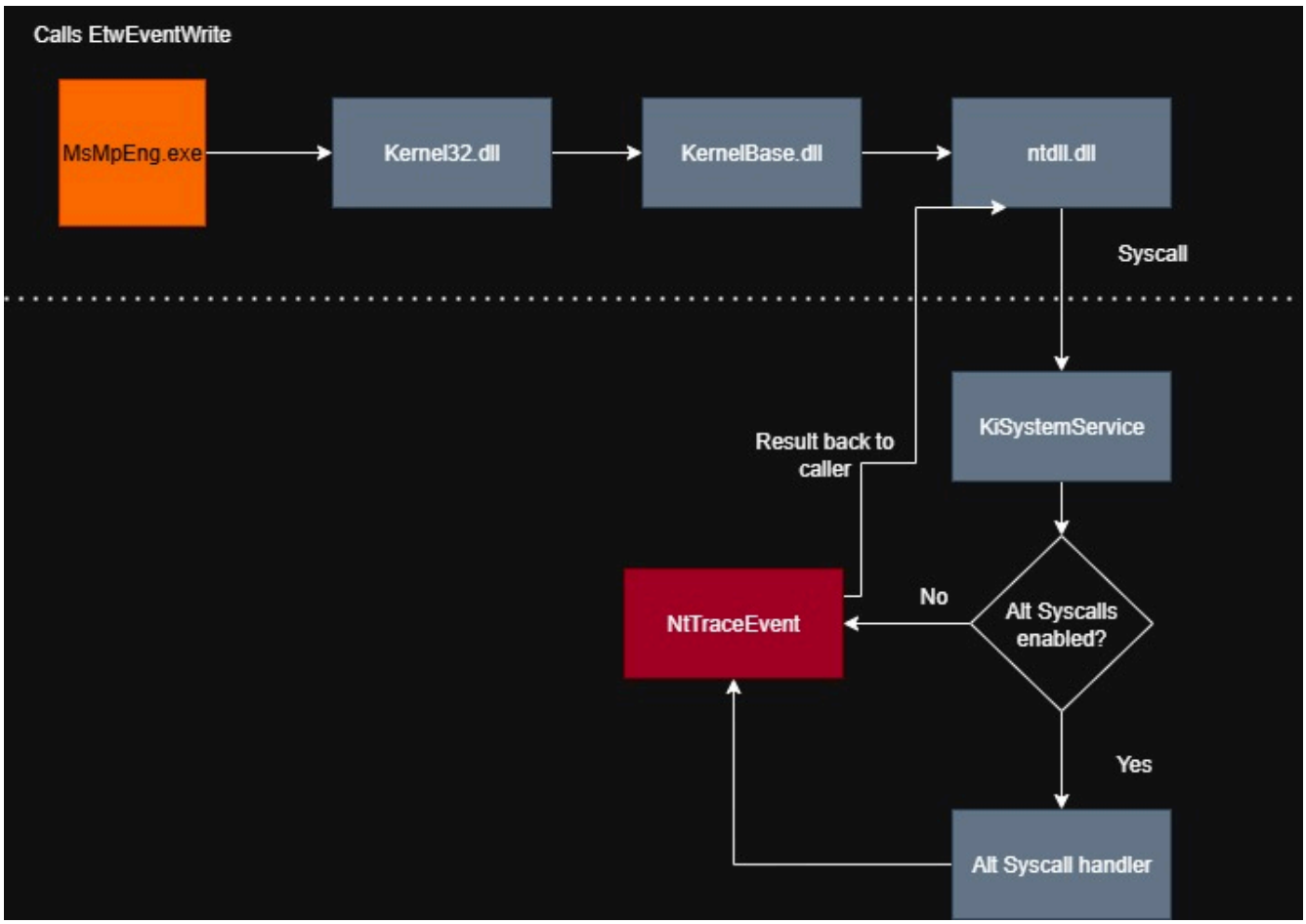
Hells Hollow: A new SSDT Hooking technique

fluxsec.red/hells-hollow-a-new-SSDT-hooking-technique-with-alt-syscalls-rootkit



Everything is a weapon in the belly of the beast.

Intro

The proof of concept code for this can be found on GitHub at [0xflux/Hells-Hollow](). If you like my project and post, please feel free to give the repo a star! :)

To see **limitations** of this technique, I have included that as a separate section below.

Hells Hollow is my name for a technique which has come about from my research into [Alt Syscalls](). In that post, I explained how when using the Alt Syscalls technique (for defensive EDR purposes) you must always return 1. This time, we are going to return something else and start messing with the Operating System, allowing you to hook and modify the **actual KTRAP_FRAME** of all syscalls (not just a copy of the trap).

Whilst in this post we focus on ETW, **Hells Hollow** can be used in any number of creative ways - this name can describe any number of malicious activities we can conduct via this technique, essentially, a SSDT bypass for rootkits that Microsoft managed to defeat via PatchGuard, for Windows 11. It is worth noting, that my tests have shown this is fully resistant to PatchGuard and 'HyperGuard' (HG testing done on a 'jailbroken' SecureBoot VM where you can add a root certificate to the OS to accept a custom signed driver, with Hyper-V fully enabled). As ever with the scientific method, I would challenge anybody to test this for yourself to see if you have a differing outcome with regards to HyperGuard sensitivity.

