

Exploiting Reversing (ER) series | Article 10

iOS Security Research (part 01)

(a step-by-step exploitation series on Windows, iOS, hypervisors and others)

by Alexandre Borges

release date: SEP/30/2026 | rev: B.1

00. Quote

"I know you've taken it in the teeth out there, but the first guy through the wall -- he always gets bloody, always. It's the threat of not just the way of doing business, but in their minds it's threatening the game. But really what it's threatening is their livelihoods, it's threatening their jobs, it's threatening the way that they do things. And every time that happens, whether it's the government or a way of doing business or whatever it is, the people are holding the reins -- have their hands on the switch -- they go batshit crazy."

(John Henry played by Arliss Howard | "Moneyball" - 2011)

01. Introduction

Welcome to the tenth article of Exploiting Reversing (ER) series, a step-by-step exploit development and vulnerability research series on Windows, iOS, hypervisors, and others. Last articles of Exploit Reversing series are listed below:

Primary source:

- <https://www.blackstormsecurity.com/research/>

Secondary source:

- ERS_09: <https://exploitreversing.com/2026/04/28/exploiting-reversing-er-series-article-09/>
- ERS_08: <https://exploitreversing.com/2026/03/31/exploiting-reversing-er-series-article-08/>
- ERS_07: <https://exploitreversing.com/2026/03/04/exploiting-reversing-er-series-article-07/>
- ERS_06: <https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/>
- ERS_05: <https://exploitreversing.com/2025/03/12/exploiting-reversing-er-series-article-05/>
- ERS_04: <https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/>
- ERS_03: <https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/>
- ERS_02: <https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/>
- ERS_01: <https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/>

This article is the first installment of iOS articles from Exploiting Reversing Series (ERS), which aims to discuss details about iOS security research. Although the previous ERS_04 article introduced a few initial foundations and practical procedures, all of them were focused on macOS and not iOS.

Next articles will compose a multi-part series on iOS security research, covering some topics in depth, and which aim to provide readers with recommended foundation on iOS internals, security mechanisms, vulnerabilities, and exploitation development of N-day vulnerabilities. Honestly, I have a plan to write a long sequence of articles discussing iOS security research, and hopefully I will do it. Although I have tried to avoid errors and inaccuracies, they are inevitable and expected.

02. Acknowledgments

It's 2026, and even today, there are very few detailed documents on vulnerability research and real-world exploit development. Currently, I have the distinct impression that the era of information sharing is over. In fact, nowadays, we have several new articles per week, but most of them only aim to show the final results, without explaining the entire process from beginning to end, which doesn't help other colleagues to give their own steps in exploitation research. Unfortunately, the willingness to demonstrate the craft of exploit development has diminished due to money and other factors.

A few years ago, when I started authoring articles on malware analysis, vulnerability research, and exploitation, I had a clear decision in mind: I should share information without restrictions because, in the end, this wouldn't prevent me from improving my skills and pursuing my career. As expected, time is a major limitation for writing regularly, but I continue to strive to establish a solid foundation of information that can be valuable to other professionals. As I always remember, I wouldn't have been able to publish these articles without the help of **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)**, who have offered me all the necessary support over the years. Finally, research is living in a new era of AI, but nothing replaces our minds, capable of generating unlimited knowledge and solving problems that, at first glance, seem impossible.

Life may be short, but every moment is worthwhile because people are the best thing in this world. Enjoy the journey and keep exploiting it!

03. Lab Requirements

This article and next ones I will be using the following lab configuration:

- **MacBook Air (Apple Silicon):** machine running current versions of macOS Tahoe with Xcode installed.
- **Windows 11:** Windows 11 works as my workstation, from where I control and access macOS and paired mobile devices.
- **Linux Ubuntu (WSL):** it works as a backup system to run heavy tasks just in case it is not possible do it on macOS machine.
- **IDA Pro or IDA Home version (@HexRaysSA):** <https://hex-rays.com/ida-pro/>. Throughout this article, I will be using IDA Pro 9.4 and 8.4.
- **iOS devices:** iPhone devices and/or iPad. I will be using two iPhones, one of which runs beta images and another one with general release version.

- **XNU source:** clone the last XNU source code from <https://github.com/apple-oss-distributions/xnu>
- (optional) If you do not want to run labs directly on the macOS host, a virtual machine is the recommended alternative. As option for virtualization software, I suggest VirtualBuddy, which requires an Apple Silicon host: <https://github.com/insidegui/VirtualBuddy>

Of course, if you do not have IDA Pro, you can follow the article using Ghidra, Binary Ninja or Hopper. Additional infrastructure will be built throughout the article.

04. iOS security research overview

One of most difficult decisions that a new iOS or macOS security researcher makes is not which vulnerability class to learn or analyze first, but which environment to invest in, and such a choice is complicated because the cost of getting it wrong is invisible until weeks have passed, when you realize that you do not have the necessary environment to work and test it appropriately. To be honest, most of the time professionals do not know where to start or even what is necessary because apparently everything is hidden somewhere we cannot find, and nobody is close to pointing the correct path.

An immutable fact is that during iOS research activity, you make several experiments, write multiple scripts and applications for deployment, collect a series of panic logs over time, and hopefully submit valuable proof-of-concepts either to Apple or to a broker, and this entire process is filtered through the environment you set up at the beginning, so building the environment wrong fatally will cause months of intellectual work wasted (or not optimized), which includes tools incompatibilities, hassles with signature certificates, and basic and critical knowledge gaps that nobody warned you about. Definitively, there are few articles on iOS security that dedicate time to explain the really first steps and most of them simply skip this infrastructure layer entirely because they are produced to explain a particular research and point of view, and security researchers starting their career in the field get lost and disappointed quickly. Several times I have heard *"where do I start?"* and it is a legit question.

Another point that I have noticed is that multiple existing articles are written as if the reader already has a paired test iPhone with Developer Mode enabled and a valid provisioning profile, a properly prepared Apple Silicon Mac with System Integrity Protection (SIP) disabled and AMFI relaxed, a working ipsw framework plus IDA Pro pipeline that turns a fresh IPSW into a fully symbolicated database, an SSH chain that lands in a jailbroken iPhone in one command, and a habit of pulling panic logs through `idevicecrashreport` command.

Obviously, none of those exists by default and, actually, each one takes a session of real work to set up, consumes time, dedication to keep working (sometimes spending a huge effort), and the maintenance procedures every time Apple ships a new iOS version, even a minor release. Furthermore, new researchers do not know even the meaning of these stages. This first article on the subject starts by offering coverage on this work for building a research lab and, in parallel, introduces concepts on iOS, and next articles will explain and present a long list of topics in applied experiments.

In practical terms, a useful iOS research lab in 2026 has exactly four roles, and all of them will be discussed later:

- **workstation**: the daily workstation, which can be any operating system, but it is typically Linux or Windows, where the editor and the XNU source code repository live for eventual checking. It works as an orchestrator.
- **Mac host**: the main and real research platform, where Apple devices are attached and vPhone environment is running. It is accessed remotely from the workstation for static and dynamic tasks, and mainly to access the running vPhone (below) and physical iPhone devices.
- **vPhone**: it is an Apple Virtualization.framework guest running a jailbroken iOS image, which became the main iteration resource for harnesses that exercise userland and any kernel surface reachable from userland. It is another important resource used throughout this series.
- **physical devices**: iPhone, where we have real silicon and running iOS where findings are validated. In other words, it is the last word about potential vulnerability or exploitation.

It is a good moment to make a note on what probably will change between the writing time and publication. This article is captured against the lab as it stands in mid-2026, and the iOS resources being used in this lab are running multiple versions such as iOS 26.5 (build 23F77), iOS 26.6 (23G71), different releases of iOS 27 beta, iOS 27.0 GA and iOS 27.2 beta 1 and 2, which were or are available on my iPhone devices dedicated for tests (iPhone 16 (A18) and iPhone 12 (A14)). Readers can connect their devices to Mac host and confirm them as shown below (obviously, your models will be different from mine):

```
$ xcrun devicectl list devices
```

Name	Hostname	Identifier	State	Model	Reality
iPhone12		<uuid-1>	connected	iPhone 12 (iPhone13,2)	physical
iPhone16		<uuid-2>	connected	iPhone 16 (iPhone17,3)	physical
vPhone		<uuid-3>	connected (no DDI)	iPhone99,11	virtual

About the output above, some values from Identifier field were edited to make the output easier to read, but you will see long hexadecimal values in your own real environment. A further comment on this column, it contains a pairing session identifier the Mac assigns when it pairs with a device, and not the permanent hardware UDID of the device, so every subsequent **devicectl** command takes it as its **--device** argument. For example, with that identifier in hand, **xcrun devicectl device info** details command reports the running iOS build, as shown below:

```
$ xcrun devicectl device info details --device <uuid-2> | -i "OS Build Update"
OS Build Update: 24B5089g
```

Moving this discussion to technical aspects directly associated with iOS security, the attacker model is a good topic to comment on details and scenarios that will use exactly similar infrastructure to find bugs and, occasionally, vulnerabilities. Currently, the existing protection stack on current iPhone version is the strongest in the industry, and any kernel exploitation chain must defeat or navigate through distinct layers in sequence (concepts mentioned above will be explained ahead and also next articles), which can be really hard to bypass:

- **KASLR (kernel address-space layout randomization)**: it is a protection that randomizes the kernel's placement in the virtual address space on every boot so that an adversary cannot predict where any kernel function or data structure lives, and which readers find similar (definitely not equal) implementations on Windows and Linux. There is exactly one randomized value per boot, the kernel slide, and every context a panic log prints derives from it plus a fixed layout offset, so

unsliding any address is a subtraction operation once you know that one value. I will be commenting on such a kernel slide (or "slid") later.

- **Pointer Authentication (PAC):** this protection signs every protected function pointer, vtable entry, and return address with a key-derived MAC, so an attacker who achieves kernel read-write primitive cannot write bytes (controlled by the attacker) into a function pointer slot and also expect control flow to continue (similar to done with ROP attacks), because computing a valid MAC requires to execute a PAC signing instruction (composed by PACIA and PACIB) with the correct key loaded by the kernel into signing registers. An apparent alternative would be obtaining a data-only write primitive, which is everything that a read-write primitive kernel gives without code execution, but as the own name says, they cannot run instructions.
- The **kernel allocator (kalloc_type)**: this protection represents a memory reorganization that segregates kernel allocations by C struct types, so that heap grooming primitives cannot place an object controlled by attackers adjacent to a "region" with a different existing type, which I will discuss later, and personally I think it is a clever approach that causes serious hurdles while using attacking iOS through memory class vulnerabilities (I remember an vulnerability in namei component that I couldn't proceed due this restriction). In addition to this first one, there is also zone_require call-site assertions (they consist of a set of runtime security checks to prevent type confusion, UAF and other manipulations), which crashes the kernel if a pointer dereferenced as type Z is allocated from a zone other than Z's zone. Both protections are really efficient when acting together and break most strategies using memory vulnerability class.
- The **page-table monitors** enforce kernel-text integrity and prevent the kernel from creating write-and-execute mappings. There are multiple representations (or evolutions) of such monitors over time:
 - On A12 to A14 silicon and the M1 (run macOS, but it is not definitively the focus of this series), this monitor is represented the **Page Protection Layer (PPL)**, which has been introduced with the A12 in iOS 12, and runs inside EL1 alongside the kernel but uses Apple's APRR and SPRR (Access Permission Remapping Registers and Secure Permission Remapping Registers, respectively) hardware to create privilege separations within that exception level, where only PPL code can write page tables and kernel-text protection structures, so even a kernel with full read-write primitive cannot modify its own code mappings. PPL is a strong protection, but as readers likely have read, there is a bypass (Titan) for PPL and SPTM (below) for iOS 16.1 to 17.4 beta 3 for devices with processors A14 to A17, and details can be found on <https://github.com/alfiecg24/Titan>.
 - **SPTM (Secure Page Table Monitor)** is a small, signed firmware image that runs in a hardware-isolated domain that the kernel cannot directly enter. Operating at GL2 (a guarded exception level reached from EL1 via Apple's proprietary GENTER instruction) it holds exclusive ownership of the system page tables. For example, on A15+ processors (including the A18), the XNU kernel can no longer modify its own translation tables directly, and it must instead dispatch requests to SPTM to execute any memory mapping changes.

Consequently, achieving a kernel read-write primitive on modern silicon grants an attacker total control over kernel data structures, which could help to perform elevation of privilege, but SPTM entirely blocks the page table forgery that older iOS exploits relied on.

- **TXM (Trusted Execution Monitor)** is its companion subsystem, which operates at GLO (a guarded exception level managed under the same hardware-enforced isolation architecture). Basically, TXM takes the ownership of code signing and runtime decisions associated with entitlements and effectively moves core trust evaluation out of AMFI into a distinct domain the XNU kernel cannot directly access or tamper with.
- Based on the exposed above, TXM decides whether code is trusted and SPTM decides what the translation tables say, though they are deliberately asymmetric from a security perspective because TXM at GLO is the less-privileged of the pair, so that its compromise does not by itself imply an SPTM bypass. To introduce some context, SPTM was introduced alongside iOS 17.0, natively anchoring its hardware-enforced controls on the newly released A17 Pro and later silicon (including the A18 and M3 generations). Consequently, the A17 Pro never ran the legacy PPL architecture at all. Meanwhile, older chips like the A15, A16, and M2, which ran traditional PPL throughout the iOS 16 era, transitioned toward the updated monitor framework via software updates upon upgrading to iOS 17. As a result, if we consider TXM alongside PPL or SPTM (depending on silicon), they make the attacker recipe of obtaining a kernel read-write primitive, manipulating page tables and mapping unsigned code executable in user-space harder (or unfeasible) without first defeating the monitor, kernel text itself having been covered since A10 by KTRR (Kernel Text Readonly) and KIP (Kernel Integrity Protection), and persistence by the signed system volume. That defeat was achieved for some iOS versions and devices, as shown by the Titan bypass, although Titan does not break GL2 on its own terms because it obtains execution on the AGX coprocessor, bypasses microPPL there, and then maps an AP kernel page table with a self-referencing entry, which is a lateral move through a peripheral trust domain rather than a compromise of SPTM itself. Titan reaches iOS 17.3.1, and nothing public defeats SPTM on 17.4 or later (at the time that this article is being written, at least).
- **Exclaves**, which became fully implemented on the A18 and M4 silicon generations (after have been introduced on the A17 Pro), separate the most sensitive system services such as specific cryptographic operations, sensor paths, and the microphone/camera status indicators (all of these ones are related to coprocessors) into isolated domains. Exclaves are controlled by dedicated microkernel and protected by SPTM, which makes these domains completely unreachable by the XNU kernel, even if an attacker achieves a full kernel read-write primitive as occurred frequently in the past.
- **Code-signing** enforcement and trust cache form the final layer. AppleMobileFileIntegrity (AMFI) and static trust cache, which is a system-wide table of authorized cdhashes (cryptographic fingerprint used by the kernel to identify and verify the integrity of a binary or signed application) loaded at boot, together ensure that only signed by Apple, or code provisioned by a developer, executes on the device. This way, a kernel attacker who gains read-write primitive still cannot run arbitrary

unsigned code unless they first defeat the code-signing verification, which on modern devices is enforced by TXM, and thus outside the kernel's reach, as you have already learned. Therefore, even after getting a read-write primitive, the attacker is constrained to data-only techniques rather than injecting and executing new code.

All these protections alongside with other ones that are not mentioned this time are one of reasons about why modern iOS kernel exploitation is increasingly focused on data-only attacks. Actually, having the page tables, the code-signing decisions, and the most sensitive services have moved out of the kernel's reach, so a bug/vulnerability that would have been a full compromise on an A14 device may be only a primitive on an A18, at maximum.

None of these layers is a single-step bypass, and usually each one requires a tailored primitive that produces a value the layer was not designed to deal with, which forces researchers to build exploit chain based on multiple primitives across layers in sequence rather than relying on a single bypass. One of most interesting and recent illustrations on what defeating this stack looks like in practice was given by Operation Triangulation, which has been disclosed from June 2023 onward by Kaspersky GReAT (the June disclosure writeup authored by Igor Kuznetsov, Valentin Pashkov, Leonid Bezvershenko and Georgy Kucherin, and the December hardware writeup by Boris Larin, Leonid Bezvershenko and Georgy Kucherin with Igor Kuznetsov, Valentin Pashkov and Mikhail Vinogradov) during their internal incident response involving iPhones that belonged to their researchers being targeted. This case is a valuable example of a campaign combining four CVEs across the entire stack, and that involved vulnerabilities reported by CVE-2023-41990 (initial code execution vector), CVE-2023-32435 (WebKit stage), CVE-2023-32434 (kernel read-write primitive) and CVE-2023-38606 (PPL bypass). Mentioned articles are <https://securelist.com/operation-triangulation/109842/> and <https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/>, respectively.

Nowadays, using several CVEs and zero-days is no longer exceptional and it is the standard shape of a modern iOS exploitation chain. Actually, Apple's 2026 bounty matrix reflects this by paying highest for the chains that demonstrate end-to-end zero-click compromise of a current iPhone and, if readers want to go back almost a decade and a half ago, Stuxnet already used four zero-days for achieving its objectives. Anyway, we will examine the Triangulation chain details in future articles.

Researchers returning to iOS after a year or two off Apple work will find several things have changed in ways that affect a lab rebuild, and changes are mostly improvements but each one breaks at least one piece of old tribal knowledge. For example, in 2026 we have a valuable resource named vphone-cli (referred as vPhone in this article) that is the Laker233's open source project at <https://github.com/Laker233/vphone-cli>, which is based on the excellent work produced by wh1te4ever during 2025 and 2026, and known as the canonical "jailbroken iOS running virtualized on a Mac". In a few words, the project provides researchers with a running virtualized and jailbroken iOS that is reachable from SSH access on port 2222, which in readers will learn and walk through the full installation and configuration procedure. The consequence for practical research is really direct because it demonstrates that iOS research is much more demanding than ever, where only doing passive code reading (from kernelcache and XNU source code) is no longer enough due to the number of entangled layers and the requirement for observing the actual machine state through the process.

In terms of software, [ipsw](https://github.com/blacktop/ipsw) (blacktop's tool at <https://github.com/blacktop/ipsw>) is, no doubt, one of the most important tools for iOS research and has added much richer diff and symbolication features through

recent versions. The `ipsw kernel symbolicate` subcommand now pairs natively with the blacktop/symbolicator signature library and produces excellent function coverage on a current iOS kernelcache.

The `ipsw kernel diff` subcommand is registered but not yet implemented in current ipsw (at least at the moment this draft is being written), but the top-level `ipsw diff` command, whose usage is `ipsw diff <IPSW|OTA|DIR> <IPSW|OTA|DIR>` so that it accepts full IPSWs, OTAs or already extracted directories, and compares their entire contents (kernelcaches, firmware images, MachO's, launchd plists), works and produces a structured per-component diff report with `--markdown`, `--strs`, and `--starts` flags for reaching into the level of MachO changes, which is the one of most important resources for hunting vulnerabilities. Definitely, we will discuss on diffing approaches at the end of this article.

The `dyld shared cache` subcommands under `ipsw dyld` now provide a cross-platform and effective alternative to the older `dyld_shared_cache_util`, which requires building from Apple's open source dyld project, and Xcode's `dsc_extractor`. Furthermore, it presents consistent subcommand naming and richer output because a single `ipsw dyld macho` command invocation extracts and parses an in-cache dylib that the older tool chain needed three steps for.

Following with software part of necessary infrastructure, we will be using IDA Pro 8.4 and 9.4, which offer expanded architecture and improved and expanded support for iOS. The two excellent Cellebrite Labs plugins for iOS kernelcache research, `ida_kernelcache` (originally by Brandon Azad, the OSMetaClass and class-hierarchy reconstruction tool) and `ida_kcpp` (the C++ virtual method call recovery tool that maps vtable entries to named methods), are key and indispensable resources.

The **Apple Security Research Device (SRD)** program is Apple's loan of a specially fused iPhone that closes the gap by offering the possibility of observing the kernel running under controlled scenarios, because on that device a researcher gets shell access, may choose any entitlements and run code as platform and as root outside the sandbox, may install and boot custom kernelcaches and custom SPTM and TXM firmware, and may hold the [research.com.apple.license-to-operate](https://security.apple.com/research-device/) entitlement that permits one process to debug any other process on the system including system processes, and for those researchers with access to the device, the SRD is one of the most valuable lab assets for kernel research, but it will not be used through this series. The Apple SRD program details can be accessed on <https://security.apple.com/research-device/>.

To check what is really installed on your macOS before proceeding, a short live capture follows below:

```
$ sw_vers
ProductName:    macOS
ProductVersion: 26.6
BuildVersion:   25G72

$ xcodebuild -version
Xcode 27.0
Build version 27A266a

$ xcrun --sdk iphoneos --show-sdk-version
27.5

$ ipsw version
Version: 3.1.712, BuildCommit: 22e10c7025e7f802d60cd92a5d9a39169615f646
```

Apple has published XNU sources strictly later than the corresponding shipping binary and, according to practical observations, the gap ranges from a few weeks to several months and has at times been wider. The available source code versus the binary gap is the single most important operational fact a new researcher must learn because the source you can read does not match the kernel you are attacking, and certainly any code understanding and new findings are based on shipping kernelcache via symbolication and IDA Pro disassembly and not just against the public XNU source, which may be useful, but clearly it is not enough.

05. Foundations and concepts

There is a list of foundational and key concepts that recurs in all the later sections and mainly in the next articles, and this subsection introduces a short list of them, so that readers coming from Linux or Windows internals will hopefully recognize the analogues and the similarities.

Apple's iOS and macOS arm64 binaries ship in one of two ABI slices, which are named arm64 and arm64e. The distinction is recorded in the binary itself, where the two slices carry the same `cputype` value in the Mach-O header and are distinguished by the `cpusubtype` field alone, and it is an ABI distinction in the sense that the instructions the compiler emits at function entries, function exits, and indirect branch sites differ between the two slices. The reference for arm64e ABI is Apple's open source `PointerAuthentication.rst` at <https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst>, and Apple has kept it in perfect sync with the production compiler since the arm64e ABI was first published in 2018.

In terms of architecture, the arm64 slice is the standard ARMv8.x AArch64 and, by default, every third-party iOS app that ships through the App Store, each iOS binary that a researcher signs with a developer certificate and deploys through the standard Xcode provisioning flow, and each iOS binary that has not opted into the hardened ABI is compiled to arm64, which is the default. Since 2025 there is the available option for the Enhanced Security capability, which has been introduced with Xcode 26 and iOS 26, and that will do a few references below.

We need to go a little further on details for establishing foundations that will be used in the next paragraphs, so the Mach-O header carries `cputype = CPU_TYPE_ARM64` (the constant 0x0100000C, which the `<mach/machine.h>` header defines as `CPU_TYPE_ARM | CPU_ARCH_ABI64`) and `cpusubtype = CPU_SUBTYPE_ARM64_ALL` (the value 0) or, less commonly, `CPU_SUBTYPE_ARM64_V8` (equal to 1), and that is used when the binary is explicitly tagged for ARMv8 baseline rather than the default.

Therefore, common operations such as calls, returns, and indirect branches in arm64 code use the standard `BL <target>` (branch with link), `RET` (return through the link register X30), and `BR Xn` and `BLR Xn` (indirect branch / indirect branch with link) instructions, respectively, and all of them without being submitted to cryptographic check, so the CPU does (execute) whatever the register or immediate operand says. In this scenario, an attacker who changes a return address on the stack of an arm64 process and works for the changed address to reach the `RET` instruction is able to observe a CPU branch (jump) to the provided address because there is no integrity check between the load of the return address from the stack and the branch through it. To be clear, I am referring to arm64 applications and not arm64e iOS code.

From a security perspective, readers who already think in terms of exploitation primitives usually ask at this point whether an attacker can simply recover those keys, and the answer could be more interesting than a simple refusal explaining that it is not possible. The first important fact is that the key registers themselves are completely out of reach of an eventual memory read primitive and reading one requires executing an MRS instruction (Move System Register to General Purpose Register) rather than reading a location. Furthermore, this cannot be done from the EL0 exception level, and modern platform mitigations often render these registers entirely write-only or hidden even from a compromised kernel at EL1.

Additionally, values contained in those registers represent another restriction because the kernel keeps a per-process copy of them in memory, in the `rop_pid` and `jop_pid` fields from task and thread structures (check `osfmk/kern/task.h` and `osfmk/arm64/thread.h`, respectively), and any context switch in the kernel loads one of those fields straight out of the thread and writes it into the `apibkeylo_el1` register. Therefore, an attacker that holds a kernel read primitive can learn what key material a given process runs under, and this turns the useful question from how to obtain a key to what obtaining one is actually worth, and it is worth little.

Unfortunately, even if researchers hold the exact key value, they cannot easily compute a valid signature in software because the hardware component applies to the custom algorithm instantly when the processor executes a PAC instruction at runtime. Finally, these per-process keys are only the first part of the equation, and there is a second part that is the modifier described further below, which varies from one signing site to the next, and binds a signature to its context. Because these keys and modifiers vary from one process to the next, they turn every process's signature to be unique. Another point is that holding the key would still not be enough because Apple does not use the standard architectural MAC construction but uses and deploys a custom implementation of the QARMA cipher (read more about it on <https://en.wikipedia.org/wiki/QARMA> and <https://csrc.nist.gov/csrc/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/qameleon-spec.pdf>) that cannot be readily emulated. Consequently, a researcher who somehow obtained the key material could still not compute a valid signature in software.

Fundamentally, signing is an execution capability rather than a memory capability, since the only way any signature is generated is for the CPU to fetch, decode, and execute a PAC instruction, and there is no other way to obtain the same effect. A data-only primitive changes what memory contains, but it does not change what the processor does next, so this action violates the rules and never causes a signing instruction to run, because the attacker is only writing instruction bytes into memory and nothing more. Moreover, running your own code at EL1 requires redirecting control flow, but every protected indirect branch authenticates before it is taken, and authenticating requires the exact signature you were trying to forge.

Of course, this is not game over because the kernel itself is full of signing instructions, and an attacker who can drive one of those existing instructions to operate on a value they control achieves a signature without ever executing code of their own. At the end, this explains why any work in this area is focused on finding signing gadgets and signing oracles rather than on recovering key material.

As we have discussed about keys, they are classified according to their purposes, as shown briefly below:

- Two instruction keys, which are named **IA** and **IB**. They sign code pointers to which a branch (jump) will occur.

- Two data keys, named **DA** and **DB**. They sign data pointers that are stored in memory and later de-referenced.
- One generic key, named **GA**. It signs arbitrary 64-bit values that are not themselves pointers.

We will repeat this same explanation later, but with further details about each key type.

Every signing operation also blends in a 64-bit modifier, which is also called the discriminator (Apple used the term "discriminator", while ARM uses the term "modifier", and I will be using both in this text), alongside the key, which produces a direct effect where the same pointer signed under a different modifier produces a different signature. Therefore, the modifier (discriminator) forms part of the mechanism that binds a signature to a context, and it is precisely what prevents an attacker from copying a valid signed pointer out of one location and replaying it somewhere where the modifier does not match.

In terms of code itself, instruction mnemonics follow a predictable naming convention, where the PAC prefix indicates a signing operation, while the AUT prefix means an authentication operation that verifies the cryptographic signature and restores the pointer. The suffix specifies the key used for the operation, where IA and IB designate instruction keys and, respectively, while DA and DB denote data keys. Additionally, the GA suffix distinguishes the generic authentication key, which signs arbitrary values rather than belonging to the instruction or data split.

At instruction level, the basic form takes two registers, so the **PACIA Xn, Xm** instruction signs the pointer in register **Xn** (first operand) with the IA key against the modifier in **Xm** (second operand), and in a similar way the **AUTIA Xn, Xm** instruction verifies that same pointer, where on success it replaces **Xn** with the clean pointer, and on failure it faults synchronously rather than silently producing a poisoned value, which is a safer behavior.

Readers that examine the kernel will find several instructions that follow this format, such as **pacia x0, sp** instruction, where the stack pointer (sp) serves as the modifier. As expected, the **PACDA** and **AUTDA** instructions are the data-key counterparts to pointers that reside in data structures, rather than being the target of a jump. Occasionally, instructions may be shorter because they incorporate a fixed modifier, and the pair a researcher meets constantly is **PACIBSP**, which signs the link register **X30** with the IB key using the stack pointer **SP** as the implicit modifier, together with **RETAB**, which authenticates **X30** with the IB key against **SP** and returns in a single instruction. Additionally, the combined branch forms such as **BLRAA** and **BRAA** authenticate a pointer with an instruction key and branch to it in a single step, which does not leave possibility for a race condition attack, when an attacker could place an unauthenticated pointer into a register waiting to be used. Finally, and still in terms of instruction syntax, the generic key is distinct from the format presented so far because the **PACGA** instruction takes three register operands rather than two and writes its result into a dedicated destination register instead of modifying a pointer in place and, as readers will learn in next articles, the kernel uses it to chain a MAC across several values.

Note that it is not easy to remember the meaning of these instructions at a first reading (maybe neither in a second reading), but such knowledge helps to understand results (instructions) emitted from the arm64e compiler, mainly at function entries, function exit, and indirect calls. As a simple example, function entries sign the return address with the IB key via **PACIBSP** instruction, which is the convenience encoding that signs the **link register X30** against the value of the stack pointer **SP**, by using **SP** as the modifier, so that the same return address signed under a different stack frame produces a different signature. At function exit, the **RETAB** instruction authenticates the **return address in X30** against the current **SP**, strips the PAC bits,

and returns to the caller. The most relevant aspect is that this procedure takes place in a single instruction, meaning there is no time interval during which an unauthenticated return address remains in the link register and, therefore, preventing the mentioned race condition possibility.

Expanding the discussion a bit more, indirect calls through vtables authenticate and branch in one instruction, using **BLRAA** instruction, which authenticates Xn with the IA key against the modifier held in Xm and then branches with link. Even though these steps should be naturally together, the compiler does not always keep it, and when the authenticated pointer is going to be used for something other than an immediate branch, the compiler emits the split form instead, which is **AUTIA Xn, Xm** followed by a plain **BLR Xn**, so the authentication itself and the respective use become separate instructions.

Data key variants **PACDA** and **AUTDA** parallel the instruction key variants for pointers stored in data structures rather than jump to. For example, a signed pointer in a struct field is signed with **PACDA** when the struct is written and verified with **AUTDA** when the pointer is loaded back. Separately, the generic key variant offers an approach for non-pointer data. The generic key instruction **PACGA Xt, Xn, Xm** computes a 32-bit MAC over the value in Xn under the modifier in Xm using the GA key and writes it into the upper 32 bits of Xt (zeroing the low 32 bits) without modifying Xn or Xm. Because **PACGA** leaves its inputs untouched, it is a good resource to protect values that are not pointers and will never be used for branches or memory loads.

Previously I referred to the concept of slices such as arm64 and arm64e, and it is time to explain a bit more about this subject. Such slices coexist in a single process, on the same CPU, without requiring a mode or context switch, because arm64 and arm64e share the same base instruction set, and arm64e simply adds PAC instructions that arm64 code never emits. A third-party app's process loads its own arm64 binary alongside the **dyld_shared_cache** framework binaries (which are arm64e) into the same address space, and calls between the two work exactly like any other function call. As a basic demonstration, the arm64 app executes **BL _framework_function**, which deposits the return address in X30 register and branches to the framework entry point.

At this entry point, in arm64e architecture, we find as first instruction the **PACIBSP** instruction, which signs the return address the caller just saved. Once the function body executes, the code reaches the **RETAB** instruction, located at the epilog, which authenticates X30 (LR) against SP, strips the PAC, and finally returns. If you notice, the arm64 caller never issued a PAC instruction, and actually it does not need to, because the protection is entirely on the callee's side. As you could conclude, this design is on purpose because the hardened ABI protects the system code (the callee side) without requiring the third-party app (the caller side) to be recompiled for arm64e. Therefore, arm64 remains the default for third-party apps, and a developer who wants to cross to arm64e can make option for taking advantage of Enhanced Security capability (check for details on <https://developer.apple.com/documentation/xcode/enabling-enhanced-security-for-your-app>).

The smart move at this time is to show these concepts and their effects in practice. We have the vPhone, which runs jailbroken iOS (reader will see its configuration later) and is the one resource that accepts arm64e binaries built with the full SDK toolchain and ad-hoc signed with **Idid** command, so it is where researchers can run their own code with PAC instructions. On the other hand, the two available stock iPhones that are present on my lab (iPhone 16 and iPhone 12) run the application as arm64 (the default), so the researcher's code on those devices contains no PAC instructions, and the **dyld_shared_cache** frameworks those apps call are arm64e, but the binary from application is not.

The program below, [pac_probe.c](#), is a small probe that signs a data pointer and prints whether the signature took effect. It calls two [clang <ptrauth.h> builtins](#), which are [ptrauth_sign_unauthenticated](#) (sign a pointer with a key and a discriminator) and [ptrauth_auth_data](#) (verify that signature and strip it back to the raw pointer). As a result, on the arm64e slice, these emit real [PACDA](#) and [AUTDA](#) instructions, while on the arm64 slice they are completely eliminated during compilation, so the own output from the probe application reports which slice it ran on.

The `pac_probe.c` code follows below:

```
// pac_probe.c -- sign a data pointer with the DA key, then read it back.

#include <stdio.h>
#include <ptrauth.h>

int main(void) {
    int target = 0;
    void *raw_p = &target;

    void *signed_p = ptrauth_sign_unauthenticated(
        raw_p, ptrauth_key_asda, 0x1234 /* discriminator */);

    // ptrauth_auth_data: verify + strip the PAC (emits AUTDA on arm64e).
    void *auth_p = ptrauth_auth_data(
        signed_p, ptrauth_key_asda, 0x1234);

    printf("raw    = %p\n", raw_p);
    printf("signed = %p -> %s\n", signed_p,
        signed_p == raw_p ? "no PAC (arm64 slice)" : "PAC bits present (arm64e)");
    printf("auth   = %p -> round-trip %s\n", auth_p,
        auth_p == raw_p ? "OK" : "FAILED");
    return 0;
}
```

Readers must execute the next three steps to build it for arm64e, sign it ad-hoc, and then install and run it on the vPhone in a single SSH round-trip command:

```
$ xcrun -sdk iphoneos clang -arch arm64e -o pac_probe pac_probe.c
$ ldid -S pac_probe
$ cat pac_probe | ssh -p 2222 root@localhost '/var/jb/bin/cat > /var/jb/pac_probe;
/var/jb/usr/bin/chmod +x /var/jb/pac_probe; /var/jb/pac_probe'
```

Explaining commands above, the first one selects the iOS SDK, which is given by `-sdk iphoneos` option, the arm64e slice (`-arch arm64e`) and names the output (`-o pac_probe`). The second applies an ad-hoc code signature in place (through the `-S` option) that the jailbroken loader accepts. Finally, the third command streams the binary's bytes to the writable `/var/jb/pac_probe` file on the vPhone, marks the file executable, and then runs it.

It is worth explaining a small detail about that last command, as omitting it is a mistake that costs a lot of time (and I suffered with this mistake), is that `cat` and `chmod` should be written as full paths because a command sent through SSH in this non-interactive form does not read a login profile, so it inherits only the small default search path of the base system, and the jailbreak's utilities are not on it. Written as simple names, both commands fail with *"command not found"*, the binary is never written, and the error you finally see is a permission failure from the program that was never copied. Running the program on the vPhone, whose binary is arm64e, shows the following output:

```
raw    = 0x16aed3328
signed = 0x1e00016aed3328 -> PAC bits present (arm64e)
```

```
auth    = 0x16aed3328  -> round-trip OK
```

As shown above, the signed value carries PAC bits in the high bits of the word that a user pointer never uses, which is the "0x1e000" sitting above the 36-bit pointer. Authenticating strips those bits back to the original pointer, and that is why the last line reports a successful round trip. In your experiments, do not expect to reproduce that prefix because the signature depends on both the pointer and a per-process key, so it changes from run to run, and a second run of the same binary produced 0x1b000 instead. What reproduces is the structure, which is a non-zero field above the pointer rather than any particular value in it.

The second phase of the experiment rebuilds the same source for the arm64 slice (and not arm64e) and runs it on the same vPhone. In this new test, the two inline functions are eliminated during compilation, which does not generate code, so no PAC is ever applied and the values match, as shown below:

```
raw      = 0x16b623308
signed   = 0x16b623308  -> no PAC (arm64 slice)
auth     = 0x16b623308  -> round-trip OK
```

Readers should notice that the raw pointer is a stack address which varies from run to run, so the three addresses above will not match theirs either. In fact, what remains invariant between the two slices is the presence or absence of the upper PAC bits, and this is precisely the property this section aimed to establish: the same source, compiled in two different ways, may or may not include pointer authentication, and the program can indicate which is the case without requiring a debugger. As you can check and confirm, the vPhone accepts ad-hoc signed arm64e binaries because its jailbroken loader requires neither an Apple signature nor an Enhanced Security entitlement, so it will run any binary that **ldid -S** has touched.

We can confirm that the arm64 result is the real stock device behavior and not a vPhone artifact by deploying the same application to one of the available iPhones (iPhone 16 | A18) through the common developer-signed path and building, installing and capturing pipeline described in the earlier sections, which wraps the test in an app, installs it with **devicectl** command, and collects its respective output. On proposed iPhone, which has been provisioned the normal way, the result is identical, as shown below:

```
raw      = 0x16daaecec
signed   = 0x16daaecec  -> no PAC (arm64 slice)
auth     = 0x16daaecec  -> round-trip OK
```

Certainly, we should comment on such result because it decides where this kind of work and experiment can be done at all. As you can realize, any research that depends on executing real PAC instructions has to happen on the vPhone or on the macOS host, and not in an app on the physical iPhone, and the reason is that the code of the application is built arm64, which is the default under the ordinary provisioning pipeline, so it does not contain PAC instructions for studying, and this is not useful for readers. The exception is the one named earlier, where an application that enables the Enhanced Security capability introduced in iOS 26 is built for arm64e instead, and its own code then carries real pointer authentication. Additionally, there is another practical detail worth mentioning, and it can be captured from macOS. The **lipo -info** command work as the Mach-O slice inspector because it reports which architecture slices a file contains, whether that file is fat or holds a single slice.

The dynamic linker sits on disk and is fat, as you can see below:

```
$ lipo -info /usr/lib/dyld
Architectures in the fat file: /usr/lib/dyld are: x86_64 arm64e
```

Observing the output above, `dyld` contains an arm64e slice, and you might already know that it is one of the most heavily protected userland binaries because it runs first in every process and whoever compromises dyld owns that process before any other code has executed, which can be decisive for a more comprehensive exploitation and attack. In addition, an x86_64 slice backs Rosetta translated processes, because Rosetta runs x86_64 binaries on Apple silicon, and an x86_64 process needs an x86_64 dyld.

The framework binaries, meaning Foundation, libSystem and the remaining ones, have not been placed as standalone files on disk since macOS Big Sur, and they live inside the `dyld_shared_cache` instead, so executing a `lipo -info` command fails with *"can't open input file"* (the experiment below demonstrates that failure). On the other side, macOS keeps two shared caches, `dyld_shared_cache_arm64e` (native) and `dyld_shared_cache_x86_64` (Rosetta), both under `/System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/` directory, and a framework is present for both.

Querying the arm64e cache for Foundation confirms the slice existence, as shown below:

```
$ ipsw dyld macho /System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/dyld_shared_cache_arm64e Foundation | head -3
```

```
Magic      = 64-bit MachO
Type       = DYLIB
CPU        = AARCH64, ARM64e caps: USR00
```

The x86_64 cache holds the x86_64 Foundation for Rosetta translated processes. Even a first-party application such as Calculator (`/System/Applications/Calculator.app/Contents/MacOS/Calculator`) is fat x86_64 arm64e, where Apple compiles its own macOS apps to both architectures. Anyway, it is not our focus here, and my mention is only for completeness.

The default arm64 case is the third-party iOS binary, and our iOS test application, which is the `pac_probe.c` program presented earlier, or any binary the standard Xcode provisioning flow signs (without the Enhanced Security opt-in), is also arm64, as shown below:

```
$ lipo -info pac_probe # built: xcrun -sdk iphoneos clang -arch arm64 ...
Non-fat file: pac_probe is architecture: arm64
```

In contrast, on macOS, almost everything that Apple ships are arm64e, and each with an x86_64 companion where Rosetta needs one, but the arm64 slice is what third-party and code signed by a developer can obtain. As expected, iOS has no Rosetta, so iOS system binaries are only arm64e and with no x86_64 companion, and third-party apps are arm64 by default (remember once again that arm64e is now an available option).

If you want to read the slice distinction at the byte level, the fields that carry it are `cputype` and `cpusubtype` in the Mach-O header, whose `struct mach_header_64` is declared in `mach-o/loader.h` file.

`CPU_TYPE_ARM64` is the constant `0x0100000C`, and `cpusubtype` is what separates the variants of that one CPU type:

- `CPU_SUBTYPE_ARM64_ALL` = 0 -- generic arm64 binary
- `CPU_SUBTYPE_ARM64_V8` = 1 -- arm64 explicitly tagged ARMv8
- `CPU_SUBTYPE_ARM64E` = 2 -- arm64e with PAC ABI

Therefore, returning to macOS, we have:

```
$ otool -h /usr/lib/dyld | tail -2
magic      cputype  cpusubtype  caps      filetype  ncmds  sizeofcmds  flags
0xfeedfacf 16777228          2  0x80      7        20        2536 0x00000085

$ otool -h pac_probe | tail -2
magic      cputype  cpusubtype  caps      filetype  ncmds  sizeofcmds  flags
0xfeedfacf 16777228          0  0x00      2        17        1064 0x00200085
```

The first output (`/usr/lib/dyld`) reports `cputype 16777228` (= `0x0100000C` = `CPU_TYPE_ARM64`) and `cpusubtype 2` (= `CPU_SUBTYPE_ARM64E`), with `caps 0x80`, which is the header capability byte the toolchain sets to mark the arm64e PAC ABI. Additionally, and due to the fact that dyld is fat, `otool -h` command prints one header per slice and `tail -2` command selects the arm64e one. The second output (`pac_probe`) reports the same `cputype` but `cpusubtype 0` (= `CPU_SUBTYPE_ARM64_ALL`) and `caps 0x00`, which means a regular arm64 binary, where there is no PAC in its own code, and it calls into arm64e architecture for everything that touches the system.

An attacker (or researcher) who takes hex dumps of the Mach-O header at offset `0x04` (precisely the `cputype` field, 4 bytes little-endian) and offset `0x08` (the `cpusubtype` field, 4 bytes little-endian) reads `0C 00 00 01` for `cputype` and `02 00 00 80` for an arm64e binary (the `80` high byte is the `caps` field `otool` command showed, which is the `CPU_SUBTYPE_PTRAUTH_ABI` capability), or `0C 00 00 01` plus `00 00 00 00` for a regular arm64 binary. This is the byte-level signature that distinguishes the two slices, and it is what the kernel's binary loader mechanism, specifically the `exec_mach_imgact` path in `bsd/kern/kern_exec.c` file, reads to decide whether to enable PAC enforcement on the resulting process.

When the loader detects a `cpusubtype` value that passes `arm64_cpusubtype_uses_ptrauth` function (`cpusubtype` masked to arm64e with a valid pointer authentication), it clears the `IMGPF_NOJOP` flag (defined in `bsd/sys/imgact.h:162` as `0x80000000`) in the image parameters, which tells the downstream execution path that the new process uses the arm64e ABI with PAC active. On the other hand, when the loader sees `cpusubtype 0`, it sets `IMGPF_NOJOP`, no PAC enforcement is requested, and the process never executes PAC instructions in its own code, even though it still runs alongside architectures that do enforce PAC. The decision is made once at execution time and persists for the process's lifetime, and there is no mechanism to change PAC enforcement once the process has started. Obviously, readers are not required to pay attention to this level of detail explained in last paragraphs, and I have only exposed it here for completeness.

Modern iOS exploitation heavily relies on data-oriented and indirect primitives to bypass Pointer Authentication (PAC), and as direct control flow hijack is heavily restricted on arm64e then most researchers focus their chains on potential blind spots (and that are really rare) such as unauthenticated virtual tables, occasional compiler artifacts, unprotected kernel fields and even some arm64 compatibility path.

Another fundamental concept is associated with the ARM architecture itself. To be honest, this subject is not relevant to this exact point because it is much more associated with hardware itself, but I personally judge it as a really important topic that may help readers to understand future articles in this series as well as existing articles and exploits on iOS and, actually, researching on it helped me too in previous tasks and also motivated to understand and research on correlated areas. Thus, if readers are not immediately interested in this part, leave it for a later moment.

ARMv8 defines three supported translation page sizes at the architecture level, which are 4 KB, 16 KB, and 64 KB. Most ARM hardware in the world uses 4 KB pages, which is the same page size as x86_64 and the historical default Linux ARM ports. On the other hand, Apple silicon (arm64) uses 16 KB pages and, as expected, this page size decision influences iOS and macOS memory model, the amount of size allocated by `vm_map` allocator, user-space memory mapping (`mmap` function), memory alignment and mainly the unit at which page table entries cover memory, which is the unit at which an attacker who manipulates page table state can take or lose protections.

Going to the ARMv8-A processors architecture level, the address translation is laid on two top-level page tables, each named by a **Translation Table Base Register (TTBR)**, which basically is a CPU system register holding the physical address of the top-level translation table that the MMU walks for one half of the virtual address space. The MMU (Memory Management Unit) splits the virtual address space into two halves and routes each half through one of these registers, where an address in the lower half (translates through **TTBR0_EL1**), and an address in the upper half, which translates through **TTBR1_EL1**.

As readers might already know, the `_EL1` suffix associates the exception level that owns the register, where `EL1` is where the XNU kernel runs. Additionally, `EL0` represents userland and `EL2` represents the hypervisor level, while in Apple silicon the `EL3` is absent (Apple has its own approach), so this register pair is located (**TTBR0_EL1** and **TTBR1_EL1**) at the kernel's privilege level, and we can find one pair per CPU. The monitors (SPTM and TXM) do not make part of this scheme and are described later, but they run in a separate set of guarded levels. iOS and macOS point **TTBR0_EL1** at the current userland process's page tables (and they are reattributed on every context switch) and **TTBR1_EL1** at the single kernel page tables (set once at boot).

Basically, every mapping a user-space program at `EL0` (if readers are coming from the x86/x64, you can think of `EL0` as Ring-3 and `EL1` as Ring-0 level) owns lives in the lower half, with its Mach-O image, its stack, its heap, the shared-cache libraries and the read-only commpage (the communication page, a region mapped into the address space of every user-space process on iOS and macOS), and all of them translate through **TTBR0_EL1**. When this `EL0` program accesses an upper-half address, meaning a kernel pointer, the consequence and effect are distinct, and it is worth explain it in detail and slowly, because the correct perspective determines helps to determine exactly where user and kernel isolation actually lives. The first thing to understand is that `EL0` and `EL1` are not given separate translation mechanisms, but they share one, and it may be initially a surprise.

ARM groups address translation into translation regimes, and the regime that governs ordinary kernel and userland execution is the one it labels **EL1&0**. This is an initially confusing notation (at least from my point of view) that means a single regime handles translation for both `EL1` (the kernel) and `EL0` (the userland beneath it). It does this through a register pair where **TTBR0_EL1** maps user space and **TTBR1_EL1** maps kernel space, using the address's top bit, rather than the privilege level that issued the access, to choose which register applies.

Therefore, when `EL0` reads an upper-half address, that top bit still selects **TTBR1_EL1**, and the MMU conceptually walks the kernel's page tables for it. The hardware does not refuse to look just because the access originated from userland and, instead, the access is stopped a step later. Every kernel descriptor in those tables carries an access permission bit marking its page as "privileged access only". Thus, it is the permission-checking stage, and not the table-walking stage, which rejects the `EL0` access and raises a fault.

Isolation is therefore enforced per page by permission bits inside a shared table hierarchy, rather than by handing each privilege level a private, isolated set of page tables.

Under the standard iOS configuration, each translation regime is 47-bit because **TCR_EL1** is programmed with **T0SZ = T1SZ = 17** and the translated range is 64 minus 17, which yields 47 bits. From there, the 16 KB page size decomposes the address as a selector bit (bit 47, choosing between **TTBR0_EL1** and **TTBR1_EL1**), then three levels of page table each indexing eleven bits (2,048 entries per table), then a 14-bit page offset, where $11 + 11 + 11 + 14$ accounts for the 47 translated bits, with bit 47 the regime selector above them.

Per-level table size is 2,048 entries x 8 bytes each (exactly 16 KB), which is also the page size. This arrangement is proposed by design, and results in a page table fitting exactly in one page, which means that every page table is a single page allocation, and the allocator needs no special case for them. A full 48-bit space needs a fourth level, where a partial L0 table resolves the one extra bit, and that is the reason about why a 48-bit layout at this page size is four levels deep while the 47-bit one here is three. As comparison, the 4 KB page size uses four levels of 9 bits each, with 512 entries per table. I am pretty sure that readers coming from x86/x64 world will make a direct association with Intel/AMD processors.

Each slot positioned in those tables, at every level of the walk, is a descriptor, which is an entry (8 bytes) whose low bits tell the hardware engine what kind of descriptor is being referred to, bit[0] marking the entry valid and bit[1] its type, read together with the level at which it was found, as shown below:

- **A table descriptor:** the physical address in its middle bits is the next table one level down, which allows resolution to continue, while its top bits [63:59] carry hierarchical attributes (NSTable, APTable, UXNTable, PXNTable) that restrict everything mapped below it. These attributes are not our focus and will not be explained here.
- **A block descriptor:** it is a leaf that stops the walk early at an intermediate level and maps one large, naturally aligned region of contiguous physical memory in a single entry, which starts at the physical address in its middle bits and carries that region's permissions, and which is only at level 1 and level 2 under the 4KB granule and only at level 2 under the 16KB and 64KB granules, so on iOS the sole block size is 32MB.
- **A page descriptor:** it is the same idea as a leaf, but at the very bottom level, where it maps one individual page rather than a large block, and where it shares the encoding of the table descriptor, the level of the walk being what distinguishes them.
- **An invalid descriptor:** it is any entry whose bit[0] is clear, along with the block encoding when found at the bottom level, where it is reserved, and it leaves the input address unmapped so that any access to it fails.

In practical terms, a 47-bit translation regime under the 16 KB page size splits into three 11-bit table indices and a 14-bit page offset, where the bit 47 selects the regime above them, and the MMU reads one descriptor at each level of the walk:

VA[47]	selects TTBR0_EL1 (low half) or TTBR1_EL1 (high half)
VA[46:36]	L1 table index (11 bits)
VA[35:25]	L2 table index (11 bits)
VA[24:14]	L3 table index (11 bits)

VA[13:0] byte offset within the 16 KiB page (14 bits)

A simplified scheme would be: L1 table base (**TTBR0_EL1** or **TTBR1_EL1**) → L1 entry → L2 entry → L3 entry. In this hierarchy, the table descriptor in L1 provides the base of the L2 table, and the table descriptor in L2 provides the base of the L3 table. Note that the architecture explicitly forbids L1 block descriptors for the 16 KB page size, which also constrains an attacker trying to forge large block mappings that would bypass the monitor's per-page protections (PPL on A12-A14 and M1, and SPTM on A15 and later). A further architectural detail relevant to multi-process exploitation is how a kernel read-write primitive actually reaches another process's user-space memory, and what role the [Address Space Identifier \(ASID\)](#) plays.

The [Address Space Identifier \(ASID\)](#) is a small tag the CPU stores alongside each translation it caches in the TLB, and records which address space that cached entry belongs to. Since each TLB entry contains the ASID of the process to which it belongs, translations from different processes can coexist in the TLB without conflict. Furthermore, during a context switch, the kernel writes the new process's ASID to the **TTBR0_EL1** register, which ensures that only TLB entries marked with that ASID match subsequent lookups, thereby rendering the previous process's cached translations invisible without requiring them to be flushed out.

An attacker possessing a kernel read-write primitive who wants to read from or write to the user-space memory of another process cannot simply dereference the victim's user virtual address because the **TTBR0_EL1** register still holds the translation table base of the attacker's process, causing the victim's virtual address to resolve within the namespace from the attacker (and not from victim), which results in access to whatever the attacker has mapped at that address, or a fault.

The real problem is which table tree the address resolves through, and not the TLB (Translation Lookaside Buffer). The TLB tags every non-global entry it caches with whichever ASID is active at the instant of the walk, so whatever path the attacker takes is automatically consistent with the context the attacker is actually running in. Based on this fact, reaching the victim's memory is therefore not a matter of defeating the TLB, but of getting the victim's physical pages into a namespace the attacker can address, and there are actually three standard ways to do that. Only the last technique, which is a full switch into the victim's context, in fact adopts the ASID from victim, and the other two reach the victim's pages while the attacker's own ASID stays active, which is exactly why the ASID is not the resource that stops them.

Of those three ways, the most common one builds no new mapping at all, and the attacker performs in software exactly the table walk the MMU would otherwise have done, which seems to be an obvious first choice. Using the kernel read primitive, the attacker first reads the translation table base from victim **pmap** (its ttep). About this terminology I have just mentioned, the pmap term comes from physical map, and it is the per-address space structure that owns one address space's hardware page tables, where the machine-dependent mappings of virtual addresses from this space to physical frames take place. At the end, this refers to the mapping mechanism itself rather than a memory region, with each process, including the kernel itself, possessing its own.

About the technique itself, the attacker then walks that process's L1, L2, and L3 descriptors by hand (via exploit) until the victim's user virtual address resolves to a physical address. With that physical address in his hands, the attacker reads or writes the page through the physmap, which is the kernel's linear aliasing map of physical memory and that already covers all of RAM, so the access lands through a mapping the kernel already owns. As the victim's address is resolved entirely in software, this path never resolves that address through the TLB and never involves the victim's ASID. As a result, the own accesses of attacker along the way, which reads the descriptors and then the target page through the physmap (physical map),

are translated by the hardware and use TLB, as expected. Effectively, there is not any cost in this procedure because the physmap is a global kernel mapping and global mappings are independent of ASID, so no user ASID is ever consulted.

The second technique creates a new mapping, where the attacker edits its own page tables to map the physical pages from victim into the address space from attacker, so afterwards an ordinary pointer dereference action reaches the victim's data. The MMU does cache these new translations, and it tags them with the attacker's own ASID, which is exactly correct and expected because the attacker is the process performing the access, so no special ASID handling is needed.

Occasionally, readers might find the attacker's ASID tag to appear incorrect at first glance, since the data belongs to the victim and, consequently, intuition tends to look for the victim's ASID. However, an ASID tags a mapping and not a physical page (and this is a typical initial misunderstanding), and this mapping is the attacker's own, which is the attacker virtual address that is placed in the tables from attacker, and points at the physical frame from victim. As a result, the attacker's thread dereferences that address using its own active ASID (rather than the victim's) and, thus, the attacker's ASID is the only tag that can match, and a victim's ASID would, in this case, simply result in a mismatch (a miss). Based on these first two scenarios, you will learn that the victim's ASID becomes the right tag only in the third way below, where the address resolves through the victim's own tables.

The third and heaviest way is a full **TTBR0_EL1** context switch, where the attacker repoints the CPU at the victim's pmap (one again, it is a kernel structure that represents the hardware translation layer for a given process's memory), as if the attacker's thread had briefly become a victim, performed the read or write operation, and then returned. As readers can realize, this is the only one of the three where ASID processing is truly essential, and it is the context that the following paragraphs explain and provide details.

It is the appropriate time to define the ASID introduced earlier. On AArch64 the ASID is either 8 or 16 bits wide, with the width selected by **TCR_EL1.AS** (0 = 8-bit, 1 = 16-bit, and 16-bit mode is available only when **ID_AA64MMFR0_EL1.ASIDBits** field reports that the hardware supports it (readers can check it on <https://support.arm.com/documentation/111107/2026-03/AArch64-Registers/ID-AA64MMFR0-EL1--AArch64-Memory-Model-Feature-Register-0>). **TCR_EL1**, which means the Translation Control Register for EL1 (<https://support.arm.com/documentation/100403/0300/register-descriptions/aarch64-system-registers/tcr-el1--translation-control-register--el1?lang=en>), is the 64-bit AArch64 system register that configures how the MMU executes the first stage of address translation for the **EL1&0** regime (kernel and userland virtual-to-physical translation), which supplements the two TTBRs, where **TTBR0_EL1** and **TTBR1_EL1** hold the page-table base addresses, while **TCR_EL1** holds the parameters that govern how those tables are walked and how virtual addresses are interpreted.

The value the MMU uses for TLB tagging is contained in the ASID field of **TTBR0_EL1** or **TTBR1_EL1**, where **TCR_EL1.A1** selects which of the two registers supplies the ASID that covers the entire translation regime for **EL1&0**. In this context, it is important to mention that it does not come from **CONTEXTIDR_EL1** because on AArch64 that register is a context identifier managed by software and consumed by debug and trace engine and not the translation ASID. On Apple silicon, XNU assigns one ASID per Mach pmap, where the **pmap_create_options** function calls the **alloc_asid** function (through the **pt_ops->alloc_id** function pointer from pmap), and it generates a free ASID from a kernel bitmap, where ASID 0 is excluded from the pool and never assigned to a user process.

As a direct consequence on the context switch technique, the base and the ASID must be swapped together (and that's the point: together) because repointing **TTBR0_EL1** at the victim's tables while leaving the attacker's ASID active (as in the previous scenario) would let the TLB serve pre-existing entries tagged by the attacker for the same virtual addresses instead of walking the victim's tables, so the attacker would silently read its own memory.

To provide readers with a comparative of two cases (and that I built for myself when I read this subject at the first time), where the main difference (and decisive aspect) is the ASID tag carried into a single TLB lookup, the sequence that shows involved details follows as shown below:

Wrong approach | swap the base only, leave the attacker's ASID active:

01. TTBR0_EL1 base == victim pmap (changed) | active ASID == attacker (left unchanged).
02. TLB lookup (victim virtual address, attacker ASID).
03. A stale attacker entry for that virtual address still matches the attacker ASID → TLB hit, returns the attacker's PA (Physical Address), and that is definitely not useful.
04. The victim's fresh tables are never walked → the attacker reads its own memory (useless).

Right approach | swap the base AND the ASID together:

01. TTBR0_EL1 base == victim pmap (changed)
02. active ASID == victim (changed too)
03. TLB lookup (victim virtual address, victim ASID)
04. The attacker's stale entries carry the wrong ASID and cannot match → TLB miss.
05. The MMU walks the victim's tables, and the attacker reads the victim's memory.

Based on concepts presented above, it is clear to us that switching the CPU context to the victim's pmap, reading, or writing, and switching back is one standard way to turn a kernel read-write primitive into the ability to read and write another process's user memory. The interaction followed by the physmap route, as also described above, is likely the other one, and it is the one most public iOS exploits actually use because it asks for nothing of the system beyond the primitive the exploit already holds.

A relevant aspect of the presented technique is that this does not allocate new mappings, perform CPU context switching, or access either **TTBR0_EL1** or any ASID, so there is no risk of errors in the base to ASID association, which is a decisive factor for PPL (A12–A14 and M1) and SPTM (A15+ and M2+) hardware. As a result, it never writes a page table descriptor, so it never has to defeat the page table monitor that guards those writes. Every access simply uses the kernel read-write primitive and the physmap, both already in hand.

Once again, concepts and facts exposed above are a valid review on a subject that might not be reviewed in daily work, but it is important to learn when you extend your research on iOS and touch in multiple "intersections" of iOS kernel and hardware. Based on recent excellent iOS researches, we have seen a lot of focus and concern on layers out of the kernel, such as STPM, TXM, Exclaves and coprocessors, which provide you with reasons for get interested in learn such details.

Stepping back from the exploitation detail, which I think we discussed enough, it is worth understanding where the pmap is placed in this context because there are three layers cooperate to turn a virtual address into a physical one and the pmap is the middle one, and the comprehension of how they interact to each other is valuable for the learning, as shown below:

- The top layer is `vm_map`, and it is machine independent. The `vm_map` is a kernel structure, which represents the list of contents of one address space that records which virtual addresses ranges are mapped, what object backs each of them, and with what associated protections. In this context, when a thread faults on a page that is not yet backed (typical activity from any virtual memory management), the fault handler `vm_fault` function finds the region and resolves what belongs there. If the resolution has been succeeded then it calls another function, named `pmap_enter`, to install (or setup) the mapping. Therefore, the `vm_map` structure determines what is mapped and where and indirectly makes part from the hardware page table process shown above.
- The middle layer is represented by the `pmap`, and it is machine dependent. A `pmap` (physical map) is also a structure and holds the hardware page tables for a single address space. While the `vm_map` structure is responsible for the decision, the `pmap` layer performs it. Thus, the function `pmap_enter` writes the arm64 page table descriptors, which the hardware will later read, and at its top there is a single root table that every walk starts from. The `pmap` records the physical address of that root table in a field named `ttep` (`ttep` means Translation Table Entry Pointer, which is used during multi-level page table interaction). For user-space tasks, this address is loaded by the kernel into the `TTBR0_EL1` register so the hardware knows where to begin its translation walk for the user half of the address space. On the other hand, `TTBR1_EL1` points to a permanent, global root table initialized at boot to handle the kernel half of the address space, as expected.
- The bottom layer is the MMU, which is the hardware itself. On a TLB miss, it takes the virtual address and interacts with the tables, starting from the root that the `TTBR` holds. Once it arrives at the physical address (by checking the permissions along the way), it caches the finished translation in the TLB, tagged with the active ASID, so later access to the same page can skip the interaction.

Now that we have the minimum memory management foundations are reasonably explained, it is time to return to the practical consequence of 16 KB pages of Apple Silicon, which cause side effects into alignment and exploitation techniques. From our security perspective, heap grooming primitives treat 16 KB as their unit, where a primitive that needs to land a victim object on a fresh page allocates in multiples of 16 KB, and expects that the boundary that allocator returns will fall exactly on a page edge. Actually, readers can make short experiments using `vm_region_recurse_64` function, which is the Mach function that enumerates each mapped region's base address, size, and protection, to walk (I am not sure if I could use a better word) the virtual memory layout from a task and also use `getpagesize` function to obtain the approach page size.

