Eight Degrees of Exploitability

Field Guide to Data-Only Windows Kernel Exploitation



Seoul | Korea (2026)

Who am I



Alexandre Borges

@ale_sp_brazil

Vulnerability Researcher
& Exploit Developer

Blackstorm Security

- COMPANY
  Blackstorm Security — blackstormsecurity.com
- WRITING
  blackstormsecurity.com/research + exploitreversing.com
- RESEARCH
  iOS and Windows
- BOOK
  Gray Hat Hacking, 7th Edition — co-author (upcoming)
- EDITORIAL
  Forensic Science Intl: Digital Investigation — board member

Agenda

1	**Introduction and framing**	the thesis, the scope, the spine	
2	**The bug and the PoC wall**	CVE-2024-30085 — one 16-byte write	
3	**Overflow → SYSTEM**	exploits 01 & 02	ERS 06
4	**Escaping the trap**	exploits 03 & 04	ERS 07
5	**The "pure" exploit**	exploits 05 & 06	ERS 08
6	**Impact**	PPL → LSASS, exploits 07 & 08	ERS 09
7	**Synthesis and the cleanup tax**	what survives, for defenders	

Based on ERS 06-09, published on blackstormsecurity.com/research and exploitreversing.com

❑ Section 01 | Introduction and Framing

Framing

"This is the story of one **16-byte bug**
and the **eight exploits** I built on top of it."

NOT the exploit code → **its ENGINEERING**

This talk is not about the exploit code itself, but rather its engineering.

Scope

BUG

CVE-2024-30085 — cldflt.sys

N-day, patched Jun 2024

STUDY

exploitation only

nothing about 0-day

PRIMITIVE

one bug → one primitive

16-byte write past a paged-pool buffer

APPROACH

100% data-only, by construction

HVCI / SMEP / SMAP / kCFG sidestepped

DEMONSTRATED ON

Windows 11 23H2 (22631.2428)

8 exploits, each kills the last's limit

REACH

7 of 8: user → SYSTEM

exploit 2: Admin → SYSTEM

Windows 11 24H2+ restricts the medium-IL handle leak — but it does NOT break all eight.

Reaching SYSTEM is the easy half — surviving my own corruption is the hard half.

You should know...

TECHNIQUES

PreviousMode, I/O Ring, etc.

patched over time

THE BUG

2024 vulnerability

Windows 11 23H2

THE FIELD MOVES

new techniques used today

refined over weeks & months

THE MYTH

an ‘arsenal of techniques’

→

WHAT ACTUALLY MATTERS

engineering + creativity

The focus is not the techniques — it is the engineering to overcome the challenges.

Exploit Map

One bug → eight exploits — each one kills the limitation of the last.

● deep dive ○ transition

[1] ALPC one-shot write	any user → SYSTEM	ERS 06
[2] Parent spoofing	Admin only	ERS 06
[3] PreviousMode flip	unlimited write; clean exit	ERS 07
[4] I/O Ring write ed.01	8B precise, reusable	ERS 07
[5] I/O Ring +read ed.02	24 → 15 stages	ERS 08
[6] "Pure" I/O Ring ed.03	13 stages, zero ALPC	ERS 08
[7] PreviousMode (general)	universal Nt-API	ERS 09
[8] PPL bypass "E5"	→ LSASS cred dump	ERS 09

The same 16-byte write, eight times — each edition erases the last one's limitation.

Why data-only wins

Every mitigation watches for CODE or a hijacked CALL — we do neither.

MITIGATION	WHAT IT CATCHES	WHY WE SLIP PAST
DEP / W^X / HVCI	an executable attacker page	→ no page made exec; we rewrite DATA
SMEP / SMAP	kernel run/read of a user page	→ never touch a user page from kernel
CFG / kCFG / XFG	a guarded INDIRECT call	→ no new call; flip fields + a fnptr
CET / shadow stack	a corrupted RETURN address	→ no return address is touched
PatchGuard	tampered kernel code / globals	→ edit per-process token + thread data

We don't DEFEAT these — we give them nothing to catch. (ERS 06, Sec.15)

Stages of exploit development

From a single vulnerable line to eight reliable, self-cleaning exploits.