This technique is only available to a Rootkit, a piece of malware operating within the Windows Kernel. Whilst that may sound a high barrier for attack, Microsoft say that implementing Virtualisation Based Security (VBS) showed 60% fewer malware reports to Defender - indicating the Kernel is a legitimate and highly sought after attack surface. Whilst VBS prevent's the loading of Rootkits under certain circumstances, it does not prevent rootkits loading from a legitimately signed driver or via exploits. Therefore, the kernel is a highly valuable target for threat actors.

Touching briefly on our POC here - Events Tracing for Windows (ETW) is a critical component of how the operating system, and programs running on it, log events which can be consumed en-masse by other programs. One key purpose of this is for security. I am not going to go over what ETW is here, I have already explained that in another blog post. There is no shortage of documented usermode ETW bypass methods of disabling ETW in userland, the most common of these tends to be the patching of NtTraceEvent in ntdll.

Another caveat to this technique specific for ETW, it will only disable logging for ETW events which are logged via the user-mode calls. Direct ETW logging from the kernel is not defeated by this technique. That still leaves a significant enough attack surface for us, as per classic ETW evasion techniques report.

Limitations

Thanks to some testing by Xacone, we now know that **HVCI** prevents writing to the `PspServiceDescriptorGroupTable` structure; so that is one limitation of this technique. From my own testing, it appears that this is still resistant to both PatchGuard and HyperGuard under VBS. I used ssde to load my driver whilst Secure Boot and VBS were enabled, of which it is my understanding should be enough to test it against HyperGuard. This was done with debug mode off, which should also allow PatchGuard full authority to detect and block (BugCheck) the technique.

This research is valid only for Windows 11 (tested on 24H2).