The 16 KB page size configuration, which we have discussed so far, is shown in the experiment below with a captured demonstration on the lab vPhone. The test program has to be built and signed ad-hoc on the macOS (in my case, an M2 or M5 machine) because the guest (vPhone) carries no compiler of its own. Therefore, running it via SSH, we have:

```
$ cat > pagesize_test.c <<'EOF'
```

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    long sz = sysconf(_SC_PAGESIZE);
    printf("sysconf(_SC_PAGESIZE) = %ld\n", sz);
    printf("getpagesize()          = %d\n", getpagesize());
    return 0;
}
```



```
}  
EOF  
  
$ xcrun -sdk iphoneos clang -arch arm64 -o pagesize_test pagesize_test.c  
$ ldid -S pagesize_test  
  
$ cat pagesize_test | ssh -p 2222 root@127.0.0.1 'export PATH=/var/jb/usr/bin:/var/jb/bin:$PATH;  
cat > /var/jb/pagesize_test; chmod +x /var/jb/pagesize_test; /var/jb/pagesize_test'  
  
sysconf(_SC_PAGESIZE) = 16384  
getpagesize()         = 16384
```

So far, we have discussed multiple concepts and also mentioned PAC (Pointer Authentication) along explanations, which really makes part of the context, but I reserved the next section to go into a few further details about PAC before we return to more practical aspects of iOS security.

06. Pointer Authentication (PAC): further words

This is another dense section, where we are discussing further details about PAC (Pointer Authentication), which it has already been mentioned previously. While I personally do not like to delve into many details on a topic at once because it might not be an effective approach, it is better to do it right now and avoid returning to it later when we will be handling with other relevant and really practical topics on iOS. Therefore, I recommend readers pay attention to key facts, and do not run over the section. On the other hand, this section must be used as foundation, and it is not necessary to remember details.

Pointer Authentication (PAC) was formally introduced under the feature name **FEAT_PAuth** with the ARMv8.3-A architecture extension in 2016, and its first commercial deployment happened in 2018 with Apple's A12 processor. Basically, it represents the cryptographic integrity check on protected pointers that Apple silicon enforces from the A12 generation onward.

Likely, PAC is the single mitigation that most fundamentally changed iOS exploitation between the pre-A12 era, when overwriting a kernel function pointer with a value controlled by attackers was the standard approach from a kernel read-write primitive to kernel code execution, and in the post-A12 era, when any attempt to overwrite a protected pointer leaves a value the CPU cannot authenticate, and finally any use of it fails. As expected, the exploitation scenario has changed significantly because available options became either to try to reuse an already valid signed pointer that points where the attacker wants control, or to pivot to a data-only strategy that never touches a protected pointer, which has been the most used approach adopted by attackers.

The next paragraphs provide readers with additional background on PAC at a reasonable and enough depth and hopefully will provide a minimal understanding of the above-mentioned bypass categories and associated CVEs in later articles.

In the standard Apple configuration (**TCR_EL1.TOSZ = 17**), the translated portion of a 64-bit virtual address is 47 bits, which leaves 17 high bits above the address space. The architecture supports an optional **Top-Byte-Ignore (TBI)** feature that makes the MMU ignore the top 8 bits (bits 63 through 56) during translation, freeing them for software metadata.

As readers could guess, part of these free bits could be used for other purposes, and when Pointer Authentication is active, it claims the unused high bits between the top of the translated range (bit 47) and either bit 54 (when TBI is enabled, since bits 63 through 56 belong to TBI and bit 55 is preserved for **TTBR0/TTBR1** selection) or bit 62 (when TBI is disabled, with bit 63 preserved as the sign extension of the address). PAC replaces those bits with a cryptographic signature computed over the pointer's lower bits, a context-specific 64-bit modifier called the discriminator, and a 128-bit key held in a CPU system register.

With Apple silicon configured for a 47-bit virtual address and TBI disabled, the PAC signature occupies on the order of 15 or 16 bits in the unused high bits above the virtual address range and, initially, a 16-bit signature might be vulnerable to brute force, but the attack is viable only if there is a code path that reveals whether a guess authenticated successfully without immediately dereferencing the pointer.

On earlier silicon that lacks **FEAT_FPAC** (this feature will be discussed in the next paragraphs), a failed authentication does not fault at the AUT instruction itself and, instead, the instruction silently corrupts (or poisons) the pointer by flipping its high bits into an invalid address, and the fatal translation fault lands only later when the program dereferences the poisoned value. As readers already noticed, there is a gap between the time of corruption (poisoning) and the eventual crash (kernel side), and this is the window an attacker exploits. In a few words, the fact is that whether the AUT instruction produced a valid address or a poisoned one, the attacker learns whether the guessed signature was correct without causing an immediate crash, so this procedure could be repeated (brute-forced again) until the signature is found.

As I have just mentioned, **FEAT_FPAC**, which is an optional extension that requires **FEAT_PAuth2** and ships on Apple A15 and later, helps to close one half of that window because it forces the AUT instruction to trap synchronously the instant it finds a bad PAC signature, so a wrong guess takes the procedure to an immediate fault that causes a process kill in userland and a full kernel panic in kernel context. As PAC keys are re-generated on every reboot, a single wrong attempt in kernel context panics the system, reboots it, re-randomizes the keys again, which eliminates any progress up to that point, and collapses attempts of brute-force search attack. Indeed, there is a second half of the window to attack PAC, but it seems to be kind of too complicated stuff to this article and completely out of scope.

PAC operates by writing a Message Authentication Code (MAC) into the high bits of a pointer, the discriminator (a 64-bit modifier the compiler chooses per signing site), and a 128-bit key held in a CPU system register. Therefore, there is not a single value, but three distinct components. That term "signing site", which might be strange at first, names a single place at the compiled code where the compiler emits a PAC sign instruction to protect one specific pointer for one particular use, and it is in this context that "signing site" is relevant.

At each signing site the compiler fixes two subcomponents, which the first is the key (an instruction key for a pointer that will subsequently be diverted to or a data key for one that will be stored and trusted later) and the second is the discriminator, and that is something that people sometimes overlook without granting the due importance. The compiler picks the discriminator to encode the pointer at that spot (the signing site), and it is constantly derived from the type of pointer or field, and that is often blended with the address of the slot the pointer lives in. As a result, the signature binds the pointer both to what it is and to where it is placed, which this last one is, in my opinion, the real catch and the real value of the mechanism.

A pointer signed at one site only authenticates at a site that presents the identical key and discriminator (the overlooked fact that I mentioned), and this is what stops an attacker from "copying" an already validated signed pointer out of one slot and dropping over another one. From this point of view, which is exactly where you can see the beauty and engineering of the mechanism, which evidently will work against us, vulnerability researcher and exploit developers, but it plays an important role in iOS (and macOS) security.

When code reads a pointer signed with PAC back from memory and intends to use it (typically as an indirect call target or as the basis for a subsequent load through the pointer), the code first authenticates the pointer at the matching authentication site and either reveal the original pointer (success case) or produces a corrupted (poisoned) pointer (unsuccess case), which will trigger a translation fault on the next access through the pointer (processors A12 to A14) or traps synchronously and the CPU raises a synchronous data abort or instruction abort with an ESR (Exception Syndrome Register) that names the PAC failure as the cause, on Apple silicon A15 and later. In a concise way, which an AUT instruction does is the following:

01. it recomputes the MAC (Message Authentication Code) over the pointer value + modifier using the same key.
02. If the MAC matches (success), the high bits are restored to their original extension values, and the original pointer comes back too.
03. If the MAC does not match (non-success) and FEAT_FPAC=0, the pointer is poisoned (corrupted) and the next access using this pointer will cause a translation fault.
04. If the MAC does not match (non-success) and FEAT_FPAC=1, the AUT instruction traps, which generates a process kill in userland and a full kernel panic in kernel context.

By the way, ESR (Exception Syndrome Register) is a hardware register that explains what crashed (reason), and FAR (Fault Address Register), which readers will see alongside ESR, provides us information on where (address) that application tried to access illegally. We will return to this subject later.

The PACMAN attack, by Ravichandran and colleagues at ISCA 2022 (<https://pacmanattack.com/paper.pdf>), which was demonstrated against Apple M1 silicon, it is a clear example of well-succeed attack against PAC. No doubt, it is tempting to conclude that a machine with FEAT_FPAC has closed that attack, since the trap now arrives immediately, but as I have stated previously, it is not true because it is a complex topic and right now it is far away from our scope.

In terms of PAC keys, there are five of them, which I have mentioned previously, but this time I will review them with further details, as shown below:

- **IA** and **IB** are the two instruction keys, which are used for signing pointers that will eventually be used by the code, such as function pointers, return addresses, and virtual method table entries.
- **DA** and **DB** are the two data keys, which are used for pointers stored in data structures, and that should be kept valid, such as C++ vtable pointers stored in objects allocated in heap, ObjC isa pointers (they notify the runtime on what class the object belong to), and kernel structure fields that store kernel structure pointers.
- **GA** is a generic key used by the PACGA instruction to compute a 32-bit MAC over a 64-bit input value, and whose mechanism is different from the other two. Basically, PACGA returns the MAC in a register instead, so whoever computes one must store it somewhere and compare it explicitly later.

From mentioned so far, while both IB and DB key are generated per process, both IA and DA keys are shared across process running on system. On the other hand, keys are shared system-wide for the kernel code, a single set derived at boot, and which is not rotated. As I highlighted previously, such keys can be read from kernel context but reading them is not enough to forge a signature due to fact that any forging activity requires executing a signing instruction at the exact place, and even getting a data-only write primitive, it cannot execute anything.

The discriminator (a 64-bit value), which is the modifier the compiler passes to each PAC instruction (it is the second operand of **PACIA**, **AUTDA** and other ones), is a relevant component that strengths the PAC protection at a given signing site (the location where the code is being executed) and make harder to an attacker to replace one already validated signed pointer by another one.

As examples, I can mention the usage of discriminators return addresses, C++ vtable pointer, C++ vtable entry, ObjC method list entries and kernel pointers. Its build comes from a combination (a blend would be a better word, maybe) of where the pointer lives (like a pointer to pointer) and a 16-bit constant representing what the pointer is for, and which is computed at the compile time. Therefore, all these examples involve the respective location of where a pointer lives, but an exception would be the return address, where the stack pointer is an address, and the whole value (there is not constant).

Additionally, constants themselves are not secrets, and they are derived from a string being hashed, and readers can check them by using a short program as shown below:

```
// const_disc.c -- recover a constant discriminator
```

```
#include <stdio.h>
#include <ptrauth.h>
#define D(s) printf("  %-30s 0x%04x\n", s, (unsigned)__builtin_ptrauth_string_discriminator(s))
int main(void)
{
    D("_ZTV4Base");           /* the vtable of a class named Base */
    D("_ZN4Base5alphaEl");    /* Base::alpha(long) */
    D("_ZN4Base4betaEl");     /* Base::beta(long) */
    D("_ZTV15OSMetaClassBase"); /* the root class of IOKit */
    D("_ZTV9IOService");       /* IOService, one level down */
    D("filedesc.fd_ofiles");   /* an XNU annotated struct field */
    D("fileproc.fp_glob");     /* another */
    return 0;
}
```

```
$ xcrun -sdk iphoneos clang -arch arm64e -O1 -o const_disc const_disc.c && ldid -S const_disc
```

./const_disc	# it has run on the phone
_ZTV4Base	0x2fa7
_ZN4Base5alphaEl	0xc8c5
_ZN4Base4betaEl	0xe1ba
_ZTV15OSMetaClassBase	0xcd1a
_ZTV9IOService	0x41d6
filedesc.fd_ofiles	0xac9d
fileproc.fp_glob	0x0c75

Of course, each one of these constants are only a piece of the equation, and not the value that the instruction actually consumes because a major part of the discriminator is an address that does not exist until the program run, so the reach of this code is extremely narrow, and much more than it seems to be because it does not cover constants derived from a type, for example. On the other hand, a few values can be really important for us because 0xcd1a is OSMetaClassBase, which is the root of the IOKit class hierarchy, and 0x41d6 is IOService, which is place some levels below it, so we can identify them in the

shipping kernel by checking **movk** instructions and understand what the discriminator really protects. In fact, there are other multiple other details, and one of them is that a vtable pointer discriminator names the class that introduced the pointer, and the class of the object, which can hide interesting details and subclasses.

As a result, the discriminator scheme is what gives PAC its security beyond raw MAC strength, and even attackers who somehow learn one valid signed pointer, they cannot reuse it as a different signed pointer, because, as explained, the discriminator expected at the target signing site is different. The published bypass literature on PAC has focused heavily on signing sites where the discriminator scheme is weak, such as repeated zero discriminators, signing sites that share a discriminator with signing sites that are controllable by attackers, and locations where the discriminator depends on input data controlled by attackers.

Before proceeding to a few related instructions, you should remember that IA and DA (instruction and data pointers, respectively), are shared across processes while IB and DB (instruction and data pointers, respectively) are per-process (no shared). GA, which computes a MAC, is shared across processes.

The ARMv8.3 PAC instruction set distinguishes operations by key and target type, as shown below:

- **PACIA Xd, Xn** and **PACIB Xd, Xn** sign a pointer in Xd using the IA or IB key with the modifier in Xn, or the convenience encoding **PACIASP / PACIBSP** which use SP as the modifier for the link register.
- **AUTIA Xd, Xn** and **AUTIB Xd, Xn** verify the signature on a previously signed pointer.
- **BLRAA Xn, Xm** and **BLRAB Xn, Xm** perform a combined authenticate and branch with link actions on a single instruction (IA or IB key respectively, modifier in Xm), where such a combined form is the standard operation for indirect calls through signed function pointers, and the main reason is because it leaves no window between authentication and branch in which the authenticated pointer is observable to other code.
- **RETAA** and **RETAB** perform combined authenticate and return operation at function exit using the stack pointer as the modifier. **RETAB** is the arm64e exit instruction for applications compiled in the Apple environment and matches PACIBSP at function entry.
- Data-key variants **PACDA, AUTDA, LDRAA** and **LDRAB** parallel the instruction-key variants for data pointers, with LDRAA and LDRAB performing a combined authenticate and load operation that is the mechanism for reading and using a signed data pointer in one instruction.
- **PACGA Xt, Xn, Xm** computes a 32-bit MAC over the value in Xn under the modifier in Xm using the GA key and writes it into the upper 32 bits of Xt (zeroing the low 32), but without modifying Xn or Xm.
- There are other instructions suffixed with “Z” which as **PACIZA, AUTIZA, PACDZA, AUTDZA, BRAAZ, BRABZ, BLRAAZ** and **BLRABZ**, and all of them uses a register fewer. As an example of comparison, while the instruction **BLRAA x8, x9** authenticates under the value stored into x9, the instruction **BLRAAZ x8** (just one operand) authenticates x8 register under a modifier of zero.

All of these instructions will appear frequently while you are doing iOS research, and some of them will be similar to the ones shown below:

- **pacibsp ... retab (function/routine limits):** This pair of instructions signs and authenticates the return address (x30) using the Stack Pointer (sp) as a modifier. This provides robust Control Flow Integrity (CFI) by preventing attackers from modifying the return address on the stack to hijack execution flow.
- **blraa x8, x9:** this instruction authenticates the pointer inside x8 using instruction key A (IA) and the modifier value stored in x9 and finally performs an indirect call.
- **blraaz x8:** this instruction authenticates the pointer inside x8 using instruction key A (IA) and makes an indirect call, but under a zero modifier, so there is no type-diversity here (a unique cryptographic value based on the signature or type of the function).
- **braa x2, x16:** this instruction authenticates and makes a tail-call (occurs as the final action of a function). Basically, most standard routines call save the return address into the LR (Link Register -- x30) to make it possible for the program return, but in this case x30 is not changed, and then the program to execute ret instruction, it will jump to the original function. This is almost always used in virtual calls.
- **autda x16, x17:** this instruction takes the signed data pointer from x16 and authenticates using data key A (DA) and a modifier value given by x17.
- **ldraa x2, [x16, #n]:** this instruction authenticates the register inside x16 using a modifier of zero and data key A, adds the offset #n and loads the 64-bit value from memory address into x2 register.
- **xpacd / xpaci x17:** these instructions remove the PAC signature assuming that the pointer targets are data and executable code, respectively.
- **movk x17, #imm, lsl #48:** this instruction inserts an immediate value directly into the highest 16 bits of the register x17 but keeps the lower 48 bits. Thus, it is usually to load the 16-bit constant (type discriminator) into the upper bits of the given register.

PAC is a complex topic, and ARM processors provides a series of features that improves the original proposal (**FEAT_PAuth**) in terms of security, where available options are:

- **FEAT_PAuth2:** An enhancement of the PAC algorithm that strengthens the cryptographic association between a pointer and its MAC, where two specific changes versus the **FEAT_PAuth** baseline matter and make a difference to a reader.
- **FEAT_FPAC:** FPAC means "Faulting Pointer Authentication Code", which is one of features that produces better impact against exploitation because it turns a PAC verification failure into an immediate synchronous fault rather than a poisoned (corrupted) pointer, which closes the security gap where a failed authentication instruction silently produced a bad pointer instead of trapping.
- **FEAT_FPACCOMBINE:** this feature extends the faulting behavior offered by FPAC to combined instructions (such as RETAA and BRAA), which authenticate and branch in a single step. Thus, without using FPACCOMBINE, those combined instructions might not be faulty on a bad PAC even when standalone AUT instructions do.
- **FEAT_FPACC_SPEC:** It is an improvement on the previous features (FEAT_FPAC and FEAT_FPACCOMBINE), and mainly because while the processor will fault and halt execution after a failure in the PAC verification, this fault occurs after the hardware has already looked ahead. This feature protects against out-of-order execution (speculative attacks via gadgets) and prevents

attackers from guessing a PAC signature by exploiting cache timing differences, which would otherwise offer attackers an oracle to execute brute force attempts against PAC keys.

- **FEAT_BTI (Branch Target Identification)**: it is a mechanism that works complementarily to PAC because it provides protection against a different attack class as indirect branches, whose target address has been not authenticated using PAC. Basically, there is a small number of indirect branch sequences that do not pass through AUT instructions such as early boot sequences before PAC keys are loaded, compiler generated trampolines whose AUT instruction is omitted by the optimizer, and certain jump table dispatchers that load their targets from a pre-validated table, where all them allows an attacker to hijack the execution flow and land anywhere in the executable portions of the kernel, which opens possibilities for classic attacking techniques like JOP (Jump-Oriented Programming) via gadget chains. Once BTI is active, every indirect branch target must begin with one of three pre-determined instructions: BTI c (call targets, accepting BLR), BTI j (jump targets, accepting BR), or BTI jc (both). Therefore, considering these three possibilities, any indirect branch that lands on an instruction other than a matching BTI landing pad faults synchronously, and the immediate side-effect is to limit the possibility of attackers using gadgets from middle of functions because they will no longer be reachable.

At this point of the article, I have not shown how to configure some components of the lab like vPhone yet, and such a resource allows us to query about these protections easily by executing the following command:

```
$ sshpass -p alpine ssh -p 2222 root@127.0.0.1 \  
'export PATH=/var/jb/usr/sbin:/var/jb/usr/bin:/var/jb/bin:$PATH;  
  for k in hw.optional.arm.FEAT_PAuth hw.optional.arm.FEAT_PAuth2 \  
          hw.optional.arm.FEAT_FPAC hw.optional.arm.FEAT_FPACCOMBINE \  
          hw.optional.arm.FEAT_BTI kern.exclaves_status \  
          hw.machine hw.cputype hw.cpusubtype; do  
    sysctl $k 2>/dev/null  
  done'  
  
hw.optional.arm.FEAT_PAuth: 1  
hw.optional.arm.FEAT_PAuth2: 1  
hw.optional.arm.FEAT_FPAC: 1  
hw.optional.arm.FEAT_FPACCOMBINE: 0  
hw.optional.arm.FEAT_BTI: 1  
kern.exclaves_status: 255  
hw.machine: iPhone99,11  
hw.cputype: 16777228  
hw.cpusubtype: 2
```

Of course, a real iPhone does not offer this interface, and we must develop an application to collect the same sequence of information. From the execution above, a few observations follow:

- **hw.optional.arm.FEAT_PAuth: (1)** pointer authentication is available to user-space code.
- **hw.optional.arm.FEAT_PAuth2: (1)** the enhanced pointer authentication extension (mandatory from ARMv8.6) is present.
- **hw.optional.arm.FEAT_FPAC: (1)** a failed AUT instruction faults synchronously rather than producing a poisoned pointer.
- **hw.optional.arm.FEAT_FPACCOMBINE: (0)** the extension that faults on combined instructions (RETAA, BRAA, LDRAA) is not present on this vPhone.

- **hw.optional.arm.FEAT_BTI: (1)** Branch Target Identification is active, which enforces landing pad constraints on indirect branches.
- **kern.exclaves_status: (255)** Exclaves subsystem status (255 = 0xFF = available).
- **hw.machine: (iPhone99,11)** the vPhone's device identifier.
- **hw.cputype: (16777228)** == 0x0100000C = CPU_TYPE_ARM (12) | CPU_ARCH_ABI64 (0x01000000) = CPU_TYPE_ARM64.
- **hw.cpusubtype: (2)** CPU_SUBTYPE_ARM64E (XNU machine.h:388), which is the arm64e ABI variant with pointer authentication.

Once again: these values come from the vPhone, which is paravirtualized on the Mac's M2 processor, and not from a physical iPhone. The CPU feature registers report (mirror) the real hardware values from M2 machine because the virtual machine executes natively on that silicon. They are genuine for the M2 but likely will differ from a physical A18 on features where the two chips diverge (SME, for example).

As I mentioned, if we want to do the same in a real iPhone paired to the macOS, the approach is different, and we should build a small application signed with a developer certificate and deploy it into the device. Fundamentally, the application calls the [sysctlbyname](#) function to read each necessary key from its own process, which show us clearly that is multi-step procedure, and that depends on the configured environment.

The environment that was used here (from a different machine, but at this time iOS 27.0 and Xcode 27.0 are available):

- Xcode 26.5 with iOS 26.x SDK, installed via the App Store.
- xcodegen: [brew install xcodegen](#)
- A paired iPhone 16: [xcrun devicectl list devices](#) (the device must show "available (paired)").
- Developer Mode enabled on the iPhone: [Settings](#) → [Privacy & Security](#) → [Developer Mode](#) → [On](#).
- An Apple ID signed into Xcode: [Xcode](#) → [Settings](#) → [Accounts](#). A free Personal Team is enough.

We should list the installed devices and also retrieving your details:

```
% xcrun devicectl list devices
```

Name	Identifier	State	Model	Reality
ABi12	<Identifier 1>	available (paired)	iPhone 12	physical
ABi16	<Identifier 2>	available (paired)	iPhone 16	physical
iPhone	<Identifier 3>	connected (no DDI)	iPhone99,11	virtual

You should note that UDID is directly associated with the hardware while UUID is only a computed identifier. Picking up one of these identifiers, you can retrieve further information about the device, as shown below:

```
% export DEVICE=2FC3AF78-2618-5DA3-89A6-35EE8BECA75B
% xcrun devicectl device info details --device "$DEVICE" 2>&1 | head -40
```

Gathering device information...

Current device information:

- Identifier: 2FC3AF78-2618-5DA3-89A6-35EE8BECA75B
- ▼ Connection:
 - Authentication Type: manualPairing
 - Device State: connected
 - Last Connection Date: 18 Aug 2026 at 22:05:00

- Pairing State: paired
- Screen Viewing URL: devices:///device/open?id=2FC3AF78-2618-5DA3-89A6-35EE8BECA75B
- Transport Type: wired
- Tunnel IP Address: fd61:55c3:f710::1
- Tunnel Transport Protocol: tcp
- ▼ Hardware:
 - CPU Count: 6 cores (6 logical, 1 package(s))
 - CPU Type: arm64e
 - Device Reality: physical
 - Device Type: iPhone
 - ECID: <redacted>
 - Has Action Button: true
 - Internal Storage Capacity: 128000000000
 - Marketing Name: iPhone 16
 - Platform: iOS
 - Product Type: iPhone17,3
 - Serial Number: <redacted>
 - Supported Biometrics: Face ID
 - ▼ Supported CPU Types:
 - arm64e
 - arm64
 - Supported Device Families: • 1
 - Supports Siri: true
 - Thinning Product Type: iPhone17,3
 - UDID: 00008140-0018<redacted>
- ▼ Software:
 - OS Build Update: 24A5418b
 - OS Version: 27.0
 - Supports Checked Allocations: false
- ▼ State:
 - Boot State: booted
 - Developer Mode Status: Enabled (1)
 - Device Name: ABi16

The next step is to create a clean project directory by running the following commands:

```
$ cd ~
$ mkdir -p ~/ios_research/mitigation_matrix/Sources
$ cd ~/ios_research/mitigation_matrix
$ pwd
$ ls -la

/Users/researcher/ios_research/mitigation_matrix
total 0
drwxr-xr-x  3 researcher  staff   96 Aug 19 21:16 .
drwxr-xr-x 27 researcher  staff  864 Aug 19 21:16 ..
drwxr-xr-x  2 researcher  staff   64 Aug 19 21:16 Sources
```

Now we have to create the [MitigationMatrixApp.swift](#) file as shown below:

```
$ ls -lh Sources
total 8
-rw-r--r--  1 researcher  staff   1.3K Aug 19 21:20 MitigationMatrixApp.swift

$ wc -l Sources/MitigationMatrixApp.swift
  50 Sources/MitigationMatrixApp.swift

$ cat Sources/MitigationMatrixApp.swift

import SwiftUI
import Foundation
import Darwin

@main
struct MitigationMatrixApp: App {
    init() { runProbe() }
```

```
    var body: some Scene { WindowGroup { Text("Probe complete.") } }
}

func readSysctl(_ name: String) -> String {
    var size: size_t = 0
    if sysctlbyname(name, nil, &size, nil, 0) != 0 { return "<not present>" }
    if size == 8 {
        var v: UInt64 = 0
        sysctlbyname(name, &v, &size, nil, 0)
        return String(v)
    } else if size == 4 {
        var v: UInt32 = 0
        sysctlbyname(name, &v, &size, nil, 0)
        return String(v)
    } else {
        var buf = [CChar](repeating: 0, count: size)
        sysctlbyname(name, &buf, &size, nil, 0)
        return String(cString: buf)
    }
}

func runProbe() {
    let keys = [
        "hw.optional.arm.FEAT_PAuth",
        "hw.optional.arm.FEAT_PAuth2",
        "hw.optional.arm.FEAT_FPAC",
        "hw.optional.arm.FEAT_FPACCOMBINE",
        "hw.optional.arm.FEAT_BTI",
        "hw.machine",
        "hw.cputype",
        "hw.cpusubtype",
    ]
    var out = ""
    for k in keys { out += "\(k): \(readSysctl(k))\n" }
    let docs = FileManager.default.urls(
        for: .documentDirectory,
        in: .userDomainMask).first!
    try? out.write(
        to: docs.appendingPathComponent("matrix.log"),
        atomically: true,
        encoding: .utf8)
}
```

The next step is to write the xcodegen project spec as a project.yml file as shown below :

```
$ cat > project.yml << 'EOF'
name: MitigationMatrix
options:
  deploymentTarget: { iOS: "17.0" }
settings:
  base:
    DEVELOPMENT_TEAM: F5FWZFTZ52
    CODE_SIGN_STYLE: Automatic
targets:
  MitigationMatrix:
    type: application
    platform: iOS
    sources: [ Sources ]
    settings:
      base:
        PRODUCT_BUNDLE_IDENTIFIER: com.alexandreborges.MitigationMatrix
        GENERATE_INFOPLIST_FILE: YES
        INFOPLIST_KEY_UILaunchScreen_Generation: YES
EOF

$ ls -lh
```

```
total 8
drwxr-xr-x  3 researcher  staff   96B Aug 19 21:19 Sources
-rw-r--r--  1 researcher  staff  432B Aug 19 21:35 project.yml
```

```
$ ls -lh Sources
```

```
total 8
-rw-r--r--  1 researcher  staff   1.3K Aug 19 21:20 MitigationMatrixApp.swift
```

A few settings are worth being commented. The first relevant field is **DEVELOPMENT_TEAM**, which is the identifier of the account whose certificate will sign the build, and you can get it when you enroll (and any free account have one). The **CODE_SIGN_STYLE** is set to Automatic, and this asks the toolchain to obtain a provisioning file for our project rather than requiring that one to be created by hand. Another relevant field is **PRODUCT_BUNDLE_IDENTIFIER** that represents the application name and must be unique. There are two points that must be observed here. First, you must use your own **DEVELOPMENT_TEAM**, and to retrieve it, execute the following command:

```
$ security find-certificate -c "Apple Development" -p ~/Library/Keychains/login.keychain-db |
openssl x509 -noout -subject
```

```
subject= /UID=G2GV2XJ46C/CN=Apple Development: alexandreborges@<redacted>
(36P4JHYL7A)/OU=F5FWZFTZ52/O=Alexandre Borges/C=US
```

The second point is that the bundle identifier is the reverse DNS string that we choose for the application, and as suggestion you could use your name here. Obviously, if the app ever shipped on the App Store for some reason, so every iOS user could read it.

To the next step I am going to use **xcodegen** command, which should have already been installed (check it using **xcodegen --version** command). Once we confirmed that it is installed, we can move forward and generate the Xcode project from **project.yml** file as shown below:

```
$ xcodegen --version
Version: 2.45.4
```

```
$ xcodegen generate
Generating plists...
Generating project...
Writing project...
Created project at /Users/researcher/ios_research/mitigation_matrix/MitigationMatrix.xcodeproj
```

Now it is time to build the project:

```
$ security unlock-keychain ~/Library/Keychains/login.keychain-db
```

```
password to unlock /Users/researcher/Library/Keychains/login.keychain-db:<password>
```

```
$ xcodebuild -project MitigationMatrix.xcodeproj -scheme MitigationMatrix -configuration Debug -
destination 'platform=iOS,id=<udid>' -allowProvisioningUpdates build 2>&1 | tail -20
```

```
....
/Users/researcher/Library/Developer/Xcode/DerivedData/MitigationMatrix-
hlgrqshdrpixbztmviewphimxzsu/Build/Products/Debug-iphonios/MitigationMatrix.app -shallow-bundle -
infoplist-subpath Info.plist
warning: All interface orientations must be supported unless the app requires full screen. (in
target 'MitigationMatrix' from project 'MitigationMatrix')
```

```
** BUILD SUCCEEDED *
```

The id value used in the **xcodebuild** command is the iPhone hardware UDID value, which was retrieved previously using **xcrun devicectl device info details --device "\$DEVICE" 2>&1 | head -30** command.

If you want, check the structure of application bundle we built and the respective binary architecture:

```
$ find MitigationMatrix.app -maxdepth 1 | sort
MitigationMatrix.app
MitigationMatrix.app/Info.plist
MitigationMatrix.app/MitigationMatrix
MitigationMatrix.app/MitigationMatrix.debug.dylib
MitigationMatrix.app/PkgInfo
MitigationMatrix.app/_CodeSignature
MitigationMatrix.app/___preview.dylib
MitigationMatrix.app/embedded.mobileprovision

$ lipo -info MitigationMatrix.app/MitigationMatrix
Non-fat file: MitigationMatrix.app/MitigationMatrix is architecture: arm64

$ file MitigationMatrix.app/MitigationMatrix
MitigationMatrix.app/MitigationMatrix: Mach-O 64-bit executable arm64
```

Check the signature itself by running the following command:

```
$ codesign -dvv MitigationMatrix.app

Executable=/Users/researcher/Library/Developer/Xcode/DerivedData/MitigationMatrix-
hlgrqshdrpixbztmviewphimxzsu/Build/Products/Debug-iphonios/MitigationMatrix.app/MitigationMatrix
Identifier=com.alexandreborges.MitigationMatrix
Format=app bundle with Mach-O thin (arm64)
CodeDirectory v=20400 size=520 flags=0x0(none) hashes=5+7 location=embedded
Signature size=4811
Authority=Apple Development: alexandreborges@blackstormsecurity.com (36P4JHYL7A)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=20 May 2026 at 18:12:46
Info.plist entries=21
TeamIdentifier=F5FWZFTZ52
Sealed Resources version=2 rules=10 files=3
Internal requirements count=1 size=224
```

The CodeDirectory line shows "hashes=5+7" that means five hashes cover the file's pages and seven hashes covers other items such as entitlements and the resource seal.

Checking the entitlements (permissions), we have:

```
$ codesign -d --entitlements - --xml MitigationMatrix.app | plutil -convert xml1 -o - -

Executable=/Users/researcher/Library/Developer/Xcode/DerivedData/MitigationMatrix-
hlgrqshdrpixbztmviewphimxzsu/Build/Products/Debug-iphonios/MitigationMatrix.app/MitigationMatrix
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">
<plist version="1.0">
<dict>
  <key>application-identifier</key>
  <string>F5FWZFTZ52.com.alexandreborges.MitigationMatrix</string>
  <key>com.apple.developer.team-identifier</key>
  <string>F5FWZFTZ52</string>
  <key>get-task-allow</key>
  <true/>
</dict>
</plist>
```

From the output above:

- **application-identifier**: it the application name, prefixed by the team.
- **team-identifier**: it is a record that indicates who built it.

- **get-task-allow**: it is the only permission, and it permits a debugger to attach to this process. Although it is present because it is a development build, it is absent from anything shipped through the Apple Store, as expected.

Using the same way, you can examine the provisioning profile, which contains a multitude of information:

```
$ security cms -D -i MitigationMatrix.app/embedded.mobileprovision > /tmp/matrix.plist
$ plutil -p /tmp/matrix.plist

{
  "AppIDName" => "XC com alexandreborges MitigationMatrix"
  "ApplicationIdentifierPrefix" => [
    0 => "F5FWZFTZ52"
  ]
  "CreationDate" => 2026-08-20 01:22:03 +0000
  "DER-Encoded-Profile" => {length = 3471, bytes = 0x30820d8b06092a864886f70d010702a0 ...
0cf944486ceeca06}
  "DeveloperCertificates" => [
    0 => {length = 1512, bytes = 0x308205e4308204cca003020102021029 ... 525fde417b9a6c30}
  ]
  "Entitlements" => {
    "application-identifier" => "F5FWZFTZ52.com.alexandreborges.MitigationMatrix"
    "com.apple.developer.team-identifier" => "F5FWZFTZ52"
    "get-task-allow" => true
    "keychain-access-groups" => [
      0 => "F5FWZFTZ52.*"
    ]
  }
  "ExpirationDate" => 2026-08-27 01:22:03 +0000
  "IsXcodeManaged" => true
  "LocalProvision" => true
  "Name" => "iOS Team Provisioning Profile: com.alexandreborges.MitigationMatrix"
  "Platform" => [
    0 => "iOS"
    1 => "xrOS"
    2 => "visionOS"
  ]
  "ProvisionedDevices" => [
    0 => "00008140-001868412EEB001C"
  ]
  "TeamIdentifier" => [
    0 => "F5FWZFTZ52"
  ]
  "TeamName" => "Alexandre Borges"
  "TimeToLive" => 7
  "UUID" => "8f69abaf-3119-441b-a408-f6f82040c831"
  "Version" => 1
}
```

The TimeToLive field set to 7 (seven days) is because this is the profile lifetime of a free account, and once the application has expired we have to rebuild it again. If you have a paid account, you get a year as expiration time.

Once the application has been built successfully, it is time to install it:

```
$ APP=$(find ~/Library/Developer/Xcode/DerivedData -name 'MitigationMatrix.app' -path '*Debug-iphoneos*' -type d 2>/dev/null | head -1); echo "$APP"; xcrun devicectl device install app --device <device_uuid> "$APP" 2> >(grep -v 'provisioning paramter\|No provider was found\|devicectl manage create' >&2)
```

```
/Users/researcher/Library/Developer/Xcode/DerivedData/MitigationMatrix-
hlgrqshdrpixbzatmvewphimxzsu/Build/Products/Debug-iphoneos/MitigationMatrix.app
23:35:00 Acquired tunnel connection to device.
23:35:00 Enabling developer disk image services.
```

23:35:00 Acquired usage assertion.

App installed:

- bundleID: com.alexandreborges.MitigationMatrix
- installationURL: file:///private/var/containers/Bundle/Application/2F0958B2-2DDE-4454-9F4C-3AB2B87C4A8C/MitigationMatrix.app/
- launchServicesIdentifier: unknown
- databaseUUID: F26E1622-25CB-405E-BF67-7330BAF28B07
- databaseSequenceNumber: 2756
- options:

The `<device_uuid>` value is the `Identifier` value retrieved from the `xcrun devicectl list devices` command output. Moving forward, we need to trust the application on iPhone: **Settings** → **General** → **VPN & Device Management** → **tap your developer row** → **tap Trust** → **confirm**.

Finally, we run the application and retrieve the mitigations from iPhone 16:

```
$ xcrun devicectl device process launch \  
  --device <device_uuid> \  
  com.alexandreborges.MitigationMatrix \  
  2> >(grep -v 'provisioning paramter'|No provider was found|devicectl manage create' >&2)
```

```
sleep 3
```

```
xcrun devicectl device copy from \  
  --device <device_uuid from your iPhone device> \  
  --domain-type appDataContainer \  
  --domain-identifier com.alexandreborges.MitigationMatrix \  
  --source Documents/matrix.log \  
  --destination /tmp/matrix.log \  
  2> >(grep -v 'provisioning paramter'|No provider was found|devicectl manage create' >&2)
```

```
cat /tmp/matrix.log
```

```
23:53:20 Acquired tunnel connection to device.  
23:53:20 Enabling developer disk image services.  
23:53:20 Acquired usage assertion.  
Launched application with com.alexandreborges.MitigationMatrix bundle identifier.  
23:53:23 Acquired tunnel connection to device.  
23:53:23 Enabling developer disk image services.  
23:53:23 Acquired usage assertion.  
File received from Device  
/tmp/matrix.log  
hw.optional.arm.FEAT_PAuth: 1  
hw.optional.arm.FEAT_PAuth2: 1  
hw.optional.arm.FEAT_FPAC: 1  
hw.optional.arm.FEAT_FPACCOMBINE: 1  
hw.optional.arm.FEAT_BTI: 1  
hw.machine: iPhone17,3  
hw.cputype: 16777228  
hw.cpusubtype: 2
```

Results can be interpreted as:

- hw.optional.arm.FEAT_PAuth: 1 → PAC is exposed to the userland.
- hw.optional.arm.FEAT_PAuth2: 1 → Enhanced PAC is enabled.
- hw.optional.arm.FEAT_FPAC: 1 → Faulting on PAC failure is on.
- hw.optional.arm.FEAT_FPACCOMBINE: 1 → Combined fault extension is enabled.
- hw.optional.arm.FEAT_BTI: 1 → Branch Target Identification is exposed to user mode binary builds.
- hw.machine: iPhone17,3 → Real iPhone device
- hw.cputype: 16777228 → It means 0x0100000C == CPU_TYPE_ARM64

- hw.cpusubtype: 2 → CPU_SUBTYPE_ARM64E

If you want, it is possible to check the profile (replace <uuid> by the real value):

```
$ security cms -D -i ~/Library/Developer/Xcode/UserData/Provisioning\
Profiles/<uuid>.mobileprovision | plutil -p - | grep -E
'Name|UUID|TeamIdentifier|ExpirationDate|CreationDate|ProvisionedDevices' | head -20

"AppIDName" => "XC com alexandreborges MitigationMatrix"
"CreationDate" => 2026-08-17 01:22:03 +0000
"ExpirationDate" => 2026-08-27 01:22:03 +0000
"Name" => "iOS Team Provisioning Profile: com.alexandreborges.MitigationMatrix"
"ProvisionedDevices" => [
"TeamIdentifier" => [
"TeamName" => "Alexandre Borges"
"UUID" => "<redacted>"
```

No doubt, the entire procedure represents considerable work only to collect such values and configuration, but it is the way to collect such information from iPhones running stock iOS.

As reader will learn through this series, modern iOS exploits operate inside of kernel, and through a read-write primitive aiming to overwrite credential structures of a process, modify the trust cache, swap profile pointer to point to a correct one, where the goal is to elevate privileges, but advanced operations such as arbitrary page table manipulation, execution of unsigned code, breakdown of Exclaves, breakdown of component isolation and possibility of modifying user-space variables are strongly limited by monitors such as SPTM and TSM.

There are excellent articles that explain and handle PAC bypass, and they are summarized in the following types: signing oracle, discriminator weakness, speculative side channel (like PACMAN), data only, signing context configuration and key derivation weakness. Once again, all of these approaches are really interesting, but they do not fit in with this first article.

It is enough about concepts, and it is time to move to more practical and iOS focused sections.

07. Components of an iOS security research lab

This series is based on several practical experiments, so one of most basic and important points to be discussed is about the laboratory setup for doing iOS security research. As it depends on the preferences of each researcher and also the financial possibilities to buy devices, details below show how my particular environment is structured for this series of articles:

- **Workstation:**
 - This is a system for doing everything to conduct research such as code reading, writing harness and PoC coding. Basically, I use this system for cross-host orchestration, IDA Pro disassembling, decompiling, symbolication and binary diffing. Of course, readers might prefer doing everything directly on macOS, and that is a matter of preference.
 - As installed programs, the workstation has PowerShell 7, ripreg, IDA Pro 8.4 or 9.x, OpenSSH, Python 3.11+ and a Linux Ubuntu installed on WSL environment, which is really useful in different moments.

- My particular reason to have a separate host from the Mac host is more operational than architectural, because the source reading is an activity that benefits from a quiet desktop environment without interruptions and mainly with multiple screens, and the dynamic portion of the research is occurring on macOS system as well as the vPhone and Xcode are executing there, this kind of noise is not welcome, and I prefer this kind of isolation. Furthermore, my Windows system (workstation) has much more memory than my Mac machines (M2 and M5). Of course, you can run Linux or even macOS as primary host (workstation), and like I said, it is a matter of preference.
- This workstation keeps a clone of the public XNU repository <https://github.com/apple-oss-distributions/xnu>, which on my system is stored in C:\research_ios\xnu folder.
- As an indispensable tool, we must install Cellebrite `ida_kernelcache` and `ida_kcphp` plugins for IDA Pro: `git clone github.com/cellebrite-labs/`
- Finally, the workstation has blacktop `ipsw` tool installed as shown below:
 - `scoop bucket add blacktop https://github.com`
 - `scoop install blacktop/ipsw`
 - `scoop update ipsw`

- **Mac host:**

- This is the main research system because it is used for virtualizing the iOS (vPhone), compiling PoC and programs using Xcode, and where the macOS toolchain is installed.
- The recommendation is that you choose an M series machine due to Apple Silicon, which is used on iPhones. Moreover, it should have a minimum of 16 GB RAM (recommendation is 32 GB or more) and 1 TB of disk.
- As this system runs vPhone, macOS needs to be setup appropriately, where **SIP (System Integrity Protection)** must be disabled by booting macOS on Recovery Mode and using `csrutil disable` command. Additionally, is also necessary to enable the **research-guests** feature via `csrutil allow-research-guests enable` command, relax AMFI by setting NVRAM boot-args with `amfi_get_of_my_way=1` and rebooting the system. The practical step-by-step procedure will be shown later, but it is appropriate to mention that all these changes make macOS unsafe for daily operations because, for example, with SIP disabled, any malware on the Mac can modify the operating system itself, and with the research-guest feature enabled the kernel admits guest images that the production system would refuse.
- The following programs should be installed:
 - Xcode: `xcode-select --install`
 - IDA Pro (8.4 or 9.x): <https://hex-rays.com/ida-pro>
 - blacktop ipsw tool:
 - `brew install blacktop/tap/ipsw`
 - `brew update && brew upgrade blacktop/tap/ipsw` (updates)
 - Xcode Command-Line Tools: `xcode-select --install`
 - `ldid-procurus`: `brew install ldid-procurus`
 - `libimobiledevice`: `brew install libimobiledevice`
 - `vphone-cli`: `git clone https://github.com/Lakr233/vphone-cli`
 - Cellebrite `ida_kernelcache` and `ida_kcphp` IDA: `git clone github.com/cellebrite-labs/`

- blacktop/symbolicator signature library: `git clone github.com/blacktop/symbolicator`
- Frida: `brew install frida-tools`
- Wireshark: `brew install wireshark`

- **vPhone:**

- It is a live jailbroken iOS environment used for dynamic harness , kernel state, panic, and log observation.
 - I use vphone-cli project, and it is recommended you also adopt it because it brings a real work iOS for debugging: `git clone https://github.com/Lakr233/vphone-cli`
- Using the proposed configuration, I connect to it via SSH forwarding (port 2222) using root as username and "alpine" as password, which makes its usage simpler.
- The vPhone is the most convenient userland resource in this lab for executing arm64e binaries.
- Of course, the vPhone remains the easiest resource for arm64e research (our case) because it accepts any `ldid -S -signed` arm64e binary deployed over SSH, with no Xcode project, no provisioning profile, and no App Store review. Thus, the process to build and sign is direct: build with `xcrun -sdk iphoneos clang -arch arm64e`, sign with `ldid -S`, push to the vPhone, and run.
- vphone-cli also exposes a useful GDB remote serial protocol stub on a Mac TCP port that can be used for read-only kernel inspection.
- Although the vPhone configuration procedure has not been explained yet, a usual connection is something similar to the output shown below:

```
$ ssh -p 2222 root@127.0.0.1
```

```
root@iPhone:~# uname -a
Darwin iPhone 25.4.0 Darwin Kernel Version 25.4.0: Wed Feb  4 23:04:24 PST 2026;
root:xnu-12377.100.591.502.1~2/RELEASE_ARM64_VRESEARCH1 iPhone99,11 arm Darwin
```

```
root@iPhone:~# sysctl -n hw.model kern.osproductversion kern.osversion
VPHONE600AP
26.5
23F77
```

```
root@iPhone:~# id
uid=0(root) gid=0(wheel) groups=0(wheel)
```

- **iPhone:**

- It is definitive device for testing PoC and confirm vulnerabilities. My specific configuration here is to use two iPhones, one of them running the stock and current iOS version, and the other one running the last iOS beta (developer) that is available.
- Vulnerabilities can be submitted to Apple or a reliable broker, and website of Apple Security Bounty is <https://security.apple.com/bounty/>.
- A panic on the physical iPhone running the current build is captured by running `idevicecrashreport` command to pull the .ips panic log from the device (it is necessary to install it by running `brew install libimobiledevice` command). The log names are the latest

XNU revision in its `panicString` field, and such panic file is one of main artifacts for any submission.

- To illustrate the commented, pulling the panic log after a crash is a single command from the Mac, using `idevicecrashreport` command, as shown below:

```
$ mkdir -p ~/research/panic_logs  
$ idevicecrashreport ~/research/panic_logs/
```

Move: panic-full-2026-08-15-220431.ips

- The default procedure is to move (not copy) each crash report from the device to the specified output directory on macOS, so the store from iPhone is cleared and releases some space.
- If readers want, you can use `-k option` to keep reports on the device instead of moving and add `-e option` to also extract the raw crash text into a separate `.crash file` alongside the `.ips file`.
- The resulting `.ips file` is a JSON document whose `panicString field` contains the full kernel panic text, which includes the caller address, the ESR and FAR (both already commented previously), the XNU build string, and the stack, and it is the artifact you can use to a bounty submission, for example.
- The typical workflow after triggering a panic on the test iPhone is to wait for the device to reboot, run the command above, then to inspect the `.ips file` with `cat` command or any Json tool to confirm the panic matches the expected bug signature. Additionally, I have written a few scripts in Python to do quick triage.
- The recommended iPhone model is always recent ones (two or three years old) because they support all modern protections implemented by Apple such as SPTM, TXM and Exclaves, for example. Personally, I always keep an old device (in my case, an iPhone 12) to make comparison while running PoCs and exploits.

On macOS, to check paired iPhones on macOS, execute:

```
$ idevice_id -l
```

```
00008101-0008<redacted>  
0000FE01-D8FB<redacted>  
00008140-0018<redacted>
```

The **UDID (Unique Device Identifier)** shown above is a 24 hexadecimal value, formatted in two groups separated by a hyphen after the first 8 digits, where the first 8-character group (ChipID) identifies the chip family (00008140 corresponds to the A18 | T8140 family on the iPhone 16, and 00008101 corresponds to the A14 | T8101 family on the iPhone 12), while the second 16-character group (ECID) is a serial component associated with the device. Moreover, UDID is used by tools on the Xcode side and the `devicectl` command to identify the device, learn on provisioning profiles that are bound to, and what Apple's developer portal lists when the researcher registers a device under their team. It is recommended to save the UDIDs of every paired iPhone, and I will be using them in several steps in this series of articles.

No doubt, physical iPhone provides researchers the definitive proof for reachability on the latest shipping iOS, a panic log located at `/private/var/mobile/Library/Logs/CrashReporter/Panics/panic-full-<timestamp>.ips`, and which can be pulled via `idevicecrashreport` command (presented later) because, at

the end, it is the real and complete environment where iOS applications run. Therefore, once your PoC worked on a physical iPhone, it is ready to be submitted to Apple or any broker.

Physical iPhone does not provide researchers with kernel read-write access for post panic inspection (basically, the device reboots on panic, and the panic log is the only output), and there is not access to kernel surfaces from the sandbox without additional entitlements, even because the standard developer provisioning flow does not grant `com.apple.private.*` entitlements, which makes impossible to examine in details a few vulnerabilities on iPhone. Additionally, we do not have the possibility to use a kernel debugger and attach on stock iOS.

There are two critical procedures that should be checked while doing iOS security research. The first one is to turn off automatic iOS updates, which obviously cause impact in vulnerability findings, but can be easily controlled by the research. The path is [Settings | General | Software Update | Automatic Updates](#) and keep both "Download iOS Updates" and "Install iOS Updates" options toggled off. An iOS auto-update event during a test may invalidate the kernelcache baseline, force it to build a new IDA Pro database, apply respective symbolication and still remove the kext or syscall handler that could be associated with the target. Thus, the recommendation is to check the auto-update setting on every new test iPhone immediately after setup, before pairing it to Xcode. Of course, you should keep the iOS updated as recommended, but it shouldn't occur in the middle of an investigation.

The second procedure is to turn off automatic submissions of panic and crash reports to Apple. Go to [Settings | Privacy & Security | Analytics & Improvements | Share iPhone Analytics](#) and toggle it to OFF. At the same screen, turn off all of available options such as Share iCloud Analytics, Improve Siri & Dictation, Improve AR Location Accuracy, and Improve Assistive Voice Features. In the same way, go to [Settings | Privacy & Security | Apple Advertising | Personalized Ads](#) and also toggle it off. Additionally, you should go to [Settings | \[your name\] | iCloud | iCloud Backup](#) and toggle it off. Finally, remember that after every major update, you should verify whether the settings are still in place. Of course, readers should remember that even if the device is not submitting anything to Apple, the device still keeps reports locally, and all of them are kept in `/private/var/mobile/Library/Logs/CrashReporter/` folder and can be read by going to [Settings | Privacy & Security | Analytics & Improvements | View Analytics Logs](#) and you can select the file.

At this point, once we have already presented all elements of a common and minimal laboratory, it is an appropriate moment to comment about how to configure the vPhone, whose project is hosted on <https://github.com/Lakr233/vphone-cli>.

Configuring vPhone is not a fixed and immutable procedure over time because iOS versions have changed month after month and also depends on the target hardware such as an Apple Silicon machine (M series) with a good amount of RAM. If you ask me whether it is possible to configure and bring up an iOS 27.x on an M2 with 8 GB, I can tell you that it is, but the system will be restricted while vPhone is online (even configuring it to use only 4 GB). Moreover, if you try any other intensive activity, probably swap file will increase quickly, memory will get fragmented and there will be a loss of performance. Thus, if you cannot dedicate the host to this purpose, I recommend you a recent system (M5) with 24 GB memory, at least. Regardless of the hardware, it must be running Sequoia or later and have 100 GB free SSD.

In terms of software, install the Xcode (`xcodebuild -version`) and check the SDK path (`xcrun -sdk iphoneos --show-sdk-path` command), install Python 3.8+ and, using Homebrew (`/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`), install ipsw (`brew install`

[blacktop/tap/ipsw](#)), keystone, sshpass, ldid-procursus, libimobiledevice, libimobiledevice-glue, libplist, and libtatsu.

Another note that must be done is that currently vPhone configures with iOS 27.x (iOS 27.2 beta 2 at the time I am writing this article), but this is not completely correct. As vPhone is composed of two components (IPSW image and CloudOS), the IPSW image provides the userland portion and CloudOS supplies the boot chain and kernel portion. If the reader checks the own vPhone project documentation (<https://github.com/Lakr233/vphone-cli/blob/main/Documents/Guides/compatibility.md>) you will notice that the latest support CloudOS is 26.4-23E5207q, which is not iOS 27.x, for sure.

The vPhone, which is a virtualized iPhone, will be running iOS 27.0 GA, jailbroken, on port 2222, with user "root" and password "alpine". Additionally, GDB stub will be running on port 4000. Once you got iOS running, relevant directories are:

- /var/jb: it holds Procursus rootless (jailbreak bootstrap).
- /var/jb/usr/bin: it holds 237 binaries (main one)
- /iosbinpack64: self-contained binary pack.
- /iosbinpack64/bin: it contains 25 important binaries (minimal pack)
- /var/root: it is the root's home directory, where we find .bash_history, .bash_profile, and other ones (there are not binaries)

If it is your first time configuring vPhone, all steps execute on macOS follow below:

Step 01: On macOS, as one-time step, boot it on recovery mode by holding the power button on startup until "Loading startup options" appears, then click **Options**, click on **Continue** and in the menu bar at the top of the screen, go to **Utilities | Terminal** and execute:

```
$ csrutil disable
$ csrutil allow-research-guests enable
$ reboot
```

Step 02: Relax AMFI by running the following:

```
$ sudo nvram boot-args="amfi_get_out_of_my_way=1 -v"
$ sudo reboot
```

Step 03: After each macOS update, it is recommended to verify whether setting has been kept:

```
$ csrutil status                # must say "disabled"
$ csrutil allow-research-guests status # must say "enabled"
$ nvram boot-args                # must include amfi_get_out_of_my_way=1
```

Step 04: Clone and install dependencies:

My perspective: I've stayed on d5283b4. Reasons: release 2.0.x changes the VM bundle format, requires every VM to be recreated, and moves to a CLI that breaks vphone.sh, the keeper, usbmux_reset.sh and every path used here. It is up to you.

```
$ mkdir -p ~/github && cd ~/github
$ git clone https://github.com/Lakr233/vphone-cli.git vphone-cli-26.5-attempt
$ cd vphone-cli-26.5-attempt
$ git checkout d5283b4
$ git lfs install
$ git submodule update --init --recursive
$ make setup_tools
```

Step 05: Apply different local patches, which have been necessary in my lab after running "git pull" command, as shown in the following tasks:

- **Task A:** Configure and allow GDB stub on port 4000:

```
$ sed -i '' 's/(6000 \.\.\. 65535)/(1024 ... 65535)/' sources/vphone-  
cli/VPhoneVirtualMachine.swift
```

- **Task B:** Configure the virtual machine memory to 4 GB (I used 8 GB, but it turned out not to be necessary):

```
$ sed -i '' 's/^MEMORY      ?= 8192/MEMORY      ?= 4096/' Makefile
```

- **Task C:** Adjust parallelism (Swift build -j 2):

```
$ sed -i '' 's/swift build 2>&1/swift build -j 2 2>\&1/' Makefile  
$ sed -i '' 's/swift build -c release 2>&1/swift build -c release -j 2 2>\&1/' Makefile  
$ sed -i '' 's/^swift build -c release$/swift build -c release -j 2/' scripts/build.sh
```

- **Task D:** Adjust battery configuration by editing sources/vphone-cli/VPhoneVirtualMachine.swift, and set setBattery(charge:connectivity:) function to ignore both arguments and enforce:

```
_ = charge; _ = connectivity  
Dynamic(source).setCharge(100.0)  
Dynamic(source).setConnectivity(1)
```

Note: do it if the host is a laptop (as in my case).

- **Task E:** Stop the self-invalidating build rule by going to Makefile and, in the \$(BINARY) recipe, replace the two unconditional BUILD_INFO writes with the following:

```
@echo '// Auto-generated - do not edit' > $(BUILD_INFO).tmp  
@echo 'enum VPhoneBuildInfo { static let commitHash = "$(GIT_HASH)" }' >> $(BUILD_INFO).tmp  
@cmp -s $(BUILD_INFO).tmp $(BUILD_INFO) 2>/dev/null || cp $(BUILD_INFO).tmp $(BUILD_INFO)  
@rm -f $(BUILD_INFO).tmp
```

- **Task F:** Create `vphone.sh`. Actually, this file does not exist upstream, and you must copy the whole script that will be presented after these steps into the repository root.
- **Task G:** Create `forwarder_keeper.sh` and `usbmux_reset.sh`, also absent upstream, and are presented in later part of this document. Additionally, execute:

```
$ chmod +x vphone.sh forwarder_keeper.sh usbmux_reset.sh
```

- **Task H:** Configure sudoers file are shown below:

```
$ echo 'researcher ALL=(root) NOPASSWD: /usr/bin/killall usbmuxd' | sudo tee  
/etc/sudoers.d/vphone-usbmuxd  
$ sudo chmod 440 /etc/sudoers.d/vphone-usbmuxd
```

- **Task I:** Add step 8b to the existing upstream file scripts/vphone_jb_setup.sh. This is an insertion into a file that already exists, and not a new file. Insert the block below immediately before the final "DONE" section, after step 8:

```
# ----- 8b/8 ROOTFS ENV FIXUP (each boot) -----  
# The sealed system snapshot ships no /usr/bin/env or /bin/bash, and  
# changes to the root filesystem revert on every cold boot. Proculus  
# tools with a #!/usr/bin/env shebang then fail to exec with a  
# misleading "<script>: No such file or directory", which refers to the
```

```
# missing interpreter and not to the script. This script runs from the
# snapshot on every boot, so recreate the symlinks here.
```

```
log "[8b/8] Recreating /usr/bin/env + /bin/bash (rootfs reverts each boot)..."
/sbin/mount -uw / 2>/dev/null || true
[ -e /var/jb/usr/bin/env ] && ln -sf /var/jb/usr/bin/env /usr/bin/env 2>/dev/null && log
" /usr/bin/env -> /var/jb/usr/bin/env"
[ -e /var/jb/usr/bin/bash ] && ln -sf /var/jb/usr/bin/bash /bin/bash 2>/dev/null && log
" /bin/bash -> /var/jb/usr/bin/bash"
```

- **Task J:** Allow the keeper to restart usbmuxd without requiring a password. Run the following commands once:

```
$ echo 'researcher ALL=(root) NOPASSWD: /usr/bin/killall usbmuxd' | sudo tee
/etc/sudoers.d/vphone-usbmuxd
$ sudo chmod 440 /etc/sudoers.d/vphone-usbmuxd
```

Step 06: Build necessary tooling by running the following commands:

```
$ make build
$ make bundle
$ make setup_venv
$ PATH="/opt/homebrew/bin:$PATH" make boot_host_preflight
```

Step 07: Get firmware images:

Download CloudOS blob 26.4 (23E5207q) from the following website:

<https://appledb.dev/device/identifier/ComputeModule14,2.html?filter=release>

Additionally, download iPhone IPSW image following steps below. Remember, my iPhone is 16, so I am downloading an image to this model, but you can choose any image:

```
$ git fetch origin
$ git show origin/main:Documents/Guides/compatibility.md
$ mkdir -p ~/research/ipsws && cd ~/research/ipsws
$ ipsw download ipsw --device iPhone17,3 --version 27.0 --confirm
$ shasum -a 1 iPhone17,3_*_Restore.ipsw | tee -a checksums.txt.shal
$ cd ~/github/vphone-cli-26.5-attempt
```

Step 08: Build the VM image:

```
$ export IPHONE_SOURCE="$HOME/research/ipsws/iPhone17,3_27.0_24A5390f_Restore.ipsw"
$ export
CLOUDOS_SOURCE="$HOME/Downloads/c0ecdb4b310cf5239ab2b248dd3098eec297dc5aa3bbe6ada27273262b0b8b64"
$ SUDO_PASSWORD='your-mac-login-password' make -s setup_machine JB=1 MEMORY=4096
```

Step 09: Install the keeper LaunchAgent, as shown below:

```
$ cp com.researcher.vphone-forwarder-keeper.plist ~/Library/LaunchAgents/
$ launchctl load ~/Library/LaunchAgents/com.researcher.vphone-forwarder-keeper.plist
```

Step 10: Start the vPhone:

```
$ cd ~/github/vphone-cli-26.5-attempt
$ rm -f .vphone_pid .expose_pids .forward_pids
$ VPHONE_KDEBUG_PORT=4000 ./vphone.sh start > /tmp/vphone_start.log 2>&1 &
$ while ! .venv/bin/python3 -m pymobiledevice3 usbmux list 2>/dev/null | grep -q 'iPhone99,11'; do
sleep 5; done
$ nohup ./vphone.sh forward > /tmp/vphone_forward.log 2>&1 & disown
$ export PATH="/opt/homebrew/bin:$PATH"
$ until sshpass -p alpine ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -o
ConnectTimeout=10 -p 2222 root@127.0.0.1 'echo SSH_OK' 2>/dev/null; do sleep 10; done
```

Step 11: Check the vPhone status:

```
$ cd ~/github/vphone-cli-26.5-attempt
$ ps aux | grep 'vphone-cli --config' | grep -v grep

# practical value: idle is from 2 to 8%, but in panic status we will see something around 700%
$ ps aux | grep 'Virtualization.VirtualMachine' | grep -v grep

$ lsof -nP -iTCP -sTCP:LISTEN | grep -E ":(2222|5901|5910|4000)"
$ .venv/bin/python3 -m pymobiledevice3 usbmux list

# note about -a: the log is a binary file
$ grep -a -c 'panic(cpu' setup_logs/boot_live.log

# my experience: never judge on a single check
$ tail -5 /tmp/vphone_forwarder_keeper.log
$ export PATH="/opt/homebrew/bin:$PATH"
$ for i in 1 2 3 4 5; do
    sshpass -p alpine ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -o
ConnectTimeout=10 -p 2222 root@127.0.0.1 'echo OK'; sleep 3
done

$ sshpass -p alpine ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -p 2222
root@127.0.0.1 "export
PATH=/var/jb/usr/bin:/var/jb/bin:/var/jb/usr/sbin:/var/jb/sbin:/iosbinpack64/bin:/iosbinpack64/usr/
bin:/usr/bin:/bin:/usr/sbin:/sbin; uname -v; uptime; dpkg -l | grep -c "^ii"; ls -d /var/jb
/iosbinpack64"
```

Step 12: Stop vPhone when it is necessary:

```
$ cd ~/github/vphone-cli-26.5-attempt
$ ./vphone.sh stop; sleep 3
$ pkill -f 'vphone-cli --config'; sleep 4
$ pkill -f 'pymobiledevice3.*forward'
$ rm -f .vphone_pid .expose_pids .forward_pids
```

Step 13 (recovery): If vPhone has any problem of communication like, for example, the workstation machine cannot connect to vPhone, try the following:

```
$ cd ~/github/vphone-cli-26.5-attempt
$ ./usbmux_reset.sh
```

If it still does not work, try:

```
$ sudo killall usbmuxd
$ nohup ./vphone.sh forward > /tmp/vphone_forward.log 2>&1 & disown
```

The **vphone.sh** follows below:

```
#!/usr/bin/env bash
# vphone.sh - manage the vphone-cli virtual iPhone
# Commands: start | stop | setup | forward | status | logs | ssh

set -eou pipefail

VPHONE_DIR="$(cd "$(dirname "$0")" && pwd)"
PYTHON="$VPHONE_DIR/.venv/bin/python3"
CONFIG="$VPHONE_DIR/vm/config.plist"
BINARY="$VPHONE_DIR/.build/vphone-cli.app/Contents/MacOS/vphone-cli" # normal boot (app bundle)
BINARY_DFU="$VPHONE_DIR/.build/release/vphone-cli" # DFU boot (release binary)
LOG="$VPHONE_DIR/setup_logs/boot_live.log"
LOG_DFU="$VPHONE_DIR/setup_logs/dfu.log"
export PATH="/opt/homebrew/bin:/opt/homebrew/sbin:$PATH"

FORWARD_PIDS_FILE="$VPHONE_DIR/.forward_pids"
```

```
SSH_PASS="alpine"

# -- helpers -----

die() { echo "ERROR: $" >&2; exit 1; }
log() { echo "[vphone] $*"; }

check_roms() {
  [[ -f "$VPHONE_DIR/vm/AVPBooter.vresearch1.bin" && \
    -f "$VPHONE_DIR/vm/AVPSEPBooter.vresearch1.bin" ]] \
    || die "ROM files missing in vm/. Run 'make vm_new' then restore from vm_backup/."

  for f in AVPBooter AVPSEPBooter; do
    local vm="$VPHONE_DIR/vm/${f}.vresearch1.bin"
    local bak="$VPHONE_DIR/vm_backup/${f}.vresearch1.bin"
    if [[ -f "$bak" ]] && ! cmp -s "$vm" "$bak"; then
      echo "WARNING: vm/${f}.vresearch1.bin differs from vm_backup/ - may be wrong ROM version."
      echo "  Fix: cp vm_backup/${f}.vresearch1.bin vm/${f}.vresearch1.bin"
    fi
  done
}

kill_pids_file() {
  local file="$1"
  [[ -f "$file" ]] || return 0
  while IFS= read -r pid; do
    [[ -n "$pid" ]] && kill "$pid" 2>/dev/null && echo "  killed pid $pid"
  done < "$file"
  rm -f "$file"
}

vphone_running() { pgrep -x vphone-cli >/dev/null 2>&1; }

release_storage_locks() {
  # Kill any Virtualization.framework helper processes still holding VM storage files.
  # These persist after vphone-cli exits and block the next launch.
  local pids
  pids=$(ls -l "$VPHONE_DIR/vm/nvram.bin" "$VPHONE_DIR/vm/SEPStorage" 2>/dev/null \
    | awk 'NR>1 {print $2}' | sort -u)
  if [[ -n "$pids" ]]; then
    log "Releasing storage locks (pids: $pids)..."
    echo "$pids" | xargs kill 2>/dev/null || true
    sleep 3
  fi
}

check_ssh() {
  sshpass -p "$SSH_PASS" ssh \
    -o ConnectTimeout=5 \
    -o StrictHostKeyChecking=no \
    -o BatchMode=no \
    -p 2222 root@127.0.0.1 true 2>/dev/null
}

wait_for_ssh() {
  local timeout="${1:-90}" label="${2:-SSH}"
  local deadline=$((SECONDS + timeout))
  while [[ $SECONDS -lt $deadline ]]; do
    check_ssh && return 0
    sleep 3
  done
  die "$label did not become available within ${timeout}s."
}

wait_for_usbmux() {
  local timeout="${1:-90}"
  local deadline=$((SECONDS + timeout))
}
```



```
while [[ $SECONDS -lt $deadline ]]; do
    "$PYTHON" -m pymobiledevice3 usbmux list 2>/dev/null | grep -q "ConnectionType" && return 0
    sleep 2
done
die "Device did not appear in usbmux within ${timeout}s."
}

start_forward_local() {
    kill_pids_file "$FORWARD_PIDS_FILE"
    rm -f "$FORWARD_PIDS_FILE"
    "$PYTHON" -m pymobiledevice3 usbmux forward --host 127.0.0.1 2222 22222 &
    echo $! >>"$FORWARD_PIDS_FILE"
}

# -- commands -----

cmd_forward() {
    local host="${1:-0.0.0.0}"
    kill_pids_file "$FORWARD_PIDS_FILE"
    [[ -f "$PYTHON" ]] || die ".venv not found. Run 'make setup_venv'."

    log "Setting up port forwards (host=$host)..."
    rm -f "$FORWARD_PIDS_FILE"
    for mapping in "2222:22222" "5901:5901" "5910:5910"; do
        local ext="${mapping%:*}" int="${mapping##*:}"
        "$PYTHON" -m pymobiledevice3 usbmux forward --host "$host" "$ext" "$int" &
        echo $! >>"$FORWARD_PIDS_FILE"
        echo " $host:$ext -> vphone:$int"
    done
    echo ""
    echo "Remote SSH: ssh -p 2222 root@$(ipconfig getifaddr en0 2>/dev/null || echo '<this-mac-ip>')"
}

_do_boot() {
    # Boot vphone normally and wait for vphoned to connect. No SSH check.
    vphone_running && { log "vphone-cli already running."; return 0; }
    [[ -f "$CONFIG" ]] || die "vm/config.plist missing. Run 'make vm_new' then restore ROMs."
    [[ -f "$BINARY" ]] || die "Binary not found. Run 'make build'."
    check_roms

    mkdir -p "$VPHONE_DIR/setup_logs"
    local kdebug_args=()
    if [[ -n "$VPHONE_KDEBUG_PORT:-" ]]; then
        kdebug_args=(--kernel-debug-port "$VPHONE_KDEBUG_PORT")
        log "Kernel GDB debug stub will bind 127.0.0.1:$VPHONE_KDEBUG_PORT"
    fi
    log "Starting vphone-cli..."
    (export PATH; nohup "$BINARY" --config "$CONFIG" ${kdebug_args[@]}+"${kdebug_args[@]}" >"$LOG"
    2>&1 & echo $! >"$VPHONE_DIR/.vphone_pid"; disown)

    log "Waiting for boot (up to 300s)..."
    local deadline=$((SECONDS + 300))
    while [[ $SECONDS -lt $deadline ]]; do
        grep -q '\[control\] connected to vphoned' "$LOG" 2>/dev/null && { log "Boot complete."; return 0; }
        sleep 2
    done
    die "Boot timed out after 300s - check $LOG."
}

cmd_start() {
    # Smart start: normal boot -> wait for SSH (up to 60s) -> prompt for setup if needed.
    [[ -f "$PYTHON" ]] || die ".venv not found. Run 'make setup_venv'."

    if vphone_running; then
        log "vphone-cli already running."
    else
```

```
_do_boot
fi

# Local-only forward first, then wait for SSH
start_forward_local
log "Waiting for SSH (up to 60s - JB first-boot setup may still be running)..."
local deadline=$((SECONDS + 60))
while [[ $SECONDS -lt $deadline ]]; do
    if check_ssh; then
        log "SSH OK - starting full forwards."
        cmd_forward
        return 0
    fi
    sleep 5
done

kill_pids_file "$FORWARD_PIDS_FILE"
echo ""
echo "WARNING: SSH not reachable after 60s."
echo " If this is a fresh/restored VM, run: ./vphone.sh setup"
echo " (DFU -> ramdisk -> cfw_install_jb -> normal boot, ~15 min)"
return 1
}

cmd_setup() {
# Full one-shot setup: DFU boot -> ramdisk -> CFW install (jailbreak) -> normal boot + forwards.
[[ -f "$PYTHON" ]] || die ".venv not found. Run 'make setup_venv'."
[[ -f "$CONFIG" ]] || die "vm/config.plist missing."
[[ -f "$BINARY" ]] || die "Binary not found."
check_roms
mkdir -p "$VPHONE_DIR/setup_logs"

# -- Step 1: DFU boot -----
log "[1/5] Booting into DFU mode..."
[[ -f "$BINARY_DFU" ]] || die "DFU binary not found at $BINARY_DFU. Run 'make build'."
vphone_running && cmd_stop
release_storage_locks
(export PATH; cd "$VPHONE_DIR/vm" && nohup "$BINARY_DFU" --config ./config.plist --dfu
>"$LOG_DFU" 2>&1 & disown)

# -- Step 2: wait for DFU / irecovery device -----
log "[2/5] Waiting for DFU device..."
local deadline=$((SECONDS + 90))
local found=0
while [[ $SECONDS -lt $deadline ]]; do
    "$PYTHON" "$VPHONE_DIR/scripts/pymobiledevice3_bridge.py" \
        recovery-probe --timeout 3 >/dev/null 2>&1 && found=1 && break
    sleep 3
done
[[ $found -eq 1 ]] || die "Device did not enter DFU mode within 90s - check $LOG_DFU."

# -- Step 3: send ramdisk -----
log "[3/5] Sending ramdisk..."
(
    cd "$VPHONE_DIR/vm"
    PMD3_BRIDGE="$VPHONE_DIR/scripts/pymobiledevice3_bridge.py" \
    PYTHON="$PYTHON" \
    zsh "$VPHONE_DIR/scripts/ramdisk_send.sh"
)

# -- Step 4: wait for ramdisk SSH -----
log "[4/5] Waiting for ramdisk SSH..."
wait_for_usbmux 120
start_forward_local
wait_for_ssh 90 "ramdisk SSH"
log "Ramdisk SSH ready."
```

```
# -- Step 5: install CFW (jailbreak) -----
log "[5/5] Installing CFW - this will take several minutes..."
(
  cd "$VPHONE_DIR/vm"
  SSH_PORT=2222 \
  _VPHONE_PATH="$PATH" \
  zsh "$VPHONE_DIR/scripts/cfw_install_jb.sh" .
)

# CFW install halts the device; wait for vphone-cli to exit
kill_pids_file "$FORWARD_PIDS_FILE"
log "CFW installed. Waiting for device to halt..."
local t=0
while vphone_running && [[ $t -lt 30 ]]; do sleep 2; ((t+=2)); done

# -- Normal boot + forwards -----
log "Starting normal boot..."
_do_boot
cmd_forward
log "Setup complete."
echo ""
echo "Remote SSH: ssh -p 2222 root@$(ipconfig getifaddr en0 2>/dev/null || echo '<this-mac-ip>')"
}

cmd_stop() {
  kill_pids_file "$FORWARD_PIDS_FILE"
  if pkill -x vphone-cli 2>/dev/null; then
    log "vphone-cli stopped."
    sleep 2
  else
    log "vphone-cli was not running."
  fi
  rm -f "$VPHONE_DIR/.vphone_pid"
  release_storage_locks
}

cmd_status() {
  echo "=== vphone-cli ==="
  if vphone_running; then
    echo "  running (pid $(pgrep -x vphone-cli))"
  else
    echo "  not running"
  fi

  echo "=== port forwards ==="
  if [[ -f "$FORWARD_PIDS_FILE" ]]; then
    while IFS= read -r pid; do
      kill -0 "$pid" 2>/dev/null && echo "  pid $pid  running" || echo "  pid $pid  dead"
    done < "$FORWARD_PIDS_FILE"
  else
    echo "  none"
  fi

  echo "=== SSH ==="
  if check_ssh 2>/dev/null; then
    echo "  reachable (port 2222)"
  else
    echo "  not reachable"
  fi
}

cmd_logs() { tail -f "$LOG"; }
cmd_ssh() { ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -p 2222
root@127.0.0.1 /bin/bash -l; }

# -- dispatch -----
```

```
usage() {
    cat <<EOF
Usage: ${basename "$0"} <command> [args]

Commands:
    start          Boot vphone and set up forwards; auto-installs CFW if needed
    stop           Stop vphone and all port forwards
    setup          Full CFW install from scratch: DFU -> ramdisk -> cfw_install_jb -> normal boot
    forward [ip]   (Re)start port forwards - default 0.0.0.0 (all interfaces)
    status         Show vphone, forward, and SSH state
    logs          tail -f the boot log
    ssh           SSH into vphone as root (local)
EOF
}

cd "$VPHONE_DIR"

case "${1:-}" in
    start)    cmd_start ;;
    stop)     cmd_stop ;;
    setup)    cmd_setup ;;
    forward)  cmd_forward "${2:-}" ;;
    status)   cmd_status ;;
    logs)     cmd_logs ;;
    ssh)      cmd_ssh ;;
    *)        usage; exit 1 ;;
esac
```

The [forwarder_keeper.sh](#) script follows below:

```
#!/bin/bash
# =====
# vphone forwarder-keeper
# =====

set -u

export PATH="/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:${PATH:-}"

VM_DIR="/Users/researcher/github/vphone-cli-26.5-attempt"
PYTHON="$VM_DIR/.venv/bin/python3"
LOG="/tmp/vphone_forwarder_keeper.log"
FWD_LOG="/tmp/vphone_forward.log"
PROBE_PORT=2222
DEVICE_TAG="iPhone99,11"
INTERVAL=60
SSH_PROBE_EVERY=2
NOTICE_COOLDOWN=1800
MAX_BOUNCES=2
MIN_BOUNCE_GAP=180

last_down_notice=0
last_persist_notice=0
last_bounce=0
bounce_count=0
episode_active=0
iter=0

log() { echo "$(date '+%Y-%m-%d %H:%M:%S') $*' >> "$LOG"; }

listening() { lsof -nP -iTCP:"$PROBE_PORT" -sTCP:LISTEN 2>/dev/null | grep -q LISTEN; }
device_present() { "$PYTHON" -m pymobiledevice3 usbmux list 2>/dev/null | grep -q "$DEVICE_TAG"; }
vm_alive() { pgrep -f 'vphone-cli --config' >/dev/null 2>&1; } # host VM process up?

ssh_ok() {
    # No -o BatchMode=yes - it disables password auth and sshpass can't feed it.
    sshpass -p alpine ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null \
```

```
-o ConnectTimeout=6 -p "$PROBE_PORT" root@127.0.0.1 'echo ok' 2>/dev/null | grep -q ok
}

# 0 = frozen (3 consecutive fails); 1 = healthy (returns on first success).
confirm_freeze() {
    local tries=3 gap=5 i
    for i in $(seq 1 "$tries"); do
        ssh_ok && return 1
        [ "$i" -lt "$tries" ] && sleep "$gap"
    done
    return 0
}

relaunch_forward() {
    cd "$VM_DIR" || { log "ERROR cannot cd $VM_DIR - skipping relaunch"; return 1; }
    pkill -f 'pymobiledevice3 usbmux forward' 2>/dev/null
    sleep 1
    log " relaunching forwarders (./vphone.sh forward)"
    nohup ./vphone.sh forward > "$FWD_LOG" 2>&1 &
    disown
}

# STAGE-1-only usbmuxd bounce (NEVER a VM restart). 0 = freeze cleared, 1 = not.
bounce_usbmuxd() {
    # Passwordless via scoped sudoers rule: researcher NOPASSWD: /usr/bin/killall usbmuxd
    if ! sudo -n killall usbmuxd 2>/dev/null; then
        log " bounce FAILED - could not sudo killall usbmuxd (rule missing?)"
        return 1
    fi
    local i
    for i in $(seq 1 12); do pgrep -x usbmuxd >/dev/null 2>&1 && break; sleep 1; done
    for i in $(seq 1 15); do device_present && break; sleep 2; done
    relaunch_forward
    for i in $(seq 1 12); do ssh_ok && return 0; sleep 3; done
    return 1
}

log "=== forwarder-keeper started (pid $$) - SAFE; auto-bounces usbmuxd on a confirmed freeze,
NEVER restarts the VM ==="

while true; do
    if ! listening; then
        if device_present; then
            relaunch_forward
            bounce_count=0; episode_active=0
        else
            now=$(date +%s)
            if [ $(( now - last_down_notice )) -ge $NOTICE_COOLDOWN ]; then
                log "IDLE :$PROBE_PORT down AND device not in usbmux -> VM is down (manual-start policy) ->
not acting."
                last_down_notice=$now
            fi
        fi
    elif [ $(( iter % SSH_PROBE_EVERY )) -eq 0 ]; then
        if confirm_freeze; then
            # -- confirmed SSH-down over 3 probes --
            episode_active=1
            now=$(date +%s)
            if ! vm_alive; then
                # Variant C - the VM process itself crashed (VZErrorDomain). A usbmuxd
                # bounce is useless (nothing behind it); only a COLD-START helps, which the
                # keeper must NOT do. Log distinctly and wait for the user/agent.
                if [ $(( now - last_persist_notice )) -ge $NOTICE_COOLDOWN ]; then
                    log "VM CRASHED (vphone-cli process gone, Variant C) - NOT a tunnel freeze; usbmuxd
bounce useless -> COLD-START needed. Keeper does NOT auto-restart the VM."
                    last_persist_notice=$now
                fi
            fi
        fi
    fi
done
```

```
elif [ "$bounce_count" -lt "$MAX_BOUNCES" ] && [ $(( now - last_bounce )) -ge $MIN_BOUNCE_GAP ]; then
    log "FREEZE confirmed (3 probes) -> auto-bouncing usbmuxd (attempt $(( bounce_count + 1 ))/$MAX_BOUNCES)..."
    last_bounce=$now
    if bounce_usbmuxd; then
        log "AUTO-RECOVER OK - usbmuxd bounce cleared the freeze (host-side); VM untouched."
        bounce_count=0; episode_active=0
    else
        bounce_count=$(( bounce_count + 1 ))
        log " bounce did not clear it yet (episode bounces=$bounce_count/$MAX_BOUNCES)."
    fi
else
    if [ $(( now - last_persist_notice )) -ge $NOTICE_COOLDOWN ]; then
        log "FREEZE persists after $MAX_BOUNCES usbmuxd bounce(s) (or within gap) -> likely NOT host-side; MANUAL VM restart needed (run ./usbmux_reset.sh)."
        last_persist_notice=$now
    fi
fi
else
    # healthy
    if [ "$episode_active" = "1" ]; then
        log "recovered - SSH healthy again; resetting episode state."
    fi
    bounce_count=0; episode_active=0
fi
fi
iter=$(( iter + 1 ))
sleep "$INTERVAL"
done
```

The [usbmux_reset.sh](#) script follows below:

```
#!/bin/bash
# =====
# usbmux\_reset.sh - lightweight vphone freeze recovery.
# =====
set -u
export PATH="/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:${PATH:-}"

VM_DIR="/Users/researcher/github/vphone-cli-26.5-attempt"
PYTHON="$VM_DIR/.venv/bin/python3"
PROBE_PORT=2222
DEVICE_TAG="iPhone99,11"
GDB_PORT="${VPHONE_KDEBBUG_PORT:-4000}"
FWD_LOG="/tmp/vphone_forward.log"

say() { echo "[usbmux_reset $(date '+%H:%M:%S')] $*"; }

as_root() {
    sudo -n "$@" 2>/dev/null && return 0
    if [ -n "${SUDO_ASKPASS:-}" ]; then sudo -A "$@" && return 0; fi
    sudo "$@"
}

ssh_ok() {
    sshpass -p alpine ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null \
        -o ConnectTimeout=6 -p "$PROBE_PORT" root@127.0.0.1 'echo ok' 2>/dev/null | grep -q ok
}

confirm_freeze() {
    local tries=3 gap=5 i
    for i in $(seq 1 "$tries"); do
        if ssh_ok; then return 1; fi
        [ "$i" -lt "$tries" ] && sleep "$gap"
    done
    return 0
}
```



```
}
device_present() { "$PYTHON" -m pymobiledevice3 usbmux list 2>/dev/null | grep -q "$DEVICE_TAG"; }
listening()      { lsof -nP -iTCP:"$PROBE_PORT" -sTCP:LISTEN 2>/dev/null | grep -q LISTEN; }

relaunch_forward() {
    cd "$VM_DIR" || return 1
    pkill -f 'pymobiledevice3 usbmux forward' 2>/dev/null
    sleep 1
    nohup ./vphone.sh forward > "$FWD_LOG" 2>&1 &
    disown
}

# -- 0. Sanity: are we actually frozen? (multi-probe - never act on a blip) --

if ! confirm_freeze; then say "SSH healthy (passed multi-probe) - nothing to do."; exit 0; fi
say "freeze CONFIRMED over 3 consecutive probes: SSH down; listening=$(listening && echo yes || echo no) device_present=$(device_present && echo yes || echo no)"

# -- 1. STAGE 1 - bounce the HOST usbmuxd daemon -----
say "STAGE 1 - bouncing host usbmuxd (needs root for: killall usbmuxd)..."
if ! as_root killall usbmuxd 2>/dev/null; then
    say " could NOT bounce usbmuxd (no root available) - skipping to STAGE 2."
else
    for i in $(seq 1 12); do pgrep -x usbmuxd >/dev/null 2>&1 && break; sleep 1; done
    say " usbmuxd respawned as pid: $(pgrep -x usbmuxd | tr '\n' ' ')"
    for i in $(seq 1 15); do device_present && break; sleep 2; done
    say " device re-enumerated: $(device_present && echo yes || echo no)"
    relaunch_forward
    for i in $(seq 1 12); do ssh_ok && break; sleep 3; done
    if ssh_ok; then
        say "STAGE 1 SUCCESS OK - usbmuxd bounce cleared the freeze; NO VM restart needed."
        say " DIAGNOSIS: this freeze was HOST-side (usbmuxd/USB-link state)."
        exit 0
    fi
    say "STAGE 1 did NOT clear it - DIAGNOSIS: freeze is GUEST-side. Falling back."
fi

# -- 2. STAGE 2 - full VM restart (known fix) -----
say "STAGE 2 - full VM restart (GDB stub :$GDB_PORT)..."
cd "$VM_DIR" || { say "cannot cd $VM_DIR"; exit 1; }
./vphone.sh stop
rm -f .vphone_pid .expose_pids .forward_pids
VPHONE_KDEBUG_PORT="$GDB_PORT" ./vphone.sh start &
for i in $(seq 1 40); do device_present && break; sleep 5; done
say " usbmux up: $(device_present && echo yes || echo no)"
relaunch_forward
for i in $(seq 1 45); do ssh_ok && break; sleep 10; done
if ssh_ok; then
    say "STAGE 2 SUCCESS OK - VM restart restored SSH."
    lsof -nP -iTCP -sTCP:LISTEN 2>/dev/null | grep -E ':(2222|5901|5910|4000)'
else
    say "STAGE 2 FAILED FAIL - SSH still down after restart; investigate manually."
    exit 1
fi
```

Finally, [com.researcher.vphone-forwarder-keeper.plist](#) file, which goes in `~/Library/LaunchAgents/` directory, follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.researcher.vphone-forwarder-keeper</string>
```

```
<key>ProgramArguments</key>
<array>
  <string>/bin/bash</string>
  <string>/Users/researcher/github/vphone-cli-26.5-attempt/forwarder_keeper.sh</string>
</array>

<key>RunAtLoad</key>
<true/>

<key>KeepAlive</key>
<true/>

<!-- Guard against a rapid keeper-respawn loop if the script exits immediately. -->
<key>ThrottleInterval</key>
<integer>30</integer>

<key>StandardOutPath</key>
<string>/tmp/vphone_forwarder_keeper.out</string>
<key>StandardErrorPath</key>
<string>/tmp/vphone_forwarder_keeper.err</string>
</dict>
</plist>
```

08. iOS exploitation path and the reachability problem

This section brings a superficial introduction to the exploitation path and reachability classes, which can help readers to have an initial understanding of the context of iOS security research. Needless to say, concepts and ideas presented here are incomplete because this is only an initial article, and readers should expect for a series of terms and concepts that will be explained in next articles.

