670.2

Getting to Know Your Target

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

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

SEC670.2 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Getting to Know Your Target

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.2
Welcome to Section 2 of SEC670. In this section, we will be getting to know the target very well by creating various tools to obtain detailed information.

Table of Contents (1)

This page intentionally left blank.

Table of Contents (2)

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 2

Gathering Operating System Information

Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration

Lab 2.2: ProcEnum

Lab 2.3 CreateToolhelp

Lab 2.4 WTSEnum

Installed Software

Directory Walks

Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will dive into how to gather OS specific information and why it is important.

Objectives

Our objectives for this module are:

- Discuss the importance of determining OS information
- Explore public methods to retrieve information about the system
- Explore undocumented method to retrieve information about the system

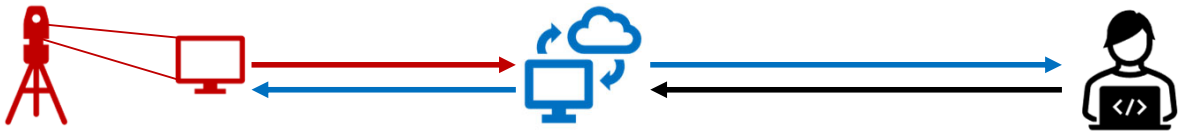
Objectives

The objectives for this module are to understand how important it is to determine what OS version your target is running and/or what service pack your target has, explore a few public methods for retrieving system information, as well as explore some undocumented methods. There are some methods that are more reliable than others, as well as some methods that might return inaccurate information.

Survey Script

Survey the host and determine where you are.

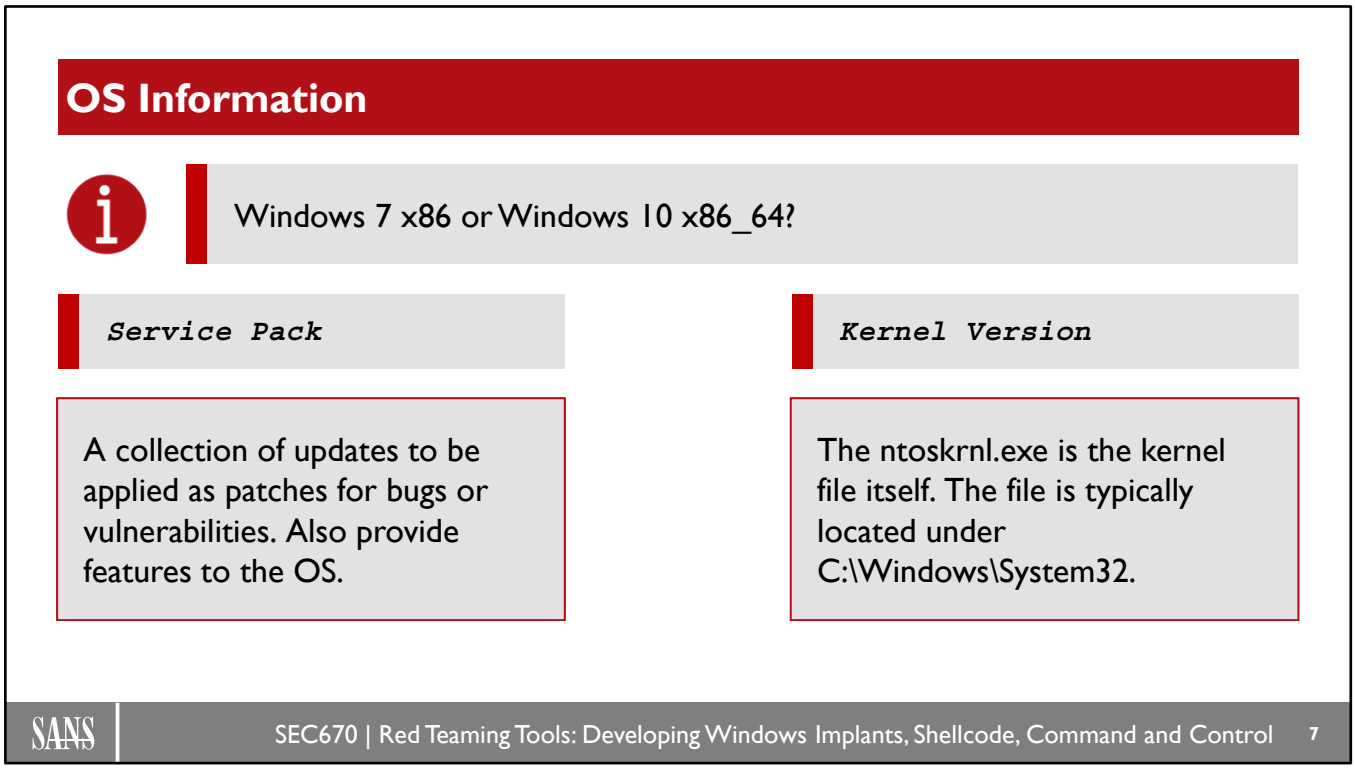
Knowing the system that you are on is vital to the success of your operation. A host survey tool can query various components and report back its findings that can then be used to determine the next action.



Survey Script

If you are a red teamer, or a penetration tester, or have taken SEC560, then you may already know that one of the first tasks that you would typically perform is some recon. Recon is such a broad term that can encompass many things, like users, networks, shares, etc. The focus at this point is to learn more about the system. The more detailed information you can gather, the more informed decisions can be made, like what the next action you would take next is. Privilege escalation could be an example of one such action to take after you have gained some insight into what that system is being used for and what value it might be for you. There are several tools that exist today that make this task trivial. There should be some questions coming to mind here. How do they get that information? What APIs are they using? What logs, if any, are generated from calling certain APIs? Since this is a developer-focused course, you will be creating a recon capability.

One of the tools you could create would be a tool that would execute on the target and query several pieces of information about the target. There can be a standard set of information that you might want to gather, like applications installed, especially any AV/EDR solutions. There can also be specific information that could directly relate to the primary goal of an operation, like looking for a specific folder or file. If that folder or file is not there, then the tool can clean itself off target and be done.



OS Information

Perhaps one of the most important pieces of information to gather first would be the exact version of the operating system, if that is not known already. Typically, you would know at least some basic information about the target beforehand, like if it is a Windows 7 or Windows 10 target, but you might not know the exact service pack or what version of ntoskrnl.exe is on the target. The more details you can gather the better, because Windows 10 is not good enough, especially if someone would want to bring in additional payloads that would be specific to the target. If your tool incorrectly said the target was x64 when it was in fact x86, well, let us just hope a process crashes instead of the target system being bug checked, a.k.a. blue screen (BSOD). To aid us in gathering some of this information, there are a number of APIs that we can call. However, certain APIs might not be available on older versions of Windows. Just because an API is available to use on Windows 10 does not automatically mean that it will be available to use on an older version like Windows XP Service Pack 1. The Windows terminal services (WTS) API family is one such example and some of the WTS APIs will be used later for process enumeration.

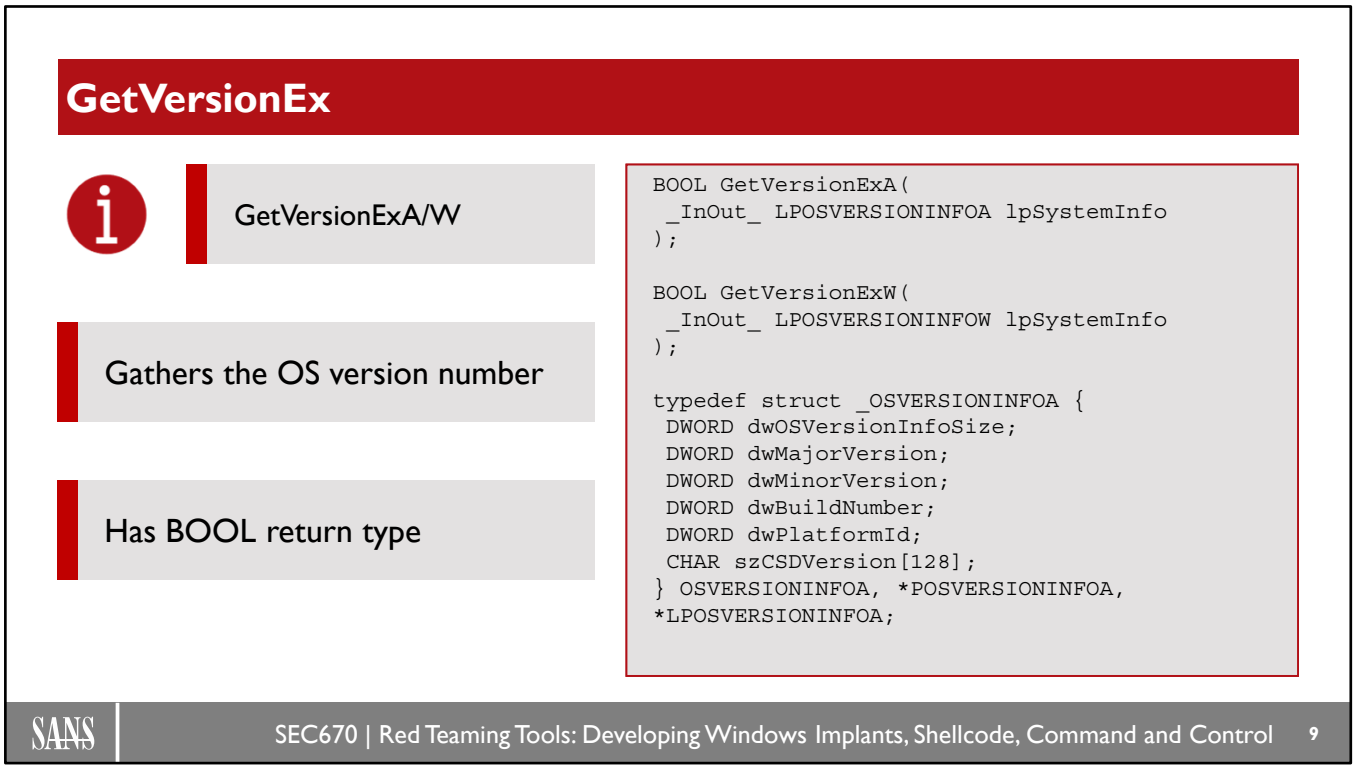
Windows Versions

Windows releases and their respective version numbers

Windows XP	5.1
Windows Server 2003	5.2
Windows Vista / Server 2008	6.0
Windows 7 / Server 2008 R2	6.1
Windows 8 / Server 2012	6.2
Windows 8.1 / Server 2012 R2	6.3
Windows 10 / Server 2016	10

Windows Versions

When you are querying the target to determine the specific version of the OS, you will not find something that tells you that the target is a Windows Vista system. Instead, you would be given back something like 6.1 to indicate Windows 7. The table is simply here for an easy reference when you are going about gathering information about the OS.



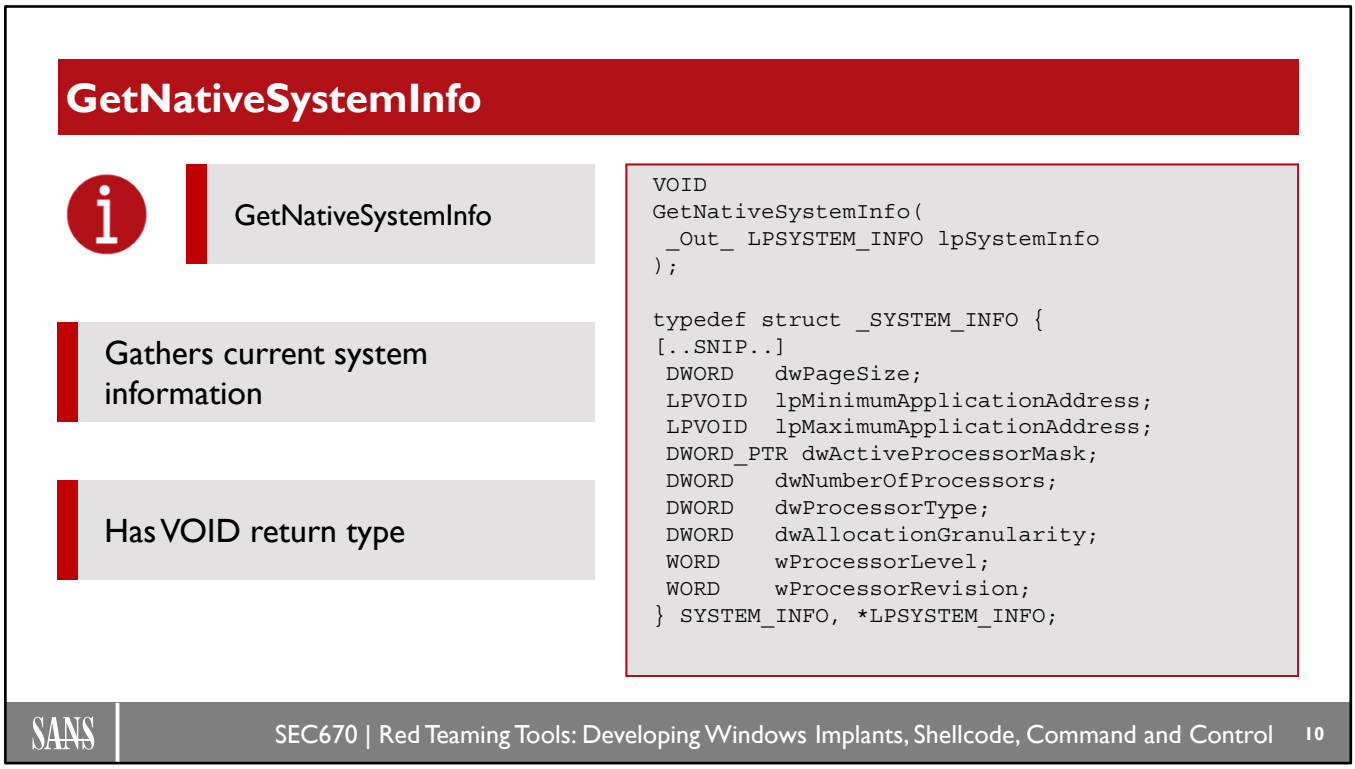
GetVersionEx

The ***GetVersionEx*** API can be used in gathering the Operating System's version number, such as one of the ones listed on the previous slide. Here on this slide are two versions of this API, the ANSI and the Unicode versions. The function takes a single argument for a pointer to an OSVERSIONINFO struct. Before we look at the struct, there is some interesting behavior that we need to understand. By default, if you were to call this API on your Windows 10 Dev VM, you would not get back version number 10. The highest you might get back is version 6.2, which would indicate Windows 8.

The reason for this behavior is mainly compatibility. Back in the day, programs written for XP and Vista would check the version numbers and if they were higher than 5 and 1 (5.1), then the application knew they were on something beyond XP. The check would fail, though when version 6.0 was released as the minor version number of 0 is not >= 1. Windows came up with a solution to never increase the major version number, only the minor. This obviously ended when Windows 10 was released because they are not using version 6.4, they are indeed using version 10.0. The only way for this function to return the correct version number is to account for a possible higher operating system that would be declared in its manifest file. The manifest file is purely XML data and nothing more.

Please note that Windows recently deprecated this API and would really like for you to use the newer version helper functions like IsWindows7OrGreater, or IsWindows10OrGreater, etc.

dwOSVersionInfoSize; the size of the struct and should be set using size of (OSVERSIONINFO)
dwMajorVersion; the major version number
dwMinorVersion; the minor version number
dwBuildNumber; the build number
dwPlatformId; the OS platform
szCSDVersion[128]; if there is a service pack installed, then this would the value for it, like "Service Pack 1".



GetNativeSystemInfo

The ***GetNativeSystemInfo*** API can be used in gathering some specific information about the target system for WoW64 applications. This will also work for x64 applications, but the results will be noticeably different. Since the function has a VOID return type, it will not return anything to the caller, so there would be no point in trying to assign its return value to a variable. What the API needs is a pointer to a structure that the API will fill out according to the information it queries. This is very common behavior for Windows APIs. In fact, most Windows APIs expect to be given a properly initialized structure to fill out for you. Here is a breakdown of the one and only argument to ***GetNativeSystemInfo***.

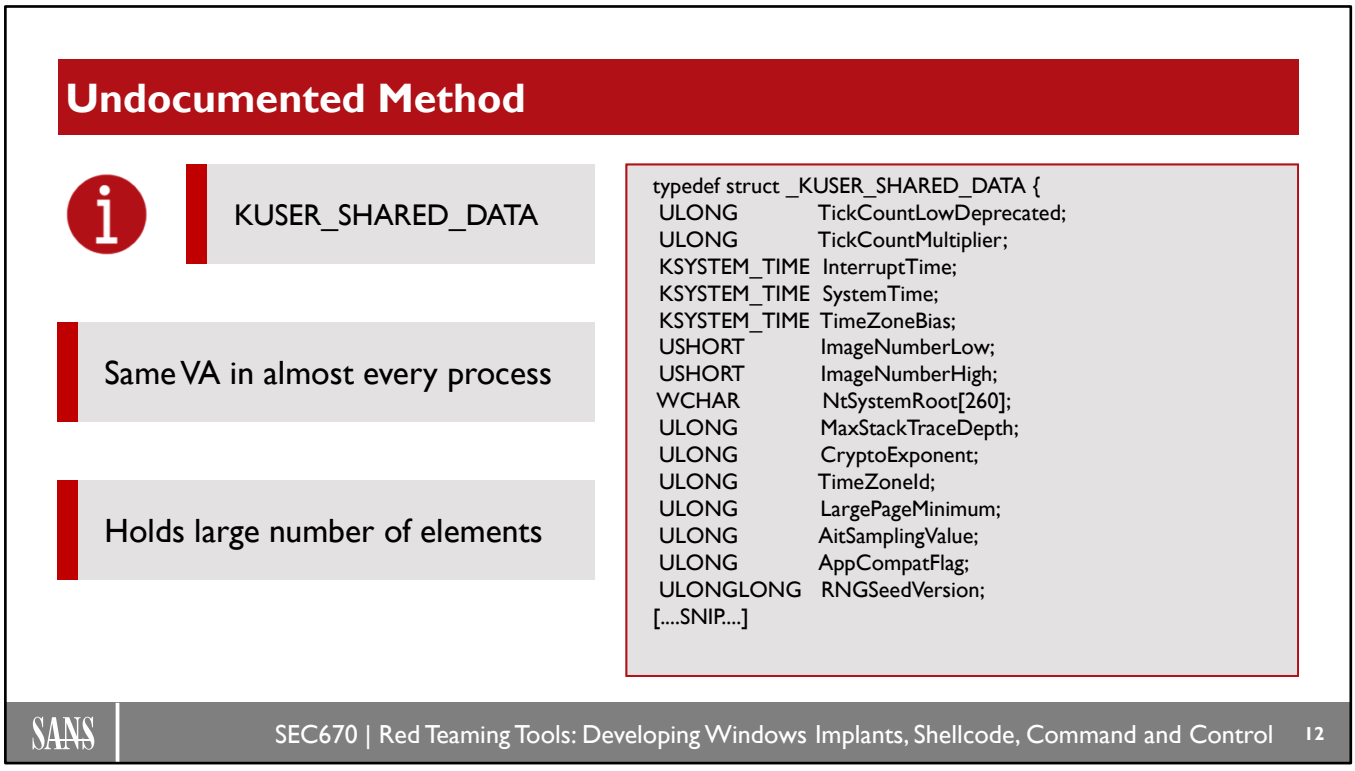
lpSystemInfo, as annotated by the SAL markup, it is an out parameter meaning the function requires write permissions to the variable. The variable, or pointer, must be of type LPSYSTEM_INFO, which is a structure that holds around 10 or so members.

Let us take a more detailed look at the SYSTEM_INFO struct.

dwOemId; is no longer being used but is simply maintained for compatibility. We need to use wProcessorArchitecture instead.
wProcessorArchitecture; the installed architecture of the process that is installed on the target. This field will have one of the values listed below:

9 - PROCESSOR_ARCHITECTURE_AMD64 (x64 AMD or Intel)
5 - PROCESSOR_ARCHITECTURE_ARM (ARM)
12 - PROCESSOR_ARCHITECTURE_ARM64 (ARM64)
6 - PROCESSOR_ARCHITECTURE_IA64 (Intel Itanium-based)
0 - PROCESSOR_ARCHITECTURE_INTEL (x86)
0xffff - PROCESSOR_ARCHITECTURE_UNKNOWN

wReserved; reserved for supposedly something amazing in the future? Who knows?
dwPageSize; the page size along with the granularity of page protection and the commitment. *VirtualAlloc* relies on this value for its operations.
lpMinimumApplicationAddress; this is a pointer to the lowest memory address that will be made accessible to programs and their DLLs.
lpMaximumApplicationAddress; the exact opposite as the previous member.
dwActiveProcessorMask; the set of processors that are configured on the system in the form of a mask, 0-31 bits each one indicating the processor.
dwNumberOfProcessors; how many logical processors are in the current group.
GetLogicalProcessorInformation relies on this value.
dwProcessorType; this is obsolete so do not rely on it.
dwAllocationGranularity; for virtual memory allocations, this is the granularity for the starting address.
wProcessorLevel; the processor level that is dependent on the architecture.
wProcessorRevision; the processor revision that is dependent on the architecture.



Undocumented Method

The undocumented method of retrieving system information is to query the KUSER_SHARED_DATA struct that is present in virtually every single process at the same Virtual Address: 0x7FFE0000. KUSER_SHARED_DATA is a massive structure that is defined in the ntddk.h header file. The structure stores an enormous amount of information that we can leverage for our needs of gathering system information and then some. However, this is not the recommended method of doing so, but we do not really care too much about that. Implant developers and malware authors tend to favor the undocumented methods more than anything else. Knowing the correct offsets for what information you need can be very useful instead of copying everything over from this structure. Currently, the offsets for the Major Version and Minor Version have been offsets 0x26C and 0x270, respectively. The Build Number can be found at offset 0x260. For more details about this structure, check out the header files and various online resources like Geoff Chappell's documentation.