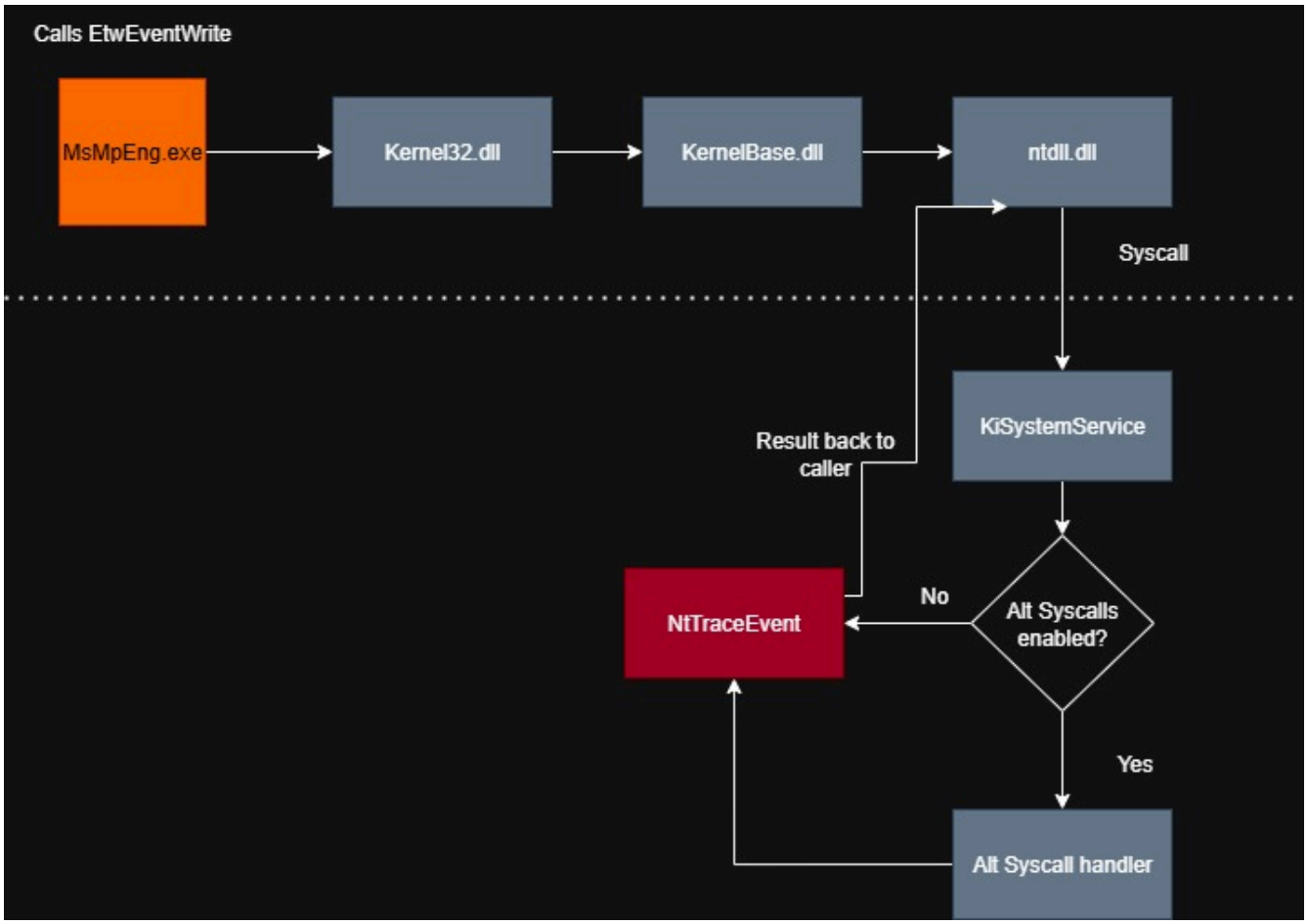
The belly of the beast

As a refresher, **Alt Syscalls** is a highly undocumented (and apparent not yet stabilised) feature of the modern Windows Kernel which allows for alternative system call handling, first discovered on Windows 10 (6 years ago) by 0xcpu, and written up nicely by Xacone, it was then reverse engineered for Windows 11 by me after significant key internal kernel changes. Before Patch Guard came along, adversaries, via a Rootkit, were able to alter the System Service Dispatch Table to take control of System Calls. Whilst Patch Guard is responsible for a number of things, it's primary purpose is in the prevention of rootkits, one of the most notable features of PatchGuard as reported by cert.it, is you guessed it, preventing rootkits from taking over, or reading, syscalls. I bring this thwarted technique back to life with **Hells Hollow**.

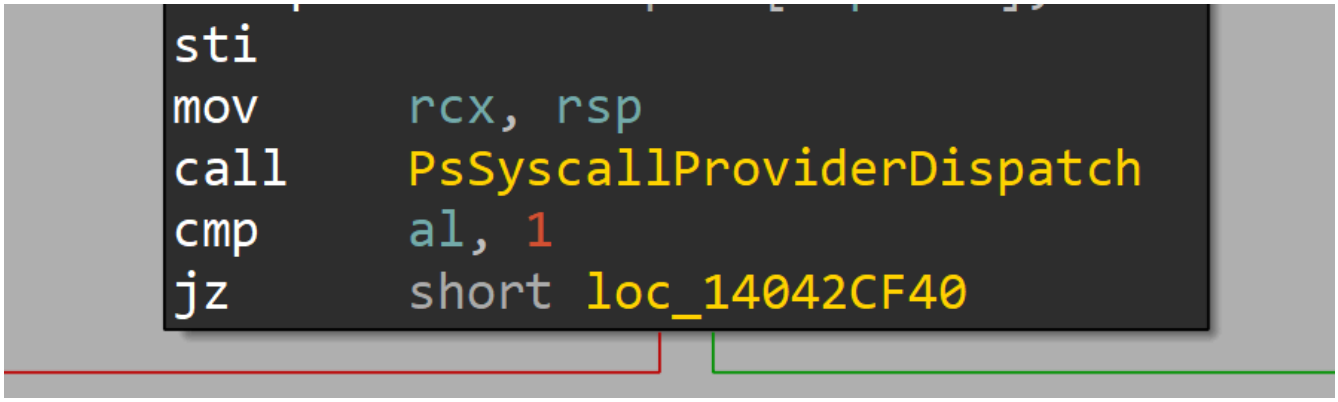
Since then; this technique was totally off limits to rootkits, unless you had a PG bypass, or relying on (luck based) synchronisation.

Not only does **Hells Hollow** allow us to intercept the system call, we can completely take control of it - preventing the actual dispatch routine to take place if we should so choose, or altering the arguments provided to be dispatched.

An ordinary System Call (including the Alt Syscall flow) can be visualised as below (**note**, you can find the Hells Hollow version of this diagram below in this post):



In my previous post on Alt Syscalls, I made the comment we must always **return 1** from the function, and didn't provide any more context at the time. The logic checking this can be seen as below:



If however, we return zero, we can short-circuit the syscall reaching the **Nt** implementation in **ntoskrnl** entirely, and execution returns to usermode. Further, we can modify the trap frame directly, as well as control the return code back to usermode!

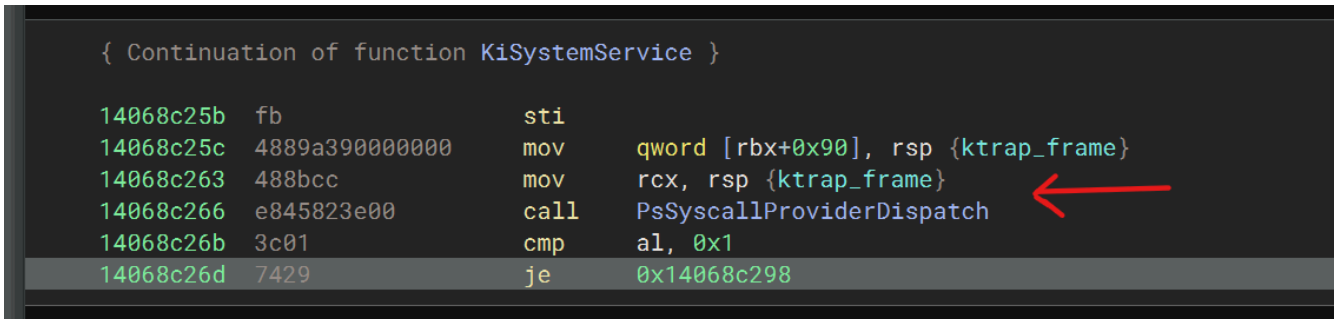
The Devils trap

Update: @sixtyvividtails has pointed out that instead of doing everything I have written below to get the trap, we can get the address directly from the KTHREAD. I have updated the POC to use this technique as pointed out by them, as it is far easier. That said, I am leaving the below in this post as a record of my research. Enjoy!

