Assembly Review

To be provided by another instructor.

Windows Internals Crash Course

Windows Internals

Windows internals is a large and complex body of knowledge. We'll review the basics of this complex subject but, it can be an active area of practical research, technique development, and consistent learning for even the most advanced cyber security practitioners.

User Mode vs. Kernel Mode

The Windows operating system is split into two distinct modes, User Mode and Kernel Mode. Most people and developers will remain in User Mode and have no need to directly interact with the kernel. The programs that you most commonly interact with are in User Mode. Think of your browsers, the calcualator, word processors, Integrated Development Environments (IDEs), etc; these are all in User Mode.

Kernel Mode is the most privileged area of the operating system and handle all of the complex processing that most users do not want to think about. The kernel handles memory management, file access, interacting with hardware, execution scheduling, and more. The Windows kernel is located in the NTOSKRNL.exe file. This is loaded shortly after the computer powers on while the system is completing the boot process.

These two modes are implemented as two sides of a strict security boundary to protect users from malicious capabilities and sometimes even themeselves. Learning about both will give a better picture of how a cyber security practitioner can both attack and defend this operating system.

Process, Threads, and Scheduling

One of the foundational components of the Windows operating system is the concept of processes and threads. A process is a logical collection of information, including executable code, that comprises a running program. This process has large amounts of information that allow the operating system to provide the resources needed to execute the program. Within each process, there are one, or many, threads executing. Each thread also contains a lot of information about the code being executed but is isolated to only that thread. Each thread only knows about itself and does not worry about other threads running within the same process.

The kernel uses a "scheduler" to start, pause, and stop executing threads. When a thread starts, it's given a specific amount of time to execute before

the operating system pauses it, saves off information, and let's another thread execute. This scheduling gives the illusion of a system that can run multiple programs simultaneously. While modern hardware may have multiple cores, which will allow for true multiprocessing, even a single core CPU can give the illusion of the operating system running multiple programs at the same time simply by efficiently executing, pausing, and running a sequence of threads.

Memory Management

Computer memory is foundational to the computer as a system, not just an operating system or software. Any computer must have space to store its information. Executable code and data are all stored in memory so the computer can execute instructions with data. While simple in concept, the impelemntation can become quite complex when considering physical memory and virtual memory in a modern operating system. Luckily, the kernel handles this for us and when a program needs memory for its operation, it can request it from the operating system and the operating system will prepare the memory for usage by the program.

Since there has been a rise in exploitation attempts against the Windows operating system overtime, Microsoft, along with its partners, have implemented memory security protections including DEP, ASLR, which are critical to understand for binary exploitation on the Windows operating system.

Windows Registry

One more critical component of the Windows operating system is the Windows Registry. This is used throughout the operating system for a variety of use cases including system settings, process settings, display settings, security settings, enterprise configurations, forensics, and even more. It is also used by both User Mode and Kernel mode applications.

Windows API Calls

Combining all of these ideas together, users and developers need a safe way to interact with the kernel in order to run programs using hardware, graphics, memory, etc. This brings us to the Windows API. Each program calls exported functions by a variety of dynamically linked library (.DLL) files in order to securely interact with the operating system. There are DLLs that are used in every program, some that are linked dynamically when the program is loaded by the operating system, and some that a developer can dynamically load themselves for unique purposes. These allow for users and developers to correctly and securely use the operating system to its full potential and arrive at the user experience that we are all familiar with on the Windows operating system.

Some examples of functions include VirtualAlloc, CreateFile, CreateProcess, LoadLibrary, GetProcAddress, and so many more. As you learn more, you'll even learn that these are abstracted away from the system calls that actually

cause the low-level interrupt and mode change between User Mode and Kernel Mode. We will talk more about system calls later in this course.

PE File vs. Shellcode

PE File

The Portable Executable (PE) file structure is well-documented and understanding this structured format for a program is critical to understanding how the Windows operating system works. One of the best resources to learn about Windows internals is the documentation provided by Microsoft called MSDN.