References:
https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntexapi_x/kuser_shared_data/index.htm
https://www.vergiliusproject.com/kernels/x64/Windows%2010%20|%202016/2004%2020H1%20(May%202020%20Update)/_KUSER_SHARED_DATA

Source Code Review

Source code review!

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

Lab 2.1: OS Info

Gathering information about the OS and target

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

Lab 2.1: OS Info
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to understand how you can retrieve various information about the OS of your target.

Module Summary

Discussed how obtaining accurate system information is key

Covered documented and recommended methods to obtain the information

Covered undocumented methods to obtain the information

Module Summary

In this module, we discussed why you would want to know the exact details of your target's OS version and architecture, we also explored a few Windows APIs that enable us to do so, and finally, we took a look at an undocumented method by means of KUSER_SHARED_DATA.

Unit Review Questions

What structure can be found at VA 0x7FFE0000?

A EPROCESS

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Questions
Q: What structure can be found at VA 0x7FFE0000?

A: EPROCESS

B: KPROCESS

C: KUSER_SHARED_DATA

Unit Review Answers

What structure can be found at VA 0x7FFE0000?

A EPROCESS

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Answers
Q: What structure can be found at VA 0x7FFE0000?

A: EPROCESS

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

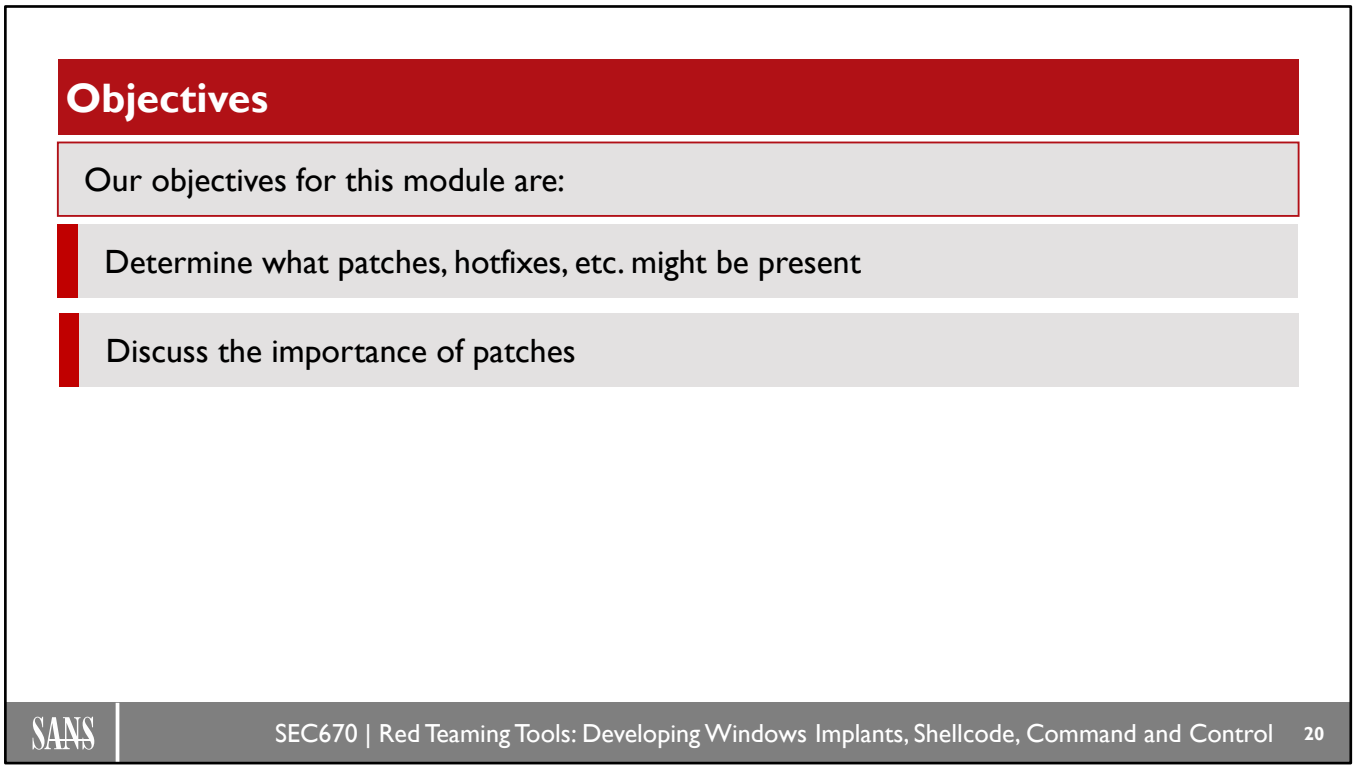
Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will discuss how to gather information about service packs, hotfixes, and patches, as well as why the information might be important.



Objectives
The objectives are to determine what patches or hotfixes a system might have and how they might affect an operation.

Windows Hotfixes

Used to fix critical issues in software

Also referred to as Quick Fix Engineering (QFE) updates, hotfixes are used to apply a vital fix to software applications. Users that have Windows updates set to automatic will have hotfixes downloaded without much user intervention. The only exception would be a reboot.

Windows Hotfixes

Windows updates bring with them any number of things, but the emphasis here would be hotfixes. The term "hot fix" traditionally would mean that a patch to a software program can be applied while the system was still running. You may have heard the phrases "hot swappable," "hot patching," etc. which all mean similar things, but the common component is "hot" where the system or device does not have to go through a shutdown procedure; hot swapping hard drives when one drive becomes full, or in our case, issuing a quick fix to a certain application while the system is still running, and the user is presumably performing work. Some Windows updates do, however, require a reboot for a change to come into effect, but there are times when a reboot is not required to be performed. It is important for us to know what hotfixes have been applied so that time is not wasted attempting to execute an exploit against something that has already been patched. Doing so could tip off the user/admin to our presence on the system along with generating unnecessary logs.

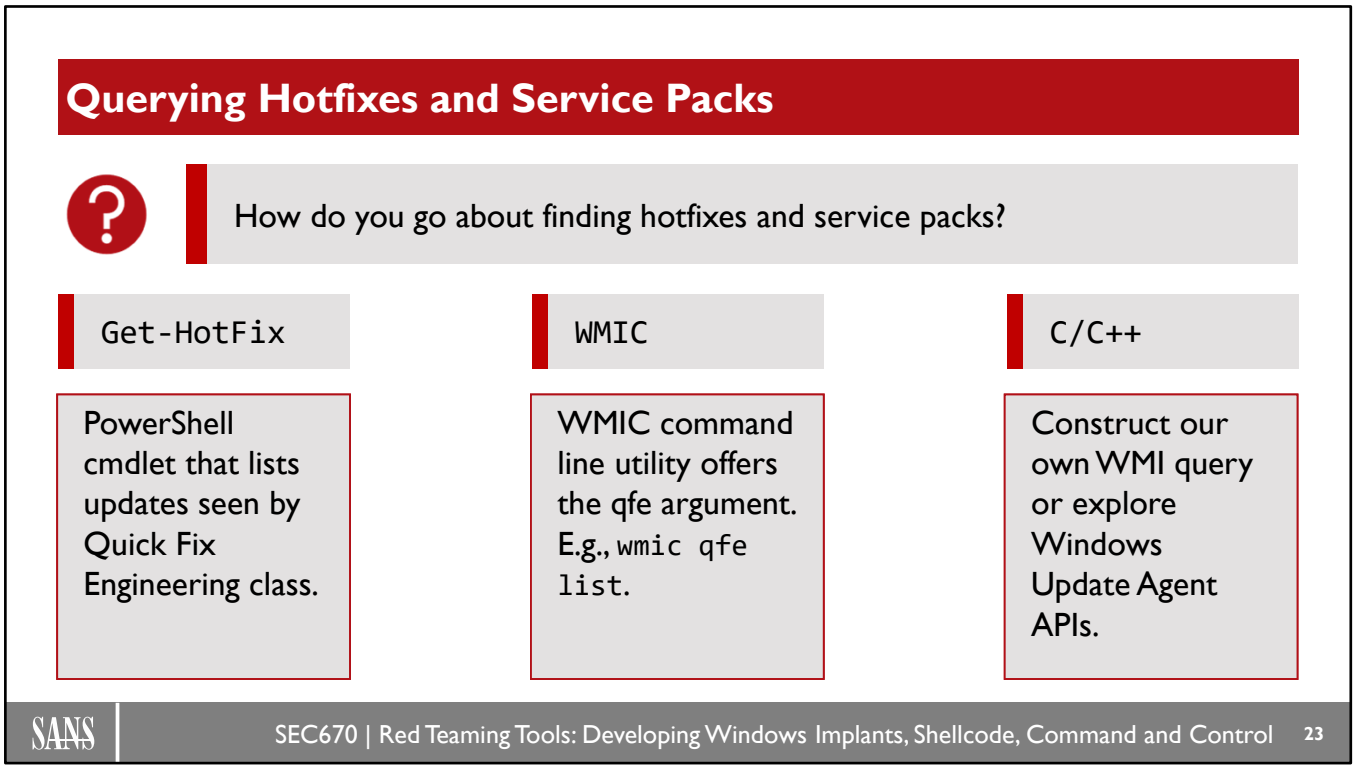
Service Packs

Bundled hotfixes

Each service pack brings with it a grouping of one or more hotfixes that will be applied to the OS. Each service pack that targets a particular OS version will have all previous hotfixes that former service packs brought with it so that a user can jump straight to the most recent service pack without installing each one sequentially.

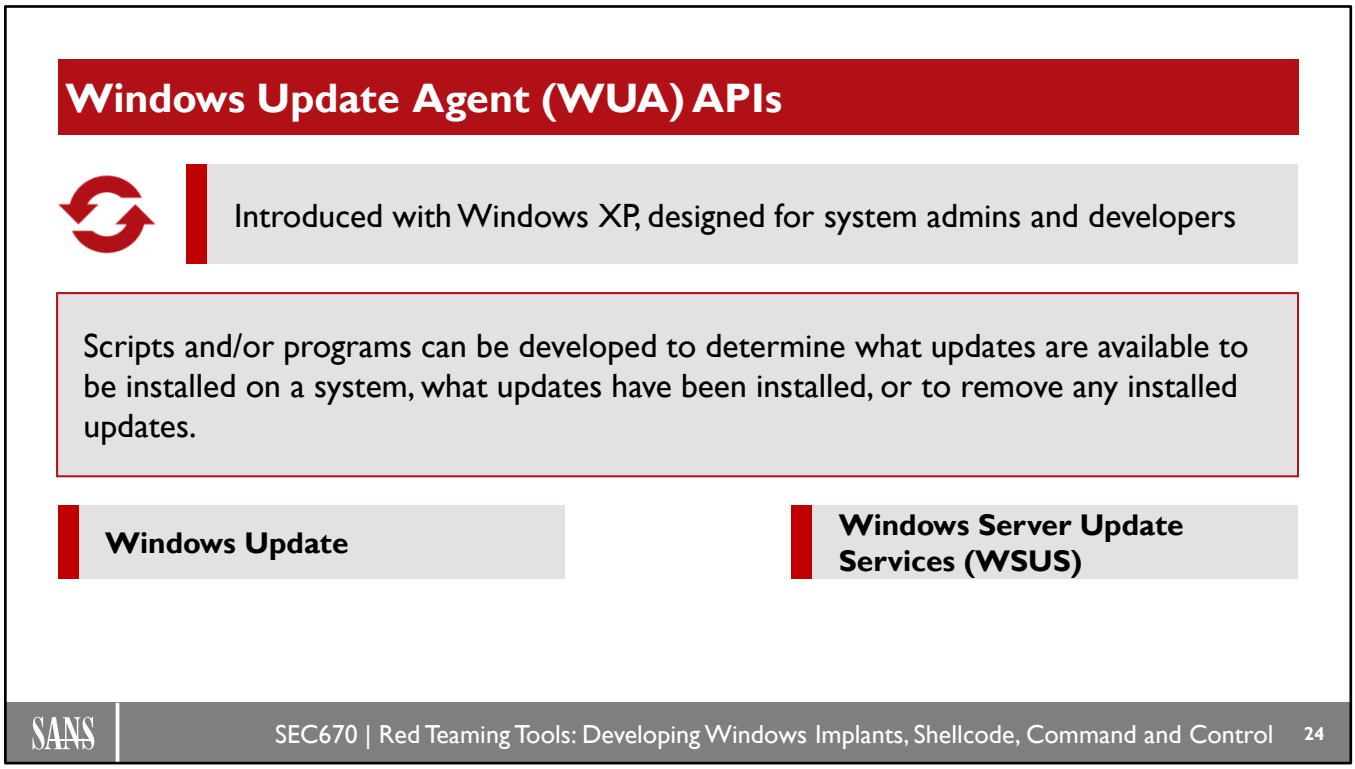
Service Packs

It really would not make much sense for Windows to push down hotfixes by themselves one at a time, but to rather bundle them up in what is called a service pack. Bundling the hotfixes together is pretty great because it makes updating a lot more efficient. Each OS version can have any number of service packs issued, like Windows XP had three of them: SP1, SP2, and SP3. Each service pack might also mean that exploits might have to be adjusted depending on the target's SP level. Within the Metasploit Framework, the option to choose a specific target is made available. An exploit targeting a vulnerable FTP server might need to have the target be specified, say for Windows XP SP3 versus XP SP1. The same considerations must be taken if your implant is going to exploit anything from local privilege escalation to persistence, etc.



Querying Hotfixes and Service Packs

Windows provides users and admins with a number of options to go about querying patches, or hotfixes, which have been applied to a system. Perhaps the easiest method would be to drop into a PowerShell instance and run the Get-HotFix cmdlet. The downside is that the cmdlet does not return everything because on the back end it only queries the WMI Win32_QuickFixEngineering class. Knowing that, we can easily craft our own WMI (Windows Management Instrumentation) query and completely avoid using PowerShell. Another alternative is to use the *wmic qfe list* command. The QFE part of the command gives away what it is querying, and it also indicates that it will show the same results as Get-Hotfix. Lastly, it will have the same downside that the Get-HotFix cmdlet does. If you were to look at the Windows Programs and Features listing, you might notice a difference in the listing. This is because the QFE WMI class only sees certain types of updates. We want a more complete view and as such, we must utilize the Windows Update Agent APIs. We can use them as a standalone or in combination with what the QFE class returns.



Windows Update Agent (WUA) APIs

Programmatically creating a solution is often more complicated than using pre-built programs or tools like the wmic tool. We turn to this option when existing tools fail or do not show us the entire picture we are hoping to see. Enter the WUA API family. Since its debut with Windows XP, it has been used by system administrators and developers alike, to determine what updates computers in their organization require. What we want to use it for is to determine what updates have already been applied because we like to go after the lowest hanging fruit. WUA is a set of COM interfaces, and we must create instances of whichever interface we need. Before the WUA APIs are made available to us, we must reference the proper header file and its respecting lib file; Wuapi.h and Wuguid.lib, respectively. To create the correct interface, we need to choose the COM object that is most suited for our needs, and that could be the *UpdateSession*, *UpdateSearcher*, and *SearchResult* WUA objects.

WUA UpdateSession Object

IUpdateSession

Represents update session object

Search, download, install, uninstall

```
#include <wuapi.h>

HRESULT res = CoInitialize(NULL);

IUpdateSession* upSsn;

CoCreateInstance(
   ...,
   (PVOID*)&upSsn
);
```


WUA UpdateSession Object

Because the WUA APIs are all COM-based, the calling thread must initialize the COM library by calling the ***CoInitialize*** API. The WUA *UpdateSession* object is a COM interface, hence the naming convention of adding the letter "I" to the API name. Before we can do anything useful, we must create an instance of the object by calling the ***CoCreateInstance*** API and passing in the address to our pointer variable *IUpdateSession*. The pointer variable is going to receive the pointer to the interface if the function call succeeds. We use the ***CoCreateInstance*** API because it will create and initialize the object of whatever class we are passing in for the CLSID parameter. It is from this newly created object that we will then be able to create other objects using methods provided by the *UpdateSession* object.

WUA UpdateSearcher Object

IUpdateSearcher

Created by the UpdateSearcher coclass

Used to search for updates on the target system

```
#include <wuapi.h>

...
IUpdateSession* upSsn;
IUpdateSearcher* upSearch;
ISearchResult* results;

upSsn->CreateUpdateSearcher(&upSearch);
...
upSearch->Search(criteria, &results);
```


WUA UpdateSearcher Object

The previous slide showed how to create the *UpdateSession* object and now that it is created, we can call its ***CreateUpdateSearcher*** method to make the *UpdateSearcher* object. We need to create the *UpdateSearcher* object because it provides a method that we need to start conducting a search of updates that have been applied to the system. The method of interest for us is the ***Search*** method. The ***Search*** method requires two arguments: the criteria and a pointer to the *UpdateSearcher* pointer variable. The criteria argument is where we can specify our search criteria—think of it like a filter of sorts. The criteria can be created as a string like the following: "IsInstalled=1". The criteria string can contain several filters using the "or" operator so you could expand the string to be "IsHidden=1 or IsInstalled=1". There are (sometimes) updates that are marked as hidden on the computer, but we want to see them anyway, so we set that variant to true. The collection of results that match the specified criteria will be stored in the results *ISearchResult* pointer variable. It is from this object that you can start to process the results from the search as it exposes several methods of interest to us.

WUA SearchResult Object

ISearchResult

Used to represent search results

Has methods that can query updates from a resulting search

```
// interface collection of updates from a
resulting search

ISearchResult* results;
IUpdateCollection* upList;
LONG upSize;

upSsn->CreateUpdateSearcher(&upSearch);
upSearch->Search(criteria, &results);

results->get_Updates(&upList);
upList->get_Count(&upSize);
```


WUA SearchResult Object

The *SearchResult* WUA object represents the collection of updates that matched the search criteria. To get the collection interface of those updates from the search, we will need to call ***get_Updates*** method from the *SearchResult* object. The argument we pass to the method is the address to the pointer variable of an *IUpdateCollection* pointer. The *UpdateCollection* object is used to represent the list of updates, which is an ordered list of updates. We can then use this ordered list and iterate over it based on the size of the list that can be gathered after calling the ***get_Count*** method. We pass the ***get_Count*** method the address to a LONG variable to store the size of the update list. Finally, we can gather the details of the updates we found on the system and either print them out to the terminal window or write the results to a log file somewhere on disk.

Sample Code

```
...
IUpdate* upItem;
BSTR upName;

results->get_Updates(&upList);
upList->get_Count(&upSize);

LONG index = 0;
for ( ; index < upSize; index++)
{
 upList->get_Item( index, &upItem );
 upItem->get_Title( &upName );
 ...
}
```


Sample Code

This slide just holds some pseudo code for how one could possibly iterate over a collection of results. Some new variables here are the pointer variable for the *IUpdate* collection object, which is used to obtain whatever properties and methods an update might have. The next variable is a basic string variable named *upName* that is short for the update name. As we iterate over the collection, the *upName* will be updated to the one at the proper index in the collection. The *IUpdate* collection object exposes other methods too, like ***get_Type***, to get the type of the update; ***get_Title***, to get the title of the update; or ***get_KBArticleIDs***, to get the collection of KB article IDs that are tied to the update.

Specifically, the code on the slide will get the current update item at the index specified, update the *upItem* variable so we can get the title of the current update, and move on with the iteration process after that. To be even more detailed and thorough, you could do more during this loop by calling some of the other *IUpdate* methods, but this is enough to get you started.

Module Summary

Discussed the importance of updates and patches

Learned how to obtain information about patches

Used the WUA APIs

Module Summary
In this module, we discussed hotfixes, service packs, and how to get information about them using the WUA APIs.

Unit Review Questions

What PowerShell cmdlet queries the Quick Fix Engineering class?

A Get-HotFix

B Get-Updates

C Get-ServicePack

Unit Review Questions
Q: What PowerShell cmdlet queries the Quick Fix Engineering class?

A: Get-HotFix

B: Get-Updates

C: Get-ServicePack

Unit Review Answers

What PowerShell cmdlet queries the Quick Fix Engineering class?

A Get-HotFix

B Get-Updates

C Get-ServicePack

Unit Review Answers
Q: What PowerShell cmdlet queries the Quick Fix Engineering class?

A: Get-HotFix

B: Get-Updates

C: Get-ServicePack

Unit Review Questions

What is the update family of APIs used to query hotfixes?

A WUA

B LUA

C FUA

Unit Review Questions
Q: What is the update family of APIs used to query hotfixes?

A: WUA

B: LUA

C: FUA

Unit Review Answers

What is the update family of APIs used to query hotfixes?

A WUA

B LUA

C FUA

Unit Review Answers
Q: What is the update family of APIs used to query hotfixes?

A: WUA (Windows Update Agent)

B: LUA

C: FUA

Unit Review Questions

What WUA object is used to find updates on a system?

A SearchResult

B UpdateSearcher

C UpdateSession

Unit Review Questions
Q: What WUA object is used to find updates on a system?

A: SearchResult

B: UpdateSearcher

C: UpdateSession

Unit Review Answers

What WUA object is used to find updates on a system?

A SearchResult

B UpdateSearcher

C UpdateSession

Unit Review Answers
Q: What WUA object is used to find updates on a system?

A: SearchResult

B: UpdateSearcher

C: UpdateSession

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will look at the how and why when it comes to process enumeration.