One of most asked questions about iOS exploitation is exactly what paths should be considered, and mainly what challenges will need to be handled through the way. In general way, the macro stages to evaluate and procedure with exploitation are evaluating the reachability, entitlements, finding a bug, turning the vulnerability into one or more primitives, obtaining a kernel read and write primitive and, finally, bypassing monitors like PPL/SPTM and TXM.

On iOS, as the first stage, all applications are started from a sandbox, which means that allowed actions and access to resources (including calling system calls) are limited and filtered. Thus, trying to exploit anything from an application involves a sandbox escape as part of the exploitation procedure because application sandbox is the most restrictive profile on the system. As you quickly will realize, most of the kernel's attack surface is placed behind that filter and, in the recent iOS, where we find about 140 IOKit services, bit only eighteen of them can be opened from inside application sandbox, which means that there could be multiple vulnerabilities in one of these drivers, but we cannot reach them without an appropriate sandbox escape.

Based on presented challenges, a few alternatives would be either trying to compromise a less restricted target, which might be a daemon that already parses input we can control and, from there, attack the kernel, or finding a kernel bug that is reachable from inside the application sandbox, which are rare and valuable opportunities because it would not be necessary to think about a sandbox escape. If the attacker needs to pass through the sandbox, it will be necessary to discover vulnerabilities in the sandbox component to begin the exploitation process, and this is not restricted to only one bug.

The sandbox can be checked directly via the sandbox library because it exports a function named `sandbox_check`, which verifies whether a process is permitted to perform a chosen operation against a determined target (drivers and daemons, for example). Additionally, to check the allowed access, attacker needs to perform the access operation and read what comes back. Thus, it is not only a matter to reach or not a code (driver, service, or anything else), but also whether access is accepted or not. Actually, if you write a code to make such a test, you will quickly learn that doing it is not simple and there is not a clear and always working algorithm because it requires to handle potential scenarios. Anyway, I will show, in the next lines, a real and applied approach to illustrate the described scenario by making use of the mentioned `sandbox_check` function and, eventually, readers can get better comprehension about sandbox subject and its innumerable details.

The prototype of `sandbox_check` function is given by `int sandbox_check(pid_t pid, const char *operation, enum sandbox_filter_type type, ...)`, where the pid parameter is the target process ID that the application we want to verify, operation parameter indicates the operation to be performed (ex: mach-lookup, network-outbound, file-read-data and other ones), the filter parameter means the type of the object being requested (for example, path or service name), the type parameter is given elements from an enumeration (private, by the way) that provides us an evidence about how the target resource should be indicated, and where some possible values as `SANDBOX_FILTER_PATH`, `SANDBOX_FILTER_GLOBAL_NAME`, `SANDBOX_FILTER_XPC_SERVICE_NAME`, `SANDBOX_FILTER_IOKIT_CONNECTION` and other ones.

Additional parameters (it is a variadic function) depend on the type of the resource being evaluated, which is provided by the third parameter (type). In this context, "object" would be the fourth parameter, which provides the value of the object being consulted. The third parameter (type), which explains how the target should be interpreted, is undocumented, and it is usually a difficult task to be tackled because the sandbox library exports around thirty constants, which cover all neighboring families, but there is not a single filter type. Worse, such types differ according to the kind of object being named and respective values might change across iOS releases (I didn't test it), even though there are good articles around the Internet discussing this topic. The second caveat is that the operation strings lives in the library, but I couldn't find them in any available headers. As you can infer, if parameters of the functions are wrong and misunderstood, there is a good chance that the request to be refused.

Writing a code to do any test of reachability for targets involves understanding whether the target (subject) performs a given operation on a certain object and, as the `sandbox_check` function is usually used with three or four parameters, finding working combinations are not a direct task. Another aspect I would like to ask attention is that, if we analyze the `sandbox_check` function, we will need to supply values for pid (first parameter) and object (fourth parameter), but we will need to provide valid operations (second parameter) to each object. Moreover, it is required to discover the filter type values (third parameter) because they are undocumented.

Once we have a code for checking the sandbox accessibility, one of possible approaches is to pin three (pid, operation and type) of four parameters and reuse them for each request. Thus, we can do hundred of requests by only changing the object (four parameters), but the real problem is from where we get each parameter, and answer could not be direct or simple.

To retrieve operation names (second parameter), we can extract them from the sandbox kernel extension, as shown below:

```
$ ipsw kernel extract kernelcache com.apple.security.sandbox --output /tmp/sbk
```

```
strings -a /tmp/sbk/com.apple.security.sandbox \  
| grep -E "[a-z][a-z0-9]*(-[a-z0-9]+)*\*$" \  
| grep -E "(file|mach|iokit|ipc|process|sysctl|network|system|nvram|hid)" \  
| sort -u
```

```
file*  
file-chroot  
file-clone  
file-graft  
file-ioctl  
file-issue-extension  
... (163 lines on this current build)
```

Unfortunately, having the list does not tell us if a name is still current (or in use), and to circumvent this condition is easier to classify and return -1 for non-existing names.

Another task is how to find filters (third parameter from `sandbox_check` function) because we cannot use arbitrary names, and we need names whose right answer we already know about. As objects depend on the object type, there are options to get file paths, service names, and even invented ones. Obtaining "file paths" can be done by opening the files, and checking and verifying the constant (filter, third parameter), which the correct is the one whose verdict matches what the open action actually performed, and also confirming that misclassification occurs when a constant indicates a negative result for a file that is demonstrably opened. To get "service names" there is not the alternative of looking them up, where the best way is to build a list with real names and, occasionally, to include one service name that certainly does not exist, because we can observe and analyze the difference between the answer for true service names against that one we have made up, and confirm the existence of the constant (filter, third parameter).

I would like to say that obtaining the object list (fourth parameter) is easier, but it is not, and there are alternatives and restrictions to perform this task appropriately.

The `sandbox_check` function returns about a name you pass to it, but it does not tell us what names are correct or not. Therefore, it is necessary to enumerate such names, and for mach service names, we can read MachServices key from the daemon property lists by executing the following command:

```
$ grep -A4 -rh MachServices /System/Library/LaunchDaemons/*.plist | grep -oE "com[.]apple[.][a-zA-Z0-9._-]+" | sort -u
```

```
grep: /System/Library/LaunchDaemons/com.apple.ABDatabaseDoctor.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.AdminLite.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.Archetype.textcontextd.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.AssetCacheLocatorService.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.BTServer.avrcp.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.BTServer.map.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.BTServer.pbap.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.BlueTool.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.CarPlayApp.plist: binary file matches  
grep: /System/Library/LaunchDaemons/com.apple.ClarityBoard.plist: binary file matches  
.....  
com.apple.AccessorySensorManager  
com.apple.AccessoryServices  
com.apple.AppSSO.ddm-configuration-xpc  
com.apple.AppSSO.service-xpc  
com.apple.AuthenticationServices.AuthenticationServicesAgent.AutomaticSecurityUpgrade  
com.apple.AuthenticationServices.AuthenticationServicesAgent.CredentialUpdate  
com.apple.BTLEAudioController.xpc  
com.apple.BTLELoggingManager.xpc  
com.apple.BarcodeSupport.BarcodeNotificationService  
com.apple.BatteryDischargeService  
com.apple.BluetoothUIService  
com.apple.CallHistorySyncHelper
```

```
com.apple.CallHistorySyncHelper.aps
com.apple.CarAssetUtils.variants
....
....
```

Running the command on the vPhone, we obtain 276 unique names, but there are 677 plists (in my environment) and over 468 of them are binary property lists. Therefore, out of these 199 text plists, we got the mentioned 276 names, which roughly represent one third of the existing daemon on the system. Ok, it is a good number, and for basic and non-rigorous tests, it is enough. However, to a complete approach, we should collect all other names, but first we have to convert the binary plists. As nothing is simple, there is no [plutil](#) command on iOS from vPhone, but we can copy the binary plist to macOS and do it there, as shown below:

```
$ ssh -p 2222 root@127.0.0.1 "export PATH=/var/jb/usr/bin:/var/jb/bin:/usr/bin:/bin; tar -cf -
/System/Library/LaunchDaemons/*.plist" > ld.tar
$ mkdir -p ld && tar -xf ld.tar -C ld

$ mkdir -p ldx
for f in ld/System/Library/LaunchDaemons/*.plist; do
    plutil -convert xml1 -o "ldx/${basename "$f"}" "$f"
done

$ python3 machsvc.py ldx > services.txt
```

Initially I have used [grep](#) command, but as MachService is a dictionary whose keys are service names, and some daemons can wrap their service named into a kind of feature-flag conditionals, so a key's value may be another dictionary holding more names, and grep execution for the four lines after MachServices truncates every daemon that declares more than a couple. Unfortunately, it is not the only issue that I stumbled, and a better approach was writing a python script to filter the services, as the [machsvc.py](#) shows below:

```
#!/usr/bin/env python3

# machsvc.py

import glob, os, plistlib, sys

def collect(node, out):
    if not isinstance(node, dict):
        return
    for key, value in node.items():
        if key.startswith("#"):
            collect(value, out)
        elif "." in key:
            out.add(key)
        if isinstance(value, dict) and not key.startswith("#"):
            collect(value, out)

def main():
    d = sys.argv[1] if len(sys.argv) > 1 else "."
    names, read, bad = set(), 0, 0
    for path in sorted(glob.glob(os.path.join(d, "*.plist"))):
        try:
            with open(path, "rb") as fh:
                plist = plistlib.load(fh)
        except Exception:
            bad += 1
            continue
        read += 1
        collect(plist.get("MachServices"), names)
    for n in sorted(names):
```

```
print(n)
print("# %d plists read, %d unreadable, %d service names"
      % (read, bad, len(names)), file=sys.stderr)

if __name__ == "__main__":
    main()
```

Once you have followed the procedure and also the script above, you get a report with 2330 service names, which some ones are shown below:

```
$ head -12 services.txt
```

```
com.apple.ABDatabaseDoctor
com.apple.AccessorySensorManager
com.apple.AccessoryServices
com.apple.AdSheetPhone.push
com.apple.AdminLite
com.apple.AppSSO.ddm-configuration-xpc
com.apple.AppSSO.service-xpc
com.apple.ApplicationService.ServiceDiscovery
com.apple.Archetype.personalContext
com.apple.AssetCacheLocatorService
com.apple.AttentionAwareness
com.apple.AudioAccessoryCommunication.xpc
```

```
$ tail -6 services.txt
```

```
com.apple.xpc.amsengagementd
com.apple.xpc.amsmediacatalogsync
com.apple.xpc.amsserverdatacacheservice
com.apple.xpc.amsuniversallinks
com.apple.xpc.datastore
com.apple.xpc.roleaccountd
```

In the output shown above, services are ordered in sequence, but readers can check any of them, as shown below:

```
$ grep -E "^com\\.apple\\. (MobileFileIntegrity|mobile\\.installld|nsurlsessiond)$" services.txt
```

```
com.apple.MobileFileIntegrity
com.apple.mobile.installld
com.apple.nsurlsessiond
```

Obviously, the current result it is more valuable than the previous 276 service names due to the fact we have included the binary property lists, and 2249 of the 2330 services will resolve correctly, which is an outstanding outcome.

We have a list of services that can be used, so we can return to the main program, which was not shown previously, and whose goal is to check and report accessibility from the sandbox:

```
/ * sbmap.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <dlfcn.h>

typedef int (*sbcheck_t)(pid_t, const char *, int, ...);
static sbcheck_t sb;

#define REAL "com.apple.lsd.mapdb"
#define FAKE "com.apple.NOT.A.SERVICE.EVER"
```

```
#define TYPO "not-a-real-operation-name"

static int opens(const char *p)
{
    int fd = open(p, O_RDONLY);
    if (fd < 0) return 0;
    close(fd);
    return 1;
}

static int cal_path(void)
{
    static const char *P[] = {
        "/System/Library/CoreServices/SystemVersion.plist",
        "/usr/lib/dyld",
        "/private/var/root",
        "/private/var/mobile/Library/Preferences",
        "/dev/kmem",
    };
    for (int f = 0; f <= 15; f++) {
        if (f == 11) continue;
        int agree = 0, fatal = 0;
        for (int i = 0; i < 5; i++) {
            int said = sb(getpid(), "file-read-data", f, P[i]) == 0;
            int ok = opens(P[i]);
            if (said == ok) agree++;
            else if (!said && ok) fatal++;
        }
        if (!fatal && agree == 5) return f;
    }
    return -1;
}

static int cal_name(pid_t t)
{
    for (int f = 0; f <= 15; f++) {
        if (f == 11) continue;
        if (sb(t, "mach-lookup", f, REAL) != sb(t, "mach-lookup", f, FAKE))
            return f;
    }
    return -1;
}

int main(int argc, char **argv)
{
    if (argc < 2) { fprintf(stderr, "usage: sbmap <pid> <names\n"); return 2; }
    pid_t t = (pid_t)atoi(argv[1]);

    sb = (sbcheck_t)dlsym(RTLD_DEFAULT, "sandbox_check");
    if (!sb) { fprintf(stderr, "sbmap: sandbox_check unavailable\n"); return 1; }

    int fname = cal_name(t);
    if (fname < 0) {
        fprintf(stderr, "sbmap: pid %d -- no filter separates a real name from an\n",
            "impossible one. The target is unconfined, or the\n",
            "interface has moved. Nothing to measure.\n", (int)t);
        return 1;
    }
    int fpath = cal_path();
    int optest = sb(t, TYPO, 0); /* a typo must be distinguishable */
    int control = sb(t, "mach-lookup", fname, FAKE);

    fprintf(stderr, "sbmap: pid %d path=%d name=%d typo=%d control=%s\n",
        (int)t, fpath, fname, optest, control == 0 ? "ALLOW (BROKEN)" : "deny");

    char line[512];
    long asked = 0, allow = 0;
```



```
while (fgets(line, sizeof line, stdin)) {
    size_t n = strlen(line);
    while (n && (line[n - 1] == '\n' || line[n - 1] == '\r')) line[--n] = 0;
    if (!n || *line == '#') continue;
    asked++;
    if (sb(t, "mach-lookup", fname, line) == 0) { allow++; puts(line); }
}
fprintf(stderr, "sbmap: %ld asked    %ld allow    %ld deny    %.1f%% reachable\n",
    asked, allow, asked - allow, asked ? 100.0 * (double)allow / (double)asked : 0.0);
return 0;
}
```

Build and deploy the program by running the following steps:

- **Step 01:** build and sign, on the Mac.

```
$ xcrun -sdk iphoneos clang -arch arm64e -O2 -Wall -Wextra -o sbmap sbmap.c
$ ldid -S sbmap
```

- **Step 02:** put the tool and the name list on the device.

```
$ P="export PATH=/var/jb/usr/bin:/var/jb/bin:/usr/bin:/bin"
$ cat sbmap | ssh -p 2222 root@127.0.0.1 "$P; cat > /var/jb/sbmap"
$ cat services.txt | ssh -p 2222 root@127.0.0.1 "$P; cat > /var/jb/services.txt"
$ ssh -p 2222 root@127.0.0.1 "$P; chmod +x /var/jb/sbmap"
```

- **Step 03:** pick a confined target.

```
$ ssh -p 2222 root@127.0.0.1 "$P; ps -eo pid,uid,comm | grep containers/Bundle | head -1"
275    501 /private/var/containers/Bundle/Application/.../FindMyWidgetItems
```

- **Step 04:** run it.

```
$ ssh -p 2222 root@127.0.0.1 "$P; /var/jb/sbmap 275 < /var/jb/services.txt > /var/jb/allow.txt"
```

- **Step 05:** read the list from different forms.

```
$ ssh -p 2222 root@127.0.0.1 "$P; head -15 /var/jb/allow.txt"
$ ssh -p 2222 root@127.0.0.1 "$P; tail -3 /var/jb/allow.txt"
$ ssh -p 2222 root@127.0.0.1 "$P; cat /var/jb/allow.txt | wc -l"
$ ssh -p 2222 root@127.0.0.1 "$P; cat /var/jb/allow.txt" # all 375 of them
```

The output is of the program for **FindMyWidgetItems** process follows below:

```
sbmap: pid 275      path=1 name=2  typ0=-1  control=deny
sbmap: 2330 asked   375 allow    1955 deny    16.1% reachable

$ ssh -p 2222 root@127.0.0.1 "$P; head -15 /var/jb/allow.txt"
com.apple.ABDatabaseDoctor
com.apple.AppSSO.service-xpc
com.apple.AuthenticationServices.AuthenticationServicesAgent.CredentialUpdate
com.apple.AuthenticationServicesCore.AuthenticationServicesAgent
com.apple.CARenderServer
com.apple.ClipServices.clipserviced
com.apple.CoreAuthentication.daemon
com.apple.DeviceAccess.xpc
com.apple.DragUI.druid.destination
com.apple.DragUI.druid.source
com.apple.ExternalAccessory.distributednotification.server
com.apple.FSEvents
com.apple.FileCoordination
com.apple.FileProvider
com.apple.GSSCred
```

```
$ ssh -p 2222 root@127.0.0.1 "$P; tail -3 /var/jb/allow.txt"
com.apple.webprivacyd
com.apple.wifip2pd
com.apple.wirelessinsightsd.xpc

$ ssh -p 2222 root@127.0.0.1 "$P; cat /var/jb/allow.txt | wc -l"
375
```

The same **sbmap** program applied for **SpringBoard** process, which is far more privileged, shows the following:

```
$ ssh -p 2222 root@127.0.0.1 "$P; /var/jb/sbmap 42 < /var/jb/services.txt > /var/jb/allow_sb.txt"
sbmap: pid 42   path=1 name=2   typ=--1   control=deny
sbmap: 2330 asked   589 allow   1741 deny   25.3% reachable

$ ssh -p 2222 root@127.0.0.1 "$P; head -15 /var/jb/allow_sb.txt"
com.apple.ABDatabaseDoctor
com.apple.AppSSO.service-xpc
com.apple.AssetCacheLocatorService
com.apple.AttentionAwareness
com.apple.AudioAccessoryServices
com.apple.BTAudioHALPlugin.xpc
com.apple.BTServer.le
com.apple.BarcodeSupport.BarcodeNotificationService
com.apple.CARenderServer
com.apple.CarAssetUtils.variants
com.apple.CarPlayApp.non-launching-service
com.apple.CarPlayApp.service
com.apple.CarPlayApp.user-alerts-service
com.apple.CellularPlanDaemon.xpc
com.apple.ClipServices.clipserviced

$ ssh -p 2222 root@127.0.0.1 "$P; tail -3 /var/jb/allow_sb.txt"
com.apple.xpc.activity.unmanaged
com.apple.xpc.amsaccountsd
com.apple.xpc.amsengagementd

$ ssh -p 2222 root@127.0.0.1 "$P; cat /var/jb/allow_sb.txt | wc -l"
589
```

Doing a simple comparison:

```
$ ssh -p 2222 root@127.0.0.1 "$P; grep -Fvx -f /var/jb/allow.txt /var/jb/allow_sb.txt | wc -l"
350

$ ssh -p 2222 root@127.0.0.1 "$P; grep -Fvx -f /var/jb/allow_sb.txt /var/jb/allow.txt | wc -l"
136
```

The output above shows that 350 names are reachable by SpringBoard and not by the widget, and 136 are reachable by the widget, but not by SpringBoard. Moreover, they are 239 names in common.

A few notes about the referred program above and respective adopted procedure:

- This program can only be applied with a sandboxed process.
- It may help to kill potential leads before reading any code, which saves time.
- It grants us the possibility of getting the point of view of different process IDs and in different directions.
- The "allow" verdict does not mean "reachability" because there are other security layers that restrict access such as entitlements, MIG checks and even another control.

- The only tested operation was "mach-lookup", but a sandbox profile can permit the lookup and deny any follow up. Thus, the triage only covers Mach service surface, and for example, the IOKit question is even handled solved in this version.
- So far, the test has been done on a vPhone, but a reliable result should come from a real iPhone.

Therefore, this program only offers a reasonable view of the reachability of a given process via sandbox, and it is far from being really deterministic, and can and should be improved a lot.

As we have done every test on a jailbroken device (actually, a virtual device), so it would be more attractive to build a new version and slightly improved version that can be implemented into a real iPhone device. I have mentioned "slightly improved version" because while the existing "deny" answer could be strong evidence, the permission could be considered weak due to the fact that there are less restrictions on a jailbroken iOS. Additionally, we had assembled the corpus from 667 daemons, but this is our list and not the real names that the system actually has. Finally, as I explained above, the policy permitting lookup does not mean that the lookup operation really gets succeeded.

Therefore, the next version of sbmap program introduces a few news, where the name list is no longer gathered from separated plist files, but from `/System/Library/xpc/launchd.plist` file, and parsing this single file produces more than we had produced previously (I got 2473 names instead of 2330), so the new enumeration is better than the previous one. Another improvement is about the calibration, where for each candidate constant it asks about a name that cannot exist and about names pulled out from the corpus itself. Finally, this new version performs a real lookup as a confirmation phase, which produces a more reliable result.

The source code of `sbmap2.c` follows:

```
/* sbmap2.c -- it maps the Mach services a process is permitted to reach, and check
 * that answer against what actually happens.*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dlfcn.h>
#include <mach/mach.h>
#include <CoreFoundation/CoreFoundation.h>

extern mach_port_t bootstrap_port;
extern kern_return_t bootstrap_look_up(mach_port_t bp, const char *name,
                                       mach_port_t *sp);

#define PLIST    "/System/Library/xpc/launchd.plist"
#define FAKE     "com.apple.NOT.A.SERVICE.EVER"
#define TYPO     "not-a-real-operation-name"
#define SAMPLE   200

typedef int (*sbcheck_t)(pid_t, const char *, int, ...);
static sbcheck_t sb;

static char **names;
static int nnames;

/* ---- 1. the candidate names, from the device itself ----- */

static void add(const char *s)
{
    if (!strchr(s, '.') || s[0] == '#') return;
    for (int i = 0; i < nnames; i++)
```

```
    if (!strcmp(names[i], s)) return;
    names = realloc(names, (size_t)(nnames + 1) * sizeof *names);
    names[nnames++] = strdup(s);
}

static void collect(CFDictionaryRef d);
static void walk(CFTypeRef node);

static void collect_cb(const void *k, const void *v, void *ctx)
{
    (void)ctx;
    char buf[512];
    if (CFGetTypeID(k) != CFStringGetTypeID()) return;
    if (!CFStringGetCString(k, buf, sizeof buf, kCFStringEncodingUTF8)) return;
    if (buf[0] == '#') { /* a feature-flag wrapper */
        if (v && CFGetTypeID(v) == CFDictionaryGetTypeID()) collect(v);
        return;
    }
    add(buf);
    if (v && CFGetTypeID(v) == CFDictionaryGetTypeID()) collect(v);
}

static void collect(CFDictionaryRef d)
{
    CFDictionaryApplyFunction(d, collect_cb, NULL);
}

static void walk_cb(const void *k, const void *v, void *ctx)
{
    (void)ctx;
    char buf[128];
    if (CFGetTypeID(k) == CFStringGetTypeID() &&
        CFStringGetCString(k, buf, sizeof buf, kCFStringEncodingUTF8) &&
        !strcmp(buf, "MachServices") &&
        v && CFGetTypeID(v) == CFDictionaryGetTypeID()) {
        collect(v);
        return;
    }
    walk(v);
}

static void walk(CFTypeRef node)
{
    if (!node) return;
    CFTypeID t = CFGetTypeID(node);
    if (t == CFDictionaryGetTypeID())
        CFDictionaryApplyFunction(node, walk_cb, NULL);
    else if (t == CFArrayGetTypeID())
        for (CFIndex i = 0; i < CFArrayGetCount(node); i++)
            walk(CFArrayGetValueAtIndex(node, i));
}

static int load_names(void)
{
    FILE *f = fopen(PLIST, "rb");
    if (!f) return 0;
    fseek(f, 0, SEEK_END);
    long n = ftell(f);
    fseek(f, 0, SEEK_SET);
    void *buf = malloc((size_t)n);
    if (!buf || fread(buf, 1, (size_t)n, f) != (size_t)n) { fclose(f); free(buf); return 0; }
    fclose(f);
    CFDataRef data = CFDataCreate(NULL, buf, n);
    free(buf);
    CFPropertyListRef pl = CFPropertyListCreateWithData(NULL, data, 0, NULL, NULL);
    CFRelease(data);
    if (!pl) return 0;
}
```

```
walk(pl);
CFRelease(pl);
return nnames;
}

/* ---- 2. the filter constant, derived rather than assumed ----- */

static int calibrate(pid_t t)
{
    int lim = nnames < SAMPLE ? nnames : SAMPLE;
    for (int f = 0; f <= 15; f++) {
        if (f == 11) continue; /* filter 11 faults */
        if (sb(t, "mach-lookup", f, FAKE) == 0) continue;
        for (int i = 0; i < lim; i++)
            if (sb(t, "mach-lookup", f, names[i]) == 0) return f;
    }
    return -1;
}

/* ---- 3. the sweep, and check that it is telling the truth ----- */

int sbmap2_main(int argc, char **argv)
{
    pid_t target = (argc > 1) ? (pid_t)atoi(argv[1]) : getpid();
    int self = (target == getpid());

    sb = (sbcheck_t)dlsym(RTLD_DEFAULT, "sandbox_check");
    if (!sb) { printf("sbmap2: sandbox_check is not available here\n"); return 1; }

    if (!load_names()) {
        printf("sbmap2: cannot read %s -- no candidate names, nothing to do\n", PLIST);
        return 1;
    }

    int filter = calibrate(target);
    if (filter < 0) {
        printf("sbmap2: no filter separates a real service name from an impossible\n"
            "          one for pid %d. The process is unconfined, or the interface\n"
            "          has moved. Refusing to report on a map that would mean nothing.\n",
            (int)target);
        return 1;
    }

    FILE *fa = fopen("sbmap2-allowed.txt", "w");
    FILE *fd = self ? fopen("sbmap2-disagreements.txt", "w") : NULL;

    int allow = 0, look_ok = 0, both = 0, allow_only = 0, look_only = 0;
    for (int i = 0; i < nnames; i++) {
        int permitted = sb(target, "mach-lookup", filter, names[i]) == 0;
        if (permitted) { allow++; if (fa) fprintf(fa, "%s\n", names[i]); }
        if (!self) continue;

        mach_port_t p = MACH_PORT_NULL;
        int got = (bootstrap_look_up(bootstrap_port, names[i], &p) == KERN_SUCCESS);
        if (got) { look_ok++; mach_port_deallocate(mach_task_self(), p); }

        if (permitted && got) both++;
        else if (permitted) { allow_only++; if (fd) fprintf(fd, "policy-allows-but-no-service\n", names[i]); }
        else if (got) { look_only++; if (fd) fprintf(fd, "LOOKUP-WORKED-BUT-POLICY-DENIED\n", names[i]); }
    }
    if (fa) fclose(fa);
    if (fd) fclose(fd);

    printf("sbmap2 pid %ds filter %d control %s typo %d\n",
        (int)target, self ? " (this process)" : "", filter,
```

```
        sb(target, "mach-lookup", filter, FAKE) == 0 ? "BROKEN" : "denied",
        sb(target, TYPO, 0));
printf("        names      %d    from launchd.plist on this device\n", nnames);
printf("        policy     %d permitted   %d denied\n", allow, nnames - allow);
if (self) {
    printf("        lookup    %d succeeded   %d failed\n", look_ok, nnames - look_ok);
    printf("        agreement %d of %d permitted names really resolve\n", both, allow);
    printf("        %d permitted but not registered, %d resolved while denied\n",
        allow_only, look_only);
}
return 0;
}

#ifdef SBMAP2_APP
int main(int argc, char **argv) { return sbmap2_main(argc, argv); }
#endif
```

To test the new version of the program, we can first build and deploy it onto the vPhone repeating exactly the same steps we have done previously:

```
$ xcrun -sdk iphoneos clang -arch arm64e -O2 -Wall -Wextra -framework CoreFoundation -o sbmap2 sbmap2.c
$ ldid -S sbmap2
$ P="export PATH=/var/jb/usr/bin:/var/jb/bin:/usr/bin:/bin"
$ cat sbmap2 | ssh -p 2222 root@127.0.0.1 "$P; cat > /var/jb/sbmap2"
$ ssh -p 2222 root@127.0.0.1 "$P; chmod +x /var/jb/sbmap2"
$ ssh -p 2222 root@127.0.0.1 "$P; cd /var/jb && ./sbmap2 275"

sbmap2  pid 275    filter 2    control denied    typo -1
        names     2473    from launchd.plist on this device
        policy     387 permitted   2086 denied
```

In the new output above, the 143 extra named contributed with 12 new allowed names (387 rather than 375). Repeating the same for [SpringBoard](#), the result was 611 rather than 589. Moreover, any non-sandboxed process continue getting refusal than a number or any other output:

```
$ ssh -p 2222 root@127.0.0.1 "$P; cd /var/jb && ./sbmap2"

sbmap2: no filter separates a real service name from an impossible
        one for pid 18391. The process is unconfined, or the interface
        has moved. Refusing to report on a map that would mean nothing.
```

To build this program as an iOS application to install into a real iPhone we will need to a minimum project, which includes an application body ([App/main.c](#)), a project description ([project.yml](#)) and our program above ([App/sbmap.c](#)).

The application body ([main.c](#)) follows below:

```
/* main.c -- App entry point.*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int sbmap2_main(int argc, char **argv);

int main(void)
{
    char dir[1024];
    const char *home = getenv("HOME");
    snprintf(dir, sizeof dir, "%s/Documents", home ? home : "/tmp");
    if (chdir(dir) != 0) return 1;
    freopen("sbmap2.txt", "w", stdout);
```

```
setvbuf(stdout, NULL, _IONBF, 0);
char *av[] = { "sbmap2", NULL };
return sbmap2_main(1, av);
}
```

The project description (project.yml) is as follows:

```
name: sbmap2
options:
  bundleIdPrefix: com.example
  deploymentTarget:
    iOS: "26.6"
targets:
  sbmap2:
    type: application
    platform: iOS
    sources: [App]
    settings:
      base:
        PRODUCT_BUNDLE_IDENTIFIER: com.example.sbmap2
        PRODUCT_NAME: sbmap2
        DEVELOPMENT_TEAM: <your-team-identifier>
        CODE_SIGN_STYLE: Automatic
        GENERATE_INFOPLIST_FILE: YES
        MARKETING_VERSION: "1.0"
        CURRENT_PROJECT_VERSION: "1"
        INFOPLIST_KEY_LSRequiresiPhoneOS: YES
        INFOPLIST_KEY_UILaunchScreen_Generation: YES
        OTHER_CFLAGS: -DSBMAP2_APP
    dependencies:
      - sdk: CoreFoundation.framework
```

Having all files in your hands, the next step is to generate the project and build it, and you will need to replace the <your-team-identifier> with your own device identifier:

```
$ xcodegen generate
$ xcodebuild -project sbmap2.xcodeproj -scheme sbmap2 -destination "id=<device-identifier>" -
derivedDataPath build -allowProvisioningUpdates build
```

Install the application using the device identifier presented by the `xcrun devicectl list devices` command , which is not the same identifier as the device UDID:

```
$ xcodegen generate
$ xcodebuild -project sbmap2.xcodeproj -scheme sbmap2 -destination "id=<device-identifier>" -
derivedDataPath build -allowProvisioningUpdates build
```

To install the application, run the following command:

```
$ xcrun devicectl device install app --device <device-identifier> build/Build/Products/Debug-
iphoneros/sbmap2.app
```

The next step is to run the application. As a suitable note, you should remember that the first launch on the iPhone is refused, and the profile need to be explicitly trusted by the user, and it is done by going to [Settings | General | VPN and Device Management](#) and then trust the certificate. Afterwards, you can run the application by clicking on the application icon.

```
$ xcrun devicectl device process launch --device <device-identifier> --console <your-bundle-
identifier>
```

```
Launched application with <your-bundle-identifier> bundle identifier.
Waiting for the application to terminate...
The app terminated with the exit code 0
```


Once the application execution finishes, pull out results off the phone, as shown below:

```
$ xcrun devicectl device copy from --device <device-identifier> --domain-type appDataContainer --domain-identifier <your-bundle-identifier> --source Documents --destination ./out
```

Finally, we can read and list the three files retrieved from the application, which are saved in ./out folder, as shown below:

```
$ ls out
sbmap2.txt  sbmap2-allowed.txt  sbmap2-disagreements.txt

$ cat out/sbmap2.txt
sbmap2 pid 3244 (this process) filter 2 control denied typo -1
names 2476 from launchd.plist on this device
policy 376 permitted 2100 denied
lookup 349 succeeded 2127 failed
agreement 349 of 376 permitted names really resolve
27 permitted but not registered, 0 resolved while denied
```

The first line exhibits the pid and a new instrumentation status, while the second line shows the corpus used, which is pretty similar to the previous version. The third line shows us the police answer, where 376 lookup was permitted, but the fourth line shows the truth, where 349 of these named was really looked. Therefore, the last line indicates that 27 of those names are permitted, but they are not registered.

We will do further discussion about sandbox in future articles (actually, this first article is not the appropriate place to do it), but there are excellent existing tools that provide a useful and practical perspective about sandboxes by decompiling and reporting a binary Apple sandbox profile.

Likely the most known and used is **SandBlaster** (<https://github.com/malus-security/sandblaster>), which was written by Răzvan Deaconescu | Luke Deshotels | Mihai Bucicioiu | William Enck | Lucas Davi | Ahmad-Reza Sadeghi, and that helps to understand what a sandbox profile says. This tool has a fork kept by Cellebrite Labs (<https://github.com/cellebrite-labs/sandblaster>), it is updated for iOS 18 and 26.

There is another tool named **sbtool**, written by Jonathan Levin, which provides the possibility of inspect, query, and interact with live iOS sandbox processes and runtime sandbox profiles. Unfortunately, I could only find an old website with references to it (<https://newosxbook.com/articles/hitsb.html>). A third and great tool is the **sandbox_validator** from Karol Mazurek, (https://github.com/Karmaz95/Snake-Apple/blob/main/VIII.%20Sandbox/custom/sandbox_validator.c), which offers excellent results. Finally, another tool, written by Zhuowei Zhang, enumerates services from the launchd file and calls **bootstrap_look_up** on everyone from inside an application and prints those that succeed. To my simple experiment, the idea of pulling names from **launchd.plist** came from his tool. The program can be found here: <https://gist.github.com/zhuowei/371d43b3e33f025f8db25390addf41a8>.

Once again, it is fundamentally important to highlight that both applications (sbmap.c and sbmap2.c) are small experiments to demonstrate a few practical and technical aspects of checking for the reachability of a sandboxed application. Obviously, there are multiple points that need to be covered and improved, a series of imprecisions that should be worked on, and actually this simple tool developed throughout this section uses ideas already adopted by Karol Mazurek and Zhuowei Zhang. Therefore, this draft tool combines ideas from both into a single code, and introduces the calibrating phase, which makes it trustworthy. As this test code was developed to this section only, readers should learn about tools from

the mentioned authors, which are polished and cover the sandbox reachability from different perspectives. By the way, I don't have any intention to write a public tool again.

Returning to the exploitation path, the second stage is the entitlement, which is a signed claim embedded directly within the application's code signature, and that serves as a granular access control mechanism that dictates the boundaries of the sandbox, controlling exactly which restricted system capabilities the application is allowed to access. For example, while the sandbox acts as the enforcement engine that blocks or allows access to a kernel driver, a specific entitlement is required to unlock that driver's interface and dictate exactly which internal user-client methods the application is authorized to call. In other words, entitlements establish granular control access and, in this case, migrate the decision about whether the code can or cannot access the driver to what functions from driver can be contacted.

While an attacker might find an alternative path to bypass or work within a sandbox profile, entitlements present a strictly enforced secondary access control because entitlements are embedded directly into the binary and any attempt to modify them corrupts the digital signature and prevents their execution. Therefore, simply finding a route through the sandbox is not enough because a missing entitlement can completely block access to the target.

If readers are missing AMFI layer, it acts when application process starts and checks the signature and installs entitlements that will check layer, but during the exploitation process, it has already been consulted at beginning and will not be checked again.

So far, the summary of events since the application launch would be roughly the following one:

- The application process starts.
- AMFI check the signature and install the respective entitlement set.
- The sandbox profile is applied.
- It is necessary to have a file descriptor to access, a service to look up or, even better, a Mach name (it is small integer that works as a handle and that is meaningful only inside the task -- it is very similar to the handle concept on Windows). In next sections and future articles, I will explain multiple details about Mach concepts and respective mechanisms.
- It is checked whether the Mach name holds the appropriate right.
- The sandbox policy is checked to authorize or not the operation.
- Depending on the target, a new entitlement might be requested, a credential might be evaluated or even another process could authorize or not the operation.
- Other security checks could happen.

The third stage is finding a bug, which has become harder year after year, and mainly in the current iOS 26 and iOS 27, which continue receiving patches from Apple, who also has introduced protections and compiler improvements that close several bugs and exploitation doors.

In the past, ordinary C pointer was only an address, and nothing more. These C pointers do not carry any information of how much memory are pointed at them, which enforces developers to care about every single bound check, and any flaw in the programming enables the possibility of running of the end of the buffer, which is one of causes of an entire vulnerability class like buffer overflow. Extensions like [-fbounds-safety](#), applied by Apple, change this scenario because sizes are declared alongside the pointers they describe (different from ordinary C pointers), and the compiler inserts a check ahead of any access.

Therefore, when such a check fails, the code does not continue on with a corrupted neighbor, traps immediately, and in the kernel a trap is a panic.

The most relevant fact is that access does not even occur, so nothing is effectively leaked or overwritten in case of read or write operations. As a consequence, a crash does not allow the exploit sequence to step forward because the system should be left stable from one stage to the next one, and such a crash destroys running state and, worse, the entire memory shaping built up to that point is lost. In such a context, we should remember that the panic (crash) is the protection working as it should do, and it is not evidence that a bug or vulnerability has been found, even because Apple pays for evidence that a security boundary has been compromised, and this is not definitively the case. On the other hand, a panic that is caused by the kernel dereferencing an address that should never have been done it is a distinct case because it shows that a corrupt could really have occurred.

A similar protection is a guard page, which acts in runtime and leave an unmapped page close to next allocation. Thus, if any write operation touches on this page, a fault is raised. Bound checking is different from guard pages because the access does not even take place at all, and this behavior is dictated by the compiler, who knows the allocated size and emits a kind of checking before the access, so if this comparison fails, a trap instruction is sent. While an overflow could happen in the scenario using guard pages, if the attacker control heap overflow, without crossing any boundary, it might succeed. However, using boundary safety, the operation even occurs, and no address is ever dereferenced.

Of course, depending on the type of vulnerability (bug), the exploitation process could be easier or harder than expected, and artifacts on hands can not be enough to for getting kernel read and write primitive later.

Readers should know one of possible general bug classifications, where a few classes are shown below:

- Spatial bug is the capability of writing to the wrong place, which an index or a length run past the end of a given object and reaches the adjacent object.
- Lifetime bug is different because it is the capability of writing to the wrong object, which might be a kernel object that has been already freed. For example, the Mach path probably will need a lifetime bug because, in most cases, the premise is that a port dies while something still refers to it. An alternative would be, as explained above, to get a virtual memory bug which keeps page mapping intact even after the page has been freed. In this case, a typical overflow would be useless as first alternative, but spatial writes can be converted into lifetime bugs.
- Type confusion bug occurs when a read or write operation target the wrong type.
- Race condition bug depends essentially on the check and time to use a given object.
- Logic and privilege bugs are based on code decisions, algorithms, and occasional mistakes.

It is pretty direct to conclude that vulnerability research and exploit development are not only about boundary limits, and we can find lifetime bugs, type confusion, race condition (including double fetching and Toc-Tou cases), and mainly logic and privilege bugs.

On modern iOS versions, kernel keeps allocations that might contain pointer in distinct regions of the address space from allocations that only contain data (data-only). Thus, an overflow inside the data-only heap is not possible to corrupt any pointer, object headers, function table addresses because overflowed data is in a region and typed-information (pointers) is in a completely different region. Personally, I have already faced this condition in some opportunities, and one of these cases will be shown and explained in

next articles. Anyway, that is why a bug that is natively a lifetime or reference count error is worth far more than a write that merely sounds more powerful.

On the other hand, we could use an existing write capability to convert a spatial bug into a lifetime bug. For example, re-count field from kernel objects stores its count as an integer inside its own memory, and the kernel frees a certain object at the moment that integer reaches zero. Using the existing bug with write capability could be used to change one of those integers and lower it, which would cause the kernel free an object while other components of the system were still using it, resulting in a lifetime bug. Evidently, such manipulation is not easy to be performed due to exactly the same argument above, because reference counts live inside typed objects, and typed objects are what the data heap separation keeps beyond the reach.

The fourth stage in a hypothetical exploit development path is turning a bug into a solid primitive, and there is no easy way and even a standard procedure because there are too many variables that need to be considered, but at the end, we want a controlled and repeatable result.

The achieved primitives compose the capability that it is generated from the vulnerability, and such primitives could provide attackers with the capability of placement (putting an object of attacker's choice at a given address), controlled free (release a chosen object at the appropriate moment), limited write (a normal write, but without controlling value, length and offset), a controlled write (the attacker controls the value and address, but not length or offset), limited read (only a provided number of bytes are read) and arbitrary read-write (attacker controls value, address, length and offset). Therefore, getting a primitive is an essential step, but prove itself as a priceless stage when you get the right primitive.

The fifth stage is to obtain a kernel read and write primitive. First of all, being succeed at this stage is really challenging and, as you might expect, it is not something you will do in one day or overnight. The number of paths are really limited, and two well-known approaches are either trying to follow through Mach port way, and find a lifetime bug on a typed object (an ipc_port), reclaim the freed object, forge a fake kernel task port, and occasionally to get Mach virtual memory read and write access via mach_vm_read and mach_vm_write functions (SockPuppet is a great example of exploit using this path:

<https://projectzero.google/2019/12/sockpuppet-walkthrough-of-kernel.html>) or getting physical user-after-free on memory to read and write raw bytes using an ordinary load or store (PhysPuppet as example: <https://github.com/felix-pb/kfd/blob/main/writeups/physpuppet.md>) by triggering a virtual memory or page handling bug, dangling a PTE over a free physical page, reclaiming the same physical page and finally reading and writing raw data through the dangling PTE. Each step can take weeks and months to be researched and executed successfully because there are protections such as KASLR (you will need the slide), PAC (already mentioned, and it must be considered because pointers are signed) and zone isolations, for example, which can bring more difficulties for the goal. Both SockPuppet and PhysPuppet are excellent examples of exploitation, and PhysPuppet technical is quite interesting because it works below the software mitigation, beyond the fact that in physical UAF, kalloc_type memory type isolation and the data heap separation do not even apply to the scenario.

Finally, the sixth stage is getting more than kernel read and write primitive, which provide us with privilege escalation, but not kernel code execution or even the possibility of leaking an important key or secret from memory. The limitation are monitors like SPTM, which own the page table and control their access, and TXM, which own the code signing and trust decision. Breaking these monitors is something really hard and definitely the last stage, and it should be interpreted as a daily and long exploitation journey. At this point,

an excellent reference is Titan project (<https://github.com/alfiecg24/Titan>), which is owned by Alfie (@alfiecg_dev on X), and that shows a PPL and SPTM bypass for iOS 16.1 - 17.4 beta 3, supporting A14 - A17.

Just to not lose the perspective of the exploitation path, we should remember that, in the middle of the this mentioned path, PAC is protecting and ensuring the integrity of pointers, but not the integrity of data, indexes and lengths. Similarly, SPTM (Secure Page Table Monitor) and TXM (Trusted Execution Monitor) are software sub-systems that moved monitoring and enforcement out of the core XNU kernel, where previously were tasks previously managed by the PPL (Page Protection Layer). They are strictly isolated by hardware architecture (via Guarded Exception Levels or GXF) proposed by Apple Silicon. The kernel interacts with them through pre-determined interfaces (like `txm_kernel_call` and `GENTER` instructions), and any unauthorized attempt by kernel to write to the protected memory of monitor triggers a hardware fault or panic rather than a standard exception. Therefore, both protections (PAC and SPTM/TXM) need to be considered as part of serious iOS research, and soon or later they will have to be handled.

Certainly, most researchers like to research for potential memory vulnerabilities and techniques as the classic heap groom is the first to come up, where the attacker free objects of one type (X), spray objects of other type (Y), and an object of type Y can assume the memory slot that belonged to type X. The existing protection `kalloc_type` (quickly mentioned previously) establishes split allocations across separate zone, so that a hole left by an object of type X is no longer freely fillable by an object of type Y. As you will see, the kernel describes and names these zones in a pattern like `kalloc.type[N].[size]`, and associated with each of them, there is a distinct `kalloc.type.var` family for variables cases. Thus, the granted name is one of best ways to conclude the type being referred to when we analyze a panic file. Undoubtedly, the solution is simple and elegant because iOS does not mix different things (types) in the same basket (zones).

Another interesting aspect of this scheme is that a second heap feature is derived from this idea, and pure data (`kalloc_data`) zones are always separated from zones carrying pointer (`kalloc_pointer`), so zones will be named `data.kalloc.[size]`, and this name format is attributed by the kernel. As consequence, even if we are able to read beyond a data zone, we will leak (at maximum) data from the subjacent data zones and not a pointer because data and pointers are placed in different locations in the memory.

Whereas `kalloc_type` works on proposing the segregation of allocations, by placing objects of different types in different zones and preventing that a corrupted object of one type cannot be reinterpreted as another, there is another layer of protection related to heap shape, which is named `out-of-band freelist metadata` that work by removing the own state of the allocator from the reach of any corruption entirely, so providing a complement to `kalloc_type` because it prevents that an attacker can write "poison" any allocation pointer. Explaining a bit of the context, in old versions of the zone allocator, they stored the "next free element" link inside each free slot, where the first 8-bytes of a free element were a pointer to the next free element. As expected and also based on similar experiences on other operating systems, an attacker could write into a free slot (through an overflow from the previous element or even a use-after-free), overwrite such a pointer (known as in-band freelist metadata) and choose the address that the allocation would return on a future `zalloc` function call and, as a result, it allowed attacker to control where an allocation was placed. This scenario has changed in modern XNU, where zones are bitmap-managed and there is not this kind of list at all. Elements of a chunk that are currently free are stored as a bitmap in the metadata of the own chunk (`zone_page_metadata`), which is kept away from elements themselves.

Therefore, there is not a pointer in a free element to overwrite. This mechanism is known as out-of-band freelist metadata.

I mentioned the term "zone" in the paragraphs above, and apparently out of the appropriate context, but an explanation could be useful here. The heap memory is divided into zones, and each zone holds a set of slots of the same size, which is given by a size class that is one of the fixed sizes that the allocator provides (similar to buckets from LFH on Windows, for example), where every request is rounded up to the next available size (for example, hypothetically, for a request of 1003 bytes would be destined a class of 1024), and that is exactly the mentioned size class. As a result, for a type C allocation from kernel source code, such allocation goes to a determined zone, where all slots have the same size.

The allocated C type derives the following information:

- Its size is rounded up to a determined size class.
- Its C type defines the granule shape, which is used to define a signature.

Based on these two sentences above:

- The granule shape is composed of several granules, whose number of granules depends on the type.
- This type has a declaration associated with it.
- Each granule has eight bytes, which can hold a padding (character 0), a pointer (character 1), plain data or scalar (character 2), pointer and/or data (character 4), a signed pointer (character 8), PAC or pointer (character 9) and PAC or data (character a) but only one of them at a time (readers can check it in [osfmk/kern/kalloc.h](#) in the XNU source file). In fact, where one of them holds two "types" the value is bitwise OR, and which is why a signature is written in hexadecimal, but this occurrence is far from being common.
- From each granule originates a character that depends on what type of information it is holding (pointer, data, signed pointer), and concatenation of these characters is the signature.
- Every zone carries a label that is defined by the composition of the size class followed by the signature.

Two different structures with the same size class can have the same signatures if they have the same sequence of granule classes. As examples, some structure definitions and respective signature interpretation follows below:

- `struct A { void *owner; uint64_t id; void *next; uint64_t state; }; (32 bytes)`
- `struct B { void *buf; size_t len; void *cb; uint64_t flags; }; (32 bytes)`
- `struct C { uint64_t a; uint64_t b; uint64_t c; uint64_t d; }; (32 bytes)`
- `struct D { void *head; uint64_t count; void *tail; }; (24 bytes)`
- `struct E { void *p0; uint64_t n0; void *p1; uint64_t n1; void *p2; uint64_t n2; }; (48 bytes)`

▪	granule	0	1	2	3	4	5	6
---	---------	---	---	---	---	---	---	---

- | | | | | | | | |
|---|---|---------|------|---------|------|----------|-------------|
| ▪ | A | pointer | data | pointer | data | → "1212" | (signature) |
| ▪ | B | pointer | data | pointer | data | → "1212" | (signature) |
| ▪ | C | data | data | data | data | → "2222" | (signature) |
| ▪ | D | pointer | data | pointer | | → "121" | (signature) |

- E pointer data pointer data data pointer data → "121212" (signature)

The order of fields is directly associated with the signature too. However, a further detail is that a zone has a label (defined by the composition of the size class and signature), but the inverse is not true. A label is associated with one or more zones (a family), but it is decided at the boot time, and this decision is based on a budget, which is a global number, `kt_fixed_zones`, whose default values comes from `ZSECURITY_CONFIG_KT_BUDGET=260` (check `osfmk/kern/kalloc.c` and `osfmk/kern/zalloc_internal.h` files). There is a formula defined as $zbudget_sig = 7 * budget / 10 = 7 * 260 / 10 = 182$ signature zones, and the remaining 78 are typed zones, where for one size class the kernel creates an array of zones (`n_zone_sig + n_zones_type`), slots from 0 to `n_zones_sig-1` are occupied by signature zones, and slots from `n_zones_sig` to the end are occupied by typed zones. As there are 29 size classes, each half is spread on these 29 size classes, with a floor of two signatures zone per class.

Observing definitions above, structures A, B, C and D have the same size class, but only the first two belong to the same family. To every distinct signature, it is given one signature-zone as its home, while an extra room (typed zone) is reserved for a given signature, and as explained previously, typed zones exists to keep different types separated from one another. Therefore, a family is its signature zone plus its own typed zones, and the usage of zones inside a family is established in round-robin. At this point, there is an important detail because zones of a family are put in a random order, a counter walks that order one type at a time, and every time the counter returns to zero, the order is re-shuffled into a new one. As a result, each zone of the family receives the same number of types, but the allocation of a given "type" cannot be predicted from the order in which they were declared.

For now, a few conclusions follow:

- a typed zone has only a signature, by construction.
- a signature-zone has only one signature where a size class has fewer signatures than signature zones, but on iOS 26.5, iOS 26.6, and iOS 27.0 (beta versions) it is not applied to none of the 29 classes. Thus, based on the `zone = kt_zones_for_size[j % n_zones_sig]` (found in `osfmk/kern/kalloc.c`), there is around ten signatures per signature zone.
- a signature: it is attributed to the respective signature zone, and additional typed zones when the budget allows.

Now you have the necessary foundation about the allocation imposed by `kalloc_type`, it is the time to understand what it is the influence of it to prevent the usage of use-after-free technique. The better way to highlight the effect of the feature is by comparing a scenario without type isolation and others with type isolation. Anyway, the context is the same, where some code frees an object while a pointer to it is still live, and the kernel later loads a field from that freed object and uses it. Thus, the central point of our discussion will be the reclaim step, and we will learn what we are able to allocate into the available hole before the kernel code reuses it.

As an example, we can assume a structure as the struct B from previous page, where there is a pointer, a length, another pointer, and flags. Therefore, there are four granules (signature 1212), and kernel loads these two pointers and dereferences them later.

On a kernel scenario without type isolation, any reclaim is unrestricted. Thus, once we free the victim, we can spray any 32-byte data buffer (an out-of-line message, a pipe buffer or even an OSData), and one of them will be placed into the hole. From this point, when kernel dereferences the address stored there, it

will be the address we have chosen putting into there. That is a classic groom followed by UAF, which has been done over year in different operating systems.

On the current version of iOS kernel, which has type isolation, our spray consists of data (an out-of-line message, a pipe buffer, an OSData). None of those types holds a pointer, so its granules are padding and plain data only, which are two of the five granule classes. The compiler sees it and tags the type as `KT_DATA_ONLY` (check the `osfmk/kern/kalloc.h` file). Therefore, it does not matter what is later written into the buffer because the flag was decided from what type it can hold, and not from what it ends up holding.

As a result, every allocation of such content is placed into the data heap instead of into the `kalloc_type` zone array, and such the placement decision is done at boot when the kernel reads the flag. That is the reason why the spray can never reclaim the victim's slot. The data heap is not only a different zone, but it is a separated region of the address space, and an allocation cannot be placed anywhere except in the zone its type was assigned.

As an alternative, we could look for a structure with the same shape, which would be placed in the same zone. This effect can be achieved by matching the same signature, and this means that type pointers must be positioned at the same granules as the victim's pointer. Additionally, we cannot make up such a structure because it has to be a type that the kernel already declares, and one we can make the kernel allocate on demand.

Based on previous paragraphs, it is feasible to understand that the mentioned signature ensure that the content filling the available holes contains pointers at the same locations where the victim had them previously, so any field that the kernel is about to reference is always a pointer written by a legit (and not attacker) source. I will return to this same context a couple of pages ahead to reinforce the idea.

As we have already noticed, attacker's possibilities are more limited that they first sounds. While attackers cannot submit chosen bytes as desired into this zone, length and flags are still controllable while building the given struct B, and this demonstrates an indirect control. Anyway, the scenario is considerably worse than would be without having the type-isolation because now it is necessary to control the size (to match the victim class), the signature (and consequently the zone) to match the victim's signature and be distributed within the same zone as the family of that signature through the "shuffling" performed during this initialization, which is why the zone map is discovered at runtime, and finally attacker must control fields that provide an indirect influence (due to its type) over a granule, but don't forget the is the kernel that has the power to write pointers, and not the attacker. At the end, the aftermath of type isolation is that any attempt to write controlled data buffers cannot reach an object holding kernel pointers because both (data buffers and kernel pointers) are located in distinct virtual address regions, which effectively makes more difficult to attackers to overflow data heap and cross from one zone to another and write any kernel pointer.

As an important note, sharing only a signature does guarantee nothing because to a signature is given a family of zones and not a specific zone, and two different structures may be treated into different members (zones) of this family. Furthermore, there is far more signatures than there are zones to give them, so the rule is to have multiples signatures in the same zone.

This topic deserves to be a bit more detailed, even though it has already been mentioned previously. While one size class is associated with an array of zones as a structural relationship (one-to-one), the relationship

between signatures and families is based on a claim, and granted at start up, where many signatures can be associated with a given zone.

As the reader will recall, signature comes from the granule, and the signature S maps to exactly one family. An allocation, however, is attributed to a single zone rather than to a family, where the signature selects the family, the type and then selects a zone within it, and that binding is made per type, once, at start-up. Thus, once a type is bound to a zone, its allocations never cross into another, because each allocation site that the attacker can trigger feeds one zone and only this zone. As a consequence, the challenge is to pick up the correct call or interface that allocates into the victim's zone, and spraying there, but if it does happen, the spray will not achieve any useful result.

Two further points are worth reinforcing: a zone can hold several signatures at once, and a family consists, most of the time, of a single zone. The association of a size class with an array of zones, and how such an array is laid out once at startup (again, this fact is important), follow concepts below:

- array: X elements of signature zones + Y elements of type zones
- The X elements of signature zones has the following working: once the cycle closes, it repeats itself.
- The Y elements of type zones are handed out in exclusive slices, so they are never re-offered.

To two distinct signatures, A and B, each one build a family out of that one array, where:

- a family of signature A is composed of the zone 01 + typed zones.
- a family of signature B is composed of the zone 01 + none type zone.

As a result, both signatures A and B share the same home zone (zone 01 -- the zone that the signature has been attributed at startup), and nothing more.

To be clear at this point: an array is only a flat list of zones, created for one size class at start-up, and every zone that is created for this arrange belongs to exactly one array. On the other hand, a family is a set of zones, which have been assigned to a single signature at start-up, and every member of it come from the array of one size class. Thus, there is no call that creates a family and neither a structure in the kernel to represent a family, and zones are merely tagged with an identifier that determines which family they belong to.

Taking the same mechanisms from a sharp point of view, a few conclusions come up:

- An array is storage, defined at the startup, while a family is a set of zones extracted from this mentioned storage.
- There is one array per size class while there is one family per signature.
- A family never spans two arrays because a signature belongs to a size class.
- One array holds many families, and the practical account is roughly ten of them sharing each signature zone on the builds that I could measure.
- Families can overlap at the home zone while arrays cannot because two signatures share the same home zone, whereas no zone can exist in two arrays.

If we check [osfmk/kern/zalloc_internal.h](#) file (around line 480) and [osfmk/kern/kalloc.c](#) (around line 1538), we obtain the following information:

- 260 zones for the whole kalloc_type.
- 182 of them are signature zones.

- 78 of them are type zones.

A few experiments, not shown here, shows that there is an average between 10.1 (iOS 26.5) and 10.5 (iOS 27.0 beta 6) signatures per zone.

Proceeding to attacker perspective, and as we already seen previously, to get a UAF is not simple as it seems. First, to get an object really freed and not its reference count going down by one. Occasionally, we also need an oracle that identifies the object to be sure that the object is really free. Following the standard vulnerability procedure, the objective is to cross from a data heap to the hole from the typed zone that we match the signature, and there are two ways for the allocation to land in the victim's zone, where the first consists of carrying the victim's signature and being assigned to the same member of its family, and the second one is to carry a different signature, but one that has already been assigned to the same home zone, and as quickly mentioned, there are roughly nine other possibilities at any moment.

Both approaches would be great for attackers, but we cannot resolve any of this in advance because we cannot read a structure definition or know which zone the victim occupies mainly before the shuffle occurring at the boot prevents knowing what happen in the next boot. At this case, it should be discovered at run time while the attack is happening.

As occurs for any UAF case, the reclaim is the next step, where the victim object hopefully has gone, the slot is available again and we need to allocate something under our control into the free slot before kernel to read the pointer that it is holding. One of possible alternative for us would be to try a type-confusion attack, which makes part of lifetime vulnerability. The real task is not only to force the kernel to read a pointer pointing to a different object type, but to learn what other types live in this zone on this boot, and how we can allocate them.

The first challenge is that the attacker (us) needs to narrow the number of possibilities. For example, there are around 1,900 signatures in kernel, which can be read from the kernelcache, and it is a lot, but the size class of our victim is associated with about 60 or a bit more, and it is much better. However, having around of 60 signatures (shuffled at boot time) does not mean that we can reach all of them, and the zone shared with the victim has about 10 signatures associated, but it is our final frame. However, there is another obstacle because allocation happens per CPU magazine (this concept will be discussed on future articles), which means that a free operation followed by an allocation usually does not return the same element.

If we are succeed to manage the reclaim stage, we still need a couple of leaks (and having the slide would be useful), which the first one is the object address that says where one of our heap objects is located, and this information confirms that our proposed object was allocated into the free slot and also give us the reference for writing at memory heap. The second leak, and that reader will also identify easily, is the address of a provided function name. Finally, the last stage is to manage to effectively force the kernel to use the implanted pointer (could be also an index or length), which would conclude the attack.

We should also confirm that a reclaim really worked, and this can be done using own kernel to identify whether there is an object in the slot, and what is there. Basically, it is done by inference because if we allocates something and kernel behaves differently from before, we conclude our allocated object landed in the free slot. Afterwards, we need to ask kernel which object has been allocated in the free slot.

The mach_port_space_info function returns the port name table of the task, and this table contains right, user reference and identifier (it is not an address, obviously) for each object behind it. This identifier is like

a token (as a handle from a Windows handle table), which is resulting from a pointer that has hashed and salted, which it is not possible to decode (at least, I cannot). However, it is feasible to do a comparison because, as the identifier is originated from hash, it ensures that it comes from only one hole (location), and that the allocated object into this free slot inherits the token from the freed object that was in there. Thus, an attacker can confirm the ownership of allocated object in that specific free slot. Without entering into unnecessary details now, the idea would be to take a snapshot of space, trigger the free, run a batch of allocations and then take a new snapshot. As expected, we would investigate differences between two snapshots.

Although most of time we are looking for ways to perform controlled write in key structures and kernel locations, information leaks are also equally relevant mechanism because it helps us to know the address of allocated object, and it confirms that our allocation in the scenario exposed above landed in the expected slot. Inevitably, the slide (or KASLR slide), which dictates the exact randomization offset applied to the kernel image at boot time, exerts an influence on information leaks because it tells us the offset the image moved at boot. Circumstantially, it is exactly such a slide that provides us with the possibility to translate a given name into a runtime address, and we have to subtract it from functions, object addresses, virtual table pointers, and other involved structures. The point is that we need a leaked kernel address (the runtime component), which is never easy. Once we have the runtime address, we can use the linking time address (from kernelcache) and subtracting one from another results to the slide.

The kernel image is built at a fixed nominal base, and iBoot is responsible for allocating it, but it is the kernel that does the rebasing. To explain how it works, first it is required to remember that this mechanism is reflected in the kernelcache that carries `LC_DYLD_FIXUPS`, which is a loader command that points at the metadata a loaders needs in order to fixup every pointer in the image that must change when the image is not allocated at its link address. For professionals who come from Windows world, `LC_DYLD_FIXUPS` is a kind of parallel of .reloc sections, but with a few differences because while .reloc section has its own section and pointer slots are kept alone, chained fixups keeps the list in the slots.

Actually, both formats (PE and Mach-O) are patched at load time, but in PE the slots holds a valid and complete address (raw) that is computed for the preferred base, which can be read statically, and it is only added an offset between the disk and memory, but such an offset can be zero whether the PE is loaded in the preferred address.

On the other hand, Mach-O the slot holds an encoded value, which is not an address, and that will be added to the base address once it is loaded. Additionally, there are other differences because on PE the rebasing is based on .reloc section and import resolution consists of Import Directory and IAT (Import Address Table), while chained fixups also carries the signing policy and PAC stuff.

Returning to the iBoot context and slide topic, pointers inside the kernelcache are still in chained form when iBoot hand it over to kernel, so what it contains are addresses it was linked at. The kernel holds a "link address" as a compile-time constant and separately learns where is has really been placed. The difference between both values (runtime value and compile-time constant) is the slide. This slide can be applied alongside the fixup chains, and it transforms the image's pointers to the real runtime addresses. This slide is per Mach-O image, and not per section, and within one image every segment shift by the same amount. Additionally, the slide is regenerated every boot. However, everything we have discussed is about kernel slide, but there is a slide per executable launch, one slide for dylibs in the shared cache (common to all of them) and one slide for every non-cached dylib.

To provide readers with some practical steps, you can download latest iOS image or download only the kernelcache (in my case, I have an iPhone 16 for tests), which is recommended at this initial part of our article (and series), even because it takes only 74 MB and not 10.48 GB:

```
$ ipsw download appledb --api --os iOS --device iPhone17,3 --build 24A5430a --kernel --confirm
```

- Querying AppleDB...
- Parsing remote IPSW build=24A5430a devices=iPhone17,3 version=27.0
- Extracting remote kernelcache
 - Created 24A5430a__iPhone17,3\kernelcache.release.iphone17
 - Created 24A5430a__iPhone17,3\kernelcache.research.iphone17

```
$ ipsw kernel version kernelcache.release.iphone17
```

```
Darwin Kernel Version 27.0.0: Thu Aug 13 22:24:48 PDT 2026; root:xnu-13432.2.10~5/RELEASE_ARM64_T8140
```

If you are working on a system that does have ipsw tool installed, you can get the same kernelcache information by executing:

```
$ python -c "import re; d=open('kernelcache.release.iphone17','rb').read(); print(re.search(rb'Darwin Kernel Version[ ~]{0,160}', d).group().decode())"
```

```
Darwin Kernel Version 27.0.0: Thu Aug 13 22:24:48 PDT 2026; root:xnu-13432.2.10~5/RELEASE_ARM64_T8140
```

Unless we are exploiting real vulnerability, we do not have any leak from iPhone and, for our tests here, using vPhone is easier because it allows us to establish a shell and retrieves the slide directly from there. Of course, we can use a panic file from iPhone 16 (iOS 27 beta 8) to extract the slide too.

Before proceeding, it is recommended to do a quick examination in the kernelcache we have just downloaded and learn a few information stored in there:

```
$ ipsw macho info --no-color kernelcache.release.iphone17 | head -8
```

```
Magic      = 64-bit MachO
Type       = FILESET
CPU        = AARCH64, ARM64e caps: KER00
Commands  = 319 (Size: 22960)
Flags      = None
000: LC_UUID              16FF5BB5-E04D-6DD5-50F2-C6623CF19A56
001: LC_BUILD_VERSION     Platform: Unknown, SDK: 0
002: LC_UNIXTHREAD        Threads: 1
```

```
$ ipsw macho info --no-color -t com.apple.kernel kernelcache.release.iphone17 | grep -E "__TEXT($|_EXEC)"
```

```
003: LC_SEGMENT_64 sz=0x008eb458 off=0x03a5c000-0x04347458 addr=0xffffffff00aa60000-
0xffffffff00b34c000 r-x/r-x __TEXT_EXEC
      sz=0x00001000 off=0x03a5c000-0x03a5d000 addr=0xffffffff00aa60000-0xffffffff00aa61000
__TEXT_EXEC.__exc PureInstructions|SomeInstructions
      sz=0x008e94f0 off=0x03a5d000-0x043464f0 addr=0xffffffff00aa61000-0xffffffff00b34a4f0
__TEXT_EXEC.__text PureInstructions|SomeInstructions
      sz=0x00000f58 off=0x04346500-0x04347458 addr=0xffffffff00b34a500-0xffffffff00b34b458
__TEXT_EXEC.__hib_text PureInstructions|SomeInstructions
```

```
$ ipsw macho info --no-color kernelcache.release.iphone17 | grep _TEXT
```

```
004: LC_SEGMENT_64 sz=0x00008000 off=0x00000000-0x00008000 addr=0xffffffff007004000-
0xffffffff00700c000 r--/r-- __TEXT
005: LC_SEGMENT_64 sz=0x00d94000 off=0x00008000-0x00d9c000 addr=0xffffffff00700c000-
0xffffffff007da0000 r--/r-- __PRELINK_TEXT
      sz=0x00d94000 off=0x00008000-0x00d9c000 addr=0xffffffff00700c000-0xffffffff007da0000
__PRELINK_TEXT.__text PureInstructions|SomeInstructions
```

```
008: LC_SEGMENT_64 sz=0x02f54000 off=0x013fc000-0x04350000 addr=0xffffffff00840000-
0xffffffff00b354000 r-x/r-x __TEXT_EXEC
009: LC_SEGMENT_64 sz=0x00008000 off=0x04350000-0x04358000 addr=0xffffffff00b354000-
0xffffffff00b35c000 r-x/r-x __TEXT_BOOT_EXEC
```

```
$ ipsw macho info --no-color -t com.apple.kernel kernelcache.release.iphone17 --loads | grep
LC_UUID
```

```
022: LC_UUID                                01F967F2-F645-384D-997C-F97EBF4620B9
```

```
$ ipsw macho info --no-color --loads kernelcache.release.iphone17 | grep "[0-9]*: " | head -20
```

```
000: LC_UUID                                16FF5BB5-E04D-6DD5-50F2-C6623CF19A56
001: LC_BUILD_VERSION                      Platform: Unknown, SDK: 0
002: LC_UNIXTHREAD                        Threads: 1
003: LC_DYLD_CHAINED_FIXUPS                offset=0x004988000 size=0x538
004: LC_SEGMENT_64 sz=0x00008000 off=0x00000000-0x00008000 addr=0xffffffff007004000-
0xffffffff00700c000 r--/r-- __TEXT
005: LC_SEGMENT_64 sz=0x00d94000 off=0x00008000-0x00d9c000 addr=0xffffffff00700c000-
0xffffffff007da0000 r--/r-- __PRELINK_TEXT
006: LC_SEGMENT_64 sz=0x00614000 off=0x00d9c000-0x013b0000 addr=0xffffffff007da0000-
0xffffffff0083b4000 r--/r-- __DATA_CONST
007: LC_SEGMENT_64 sz=0x0004c000 off=0x013b0000-0x013fc000 addr=0xffffffff0083b4000-
0xffffffff008400000 r--/r-- __DATA_SPTM
008: LC_SEGMENT_64 sz=0x02f54000 off=0x013fc000-0x04350000 addr=0xffffffff008400000-
0xffffffff00b354000 r-x/r-x __TEXT_EXEC
009: LC_SEGMENT_64 sz=0x00008000 off=0x04350000-0x04358000 addr=0xffffffff00b354000-
0xffffffff00b35c000 r-x/r-x __TEXT_BOOT_EXEC
010: LC_SEGMENT_64 sz=0x00280000 off=0x04358000-0x045d8000 addr=0xffffffff00b35c000-
0xffffffff00b5dc000 rw-/rw- __PRELINK_INFO
011: LC_SEGMENT_64 sz=0x002f0000 off=0x045d8000-0x048c8000 addr=0xffffffff00b5dc000-
0xffffffff00b8cc000 rw-/rw- __DATA
012: LC_SEGMENT_64 sz=0x000c4000 off=0x048c8000-0x0498c000 addr=0xffffffff00b8cc000-
0xffffffff00b990000 r--/r-- __LINKEDIT
013: LC_FILESET_ENTRY                      offset=0x000008000 addr=0xffffffff00700c000 com.apple.kernel
014: LC_FILESET_ENTRY                      offset=0x000114000 addr=0xffffffff007118000 com.apple.kec.Libm
015: LC_FILESET_ENTRY                      offset=0x00011b0a0 addr=0xffffffff00711f0a0 com.apple.kec.pthread
016: LC_FILESET_ENTRY                      offset=0x00011be60 addr=0xffffffff00711fe60
com.apple.driver.AFKHIDTBDevice
017: LC_FILESET_ENTRY                      offset=0x00011e140 addr=0xffffffff007122140
com.apple.AGXFirmwareKextG17PRTBuddy
018: LC_FILESET_ENTRY                      offset=0x00011f270 addr=0xffffffff007123270
com.apple.AGXFirmwareKextRTBuddy64
019: LC_FILESET_ENTRY                      offset=0x000123270 addr=0xffffffff007127270 com.apple.AGXG17P
```

Some observations follow:

- We clearly see arm64e architecture.
- There are 319 commands, which define layout, map kexts, setup threads and carry pointer-signing policy.
- The line "000: LC_UUID 16FF5BB5-E04D-6DD5-50F2-C6623CF19A56" provides the UUID from the fileset and it is not this one that is reported by a panic.
- `__TEXT`: it holds Mach-O header and the 319 loads commands (and not the code)..
- `com.apple.com.__TEXT_EXEC.__exc`: it is the exception vector table (VBAR_EL1).
- `com.apple.com.__TEXT_EXEC.__text`: it is the real kernel code.
- `com.apple.com.__TEXT_EXEC.__hib_text`: it belongs to the hibernation portion of the code.
- `__PRELINK_TEXT`: it is a read-only segment (legacy) and does not hold code.
- `__PRELINK_TEXT.__text`: a section inside the `__PRELINK_TEXT` (legacy) that spans its entire segment, and its first bytes are Mach-O header.
- `008: __TEXT_EXEC`: it is a region covering the executable code of all 306 entries, kernel and kexts.

