Subverting the Windows Kernel

with VBS / HVCI / kCET enabled

Table hijacking to gain arbitrary kernel code execution

EuskalHack Security Congress IX

Juan Sacco <jsacco@exploitpack.com> - https://exploitpack.com

Who am I ?

$ whoami: Exploit Writer, Reverse Engineer, OSCP, OSCE, Advanced Corelan, IDA Pro Advanced training

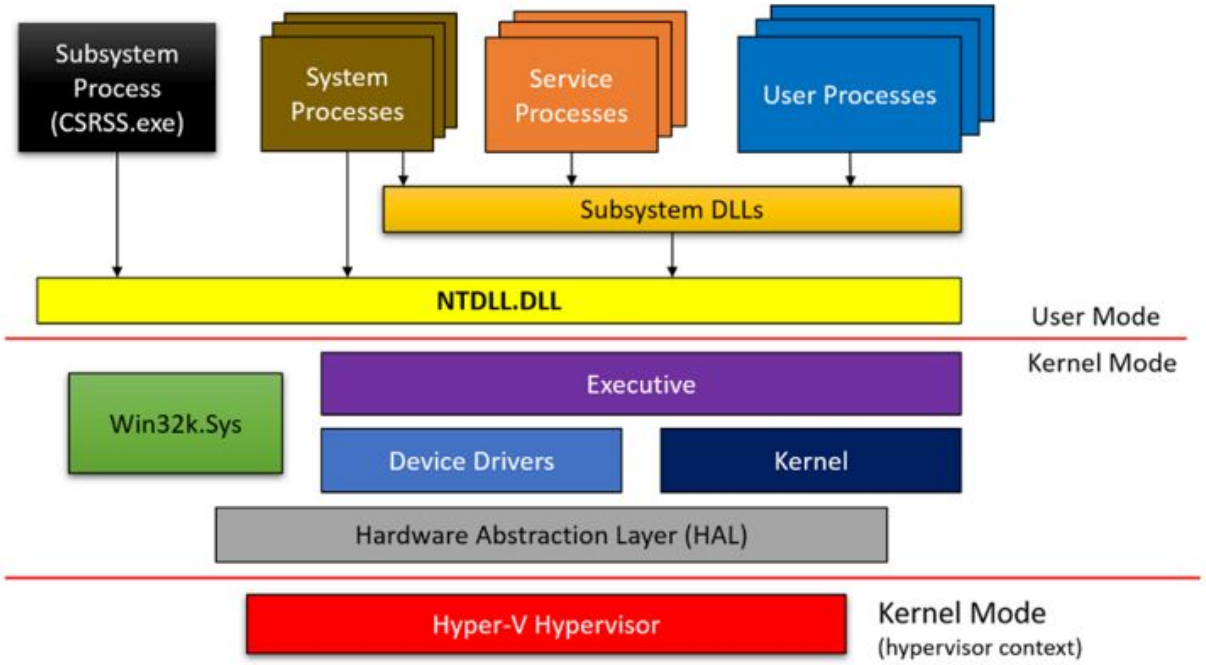
Principal developer and co-founder of https://exploitpack.com

What is the Windows Kernel?

The Windows kernel is the privileged execution environment of the operating system.

It executes in kernel mode and provides the mechanisms required for scheduling, interrupt and exception dispatch, synchronization, memory management, object management, process… On current Windows systems, the core kernel and executive services are implemented primarily in ntoskrnl.exe.

Windows uses a hardware-enforced separation between user mode and kernel mode.