Reach the bug

1. Find the exact vulnerable line
2. Understand the data restrictions to reach it
3. Write a PoC that reaches the vulnerable line
4. Cause a crash

crash = you reached the line

Build exploits

5. Promote the bug to one or more primitives
6. Write the first exploit
7. Other exploits → different mechanisms & primitives

one bug → many weapons

Make it reliable

8. Work on stability
9. Work on repeatability
10. Perfect cleanup → no crash at the exit

reliable + silent exit

A bug is not an exploit — reliability and a clean exit are the real work.

❑ Section 02 | The bug and the PoC wall

The whole bug — one missing compare

DRIVER	ROUTINE	HEAP DEST
cldflt.sys	**HsmIBitmapNORMALOpen**	**PagedPool 0x1000 'mBsH'**

● VULNERABLE

no guard

```
Element_Length = ElementInfos[4].Length;
// ElementInfos[4] is attacker-controlled

memmove(dst, Src, Element_Length);
```

no bound check → 16B past the 0x1000 buffer

● PATCHED (Jun 2024)

+1 compare

```
Element_Length = ElementInfos[4].Length;

if ( Element_Length > 0x1000 )
    return status;

memmove(dst, Src, Element_Length);
```

The entire fix is ONE added compare — CWE-122 heap buffer overflow.

Exactly where the bug is

The length is checked on one branch — and skipped on the other.

● if: length is bounded

```
if ( Src && Element_Length - 1 <= 0xFFE ) {
    ...
    dst = ExAllocatePoolWithTag(PagedPool, 0x1000, 'mBsH');
    memmove(dst, Src, Element_Length);
    if ( Element_Length < 0xFFC ) ...
}
```

length in [1 .. 0xFFF] → copy fits the 0x1000 buffer

● else: length is NOT bounded

```
else {
    dst = ExAllocatePoolWithTag(PagedPool, 0x1000, 'mBsH');

    memmove(dst, Src, Element_Length);
}
```

no length check here — oversized length routes in

Element_Length = 0x1010 → 16B past the 0x1000 buffer

The guard sits only on the if-branch — oversize the length and the else copies it unchecked.

One unforgiving primitive

memmove writes 0x1010 bytes into a 0x1000 buffer — 16 bytes spill into the neighbour.

dest buffer — 0x1000

ExAllocatePool(PagedPool, 0x1000, 'mBsH')

0x10
16B

adjacent pool object

first bytes overwritten

The primitive you get

16-byte minimum THESIS

not 8 → the one fact that drives the whole talk

Fixed position

overwrites the neighbour's FIRST bytes

Write only

no read, no info leak yet

16-byte, fixed-position, write-only — every one of the eight exploits is built from THIS.

The reparse "onion"

Attacker data must pass every layer before a single OOB byte lands.

REPARSE_DATA_BUFFER (HSM / Cloud reparse point) — checks: valid reparse tag

FeRp (OUTER) magic 0x70526546 — checks: magic + CRC32 + compression

_HSM_REPARSE_DATA / _HSM_DATA headers — 0x10 each

CRC32 over [0x0C .. end] | compressed flag set

MaxElements = 0x0A → 10 x _HSM_ELEMENT_INFO (8B)