Objectives

Our objectives for this module are:

- Understand the need for process enumeration
- Take a deeper look at processes
- Explore the various methods to enumerate processes

Objectives

The objectives for this module are to understand the need for enumerating processes. Furthermore, to understand processes even more, we will look at what processes are, how they are created, different process states, and the several methods involved with enumeration. Let's get to it.

Why Enumerate Processes?

Must find out what applications are running

An important part of conducting a survey is gathering a list of running processes. Depending on what processes you find, your operation may come to a halt, or you might deem the target safe for further operations.

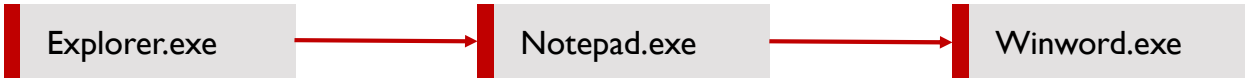
Why Enumerate Processes?

One of the features that implants often have is the ability to enumerate processes. If you are familiar with Metasploit's Meterpreter session, you might know that it can enumerate processes. There are several reasons why a red team operator might want to gather a list of running processes. One reason could be to find a suitable target process for shellcode injection. Another reason could be to find out if the target has a vulnerable application that could then be matched to an exploit to aid in escalation of privileges, should your initial access not be elevated already. Yet another reason to enumerate processes would be to determine if there is any security product present on the system. Depending on the security product installed on the system, you might decide to halt or suspend operating on that target. Many solutions have a cloud-based portion that will take your tooling and conduct analysis in its cloud engine. If you do not want your tooling to be siphoned off to their cloud, then perhaps you need to clean off the target.

How Are Processes Created?

The main method to create a process is an API call.

The CreateProcess API is one of the main drivers for creating a process on a system. The API is complex, it has many arguments, and requires the kernel to kick in and make the system object in system space. The kernel tracks all processes and keeps the process objects organized in a linked list.



How Are Processes Created?

Process creation can be kicked off by calling the ***CreateProcess*** API. User programs can easily make calls to ***CreateProcess***, but they do not have to do any heavy lifting. The kernel will eventually take over and it is up to the kernel to create a process object in system space, inject the main thread into the process, and append the process object to the existing linked list of process objects. There is more that the kernel does for process creation, but those items hit the general responsibilities. There are two types of process objects that are linked together: the *_EPROCESS* object and the *_KPROCESS* object. Both objects, together, represent a single process and each one holds different information about a process that is important for various Windows subsystems. The *KPROCESS* object is the first member of the *EPROCESS* object and is intentionally designed this way. If you have the address of the *EPROCESS* object, you also have the *KPROCESS* as well since it is at offset 0x00.

_EPROCESS

EPROCESS

Kernel object representing processes

DKOM attacks can unlink processes

```
kd> dt nt!_EPROCESS


//0xa40 bytes (sizeof)
struct _EPROCESS
{
 struct _KPROCESS Pcb;              //0x0
 struct _EX_PUSH_LOCK ProcessLock;  //0x438
 VOID* UniqueProcessId;             //0x440
 struct _LIST_ENTRY ActiveProcessLinks; //0x448
 struct _EX_RUNDOWN_REF RundownProtect; //0x458

 [..snip..]

}
```


_EPROCESS

The *_EPROCESS* object is one of the structures that Microsoft determines to be opaque to user mode apps and to developers. The *EPROCESS* object belongs to the Windows Executive subsystem since it needs to have access to all information to manage the process. From the small snippet on the slide, you can see some of the information stored in it. One of the best methods to peeking inside of opaque structures is to look at them during a kernel debugging session. KDNET makes getting kernel debugging working with minimal effort. Another way is to look at the Vergilius Project, which has done an amazing job at documenting various kernel structures from many Windows versions. Check out their site here: https://www.vergiliusproject.com.

The *ActiveProcessLinks* is an interesting one because its type is *_LIST_ENTRY*, indicating that it is used to link to other objects. Windows uses this type to make a doubly linked list. The *LIST_ENTRY*'s FLINK would take us to the next *EPROCESS* struct in the list. Many programs indirectly walk this list when enumerating processes like the PowerShell cmdlet *Get-Process* or executing *tasklist* at the command line. When an attacker can get into the kernel, these objects can be directly manipulated, and a process can be unlinked from the chain and therefore hidden from users.

_KPROCESS

KPROCESS

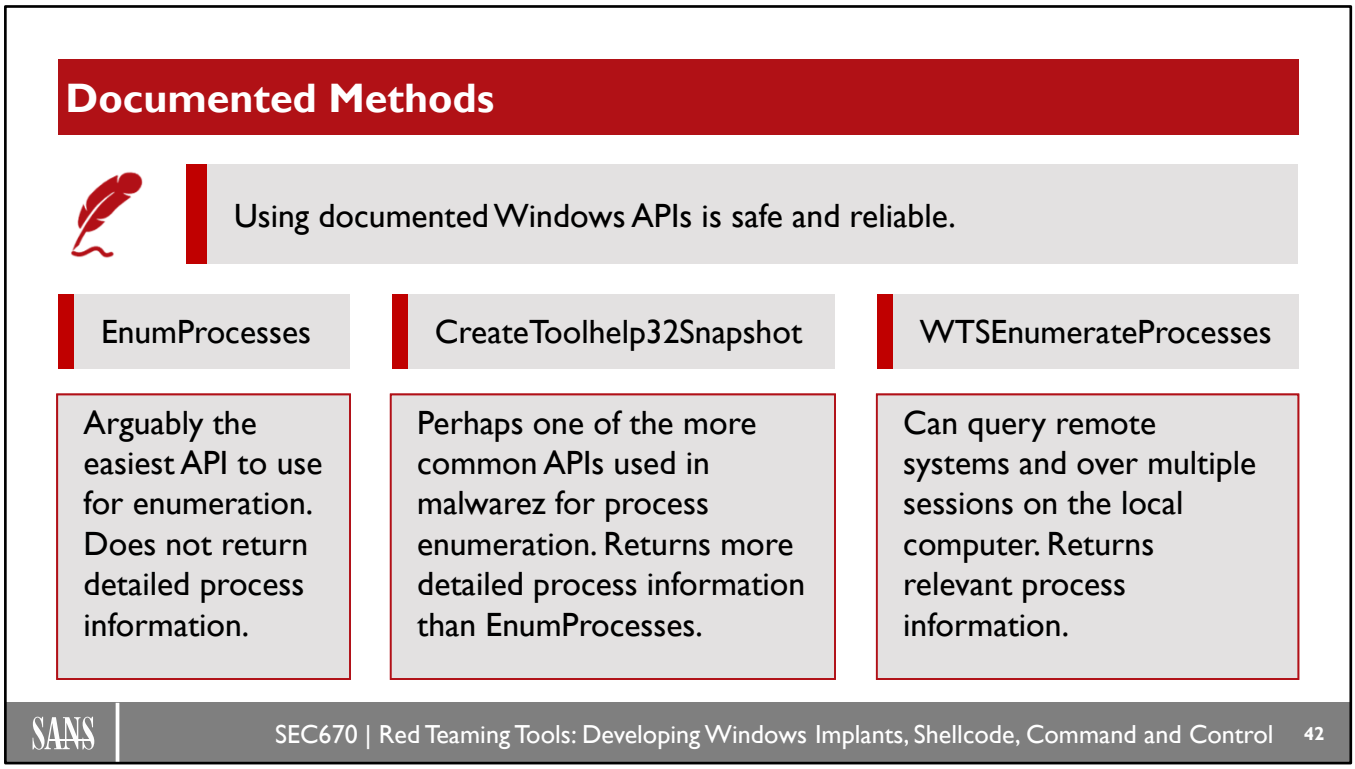
Kernel object representing processes

Used by the lower layer of the Kernel

```
//0x438 bytes (sizeof)
struct _KPROCESS
{
 struct _DISPATCHER_HEADER Header;   //0x0
 struct _LIST_ENTRY ProfileListHead; //0x18
 ULONGLONG DirectoryTableBase;       //0x28
 struct _LIST_ENTRY ThreadListHead;  //0x30
 ULONG ProcessLock;                  //0x40
 ULONG ProcessTimerDelay;            //0x44
 ULONGLONG DeepFreezeStartTime;      //0x48
 struct _KAFFINITY_EX Affinity;      //0x50
 ULONGLONG AffinityPadding[12];      //0xf8
 struct _LIST_ENTRY ReadyListHead;   //0x158
 [..snip..]

}
```


_KPROCESS

The *_KPROCESS* object is important to the core of the kernel and is not exposed via the Object Manager like the *EPROCESS* object is. Some of the members of the KPROCESS object are important for thread scheduling, like what threads are ready to run (*ReadyListHead*), tracking quantum, priorities, CPU affinity, linking of threads, etc. For example, the *ThreadListHead* is of type *_LIST_ENTRY* so we know it is part of a doubly linked list. The list is a chain of Threads that have been created in the process which is represented by the *KPROCESS* object. Another important member is the *DirectoryTableBase* of type *ULONGLONG*, which will hold the physical address of the process' Page Directory Table, an important item for Virtual Address Translation.



Documented Methods

When it comes to reliability and stability, using officially supported and documented APIs is great because you know the API should work as advertised. For the documented methods of enumerating processes, we have three options. The first option is the ***EnumProcesses*** API, which is a very simple API to understand and implement. One of the major drawbacks about ***EnumProcesses*** is the lack of detailed information about each enumerated process. If you wanted to get more information with this API, then you could attempt to open a handle to the process via its *ProcessId*. The second option is to use the ***CreateToolhelp32Snapshot*** API, which offers much more information about each process. The downside to this API is that because it only takes a snapshot of the currently mapped processes, you will miss any new processes after the snapshot is taken. The last one we will cover is ***WTSEnumerateProcesses***, which offers a nice feature of remote process enumeration. Some of the above APIs might not be available for every version of Windows, so you must be sure to find out and test.

EnumProcesses API

EnumProcesses()

Used to obtain the process IDs on the system

Has BOOL return type

```
BOOL EnumProcesses(
 _Out_  DWORD   *lpidProcess,
 _In_   DWORD   cb,
 _Out_  LPDWORD lpcbNeeded
);
```


EnumProcesses API

As mentioned on the previous slide, the ***EnumProcesses*** API is incredibly easy to use when it comes to process enumeration. The API can be chosen over the other options if you do not care about getting detailed information about the processes on the system. The API will only return the process IDs of each process object that the kernel has created at the time the API is called. There are three arguments required: two out parameters and a single in parameter. Let us break down some of these arguments.

lpidProcess, is an out DWORD pointer to the array that will hold the process IDs.

Cb, is an in DWORD that indicates the size, in bytes, of the array.

lpcbNeded, is an out LPDWORD that will indicate how many bytes were placed in the array.

Because the function requires a buffer, the array, it needs to be large enough to hold all the PIDs. The function will not indicate beforehand how large the buffer needs to be, so we can just make a large one to err on the side of caution and avoid the need of having to call the API twice.

Example: EnumProcesses

```
if ( !EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize) )
{
    // fail and bail code here
    goto fail_and_bail;
}

// iterate over the results
for ( DWORD i = 0; i < dwCount; i++ )
{
    HANDLE hProc = OpenProcess( PROCESS_QUERY_LIMITED_INFORMATION, FALSE,
dwProcList[i] );
    // do something with the handle if OpenProcess succeeds
}
```


Example: EnumProcesses

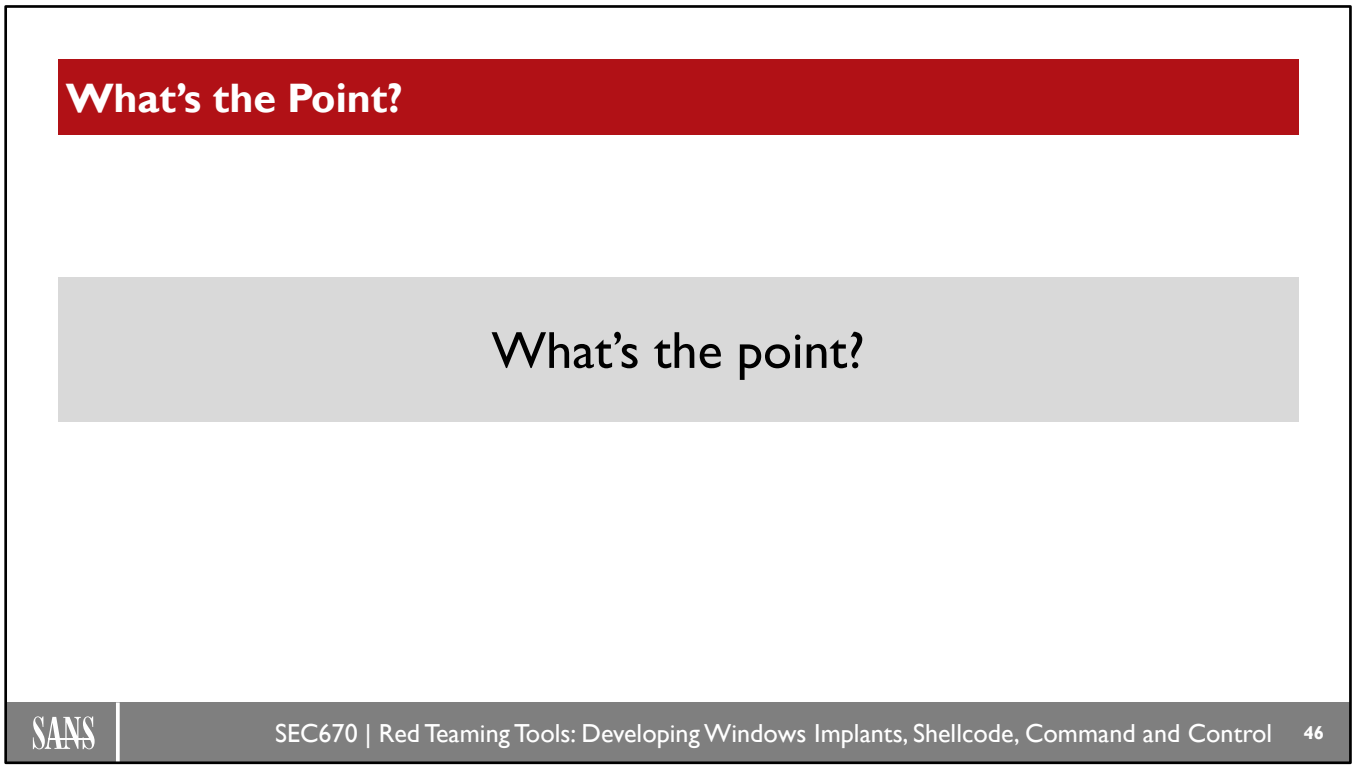
This example of ***EnumProcesses*** is a small snippet for how to call this API. There is not a lot of code here to make this work because it is a very simple API to use, and it does not return a lot of information. The API will fill out a list that will hold the PIDs of the processes that are currently mapped into memory. The API does not tell us how many processes were found, so we must do that on our own. We can use some simple math and figure out how many entries there are in the array of PIDs. To determine the number of processes, divide the *dwRealSize* by the size of the DWORD data type. Now that the count has been calculated, we can use that value in a *for* loop so we can iterate over the array. Each iteration will execute the *wprintf* statement showing the user the PID that is currently being processed. Though, your implant would not be making *printf* type of function calls because there would be no terminal to see the results. Instead, it would be more beneficial to create a log file or place the results in a memory buffer to later be sent back to your C2 infrastructure. You can also gather more information about each PID by calling ***OpenProcess*** against each one. If you are successful with that operation, you could then get more details about that process via the handle.

Lab 2.2: ProcEnum

Using EnumProcesses, enumerate the processes on the system.

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

Lab 2.2: ProcEnum
Please refer to the eWorkbook for the details of the lab.



What's the Point?
The point of the lab was to explore the ease of use for this API. It does have a few drawbacks, like the limited information, but that can be accounted for by opening a process handle to each PID returned by the API.

CreateToolhelp32Snapshot API

CreateToolhelp32Snapshot()

Creates a snapshot of a process

Can take snapshots of heaps and threads as well

```
HANDLE CreateToolhelp32Snapshot(
 _In_ DWORD dwFlags,
 _In_ DWORD th32ProcessID
);

BOOL Process32First(
 _In_  HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);

BOOL Process32Next(
 _In_  HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);
```


CreateToolhelp32Snapshot API

The ***CreateToolhelp32Snapshot*** API was discussed during the Create APIs module during Section 1, but it is being included again here as it specifically relates to enumerating processes. The API will create a snapshot of the specified processes, if any, and return a handle to that snapshot. Emphasis must be placed on the "snapshot" since you could easily miss a new process being created as this is not a dynamic view. You can use the handle to perform your queries and extract the information you are looking for like a specific process name or module name the process has loaded. The function is relatively easy to call since it only takes two parameters which are of the same type. The *dwFlags* parameter is the most important as it dictates what data should be collected in the snapshot. There are seven flags that can be passed here, but the most interesting flag for us in this use case is **TH32CS_SNAPPROCESS** because, as the name implies, will grab all processes that have been mapped into memory. There are other flags that can be used to capture other useful information, like modules and threads if you need to enumerate those items.

After the snapshot has been taken, you can perform our process enumeration using the ***Process32First*** function. The ***Process32First*** function will gather information about the first process in the snapshot. The function requires two parameters: the handle to the snapshot and a pointer to a variable of type *PROCESSENTRY32* structure. Typically, what you would see is a call to ***Process32First*** and then a loop that uses the ***Process32Next*** function to determine when to break out of the loop. ***Process32Next*** requires the same arguments as the ***Process32First*** function. The next slide will show a small snippet of code of what this could look like.

Example: CreateToolhelp32Snapshot

```
DWORD lastError = 0;
HANDLE snapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

PROCESSENTRY32 pe32 = { 0 };
pe32.dwSize = sizeof PROCESSENTRY32;

printf("%20s %7s\n", "Process Name", "PID");
printf("..........................................\n");
do {
    printf("%20ws", pe32.szExeFile);
    printf("%8d\n", pe32.th32ProcessID);
} while (Process32Next(snapShot, &pe32));

CloseHandle(snapShot);
return ERROR_SUCCESS;
```


Example: CreateToolhelp32Snapshot

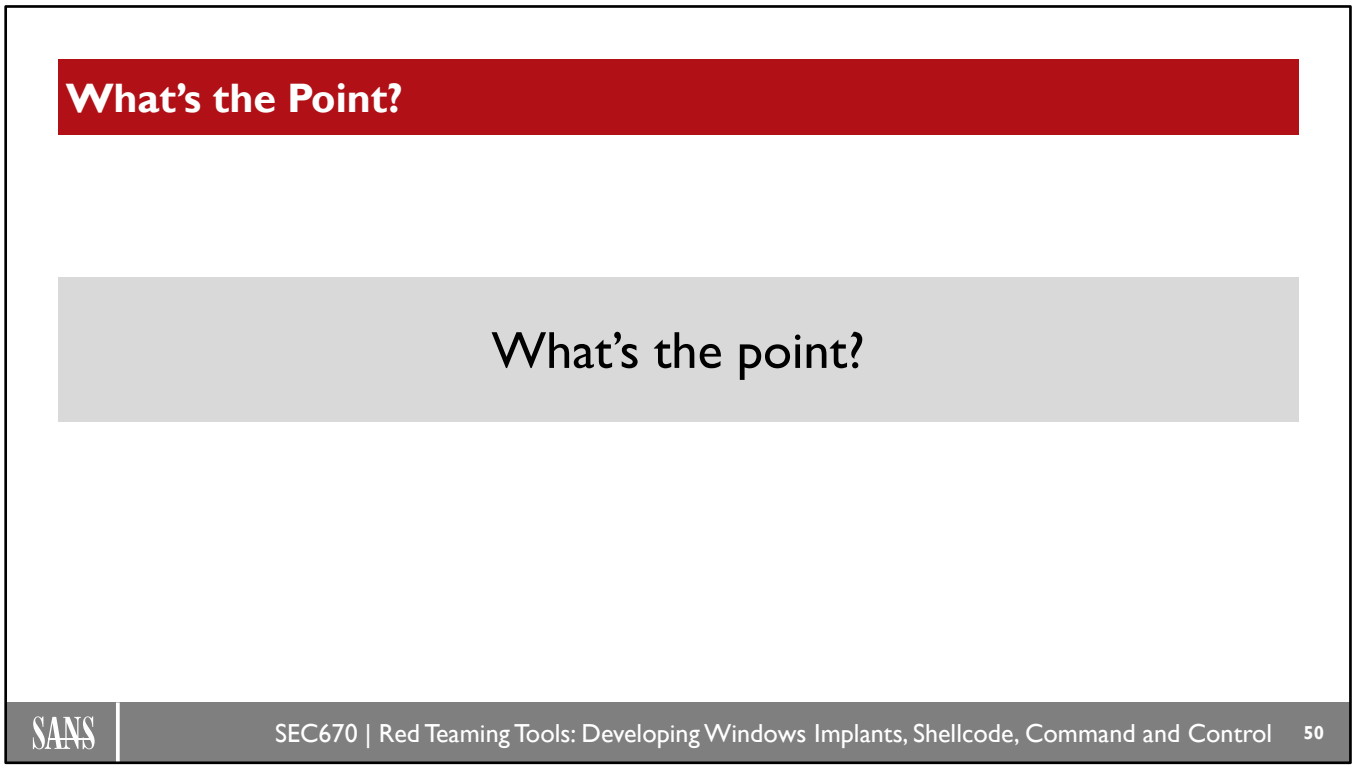
The example here intentionally omits error checking and the call to ***Process32First***, due to size limitations on the slide. Regardless, the main points are represented here, starting with the call to the ***CreateToolhelp32Snapshot*** function on the second line. We are only interested in capturing processes in this snapshot and we are not specifying a process ID as indicated by NULL. The function returns a handle value which is saved off in the *snapShot* variable. In the full code, this would be error checked against **INVALID_HANDLE_VALUE** to ensure the snapshot was created successfully. The next couple of lines create the *pe32* variable of struct type **PROCESSENTRY32** and since it has a size field, it should be set to the size of the struct. The next important part in this code snippet is the do while loop. The body of the loop is simply printing out the name of the executable and the corresponding process ID. The exit condition is met when the ***Process32Next*** function returns FALSE since it is a BOOL return type. To understand the process even more and what other information is available, research the **PROCESSENTRY32** definition in the TlHelp32.h header file.