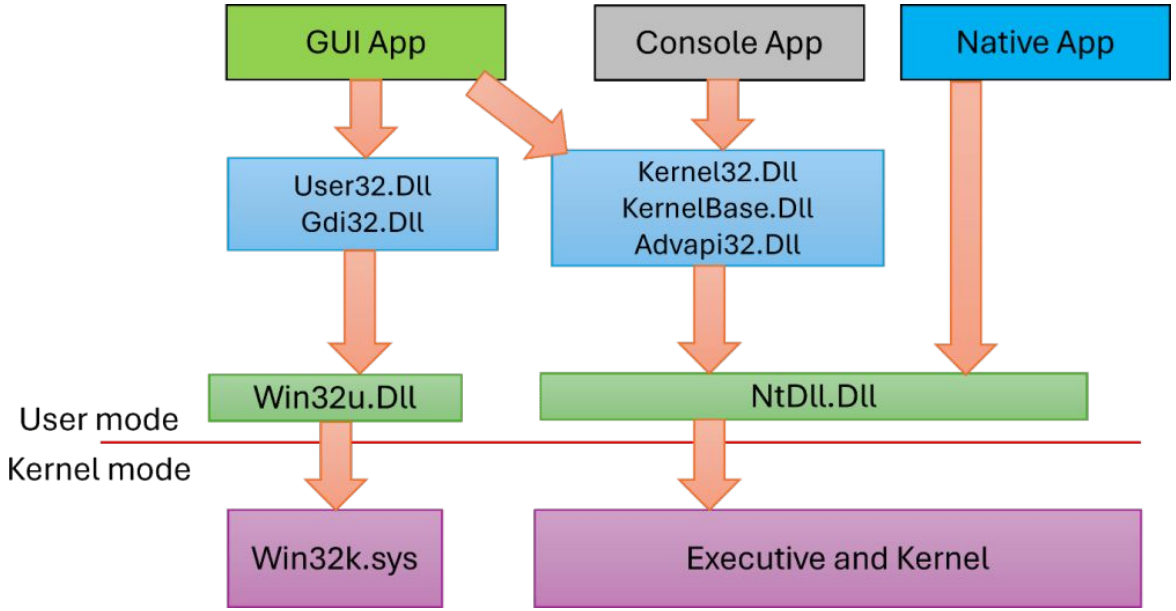


Windows System Architecture

Windows is organized around a user-mode and kernel-mode separation.

User mode contains applications, services, subsystem processes, and the Win32 API layer.

Kernel mode contains the executive, the low-level kernel, the hardware abstraction layer, and kernel-mode drivers. The executive provides high-level operating system services, including the memory manager, object manager, I/O manager, process manager, security reference monitor, registry...



And a Syscall flow looks like this:

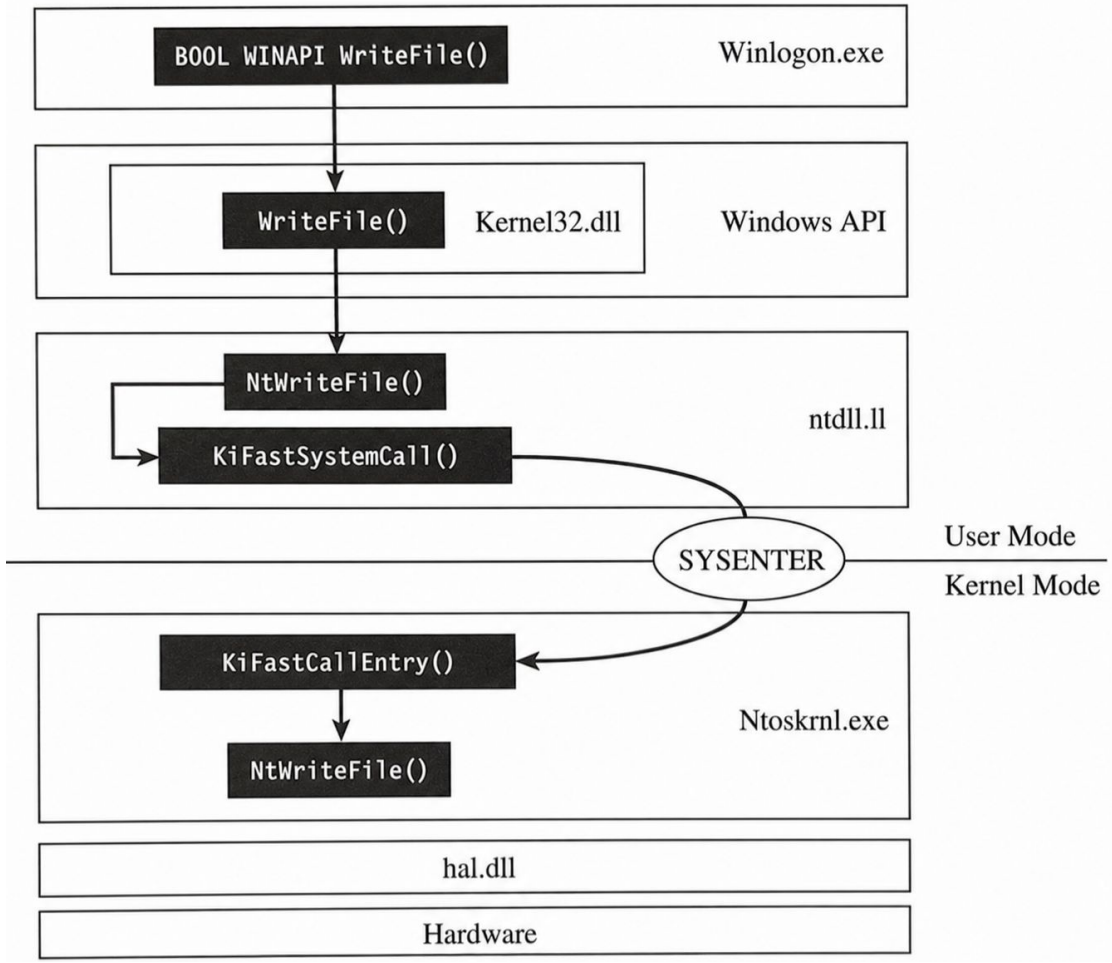
On x64, the user-mode call site follows the standard Windows x64 calling convention before the privilege transition.

The first four integer or pointer arguments are placed in RCX, RDX, R8, and R9; additional arguments are passed on the user stack.

The native API stub then loads the system service number into EAX.

Because the syscall instruction overwrites RCX with the user return address and R11 with the saved user RFLAGS, the stub copies the original first argument from RCX into R10...

A typical x64 stub therefore has the form mov r10, rcx, mov eax, ServiceId, syscall, ret.