Element[4] → carries the nested BtRp

BtRp (NESTED) magic 0x70527442 — checks: magic + 5 elements

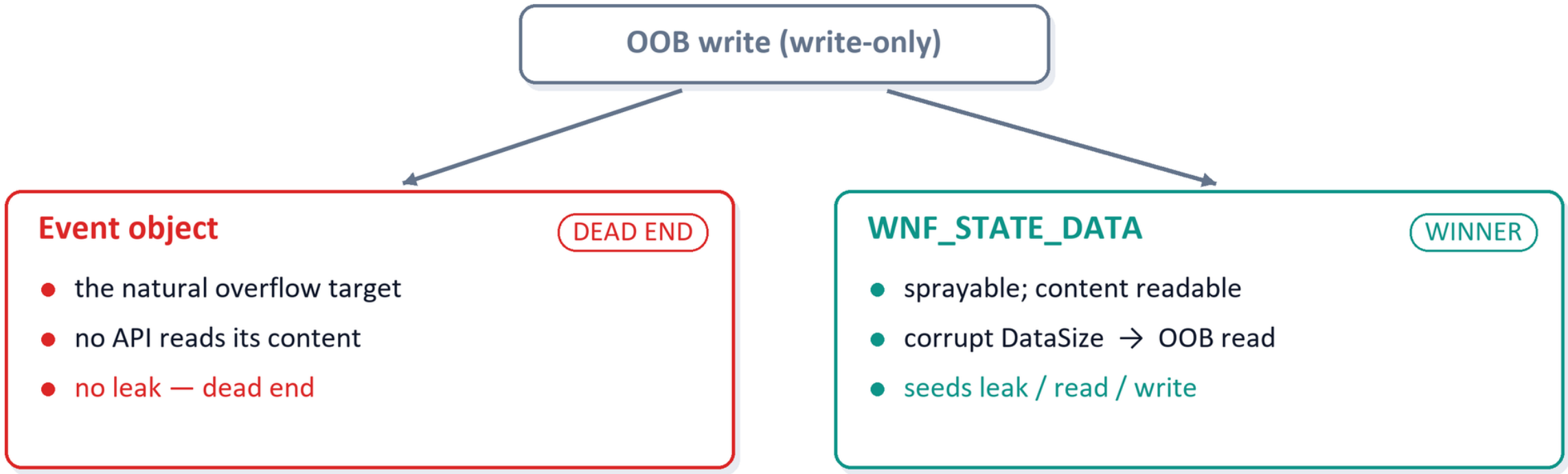
Element[4].Length = 0x1010 → sets the overflow size

attacker bytes → memmove copies 0x1010 → 16B OOB

Magic + CRC32 + compression + structure — THEN one overflow byte lands.

Write, but no read

I have a WRITE, no READ, no KASLR — which overflow target yields a leak?



One choice — WNF_STATE_DATA — seeds the leak / read / write toolkit for all eight exploits.

Choosing the readable object

Which sprayable neighbour can I READ back, to turn the write into a leak?

kernel objects

NO LEAK

- Event / Semaphore / Mutant
- I/O Completion port
- sprayable, but no content read
- Query returns STATE only
- cannot leak a pointer

WNF_STATE_DATA

WINNER

- heap object (not a kernel object)
- PagedPool — like the 'mBsH' buffer
- grooms adjacent to the overflow
- readable: NtQueryWnfStateData
- freely sprayable (0x1000)
- the readable window

ALPC handle table + pipe attributes are not rejected — they return later: the leak source and the arbitrary read.

WNF_STATE_DATA: a readable, sprayable PagedPool object — the window that turns the write into a leak.

❑ Section 03 | Overflow → System | Exploits 1-2

Exploit 1: four primitives, one chain

One 16-byte write, chained through four objects — never shellcode.