Lab 2.3: CreateToolhelp

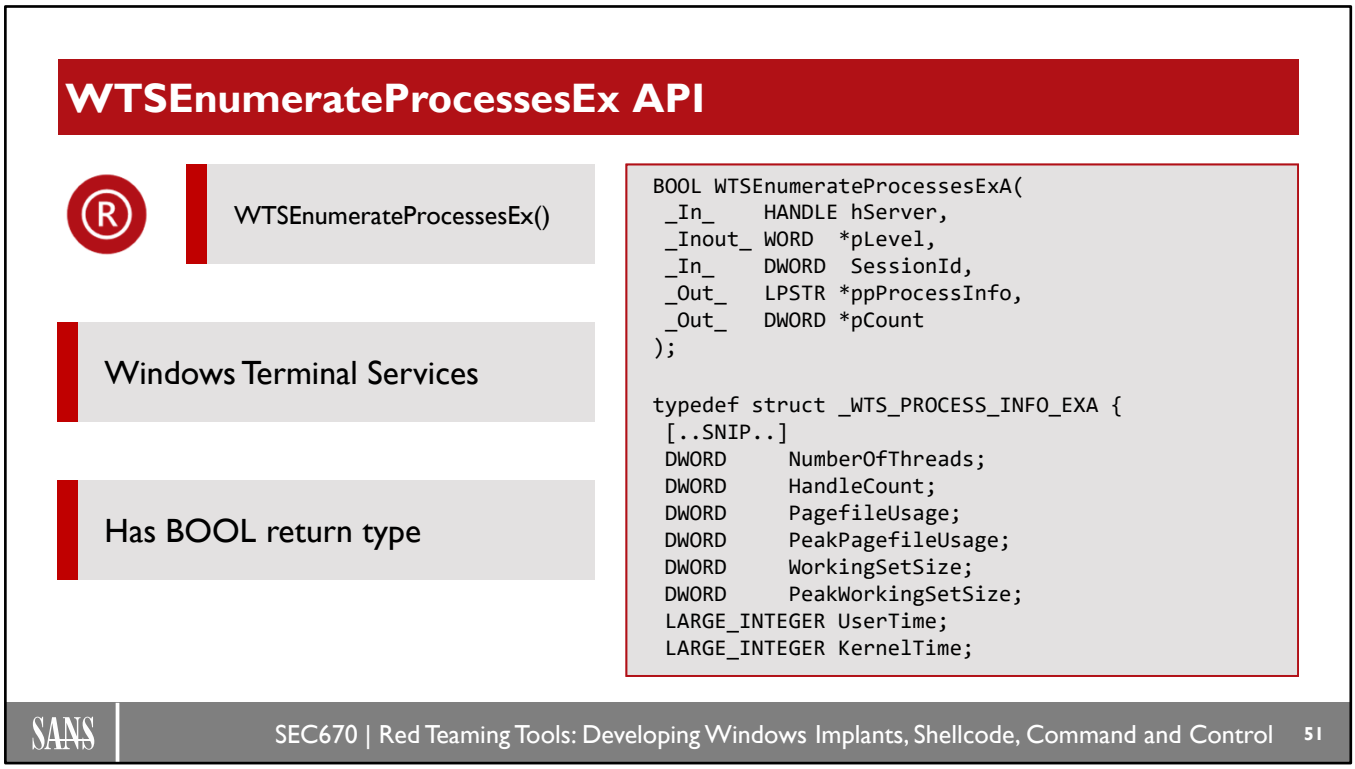
Using CreateToolhelp32Snapshot, enumerate the processes on the system.

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

Lab 2.3: CeateToolhelp
Please refer to the eWorkbook for the details of the lab.



What's the Point?
The point of the lab was to explore one of the more popular methods of enumerating processes. The major downside to this method is you can miss newly created processes after the snapshot has been taken.



WTSEnumerateProcessesEx API

There is an entire family of WTS APIs that are defined in the wtsapi32.h header file. The WTS prefix is the abbreviation for Windows Terminal Services, so think Remote Desktop. With that being said, a big advantage of using the WTS APIs for enumerating processes is that we could do so against remote systems that are configured for it. Specifically, there is a registry key that must be configured to enable the remote interrogation, but if it is there then remote process queries are possible. The API will return very detailed information about each process that is running on a local or remote system. The information about each process is returned via the *ppProcInfo* parameter, which is a pointer to a *WTS_PROCESS_INFO* struct. This will be an array of *WTS_PROCESS_INFO* struct entries. Let us go ahead and break down the API's parameters.

hServer, a handle returned by the ***WTSOpenServer*** API. The handle can be for a local or remote system.

pLevel, used to determine what level of information you would like returned. Passing 1 would give the extended version of the *WTS_PROCESS_INFO* struct.

SessionId, used when you would want to query a different session on the system. You can, of course, query all sessions, just pass in WTS_ANY_SESSION.

ppProcInfo, the out parameter so the function can file out the *WTS_PROCESS_INFO(Ex)* structure with the detailed information for the process.

pCount, the out parameter that will indicate how many structures were created by the API, in other words, the number of processes.

When you are done processing the information in the returned buffer, we must free it using the ***WTSFreeMemoryEx*** API.

Example: WTSEnumerateProcessesEx

```
if (!WTSEnumerateProcessesEx(WTS_CURRENT_SERVER_HANDLE, &dwLevel, WTS_ANY_SESSION, (PWSTR*)&procInfoEx, &dwCount))
{
    wprintf(L"WTSEnumerateProcessesExW failed with error: %d\n", GetLastError());
    return FALSE;
}

if (NULL == procInfoEx) { ... }

wprintf(L"%-20s [%5s] [%6s] [%7s] [%6s] [%-20s]\n", L"ImageName", L"PID", L"Threads", L"Handles", L"Session", L"UserName");
wprintf(L"================================================================================\n");

// iterate over the results
DWORD dwIndex = 0;
for (; dwIndex < dwCount; dwIndex++)
{
    auto pProcInfo = procInfoEx + dwIndex;
    wprintf(L"%-20.19s [%5d] [%7d] [%7d] [%7d] [%20s]\n",
            pProcInfo->pProcessName,
            pProcInfo->ProcessId,
            pProcInfo->NumberOfThreads,
            pProcInfo->HandleCount,
            pProcInfo->SessionId,
            (PCWSTR)GetUserNameFromSid(pProcInfo->pUserSid) );
}
```


Example: WTSEnumerateProcessesEx

The example here shows how to use the API to make a local query. The amazing part about this API is that it provides detailed information about each process. We can even query the session to see what session is tied to a certain process. This can lead down a path of injecting into processes that are in a different session. This method is called cross-session process injection. Unlike the ***EnumProceses*** API, this one can return the count to you, which is a nice touch. Just like the other API, once the count is known, we can iterate over the array and process the structures that the API created to represent each process. Inside the body of the for loop, you can see some of the processing of this information. It is inside the loop body that you can decide to perform other operations, like writing the output to a log file or getting the data ready to send back to your C2 server.

Lab 2.4: WTSEnum

Using WTSEnumerateProcesses, enumerate processes on the system.

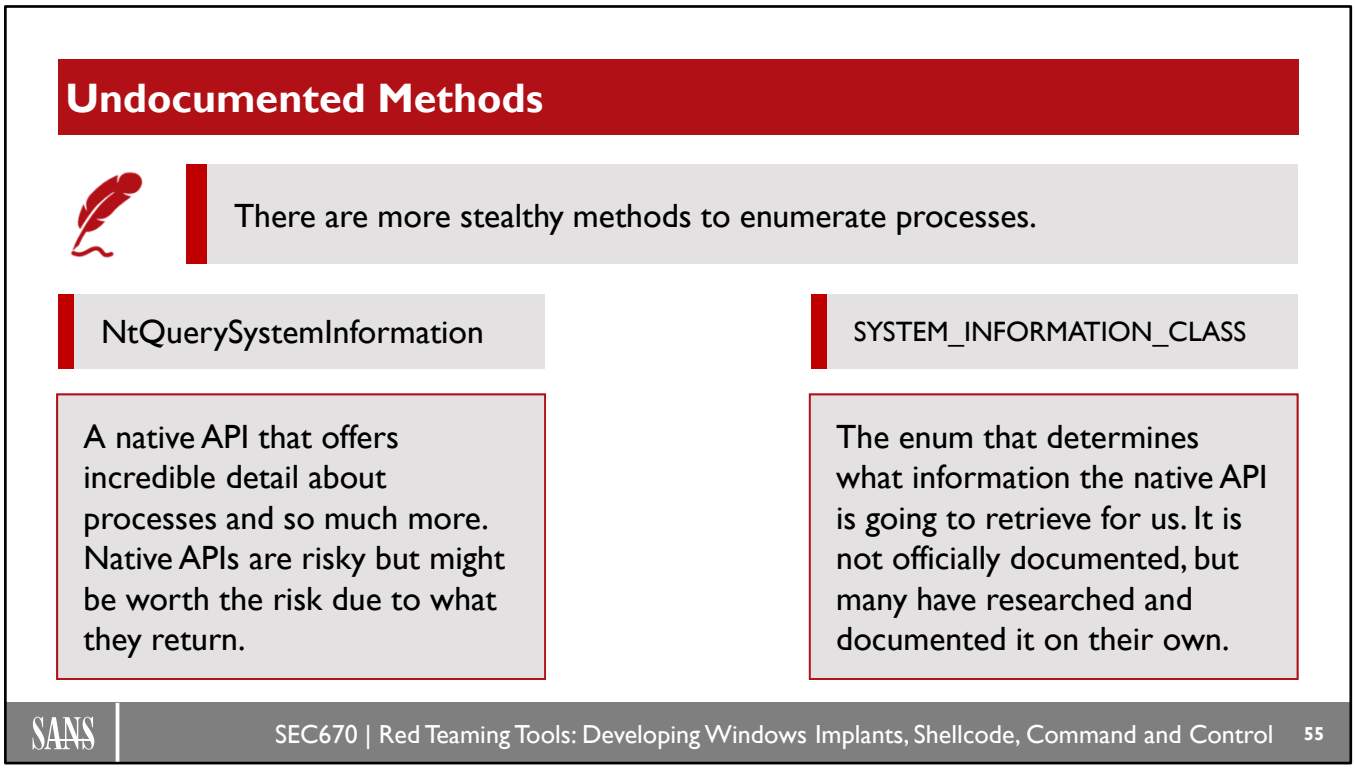
Please refer to the eWorkbook for the details of the lab.

Lab 2.4: WTSEnum
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of the lab was to explore another method to enumerate processes. Using the Windows Terminal Services is nice because you have the potential to query remote targets.



Undocumented Methods

Native APIs tend to be very risky to use as they are not officially documented by Windows. The APIs could change without Microsoft making an announcement of any kind, so your tool could work just fine on one version of Windows and then come time for an update, the native API you were using breaks, rendering your tool useless. When it comes to process enumeration, there is an amazing native API that we can use called ***NtQuerySystemInformation***. MSDN does have some documentation on it and the main description for it says, "**NtQuerySystemInformation** may be altered or unavailable in future versions of Windows. Applications should use the alternate functions listed in this topic." Despite that warning, we are going to use it anyway because we do not have to heed to their warning; we are creating an implant after all.

The ***NtQuerySystemInformation*** API relies heavily on the *SYSTEM_INFORMATION_CLASS* enum, which is not formally documented. In this undocumented enum are entries that we can use to specify what type of system information we are interested in seeing. The enum entry that would be of interest for process enumeration would be *SystemProcessInformation*. MSDN does not list every entry for the enum, but several GitHub projects, like the one for the x64dbg debugger, have kindly posted their hard efforts for us.

References:
https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation
https://github.com/x64dbg/x64dbg/blob/development/src/dbg/ntdll/ntdll.h

NtQuerySystemInformation API

NtQuerySystemInformation

Grabs specific information about the system

Has NTSTATUS return type

```
NTSTATUS
NtQuerySystemInformation(
 _In_ SYSTEM_INFORMATION_CLASS InfoCls,
 _Inout_   PVOID SystemInformation,
 _In_      ULONG SystemInformationLength,
 _Out_opt_ PULONG ReturnLength
);

// enum entry SystemProcessInformation

// SYSTEM_PROCESS_INFORMATION struct
```


NtQuerySystemInformation API

As mentioned on the previous slide, the ***NtQuerySystemInformation*** function is a native function as annotated by the Nt prefix in the function name. Even though MSDN has documentation online for this function, they do not recommend we use it. Instead, alternate functions are listed as suggestions to use to retrieve information about the system. The reasoning behind this is that the function could "break" or be removed from any future version release. This is not likely to happen, so we use this function anyway because it can return some very rich information not only about processes but also about the system. The function is extremely useful, and its true power comes from what SYSTEM_INFORMATION_CLASS is passed into it. To better understand how to use this function, we can break down the parameters in detail.

InfoCls is an abbreviation for *SystemInformationClass*, of type SYSTEM_INFORMATION_CLASS, which is a massive enum. Since we are talking about process information, we will be using the *SystemProcessInformation* enum entry. Each enum entry has a corresponding structure and ours will be the SYSTEM_PROCESS_INFORMATION struct. What will be returned to use is an array of these structures where each entry represents a process that has been mapped into memory.

SystemInformation, of type PVOID, is the pointer to the buffer to hold the information to be returned from the function. The size of this buffer must be known ahead of time, and it can also vary based on the information that is being requested. To get this information, the function will need to be called twice with the first call used just to get the size of the buffer. Then you can allocate a buffer of the correct size for the second call.

SystemInformationLength, of type ULONG, is the size of the buffer that the *SystemInformation* parameter points.

ReturnLength, of type PULONG, is not required to be passed, but if used, it will hold the actual size of the requested information.

SYSTEM_PROCESS_INFORMATION Struct

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
  ULONG NextEntryOffset;
  ULONG NumberOfThreads;
  LARGE_INTEGER WorkingSetPrivateSize;   // Since Vista
  ULONG HardFaultCount;                  // Since Windows 7
  ULONG NumberOfThreadsHighWatermark;    // Since Windows 7
  ULONGLONG      CycleTime;              // Since Windows 7
  LARGE_INTEGER  CreateTime;
  LARGE_INTEGER  UserTime;
  LARGE_INTEGER  KernelTime;
  UNICODE_STRING ImageName;
[..SNIP..]
  HANDLE UniqueProcessId;
  HANDLE InheritedFromUniqueProcessId;
  SYSTEM_THREAD_INFORMATION Threads[1];
[..SNIP..]
```


SYSTEM_PROCESS_INFORMATION Struct
Here is the struct as documented by the researchers and developers of x64dbg.

typedef struct _SYSTEM_PROCESS_INFORMATION
{
 ULONG NextEntryOffset;
 ULONG NumberOfThreads;
 LARGE_INTEGER WorkingSetPrivateSize; // Since Vista
 ULONG HardFaultCount; // Since Windows 7
 ULONG NumberOfThreadsHighWatermark; // Since Windows 7
 ULONGLONG CycleTime; // Since Windows 7
 LARGE_INTEGER CreateTime;
 LARGE_INTEGER UserTime;
 LARGE_INTEGER KernelTime;
 UNICODE_STRING ImageName;
 KPRIORITY BasePriority;
 HANDLE UniqueProcessId;
 HANDLE InheritedFromUniqueProcessId;
 ULONG HandleCount;
 ULONG SessionId;
 ULONG_PTR UniqueProcessKey; // Since Vista (requires SystemExtendedProcessInformation)
 SIZE_T PeakVirtualSize;
 SIZE_T VirtualSize;
 ULONG PageFaultCount;
 SIZE_T PeakWorkingSetSize;
 SIZE_T WorkingSetSize;
 SIZE_T QuotaPeakPagedPoolUsage;
 SIZE_T QuotaPagedPoolUsage;

```
SIZE_T QuotaPeakNonPagedPoolUsage;
  SIZE_T QuotaNonPagedPoolUsage;
  SIZE_T PagefileUsage;
  SIZE_T PeakPagefileUsage;
  SIZE_T PrivatePageCount;
  LARGE_INTEGER ReadOperationCount;
  LARGE_INTEGER WriteOperationCount;
  LARGE_INTEGER OtherOperationCount;
  LARGE_INTEGER ReadTransferCount;
  LARGE_INTEGER WriteTransferCount;
  LARGE_INTEGER OtherTransferCount;
  SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

Example: NtQuerySystemInformation

```
Status = NtQuerySystemInformation(
    SystemProcessInformation,
    SystemInformation,
    SystemInformationLength,
    &ReturnLength);
// error check again
if (!NT_SUCCESS(Status)) { ... }

auto SystemProcess = (PSYSTEM_PROCESS_INFORMATION)SystemInformation;
ULONG dwIndex = 0;
for (; ; dwIndex)
{
    // convert the handle to a ULONG
    auto Pid = HandleToULong(SystemProcess->UniqueProcessId);
    // if the PID is 0 it is for the Idle process, else we have another process we can grab the name for it
    auto Name = Pid == 0 ? L"[Idle]" : CString(SystemProcess->ImageName.Buffer, SystemProcess->ImageName.Length / sizeof(WCHAR));
    auto SessionId = SystemProcess->SessionId;
    auto HandleCount = SystemProcess->HandleCount;
    auto ThreadCount = SystemProcess->NumberOfThreads;
    auto NPP = SystemProcess->QuotaNonPagedPoolUsage;

    wprintf(L"%-5d  %-20.19s  [%7d]  [%7d]  [%7d]  [%10lld]\n", Pid, (LPCWSTR)Name, SessionId, HandleCount, ThreadCount, NPP);
```


Example: NtQuerySystemInformation

Whenever you decide to use native APIs, you are also deciding whether to accept the risk that comes with the usage. They do offer amazing benefits, like increasing your chances of bypassing AV/EDR detection and being able to obtain more detailed information like what this example shows. Hackers take risks all the time and developers must take risks too to increase the chances of a red team operator being successful for the operation at hand. Once the API call has been made, we have a linked list that we can iterate over. Each entry in the list is of type PSYSTEM_PROCESS_INFORMATION, and we can use it and some of its members to keep advancing to the next entry until there are no more entries. There is so much information that it will not fit on the slide, but the best way to see all this information is online using the MSDN documentation and some of the third party sources, like x64dbg's GitHub repo.

Bottom line: certain native functions are well worth the risk based solely on what they can offer.

Module Summary

- Discussed the reason for enumerating processes
- Explored the structures the kernel uses to represent processes
- Explored various methods for process enumeration

Module Summary

In this module, we discussed why it is important to enumerate processes on a system, the structures the kernel uses to represent processes in system address space, and several methods to enumerate processes on local and remote systems.

Unit Review Questions

What undocumented API can be used to enumerate processes?

A EnumProcesses()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Questions
Q: What undocumented API can be used to enumerate processes?

A: EnumProcesses()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Unit Review Answers

What undocumented API can be used to enumerate processes?

A EnumProcesses()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Answers
Q: What undocumented API can be used to enumerate processes?

A: EnumProcesses()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Unit Review Questions

What API could be used to obtain more information about a process given only its PID?

A. OpenProcess()

B. WTSEnumerateProcessesEx()

C. NtQuerySystemInformation()

Unit Review Questions
Q: What API could be used to obtain more information about a process given only its PID?

A: OpenProcess()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Unit Review Answers

What API could be used to obtain more information about a process given only its PID?

A	OpenProcess()
B	WTSEnumerateProcessesEx()
C	NtQuerySystemInformation()

Unit Review Questions
Q: What API could be used to obtain more information about a process given only its PID?

