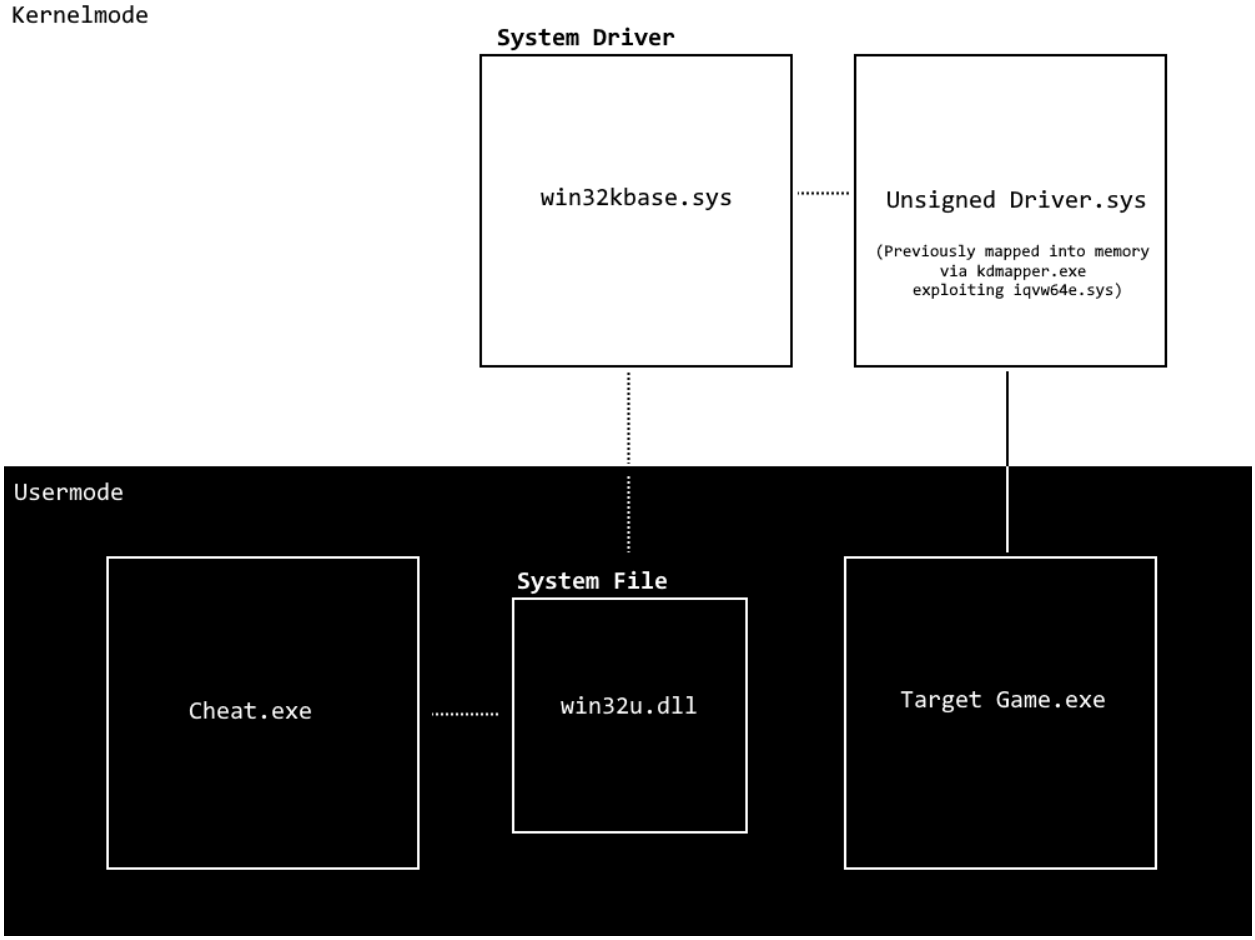
Exploiting Win32kbase Windows System Driver using Return Oriented Programming (ROP) for Kernel-Usermode Communication to Evade Detection from Easy Anti Cheat

Ethan Provost

To preface this report, we will be using KDMapper, a manual mapping tool that will allow us to load our unsigned kernel driver via the vulnerable **iqvw64e.sys** intel driver. We will call this our unsigned driver, which will be used to handle reading and writing memory requests from the usermode (cheat) application. Figure 1 depicts the structure of the PoC.