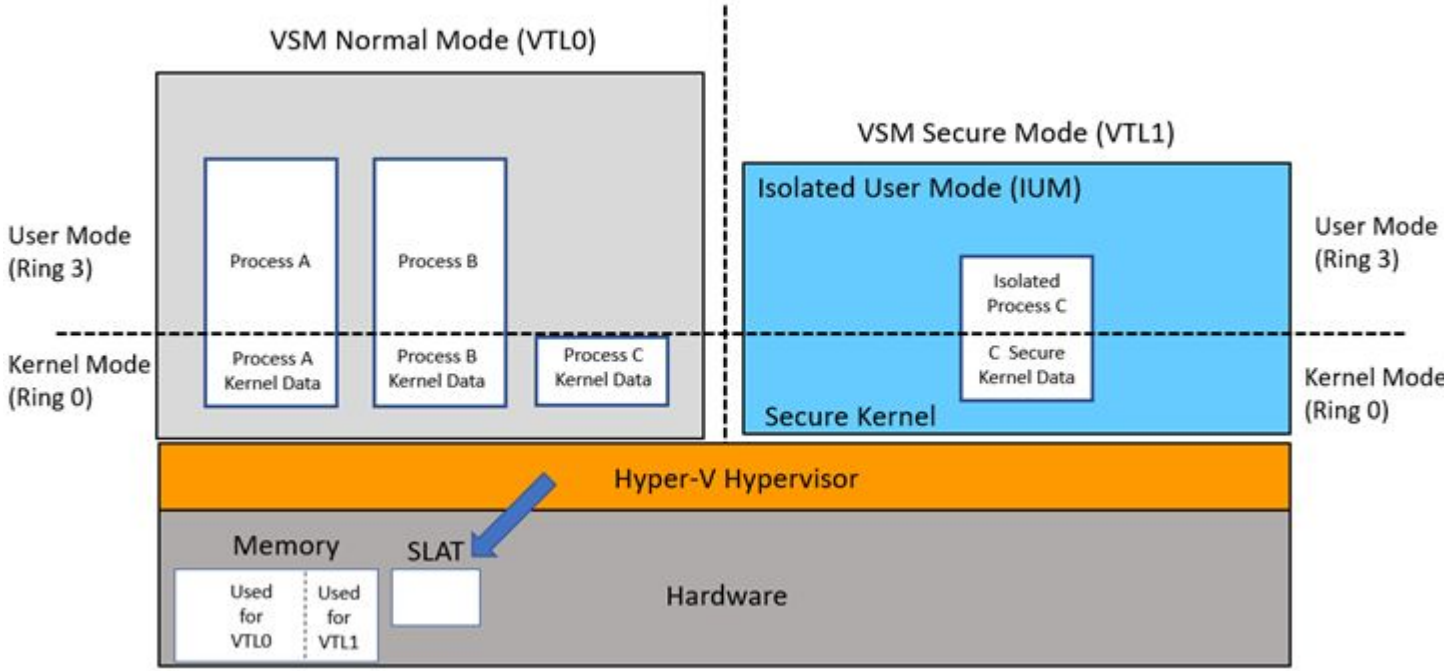
VBS (Virtualization Based-Security) VTL0/VTL1

Virtualization-Based Security, or VBS, is the Windows security architecture that uses the hypervisor to create an isolated execution environment for security-critical operating...

In the VBS model, the ordinary Windows kernel continues to run in VTL0, while a more privileged isolated environment runs in VTL1.

Microsoft describes this isolated environment as the root of trust of the operating system in the event that the normal kernel is compromised.

The secure kernel and isolated user-mode trustlets execute in this higher trust level, and access to VTL1 memory is mediated by the hypervisor rather than by the VTL0 kernel...



Why VBS/HVCI changes the exploitation scene

The hypervisor enforces kernel code integrity and W^X at the second-level translation layer.

The kernel cannot simply map attacker-controlled bytes as executable kernel code.

kCET shadow stacks is meant to stop classic return-address ROP. JOP/COP are only constrained if enforcement such as IBT is active for the relevant kernel code; JOP uses indirect jmps; COP uses indirect calls; neither depends on corrupting a ret address the way classic ROP does.

And PatchGuard?

Patchguard is meant to detect and prevent persistent modifications in Kernel components.

The main issue with HVCI

HVCI blocks new executable bytes. It does not make every kernel data structure immutable, nor does it remove or prevent vulnerable signed drivers.

PART 1

From Physical Memory to Kernel Control

The setup and initial kick-off

A signed-driver physical memory primitive can be subverted into providing kernel execution.

Setup

A MS Windows target with VBS, HVCI and kCET enabled.

Primitive

Read/write physical memory from user mode to Kernel via a vulnerable driver and a VA->PA translator.

Expected result

Trampoline functionality via table hijacking

Primitive exploitation

Different vulnerable drivers, one internal interface, the same Kernel

Vulnerable signed drivers turn vendor kernel code into an attacker-controlled primitive. Even with VBS/HVCI enabled, a driver vulnerability that exposes kernel or physical memory access can mutate trusted kernel state and reuse existing executable code.

First-party drivers are not outside the attack surface. They are high-value kernel gateways: trusted, widely deployed, hard to block, and reachable from user mode.