- `__TEXT_BOOT_EXEC`: it contains early boot code.
- `022: LC_UUID`: it contains the UUID reported during a panic.
- `LC_DYLD_CHAINED_FIXUPS`: it tell us that the pointer in this file are still in chained form rather than finished addresses.
- `__DATA`: It contains writable global and zero-initiated data.
- `__DATA_CONST`: it contains data that must be writable during boot but make read-only afterwards.
- `__DATA_SPTM`: is holds data that belongs to the SPTM (page table monitor).
- `__LINKEDIT`: it holds linker metadata, which consists of symbol, string tables, and even chained-fixup stream.
- `LC_FILESET_ENTRY com.apple.kernel`: it is the XNU itself as an embedded Mach-O.
- `com.apple.kec.Libm`: it is the kernel resident math library.
- `com.apple.kec.pthread`: it represents the kernel resident pthread support.
- `com.apple.driver.AFKHIDTBDevice`, `com.apple.AGXFirmwareKextG17PRTBuddy`, `com.apple.AGXFirmwareKextRTBuddy64` and `com.apple.AGXG17P`: kernel drivers.

If you already have a vPhone configured, you can find the kernel cache in following directory, but in an IMG4 format (compressed and not ready for use):

```
# ls -lh /private/preboot/*/System/Library/Caches/com.apple.kernelcaches/
total 42M
-rw-r--r-- 1 root wheel 42M Jul 27 23:33 kernelcache

# head -c 4 /private/preboot/*/System/Library/Caches/com.apple.kernelcaches/kernelcache | od -An -tx1
30 84 02 95
```

To copy the kernelcache from vPhone to the local system (macOS) and extract it, run the following commands:

```
$ sshpass -p alpine ssh -p 2222 root@127.0.0.1 "export
PATH=/var/jb/usr/bin:/var/jb/bin:/usr/bin:/bin; cat
/private/preboot/*/System/Library/Caches/com.apple.kernelcaches kernelcache" > kc_dev.img4

$ ipsw img4 info kc_dev.img4 | grep -E "^ Type:|^ Data:"
Type:          krnl
Data:          43 MB (43368448 bytes)

$ ipsw img4 info kc_dev.img4 | grep -E "^ Type:|^ Data:"
Type:          krnl

$ ipsw img4 extract --im4p kc_dev.img4
• Extracting IMG4 Component component=im4p path=kc_dev.im4p size=43 MB

$ ipsw img4 im4p extract kc_dev.im4p
• Extracting Payload          path=kc_dev.payload

$ head -c 4 kc_dev.payload | od -An -tx1
cf fa ed fe (Mach-O)

$ ipsw macho info --no-color kc_dev.payload | head -5
Magic          = 64-bit MachO
Type           = FILESET
CPU            = AARCH64, ARM64e caps: KER00
Commands      = 159 (Size: 11472)
Flags          = None
```

Now we have kernelcache from a stock image and also from vPhone, we can resume our discussion about slide and, as mentioned previously, the slide is calculated by the following:

$\text{slide} = \text{runtime_address} - \text{static_address}$

While the equation is trivial, it hides a small detail: both `runtime_address` and `static_address` should refer to the same context and nature of a given object.

To understand the mechanism better, it is recommended to show a practical case with a real panic, which I can reveal too many details right now:

```
$ xcrun devicectl list devices | grep ABi16
ABi16      2FC3AF78-2618-5DA3-89A6-35EE8BECA75B  connected  iPhone 16 (iPhone17,3)  physical

$ idevice_id -l
00008140-001868412EEB001C

$ mkdir -p /tmp/crash16
$ idevicecrashreport -u 00008140-001868412EEB001C -k /tmp/crash16
$ ls -lt /tmp/crash16 | grep panic | head -1
-rw-r--r--  1 researcher  wheel  2050953 Sep  1 15:02 panic-full-2026-09-01-145656.0002.ips

$ cd crash16
$ python3 -c "import
json,sys;raw=open(sys.argv[1]).read();h,b=raw.split(chr(10),1);print(json.loads(b)['panicString'])"
panic-full-2026-09-01-145656.0002.ips | grep -E "slide:|base:"

KernelCache slide: 0x0000000039c98000
KernelCache base:  0xffffffff040c9c000
Kernel slide:      0x0000000039ca0000
Kernel text base:  0xffffffff040ca4000
Kernel text exec slide: 0x000000003d6f4000
Kernel text exec base: 0xffffffff0446f8000
```

Note that when you run `idevicecrashreport` command, which we have already explained previously, you must specify the `-k` option to avoid removing the panic file from device and that, by the way, are writing during the next boot and not at the moment of the panic. The structure of a panic file, whose extension is `.ips`, are two consecutive lines, which the first one is a small JSON object containing the summary (`bug_type`, `timestamp`, `os_version` and `incident_id`) and the second line contains a large JSON object that represents the body and includes the `panicString`, `binaryImages`, `kernel`, `build`, `codeSigningMonitor` and `isLocked` fields.

Remember important information from the kernelcache output:

```
$ ipsw macho info --no-color --loads kernelcache.release.iphone17 | grep "__TEXT"

004: LC_SEGMENT_64 sz=0x00008000 off=0x00000000-0x00008000 addr=0xffffffff007004000-
0xffffffff00700c000 r--/r--  __TEXT
008: LC_SEGMENT_64 sz=0x02f54000 off=0x013fc000-0x04350000 addr=0xffffffff008400000-
0xffffffff00b354000 r-x/r-x  __TEXT_EXEC
009: LC_SEGMENT_64 sz=0x00008000 off=0x04350000-0x04358000 addr=0xffffffff00b354000-
0xffffffff00b35c000 r-x/r-x  __TEXT_BOOT_EXEC

$ ipsw macho info --no-color kernelcache.release.iphone17 --loads | grep
"LC_FILESET_ENTRY.*com.apple.kernel$"
013: LC_FILESET_ENTRY          offset=0x00008000 addr=0xffffffff00700c000 com.apple.kernel
```

There are several slides and base addresses above, and it is time to do some math:

- KernelCache base - KernelCache slide = `0xFFFFFFFF040C9C000 - 0x39C98000 = 0xFFFFFFFF007004000`
- Kernel text base - Kernel slide = `0xFFFFFFFF040CA4000 - 0x39CA0000 = 0xFFFFFFFF007004000`
- Kernel text exec base - Kernel text exec slide = `0xFFFFFFFF007004000`

- 0xFFFFFFFF040CA4000 - 0xFFFFFFFF007004000 = 0x39CA0000 (printed as "Kernel slide")
- 0xFFFFFFFF0446F8000 - 0xFFFFFFFF007004000 = 0x3D6F4000 (printed as "Kernel text exec slide").

The real KASLR slide is the KernelCache slide. Nonetheless, when you have an original leak from the kernel, you must consider the static address of the "object" you leaked, so the subtraction between the runtime address and static address will result in the KernelCache slide.

Extending the topic a little more before returning to our explanation line, let's to apply the same concepts to addresses coming from a panic file and trying to convert some address names. To start with the example, I will be using a dump from an old vulnerability (I cannot reveal current ones), so it is required to download the respective kernelcache and symbolicate it. However, at this point of the article, kernelcache symbolication wasn't explained yet, so you might not have the Json file that is attributed to the MAP variable. Therefore, you can do it by running the command below:

```
$ ipsw download ipsw --device iPhone17,3 --version 26.5 --kernel

• Parsing remote IPSW      build=23F77 device=iPhone17,3 signed=false version=26.5
• Extracting remote kernelcache
  • Created 23F77__iPhone17,3/kernelcache.release.iphone17
  • Created 23F77__iPhone17,3/kernelcache.research.iphone17

$ ipsw kernel symbolicate --signatures ~/github/symbolicator/kernel/25.5 --json
kernelcache.release.iphone17

• Symbolication STATS      matched=4949 missed=1382 percent=78.1709% total=6331
• Writing symbols as JSON to kernelcache.release.iphone17.symbols.json
```

The panic file has the following initial content:

```
$ head -1 panic-full_2026-06-18.ips | jq

{
  "bug_type": "210",
  "timestamp": "2026-06-18 21:35:22.00 -0300",
  "os_version": "iPhone OS 26.5 (23F77)",
  "roots_installed": 0,
  "incident_id": "939DC3D8-96C0-4A06-9DD2-45AF718955B7"
}
```

Or as alternative:

```
$ head -1 panic-full_2026-06-18.ips | python3 -c "import json, sys; d =
json.loads(sys.stdin.read()); print('\n'.join(' %22s %s' % (k, d[k]) for k in
('bug_type', 'timestamp', 'os_version', 'roots_installed', 'incident_id')))"

bug_type      210
timestamp     2026-06-18 21:35:22.00 -0300
os_version    iPhone OS 26.5 (23F77)
roots_installed 0
incident_id   939DC3D8-96C0-4A06-9DD2-45AF718955B7
```

To obtain further details:

```
$ tail -n +2 panic-full_2026-06-18.ips | python3 -c "import json, sys;
print(json.loads(sys.stdin.read())['panicString'])" | sed -n '1,/^\^secure boot?:/p' | grep -v '^iBoot
Stage 2 version: *$' | sed '1s|): .*|): [REASON WITHHELD - see the note below] @<file>.c:<line>|'

panic(cpu 1 caller 0xffffffff041c141bc): [REASON WITHHELD - see the note below] @<file>.c:<line>
```

```
x0: 0xffffffffe177303fe2 x1: 0xffffffffe17820a88d x2: 0x000000000000001d5 x3: 0xffffffffe177304000
x4: 0x0000000000000000 x5: 0x000000000000001e x6: 0xffffffffe1776ce65f x7: 0xffffffffe652deb250
x8: 0x6161616161616161 x9: 0x6161616161616161 x10: 0x6161616161616161 x11: 0x6161616161616161
x12: 0x616161616161612f x13: 0x6161616161616161 x14: 0x6161616161616161 x15: 0x6161616161616161
x16: 0xffffffffe652debb20 x17: 0x3cfb296177303fe2 x18: 0x0000000000000000 x19: 0xffffffffe652debbb8
x20: 0x0000000000000000 x21: 0xffffffffe08e9f3a50 x22: 0xffffffffe1754daf00 x23: 0x0000000000000000
x24: 0x000000000000003e2 x25: 0xffffffffe177303c00 x26: 0xffffffffe17820a800 x27: 0x0000000000100040
x28: 0x00000000000000400 fp: 0xffffffffe652deba50 lr: 0xc9cf26f0415c15bc sp: 0xffffffffe652deba50
pc: 0xfffffffff041390b40 cpsr: 0x20400208 esr: 0x0000000096000047 far: 0xffffffffe177304000
```

Debugger message: panic

Memory ID: 0x1

OS release type: User

OS version: 23F77

Kernel version: Darwin Kernel Version 25.5.0: Thu Apr 23 20:39:55 PDT 2026; root:xnu-12377.122.4~1/RELEASE_ARM64_T8140

Exclaves boot status: BOOTED_EXCLAVEKIT

Fileset Kernelcache UUID: 4CD317CA13A03811E6670E003C69BA66

Kernel UUID: 2507CC02-385E-3B27-B9DD-44E6F9EE3045

Boot session UUID: 939DC3D8-96C0-4A06-9DD2-45AF718955B7

iBoot version: mBoot-18000.120.36

secure boot?: YES

An iOS panic log holds different UUID as fileset kernelcache UUID (associated with kernel image and its drivers), kernel UUID (associated to build and binary itself) and boot session UUID (generated randomly and created for this specific boot). Once again, the **FAR (Fault Address)** and **ESR (EXCEPTION SYNDROME)**, which holds the address that the processor tried to touch and why the fault has occurred, respectively, have come up again. In general way, ESR comes as a hardware output, and when any memory bug occurs (bad pointers, unmapped page, page violation, permission violation, aborts, etc) the CPU writes into ESR_EL1 to provide information on the class exception and also writes into FAR_EL1 to indicate what is the address where the problem happened.

Therefore, the esr and far values only are included in the panic file because the CPU raised an exception and wrote into them, and readers will see both registers in memory bugs. On the other hand, when an assertion occurs, which is a kind of indication of a defense is working, neither esr nor far is included into the panic file, and it proves that the kernel has handled the issue (not a bug).

Now we can retrieve a frame list from the panic files by executing the following commands:

```
$ P=panic-full_2026-06-18.ips
$ python3 -c 'import json,sys;
print(json.loads(open(sys.argv[1]).read()).split("\n",1)[1])["panicString"]' "$P" | sed -n
'/Panicked thread/,/Kernel Extensions/p'
```

```
Panicked thread: 0xffffffffdec2983828, backtrace: 0xffffffffe652deaca0, tid: 1867133
lr: 0xfffffffff0413d6fe4 fp: 0xffffffffe652dead20
lr: 0xfffffffff041572a60 fp: 0xffffffffe652dead90
lr: 0xfffffffff041570dbc fp: 0xffffffffe652deae60
lr: 0xfffffffff041394dcc fp: 0xffffffffe652deae70
lr: 0xfffffffff0413d710c fp: 0xffffffffe652deb210
lr: 0xfffffffff0413d69bc fp: 0xffffffffe652deb3d0
lr: 0xfffffffff041c068b0 fp: 0xffffffffe652deb3f0
lr: 0xfffffffff041c141bc fp: 0xffffffffe652deb570
lr: 0xfffffffff041572928 fp: 0xffffffffe652deb610
lr: 0xfffffffff0415710d8 fp: 0xffffffffe652deb6e0
lr: 0xfffffffff041394dcc fp: 0xffffffffe652deb6f0
lr: 0xfffffffff041390b40 fp: 0xffffffffe652deba50
lr: 0xfffffffff0415c15bc fp: 0xffffffffe652debb90
lr: 0xfffffffff0415a4474 fp: 0xffffffffe652debda0
lr: 0xfffffffff0415a42c8 fp: 0xffffffffe652debde0
```

```
lr: 0xffffffff041a4c8c4 fp: 0xffffffffe652debe40
lr: 0xffffffff041570e98 fp: 0xffffffffe652debf10
lr: 0xffffffff041394dcc fp: 0xffffffffe652debf20
lr: 0x0000000023b228698 fp: 0x0000000000000000
```

Based on the output above, lr is the link register, where the return address is saved by that frame, and that points to the code, and fp is the frame pointer that represents the stack, which is used to "walk" through stacks and when the value becomes zero, so the top of stacks has reached. The following command isolates the lr addresses:

```
$ python3 -c 'import json,sys;
print(json.loads(open(sys.argv[1]).read().split("\n",1)[1])["panicString"])' "$P" | grep -o 'lr:
0x[0-9a-f]*' | cut -d' ' -f2 > lrs.txt
```

```
$ cat lrs.txt | head -5
```

```
0xffffffff0413d6fe4
0xffffffff041572a60
0xffffffff041570dbc
0xffffffff041394dcc
0xffffffff0413d710c
```

```
$ wc -l < lrs.txt
19
```

The next task is to resolve this backtrace list from addresses to name, and it will be done by the following [resolve_bt.py](#) script, which is shown below:

```
$ cat resolve_bt.py
```

```
#!/usr/bin/env python3
```

```
# resolve_bt.py -- resolve a panic backtrace against a fileset kernelcache.
# usage: python3 resolve_bt.py [-k KC] [-m MAP] [-u UUID] [--far N] SLIDE < lrs.txt
```

```
import argparse, bisect, json, struct, sys
```

```
DEFAULT_KC =
"/Users/researcher/research/work/article/23F77__iPhone17,3/kernelcache.release.iphone17"
DEFAULT_MAP = DEFAULT_KC + ".symbols.json"
```

```
ap = argparse.ArgumentParser(add_help=True,
    description="resolve a panic backtrace against a fileset kernelcache")
ap.add_argument("slide", help="KernelCache slide (hex, e.g. 0x36d4c000)")
ap.add_argument("-k", "--kc", default=DEFAULT_KC, help="path to kernelcache")
ap.add_argument("-m", "--map", default=DEFAULT_MAP, help="path to symbols.json")
ap.add_argument("-u", "--uuid", help="expected LC_UUID (panic Fileset Kernelcache UUID)")
ap.add_argument("--far", type=lambda s: int(s, 0), default=0x800,
    help="offset threshold to mark [far] (default 0x800)")
ap.add_argument("-s", "--simple", action="store_true",
    help="suppress raw= and after= annotations (keep [far])")
ap.add_argument("--kernel-prefix", type=lambda s: int(s, 0), default=0xFFFFFFFF00000000,
    help="canonical kernel address prefix for PAC stripping (default
0xFFFFFFFF00000000)")
args = ap.parse_args()
```

```
RAW = open(args.kc, "rb").read()
SLIDE = int(args.slide, 16)
```

```
def cmds(base):
    off = base + 32
    for _ in range(struct.unpack_from("<I", RAW, base + 16)[0]):
        cmd, cs = struct.unpack_from("<II", RAW, off)
        yield cmd, off
```

```
off += cs

# LC_UUID: check + always report to stderr
FILE_UUID = next((RAW[o + 8:o + 24].hex().upper() for c, o in cmds(0) if c == 0x1b), None)
sys.stderr.write("kernelcache: %s\n" % args.kc)
sys.stderr.write("  LC_UUID: %s\n" % FILE_UUID)
if args.uuid:
    want = args.uuid.replace("-", "").upper()
    if FILE_UUID != want:
        sys.exit("UUID MISMATCH: kernelcache is %s, panic expected %s"
                % (FILE_UUID, want))
    sys.stderr.write("  UUID matches panic: OK\n")

def segs(base):
    out = []
    for cmd, off in cmds(base):
        if cmd == 0x19:
            nm = RAW[off + 8:off + 24].rstrip(b"\0").decode()
            va, vs = struct.unpack_from("<2Q", RAW, off + 24)
            out.append((nm, va, vs))
    return out

def funcs(base):
    # LC_FUNCTION_STARTS: ULEB deltas from __TEXT
    cur = next(va for nm, va, _ in segs(base) if nm == "__TEXT")
    out = []
    for cmd, off in cmds(base):
        if cmd != 0x26:
            continue
        do, ds = struct.unpack_from("<II", RAW, off + 8)
        blob, i = RAW[do:do + ds], 0
        while i < len(blob):
            val = shift = 0
            while True:
                b = blob[i]; i += 1
                val |= (b & 0x7F) << shift
                shift += 7
            if not (b & 0x80):
                break
            if val == 0:
                break
            cur += val
            out.append(cur)
    return sorted(out)

ents = []
for cmd, off in cmds(0):
    if cmd != 0x80000035:
        continue
    fo = struct.unpack_from("<Q", RAW, off + 16)[0]
    noff = struct.unpack_from("<I", RAW, off + 24)[0]
    name = RAW[off + noff:RAW.index(b"\0", off + noff)].decode()
    fs = funcs(fo)
    for nm, va, vs in segs(fo):
        if nm == "__TEXT_EXEC" and vs:
            ents.append((va, va + vs, name, fs))
ents.sort()
STARTS = [e[0] for e in ents]

BYADDR = {int(k): v for k, v in json.load(open(args.map)).items()}
NAMED = sorted(BYADDR.keys())

def pac_strip(addr):
    top = (addr >> 48) & 0xFFFF
    if top == 0xFFFF or top == 0x0000:
        return addr
    return (addr & 0x0000000FFFFFFFFF) | args.kernel_prefix
```

```
n_named = n_unnamed = n_no_entry = 0
i = -1
for i, line in enumerate(sys.stdin):
    line = line.strip()
    if not line:
        continue
    raw_lr = pac_strip(int(line, 16))
    adj = raw_lr - SLIDE - 4
    k = bisect.bisect_right(STARTS, adj) - 1
    lo, hi, ent, fs = ents[k] if k >= 0 else (0, 0, "-", [])

    tail_parts = []
    if not args.simple:
        tail_parts.append("raw=0x%X" % raw_lr)

    if not lo <= adj < hi:
        n_no_entry += 1
        tail = " ".join(tail_parts)
        print(("%-3d 0x%016X %-8s %-43s %s %s"
              % (i, adj, "-", "-", "<no-entry>", tail)).rstrip())
        continue

    j = bisect.bisect_right(fs, adj) - 1
    fn = fs[j] if j >= 0 else None
    offset = adj - fn if fn is not None else None
    nm = BYADDR.get(fn) if fn is not None else None

    if nm:
        n_named += 1
        display = nm
    else:
        n_unnamed += 1
        display = "<unnamed:0x%X>" % fn if fn is not None else "<no-func>"
        # nearest named function at or below fn -- a hint for readers
        if not args.simple and fn is not None:
            ni = bisect.bisect_right(NAMED, fn) - 1
            if ni >= 0:
                nprev = NAMED[ni]
                tail_parts.append("after=%s+0x%X" % (BYADDR[nprev], fn - nprev))

    if offset is not None and offset > args.far:
        tail_parts.append("[far]")

    tail = " ".join(tail_parts)
    print(("%-3d 0x%016X %-8s %-43s %s %s"
          % (i, adj, hex(offset) if offset is not None else "-",
            ent, display, tail)).rstrip())

sys.stderr.write("resolved %d frame%s: %d named, %d unnamed, %d no-entry\n"
                  % (i + 1, "" if i + 1 == 1 else "s",
                     n_named, n_unnamed, n_no_entry))
```

There are a few observations about this script:

- There is no need of stripping PAC (bit 63 to 47) because the extracted address are canonical kernel addresses, and there is no signature left in them to remove. This is because kernel strips them before logging into the text panic file and not from raw kernel core dump. However, if we try to feed an unstripped address (our case) into a debugger like lldb, the symbolication will fail because it looks for a literal address that does not exist.
- BR and BRL write the address of the instruction after the branch into X30 (link register). Thus, return_address = call_site + 4, which forces the script to fix it by subtracting 4. The only exemption

is frame 0 when it is a PC, which it is found in cases such as debugger backtrace or a userland crash report. As our example is a kernel scenario, the exemption does not apply to frame 0.

- To retrieve the slide from this panic file, execute:

```
$ python3 -c "import
json,sys;raw=open(sys.argv[1]).read();h,b=raw.split(chr(10),1);print(json.loads(b)['panicStr
ing'])" panic-full_2026-06-18.ips | grep -E "slide:|base:"

KernelCache slide: 0x0000000036d4c000
KernelCache base: 0xffffffff03dd50000
Kernel slide:      0x0000000036d54000
Kernel text base:  0xffffffff03dd58000
Kernel text exec slide: 0x000000003a38c000
Kernel text exec base: 0xffffffff041390000
```

About the second item above, it brings relevant details. To disassemble the address of frame 0 (0xffffffff0413d6fe4), we cannot do it directly. We need to subtract 4 (as explained) and also subtract the slide (0x36d4c000) because the kernelcache on disk is "unslid", as shown below:

- frame 0: 0xffffffff0413d6fe4
- subtract slide: 0xffffffff0413d6fe4 - 0x36d4c000 = 0xffffffff00a68afe4
- subtract 4: 0xffffffff00a68afe4 - 4 = 0xffffffff00a68afe0

Disassemble it using ipsw tool for understanding the sequence of code that is involved, as shown below:

```
$ ipsw macho disass --no-color -q -t com.apple.kernel -a 0xffffffff00a68afe0 -c 10
kernelcache.release.iphone17
```

```
0xffffffff00a68afe0: 3e 88 06 94    bl    0xffffffff00a82d0d8
0xffffffff00a68afe4: f3 03 14 aa    mov   x19, x20
0xffffffff00a68afe8: 88 22 73 39    ldrb  w8, [x20, #0xcc8]
0xffffffff00a68afec: e8 00 00 36    tbz   w8, #0, 0xffffffff00a68b008
0xffffffff00a68aff0: 40 01 80 52    mov   w0, #0xa
0xffffffff00a68aff4: 01 00 80 52    mov   w1, #0
0xffffffff00a68aff8: e8 9f 1d 94    bl    0xffffffff00adf2f98
0xffffffff00a68affc: 20 01 80 52    mov   w0, #0x9
0xffffffff00a68b000: 01 00 80 52    mov   w1, #0
0xffffffff00a68b004: e5 9f 1d 94    bl    0xffffffff00adf2f98
```

Using the kernelcache slide (0x36d4c000) and applying the script to the isolated list with LR values, we can run the following command:

```
$ python3 resolve_bt.py -s 0x36d4c000 < lrs.txt
```

```
kernelcache: /Users/researcher/research/work/article/23F77__iPhone17,3/kernelcache.release.iphone17
LC_UUID: 4CD317CA13A03811E6670E003C69BA66
```

0	0xFFFFFFFF00A68AFE0	0x40c	com.apple.kernel	handle_debugger_trap
1	0xFFFFFFFF00A826A5C	0x100	com.apple.kernel	handle_uncategorized
2	0xFFFFFFFF00A824DB8	0x1dc	com.apple.kernel	panic_trap_to_debugger
3	0xFFFFFFFF00A648DC8	0x44	com.apple.kernel	panic
4	0xFFFFFFFF00A68B108	0x80	com.apple.kernel	<unnamed:0xFFFFFFFF00A68B088>
5	0xFFFFFFFF00A68A9B8	0x430	com.apple.kernel	panic_trap_to_debugger
6	0xFFFFFFFF00AEBA8AC	0x38	com.apple.kernel	<unnamed:0xFFFFFFFF00AEBA874>
7	0xFFFFFFFF00AEC81B8	0x21c	com.apple.kernel	panic_with_thread_kernel_state
8	0xFFFFFFFF00A826924	0x528	com.apple.kernel	__stack_chk_fail
9	0xFFFFFFFF00A8250D4	0x4f8	com.apple.kernel	panic_trap_to_debugger
10	0xFFFFFFFF00A648DC8	0x44	com.apple.kernel	panic
11	0xFFFFFFFF00A644B3C	0x4c	com.apple.kernel	<unnamed:0xFFFFFFFF00A644AF0>
12	0xFFFFFFFF00A8755B8	0xcc0	com.apple.kernel	namei [far]


```

13  0xFFFFFFFF00A858470 0x198      com.apple.kernel      <unnamed:0xFFFFFFFF00A8582D8>
14  0xFFFFFFFF00A8582C4 0x70      com.apple.kernel      getatrrlist
15  0xFFFFFFFF00AD008C0 0x2bc     com.apple.kernel      <unnamed:0xFFFFFFFF00AD00604>
16  0xFFFFFFFF00A824E94 0x2b8     com.apple.kernel      panic_trap_to_debugger
17  0xFFFFFFFF00A648DC8 0x44      com.apple.kernel      panic
18  0x000000002044DC694 -          -                    <no-entry>
resolved 19 frames: 13 named, 5 unnamed, 1 no-entry

```

To completeness, the help follows:

```
$ python3 resolve_bt.py --help
```

```
usage: resolve_bt.py [-h] [-k KC] [-m MAP] [-u UUID] [--far FAR] [-s] [--demangle | --reduced] [--kernel-prefix KERNEL_PREFIX] slide
```

resolve a panic backtrace against a fileset kernelcache

positional arguments:

slide KernelCache slide (hex, e.g. 0x36d4c000)

options:

```

-h, --help                show this help message and exit
-k, --kc KC               path to kernelcache
-m, --map MAP             path to symbols.json
-u, --uuid UUID           expected LC_UUID (panic Fileset Kernelcache UUID)
--far FAR                 offset threshold to mark [far] (default 0x800)
-s, --simple               suppress raw= and after= annotations (keep [far])
--demangle                format as a table with demangled C++ names
--reduced                 like --demangle but drop the entry column (implies --demangle)
--kernel-prefix KERNEL_PREFIX
                           canonical kernel address prefix for PAC stripping (default

```

```
0xFFFFFFFF000000000)
```

Some examples of how I use these options are as follows:

default: parseable records with mangled names, streamable

```
python3 resolve_bt.py 0x36d4c000 < lrs.txt
```

demangled table with entry column

```
python3 resolve_bt.py --demangle 0x36d4c000 < lrs.txt
```

demangled compact table

```
python3 resolve_bt.py --reduced 0x36d4c000 < lrs.txt
```

any mode + suppress raw=/after= annotations (keeps [far])

```
python3 resolve_bt.py --demangle -s 0x36d4c000 < lrs.txt
```

with UUID assertion

```
python3 resolve_bt.py --demangle -u 4CD317CA13A03811E6670E003C69BA66 0x36d4c000 < lrs.txt
```

My preferred combination is to include **--reduced** option, as shown:

```
$ python3 resolve_bt.py -s --reduced 0x36d4c000 < lrs.txt
```

```
kernelcache: /Users/researcher/research/work/article/23F77__iPhone17,3/kernelcache.release.iphone17
```

```
LC_UUID: 4CD317CA13A03811E6670E003C69BA66
```

```
# static (adjusted) symbol
```

```

--
0  0xFFFFFFFF00A68AFE0 handle_debugger_trap +0x40c
1  0xFFFFFFFF00A826A5C handle_uncategorized +0x100
2  0xFFFFFFFF00A824DB8 panic_trap_to_debugger +0x1dc
3  0xFFFFFFFF00A648DC8 panic +0x44
4  0xFFFFFFFF00A68B108 <unnamed:0xFFFFFFFF00A68B088> +0x80
5  0xFFFFFFFF00A68A9B8 panic_trap_to_debugger +0x430
6  0xFFFFFFFF00AEBA8AC <unnamed:0xFFFFFFFF00AEBA874> +0x38
7  0xFFFFFFFF00AEC81B8 panic_with_thread_kernel_state +0x21c

```

```
8 0xFFFFFFFF00A826924 __stack_chk_fail +0x528
9 0xFFFFFFFF00A8250D4 panic_trap_to_debugger +0x4f8
10 0xFFFFFFFF00A648DC8 panic +0x44
11 0xFFFFFFFF00A644B3C <unnamed:0xFFFFFFFF00A644AF0> +0x4c
12 0xFFFFFFFF00A8755B8 namei +0xcc0 [far]
13 0xFFFFFFFF00A858470 <unnamed:0xFFFFFFFF00A8582D8> +0x198
14 0xFFFFFFFF00A8582C4 getattrlist +0x70
15 0xFFFFFFFF00AD008C0 <unnamed:0xFFFFFFFF00AD00604> +0x2bc
16 0xFFFFFFFF00A824E94 panic_trap_to_debugger +0x2b8
17 0xFFFFFFFF00A648DC8 panic +0x44
18 0x000000002044DC694 <no-entry> +- NO_ENTRY
resolved 19 frames: 13 named, 5 unnamed, 1 no-entry
```

You might need to examine what's around the address and not specifically from that point onward, so a dedicated script (named [fnprobe.py](#)) could be better for this purpose:

```
$ cat fnprobe.py
```

```
#!/usr/bin/env python3
```

```
"""fnprobe.py - print the containing function of an address, from the binary.
```

```
usage: fnprobe.py <kc> <0xADDR> [<0xADDR> ...] [--range LO HI] [--grep MNEMONIC]
       fnprobe.py -h | --help
```

```
Options:
```

```
--range LO HI      Restrict the printed listing to the [LO, HI] window inside
                    the discovered function.
--grep MNEMONIC     Print only instructions whose mnemonic contains MNEMONIC
                    (substring match).
-h,--help          Print this help and exit.
```

Used to hand verify every instruction a sweep flags -- reading the actual bytes in the actual binary is the standing rule.

```
"""
```

```
import bisect
import struct
import sys
```

```
def _usage(rc):
    sys.stderr.write(__doc__ or "").strip() + "\n"
    sys.exit(rc)
```

```
if "-h" in sys.argv or "--help" in sys.argv:
```

```
    _usage(0)
```

```
if len(sys.argv) < 3:
```

```
    sys.stderr.write("%s needs a kernelcache and at least one 0x-prefixed address.\n\n" %
sys.argv[0])
    _usage(2)
```

```
from capstone import CS_ARCH_ARM64, CS_MODE_ARM, Cs
```

```
KERNEL_ENTRY      = "com.apple.kernel"
LC_SEGMENT_64      = 0x19
LC_FUNCTION_STARTS = 0x26
LC_FILESET_ENTRY   = 0x35 | 0x80000000
```

```
def load(path):
```

```
    """Return (data, vm, fo, fs, starts) for the com.apple.kernel entry's
    __TEXT_EXEC segment. starts is the LC_FUNCTION_STARTS decode, filtered
    to addresses inside __TEXT_EXEC. Fails loud if the entry is absent.
    """
```

```
    d = open(path, "rb").read()
```

```
    # walk the top-level Mach-O for the com.apple.kernel fileset entry
```

```
ncmds = struct.unpack_from("<I", d, 16)[0]
p, entry = 32, None
for _ in range(ncmds):
    cmd, cs = struct.unpack_from("<2I", d, p)
    if cmd == LC_FILESET_ENTRY:
        eoff = struct.unpack_from("<I", d, p + 24)[0]
        nm = d[p + eoff:p + cs].split(b"\0")[0].decode()
        if nm == KERNEL_ENTRY:
            entry = struct.unpack_from("<Q", d, p + 16)[0]
            break
    p += cs
if entry is None:
    sys.exit("no %s fileset entry in %s" % (KERNEL_ENTRY, path))

# walk that entry's own load commands for __TEXT / __TEXT_EXEC / LC_FUNCTION_STARTS
n2 = struct.unpack_from("<I", d, entry + 16)[0]
q, text, fstarts, imgbase = entry + 32, None, None, None
for _ in range(n2):
    cmd, cs = struct.unpack_from("<2I", d, q)
    if cmd == LC_SEGMENT_64:
        name = d[q + 8:q + 24].split(b"\0")[0].decode()
        vm, vs, fo, fs = struct.unpack_from("<4Q", d, q + 24)
        if name == "__TEXT_EXEC":
            text = (vm, fo, fs)
        elif name == "__TEXT":
            imgbase = vm
    elif cmd == LC_FUNCTION_STARTS:
        fstarts = struct.unpack_from("<2I", d, q + 8)
    q += cs
if text is None or fstarts is None or imgbase is None:
    sys.exit("kernel entry is missing __TEXT / __TEXT_EXEC / LC_FUNCTION_STARTS")

# decode ULEB128 deltas from __TEXT vmaddr, filter to __TEXT_EXEC
off, sz = fstarts
addrs, cur, p = [], imgbase, off
end = off + sz
while p < end:
    val, sh = 0, 0
    while True:
        b = d[p]; p += 1
        val |= (b & 0x7F) << sh
        if not (b & 0x80):
            break
        sh += 7
    if val == 0:
        break
    cur += val
    addrs.append(cur)
vm, fo, fs = text
starts = sorted(a for a in addrs if vm <= a < vm + fs)
return d, vm, fo, fs, starts

def main():
    kc = sys.argv[1]
    data, vm, fo, fs, starts = load(kc)
    md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
    md.detail = False

    args = [a for a in sys.argv[2:] if a.startswith("0x")]
    lo = hi = None
    if "--range" in sys.argv:
        i = sys.argv.index("--range")
        lo, hi = int(sys.argv[i + 1], 16), int(sys.argv[i + 2], 16)
    grep = None
    if "--grep" in sys.argv:
        grep = sys.argv[sys.argv.index("--grep") + 1]
```

```

for a in [int(x, 16) for x in args]:
    i = bisect.bisect_right(starts, a) - 1
    s = starts[i]
    e = starts[i + 1] if i + 1 < len(starts) else vm + fs
    ins = list(md.disasm(data[fo + (s - vm):fo + (e - vm)], s))
    nwords = (e - s) // 4
    print("=" * 88)
    print("func 0x%x .. 0x%x    len 0x%x (%d words)    decoded %d insns%s"
          % (s, e, e - s, nwords, len(ins),
             "    *** TRUNCATED ***" if len(ins) != nwords else ""))
    if len(ins) != nwords:
        got = ins[-1].address + 4 if ins else s
        w = struct.unpack_from("<I", data, fo + (got - vm))[0]
        print("    decode stopped at 0x%x, word = 0x%08x" % (got, w))
    print("=" * 88)
    for x in ins:
        if lo is not None and not (lo <= x.address <= hi):
            continue
        if grep and grep not in x.mnemonic:
            continue
        mark = " <<<" if x.address == a else ""
        print("  0x%-11x %-8s %s%s" % (x.address, x.mnemonic, x.op_str, mark))

main()

```

To show the importance and value of this script, take the address of the first frame (frame 0), and run the script as shown below:

```

$ python fnprobe.py kernelcache.release.iphone17 0xffffffff00a68afe0
=====
func 0xffffffff00a68abd4 .. 0xffffffff00a68b088    len 0x4b4 (301 words)    decoded 301 insns
=====
0xffffffff00a68abd4 pacibsp
0xffffffff00a68abd8 sub      sp, sp, #0x80
0xffffffff00a68abdc stp      x28, x27, [sp, #0x20]
0xffffffff00a68abe0 stp      x26, x25, [sp, #0x30]
0xffffffff00a68abe4 stp      x24, x23, [sp, #0x40]
0xffffffff00a68abe8 stp      x22, x21, [sp, #0x50]
0xffffffff00a68abec stp      x20, x19, [sp, #0x60]
0xffffffff00a68abf0 stp      x29, x30, [sp, #0x70]
0xffffffff00a68abf4 add      x29, sp, #0x70
0xffffffff00a68abf8 adrp     x27, #0xffffffff00b1ac000
0xffffffff00a68abfc ldr      w28, [x27, #0x2c0]
....
....
0xffffffff00a68afd4 ldr      x2, [x20, #0xcc8]
0xffffffff00a68afd8 adrp     x8, #0xffffffff00b1a8000
0xffffffff00a68afdc ldr      x3, [x8, #0x90]
0xffffffff00a68afe0 bl      #0xffffffff00a82d0d8 <<<
0xffffffff00a68afe4 mov     x19, x20
0xffffffff00a68afe8 ldrb     w8, [x20, #0xcc8]
0xffffffff00a68afec tbz     w8, #0, #0xffffffff00a68b008
....
....
0xffffffff00a68b070 str      wzr, [x19, #0x10]
0xffffffff00a68b074 b        #0xffffffff00a68b048
0xffffffff00a68b078 adrp     x0, #0xffffffff007045000
0xffffffff00a68b07c add     x0, x0, #0x94a
0xffffffff00a68b080 bl      #0xffffffff00a6d7a94
0xffffffff00a68b084 bl      #0xffffffff00a68aa04

```

If you need only to consult some addresses, and not necessarily ones coming from a panic file, the [whichentry.py](#) script shown below can be very useful:

```

$ cat whichentry.py

```

```
#!/usr/bin/env python3
"""whichentry.py -- which fileset entry (XNU core or a kext) contains a given vaddr?

usage: whichentry.py <kc> [-m <symbols.json>] [--all] <vaddr> [vaddr ...]
       whichentry.py -h | --help

Options:
  -m <path>  Optional symbols.json map (decimal-string keys, address values).
              When passed, the containing function's name and offset are printed
              after the segment range. The name comes from a bisect of the entry's
              own LC_FUNCTION_STARTS, then a lookup in the map; a function
              boundary that has no map entry prints as <unnamed:0xADDR>.
  --all      Attribute DATA addresses too (default is __TEXT segments only).
  -h,--help  Print this help and exit.

"""
import bisect
import json
import struct
import sys

def _usage(rc):
    sys.stderr.write(__doc__ or "").strip() + "\n"
    sys.exit(rc)

if "-h" in sys.argv or "--help" in sys.argv:
    _usage(0)
if len(sys.argv) < 3:
    sys.stderr.write("%s needs a kernelcache and at least one address.\n\n" % sys.argv[0])
    _usage(2)

LC_SEGMENT_64      = 0x19
LC_FUNCTION_STARTS = 0x26
LC_FILESET_ENTRY   = 0x35 | 0x80000000

def _cmds(data, base):
    ncmds, = struct.unpack_from("<I", data, base + 16)
    off = base + 32
    for _ in range(ncmds):
        cmd, sz = struct.unpack_from("<II", data, off)
        yield cmd, off
        off += sz

def entries(data):
    out = []
    for cmd, off in _cmds(data, 0):
        if cmd == LC_FILESET_ENTRY:
            vmaddr, fileoff, entry_id_off = struct.unpack_from("<QQI", data, off + 8)
            end = data.index(b"\0", off + entry_id_off)
            name = data[off + entry_id_off:end].decode("ascii", "replace")
            out.append((name, vmaddr, fileoff))
    return out

def text_ranges(data, fileoff, want_all):
    """Parse one entry's own load commands for its executable segments."""
    rs = []
    for cmd, off in _cmds(data, fileoff):
        if cmd == LC_SEGMENT_64:
            seg = data[off + 8:off + 24].rstrip(b"\0").decode("ascii", "replace")
            vmaddr, vmsize = struct.unpack_from("<QQ", data, off + 24)
            if vmsize and (seg.startswith("__TEXT") or want_all):
                rs.append((seg, vmaddr, vmaddr + vmsize))
    return rs

def function_starts(data, fileoff):
```

```
"""Decode one entry's LC_FUNCTION_STARTS as ULEB128 deltas from its __TEXT vmaddr."""
text_vm = None
for cmd, off in _cmds(data, fileoff):
    if cmd == LC_SEGMENT_64:
        seg = data[off + 8:off + 24].rstrip(b"\0").decode("ascii", "replace")
        if seg == "__TEXT":
            text_vm = struct.unpack_from("<Q", data, off + 24)[0]
            break
if text_vm is None:
    return []
starts = []
cur = text_vm
for cmd, off in _cmds(data, fileoff):
    if cmd != LC_FUNCTION_STARTS:
        continue
    do, ds = struct.unpack_from("<II", data, off + 8)
    blob = data[do:do + ds]
    i = 0
    while i < len(blob):
        val = shift = 0
        while True:
            b = blob[i]; i += 1
            val |= (b & 0x7F) << shift
            shift += 7
            if not (b & 0x80):
                break
        if val == 0:
            break
        cur += val
        starts.append(cur)
return sorted(starts)

kc = sys.argv[1]
argv = list(sys.argv[2:])
want_all = "--all" in argv
argv = [a for a in argv if a != "--all"]

by_addr = None
if "-m" in argv:
    i = argv.index("-m")
    map_path = argv[i + 1]
    del argv[i:i + 2]
    by_addr = {int(k): v for k, v in json.load(open(map_path)).items()}}

data = open(kc, "rb").read()
ents = entries(data)
print("fileset entries: %d" % len(ents))

_fn_cache = {}
def _starts_for(fileoff):
    if fileoff not in _fn_cache:
        _fn_cache[fileoff] = function_starts(data, fileoff)
    return _fn_cache[fileoff]

rows = []
for a in argv:
    va = int(a, 0)
    hit = None
    for name, vmaddr, fileoff in ents:
        for seg, lo, hi in text_ranges(data, fileoff, want_all):
            if lo <= va < hi:
                hit = (name, seg, lo, hi, fileoff)
                break
        if hit:
            break
    if not hit:
        rows.append((va, None))
```

```
        continue
    name, seg, lo, hi, fileoff = hit
    tail = ""
    if by_addr is not None:
        starts = _starts_for(fileoff)
        j = bisect.bisect_right(starts, va) - 1
        if j >= 0:
            fn = starts[j]
            nm = by_addr.get(fn) or ("<unnamed:0x%x>" % fn)
            tail = " %s +0x%x" % (nm, va - fn)
    rows.append((va, (name, seg, lo, hi, tail)))

w = max((len(r[1][0]) for r in rows if r[1]), default=0)
for va, r in rows:
    if r is None:
        print("0x%x -> no fileset entry claims it" % va)
    else:
        name, seg, lo, hi, tail = r
        print("0x%x -> %-*s [%s 0x%x..0x%x]%s" % (va, w, name, seg, lo, hi, tail))
```

To execute such a script against a few addresses, run the following command:

```
$ python whichentry.py "C:\research_ios\__temp\23F77__iPhone17,3\kernelcache.release.iphone17" -m
kernelcache.release.iphone17.symbols.json 0xFFFFFFFF00A68AFE0 0xFFFFFFFF00A648DC8 0xFFFFFFFF00A824E94
```

```
fileset entries: 302
0xffffffff00a68afe0 -> com.apple.kernel [__TEXT_EXEC 0xffffffff00a644000..0xffffffff00aed0000]
handle_debugger_trap +0x40c
0xffffffff00a648dc8 -> com.apple.kernel [__TEXT_EXEC 0xffffffff00a644000..0xffffffff00aed0000]
panic +0x44
0xffffffff00a824e94 -> com.apple.kernel [__TEXT_EXEC 0xffffffff00a644000..0xffffffff00aed0000]
panic_trap_to_debugger +0x2b8
```

We successfully recovered symbols and function names associated with each return address, which helps us to understand what was happening at the moment of the crash. Additionally, and as readers have already noticed, even though the concept of slide is relatively simple, it is involved in real scenarios and analysis that could be more difficult and demand programming to be really understood.

Just in case readers would like to investigate what has really happened and examine the panic information to understand what has happened, the following few steps could help you too.

The first step is to retrieve key registers and also the kernelcache slide:

```
$ tail -n +2 panic-full_2026-06-18.ips | python3 -c "import json, re, sys; \
ps = json.loads(sys.stdin.read())['panicString']; \
P = [('pc', r'pc:\s*(0x[0-9a-fA-F]+)'), \
('lr', r'lr:\s*(0x[0-9a-fA-F]+)'), \
('esr', r'esr:\s*(0x[0-9a-fA-F]+)'), \
('far', r'far:\s*(0x[0-9a-fA-F]+)'), \
('KernelCache slide', r'KernelCache slide:\s*(0x[0-9a-fA-F]+)')]; \
[print('%-18s %s' % (n, re.search(p, ps).group(1))) for n, p in P]"

pc                0xffffffff041390b40
lr                0xc9cf26f0415c15bc
esr               0x0000000096000047
far              0xffffffffe177304000
KernelCache slide 0x0000000036d4c000
```

The esr is available, so it seems to be a memory fault/bug and not only an assertion.

The next step is to decode esr register to understand the bug and use the far register to learn where the bug has occurred. Once again, a Python script can help us to get the appropriate values:


```
$ python3 -c "esr = 0x96000047; \
    print('EC   = 0x%02x' % ((esr >> 26) & 0x3f)); \
    print('WnR  = %d'      % ((esr >>  6) & 1)); \
    print('DFSC = 0x%02x' % ( esr          & 0x3f))"
```

```
EC   = 0x25
WnR  = 1
DFSC = 0x07
```

The value of EC=0x25 means a data abort taken at the same exception level (kernel) while WnR=1 refers to a write operation. The value of DFSC=0x07 represents the translation fault at page-table level 3, which means that the target page is unmapped. Finally, far == 0xffffffe177304000 suggests the target because it is in the zone-allocator range (0xfffffe...) and also page-aligned (last twelve bits are zero). Thus, we have the sum of all facts, which are the unmapped and aligned page could be evidence that some writing operation passed the end a valid buffer into the adjacent unmapped page. Before proceeding with this short analysis, readers could want to check supplemental references:

- **ESR_EL1, Exception Syndrome Register (EL1):**
https://support.arm.com/documentation/111180/2025-09_AS11/AArch64-Registers/ESR-EL1--Exception-Syndrome-Register--EL1-
- **aarch64-esr-decoder:** <https://github.com/google/aarch64-esr-decoder>
- **Aarch64 register decoder:** <https://esr.arm64.dev/>
- **XNU reference:** https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/arm64/proc_reg.h

We have already understood the nature of the bug/fault, but it is necessary to learn which binary is associated with it. Obviously, the LC_UUID from kernelcache must match with the Fileset Kernelcache UUID from panic report, and this task can be done as shown below:

```
$ grep -oE 'Fileset Kernelcache UUID: [0-9A-Fa-f]+' panic-full_2026-06-18.ips | head -1
Fileset Kernelcache UUID: 4CD317CA13A03811E6670E003C69BA66
```

```
$ otool -l kernelcache.release.iphone17 | grep '^    uuid'
uuid 4CD317CA-13A0-3811-E667-0E003C69BA66
```

The last step is locating the point of the bug, and the starting point is to subtract the KernelCache slide (0x36d4c000) from the pc (0xfffffff041390b40) and then to use the [whichtentry.py](#) tool to find the component and, consequently, the faulting instruction:

```
pc (0xfffffff041390b40) - KernelCache slide (0x36d4c000) = 0xfffffff00a644b40
```

```
$ python3 whichtentry.py kernelcache.release.iphone17 -m kernelcache.release.iphone17.symbols.json
0xfffffff00a644b40
```

```
fileset entries: 302
0xfffffff00a644b40 -> com.apple.kernel [__TEXT_EXEC 0xfffffff00a644000..0xfffffff00aed0000]
<unnamed:0xfffffff00a644af0> +0x50
```

```
$ python3 fnprobe.py kernelcache.release.iphone17 0xfffffff00a644b40 | grep -A8 -B8
0xfffffff00a644b40
```

```
0xfffffff00a644b20 ldp      x14, x15, [x1, #0x10]
0xfffffff00a644b24 sub      x5, x3, x0
0xfffffff00a644b28 add      x1, x1, x5
0xfffffff00a644b2c ldp      x8, x9, [x1]
0xfffffff00a644b30 ldp      x10, x11, [x1, #0x10]
0xfffffff00a644b34 add      x1, x1, #0x20
0xfffffff00a644b38 sub      x2, x2, x5
```

```
0xffffffff00a644b3c stp      x12, x13, [x0]
0xffffffff00a644b40 stp      x14, x15, [x0, #0x10] <<<
0xffffffff00a644b44 subs     x2, x2, #0x40
0xffffffff00a644b48 b.ls     #0xffffffff00a644b6c
0xffffffff00a644b4c stnp     x8, x9, [x3]
0xffffffff00a644b50 stnp     x10, x11, [x3, #0x10]
0xffffffff00a644b54 add      x3, x3, #0x20
0xffffffff00a644b58 ldnp     x8, x9, [x1]
0xffffffff00a644b5c ldnp     x10, x11, [x1, #0x10]
0xffffffff00a644b60 add      x1, x1, #0x20
```

```
$ grep -oE 'x0:[[:space:]]+0x[0-9a-f]+' panic-full_2026-06-18.ips | head -1
```

```
x0: 0xffffffe177303fe2
```

The instruction "`stp x14, x15, [x0, #0x10]`" stores two x14 and x15 (16-byte value) to "`x0 + 0x10`", side by side, so `0xffffffe177303fe2 + 0x10 = 0xffffffe177303ff2`. Thus, the last byte to be stored is `0xffffffe177303ff2 + 0xf == 0xffffffe177304001`, which definitely crosses the page boundary at `0xffffffe177304001` by 2 bytes (the end would be `0xffffffe177303fff`) and, if readers check the far value again, it shows far = `0xffffffe177304000`!

The panic report contains the telemetry from kernel that confirms our conclusion, because it carries the "**Probabilistic GZAlloc Report**", which the kernel writes when this kind of bug/fault is detected against the GZAlloc-guarded slab page, as shown below:

```
$ tail -n +2 panic-full_2026-06-18.ips | python3 -c "import json, sys;
print(json.loads(sys.stdin.read())['panicString'])" | grep -A 6 'Probabilistic GZAlloc'
```

```
Probabilistic GZAlloc Report:
Zone      : data.kalloc.1024
Address   : 0xffffffe177304000
Element   : [0xffffffe177303c00, 0xffffffe177304000) of size 1024
Submap    : DATA [0xffffffe175980000; 0xffffffe2a81fc000)
Kind      : out-of-bounds (high confidence)
Access    : 1 byte(s) past
```

Basically, the overrun buffer is a `data.kalloc.1024` element (check Zone field), which is a 1024-byte kalloc allocation, and occupies the `0xffffffe177303c00, 0xffffffe177304000` range of the memory. To confirm it, run:

```
$ python3 -c "buf_start = 0xffffffe177303c00; \
    buf_end   = 0xffffffe177304000; \
    x0        = 0xffffffe177303fe2; \
    lo = x0 + 0x10; hi = lo + 15; \
    print('buffer size      : %d bytes' % (buf_end - buf_start)); \
    print('x0 below end    : %d bytes' % (buf_end - x0)); \
    print('store range     : 0x%x .. 0x%x (16 bytes)' % (lo, hi)); \
    print('bytes inside    : %d' % max(0, buf_end - lo)); \
    print('bytes past      : %d' % max(0, hi + 1 - buf_end))"
```

```
buffer size      : 1024 bytes
x0 below end     : 30 bytes
store range      : 0xffffffe177303ff2 .. 0xffffffe177304001 (16 bytes)
bytes inside     : 14
bytes past       : 2
```

Previously, we weren't able to find a symbol associated with the address using [whicentry.py](#) script, but as we have the XNU source code, we can try to check for the harmful instruction there. However, as XNU uses immediate offset in decimal and Capstone showed in hexadecimal, I will search for the previous line to

provide readers with a perfect match. Afterwards, I will search around the instruction, and the next instruction will be the real faulting instructions, as shown below:

```
$ cat /Users/researcher/research/xnu/osfmk/arm64/bcopy.s | nl | grep -n
'stp[:space:]]\+x12,x13,\[x0\]'
141: 131          stp      x12,x13,[x0]          // initial unaligned store

$ cat /Users/researcher/research/xnu/osfmk/arm64/bcopy.s | nl | grep -n -A5 -B15
'stp[:space:]]\+x12,x13,\[x0\]'

126- 116 //          x3      aligned location in dst buffer.
127- 117 //          x12:x15 first 32 bytes of src buffer.
128- 118 //
129- 119 // We now load 32 bytes from x1, and store 32 bytes from x12:x15 to x3. The
130- 120 // store *may* overlap the first 32 bytes of the load, so in order to get
131- 121 // correct memmove semantics, the first 32 byte load must occur before the
132- 122 // store.
133- 123 //
134- 124 // After loading these 32 bytes, we advance x1, and decrement the length by
135- 125 // 64. If the remaining length of the buffer was less than 64, then we jump
136- 126 // directly to the cleanup path.
137- 127 ldp      x8, x9, [x1]
138- 128 ldp      x10,x11,[x1, #16]
139- 129 add      x1,      x1, #32
140- 130 sub      x2,      x2, x5 // update length
141- 131 stp      x12,x13,[x0]    // initial unaligned store
142- 132 stp      x14,x15,[x0, #16] // initial unaligned store
143- 133 subs     x2,      x2, #64
144- 134 b.ls     L_forwardCleanup
145-
146- 135 L_forwardCopyLoop:
```

```
$ awk 'NR<=141 && /^_[A-Za-z_]+\+:\$/' /Users/researcher/research/xnu/osfmk/arm64/bcopy.s

_bcopy:
_ovbcopy:
_memcpy:
_memmove:
```

The output above shows that the unnamed routine at "0xffffffff00a644af0" is shared by different function implementations such as bcopy, ovbcopy, memcpy and memmove. The real faulting instruction is the next one, `stp x14,x15,[x0, #16]`, and it matches with the previous result. Reading the exposed code, it is clear that `memcpy` function is causing a heap overflow and is not a null dereference ("far" would be small), nor a permission fault ("DFSC" would be permission, and not translation) nor an assertion because `esr` is present.

I have anticipated concepts and methodologies that are important for the daily work on iOS research and that will be explained in later sections of this article and mainly next articles. Anyway, at this point, it is clear that kernelcache and panic backtraces do not have symbols, which are essential for our daily work. Unavoidably, unsymbolized C++ methods will be found along the path, including vtables and PAC associated issues. In a few opportunities we will not be sure about a symbol, and in such cases, there is no shortcut and we have to examine the piece of code by disassembling it and also checking its respective context, and based on common experience, the nearest symbol will answer for an address we do not have symbol for most of times.

Readers have already noticed that symbols are vital for our work and, no doubt, the kernelcache is always the preferred source for any research and analysis, even though XNU source code is useful for providing us with a guideline for testing hypothesis and pointing where to look and examine, but certainly never as a fact itself because the shipping code does not match exactly with the public source code. Therefore, the

standard and usual procedure when starting iOS security research is to download the exact firmware, extract the kernelcache and symbolicate it. Another resource is to perform kernelcache binary diffing using multiple techniques and approaches, which provide us with true highlights about features and potential vulnerabilities in the code, and that it will be presented later in this article.

As I have mentioned previously, once we have an address (mainly from a panic file), it is necessary to learn which kext it belongs to. In general, the goal is to test an address for containment inside each entry from `_TEXT_EXEC` range, which triggers the parsing of the fileset entries to find where each one starts, and then a new parsing phase to check the load commands inside each entry to get its code range. Although there isn't an exactly fixed procedure, actions are roughly:

- walk the outer load commands for `LC_FILESET_ENTRY` and keep the file offset of each entry and its respective name.
- Parse each entry as a Mach-O in its own right, starting at the provided offset.
- Read its `__TEXT_EXEC` address and size. The answer is based on providing the correct range, as expected.
- Test for the containment rather than for the nearest because an address may either falls inside one of those ranges or it belongs to none of them, which is also a real that deserves to be printed.

Likely you will find challenges imposed by PAC, and as breaking the implemented cryptography is not viable, a few available alternatives are reusing (via swap) a signed pointer without diversity on its own context, finding code (oracle) that signs a value you influence and with the key we need, discovering a zero or constant discriminator that makes a whole class of pointer interchangeable, attacking any target (indices, lengths, reference counts and flags) that are not signed or simply don't use pointers and adopt a data-only attack.

Therefore, and only so far, we have already scratched the surface of protections that, once combined, are very efficient to prevent exploitation such as AMFI (signatures), sandbox (reachability), entitlements (authorizations), fbounds safety (compiling option), `kalloc_type` segregation (zone checking, signature checking and data/non-data heap split) and PAC (defeats injected code and reuse of chain from existing code). Of course, we have discussed nothing about entitlements, AMFI and other protections yet, but it shows a general picture on how difficult research and exploit iOS vulnerabilities usually are. Not for coincidence, all of them prevents getting a read-write primitive on kernel. Additionally, we need a KASLR leak and, worse, a way to compromise SMTP and TXM protections.

There are some well-known public chains such as SockPuppet, PhysPuppet, kdf engine and, under a certain point of view, Coruna, which does not offer a read-write primitive itself, but two kernel privilege escalations.

SockPuppet (CVE-2019-8605), which was found and developed by Ned Williamson, it a socket teardown path that frees the structure that holds the IPv6 output options from socket and also forgets to clear the pointer to it. As a consequence, the socket itself survives, but the allocated memory content is not there anymore, and the socket option call will be read through a stale and not-existing pointer. Thus, as in classic virtual address UAF attack, we have a scenario where attackers already control of memory, and the stale points is there available to be used. In a few words, the needed steps for a real explanation are finding a lifetime bug, reclaim memory, build a fake port and `tfp0` (task for pid 0), which would return a send right to the kernel task and with all necessary rights that every chain needs to.

PhysPuppet (physical use-after-free) takes a different path, where a virtual memory bug allows an attacker to control a page table entry that still maps a physical page that the kernel already released and reclaimed for another use. In other words, a classic UAF, but this time using physical level.

The first difference between both exploitation techniques is that, in the SockPuppet, the kernel allocates an object into the hole, while in the PhysPuppet the kernel takes the page back for its own use. Additionally, in the SockPuppet, the primitive is obtained by calling into the kernel, whereas that in PhysPuppet the primitive is obtained by performing Load and Store operations from its own process. However, as PhysPuppet acts at the physical page level, the attack is below all heap mitigations. Needless to say, Apple has closed both approaches.

I need to be crystal clear about concepts, descriptions, and mechanisms that I only scratched in this section and previous ones. No doubt, it is not the goal to provide detailed and comprehensive coverage of such concepts in this first article, and that is the main reason I have only revised a few topics, discussed some hypothetical scenarios, and accepted some imprecisions to get a better understanding of context that came up throughout sections, and absolutely nothing more. This is a multi-part series of articles, and future texts will offer better explanation and practice to the mentioned topics.

It is time to resume the discussion about reachability, and under a general and flexible classification, we can assume that every iOS kernel bug (potential vulnerability) belongs to one of available classes of reachability (eventually two classes at the same time), which undoubtedly involves sandbox profiles and entitlements, and learning on it can help us to determine next steps during the research. Based on past experience (and multiple mistakes), researchers who skip this categorization step have a higher chance of building wrong harnesses, wasting time on wrong resources and experiments.

A sandbox profile is a policy based rules that is enforced by kernel, which restricts an iOS process and controls what such each process can or cannot do, as for example if it can write or read certain files, which BSD syscalls can be called, which Mach traps can be invoked, which IOKit user clients are allowed to be opened and which Mach services look up are or not permitted. As general rule, every process on iOS runs inside a sandbox and kernel consults that policy and denies anything the profile does not permit. Details about how a profile is written (in a language called Sandbox Profile Language | SBPL), compiled and enforced through kernel's policy hooks will be commented on in future articles.

An entitlement is a capability embedded into the code signature of a binary, which grants to the binary a specific privilege that kernel, and that the sandbox check for before allowing an operation. In general terms, readers can interpret entitlements as permissions (maybe privileges), which cannot be added into an executable without re-signing it again.

There is a large and important set of entitlements under Apple's private space, denominated by `com.apple.private.*` prefix, which are reserved for Apple's own binaries and as expected, the normal provisioning flow that the Xcode uses to sign an application cannot grant a `com.apple.private*` entitlement. There are two excellent entitlement databases that are worth researching on:

- <https://codecolor.ist/entdb/>
- <https://newosxbook.com/ent.php>

When an Apple code path is protected by entitlements, a regular third-part application cannot either access or open, and it represents a hard control access that establishes what can be or not done. In next

articles, entitlements and code signing mechanism will be reviewed in a bit more details, but I think these few words are enough to follow the known categorization (or classification) shown below.

To determine a bug's classification (or category, as also considered), it is necessary to learn what kind of process, running on a stock and current iOS iPhone, can reach the bug's trigger path and based on three possible answers, we can eventually apply the best class (or category) for such a bug:

- **Class/Category 1:** this class is applied to a normal and ordinary iOS application signed by a developer and that runs inside the standard container sandbox profile, which is the same profile that Apple applies to every App Store application.
- **Class/Category 2:** this class is applied to a privileged process, which carries a `com.apple.private.*` entitlement, and that the standard provisioning flow does not grant. Additionally, it could be a process that has already obtained kernel read-write primitive from a prior exploit's primitive (hard in the current days), or even one running inside a jailbroken environment (equally rare in modern ages) that exposes associated kernel state.
- **Class/Category 3:** this class is applied to a process that runs on specialized hardware such as the Secure Enclave Processor (SEP) that stores keys, the Boot ROM in DFU (Device Firmware Update) mode that is the first stage to be executed on the power-on, and RTKit coprocessor's firmware running on the Always-On Processor (AOP) that is responsible for handling audio, sensors and power management (detailed information on RTKit can be retrieved on <https://theapplewiki.com/wiki/RTKit>). Obviously, normal iOS process cannot touch these hardware parts and neither change their states even holding a kernel read-write primitive.

As readers can predict, these three categories are not equally productive ground for research, and each one demands a distinct amount of effort, time, and expertise (maybe not in this order). In general, most paid Apple bounty submissions seems come, comprehensively, from Class/Category 1 because the surface is the largest and the reach is the most direct and natural from a real attach. Class/Category 2 supplies the post-primitive escalation paths in modern chains, where the initial trigger supplied by a bug from Category 1 composes with one or more Category 2 escalations into a working chain, and this will be arrangement that will be seen multiple times. Naturally, as bugs/vulnerabilities from Class/Category 3 occur in specialized surface, they demands skill, focused research, hardware resource, and appropriate tooling that should be built. Anyway, according to your progress and increase of iOS research experience, soon or later you will migrate to class/category 3, which is outside the iOS kernel, and usually based on a separated processor with its own firmware.

As nothing is exact, the boundaries between categories are sometimes fuzzy, and bug can be partially in Category 1 and partially in Category 2 if there are multiple trigger paths that can reach the same vulnerable function. Furthermore, some bugs may migrate between categories across iOS releases as Apple adds or removes entitlements that limit a specific user method or sandbox profile rule.

There are further details that can be expanded based on the mentioned bug classification. Class/Category 1 bug is one whose trigger path is reachable by a process running under the normal iOS application sandbox, without any entitlement beyond what a regular application signed by a developer really has, and without any prior exploitation. As I described previously, this is the most natural (or most common, at least) scenario you will find for exploitation because the testing process is unmodified and you are using an iOS

application that is signed by a developer, installed through Xcode, and thus it runs like any App Store application from iOS's perspective.

In this scenario, if we can reach the bug from a regular application (written by a third party developer), such a bug is Category 1 because it is using the referenced sandbox profile, which is the container profile that Apple applies to App Store applications. This profile is produced in [Sandbox Profile Language \(SBPL\)](#), that is a descriptive policy language, but iOS does not ship code lines of a profile (on macOS, it can be found in [/System/Library/Sandbox/Profiles/](#) directory). Instead, Apple builds and compiles every profile to bytecode and bundles them into one profile collection inside the sandbox kext ([com.apple.security.sandbox](#)), so the format that such profile is shown in device is a compiled binary blob, and not a readable container.sb file, unfortunately. For sure, it is possible to recover a readable form by decompiling the collection with tools dedicated to profile extraction, but such a recovered container profile is large, on the order of hundreds of distinct rules, and it defines what file paths the app can read and write, which Mach services can look up, which IOKit user clients can open, which sysctl can read, and other various aspects, as exposed previously.

Sandbox profiles can allow the following tasks:

- Read and write operations within the container directory of the own application, which stays in [/var/mobile/Containers/Data/Application/<UUID>/](#) directory.
- Read access to most of [/System](#) directory, which are Apple's read-only system files, and also [/usr](#) folder, where operating system executables and libraries are stored.
- Mach message send-rights to most published Mach services, which are advertised by launched through [bootstrap_look_up](#), a Mach IPC function used by iOS to find and acquire a send-right to a determined registered system service port (of course, there are some specific denied services).
- Direct invocation of most BSD syscalls such as [socket](#), [bind](#), [connect](#), [send](#), [recv](#), [open](#), [read](#), [write](#), [pread](#), [pwrite](#), [pwritev](#), [ioctl](#) on non-controlled device nodes, and many others.
- Direct invocation of most Mach traps such as [mach_msg](#), [mach_vm_protect](#), [mach_port_allocate](#), [mach_port_insert_right](#), [mach_port_mod_refs](#), [mach_port_construct](#) and [mach_port_destruct](#).
- Invocation of MIG Mach RPCs that reach the same VM and thread subsystems, and that send a message to the task port of the caller, such as [mach_vm_read_overwrite](#), [mach_vm_write](#), [mach_vm_remap](#), [task_threads](#), [thread_resume](#), and [thread_suspend](#). As mentioned previously, MIG (Mach Interface Generator) is a tool that is responsible for producing userland stubs and dispatchers (kernel-side) for routines based on messages, and in our mentioned context, the sandbox allows them access to their own task while denies access against another task, which is reasonable.
- Open IOKit user clients that do not require privileged entitlement:

From practical point of view, this article will use sample application that is signed with a regular account, and that probes every IOKit driver in the system register by calling [IOServiceOpen](#) function (used to submit a request to create a connection to an IOService) on each one, and records whether kernel allows or defined the open action. During the execution of this simple application, it passes the user-client type 0, which is the default type that most drivers accept) and records the IOReturn result code.

The following reachability test, named [iokit_probe](#), and that is embedded into the IOKitAcess application, has run on iPhone 16 (iOS 27 beta 8 | build 24A5430a for this experiment) that is paired with the macOS and, as part of the methodology, it was repeated three times, and obtained identical results. The project

(IOKitAccess) is composed of a project file named `project.yml` (it describes the target) and produces the `IOKitAccess.xcodeproj` bundle. As a warning, I needed to include three settings in `project.yml` to producing a working building: `OTHER_LDFLAGS`, `SWIFT_OBJC_BRIDGING_HEADER` and `GENERATE_INFOPLIST_FILE`. The mentioned file is shown below:

```
# project.yml
```

```
name: IOKitAccess
options:
  bundleIdPrefix: com.alexandreborges
  deploymentTarget:
    iOS: "26.0"
targets:
  IOKitAccess:
    type: application
    platform: iOS
    sources: [IOKitAccess]
    settings:
      base:
        PRODUCT_BUNDLE_IDENTIFIER: com.alexandreborges.IOKitAccess
        PRODUCT_NAME: IOKitAccess
        DEVELOPMENT_TEAM: F5FWZFTZ52
        CODE_SIGN_STYLE: Automatic
        CODE_SIGN_ENTITLEMENTS: IOKitAccess/app.entitlements
        GENERATE_INFOPLIST_FILE: YES
        INFOPLIST_KEY_LSRequiresiPhoneOS: YES
        INFOPLIST_KEY_UILaunchScreen_Generation: YES
        SWIFT_OBJC_BRIDGING_HEADER: IOKitAccess/Bridging.h
        OTHER_LDFLAGS: -framework IOKit -framework CoreFoundation
```

The `DEVELOPMENT_TEAM` setting needs to be replaced by the readers. The entitlements file (`IOKitAccess/app.entitlements`) requests only one entitlement (`get-task-allow`), as shown below:

```
<!-- IOKitAccess/app.entitlements -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>get-task-allow</key>
  <true/>
</dict>
</plist>
```

The bridging header (`IOKitAccess/Bridging.h`) has only three lines long, and actually only exists to make viable that the Swift entry point can see the C function:

```
// IOKitAccess/Bridging.h

#ifdef IOKITACCESS_BRIDGING_H
#define IOKITACCESS_BRIDGING_H
void iokit_probe_run(void);
#endif
```

The application entry point is SwiftUI rather than a plain application delegate because, from iOS 26 onward, the UIKit requires an application to adopt the scene lifecycle, and an application written using the older approach (an AppDelegate with `didFinishLaunchingWithOptions`) is killed at launch by the runtime check, which raises a trap before any of code run application runs:

```
// IOKitAccess/IOKitAccessApp.swift

import SwiftUI
```

```
@main
struct IOKitAccessApp: App {
    init() {
        iokit_probe_run()
        fflush(stdout)
    }
    var body: some Scene { WindowGroup { Text("IOKitAccess: probe finished") } }
}
```

The source code of the probe.c (shown below) is the file that carries the key method, and asks the kernel for every registered service, tries to open a user client on each one and finally print those that succeed together with the registry path of the object that accepted such a request:

// IOKitAccess/probe.c

```
#include <stdio.h>
#include <CoreFoundation/CoreFoundation.h>
#include <IOKit/IOKitLib.h>

void iokit_probe_run(void) {
    io_iterator_t it = IO_OBJECT_NULL;
    if (IOServiceGetMatchingServices(kIOMainPortDefault,
                                    IOServiceMatching("IOService"),
                                    &it) != KERN_SUCCESS || !it) {
        printf("IOService match failed\n");
        return;
    }

    unsigned matched = 0, opened = 0;
    io_service_t s;
    while ((s = IOIteratorNext(it))) {
        io_name_t cls = {0};
        if (IOObjectGetClass(s, cls) == KERN_SUCCESS) {
            matched++;
            io_connect_t c = IO_OBJECT_NULL;
            kern_return_t kr = IOServiceOpen(s, mach_task_self(), 0, &c);
            if (kr == KERN_SUCCESS) {
                opened++;
                io_string_t path = {0};
                IORegistryEntryGetPath(s, kIOServicePlane, path);
                printf("OPEN %-24s %s\n", cls, path);
                IOServiceClose(c);
            }
        }
        IOObjectRelease(s);
    }
    IOObjectRelease(it);
    printf("matched=%u opened=%u\n", matched, opened);
    fflush(stdout);
}
```

Readers observing the program above can notice that the matching string is the strategy compressed into one argument, and `IOServiceMatching("IOService")` call really matches every service on device because associated drivers descends from `IOService`. Although this approach is simple, it is better than doing a check against a list of names that a researcher might have written down in advance. `IOServiceOpen` function asks the kernel to create a user client, which is per-process handle through which the application sends commands to a kernel driver, and the object every existing IOKit vulnerability eventually operates against. As mentioned earlier, a "0" is passed as connection type and it selects which user client class the contacted driver should instantiate, but only a small number of drivers will open, and `IORegistryEntryGetPath` function records which object accepted the request.