Format

Microsoft published a large page on the PE format located [[https://learn.microsoft.com/en-us/windows/win32/debug/pe-format]]. We'll walk through some of the sections and components of the format. The first three major sections of the PE file are the DOS Header, the COFF/PE Header, and the Optional Header.

DOS Header This is a small header at the beginning of every PE file. You'll likely recognize it from the "MZ" characters or the "This program cannot be run in DOS mode" string that exists in this header. At location 0x3C in the DOS header, one can find the offset to the PE Header.

PE Header The PE Header contains much more information that Windows uses when loading the program into memory. It contains information like the architecture of the binary, the number of sections with in the PE file, and characteristics of the PE File which include a flag to determine if the PE file is an executable (.EXE) or a dynamically linked library (.DLL).

Optional Header The Optional Header is not optional as it contains critical and valuable information about the PE file used by the Windows operating system to load the program. Fields like the size and location of important sections within the PE file, the offset of the Original Entry Point (OEP), process values such as the default size of the stack, a checksum, and even more data.

Data Directories

The data directories will be especially useful to us in this course as we will routinely interact with the Export Address Table (EAT) in order to dynamically resolve virtual addresses of functions in memory so we can use them with our shellcode. Other data directories exist such as the Import Table, Resource Table, the Import Address Table, and the base relocation table. There are many more data directories that deserve their own time for reserach and understanding that will not be provided in this course.

PEB

Now that we have a basic understanding of Windows internals, let's move towards our objective of understanding and writing shellcode. This starts with understanding the Process Environment Block (PEB) structure. Microsoft has published a version of this structure on MSDN, [[https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb]], but we can get an even more detailed one by using a debugging tool, such as WinDBG, to download the symbols from microsoft and view them within the debugger. We will see this later in the course when we get to analyzing shellcode.

There are also many other blogs and posts that talk about the PEB structure and individual practitioners have documented fields that microsoft has not made public. Given that this structure is not guaranteed to be the same across operating system versions, it's important to understand this structure and how and where it's used as part of the Windows operating system and the execution and maintenance of a process object.

For shellcode, we will need to "walk the PEB" or "traverse the PEB" in order to find exported functions located in other DLLs that are loaded into memory. The reason we need to walk the PEB for this is that each process keeps lists of DLLs that are loaded into memory for each process. Each of these lists hold the same information but the ordering is different and can be used to our advantage when locating specific functions within specific DLLs we know are guaranteed to exist.

First, we need to know how to find the PEB. The way the Windows operating system is implemented, Windows places a critical structure in the `fs` register. This data structure is called the Thread Environment Block (TEB). It contains information necessary for the execution of the thread. One of the members of the TEB is a pointer to the PEB. For 32-bit, it's offset 0x30. We can access it like this:

```
mov EAX, fs:[0x30]
```

The `EAX` register now contains a pointer to the PEB. We can now use this in our shellcode to start walking the PEB.

We first look at the Ldr member of the PEB. This member is of type PPEB_LDR_DATA, or in english, a pointer to a PEB_LDR_DATA structure. This structure is also semi-documented on MSDN, [[https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data]]. We can only see here the InMemoryOrderModuleList. This is one of the lists that contains a list of the loaded DLLs in memory. There are actually two other lists that are part of the PVOID Reserved2[3] member. These are called InLoadOrderList and InInitializationOrderList. You may also see it with "Link" instead of "List" for the named members. These pointers point to a doubly linked list of LDR_DATA_TABLE_ENTRY structs. This struct is also documented on the same page as the PEB_LDR_DATA structure.

The LDR_DATA_TABLE_ENTRY contains lots of valuable information about each DLL that is loaded into memory including the base address and the DLL Name. We will use this information to functionally implement the LoadLibrary Windows API call. The LoadLibrary API call takes in a file name or path to a DLL and loads it into memory for the program to use. Often this is the virtual address of the DLL image in memory and specifically for our implementation it will definitely be the base address of the DLL.

From this point, we need to implement our own version of GetProcAddress. The GetProcAddress API call takes in a DLL and a string that is the function name to resolve and returns the virtual address of the function. So for example, calling GetProcAddress(pKernel32, "MessageBoxA"), where pKernel32 is the base address of Kernel32.dll in memory, it will reutrn the address of the function MessageBoxA so it can be used within the program.