A: OpenProcess()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

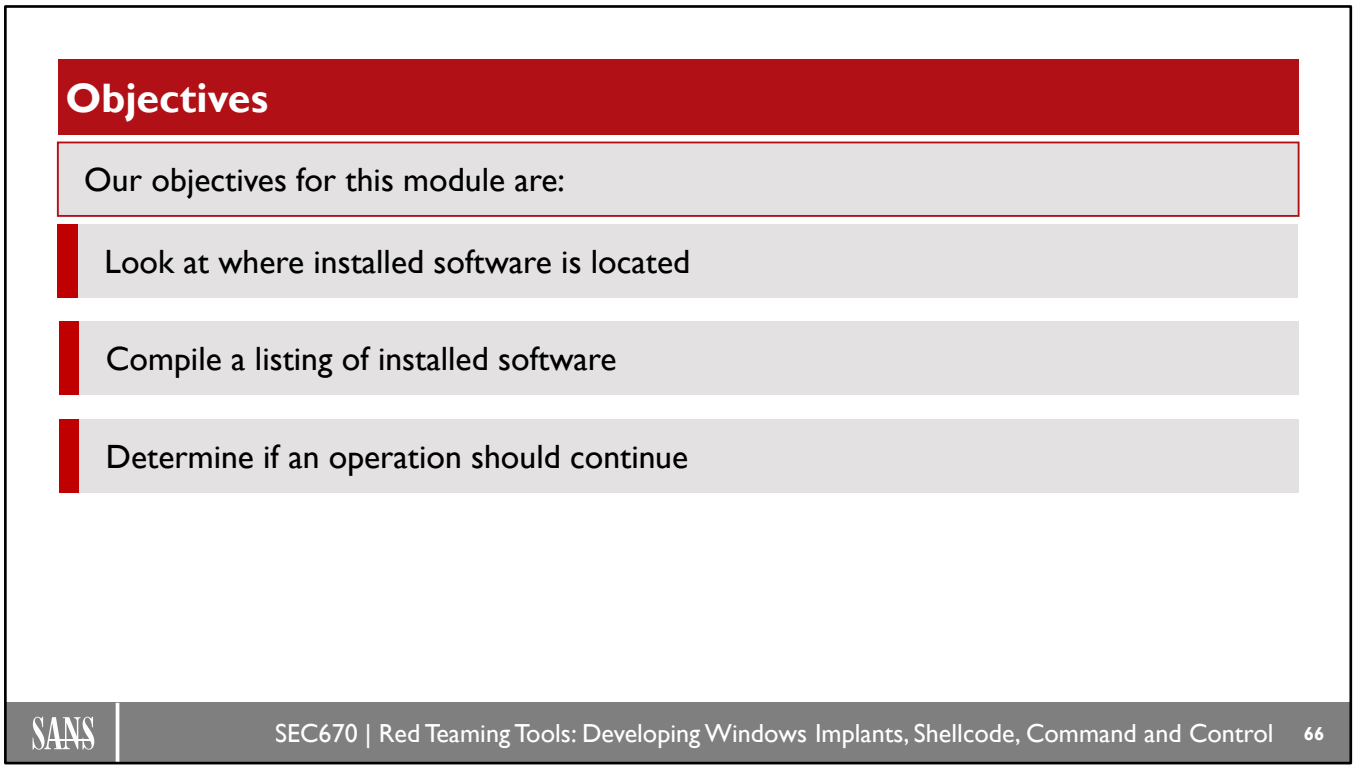
Services and Tasks

Network Information

Registry Information

Bootcamp

Finding installed software can tell you a great deal about a target. Let us dive in, shall we?



Objectives
The objectives for this module are to know where to look for installed software, compile a listing of all installed programs, and determine if an operation should continue given the presence, or absence, of software.

Where to Look?

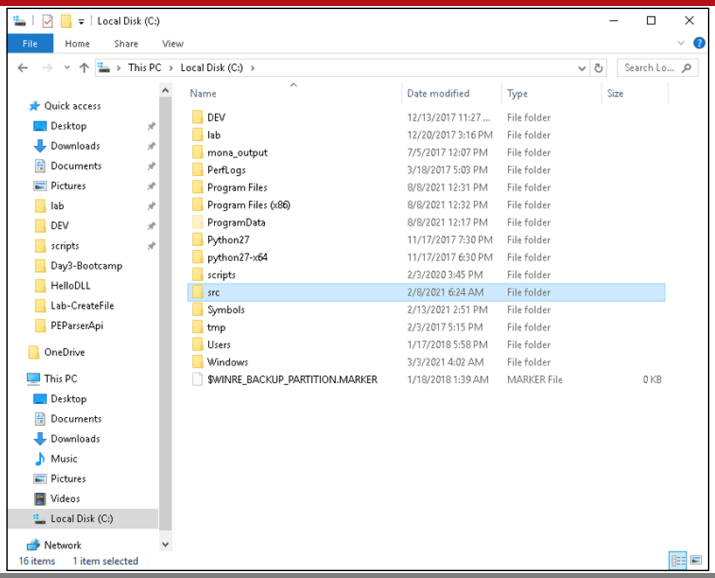
Where can you find 32-bit and 64-bit applications?

C:\Program Files	C:\Program Files (x86)	C:\
It should be safe to assume that entries found in this folder are 64-bit applications	A similar assumption can be made for entries found in this location; that they will be 32-bit applications	Some apps, like Python, install at the root system drive, although not very common

Where to Look?

One of the goals of recon is to determine what applications are installed on your target. Maybe you want to see if an already known vulnerable application is there, or perhaps you want to make sure a certain application is not installed. In either case, it is good to know where to find the installation folders. The assumption here is that the target is a 64-bit Windows installation. Based on that assumption, 64-bit applications will be located at the "C:\Program Files" directory or the C:\Progra~1 for using the 8.3 short name convention that NTFS supports. The next folder name really gives away what the purpose is: "C:\Program Files (x86)". The x86 portion in the folder name indicates that 32-bit applications are in this folder. The root of the system drive, typically represented by the letter C:\, does contain some entries of applications that have been installed there. Python 2.7 used to be one such example, among others. With all this being said, users that are going through an installation process typically can choose where to install the application. If your survey tool is only checking these three locations, you might miss one if a user decided to install an application in their Documents or Downloads folder.

C:\

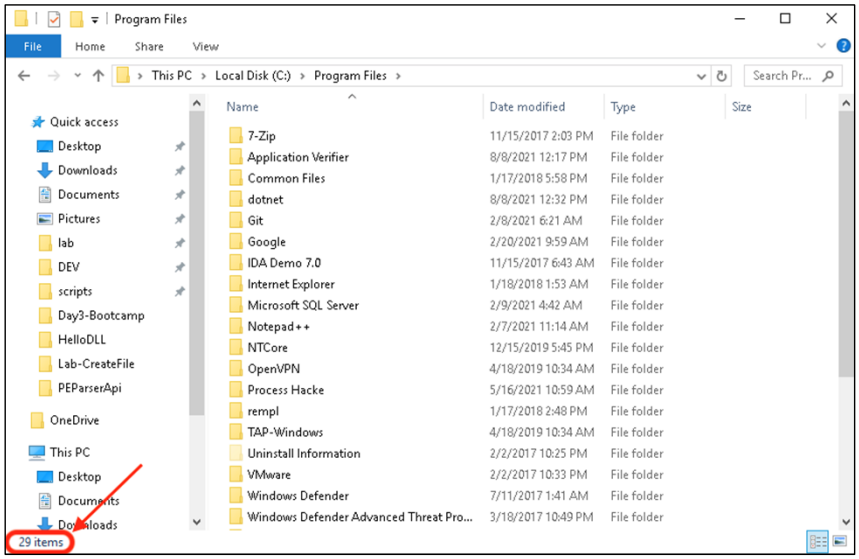


C:

The system's root drive is typically annotated by the drive letter C, but it can be another letter. Because it can be any letter, it is best to not hard code file paths until you have determined the system root drive. Once done, you can append your file paths to that returned value. Back to the current topic, as you can see from the screenshot, there is not a lot being listed here as far as figuring out what software might be installed. We do have Python27 as a dead giveaway, but beyond that we would have to look elsewhere for our information.

One thing to note with folders like these would be permissions, because not everyone can write to these folders without the proper permissions.

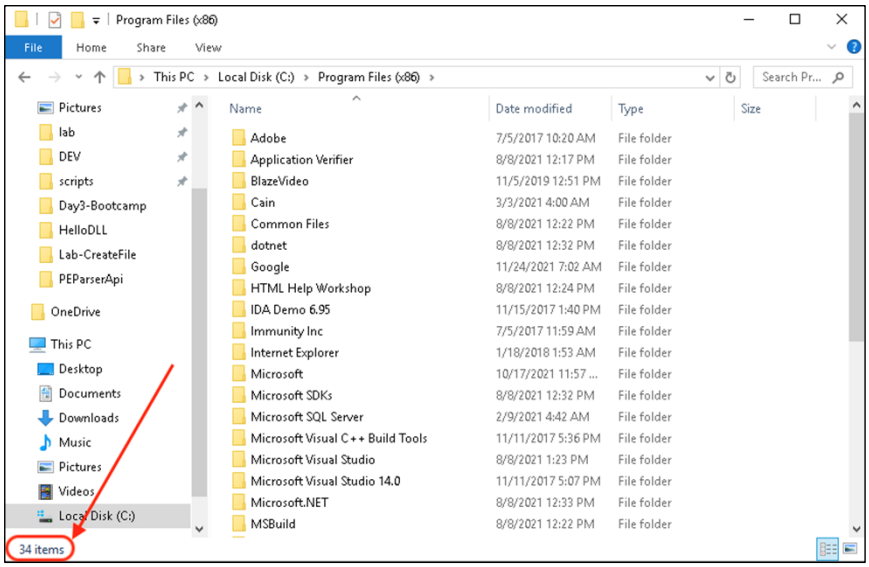
C:\Program Files



C:\Program Files

This screenshot shows the contents of the Program Files directory. The directory contains 29 entries, which could be a possible indicator that there are at least 29 applications that have been installed on this system; at *least* 29 items because some folders could easily hold other programs. If your tool was collecting this information then it would allow a red team operator to get a glimpse as to what 64-bit applications are here, and based on the applications, a guess could be made as to what the system's purpose is. Seeing applications like Notepad++, Process Hacker, VMware, etc. could indicate that this machine could be a research VM. This screenshot was taken from a Windows 10 Dev VM.

C:\Program Files (x86)



C:\Program Files (x86)

This screenshot shows the contents of the Program Files x86 directory. This directory contains 34 entries, which could indicate that there are at least 34 applications installed on this system that are 32-bit. An operator might be able to make a better educated guess as to what this system is being used for after seeing several entries for development software.

Should Operations Continue?

When should you abort an operation?

Aborting an operation based solely on a single application being installed is quite the decision to make. If you have no idea what the application does or what it would do if you drop more tools on the system, it could be a good decision to back off. This would allow you more time to conduct some research and hit the target later, possibly.

Should Operations Continue?

When deciding if you should continue with an operation, there is not always a clear yes or no answer. The more enumeration you do, the more informed your decision could be and the more confident you can feel about the decision. In a makeshift scenario, let us say you have gathered the following information: OS info/version, service pack/hotfixes, processes, and installed software. You might have enough information to make an informed decision. If you noticed that there was an application that detects your tool, then perhaps it would be a good idea to back off and find a different system to target. For the applications that you have never heard of before and have not had any time to research, you might have to decide if the risk of bringing more tools down is worth the reward. All of this should be done ahead of time, as much as possible. That way, when you come to a decision point, you already know what to do. The discussions could happen in an operational pre-brief where your team has a chance to brainstorm together. GO and NOGO criteria can be made during these pre-briefs where you know if it is okay to continue or if you have to back off right away.

Module Summary

Explored where applications can be installed

Learned to make a decision to continue or abort based on the listing of software

Module Summary
In this module, we explored where some applications might be installed, and we also discussed how GO/NO-GO decisions can be made based on gathered information.

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will discuss a feature for enumerating directories. Many implants today already implement a directory listing feature, so we should put one in ours.

Objectives

Our objectives for this module are:

- Understand how to enumerate files in a directory
- Understand how to implement a recursive directory walk; dirwalk
- Learn how to locate a specific file of interest

Objectives

The objectives for this module are to understand how to enumerate files in a directory, understand how to implement a recursive directory walk, and learn how to locate a specific file of interest.

Directory Enumeration

Directory listings is nothing new.

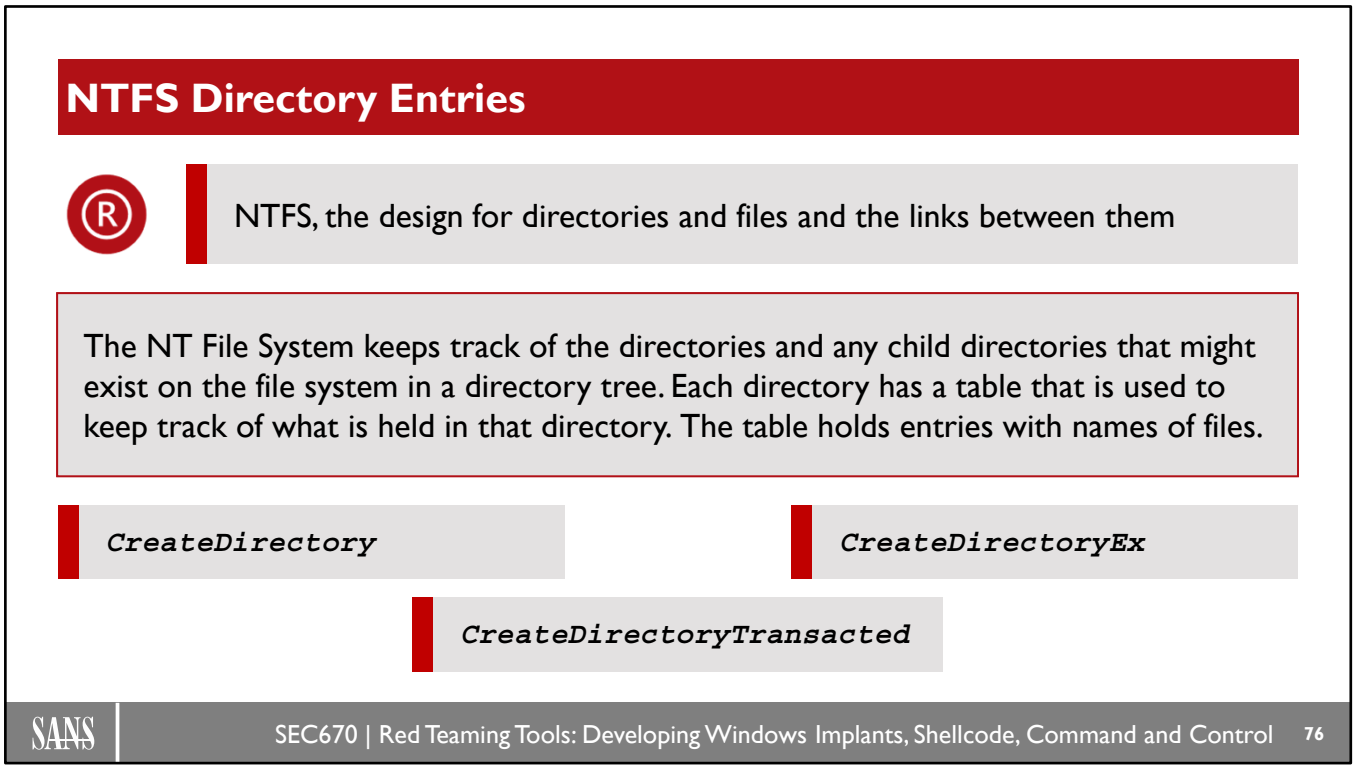
Many popular frameworks have implants that can perform directory listings. The famous Meterpreter session from the Metasploit Framework offers operators the ability for perform a directory listing. Native Windows binaries also perform directory listings, so it is not a behavior that should be categorized as malicious or suspicious.

Directory Enumeration

Directory enumeration is a very simple feature to implement programmatically, and it can be done in a number of different ways. MSDN provides a few simple examples of using the primary APIs involved and it is easy to build upon those examples. If you are familiar with the Metasploit Framework and its Meterpreter session, you might be familiar with its *ls* command to list the contents of a directory. Of course, *ls* is a native Linux command, but Meterpreter implements this as a platform agnostic function for Windows or Linux in their source code. The C source code for Meterpreter's *ls* command uses the exact same Windows APIs that we will be discussing in this module and for our lab. You can browse to their GitHub page to look at the code to get an idea of what is happening behind the scenes.

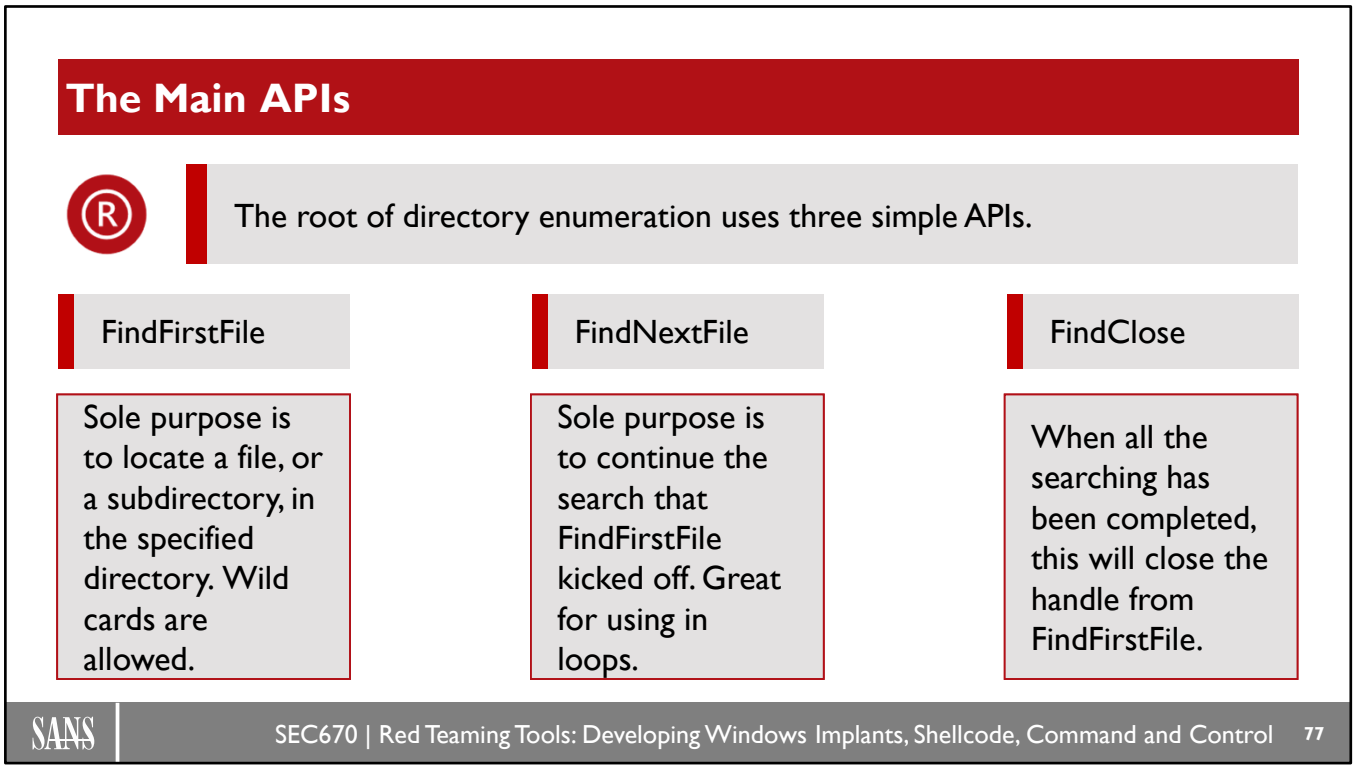
Enumerating directories is an important feature to implement because you would want to give your operators the ability to see what is in a certain directory. Perhaps the operator will want to download a file of interest or want to see what files are currently in a folder before making the decision to drop a file of their own in that same folder. The operator will do whatever in the end, but we must at least give an operator the ability to perform a directory listing.

Here is the URL to Raymond Chen's article for BFS: https://devblogs.microsoft.com/oldnewthing/20050203-00/?p=36533.



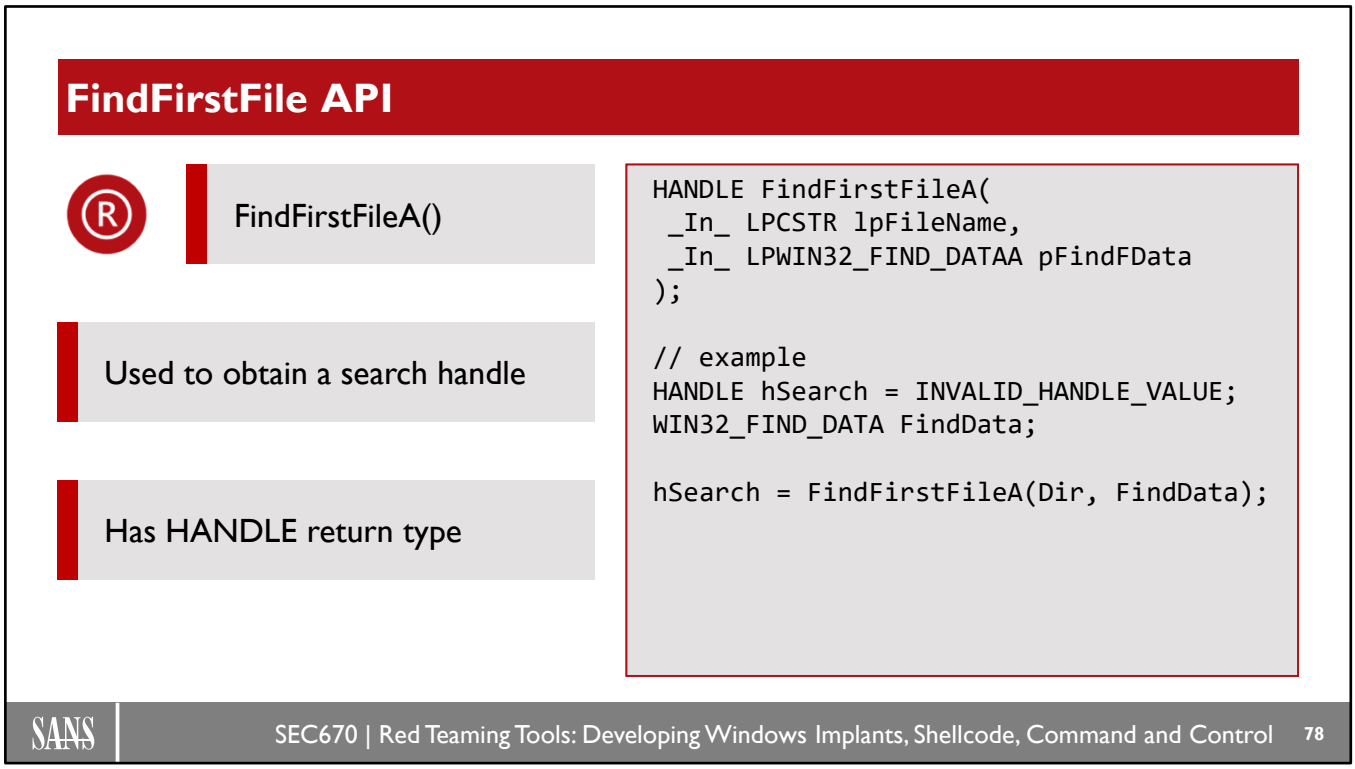
NTFS Directory Entries

You might already be familiar with how Explorer shows directories and the files in them, but what really makes that happen? When a directory is created via one of the three APIs listed on the slide—***CreateDirectory***, ***CreateDirectoryEx***, or ***CreateDirectoryTransacted***—it will have a corresponding directory table made for it. Whenever a file is created or moved into this directory, an entry will be placed into the table, and the entry will have the name of the file. There can be multiple entries for the same file in the directory table called links. A hard link is created when there is another entry in the table that is made for the same file. Trying to view the table itself is not important to us as that starts to dive down into the world of forensics. We will simply be using the APIs to do the querying of the tables for us.