1. LEAK
- WNF spray
- DataSize → 0xFF8
- OOB read

→

2. KASLR
- _KALPC_RESERVE
- groomed next door
- leak kernel ptr

→

3. READ
- corrupt pipe Flink
- any kernel addr
- arbitrary read

→

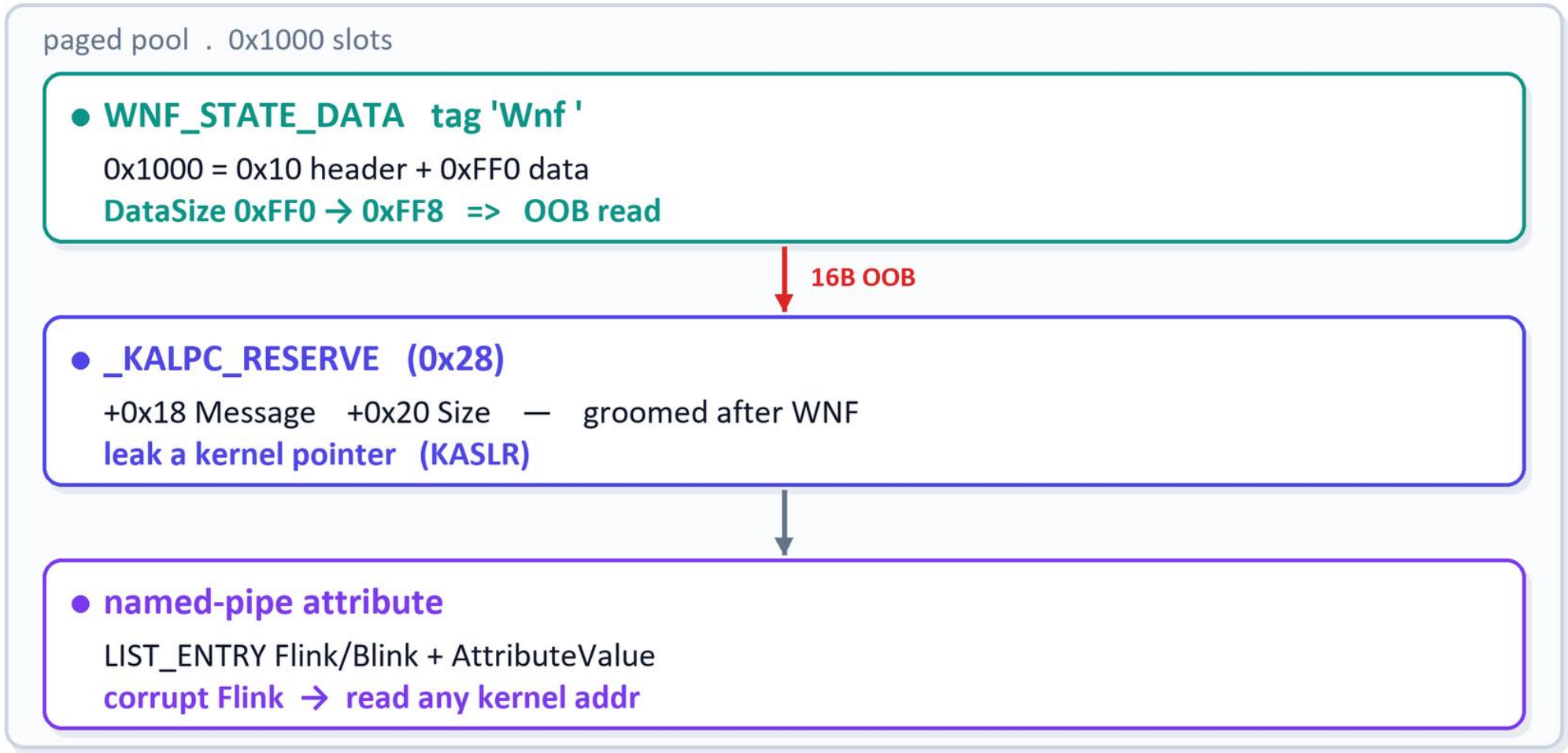
4. WRITE
- one-shot ALPC
- Token+0x40 privs
- grant all privs

any user → SYSTEM (~23 stages + a repair stage)

Exploit 1's KASLR is heap adjacency (no API leak) — it survives the 24H2 handle-leak removal.

OOB write → leak → read

Three neighbours groomed in paged pool (0x1000 slots) — one field each turns write into read.



One field each: a length (WNF 0xFF0 → 0xFF8) or a pointer (KALPC / pipe Flink).

