Module 3 - Reverse Engineering Indirect Syscalls and VEH

Module 3: Indirect Syscalls and Exception Handlers

Module Overview

This module will cover a technique that malware authors use to evade detection by Malware Sandbox tools and Endpoint Detection and Response (EDR) tools, by skipping over any hooks that the sandbox or EDR have placed around Windows API functions to call those functions directly. We will also cover a technique designed to fool malware analysts by putting some code into a Vectored Exception Handler (VEH) or Structured Exception Handler (SEH) and then intentionally triggering an error condition that will make the program's control flow jump to the exception handler code.

We'll see how these two techniques can be combined by using an exception handler to modify Windows API function calls in-flight, so that it looks like the program is calling one function (something benign and uninteresting) but at the last minute it actually calls a different function entirely!

Part 1: Indirect Syscalls made easy with SysWhispers

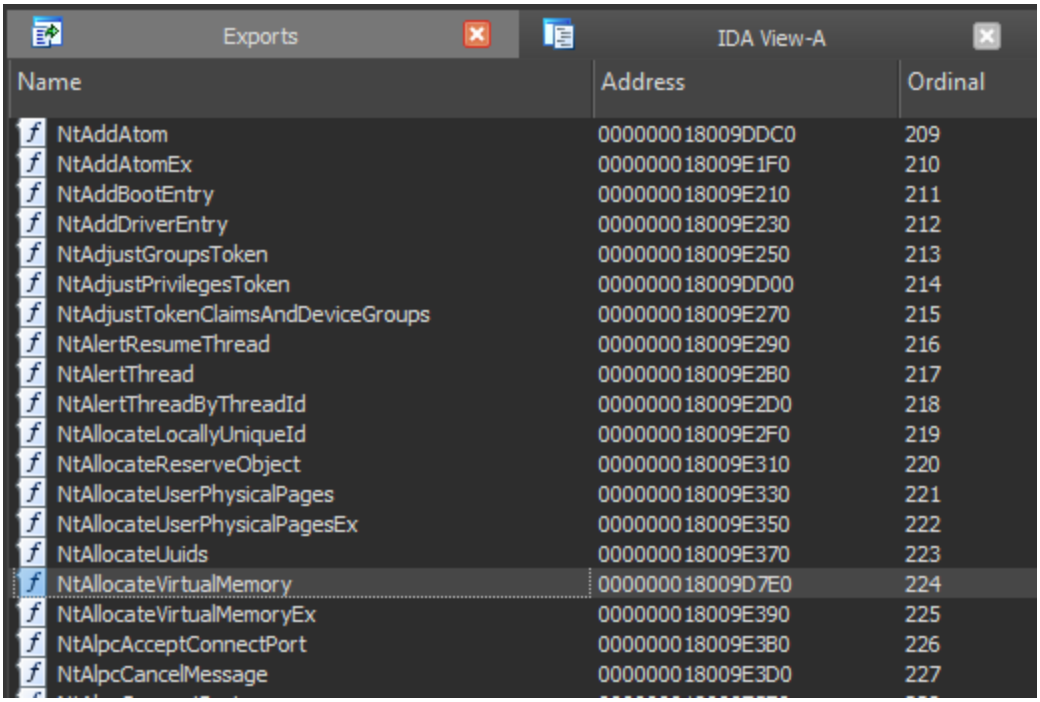
What are System Calls (syscalls)?

User-run processes must ask the OS to perform any actions with files, network connections, other processes, or anything interesting at all. That is so that the OS can do its job to check permissions of the user or service account that is requesting the operation and then enforce those permissions to allow or deny the action it requested. Every OS has a set of API functions that are documented for user programs to call if they wish to ask the OS to perform these actions on their behalf. For example, Windows has an API function called **VirtualAlloc**, documented in *memoryapi.h* and implemented in kernel32.dll, which allows a process to request allocation of memory in its own process address space. There's another Windows API function called **VirtualAllocEx**, documented in the same *memoryapi.h* header file and also implemented in kernel32.dll, which allows a process to request allocation of memory in another process memory address space.

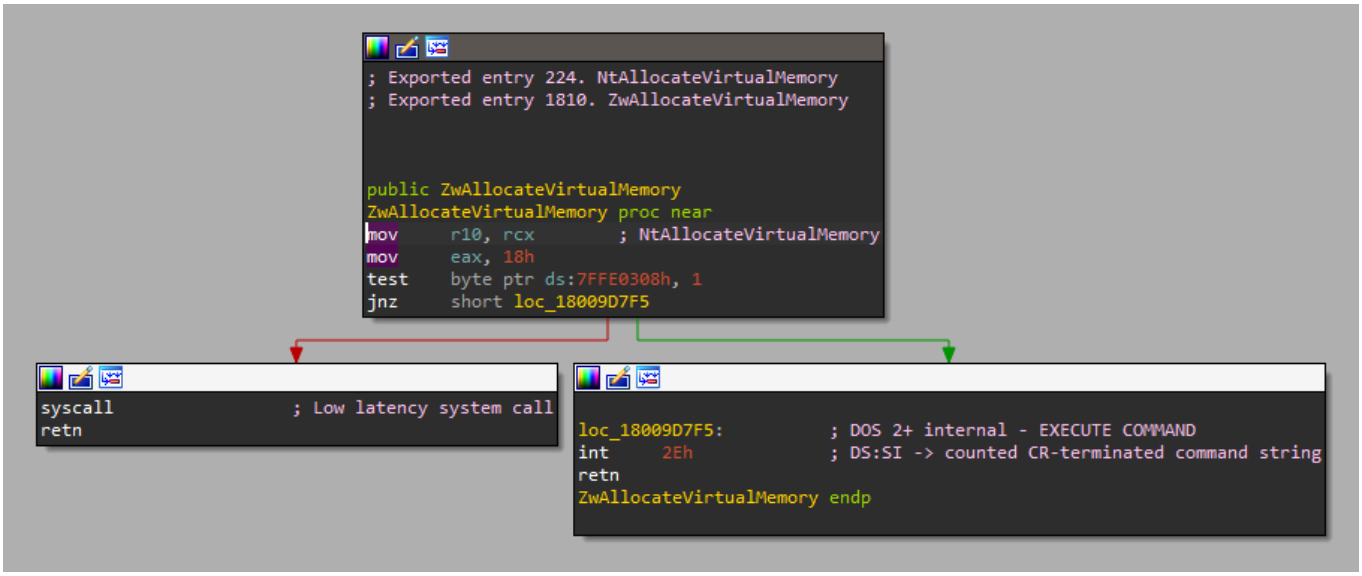
The documented Windows API functions don't perform the action directly, though. **VirtualAllocEx** calls a lower-level function called **NtAllocateVirtualMemory**, which is

implemented in ntdll.dll. Normally when you are reverse engineering and debugging a program, you would see the call stack show that the user program called kernel32.dll, and then kernel32.dll made a call to ntdll.dll. Let's use IDA to peek into ntdll and see how this function is implemented.

1. Open IDA and load a new file: C:\windows\system32\ntdll.dll
2. You will have to choose a different directory to save your IDA database file, because you don't want to (and shouldn't) save IDA files in system32.
3. Navigate to the Exports tab to see the functions that ntdll exports
4. Scroll down to NtAllocateVirtualMemory and double-click it



It may be somewhat surprising to you that these functions are so short and simple. Here is the complete code of **NtAllocateVirtualMemory**



The same function in code view shows the relative addresses of each instruction, which will be helpful information for malware authors hoping to abuse Windows functions to avoid detection.

```
.text:000000018009D7E0                 public ZwAllocateVirtualMemory
.text:000000018009D7E0 ZwAllocateVirtualMemory proc near       ; CODE XREF: RtlExtendMemoryZone+61↑p
.text:000000018009D7E0                                         ; LdrEnsureMrdataHeapExists+61↑p ...
.text:000000018009D7E0                 mov     r10, rcx        ; NtAllocateVirtualMemory
.text:000000018009D7E3                 mov     eax, 18h
.text:000000018009D7E8                 test    byte ptr ds:7FFE0308h, 1
.text:000000018009D7F0                 jnz     short loc_18009D7F5
.text:000000018009D7F2                 syscall                 ; Low latency system call
.text:000000018009D7F4                 retn
.text:000000018009D7F5 ; ---------------------------------------------------------------------------
.text:000000018009D7F5
.text:000000018009D7F5 loc_18009D7F5:                          ; CODE XREF: ZwAllocateVirtualMemory+10↑j
.text:000000018009D7F5                 int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.text:000000018009D7F5                                         ; DS:SI -> counted CR-terminated command string
.text:000000018009D7F7                 retn
.text:000000018009D7F7 ZwAllocateVirtualMemory endp
```

If you wish, take a look at a few other exported functions from NTDLL and notice that they all follow the same pattern, just with a different number, which we will examine in a moment.

Before going further, take a few minutes to think critically about this function from a malware author's point of view. Knowing that this function is probably hooked by EDR and sandboxes, how would you get around those hooks without tripping them?

As with all fastcall functions, the arguments to these Windows API functions will be stored in registers *rcx, rdx, r8, r9*. If there are more than 4 arguments, the rest are passed on the stack.

Notice that the function starts by moving the contents of register rcx into r10, because on x64 Windows, the syscall instruction expects the first argument in r10 -- not rcx as in fastcall -- per the syscall calling convention. In the next instruction (3 bytes later) it moves the immediate value 0x18 hex (24 decimal) into eax. The value 0x18 hex is the *System Service Number (SSN)* that stands for **NtAllocateVirtualMemory** in this build of Windows 10. You can look up the SSN values for all the other System calls for different versions and builds of Windows from XP

through 11 and Server on this website. You might notice that some system calls have the same SSN consistently while others have changed multiple times even within different builds of Windows 10. That's because these numbers are not meant to be used by programmers in user processes - they are supposed to be opaque internal values used only by Windows internal system components.

As a result of the fact that system service numbers can change between builds, most malware that abuses system calls for evasion will have some code that looks up the correct SSN for the system it is running on by reading ntdll.dll and/or kernel32.dll

At 8 bytes into the function, after setting the correct SSN in eax, the ntdll code will next test a byte at address 0x7FFE0308h to see if it is 1 or not. That is the address of the SharedUserData struct, SystemCall member. The documentation says "On AMD64, this value is initialized to a nonzero value if the system operates with an altered view of the system service call mechanism." In practice, for our purpose we don't really care about this, but it will just determine whether to call the system using the *syscall* instruction or by using the interrupt instruction *int 2Eh* instead.

Regardless of whether it uses the instruction *syscall* or *int 2Eh* interrupt, the result is the same: it transfers control to the kernel and the kernel will interpret the value that is in eax to tell it which system function the user process is requesting it to call.

That means, malware authors can abuse this function by setting up the arguments for the function they wish to call, making sure that the correct value for SSN is in eax, making sure the argument that should be in rcx is stored in r10, and then doing any one of the following:

- Calling the *syscall* instruction directly (this stands out as the most suspicious)
- Jumping into ntdll just before syscall would be called (less suspicious)
- Calling a normal Windows API function to go through the normal call chain pattern, but swapping out the SSN at the last minute (least suspicious)

The first technique, calling syscall directly from user code, is not very sophisticated and is easily spotted when reverse engineering, because no user program should ever call the syscall instruction. We will briefly analyze this technique but mainly focus on learning the other two techniques.