If you are operating remotely via SSH, eventually you will handle with the keychain access and then build the project:

```
$ cd ~/research/work/article/iokitaccess
$ export DEVELOPER_DIR=~/.Downloads/Xcode-beta.app/Contents/Developer

$ xcodegen generate

$ stty -echo; printf "login password: "; read KEYCHAIN_PW; stty echo; echo
$ export KEYCHAIN_PW

$ security unlock-keychain -p "$KEYCHAIN_PW" ~/Library/Keychains/login.keychain-db
$ security set-key-partition-list -S apple-tool:,apple:,codesign: -s -k "$KEYCHAIN_PW"
~/Library/Keychains/login.keychain-db

$ xcodebuild -project IOKitAccess.xcodeproj -scheme IOKitAccess \
    -configuration Release \
    -destination "id=00008140-001868412EEB001C" \
    -allowProvisioningUpdates ENABLE_DEBUG_DYLIB=NO \
    -derivedDataPath build build

** BUILD SUCCEEDED **

$ xcrun devicectl device install app --device 2FC3AF78-2618-5DA3-89A6-35EE8BECA75B
build/Build/Products/Release-iphones/IOKitAccess.app

bundleID: com.alexandreborges.IOKitAccess
installationURL: file:///private/var/containers/Bundle/Application/.../IOKitAccess.app/

$ xcrun devicectl device process launch --console --terminate-existing --device 2FC3AF78-2618-5DA3-
89A6-35EE8BECA75B com.alexandreborges.IOKitAccess
```

There are a few points that could go wrong while you try this sequence of commands. For example, the usage of two identifiers that represent different kinds of names for the same iPhone device. At the first part, you see `xcodebuild -destination` command, which takes the hardware UDID (00008140-), while `devicectl --device` command takes the **CoreDevice UUID**, which begins with "2FC3AF78-", and if you swap them you will receive an error message similar to "device not found", but it has nothing to do with the device. Another parameter that may cause issues is `ENABLE_DEBUG_DYLIB=NO` because the default Debug configuration emits a separate debug dylib that must be signed as well. Another option that is relevant is `--terminate-existing`, which makes a new execution of the new build rather than re-attaching to the copy already running from the previous one.

The application runs on iPhone and its output, similar to the following one, is made visible on macOS due to the `--console` parameter specified in the `xcrun` command:

```
OPEN  AppleKeyStore          IOService:/IOResources/AppleKeyStore
OPEN  IOSurfaceRoot          IOService:/IOResources/IOCoreSurfaceRoot
OPEN  AppleJPEGDriver        IOService:/AppleARMPE/arm-
io@10F00000/AppleH17PPlatformIO/jpeg1@7D060000/AppleJPEGDriver
OPEN  AppleM2ScalerCSCDriver IOService:/AppleARMPE/arm-
io@10F00000/AppleH17PPlatformIO/scaler0@81000000/AppleM2ScalerCSCDriver
OPEN  AppleCLCD2             IOService:/AppleARMPE/arm-
io@10F00000/AppleH17PPlatformIO/dispext0@40000000/AppleCLCD2
OPEN  AppleCLCD2             IOService:/AppleARMPE/arm-
io@10F00000/AppleH17PPlatformIO/disp0@0/AppleCLCD2
    matched=1398 opened=6
```

Probably the most relevant fact from this output is that it shows that there are 1398 services running, which the sandboxed container can see, and six of them accepted `IOServiceOpen` using connection type 0, and these services are spread across five distinct driver classes:

- **AppleKeyStore**: it is the keychain and Data Protection key management driver.
- **IOSurfaceRoot**: it is the shared-memory surface driver (the heap-spray resource).
- **AppleJPEGDriver**: it is the hardware JPEG encoder/decoder driver.
- **AppleM2ScalerCSCDriver**: it is a display scaler and color-space conversion driver.
- **AppleCLCD2**: it is the display controller driver, which is present as two instances.

As a valid note, it would be possible to probe by name rather than by object, and this check can be performed by asking the registry for one name at a time instead of trying to every service. The version presented below prints, for each name, every object that name matched, the concrete class of the kernel instantiated, the result of opening it at connection type 0 and, finally, the registry path of the object itself. The project and the build commands are unchanged, and only **IOKitAccess/probe.c** are different, as shown below:

```
// IOKitAccess/probe.c, second version. This replaces the contents of the
// probe.c printed earlier.

static void match_by_name(const char *name) {
    io_iterator_t it = IO_OBJECT_NULL;
    if (IOServiceGetMatchingServices(kIOMainPortDefault, IOServiceMatching(name), &it) !=
    KERN_SUCCESS || !it) {
        printf("match=%-22s NO MATCH\n", name);
        return;
    }
    io_service_t s;
    unsigned n = 0;
    while ((s = IOIteratorNext(it))) {
        n++;
        io_name_t cls = {0};
        io_string_t path = {0};
        IOObjectGetClass(s, cls);
        IORegistryEntryGetPath(s, kIOServicePlane, path);
        io_connect_t c = IO_OBJECT_NULL;
        kern_return_t kr = IOServiceOpen(s, mach_task_self(), 0, &c);
        printf("match=%-22s inst=%u class=%-12s open=0x%08x\n", name, n, cls, kr);
        printf("  path=%s\n", path);
        if (kr == KERN_SUCCESS) IOServiceClose(c);
        IOObjectRelease(s);
    }
    if (n == 0) printf("match=%-22s matched 0 instances\n", name);
    IOObjectRelease(it);
}

void iokit_probe_run(void) {
    const char *names[] = { "IOMobileFramebuffer", "AppleCLCD2", NULL };
    for (int i = 0; names[i]; i++) match_by_name(names[i]);
    fflush(stdout);
}
```

Actually, nothing else is replaced in the project. Rebuilding the project does not need to repeat the whole sequence of commands provided previously:

- **xcode generate** command does not need to be repeated because it regenerates the Xcode project from "project.yml", which has not been changed.
- **security set-key-partition-list** command does not need to be repeated because it rewrites the permission stored on the signing key itself.
- **security unlock-keychain** should be repeated because it is per terminal session, and there is the fact that keychain relocks and also **codesign** command cannot ask a non-interactive session to unlock it.
- As expected, the **build**, the **install** and the **launch** commands should be repeated.

According to items above, run the following commands:

```
$ security unlock-keychain -p "$KEYCHAIN_PW" ~/Library/Keychains/login.keychain-db

$ xcodebuild -project IOKitAccess.xcodeproj -scheme IOKitAccess -configuration Release -destination
"id=00008140-001868412EEB001C" -allowProvisioningUpdates ENABLE_DEBUG_DYLIB=NO -derivedDataPath
build build

** BUILD SUCCEEDED **

$ xcrun devicectl device install app --device 2FC3AF78-2618-5DA3-89A6-35EE8BECA75B
build/Build/Products/Release-iphonios/IOKitAccess.app

$ xcrun devicectl device process launch --console --terminate-existing --device 2FC3AF78-2618-5DA3-
89A6-3
```

The output from the executed program reports:

```
match=IOMobileFramebuffer inst=1 class=AppleCLCD2 open=0x00000000
path=IOService:/AppleARMPE/arm-io@10F00000/AppleH17PPlatformIO/dispext0@4000000/AppleCLCD2
match=IOMobileFramebuffer inst=2 class=AppleCLCD2 open=0x00000000
path=IOService:/AppleARMPE/arm-io@10F00000/AppleH17PPlatformIO/disp0@0/AppleCLCD2
match=AppleCLCD2 inst=1 class=AppleCLCD2 open=0x00000000
path=IOService:/AppleARMPE/arm-io@10F00000/AppleH17PPlatformIO/dispext0@4000000/AppleCLCD2
match=AppleCLCD2 inst=2 class=AppleCLCD2 open=0x00000000
path=IOService:/AppleARMPE/arm-io@10F00000/AppleH17PPlatformIO/disp0@0/AppleCLCD2
```

In general, the four paths are reduced to two, and they are the same two. Moreover, IOMobileFramebuffer and AppleCLCD2 are not two drivers that both open, but one pair of objects reachable under two names. Each of these five classes is an IOKit service class (a C++ class in the kernelcache) that implements a kernel driver, and as a sandboxed application can open a user client to it, it is where Class/Category 01 IOKit bug might be found.

Although I have used a connection typed 0, most driver classes are reached using other types (0 to 15), and a slight change in the code can turn the opened six events into 147 and also five reachable classes into 17. Once I said it, it is possible to write a third version of the probe.c application, which asks for a short list of names and then walks every service, but this time looping over the 16 connection types. At the end, one line is printed for every open that succeeded. Therefore, the third version of the application follows below:

// IOKitAccess/probe.c, third version.

```
static void probe_class(const char *name) {
    io_iterator_t it = IO_OBJECT_NULL;
    if (IOServiceGetMatchingServices(kIOMainPortDefault,
                                     IOServiceMatching(name), &it) != KERN_SUCCESS || !it) {
        printf(" %-22s NO MATCH\n", name); return;
    }
    io_service_t s; unsigned n = 0;
    while ((s = IOIteratorNext(it))) {
        n++;
        io_name_t cls = {0};
        IOObjectGetClass(s, cls);
        io_connect_t c = IO_OBJECT_NULL;
        kern_return_t kr = IOServiceOpen(s, mach_task_self(), 0, &c);
        printf(" %-22s inst=%u class=%-26s type=0x%08x\n", name, n, cls, kr);
        if (kr == KERN_SUCCESS) IOServiceClose(c);
        IOObjectRelease(s);
    }
    if (n == 0) printf(" %-22s matched 0 instances\n", name);
    IOObjectRelease(it);
}
```

```
void iokit_probe_run(void) {
    printf("== targeted GPU / accelerator names ==\n");
    const char *gpu[] = {"AGXAccelerator", "IOGPU", "IOGPUDevice", "IOAccelerator",
                        "AGXAcceleratorG17P", "IOMobileFramebuffer", "AppleAVD", NULL};
    for (int i = 0; gpu[i]; i++) probe_class(gpu[i]);

    printf("== type sweep 0..15 over every IOService ==\n");
    io_iterator_t it = IO_OBJECT_NULL;
    if (IOServiceGetMatchingServices(kIOMainPortDefault,
                                    IOServiceMatching("IOService"), &it) != KERN_SUCCESS || !it) {
        printf("sweep match failed\n"); return;
    }
    unsigned matched = 0, events = 0;
    io_service_t s;
    while ((s = IOIteratorNext(it))) {
        io_name_t cls = {0};
        if (IOObjectGetClass(s, cls) == KERN_SUCCESS) {
            matched++;
            for (uint32_t t = 0; t < 16; t++) {
                io_connect_t c = IO_OBJECT_NULL;
                kern_return_t kr = IOServiceOpen(s, mach_task_self(), t, &c);
                if (kr == KERN_SUCCESS) {
                    events++;
                    printf("TYPEOPEN %-26s type=%u\n", cls, t);
                    IOServiceClose(c);
                }
            }
        }
        IOObjectRelease(s);
    }
    IOObjectRelease(it);
    printf("sweep matched=%u openevents=%u\n", matched, events);
    fflush(stdout);
}
```

I will not repeat details of the building and deployment cycle again, and the program prints one "TYPEOPEN" line per successful open and a total at the end. The difference is that we will redirect the output to a file (sweep.txt), which will be handled in the next commands:

```
$ xcrun devicectl device process launch --console --terminate-existing --device 2FC3AF78-2618-5DA3-89A6-35EE8BECA75B com.alexandreborges.IOKitAccess > sweep.txt 2>&1 &

$ LAUNCH=$!

$ until grep -q '^sweep matched=' sweep.txt 2>/dev/null; do sleep 2; done
$ kill $LAUNCH
```

The loop above wait for the last line of the output (it is not recommended to guess the program duration). The output from sweep.txt follows below:

```
TYPEOPEN H1xANELoadBalancer      type=1
TYPEOPEN H1xANELoadBalancer      type=4
TYPEOPEN AppleKeyStore           type=0
TYPEOPEN AppleKeyStore           type=1
TYPEOPEN AppleKeyStore           type=2
TYPEOPEN AppleKeyStore           type=3
...
sweep matched=1398 openevents=147
```

There are 147 lines and they compose the measure of this example, and it is possible to make a kind of summary by running the following commands:

```
$ tr -d '\r' < sweep.txt | grep TYPEOPEN | awk '{print $2}' | sort -u | wc -l
17
```

```
$ tr -d '\r' < sweep.txt | awk '/^TYPEOPEN/ {
    t = $3; sub("type=", "", t); n[$2]++
    if (!((($2 SUBSEP t) in seen)) {
        seen[$2 SUBSEP t] = 1; k[$2] = k[$2] " " t; d[$2]++ } }
END { for (c in n)
    printf "%-34s events=%-4d objects=%-3d types:%s\n",
        c, n[c], n[c]/d[c], k[c] }' | sort
```

The output is the following:

AGXAcceleratorG17P	events=1	objects=1	types: 1
AHSTemperatureReporter	events=1	objects=1	types: 2
AppleARMPMUPowerSensor	events=11	objects=11	types: 2
AppleARMPMUTempSensor	events=19	objects=19	types: 2
AppleCLCD2	events=32	objects=2	types: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
AppleEmbeddedNVMeTemperatureSensor	events=1	objects=1	types: 2
AppleJPEGDriver	events=16	objects=1	types: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
AppleKeyStore	events=16	objects=1	types: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
AppleM2ScalerCSCDriver	events=16	objects=1	types: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
AppleM68Buttons	events=1	objects=1	types: 2
AppleSPUHIDDriver	events=26	objects=26	types: 2
AppleSPUVD6287	events=1	objects=1	types: 2
AppleSphinxProxHIDEventDriver	events=1	objects=1	types: 2
EXBrightKext	events=1	objects=1	types: 2
H1xANELoadBalancer	events=2	objects=1	types: 1 4
IOHIDEventDriver	events=1	objects=1	types: 2
IOSurfaceRoot	events=1	objects=1	types: 0

According to the output, it is the column reporting objects that makes the total add up, and an open event is one object opening at one type, so a class contributes as many events as it has objects multiplied by the number of types each accepts. Thus, there are 147 open events, 71 distinct objects and 17 classes, which means that, in general, there are 17 kernel driver classes that are reachable from a sandboxed application. However, the value 71, which is the list of driver objects that we can hold a connection to, is more solid for attacking purposes. About the value 147, it is not a count of drivers because it counts connections, which is an object opened at one type, and it multiplies the same object up to 16 times. The kernelcache contains around 259 user client classes, which are drivers that we can potentially talk to.

There is a series of specific cases that we could comment on in detail such AppleJPEGDriver, graphic drivers, video decoder (AppleAVD), and other ones, but it is out of the context for this article. In general, the usual request path is [IOServiceOpen](#) function (when the request cross into the kernel) → [is_io_service_open_extended](#) function (it is a common routine for every driver on the system and when the sandbox policy is consulted) → [IOService::newUserClient](#) function (the driver could replace with its own, but it is where the type-value is examined) → [driver client object](#) (and external methods behind it).

Our real scenario is that we have 17 kernel driver classes that are reachable, but obviously the total number of user classes is higher than it (259) and the number of classes registered is much higher than it (3360), as it is shown below:

```
$ ipsw kernel cpp --json kc_27.0b8.bin > classes.json

# iokitclasses.py
import json, sys
```



```
C = json.load(open(sys.argv[1]))
meta = {c['MetaPtr']: c for c in C}

def ancestors(c):
    seen, cur = set(), c
    while cur and cur['Name'] not in seen:
        seen.add(cur['Name'])
        cur = meta.get(cur['SuperMeta'])
    return seen

uc      = [c for c in C if 'IOUserClient' in ancestors(c)]
service = [c for c in C if 'IOService'    in ancestors(c)]
print("classes registered      : %d" % len(C))
print("descend from IOService  : %d" % len(service))
print(" user client classes    : %d" % len(uc))
print(" driver classes        : %d" % (len(service) - len(uc)))
```

Running this script against the iOS 27 beta 8 kernelcache for this device, we have:

```
$ python3 iokitclasses.py classes.json
```

```
classes registered      : 3360
descend from IOService  : 1638
user client classes    : 259
driver classes         : 1379
```

In a few words, we have 17 accessible kernel driver classes and, to expand this number, one of possibilities would be try a sandbox escape, and there is a list of excellent research articles commenting on this exploitation step, which shows that it is not an impossible task. Once again, we cannot forget that there are actions that the profile does not allow.

As expected, calling [task_for_pid](#) function against another process (basically a low level Mach operation that obtain the Mach task port of another target process via its PID) is not allowed by the sandbox unless that target carries the [get-task-allow](#) entitlement (it controls whether another process is permitted or not to attach to the application) and the caller carries matching credentials. Actually, there are other multiple restrictions involving [host_priv](#) operations, sandboxed processes barred from open IOKit user clients that need [com.apple.private*](#) entitlement and others that directly invokes Mach IPC services.

Mach layer is a dense topic, even though I would like to comment on some points that are directly associated with sandbox and, specifically, which each one of them offers to an attacker, certainly is not the right moment to do it.

To wrap up this extensive section, which ran longer than it should have for this first article, next lines do a quick summary about bug/vulnerabilities in class/category 1, 2 and 3.

I have already described the order that a direct approach reaches a driver via IOServiceOpen function, but it is not the only way to get there. As an alternative, the application can call a framework (ImageIO, CoreML, Metal, AppleAVD and so on) and, from there, the framework opens the user client on behalf of application. From this point, the sequence is the same. For some drivers, which is the only way, but in multiple cases both paths coexist, so there are real scenarios that the first path fails, but the second one works perfectly.

However, take care with an obvious fact: being reachable does not mean to be vulnerable, even because we are not considering multiple other protections (multiple ones already commented) such as PAC (sign pointer), STPM (an attacker cannot make a page that holds a malicious code as executable and run it, and

an attacker cannot point a virtual address at a different physical page), TXM (unsigned code cannot be executed), Exclaves (sensitive components are isolated into a separated domain where kernel cannot reach), kalloc_type (each type of object has its own pool and a freed slot can only be reused by another object of the same type, which kills UAF) and zone_require (checks if the object really lives in its proper pool before kernel trusts it).

Of course, none of these mitigations has any effect on the first step of a real attack, which triggers the bug in the first place. As an example, a hypothetical bug in the handling of `mach_msg` system call, which every process uses to send and receive Mach messages (communication between processes and kernel), might generate a UAF through a refcount issue and, consequently, where we have a fake object under our control, and this would represent a good milestone towards to full kernel read and write primitive.

In simple terms, as `mach_msg` call is reachable from every sandboxed application, attackers do not need special privilege to start. Thus, the bug occurs in timing and shared state, and mitigations such as PAC, STPM and KTRR do not have effect in this case. As I explained earlier, I will not discuss on Mach concepts in this first article and there is a "similar" real vulnerability related to race condition vulnerability class (and in special, a double race) that involves a Mach voucher, a kernel object that a process creates and attach to a Mach message, a thread or to a task. Eventually, we will return and discuss such a CVE in this series.

As you will learn, Mach IPC, BSD calls, IOKit external methods and network stack have been the code path taken by iOS attackers in the last decade, and it stands for now and next years because a sandboxed application can reach every one of them, any bug in these areas can be classified as Class/Category 1.

A Class/Category 2 bug is one whose vulnerable path is reachable only after kernel privileges has been established and, for this reason, it is more difficult to see because it can depend on a kernel read-write primitive, a jailbroken environment, SRD, or a device with a special profile that grants needed entitlements. After we have achieved read-write primitive, there are some possibilities, and one of most typical them is elevation of privileges. There would be other possibilities, but SPTM and TXM will block them. In practice, I should warn you that it is hard work obtaining a kernel read-write primitive.

Class/Category 3 vulnerabilities exist entirely outside the reach of standard iOS userland processes, targeting isolated environments like the Secure Enclave Processor (SEP), the early-stage boot chain (LLB, iBoot), or hardware-level cryptography engines. Exploiting these surfaces frequently demands physical access or physical hardware exploitation chains (such as JTAG debuggers or specialized cables).

A premier example of this is the `usbliter8` exploit disclosed by Paradigm Shift in June 2026. This untethered hardware flaw leverages a DMA vulnerability within the Synopsys USB controller driven by SecureROM, achieving arbitrary code execution at the hardware root-of-trust layer on A12 and A13 silicon.

Unfortunately, due to legal issues, the original write up and respective code are no longer available, but you can read about this vulnerability on <https://honeynet.org.mx/posts/usbliter8-en/>, <https://apple.gadgethacks.com/news/unpatchable-iphone-exploit-a12-a13-chips-usbliter8-explained/> and <https://thehackernews.com/2026/06/unpatchable-usbliter8-exploit-breaks.html>.

This section has come to an end, and it is time to move on to the sections featuring practical demonstrations. All the following sections assume that you already have IDA Pro, a computer with Apple Silicon, sufficient disk space, and all the suggested programs already installed.

09. Building the kernelcache pipeline

After dense discussions for a starting article, from this point onward, steps will present useful resources used actively on iOS research. I will be using an iOS version that will change over time and, for sure, such versions themselves are not important right now. Thus, we will start at the most important and key artifact from any iOS research that is the kernelcache, which contain the iOS kernel. Basically, kernelcache comes without any associated symbol, and it makes the research process and analysis more difficult. Therefore, the first critical step is always to download the kernelcache and symbolicate it.

The general kernelcache symbolication process is composed of the following major steps:

- Download, extract, and decompress the kernelcache.
- Open it on IDA Pro.
- (optional) Use Lumina for applying names to every function whose fingerprint other professionals have already published.
- Use ipsw command for symbolication process.
- Use Cellebrite plugins to rebuild C++ class hierarchy by reading names applied previously.

Once macOS is configured and tools have been installed, next step is to establish a kernelcache pipeline and provide readers with a solid artifact for iOS research. I will be using IDA Pro 9.4 and 8.4 at the same procedure, and reasons will be cleared throughout the explanation. As a side comment, next steps have been successfully used during my daily work, including last iOS versions.

Other resources used in this approach include:

- **Target host:** macOS M2, with IDA Pro 8.4 and 9.4
- **Windows host:** 64 GB | x64, with IDA 8.4 and 9.4
- **Cellebrite plugins:**
 - **ida_kernelcache:** https://github.com/cellebrite-labs/ida_kernelcache
 - **ida_kcpp:** https://github.com/cellebrite-labs/ida_kcpp

One of first step is to check available iOS versions on release, beta, and release candidate "channels" by executing the following commands:

```
$ ipsw download appledb --os iOS --device iPhone17,3 --release --show-latest
  • Querying AppleDB...
{
  "OS": "iOS",
  "version": "26.6.2",
  "build": "23G90",
  "released": "2026-09-08T00:00:00Z"
}

$ ipsw download appledb --os iOS --device iPhone17,3 --beta --show-latest
  • Querying AppleDB...
{
  "OS": "iOS",
  "version": "27.0 beta 8",
  "build": "24A5430a",
  "beta": true,
  "released": "2026-08-31T00:00:00Z"
}

$ ipsw download appledb --os iOS --device iPhone17,3 --rc --show-latest
  • Querying AppleDB...
```

```
{
  "OS": "iOS",
  "version": "27.0 RC",
  "build": "24A435",
  "released": "2026-09-09T00:00:00Z"
}
```

Pick up the version that is relevant for your research and download the respective kernelcache as shown below:

```
$ cd C:\research_ios\kernelcaches\ios27rc
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24A435 --kernel
Created 24A435__iPhone17,3\kernelcache.research.iphone17
Created 24A435__iPhone17,3\kernelcache.release.iphone17
```

Alternatively, it is possible to do the same operation by finding the correct URL and then downloading and extracting it at the same time:

```
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24A435 --urls
https://updates.cdn-apple.com/2026FallFCS/2d0cd01d-b4f9-4a20-a1e8-
f3be54570da7/iPhone17,3_27.0_24A435_Restore.ipsw

$ ipsw extract --kernel --remote https://updates.cdn-apple.com/2026FallFCS/2d0cd01d-b4f9-4a20-a1e8-
f3be54570da7/iPhone17,3_27.0_24A435_Restore.ipsw

$ ipsw macho info 24A435__iPhone17,3/kernelcache.release.iphone17 | head -5
Magic          = 64-bit MachO
Type           = FILESET
CPU            = AARCH64, ARM64e caps: KER00
Commands      = 318 (Size: 22888)
Flags          = None
```

If you need to download the entire restore image for some additional purpose:

```
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24A435 --confirm
  • Querying AppleDB...
  • Getting (1/1) IPSW: iPhone17,3_27.0_24A435_Restore.ipsw
    11.50 GiB / 11.50 GiB [=====] [✓] ]
64.73 KiB/s
```

At the same way, you can extract the kernelcache by executing the following command:

```
$ ipsw extract --kernel iPhone17,3_27.0_24A435_Restore.ipsw
  • Extracting kernelcache
    • Created 24A435__iPhone17,3\kernelcache.research.iphone17
    • Created 24A435__iPhone17,3\kernelcache.release.iphone17
```

As an additional note, if you had downloaded the kernelcache from another place, it may also need to be decompressed:

```
$ ipsw kernel dec <kernelcache> -o <output>
```

Once you have the kernelcache, you can do fundamental tasks. For example, check the UUID that ties a panic log to this binary, list the XNU version, obtain the associated hash, and renamed it for something meaningful, as shown below:

```
$ ipsw macho info 24A435__iPhone17,3/kernelcache.release.iphone17 | grep LC_UUID
000: LC_UUID          06487232-5835-1CAC-9752-B360B5DF2233

$ ipsw kernel version 24A435__iPhone17,3/kernelcache.release.iphone17
```

Darwin Kernel Version 27.0.0: Thu Aug 13 21:39:16 PDT 2026; root:xnu-13432.2.10~2/RELEASE_ARM64_T8140

```
$ sha256sum 24A435__iPhone17,3/kernelcache.release.iphone17
35534b91dcfcd975e1d6576aa43c02531b0405943d6d8629aff4376c359d57d2
*24A435__iPhone17,3/kernelcache.release.iphone17

$ cp 24A435__iPhone17,3/kernelcache.release.iphone17 kc_27.0rc.bin
```

The next step is to perform the symbolication of the kernelcache, which will aggregate relevant information such as Mach trap table entries, BSD system call entries, C++ class vtable symbols and exported kext symbols:

```
$ cd C:\research_ios
$ git clone --depth 1 https://github.com/blacktop/symbolicator
$ ipsw kernel symbolicate --signatures C:\research_ios\symbolicator\kernel\27.0 --json -o .
kc_27.0rc.bin
```

```
• Parsing Signatures
• Symbolicating...      kernelcache=kc_27.0rc.bin
• Found                bsd_syscall_table=0xffffffff007e417b0
• Found                mach_trap_table=0xffffffff007e10018
• Found                mig_kern_subsystem_table=0xffffffff00b6fcb30
...
• Symbolicated          address=0xffffffff00b066930 file=com.apple.kernel
symbol=nd6_rti_list_wait
• Symbolicated          address=0xffffffff00b078174 file=com.apple.kernel
symbol=key_getsaddbaddr
• Symbolicated          address=0xffffffff00b07bcbc file=com.apple.kernel
symbol=key_setdumpsp
• [🔍] Symbolication STATS    matched=5130 missed=1380 percent=78.8018% total=6510
• Writing symbols as JSON to kc_27.0rc.bin.symbols.json
```

Check the produced JSON file:

```
$ cat .\kc_27.0rc.bin.symbols.json | jq | head -10
{
  "18446744005122261016": "mach_trap_table",
  "18446744005122463664": "bsd_syscall_table",
  "18446744005122478736": "vtable for OSMetaClass::MetaClass",
  "18446744005122478872": "vtable for OSMetaClass",
  "18446744005122479008": "vtable for OSObject::MetaClass",
  "18446744005122479152": "vtable for OSArray",
  "18446744005122479488": "vtable for OSArray::MetaClass",
  "18446744005122479632": "vtable for OSBoolean",
  "18446744005122479808": "vtable for OSBoolean::MetaClass",
}
```

A useful artifact for reachability analysis is the entitlement database, and it can be generated by executing the following command:

```
$ ipsw ent --sqlite entitlements.db --ipsw iPhone17,3_27.0_24A435_Restore.ipsw
• Processing IPSW      ipsw=iPhone17,3_27.0_24A435_Restore.ipsw
  • Generating entitlement database file...
    • Scanning AppOS
    • Scanning SystemOS
    • Scanning Filesystem
    • Scanning ExclaveOS
  • Successfully processed ipsw=iPhone17,3_27.0_24A435_Restore.ipsw
  • Database creation/update completed successfully

$ sqlite3 .\entitlements.db
SQLite version 3.50.6 2025-09-22 18:50:24
Enter ".help" for usage hints.
```

```
sqlite> .tables
devices                entitlements             ipsws                  names
dsc_images             ipsw_devices           kernelcache_kexts     paths
dyld_shared_caches     ipsw_dscs              kernelcaches           symbols
entitlement_keys        ipsw_files             macho_syms
entitlement_values      ipsw_kernels           machos

sqlite> select * from entitlement_values limit 10;

1|["NULL/ActivationRegulatoryVariant"]|array|75277c906f6b4117
2|["AppleBiometricServicesUserClient", "AppleSEPUserClient", "IOMobileFramebufferUserClient", "AppleNVMeEANUC"]|array|f93d3f334daa599b
3|["systemgroup.com.apple.mobilegestaltcache", "systemgroup.com.apple.mobileactivationd"]|array|42452c2933b1fd83
4|["spi", "identity"]|array|1f6b6eb52c20bf63
5|true|bool|4026e0ec651f149c
6|["com.apple.safetyalerts"]|array|ac219fa5bf2a8945
7|["visualIdentifier", "entityAliasECR", "entitySubgraph", "peopleSubgraph", "inferenceFeaturesECR", "personEntityRelevanceRanking", "entitySimilarityFeatures"]|array|9c45f2534a0de12b
8|com.apple.ImagePlayground.InlineExtension|string|648c2550f00e14b4
9|["kTCCServicePhotos", "kTCCServiceAddressBook", "kTCCServiceCamera"]|array|328a00ac06a3e9e7
10|["com.apple.photos.ImageConversionService", "com.apple.assistant.cdm", "com.apple.modelmanager", "com.apple.modelcatalog.catalog", "com.apple.siri.uaf.service", "com.apple.mobileasset.autoasset", "com.apple.mobileassetd.v2", "com.apple.photoanalysisd", "com.apple.photos.service", "com.apple.mediaanalysisd.service.public", "com.apple.sage.summarization", "com.apple.generativeexperiences.summarization", "com.apple.generativeexperiences.availabilityService", "com.apple.feedbackd.centralized-feedback", "com.apple.extensionkit.service", "com.apple.intelligenceplatform.EntityResolution", "com.apple.intelligenceplatform.View", "com.apple.biome.access.user", "com.apple.duetactivityscheduler", "com.apple.ind.cloudfeatures", "com.apple.biome.access.user", "com.apple.stickers.recency", "com.apple.ciphermld"]|array|78611dfefbf6b62b
```

Effectively, do a query and check for where a given entitlement is used:

```
$ ipsw ent --sqlite entitlements.db --key com.apple.private.iomfb --limit 500
```

- Found 9 entitlements matching criteria

```
com.apple.private.iomfb.set-block-client-profile /usr/libexec/IOMFB_bics_daemon [27.0 24A435]
- iomfb_bics_daemon
com.apple.private.iomfb.set-block /usr/libexec/IOMFB_bics_daemon [27.0 24A435]
- true
com.apple.private.iomfb.set-block /usr/libexec/mobile_obliterator [27.0 24A435]
- true
com.apple.private.iomfb.set-block-client-profile /usr/libexec/thermalmonitord [27.0 24A435]
- thermal_monitor
com.apple.private.iomfb.set-block /usr/libexec/thermalmonitord [27.0 24A435]
- true
com.apple.private.iomfb.set-block-client-profile /usr/bin/IOMFB_FDR_Loader [27.0 24A435]
- iomfb_fdr_loader
com.apple.private.iomfb.set-block /usr/bin/IOMFB_FDR_Loader [27.0 24A435]
- true
com.apple.private.iomfb.set-block-client-profile /usr/libexec/backboardd [27.0 24A435]
- backboardd
com.apple.private.iomfb.set-block /usr/libexec/backboardd [27.0 24A435]
- true
```

The database above, search by the ipsw tool command, helps to answer which shipped binaries hold a given entitlement, which is very appropriate. As a supplement, it is possible to produce a script that shows another part of the equation, from the kernel side, which function references (and not necessarily tests) a determined entitlement (string), that can could be declared and checked by nobody. Therefore, we are taking a double-side approach and trying to understand what binaries carries a given entitlement (user application side) and which functions, from the kernel side requests such entitlement. Of course, this

verification is only from the entitlement point of view, and we are not considering other protections or mitigation that can be involved in context. The script follows below:

```
$ cat entitle.py
```

```
#!/usr/bin/env python3
"""entitle.py - kernel side entitlement check.

USAGE
    python entitle.py                every entitlement string, with its reference count
    python entitle.py orphans        ***** declared and referenced by NOBODY - the bug shape
    python entitle.py <substring>    entitlements matching, and who references each
    python entitle.py fn <name|0xaddr> which entitlements THIS function references
    python entitle.py check
"""

import os
import sys

sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))
import calib # noqa: E402
import kc # noqa: E402
import xref # noqa: E402

PREFIXES = ("com.apple.", "platform-application", "get-task-allow",
            "dynamic-codesigning", "task_for_pid")

MANGLED = [False]

def is_entitlement(s):
    if not s or len(s) < 8 or len(s) > 160:
        return False
    if " " in s or "%" in s or "/" in s:
        return False
    if not s.startswith(PREFIXES):
        return False
    return "." in s or "-" in s

def strings(im):
    if hasattr(im, "_ents"):
        return im._ents
    out = {}
    for pfx in PREFIXES:
        for va, s in im.find_cstrs(pfx, limit=20000):
            if is_entitlement(s) and s not in out:
                out[s] = va
    im._ents = out
    return out

def refs(im):
    if hasattr(im, "_entrefs"):
        return im._entrefs
    idx = xref.index(im)
    ents = strings(im)
    out = dict((e, []) for e in ents)
    for f, ss in idx["strefs"].items():
        for s in ss:
            if s in out:
                out[s].append(f)
    im._entrefs = out
    return out

def orphans(im):
    r = refs(im)
    return sorted([e for e, fs in r.items() if not fs])

def to_report(im, sel, cal, title):
```



```
rep = calib.Report("entitle", image=im.tag, target=title, cal=cal)
if not cal.calibrated():
    return rep
ents, r = strings(im), refs(im)
rep.fact("entitlement_strings", len(ents))
rep.fact("orphans", len(orphans(im)))
rep.fact("matched", len(sel))
rep.note("!" an entitlement with no reference is a policy that is not wired up; "
        "an absence of a reference is NOT proof that no check exists - it may live in a "
        "sandbox profile or be set through setProperty at runtime.")
for e in sel:
    fs = r.get(e, [])
    rep.row(entitlement=e, addr="0x%x" % ents[e], references=len(fs),
            referenced_by=[im.show(f)[:70] for f in fs[:8]],
            kexts=sorted(set((im.kext_of(f) or "com.apple.kernel") for f in fs))[:6]))
return rep

C_USED = "com.apple.private.amfi.can-load-trust-cache"
C_ORPHAN = "com.apple.private.amfi.can-check-trust-cache"
C_FAKE = "com.apple.private.this.cannot.exist.banana"

def selftest(tag=kc.DEFAULT):
    im = kc.load(tag, announce=False)
    cal = calib.Calibration("entitle.py", target=im.tag)
    ents, r = strings(im), refs(im)
    cal.pos("entitlement strings are found in quantity", len(ents) > 500,
            "%d strings" % len(ents))
    cal.pos("a known CHECKED entitlement is present and referenced",
            C_USED in r and len(r[C_USED]) >= 1,
            "%d reference(s)" % len(r.get(C_USED, [])))
    cal.truth("the recorded orphan entitlement is present and referenced by NOBODY",
            C_ORPHAN in ents and len(r.get(C_ORPHAN, [])) == 0,
            "present=%s refs=%d" % (C_ORPHAN in ents, len(r.get(C_ORPHAN, []))))
    cal.null("an entitlement that cannot exist is not found", C_FAKE not in ents)
    cal.neg("not everything is an orphan - most are referenced",
            len(orphans(im)) < len(ents),
            "%d orphans of %d" % (len(orphans(im)), len(ents)))
    return cal

calib.register("entitle", selftest)

def main():
    argv = sys.argv[1:]
    as_json = calib.wants_json(argv)
    if "--mangled" in argv:
        argv.remove("--mangled")
        MANGLED[0] = True
    im = kc.arg_image(argv)
    mode = argv.pop(0) if argv and argv[0] in ("check", "orphans", "fn") else None
    if mode == "check":
        c = selftest(im.tag)
        if as_json:
            calib.Report("entitle", image=im.tag, target="controls", cal=c).emit(True)
        else:
            c.report()
        return
    tok = argv.pop(0) if argv and mode == "fn" else (argv.pop(0) if argv and not mode else "")
    cal = selftest(im.tag)
    ents, r = strings(im), refs(im)

    if mode == "fn":
        a = xref._resolve(im, tok)
        if a is None:
            print("no such function"); return
        f = im.func_at(a)
        idx = xref.index(im)
        got = [s for s in idx["strefs"].get(f, []) if s in ents]
```

```
if as_json:
    rep = calib.Report("entitle", image=im.tag, target=tok, cal=cal)
    rep.fact("function", "0x%x" % f)
    rep.fact("name", im.show(f))
    for e in got:
        rep.row(entitlement=e)
    rep.emit(True)
    return
cal.require()
_n = im.show(f)
_d = _n if MANGLED[0] else kc.demangle(_n)
print("\n%s references %d entitlement(s):" % (_d, len(got)))
if _d != _n:
    print("    [%s]" % _n)
for e in got:
    print("    %s" % e)
return

if mode == "orphans":
    sel = orphans(im)
    title = "orphans"
else:
    sel = sorted(e for e in ents if tok.lower() in e.lower())
    title = tok

if as_json:
    to_report(im, sel, cal, title).emit(True)
    return
cal.require()
if mode == "orphans":
    print("\n***** %d ENTITLEMENT(S) DECLARED AND REFERENCED BY NOBODY, of %d total\n"
          % (len(sel), len(ents)))
    print("    A policy written down and not wired up is the missing authorization shape.")
    print("    ! An absence of a reference is NOT proof that no check exists - it may live in")
    print("    a sandbox profile, or be set through setProperty at runtime.\n")
    for e in sel[:80]:
        print("    0x%-14x %s" % (ents[e], e))
    if len(sel) > 80:
        print("    ... %d more" % (len(sel) - 80))
    return

print("\n%d entitlement(s) matching %r (of %d)\n" % (len(sel), tok, len(ents)))
for e in sel[:60]:
    fs = r.get(e, [])
    mark = "    ***** REFERENCED BY NOBODY" if not fs else ""
    print("    %-64s %3d ref(s)%s" % (e[:64], len(fs), mark))
    for f in fs[:4]:
        nm = im.show(f)
        if not MANGLED[0]:
            nm = kc.demangle(nm)
            cut = nm.find("(")
            if cut > 0:
                nm = nm[:cut]
        print("    %-56s [%s]" % (nm[:56], im.kext_of(f) or "com.apple.kernel"))

if __name__ == "__main__":
    main()
```

Testing the script against the provided kernelcache and entitlement, we have:

```
$ python tools\entitle.py C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin
com.apple.private.iomfb
```

```
IMAGE C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin (kc_27.0rc.bin) 197748 starts, 46460
symbols
CONTROLS entitle.py C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin CALIBRATED 5/5
```

2 entitlement(s) matching 'com.apple.private.iomfb' (of 2836)

```
com.apple.private.iomfb.set-block                                1 ref(s)
  IOMobileFramebufferUserClient::isSetBlockEnumAllowed
[com.apple.iokit.IOMobileGraphicsFamily]
  com.apple.private.iomfb.set-block-client-profile              1 ref(s)
  IOMobileFramebufferUserClient::detectSetBlockClientType
[com.apple.iokit.IOMobileGraphicsFamily]]
```

Based on the script, the `method isSetBlockEnumAllowed` of the class `IOMobileFramebufferUserClient`, in the text `com.apple.iokit.IOMobileGraphicsFamily`, tests for the `com.apple.private.iomfb.set-block` entitlement. Please, pay attention to the fact that the kernel side reports what tests an entitlement, but it does have one. Actually, the text does not carry entitlement in all, and only a process does.

We can invert the question and, given the function and the kernelcache, we could request what entitlement is required by the provided function:

```
$ python tools\entitle.py fn isSetBlockEnumAllow C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin
IMAGE C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin (kc_27.0rc.bin) 197748 starts, 46460
symbols
CONTROLS entitle.py C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin CALIBRATED 5/5

IOMobileFramebufferUserClient::isSetBlockEnumAllowed(task*, unsigned int, IOMFBClientType, unsigned
int) references 1 entitlement(s):
  [__ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej]
  com.apple.private.iomfb.set-block
```

As we know the function, we can print its disassembly by using another script, as shown below:

```
$ cat readfn.py

#!/usr/bin/env python3

"""readfn.py - prints a function
"""

import sys
import os

sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))
import calib # noqa: E402
import kc # noqa: E402

def resolve(im, va):
    n, trust = im.fname(va)
    if not n:
        return "sub_%x" % va
    if trust == "start":
        return n
    return "~%s" % n

def resolve_target(im, target):
    if target.lower().startswith("0x"):
        try:
            return int(target, 16), None
        except ValueError:
            return None, None
    va = im.addr_of(target)
    if va is not None:
        return va, None
    hits = im.find_syms(target, limit=12)
    starts = [(a, n) for a, n in hits if im.func_at(a) == a]
    pick = starts or hits
```

```
if len(pick) == 1:
    return pick[0][0], None
if len(pick) > 1:
    return None, pick
return None, None

def main():
    argv = sys.argv[1:]
    raw = "--raw" in argv
    calls_only = "--calls" in argv
    argv = [a for a in argv if not a.startswith("--")]
    if "--image" in sys.argv:
        im = kc.load(sys.argv[sys.argv.index("--image") + 1])
    else:
        im = kc.arg_image(argv)
    if not argv:
        print(__doc__)
        return 2

    va, amb = resolve_target(im, argv[0])
    if amb:
        print("%r matches %d symbols - name one exactly:" % (argv[0], len(amb)))
        for a, n in amb[:12]:
            print("    0x%-14x %s" % (a, n))
        return 2
    if va is None:
        print("no such symbol: %s" % argv[0])
        return 2

    start = im.func_at(va) or va
    end = im.next_start(start)
    off = im.f_off(start)
    blob = im.data[off:off + (end - start)]
    insns = list(im.disasm_all(blob, start))
    if getattr(im, "disasm_skipped", 0):
        print("! %d bytes were undecodable and skipped" % im.disasm_skipped)
    _raw = resolve(im, start)
    print("%s %s" % (_raw, im.kext_of(start) or "com.apple.kernel"))
    _bad = kc.demangle_check()
    if _bad:
        print("! %s" % _bad)
    else:
        _dm = kc.demangle(_raw.lstrip("~").split("+")[0])
        if _dm != _raw:
            print("    = %s" % _dm)
    print("0x%x .. 0x%x    %d bytes, %d instructions" % (start, end, end - start, len(insns)))
    print("-" * 78)

    for ins in insns:
        m, o = ins.mnemonic, ins.op_str
        note = ""
        if m in ("bl", "b", "blr", "braa", "braaz", "blraa", "blraaz") and o.startswith("#"):
            t = int(o[1:], 16)
            real = im.stub_target(t)
            note = "    -> %s" % resolve(im, real)
            if real != t:
                note += "    (via stub 0x%x)" % t
        elif m in ("adrp", "adr"):
            note = ""
        if calls_only and not note:
            continue
        if raw:
            print("0x%x %s %s%s" % (ins.address, m, o, note))
        else:
            print("0x%08x %-8s %-44s%s" % (ins.address, m, o, note))
    return 0
```

```
def selftest(tag=kc.DEFAULT):
    im = kc.load(tag, announce=False)
    cal = calib.Calibration("readfn.py", target=im.tag)

    named = sorted(a for a in im.STARTS if im.SYM.get(a))
    start = named[len(named) // 2] if named else None
    cal.pos("a named function START resolves to its plain symbol, with no trust marker",
            start is not None and resolve(im, start) == im.SYM[start],
            "0x%x -> %s" % (start or 0, resolve(im, start) if start else "NO NAMED START"))
    cal.neg("an address INSIDE that function is marked '~', so the trust level is not hidden",
            start is not None and resolve(im, start + 4).startswith("~"),
            resolve(im, start + 4) if start else "-")
    cal.null("an address with no symbol at all prints as sub_<addr>",
            resolve(im, 0x1000).startswith("sub_"))

    uniq = None
    for a in named:
        n = im.SYM[a]
        if len([1 for b, m in im.SYM.items() if n in m]) == 1 and len(n) > 12:
            uniq = (a, n)
            break
    cal.pos("an exact symbol resolves to its address",
            uniq is not None and resolve_target(im, uniq[1])[0] == uniq[0],
            "%s -> 0x%x" % (uniq[1][:40], uniq[0]) if uniq else "no unique symbol found")
    amb = resolve_target(im, "externalMethod")
    cal.neg("an AMBIGUOUS name is REFUSED, not guessed",
            amb[0] is None and bool(amb[1]),
            "%d candidate(s) returned instead of a pick" % len(amb[1] or []))
    cal.null("a name no symbol contains resolves to nothing, and does not raise",
            resolve_target(im, "NoSuchSymbolBanana") == (None, None))
    cal.pos("a hex address is taken verbatim",
            resolve_target(im, "0x%x" % (start or 4))[0] == (start or 4),
            "0x%x" % (start or 4))

    worse = None
    for a in named[:4000]:
        end = im.next_step(a) if hasattr(im, "next_step") else im.next_start(a)
        n = (end - a) // 4
        if n < 8 or n > 2000:
            continue
        off = im.f_off(a)
        if off is None:
            continue
        full = len(list(im.disasm_all(im.data[off:off + (end - a)], a)))
        if full > len(im.body(a, cap=end - a)):
            worse = (a, len(im.body(a, cap=end - a)), full)
            break
    cal.truth("the resumable read is compared against the stopping one on this image",
            True,
            ("0x%x: body() %d instruction(s), disasm_all %d - the difference is why this "
             "tool uses the latter" % worse) if worse
            else "no function on this image carries an undecodable word in the sample")
    return cal

calib.register("readfn", selftest)

if __name__ == "__main__":
    sys.exit(main())
```

As an example, running the script with one of found functions we have:

```
$ python .\tools\readfn.py C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin
_ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej | head -20
```

```
IMAGE C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin (kc_27.0rc.bin) 197748 starts, 46460
symbols
```

```
__ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej
com.apple.iokit.IOMobileGraphicsFamily
= IOMobileFramebufferUserClient::isSetBlockEnumAllowed(task*, unsigned int, IOMFBClientType,
unsigned int)
```

0xffffffff00a17e8dc .. 0xffffffff00a17e9f0 276 bytes, 69 instructions

```
-----
0xffffffff00a17e8dc bti      c
0xffffffff00a17e8e0 pacibsp
0xffffffff00a17e8e4 sub      sp, sp, #0x40
0xffffffff00a17e8e8 stp      x29, x30, [sp, #0x30]
0xffffffff00a17e8ec add      x29, sp, #0x30
0xffffffff00a17e8f0 adrp     x8, #0xffffffff00b974000
0xffffffff00a17e8f4 ldrb     w8, [x8, #0xa88]
0xffffffff00a17e8f8 tbz      w8, #0, #0xffffffff00a17e904
0xffffffff00a17e8fc mov      w0, #1
0xffffffff00a17e900 b        #0xffffffff00a17e9b8      ->
~__ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej+0xdc
0xffffffff00a17e904 cmp      w2, #2
0xffffffff00a17e908 b.eq     #0xffffffff00a17e9a4
0xffffffff00a17e90c cmp      w2, #1
0xffffffff00a17e910 b.ne     #0xffffffff00a17e974
0xffffffff00a17e914 adrp     x8, #0xffffffff0082cd000
0xffffffff00a17e918 add      x8, x8, #0xe40
```

It is possible to provide the address and obtain the same result:

```
$ python .\tools\readfn.py C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin 0xffffffff00a17e8dc |
head -20
```

```
IMAGE C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.bin (kc_27.0rc.bin) 197748 starts, 46460
symbols
```

```
__ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej
com.apple.iokit.IOMobileGraphicsFamily
= IOMobileFramebufferUserClient::isSetBlockEnumAllowed(task*, unsigned int, IOMFBClientType,
unsigned int)
```

0xffffffff00a17e8dc .. 0xffffffff00a17e9f0 276 bytes, 69 instructions

```
-----
0xffffffff00a17e8dc bti      c
0xffffffff00a17e8e0 pacibsp
0xffffffff00a17e8e4 sub      sp, sp, #0x40
0xffffffff00a17e8e8 stp      x29, x30, [sp, #0x30]
0xffffffff00a17e8ec add      x29, sp, #0x30
0xffffffff00a17e8f0 adrp     x8, #0xffffffff00b974000
0xffffffff00a17e8f4 ldrb     w8, [x8, #0xa88]
0xffffffff00a17e8f8 tbz      w8, #0, #0xffffffff00a17e904
0xffffffff00a17e8fc mov      w0, #1
0xffffffff00a17e900 b        #0xffffffff00a17e9b8      ->
~__ZN29IOMobileFramebufferUserClient21isSetBlockEnumAllowedEP4taskj15IOMFBClientTypej+0xdc
0xffffffff00a17e904 cmp      w2, #2
0xffffffff00a17e908 b.eq     #0xffffffff00a17e9a4
0xffffffff00a17e90c cmp      w2, #1
0xffffffff00a17e910 b.ne     #0xffffffff00a17e974
0xffffffff00a17e914 adrp     x8, #0xffffffff0082cd000
0xffffffff00a17e918 add      x8, x8, #0xe40
```

Considering tools and scripts so far, we have:

- [ipsw tool](#): it can check what entitlement an application holds.
- [entitle.py](#): it can check if which function holds a reference for a given entitlement.
- [readfn.py](#): it disassembles the provided function and enables to check, by interpreting the code, if the entitlement is really being checked (required) or not.

Next step is to create an IDA database of the kernelcache. While doing this task graphically, via IDA Pro GUI, it certainly works well, in the daily research work we need to be faster, and using command line to such a work is the natural way to go.

So far we have two artifacts, which are the own kernelcache (kc_27.0rc.bin) and its respective symbols, given by the [kc_27.0rc.bin.symbols.json](#) file. To build the IDA database, you can use IDA Pro 9.4 or IDA Pro 8.4, and as multiple researchers have the latter, I will be showing steps for both versions. Another important aspect of this work is where it is going to be done. No doubt, it could be done on macOS system (running on M2 or M5, in my case), but as it is heavy work, I always perform it out of macOS, even because the machine where Windows is running has far more hardware resources. Therefore, next steps will be to use my directory locations as reference, and you can adapt it to your configuration.

The first 9 steps of the procedure must be done with IDA Pro 9.4, as shown below:

- **Step 01:** Open a PowerShell window. Although it is possible to use Command prompt, I had some issues and I recommend avoiding using it.
- **Step 02:** Set up the two necessary paths:
 - `$ida = "C:\Program Files\IDA Professional 9.4\idat.exe"`
 - `$base = "C:\research_ios\kernelcaches\ios27rc"`
- **Step 03:** Confirm that kernelcache and build_idb.py script (it will be presented after this procedure) are placed in the same folder. Additionally, you should notice that there are location variables to be setup inside of the script:
 - `Get-ChildItem "$base\kc_27.0rc.bin", "$base\build_idb.py"`
- **Step 04:** As issues due to plugin update can come up over time, the exact versions for both Cellebrite plugins are listed below:
 - `git -C C:\IDAScripts\ida_kernelcache log -1 --format=%h # 81ca5fc`
 - `git -C C:\IDAScripts\ida_kcpp log -1 --format=%h # acab27f`
 - (only if it is needed) `git -C C:\IDAScripts\ida_kernelcache checkout 81ca5fc`
 - (only if it is needed) `git -C C:\IDAScripts\ida_kcpp checkout acab27f`
- **Step 05:** Create C:\tmp directory because ida_kcpp plugin installs a log handler on the literal path "/tmp/ida_kcpp.log":
 - `if (-not (Test-Path 'C:\tmp')) { New-Item -ItemType Directory 'C:\tmp' }`
- **Step 06:** Delete any database that already exists:
 - `Remove-Item "$base\kc_27.0rc.bin.id4" -ErrorAction SilentlyContinue`
- **Step 07:** Run the script to build the database (it takes something between 20 and 30 minutes), and there are important observations such as `-A` option means batch mode with no dialogs, `-S<script>` option runs the script after the load finishes, and `-L<file>` option captures the log. Take care: there is **no** space after `-S` or `-L`, and space makes IDA treat the path as a second input file:
 - `& $ida -A "-S$base\build_idb.py" "-L$base\ida94.log" "$base\kc_27.0rc.bin"`

- **Step 08:** Open a second PowerShell window to follow the task being executed on step 05:

```

▪ $base = "C:\research_ios\kernelcaches\ios27rc"
▪ Get-Content "$base\build_idb_progress.txt" -Wait

[ 0.0s] python 3.13.7, IDA 9.4
[ 0.0s] waiting for auto-analysis
[ 524.9s] auto-analysis complete
[ 526.4s] imported 29104 ipsw symbols
[ 743.4s] ida_kernelcache (81ca5fc) complete
[ 898.1s] kcopp init_if_needed done
[ 898.1s] kcopp sync: ida_kcopp: fixing classes vtables (0/3311)
[ 907.8s] kcopp sync: ida_kcopp: fixing classes vtables (250/3311)
...
[ 1758.5s] kcopp sync: ida_kcopp: fixing classes vtables (3250/3311)
[ 1767.1s] ida_kcopp (acab27f) complete - 3312 sync callbacks
[ 1768.2s] named functions: 69564
[ 1819.8s] 346345 TOTAL symbols exported to
C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.symbols.exported.json
[ 1819.8s] DONE

```

- **Step 09:** Confirm the output:

```

▪ Get-ChildItem "$base\kc_27.0rc.bin.i64", "$base\kc_27.0rc.symbols.exported.json"

```

Directory: C:\research_ios\kernelcaches\ios27rc

Mode	LastWriteTime	Length	Name
-a----	9/16/2026 2:14 PM	1044047658	kc_27.0rc.bin.i64
-a----	9/16/2026 2:14 PM	17913684	kc_27.0rc.symbols.exported.json

The next steps are equivalent tasks for IDA Pro 8.4:

- **Step 10:** As ida_kcopp writes a hardcoded log path `C:\tmp\ida_kcopp.log` on Windows, so `C:\tmp` folder must exist:
 - `if (-not (Test-Path 'C:\tmp')) { New-Item -ItemType Directory 'C:\tmp' }`
- **Step 11:** Confirm that `ida_kernelcache` version matches with presented in this article. Actually, a checkout left on the wrong commit results in silent failure. It is important to mention that I keep different copies of `ida_kernelcache plugin` (`C:\IDAScripts\ida_kernelcache` and `C:\IDAScripts\ida_kernelcache_84` folders) to avoid going back to an exact version:
 - `git -C C:\IDAScripts\ida_kernelcache_84 log -1 --format=%h # 2f06467`
 - (only if it is needed) `git -C C:\IDAScripts\ida_kernelcache_84 checkout 2f06467`
- **Step 12:** Copy the binary to work with an independent copy of kernelcache for IDA Pro 8.4:
 - `$base = "C:\research_ios\kernelcaches\ios27rc"`
 - `Copy-Item "$base\kc_27.0rc.bin" "$base\kc_27.0rc.ida84.bin"`
 - `(Get-FileHash "$base\kc_27.0rc.bin").Hash -eq (Get-FileHash "$base\kc_27.0rc.ida84.bin").Hash`
- **Step 13:** Set the correct IDA Pro 8.4 executable, as shown below:
 - `$ida = "C:\Program Files\IDA Pro 8.4\idat64.exe"`
- **Step 14:** Delete any existing IDA Pro 8.4 database:

- `Remove-Item "$base\kc_27.0rc.ida84.bin.i64" -ErrorAction SilentlyContinue`
- **Step 15:** Run the script to build the database following the same rules from Step 07:
 - `$ida -A "-S$base\build_ida84.py" "-L$base\ida84.log" "$base\kc_27.0rc.ida84.bin"`
- **Step 16:** Open a second PowerShell window to follow the task being executed on step 15:
 - `Get-Content "$base\build_ida84_progress.txt" -Wait`

```
[ 0.0s] python 3.13.7, IDA 8.4
[ 0.0s] waiting for auto-analysis
[ 757.5s] auto-analysis complete
[ 760.0s] imported 35768 of 46460 ipsw symbols
[ 1025.0s] ida_kernelcache (2f06467) complete
[ 1246.9s] kcpp init_if_needed done
[ 1246.9s] kcpp sync: ida_kcpp: fixing classes vtables (0/3310)
[ 1247.0s] kcpp sync: ida_kcpp: fixing classes vtables (250/3310)
...
[ 1231.7s] kcpp sync: ida_kcpp: fixing classes vtables (3250/3310)
[ 1231.8s] ida_kcpp (acab27f) complete on IDA 8.4 - 3311 sync callbacks
[ 1233.2s] named functions: 36737
[ 1284.8s] 310698 TOTAL symbols exported to
C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.symbols.ida84.json
[ 1284.8s] DONE
```

- **Step 17:** Confirm the output:
 - `Get-ChildItem "$base\kc_27.0rc.ida84.bin.i64", "$base\kc_27.0rc.symbols.ida84.json"`

Directory: C:\research_ios\kernelcaches\ios27rc

Mode	LastWriteTime	Length	Name
-a----	9/16/2026 4:07 PM	1078597625	kc_27.0rc.ida84.bin.i64
-a----	9/16/2026 4:06 PM	15529278	kc_27.0rc.symbols.ida84.json

Both `build_idb.py` (for IDA Pro 9.4) and `build_ida84.py` (for IDA Pro 8.4) that were used in these procedures, and that can be improved. Once again, remember to change the kernelcache directory within them for repeated operations.

First, the `build_idb.py` script, which has been used for IDA Pro 9.4:

`build_idb.py`

```
import json
import os
import sys
import time
import ida_name
import idaapi
import idautils
import idc

try:
    BASE = os.path.dirname(os.path.abspath(__file__))
except NameError:
    BASE = r"C:\research_ios\kernelcaches\ios27rc"

IPSW_SYMS = BASE + r"\kc_27.0rc.bin.symbols.json"
KCPLUGIN = r"C:\IDAScripts\ida_kernelcache"
KCPP = r"C:\IDAScripts\ida_kcpp"
OUT = BASE + r"\kc_27.0rc.symbols.exported.json"
LOG = BASE + r"\build_idb_progress.txt"
```

```
T0 = time.time()

def note(msg):
    line = "[%7.1fs] %s" % (time.time() - T0, msg)
    print(line)
    try:
        with open(LOG, "a") as f:
            f.write(line + "\n")
    except Exception:
        pass

note("python %s, IDA %s" % (sys.version.split()[0], idaapi.get_kernel_version()))
note("waiting for auto-analysis")
idaapi.auto_wait()
note("auto-analysis complete")

imported = 0
try:
    with open(IPSW_SYMS) as f:
        raw = json.load(f)
    for addr_str, name in raw.items():
        if name == "<unknown>" or not addr_str.isdigit():
            continue
        addr = int(addr_str)
        current = idc.get_func_name(addr) or idc.get_name(addr, ida_name.GN_VISIBLE)
        if current and not current.startswith("sub_") and not current.startswith("loc_"):
            continue
        clean = name.replace(" ", "_").replace(":", "__")
        if ida_name.set_name(addr, clean, ida_name.SN_NOWARN | ida_name.SN_NOCHECK):
            imported += 1
        note("imported %d ipsw symbols" % imported)
except Exception as e:
    note("ipsw import FAILED: %s" % e)

sys.path.insert(0, KCPLUGIN)
try:
    import ida_kernelcache

    ida_kernelcache.kernelcache_process()
    note("ida_kernelcache (81ca5fc) complete")
except Exception as e:
    import traceback

    note("ida_kernelcache FAILED: %s: %s" % (type(e).__name__, e))
    for ln in traceback.format_exc().strip().split("\n")[-6:]:
        note("    " + ln.strip())

# ---- ida_kcpp -----

sys.path.insert(0, KCPP)
_ksync = [0]

def _cb(msg):
    _ksync[0] += 1
    if _ksync[0] % 250 == 1:
        note("    kcpp sync: %s" % msg)
try:
    from ida_kcpp import logic

    logic.init_if_needed(only_collect=False)
    note("kcpp init_if_needed done")
    logic.perform_initial_sync(_cb)
    note("ida_kcpp (acab27f) complete - %d sync callbacks" % _ksync[0])
except Exception as e:
    import traceback
```

```
note("ida_kcpp FAILED: %s: %s" % (type(e).__name__, e))
for ln in traceback.format_exc().strip().split("\n")[-6:]:
    note("    " + ln.strip())

symbols = {}
for ea in idautils.Functions():
    name = idc.get_func_name(ea)
    if name and not name.startswith("sub_"):
        symbols[str(ea)] = name
note("named functions: %d" % len(symbols))

for seg_ea in idautils.Segments():
    seg = idaapi.getseg(seg_ea)
    if not seg:
        continue
    ea = seg.start_ea
    while ea < seg.end_ea and ea != idaapi.BADADDR:
        name = idc.get_name(ea, ida_name.GN_VISIBLE)
        if name and not name.startswith("sub_") and not name.startswith("loc_"):
            symbols.setdefault(str(ea), name)
        nxt = idc.next_head(ea, seg.end_ea)
        if nxt == idaapi.BADADDR or nxt <= ea:
            break
        ea = nxt

with open(OUT, "w") as f:
    json.dump(symbols, f)
note("%d TOTAL symbols exported to %s" % (len(symbols), OUT))
note("DONE")
idc.qexit(0)
```

Now, the **build_ida84.py** script, which has been used for IDA Pro 8.4:

```
# build_ida84.py

import json
import os
import sys
import time
import ida_name
import idaapi
import idautils
import idc

try:
    BASE = os.path.dirname(os.path.abspath(__file__))
except NameError:
    BASE = r"C:\research_ios\kernelcaches\ios27rc"
IPSW_SYMS = BASE + r"\kc_27.0rc.bin.symbols.json"
KC84 = r"C:\IDAScripts\ida_kernelcache_84"
KCPP = r"C:\IDAScripts\ida_kcpp"
OUT = BASE + r"\kc_27.0rc.symbols.ida84.json"
LOG = BASE + r"\build_ida84_progress.txt"

T0 = time.time()

def note(msg):
    line = "[%7.1fs] %s" % (time.time() - T0, msg)
    print(line)
    try:
        with open(LOG, "a") as f:
            f.write(line + "\n")
    except Exception:
        pass

note("python %s, IDA %s" % (sys.version.split()[0], idaapi.get_kernel_version()))
note("waiting for auto-analysis")
```

```
idaapi.auto_wait()
note("auto-analysis complete")

imported = 0
try:
    with open(IPSW_SYMS) as f:
        raw = json.load(f)
        for addr_str, name in raw.items():
            if name == "<unknown>" or not addr_str.isdigit():
                continue
            addr = int(addr_str)
            current = idc.get_func_name(addr) or idc.get_name(addr, ida_name.GN_VISIBLE)
            if current and not current.startswith("sub_") and not current.startswith("loc_"):
                continue
            clean = name.replace(" ", "_").replace(":", "__")
            if ida_name.set_name(addr, clean, ida_name.SN_NOWARN | ida_name.SN_NOCHECK):
                imported += 1
        note("imported %d of %d ipsw symbols" % (imported, len(raw)))
except Exception as e:
    note("ipsw import FAILED: %s" % e)

sys.path.insert(0, KC84)
try:
    import ida_kernelcache

    ida_kernelcache.kernelcache_process()
    note("ida_kernelcache (2f06467) complete")
except Exception as e:
    import traceback

    note("ida_kernelcache FAILED: %s: %s" % (type(e).__name__, e))
    for ln in traceback.format_exc().strip().split("\n")[-6:]:
        note("    " + ln.strip())

sys.path.insert(0, KCPP)
_ensure = [0]

def _cb(msg):
    _ensure[0] += 1
    if _ensure[0] % 250 == 1:
        note("    kcpp sync: %s" % msg)

try:
    from ida_kcpp import logic

    logic.init_if_needed(only_collect=False)
    note("kcpp init_if_needed done")
    logic.perform_initial_sync(_cb)
    note("ida_kcpp (acab27f) complete on IDA 8.4 - %d sync callbacks" % _ensure[0])
except Exception as e:
    import traceback

    note("ida_kcpp FAILED ON 8.4: %s: %s" % (type(e).__name__, e))
    note("***** if this is an API error, check out bb23a96 into C:\\IDAScripts\\ida_kcpp_84")
    for ln in traceback.format_exc().strip().split("\n")[-8:]:
        note("    " + ln.strip())

symbols = {}
for ea in idutils.Functions():
    name = idc.get_func_name(ea)
    if name and not name.startswith("sub_"):
        symbols[str(ea)] = name
note("named functions: %d" % len(symbols))

for seg_ea in idutils.Segments():
    seg = idaapi.getseg(seg_ea)
    if not seg:
```

```

        continue
    ea = seg.start_ea
    while ea < seg.end_ea and ea != idaapi.BADADDR:
        name = idc.get_name(ea, ida_name.GN_VISIBLE)
        if name and not name.startswith("sub_") and not name.startswith("loc_"):
            symbols.setdefault(str(ea), name)
        nxt = idc.next_head(ea, seg.end_ea)
        if nxt == idaapi.BADADDR or nxt <= ea:
            break
        ea = nxt

with open(OUT, "w") as f:
    json.dump(symbols, f)
note("%d TOTAL symbols exported to %s" % (len(symbols), OUT))
note("DONE")
idc.qexit(0)

```

In every procedure so far, we have generated directly or indirectly a symbol file and merging the existing ones will be useful for us. Before proceeding, it is necessary to confirm whether symbols files are present:

```
$ Get-ChildItem -Path "$base\kc_27.0rc.bin.symbols.json", "$base\kc_27.0rc.symbols.exported.json",
"$base\kc_27.0rc.symbols.ida84.json"
```

Mode	LastWriteTime	Length	Name
-a----	9/13/2026 10:02 PM	3528625	kc_27.0rc.bin.symbols.json
-a----	9/16/2026 2:14 PM	17913684	kc_27.0rc.symbols.exported.json
-a----	9/16/2026 4:06 PM	15529278	kc_27.0rc.symbols.ida84.json

I will use a developed script named `merge_symbols.py`, which will make the merge appropriately, and as you have already realized, file names have been setup according to the kernelcache files:

```

#!/usr/bin/env python3
r"""
    usage: python merge_symbols.py [--write]
"""
import json
import os
import re
import sys

BASE = os.path.dirname(os.path.abspath(__file__))

IPSW = os.path.join(BASE, "kc_27.0rc.bin.symbols.json")
IDA94 = os.path.join(BASE, "kc_27.0rc.symbols.exported.json")
IDA84 = os.path.join(BASE, "kc_27.0rc.symbols.ida84.json")
OUT = os.path.join(BASE, "kc_27.0rc.symbols.MERGED.json")

REQUIRED = ("ipsw", "ida94", "ida84")

DATA_LABEL = re.compile(r"^(off_|qword_|dword_|byte_|word_|unk_|asc_|def_|jpt_|stru_)")
CODE_LABEL = re.compile(r"^(sub_|loc_|locret_|nullsub_|j_)")

def load(p):
    if not os.path.exists(p):
        print(" ! ABSENT, THIS LEG CONTRIBUTES NOTHING: %s" % os.path.basename(p))
        return {}
    with open(p) as f:
        return json.load(f)

def refuse_anchor_file():
    found = sorted(f for f in os.listdir(BASE)
                   if f.startswith("anchored_") and f.endswith(".json"))
    if found:
        print("")

```

```
print("***** REFUSING TO MERGE - an anchor file is present and this script no longer has")
print("      a tier that consumes one. Step 4b is retired.")
for f in found:
    print("          %s" % f)
print("      Move it aside to merge without it, or restore the anchor tier deliberately")
print("      - and if you restore it, it goes FIRST, because an anchor must never")
print("      override a real symbol.")
sys.exit(2)

def main():
    refuse_anchor_file()
    ipsw, ida94, ida84 = load(IPSW), load(IDA94), load(IDA84)
    print("inputs: ipsw %d  ida9.4 %d  ida8.4 %d"
          % (len(ipsw), len(ida94), len(ida84)))

    present = [n for n, d in zip(REQUIRED, (ipsw, ida94, ida84)) if d]
    print("legs:      %d of %d contributed - %s" % (len(present), len(REQUIRED), ", ".join(present)))
    if len(present) < len(REQUIRED):
        missing = [n for n in REQUIRED if n not in present]
        print(" ! ***** THIS IS A SHORT MERGE. Missing: %s. That is legitimate when the "
              "secondary" % ", ".join(missing))
        print("      IDA pass was not run - the shipping build is that case - but the output is NOT")
        print("      comparable with a figure recorded from a full merge.")

    merged = {}
    dropped = 0
    for src in (ipsw, ida94, ida84):
        for addr, name in src.items():
            if not name or name == "<unknown>":
                continue
            if DATA_LABEL.match(name):
                dropped += 1
                continue
            merged[addr] = name

    real = sum(1 for v in merged.values() if not CODE_LABEL.match(v))
    print("dropped %d IDA data label writes (RULE 1)" % dropped)
    print("merged:  %d entries, %d real, %d code placeholders"
          % (len(merged), real, len(merged) - real))

    wins = {"ida84 over ida94": 0, "ida84 over ipsw": 0, "ida94 over ipsw": 0}
    for a, n in ipsw.items():
        if n == "<unknown>":
            continue
        f = merged.get(a)
        if f is None or f == n:
            continue
        if a in ida84 and ida84[a] == f:
            wins["ida84 over ipsw"] += 1
        elif a in ida94 and ida94[a] == f:
            wins["ida94 over ipsw"] += 1
    for a, n in ida94.items():
        f = merged.get(a)
        if f is not None and f != n and a in ida84 and ida84[a] == f:
            wins["ida84 over ida94"] += 1
    for k, v in wins.items():
        print(" %-20s %7d" % (k, v))

    if "--write" in sys.argv:
        with open(OUT, "w") as f:
            json.dump(merged, f)
        print("wrote %s (%d bytes)" % (OUT, os.path.getsize(OUT)))
    else:
        print("(dry run - pass --write to produce %s)" % os.path.basename(OUT))

if __name__ == "__main__":
    main()
```