On first glance and whilst exploring ways in which to exploit the Alt Syscalls mechanism, my first thought was “how can we modify the KTRAP_FRAME such that we have full control over a system call”? Well, we would need access to this! On first inspection, there was no **KTRAP_FRAME** on the stack nearby. In fact, a reference to it lived solely in registers which were since overwritten, and in other cases (such as below) what was available was the data in the dereference of the KTRAP_FRAME.

```
140749124    int64_t PspSyscallProviderServiceDispatchGeneric(struct _KTRAP_FRAME_1* ktrap_frame, void* jmp_address,
140749124      uint64_t flags, int32_t arg4, int64_t* arg5)

140749124  4053              push    rbx {__saved_rbx}
140749126  56                push    rsi {__saved_rsi}
140749127  57                push    rdi {__saved_rdi}
140749128  4881ece0000000    sub     rsp, 0xe0
14074912f  488b05aa136c00    mov     rax, qword [rel __security_cookie]
140749136  4833c4            xor     rax, rsp {var_f8}
140749139  48898424d0000000  mov     qword [rsp+0xd0 {var_28}], rax
140749141  488b4138          mov     rax, qword [rcx+0x38 {_KTRAP_FRAME_1::Rcx}]
140749145  488bf2            mov     rsi, rdx
140749148  488bbc2420010000  mov     rdi, qword [rsp+0x120 {arg5}]
140749150  4889442430        mov     qword [rsp+0x30 {Rcx}], rax
140749155  488b4140          mov     rax, qword [rcx+0x40 {_KTRAP_FRAME_1::Rdx}]
140749159  4889442438        mov     qword [rsp+0x38 {Rdx}], rax
14074915e  488b4148          mov     rax, qword [rcx+0x48 {_KTRAP_FRAME_1::R8}]
140749162  4889442440        mov     qword [rsp+0x40 {R8}], rax
140749167  488b4150          mov     rax, qword [rcx+0x50 {_KTRAP_FRAME_1::R9}]
14074916b  4889442448        mov     qword [rsp+0x48 {R9}], rax
140749170  418bd9            mov     ebx, r9d
140749173  4584c0            test    r8b, r8b
140749176  7422              je      0x14074919a

140749178  488b8980010000    mov     rcx, qword [rcx+0x180 {_KTRAP_FRAME_1::Rsp}]
14074917f  488d542450        lea     rdx, [rsp+0x50 {out_params}]
140749184  4883c128          add     rcx, 0x28
```

At this point, I started walking the stack frames backwards, until I got back into the start of this whole descent into madness, **KiSystemServiceUser**. If we are dereferencing the KTRAP_FRAME, then surely it must exist on the stack somewhere. And, indeed it does! Looking in **KiSystemServiceUser**, we pass the address of the KTRAP_FRAME into our Alt Syscall dispatch function (**PsSyscallProviderDispatch**), and that so happens to be at the stack pointer of that particular frame:



Great! Well, seeing as though we are only a few stack frames below in our callback routine (directly below) - we can just walk back up the stack to get the original `KTRAP_FRAME`!

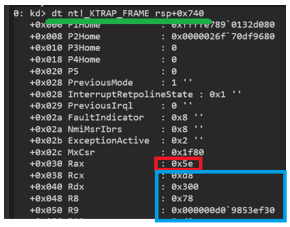
I calculated this my meticulously stepping through everything:

```
Actual trap                                          ffffab8a79276ae0
PsSyscallProviderDispatch                            ffffab8a79276ad8 -> 8
                                                     ffffab8a79276a80 -> 0x58
PsSyscallProviderDispatchGeneric                     ffffab8a79276a78 -> 8
Immediately before call dispatch                     ffffab8a79276980 -> 0xF8
Inside dispatch                                      ffffab8a79276978 -> 8
Callback                                             ffffab8a79276978 -> (no stack a
sub rsp inside the callback                          ffffab8a792763a0 -> 0x5D8 (Sta

total                                                = 0x740
```

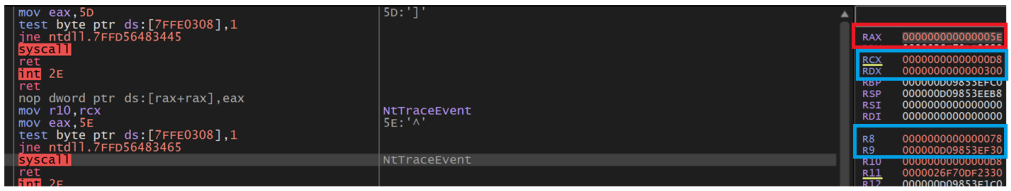
So, here we have an absolute offset from the start of our callback function, **168h** bytes, **PLUS** the stack subtraction of our callback routine. That callback stack is going to be the clinch when it comes to automating this process, which is a future endeavour.

In an easier sentence, we calculate **rsp+740h** from our callback to find the start of the **KTRAP_FRAME**.

So, from WinDbg, we can check our math is correct by coercing that address to a **_KTRAP_FRAME** struct. On the left of the image you can see the kernel debugger (via WinDbg) from my host machine, and on the right you can see **x64dbg** running on the guest which is attached to a process making the relevant syscall. As we are focusing just on ETW here, we are making the syscall into NtTraceEvent, which on Windows 11 has the SSN (System Service Number) **0x5E**. In the red boxes, you can see the RAX register is set to 0x5E, and the blue boxes show the first four parameters to the function; equivalent on each side.