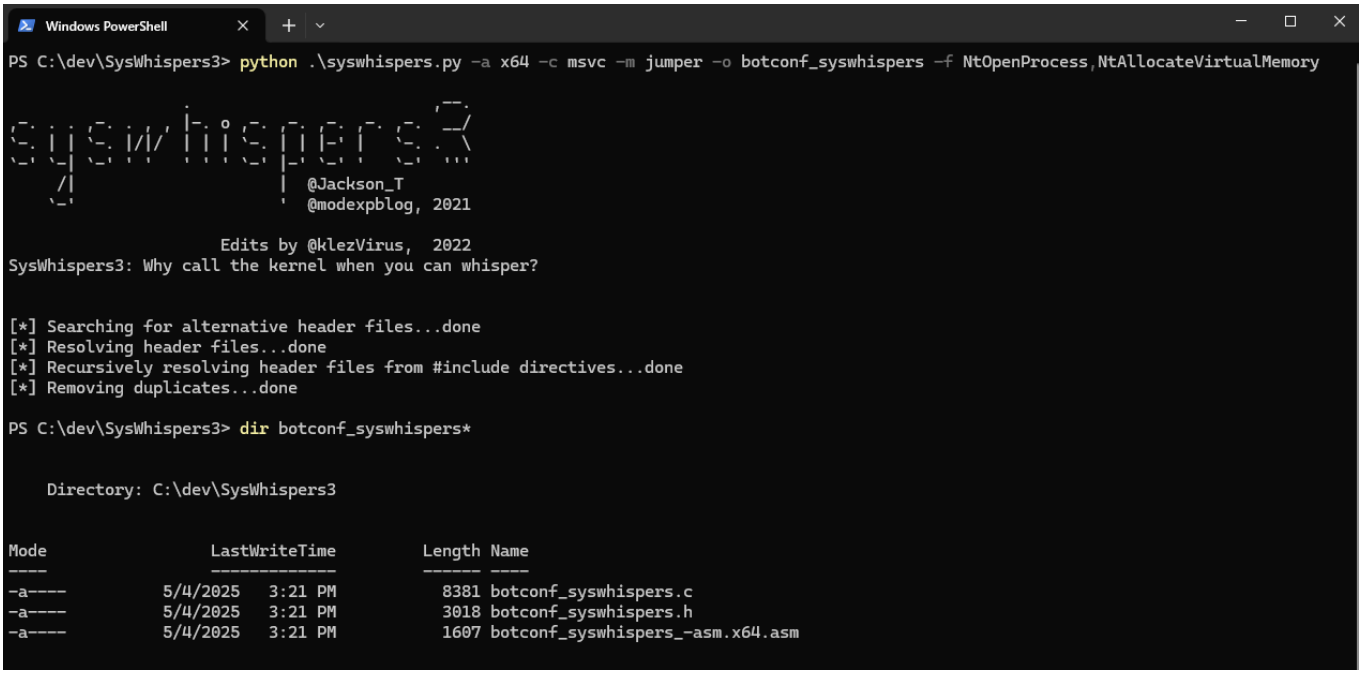
The second technique, jumping into ntll just before syscall would be called, is how SysWhispers works. SysWhispers is an open source project that uses a Python script to generate assembly code, c code, and a header file for a malware author to include in their Visual Studio (or other IDE) and use as a part of their malware program. It doesn't stand on its own, it is always included as a piece of another program.

The third technique is similar to SysWhispers, but instead of jumping directly into ntdll, it sets up a normal call chain going through a documented Windows API function just as programs normally should. In the example we'll analyze in this module, the code will use a technique called "MutationGate" to set a Vectored Exception Handler and cause an exception (using a hardware breakpoint) right at the point in the call to a benign, non-suspicious function to change the SSN to call the system function that the malware author actually wanted to call. In other words, it disguises a dangerous system call as if it were a non-dangerous, non-interesting function call.

SysWhispers3

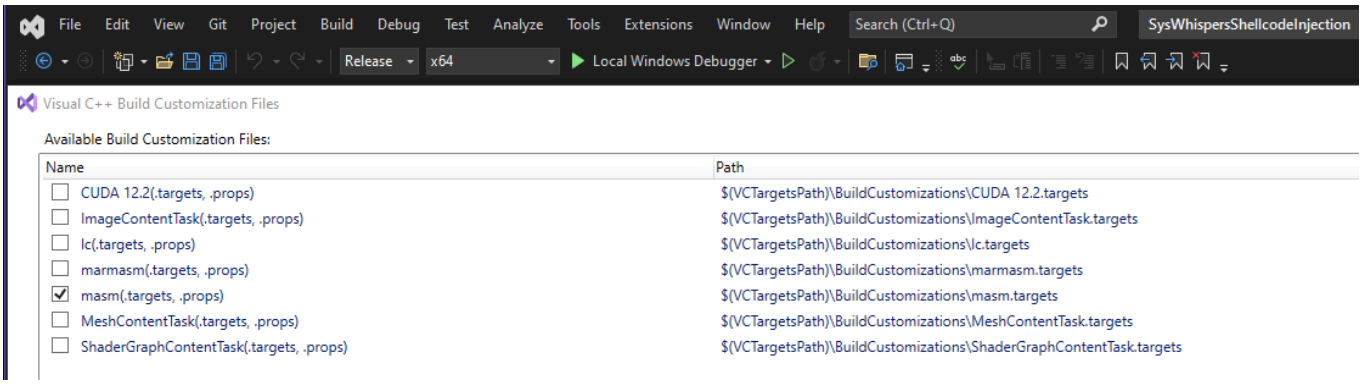
We've already done this part for you so you don't have to, but if you learn best by doing every step yourself, here's what you would do:

1. Download or clone the SysWhispers3 project: https://github.com/klezVirus/SysWhispers3
2. Make sure you have Python 3 installed
3. run "*python .\syswhispers.py --help*" to make sure it runs
4. run "*python .\syswhispers.py -a x64 -c msvc -m jumper -o defcon_syswhispers -f NtOpenProcess,NtAllocateVirtualMemory*"
5. Notice that we asked SysWhispers to build files so that we can call two functions: **NtOpenProcess** and **NtAllocateVirtualMemory** but you could include many other functions in that list if you wish, just by listing each function separated by a comma.
6. Note the output files created that start with "defcon_syswhispers"



Follow the instructions in the SysWhispers3 github README.md to add these three files as resources in your Visual Studio project. Note that in the community edition of Visual Studio

2022, you can find the Build Customizations menu by first selecting the project (not the solution) in the Solution Explorer, then clicking the "Project" top level menu bar item, then "Build Customizations"



This is slightly different from what the SysWhispers3 github instructions say, so hopefully this saves you a bit of time hunting through menus to find where it went.

Quick Review of Microsoft x64 calling convention

You can skip this if you're already familiar with how functions are called. If you want a deeper explanation with history, this Wikipedia article is a good source (scroll down to "Microsoft x64 calling convention").

By default when compiling a 64-bit program for Windows, the first four arguments to the function will be stored in the registers *rcx, rdx, r8, and r9*, and the rest of the arguments (if there are more than four) will be pushed onto the stack. The return value from the function when it is finished will be stored in the register *rax*.

For example, when calling NtAllocateVirtualMemory, there are six arguments passed to the function:

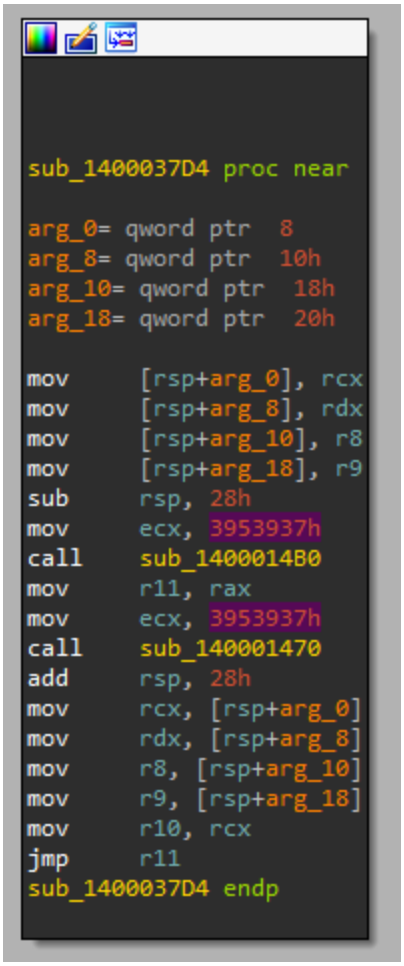
```
[in] HANDLE ProcessHandle,
[in, out] PVOID *BaseAddress,
[in] ULONG_PTR ZeroBits,
[in, out] PSIZE_T RegionSize,
[in] ULONG AllocationType,
[in] ULONG Protect
```

So, before calling this function, the ProcessHandle will be stored in rcx, the pointer to the BaseAddress will be stored in rdx, the pointer to the unsigned long integer ZeroBits will be stored in r8, the pointer to the size_t RegionSize will be stored in r9, the unsigned long integer Protect will be pushed onto the stack, and then the unsigned long integer AllocationType will be pushed onto the stack.

When NtAllocateVirtualMemory returns, the NTSTATUS value indicating whether it succeeded or not will be stored in rax.

Reverse Engineering Malware that uses SysWhispers3

When a malware author compiles a program using code generated by SysWhispers3, you will not see any references to the system functions they are calling directly. For example, the string "**NtAllocateVirtualMemory**" will not be in the Import Address Table. Your reverse engineering tools won't be able to tell you what function is being called. Instead, you will see a call to a function in the program that looks like this:



```
sub_1400037D4 proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     [rsp+arg_0], rcx
mov     [rsp+arg_8], rdx
mov     [rsp+arg_10], r8
mov     [rsp+arg_18], r9
sub     rsp, 28h
mov     ecx, 3953937h
call    sub_140001480
mov     r11, rax
mov     ecx, 3953937h
call    sub_140001470
add     rsp, 28h
mov     rcx, [rsp+arg_0]
mov     rdx, [rsp+arg_8]
mov     r8, [rsp+arg_10]
mov     r9, [rsp+arg_18]
mov     r10, rcx
jmp     r11
sub_1400037D4 endp
```

Let's take a look at the patterns in this very short function. First, notice that the beginning and the end of the function are similar. The first five instructions are saving the contents of *rcx, rdx, r8, and r9* to the stack, then subtracting 0x28 bytes from the stack pointer to make room for those four saved values. Remember that in the Microsoft x64 calling convention, the first four arguments to the function are stored in *rcx, rdx, r8, r9*, and this prevents problems in case those registers get overwritten during this function or the functions that are called. The last five instructions just before the final jmp are the reverse, adding 0x28 bytes to the stack pointer, and restoring the four values from the stack to *rcx, rdx, r8, and r9* -- you can effectively ignore these

instructions and remember that they are just saving and restoring the state of those registers, which contain the arguments for the system call, so that they don't get accidentally overwritten.