PART 2

ROP Chain via Suspended Threads

Abusing Suspended Threads to get our ROP executed

The ROP chain journey for a Token-Swap proof of concept

This demo creates a suspended dummy thread, leaks its KTHREAD, and uses the available physical-memory primitive to locate and modify a return slot on that thread's kernel stack.

The original kernel return address is replaced with a small ROP chain that runs when the thread is resumed.

The chain performs a direct privileged write to replace the current process _EPROCESS.Token with a SYSTEM token value, then restores the original return address and repairs RSP so kernel execution can continue normally.

Without kCET, the CPU follows the corrupted kernel stack and the ROP chain executes; with kCET enabled, the architectural return stack no longer matches the protected shadow stack, so the same ROP chain is blocked.

Abusing Suspended Threads to get our ROP executed

The ROP chain journey for a Token-Swap proof of concept

1. Load current process token slot into RDX
2. Load patched SYSTEM token fast-ref into RAX
3. Write RAX into [RDX]
4. Restore the original return address at retAddr
5. Pivot RSP back to retAddr
6. Return normally into original kernel control flow

```
pop rdx ; ret                   ; token slot
pop rax ; ret                   ; patched SYSTEM token
mov [rdx], rax ; ret            ; token write

pop rdx ; ret                   ; original return slot address
pop rax ; ret                   ; original return value
mov [rdx], rax ; ret            ; restore original return slot

pop rsp ; ret                   ; restore RSP to original return slot
ret                             ; returns to originalRet with RSP == retAddr + 8
```

PART 2

Native SSDT Hijack

The native syscall dispatch table

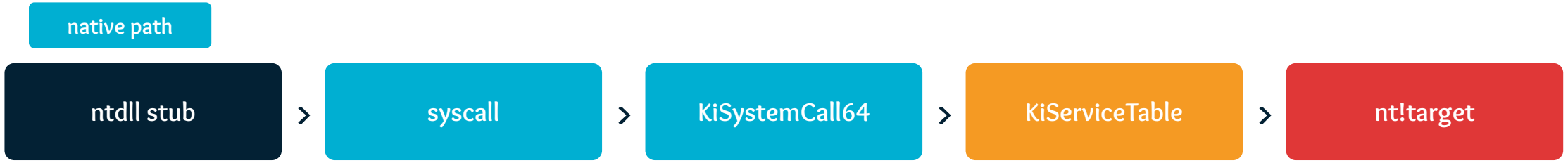
The SSDT is still the spine of the ntoskrnl syscall path.

```
raw = KiServiceTable[index]
offset = raw >> 4
handler = KiServiceTable +
offset
argbytes = raw & 0xF
call handler
```

Entries are compressed 32-bit signed offsets, not function pointers.

A writable table entry becomes a temporary call-routing primitive.

The target must be reachable by the SSDT entry encoding range.



SSDT hijack implementation

The project borrows a syscall slot, writes the encoded target, triggers it, then restores.

Resolve KeServiceDescriptorTable by patterning KiSystemCall64 or symbols.

Use NtSetQuotaInformationFile as the controllable native syscall slot. (Can be changed of course)

Verify direct physical write to KiServiceTable, with PDE permission where needed.

Call DbgPrint/memcpy/Ps* helpers through the selected syscall path.