To do this, we will walk the PEB to find the base address of the DLL we need to resolve a function from. Once we have that address, we walk the PE File using our knowledge of the structure to find the Export Address Table (EAT). Each member of the EAT contains a pointer to the name of the function. We can iterate through the table looking for the function name that matches the function name we are looking for. Once we find it, we use that index to look up the ordinal of the function being exported. The ordinal is the index in the function address component of the EAT member which will allow use to compute the function address of our desired routine. We can now call this address for this function like we would with any other function since it has been resolved.

GetProcAddress Hashing Techniques

Expanding on our LoadLibrary and GetProcAddress implementation, we need to consider the ability for Anti-virus and security protection mechanisms to hinder our shellcode. Including the string "CreateProcessA" is much more suspicious than a random series of bytes which could mean the same thing. In fact, we can use this idea to implement our own hashing algorithm to obfuscate the functions we are interested in and still resolve it correctly.

We will actually practice this in an upcoming lab! The idea goes as follows. If we take our function name string and apply a hashing function to it to produce a unique number then when we are resolving functions we can apply the same hashing function to each function string we are searching through. If we find a function where the hashes match, we have found our function, and no strings are present in our shellcode.

Get Program Counter (PC) Techniques

One last thing before we go to our lab, we need to talk about getting the Program Counter, or instruction pointer, value. Basically, how do we know where we are executing in memory? There is a common trick using the CALL/POP

instructions to locate oneself in memory. The way it works is this. When the shellcode starts, the first instruction is a CALL instruction to the next instruction. As part of the CALL instruction, the return address is pushed onto the stack. The return address in this case is the next instruction so the address of current instruction is located on the stack. We can then pop this value from the stack and we have our current instruction address in a register to be used as we need for our shellcode.

```
call get_pc
get_pc:
  pop eax  ; EAX now has our current instruction address
```

Hashing Technique Demo

Now we're going to use the provided WinApiHash.py script that has a simple Windows API function name hashing algorithm implemented within it. It also takes a key to make the hash more unique. The key can be changed as part of the shellcode to prevent null bytes, signatures, bad characters without even changing the algorithm. If it comes to it, the algorithm could be switched up as well by modifying the hashing operations done to compute it. Feel free to experiement with the code and generate hashes.

Keep in mind, this hashing also will have to be implemented in shellcode to resolve to the same value, so making it too complicated can have negative effects on the implementation of your shellcode.

Now it's your turn, give some of these a try and use MSDN to look up other functions to play around with the API hashing. Try:

- Key: 0xabcd1234, Name: VirtualAlloc
- Key: 0xabcd1235, Name: VirtualAlloc
- Key: 0x11111111, Name: CreateFileA
- Key: 0x1F1F1F1F, Name: LoadLibraryW

Rolling Your Own Shellcode

Removing bad characters

Substituting instructions

One of the most common substitutions is for setting values equal to zero. You can absolutely use `MOV EAX, 0x0` and this will assign the EAX register to be zero. However, this will contain at least one null byte because of the constant 0x0 being present in the instruction.

A common trick to avoid this is to XOR a register with itself, `XOR EAX, EAX`. Any value XOR'd with itself is 0; the XOR operation just has this mathematical property.

It natually follows from this that any small value that you need can be created by XOR-ing a register and then incrementing the register until you get the value you need. However, at some point, there is a tradeoff between the number of `INC` calls you need and the actual value you need.

Again, for small values, you can use `LEA` as another tactic. `LEA` stands for Load Effective Address. It has some weird usage, but any single byte value can be used like the following to get a byte into a register.

```
XOR EAX, EAX
LEA EAX, [EAX+0x45]
```

This will place the byte 0x45 into the EAX register. This works because we zero out the EAX register and then we add 0x45 to EAX, which is zero, to create an "address" of 0x45. We then move that value into EAX for storage.

This leads into our next subsection, arithmetic construction of values.