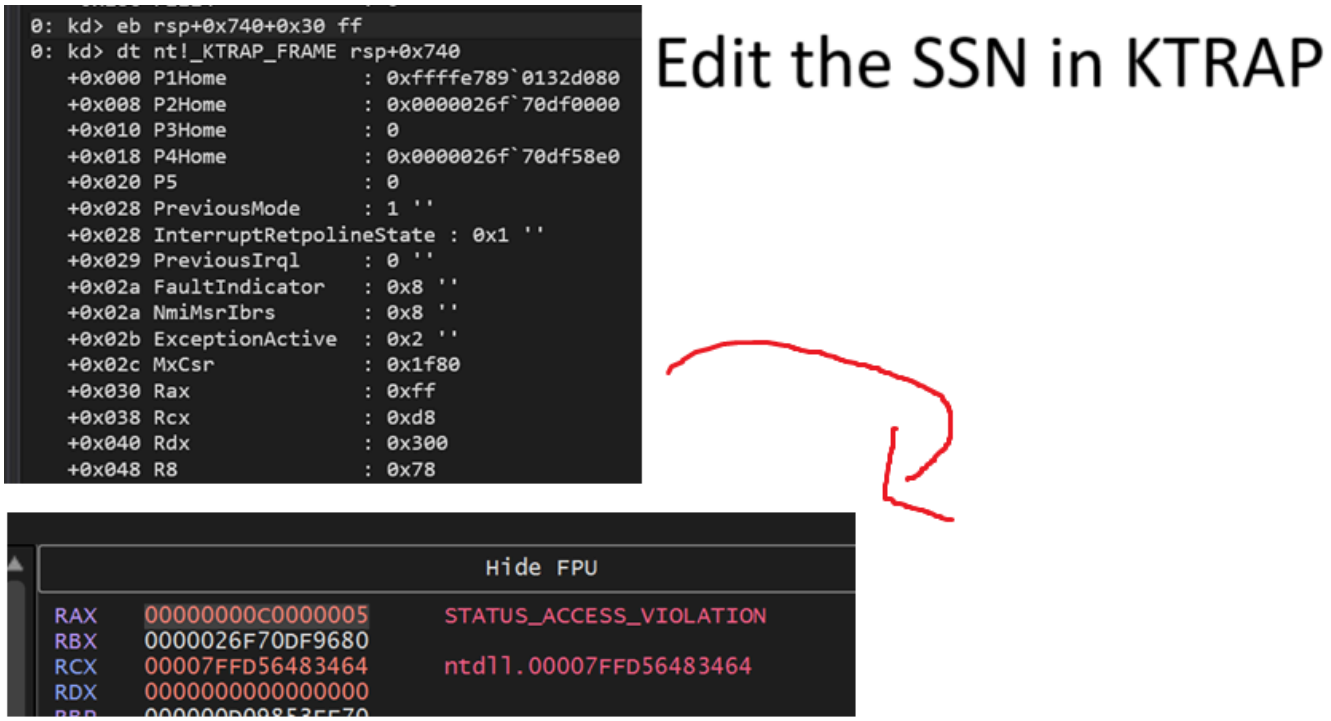
Kernel debugger looking at KTRAP



x64dbg in the process making the syscall

Modifying the trap

Next we need to modify the **_KTRAP_FRAME** directly, well, this is as easy as calculating the above offset, and writing to the address. For example, if we alter the SSN of the system call (and force the Operating System to handle the syscall after our dispatch routine), you can see in the result of the syscall, we get an error code (in this case, access violation):



We made the kernel try dispatch NtGetMUIRegistryInfo (SSN 0xff) of which the third parameter that function expects is a valid memory address; which looking at our r8 parameter, it is not, thus, it follows it is correct to receive an access violation when calling that Nt function.

So; this allows us to fully intercept and modify system calls.

Note that in the above case, we forced the kernel to dispatch the syscall by returning 1 from our callback. What follows is returning 0 to bypass kernel dispatching, and in turn, returning our own result back to the syscall caller. You could of course, just return 0 if you want to drop

a syscall without the added step here of altering the return value. In which case, you nullify the syscall.

Modifying the syscall return value

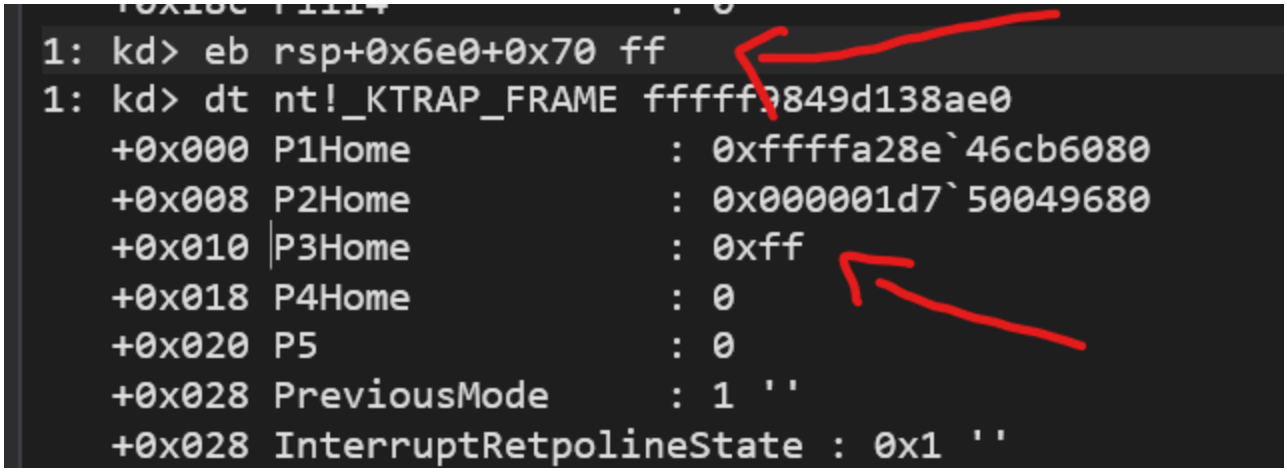
Finally, we may want to modify the actual value returned into **rax** from the 'apparent' syscall (perhaps you can exploit / trick an application into thinking it was successful, when it wasn't under circumstances you control).