Figure 1



Overview

As seen above, our cheat program will call a vulnerable function, in this case NtCompositionInputThread, in win32u.dll, and pass a custom structure (that contains our read/write commands) as a parameter. This parameter will then be passed to the function in kernel space via win32kbase.sys. This will now allow our commands to be in kernel space. The kernel driver will then interpret those instructions by hooking NtCompositionInputThread and processing the commands, it will return the modified communication struct to the usermode application that can use that data to render an ESP, move the camera for an aimbot, and more.

Figure 2:


```
; __int64 __fastcall NtCompositionInputThread(void *, void *, int)
                public NtCompositionInputThread
NtCompositionInputThread proc near      ; DATA XREF: .data:00000001C02411F8↓o
                                        ; .pdata:00000001C026C308↓o

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                mov     [rsp+arg_0], rbx ; arg_0 stored in rcx
                                         ; (https://learn.microsoft.com/en-us/cpp/build/x64
                mov     [rsp+arg_8], rsi
                push    rdi
                sub     rsp, 20h
                mov     rax, cs:qword_1C024DE48 ; replace with ROP gadget ; 0x1c01aaa73
                mov     ebx, r8d
                mov     rdi, rdx
                mov     rsi, rcx
                test    rax, rax
                jz      short loc_1C01F1F4C
                call    cs:__guard_dispatch_icall_fptr ; jmp rax
                jmp     short loc_1C01F1F51
```

Explanation

The first step in implementing our exploit is to locate the *gadget* we are going to use. A *gadget* is a small set of assembly instructions, in the form of bytes, that is located somewhere throughout a binary. The presence of these bytes do not have to make up the instructions we want to execute, but simply be present. For example we may need to utilize a `pop rdi ; ret` *gadget*, so we need to search our target binary for this string literal "\x5F\xC3". Once we have located it, we can exploit an existing `jmp` instruction to jump to our *gadget* address. It will then execute the `pop rdi ; ret` instruction even if the actual instructions at that address make up some other data.

It is important to know where the parameters of a function are stored in memory. The first parameter is stored in the RCX register and the second in RDX, as the Microsoft x64 Calling Convention describes, "Integer arguments are passed in registers RCX, RDX, R8 and R9." Any functions that exceed this number of parameters, will push those parameters into the stack. (https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170)

With this information, we will need to locate a `push rcx; ret` *gadget*, represented by the following string literal: "\x51\xC3". In this instance of the win32kbase.sys binary, I have located the *gadget* bytes in the instruction `cmp r13; cs:gspepInit` at 0x1c01aa73.

Figure 3:

```
.text:00000001C01AAA73 4C 3B 2D 51 C3 1D 00    cmp     r13, cs:gpepInit ; Compare Two Operands
.text:00000001C01AAA7A 0F 84 C0 FE FF FF       jz      loc_1C006AD3D   ; Jump if Zero (ZF=1)
.text:00000001C01AAA80 85 C0                   test    eax, eax        ; Logical Compare
```

Once we have located our *gadget* in the vulnerable win32kbase.sys, we can swap the pointer of QWORD_1C024DE48 (located in the mov rax instruction in *Figure 2*) to the pointer to the gadget, 0x1c01aa73, using the kernel function InterlockedExchangePointer.

(https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-interlockedexchangepointer)

Once this .data pointer swap is in place, calling the NtCompositionInputThread function will move the *gadget* pointer into rax, and then redirect code flow to our *gadget*, due to the function calling `__guard_dispatch_icall_fptr` which contains instructions `jmp rax`. At the gadget pointer, it will execute `push rcx ; ret` which will push whatever is in RCX onto the stack, and then return. Essentially, we can execute *any* arbitrary unsigned code by passing the address to that unsigned code into RCX. We will use this by making RCX the address to our unsigned driver's communication function.