Arithmetic construction

Building on our previous example, we can repeat that LEA technique to fill in a register using `LEA` and the `SHL` instruction. The `SHL` instruction shifts a value to the left by the number of bits specified as an argument. Given that each DWORD is 32bits, and each byte takes up 8 bits. We can create any DWORD following this trick.

```
XOR EAX, EAX
LEA EAX, [EAX+0x12]
SHL EAX, 8
LEA EAX, [EAX+0x34]
SHL EAX, 8
LEA EAX, [EAX+0x56]
SHL EAX, 8
LEA EAX, [EAX+0x78]
```

This will place the value, 0x12345678 into EAX with no null bytes.

We can also use AND, OR, NOT, NEG, and XOR to create values in any register without using null bytes. We can also use math operations like ADD, SUB, MUL, and DIV to get us the numbers we need too. This requires a bit of pre-computation before putting the values in your shellcode, but once you have them the operation will run correctly and produce your value. Let's say we need a value that has a null byte in it, we can use math to create it.

Let's create 'cmd.exe\0' using math. Unfortunately we can see that this will require a null byte. Since those are not allowed for this example, we need a way to compute it instead of passing it in.

'cmd.' is 0x63 0x6d 0x64 0x2e in hexadecimal and in little endian we get the hex constant 0x2e646d63. Let's use XOR to compute this. We can get

0xFFFFFFFF from `XOR EAX, EAX` followed by a `DEC EAX`. Then we can use a constant, 0xd19b929c, which also has no null bytes. When we XOR -1 and our new constant, we get our desired hex value.

Now we need our hex constant that has a null value, 0x00657865. We can take advantage of knowing that the opposite (or NEG) of 0x00 is 0xFF. We can then use 0xFF9A879A, which is null free, and call the NEG instruction on the register that holds it to compute our value.

Lastly, we will want these values on the stack so we can use the string as an argument to a function call. An example might look something like this:

```
MOV EAX, 0xFF9A879A
NEG EAX
PUSH EAX
XOR EAX, EAX
DEC EAX
XOR EAX, 0xd19b929c
PUSH EAX
```

The possibilities to compute values are unlimited and rely only on your creativity and the restricted characters for your shellcode.

Using encoders

Lastly, we come to encoders. This is the more progammatic way to solve the issue. What if we had a stub of code that could just decode all of our code? We could give it a key and that key would allow the encoder to do all of the decoding in memory. We wouldn't need to manually compute values or instructions, we simply write our code and then encoded it and prepend a stub that decodes it to the shellcode.

Encoders can be a little tricky as they are often targeted by Antivirus companies because they themselves cannot be encoded. This means they are a prime target for signature based detection.

One of the most famous encoders when first learning about shellcode is the shikata ga nai encoder included in Metasploit. This is polymorphic XOR additive feedback encoder. In other words, it uses XOR and addition of values to compute the key which feeds into the decoding process. This makes it difficult to decrypt with a static key but once the algorithm is known by an analyst it can be trivially decoded.

For this course, we'll do a simple XOR encoder to demonstrate the technique. The idea is that every byte in the payload will be XOR encoded with a specific value. This will obfuscate the bytes and potentially eliminate any null bytes. However, there is a chance that the byte used as the key is present in the shellcode and thus would make a new null byte. This problem can be solved with multiple encodings or selecting a key that is not used anywhere in the shellcode.

First, we're going to need a python script to do the XOR encoding for us and we'll have it select a key that's not used in the shellcode (if possible). Then we'll write our XOR decoder stub. Finally, we'll put it all together and have it take in our shellcode and spit out an encoded shellcode for us.

Use the XOREncode_skeleton.py and the xor_decode.asm files to complete this activity. You will need to fill in the missing functionality for the python script and assemble the xor decoder in order to get the bytes to prepend to the shellcode.

Creating Multi-WinAPI Shellcode

Looking up Windows APIs and Structures

The Windows API is notoriously large and complex. It's very unlikely anyone will memorize or learn the entire API, which is likely why Microsoft publishes all of the docs on MSDN. You can search on MSDN for the function you are looking for and see a large amount of information about the functions, structures, and often even a usage example of how to correctly use the call.