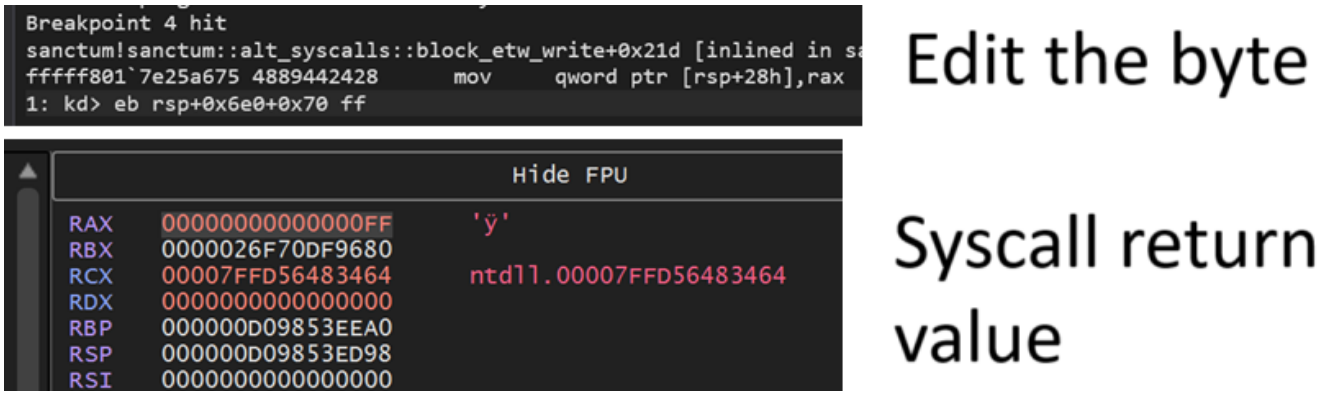
Returning to our disassembly once more, we can see that some stack variable (at **rsp+70h**) is placed into the **KTRAP_FRAME** + 30h, which is for the **rax** register (or, stack location for what goes back into the register).

Doing a little math over the stack layout, the distance from rsp within my callback to rsp at that moment of time in **PsSyscallProviderDispatch**, is **0x6e0**. Thus, **0x6e0 + 0x70 = KTRAP return value**.

It follows therefore, that from within our callback, we can simply modify this memory address to contain some value we want returned back to the caller, such to the effect of **rsp+0x6e0+0x70**.