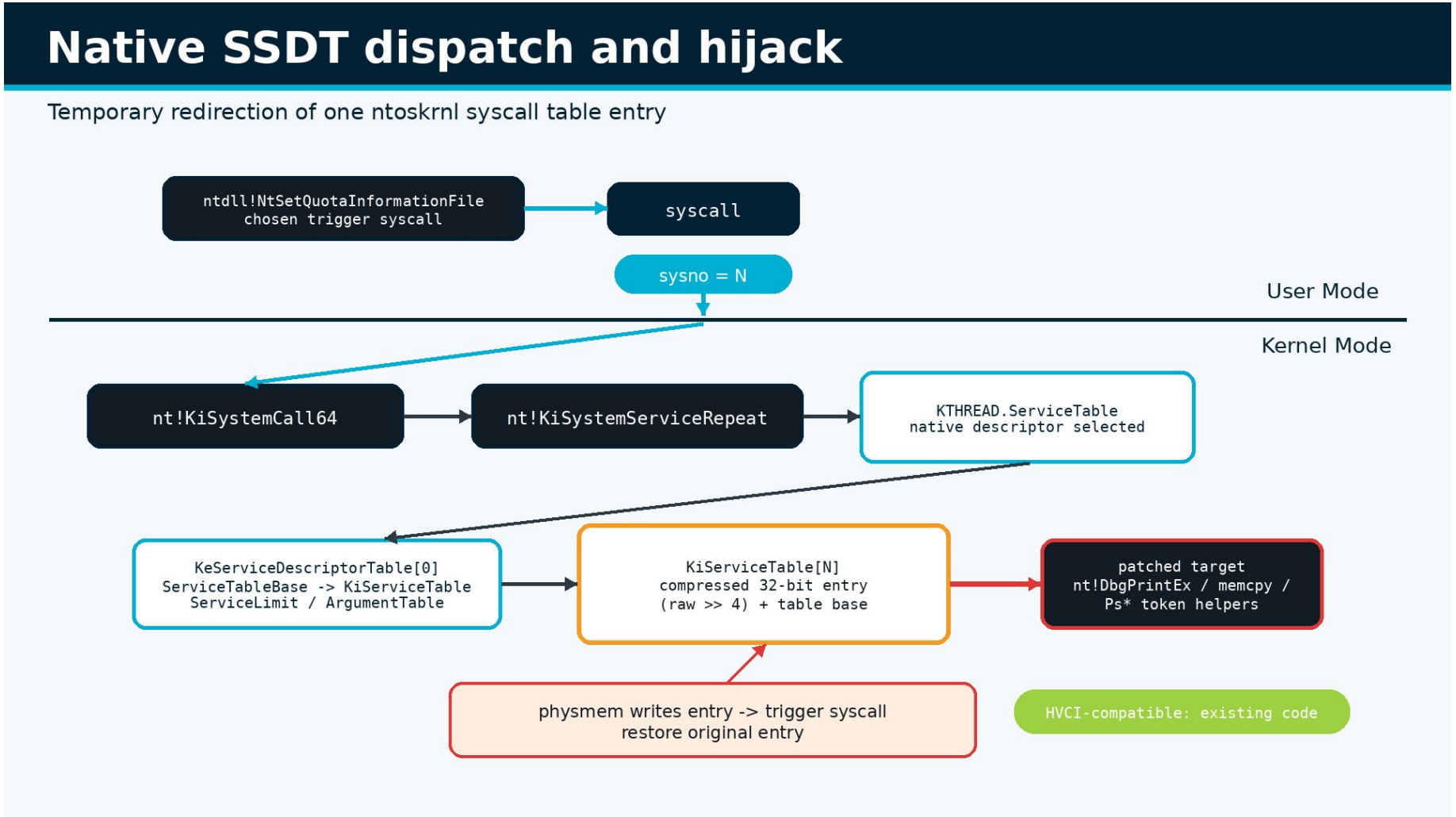
Restore the original entry and validate guard entries.

```
write_ssdt_entry32(slot, target)
InvokeSyscallBlob(...)
write_ssdt_entry32(slot, original)
verify(previous, current, next)
```

Proof of concept payloads

DbgPrint(): visible kernel execution signal
Token swap: privilege outcome without new executable kernel memory

SSDT hijack implementation



Why this works with HVCI on

No new kernel code

The redirect lands on existing signed kernel image bytes already executable under code integrity policy.

No ROP stack

The syscall dispatcher performs a normal call into the selected routine; this avoids the classic return-chain shape kCET is built to break.

Data mutation only

The attacker-controlled operation is a data write to dispatch metadata plus controlled user-mode arguments.