Let's walk through an example to illustrate the point. We'll look at the `CreateFileW`. When we first go to MSDN, we can search for our function that we want to learn about. We'll search for `CreateFileW` and navigate to the page with information about the function, here.

We can immediately see a summary of the function, it's purpose, what it does, and even the syntax for the function definition. If we continue scrolling down, we can learn more about each parameter, default values, and what kind of information is expected for each parameter.

For example, we can see that dwDesiredAccess and dwShareMode require specific values from definitions within the operating system. We can navigate to any of those pages to learn more about "Generic Access Rights". We can also see a short list of possible values for dwShareMode based on how we want the file to be shared that we are creating.

Not all parameters are required, which can be very nice while writing shellcode as it means a `PUSH 0x0` instead of setting up a complex structure. The lpSecurityAttributes are an optional parameter and if we aren't worried about them for our file, then we can just skip it and use 0 instead.

Finally, we scroll down to the return value. This can give us more information about what's expected from a successful and unsuccessful call to this function. `CreateFileW` returns a handle to the opened file. This handle "represents" the file to the operating system but is not the file itself. Other Windows APIs will take this handle and perform actions on the file only with that handle as opposed to the file name, the full path name, etc.

CreateFileW is unique in these docs as the "Remarks" section has lots of additional content to understand what is going on with files. `CreateFileW` is a very complex call and can handle many resources, paths, and types of files. Any developer could be accessing any of those resources and could look at this page to better understand how `CreateFileW` handles those cases. While it's not always necessary to read everything on these pages, these sections and ultimately these docs, are excellent resources for learning more about Windows internals.

Lastly, if you ever need it, there are examples towards the bottom of the docs. Not every function always has examples, but `CreateFileW` does and many other do as well.

Moving on, let's look at a structure, `PROCESS_INFORMATION`, here.

This structure will be used in an upcoming lab and we can see that Microsoft documents this structure for us as well. It contains the syntax for the struct, the data types, and names, and even pointer aliases for the structure.

Each of the members of the structure are summarized on the page. Lastly, we can see in the "Remarks" section a good tip. Make sure to clean up handles when you are done using them. It is reassuring the system will eventually clean them all up, but not until the parent process is terminated, per the documentation. This can be useful information for resource management on the operating system; these details can sometimes be the difference on being caught by AV/EDR or not!

Shellcode Using Windows Syscalls

Background

In general, most guidance you will read about using Windows system calls is that you shouldn't. There is a strong reason for that, and that's because Microsoft makes no guarantees that these system call numbers, or System Service Numbers (SSNs) will not change between versions. Instead, Microsoft exposes high level wrappers, like what most developers use and what we've been dynamically resolving in our shellcode, to make sure backwards compatibility is not compromised.

This is great from a building project perspective as developers can be sure that their `CreateFile` function will never change and they can use it across versions. There is no such guarantee about `NtCreateFile`. `NtCreateFile` is the exposed API by ntdll.dll that explicitly calls the correct SSN and causes the switch from user to kernel-mode.

This technique goes even one step further and actually loads the arguments in the correct registers or on the stack and calls the SSN itself by placing it into the EAX register and calling the `SYSCALL/SYSENTER` (or `INT 0x2E`) instruction.

Now as we've established, this is less than stellar for backwards compatibility but, one of the biggest hinderances for running shellcode is the presence of Anti-Virus. AV will use all sorts of detection mechanisms and hooks in the code to attempt to catch and block your code. By calling the SSN yourself, you guarantee that at no point between the `CreateFile` and `NtCreateFile` was a hook used to capture what you are doing. The reason we can be sure is that you didn't call any of those wrappers in your shellcode and therefore no hook was ever triggered either.

In addition, this technique bypasses the need for resolving functions and libraries because the code is implemented in your shellcode. However, the trade-off here is that a lot of upfront work was needed to make sure you have all the calls, arguments, and parameters setup correctly before you made that call. In some cases that upfront work is an excellent trade-off for secrecy and effectiveness.

For this course, we look at what these code snippets look like and how we can implement them. We will also look at SSN resolving using ShellWasp to speed up the development process.