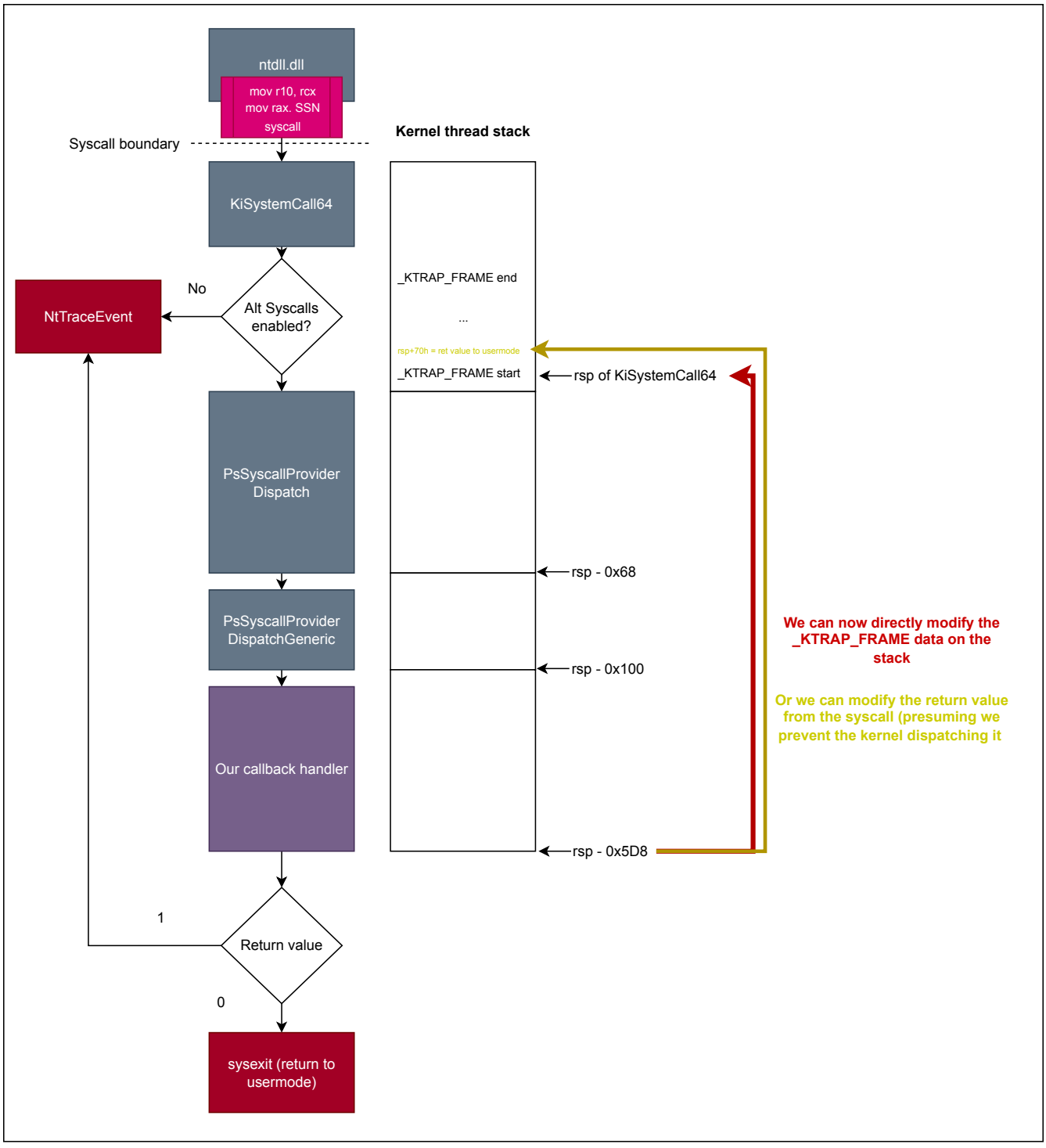
In fact, it follows (based on the stack offsets) that the value we edit at `rsp+0x6e0+0x70`, is actually the **P3Home** parameter of the **_KTRAP_FRAME**. You can see this in the below visual aid, with relevant offsets added.





Visual aid

I have prepared the below visual aid which helps I think in conceptualising this.



Proof of concept for ETW

Back to the original purpose of this, yet another technique to bypassing Events Tracing for Windows - this time with my **Hells Hollow** technique :).

In my previous blog post on evading usermode ETW, we used a Windows Rust project for interfacing with ETW as a simple testbed. The result of this using Hells Hollow is exactly the same as patching ntdll with a **return** instruction; 0 ETW logs.

The proof of concept code here gets the original **_KTRAP_FRAME**, prints data about it, and modifies the return value of what goes back to userland (0xff is returned in rax). Note, I'm only doing this for processes with "hello_world" in their process name so we can make these tests manageable (which is the process name of the Microsoft Rust ETW example code we are running the test against). We return 0, such that the actual syscall isn't dispatched by the normal SSDT, in effect, this is our working SSDT bypass in action:

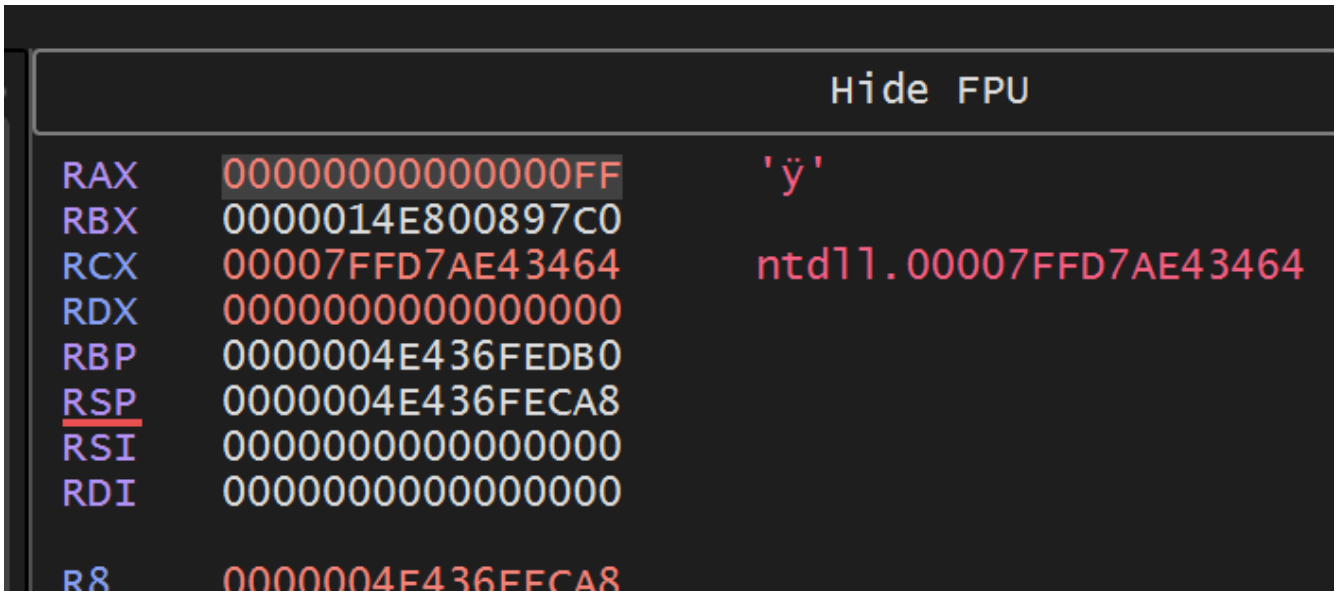
```
#[inline(always)]
fn block_etw_write(
    ssn: u32,
    args_base: *const c_void,
) -> Result<i32, ()> {

    let proc_name = get_process_name().to_lowercase();

    if proc_name.contains("hello_world") {
        println!("Found hello world");

        let mut rsp_val: u64 = 0;

        unsafe {
            asm!(
                "mov {out}, rsp",
                out = out(reg) rsp_val,
                options(nomem, nostack, preserves_flags),
            );
        }

        // rsp + offset of stack frames calculated.
        let trap_addr = (rsp_val + 0x540 + 0x210) as *mut _KTRAP_FRAME;

        println!("Addr: {:p}", trap_addr);

        let mut ktrap: _KTRAP_FRAME = unsafe { *trap_addr };

        // change the return value to usermode
        unsafe { (*trap_addr).P3Home = 0xff };

        // print the SSN
        println!("RAX: {:X}", ktrap.Rax);

        return Ok(0);
    }

    Ok(1)
}
```

```
#[inline(always)]
fn get_process_name() -> String {
    let mut pkthread: *mut c_void = null_mut();

    unsafe {
        asm!(
            "mov {}, gs:[0x188]",
            out(reg) pkthread,
        )
    };
    let p_eprocess = unsafe { IoThreadToProcess(pkthread as PETHREAD) } as *mut c_void

    let mut img = unsafe { PsGetProcessImageFileName(p_eprocess) } as *const u8;
    let mut current_process_thread_name = String::new();
    let mut counter: usize = 0;
    while unsafe { core::ptr::read_unaligned(img) } != 0 || counter < 15 {
        current_process_thread_name.push(unsafe { *img } as char);
        img = unsafe { img.add(1) };
        counter += 1;
    }

    current_process_thread_name
}
```