OOB write -> the ALPC write

- **fake _KALPC_RESERVE (user memory)**
  - set only Size + Message
  - Message -> the fake message

.Message →

- **fake _KALPC_MESSAGE**
  - ExtensionBuffer -> Token+0x40
  - ExtensionBufferSize = 0x10 (16 bytes)

Handles[0] ↑

- **_ALPC_PORT Handles[0]**
  - entry normally -> the REAL reserve
  - reuse the WNF OOB write: swing it -> fake

NtAlpcSendWaitReceivePort -> kernel delivers through the fake reserve, copies 16 bytes

↓

- **=> write lands at _TOKEN.Privileges (Token+0x40)**
  - Present + Enabled = all-ones => every privilege granted

The one-shot write — and why it bugchecks

16 bytes grant every privilege — but the same write leaves a corruption it must undo.

- **fake _KALPC_MESSAGE**
  - ExtensionBuffer → target
  - ExtensionBufferSize = 0x10 (16B)

16B →

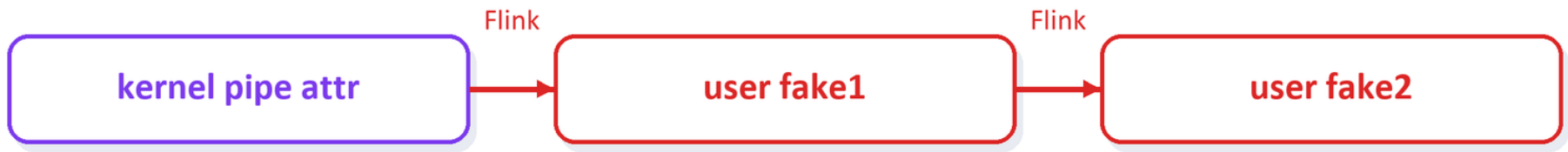
- **_TOKEN.Privileges (Token+0x40)**
  - Present + Enabled → all-1s
  - every privilege (8B silently no-op)

- **The price: a corruption it cannot repair**
  - the same overflow smashed a pipe Flink (a doubly-linked list)
  - on process exit the kernel walks it → into freed user memory
  - BUGCHECK
  - a dedicated REPAIR stage had to be invented (exploits 3+ escape it)

The grant works — but the same write bugchecks the box on the way out.

Why it bugchecks — and the repair

The same overflow that granted the token also smashed a pipe attribute Flink.



- **On process exit → BUGCHECK**
  - NpRemoveAllAttributesFromList walks the list
  - it follows the Flink into freed user memory
  - BUGCHECK
  - you reached SYSTEM; the box dies on exit

- **The repair (Stage 22b)**
  - the ALPC one-shot was spent on the token
  - reuse the WNF overflow — the only write left
  - rewrite the ORIGINAL kernel Flink over the fake
  - corrupt again to repair → clean exit

Cleanup ordering: do not VirtualFree the fake KALPC until the ALPC ports close.
Corrupted WNF entries stay leaked — touching their pool metadata crashes.

Exploits 3+ escape this trap — a reusable write can repair cleanly.

❖ DEMO 01: ANY USER TO SYSTEM (EXPLOIT 1)

❑ Section 04 | Escaping the trap | Exploits 3-4

Exploit 3: spend the write on PreviousMode

One byte flips the kernel's trust — and this write is reusable, so it can clean up after itself.

- **_KTHREAD.PreviousMode +0x232**
  - one ALPC write
  - 0x01 → 0x00
  - UserMode → KernelMode

→

- **every Nt* handler trusts you**
  - pointer probes skipped
  - kernel addresses accepted
  - NtWriteVirtualMemory =
  - unlimited kernel write

→

- **reusable => you can repair**
  - callable as often as you like
  - un-corrupt token / Flink /
  - PreviousMode itself
  - clean exit, no bugcheck

New cost: locate your own _KTHREAD via the handle table (the 24H2 dependency).

One byte turns a benign syscall into arbitrary, repeatable kernel R/W — and exits clean.

Precision: read-modify-write around the flip

The write is 16 bytes minimum; PreviousMode is 1 byte — the other 15 are live scheduler state.