The next instruction is:
`mov ecx, 3953937h

That immediate value is the hash of **NtAllocateVirtualMemory** which SysWhispers will use to hunt for the correct address of the function in ntdll and again to hunt for the correct SSN to use for the version and build of Windows that it is running on.
There are two calls to functions. In this build, those functions were at particular addresses, but those will change with every program and aren't useful to remember. Just note that the first function call locates the address that is 8 bytes past the start of **NtAllocateVirtualMemory**, right at the spot where this instruction is called:

`test 0x7FFE0308h, 1
which is after the SSN has been set in eax, and just before the syscall instruction or the int 0x2E instruction is called. This is the address that we want to jump to. We store this in r11, and we will jump to that address at the end of this function.

The next instruction is just a repeat of setting the hash value for **NtAllocateVirtualMemory** into ecx, just in case ecx got overwritten.
The next function called (`sub_140001470` in this sample) will get the correct SSN for **NtAllocateVirtualMemory** which happens to be 0x18 in every version of Windows 10 and Windows 11. **Remember that the return value from functions are stored in rax, so as long as we don't modify rax after this point, when syscall gets called, the correct SSN will be in rax.**

At this point, we have all we need to call the system, so we just restore those four values to *rcx, rdx, r8 and r9* from where we had saved them on the stack, then move *rcx* to *r10* just like the real Windows API call does. In the case of **NtAllocateVirtualMemory**, the first argument is the process handle.

We end this function with a *jmp* instruction to r11, which holds the address in ntdll that we want to jump to, just before calling the system call, and happily (for the malware author) after any hooks that the EDR or sandbox may have installed. Recall that we previously saved the return value from `sub_1400014B0` (which finds the right address to jump to) into r11.

Whenever you are reverse engineering and see a tiny little function that looks like this, you can be pretty sure that it was either generated by SysWhispers3 using the "jumper" method, or borrowed from code inspired by SysWhispers.

How to find all the SysWhispers functions in the SysWhispersShellcodeInjection sample

We're going to focus on this sequence of instructions that always appears at the end of the SysWhispers function:

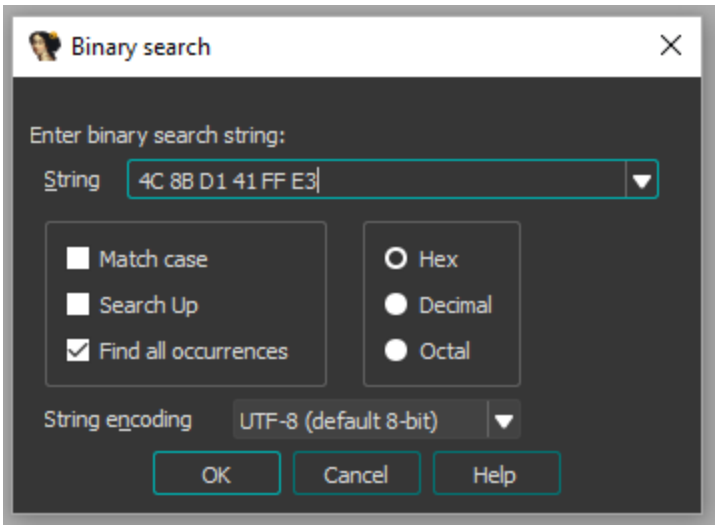
```
mov     r10, rcx        ; 4C 8B D1
jmp     r11             ; 41 FF E3
```

In IDA, we can search for text (which searches strings, disassembly, etc) or we can search for a sequence of bytes. Text is easier for you to type in the search box, but not as powerful, because you can search for one instruction "mov r10, rcx" or the other "jmp r11" but not both together. If you use text search, you also have to remember the correct number of spaces between the instruction and the operands (5 spaces), which is annoying and frustrating if you get it wrong.

The better way to search is to note that `mov r10, rcx` in machine code is `4C 8B D1` and `jmp r11` in machine code is `41 FF E3`, so we can use IDA to search for those six bytes as a sequence. Click on the Search menu item, then "Sequence of bytes..." and enter:

```
4C 8B D1 41 FF E3
```

(the spaces between the bytes are important)



Remember to look for these overall patterns:

1. Temporarily storing the values from *rcx, rdx, r8, r9* and restoring them just before a jmp or call
2. Calling a function that returns an address in ntdll, and storing that address on a register later used as the target for the jmp or call instruction at the end
3. Calling a function or hard-coding a number that corresponds to an SSN for some function (hint: use this table to find which function it is)

That's pretty handy knowledge, so let's generate some SysWhispers code using the other methods as well and compile each one to see what they look like! We used the "jumper" method for the example above, so we'll try each of the following:

1. embedded
2. egg_hunter
3. jumper_randomized

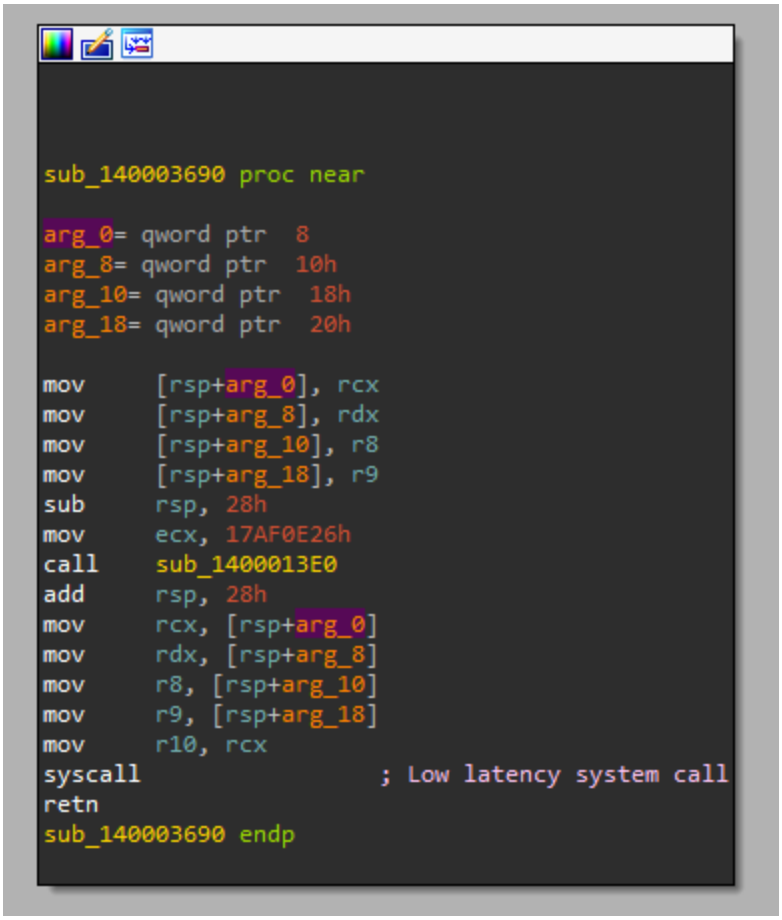
SysWhispers3 Embedded method

To produce this code, we ran "*python syswhispers.py*" again just as before, but instead of using the method jumper with "*-m jumper*" we changed the method to embedded with "*-m embedded*"

The code produced is very similar, but instead of finding the address in ntdll and jumping to it, this one calls *syscall* directly from user code. This is a lot more obvious and less sneaky for the malware because any syscall instruction in a user program stands out as weird.

In this sample, the call to `sub_1400013E0` returns the correct SSN for the function that the malware author wished to call. The return value is in *rax*, as usual, and since we don't overwrite *rax* afterward, that value is still in *rax* when the syscall instruction is called.

If you ever see this pattern when reverse engineering malware, you can easily recognize it for what it is:



```
sub_140003690 proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     [rsp+arg_0], rcx
mov     [rsp+arg_8], rdx
mov     [rsp+arg_10], r8
mov     [rsp+arg_18], r9
sub     rsp, 28h
mov     ecx, 17AF0E26h
call    sub_1400013E0
add     rsp, 28h
mov     rcx, [rsp+arg_0]
mov     rdx, [rsp+arg_8]
mov     r8, [rsp+arg_10]
mov     r9, [rsp+arg_18]
mov     r10, rcx
syscall                 ; Low latency system call
retn
sub_140003690 endp
```

SysWhispers3 Egg-Hunter Method

This is the trickiest method of all, because it creates code that won't actually run. To use it in practice, the malware author needs to write some other code in a loader that will hunt for a pattern of bytes (the "egg") in the memory of the thread containing this "broken" code, and replace those bytes with a different sequence of bytes that will "fix" the code so that it does work.

Here is the output of the SysWhispers3 script which shows you the bytes to look for and what bytes to replace them with:

```
PS C:\dev\SysWhispers3> python .\syswhispers.py -a x64 -c msvc -m egg_hunter -o mutationgate_syswhispers -f NtCreateThreadEx,NtCreateUserP
rocess,NtWriteVirtualMemory,NtOpenProcess,NtAllocateVirtualMemory,NtClose

                  .                          ,--.
,-. . . ,-. . , , |-. o ,-. ,-. ,-. ,-. ,-.  __/
`-. | | `-. |/|/  | | | `-. | | |-' |   `-. .  \
`-' `-| `-' ' '   ' ' ' `-' |-' `-' '   `-'  '''
     /|                    |  @Jackson_T
    `-'                    '  @modexpblog, 2021

                      Edits by @klezVirus,  2022
SysWhispers3: Why call the kernel when you can whisper?


[*] Searching for alternative header files...done
[*] Resolving header files...done
[*] Recursively resolving header files from #include directives...done
[*] Removing duplicates...done
[*] With the egg-hunter, you need to use a search-replace functionality:
  unsigned char egg[] = { 0x6a, 0x0, 0x0, 0x65, 0x6a, 0x0, 0x0, 0x65 }; // egg
  unsigned char replace[] = { 0x0f, 0x05, 0x90, 0x90, 0xC3, 0x90, 0xCC, 0xCC }; // syscall; nop; nop; ret; nop; int3; int3
```

This is all done to avoid having the instruction "syscall" in the compiled program, or from having it jump directly into an address in ntdll. After the "egg" has been replaced with the correct code, the syscall instruction will be in there.