Figure 4 (Communication Structure):

```
typedef struct KernelRequest
{
    int magic;
    ULONG64 source_pid;
    ULONG64 source_address;
    ULONG64 target_pid;
    ULONG64 target_address;
    ULONG64 size;
    const char* module_name;
    void* output;
    RequestCode code;
}KernelRequest, * PKernelRequest;
```

The communication function also known as the *FunctionHook* function, processes commands that are passed through the second parameter in RDX as a pointer to the communication structure. It will parse the communication structure (*Figure 4)* and execute commands we implement in our unsigned driver such as GetBaseAddress, WriteMemory, and ReadMemory. Because we are redirecting code flow from an NT Win32kbase function to this function, our communication function will need to reflect the NT function we are calling in usermode. In this instance, we are calling NtCompositionInputThread, so our communication function will have the following function signature:

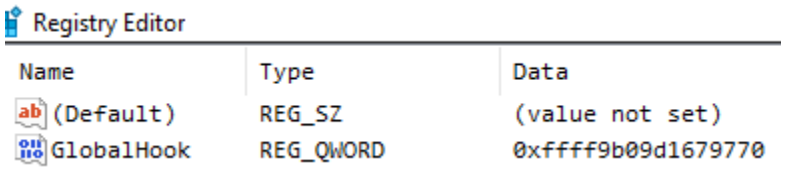
Win32kbase:

```
; __int64 __fastcall NtCompositionInputThread(void *, void *, int)
```

Recreated in our Unsigned Driver:

```
__int64 __fastcall FunctionHook(uint64_t rcx, uint64_t rdx, uint64_t r8)
```

Now we need to find a way to store the address of our kernel communication function. For this exploit, by creating a registry entry in the kernel, I will be storing the address to our *FunctionHook* in a registry key as a DWORD. This will be useful later as we will be able to retrieve registry data from our usermode component.



Now we will be focusing on the code in the usermode component. Suppose we are calling NtCompositionInputThread from usermode in win32u.dll (See the boxed/outlined function call in *Figure 5*). In that case, we can pass a pointer to our *FunctionHook* as our first parameter by accessing the registry key we created. Secondly, we will pass in a pointer to the communication structure as the second parameter to this function. That struct will contain data the unsigned driver will use to read and write to the target process.

Figure 5 (Usermode)

```
class Driver
{
private:
    void* function_address = nullptr;
    __int64(__fastcall* function)(uint64_t, uint64_t, uint64_t, uint64_t);
    uint64_t hook_base = NULL;
    DWORD pid = NULL;
public:
    bool DriverInit()
    {
        hook_base = Registry::ReadRegistry(_("SOFTWARE\\Unnamed"), _("GlobalHook"));
        if (!hook_base)
        {
            printf(_("[!] Error Map Driver\n"));
            return false;
        }

                LoadLibraryA(_("user32.dll"));
                LoadLibraryA(_("win32u.dll"));
                function_address = GetProcAddress(GetModuleHandleA(_("win32u.dll")), _("NtCompositionInputThread"));
                if (!function_address)
                {
            printf(_("[!] Failed To Find Function Address\n"));
                        return false;
                }
                *(PVOID*)&function = function_address;
        return true;
    }
    template<typename t> t read(uint64_t address)
    {
        if (address)
        {
            KernelRequest request;
            t buffer = {};
            request.magic = 0x1337;
            request.code = RequestCode::read;
            request.target_pid = pid;
            request.source_pid = GetCurrentProcessId();
            request.source_address = address;
            request.target_address = (uint64_t)&buffer;
            request.size = sizeof(t);

            function(hook_base, (uint64_t)&request, 0, 0);
            return buffer;
        }
        return t();
    }
}
```

Conclusion

With this in place, we have hooked NtCompositionInputThread from our unsigned driver, and can send and process the communication structure and execute its commands.

One question that arose with my research was if we would be able to use the third or fourth parameter of a function if the first is being occupied or is vital to reaching the point of code execution that we need. In this scenario, if we are using the second parameter, according to the x64 calling convention, it would be stored in RDX. Therefore, we would need to find a gadget with the following assembly and shellcode: `push rdx ; ret` and "\x52\xC3" respectively. In the unsigned driver and usermode, we would need to send and process our communication struct through the R8 register.

In *Figure 6*, you will see the driver operating to find the base address of an Easy Anti Cheat protected game, Apex Legends.

Figure 6 (Proof of Concept):