The Main APIs

The act of enumerating directories is not malicious, nor should it even be deemed as suspicious. There are several native Windows applications that have this functionality built into it. Explorer must do this very same thing; in fact, we could search through many executables in the System32 folder to determine how many of them import these functions. CTF players could whip up a directory enumeration script to locate flag files or whatever their objective is. Nation State actors might enumerate directories to see if any files are worth downloading for further analysis. Actors that might be interested in espionage might conduct recursive directory walks looking for blueprints of engine designs for fighter jets, ships, stealth technology, etc. To get this done, there are two primary APIs that do the bulk of the work: ***FindFirstFile*** and ***FindNextFile***. ***FindClose*** is simply to clean things up when the search is done. The APIs also have extended versions like ***FindFirstFileEx***, so we could pass in extra attributes that the file should have. There is also a transacted API like ***FindFirstFileTransacted***, though it is not recommended to be used.



FindFirstFile API
The first API that we need to use to kick of the directory walk is none other than the ***FindFirstFile*** API. Like most other APIs that have string arguments, it is just a macro that is expanded to support Unicode or ANSI depending on your project settings. The example on the slide is using the ANSI version. Should you have to deal with non-English characters, like Chinese, you better use the Unicode versions. The main purpose of the API is to return a search handle that the ***FindNextFile*** API would then use to continue searching for a file in a directory. The ***FindFirstFileA*** function has just two parameters, which are broken down below.

lpFileName, of type LPCSTR, is the directory and filename that should be searched. The filename can have wildcards in it too, which is nice when you do not know an extension for a file.

pFindFData, of type LPWIN32_FIND_DATAA, is the pointer to a WIN32_FIND_DATA structure that will be filled out by the API as it retrieves the information about a file or subdirectory.

If the function succeeds, a file search handle will be returned and the WIN32_FIND_DATA structure will start to be filled out as you progress. On failure, it will return INVALID_HANDLE_VALUE, so you would have to call ***GetLastError*** to determine what really was the cause.

WIN32_FIND_DATA Struct

```
typedef struct _WIN32_FIND_DATAA {
 DWORD    dwFileAttributes;
 FILETIME ftCreationTime;
 FILETIME ftLastAccessTime;
 FILETIME ftLastWriteTime;
 DWORD    nFileSizeHigh;
 DWORD    nFileSizeLow;
 DWORD    dwReserved0;
 DWORD    dwReserved1;
 CHAR     cFileName[MAX_PATH];
 CHAR     cAlternateFileName[14];
 DWORD    dwFileType;
 DWORD    dwCreatorType;
 WORD     wFinderFlags;
} WIN32_FIND_DATAA, *PWIN32_FIND_DATAA, *LPWIN32_FIND_DATAA;
```


WIN32_FIND_DATA Struct

The WIN32_FIND_DATA structure is filled with useful information. Many of the struct members do not need any explanation, like FileAttributes, CreationTime, LastAccessTime, LastWriteTime, etc. One that might need some explaining is the AlternateFileName with the fixed size of 14. This is for the short file naming conventions that NTFS supports. Specifically, it is an 8.3 convention with the filename containing 8 letters, then the dot, then the 3-letter extension. As an example, if you have a long file name, like backup-picture.jpg, it could look like this: BACKUP~1.JPG. The other members are not that interesting to us, so we can simply ignore them.

FindNextFile API

FindNextFileA()

Used to continue a search

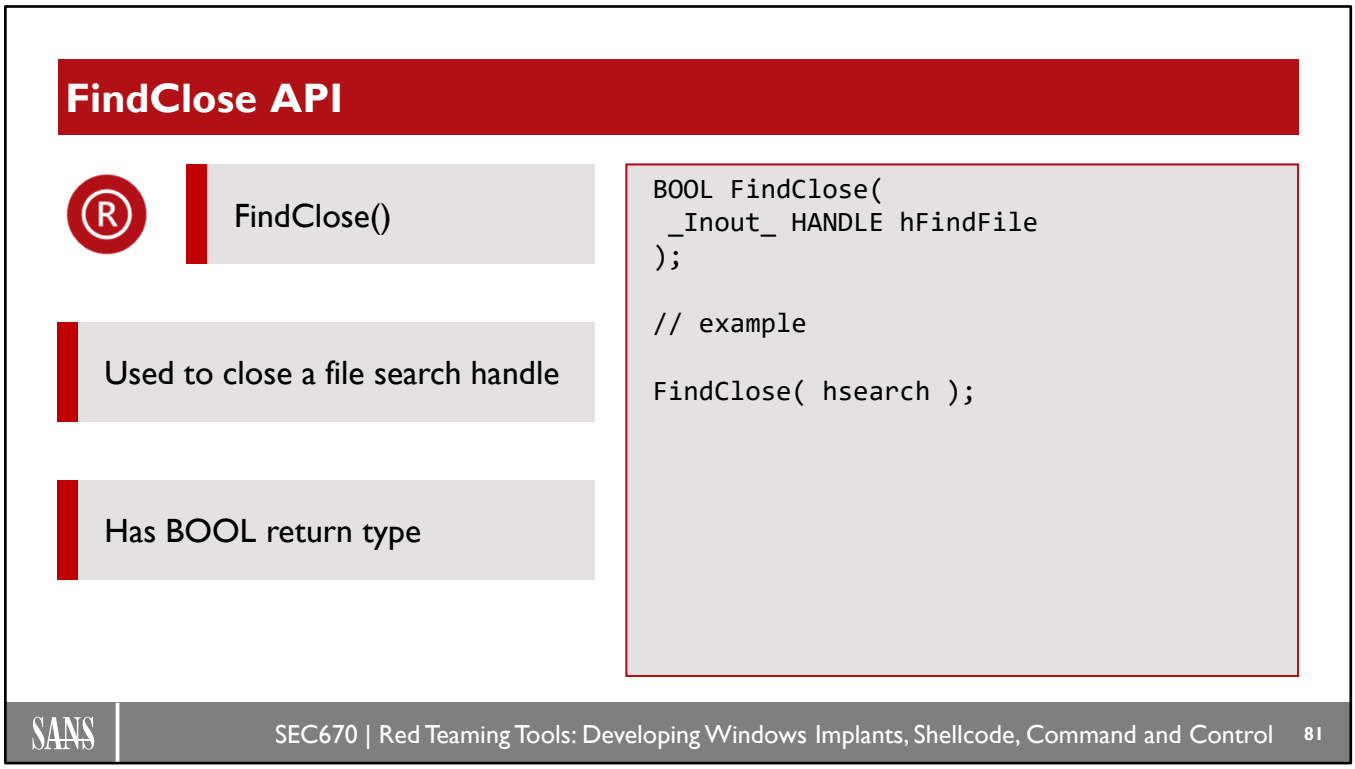
Has BOOL return type

```
BOOL FindNextFileA(
 _In_ HANDLE hFindFile,
 _In_ LPWIN32_FIND_DATAA pFindFileData
);

// example

do {

 // do stuff with the info

} while (
  FindNextFileA(hSearch, FindData) != 0
);
```


FindNextFile API

The next API we need to use to implement our directory walk is the ***FindNextFile*** API, which is really a macro that expands to ***FindNextFileA*** for ANSI, or ***FindNextFileW*** for Unicode. For this example, we will be using the ANSI version of the macro. The ***FindNextFileA*** function will not work on its own—it must have the file search handle that the ***FindFirstFileA*** function returns. Just like ***FindFirstFileA***, this function has two parameters: a handle to a valid file search handle, and a pointer to a WIN32_FIND_DATAA structure. Since handles are nothing new to us at this point and since the structure is the same one that is used in the ***FindFirstFileA*** function, there is no need to break down the parameters. Because the return type is BOOL, it is perfect to use in a loop like a while loop. Each iteration of the loop for each file found in a directory would have the structure filled out for the respective file. When the function returns false, the loop would break and be done.

If the function fails for whatever reason, the structure may not have been filled out properly, so it would be best to call ***GetLastError*** to see what really happened.



FindClose API

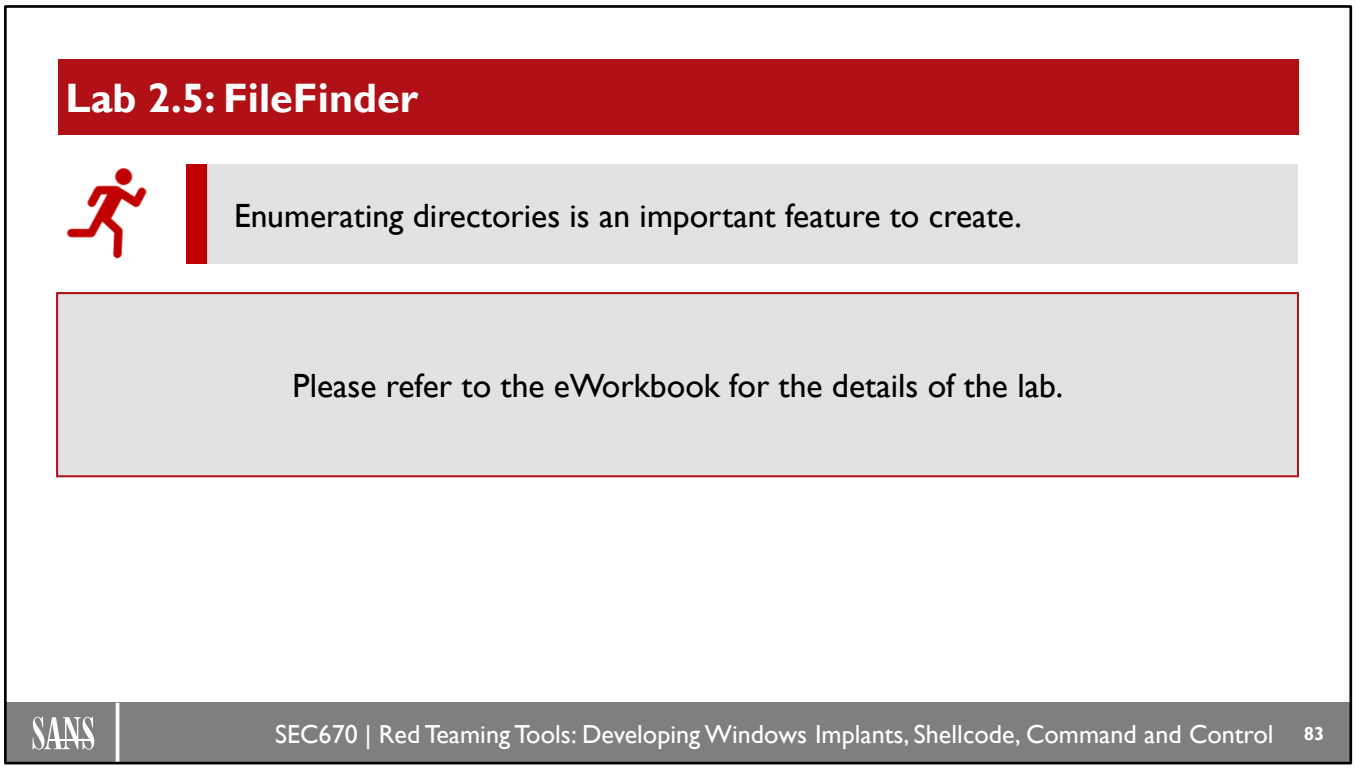
After you are done searching for files or performing your directory walk, the file search handle should be closed out. The ***FindClose*** API can do this for us, and it is a very simple API to understand and implement in code. ***FindClose*** only takes one argument and that is a valid file search handle. The return type is BOOL so you could check to see if the function was successful or not, and if it was not successful, you would call ***GetLastError*** to see the details.

Example: FindFirstFile, FindNextFile, FindClose

```
// attempt to enumerate the hard links to the file
hFileToFind = FindFirstFileW(fileName.GetBuffer(), &findData);
// error check
if (INVALID_HANDLE_VALUE == hFileToFind) { ... }
else
{
    do
    {
        // do work...
    } while (FindNextFileW(hFileToFind, &findData));
}
```


Example: FindFirstFile, FindNextFile, FindClose

The short example on the slide shows very basic usage for the two main APIs that are involved in enumerating a directory. The ***FindFirstFileW*** API is used to kick off the process of enumeration. The first argument being passed to the API is a CString type that has a method ***GetBuffer()*** to get a pointer to the buffer. This is done to satisfy the requirement of the API. The function can fail, so be sure to check for success for failure. The next part is the do/while loop that will continue as long as the ***FindNextFileW*** function keeps returning TRUE or 1. In the body of the do/while loop is where your processing of each entry would be done. You can do matching if you are looking for a certain file name, extension type, skip directories, etc. If you are keeping a list of everything being discovered, this is where you would be appending to that list.



Lab 2.5: FileFinder
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was the explore how you can programmatically enumerate a directory to find a file, and if you had time, enumerate any subdirectories.

Module Summary

Discussed why we would perform a directory walk

Learned how to perform a directory walk to find a file

Discovered the main APIs involved

Module Summary
In this module, we discussed the how to perform a directory walk and why we would do one in the first place. A recursive walk from the system root could take a while, but at the same time it could yield some great information. There were only three APIs involved with this, but there were two that did the heavy lifting: ***FindFirstFile*** and ***FindNextFile***.

Unit Review Questions

What user-mode structure holds the attributes of a file?

A WIN32_FIND_DATA

B KUSER_SHARED_DATA

C FILE_OBJECT

Unit Review Questions
Q: What user-mode structure holds the attributes of a file?

A: WIN32_FIND_DATA

B: KUSER_SHARED_DATA

C: FILE_OBJECT

Unit Review Answers

What user-mode structure holds the attributes of a file?

A WIN32_FIND_DATA

B KUSER_SHARED_DATA

C FILE_OBJECT

Unit Review Answers
Q: What user-mode structure holds the attributes of a file?

A: WIN32_FIND_DATA

B: KUSER_SHARED_DATA

C: FILE_OBJECT

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This module will discuss the importance and benefits of gathering information about the users of a system. The module will of course discuss the Windows APIs that might be involved with retrieving user information.

Objectives

Our objectives for this module are:

- Discuss the importance of gathering user information
- Understand how to programmatically gather information about users

Objectives
The objectives for this module are to understand how to programmatically gather information about users and discuss the importance behind it.

User Information

Who's who on the system

It is always a good idea to see what users are on the system. Limited privileged users are one thing, but finding out if a user is part of the Administrators group is great. You might even get lucky enough to see a Domain admin logged into a system!

User Information

Another part of conducting recon is gathering user account information. From the command line, there are some common tools that can be used to do this, like the *net* command line utility. The *net* command does more than gather user information but if you pass in the *net user* or the *net localgroup* options, it will gather user information and groups on the system, among other things. Using the command line is nice when you have a shell, but we can also do this programmatically. Why do we care about users in the first place? Well, if a user is part of the Administrators group, that could give us a solid option of escalating our privileges or attempting a UAC bypass. If the system is part of an Active Directory domain, something you would find in most mature Windows environments, you might see a domain admin logged into the system. The hash for that account would be great to grab and then further pivot around the domain. Dumping creds and hashes for accounts can help with our lateral movement, or as some say our East and West movement, something that SEC565 dives into.

GetUserName API

GetUserNameA()

Used to obtain the current username

Has BOOL return type

```
BOOL GetUserNameA(
 _Out_   LPSTR   lpBuffer,
 _Inout_ LPDWORD pcbBuffer
);

// example

PSTR userName;
DWORD cbSize = 32767;

GetUserNameA(userName, &cbSize);
```


GetUserName API

The ***GetUserName*** API is yet another macro that can expand to support ANSI or Unicode, depending on project settings. Keeping with the trend of showing ANSI versions, we will break down the ANSI version of this API. ***GetUserNameA*** requires two parameters: *lpBuffer*, and *pcbBuffer*. We can take a look at those in more detail below.

lpBuffer, of type LPSTR, is a pointer to a buffer that will end up storing the username.

pcbBuffer, of type LPDWORD, is a pointer to a variable that will indicate the size of the buffer pointed to by lpBuffer.

The function will indicate success by returning nonzero. Failures result in zero, which does not give us any real insight as to why it may have failed. For that, we would have to call our favorite error function ***GetLastError***. There are a few caveats with this function in that it will not return the username that one might expect. For example, threads have the ability to impersonate tokens/users and as such, the API would return the username of the client that is currently being impersonated.

GetUserProfileDirectory API

GetUserProfileDirectoryA()

Used to obtain the root directory of the user's profile

Has BOOL return type

```
USERENVAPI
BOOL
GetUserProfileDirectoryA(
 _In_      HANDLE  hToken,
 _Out_opt_ LPSTR   lpProfileDir,
 _In_opt_  LPDWORD lpcchSize
);
```


GetUserProfileDirectory API

The ***GetUserProfileDirectory*** API is useful when you would like to know the path of the root folder for the username that was passed into the function. The ***GetUserProfileDirectory*** API is another macro, so we should know what that means by now. The function only has one required parameter and two optional parameters. Even though a few of the parameters are optional, we can dive into the details of each one.

hToken, of type HANDLE, is a valid handle to a token for the user that can be gathered from calling a function like ***OpenProcessToken*** or ***OpenThreadToken***.

lpProfileDir, of type LPSTR, is a pointer to a variable that will hold the path to the user's profile directory.

lpcchSize, of type LPDWORD, is a pointer to the size of the buffer pointed to by lpProfileDir.

One nice thing about this function is that if your *lpProfileDir* buffer is not big enough, the needed size will be placed in *lpcchSize* by the function. This would allow you to make the call a second time using the correct size. On that note, your first call for functions like these should be with a zero-sized buffer. This would make the function fail, forcing it to give us the proper size. Once you have that, you can make the second call with the correct size. This is the preferred way and the most reliable method.

NetUserEnum API

NetUserEnum()

Used to obtain information about all user accounts

Has NET_API_STATUS return type

```
NET_API_STATUS
NET_API_FUNCTION
NetUserEnum(
 _In_    LPCWSTR servername,
 _In_    DWORD   level,
 _In_    DWORD   filter,
 _Out_   LPBYTE *bufptr,
 _In_    DWORD   prefmaxlen,
 _Out_   LPDWORD entriesread,
 _Out_   LPDWORD totalentries,
 _Inout_ PDWORD  resume_handle
);
```


NetUserEnum API

Getting account information about a single user account is fine, but it is also great to get account information about all user accounts on a system. The ***NetUserEnum*** API is perfect for this because it will return information from all user accounts on the local or remote system. The information returned from the API might seem similar to the output you may have seen after executing the *net user* command. The API has several parameters that need some explaining.

servername, of type LPCWSTR, is the pointer to a constant wide char string for the server. Passing in NULL here would indicate the local computer.

level, of type DWORD, is the level of information you intend to see. There are seven values that can be passed in here:
- 0 - is for user account names. There will be an array of USER_INFO_0 structures.
- 1 - is for user account details. There will be an array of USER_INFO_1 structures.
- 2 - is for account details and logon information. There will be an array of USER_INFO_2 structures.
- 3 - is for all of the above and now profile information. The array will have USER_INFO_3 structures.
- 10 - is for user account names and comments. The array will have USER_INFO_10 structures.
- 11 - is for more detailed account information. The array will have USER_INFO_11 structures.
- 20 - is for the user's names account attributes. The array will have USER_INFO_20 structures.

filter, of type DWORD, is used to specify what accounts should be included in the search.

bufptr, of type LPBYTE, is the pointer to the buffer that will end up holding the returned information. You are required to free this buffer using ***NetApiBufferFree*** when done, even if the API fails.

prefmaxlen, of type DWORD, is the maximum length of the data, in bytes. We do not need to specify a value here other than MAX_PREFERRED_LENGTH so that the function makes the proper space for the data.

entriesread, of type LPDWORD, is a pointer to the variable that will hold the number of entries the function queried.

totalentries, of type LPDWORD, is a pointer to the variable that will hold the number of entries that could have been queried from a position called the resume position.

resume_handle, of type PDWORD, is a pointer to a variable that is used as the resume handle. The resume handle can be used to continue searching user accounts and if this is what you want to do, then zero (0) should always be used for the first call. If you do not care about this, then passing NULL here is just fine.