Even as tricky as this is, you can still recognize the pattern when you see it as you reverse engineer a malware sample. Here is the function:

```
.text:00000001400036E8 loc_1400036E8:                          ; CODE XREF: main+4CD↑p
.text:00000001400036E8                 mov     [rsp+8], rcx
.text:00000001400036ED                 mov     [rsp+10h], rdx
.text:00000001400036F2                 mov     [rsp+18h], r8
.text:00000001400036F7                 mov     [rsp+20h], r9
.text:00000001400036FC                 sub     rsp, 28h
.text:0000000140003700                 mov     ecx, 0B91190Bh
.text:0000000140003705                 call    sub_1400013E0
.text:000000014000370A                 add     rsp, 28h
.text:000000014000370E                 mov     rcx, [rsp+8]
.text:0000000140003713                 mov     rdx, [rsp+10h]
.text:0000000140003718                 mov     r8, [rsp+18h]
.text:000000014000371D                 mov     r9, [rsp+20h]
.text:0000000140003722                 mov     r10, rcx
.text:0000000140003725                 push    0
.text:0000000140003727                 add     [rbp+6Ah], ah
.text:0000000140003727 ; ---------------------------------------------------------------------------
.text:000000014000372A                 dw 0
.text:000000014000372C                 dd 8948C365h
.text:0000000140003730                 dq 102454894808244Ch, 4C894C182444894Ch, 0E5B928EC83482024h
.text:0000000140003748                 dq 0FFFFDC90E8C054DDh, 244C8B4828C48348h, 8B4C1024548B4808h
.text:0000000140003760                 dq 20244C8B4C182444h, 6A6500006AD18B4Ch, 0CCCCCCCCC3650000h
.text:0000000140003778                 dq 0CCCCCCCCCCCCCCCCh, 6666CCCCCCCCCCCCh, 841F0Fh
.text:0000000140003790 ; [0000001E BYTES: COLLAPSED FUNCTION __security_check_cookie. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00000001400037AE algn_1400037AE:                         ; DATA XREF: .pdata:0000000140041324↓o
```

Notice how we can still see very familiar patterns:

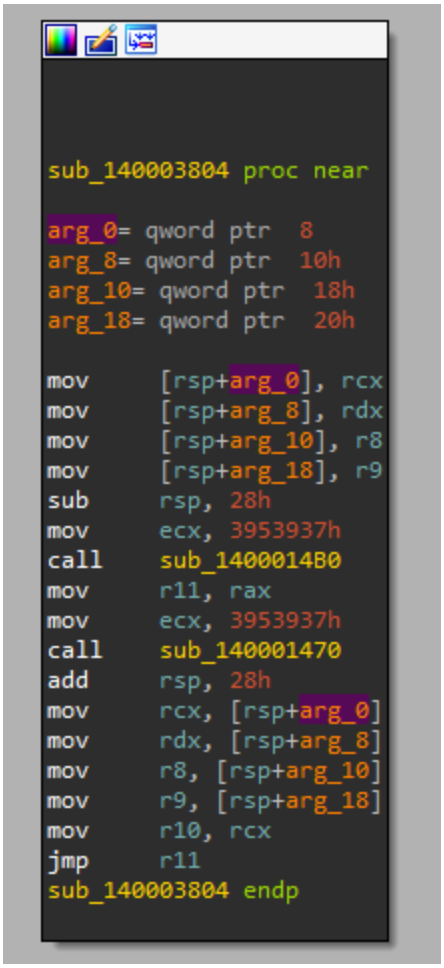
1. The four x64 function argument registers *rcx, rdx, r8, r9*
2. Subtracting 0x28 from the stack pointer
3. Putting an immediate value (the hash) into ecx
4. Calling a function that retrieves the correct SSN for the system call
5. Adding 0x28 to the stack pointer and restoring *rcx, rdx, r8, r9* values from the stack
6. Moving rcx to r10 (just like Windows does)
   These patterns are just like the setup to jump or call the syscall instruction that we've seen before, but instead of the syscall or jump, we see a mess of weird instructions and bytes that don't translate to instructions and our disassembler gets confused. If you see a pattern like this when reverse engineering malware, look for some other function that searches for a byte pattern and replaces these bytes in this function with the fixed version. Note that the function that hunts for the egg and replaces it could be in the same program/binary as the one you're analyzing, or it could be a completely separate program, referred to as a loader, which downloads the binary you're analyzing and modifies it in memory.

If the egg hunting and replacing code is implemented in a separate program (a loader) and you don't have that program, this code just won't work, so don't be surprised if it seems to be broken!

SysWhispers3 with jumper-randomized method

The randomized jumper method looks a lot like the jumper method we analyzed first. In fact, it is exactly the same except instead of jumping into the middle of the NT system function that we wished to call, SysWhispers3 picks a random other NT system function to jump into instead. The randomization happens at runtime, which means that every time it runs, it could call a different function, possibly leading to your confusion if you're debugging it multiple times!

Never fear, however, because the pattern of this function should still be very familiar and recognizable to you:



```
sub_140003804 proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     [rsp+arg_0], rcx
mov     [rsp+arg_8], rdx
mov     [rsp+arg_10], r8
mov     [rsp+arg_18], r9
sub     rsp, 28h
mov     ecx, 3953937h
call    sub_1400014B0
mov     r11, rax
mov     ecx, 3953937h
call    sub_140001470
add     rsp, 28h
mov     rcx, [rsp+arg_0]
mov     rdx, [rsp+arg_8]
mov     r8, [rsp+arg_10]
mov     r9, [rsp+arg_18]
mov     r10, rcx
jmp     r11
sub_140003804 endp
```

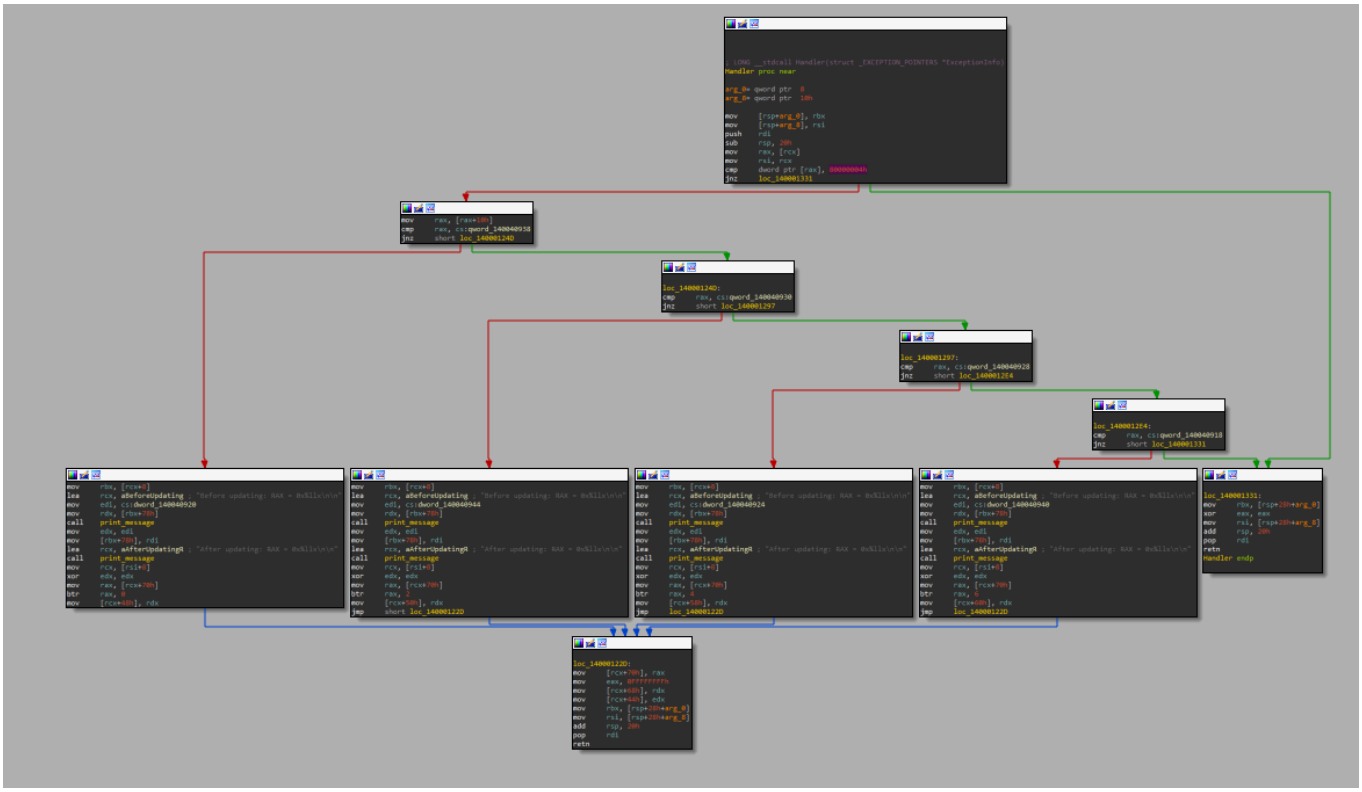
The first call instruction will return the address inside a randomly chosen function of ntdll, at a suitable place to jump to just after setting the SSN in rax and moving rcx to r10, but it will be a different address each time it runs.

The second call instruction will return the correct SSN for the function we wish to call (**NtAllocateVirtualMemory** in this example), and that's where you can set a breakpoint when debugging to figure out which system call is really being made.

The rest of this function with the saving registers and restoring them should be familiar to you, because it is the same as the other SysWhispers generated code.

Lab: SysWhispers Reverse Engineering

Find the lab sample named "SysWhispersShellcodeInjection.exe" and analyze it safely in your VM with IDA and x64dbg. Note: this is a sample written for this workshop. It will inject shellcode, but nothing dangerous or destructive or invasive to your privacy will happen if you run it. Answer the following questions, then check your answers in the SysWhispersShellcodeInjection.cpp source code file. Try answering via static or dynamic analysis before checking the source to reinforce your reverse engineering skills.



1. This sample won't run unless it finds another process with a particular name is running (Evasion 001). What is the name of the process it is looking for?
    1. Note: You can copy the "InjectionTargetDummy.exe" program (choose the 64-bit version) and rename it to be the name of the process that this sample is looking for, if you want to convince it to run.
2. The sample calls four system functions (all starting with "Nt") from ntdll using SysWhispers. They are labeled in the output as "Evasion 002", "Evasion 003", "Evasion 004", and "Evasion 005". What are the names of the system functions that it calls?

- Evasion002:
- Evasion003:
- Evasion004:
- Evasion005:

3. Bonus question: What method of SysWhispers does this sample use? (Choose one, multiple-choice): (A) Embedded, (B) Egg Hunter, (C) Jumper, (D) Jumper-Randomized
4. Bonus question: What process is spawned by the injected shellcode, if successful?

Now that we've explored how SysWhispers can avoid hooks, let's look at an even more evasive trick: using Vectored Exception Handlers to dynamically mutate a syscall mid-flight!

MutationGate and Vectored Exception Handlers

Now that we've examined SysWhispers up close, let's consider a newer and slightly sneakier way to evade detection and disguise which system call the malware really intends to use. This sample is based on an open source project called MutationGate and its associated technical article.