Running the script against our files we have:

```
$ $base="C:\research_ios\kernelcaches\ios27rc"
$ python C:\research_ios\kernelcaches\ios27rc\merge_symbols.py --write

inputs: ipsw 46460   ida9.4 346345   ida8.4 310698
legs:    3 of 3 contributed - ipsw, ida94, ida84
dropped 143104 IDA data label writes (RULE 1)
merged:  286569 entries, 281247 real, 5322 code placeholders
         ida84 over ida94      15922
         ida84 over ipsw       8109
         ida94 over ipsw       295
wrote C:\research_ios\kernelcaches\ios27rc\kc_27.0rc.symbols.MERGED.json (15370482 bytes)
```

```
$ Get-Content kc_27.0rc.symbols.MERGED.json | jq | head -15
{
  "18446744005122261016": "mach_trap_table",
  "18446744005122463664": "bsd_syscall_table",
  "18446744005122478736": "vtable_for_OSMetaClass__MetaClass",
  "18446744005122478872": "vtable_for_OSMetaClass",
  "18446744005122479008": "vtable_for_OSObject__MetaClass",
  "18446744005122479152": "vtable_for_OSArray",
  "18446744005122479488": "vtable_for_OSArray__MetaClass",
  "18446744005122479632": "vtable_for_OSBoolean",
  "18446744005122479808": "vtable_for_OSBoolean__MetaClass",
  "18446744005122479952": "vtable_for_OSCollection__MetaClass",
  "18446744005122480104": "vtable_for_OSCollectionIterator",
  "18446744005122480280": "vtable_for_OSCollectionIterator__MetaClass",
  "18446744005122480424": "vtable_for_OSData",
  "18446744005122480728": "vtable_for_OSData__MetaClass",
```

```
$ Get-ChildItem -Path "$base\kc_27.0rc*.*", "$base\merge_symbols.py"
```

Mode	LastWriteTime	Length	Name
-a----	9/13/2026 9:59 PM	77987840	kc_27.0rc.bin
-a----	9/16/2026 2:14 PM	1044047658	kc_27.0rc.bin.i64
-a----	9/13/2026 10:02 PM	3528625	kc_27.0rc.bin.symbols.json
-a----	9/13/2026 9:59 PM	77987840	kc_27.0rc.ida84.bin
-a----	9/16/2026 4:07 PM	1078597625	kc_27.0rc.ida84.bin.i64
-a----	9/16/2026 2:14 PM	17913684	kc_27.0rc.symbols.exported.json
-a----	9/16/2026 4:06 PM	15529278	kc_27.0rc.symbols.ida84.json
-a----	9/16/2026 8:41 PM	15370482	kc_27.0rc.symbols.MERGED.json
-a----	9/16/2026 8:33 PM	6983	merge_symbols.py

During daily iOS research, you will be using resources such as ipsw tool and scripts to accelerate any investigation, and a few examples follow below. For example, obtaining the disassembled code at a virtual address can be done by executing:

```
$ ipsw macho disass --fileset-entry com.apple.kernel --count 20 --vaddr 0xffffffff00ad95488
kc_27.0rc.bin
```

```
• Loaded 1 symbols from cache file: kc_27.0rc.bin.a2s
sub_ffffffff00ad95488:
0xffffffff00ad95488: 7f 23 03 d5   pacibsp
0xffffffff00ad9548c: ff 03 02 d1   sub sp, sp, #0x80
0xffffffff00ad95490: fc 6f 02 a9   stp x28, x27, [sp, #0x20]
0xffffffff00ad95494: fa 67 03 a9   stp x26, x25, [sp, #0x30]
0xffffffff00ad95498: f8 5f 04 a9   stp x24, x23, [sp, #0x40]
0xffffffff00ad9549c: f6 57 05 a9   stp x22, x21, [sp, #0x50]
0xffffffff00ad954a0: f4 4f 06 a9   stp x20, x19, [sp, #0x60]
0xffffffff00ad954a4: fd 7b 07 a9   stp fp, lr, [sp, #0x70]
0xffffffff00ad954a8: fd c3 01 91   add fp, sp, #0x70
0xffffffff00ad954ac: f3 03 00 aa   mov x19, x0
0xffffffff00ad954b0: 08 1c 40 f9   ldr x8, [x0, #0x38]
0xffffffff00ad954b4: e8 03 00 f9   str x8, [sp]
```

```
0xffffffff00ad954b8: 08 5c 41 b9 ldr w8, [x0, #0x15c]
0xffffffff00ad954bc: e8 0b 00 b9 str w8, [sp, #0x8]
0xffffffff00ad954c0: 14 b0 40 f9 ldr x20, [x0, #0x160]
0xffffffff00ad954c4: 1a 14 40 b9 ldr w26, [x0, #0x14]
0xffffffff00ad954c8: 1b 50 41 b9 ldr w27, [x0, #0x150]
0xffffffff00ad954cc: 9a 00 08 37 tbzn w26, #0x1, loc_ffffffff00ad954dc ; ↪ 0x10
0xffffffff00ad954d0: ff 0b 00 f9 str xzr, [sp, #0x10]
0xffffffff00ad954d4: f4 03 00 b5 cbnz x20, loc_ffffffff00ad95550 ; ↪ 0x7c
```

To lookup a symbol address, execute:

```
$ ipsw kernel symbolicate --lookup 0xffffffff00ad95488 kc_27.0rc.bin.symbols.json
```

- Looking up symbol

```
0xffffffff00ad95488 namei
```

There are other small and bare tools (all of them use Capstone engine), which could be used for common daily operations. The first tool is named [kdis.py](#) (kernel disassemble), which offers quick disassembled code on command line:

```
$ $KC = "C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin"
$ python kdis.py $KC 0xffffffff00ad95488 12
```

```
0xffffffff00ad95488 pacibsp
0xffffffff00ad9548c sub     sp, sp, #0x80
0xffffffff00ad95490 stp     x28, x27, [sp, #0x20]
0xffffffff00ad95494 stp     x26, x25, [sp, #0x30]
0xffffffff00ad95498 stp     x24, x23, [sp, #0x40]
0xffffffff00ad9549c stp     x22, x21, [sp, #0x50]
0xffffffff00ad954a0 stp     x20, x19, [sp, #0x60]
0xffffffff00ad954a4 stp     x29, x30, [sp, #0x70]
0xffffffff00ad954a8 add     x29, sp, #0x70
0xffffffff00ad954ac mov     x19, x0
0xffffffff00ad954b0 ldr     x8, [x0, #0x38]
0xffffffff00ad954b4 str     x8, [sp]
```

The [kdis.py](#) script code follows:

```
#!/usr/bin/env python3
```

```
""" kdis.py: Disassemble a kernelcache vaddr range, and rank BL targets.
USAGE:
    dis.py <kc> <vaddr> [count]      disassemble count instructions at vaddr
    dis.py <kc> rank [n]              the n most called BL targets in __TEXT_EXEC
"""
```

```
import struct
import sys
```

```
def _argguard():
    import sys as _s
    if len(_s.argv) > 1:
        return
    d = __doc__ or ""
    i = d.find("INVOCATION")
    if i < 0:
        i = d.find("invocation")
    _s.stderr.write("%s needs arguments; it has no defined bare run.\n\n" % _s.argv[0])
    _s.stderr.write(d[i:i + 700] if i >= 0 else
        "No INVOCATION block in this file's docstring - that is the bug.\n")
    _s.stderr.write("\n")
    _s.exit(2)
```

```
_argguard()

LC_SEGMENT_64 = 0x19

def segments(data):
    ncmts = struct.unpack_from("<I", data, 16)[0]
    segs, p = [], 32
    for _ in range(ncmts):
        cmd, cmdsize = struct.unpack_from("<2I", data, p)
        if cmd == LC_SEGMENT_64:
            name = data[p + 8:p + 24].split(b"\x00")[0].decode()
            vmaddr, vmsize, fileoff, filesize = struct.unpack_from("<4Q", data, p + 24)
            segs.append((name, vmaddr, fileoff, filesize))
            p += cmdsize
    return segs

def f_off(segs, va):
    for _, vm, fo, fs in segs:
        if fs and vm <= va < vm + fs:
            return fo + (va - vm)
    return None

def rank(data, segs, n):
    from collections import Counter
    c = Counter()
    for name, vm, fo, fs in segs:
        if "TEXT" not in name or not fs:
            continue
        cnt = fs // 4
        ws = struct.unpack_from("<%dI" % cnt, data, fo)
        for i in range(cnt):
            w = ws[i]
            if (w & 0xFC000000) != 0x94000000:
                continue
            imm = w & 0x03FFFFFF
            if imm & (1 << 25):
                imm -= (1 << 26)
            c[vm + i * 4 + imm * 4] += 1
    return c.most_common(n)

def main():
    kc = sys.argv[1]
    data = open(kc, "rb").read()
    segs = segments(data)

    if sys.argv[2] == "rank":
        n = int(sys.argv[3]) if len(sys.argv) > 3 else 40
        for va, k in rank(data, segs, n):
            print(" 0x%-14x %6d callers" % (va, k))
        return

    va = int(sys.argv[2], 0)
    count = int(sys.argv[3], 0) if len(sys.argv) > 3 else 40
    off = f_off(segs, va)
    if off is None:
        print("vaddr not mapped")
        return
    from capstone import Cs, CS_ARCH_ARM64, CS_MODE_LITTLE_ENDIAN
    md = Cs(CS_ARCH_ARM64, CS_MODE_LITTLE_ENDIAN)
    for ins in md.disasm(data[off:off + count * 4], va):
        print("0x%-12x %-8s %s" % (ins.address, ins.mnemonic, ins.op_str))

main()
```

The second tool is [blxref.py](#), which is used to discover who branches to a specific address:

```
$ $KC = "C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin"
```

```
$ python blxref.py $KC 0xffffffff00ad95488 | head -10
```

```
target 0xffffffff00ad95488 : 80 branch site(s)
BL 0xffffffff00ad788b0
BL 0xffffffff00ad791fc
BL 0xffffffff00ad79a94
BL 0xffffffff00ad7a764
BL 0xffffffff00ada1a7c
BL 0xffffffff00ada90f8
BL 0xffffffff00adad324
BL 0xffffffff00adad94c
BL 0xffffffff00adaed74
```

The [blxref.py](#) tool code follows:

```
#!/usr/bin/env python3
```

```
"""blxref.py - every BL / B that branches to a given target address.
```

```
USAGE:
```

```
    blxref.py <kc> <target_vaddr> [--seg __TEXT_EXEC]
"""
```

```
import struct
import sys
```

```
def _argguard():
```

```
    import sys as _s
    if len(_s.argv) > 1:
        return
    d = __doc__ or ""
    i = d.find("INVOCATION")
    if i < 0:
        i = d.find("invocation")
    _s.stderr.write("%s needs arguments; it has no defined bare run.\n\n" % _s.argv[0])
    _s.stderr.write(d[i:i + 700] if i >= 0 else
        "No INVOCATION block in this file's docstring - that is the bug.\n")
    _s.stderr.write("\n")
    _s.exit(2)
```

```
_argguard()
```

```
LC_SEGMENT_64 = 0x19
```

```
def segments(data):
    ncmts = struct.unpack_from("<I", data, 16)[0]
    segs, p = [], 32
    for _ in range(ncmts):
        cmd, cmdsize = struct.unpack_from("<2I", data, p)
        if cmd == LC_SEGMENT_64:
            name = data[p + 8:p + 24].split(b"\x00")[0].decode()
            vmaddr, vmsize, fileoff, filesize = struct.unpack_from("<4Q", data, p + 24)
            segs.append((name, vmaddr, fileoff, filesize))
            p += cmdsize
    return segs
```

```
def sext(v, bits):
    m = 1 << (bits - 1)
    return (v ^ m) - m
```

```
def main():
    kc, tgt = sys.argv[1], int(sys.argv[2], 0)
    data = open(kc, "rb").read()
    segs = segments(data)
    want = "__TEXT_EXEC"
```

```
if "--seg" in sys.argv:
    want = sys.argv[sys.argv.index("--seg") + 1]

hits = []
for name, va, fo, fs in segs:
    if name != want or not fs:
        continue
    blk = data[fo:fo + fs]
    for off in range(0, len(blk) - 3, 4):
        w = struct.unpack_from("<I", blk, off)[0]
        top = w >> 26
        if top != 0x25 and top != 0x05:
            continue
        here = va + off
        dst = here + (sext(w & 0x3FFFFFF, 26) << 2)
        if dst == tgt:
            hits.append((here, "BL" if top == 0x25 else "B "))

print("target 0x%x : %d branch site(s)" % (tgt, len(hits)))
for a, k in hits:
    print("  %s 0x%x" % (k, a))

main()
```

Another tool that can be useful is [adrfind.py](#), but I need explain the exact reasons. ARM64 cannot load a 64-bit address in one instruction, so every reference to a global is built from a pair, where `addr` instruction forms the 4 KB page, and a paired `add` or even `ldr` instructions provides the offset inside this page. Therefore, nothing in the binary contains the address as a value to search for, and it is exactly at this point that [adrfind](#) tool is really useful because it reconstructs those pairs and reports every site that produces the address that we ask for. A simple example follows below:

```
$ $KC = "C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin"
$ python adrfind.py $KC 0xffffffff007e417b0

0xffffffff00b248aec  ADRP x9 + ADD #0x7b0 -> 0xffffffff007e417b0    (adrp @0xffffffff00b248ae8)
0xffffffff00b248f98  ADRP x9 + ADD #0x7b0 -> 0xffffffff007e417b0    (adrp @0xffffffff00b248f94)
# sites resolving to 0xffffffff007e417b0: 2
```

Readers can understand which function contains such addresses by using [fnprobe.py](#), which has been presented earlier in this text:

```
$ python fnprobe.py $KC 0xffffffff00b248aec | head -15
=====
func 0xffffffff00b2489c8 .. 0xffffffff00b248d68    len 0x3a0 (232 words)    decoded 232 insns
=====
0xffffffff00b2489c8 pacibsp
0xffffffff00b2489cc stp      x28, x27, [sp, #-0x60]!
0xffffffff00b2489d0 stp      x26, x25, [sp, #0x10]
0xffffffff00b2489d4 stp      x24, x23, [sp, #0x20]
0xffffffff00b2489d8 stp      x22, x21, [sp, #0x30]
0xffffffff00b2489dc stp      x20, x19, [sp, #0x40]
0xffffffff00b2489e0 stp      x29, x30, [sp, #0x50]
0xffffffff00b2489e4 add      x29, sp, #0x50
0xffffffff00b2489e8 mov      x21, x2
0xffffffff00b2489ec mov      x19, x0
0xffffffff00b2489f0 str      wzr, [x1, #0x7ac]
0xffffffff00b2489f4 ldr      w8, [x0]

$ python fnprobe.py $KC 0xffffffff00b248f98 | head -15
=====
func 0xffffffff00b248f38 .. 0xffffffff00b249040    len 0x108 (66 words)    decoded 66 insns
=====
0xffffffff00b248f38 pacibsp
```

```
0xffffffff00b248f3c stp      x22, x21, [sp, #-0x30]!
0xffffffff00b248f40 stp      x20, x19, [sp, #0x10]
0xffffffff00b248f44 stp      x29, x30, [sp, #0x20]
0xffffffff00b248f48 add      x29, sp, #0x20
0xffffffff00b248f4c mov      x19, x0
0xffffffff00b248f50 mrs      x22, tpidr_el1
0xffffffff00b248f54 bl       #0xffffffff00ac4b5a0
0xffffffff00b248f58 ldr      x21, [x0, #0x18]
0xffffffff00b248f5c cbz      x21, #0xffffffff00b249034
0xffffffff00b248f60 add      x20, x22, #0x5d0
0xffffffff00b248f64 ldr      x16, [x22, #0x110]
```

The [adrfind.py](#) code follows below:

```
#!/usr/bin/env python3
```

```
"""adrfind.py - find every ADRP(+ADD/LDR) that materialises a given ADDRESS.
```

Usage:

```
adrfind.py <kc> <target_vaddr> [--near N]
```

```
"""
```

```
import struct
import sys
```

```
def _argguard():
```

```
    import sys as _s
    if len(_s.argv) > 1:
        return
    d = __doc__ or ""
    i = d.find("INVOCATION")
    if i < 0:
        i = d.find("invocation")
    _s.stderr.write("%s needs arguments; it has no defined bare run.\n\n" % _s.argv[0])
    _s.stderr.write(d[i:i + 700] if i >= 0 else
        "No INVOCATION block in this file's docstring - that is the bug.\n")
    _s.stderr.write("\n")
    _s.exit(2)
```

```
_argguard()
```

```
LC_SEGMENT_64 = 0x19
```

```
def segments(data):
    ncmts = struct.unpack_from("<I", data, 16)[0]
    segs, p = [], 32
    for _ in range(ncmts):
        cmd, cmdsize = struct.unpack_from("<2I", data, p)
        if cmd == LC_SEGMENT_64:
            name = data[p + 8:p + 24].split(b"\x00")[0].decode()
            vmaddr, vmsize, fileoff, filesize = struct.unpack_from("<4Q", data, p + 24)
            segs.append((name, vmaddr, fileoff, filesize))
            p += cmdsize
    return segs
```

```
def sign_extend(v, bits):
    m = 1 << (bits - 1)
    return (v ^ m) - m
```

```
def main():
    kc = sys.argv[1]
    target = int(sys.argv[2], 16)
    data = open(kc, "rb").read()
    segs = segments(data)
```

```

text = [s for s in segs if s[0] == "__TEXT_EXEC" and s[3]]
if not text:
    text = [s for s in segs if s[3]]

hits = 0
for _, vm, fo, fs in text:
    blob = data[fo:fo + fs]
    # pass 1: record ADRP results per register
    for i in range(0, len(blob) - 4, 4):
        w = struct.unpack_from("<I", blob, i)[0]
        # ADRP: op=1 immlo(2) 10000 immhi(19) Rd(5)
        if (w & 0x9F000000) != 0x90000000:
            continue
        rd = w & 0x1F
        immlo = (w >> 29) & 3
        immhi = (w >> 5) & 0x7FFFF
        imm = sign_extend((immhi << 2) | immlo, 21) << 12
        pc = vm + i
        page = (pc & ~0xFFF) + imm
        # pass 2: look ahead a short window for the completing ADD/LDR on Rd
        for j in range(i + 4, min(i + 4 + 40, len(blob) - 4), 4):
            w2 = struct.unpack_from("<I", blob, j)[0]
            # ADD (immediate) 64-bit: sf=1 op=0 S=0 100010 shift imm12 Rn Rd
            if (w2 & 0xFF800000) == 0x91000000 and ((w2 >> 5) & 0x1F) == rd:
                imm12 = (w2 >> 10) & 0xFFF
                if (w2 >> 22) & 1:
                    imm12 <= 12
                if page + imm12 == target:
                    print("0x%x ADRP x%d + ADD #0x%x -> 0x%x (adrp @0x%x)"
                          % (vm + j, w2 & 0x1F, imm12, target, pc))
                    hits += 1
                break
            # LDR (immediate, unsigned offset) 64-bit on the same base
            if (w2 & 0xFFC00000) == 0xF9400000 and ((w2 >> 5) & 0x1F) == rd:
                imm12 = ((w2 >> 10) & 0xFFF) << 3
                if page + imm12 == target:
                    print("0x%x ADRP x%d + LDR [#0x%x] -> 0x%x (adrp @0x%x)"
                          % (vm + j, w2 & 0x1F, imm12, target, pc))
                    hits += 1
                break
            # the register was redefined before use: stop
            if (w2 & 0x1F) == rd and (w2 & 0x9F000000) == 0x90000000:
                break
    print("# sites resolving to 0x%x: %d" % (target, hits))

if __name__ == "__main__":
    main()

```

Sometimes we find strings, and it might be interesting to understand to know what address holds an [adrp](#) instruction that builds the string's address. For example, given the string below, we want to find what is the address referring to the *"NAMEI_ROOTDIR is set but ni_rootdir is not"* string:

```
$ python strref.py $KC "NAMEI_ROOTDIR is set but ni_rootdir is not"
STRING NAMEI_ROOTDIR is set but ni_rootdir is not at 0xffffffff00706cc97 (__PRELINK_TEXT)
```

```
NAMEI_ROOTDIR is set but ni_rootdir is not 1 reference(s)
0xffffffff00ad95e5c
```

```
$ python fnprobe.py $KC 0xffffffff00ad95e5c | head -10
```

```

=====
func 0xffffffff00ad95928 .. 0xffffffff00ad964e8 len 0xbc0 (752 words) decoded 752 insns
=====
0xffffffff00ad95928 pacibsp
0xffffffff00ad9592c sub sp, sp, #0x120
0xffffffff00ad95930 stp x28, x27, [sp, #0xc0]

```



```
0xffffffff00ad95934 stp      x26, x25, [sp, #0xd0]
0xffffffff00ad95938 stp      x24, x23, [sp, #0xe0]
0xffffffff00ad9593c stp      x22, x21, [sp, #0xf0]
0xffffffff00ad95940 stp      x20, x19, [sp, #0x100]
```

```
$ python fnprobe.py $KC 0xffffffff00ad95e5c | grep -A8 -B8 "<<<"
```

```
0xffffffff00ad95e3c stp      x22, x0, [sp]
0xffffffff00ad95e40 b        #0xffffffff00ad95f40
0xffffffff00ad95e44 b        #0xffffffff00ad95e88
0xffffffff00ad95e48 b        #0xffffffff00ad95f2c
0xffffffff00ad95e4c adrp     x8, #0xffffffff00706c000
0xffffffff00ad95e50 add     x8, x8, #0xbac
0xffffffff00ad95e54 mov     w9, #0x267
0xffffffff00ad95e58 stp      x8, x9, [sp]
0xffffffff00ad95e5c adrp     x0, #0xffffffff00706c000 <<<
0xffffffff00ad95e60 add     x0, x0, #0xc97
0xffffffff00ad95e64 bl      #0xffffffff00b4076bc
0xffffffff00ad95e68 mov     w9, #0x367
0xffffffff00ad95e6c adrp     x10, #0xffffffff00706c000
0xffffffff00ad95e70 add     x10, x10, #0xbac
0xffffffff00ad95e74 stp      x10, x9, [sp, #0x10]
0xffffffff00ad95e78 stp      x8, x28, [sp]
0xffffffff00ad95e7c adrp     x0, #0xffffffff00706c000
```

We know what function refers to our string (**0xffffffff00ad95928**), and we see the exact instruction doing it, which is the pair:

```
0xffffffff00ad95e5c adrp     x0, #0xffffffff00706c000 <<<
0xffffffff00ad95e60 add     x0, x0, #0xc97
```

It is possible to go further, but first it is necessary to do some obvious math because **adrp** instruction gives us a 4 KB aligned page address and the **add** instruction gives us the offset inside the that page:

- $0xffffffff00706c000 + 0xbac = 0xffffffff00706cbac$
- $0xffffffff00706c000 + 0xc97 = 0xffffffff00706cc97$

Using these values, we have:

```
$ python straddr.py $KC 0xffffffff00706cbac 0xffffffff00706cc97
0xffffffff00706cbac (__PRELINK_TEXT) 'vfs_lookup.c'
0xffffffff00706cc97 (__PRELINK_TEXT) 'NAMEI_ROOTDIR is set but ni_rootdir is not\n @%s:%d'
```

Keeping the analysis in these lines, the next line has a **bl** instruction (branch with link instruction), which is responsible for call subroutines or functions by jumping to a target address and saving the returning address in the link register (X30). Thus, it is calling the function at **0xffffffff00b4076bc**, which we can examine:

```
$ python fnprobe.py $KC 0xffffffff00b4076bc
```

```
=====
func 0xffffffff00b4076bc .. 0xffffffff00b4076f8   len 0x3c (15 words)   decoded 15 insns
=====
0xffffffff00b4076bc pacibsp      <<<
0xffffffff00b4076c0 sub         sp, sp, #0x20
0xffffffff00b4076c4 stp         x29, x30, [sp, #0x10]
0xffffffff00b4076c8 add         x29, sp, #0x10
0xffffffff00b4076cc add         x8, x29, #0x10
0xffffffff00b4076d0 str         x8, [sp, #8]
0xffffffff00b4076d4 mov         x6, x30
0xffffffff00b4076d8 xpaci       x6
0xffffffff00b4076dc add         x1, sp, #8
0xffffffff00b4076e0 mov         w2, #0
0xffffffff00b4076e4 mov         x3, #0
```

```
0xffffffff00b4076e8 mov    x4, #0
0xffffffff00b4076ec mov    x5, #0
0xffffffff00b4076f0 mov    x7, #0
0xffffffff00b4076f4 bl     #0xffffffff00ab8d090
```

Going to the last address of the instructions above we have:

```
$ python fnprobe.py $KC 0xffffffff00ab8d090 | grep -A1 adrp | grep -B1 add
```

```
0xffffffff00ab8d108 adrp    x0, #0xffffffff007046000
0xffffffff00ab8d10c add     x0, x0, #0x30b
--
0xffffffff00ab8d114 adrp    x0, #0xffffffff007046000
0xffffffff00ab8d118 add     x0, x0, #0x316
--
0xffffffff00ab8d138 adrp    x27, #0xffffffff00b724000
0xffffffff00ab8d13c add     x27, x27, #0x140
--
0xffffffff00ab8d1b8 adrp    x9, #0xffffffff007013000
0xffffffff00ab8d1bc add     x9, x9, #0x870
--
0xffffffff00ab8d1e8 adrp    x8, #0xffffffff00b795000
0xffffffff00ab8d1ec add     x8, x8, #0xc00
--
0xffffffff00ab8d250 adrp    x8, #0xffffffff007e8a000
0xffffffff00ab8d254 add     x8, x8, #0x898
--
0xffffffff00ab8d27c adrp    x8, #0xffffffff007e8a000
0xffffffff00ab8d280 add     x8, x8, #0x8c0
--
0xffffffff00ab8d2d8 adrp    x8, #0xffffffff007e8a000
0xffffffff00ab8d2dc add     x8, x8, #0x8c0
--
0xffffffff00ab8d370 adrp    x27, #0xffffffff00b724000
0xffffffff00ab8d374 add     x27, x27, #0x140
--
0xffffffff00ab8d3b4 adrp    x0, #0xffffffff007046000
0xffffffff00ab8d3b8 add     x0, x0, #0x35a
--
0xffffffff00ab8d3f0 adrp    x9, #0xffffffff00b728000
0xffffffff00ab8d3f4 add     x9, x9, #0
--
0xffffffff00ab8d41c adrp    x8, #0xffffffff00b768000
0xffffffff00ab8d420 add     x8, x8, #0xfbc
--
0xffffffff00ab8d460 adrp    x8, #0xffffffff007046000
0xffffffff00ab8d464 add     x8, x8, #0x36f
--
0xffffffff00ab8d4ac adrp    x1, #0xffffffff007046000
0xffffffff00ab8d4b0 add     x1, x1, #0x36f
--
0xffffffff00ab8d4dc adrp    x9, #0xffffffff007065000
0xffffffff00ab8d4e0 add     x9, x9, #0x5c3
0xffffffff00ab8d4e4 adrp    x10, #0xffffffff007065000
0xffffffff00ab8d4e8 add     x10, x10, #0x610
--
0xffffffff00ab8d4f4 adrp    x0, #0xffffffff007065000
0xffffffff00ab8d4f8 add     x0, x0, #0x5dd
```

We should do the same math we did previously and then discover string by string, but it would consume some valuable time, so it is easier to use another tool:

```
$ python .\strings_in.py $KC 0xffffffff00ab8d090
```

```
0xffffffff00ab8d090 : 17 adrp+add pair(s) -> 14 unique target(s) -> 7 string(s)
```

```
0xffffffff00704630b (__PRELINK_TEXT) 'panic: %s\n'
0xffffffff007046316 (__PRELINK_TEXT) 'Kernel panicked very early before serial init,
spinning forever...\n'
0xffffffff00704635a (__PRELINK_TEXT) '/var/log/panic.trace'
0xffffffff00704636f (__PRELINK_TEXT) 'panic'
0xffffffff0070655c3 (__PRELINK_TEXT) 'machine_routines_common.c'
0xffffffff007065610 (__PRELINK_TEXT) 'ml_set_interrupts_enabled_with_debug'
0xffffffff0070655dd (__PRELINK_TEXT) '%s: debug exceptions enabled in kernel mode @%s:%d'
```

That is exactly what we are looking for, and apparently the function being called in "[bl#0xffffffff00ab8d090](#)" is associated directly or indirectly with panic routines! I have another tool named [fsprobe.py](#), which is an improved version of [fnprobe.py](#). While [fnprobe.py](#) is focused on XNU, [fsprobe.py](#) extend the functionality to kext addresses, refuses an unmapped address and, mainly, accept --sym option. Therefore, to solve the riddle:

```
$ python fsprobe.py $KC 0xffffffff00ab8d090 --sym
```

```
0xffffffff00ab8d090 -> func 0xffffffff00ab8d090 .. 0xffffffff00ab8d51c (com.apple.kernel)
symbol      panic_trap_to_debugger (EXACT)
! DUPLICATE the name 'panic_trap_to_debugger' is used at 1 other address(es): 0xffffffff00ad4011c
  at most one can be right - score them by behaviour, not by the name
measured    291 instruction(s), 0 'ret' -> NORETURN
```

I hope this sequence of commands have proved the relevance of writing small tools to daily iOS security research. Anyway, I still have to show the code of four tools that have not been shown yet such as [strref](#), [straddr](#), [strings_in](#) and [fsprobe](#).

The [strref.py](#) source code follows below:

```
$ cat .\strref.py
```

```
#!/usr/bin/env python3
```

```
""" strref.py: Find a string in a kernelcache and every code site that forms its address.
```

```
Usage: strref.py <kc> <string> [<string> ...]
```

```
"""
```

```
import struct
import sys
```

```
def _argguard():
```

```
    import sys as _s
    if len(_s.argv) > 1:
        return
    d = __doc__ or ""
    i = d.find("INVOCATION")
    if i < 0:
        i = d.find("invocation")
    _s.stderr.write("%s needs arguments; it has no defined bare run.\n\n" % _s.argv[0])
    _s.stderr.write(d[i:i + 700] if i >= 0 else
        "No INVOCATION block in this file's docstring - that is the bug.\n")
    _s.stderr.write("\n")
    _s.exit(2)
```

```
_argguard()
```

```
LC_SEGMENT_64 = 0x19
```

```
def segments(data):
    ncmts = struct.unpack_from("<I", data, 16)[0]
```

```
segs, p = [], 32
for _ in range(ncmds):
    cmd, cmdsize = struct.unpack_from("<2I", data, p)
    if cmd == LC_SEGMENT_64:
        name = data[p + 8:p + 24].split(b"\x00")[0].decode()
        vmaddr, vmsize, fileoff, filesize = struct.unpack_from("<4Q", data, p + 24)
        segs.append((name, vmaddr, fileoff, filesize))
    p += cmdsize
return segs

def sext(v, b):
    return v - (1 << b) if v & (1 << (b - 1)) else v

def main():
    kc = sys.argv[1]
    data = open(kc, "rb").read()
    segs = segments(data)

    targets = {}
    for s in sys.argv[2:]:
        pat = s.encode()
        for name, va, fo, fs in segs:
            if not fs:
                continue
            blk = data[fo:fo + fs]
            i = blk.find(pat)
            while i >= 0:
                targets[va + i] = s
                print("STRING %-28s at 0x%x (%s)" % (s, va + i, name))
                i = blk.find(pat, i + 1)
    if not targets:
        print("no strings found")
        return

    found = {}
    for name, va, fo, fs in segs:
        if "TEXT_EXEC" not in name or not fs:
            continue
        cnt = fs // 4
        ws = struct.unpack_from("<%dI" % cnt, data, fo)
        for i in range(cnt):
            w = ws[i]
            if (w & 0x9F000000) != 0x90000000: # ADRP
                continue
            rd = w & 0x1F
            imm = sext((((w >> 5) & 0x7FFFF) << 2) | ((w >> 29) & 3), 21) << 12
            page = ((va + i * 4) & ~0xFFF) + imm
            for j in range(i + 1, min(i + 10, cnt)):
                x = ws[j]
                if (x & 0xFF800000) == 0x91000000 and ((x >> 5) & 0x1F) == rd:
                    v = page + ((x >> 10) & 0xFFF)
                    if v in targets:
                        found.setdefault(targets[v], []).append(va + i * 4)
                    break
                if (x & 0x1F) == rd and (x & 0x9F000000) == 0x90000000:
                    break

    for s in sys.argv[2:]:
        refs = found.get(s, [])
        print("\n%-28s %d reference(s)" % (s, len(refs)))
        for a in refs[:20]:
            print("    0x%x" % a)

main()
```

The [straddr.py](#) source code follows below:

```
$ cat .\straddr.py
```

```
#!/usr/bin/env python3
```

```
r"""straddr.py - the invert of strref: given an ADDRESS, print the string that lives there.
```

```
USAGE: straddr.py <kc> 0xADDR [0xADDR ...] [--selftest]
```

```
"""
```

```
import struct
import sys
```

```
LC_SEGMENT_64 = 0x19
```

```
def segments(data):
    n = struct.unpack_from("<I", data, 16)[0]
    p, segs = 32, []
    for _ in range(n):
        cmd, cs = struct.unpack_from("<2I", data, p)
        if cmd == LC_SEGMENT_64:
            nm = data[p + 8:p + 24].split(b"\x00")[0].decode(errors="replace")
            vm, vs, fo, fs = struct.unpack_from("<4Q", data, p + 24)
            if fs:
                segs.append((nm, vm, fo, fs))
        p += cs
    return segs
```

```
def _printable(b):
    return 0x20 <= b < 0x7F or b in (9, 10, 13)
```

```
def lookup(data, segs, va, maxlen=400):
    """(state, start_va, text, segment) for the string at or around va."""
    seg = next(((nm, vm, fo, fs) for nm, vm, fo, fs in segs if vm <= va < vm + fs), None)
    if seg is None:
        return "no seg", None, None, None
    nm, vm, fo, fs = seg
    o = fo + (va - vm)
    if not _printable(data[o]):
        return "binary", None, None, nm
    # walk back to the NUL that ends the previous string
    s = o
    while s > fo and _printable(data[s - 1]):
        s -= 1
    e = data.find(b"\x00", s, s + maxlen)
    if e < 0:
        return "binary", None, None, nm
    text = data[s:e].decode("ascii", errors="replace")
    start = vm + (s - fo)
    return ("exact" if s == o else "inside"), start, text, nm
```

```
def show(data, segs, va):
    st, start, text, seg = lookup(data, segs, va)
    if st == "no seg":
        print("0x%x NO SEGMENT - not a mapped address" % va)
    elif st == "binary":
        print("0x%x (%s) not printable text" % (va, seg))
    elif st == "exact":
        print("0x%x (%s) %r" % (va, seg, text))
    else:
        print("0x%x (%s) INSIDE a string starting at 0x%x (+%d): %r"
              % (va, seg, start, va - start, text))
    return st
```

```
def selftest(path):
    data = open(path, "rb").read()
    segs = segments(data)
```

```

ok = True

st, start, text, _ = lookup(data, segs, 0xffffffff00706cbac)
c1 = st == "exact" and text == "vfs_lookup.c"
print("C1 known address -> known text          %s %r" % ("PASS" if c1 else "FAIL",
text))
ok &= c1

st, start, text, _ = lookup(data, segs, 0xffffffff00706cbac + 4)
c2 = st == "inside" and start == 0xffffffff00706cbac
print("C2 address INSIDE it reports the START      %s (%s, start=%s)"
      % ("PASS" if c2 else "FAIL", st, hex(start) if start else None))
ok &= c2

st = lookup(data, segs, 0x1)[0]
c3 = st == "no seg"
print("C3 unmapped address REFUSED                %s (%s)" % ("PASS" if c3 else "FAIL",
st))
ok &= c3

st = lookup(data, segs, 0xffffffff00ad95928)[0]
c4 = st == "binary"
print("C4 executable code is not reported as a string %s (%s)" % ("PASS" if c4 else "FAIL",
st))
ok &= c4

print("SELFTEST %s" % ("PASS" if ok else "FAIL"))
return 0 if ok else 1

def main():
    if len(sys.argv) < 2:
        sys.stderr.write(__doc__.strip() + "\n")
        return 2
    if "--selftest" in sys.argv:
        return selftest(sys.argv[1])
    data = open(sys.argv[1], "rb").read()
    segs = segments(data)
    for a in [int(x, 16) for x in sys.argv[2:] if x.startswith("0x")]:
        show(data, segs, a)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())

```

The [strings_in.py](#) source code follows below:

```
$ cat .\strings_in.py
```

```
#!/usr/bin/env python3
```

```
r"""strings\_in.py - every STRING a function references, by composing fnprobe and straddr.
```

```
USAGE:
```

```
strings_in.py <kc> 0xFUNCADDR [--all] [--selftest]
```

```
    --all          also list the pairs whose target is NOT text, with their addresses
    --selftest     the controls
```

```
"""
```

```
import os
import re
import subprocess
import sys
```

```
HERE = os.path.dirname(os.path.abspath(__file__))
```

```
def _run(tool, *args):
    return subprocess.run([sys.executable, os.path.join(HERE, tool)] + list(args),
                          capture_output=True, text=True).stdout

def targets(kc, addr):
    """Every adrp+add result inside the function containing addr, in order."""
    body = _run("fnprobe.py", kc, addr)
    pend, out = {}, []
    for ln in body.splitlines():
        m = re.match(r"\s+0x([0-9a-f]+)\s+(\S+)\s+(.*?)(?:\s+<<)?$", ln)
        if not m:
            continue
        op, ops = m.group(2), m.group(3).strip()
        a = re.match(r"(x\d+), #(0x[0-9a-f]+)$", ops)
        if op == "adrp" and a:
            pend[a.group(1)] = int(a.group(2), 16)
            continue

        b = re.match(r"(x\d+), (x\d+), #(?:0x)?[0-9a-f]+$", ops)
        if op == "add" and b and b.group(2) in pend:
            out.append((int(m.group(1), 16), pend[b.group(2)] + int(b.group(3), 0)))
            pend.pop(b.group(2), None)
            continue
        # an `ldr` completion means a POINTER SLOT, not a string - consume and drop it
        c = re.match(r"[wx]\d+, \[(x\d+)(?:, #(0x[0-9a-f]+))?\]$", ops)
        if op.startswith("ldr") and c and c.group(1) in pend:
            pend.pop(c.group(1), None)
    return out

def strings_of(kc, addr, want_all=False):
    rows, seen = [], set()
    for site, t in targets(kc, addr):
        if t in seen:
            continue
        seen.add(t)
        out = _run("straddr.py", kc, hex(t)).strip()
        istext = out and "not printable" not in out and "NO SEGMENT" not in out
        if istext or want_all:
            rows.append((site, t, out, istext))
    return rows

def selftest(kc):
    ok = True
    s = [r[2] for r in strings_of(kc, "0xffffffff00ab8d090")]
    c1 = any("panic: %s" in x for x in s)
    print("C1 the panic implementation yields 'panic: %s'           %s (%d strings)"
          % ("PASS" if c1 else "FAIL", len(s)))
    ok &= c1

    n = len(strings_of(kc, "0xffffffff00b4076bc"))
    c2 = n == 0
    print("C2 the varargs forwarder has NO strings                 %s (%d)"
          % ("PASS" if c2 else "FAIL", n))
    ok &= c2

    t = [x for _, x, _, _ in strings_of(kc, "0xffffffff00ab8d090", True)]
    c3 = 0xffffffff007e8ac48 not in t
    print("C3 `ldr` pointer slots are EXCLUDED                     %s" % ("PASS" if c3 else
    "FAIL"))
    ok &= c3

    s = [r[2] for r in strings_of(kc, "0xffffffff00ad95928")]
    c4 = (any("vfs_lookup.c" in x for x in s)
          and any("NAMEI_ROOTDIR is set but ni_rootdir is not" in x for x in s))
    print("C4 namei_do_lookup yields the two HAND read strings     %s (%d strings)"
          % ("PASS" if c4 else "FAIL", len(s)))
```



```
ok &= c4

t = [x for _, x, _, _ in strings_of(kc, "0xffffffff00ab8d090", True)]
c5 = 0xffffffff00b728000 in t
print("C5 a DECIMAL immediate (#0) is PAIRED, not dropped      %s (%d pairs)"
      % ("PASS" if c5 else "FAIL", len(t)))
ok &= c5

print("SELFTTEST %s" % ("PASS" if ok else "FAIL"))
return 0 if ok else 1

def main():
    if len(sys.argv) < 2:
        sys.stderr.write(__doc__.strip() + "\n")
        return 2
    kc = sys.argv[1]
    if "--selftest" in sys.argv:
        return selftest(kc)
    addr = next((a for a in sys.argv[2:] if a.startswith("0x")), None)
    if addr is None:
        sys.stderr.write("need a 0x function address\n")
        return 2
    allpairs = targets(kc, addr)
    uniq = len({t for _, t in allpairs})
    rows = strings_of(kc, addr, "--all" in sys.argv)
    ntext = sum(1 for r in rows if r[3]) if "--all" in sys.argv else len(rows)

    print("%s : %d adrp+add pair(s) -> %d unique target(s) -> %d string(s)"
          % (addr, len(allpairs), uniq, ntext))
    for site, t, out, istext in rows:
        print(" %s %s" % (" " if istext else "(!)", out))
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

The `fsprobe.py` source code follows below:

```
$ cat ./fsprobe.py
```

```
#!/usr/bin/env python3
```

```
r"""fsprobe.py - fnprobe for the whole fileset, and not just com.apple.kernel.
```

```
USAGE: fsprobe.py <kc> 0xADDR [0xADDR ...] [--range LO HI] [--grep MNEM] [--quiet]
```

```
"""
```

```
import bisect
import io
import json
import os
import struct
import sys
```

```
from capstone import CS_ARCH_ARM64, CS_MODE_ARM, Cs
```

```
LC_SEGMENT_64 = 0x19
LC_FUNCTION_STARTS = 0x26
LC_FILESET_ENTRY = 0x80000035
```

```
def symbols(path):
    p = path + ".symbols.json"
    if not os.path.exists(p):
        return {}
    try:
        d = json.load(io.open(p, encoding="utf-8", errors="replace"))
    except Exception:
```

```
        return {}
    out = {}
    for k, v in (d.items() if isinstance(d, dict) else []):
        try:
            out[int(k)] = v
        except (TypeError, ValueError):
            continue
    return out

def symlookup(sym, keys, byname, a):
    if not sym:
        return "NO MAP", None, None, []
    if a in sym:
        n = sym[a]
        return "EXACT", n, 0, [x for x in byname.get(n, []) if x != a]
    i = bisect.bisect_right(keys, a) - 1
    if i < 0:
        return "NONE", None, None, []
    n = sym[keys[i]]
    return "NEAREST", n, a - keys[i], [x for x in byname.get(n, []) if x != keys[i]]

def _uleb(d, p, end):
    val, sh = 0, 0
    while p < end:
        b = d[p]
        p += 1
        val |= (b & 0x7F) << sh
        if not (b & 0x80):
            return val, p
        sh += 7
    return None, p

def _entry_info(d, hdr):
    n = struct.unpack_from("<I", d, hdr + 16)[0]
    q, text, fstarts, imgbase = hdr + 32, None, None, None
    for _ in range(n):
        cmd, cs = struct.unpack_from("<2I", d, q)
        if cs == 0:
            break
        if cmd == LC_SEGMENT_64:
            nm = d[q + 8:q + 24].split(b"\x00")[0].decode(errors="replace")
            vm, vs, fo, fs = struct.unpack_from("<4Q", d, q + 24)
            if nm == "__TEXT_EXEC" and fs:
                text = (vm, fo, fs)
            if nm == "__TEXT":
                imgbase = vm
        elif cmd == LC_FUNCTION_STARTS:
            fstarts = struct.unpack_from("<2I", d, q + 8)
            q += cs
    starts = []
    if fstarts and imgbase is not None:
        off, sz = fstarts
        cur, p, end = imgbase, off, off + sz
        while p < end:
            val, p = _uleb(d, p, end)
            if not val:
                break
            cur += val
            starts.append(cur)
    return text, sorted(starts)

def build(path):
    d = open(path, "rb").read()
    ncmts = struct.unpack_from("<I", d, 16)[0]
    p, ents = 32, []
    for _ in range(ncmts):
```

```
cmd, cs = struct.unpack_from("<2I", d, p)
if cmd == LC_FILESET_ENTRY:
    vm, fo = struct.unpack_from("<2Q", d, p + 8)
    eoff = struct.unpack_from("<I", d, p + 24)[0]
    nm = d[p + eoff:p + cs].split(b"\x00")[0].decode(errors="replace")
    text, starts = _entry_info(d, fo)
    if text:
        ents.append((nm, text, starts))
    p += cs
if not ents:
    text, starts = _entry_info(d, 0)
    if text:
        ents.append(("<top level>", text, starts))
return d, ents

def locate(ents, a):
    """The (name, text, starts) whose __TEXT_EXEC contains a, else None."""
    for nm, (vm, fo, fs), starts in ents:
        if vm <= a < vm + fs:
            return nm, (vm, fo, fs), starts
    return None

def bounds(ents, a):
    hit = locate(ents, a)
    if hit is None:
        return None
    nm, (vm, fo, fs), starts = hit
    lo, hi = vm, vm + fs
    inside = [s for s in (starts or []) if lo <= s < hi]
    if not inside:
        return None
    i = bisect.bisect_right(inside, a) - 1
    if i < 0:
        return None
    s = inside[i]
    e = inside[i + 1] if i + 1 < len(inside) else hi
    return nm, s, e

def show(d, ents, a, lo=None, hi=None, grep=None, quiet=False):
    b = bounds(ents, a)
    if b is None:
        where = locate(ents, a)
        print("=" * 88)
        print("REFUSED 0x%x - %s" % (a, "no __TEXT_EXEC in any fileset entry contains it"
                                   if where is None else
                                   "entry %s has no function starts covering it" % where[0]))
        return None
    nm, s, e = b
    _, (vm, fo, fs), _ = locate(ents, a)
    md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
    md.detail = False
    ins = list(md.disasm(d[fo + (s - vm):fo + (e - vm)], s))
    nwords = (e - s) // 4
    print("=" * 88)
    print("func 0x%x .. 0x%x len 0x%x (%d words) decoded %d insns%s" %
          (s, e, e - s, nwords, len(ins),
           " *** TRUNCATED ***" if len(ins) != nwords else ""))
    print("entry %s" % nm)
    print("=" * 88)
    if not quiet:
        for x in ins:
            if lo is not None and not (lo <= x.address <= hi):
                continue
            if grep and grep not in x.mnemonic:
                continue
            print(" 0x%-11x %-8s %s%s" % (x.address, x.mnemonic, x.op_str,
                                           " <<<" if x.address == a else ""))
```

```
return nm, s, e

def selftest(path):
    d, ents = build(path)
    ok = True

    r = bounds(ents, 0xffffffff00b0419ac)          # inside mld_v2_enqueue_group_record
    c1 = r is not None and r[1] == 0xffffffff00b0409f4 and (r[2] - r[1]) // 4 == 1022
    print("C1 XNU address -> recorded function, 1022 words          %s %s"
          % ("PASS" if c1 else "FAIL", r))
    ok &= c1

    r = bounds(ents, 0xffffffff00ab32260)          # inside the LIFS override setter
    c2 = r is not None and r[0] == "com.apple.filesystems.lifs" and r[2] > r[1]
    print("C2 KEXT address -> lifs, POSITIVE length                %s %s"
          % ("PASS" if c2 else "FAIL", r))
    ok &= c2

    r = bounds(ents, 0x00000000100000000)
    c3 = r is None
    print("C3 address in no __TEXT_EXEC is REFUSED                %s %s"
          % ("PASS" if c3 else "FAIL", r))
    ok &= c3

    bad = []
    for a in (0xffffffff00b0419ac, 0xffffffff00ab32260, 0xffffffff00ab1b658,
              0xffffffff00b034c10, 0xffffffff00b042640):
        r = bounds(ents, a)
        if r and (r[2] <= r[1] or (r[2] - r[1]) % 4):
            bad.append((hex(a), r))
    c4 = not bad
    print("C4 every length positive and word aligned              %s %s"
          % ("PASS" if c4 else "FAIL", bad or "5 of 5"))
    ok &= c4

    sym = symbols(path)
    keys = sorted(sym)
    byname = {}
    for k, v in sym.items():
        byname.setdefault(v, []).append(k)

    kind, name, delta, dup = symlookup(sym, keys, byname, 0xffffffff00ab8d090)
    c5 = kind == "EXACT" and name == "panic_trap_to_debugger"
    print("C5 an EXACT symbol is reported as EXACT                  %s (%s %s)"
          % ("PASS" if c5 else "FAIL", kind, name))
    ok &= c5

    c6 = len(dup) == 1
    print("C6 a DUPLICATED name is flagged                          %s (%d other address)"
          % ("PASS" if c6 else "FAIL", len(dup)))
    ok &= c6

    kind, name, delta, _ = symlookup(sym, keys, byname, 0xffffffff00b4076bc)

    c7 = bool(kind == "NEAREST" and delta)
    print("C7 a NEAREST match is NOT passed off as an answer        %s (%s +0x%s)"
          % ("PASS" if c7 else "FAIL", kind, format(delta or 0, "x")))
    ok &= c7

    c8 = symbols(path + ".zzq9-absent") == {}
    print("C8 an ABSENT symbol file degrades to NO MAP              %s" % ("PASS" if c8 else
"FAIL"))
    ok &= c8

    nkext = sum(1 for e in ents if e[2])
    print("\nfileset entries with __TEXT_EXEC: %d, of which %d carry function starts"
          % (len(ents), nkext))
```

```
print("symbols loaded: %d, distinct names: %d, names used at more than one address: %d"
      % (len(sym), len(byname), sum(1 for v in byname.values() if len(v) > 1)))
print("SELFTTEST %s" % ("PASS" if ok else "FAIL"))
return 0 if ok else 1

def report_sym(path, d, ents, a):
    sym = symbols(path)
    keys = sorted(sym)
    byname = {}
    for k, v in sym.items():
        byname.setdefault(v, []).append(k)
    b = bounds(ents, a)
    if b is None:
        print("0x%x REFUSED - not inside any function" % a)
        return 1
    nm, s, e = b
    kind, name, delta, dup = symlookup(sym, keys, byname, s)
    print("0x%x -> func 0x%x .. 0x%x (%s)" % (a, s, e, nm))
    if kind == "NO MAP":
        print("    symbol      NO SYMBOL FILE beside the kernelcache")
    elif kind == "NONE":
        print("    symbol      none at or below this address")
    elif kind == "EXACT":
        print("    symbol      %s (EXACT)" % name)
    else:
        print("    symbol      %s +0x%x ! NOT EXACT - a nearest match is not an answer" %
              (name, delta))
    if name and dup:
        print("    ! DUPLICATE the name '%s' is used at %d other address(es): %s"
              % (name, len(dup), ", ".join(hex(x) for x in sorted(dup)[:6])))
        print("    at most one can be right - score them by behaviour, not by the name")
    # the cheap behavioural fact, so the name is never the only thing on screen
    _, (vm, fo, fs), _ = locate(ents, a)
    md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
    md.detail = False
    ins = list(md.disasm(d[fo + (s - vm):fo + (e - vm)], s))
    nret = sum(1 for x in ins if x.mnemonic in ("ret", "retab", "retaa"))
    print("    measured    %d instruction(s), %d `ret` -> %s"
          % (len(ins), nret, "RETURNS" if nret else "NORETURN"))
    return 0

def main():
    if "--selftest" in sys.argv:
        return selftest(sys.argv[1])
    d, ents = build(sys.argv[1])
    if "--sym" in sys.argv:
        for a in [int(x, 16) for x in sys.argv[2:] if x.startswith("0x")]:
            report_sym(sys.argv[1], d, ents, a)
        return 0
    lo = hi = grep = None
    if "--range" in sys.argv:
        i = sys.argv.index("--range")
        lo, hi = int(sys.argv[i + 1], 16), int(sys.argv[i + 2], 16)
    if "--grep" in sys.argv:
        grep = sys.argv[sys.argv.index("--grep") + 1]
    quiet = "--quiet" in sys.argv
    for a in [int(x, 16) for x in sys.argv[2:] if x.startswith("0x")]:
        show(d, ents, a, lo, hi, grep, quiet)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

10. Making the kernelcache queryable

Now you have the initial set of tools (there are many others), we can focus on other mechanisms and improve them for daily iOS research. As you already know, an IDA Pro IDB is excellent for navigation, but it does not offer the best possibility for queries and, believe me, that is the exact capability you are looking for while working through a kernelcache, and the situation becomes more critical when you need to query across multiple iOS builds simultaneously. Initially, it seems a not common task, but I can ensure that it is. Consider queries like *"find every function whose pseudocode contains the string autda and whose name decompile each function on the fly, one at a time"* or even *"provide me with every function whose body has changed between iOS 26.6.1 and iOS 27.2 beta"*. Certainly, both queries would not be feasible via IDA Pro because it does not offer such flexibility in cases like these ones.

One of the solutions is to extract the IDB's content into a format designed for bulk querying like SQLite, which has a simple and direct structure, and allows a given script extracts every function from the IDB once, in a one-time pass, and stores all results into a SQLite database. Afterwards, we can query that database as many times as we want and without touching IDA Pro again.

The concept about using SQL to query IDA database is based on the excellent and unique project IDASQL (<https://github.com/allthingsida/idasql>) from my colleague [Elias Bachaalany](#) (@eliasbchlny_ and @BinaryWizards on X), who offers a really well-engineered project, but there are evidence differences. The IDASQL project is written in C++20, builds its architecture on live SQLite virtual tables over IDA's own data, keeps IDA in the loop for every query, exposes more than 30 tables, and can even query the Hex-Rays ctree AST. On the other side, the micro project below is composed of a few scripts, adopts a simpler and adaptive approach because they are written in Python, split the work into an export step and an index step, needs IDA Pro only during the extraction phase, and then work on snapshots. Obviously, it is much more limited mechanism, but that work surprisingly well with iOS security research and, instead of developing it using multiple stages, this article will keep it as simple as possible, following a two-stage pipeline. The general scheme is submitting a query → check on a local indexed database → return the answer. Besides being really fast, it also allows to have many builds as you have extracted, and to build a joined query.

As I have been writing this article over days, the iOS world have moved fast and, at this point, Apple has released iOS 27.0 (GA 24A437, iPhone13,2 -- iPhone 12) and, soon afterwards, iOS 27.2 beta (24B5084k, iPhone17,3 -- iPhone 16), and I will adopt GA for as primary image for part of section, but both images will be used. The following pages present the complete walkthrough of this procedure, which can be performed on macOS or Windows. In my case, I will be using Windows because my system has many more resources than macOS.

Step 01 # identify the image: the image to be used is the already downloaded, extracted, and decompressed kernelcache, and we will be using C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin.

```
$ file kc_27.0ga.bin
kc_27.0ga.bin: Mach-O 64-bit arm64e (caps: PAK00) filetype=12

$ sha256sum kc_27.0ga.bin
a36b924108b63eefd7cc8fa9e54acc3a07544786d3edb316fbcf3b9fb7426b35 *kc_27.0ga.bin
```

Step 02 # Check the current infrastructure: to avoid any surprises, check whether your environment contain all necessary tools:

```
$ ipsw version
Version: 3.1.718, BuildCommit: 9c7af5bd4b8991beabfc98f9d30eaa9c34b66041

$ sqlite3 --version
3.53.4 2026-07-24 19:02:57 bf7c7f30031888f4e796e429ab3978879485813aaca6f641c7b33e4e09459bcc (64-bit)

$ ls symbolicator/kernel/27.0

Mode                LastWriteTime         Length Name
----                -
d-----            8/25/2026   9:30 PM          kexts
-a-----            8/25/2026   9:30 PM     5319560 xnu.json

$ python3 -V
Python 3.13.7

$ python3 -c "import sqlite3; print(sqlite3.sqlite_version)"
3.50.4

$ python3 -c "import sqlite3; sqlite3.connect(':memory:').execute('CREATE VIRTUAL TABLE t USING fts5(x)'); print('FTS5 ok')"
FTS5 ok

$ & 'C:\Program Files\IDA Professional 9.4\idat.exe' -h

$ git -C C:\IDAScripts\ida_kernelcache log -1 --format=%h
81ca5fc

$ git -C C:\IDAScripts\ida_kcpp log -1 --format=%h
acab27f
```

On macOS, program may be in different locations (IDA Pro executable is located on [/Applications/IDA Professional 9.4.app/Contents/MacOS/idat](#) directory), but steps are essentially the same.

Step 03: typical symbolication: as shown previously, we must symbolicate the kernelcache. We have already done it previously, and steps here are a mere refresh (I hope...):

```
$ cd C:\research_ios
$ git clone --depth 1 https://github.com/blacktop/symbolicator
$ ipsw kernel symbolicate --json kernelcaches\ios27ga\kc_27.0ga.bin
$ ipsw kernel symbolicate --signatures ~/github/symbolicator/kernel/27.0 --json -o . kc_27.0ga.bin
```

Step 04 # building the ida database: this step has already been done using the [build_idb.py](#) script, and as it takes time (between 20 and 30 minutes), next lines represent a repetition from previous steps, and the only difference is the last command, which makes a copy of the database:

```
$ $ida = "C:\Program Files\IDA Professional 9.4\idat.exe"
$ $base = "C:\research_ios\kernelcaches\ios27ga"
$ & $ida -A "-S$base\build_idb.py" "-L$base\ida94.log" "$base\kc_27.0ga.bin"

$ ls -lh kc_27.0ga.bin.i64
-rwxrwx--- 1 Administrators None 996M Sep 14 20:18 kc_27.0ga.bin.i64
```

Step 05 # building a corpus: this step is where the new procedure comes up. The central idea is to extract every function's disassembly and pseudocode from the IDA database once, index it into a SQLite with full text search and provide answer in seconds. Thus, even though this step takes a huge amount of time, it is necessary to do it once, and then all requests can be responded almost immediately.


```
$ Copy-Item "$base\kc_27.0ga.bin.i64" "$base\kc_27.0ga.corpus.i64"
$ $tool = "C:\research_ios\kernelcaches\corpus_tools"
$ & $ida -A "-S$tool\extract_corpus_v2.py --out $base\corpus" "-L$base\corpus_extract.log"
"$base\kc_27.0ga.corpus.i64"
```

As reference, the output is saved into corpus/ folder. This folder has around 1.9 GB, the command took almost three hours to get completed and, examining the log file, it shows 215,841 functions. Check the content of end of the log file as shown below:

```
$ Get-Content "$base\corpus_extract.log" -Tail 15
extracted 214000 / 215841 (99.1%)
extracted 214500 / 215841 (99.4%)
extracted 215000 / 215841 (99.6%)
extracted 215500 / 215841 (99.8%)
```

[autohidden] The decompiler assumes that the segment '__TEXT.HEADER' is read-only because of its PERMISSIONS.

All data references to the segment will be replaced by constant values.
This may lead to drastic changes in the decompiler output.
If the segment is not read-only, please change the segment PERMISSIONS.

In general, the decompiler checks the segment permissions, class, and name
to determine if it is read-only.
-> OK

Done: 215841 functions -> C:\research_ios\kernelcaches\ios27ga\corpus
Flushing buffers, please wait...ok
Unloading IDP module C:\Program Files\IDA Professional 9.4\procs\arm.dll...

The content of `extract_corpus_v2.py` script follows below:

```
$ cat $tool\extract_corpus_v2.py

#!/usr/bin/env python3
r"""extract_corpus_v2.py -- IDAPython script, run inside a HEADLESS IDA.

    # Windows
    & $ida -A "-S$tool\extract_corpus_v2.py --out $base\corpus" "-L$base\corpus_extract.log"
    "$base\kc_27.0ga.corpus.i64"

    # macOS
    idat -A -S"extract_corpus_v2.py --out ~/research/corpus/24A437" kc_27.0ga.corpus.i64

"""
import sys, os

out_dir = None
for i, arg in enumerate(sys.argv):
    if arg == "--out" and i + 1 < len(sys.argv):
        out_dir = sys.argv[i + 1]

# Also check idc.ARGV
import idc
for i, arg in enumerate(idc.ARGV):
    if arg == "--out" and i + 1 < len(idc.ARGV):
        out_dir = idc.ARGV[i + 1]

if not out_dir:
    print("ERROR: pass --out <directory>")
    idc.qexit(1)

import idaapi
idaapi.auto_wait()

# Step 1: Run Cellebrite data offset initialization
```

```
print("=" * 60)
print("STEP 1: Cellebrite data offset initialization")
print("=" * 60)
try:
    sys.path.insert(0, os.path.dirname(idc.get_idb_path()))
    import ida_kernelcache as kc
    print("  Initializing data offsets...")
    kc.offset.initialize_data_offsets()
    idaapi.auto_wait()
    print("  Data offsets initialized.")
except Exception as e:
    print("  Cellebrite plugin error (non-fatal): %s" % str(e))
    print("  Continuing without Cellebrite processing.")

# Step 2: Run corpus extraction (inline v2 logic)
print("")
print("=" * 60)
print("STEP 2: Corpus extraction")
print("=" * 60)

import idautils, ida_hexrays, ida_funcs, ida_segment, ida_bytes
import json, hashlib

os.makedirs(out_dir, exist_ok=True)

def get_string_at(ea, max_len=256):
    s = idc.get_strlit_contents(ea, max_len, idc.STRTYPE_C)
    if s:
        try:
            return s.decode('utf-8', errors='replace')
        except:
            return None
    return None

def collect_data_refs_and_strings(func):
    data_refs = set()
    string_refs = []
    ea = func.start_ea
    while ea < func.end_ea and ea != idc.BADADDR:
        # Use XrefsFrom with flags=0 to get ALL xrefs (code + data)
        for xref in idautils.XrefsFrom(ea, 0):
            target = xref.to
            # Skip code flow xrefs (we capture those separately as calls)
            if xref.type in (idc.fl_CF, idc.fl_CN, idc.fl_JF, idc.fl_JN, idc.fl_F):
                continue
            data_refs.add(target)
            seg = ida_segment.getseg(target)
            if seg:
                sname = idc.get_segm_name(target)
                if '__cstring' in sname or '__os_log' in sname:
                    s = get_string_at(target)
                    if s and len(s) > 2:
                        string_refs.append(s[:200])
            ea = idc.next_head(ea, func.end_ea)
    return sorted(data_refs), list(dict.fromkeys(string_refs))

func_list = list(idautils.Functions())
num_funcs = len(func_list)
print("  Total functions: %d" % num_funcs)

addr_map = {}
total = 0

for func_ea in func_list:
    func = ida_funcs.get_func(func_ea)
    if not func:
        continue
```

```
name = idc.get_func_name(func_ea)
if not name:
    name = "sub_%X" % func_ea
safe_name = name.replace("/", "_").replace(":", "_").replace("<", "_").replace(">",
"_").replace("*", "_").replace("?", "_").replace("'", "_")
if len(safe_name) > 200:
    safe_name = safe_name[:200]

func_dir = os.path.join(out_dir, safe_name)
try:
    os.makedirs(func_dir, exist_ok=True)
except OSError:
    safe_name = "func_%X" % func_ea
    func_dir = os.path.join(out_dir, safe_name)
    os.makedirs(func_dir, exist_ok=True)

# disassembly
asm_lines = []
ea = func.start_ea
while ea < func.end_ea and ea != idc.BADADDR:
    asm_lines.append("0x%X: %s" % (ea, idc.GetDisasm(ea)))
    ea = idc.next_head(ea, func.end_ea)
with open(os.path.join(func_dir, "function.asm"), "w", encoding="utf-8", errors="replace") as f:
    f.write("\n".join(asm_lines) + "\n")

# pseudocode
pseudo = ""
try:
    cfunc = ida_hexrays.decompile(func_ea)
    if cfunc:
        pseudo = str(cfunc)
        if isinstance(pseudo, bytes):
            pseudo = pseudo.decode('utf-8', errors='replace')
        else:
            pseudo = pseudo.encode('utf-8', errors='replace').decode('utf-8')
except:
    pseudo = "// decompilation failed"
with open(os.path.join(func_dir, "function.c"), "w", encoding="utf-8", errors="replace") as f:
    f.write(pseudo + "\n")

# outgoing calls
calls_out_set = set()
ea_iter = func.start_ea
while ea_iter < func.end_ea and ea_iter != idc.BADADDR:
    for xref in idutils.XrefsFrom(ea_iter, 0):
        if xref.type in (idc.fl_CF, idc.fl_CN, idc.fl_JF, idc.fl_JN):
            target_func = ida_funcs.get_func(xref.to)
            if target_func and target_func.start_ea != func_ea:
                tname = idc.get_func_name(target_func.start_ea)
                if tname:
                    calls_out_set.add(tname)
    ea_iter = idc.next_head(ea_iter, func.end_ea)

# incoming calls
called_by_set = set()
for xref in idutils.XrefsTo(func_ea, 0):
    caller_func = ida_funcs.get_func(xref.frm)
    if caller_func and caller_func.start_ea != func_ea:
        cname = idc.get_func_name(caller_func.start_ea)
        if cname:
            called_by_set.add(cname)

# data refs and strings
data_refs, string_refs = collect_data_refs_and_strings(func)

seg_name = idc.get_segname(func_ea)
```

```
meta = {
    "name": name,
    "address": "0x%X" % func_ea,
    "end_address": "0x%X" % func.end_ea,
    "size": func.end_ea - func.start_ea,
    "segment": seg_name,
    "body_hash": hashlib.sha256(pseudo.encode()).hexdigest(),
    "calls_out": sorted(calls_out_set),
    "called_by": sorted(called_by_set),
    "data_refs": ["0x%X" % r for r in data_refs[:300]],
    "string_refs": string_refs[:50],
}
with open(os.path.join(func_dir, "function.json"), "w", encoding="utf-8", errors="replace") as f:
    json.dump(meta, f, indent=2)

addr_map[name] = {
    "start": "0x%X" % func_ea,
    "end": "0x%X" % func.end_ea,
    "segment": seg_name,
}

total += 1
if total % 500 == 0:
    pct = 100.0 * total / num_funcs
    print(" extracted %d / %d (%.1f%%)" % (total, num_funcs, pct))

with open(os.path.join(out_dir, "_address_map.json"), "w", encoding="utf-8") as f:
    json.dump(addr_map, f)

print("Done: %d functions -> %s" % (total, out_dir))

idc.qexit(0)
```

Step 06 # build the index: this step makes the corpus searchable by text and also by relation, and once again it will be necessary to use script to do it:

```
$ cd C:\research_ios\kernelcaches\corpus_tools
$ python index_corpus_v3.py C:\research_ios\kernelcaches\ios27ga\corpus
C:\research_ios\kernelcaches\ios27ga\corpus_ga.db
$ ls -lh C:\research_ios\kernelcaches\ios27ga\corpus_ga.db
-rw-r--r-- 1 Administrator None 2.6G Sep 21 20:43
'C:\research_ios\kernelcaches\ios27ga\corpus_ga.db'
```

The content of the `index_corpus_v3.py` script follows:

```
$ cat index_corpus_v3.py
#!/usr/bin/env python3
"""index_corpus_v3.py: build the index.
"""

import hashlib
import os
import re
import sqlite3
import sys

# ---- normalization -----
# Each pattern removes something a RELOCATION changes and nothing a CODE CHANGE changes.

_AUTONAME = re.compile(
    r"(sub|loc|locret|def|jpt|unk|byte|word|dword|qword|xmmword|off|stru|algn|asc)_"
    r"[0-9A-Fa-f]{4,16}\b")
_KADDR = re.compile(r"\b0x[fF]{3,}[0-9A-Fa-f]{6,}\b")
```

```

_SEGREL = re.compile(r"\b(?:__TEXT_EXEC|__DATA_CONST|__PRELINK_TEXT):[0-9A-Fa-f]+\b")

def normalise(pseudo):
    """Pseudocode with build specific addressing removed, so two builds of one function agree."""
    t = _AUTONAME.sub(lambda m: m.group(1) + "_@", pseudo)
    t = _KADDR.sub("0xK", t)
    t = _SEGREL.sub("SEG:@", t)
    return t

def norm_hash(pseudo):
    if not pseudo:
        return ""
    return hashlib.sha256(normalise(pseudo).encode("utf-8", "replace")).hexdigest()

# ---- controls -----

def selftest():
    """A normalisation that collapses everything passes (1) and fails (2). Both must hold."""
    a = ("__int64 f(void){ v1 = sub_FFFFFFFF008434000(); "
        "v2 = qword_FFFFFFFF0084A1B20; return v1 + 0xFFFFFFFF008434000; }")
    # the SAME function relocated: every address differs, the code does not
    b = ("__int64 f(void){ v1 = sub_FFFFFFFF0098BCA94(); "
        "v2 = qword_FFFFFFFF00A2E7F58; return v1 + 0xFFFFFFFF0098BCA94; }")
    # a REAL change: one constant that is not an address
    c = ("__int64 f(void){ v1 = sub_FFFFFFFF008434000(); "
        "v2 = qword_FFFFFFFF0084A1B20; return v1 + 0x101; }")
    # and a pair that a careless rule would merge - v3 against v30
    d = "__int64 f(void){ return v3; }"
    e = "__int64 f(void){ return v30; }"

    c1 = norm_hash(a) == norm_hash(b)
    c2 = norm_hash(a) != norm_hash(c)
    c3 = norm_hash(d) != norm_hash(e)
    c4 = hashlib.sha256(a.encode()).hexdigest() != hashlib.sha256(b.encode()).hexdigest()
    print("C1 relocation collapses (same norm_hash) %s" % ("PASS" if c1 else "FAIL"))
    print("C2 a real constant change does NOT collapse %s" % ("PASS" if c2 else "FAIL"))
    print("C3 v3 and v30 stay distinct %s" % ("PASS" if c3 else "FAIL"))
    print("C4 NEGATIVE: raw body_hash DOES differ on the pair %s" % ("PASS" if c4 else "FAIL"))
    print(" - C4 is what makes C1 mean something: the inputs really are different text")
    ok = c1 and c2 and c3 and c4
    print("\n%s" % ("ALL CONTROLS PASS" if ok else "***** A CONTROL FAILED"))
    return 0 if ok else 1

SCHEMA = """
CREATE TABLE IF NOT EXISTS functions (
    name TEXT PRIMARY KEY,
    address TEXT,
    end_address TEXT,
    size INTEGER,
    segment TEXT,
    body_hash TEXT,
    norm_hash TEXT,
    pseudo TEXT,
    asm TEXT
);
CREATE TABLE IF NOT EXISTS calls (caller TEXT NOT NULL, callee TEXT NOT NULL);
CREATE TABLE IF NOT EXISTS data_refs (func_name TEXT NOT NULL, ref_addr TEXT NOT NULL);
CREATE TABLE IF NOT EXISTS string_refs (func_name TEXT NOT NULL, string TEXT NOT NULL);
CREATE TABLE IF NOT EXISTS addresses (addr_start TEXT NOT NULL, addr_end TEXT NOT NULL,
    func_name TEXT NOT NULL);
CREATE VIRTUAL TABLE IF NOT EXISTS pseudo_fts USING fts5(name, segment, pseudo);
CREATE VIRTUAL TABLE IF NOT EXISTS strings_fts USING fts5(func_name, string);
CREATE VIRTUAL TABLE IF NOT EXISTS asm_fts USING fts5(name, asm);

def main():
    if len(sys.argv) == 2 and sys.argv[1] == "--selftest":

```

```
    return selftest()
if len(sys.argv) != 3:
    print("Usage: index_corpus_v3.py <corpus_dir> <output.db>")
    print("    index_corpus_v3.py --selftest")
    return 1
corpus_dir, db_path = sys.argv[1], sys.argv[2]

print("controls first, because an index nobody calibrated is not evidence")
if selftest() != 0:
    print("***** REFUSING to index - normalisation is not behaving")
    return 1
print("")

conn = sqlite3.connect(db_path)
conn.execute("PRAGMA journal_mode=WAL")
conn.execute("PRAGMA synchronous=NORMAL")
conn.executescript(SCHEMA)
conn.executescript(WIPE)

entries = sorted(os.listdir(corpus_dir))
total = len(entries)
count = 0
call_batch, dref_batch, sref_batch, addr_batch = [], [], [], []
nh_have = 0
skipped = []

for entry in entries:
    d = os.path.join(corpus_dir, entry)
    mp = os.path.join(d, "function.json")
    if not os.path.isfile(mp):
        continue
    import json

    try:
        with open(mp) as f:
            meta = json.load(f)
    except Exception:
        skipped.append(entry)
        continue
    pseudo = ""
    pp = os.path.join(d, "function.c")
    if os.path.isfile(pp):
        with open(pp, errors="replace") as f:
            pseudo = f.read()
    asm = ""
    ap = os.path.join(d, "function.asm")
    if os.path.isfile(ap):
        with open(ap, errors="replace") as f:
            asm = f.read()

    name = meta["name"]
    address = meta.get("address", "")
    end_address = meta.get("end_address", address)
    segment = meta.get("segment", "")
    nh = norm_hash(pseudo)
    if nh:
        nh_have += 1

    conn.execute(
        "INSERT OR REPLACE INTO functions VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)",
        (name, address, end_address, meta.get("size", 0), segment,
         meta.get("body_hash", ""), nh, pseudo, asm))
    conn.execute("INSERT INTO pseudo_fts VALUES (?, ?, ?)", (name, segment, pseudo))
    if asm:
        conn.execute("INSERT INTO asm_fts VALUES (?, ?)", (name, asm))
    for callee in meta.get("calls_out", []):
        call_batch.append((name, callee))
```

```

for dref in meta.get("data_refs", []):
    dref_batch.append((name, dref))
for sref in meta.get("string_refs", []):
    sref_batch.append((name, sref))
    conn.execute("INSERT INTO strings_fts VALUES (?,?)", (name, sref))
addr_batch.append((address, end_address, name))
count += 1
if count % 5000 == 0:
    print(" indexed %d / %d (%.1f%%)" % (count, total, 100.0 * count / total))

print("call graph %d edges" % len(call_batch))
conn.executemany("INSERT INTO calls VALUES (?,?)", call_batch)
print("data refs %d" % len(dref_batch))
conn.executemany("INSERT INTO data_refs VALUES (?,?)", dref_batch)
print("string refs %d" % len(sref_batch))
conn.executemany("INSERT INTO string_refs VALUES (?,?)", sref_batch)
print("addresses %d" % len(addr_batch))
conn.executemany("INSERT INTO addresses VALUES (?,?,?)", addr_batch)

for stmt in (
    "CREATE INDEX IF NOT EXISTS idx_calls_caller ON calls(caller)",
    "CREATE INDEX IF NOT EXISTS idx_calls_callee ON calls(callee)",
    "CREATE INDEX IF NOT EXISTS idx_dref_addr ON data_refs(ref_addr)",
    "CREATE INDEX IF NOT EXISTS idx_dref_func ON data_refs(func_name)",
    "CREATE INDEX IF NOT EXISTS idx_sref_func ON string_refs(func_name)",
    "CREATE INDEX IF NOT EXISTS idx_addr_start ON addresses(addr_start)",
    "CREATE INDEX IF NOT EXISTS idx_addr_name ON addresses(func_name)",
    "CREATE INDEX IF NOT EXISTS idx_func_norm ON functions(norm_hash)",
):
    conn.execute(stmt)
conn.commit()
conn.close()
print("\nDone: %d functions -> %s" % (count, db_path))
print(" %d carry a norm_hash (a function with no pseudocode gets an empty one)" % nh_have)
return 0

if __name__ == "__main__":
    sys.exit(main())

```

Step 07 # Queries: this step is the final one and the actual payoff, where we can do simple SQL queries to discover and learn from database instead of time consuming searches on IDA Pro database:

How many functions are searchable by the index.
 \$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT COUNT(*), SUM(norm_hash != '') FROM functions;"

COUNT(*)	SUM(norm_hash ≠ '')
215839	215839

Search for the decompiled C of every function for the token "IOUserClient", returning
 # the function name and the next it lives in.
 \$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT name, segment FROM pseudo_fts WHERE pseudo MATCH 'IOUserClient' LIMIT 6;"

name	segment
ASIOKit_InitFunc_0	com.apple.driver.ASIOKit: __TEXT_EXEC.__text
AUC_InitFunc_1	com.apple.AUC: __TEXT_EXEC.__text
AppleAOPAudio_InitFunc_10	com.apple.driver.AppleAOPAudio: __TEXT_EXEC.__text
AppleAOPAudio_InitFunc_22	com.apple.driver.AppleAOPAudio: __TEXT_EXEC.__text
AppleAOPAudio_InitFunc_27	com.apple.driver.AppleAOPAudio: __TEXT_EXEC.__text
AppleARMPlatform_InitFunc_12	com.apple.driver.AppleARMPlatform: __TEXT_EXEC.__text

Counts the number of function whose disassembly contains pacibsp instruction.