NetLocalGroupEnum API

NetLocalGroupEnum()

Used to obtain local group information

Has NET_API_STATUS return type

```
NET_API_STATUS
NET_API_FUNCTION
NetLocalGroupEnum(
 _In_    LPCWSTR  servername,
 _In_    DWORD    level,
 _Out_   LPBYTE  *bufptr,
 _In_    DWORD    prefmaxlen,
 _Out_   LPDWORD  entriesread,
 _Out_   LPDWORD  totalentries,
 _Inout_ PDWORD_PTR resumehandle
);
```


NetLocalGroupEnum API

In addition to gathering user information, we can gather information about the groups that might be present on a local or remote system. The ***NetLocalGroupEnum*** API is pretty much the equivalent to executing the *net localgroup* command from the command line. Perhaps the API should have been named NetLocalOrRemoteGroupEnum since it is not bound to local systems only. The API has almost the exact same parameters to the ***NetUserEnum*** API except for the level parameter. The level parameter only has two possible options: 0 and 1. Level 0 is for returning only the names of the local groups, which will be an array of LOCALGROUP_INFO_0 structures. Level 1 is for requesting the group names and the comments that are tied to them, if any. There will be an array of LOCALGROUP_INFO_1 structures for Level 1. Outside of the level parameter, the remaining ones are the same as the ***NetUserEnum*** API.

Additional Information

Additional headers and APIs that could be of interest

`lmaccess.h`	`lmuse.h`
`NetGroupGetUsers`	`NetUseEnum`
`NetLocalGroupGetMembers`	`NetUseGetInfo`

Additional Information

The lmaccess and the lmuse header files offer additional APIs that might be of interest when querying user and user group information. The listing on the slide is not an exhaustive list but merely a small sampling of what else is out there that can be used. Depending on the information that you are wanting to gather, you can implement the logic to get it. The more you add the more robust your survey tool will be for operators using it.

Source Code Review

Source code review!

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

Module Summary

Discussed reasons to gather user information

Explored several APIs that allow us to retrieve user and group information

Module Summary
In this module, we discussed why it might be important to gather information about the user and groups on systems. We also took a detailed look at several of the APIs that are available for us to use.

Unit Review Questions

What is one API to obtain a username?

A GetUserName()

B NetUserEnum()

C NetLocalGroupEnum()

Unit Review Questions
Q: What is one API to obtain a username?

A: GetUserName()

B: NetUserEnum()

C: NetLocalGroupEnum()

Unit Review Answers

What is one API to obtain a username?

A GetUserName()

B NetUserEnum()

C NetLocalGroupEnum()

Unit Review Answers
Q: What is one API to obtain a username?

A: GetUserName()

B: NetUserEnum()

C: NetLocalGroupEnum()

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

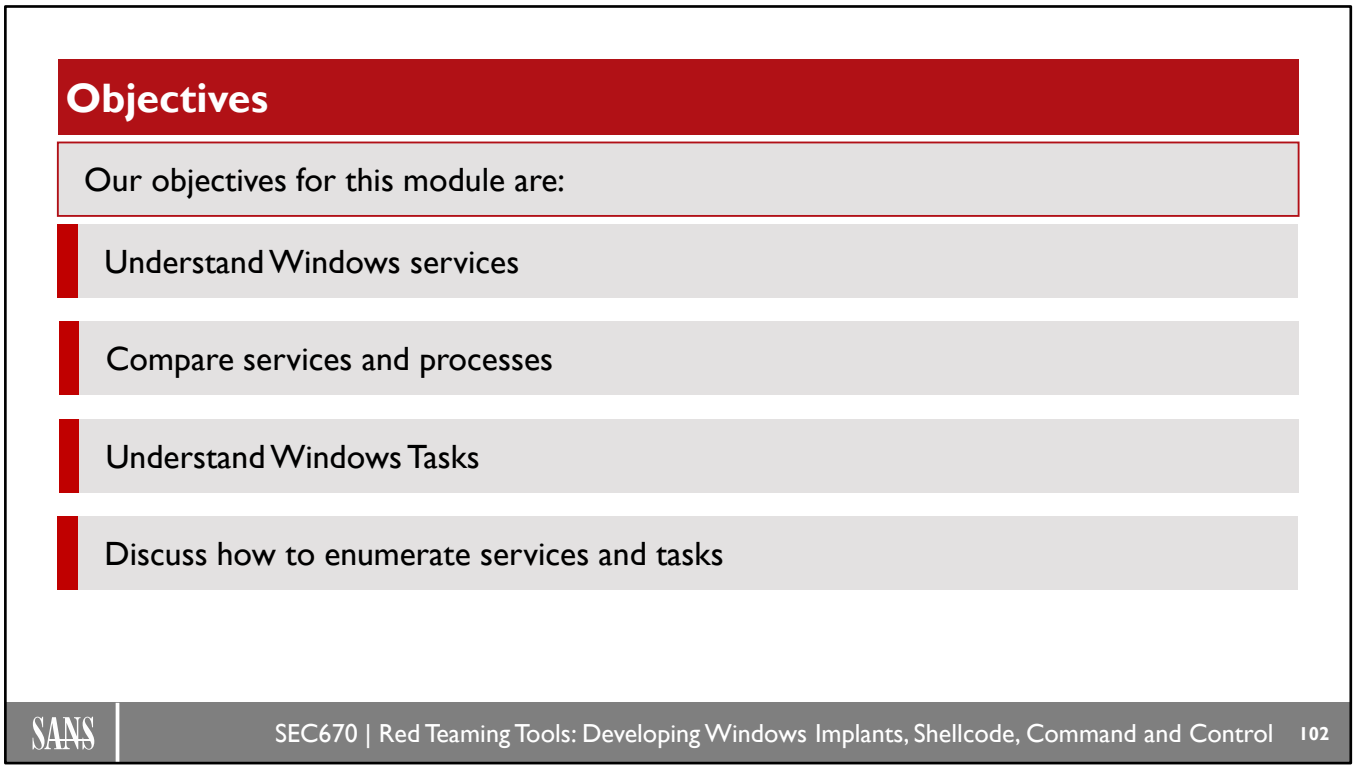
Services and Tasks

Network Information

Registry Information

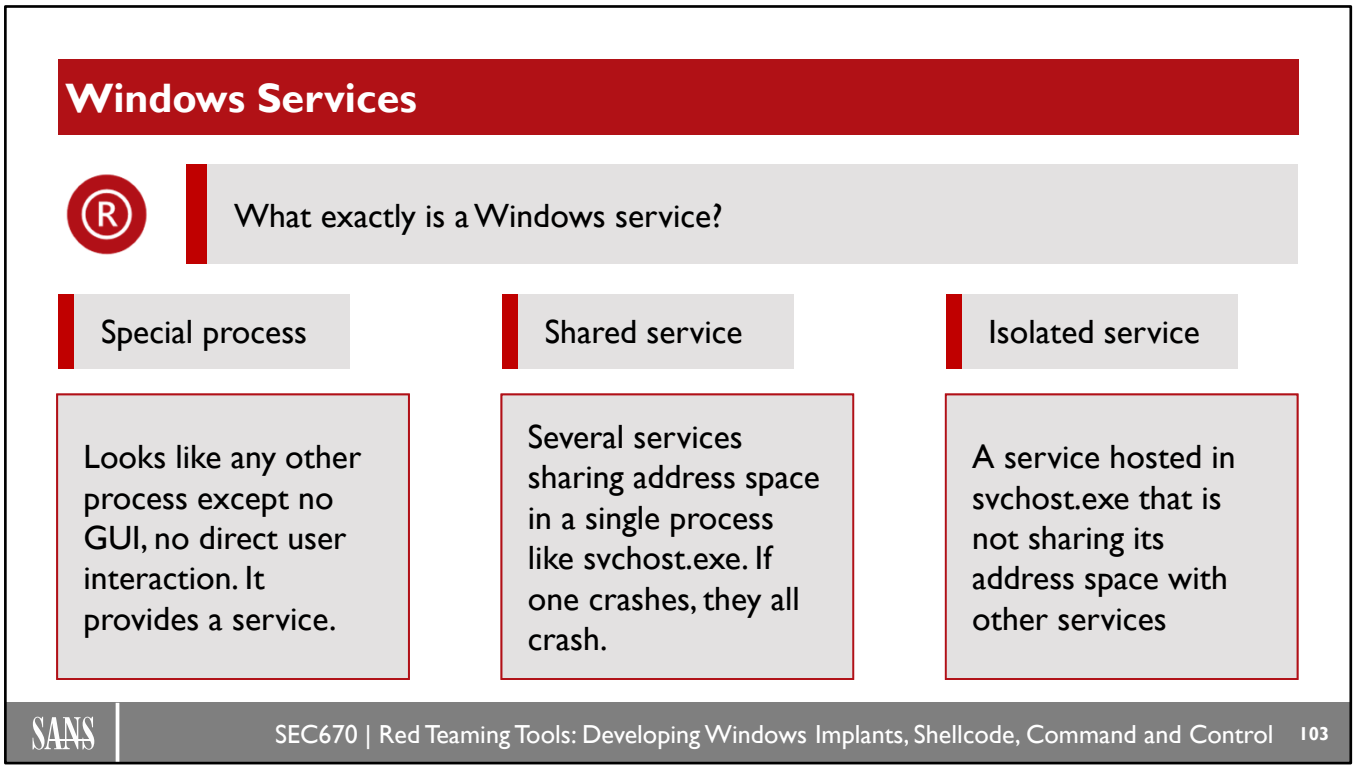
Bootcamp

In this module, we will discuss how to enumerate services and tasks during the execution of your survey tool.



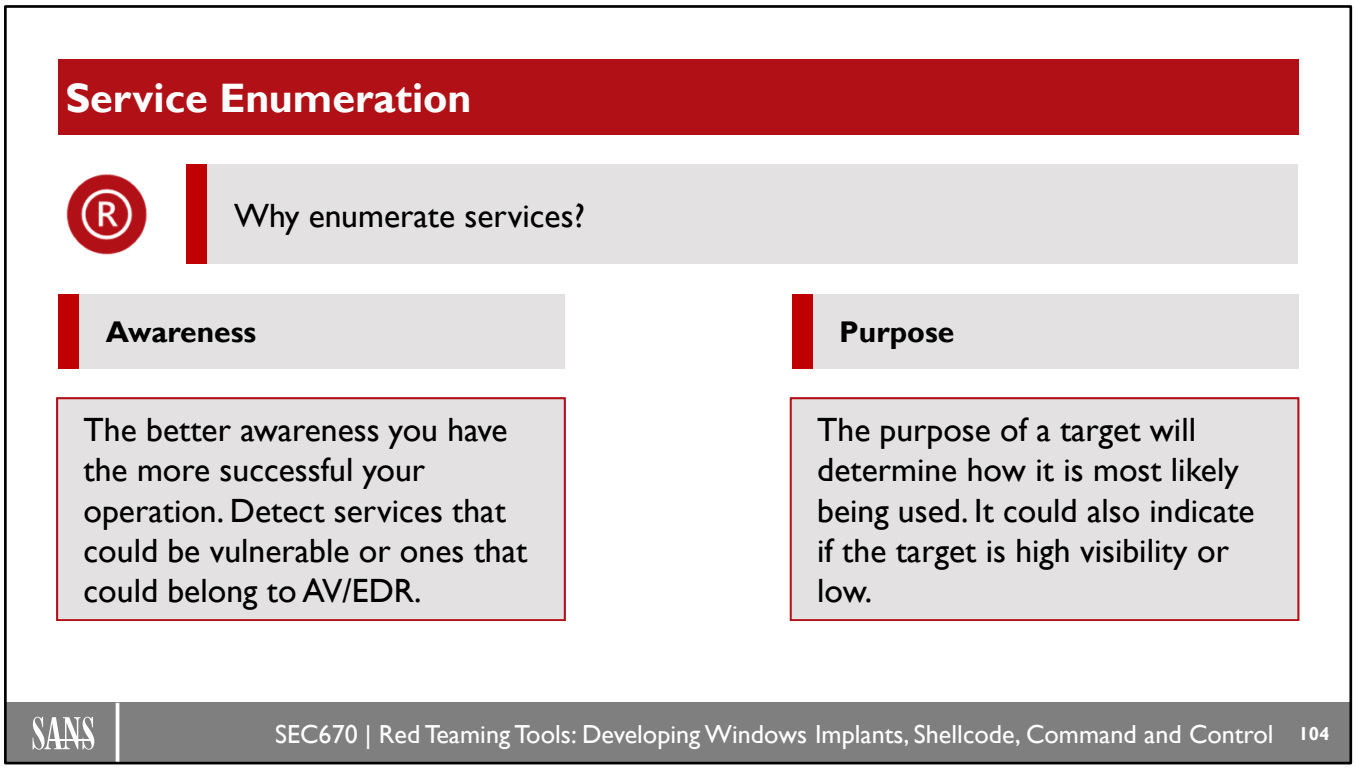
Objectives

The objectives for this module are to understand what a Windows service is, compare services and processes, understand what Windows Tasks are, and how to enumerate them all.



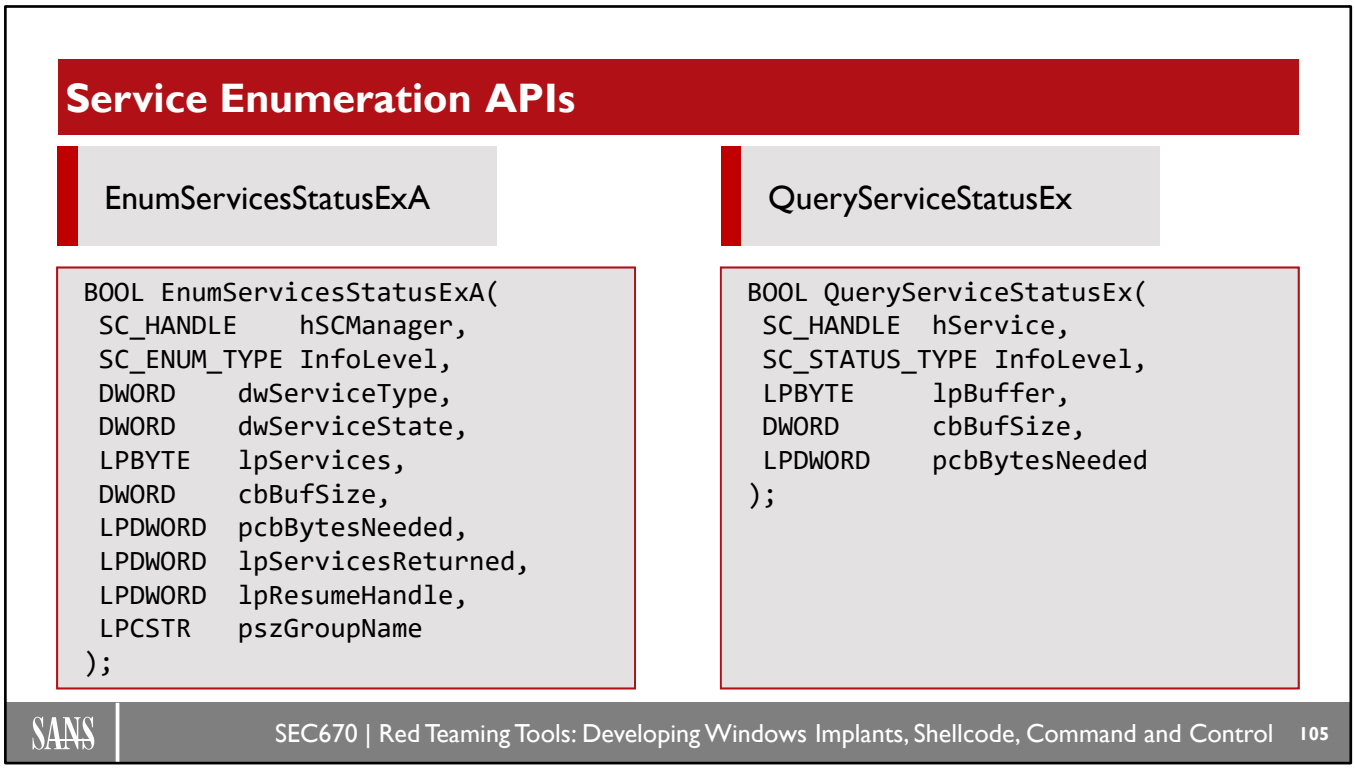
Windows Services

Even if you are not that familiar with Windows, you might be familiar with services in general. Linux has services too, typically called daemons, and they provide a certain set of services to users, like FTP access to some FTP server. Windows services are not that different as they still provide a service to users. Windows services are just special processes and they typically do not have a GUI. Most services do not have direct configuration or interaction with users on the system. The services operate in the background and mind their own business until a user needs whatever functionality they provide. The most interaction a user might have with a service is to perform some action against it, like Stop or Start actions. Another interesting item with Windows services is that they can share address spaces. The svchost.exe process acts like a container of sorts so that several services can be hosted in a single svchost.exe process. The downside of shared services is that if one service crashes for some abnormal reason, like exploit attempts, the other services go down with it. This is where isolated services come to play. They are services that can be hosted in a single svchost.exe process but they will not be sharing the address space with any other service. Later in the course, we will look at how we can make that choice programmatically when we create our own service.



Service Enumeration

We typically conduct service enumeration for similar reasons that we conduct process enumeration. Services, just like processes, can tell us what a target's purpose is or how it is being used. Certain services would be specific to servers like DHCP, DNS, FTP, to name a few, and you would not expect to find those services running on a client workstation. AV/EDR products can also have services running that handle certain portions of its functionality like scanning, file submission, watchdog services, etc. Finding those could be beneficial in making the decision to continue operating on that target. Again, if you did not expect that your target would have an AV solution installed, it might not be the best idea to continue and bring down additional tools. Another reason for service enumeration is to identify possible vulnerable services that could be exploited for privilege escalation and/or persistence, both of which will be talked about later in the course in detail.

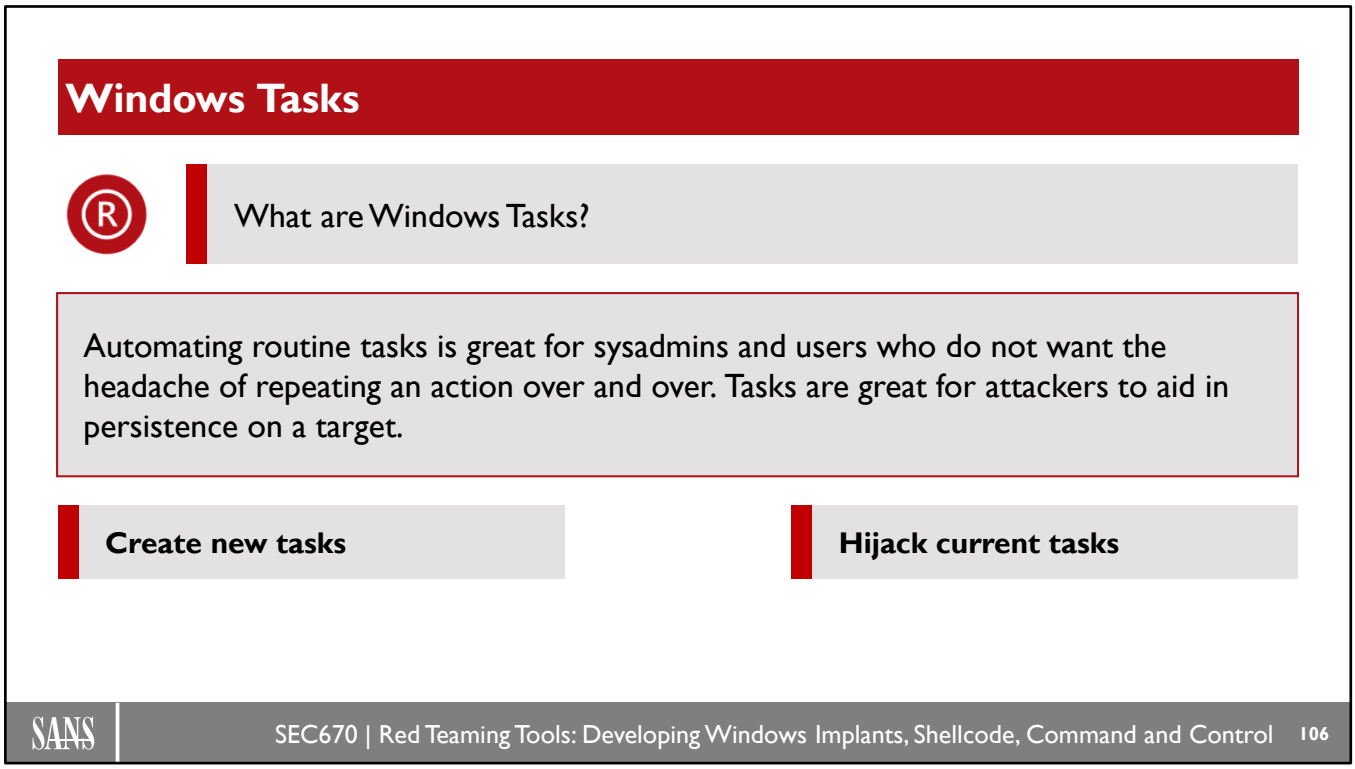


Service Enumeration APIs

How exactly can we go about enumerating services? You might know how to do this via the command line using the sc.exe utility or with the PowerShell cmdlet Get-Service, but we are doing this programmatically in C. We have two options at our disposal for getting this done. We could create WMI queries to do this or we can use a few Win32 APIs to achieve similar results. For this instance, we will be using the service-specific Win32 APIs instead of using wmi-specific Win32 APIs. ***EnumServicesStatusEx()*** and ***QueryServiceStatusEx()*** are two great APIs that can help you enumerate services and gather more detailed information about them. The first function on the slide is ***EnumeServicesStatusExA()*** and it accepts a decent number of arguments. The function will query the Service Control Manager database of either a local target or a remote target for services and will return the name and status of each service, at a minimum. More information and data can possibly be gathered depending on the InfoLevel value passed in as an argument, which the only supported InfoLevel value documented at the time of this writing is SC_ENUM_PROCESS_INFO.