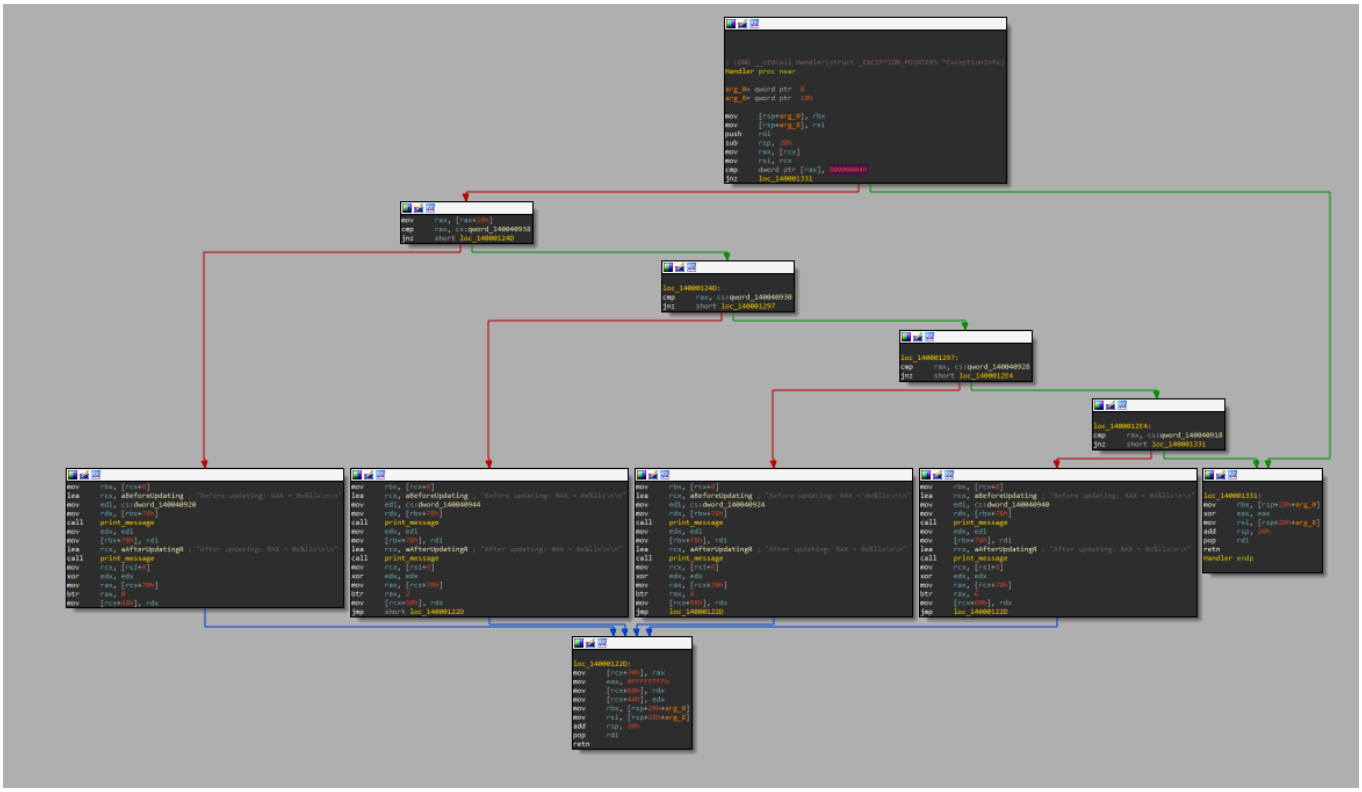
The short version of the technique is that we set a hardware breakpoint on an innocuous and innocent looking Windows API call that we guess EDR probably hasn't hooked, like **NtDrawText**. When we call **NtDrawText**, as soon as the instruction pointer hits the address where we set the hardware breakpoint, it will throw an exception. We have registered a Vectored Exception Handler that will handle the hardware breakpoint exception, so our function will get called. At that point, our exception handler code replaces the SSN value in rax, so that a different system function will be called instead of **NtDrawText** (for example, **NtOpenProcess**). One more point, if we want this diverted (or mutated) system call to work -- we have to have set up the correct arguments for **NtOpenProcess** when we called **NtDrawText**.

OK, so let's see how that code looks when we compile it into a program!

First, adding the vectored exception handler is pretty straightforward and obvious:

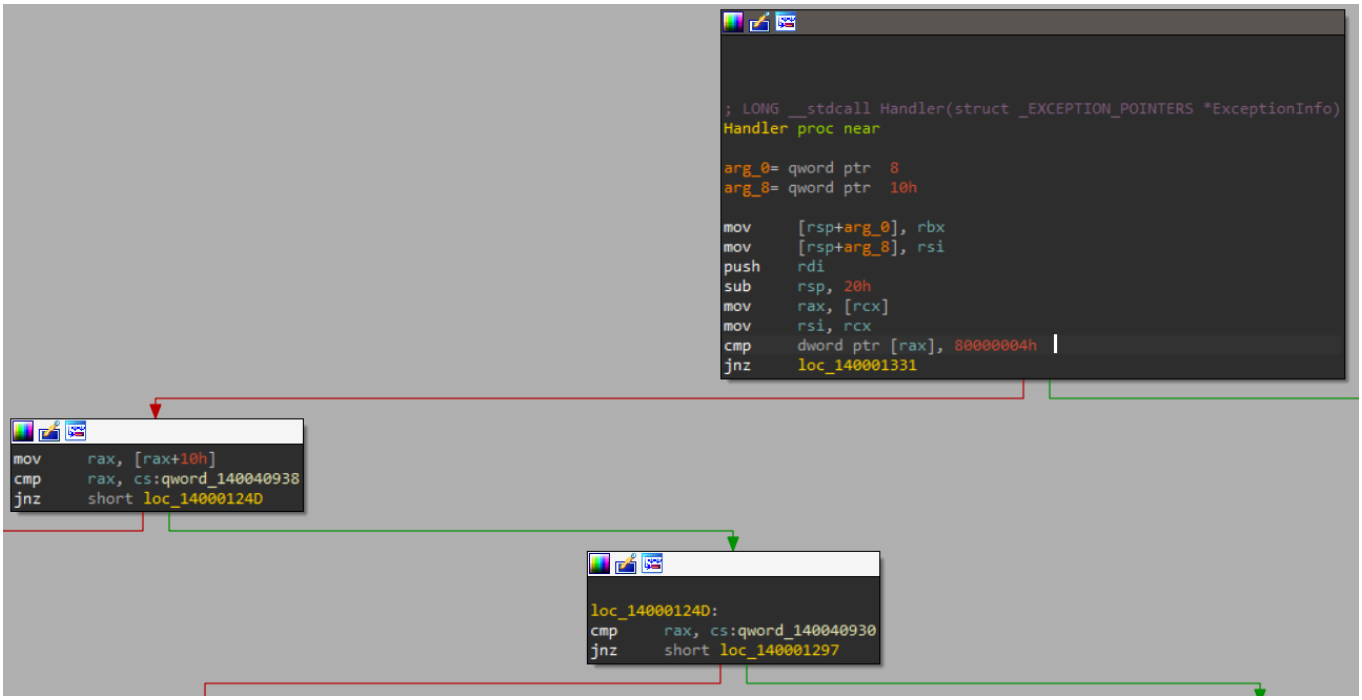
```
lea     rdx, Handler    ; Handler
mov     [rsp+7F0h+var_790], rax
mov     cs:qword_140040958, rax
lea     rcx, [rax+8]
mov     cs:qword_140040918, rcx
mov     ecx, 1          ; First
call    cs:AddVectoredExceptionHandler
lea     rdx, [rbp+6F0h+pe] ; lpContext
mov     dword ptr [rbp+6F0h+pe.szExeFile+4], 10001Fh
mov     rcx, 0FFFFFFFFFFFFFFFEh ; hThread
call    cs:GetThreadContext
test    eax, eax
jz      loc_140002A7F
```

The address that IDA named "Handler" is the function that will be called when the exception is hit. Let's look at that function, first with a high level overview of its structure:



Before even looking at the details, we can see that this exception handler basically has four main branches, each of which run very similar code with just a little bit of difference, and then some common cleanup code at the end.

There are four branches because this exception handler is called for hardware breakpoints and AMD64 processors have four hardware breakpoints, so it calls a different function depending on which address the breakpoint caused an exception. Now, let's take a closer look at the details:



Although IDA has analyzed this function as it it used standard calling convention, we know that it uses fastcall and the one and only argument, which is a pointer to the EXCEPTION_POINTERS structure, can be found in rcx.

Syntax

C++ Copy

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Members

`ExceptionRecord`

A pointer to an EXCEPTION_RECORD structure that contains a machine-independent description of the exception.

`ContextRecord`

A pointer to a CONTEXT structure that contains a processor-specific description of the state of the processor at the time of the exception.

In the instruction: ***mov, rax, [rcx]*** it dereferences the first item in the EXCEPTION_POINTERS struct, which is a pointer to an EXCEPTION_RECORD

Syntax

C++ Copy

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

The next instruction in our exception handler dereferences the value in rax, which points to the first item in the EXCEPTION_RECORD, which is the ExceptionCode. It compares that value with the immediate constant 80000004h, which is the constant for EXCEPTION_SINGLE_STEP (defined in minwinbase.h). That is the exception for a breakpoint.

If this exception was called due to a breakpoint, it will continue processing. If it was any other type of exception, it takes the path that will clean up and exit the function early.

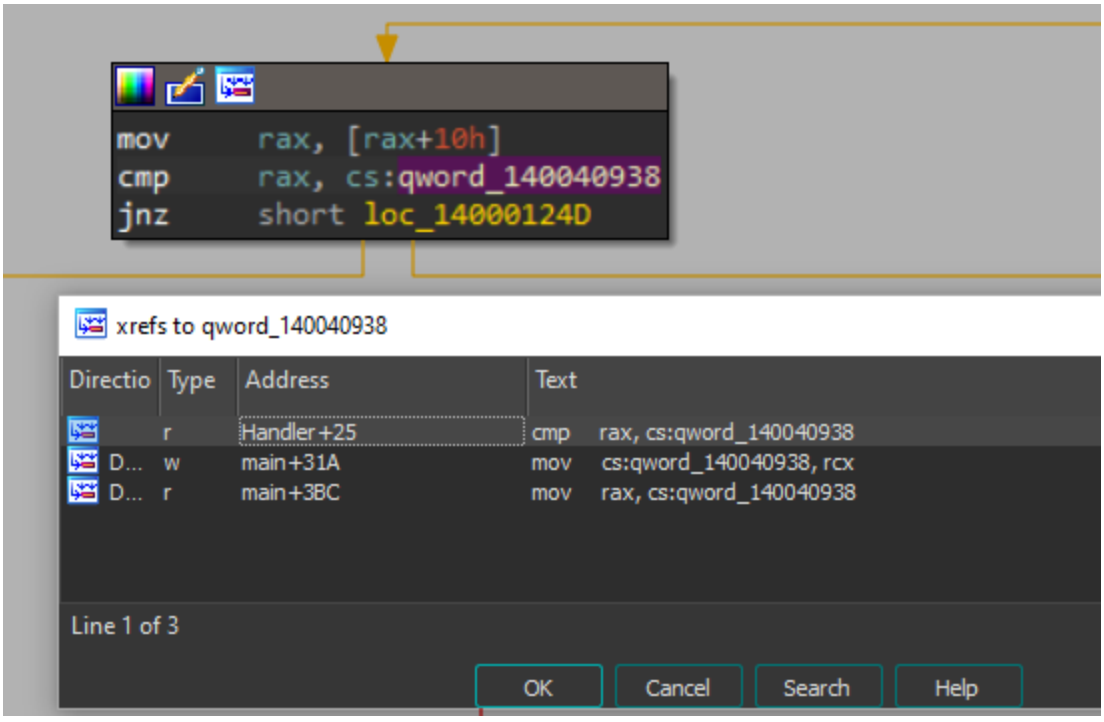
The next instruction is displayed in the next logical block of code. It takes the value in rax + 10h and moves it to rax. Remember that rax holds the pointer to the ExceptionRecord, so 0x10 bytes into it, is the ExceptionAddress. In the original C++ code, it looked like this:

```
LONG WINAPI exceptionHandler(PEXCEPTION_POINTERS exceptions)
{
    // Only handle single-step exceptions (caused by our hardware breakpoints)
    if (exceptions->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP)
    {
        // Get the address that triggered the exception
        PVOID exceptionAddr = exceptions->ExceptionRecord->ExceptionAddress;
```