- **16-byte write, 1-byte target**
  - +0x232 PreviousMode (CHAR) <- target
  - +0x233 RaiseIrql (UCHAR)
  - +0x234 SpecialApcDisable (SHORT)
  - the 15 bytes after it are
  - live scheduler state

- **read-modify-write**
  - read 16 bytes at +0x232
  - change ONLY byte[0] → 0x00
  - write all 16 back
  - memcpy(pPayload, pre_read, 16);
  - pPayload[0] = 0x00;

Corrupt the adjacent scheduler bytes and the thread crashes later.

Read 16, change one byte, write 16 back — exactly the detail write-ups skip.

Inside the window: only raw syscalls are safe

While PreviousMode = 0, your process is not a normal user process.

- **SAFE: direct ntdll syscalls**
  - NtOpenProcess
  - NtWriteVirtualMemory
  - NtDuplicateToken
  - NtSetInformationThread

- **UNSAFE: Win32 APIs**
  - many layers sit between them and the kernel
  - the kernel no longer validates for you
  - unvalidated buffers → corruption / crash

Keep the window SHORT: do the writes, then restore PreviousMode 0x00 → 0x01.

Turning off the kernel's validation turns it off for YOUR bugs too.

Why impersonating SYSTEM is not enough

You can impersonate inside the window — but only the token write makes it permanent.

Inside the window you CAN impersonate SYSTEM

NtOpenProcess(PID 4) → NtOpenProcessTokenEx → NtDuplicateToken → NtSetInformationThread

1 SecurityIdentification

duplicated token = SecID level
after the window closes,
fails with 1346 (BAD_IMPERSONATION)

2 CreateProcessW

uses the process PRIMARY token
WithTokenW needs SeImpersonate
only after the token write (circular)

3 Per-thread + temporary

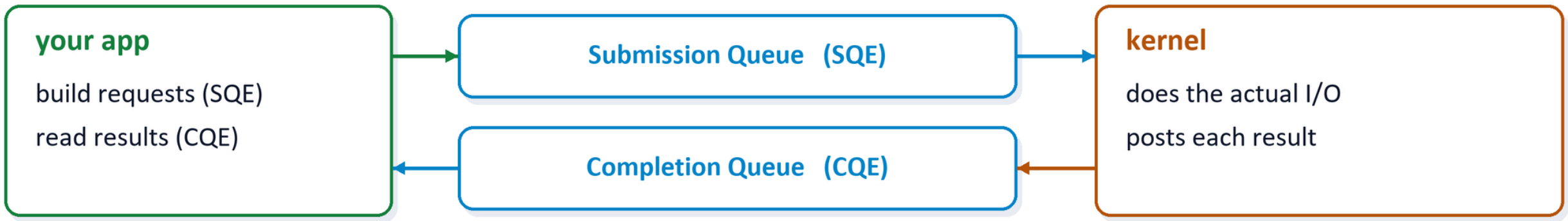
the _EPROCESS.Token write makes
the WHOLE process SYSTEM,
permanently

So do both: impersonate to show the reach, then write the token — the only thing that survives.

Impersonation is temporary — only the _EPROCESS.Token write makes the whole process SYSTEM for good.

What an I/O Ring is

Windows 11 (2021+), the io_uring analog — submit requests, read completions.



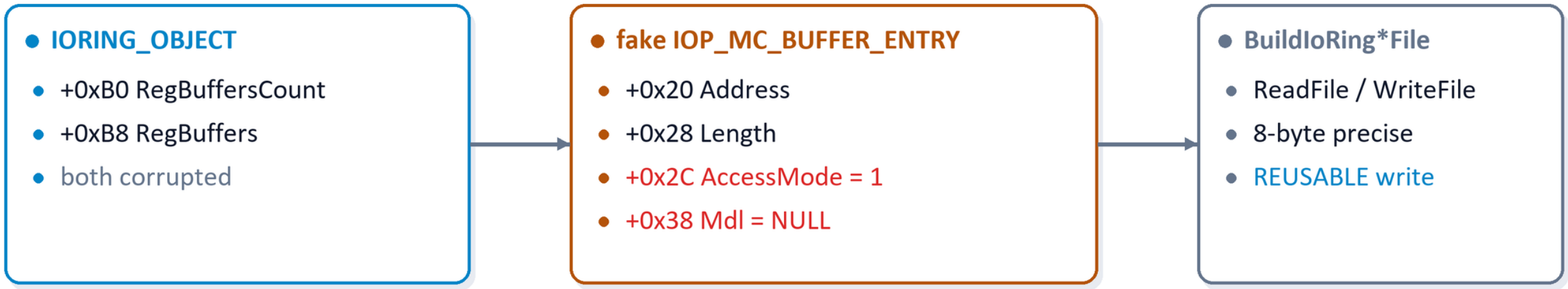
- **registered buffers — referenced by INDEX, not pointer**
  - CreateIoRing → BuildIoRingRegisterBuffers → IORING_OBJECT.RegBuffers[]
  - each slot → IOP_MC_BUFFER_ENTRY { Address, Length, ... }