Nt* Function Example

For this example, we'll open up an application in Windbg and look at the one of the Nt* exports for ntdll.dll. All of the available Nt* functions are exported out of ntdll.dll. Since ntdll.dll is always loaded into process memory, we can load any program and get access to information about it with Windbg.

We'll start with NtCreateFile on 64-bit. We can look it up in Windbg and we'll see something like:

```
0:000> x /D /f ntdll!NtCreateFile

00007ffc`3d282590 ntdll!NtCreateFile (NtCreateFile)
0:000> uf 00007ffc`3d282590
ntdll!NtCreateFile:
00007ffc`3d282590 4c8bd1          mov     r10,rcx
00007ffc`3d282593 b855000000      mov     eax,55h
00007ffc`3d282598 f604250803fe7f01 test    byte ptr
    [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`3d2825a0 7503            jne     ntdll!NtCreateFile
    +0x15 (00007ffc`3d2825a5)

ntdll!NtCreateFile+0x12:
00007ffc`3d2825a2 0f05            syscall
00007ffc`3d2825a4 c3              ret

ntdll!NtCreateFile+0x15:
00007ffc`3d2825a5 cd2e            int     2Eh
00007ffc`3d2825a7 c3              ret
```

We can see here that the the SSN is 0x55. The value that is placed into the *AX register will be used as the SSN value when it transitions from user-mode to kernel-mode. Then it calls the syscall function to make the transition.

Let's look at another one to compare and contract. We'll look at NtAccessCheck next.

```
0:000> x /D /f ntdll!NtAccessCheck

00007ffc`3d281af0 ntdll!NtAccessCheck (NtAccessCheck)
0:000> uf 00007ffc`3d281af0
ntdll!NtAccessCheck:
00007ffc`3d281af0 4c8bd1          mov     r10,rcx
00007ffc`3d281af3 b800000000      mov     eax,0
00007ffc`3d281af8 f604250803fe7f01 test    byte ptr
    [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`3d281b00 7503            jne     ntdll!NtAccessCheck
    +0x15 (00007ffc`3d281b05)  Branch

ntdll!NtAccessCheck+0x12:
00007ffc`3d281b02 0f05            syscall
00007ffc`3d281b04 c3              ret

ntdll!NtAccessCheck+0x15:
00007ffc`3d281b05 cd2e            int     2Eh
00007ffc`3d281b07 c3              ret
```

Here we can actually see that NtAccessCheck has a SSN of 0; it's the first one! However, the structure remains largely the same. By this time in the call stack, all of the parameters are where they need to be and the only thing left is to use the unique SSN to transition from user-mode to kernel-mode and begin the kernel's side of the execute of this system call.

Wow64

Before we jump into lab, we're going to talk about one more topic, WoW64. For those of you who haven't heard of it, it's the reason you can run 32-bit shellcode on your 64-bit operating systems. Windows handles the translation of architecture from the user-mode 32-bit application to the 64-bit kernel.

This translation requires specific instructions and execution paths to convert the information from 32-bit to 64-bit and this is often called Heaven's Gate.

If you didn't notice, for this lecture I was showing you 64-bit syscalls as you can see from the registers. However when working with the 32-bit version on a 64-bit operating system, the calls look slightly different.

```
0:000> u ntdll!NtCreateFile
ntdll!NtCreateFile:
```

```
77569bf0 b855000000      mov     eax,55h
77569bf5 ba501f5a77      mov     edx,offset ntdll!Wow64SystemServiceCall (775a1f50)
77569bfa ffd2            call    edx
77569bfc c22c00          ret     2Ch
77569bff 90              nop
```

Here we see that we call the Wow64SystemServiceCall function indirectly by loading it into the EDX register. This leads to a `JMP` instruction to ntdll!Wow64Transition. You can follow these steps in Windbg, if you need to see it visually or want to follow along.

If you see a bunch of garbage instructions, make sure you understand that the ntdll!Wow64Transition is a pointer to the executable code and dereference it when looking at it in a debugger.

The first instruction there is critical, `jmp 0033:774e7009`, this is the transition from 32-bit to 64-bit as it makes a far jmp to enable 64-bit mode. We wont talk too much more about this, but it does set up for our last lab, using ShellWasp.