We can see that the exception address is compared to four saved values, one at a time, to see if it matches any of them. The four saved values in this sample are labeled by IDA as:

1. qword_140040938
2. qword_140040930
3. qword_140040928
4. qword_140040918

   Let's check where those values were set by examining the cross references (xrefs) to those addresses. Use the 'x' keyboard shortcut or the right-click context menu in IDA while highlighting one of those values:



Let's look at the code at main+31A, the second xref:

```
mov     edx, 0A1920265h
mov     cs:dword_140040940, eax
mov     rcx, rdi
call    sub_140001350
mov     edx, 0DDCDED98h
mov     cs:qword_140040950, rax
mov     r14, rax
lea     rcx, [rax+8]
mov     cs:qword_140040938, rcx
mov     rcx, rdi
call    sub_140001350
mov     rcx, rdi
mov     cs:qword_140040960, rax
mov     r13, rax
lea     rdx, [rax+8]
mov     cs:qword_140040930, rdx
mov     edx, 0E6DEF784h
call    sub_140001350
mov     rcx, rdi
mov     cs:qword_140040948, rax
mov     r12, rax
lea     rdx, [rax+8]
mov     cs:qword_140040928, rdx
mov     edx, 0B735E5FCh
call    sub_140001350
lea     rdx, Handler    ; Handler
mov     [rsp+7F0h+var_790], rax
mov     cs:qword_140040958, rax
lea     rcx, [rax+8]
mov     cs:qword_140040918, rcx
mov     ecx, 1          ; First
call    cs:AddVectoredExceptionHandler
```

We that this code comes just before adding the vectored exception handler, and that all four of the qword values that we saw referenced from the Handler function are calculated and saved here. For each one, a function labeled "sub_140001350" is called with two arguments: the value from rdi is put into rcx (the first argument), and a different constant value is put into rdx (the second argument) each time just before the function "sub_140001350" is called. If we scroll up in the disassembly or use a debugger, we'll see that the first argument is the address of ntdll in memory. The list of the constants used as the second arguments are:

Constants:

1. 0A1920265h
2. 0DDCDED98h
3. 0E6DEF784h
4. 08735E5FCh

Note that whatever address is returned from the function is saved in a global variable in the code section (starting with cs:qword *followed by the address), and then that same value plus 8 is stored in the four cs:qword* variables that are used in the handler. So, we know that when the exception handler is called, it is checking to see if execution is paused at the address plus 8 of

something, and we can guess that the constant values are the hash of some function, and the sub_140001350 must be a function that finds the starting address of those functions in ntdll.

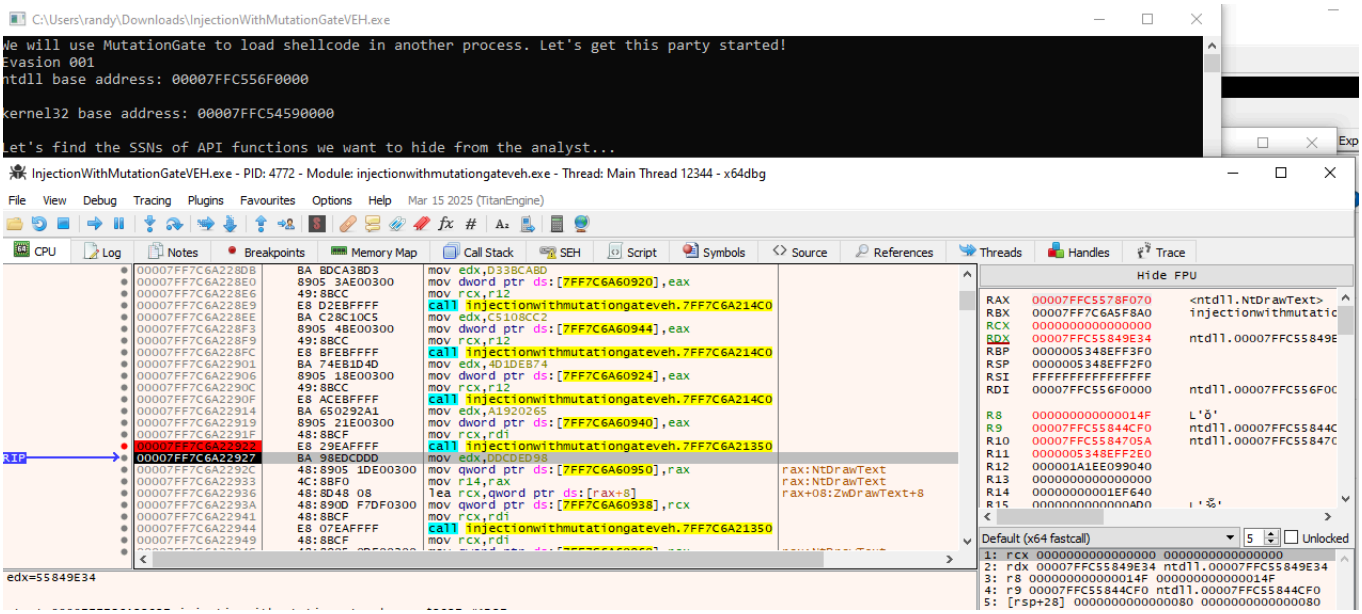
If you recall from the first section of this module, we saw that the Nt functions exported in ntdll have a certain pattern to them, and 8 bytes after the start of each function is the instruction to test a byte at address 0x7FFE0308h to see if it is 1 or not. This is the perfect place to set a breakpoint to change the SSN because it is after the SSN was set by the Nt function, after any EDR hooks, but just before the call to syscall or int 0x2E. Here is that screenshot from disassembling ntdll.dll again for reference. Notice that the function begins on address D7E0 and 8 bytes in is at D7E8:

```
.text:000000018009D7E0                 public ZwAllocateVirtualMemory
.text:000000018009D7E0 ZwAllocateVirtualMemory proc near       ; CODE XREF: RtlExtendMemoryZone+61↑p
.text:000000018009D7E0                                         ; LdrEnsureMrdataHeapExists+61↑p ...
.text:000000018009D7E0                 mov     r10, rcx        ; NtAllocateVirtualMemory
.text:000000018009D7E3                 mov     eax, 18h
.text:000000018009D7E8                 test    byte ptr ds:7FFE0308h, 1
.text:000000018009D7F0                 jnz     short loc_18009D7F5
.text:000000018009D7F2                 syscall                 ; Low latency system call
.text:000000018009D7F4                 retn
.text:000000018009D7F5 ; ---------------------------------------------------------------------------
.text:000000018009D7F5
.text:000000018009D7F5 loc_18009D7F5:                          ; CODE XREF: ZwAllocateVirtualMemory+10↑j
.text:000000018009D7F5                 int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.text:000000018009D7F5                                         ; DS:SI -> counted CR-terminated command string
.text:000000018009D7F7                 retn
.text:000000018009D7F7 ZwAllocateVirtualMemory endp
```

That is why MutationGate sets breakpoints at 8 bytes past the start of the Nt system functions that it wishes to hijack, right where the *test* instruction is called. Here is what the source code looks like:

```
// Get function pointers and their +0x8 offsets for our target functions to hijack
// (these are the benign functions that it will APPEAR that we're calling)
// NtDrawText (hash: 0xa1920265)
pNTDT = GetFuncByHash(ntdll, 0xa1920265);
pNTDTOffset_8 = (PVOID)((BYTE*)pNTDT + 0x8);     //Offset 0x8 from NtDrawText

// NtOpenMutant (hash: 0xddcded98)
pNTOM = GetFuncByHash(ntdll, 0xddcded98);
pNTOMOffset_8 = (PVOID)((BYTE*)pNTOM + 0x8);     //Offset 0x8 from NtOpenMutant

// NtPulseEvent (hash: 0xe6def784)
pNTPE = GetFuncByHash(ntdll, 0xe6def784);
pNTPEOffset_8 = (PVOID)((BYTE*)pNTPE + 0x8);     //Offset 0x8 from NtPulseEvent

// NtQueryEaFile (hash: 0xb735e5fc)
pNTQEF = GetFuncByHash(ntdll, 0xb735e5fc);
pNTQEFOffset_8 = (PVOID)((BYTE*)pNTQEF + 0x8);   //Offset 0x8 from NtQueryEaFile

// Declare our function pointers
fnNtOpenProcess pNtOpenProcess = (fnNtOpenProcess)pNTDT;
fnNtAllocateVirtualMemory pNtAllocVM = (fnNtAllocateVirtualMemory)pNTOM;
fnNtWriteVirtualMemory pNtWriteVM = (fnNtWriteVirtualMemory)pNTPE;
fnNtCreateThreadEx pNtCreateThreadEx = (fnNtCreateThreadEx)pNTQEF;

//Set up hardware breakpoints
HANDLE hExHandler = setupHardwareBp(ntdll);
```

If we run this program in a debugger (x64dbg) and set software breakpoints at the instructions in main() where these values are being set, we can see the addresses and tell which functions are being hijacked.