BuildIoRingRegisterBuffers VALIDATES Address is user-mode — so we never call it.

Corrupt RegBuffers[0] → a fake entry whose Address is KERNEL → index 0 = arbitrary R/W. (exploits 4/5/6)

I/O Ring write (edition 01)

Two corrupted fields turn a ReadFile/WriteFile into an 8-byte-precise, reusable kernel write.



Mdl = NULL, or MmUnlockPages crashes on cleanup. AccessMode = 1 accepts a non-user address.

A corrupted ring + a fake buffer entry — an 8-byte-precise, reusable kernel write.

❖ DEMO 02: PREVIOUS MODE FLIP (Exploit 3)

❑ Section 05 | The pure exploit | Exploits 5-6

Exploit 5: subtraction begins

Edition 02 — keep the write, add the read.

- **ADD: an I/O Ring READ**
  - BuildIoRingWriteFile: kernel → pipe
  - now I READ and WRITE through the ring
  - no new corruption needed

removes →

- **So these disappear**
  - the fragile pipe-attribute read
  - the entire second overflow wave
  - half the exploit, deleted

Still needed: the ALPC bootstrap — one 16-byte write to set RegBuffers.

24 stages → 15 stages — shorter, less fragile. (next: delete ALPC entirely → exploit 6)

"Pure": WNF corrupts RegBuffers directly

Same pool-size class — groom WNF next to RegBuffers[], the overflow hits it. No ALPC middleman.

- **WNF_STATE_DATA (0x1000)**
  - OOB write @0xFF0
  - 16B past the end

16B OOB →

- **I/O Ring RegBuffers[] (0x1000)**
  - RegBuffers[0] <- overwritten
  - no ALPC middleman

- **KASLR without the handle leak**
  - PsInitialSystemProcess (exported) → System EPROCESS (PID 4)

13 stages, zero ALPC — KASLR from PsInitialSystemProcess, not the handle leak.

The cost it accepts: adjacency + a 256-ring scan

Pure and ALPC-free — but it rides on heap adjacency and a short scan to find its mark.



256 I/O Rings sprayed . holes punched (white) . one corrupted (red)

- **find the corrupted ring**
  - read 8 bytes through each ring
  - corrupted one returns the fake-entry signature: Type 0x0C02 , Size 0x80

- **corrupted WNF marker**
  - ChangeStamp magic 0xC0DE ; DataSize 0xFF0 → 0xFF8

Spray 256 rings, punch holes, then scan for the fake signature — elegance has a price.

❑ Section 06 | Impact | Exploits 7-8

Exploit 7: PreviousMode as a capability

Same primitive as exploit 3 — a much bigger ambition. (Stage 22 identical; the difference is Stage 23.)

- **Exploit 3: narrow**
  - flip PreviousMode to 0
  - write ONE token
  - restore, and leave

Stage 22 identical

- **Exploit 7: a general capability**
  - run the full Nt* chain as kernel
  - open / duplicate / impersonate any token
  - write any kernel memory
  - anything the kernel can do

Same primitive, bigger idea — exploit 8 spends this generality to strip LSASS PPL.