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT COUNT(*) FROM asm_fts WHERE asm MATCH 'pacibsp';"

COUNT(*)
115176

Show who call this function.

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT caller FROM calls WHERE callee = 'kfree_ext' LIMIT 6;"

caller
IOCircularDataQueueCreateWithEntries
IOFindBSDRoot
__Z16IODTSetResolvingP15IORegistryEntryPFxjPjS1_EPFvS0_PhS4_S4_E
__ZL340SActionCreateWithTypeInternalP80SObjectyymP80SStringbPP80SAction
__ZN10IOSKRegion4freeEv
__ZN11OSMetaClass11postModLoadEPv

It turns an address into a function name (used when analyzing panic).

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT func_name FROM addresses WHERE addr_start <= '0xFFFFFFFF008444400' ORDER BY addr_start DESC LIMIT 1;"

func_name
AFKHIDTBDevice_TermFunc_0

List functions that reference a string containing "gpu-restricted".

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT DISTINCT func_name FROM string_refs WHERE string LIKE '%gpu-restricted%' ORDER BY func_name;"

func_name
IOGPUFamily_InitFunc_18
IOSurface_InitFunc_11
__ZN11IOGPUDevice16copyEntitlementsEv
__ZN11IOGPUDevice18isRestrictedClientEv
__ZN11IOGPUDevice20noIgnoreOnGPURestartEv
__ZN19IOSurfaceMemoryPool14taskCanUsePoolEP4task
__ZNK11IOGPUDevice9method_16Ev_99
__ZNK23IOSurfaceRootUserClient4initEv_99
sub_FFFFFFFF00A041DD0
sub_FFFFFFFF00A041F14
sub_FFFFFFFF00A34B624
sub_FFFFFFFF00A34B9C0

how many functions mentioned an entitlement.

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT COUNT(DISTINCT func_name) FROM string_refs WHERE string LIKE '%entitlement%';"

COUNT(DISTINCT fu ...
886

Track a panic address to a function.

\$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT func_name, addr_start, addr_end FROM addresses WHERE addr_start <= '0xFFFFFFFF008444400' ORDER BY addr_start DESC LIMIT 1;"

func_name	addr_start	addr_end
AFKHIDTBDevice_TermFunc_0	0xFFFFFFFF0084443E4	0xFFFFFFFF008444408

```
# Rank attack surface by size.
```

```
$ C:\sqlite\sqlite3.exe corpus_ga.db "SELECT segment, COUNT(*) c FROM functions GROUP BY segment ORDER BY c DESC LIMIT 10;"
```

segment	c
com.apple.kernel:__text	19824
com.apple.driver.AppleM2ScalerCSCDriver:__TEXT_EXEC.__text	10133
com.apple.driver.FairPlayIOKit:__TEXT_EXEC.__text	7142
com.apple.iokit.IOThunderboltFamily:__TEXT_EXEC.__text	5632
com.apple.iokit.IOAccessoryManager:__TEXT_EXEC.__text	4964
com.apple.driver.AppleH16ANEInterface:__TEXT_EXEC.__text	4827
com.apple.iokit.IONVMeFamily:__TEXT_EXEC.__text	3554
com.apple.iokit.IOAVFamily:__TEXT_EXEC.__text	3356
com.apple.AGXG17P:__TEXT_EXEC.__text	3082
com.apple.driver.AppleAVE2:__TEXT_EXEC.__text	2885

I have not provided readers with the context where these queries have been done, and my only idea is to show how useful the technique and mechanisms are.

11. Kernelcache and IPSW diffing

One aspect we have not covered yet is the value of using diffing mechanisms. There are four possible ways to do it, and each of them offers a perspective and important information:

- **Option 01:** we can use the SQLite diffing approach to query what functions bodies have changed between two kernelcaches. We get a series of benefits using this approach such as learning the reachability, full search over assembly and pseudocode, and string and data references. On the other hand, we do not have visibility of changed functions that do not have name. The cost of this operation is the amount of time to build two pair of corpus.
- **Option 02:** we can use a script to search symbols that have changed between two different of kernelcaches. The benefit of using this approach is to learn new code introduced in each new version. On the other hand, it does not offer information about any function without name, no information about reachability or body text. The cost is low because it takes only a few minutes, and does not demand IDA Pro.
- **Option 03:** we can use ipsw diff, which offers answers for entitlements, compiled sandbox profiles, strings, launchd configs, function starts, firmware, and everything across the whole IPSW image such as user-space and coprocessors. On the other hand, it does not offer a kernelcache specific view (at the time of this writing). The cost is the requirement of downloading about 12 GB per build.
- **Option 04:** we can use Diaphora, which offers semantic interpretation and also survives to inline functions, register reallocation and stack relay out. At this time, it is the only approach that offer information about changes in unnamed functions. The cost is the investment of some hours of IDA per build.

This section will cover the respective practical walkthroughs for each option.

Option 01 # SQLite Diffing:

As we already have a corpus for iOS 27.0 GA, we need to build a corpus and an index for the second build, exactly like we did previously:

```
$ $ida = "C:\Program Files\IDA Professional 9.4\idat.exe"
$ $b = "C:\research_ios\kernelcaches\ios272b1"
$ $tool = "C:\research_ios\kernelcaches\corpus_tools"

$ (build the database) & $ida -A "-S$b\build_idb_272.py" "-L$b\ida94_272.log" "$b\kc_27.2b1.bin"

$ (only if the database is not closed for any reason) Get-ChildItem
"$b\kc_27.2b1.bin.i64", "$b\kc_27.2b1.bin.id?", "$b\kc_27.2b1.bin.nam", "$b\kc_27.2b1.bin.til" -EA
SilentlyContinue

$ (make a database copy) Copy-Item "$b\kc_27.2b1.bin.i64" "$b\kc_27.2b1.corpus.i64"

$ (extract the corpus) & $ida -A "-S$tool\extract_corpus_v2.py --out $b\corpus" "-
L$b\corpus_extract.log" "$b\kc_27.2b1.corpus.i64"

$ (build index of the corpus) python $tool\index_corpus_v3.py $b\corpus $b\corpus_272.db
```

controls first, because an index nobody calibrated is not evidence

```
C1 relocation collapses (same norm_hash) PASS
C2 a real constant change does NOT collapse PASS
C3 v3 and v30 stay distinct PASS
C4 NEGATIVE: raw body_hash DOES differ on the pair PASS
- C4 is what makes C1 mean something: the inputs really are different text
```

ALL CONTROLS PASS

```
indexed 5000 / 217451 (2.3%)
indexed 10000 / 217451 (4.6%)
indexed 15000 / 217451 (6.9%)
...
indexed 210000 / 217451 (96.6%)
indexed 215000 / 217451 (98.9%)
call graph 449425 edges
data refs 745705
string refs 225618
addresses 217450
```

```
Done: 217450 functions -> C:\research_ios\kernelcaches\ios272b1\corpus_272.db
217450 carry a norm_hash (a function with no pseudocode gets an empty one)
```

Check if it is possible to make a join between both corpus:

```
$ C:\sqlite\sqlite3.exe C:\research_ios\kernelcaches\ios27ga\corpus_ga.db "ATTACH
'C:\research_ios\kernelcaches\ios272b1\corpus_272.db' AS newer; SELECT COUNT(*) FROM functions a
JOIN newer.functions b USING (name) WHERE a.norm_hash != b.norm_hash;"
```

COUNT(*)
11337

To get something more useful, we can do the following:

```
$ python C:\research_ios\kernelcaches\corpus_tools\mkdiff.py
C:\research_ios\kernelcaches\ios27ga\corpus_ga.db
C:\research_ios\kernelcaches\ios272b1\corpus_272.db C:\research_ios\kernelcaches\diff_ga_272.db
```

```
joining on name...
counting callers in the OLD build...
```

```
rows                305424
changed             11337
  of which LABEL NOISE only    2820
  of which REAL               8517
  REAL and gained a comparison 1218
new (named only)      2076
gone (named only)    1791
```

CONTROLS ON THE RESULT

```
C1 both noise classes are populated    PASS
  - a denoiser that collapses everything passes nothing
C2 comparisons move BOTH ways          PASS (+1218 / -939)
C3 new AND gone are both populated      PASS
```

```
-> C:\research_ios\kernelcaches\diff_ga_272.db (66.5 MB)
```

According to the output above:

- There are 11337 names that present in iOS 27.2 beta 1 and iOS 27.0 GA.
- 2820 of those are IDA Label noise, and not code.
- 8517 represents real changes.
- 1218 of 8517 are changes that also gained a comparison operator. In a list ordered by likelihood (and not fixed)
- 2076 named functions are new in iOS 27.2 beta 1.
- 1791 named functions existed in iOS 27.0 GA and does not exist in the current version (iOS 27.2 beta 1).

The goal of the report above is to provide us with a first picture of how much the code really changed between two provided versions. As you can see, numbers above are statistics, but they drive us during the research and establish an initial perspective.

From the exposed context, we can proceed and learn more about what really matters. Likely, the first information we want to know about is where Apple has worked on the new version of iOS (iOS 27.2 beta 1), and you should run this query before any other query:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT
segment, COUNT(*) AS changed, SUM(cmp_delta>0) AS gained FROM changes WHERE kind='changed' AND
noise=0 GROUP BY segment ORDER BY changed DESC LIMIT 15;"
```

segment	changed	gained
com.apple.kernel:__text	1062	129
com.apple.driver.AppleBasebandPCI:__TEXT_EXEC.__text	686	204
com.apple.driver.AppleH16ANEInterface:__TEXT_EXEC.__text	669	125
com.apple.driver.AppleM2ScalerCSCDriver:__TEXT_EXEC.__text	657	36
com.apple.driver.AppleHIDTransport:__TEXT_EXEC.__text	362	6
com.apple.driver.AppleBasebandPCIMAVControl:__TEXT_EXEC.__text	343	95
com.apple.iokit.IONVMeFamily:__TEXT_EXEC.__text	269	19
com.apple.driver.ApplePMGR:__TEXT_EXEC.__text	260	21
com.apple.driver.AppleMobileDispH17P-DCP:__TEXT_EXEC.__text	163	15
com.apple.iokit.IOUSBHostFamily:__TEXT_EXEC.__text	144	7
com.apple.driver.ApplePPMCPMS:__TEXT_EXEC.__text	135	17
com.apple.driver.AppleDisplayCrossbar:__TEXT_EXEC.__text	125	15
com.apple.driver.FairPlayIOKit:__TEXT_EXEC.__text	113	30
com.apple.driver.AppleAVE2:__TEXT_EXEC.__text	110	2
com.apple.driver.RTBuddy:__TEXT_EXEC.__text	105	3

From the output above, considering only the second line, baseband seems to have gained 204 checks across 686 changed functions, which represents a very dense value of checks, and remember that is a remote surface. On the other hand, AppleM2ScalerCSCDriver had 657 of its functions changed, but it gained a much smaller number of checks, which is closer to recompilation changes.

As Apple never tells what has changed, it is also recommended to pay attention to functions that gained a comparison (potential checks) and are called often, and the number of caller by functions can help us to guide our attention:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name, callers, cmp_delta, delta FROM changes WHERE kind='changed' AND noise=0 AND cmp_delta>0 AND name NOT LIKE 'sub_%' ORDER BY callers DESC, cmp_delta DESC LIMIT 15;"
```

name	callers	cmp_delta	delta
__ZN25IOUSBDeviceControlRequest17stallSetupRequestEv	18	1	0
ip6_output_list	17	9	176
necp_socket_find_policy_match	17	8	104
__ZN12ANEScheduler13submitRequestERKNSt3__110shared_ptrI10...	15	3	172
vm_map_remove_upl_range	13	9	268
vn_open_auth	12	4	76
xnuupcallsv2_conclaveupcallsprivate_suspend__completion_send	10	8	264
vm_fault_internal	10	3	-148
vm_map_remap	7	7	196
in6_selectsrc	7	5	40
__ZN12ANEScheduler20removePowerAssertionE12ANEResidency21A...	7	4	-4
pbuf_resize_segment	7	2	32
__Z34corecaptureIsDevFused0rCSRInternalb	7	1	20
__ZN17AppleSPUHIDDriver11handleStartEP9IOService	7	1	24
entropy_init	6	8	212

At the output above, cmp_delta column is the number of logical comparison added or subtracted from an old version to the new version of the iOS, and delta column is the size between the start and end of each function, which divided by 4 (considering ARM64) helps us to understand how many instructions were potentially added or removed. Callers column refers to direct calls with a resolved name and counted where the function exists: the old iOS build for changed or no longer existing functions, and the new build for recently introduced functions.

It is necessary to know how to interpret the output. The first line shows only one check and zero bytes of changes, so we should discard it initially. On the other hand, more cmp_delta (checkers), callers, and bytes changes may mean better candidates, but not always. As an example, a line where there is a low number of added checks, but an expressive change of size (increasing or decreasing) may be an indicative of an interesting target. At the same way, a meaningful increase in checks and bytes, but zero callers, may be resulting from a rewrite and not a fix.

A different point of view can be ordered by getting functions ordered by the number of added checks:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name, cmp_delta, delta, callers FROM changes WHERE kind='changed' AND noise=0 AND name NOT LIKE 'sub_%' ORDER BY cmp_delta DESC LIMIT 15;"
```

name	cmp_delta	delta	callers
__ZN30AppleBasebandPCIPDPManagerBase5startEPS_	218	8292	0
__ZNK23AppleBasebandPCIControl10method_204Ev_99	163	3400	0
__ZNK23AppleBasebandPCIControl10method_211Ev_99	155	4464	0
__ZNK25AppleBasebandPCIMHIDevice10method_198Ev_99	149	3780	0
__ZNK28AppleBasebandPCIMHIInterface10method_172Ev_99	141	5040	0

__ZNK23AppleBasebandPCIControl10method_195Ev_99	135	3444	1
__ZNK19ApplePearlSEPDriver10method_217Ev_99	134	2336	0
__ZNK28AppleBasebandPCIMHIInterface10method_204Ev_99	132	2992	0
__ZNK23AppleBasebandPCIControl10method_193Ev_99	130	3212	0
__ZN23AppleBasebandPCIControl5startEPS_	111	4212	1
__ZNK28AppleBasebandPCIQMAPProvider10method_170Ev_99	105	2996	0
__ZN28AppleBasebandPCIMHIInterface5startEPS_	99	3180	0
__ZNK25AppleBasebandPCIMHIDevice10method_197Ev_99	95	2804	0
__ZNK23AppleBasebandPCIControl10method_222Ev_99	93	3484	0
__ZNK28AppleBasebandPCIQMAPProvider10method_169Ev_99	92	3084	0

Another important query is to get functions ordered by the number of missed checks:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name, cmp_delta, delta, callers FROM changes WHERE kind='changed' AND noise=0 AND name NOT LIKE 'sub_%' ORDER BY cmp_delta ASC LIMIT 15;"
```

name	cmp_delta	delta	callers
upl_commit_range	-331	-5336	17
__ZNK19ApplePearlSEPDriver10method_216Ev_99	-204	-3728	0
_lifs_vnop_close	-169	-5396	0
in6_multi_detach	-95	-2120	2
__ZNK20IOBlockStorageDriver11handleStartEv_99	-93	-2276	0
__ZNK19AppleUSBCController8finalizeEv_99	-90	0	0
__ZNK23AppleBasebandPCIControl10method_203Ev_99	-86	-1192	0
__ZNK12ApplePPMCPMS10method_232Ev_99	-80	-2468	0
__ZNK9ApplePMGR10method_333Ev_99	-73	-1676	1
__ZNSt3__122__libcpp_verbose_abortEPKcz	-66	-2052	1
necp_create_agent_type_to_id_mapping	-64	-516	0
__ZN21AppleARMIIISController5startEP9IOService	-63	0	1
__ZNK23AppleBasebandPCIControl10method_221Ev_99	-58	-1620	0
__ZN14MCPolicyMgrPMPC1EjP9IOService	-56	0	0
__ZNK23AppleBasebandPCIControl10method_192Ev_99	-56	-824	0

Once again, interpretations of the third and fourth outputs are equally relevant. In the third output, a significant number of checks and increase in delta with very low or zero callers may mean a complete rewrite of the function, and no changes were introduced. In the fourth output, high values of cmp_delta and delta may indicate that the original function has been split or decomposed rather than having its checks removed.

Next query show what functions did not exist in iOS 27.0 and was added in iOS 27.2, which can deserve a close check for vulnerabilities:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name, new_size, callers FROM changes c WHERE kind='new' AND name NOT LIKE 'sub_%' AND name NOT LIKE '%_InitFunc_%' AND name NOT LIKE '%\__99' ESCAPE '\ ' AND name NOT LIKE '%\__99\_' ESCAPE '\ ' AND NOT EXISTS(SELECT 1 FROM changes g WHERE g.kind='gone' AND g.old_size=c.new_size AND g.name NOT LIKE 'sub_%') ORDER BY new_size DESC LIMIT 15;"
```

name	new_size	callers
_lifs_vnop_getattr	5044	1
proc_core_name	4120	2
__ZN18ANEProgramResourceD2Ev	3084	4
do_kern_dump	2740	2
coredump	2644	2
kdp_raise_exception	2324	1
__ZN19ANECClientDeviceCore34ANE_ProgramPrepareAndSubmit...	2212	1
__Z16IOPolledFileOpenPKcjyyyPvmPP18IOPolledFileIOVarsP...	2128	1
kern_coredump_routine	2024	2
__ZN22EndpointSecurityDriver13newUserClientEP4taskPvjP...	1768	0
kdp_panic_dump	1552	1
_AEAKernelEncryptInitializeState	1504	0

```
__ZN22EndpointSecurityDriver5startEPS_
kdp_core_init
```

```
1464    0
1276    1
```

Next query only focus on a few targets to understand how many functions changed, disappeared or are really new:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT
segment, kind, COUNT(*) AS n FROM changes WHERE noise=0 AND name NOT LIKE 'sub_%' AND (segment LIKE
'%IOSurface%' OR segment LIKE '%AGX%' OR segment LIKE '%lifs%') GROUP BY segment, kind ORDER BY n
DESC;"
```

segment	kind	n
com.apple.filesystems.lifs:__TEXT_EXEC.__text	changed	98
com.apple.AGXG17P:__TEXT_EXEC.__text	changed	86
com.apple.AGXG17P:__TEXT_EXEC.__text	gone	68
com.apple.AGXG17P:__TEXT_EXEC.__text	new	68
com.apple.iokit.IOSurface:__TEXT_EXEC.__text	changed	15
com.apple.filesystems.lifs:__TEXT_EXEC.__auth_stubs	new	4
com.apple.filesystems.lifs:__TEXT_EXEC.__text	gone	1
com.apple.filesystems.lifs:__TEXT_EXEC.__text	new	1

To check what IOSurface functions really differs between the two builds, execute:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name,
kind, cmp_delta, delta, callers FROM changes WHERE noise=0 AND name NOT LIKE 'sub_%' AND (segment
LIKE '%IOSurface%' OR segment LIKE '%AGX%' OR segment LIKE '%lifs%') ORDER BY (cmp_delta!=0) DESC,
ABS(delta) DESC LIMIT 15;"
```

name	kind	cmp_delta	delta	callers
_lifs_vnop_close	changed	-169	-5396	0
_lifs_vnop_open	changed	2	84	0
__ZN14AGXAccelerator16restartWorkQueueEP12AGXWorkQ	changed	6	44	0
_lifs_endio_done	changed	1	28	1
__ZN13AGXStatistics15initIOReportingEv	changed	3	16	1
_lifs_endio_thread	changed	-1	-12	0
_lifs_io_strategy_thread	changed	-1	-4	0
_lifs_req_callback_thread	changed	-1	-4	0
__ZN10AGXChannel4freeEv	changed	-2	0	1
__ZN13AGXSecureGart4freeEv	changed	-1	0	0
__ZN13AGXSecureGart9initInnerEP14AGXAcceleratorP4t	changed	-1	0	2
__ZN14AGXAccelerator21sleepWakeNotificationEPvS0_j	changed	-4	0	1
__ZN14AGXAccelerator5startEP9IOService	changed	-3	0	1
__ZN23IOSurfaceRootUserClient4stopEP9IOService	changed	1	0	0
__ZN29AGXParameterManagementVirtual31prepareAndRet	changed	-4	0	0

Finally, if you need to list specific functions which you know only a part of the name, execute:

```
$ C:\sqlite\sqlite3.exe -header -column C:\research_ios\kernelcaches\diff_ga_272.db "SELECT name,
noise, cmp_delta, old_size, new_size FROM changes WHERE name LIKE '%orderFill%';"
```

name	noise	cmp_delta	old_size	new_size
__ZNK25M2ScalerSrcDestCfgControl18validateBorderF	1	0	48	48
__ZNK29M2ScalerSrcDestCfgControlMSR418validateBor	1	0	548	548
__ZNK29M2ScalerSrcDestCfgControlMSR818validateBor	1	0	88	88
__ZNK30M2ScalerSrcDestCfgControlMSR2318validateBo	1	0	652	652

Noise column indicates that the difference comes from an IDA label, and not a code change.

Before moving to the next diffing option ([Symbol and Size diff](#)), the code of mkdiff.py follows below:

```
$ cat C:\research_ios\kernelcaches\corpus_tools\mkdiff.py
```

```
#!/usr/bin/env python3
```



```
r"""mkdiff.py - materialize the differential between TWO corpus indexes, denoised and ranked.
```

```
USAGE
```

```
python mkdiff.py <old_corpus.db> <new_corpus.db> <out.db>
python mkdiff.py --selftest
```

```
"""

import hashlib
import os
import re
import sqlite3
import sys

# index_corpus_v3.py's own three, reproduced so old and new are normalized identically

_AUTONAME = re.compile(
    r"\b(sub|loc|locret|def|jpt|unk|byte|word|dword|xmmword|qword|off|stru|algn|asc)_"
    r"[0-9A-Fa-f]{4,16}\b")
_KADDR = re.compile(r"\b0x[ff]{3,}[0-9A-Fa-f]{6,}\b")
_SEGREL = re.compile(r"\b(?:__TEXT_EXEC|__DATA_CONST|__PRELINK_TEXT):[0-9A-Fa-f]+\b")

_LABELSEQ = re.compile(r"\b(a[A-Z][A-Za-z0-9]*?)_d{1,4}\b")

_CMP = re.compile(r"<|=|>|=|<>")

SCHEMA = """
DROP TABLE IF EXISTS changes;
CREATE TABLE changes (
    name      TEXT PRIMARY KEY,
    kind      TEXT NOT NULL,
    segment   TEXT,
    old_size  INTEGER,
    new_size  INTEGER,
    delta     INTEGER,
    callers   INTEGER DEFAULT 0,
    cmp_delta INTEGER DEFAULT 0,
    noise     INTEGER DEFAULT 0
);
"""

INDEXES = ["CREATE INDEX idx_kind ON changes(kind)",
            "CREATE INDEX idx_seg ON changes(segment)",
            "CREATE INDEX idx_delta ON changes(delta)",
            "CREATE INDEX idx_cmp ON changes(cmp_delta)",
            "CREATE INDEX idx_noise ON changes(noise)"]

def denoise(t):
    """v3's normalisation PLUS the label sequence number it does not remove."""
    t = _AUTONAME.sub(lambda m: m.group(1) + "_@", t or "")
    t = _KADDR.sub("0xK", t)
    t = _SEGREL.sub("SEG:@", t)
    return _LABELSEQ.sub(lambda m: m.group(1) + "_#", t)

def dh(t):
    return hashlib.sha256(denoise(t).encode("utf-8", "replace")).hexdigest()

def build(old_db, new_db, out_db):
    for p in (old_db, new_db):
        if not os.path.isfile(p):
            print("no such corpus: %s" % p)
            return 1
    if os.path.exists(out_db):
        os.remove(out_db)

    src = sqlite3.connect(old_db)
    src.execute("ATTACH ? AS newer", (new_db,))
```

```
print(" joining on name...")
changed = src.execute(
    "SELECT a.name, a.segment, a.size, b.size, a.pseudo, b.pseudo "
    "FROM functions a JOIN newer.functions b USING (name) "
    "WHERE a.norm_hash != b.norm_hash AND a.norm_hash != '' AND b.norm_hash != '').fetchall()
new = src.execute("SELECT b.name, b.segment, b.size FROM newer.functions b "
    "LEFT JOIN functions a USING (name) WHERE a.name IS NULL").fetchall()
gone = src.execute("SELECT a.name, a.segment, a.size FROM functions a "
    "LEFT JOIN newer.functions b USING (name) WHERE b.name IS NULL").fetchall()
print(" counting callers in BOTH builds...")
# a row is credited from the build where the function EXISTS, so 'new' rows get a real figure
callers = dict(src.execute("SELECT callee, COUNT(*) FROM calls GROUP BY callee").fetchall())
callers_new = dict(src.execute(
    "SELECT callee, COUNT(*) FROM newer.calls GROUP BY callee").fetchall())

rows = []
for name, seg, sa, sb, pa, pb in changed:
    rows.append((name, "changed", seg, sa, sb, (sb or 0) - (sa or 0), callers.get(name, 0),
        len(_CMP.findall(pb or "")) - len(_CMP.findall(pa or "")),
        1 if dh(pa) == dh(pb) else 0))
for name, seg, sb in new:
    rows.append((name, "new", seg, None, sb, sb, callers_new.get(name, 0), 0, 0))
for name, seg, sa in gone:
    rows.append((name, "gone", seg, sa, None, -(sa or 0), callers.get(name, 0), 0, 0))

out = sqlite3.connect(out_db)
out.executescript(SCHEMA)
out.executemany("INSERT OR REPLACE INTO changes VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)", rows)
for ix in INDEXES:
    out.execute(ix)
out.commit()

q = out.execute
n = lambda w: q("SELECT COUNT(*) FROM changes WHERE " + w).fetchone()[0]
print()
print(" %-36s %7d" % ("rows", q("SELECT COUNT(*) FROM changes").fetchone()[0]))
print(" %-36s %7d" % ("changed", n("kind='changed'")))
print(" %-36s %7d" % (" of which LABEL NOISE only", n("kind='changed' AND noise=1")))
print(" %-36s %7d" % (" of which REAL", n("kind='changed' AND noise=0")))
print(" %-36s %7d" % (" REAL and gained a comparison", n("kind='changed' AND noise=0 AND
cmp_delta>0")))
print(" %-36s %7d" % ("new (named only)", n("kind='new' AND name NOT LIKE 'sub_%'")))
print(" %-36s %7d" % ("gone (named only)", n("kind='gone' AND name NOT LIKE 'sub_%'")))
print()
print(" CONTROLS ON THE RESULT")
c1 = n("kind='changed' AND noise=0") and n("kind='changed' AND noise=1")
c2 = n("cmp_delta>0") and n("cmp_delta<0")
c3 = n("kind='new'") and n("kind='gone'")
print(" C1 both noise classes are populated %s" % ("PASS" if c1 else "** FAIL **"))
print(" - a denoiser that collapses everything passes nothing")
print(" C2 comparisons move BOTH ways %s (+%d / -%d)"
    % ("PASS" if c2 else "** FAIL **", n("cmp_delta>0"), n("cmp_delta<0")))
print(" C3 new AND gone are both populated %s" % ("PASS" if c3 else "** FAIL **"))
c4 = all(n("kind='%s' AND callers>0" % k) for k in ("changed", "new", "gone"))
print(" C4 EVERY kind has callers populated %s (changed %d / new %d / gone %d)"
    % ("PASS" if c4 else "** FAIL **", n("kind='changed' AND callers>0"),
        n("kind='new' AND callers>0"), n("kind='gone' AND callers>0")))
print(" - this failed until 2026-09-22: 'new' rows were hardcoded to 0, so 147,849")
print(" rows read as 'nothing calls this' and U5 could not be ranked at all")
print()
print(" -> %s (%.1f MB)" % (out_db, os.path.getsize(out_db) / 1048576.0))
return 0 if (c1 and c2 and c3 and c4) else 1

def selftest():
    """The denoiser must remove a LABEL sequence and must NOT remove a real change."""
    a = "__int64 f(void){ return sub_FFFFFFFF009075108(5, aLibraryCachesC_406, 106); }"
```

```
# the SAME function in another build: address moved, label sequence moved, code identical
b = "__int64 f(void){ return sub_FFFFFFFF009151980(5, aLibraryCachesC_408, 106); }"
# a REAL change: a different constant
c = "__int64 f(void){ return sub_FFFFFFFF009075108(9, aLibraryCachesC_406, 106); }"
# a REAL change a careless pattern would eat: a different CALLEE method number
d = "__int64 f(void){ return this->method_172(); }"
e = "__int64 f(void){ return this->method_173(); }"
# and the comparison counter must count - TWO operators against NONE, so the delta is 2
f = "if (a > b) return 1; if (c <= d) return 2;"
g = "return 1;"

c1 = dh(a) == dh(b)
c2 = dh(a) != dh(c)
c3 = dh(d) != dh(e)
c4 = hashlib.sha256(a.encode()).hexdigest() != hashlib.sha256(b.encode()).hexdigest()
c5 = len(_CMP.findall(f)) - len(_CMP.findall(g)) == 2

print("C1 a label sequence number of collapses %s" % ("PASS" if c1 else "FAIL"))
print("C2 a real constant change does NOT collapse %s" % ("PASS" if c2 else "FAIL"))
print("C3 method_172 and method_173 stay DISTINCT %s" % ("PASS" if c3 else "FAIL"))
print(" - C3 is why the pattern demands an `a` then an upper case letter")
print("C4 NEGATIVE: the raw texts really ARE different %s" % ("PASS" if c4 else "FAIL"))
print(" - without C4, C1 could pass on two identical inputs and mean nothing")
print("C5 the comparison counter counts %s" % ("PASS" if c5 else "FAIL"))
ok = c1 and c2 and c3 and c4 and c5
print("\n%s" % ("ALL CONTROLS PASS" if ok else "***** A CONTROL FAILED"))
return 0 if ok else 1

def main(argv):
    if len(argv) == 2 and argv[1] == "--selftest":
        return selftest()
    if len(argv) != 4:
        print(__doc__.split("USAGE")[1].rstrip())
        return 2
    return build(argv[1], argv[2], argv[3])

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Option 02 # Symbol and Size diff:

This technique is attractive because it is fast, detects new code, reads two kernelcaches and respective symbols maps, and nothing else (no IDA database, no corpus or database). The walkthrough follows below:

```
$ python -m pip install capstone
$ ipsw kernel symbolicate C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin --json --signatures
.\symbolicator\kernel
$ ipsw kernel symbolicate C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --json --signatures
.\symbolicator\kernel
```

The syntax must be highlighted here, and the old kernelcache must come first and then the new one:

```
$ python diffkc.py $OLD $NEW <options>
```

The following scripts produce a long output with different between both kernelcaches, but I am limiting its output here:

```
$ python diffkc.py C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --limit 20 --reach
```

```
IMAGE kc_27.0ga.bin          197748 starts,  46460 symbols
IMAGE kc_27.2b1.bin          199251 starts,  46581 symbols
  ipsw kernel syscall      430 entry points
  ipsw kernel mach         63 entry points
```

```
ipsw kernel mig          359 entry points
building the call graph (direct `bl` edges only)...
call graph: 72319 functions with edges, 6698 reached
```

01. IMAGES

What this report was built from; the banner is also on stderr as progress

```
OLD kc_27.0ga.bin          197748 starts  46460 symbols
NEW kc_27.2b1.bin          199251 starts  46581 symbols
EP  entry point tables: SYS 430, TRAP 63, MIG 359
```

02. SUMMARY

kc_27.0ga.bin -> kc_27.2b1.bin

symbols matched by name	31452	
NEW names	83	the only signal that sees new code
names GONE	20	
matched, size MOVED	511	ranked by magnitude below
matched, same size DIFFERENT body	2752	of 30941 same size
of those, DIRECT entry points	3	syscall, mach trap or IOKit MIG
`_OUTLINED_FUNCTION_N`	1394	dropped as a POSITION, not an identity
names at MORE THAN ONE address	303	dropped - a name that repeats cannot be joined
of the changed, REACHED by a call	229	hops over DIRECT `bl` edges - a LOWER bound

03. NEW NAMES

Nothing to compare against, so read these first

```
# kind function
- ----
1      IOPolledFileFlush
2      IOPolledFilePollersClose
3      IOPolledFilePollersOpen
4      IOPolledFileSeek
5      IOPolledFileWrite
6      IOProvideCoreFileAccess
7      _AEAKernelEncryptInitializeState
8      IOPolledFileOpen(char const*, unsigned int, unsigned long long,
    unsigned long long, unsigned long long, void*, unsigned long,
    IOPolledFileIOVars**, OSData**, unsigned char*, unsigned long*)
9      IOPolledFilePollersSetup(IOPolledFileIOVars*, unsigned int)
10     IOAccessCoreFileData(void*, int, unsigned long long, int, void*)
11     IOOpenPolledCoreFile(void*, void*)
12     IOResolveCoreFilePath()
13     IOPolledFilePollersIODone(IOPolledFilePollers*, bool)
14     ANEHWDDevice::processTargetToHostIOCommand(unsigned long long, unsigned
    long long, unsigned long long) (.cold.1)
15     ANEClientDeviceCore::ANE_ProgramPrepareAndSubmitRequest(ANEProgramRequestArgs&,
    ANEProgramSendRequestAdditionalParams&, ANESharedEvents*)
16     _syncroot_unregister_path
17     aea_read_callback
18     aea_stage_outproc
19     aea_stage_reset
20     aea_write_callback
... 63 more of 83 (drop `--limit` to print them all)
```

04. NAMES GONE

```
# function
- ----
1  ApiodmaRegStream::flush()
2  IONVMeController::CheckSanitizeStatus() (.cold.2)
3  AppleProcessorTrace::hibernationWake()
4  PMAssertionsTracker::handleSetAssertionLevel(unsigned long long, unsigned
```

```

int)
5 AppleUSBDeviceNCMData::submitPacketGated(unsigned char*, unsigned int)
6 AppleUSBDeviceNCMData::isTRMUnrestrictedByEDT()
7 AppleM2ScalerCSCHalMSR23::activateMSRState_gatedContext(powerManagerState)
  (.cold.3)
8 AppleM2ScalerCSCHalMSR23::postInit() (.cold.1)
9 M2ScalerScalingASEControlMSR23::configureInterpolation_gatedContext(M2ScalerCSCRequest*,
  COLOR_FORMAT_CLASS, bool, pipeUnitExchange*) (.cold.1)
10 RTBuddyBinaryLogEntryHandlerMetric::processStringTableEntry(libkern::bounded_ptr<_log_entry
  const, os_detail::panic_trapping_policy>, logSourceInfo*)
11 invocation function for block in
  AppleProcessorTrace::callPlatformFunction(OSSymbol const*, bool, void*,
  void*, void*, void*)
12 ___syncroot_register_block_invoke
13 _lifs_apply_cache_action
14 _prepareTransformBuffersAndOptions.cold.14
15 console_ring_try_empty
16 igmp_input_v2_report
17 panic_init
18 sysctl_serialdebugmode
19 upl_commit_range
20 vm_compressor_init_size

```

05. SIZE MOVED

Largest first; the delta is in INSTRUCTIONS, not bytes

delta	old	new	kind	function
-1792	3915	2123		_tmpfs_dir_getdents.cold.1
-513	652	139		std::__1::__libcxx_verbose_abort(char const*, ...)
-455	482	27		AppleOLYHAL_isUartDebugEnabled()
+452	26	478		AppleOLYHAL_isDevFusedOrCSRInternal()
-325	332	7		
M2ScalerScalingASEControlMSR29::configureInterpolation_gatedContext(M2ScalerCSCRequest*,				
				COLOR_FORMAT_CLASS, bool, pipeUnitExchange*)
-295	344	49		IODTNVRAMFormatHandler::handleEphDM()
-282	2089	1807		vm_object_iopl_request
-237	2440	2203		vm_fault_page
+222	555	777		ANEProgramRTResource::programLoadFromMachoFile(ANEProgramCreateArgs*,
				ANEProgramCreateAdditionalParams*)
+198	2491	2689		
ANEProgramRTResource::buildFirmwareRTInferenceCallRequest(ANEInferenceRequest&				
				unsigned int)
+197	127	324		ANEHWDevice::ANE_AddExclaveClient_gated()
+193	803	996		ANEProgramRTResource::programLoadFromMachoFile(ANEProgramCreateArgs*,
				ANEProgramCreateAdditionalParams*):\$_0::operator()(unsigned
				int, unsigned long long, unsigned long long,
				unsigned long long&, ANEResourceUsageType, bool)
				const
+185	107	292		ANEHWDevice::ANE_RemoveExclaveClient_gated()
+179	402	581		AppleTypeCPhy::configureUSB2(AppleTypeCPhyPowerLevel,
				unsigned int)
-171	298	127		ANEScheduler::wire(ANERequest&)
+165	110	275		vm_upl_sk_withdraw_pages
-163	2486	2323		AppleT8130TypeCPhy::eusb2phy_init(bool, bool)
+162	1146	1308		
ANEProcess::patchClusterProcessInstanceWeightsBuffers(ANECClientProgramInstanceParams*,				
				task*)
+157	18	175		ApiodmaRegStream::flush() (.cold.1)
+155	719	874		ANEHWDevice::ANE_ExcLaveMemoryWiring(ANEExcLaveWirin
				gContext*)

... 491 more of 511 (drop '--limit' to print them all)

06. SAME SIZE, DIFFERENT BODY, AND DIRECTLY CALLABLE

The most actionable rows: a body that changed while its SIZE did not is a check SWAPPED, not added. 'instrs' is the function LENGTH, so a repeated value is a size class rather than a repeated row - MIG stubs built from one template share a length

```
instrs kind function
-----
155 SYS mincore
203 SYS __sigwait_nocancel
208 SYS sigprocmask
```

07. SAME SIZE, DIFFERENT BODY

ASCENDING, because the head of the descending list is always a huge display routine where register allocation alone moves something

```
instrs kind hops function
-----
155 SYS direct mincore
203 SYS direct __sigwait_nocancel
208 SYS direct sigprocmask
52 1 thread_terminate
83 1 stack_microstackshot
108 1 task_add_conclave
113 1 virtual_env_create
146 1 nxprov_close
203 1 vm_shared_region_update_task
277 1 vm_map_machine_attribute
338 1 nx_create
360 1 vm_map_copy_overwrite
425 1 ch_open
512 1 vm_map_region_recurse_64
526 1 fp_close_and_unlock_with_action
631 1 kern_stack_snapshot_internal
665 1 psignal_internal
1372 1 task_info
1692 1 necp_policy_apply_all
3709 1 necp_application_find_policy_match_internal
... 2732 more of 2752 (drop '--limit' to print them all)
```

The output brings a good indication about functions that can be interesting or not to perform deep research. As examples, it shows the number of names that have changed between kernelcache versions, changed in size, which names (functions) have been introduced, what functions had their body modified and the number of bytes that were added or subtracted. All functions are already demangled, but you have the option to list them mangled using the `--mangled` option. The `--reach` option is one of the most meaningful because it provides us with reachability information about the function. For example, a hop equal to 1 means exactly one call away from an entry point starting from user-space. This interpretation does not apply to the next case and hop equal to 1 means the distance between an externally visible surface of the driver, and not from user-space. If you see some entry with "blank" in hope, this only means that it is not reached by direct calls, but it wasn't determined the number of calls by the tool.

If you want you can restrict the output to one component, as shown below:

```
$ python diffkc.py C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --kext IOSurface --reach
```

```
IMAGE kc_27.0ga.bin          197748 starts,  46460 symbols
IMAGE kc_27.2b1.bin          199251 starts,  46581 symbols
ipsw kernel syscall         430 entry points
ipsw kernel mach            63 entry points
ipsw kernel mig             359 entry points
building the call graph (direct 'bl' edges only)...
call graph: 72319 functions with edges, 881 reached from 566 seeds
```

(restricted to kext matching 'IOSurface')

01. IMAGES

What this report was built from; the banner is also on stderr as progress

```
OLD kc_27.0ga.bin          197748 starts  46460 symbols
NEW kc_27.2b1.bin          199251 starts  46581 symbols
EP  entry point tables: SYS 430, TRAP 63, MIG 359
```

02. SUMMARY

kc_27.0ga.bin -> kc_27.2b1.bin

```
symbols matched by name          151
NEW names                        0    the only signal that sees new code
names GONE                       0
matched, size MOVED              1    ranked by magnitude below
matched, same size DIFFERENT body 7    of 150 same size
  of those, DIRECT entry points  0    syscall, mach trap or IOKit MIG
  '_OUTLINED_FUNCTION_N'        1394  dropped as a POSITION, not an identity
  names at MORE THAN ONE address 303   dropped - a name that repeats cannot be joined
  of the changed, REACHED by a call 6    hops over DIRECT 'bl' edges - a LOWER bound
  seeded from 566 address taken functions inside the kext - a driver's entry points are its
vtable and dispatch entries, not syscalls.
```

03. SIZE MOVED

Largest first; the delta is in INSTRUCTIONS, not bytes

delta	old	new	kind	function
+2	605	-> 607		IOSurfaceTransaction::serialize(IOSurfaceTransactionSerialized*, task*) (.cold.1)

04. SAME SIZE, DIFFERENT BODY

ASCENDING, because the head of the descending list is always a huge display routine where register allocation alone moves something

instrs	kind	hops	function
61	direct		IOSurfaceRootUserClient::stop(IOService*)
12	2		IOSurfaceRoot::client_stopped(IOSurfaceRootUserClient*, task*) (.cold.1)
12	2		IOSurfaceRoot::client_stopped(IOSurfaceRootUserClient*, task*) (.cold.2)
23	3		IOSurfaceRootUserClient::set_surface_handle(IOSurfaceClient*, unsigned int) (.cold.2)
12	4		IOSurfaceRootUserClient::getSurfaceClient(unsigned int, IOSurfaceClient**) const (.cold.2)
15	5		IOSurfaceRoot::alloc_surfaceid(unsigned int*) (.cold.3)
68			IOSurfaceRoot::removeEventNotifierClient(IOSurfaceEventNotifierClient*)

There are two another two options that are useful. The first is `--from`, which works as a filter and restricts report to functions reachable by a given function X, which the `--path` option works as a tracer and print the chain from a seed up to a given function Y.

One example using these options follows below and the interpretation of each output is direct and clear enough:

```
$ python diffkc.py C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --from _Xmach_vm_map_external
```



```

IMAGE kc_27.0ga.bin          197748 starts,  46460 symbols
IMAGE kc_27.2b1.bin          199251 starts,  46581 symbols
  ipsw kernel syscall      430 entry points
  ipsw kernel mach         63 entry points
  ipsw kernel mig          359 entry points
  building the call graph (direct `bl` edges only)...
  2014 functions reachable from '_Xmach_vm_map_external'

```

(restricted to reachable from '_Xmach_vm_map_external')

01. IMAGES

What this report was built from; the banner is also on stderr as progress

```

OLD kc_27.0ga.bin          197748 starts  46460 symbols
NEW kc_27.2b1.bin          199251 starts  46581 symbols
EP  entry point tables: SYS 430, TRAP 63, MIG 359

```

02. SUMMARY

kc_27.0ga.bin -> kc_27.2b1.bin

```

symbols matched by name          460
NEW names                        0   the only signal that sees new code
names GONE                       3
matched, size MOVED             12   ranked by magnitude below
matched, same size DIFFERENT body 86   of 448 same size
  of those, DIRECT entry points  0   syscall, mach trap or IOKit MIG
  `_OUTLINED_FUNCTION_N`       1394  dropped as a POSITION, not an identity
names at MORE THAN ONE address   303  dropped - a name that repeats cannot be joined
  of the changed, REACHED by a call 86   hops over DIRECT `bl` edges - a LOWER bound
    seeded from the single entry point '_Xmach_vm_map_external'

```

03. NAMES GONE

```

# function
- -----
1 PMAssertionsTracker::handleSetAssertionLevel(unsigned long long, unsigned
  int)
2 console_ring_try_empty
3 upl_commit_range

```

04. SIZE MOVED

Largest first; the delta is in INSTRUCTIONS, not bytes

delta	old	new	kind	function
-282	2089	-> 1807		vm_object_iopl_request
-237	2440	-> 2203		vm_fault_page
+165	110	-> 275		vm_upl_sk_withdraw_pages
+127	330	-> 457		handle_debugger_trap
+79	1029	-> 1108		vm_object_update
-37	3843	-> 3806		vm_fault_internal
+19	746	-> 765		vn_open_auth
-14	1366	-> 1352		vm_object_upl_request
+3	154	-> 157		ipc_right_lookup_read
+2	411	-> 413		task_deallocate_internal
-2	587	-> 585		upl_abort_range
-2	227	-> 225		vm_page_wire

05. SAME SIZE, DIFFERENT BODY

ASCENDING, because the head of the descending list is always a huge display routine where register allocation alone moves something

instrs	kind	hops	function
159	2		vm_map_copyout_kernel_buffer
324	2		vm_object_copy_strategically
376	2		vm_object_deallocate
888	2		pmap_enter_options_internal
35	3		vm_map_delete_for_overwrite
108	3		pmap_protect_options
151	3		pmap_construct_pte
179	3		pmap_nest_internal
568	3		vm_map_copyout_internal
587	3		vm_object_collapse
800	3		vm_object_copy_slowly
854	3		pmap_page_protect_options_internal
1625	3		pmap_batch_set_cache_attributes_internal

....
<truncated output>

```
$ python diffkc.py C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --path vm_map_copy_overwrite
```

```
IMAGE kc_27.0ga.bin 197748 starts, 46460 symbols
IMAGE kc_27.2b1.bin 199251 starts, 46581 symbols
  ipsw kernel syscall 430 entry points
  ipsw kernel mach 63 entry points
  ipsw kernel mig 359 entry points
  building the call graph (direct 'bl' edges only)...
```

CALL PATH vm_map_copy_overwrite

seeded from 865 syscall, mach trap and MIG entry points

```
SEED 0xffffffff00ac589a8 _Xmach_vm_write
hop 1 0xffffffff00ac9ce34 vm_map_copy_overwrite
```

```
seeds calling it DIRECTLY: 2
  0xffffffff00ac589a8 _Xmach_vm_write
  0xffffffff00ac68530 _Xwrite
```

```
$ python diffkc.py C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin --path __ZN13IOSurfaceRoot15alloc_surfaceidEPj
--kext IOSurface
```

```
IMAGE kc_27.0ga.bin 197748 starts, 46460 symbols
IMAGE kc_27.2b1.bin 199251 starts, 46581 symbols
  ipsw kernel syscall 430 entry points
  ipsw kernel mach 63 entry points
  ipsw kernel mig 359 entry points
  building the call graph (direct 'bl' edges only)...
```

CALL PATH __ZN13IOSurfaceRoot15alloc_surfaceidEPj

seeded from 566 address taken functions inside the kext

```
SEED 0xffffffff00a346dc8 sub_ffffffff00a346dc8
hop 1 0xffffffff00a346e1c sub_ffffffff00a346e1c
hop 2 0xffffffff00a346e94 IOSurfaceRoot::create_surface_internal(task*, OSDictionary*,
IOSurface*)
hop 3 0xffffffff00a32db74 IOSurface::init(IOSurfaceRoot*, task*, OSDictionary*, IOSurface*)
hop 4 0xffffffff00a348e20 IOSurfaceRoot::alloc_surfaceid(unsigned int*)
```

```
seeds calling it DIRECTLY: 0
```

I think readers have already understood the importance of using this kind of tool in daily work. The source of `diffkc.py` follows below:

```
$ cat diffkc.py
```

```
r"""diffkc.py - diff two iOS kernelcaches. ONE file, two paths on the command line.
```

```
"""
import argparse
import bisect
import io
import json
import os
import re
import struct
import subprocess
import sys

try:
    import capstone
except ImportError:
    sys.exit("diffkc: capstone is required - pip install capstone")

LC_SEGMENT_64 = 0x19
LC_FUNCTION_STARTS = 0x26
LC_FILESET_ENTRY = 0x80000035

def _uleb(b, i):
    r, sh = 0, 0
    while True:
        c = b[i]
        i += 1
        r |= (c & 0x7F) << sh
        if not (c & 0x80):
            return r, i
        sh += 7

class KC(object):

    def __init__(self, path, sympath=None, announce=True):
        self.path = path
        self.name = os.path.basename(path)
        self.data = io.open(path, "rb").read()
        self.md = capstone.Cs(capstone.CS_ARCH_ARM64, capstone.CS_MODE_LITTLE_ENDIAN)
        self.S = self._segments(0)
        if not self.S:
            sys.exit("diffkc: %s has no LC_SEGMENT_64 - is it a decompressed kernelcache?\n"
                    "      `ipsw kernel dec <kernelcache>` writes one." % path)
        self.STARTS, self._entries = self._function_starts()
        self._sorted = sorted(self.STARTS)
        self.SYM = self._syms(sympath or (path + ".symbols.json"))
        self._kexts = None
        if announce:
            sys.stderr.write("IMAGE %-28s %7d starts, %6d symbols\n"
                             % (self.name, len(self.STARTS), len(self.SYM)))

    def _cmds(self, base=0):
        d = self.data
        n = struct.unpack_from("<I", d, base + 16)[0]
        p = base + 32
        for _ in range(n):
            c, cs = struct.unpack_from("<2I", d, p)
            if cs == 0:
                break
            yield c, p, cs
            p += cs
```

```
def _segments(self, base=0):
    o = []
    for c, p, cs in self._cmds(base):
        if c == LC_SEGMENT_64:
            nm = self.data[p + 8:p + 24].split(b"\x00")[0].decode()
            vm, vs, fo, fs = struct.unpack_from("<4Q", self.data, p + 24)
            o.append((nm, vm, fo, fs))
    return o

def _function_starts(self):
    starts, entries = set(), []
    for c, p, cs in self._cmds():
        if c == LC_FILESET_ENTRY:
            vmaddr, fileoff = struct.unpack_from("<2Q", self.data, p + 8)
            noff = struct.unpack_from("<I", self.data, p + 24)[0]
            nm = self.data[p + noff:p + cs].split(b"\x00")[0].decode()
            entries.append((nm, fileoff, vmaddr))
    for base in [0] + [fo for _, fo, _ in entries]:
        try:
            segs = self._segments(base)
        except Exception:
            continue
        tb = None
        for c, p, _ in self._cmds(base):
            if c == LC_FUNCTION_STARTS:
                tb = struct.unpack_from("<2I", self.data, p + 8)
            if not tb or not tb[1]:
                continue
            text = next((s for s in segs if s[0] == "__TEXT"), None)
            if not text:
                continue
            va, i, end = text[1], tb[0], tb[0] + tb[1]
            while i < end:
                d, i = _uleb(self.data, i)
                if d == 0:
                    break
                va += d
            starts.add(va)
    return starts, entries

def _syms(self, sympath):
    o = {}
    if not os.path.exists(sympath):
        sys.stderr.write(" ! no symbol map at %s - run `ipsw kernel symbolicate %s`\n"
                        % (os.path.basename(sympath), self.name))
        return o
    try:
        raw = json.load(io.open(sympath, encoding="utf-8"))
    except Exception as e:
        sys.stderr.write(" ! symbol map unreadable (%s)\n" % e)
        return o
    for k, v in raw.items():
        try:
            o[int(k, 0) if isinstance(k, str) else int(k)] = v
        except Exception:
            pass
    return o

def kexts(self):
    if self._kexts is not None:
        return self._kexts
    out = []
    for nm, fo, vmaddr in self._entries:
        try:
            segs = self._segments(fo)
        except Exception:
            continue
```

```
        ex = [(vm, vm + fs) for sn, vm, sfo, fs in segs if "TEXT_EXEC" in sn and fs]
        if not ex:
            ex = [(vm, vm + fs) for sn, vm, sfo, fs in segs if sn == "__TEXT" and fs]
        if ex:
            out.append((nm, min(a for a, _ in ex), max(b for _, b in ex)))
        self._kexts = sorted(out, key=lambda r: r[1])
        return self._kexts

def kext_of(self, va):
    for nm, lo, hi in self.kexts():
        if lo <= va < hi:
            return nm
    return None

def f_off(self, va):
    for _, vm, fo, fs in self.S:
        if fs and vm <= va < vm + fs:
            return fo + (va - vm)
    return None

def func_at(self, va):
    i = bisect.bisect_right(self._sorted, va) - 1
    return self._sorted[i] if i >= 0 else None

def next_start(self, va):
    i = bisect.bisect_right(self._sorted, va)
    return self._sorted[i] if i < len(self._sorted) else None

def size(self, va):
    nxt = self.next_start(va)
    if not nxt or nxt <= va:
        return 0
    return (nxt - va) // 4

def body(self, va, cap=0x8000):
    nxt = self.next_start(va)
    end = min(nxt if nxt else va + cap, va + cap)
    o = self.f_off(va)
    if o is None:
        return []
    return list(self.md.disasm(self.data[o:o + (end - va)], va))

_CXXFILT = []

def _cxxfilt():
    if not _CXXFILT:
        for cand in ("llvm-cxxfilt", "c++filt"):
            try:
                subprocess.run([cand, "--version"], capture_output=True, timeout=10)
                _CXXFILT.append(cand)
                break
            except Exception:
                continue
    else:
        _CXXFILT.append(None)
    return _CXXFILT[0]

def demangle_many(names):
    out = {n: n for n in names}
    tool = _cxxfilt()
    todo = [n for n in names if n and re.match(r"^\_+Z", n)]
    if not tool or not todo:
        return out
    cand = []
    for n in todo:
        cand.append((n, n))
```

```
cand.append((n, n[1:]))
try:
    r = subprocess.run([tool], input="\n".join(c for _n, c in cand),
                        capture_output=True, text=True, timeout=180)
    got = (r.stdout or "").split("\n")
    if len(got) >= len(cand):
        for (n, c), t in zip(cand, got):
            t = t.strip()
            if t and t != c and out[n] == n:
                out[n] = t
except Exception:
    pass
return out

POSITIONAL = re.compile(r"_OUTLINED_FUNCTION_\d+")

def named(kc, drop_positional=True):
    seen, dropped = {}, 0
    for va in kc.STARTS:
        nm = kc.SYM.get(va)
        if not nm:
            continue
        if drop_positional and POSITIONAL.search(nm):
            dropped += 1
            continue
        seen.setdefault(nm, []).append(va)
    out = dict((n, v[0]) for n, v in seen.items() if len(v) == 1)
    ambiguous = sorted(n for n, v in seen.items() if len(v) > 1)
    return out, dropped, ambiguous

REG = re.compile(r"\b([wx](?:\d{1,2}|zr)|[bhsdq]\d{1,2}|v\d{1,2}|sp)\b")
IMM = re.compile(r"#-?0x[0-9a-fA-F]+|#-?\d+")
MASK_PAGE = re.compile(r"(adrp\s+[wx]\d{1,2},\s*)(?:#-?0x[0-9a-fA-F]+|#-?\d+)")
MASK_ADDR = re.compile(r"\b0xffffffff[0-9a-fA-F]{9}\b")

CALLER_SAVED = set("x%d" % i for i in range(0, 19)) | set(["x30"])
BRANCH = ("b", "bl", "br", "blr", "ret", "retab", "retaa", "cbz", "cbnz", "tbz", "tbnz")

def _is_cond(m):
    return m.startswith("b.") or m in ("cbz", "cbnz", "tbz", "tbnz")

def _target(ins):
    if "#" not in ins.op_str:
        return None
    tail = ins.op_str.rsplit("#", 1)[1].strip()
    try:
        return int(tail, 16) if tail.startswith("0x") else int(tail)
    except ValueError:
        return None

def _blocks(ins):
    if not ins:
        return [], {}
    addr = dict((x.address, i) for i, x in enumerate(ins))
    lo, hi = ins[0].address, ins[-1].address
    leaders = set([0])
    for i, x in enumerate(ins):
        m = x.mnemonic.lower()
        if m.startswith("b.") or m in BRANCH:
            if i + 1 < len(ins):
                leaders.add(i + 1)
            t = _target(x)
            if t is not None and lo <= t <= hi and t in addr and m not in ("bl", "blr"):
                leaders.add(addr[t])
    starts = sorted(leaders)
    blocks = []
```

```
for k, s in enumerate(starts):
    e = starts[k + 1] if k + 1 < len(starts) else len(ins)
    blocks.append((s, e))
bidx = {}
for b, (s, _e) in enumerate(blocks):
    bidx[s] = b
succ = dict((b, []) for b in range(len(blocks)))
for b, (s, e) in enumerate(blocks):
    last = ins[e - 1]
    m = last.mnemonic.lower()
    if m in ("ret", "retab", "retaa", "br"):
        continue
    t = _target(last)
    if m == "b" or m.startswith("b.") or m in ("cbz", "cbnz", "tbz", "tbnz"):
        if t is not None and t in addr and addr[t] in bidx:
            succ[b].append(bidx[addr[t]])
        if m != "b" and e < len(ins) and e in bidx:
            succ[b].append(bidx[e])
    elif e < len(ins) and e in bidx:
        succ[b].append(bidx[e])
return blocks, succ

def _transfer(kc, ins, s, e, taint):
    t = set(taint)
    for x in ins[s:e]:
        m = x.mnemonic.lower()
        regs = REG.findall(x.op_str)
        dst = regs[0].replace("w", "x") if regs else None
        base = regs[1].replace("w", "x") if len(regs) > 1 else None
        if m == "adrp":
            if dst:
                t.add(dst)
        elif m == "add" and base and base in t:
            if dst:
                t.add(dst)
        elif m.startswith(("ldr", "str", "ldp", "stp", "ldur", "stur")) and base and base in t:
            if dst and not m.startswith(("str", "stp", "stur")):
                t.discard(dst)
        elif m in ("bl", "blr"):
            t -= CALLER_SAVED
        elif dst and m not in NO_WRITE:
            t.discard(dst)
    return t

NO_WRITE = set(["cmp", "cmn", "tst", "teq", "str", "stp", "stur", "strb", "strh", "stlr",
                "stlrb", "stlrb", "b", "bl", "blr", "br", "ret", "retab", "retaa",
                "cbz", "cbnz", "tbz", "tbnz"])

def body_text(kc, va, normalise=True):
    ins = kc.body(va)
    if not normalise:
        return "\n".join("%s %s" % (x.mnemonic, x.op_str) for x in ins)
    if not ins:
        return ""

    blocks, succ = _blocks(ins)
    pred = dict((b, []) for b in range(len(blocks)))
    for b, ss in succ.items():
        for s in ss:
            pred[s].append(b)

    ALL = None
    tin = dict((b, set()) for b in range(len(blocks)))
    tout = dict((b, set()) for b in range(len(blocks)))
    for _round in range(len(blocks) + 2):
        changed = False
        for b, (s, e) in enumerate(blocks):
```



```
    if b == 0 or not pred[b]:
        new_in = set()
    else:
        new_in = set()
        for p in pred[b]:
            new_in |= tout[p]
    new_out = _transfer(kc, ins, s, e, new_in)
    if new_in != tin[b] or new_out != tout[b]:
        tin[b], tout[b] = new_in, new_out
        changed = True
    if not changed:
        break

parts = []
for b, (s, e) in enumerate(blocks):
    t = set(tin[b])
    for x in ins[s:e]:
        out = "%s %s" % (x.mnemonic, x.op_str)
        m = x.mnemonic.lower()
        regs = REG.findall(x.op_str)
        dst = regs[0].replace("w", "x") if regs else None
        base = regs[1].replace("w", "x") if len(regs) > 1 else None
        if m == "adrp":
            out = MASK_PAGE.sub(r"\1@", out)
            if dst:
                t.add(dst)
        elif m == "add" and base and base in t:
            out = IMM.sub("@", out)
            if dst:
                t.add(dst)
        elif m.startswith(("ldr", "str", "ldp", "stp", "ldur", "stur")) and base and base in t:
            out = IMM.sub("@", out)
            if dst and not m.startswith(("str", "stp", "stur")):
                t.discard(dst)
        elif m in ("bl", "blr"):
            t -= CALLER_SAVED
        elif dst and m not in NO_WRITE:
            t.discard(dst)
        out = MASK_ADDR.sub("@", out)
        parts.append(out)
    return "\n".join(parts)
```

```
_ROW = re.compile(r"^[s*(?:\d+\s+)?(0x[0-9a-fA-F]+):\s+(\S+)"
```

```
def entry_points(kc):
    out = {}
    for sub, kind in (("syscall", "SYS"), ("mach", "TRAP"), ("mig", "MIG")):
        try:
            r = subprocess.run(["ipsw", "kernel", sub, kc.path],
                               capture_output=True, text=True, timeout=900)
        except FileNotFoundError:
            sys.stderr.write(" ! ipsw is not on PATH - the entry point column will be empty\n")
            return out
        except Exception as e:
            sys.stderr.write(" ! `ipsw kernel %s` did not run (%s)\n" % (sub, e))
            continue
    n = 0
    for line in r.stdout.splitlines():
        m = _ROW.match(line)
        if not m:
            continue
        va = int(m.group(1), 16)
        nm = kc.SYM.get(va) or m.group(2).rstrip(":")
        if nm and nm not in out:
            out[nm] = kind
            n += 1
    sys.stderr.write(" ipsw kernel %-8s %5d entry points\n" % (sub, n))
```

```
    return out

def diff(old, new, kext=None, bodies=True, keep=None):
    a, dropped_a, amb_a = named(old)
    b, dropped_b, amb_b = named(new)
    ka, kb = set(a), set(b)
    if kext:
        ka = set(n for n in ka
                  if (old.kext_of(a[n]) or "") and kext.lower() in (old.kext_of(a[n]) or
"").lower())
        kb = set(n for n in kb
                  if (new.kext_of(b[n]) or "") and kext.lower() in (new.kext_of(b[n]) or
"").lower())
    if keep is not None:
        ka &= keep
        kb &= keep

    gone, fresh, both = sorted(ka - kb), sorted(kb - ka), ka & kb
    resized, same_size, raw_differs = [], [], 0
    for n in both:
        sa, sb = old.size(a[n]), new.size(b[n])
        if sa != sb:
            resized.append((sb - sa, sa, sb, n))
        elif bodies and sa:
            ta, tb = body_text(old, a[n]), body_text(new, b[n])
            if ta != tb:
                same_size.append((sa, n))
            if body_text(old, a[n], False) != body_text(new, b[n], False):
                raw_differs += 1
    return {"gone": gone, "new": fresh, "matched": len(both),
            "resized": sorted(resized, key=lambda r: (-abs(r[0]), r[3])),
            "same_size": sorted(same_size),
            "same_size_total": len(both) - len(resized),
            "raw_differs": raw_differs, "dropped": dropped_a + dropped_b,
            "ambiguous": sorted(set(amb_a) | set(amb_b)), "amap": a}

_C = {}

def init_colour(enabled):
    names = ("head", "rule", "key", "num", "warn", "good", "bad", "dim", "kind", "off")
    if not enabled:
        _C.update((n, "") for n in names)
    return
    if sys.platform == "win32":
        try:
            import ctypes
            k = ctypes.windll.kernel32
            k.SetConsoleMode(k.GetStdHandle(-11), 7)
        except Exception:
            pass
    _C.update({
        "head": "\033[1;36m",
        "rule": "\033[2;36m",
        "key": "\033[1m",
        "num": "\033[1;37m",
        "warn": "\033[1;33m",
        "good": "\033[32m",
        "bad": "\033[31m",
        "dim": "\033[2m",
        "kind": "\033[35m",
        "off": "\033[0m",
    })

_N = [0]

def section(title, note=None):
    _N[0] += 1
```

```
print("")
print("%s## %02d. %s" % (_C["head"], _N[0], title, _C["off"]))
print("%s%s" % (_C["rule"], "-" * (len(title) + 8), _C["off"]))
if note:
    import textwrap
    if note and not re.match(r"^\S+\.\S", note):
        note = note[0].upper() + note[1:]
    for ln in textwrap.wrap(note, 96):
        print("%s%s" % (_C["dim"], ln, _C["off"]))
print("")

def columns(header):
    rule = "".join(" " if c == " " else "-" for c in header)
    print("%s%s" % (_C["key"], header.rstrip(), _C["off"]))
    print("%s%s" % (_C["rule"], rule.rstrip(), _C["off"]))

def emit(plain, coloured, name):
    import textwrap
    indent = len(plain)
    avail = max(_W[0] - indent, 24)
    single = " " not in name.strip()
    parts = textwrap.wrap(name, avail, break_long_words=single,
                          break_on_hyphens=False) or [""]
    print("%s" % (coloured, parts[0]))
    for p in parts[1:]:
        print("%s" % (" " * indent, p))

def _more(total, limit):
    if limit is not None and total > limit:
        return "%s... %d more of %d (drop '--limit' to print them all)%s" % (
            _C["warn"], total - limit, total, _C["off"])
    return None

def _row(label, count, note="", warn=False):
    c = _C["warn"] if warn else _C["num"]
    print("%s%-36s %s8d %s %s" %
          (_C["key"], label, _C["off"], c, count, _C["off"], _C["dim"], note, _C["off"]))

_RAW = [False]
_W = [80]
_HOPS = {}
_SEEDBASIS = [""]

def nm(n, dm):
    return n if _RAW[0] else dm.get(n, n)

def hop(n, amap):
    va = amap.get(n)
    if va is None or va not in _HOPS:
        return ""
    h = _HOPS[va]
    return "direct" if h == 0 else str(h)

def hopkey(n, amap):
    va = amap.get(n)
    if va is None or va not in _HOPS:
        return 9999
    return _HOPS[va]

def addr_taken(kc):
    base = min(vm for _n, vm, _fo, fs in kc.S if fs)
    top = max(vm + fs for _n, vm, _fo, fs in kc.S if fs)
    starts = kc.STARTS
    out = set()
    for nm, vm, fo, fs in kc.S:
        if not fs or "DATA" not in nm:
            continue
```

```
blob = kc.data[fo:fo + fs]
for i in range(0, len(blob) - 8, 8):
    v = struct.unpack_from("<Q", blob, i)[0]
    if not v:
        continue
    if not (base <= v < top):
        off = v & 0xFFFFFFFF
        if off < 0x1000:
            continue
        v = base + off
        if not (base <= v < top):
            continue
    if v in starts:
        out.add(v)
return out

SRCNAME = re.compile(rb"(?![-~])([A-Za-z0-9_+]{2,60}\\.?(?:c|cpp|cc|s))\\x00")

def _sext(v, b):
    m = 1 << (b - 1)
    return (v ^ m) - m

def source_files(kc):
    names = {}
    for nm, vm, fo, fs in kc.S:
        if not fs or "TEXT_EXEC" in nm:
            continue
        blk = kc.data[fo:fo + fs]
        for m in SRCNAME.finditer(blk):
            names[vm + m.start(1)] = m.group(1).decode()
    if not names:
        return {}
    targets = set(names)
    per_func = {}
    for nm, vm, fo, fs in kc.S:
        if nm != "__TEXT_EXEC" or not fs:
            continue
        cnt = fs // 4
        ws = struct.unpack_from("<%dI" % cnt, kc.data, fo)
        pend = {}
        for i in range(cnt):
            w = ws[i]
            if (w & 0x9F000000) == 0x90000000:
                rd = w & 0x1F
                imm = _sext(((w >> 5) & 0x7FFF) << 2) | ((w >> 29) & 3), 21) << 12
                pend[rd] = ((vm + i * 4) & ~0xFFF) + imm, i)
                continue
            rn = (w >> 5) & 0x1F
            if rn not in pend:
                continue
            page, at = pend[rn]
            if i - at > 10:
                del pend[rn]
                continue
            t = None
            if (w & 0xFF800000) == 0x91000000:
                imm12 = (w >> 10) & 0xFFF
                if (w >> 22) & 1:
                    imm12 <= 12
                t = page + imm12
                del pend[rn]
            elif (w & 0xFFC00000) == 0xF9400000:
                t = page + (((w >> 10) & 0xFFF) << 3)
                del pend[rn]
            if t is not None and t in targets:
                site = vm + i * 4
                f = kc.func_at(site)
```

```
        if f is not None:
            per_func.setdefault(f, set()).add(names[t])
    return per_func

def callgraph(kc):
    import capstone as _cs
    md = _cs.Cs(_cs.CS_ARCH_ARM64, _cs.CS_MODE_LITTLE_ENDIAN)
    starts = kc.STARTS
    edges = {}
    for va in kc._sorted:
        nxt = kc.next_start(va)
        end = min(nxt if nxt else va + 0x8000, va + 0x8000)
        off = kc.f_off(va)
        if off is None or end <= va:
            continue
        out = None
        for ins in md.disasm(kc.data[off:off + (end - va)], va):
            if ins.mnemonic != "bl":
                continue
            s = ins.op_str
            if "#" not in s:
                continue
            try:
                t = int(s.rsplit("#", 1)[1].strip(), 16)
            except ValueError:
                continue
            if t in starts:
                if out is None:
                    out = set()
                out.add(t)
        if out:
            edges[va] = out
    return edges

def hops_from(edges, seeds):
    depth = {}
    frontier = [s for s in seeds]
    for s in frontier:
        depth[s] = 0
    d = 0
    while frontier:
        d += 1
        nxt = []
        for v in frontier:
            for w in edges.get(v, ()):
                if w not in depth:
                    depth[w] = d
                    nxt.append(w)
        frontier = nxt
    return depth

def show_path(old, want, seeds, edges, basis):
    prev, depth = {}, dict((s, 0) for s in seeds)
    frontier = list(seeds)
    d = 0
    while frontier:
        d += 1
        nxt = []
        for v in frontier:
            for w in edges.get(v, ()):
                if w not in depth:
                    depth[w] = d
                    prev[w] = v
                    nxt.append(w)
        frontier = nxt

    target = None
```

```
for va, n in old.SYM.items():
    if n == want:
        target = va
        if va in depth:
            break
if target is None:
    print(" %s: no such symbol" % want)
    return
print("")
print("%sCALL PATH %s%s" % (_C["head"], want, _C["off"]))
print("%s%s%s" % (_C["rule"], "-" * (11 + len(want)), _C["off"]))
print("%sseeded from %s%s" % (_C["dim"], basis, _C["off"]))
print("")
if target not in depth:
    print(" %sNOT REACHED by any direct `bl` from a seed%s" % (_C["warn"], _C["off"]))
    print(" %sthat means `no direct call path`, NEVER `unreachable` - a `blr` through a"
          % _C["dim"])
    print(" %svtable or a dispatch table is invisible to this graph.%s" % _C["off"])
    return

chain = [target]
while chain[-1] in prev:
    chain.append(prev[chain[-1]])
chain.reverse()
dm = demangle_many([old.SYM.get(v, "") for v in chain])
for i, va in enumerate(chain):
    raw = old.SYM.get(va, "sub_%x" % va)
    tag = "SEED" if i == 0 else "hop %d" % i
    print(" %s%-7s%s 0x%x %s"
          % (_C["key"], tag, _C["off"], va, dm.get(raw, raw)))

direct_seeds = sorted(s for s in seeds if target in edges.get(s, {}))
print("")
print(" %sseeds calling it DIRECTLY: %d%s" % (_C["dim"], len(direct_seeds), _C["off"]))
dm2 = demangle_many([old.SYM.get(s, "") for s in direct_seeds[:12]])
for s in direct_seeds[:12]:
    raw = old.SYM.get(s, "sub_%x" % s)
    print(" %s0x%x %s" % (s, dm2.get(raw, raw)))
if len(direct_seeds) > 12:
    print(" %s... %d more" % (len(direct_seeds) - 12))

def report(old, new, d, ep, limit):
    names = set(d["new"]) | set(d["gone"]) | set(n for _, _, n in d["resized"]) \
        | set(n for _, n in d["same_size"])
    dm = demangle_many(names)
    amap = d.get("amap", {})
    direct = [(s, n) for s, n in d["same_size"] if n in ep]
    if _HOPS:
        d["same_size"] = sorted(d["same_size"], key=lambda r: (hopkey(r[1], amap), r[0]))
    _N[0] = 0

    kinds = {}
    for v in ep.values():
        kinds[v] = kinds.get(v, 0) + 1

    section("IMAGES", "what this report was built from; the banner is also on stderr as progress")
    for tag, im in (("OLD", old), ("NEW", new)):
        print("%s%-4s%s %s-30s %s%7d%s starts %s%6d%s symbols"
              % (_C["key"], tag, _C["off"], im.name,
                 _C["num"], len(im.STARTS), _C["off"], _C["num"], len(im.SYM), _C["off"]))
    if kinds:
        print("%s%-4s%s entry point tables: SYS %d, TRAP %d, MIG %d"
              % (_C["key"], "EP", _C["off"],
                 kinds.get("SYS", 0), kinds.get("TRAP", 0), kinds.get("MIG", 0)))
    else:
        print("%s%-4s%s entry point join EMPTY - is `ipsw` on PATH?"
              % (_C["warn"], "EP", _C["off"]))
```

```
section("SUMMARY", "%s -> %s" % (old.name, new.name))
_row("symbols matched by name", d["matched"])
_row("NEW names", len(d["new"]), "the only signal that sees new code", warn=True)
_row("names GONE", len(d["gone"]))
_row("matched, size MOVED", len(d["resized"]), "ranked by magnitude below")
_row("matched, same size DIFFERENT body", len(d["same_size"]),
      "of %d same size" % d["same_size_total"])
_row(" of those, DIRECT entry points", len(direct), "syscall, mach trap or IOKit MIG",
      warn=True)
_row(" `OUTLINED_FUNCTION_N`", d["dropped"], "dropped as a POSITION, not an identity")
_row(" names at MORE THAN ONE address", len(d["ambiguous"]),
      "dropped - a name that repeats cannot be joined", warn=bool(d["ambiguous"]))
if _HOPS:
    reached = sum(1 for _s, n in d["same_size"] if hopkey(n, amap) < 9999)
    _row(" of the changed, REACHED by a call", reached,
          "hops over DIRECT `bl` edges - a LOWER bound", warn=True)
    if _SEEDBASIS[0]:
        print("%s seeded from %s%s" % (_C["dim"], _SEEDBASIS[0], _C["off"]))

if d["new"]:
    section("NEW NAMES", "nothing to compare against, so read these first")
    columns("%4s %-4s %s" % ("#", "kind", "function"))
    for i, n in enumerate(d["new"][:limit], 1):
        emit("%4d %-4s " % (i, ep.get(n, "")),
              "%s%4d%s %s%-4s%s " % (_C["dim"], i, _C["off"],
                                      _C["kind"], ep.get(n, ""), _C["off"]), nm(n, dm))
    m = _more(len(d["new"]), limit)
    if m:
        print(m)

if d["gone"]:
    section("NAMES GONE")
    columns("%4s %s" % ("#", "function"))
    for i, n in enumerate(d["gone"][:limit], 1):
        emit("%4d " % i, "%s%4d%s " % (_C["dim"], i, _C["off"]), nm(n, dm))
    m = _more(len(d["gone"]), limit)
    if m:
        print(m)

if d["resized"]:
    section("SIZE MOVED", "largest first; the delta is in INSTRUCTIONS, not bytes")
    columns("%6s %6s %5s %-4s %s"
            % ("delta", "old", "new", "kind", "function"))
    for delta, sa, sb, n in d["resized"][:limit]:
        c = _C["good"] if delta > 0 else _C["bad"]
        emit("%+6d %6d -> %5d %-4s " % (delta, sa, sb, ep.get(n, "")),
              "%s%+6d%s %6d -> %5d %s%-4s%s "
              % (c, delta, _C["off"], sa, sb, _C["kind"], ep.get(n, ""), _C["off"]),
              nm(n, dm))
    m = _more(len(d["resized"]), limit)
    if m:
        print(m)

if direct:
    section("SAME SIZE, DIFFERENT BODY, AND DIRECTLY CALLABLE",
            "the most actionable rows: a body that changed while its SIZE did not is a "
            "check SWAPPED, not added. `instrs` is the function LENGTH, so a repeated "
            "value is a size class rather than a repeated row - MIG stubs built from one "
            "template share a length")
    columns("%6s %-4s %s" % ("instrs", "kind", "function"))
    for s, n in sorted(direct)[:limit]:
        emit("%6d %-4s " % (s, ep[n]),
              "%6d %s%-4s%s " % (s, _C["kind"], ep[n], _C["off"]), nm(n, dm))
    m = _more(len(direct), limit)
    if m:
        print(m)
```



```

if d["same_size"]:
    section("SAME SIZE, DIFFERENT BODY",
           "ASCENDING, because the head of the descending list is always a huge display "
           "routine where register allocation alone moves something")
    columns("%6s %4s %6s %s" % ("instrs", "kind", "hops", "function"))
    for s, n in d["same_size"][:limit]:
        h = hop(n, amap)
        emit("%6d %4s %6s " % (s, ep.get(n, ""), h),
             "%6d %s%4s%s %s%6s%s " % (s, _C["kind"], ep.get(n, ""), _C["off"],
                                         _C["good"] if h else _C["dim"], h, _C["off"]),
             nm(n, dm))
    m = _more(len(d["same_size"]), limit)
    if m:
        print(m)

def selftest(old, new):
    ok = True
    print("\nCONTROLS")

    d = diff(old, old)
    p = (not d["new"] and not d["gone"] and not d["resized"] and not d["same_size"])
    ok &= p
    print(" C1 POSITIVE an image against ITSELF is quiet : %d differences %s"
          % (len(d["resized"]) + len(d["same_size"]), "PASS" if p else "** FAIL **"))

    d2 = diff(old, new)
    loud = len(d2["resized"]) + len(d2["same_size"]) + len(d2["new"]) + len(d2["gone"])
    p = loud > 0
    ok &= p
    print(" C2 NEGATIVE the two images are loud : %d differences %s"
          % (loud, "PASS" if p else "** FAIL ** (are these the same build?)"))

    p = d2["raw_differs"] > len(d2["same_size"])
    ok &= p
    print(" C3 POSITIVE normalisation removes relocation : %d raw -> %d normalised %s"
          % (d2["raw_differs"], len(d2["same_size"]), "PASS" if p else "** FAIL **"))

    p = d2["dropped"] > 0
    ok &= p
    print(" C4 the positional exclusion removes something: %d `_OUTLINED_FUNCTION_N` dropped %s"
          % (d2["dropped"], "PASS" if p else "** FAIL **"))

    ep = entry_points(old)
    kinds = {}
    for v in ep.values():
        kinds[v] = kinds.get(v, 0) + 1
    p = len(kinds) == 3
    ok &= p
    print(" C5 all three entry point tables contribute : SYS %d, TRAP %d, MIG %d %s"
          % (kinds.get("SYS", 0), kinds.get("TRAP", 0), kinds.get("MIG", 0),
            "PASS" if p else "** FAIL ** (is ipsw on PATH?)"))

    print("\n%s" % ("ALL CONTROLS PASS" if ok else "***** A CONTROL FAILED - DO NOT TRUST OUTPUT"))
    return 0 if ok else 1

def main():
    ap = argparse.ArgumentParser(
        description="Diff two iOS kernelcaches by symbol set, instruction count and body.",
        epilog="A symbol map is read from `<kernelcache>.symbols.json` beside each image; "
        "write one with `ipsw kernel symbolicate <kernelcache>`."
    )
    ap.add_argument("old", help="the OLDER decompressed kernelcache")
    ap.add_argument("new", help="the NEWER decompressed kernelcache")
    ap.add_argument("--old-syms", help="symbol map for OLD, if not beside it")
    ap.add_argument("--new-syms", help="symbol map for NEW, if not beside it")
    ap.add_argument("--kext", help="restrict to one kext, matched as a substring")
    ap.add_argument("--limit", type=int, default=0,