Just after calling the function ending in 1350 the first time, we can see the return value in rax is the address of **<ntdll.NtDrawText>**. This address and the address 8 bytes after it will both be

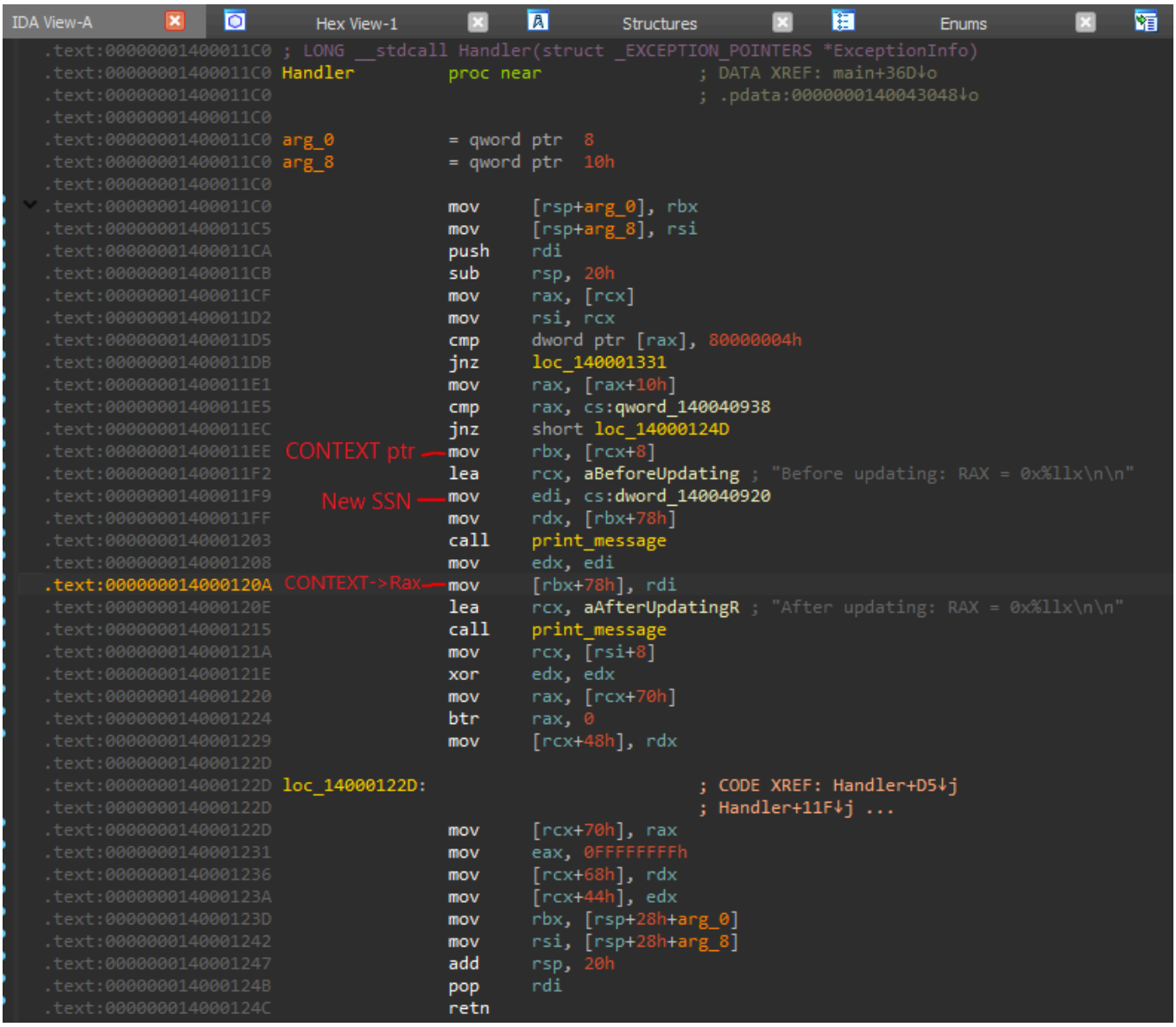
saved in global variables, and the address 8 bytes after the start of **NtDrawText** will be used when setting one of the hardware breakpoints.

We can repeat that process to find the other three target functions that are being hijacked, so we know that anywhere in the code that those functions are called, we expect a different function to actually be called. You can set a few more breakpoints in x64dbg or just single step over each instruction and see that these are the functions being targeted for replacement:

1. **NtDrawText
2. **NtOpenMutant** (which might show up in x64dbg as **ZwOpenMutant**, which is just an alias name for the same thing)
3. **NtPulseEvent** (also known as **ZwPulseEvent**)
4. **NtQueryEaFile** (also known as **ZwQueryEaFile**)

What is more important is to figure out the SSNs of the functions that are actually going to be called whenever each of these functions are called, so we can correctly translate the evasive behavior of this program.

To get that, let's set breakpoints in our exception handler and see what SSNs are being replaced. Let's take a close look at the exception handler in IDA and find the address to set a breakpoint:



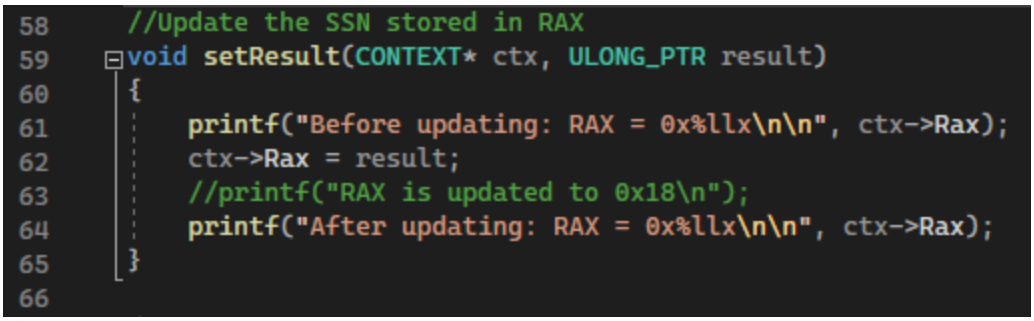
I've annotated this screenshot to show what is going on. It is helpful to understand that if an exception handler wishes to set the value of certain registers when the instruction pointer returns to the next instruction to execute, it needs to set those register values in the **CONTEXT** structure that is pointed to by the **EXCEPTION_POINTERS** struct passed in as an argument. The CONTEXT struct is defined in winnt.h. Here is a portion of that definition:

```
typedef struct DECLSPEC_ALIGN(16) DECLSPEC_NOINITALL _CONTEXT {

    //
    // Register parameter home addresses.
    //
    // N.B. These fields are for convience - they could be used to extend the
    //      context record in the future.
    //

    DWORD64 P1Home;
```

```
    DWORD64 P2Home;
    DWORD64 P3Home;
    DWORD64 P4Home;
    DWORD64 P5Home;
    DWORD64 P6Home;

    //
    // Control flags.
    //

    DWORD ContextFlags;
    DWORD MxCsr;

    //
    // Segment Registers and processor flags.
    //

    WORD   SegCs;
    WORD   SegDs;
    WORD   SegEs;
    WORD   SegFs;
    WORD   SegGs;
    WORD   SegSs;
    DWORD EFlags;

    //
    // Debug registers
    //

    DWORD64 Dr0;
    DWORD64 Dr1;
    DWORD64 Dr2;
    DWORD64 Dr3;
    DWORD64 Dr6;
    DWORD64 Dr7;

    //
    // Integer registers.
    //

    DWORD64 Rax;
    DWORD64 Rcx;
    DWORD64 Rdx;
    DWORD64 Rbx;
    DWORD64 Rsp;
    DWORD64 Rbp;
```

```
    DWORD64 Rsi;
    DWORD64 Rdi;
    DWORD64 R8;
    DWORD64 R9;
    DWORD64 R10;
    DWORD64 R11;
    DWORD64 R12;
    DWORD64 R13;
    DWORD64 R14;
    DWORD64 R15;

    //
    // Program counter.
    //

    DWORD64 Rip;
    // rest of header omitted for brevity, see winnt.h for full definition
```

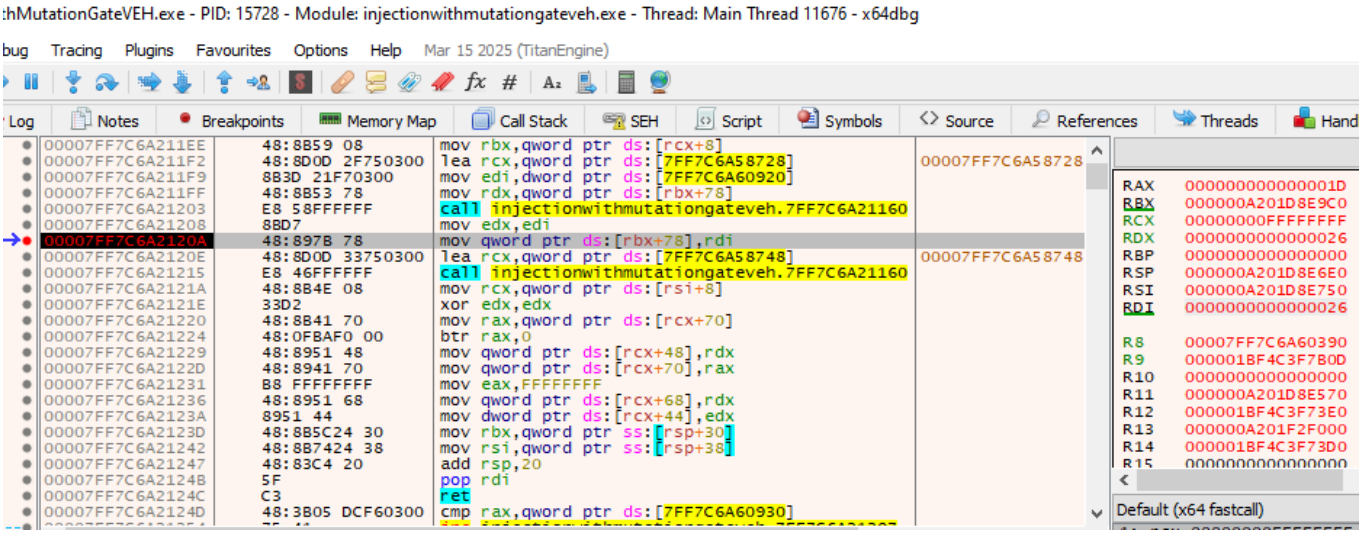
So, if we wish to set the value of the rax register from this exception handler, we need to set the value of the Rax member of the CONTEXT struct, which you can see in the exception handler source code here:



The Rax member of CONTEXT is 0x78h bytes from the start of the structure, which is why we see the instruction:

```
mov     [rbx+78h], rdi
```

Any time you are reverse engineering a sample that uses an exception handler, pay attention to the context that will be returned and any changes that are made to the context. Now let's set a breakpoint at 120A and see what the SSN value is set to in x64dbg when we run it:



The value in rdi is what will be stored in the CONTEXT->Rax member [rbx+78] and we can see that value is **0x26**. Let's look up that value in this table
Remember that the SSN used for different functions varies by OS version and build number. We can see that 0x26 in Windows XP, Server 2003, Vista, and Windows 7 means **NtAccessCheckAndAuditAlarm** but we didn't run this program on Windows XP or 7, so we need to keep looking. In Windows 8, build 8.0, SSN 0x26 means **NtMapViewOfSection**, but again that's not the OS we ran this on. In all builds of Windows 10 and 11, SSN 0x26 means **NtOpenProcess** and that is indeed the correct function in this case.

Now we know that any time we see this program calling **NtDrawText**, the function that will actually be called is **NtOpenProcess**! That's useful to understanding what this program does!