HVCI is doing its job: arbitrary unsigned kernel shellcode is restricted and blocked.

HVCI blocks unsigned kernel code, but it does not stop a vulnerable driver from rewriting trusted kernel data and steering execution into existing kernel routines.

The restore phase matters operationally and defensively: detection has a narrow time window.

PART 3

Shadow SSDT Hijack

The GUI (Win32k Shadow) syscall table

win32u-originated calls route through the Shadow SSDT into win32k.

Shadow syscalls cover NtUser*, NtGdi* and related GUI subsystem services.

The thread must run in an interactive desktop/window-station context.

The table and many useful thunks live in win32k/session space, so translation context is different.

The implementation tests candidate win32u syscalls and keeps only a working slot for the run.

Shadow route used by the demo

Redirect a low-traffic GUI syscall through a live win32kbase dispatch thunk.

1. Select trigger

Resolve win32u!NtUserGetForegroundWindow or another candidate and extract its syscall index.

2. Find thunk

Locate a win32kbase dispatch thunk with a writable RIP-relative function slot.

3. Patch it

Patch thunk slot to target routine and shadow entry to the selected thunk entry.

4. Trigger and restore

Call the win32u stub from a GUI thread, copy return value, restore shadow entry and dispatch slot.

```
ShadowPatchForCall(context, target)
ShadowInvokeVdmTrampoline(args)
ShadowRestorePatch(context)
```

Crossing the argument gap

GUI syscalls were not designed as arbitrary kernel function call adapters.

Direct register arguments cover simple calls.

Wider signatures use a kernel argument block allocated through ExAllocatePool2.

The trampoline keeps attacker-controlled bytes in user mode; kernel execution remains in existing routines.

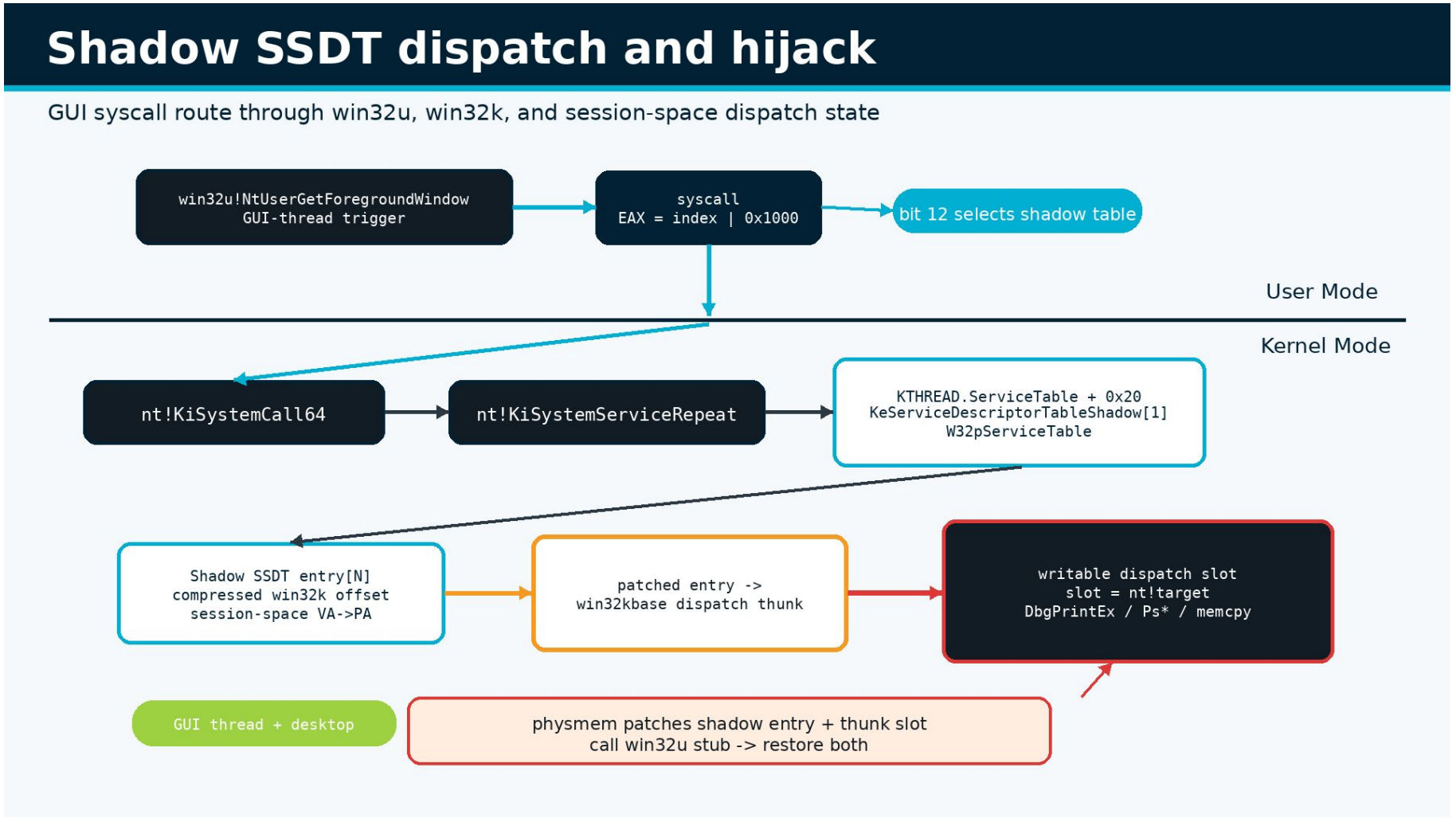
Cleanup handles both dispatch metadata and temporary kernel pool allocations.

```
kMaxKernelCallArgs = 11
arg_block = ExAllocatePool2(...)
memcpy(arg_block, user_args, ...)
call target(arg_block...)
```