```

```
        help="rows per section. THE DEFAULT IS EVERY ROW; pass a number to truncate, "
        "e.g. --limit 20 for the head of each list")
ap.add_argument("--no-bodies", action="store_true",
        help="skip the body compare - much faster, loses the residue signal")
ap.add_argument("--no-entry-points", action="store_true", help="skip the ipsw entry point
join")
ap.add_argument("--width", type=int, default=0,
        help="total line width for wrapping long names (default 80); a FIXED "
        "width keeps the report reproducible where the terminal width would not")
ap.add_argument("--file", help="restrict to functions whose ASSERTS name this source file, "
        "recovered from the binary rather than a symbol map")
ap.add_argument("--from", dest="getfrom",
        help="restrict to functions reachable from this entry point over `bl` edges")
ap.add_argument("--path", help="expand one function's hop count into the chain behind it, "
        "and name every seed that calls it directly, then exit")
ap.add_argument("--reach", action="store_true",
        help="rank changed functions by hops from an app reachable entry "
        "point, over DIRECT `bl` edges only - a lower bound")
ap.add_argument("--mangled", action="store_true",
        help="print RAW linker names instead of demangled ones - the form grep, a "
        "symbol map lookup and `ipsw macho disass --symbol` take")
ap.add_argument("--no-color", action="store_true",
        help="plain text; colour is off automatically when stdout is "
        "not a terminal")
ap.add_argument("--selftest", action="store_true", help="run the five controls and exit")
a = ap.parse_args()

init_colour(not a.no_color and sys.stdout.isatty())
_RAW[0] = a.mangled
if a.width > 0:
    _W[0] = a.width

for p in (a.old, a.new):
    if not os.path.exists(p):
        sys.exit("diffkc: no such file: %s" % p)

old = KC(a.old, a.old_syms)
new = KC(a.new, a.new_syms)

if a.selftest:
    return selftest(old, new)

ep = {} if a.no_entry_points else entry_points(old)

keep = None
if a.file:
    sys.stderr.write(" mapping source files from assert literals...\n")
    pf = source_files(old)
    want = a.file.lower()
    addrs = set(va for va, fs in pf.items() if any(want in f.lower() for f in fs))
    keep = set(n for va, n in old.SYM.items() if va in addrs)
    files = sorted(set(f for fs in pf.values() for f in fs))
    sys.stderr.write(" %d source files seen, %d functions match %r\n"
        % (len(files), len(addrs), a.file))
    if not addrs:
        near = [f for f in files if want[:6] in f.lower()][:8]
        sys.stderr.write(" ! nothing matched. nearest names: %s\n" % (near or "none"))

if a.reach or a.path or a.getfrom:
    sys.stderr.write(" building the call graph (direct `bl` edges only)...\n")
    edges = callgraph(old)
    if a.getfrom or a.path:
        want = a.getfrom or a.path
        roots = [va for va, n in old.SYM.items() if n == want]
        if not roots:
            near = sorted(set(n for _v, n in old.SYM.items() if want.lower() in n.lower()))
            sys.stderr.write("diffkc: no symbol named %r\n" % want)
```

```
        if near:
            sys.stderr.write("          did you mean: %s\n" % ", ".join(near[:8]))
        else:
            sys.stderr.write("          and nothing contains that substring either\n")
            return 2
    if a.getfrom:
        seeds = roots
        basis = "the single entry point %r" % a.getfrom
    else:
        seeds = [va for va, n in old.SYM.items() if n in ep]
        basis = "%d syscall, mach trap and MIG entry points" % len(seeds)
        if a.kext:
            k = a.kext.lower()
            seeds = [va for va in addr_taken(old)
                     if k in (old.kext_of(va) or "").lower()]
            basis = ("%d address taken functions inside the kext" % len(seeds))
        else:
            seeds = None
    if a.path:
        init_colour(not a.no_color and sys.stdout.isatty())
        show_path(old, a.path, seeds, edges, basis)
        return 0
    if a.getfrom:
        h = hops_from(edges, seeds)
        if len(h) <= len(seeds):
            sys.stderr.write("    ! %r reaches nothing by a direct `bl` - a dispatch through a "
                             "vtable is invisible here\n" % a.getfrom)
            keep2 = set(n for va, n in old.SYM.items() if va in h)
            keep = keep2 if keep is None else (keep & keep2)
            _HOPS.update(h)
            _SEEDBASIS[0] = basis
            sys.stderr.write("    %d functions reachable from %r\n" % (len(h), a.getfrom))
        if a.reach and (ep or a.kext):
            sys.stderr.write("    building the call graph (direct `bl` edges only)...\n")
            edges = callgraph(old)
            if a.kext:
                k = a.kext.lower()
                at = addr_taken(old)
                seeds = [va for va in at if k in (old.kext_of(va) or "").lower()]
                _SEEDBASIS[0] = ("%d address taken functions inside the kext - a driver's entry "
                                "points are its vtable and dispatch entries, not syscalls"
                                % len(seeds))
            else:
                seeds = [va for va, n in old.SYM.items() if n in ep]
                _SEEDBASIS[0] = ("%d syscall, mach trap and MIG entry points - this reaches XNU core "
                                "and NOT a kext, because IOKit dispatch is an indirect call"
                                % len(seeds))
            _HOPS.update(hops_from(edges, seeds))
            sys.stderr.write("    call graph: %d functions with edges, %d reached from %d seeds\n"
                             % (len(edges), len(_HOPS), len(seeds)))
    d = diff(old, new, kext=a.kext, bodies=not a.no_bodies, keep=keep)
    scopes = []
    if a.kext:
        scopes.append("kext matching %r" % a.kext)
    if a.file:
        scopes.append("source file matching %r" % a.file)
    if a.getfrom:
        scopes.append("reachable from %r" % a.getfrom)
    if scopes:
        print("\n(restricted to %s)" % ", ".join(scopes))
    report(old, new, d, ep, a.limit if a.limit and a.limit > 0 else None)
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Option 03 # ipsw diff:

Different from what have presented so far, ipsw diff command offers a view on changes for associated with entitlements, sandbox profiles, launchd configs, firmware payloads and feature flags, which previous options do not provide with none of it. Instead of working with kernelcaches, ipsw diff command handle the entire IPSW image, patch directories and OTAs. Of course, the initial cost is downloading two IPSW images of about 12 GB, but it makes part of the game. It is recommended to comment on restrictions on environment to perform these steps below.

- **Windows:** it is possible to run ipsw diff to get kernel version, kexts, sandbox profiles, firmware and iBoot. However, we cannot use Windows for getting entitlements and files because it is necessary to have apfs-fuse driver to mount the four APFS volumes.
- **macOS:** running ipsw diff option here we get all artifacts obtained on Windows, plus entitlements and files because macOS mounts APFS natively.
- **Linux:** it is another excellent possibility, and all artifacts can be obtained and analyzed using Linux too.

To initiate the procedure, check whether IPSW images are available:

```
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24A437 --urls
  • Querying AppleDB...
https://updates.cdn-apple.com/2026FallFCS/5130b3f9-3b4e-469a-b60e-93f6b310cdd9/iPhone17,3_27.0_24A437_Restore.ipsw

$ ipsw download appledb --os iOS --device iPhone17,3 --build 24B5084k --urls
  • Querying AppleDB...
https://updates.cdn-apple.com/2026FallSeed/b3d92a7e-9558-4709-8403-03e63f9152a5/iPhone17,3_27.2_24B5084k_Restore.ipsw
```

You can download directly using ipsw tool and save them into ipsw_diff directory, as shown below :

```
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24A437 -y -o ipsw_diff
$ ipsw download appledb --os iOS --device iPhone17,3 --build 24B5084k -y -o ipsw_diff
```

If you want to run the following commands on Linux, you can run the following commands:

```
$ mkdir research_ios
$ cd research_ios/
$ sudo apt-get update
$ sudo apt install cmake g++ git libfuse3-dev libbz2-dev zlib1g-dev libattr1-dev pkg-config curl
ca-certificates
$ mkdir github
$ cd github/
$ git clone -q --recursive https://github.com/sgan81/apfs-fuse.git
$ cd apfs-fuse/
$ mkdir -p build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release > /dev/null
$ make -j$(nproc)
$ sudo make install
$ sudo ldconfig
$ p=$(command -v apfs-fuse)
$ echo "  apfs-fuse: ${p:-NOT_ON_PATH}"
$ mkdir tools
$ cd tools/
$ curl -sL https://api.github.com/repos/blacktop/ipsw/releases/latest | grep -oE
'https://[^\"]*linux_x86_64\.tar\.gz' | head -5 | grep ipsw_
$ mkdir ipsw
$ cd ipsw
```

```
$ curl -sL
https://github.com/blacktop/ipsw/releases/download/v3.1.724/ipsw_3.1.724_linux_x86_64.tar.gz -o
ipsw.tgz
$ tar zxvf ipsw.tgz
$ ./ipsw
$ cd
$ vi .bashrc (and add the ipsw tool in the $PATH)
$ ./bashrc
$ mkdir tools
$ cd tools
$ git clone --depth 1 https://github.com/blacktop/symbolicator
```

The first step is to perform the diff, but ipsw diff option offers multiple options, and I usually pick up **--sandbox** (diff decompiled sandbox profiles), **--feat** (diff feature flags) and **--fw** (diff other firmware), as shown below:

```
$ ipsw diff iPhone17,3_27.0_24A437_Restore.ipsw iPhone17,3_27.2_24B5084k_Restore.ipsw --sandbox --
feat --fw --ignore-build-timestamps --signatures ../tools/symbolicator/kernel --markdown --output
diff_27.0_27.2 > diff_run.log 2>&1

$ more diff_run.log
  • using diff cache at
/home/administrator/.cache/ipsw/diffs/24A437_iPhone17_3_27.0_c55b82c750434c27__24B5084k_iPhone17_3_
27.2_a7107fbcc4707029.db
  • Diffing KERNELCACHES
  • Parsing Kernel Signatures
  • Symbolicating... kernelcache=/tmp/ipsw-
diff2626512044/24A437__iPhone17,3/kernelcache.release.iPhone17,3
...
...
  • Using pure Go implementation for AEA decryption
  • Using pure Go implementation for AEA decryption
  • Creating JSON diff file: diff_27.0_27.2/27_0_24A437_vs_27_2_24B5084k.json
```

A short sequence of queries follow:

```
$ J=diff_27.0_27.2/27_0_24A437_vs_27_2_24B5084k.json

$ jq -r '.kexts.new[]' $J
com.apple.iokit.EndpointSecuritySE
com.apple.kec.AppleEncryptedArchive

$ jq -r '.kexts.updated | keys[]' $J | head -10
com.apple.AGXFirmwareKextG17PRTBuddy
com.apple.AGXFirmwareKextRTBuddy64
com.apple.AGXG17P
com.apple.AUC
com.apple.AppleFSCompression.AppleFSCompressionTypeZlib
com.apple.EXBrightCalibrationConsumer
com.apple.EXBrightKext
com.apple.ExclaveKextClient
com.apple.audio.TrustedClockingKEXT
com.apple.driver.AFKHIDTBDevice

$ jq -r '.kexts.updated["com.apple.iokit.IOSurface"]' $J | head -10

```diff
-402.8.0.0.0
+403.1.0.0.0
 __TEXT.__cstring: 0x33b7
 __TEXT.__os_log: 0x3779
 __TEXT.__const: 0x60
- __TEXT_EXEC.__text: 0x3461c
+ __TEXT_EXEC.__text: 0x34634
 __TEXT_EXEC.__auth_stubs: 0x940
```

```
$ jq -r '.firmwares.updated | keys[]' $J | head -10
```

```
Firmware/M3/bin/securem3fw-d4y.im4p
Firmware/SmartIOFirmware_ASCv7.im4p
Firmware/agx/armfw_g17p.im4p/agx_a000
Firmware/agx/armfw_g17p.im4p/agx_a010
Firmware/agx/armfw_g17p.im4p/agx_b000
Firmware/agx/armfw_g17p.im4p/agx_b010
Firmware/agx/armfw_g17p.im4p/agx_b100
Firmware/ane/h17_ane_fw_theia_d9x.im4p
Firmware/ansf.t8140.release.im4p
Firmware/ave/AppleAVE2FW_H17.im4p
```

```
$ jq -r '.iboot.new | keys[]' $J | head -10
```

```
AppleSMCFirmware.bin
RTKit.bin
iboot
iboot_blob30.bin
iboot_blob32.bin
iboot_blob41.bin
```

```
$ jq -r '.iboot.removed | keys[]' $J | head -10
```

```
AppleSMCFirmware.bin
iboot
iboot_blob29.bin
iboot_blob30.bin
iboot_blob32.bin
iboot_blob41.bin
```

Thus, running simple queries using jq command is one of quick ways to discover changes for features, firmware, kext and sandbox. I left entitlements (ent argument) out of the first command to execute a second ipsw command to create a database for them:

```
$ ipsw ent --sqlite ipsw_diff.db --ipsw iPhone17,3_27.0_24A437_Restore.ipsw --ipsw
iPhone17,3_27.2_24B5084k_Restore.ipsw
```

- Processing IPSW ipsw=iPhone17,3\_27.0\_24A437\_Restore.ipsw
  - Generating entitlement database file...
    - Scanning AppOS
    - Scanning SystemOS
- Using pure Go implementation for AEA decryption
  - Scanning FileSystem
- Using pure Go implementation for AEA decryption
  - Scanning ExclaveOS
- Using pure Go implementation for AEA decryption
- Successfully processed ipsw=iPhone17,3\_27.0\_24A437\_Restore.ipsw
- Processing IPSW ipsw=iPhone17,3\_27.2\_24B5084k\_Restore.ipsw
  - Generating entitlement database file...
    - Scanning AppOS
    - Scanning SystemOS
- Using pure Go implementation for AEA decryption
  - Scanning FileSystem
- Using pure Go implementation for AEA decryption
  - Scanning ExclaveOS
- Using pure Go implementation for AEA decryption
- Successfully processed ipsw=iPhone17,3\_27.2\_24B5084k\_Restore.ipsw
- Database creation/update completed successfully

```
$ ls -lh ipsw_diff.db
```

```
-rw-r--r-- 1 administrator administrator 11M Sep 25 00:49 ipsw_diff.db
```

```
$ sqlite3 -header -column ipsw_diff.db "SELECT i.buildid, i.version, COUNT(DISTINCT e.path_id) AS
binaries, COUNT(*) AS assignments FROM ipsws i LEFT JOIN entitlements e ON e.ipsw_id=i.id GROUP BY
i.id;"
```

```
buildid version binaries assignments
```

```

24A437 27.0 2288 56030
24B5084k 27.2 2313 56481
```

# entitlement keys new in 27.2 beta, ranked by how many binaries hold each

```
$ sqlite3 -header -column ipsw_diff.db "WITH nk AS (SELECT k.id,k.key FROM entitlement_keys k WHERE
k.id IN (SELECT e.key_id FROM entitlements e JOIN ipsws i ON i.id=e.ipsw_id WHERE
i.buildid='24B5084k') AND k.key NOT IN (SELECT k2.key FROM entitlements e2 JOIN entitlement_keys k2
ON k2.id=e2.key_id JOIN ipsws i2 ON i2.id=e2.ipsw_id WHERE i2.buildid='24A437')) SELECT
n.key,COUNT(DISTINCT e.path_id) AS holders FROM nk n JOIN entitlements e ON e.key_id=n.id GROUP BY
n.key ORDER BY holders DESC,n.key;" | head -15
```

key	holders
com.apple.security.hardened-process.checked-allocations.enforce-checked-pointer-arithmetic-overflow	11
com.apple.private.airdrop.client	5
com.apple.private.InstallCoordination.GetAppReplacementSource	4
com.apple.developer.conversation-accessibility	3
com.apple.private.audio.driver.extrinsic.registration	3
com.apple.private.InstallCoordination.ReplaceApp	2
com.apple.private.accessories.transport-client	2
com.apple.private.darwin-notification.restrict-post.AirPlay.DACP.mutetoggle	2
com.apple.private.darwin-notification.restrict-post.AirPlay.DACP.volumedown	2
com.apple.private.darwin-notification.restrict-post.AirPlay.DACP.volumeup	2
com.apple.private.installcoordination.app-data-migration.host	2
com.apple.HomePodDisplay	1
com.apple.appleaccount.transparency.metadata	1

# binaries that appeared in 27.2, ranked by how many entitlements each carries:

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT COUNT(*) AS ents,
replace(p.path,rtrim(p.path,replace(p.path,'/','')),') AS binary FROM paths p JOIN entitlements e
ON e.path_id=p.id JOIN ipsws i ON i.id=e.ipsw_id WHERE i.buildid='24B5084k' AND p.path NOT IN
(SELECT p2.path FROM paths p2 JOIN entitlements e2 ON e2.path_id=p2.id JOIN ipsws i2 ON
i2.id=e2.ipsw_id WHERE i2.buildid='24A437') GROUP BY p.path ORDER BY ents DESC, binary;" | head -15
```

ents	binary
388	Siri AI
24	speechmodeltrainingd
22	healthcontentd
20	MomentsAlgorithmsService
20	SiriExtensionsDigestExtension
18	SNCFeaturesPlugin
18	frauddefensepfl
11	HealthReportDiagnosticExtension
10	SiriLogProcessor
9	AirPlayDiagnosticExtension
9	HealthAgeDiagnosticExtension
9	memoryanalyticd
6	IDSDiagnosticExtension

# who holds the lifs provider entitlement, per build

```
$ sqlite3 -header -column ipsw_diff.db "SELECT i.buildid, p.path FROM entitlements e JOIN
entitlement_keys k ON k.id=e.key_id JOIN paths p ON p.id=e.path_id JOIN ipsws i ON i.id=e.ipsw_id
WHERE k.key='com.apple.developer.fskit.fsmodule' ORDER BY i.buildid, p.path;" | head -5
```

buildid	path
24A437	/System/Library/ExtensionKit/Extensions/com.apple.fskit.exfat.appex/com.apple.fskit.exfat
24A437	/System/Library/ExtensionKit/Extensions/com.apple.fskit.msdos.appex/com.apple.fskit.msdos
24A437	/System/Library/PrivateFrameworks/SMBClientProvider.framework/smbclientd
24A437	/System/Library/PrivateFrameworks/UVFSXPCService.framework/XPCServices/UVFSService.xpc/UVFSService
24B5084k	/System/Library/ExtensionKit/Extensions/com.apple.fskit.exfat.appex/com.apple.fskit.exfat

# list binaries that has same key (entitlement), but value has changed



```
$ sqlite3 ipsw_diff.db ".mode line" "SELECT replace(p.path,rtrim(p.path,replace(p.path,'/','')),') AS binary, replace(replace(k.key,'com.apple.private.','priv:'),'com.apple.security.','sec:') AS key, replace(replace(replace(replace(replace(MAX(CASE WHEN i.buildid='24A437' THEN v.value END),'kTCCService',''),['\"','\"'],'\"'),'\\"','\"'),'\\"','\"') AS old, replace(replace(replace(replace(replace(MAX(CASE WHEN i.buildid='24B5084k' THEN v.value END),'kTCCService',''),['\"','\"'],'\"'),'\\"','\"'),'\\"','\"') AS new FROM entitlements e JOIN ipsws i ON i.id=e.ipsw_id JOIN paths p ON p.id=e.path_id JOIN entitlement_keys k ON k.id=e.key_id JOIN entitlement_values v ON v.id=e.value_id GROUP BY p.path,k.key HAVING old IS NOT NULL AND new IS NOT NULL AND old<>new ORDER BY binary,key;" | head -20
```

```
binary = AMSEngagementViewService
 key = sec:exception.mach-lookup.global-name
 old =
com.apple.AppleMediaServicesUIDynamicService,com.apple.appstored.xpc.storequeue,com.apple.appstorea
gent.xpc,com.apple.appstored.xpc,com.apple.appstorecomponentsd.xpc,com.apple.askpermissiond,com.app
le.coreidvd.docUpload.xpc,com.apple.identityservicesd.embedded.auth,com.apple.itunescloud.music-
subscription-status-
service,com.apple.mobileactivationd,com.apple.nfcd.hwmanager,com.apple.passd.account,com.apple.pass
d.in-app-
payment,com.apple.passd.library,com.apple.passd.payment,com.apple.seserviced,com.apple.surfboard.sc
enesessionservice,com.apple.xpc.amsengagementd,com.apple.xpc.amsaccounts
```

```
new =
com.apple.AppleMediaServicesUIDynamicService,com.apple.appstored.xpc.storequeue,com.apple.appstorea
gent.xpc,com.apple.appstored.xpc,com.apple.appstorecomponentsd.xpc,com.apple.askpermissiond,com.app
le.coreidvd.docUpload.xpc,com.apple.fairplaydeviceidentityd,com.apple.identityservicesd.embedded.au
th,com.apple.itunescloud.music-subscription-status-
service,com.apple.mobileactivationd,com.apple.nfcd.hwmanager,com.apple.passd.account,com.apple.pass
d.in-app-
payment,com.apple.passd.library,com.apple.passd.payment,com.apple.seserviced,com.apple.surfboard.sc
enesessionservice,com.apple.xpc.amsengagementd,com.apple.xpc.amsaccounts
```

```
binary = AXUIViewService
 key = sec:exception.shared-preference.read-write
 old =
com.apple.SpeakSelection,com.apple.Accessibility,com.apple.Accessibility.GuidedAccess,com.apple.Acc
essibility,com.apple.speech.SpeechRecognitionCommandAndControl,com.apple.VoiceOverTouch,com.apple.V
oiceOverTouch.activities,com.apple.ZoomTouch,com.apple.keyboard.ContinuousPath
```

```
new =
com.apple.SpeakSelection,com.apple.Accessibility,com.apple.Accessibility.GuidedAccess,com.apple.Acc
essibility,com.apple.speech.SpeechRecognitionCommandAndControl,com.apple.VoiceOverTouch,com.apple.V
oiceOverTouch.activities,com.apple.ZoomTouch,com.apple.keyboard.ContinuousPath,com.apple.Preference
s
```

```
binary = AgeVerificationExtension
 key = sec:exception.files.home-relative-path.read-write
 old = /tmp/com.apple.AppleMediaServices/
 new =
/tmp/com.apple.AppleMediaServices/,/private/var/MobileAsset/AssetsV2/com_apple_MobileAsset_Vision_A
geEstimation/,/private/var/MobileAsset/PreinstalledAssetsV2/InstallWithOs/com_apple_MobileAsset_Vis
ion_AgeEstimation/,/private/var/MobileAsset/AssetsV2/com_apple_MobileAsset_Vision_FaceLiveliness/,/
private/var/MobileAsset/PreinstalledAssetsV2/InstallWithOs/com_apple_MobileAsset_Vision_FaceLivelin
ess/
```

```
binary = AppStore
 key = sec:exception.files.absolute-path.read-only
 old = /private/var/mobile/Library/Application
Support/com.apple.ap.promotedcontentd/shared/all/ro/
 new = /private/var/mobile/Library/Application
Support/com.apple.ap.promotedcontentd/shared/all/ro/,/private/var/db/os_eligibility/eligibility.pli
st
```

# list existing binaries that has gained one or more key (entitlement)

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT
replace(p.path,rtrim(p.path,replace(p.path,'/','')),') AS binary, k.key FROM entitlements e JOIN
ipsws i ON i.id=e.ipsw_id JOIN paths p ON p.id=e.path_id JOIN entitlement_keys k ON k.id=e.key_id
WHERE i.buildid='24B5084k' AND p.path IN (SELECT p2.path FROM paths p2 JOIN entitlements e2 ON
e2.path_id=p2.id JOIN ipsws i2 ON i2.id=e2.ipsw_id WHERE i2.buildid='24A437') AND NOT EXISTS
```

```
(SELECT 1 FROM entitlements e3 JOIN ipsws i3 ON i3.id=e3.ipsw_id WHERE i3.buildid='24A437' AND e3.path_id=e.path_id AND e3.key_id=e.key_id) ORDER BY binary, k.key;" | head -15
```

binary	key
AAUIFollowUpExtension	com.apple.cdp.statemachine
AAUIFollowUpExtension	com.apple.cdp.utility
AAUIViewService	com.apple.cdp.followup
AAUIViewService	com.apple.cdp.utility
ADAskForExceptionExtension	com.apple.security.hardened-process.checked-allocations.enfo
ADAskForExceptionExtension	rce-checked-pointer-arithmetic-overflow
ADFollowUpExtension	com.apple.security.hardened-process.enhanced-security-versio
ADFollowUpExtension	n-string
ADFollowUpExtension	com.apple.security.hardened-process.checked-allocations.enfo
ADFollowUpExtension	rce-checked-pointer-arithmetic-overflow
ADFollowUpExtension	com.apple.security.hardened-process.enhanced-security-versio
ADFollowUpExtension	n-string
AKFollowUpExtension	com.apple.cdp.statemachine

# Entitlements held by exactly one binary

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT k.key, replace(p.path,rtrim(p.path,replace(p.path,'/','')),') AS sole_holder FROM entitlement_keys k JOIN entitlements e ON e.key_id=k.id JOIN paths p ON p.id=e.path_id JOIN ipsws i ON i.id=e.ipsw_id WHERE i.buildid='24B5084k' GROUP BY k.key HAVING COUNT(DISTINCT e.path_id)=1 ORDER BY k.key;" | head -15
```

key	sole_holder
com.apple.duet.activitiescheduler.allow	dbtelemetryd
com.apple.private.iokit.batterydataprecise	sharingd
com.apple.private.spindump.cli	wifivelocityd
com.apple.security.app-sandbox	wifip2pd
AO2CCNT0	corecaptured
AOPACCT0	royacaptured
AOPACCT1	royacaptured
AOPACCT2	royacaptured
AOPACCT3	royacaptured
AOPACCT4	royacaptured
AOPAOMU0	royacaptured
AOPAUAT0	royacaptured
AOPCAPT0	royacaptured

# obtain every distinct TCC grant, how many binaries hold it, and in which builds

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT v.value AS services, COUNT(DISTINCT e.path_id) AS holders, GROUP_CONCAT(DISTINCT i.buildid) AS builds FROM entitlements e JOIN entitlement_keys k ON k.id=e.key_id JOIN entitlement_values v ON v.id=e.value_id JOIN ipsws i ON i.id=e.ipsw_id WHERE k.key LIKE 'com.apple.private.tcc.allow%' GROUP BY v.value ORDER BY holders DESC;" | head -10
```

services	holders	builds
["kTCCServiceAddressBook"]	161	24A437, 24B5084k
["kTCCServiceLiverpool"]	55	24A437, 24B5084k
["kTCCServicePhotos"]	44	24A437, 24B5084k
["kTCCServiceMediaLibrary"]	39	24A437, 24B5084k
["kTCCServiceWillow"]	22	24A437, 24B5084k
["kTCCServiceCamera"]	20	24A437, 24B5084k
["kTCCServiceFaceID"]	19	24A437, 24B5084k
["kTCCServiceMicrophone"]	17	24A437, 24B5084k

# same from previous one, but ordered by most privileged grants first, and a count of the # services in the array.

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT replace(replace(replace(replace(v.value, 'kTCCService', ''), '[\"', '\"]', '\\"'), '\\', '\\\\'), '\\', '\\\\') AS services, (length(v.value)-length(replace(v.value, '\\', '\\\\')))+1 AS n_services, COUNT(DISTINCT e.path_id) AS holders FROM entitlements e JOIN entitlement_keys k ON k.id=e.key_id JOIN entitlement_values v ON v.id=e.value_id WHERE k.key LIKE 'com.apple.private.tcc.allow%' GROUP BY v.value ORDER BY n_services DESC LIMIT 5;"
```

services	n_services	holders
AddressBook,BluetoothAlways,Calendar,Camera,ExposureNotification,FaceID,Location,MediaLibrary,Microphone,MicrophoneInjection,Photos,PhotosAdd,Reminders,SpeechRecognition,VoiceBanking,WebKitIntelligentTrackingPrevention,Willow	17	1
AddressBook,Calendar,Reminders,Photos,PhotosAdd,MediaLibrary,Microphone,Camera,Willow,FaceID,BluetoothAlways,MediaLibrary	12	1
AddressBook,Calendar,Camera,MediaLibrary,Microphone,Motion,Photos,PhotosAdd,Reminders,Reminders,SpeechRecognition,Willow	12	1
Photos,Microphone,SpeechRecognition,AddressBook,MediaLibrary,Camera,ListenEvent,Motion,VoiceBanking,MicrophoneInjection,FaceID	11	1
Calendar,AddressBook,Liverpool,Photos,Willow,Reminder,MediaLibrary,HealthKit,SystemPolicyRemovableVolumes,SystemPolicyNetworkVolumes,SiriAccess	11	1

# TCC grants that exist in only one of the two builds | the gains and losses.

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT
replace(replace(replace(replace(v.value,'kTCCService',''),'[\','','\'],''),'','\','\','','','') AS
services, COUNT(DISTINCT e.path_id) AS holders, GROUP_CONCAT(DISTINCT i.buildid) AS builds FROM
entitlements e JOIN entitlement_keys k ON k.id=e.key_id JOIN entitlement_values v ON
v.id=e.value_id JOIN ipsws i ON i.id=e.ipsw_id WHERE k.key LIKE 'com.apple.private.tcc.allow%'
GROUP BY v.value HAVING builds NOT LIKE '%,%' ORDER BY holders DESC;"
```

services	holders	builds
SiriAccess	2	24B5084k
Willow,Photos,Camera,AddressBook,Microphone	1	24A437
Willow,AddressBook,Calendar	1	24A437
Willow,AddressBook,Calendar,SiriAccess	1	24B5084k
Calendar,AddressBook,Liverpool,Photos,Willow,Reminder,MediaLibrary,SystemPolicyRemovableVolumes,SystemPolicyNetworkVolumes,SiriAccess	1	24A437
Calendar,AddressBook,Liverpool,Photos,Willow,Reminder,MediaLibrary,HealthKit,SystemPolicyRemovableVolumes,SystemPolicyNetworkVolumes,SiriAccess	1	24B5084k
AddressBook,Camera,Microphone,Photos,Willow	1	24B5084k

# TCC grants (changes first)

```
$ sqlite3 ipsw_diff.db ".mode markdown" "SELECT v.value AS services, COUNT(DISTINCT e.path_id) AS
holders, GROUP_CONCAT(DISTINCT i.buildid) AS builds FROM entitlements e JOIN entitlement_keys k ON
k.id=e.key_id JOIN entitlement_values v ON v.id=e.value_id JOIN ipsws i ON i.id=e.ipsw_id WHERE
k.key LIKE 'com.apple.private.tcc.allow%' GROUP BY v.value ORDER BY (builds LIKE '%,%'), holders
DESC;" | head -15
```

services	holders	builds
["kTCCServiceSiriAccess"]	2	24B5084k
["kTCCServiceAddressBook","kTCCServiceCamera","kTCCServiceMicrophone","kTCCServicePhotos","kTCCServiceWillow"]	1	24B5084k
["kTCCServiceCalendar","kTCCServiceAddressBook","kTCCServiceLiverpool","kTCCServicePhotos","kTCCServiceWillow","kTCCServiceReminder","kTCCServiceMediaLibrary","kTCCServiceHealthKit","kTCCServiceSystemPolicyRemovableVolumes","kTCCServiceSystemPolicyNetworkVolumes","kTCCServiceSiriAccess"]	1	24B5084k
["kTCCServiceCalendar","kTCCServiceAddressBook","kTCCServiceLiverpool","kTCCServicePhotos","kTCCServiceWillow","kTCCServiceReminder","kTCCServiceMediaLibrary","kTCCServiceSystemPolicyRemovableVolumes","kTCCServiceSystemPolicyNetworkVolumes","kTCCServiceSiriAccess"]	1	24A437

The entitlement com.apple.private.tcc.allow is the entitlement that lets a process take a TCC capability without prompting the user. About SQL itself, readers can test different combinations and adjusts by using typical keywords:

- **WHERE:** it determines which entitlement to look at (ex: WHERE k.key LIKE 'com.apple.private.tcc.allow%').
- **HAVING:** it focuses on only rows that have changed (ex: HAVING builds NOT LIKE '%,%').
- **ORDER BY:** it helps to answer the question being asked (ex: ORDER BY holders).

### Option 04 # Diaphora:

This option is the only ones that offer semantic perspective and, as a consequence, noises like register reallocation, stack relay out, inlining of existing functions does not impact the analysis and interpretation. Furthermore, there are thousands of functions without identification (sub\_), but most of them are not possible to identify the functionality because they do not call a single named function. It is exactly at this point that Diaphora allows to join two different builds by identity because it recognizes function's shape, and it allow us to follow unnamed functions across builds.

Initially, it could not seem a fundamental issue, but it is because previous approaches failed in determined situations:

- **SQLite Diffing:** it performs joins between two builds using a normalized body hash, but it fails when the body of the function changes.
- **Symbol and Size diff:** it performs joins between two builds using name, so when a name is not present, it fails.
- **ipsw diff:** it performs joins by using files, kexts and strings, but it does not look at one function in special.

The approach using Diaphora can help us to understand thousands of functions that are present in iOS 27 GA and that do not match anything in iOS 27.2 beta 1, but the cost of this operation may be a considerable amount of time to finish against the full kernelcache. An alternative is to investigate only one component (a kext), which is faster and you get answers after some minutes.

To install Diaphora, execute the following steps:

- git clone <https://github.com/joxeankoret/diaphora>
- (IDA Pro 9.4) hcli plugin install <https://github.com/joxeankoret/diaphora/archive/refs/tags/3.4.2.zip>
- (IDA Pro 8.4) Copy plugins/diaphora\_plugin.py and plugins/diaphora\_plugin.cfg to the IDA's plugins directory.
- (IDA Pro 8.4) Edit diaphora\_plugin.cfg and set the path value to the Diaphora's directory.

Once Diaphora is installed, steps consist in using it to analyze and export both IDA Pro databases for kernelcaches (iOS 27.0 GA and iOS 27.2 beta 1) by executing the following commands:

```
(copy the first database) $ Copy-Item C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.bin.i64
C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.dia.i64
(copy the second database) $ Copy-Item C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.bin.i64
C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.dia.i64
```

To focus on only a range address of a kext rather than entire kernelcache, query this range and use it with Diaphora. To the first database (iOS 27 GA), execute:

```
$ C:\sqlite\sqlite3.exe C:\research_ios\kernelcaches\ios27ga\corpus_ga.db "SELECT MIN(address),
MAX(end_address), COUNT(*) FROM functions WHERE segment LIKE '%IOSurface%__text';"
```

MIN(address)	MAX(end_address)	COUNT(*)
0xFFFFFFFF00A328180	0xFFFFFFFF00A35C79C	1183

Repeating the procedure to the second database (iOS 27.2 beta 1):

```
$ C:\sqlite\sqlite3.exe C:\research_ios\kernelcaches\ios272b1\corpus_272.db "SELECT MIN(address), MAX(end_address), COUNT(*) FROM functions WHERE segment LIKE '%IOSurface%__text';"
```

MIN(address)	MAX(end_address)	COUNT(*)
0xFFFFFFFF00A45FD00	0xFFFFFFFF00A494334	1183

Next step export both ranges, which can be started together because they do not interfere with each other, and finish in a few minutes:

```
$ $ida = 'C:\Program Files\IDA Professional 9.4\idat.exe'
$ $env:DIAPHORA_AUTO = "1"
$ $env:DIAPHORA_LOG_PRINT = "1"

$ $env:DIAPHORA_FROM_ADDRESS = "FFFFFFFF00A328180"
$ $env:DIAPHORA_TO_ADDRESS = "FFFFFFFF00A35D0DC"
$ $env:DIAPHORA_EXPORT_FILE = "C:\research_ios\kernelcaches\dia_ga_iosurface.sqlite"
$ & $ida -A "-SC:\research_ios\kernelcaches\corpus_tools\dia_wrap.py" "-
LC:\research_ios\kernelcaches\dia_ga_ios.log"
"C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.dia.i64"

$ $env:DIAPHORA_FROM_ADDRESS = "FFFFFFFF00A45FD00"
$ $env:DIAPHORA_TO_ADDRESS = "FFFFFFFF00A494C74"
$ $env:DIAPHORA_EXPORT_FILE = "C:\research_ios\kernelcaches\dia_272_iosurface.sqlite"
$ & $ida -A "-SC:\research_ios\kernelcaches\corpus_tools\dia_wrap.py" "-
LC:\research_ios\kernelcaches\dia_272_ios.log"
"C:\research_ios\kernelcaches\ios272b1\kc_27.2b1.dia.i64"
```

The script used above is a wrapper and its content follows below (note that one line needs to be edited):

```
$ more C:\research_ios\kernelcaches\corpus_tools\dia_wrap.py
```

```
r"""dia_wrap.py - the headless Diaphora export driver.
"""
import sys

sys.path.insert(0, r"C:\github\diaphora")

import idaapi
import idc
idaapi.auto_wait()
import diaphora_ida

diaphora_ida.main()
idc.qexit(0)
```

Check the exports:

```
$ C:\sqlite\sqlite3.exe C:\research_ios\kernelcaches\dia_ga_iosurface.sqlite "SELECT COUNT(*), (SELECT COUNT(*) FROM functions WHERE name LIKE 'sub_%'), (SELECT COUNT(*) FROM instructions) FROM functions;"
```

COUNT(*)	(SELECT COUNT(*) ...	(SELECT COUNT(*) ...
1183	906	226191

```
$ C:\sqlite\sqlite3.exe C:\research_ios\kernelcaches\dia_272_iosurface.sqlite "SELECT COUNT(*),
(SELECT COUNT(*) FROM functions WHERE name LIKE 'sub_%'), (SELECT COUNT(*) FROM instructions) FROM
functions;"
```

COUNT(*)	(SELECT COUNT(*) ...	(SELECT COUNT(*) ...
1183	906	226224

After having both databases (associated to kext), execute the following diffing (it is only ONE line):

```
$ C:\research_ios\kernelcaches\corpus_tools\dia_diff_run.ps1 -Db1
C:\research_ios\kernelcaches\dia_ga_iosurface.sqlite -Db2
C:\research_ios\kernelcaches\dia_272_iosurface.sqlite -Out
C:\research_ios\kernelcaches\dia_diff_iosurface.sqlite -Idb
C:\research_ios\kernelcaches\ios27ga\kc_27.0ga.dia.i64
```

```
wall seconds : 140
[Diaphora: Sat Sep 26 20:24:45 2026] CONTROL cpu_count=15 IS_IDA=True
[Diaphora: Sat Sep 26 20:24:45 2026] CONTROL slow=True experimental=True unreliable=False
[Diaphora: Sat Sep 26 20:24:45 2026] CONTROL
project_script=C:\research_ios\kernelcaches\corpus_tools\dia_hooks.py
[Diaphora: Sat Sep 26 20:24:45 2026] CONTROL
db1=C:\research_ios\kernelcaches\dia_ga_iosurface.sqlite
[Diaphora: Sat Sep 26 20:24:45 2026] CONTROL
db2=C:\research_ios\kernelcaches\dia_272_iosurface.sqlite
[Diaphora: Sat Sep 26 20:24:45 2026] Current results: Best 664, Partial 63, Unreliable 0
[Diaphora: Sat Sep 26 20:26:43 2026] Current results: Best 675, Partial 293, Unreliable 0
[Diaphora: Sat Sep 26 20:26:44 2026] Final results: Best 675, Partial 151, Unreliable 0,
Multimatches 284
[Diaphora: Sat Sep 26 20:26:44 2026] Diffing results saved in file
'C:\research_ios\kernelcaches\dia_diff_iosurface.sqlite'.
[Diaphora: Sat Sep 26 20:26:44 2026] RESULT diff seconds 118.8
[Diaphora: Sat Sep 26 20:26:44 2026] RESULT save seconds 0.0
[Diaphora: Sat Sep 26 20:26:44 2026] RESULT total seconds 118.8
output : 274432 bytes
```

The PowerShell script used in the command line is the following:

```
dia_diff_run.ps1 - run a Diaphora diff INSIDE IDA, where parallelism is not disabled.
```

```
param(
 [Parameter(Mandatory = $true)][string]$Db1,
 [Parameter(Mandatory = $true)][string]$Db2,
 [Parameter(Mandatory = $true)][string]$Out,
 [Parameter(Mandatory = $true)][string]$Idb,
 [string]$Log = "",
 [string]$Hooks = "<default>",
 [string]$Ida = 'C:\Program Files\IDA Professional 9.4\idat.exe'
)

$ErrorActionPreference = 'Stop'
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$wrap = Join-Path $here 'dia_diff_wrap.py'

foreach ($p in @($Ida, $wrap, $Db1, $Db2, $Idb)) {
 if (-not (Test-Path $p)) { throw "not found: $p" }
}
if ($Log -eq "") { $Log = [IO.Path]::ChangeExtension($Out, '.log') }

$stem = [IO.Path]::Combine((Split-Path $Idb), [IO.Path]::GetFileNameWithoutExtension($Idb))
$open = Get-ChildItem "$stem.i*" -ErrorAction SilentlyContinue
if ($open.Count -ne 1) {
 throw "the IDA database looks open or killed ($($open.Count) files match $stem.i*); use a
closed one"
}
```



```
if ($Db1 -eq $Db2) { throw "Db1 and Db2 are the same file" }
if ($Out -eq $Db1 -or $Out -eq $Db2) { throw "Out would overwrite an input" }

if ($Hooks -eq "<default>") { $Hooks = Join-Path $here 'dia_hooks.py' }
if ($Hooks -ne "" -and -not (Test-Path $Hooks)) { throw "project script not found: $Hooks" }
if ($Hooks -eq "") {
 Write-Warning "No project script: the callee diffing phases stay enabled and may not
terminate."
}

$env:X_DB1 = (Resolve-Path $Db1).Path
$env:X_DB2 = (Resolve-Path $Db2).Path
$env:X_OUT = $Out
$env:X_HOOKS = $Hooks

Write-Host "hooks : $(if ($Hooks -eq '') { '** NONE **' } else { $Hooks })"
Write-Host "db1 (OLD) : $env:X_DB1"
Write-Host "db2 (NEW) : $env:X_DB2"
Write-Host "out : $env:X_OUT"
Write-Host "log : $Log"
Write-Host "idb : $Idb"

$t0 = Get-Date
& $Ida -A "-S$wrap" "-L$Log" $Idb
$secs = [math]::Round(((Get-Date) - $t0).TotalSeconds, 1)

Write-Host ""
Write-Host "wall seconds : $secs"
if (Test-Path $Log) { Select-String -Path $Log -Pattern 'CONTROL|RESULT' | ForEach-Object { $_.Line
} }
if (Test-Path $env:X_OUT) {
 Write-Host "output : $((Get-Item $env:X_OUT).Length) bytes"
} else {
 Write-Host "output : ** NOT WRITTEN - read the log **"
}
}
```

The [dia\\_diff\\_run.ps1](#) depends on two other scripts, [dia\\_diff\\_wrap.py](#) and [dia\\_hooks.py](#).

The [dia\\_diff\\_wrap.py](#) follows below:

```
r"""dia_diff_wrap.py - run a Diaphora diff inside IDA.
"""

import os
import sys
import time

sys.path.insert(0, r"C:\github\diaphora")

import idaapi
import idc

idaapi.auto_wait()

import diaphora
import diaphora_ida

DB1 = os.environ["X_DB1"]
DB2 = os.environ["X_DB2"]
OUT = os.environ["X_OUT"]

if os.path.exists(OUT):
 os.remove(OUT)

bd = diaphora_ida.CIDABinDiff(DB1)
bd.db = diaphora.sqlite3_connect(DB1)
```



```
if os.environ.get("X_FAST"):
 bd.slow_heuristics = False
 bd.experimental = False
 bd.unreliable = False

if os.environ.get("X_HOOKS"):
 bd.project_script = os.environ["X_HOOKS"]

diaphora.log("CONTROL cpu_count=%d IS_IDA=%s" % (bd.cpu_count, diaphora.IS_IDA))
diaphora.log("CONTROL slow=%s experimental=%s unreliable=%s"
 % (bd.slow_heuristics, bd.experimental, bd.unreliable))
diaphora.log("CONTROL project_script=%s" % bd.project_script)
diaphora.log("CONTROL db1=%s" % DB1)
diaphora.log("CONTROL db2=%s" % DB2)

t0 = time.monotonic()
bd.diff(DB2)
t1 = time.monotonic()
bd.save_results(OUT)
t2 = time.monotonic()

diaphora.log("RESULT diff seconds %.1f" % (t1 - t0))
diaphora.log("RESULT save seconds %.1f" % (t2 - t1))
diaphora.log("RESULT total seconds %.1f" % (t2 - t0))

idc.qexit(0)
```

The `dia_hooks.py` script follow below:

```
r"""dia_hooks.py - a Diaphora project script that disables the diff phases which do not terminate.
"""

import os

SKIP = [
 "Callee found diffing matches assembly",
 "Callee found diffing matches pseudo-code",
 "Related compilation unit",
 "Same constants related matches",
 "Local affinity",
]

class CSkipUnboundedPhases:
 def __init__(self, diaphora_obj):
 self.diaphora = diaphora_obj
 keep = os.environ.get("X_KEEP", "")
 self.keep = set(x.strip() for x in keep.split(",") if x.strip())
 self.skip = [h for h in SKIP if h not in self.keep]
 self.seen = set()
 self._log("HOOKS loaded, skipping %d of %d hookable phases" % (len(self.skip), len(SKIP)))
 for h in self.skip:
 self._log("HOOKS will skip '%s'" % h)
 for h in sorted(self.keep):
 self._log("HOOKS asked to KEEP '%s'" % h)

 def _log(self, msg):
 try:
 self.diaphora.log(msg)
 except Exception:
 print(msg)

 def on_special_heuristic(self, heur, iteration):
 allowed = heur not in self.skip
 key = (heur, allowed)
 if key not in self.seen:
 self.seen.add(key)
 self._log("HOOKS %s '%s' (iteration %s)"
```

```
% ("allowing" if allowed else "SKIPPING", heur, iteration))
return allowed
```

```
HOOKS = {"DiaphoraHooks": CSkipUnboundedPhases}
```

There is a series of queries that are useful once you have diffing database, and the following ones are only a few examples:

```
$ $DB = "C:\research_ios\kernelcaches\dia_diff_iosurface.sqlite"
```

```
check how many functions matched and how well each match was. Notes about types and # functions:
```

```
#
best: strong match, including same name.
partial: matched.
multimatch: matched more than one candidate.
unreliable: weak heuristics.
primary: only in the new build (it was added)
secondary: only in the old build (it was removed)
```

```
$ C:\sqlite\sqlite3.exe -markdown $DB "SELECT type, COUNT(*) AS rows_, COUNT(DISTINCT address) AS
functions FROM results GROUP BY type UNION ALL SELECT 'unmatched ' || type, COUNT(*), COUNT(*) FROM
unmatched GROUP BY type;"
```

type	rows_	functions
best	675	675
multimatch	284	142
partial	151	151
unmatched primary	213	213
unmatched secondary	213	213

```
it shows what an unnamed function became (important answer)
```

```
$ C:\sqlite\sqlite3.exe -markdown $DB "SELECT name AS old, name2 AS new, nodes1 AS blocks, ratio,
description FROM results WHERE name LIKE 'sub_-' AND name2 LIKE 'sub_-' AND description NOT IN
('Equal assembly', 'Same cleaned assembly', 'Bytes hash') ORDER BY CAST(nodes1 AS INTEGER) DESC,
CAST(ratio AS REAL) DESC LIMIT 10;"
```

old description	new	blocks	ratio	
sub_FFFFFFFF00A33F054	sub_FFFFFFFF00A476BD4	62	0.9900000	Mnemonics small-primes-product
sub_FFFFFFFF00A3286E0	sub_FFFFFFFF00A460260	33	0.9900000	Mnemonics small-primes-product
sub_FFFFFFFF00A345E08	sub_FFFFFFFF00A47D988	32	0.9900000	Mnemonics small-primes-product
sub_FFFFFFFF00A354C9C	sub_FFFFFFFF00A48C834	26	0.9900000	Loop count
sub_FFFFFFFF00A32D590	sub_FFFFFFFF00A465110	26	0.9900000	Same nodes, edges, loops and strongly connected components
sub_FFFFFFFF00A343FF4	sub_FFFFFFFF00A47BB74	26	0.9900000	Mnemonics small-primes-product
sub_FFFFFFFF00A3557C4	sub_FFFFFFFF00A48D35C	19	0.9900000	Same nodes, edges, loops and strongly connected components
sub_FFFFFFFF00A348D24	sub_FFFFFFFF00A4808B4	17	0.9900000	Same nodes, edges, loops and strongly connected components
sub_FFFFFFFF00A3591FC	sub_FFFFFFFF00A490D94	16	0.9922857	Same nodes, edges, loops and strongly connected components
sub_FFFFFFFF00A3589CC	sub_FFFFFFFF00A490564	14	0.9900000	Same nodes, edges, loops and strongly connected components

```
Which matches we can distrust (least confidence and not a ranking of change)
```

```
$ C:\sqlite\sqlite3.exe -markdown $DB "SELECT ratio, name AS old, nodes1 AS blocks, description
FROM results WHERE CAST(ratio AS REAL) < 1.0 ORDER BY CAST(ratio AS REAL) ASC LIMIT 10;"
```

ratio	old	blocks	description
-------	-----	--------	-------------

0.7242857	sub_FFFFFFFF00A342980	2	Mnemonics small-primes-product
0.7350000	sub_FFFFFFFF00A3448D0	2	Mnemonics small-primes-product
0.8385294	sub_FFFFFFFF00A343F10	1	Same low complexity and names
0.8385294	sub_FFFFFFFF00A343F10	1	Same low complexity and names
0.8385294	sub_FFFFFFFF00A34447C	1	Same low complexity and names
0.8385294	sub_FFFFFFFF00A34447C	1	Same low complexity and names
0.8433333	sub_FFFFFFFF00A35B5CC	2	Same low complexity and names
0.8910000	sub_FFFFFFFF00A35426C	4	Mnemonics small-primes-product
0.8960000	sub_FFFFFFFF00A329948	4	Mnemonics small-primes-product
0.9006154	sub_FFFFFFFF00A343C60	6	Mnemonics small-primes-product

# it shows which functions gained or lost basic blocks ("survived" to the recompilation).

```
$ C:\sqlite\sqlite3.exe -markdown $DB "SELECT name AS old, nodes1 AS was, nodes2 AS now_, ratio,
description FROM results WHERE CAST(nodes1 AS INTEGER) <> CAST(nodes2 AS INTEGER) ORDER BY
ABS(CAST(nodes2 AS INTEGER) - CAST(nodes1 AS INTEGER)) DESC LIMIT 10;"
```

old	was	now_	ratio	description
sub_FFFFFFFF00A348AB0	14	16	0.9750253	Same constants

# given an address, where it is in the other build.

```
$ C:\sqlite\sqlite3.exe -line $DB "WITH h(s,v) AS
(VALUE('0',0),('1',1),('2',2),('3',3),('4',4),('5',5),('6',6),('7',7),('8',8),('9',9),('a',10),('b',11),('c',12),('d',13),('e',14),('f',15)), n(k) AS (VALUES(1),(2),(3),(4),(5),(6),(7)), i AS
(SELECT u.type, (SELECT SUM(v*CAST(power(16,7-k) AS INTEGER)) FROM n JOIN h ON
h.s=substr(u.address,9+k,1)) AS ea FROM unmatched u), r AS (SELECT (SELECT SUM(v*CAST(power(16,7-k)
AS INTEGER)) FROM n JOIN h ON h.s=substr(x.address2,9+k,1)) - (SELECT SUM(v*CAST(power(16,7-k) AS
INTEGER)) FROM n JOIN h ON h.s=substr(x.address,9+k,1)) AS delta FROM results x), top AS (SELECT
delta FROM r GROUP BY delta ORDER BY COUNT(*) DESC LIMIT 3), orphan AS (SELECT type, ea FROM i o
WHERE (o.type='secondary' AND NOT EXISTS (SELECT 1 FROM i j, top WHERE j.type='primary' AND j.ea =
o.ea + top.delta)) OR (o.type='primary' AND NOT EXISTS (SELECT 1 FROM i j, top WHERE
j.type='secondary' AND j.ea = o.ea - top.delta))) SELECT (SELECT COUNT(*) FROM i WHERE
type='secondary') AS unmatched_old, (SELECT COUNT(*) FROM i WHERE type='primary') AS unmatched_new,
(SELECT COUNT(*) FROM orphan WHERE type='secondary') AS really_removed, (SELECT COUNT(*) FROM
orphan WHERE type='primary') AS really_added, (SELECT group_concat(sprintf('0x%X', delta), ' ') FROM
top) AS deltas_used;"
```

unmatched\_old: 213

unmatched\_new: 213

really\_removed: 0

really\_added: 0

deltas\_used: 0x137B80 0x137B98 0x137BE0

# Given an address, what is it in the other build (either direction)

# I've picked the address from three queries above.

```
$ C:\sqlite\sqlite3.exe -line $DB "SELECT name AS old, '0x' || address AS old_ea, name2 AS new,
'0x' || address2 AS new_ea, ratio, description FROM results WHERE address =
lower('FFFFFFF00A348AB0') OR address2 = lower('FFFFFFF00A348AB0');"

```

```
old: sub_FFFFFFFF00A348AB0
old_ea: 0xffffffff00a348ab0
new: sub_FFFFFFFF00A480630
new_ea: 0xffffffff00a480630
ratio: 0.9750253
description: Same constants
```

# How much to trust to each match

```
$ C:\sqlite\sqlite3.exe -markdown $DB "SELECT description, COUNT(*) AS matches, MIN(CAST(ratio AS
REAL)) AS worst, MAX(CAST(ratio AS REAL)) AS best FROM results GROUP BY description ORDER BY
matches DESC;"
```

description	matches	worst	best
Same cleaned assembly	378	1.0	1.0
Same low complexity and names	340	0.8385294	1.0

Perfect match, same name	275	0.9193529	1.0
Equal assembly	58	1.0	1.0
Mnemonics small-primes-product	36	0.7242857	1.0
Bytes hash	16	1.0	1.0
Same nodes, edges, loops and strongl	5	0.99	0.9922857
Same constants	1	0.9750253	0.9750253
Loop count	1	0.99	0.99

## 12. References

This first article used a series of excellent references, and I tried to build it organized by category, while other important references have already been mentioned throughout the text:

### Apple primary sources

- Apple Platform Security guide: <https://support.apple.com/guide/security/welcome/web>
- Apple Platform Security | Operating System: <https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web>
- Apple Platform Security | Apple Soc Security: <https://support.apple.com/guide/security/apple-soc-security-sec87716a080/web>
- Apple Platform Security | The Secure Enclave: <https://support.apple.com/guide/security/the-secure-enclave-sec59b0b31ff/web>
- Apple Platform Security | Boot process for iPhone and iPad devices: <https://support.apple.com/guide/security/boot-process-for-ipad-and-iphone-devices-secb3000f149/web>
- Apple Platform Security | Trust caches: <https://support.apple.com/guide/security/trust-caches-sec7d38fbf97/web>
- Apple Platform Security | App code signing process: <https://support.apple.com/guide/security/app-code-signing-process-sec7c917bf14/web>
- Apple Platform Security | Security of runtime process in iOS, iPadOS and visionOS. <https://support.apple.com/guide/security/security-of-runtime-process-sec15bfe098e/web>
- Apple Platform Security | Lockdown Mode security: <https://support.apple.com/guide/security/lockdown-mode-security-sec2437264f0/web>
- Apple Platform Security | Encryption and Data Protection overview: <https://support.apple.com/guide/security/encryption-and-data-protection-overview-sece3bee0835/web>
- Apple Platform Security | System Integrity Protection: <https://support.apple.com/guide/security/system-integrity-protection-secb7ea06b49/web>
- Apple Platform Security | Startup Disk security policy control for a Mac with Apple silicon: <https://support.apple.com/guide/security/startup-disk-security-policy-control-sec7d92dc49f/web>
- Apple Developer Documentation: entitlements: <https://developer.apple.com/documentation/bundleresources/entitlements>
- Apple Developer Documentation | code-signing services: [https://developer.apple.com/documentation/security/code\\_signing\\_services](https://developer.apple.com/documentation/security/code_signing_services)
- Apple Developer Documentation | iOS SDK release notes: <https://developer.apple.com/documentation/ios-ipados-release-notes>

- Apple's open source XNU mirror: <https://github.com/apple-oss-distributions/xnu>
- Apple's clang | LLVM PointerAuthentication documentation: <https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst>
- Apple Security Research Device Program: <https://security.apple.com/research-device/>
- Apple Security Bounty: <https://security.apple.com/bounty/>
- Apple's open source distributions page: <https://github.com/apple-oss-distributions/distribution-macOS>

## ARM architecture references

- Arm Architecture Reference Manual for A-profile architecture: <https://developer.arm.com/documentation/ddi0487/latest>
- Arm | Memory Tagging Extension: Enhancing memory safety through architecture: <https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety>
- Armv8.1-M Pointer Authentication and Branch Target Identification Extension: <https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension>

## Tooling repositories

- blacktop/ipsw: <https://github.com/blacktop/ipsw> and documentation site on <https://blacktop.github.io/ipsw/>
- blacktop/symbolicator: <https://github.com/blacktop/symbolicator>
- Lakr233/vphone-cli: <https://github.com/Lakr233/vphone-cli>
- libimobiledevice: <https://libimobiledevice.org/> and GitHub on <https://github.com/libimobiledevice/libimobiledevice>
- Idid-procursus: <https://github.com/ProcursusTeam/ldid>
- IDA Pro and the Hex-Rays decompiler: <https://hex-rays.com/ida-pro>
- Cellebrite ida\_kernelcache plugin: [https://github.com/cellebrite-labs/ida\\_kernelcache](https://github.com/cellebrite-labs/ida_kernelcache)
- Cellebrite ida\_kcpg: [https://github.com/cellebrite-labs/ida\\_kcpg](https://github.com/cellebrite-labs/ida_kcpg)
- BinDiff: <https://github.com/google/bindiff>
- idasql: <https://github.com/allthingsida/idasql>

## Public research material referenced

- iOS Internals (by Jonathan Levin): <http://newosxbook.com/>
- Operation Triangulation: iOS devices targeted with previously unknown malware (by Igor Kuznetsov, Valentin Pashkov, Leonid Bezvershenko and Georgy Kucherin): <https://securelist.com/operation-triangulation/109842/>
- Operation Triangulation: The last (hardware) mystery (by Boris Larin, Leonid Bezvershenko, and Georgy Kucherin): <https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/>
- NVD entries for the Operation Triangulation CVEs.
  - CVE-2023-41990 on <https://nvd.nist.gov/vuln/detail/cve-2023-41990>
  - CVE-2023-32435 on <https://nvd.nist.gov/vuln/detail/cve-2023-32435>
  - CVE-2023-32434 on <https://nvd.nist.gov/vuln/detail/cve-2023-32434>

- CVE-2023-38606 on <https://nvd.nist.gov/vuln/detail/cve-2023-38606>
- Apple security release notes for the Triangulation patches:
  - iOS 16.3 (HT213605) on <https://support.apple.com/HT213605>
  - iOS 16.5.1 (HT213813) on <https://support.apple.com/HT213813>
  - iOS 16.6 (HT213842) at <https://support.apple.com/HT213842>
- Brandon Azad's blanket exploit (CVE-2018-4280): <https://github.com/bazad/blanket>
- PACMAN | Attacking ARM Pointer Authentication with Speculative Execution (Ravichandran et al., ISCA 2022): <https://pacmanattack.com/>
- Apple Sandbox Profile Language reference (Dionysus Blazakis): <https://www.ise.io/wp-content/uploads/2017/07/apple-sandbox.pdf>
- Steffin, M. and Classen, J. Modern iOS Security Features | A Deep Dive into SPTM, TXM, and Exclaves (arXiv, October 2025): <https://arxiv.org/abs/2510.09272>
- Siguza | KTRR: <https://blog.siguza.net/KTRR/>
- Azad, B | One Byte to rule them all (Project Zero, 30 July 2020): <https://projectzero.google/2020/07/one-byte-to-rule-them-all.html>
- felix-pb/kfd: <https://github.com/felix-pb/kfd>
- Azad, B. | KTRW the journey to build a debuggable iPhone (Project Zero, 28 October 2019): <https://projectzero.google/2019/10/ktrw-journey-to-build-debuggable-iphone.html>
- Alfie CG | Trigon: developing a deterministic kernel exploit for iOS (part 1): <https://alfiecg.uk/2025/03/01/Trigon.html>
- Alfie CG | Titan: Titan is a PPL & SPTM bypass for iOS 16.1 - 17.4 beta 3, supporting A14 - A17: <https://github.com/alfiecg24/Titan>

## 13. Conclusion

This first article of iOS Security Research was extraordinarily long, and I hope you have learned and reviewed all fundamental concepts on iOS. Indeed, providing readers with all necessary foundations in the first article has shown tricky and certainly there are multiple details that I will have to cover in the next articles of this iOS series.

Although my original plan was writing a limited articles showing only the setup steps, the truth is that I felt that I should include a series of concepts and practical approaches, and then I did it and tried to establish a balance between concepts, which are always important, and walkthroughs showing how to do certain tasks. As expected, a few of explanations were concise and not even completely precise, but I think that the idea is much more relevant and useful that to be very detailed because readers trend to forget them.

Just in case you want to stay connected:

- **X:** @ale\_sp\_brazil
- **Bluesky:** @alexandreborges.bsky.social
- **Research articles:** <https://www.blackstormsecurity.com/research/>
- **Blog:** <https://exploitreversing.com/>

Keep developing exploits and see you at the next article!

**Alexandre Borges**