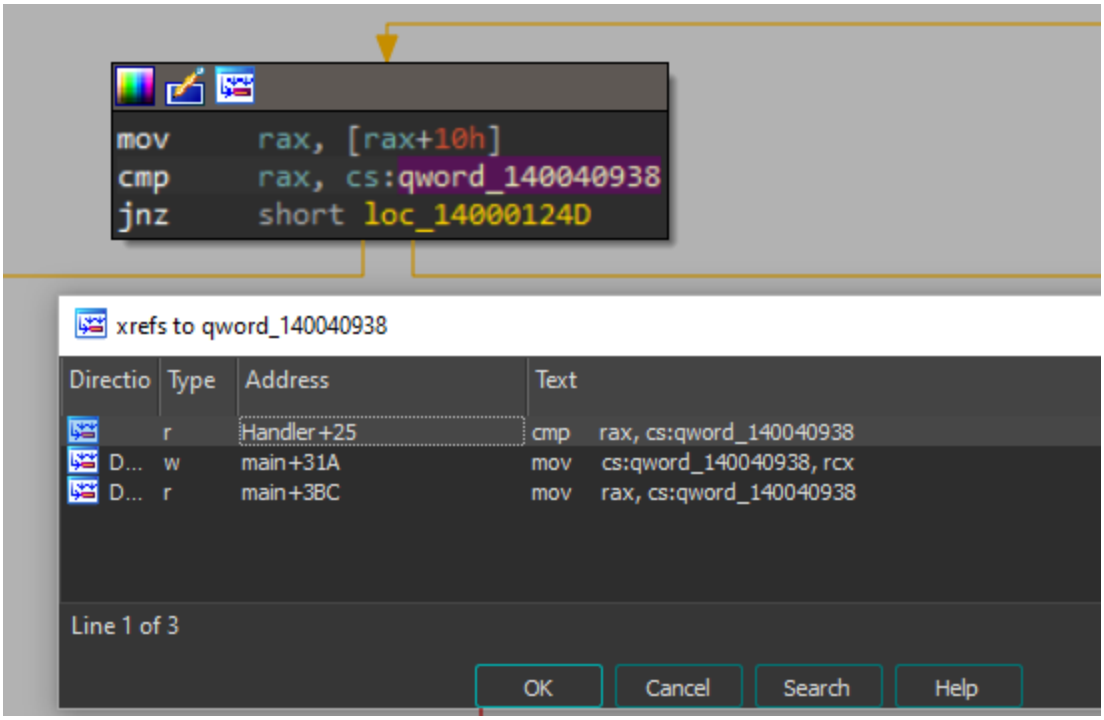
Notice that if the malware author is content with targeting only certain OS versions and builds, they could just hard-code the SSN values and not bother having the program search for the correct numbers at runtime. In this example program, we've implemented the most common approach, which is for the program to search in ntdll and possibly kernel32 as necessary to locate the correct SSN for each system function the program needs to call, using some hash of the function rather than its name.

Whether the values are hard-coded or dynamically resolved, as the reverse engineer you just need to understand the context of where the SSN will be used and know how to set a breakpoint at the right address to see what functions are being called. If the malware author went to so much trouble to hide their function calls, you can be sure that it is worth your time to figure out what they were trying to hide.

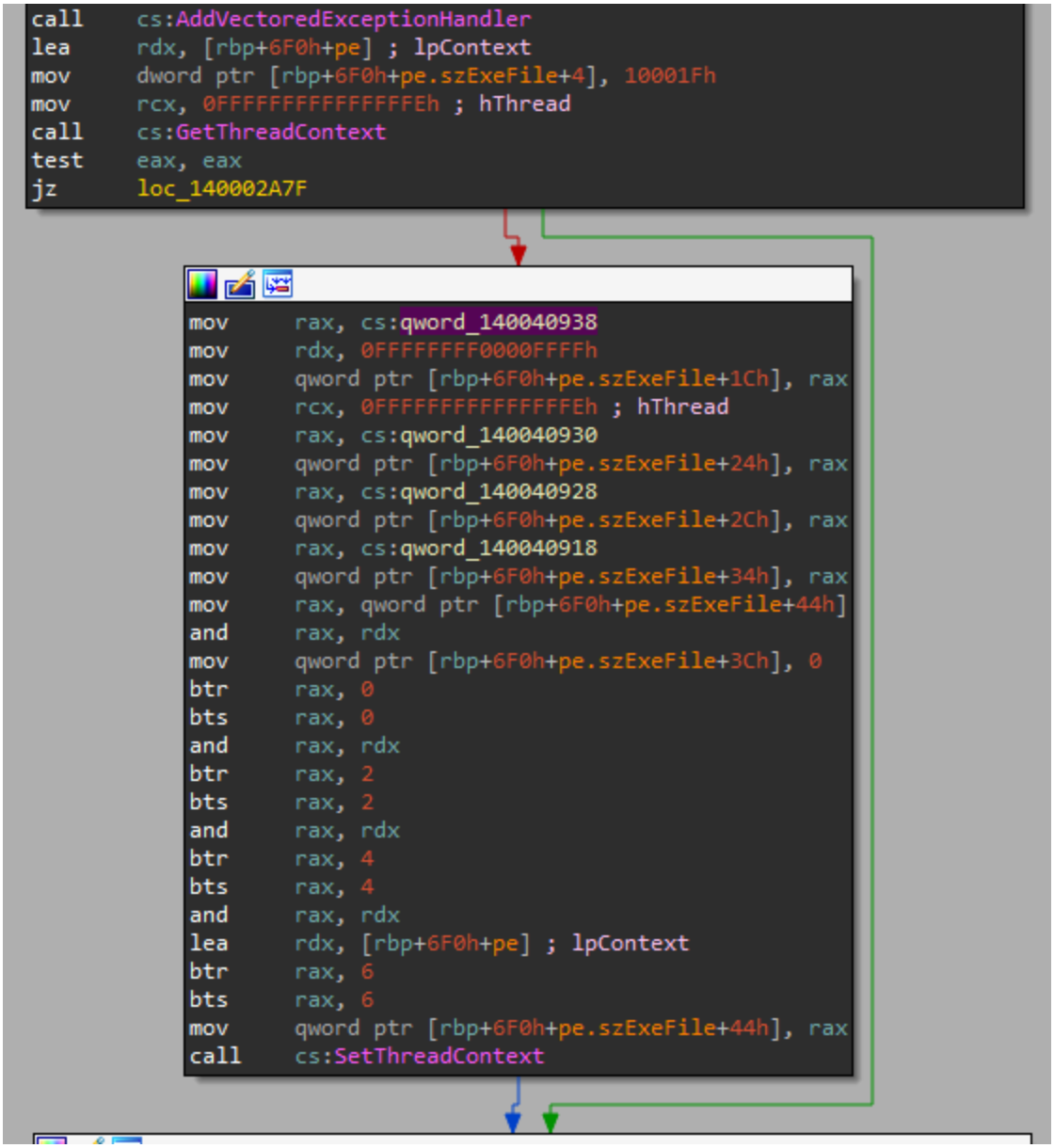
We'll save finding the other three function mappings as an exercise for the reader in the lab section below.

Let's take a quick look at how hardware breakpoints are set, so that you can recognize this pattern in the future. Remember we checked the xrefs to the global variable that held the

address of **NtDrawText** + 0x08 and found that the handler checked when it hit a breakpoint whether this was the current instruction pointer address, and we also found the place in the main() function where the program calculated the correct address and saved it in the global variable, so the one remaining xref that we haven't looked at yet (main+3BC) must be the place where the breakpoint it set, right? Let's go check it out and see!



Let's look at the code at main+3BCL



First notice the call to GetThreadContext. The first argument, in rcx, is the handle to the thread which in this case is the value -2 (or 0FFFFFFFFFFFFFFFEh as it is displayed here). That is an odd value to use as a Thread Handle, and it comes from [this AMSI bypass code gist on github](). I haven't yet found the documentation to explain why the value -2 is used here, but it's worth remembering in case you see some other malware that borrowed this code to set hardware breakpoints.

In any case, the important output from **GetThreadContext** is a pointer to the **CONTEXT** structure, which we've seen before can be used to set register values. We can see this code takes the four addresses of the target system functions that we want to hijack from the global variables where they were saved:

1. qword_140040938
2. qword_140040930
3. qword_140040928

4. qword_140040918

For each one, it moves the value (which is the address + 0x08 offset from the target function start) and then saves each address to a different member of the CONTEXT structure, at offsets:

1. 0x1C
2. 0x24
3. 0x2C
4. 0x34

I'll save you the trouble of counting offsets in the CONTEXT structure and tell you that those correspond to these members:

1. Dr0
2. Dr1
3. Dr2
4. Dr3

(You can check in the snippet of the CONTEXT structure definition above if you wish)

Registers Dr0 through Dr3 are the Debug registers used to hold the four hardware breakpoints. The code also sets Dr6 (which is at offset 0x3C) to 0, and sets Dr7 (at offset 0x44) to this very complicated-looking bit pattern that is built in rax with a bunch of instructions, but all it really does is set the bit to enable each hardware breakpoint. It first sets bits 16-31 to 0, then sets individual bits to 1 for each breakpoint. For breakpoint 0 (Dr0), it sets the bit 0 in Dr7 to 1. For breakpoint 1 (Dr1), it sets the bit 2 in Dr7 to 1. For breakpoint 2 (Dr2), it sets the bit 4 in Dr7 to 1, and for breakpoint 3 (Dr3) it sets the bit 6 in Dr7 to 1.

It's a lot simpler looking in the C++ code, which you can also see in the github gist.

```
void enableBreakpoint(CONTEXT& ctx, PVOID address, int index) {

      switch (index) {
          case 0:
              ctx.Dr0 = (ULONG_PTR)address;
              break;
          case 1:
              ctx.Dr1 = (ULONG_PTR)address;
              break;
          case 2:
              ctx.Dr2 = (ULONG_PTR)address;
              break;
          case 3:
```

```
                ctx.Dr3 = (ULONG_PTR)address;
                break;
        }

        //Set bits 16-31 as 0, which sets
        //DR0-DR3 HBP's for execute HBP
        ctx.Dr7 = setBits(ctx.Dr7, 16, 16, 0);

        //Set DRx HBP as enabled for local mode
        ctx.Dr7 = setBits(ctx.Dr7, (index * 2), 1, 1);
        ctx.Dr6 = 0;
}
```

If you can just remember the pattern of **GetThreadContext** and then setting values at any one or more of these offsets from the CONTEXT structure: 0x1C, 0x24, 0x2C, 0x34 to addresses of instructions, you can figure out that the code is setting hardware breakpoints and follow the clues to figure out what it is doing at those addresses.

As the last step, the code calls **SetThreadContext** to apply the changes that it made in the CONTEXT structure to the registers themselves.

Lab: MutationGate Reverse Engineering

Analyze the sample called *InjectionWithMutationGateVEH* with IDA and x64dbg. Set the breakpoints in x64dbg as we did in the steps above to help you get started, then answer these questions:

1. We already found out that when **NtDrawText** is called in the program, it actually calls **NtOpenProcess** by replacing the SSN with 0x26. Find the mappings between the rest of the hijacked functions to the SSN that they use as a replacement to call the real function by setting breakpoints inside the exception handler to capture the SSN values being swapped in each case. Note: The output of the program will make this easy, but don't work backwards from the console output, instead set breakpoints and check the actual SSN value for each one:
    1. **NtDrawText**: SSN is replaced by 0x26 (which is for **NtOpenProcess**)
    2. **NtOpenMutant** (also known as **ZwOpenMutant**): SSN is replaced by ??? (which is for ??)
    3. **NtPulseEvent** (also known as **ZwPulseEvent**): SSN is replaced by ?? (which is for ??)
    4. **NtQueryEaFile** (also known as **ZwQueryEaFile**) SSN is replaced by ?? (which is for ??)

2. The shellcode that gets injected is encrypted in the file and gets decrypted at runtime. What is the password used to decrypt the shellcode?
3. (bonus) What is the encryption algorithm used to encrypt/decrypt the shellcode? (This is not a trick question, it is exactly what it appears to be)