Proof of concept payloads

DbgPrint proves controlled kernel call. Token manipulation proves impact while staying compatible with HVCI's executable-memory rules.

Shadow SSDT hijack implementation



PART 4

Demo

Lets see in action

Check VBS, HVCI and kernel CET configuration on the target.

Load the vulnerable signed driver backend and initialize physmem.

Resolve ntoskrnl, system/session DTBs and VA->PA translation.

Run native SSDT DbgPrint and token payloads.

Run Shadow SSDT DbgPrint and token payloads from the GUI thread.

Confirm restoration/readback and collect debugger output.

```
physmem.exe --backend <driver>
--ssdt
physmem.exe --backend <driver>
--shadow-ssdt
kd> !dbgprint
whoami /priv
```

PART 5

Defense

What you should be watching

Reduce the attack surface

Block known-abusable signed drivers yourself, don't trust MS.

Physical memory APIs calls

Flag unusual IOCTLs that expose raw physical reads/writes.

Dispatch integrity

Watch SSDT and Shadow SSDT entries, for short mutation and session context.

Is VBS/HVCI/kCET enough to protect you?

… not really

VBS/HVCI is necessary but not sufficient against data-only attacks

Takeaways

Modern Windows exploit development is increasingly about data-only, not just shellcodes.

The SSDT and Shadow SSDT are always relevant as they define deterministic, reachable kernel tables.

VBS/HVCI/kCET change the board-rules, but physical write can still subvert the dispatch state.

Short-lived redirection plus restore reduces risk and narrows detection windows but it’s not bullet-proof.

Mitigation needs special driver allow listing, memory-integrity telemetry and dispatch-state validation altogether.

¡MUCHAS GRACIAS!
ESKERRIK ASKO!

Subverting the Windows Kernel with VBS / HVCI / kCET enabled

Questions? Preguntas?

Let’s connect! <jsacco@exploitpack.com>