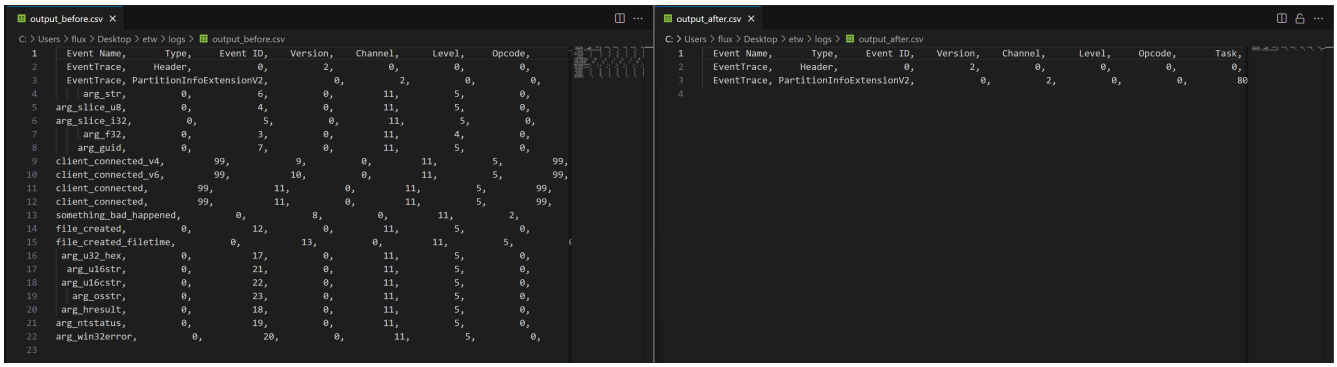
We can see the address of the **_KTRAP_FRAME**, as well as the SSN from **rax**, 0x5E!

```
1    0.00000000   Found hello world
2    0.00612670   Addr: 0xfffff481a419fae0
3    0.00976120   RAX: 5E
```

I also feel like I have posted similar images here of this way too many times; but here is a raw screenshot as evidence the above code did modify the return value:



I plan to make a video on this for the proof of concept and exploring how the internals of **Hells Hollow** works, as it may be easier to convey than a written article. For brevities sake, here is the before and after of bypassing ETW logging:



To reiterate, for ETW - this will NOT prevent logging which occurs from within the **kernel** where kernel-mode ETW events are logged. NtTraceEvent and ZwTraceEvent are not called whatsoever from within ntoskrnl, and as I have gone into in other blog posts, Kernel ETW logging is a whole separate mechanism which doesn't rely on making system calls. So as with usermode blocking of ETW, this will not be 'full spectrum'.

Summary

Through **Hells Hollow**, we are able to bring about a technique (SSDT hooking for Windows 11) which Microsoft themselves defeated many years ago because of Patch Guard.

Other creative ways in which **Hells Hollow** can be abused by rootkits (really, the limit is your imagination if you are creative enough):

- File and directory hiding

- Game cheats / anti-anti-cheats
- Process and thread hiding
- Registry manipulation and hiding (bar filter drivers)
- Network traffic filtering

If you have any feedback, please let me know either at 0xfluxsec, or at fluxsec@proton.me.