PPL bypass "E5" → LSASS

Non-admin, medium-IL — strip LSASS protection with one data write, then dump.

1	**find LSASS**	walk ActiveProcessLinks +0x448 ; match ImageFileName +0x5A8
2	**strip PPL**	_EPROCESS.Protection +0x87A : 0x41 → 0x00
3	**open + dump**	OpenProcess(LSASS) + MiniDumpWriteDump (~63 MB)
4	**creds**	pypykatz → NT hash

0x41 = PsProtectedSignerLsa-Light (Type 1, Signer 4).

One data write: 0x41 → 0x00 at _EPROCESS.Protection +0x87A — LSASS is unprotected.

Ordering — one window, one correct sequence

All of this happens inside ONE PreviousMode = 0 window, in this order.

- **PreviousMode = 0 — window OPEN**
  - 1. steal the token → _EPROCESS.Token (8B, raw EX_FAST_REF)
  - 2. strip LSASS PPL → +0x87A 0x41 → 0x00 (1 byte)
  - 3. repair the pipe Flink → so exit does not crash

- **Why this order**
  - the dump needs SeDebug
  - the token carries it
  - → steal token BEFORE PPL

restore PreviousMode 0x00 → 0x01 window CLOSES

then: OpenProcess(LSASS) → MiniDumpWriteDump (~63 MB) → pypykatz → NT hash

❖ DEMO 03: NON-ADMIN → LSASS CREDENTIALS (Exploit 8)

❑ Section 07 | Synthesis and The Cleanup Tax

The toolbox: each object contributes one field

Five primitives, three write targets — one field each, no code anywhere.

OBJECT	FIELD (one each)	EFFECT
PRIMITIVES		
WNF	DataSize 0xFF0 → 0xFF8	OOB leak / read
ALPC reserve	_KALPC_RESERVE Message	KASLR + one-shot write
pipe attribute	LIST_ENTRY Flink	arbitrary read
I/O Ring	RegBuffers +0xB8	8-byte precise R/W
PreviousMode	_KTHREAD +0x232	unlimited Nt* R/W
WRITE TARGETS		
_TOKEN.Privileges	+0x40	grant all privs
_EPROCESS.Token	+0x4B8	steal SYSTEM token
_EPROCESS.Protection	+0x87A	strip PPL

When you get a write, ask: one-shot or reusable? fixed or precise? what does cleanup cost?

The cleanup tax: honest archaeology

Stabilization, not escalation, is where the work lives.

PORT_MESSAGE under #pragma pack(1)	TotalLength 0x2C not 0x38 → messages rejected	SILENT
missing _HEAP_VS_CHUNK_HEADER prefix	fake objects rejected as bad heap metadata	SILENT
_KALPC_RESERVE mis-sized (bad source)	0x30 → corrected to 0x28	SILENT
fake entry ReferenceCount = 0	kernel skips the buffer	SILENT
VirtualFree'd the fake KALPC too early	instant bugcheck	CRASH
single-byte pipe reads	garbage → bulk-read 0x20+ aligned	SILENT
exploit 8 'PreviousMode NOT restored'	printed on SUCCESS -- a false alarm	FALSE ALARM

Almost all of them fail SILENTLY — no error, no crash. That candor is the contribution.

What still works on current Windows (26H2)

Two mitigations now bite — but the data-only approach that dodges both still lands.

- **KASLR leak removed (24H2)** — SHIPPED
  - medium-IL handle enumeration blocked

- **PreviousMode flip killed** — SHIPPED
  - KiSystemCall64 reset + bugcheck 0x1F9

I/O Ring: NOT mitigated — still a live primitive (e.g. CVE-2025-21333, 2025)

- **BREAKS**
  - 3, 4, 7, 8 — handle leak AND PreviousMode
  - 5 — handle leak only

- **SURVIVES**
  - 1, 2 — no PreviousMode, no leak
  - 6 (pure) — class-11 KASLR + I/O Ring

Exploit 6 — built purely for elegance — is the one that dodges BOTH mitigations.

THANK YOU