The other great service-specific API to use is ***QueryServiceStatusEx()***. It does not enumerate services but rather, it will obtain the status of a service that has already been enumerated. The two APIs used in conjunction with each other can provide valuable information about services on the target. The InfoLevel for this function, similar to ***EnumServicesStatusEx***, only supports one value: SC_STATUS_PROCESS_INFO.

In Section 3, we will dive much deeper into services when we discuss local damage that can be inflicted against a target.



Windows Tasks

Admins, developers, hackers, etc. tend to be lazy in the sense that all would rather script something than do it manually. Tasks are a great way to help with this because they can perform certain actions when a specific trigger condition is met. Some of the triggers could be a user logging on to the system, the system itself booting up, certain event logs, etc. From an attacker's perspective, tasks would be a great option for maintaining access to the target. Attackers can either create a new task to achieve their persistence needs or hijack current tasks to be a bit less noisy. Hijacking currently scheduled tasks could be done by adding an additional action after the default action. For enumeration purposes, we are interested in seeing what tasks are currently registered and possibly already running. Perhaps we find a task that could be of interest to hijack, or we find evidence that we are not the first ones on the target, i.e., a current task for persistence made by another attacker. Tasks might not always be the most useful, but when conducting survey scripts, it is great to be thorough.

Please note: when it comes to keeping a low profile—something you should try to keep—creating new tasks could raise your profile a good amount. Create new tasks with caution.

Enumerating Tasks v. 1.0

Using COM to enumerate Tasks

ITaskScheduler::Enum

```
HRESULT Enum(
 _Out_ IEnumWorkItems **ppEnumWorkItems
);
```

IEnumWorkItems::Next

```
HRESULT Next(
_In_  ULONG    celt,
_Out_ LPWSTR **rgpwszNames,
_Out_ ULONG   *pceltFetched
);
```


Enumerating Tasks v. 1.0

For us to enumerate tasks on a system we must turn to COM. There is an exposed interface called ***TaskScheduler***, specifically ***ITaskScheduler***, that has a method called ***Enum***, which will allow us to create an enumeration object. Then, with that object in hand, we can create ***IEnumWorkItems*** interface that allows us to enumerate whatever tasks might be present at the time the code is ran. According to MSDN, here are the high-level steps to getting this done:

- Initialize the COM library using ***CoInitialize.***
- Use ***CoCreateInstance*** to make the Task Scheduler object.
- Call the Enum method of ***ITaskScheduler*** to create the enumeration object.
- Call the Next method of the ***IEnumWorkItems*** to enumerate tasks.
- Free the resources using ***CoTaskMemFree.***

Module Summary

- Discussed what Windows services are
- Learned about services and processes
- Discussed what Windows Tasks are
- Discussed enumerating services and tasks

Module Summary
In this module, we discussed very briefly what services and tasks are. We also discussed why we would want to enumerate them, and the APIs involved for enumerating services and tasks.

Unit Review Questions

What does SERVICE_WIN32_OWN_PROCESS indicate?

A. The service shares its address space with other processes

B. The service does not share its address space with other processes

C. The service will be hidden from view

Unit Review Questions
Q: What does SERVICE_WIN32_OWN_PROCESS indicate?

A: The service shares its address space with other processes

B: The service does not share its address space with other processes

C: The service will be hidden from view

Unit Review Answers

What does SERVICE_WIN32_OWN_PROCESS indicate?

A The service shares its address space with other processes

B The service does not share its address space with other processes

C The service will be hidden from view

Unit Review Answers
Q: What does SERVICE_WIN32_OWN_PROCESS indicate?

A: The service shares its address space with other processes

B: The service does not share its address space with other processes

C: The service will be hidden from view

Unit Review Questions

How do you get the COM library ready for use in your process?

A CoCreateInstance

B CoInitialize

C CoMemFree

Unit Review Questions
Q: How do you get the COM library ready for use in your process?

A: CoCreateInstance

B: CoInitialize

C: CoMemFree

Unit Review Answers

How do you get the COM library ready for use in your process?

A CoCreateInstance

B CoInitialize

C CoMemFree

Unit Review Answers
Q: How do you get the COM library ready for use in your process?

A: CoCreateInstance

B: CoInitialize

C: CoMemFree

Unit Review Questions

What COM interface can be called to create an enumeration object?

A ITaskScheduler

B IUnknown

C IBelieve

Unit Review Questions
Q: What COM interface can be called to create an enumeration object?

A: ITaskScheduler

B: IUnknown

C: IBelieve

Unit Review Answers

What COM interface can be called to create an enumeration object?

A ITaskScheduler

B IUnknown

C IBelieve

Unit Review Questions
Q: What COM interface can be called to create an enumeration object?

A: ITaskScheduler

B: IUnknown

C: IBelieve

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This module will look at how to gather information about the network and the target's network configurations.

Objectives

Our objectives for this module are:

- Gather network information
- Gather NIC configurations

Objectives
The objectives for this module are to determine how to gather any network information we can, as well as the target's NIC configurations.

Network Information

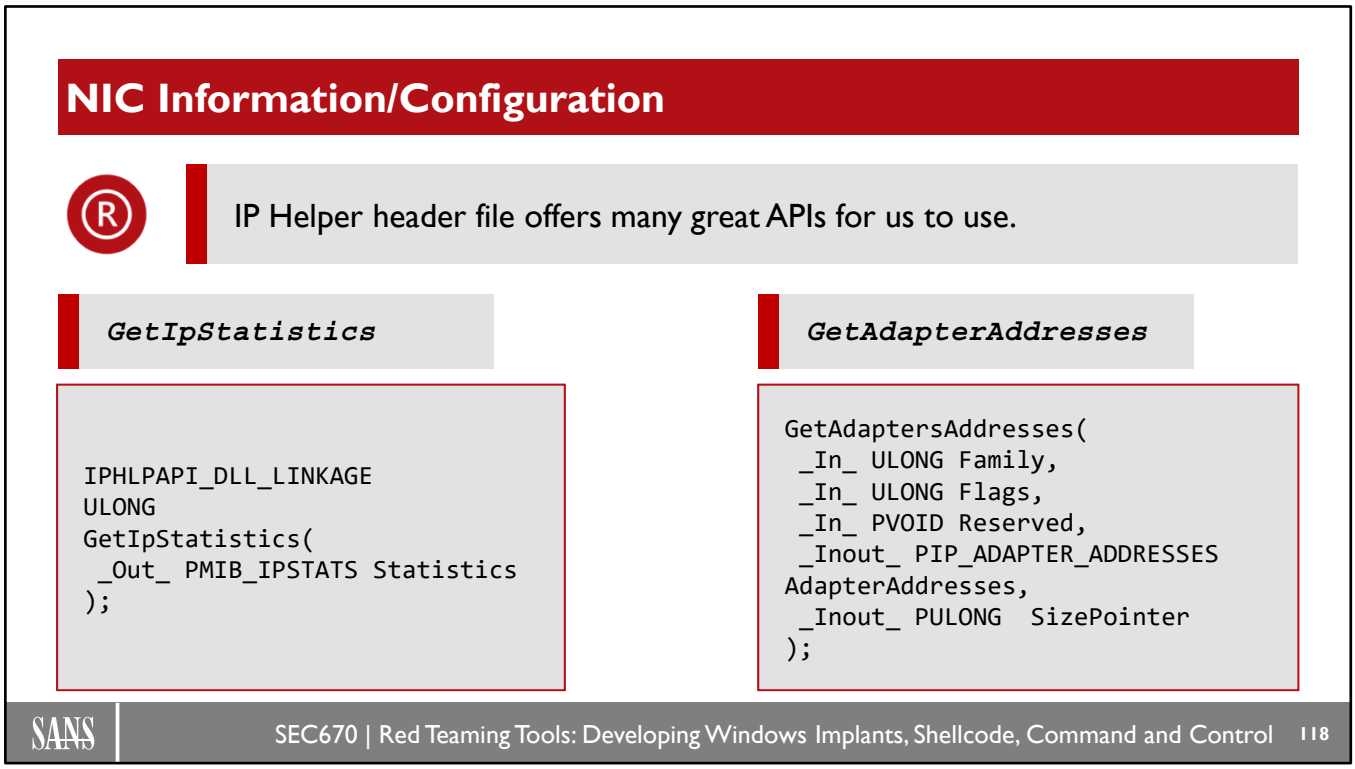
What network is the target connected to?

Most enterprise computers will have a NIC configured with an IPv6 and an IPv4 address. IPv6 will be there even if the organization does not officially support it. Some computers could have dual NICs and be what can be called dual homed. A dual homed system is connected to two different networks and offers a great pivot point into a new environment.

Network Information

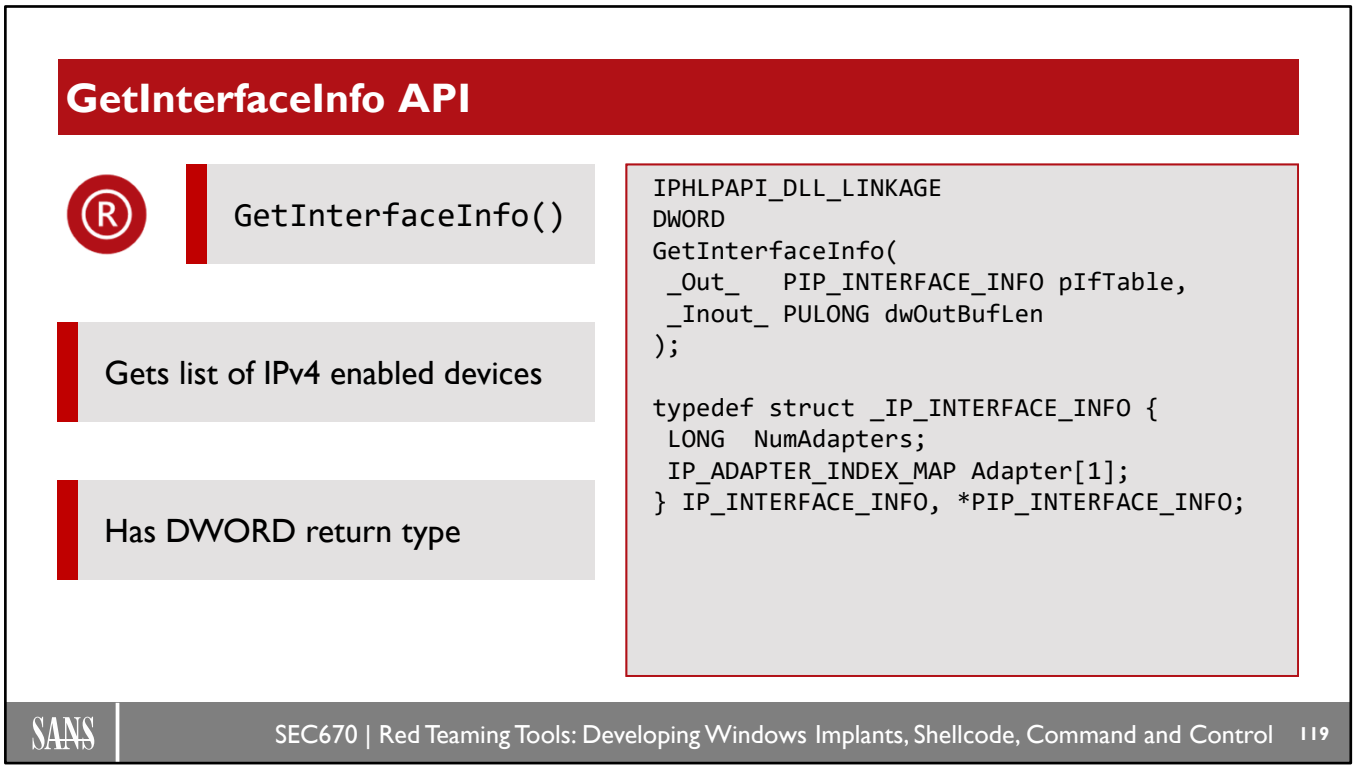
You might not be able to determine the purpose of a system just by gathering network information, but it definitely helps out. If your tool has network sniffing capabilities, you might get more information that way. Aside from sniffing packets, you can still gather some useful information. Perhaps one of the most interesting pieces of information to find out is coming across a system that has multiple NICs and is being dual homed. A dual homed system can be connected to two different networks at the same time. Some sysadmins have done this thinking it acts as a security boundary and that the NICs are separate, but Windows does its best to treat them as one. In a dual homed system, one NIC could be connected to the DMZ and the other could be connected to the trusted, internal network, or the intranet. Dual homed systems are great targets for us and other attackers because they could enable our movement throughout a network. Stuxnet would happily move through a dual homed system, as did the slammer worm way back in 2003.

We cannot forget about VLANs. These Virtual Local Area Networks virtually combine endpoints into a single broadcast domain, and it might be a good idea to look for that when you get on target. SEC660 discusses VLAN hopping and could be something of interest for developers to understand.



NIC Information/Configuration

VMs are being used more and more these days in production and if you happen to get access to a VM in production, you might see several NICs on it. Your survey tool should gather each NIC's configuration. PowerShell offers some useful cmdlets for this, like Get-NetAdapter, Get-NetAdapterHardwareInfo, and more. The Windows command line offers utilities as well, like netstat, ipconfig, and others. Other interesting information that could be gathered would be the statistics, something that is gathered by running netstat -e. There are several ways to get all of this done, like using WMI or the IpHlpApi. While WMI can be useful, it is not always the easiest to work with when there are Win32 APIs like ***GetIpStatistics***. The IP Helper header file offers many useful APIs for us, and we will take a look at some of them, like ***GetIpStatistics***, ***GetInterfaceInfo***, ***GetAdapterAddresses***, and ***GetNumberOfInterfaces***.



GetInterfaceInfo API

The ***GetInterfaceInfo*** function can be used to gather a list of interfaces on the target that have IPv4 enabled. As you can see by the SAL annotations, the function has two parameters that are written to and one that is read. We can break down the parameters to better understand them.

pIfTable, of type PIP_INTERFACE_INFO, is used as a buffer to hold the list of interfaces that have been found. It is up to us as developers to allocate enough space in the buffer to hold this information. The structure holding the information is the IP_INTERFACE_INFO structure, which has two members. The first one, *NumAdapters*, holds the number of adapters that will be stored in the array. The second member is the Adapter, which is the array of IP_ADAPTER_INDEX_MAP entries. Every structure here will be tied to an index then tied to its corresponding name.

dwOutBufLen, of type PULONG, is a pointer to some DWORD variable that will be used to give the size of the buffer to the function. If you notice, this parameter is *inout*. The out portion is for when the size is not large enough to hold the data, the function will write to the variable the correct size needed. Then you can call this function again with the correct size and it should work just fine.

A successful call would return NO_ERROR. Failed calls can return any one of the following error codes:
- ERROR_INSUFFICIENT_BUFFER: The buffer is not large enough and the correct size has been stored in *dwOutBufLen*.
- ERROR_INVALID_PARAMETER: The *dwOutBufLen* is NULL or the function cannot write to the variable.
- ERROR_NO_DATA: Could not find any network adapters.
- ERROR_NOT_SUPPORTED: The function is not supported for this version of OS.

GetIpStatistics API

`GetIpStatistics()`

Grabs the IP statistics for the system

Has ULONG return type

```
IPHLPAPI_DLL_LINKAGE
ULONG
GetIpStatistics(
 _Out_ PMIB_IPSTATS Statistics
);
```


GetIpStatistics API

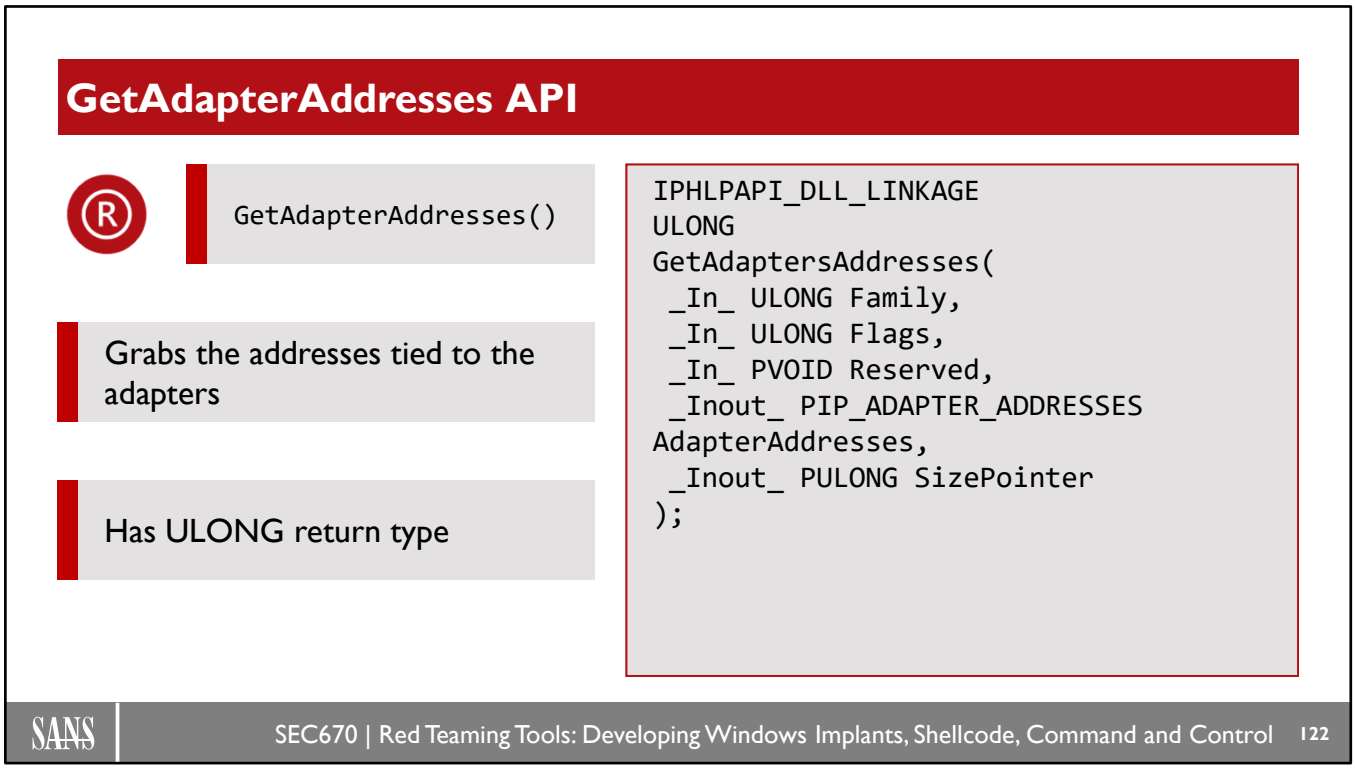
The ***GetIpStatistics*** function can help us recreate familiar Windows commands like netstat. The overall IP statistics are captured and returned to the caller of this function. The function is not very complex and accepts a single parameter: Statistics. The Statistics parameter is of type PMIB_IPSTATS, which is a struct that gets filled out by the function so that it has a place to store the information for any IP statistics that are gathered, hence the reason why the Statistics parameter is an out parameter. Upon success, the function will return NO_ERROR. When the function fails, it can return the following error code: ERROR_INVALID_PARAMTER for when the Statistics parameter is NULL or if the function is not able to write to the pointer. The structure the function fills out is very large, but the next slide holds a snippet of some of its members.

MIB_IPSTATS Struct

```
typedef struct _MIB_IPSTATS_LH {
[..SNIP..]
 DWORD dwDefaultTTL;
 DWORD dwInReceives;
 DWORD dwInHdrErrors;
 DWORD dwInAddrErrors;
 DWORD dwForwDatagrams;
 DWORD dwInUnknownProtos;
 DWORD dwInDiscards;
 DWORD dwInDelivers;
 DWORD dwOutRequests;
 DWORD dwRoutingDiscards;
 DWORD dwOutDiscards;
 DWORD dwOutNoRoutes;
[..SNIP..]
} MIB_IPSTATS_LH, *PMIB_IPSTATS_LH;
```


MIB_IPSTATS Struct

The MIB_IPSTATS structure is the structure that is filled out by the ***GetIpStatistics*** function. Some of the structure members are very useful and some you might not even care about other than just being extra detailed with information for the operator.



GetAdapterAddresses API

The ***GetAdapterAddresses*** can be used when you need to find out what adapters have what IP address. The function is great because not only can it do IPv4, but it can do IPv6 as well. ***GetAdapterAddresses*** has several **in** parameters and a few **inout** ones indicating that the function will be attempting to write to those variables. Let us break down each one of these parameters to get a better understanding of what they are.

Family, of type ULONG, is used to indicate what address family the function should get. This can be AF_INET (6), or AF_INET6 (23). The other interesting part is that you can specify both families at the same time with AF_UNSPEC. AF_UNSPEC indicates that any adapter that has IPv4 or IPv6 enabled, should have its information returned to the caller.

Flags, of type ULONG, is used to indicate what type of addresses to get. There is a long list of types that can be used for this parameter, but a few options are the following: GAA_FLAG_INCLUDE_GATEWAYS, GAA_FLAG_INCLUDE_ALL_INTERFACES, and quite a few more.

Reserved, of type PVOID, is reserved so we do not really care about it.

AdapterAddresses, of type PIP_ADAPTER_ADDRESSES, is a pointer to some variable that will act as the buffer filled with a linked list of IP_ADAPTER_ADDRESSES structures. The structure is a massive structure, and it is worth browsing to the MSDN documentation to completely understand it.

SizePointer, of type PULONG, is a pointer to the variable that stores the size of the buffer.

Upon success, the function will return ERROR_SUCCESS. Should the function ever fail it will return one of the following error codes:

- ERROR_ADDRESS_NOT_ASSOCIATED: An address has yet to be associated with the device.
- ERROR_BUFFER_OVERFLOW: The buffer size indicated is not large enough to hold the requested information.
- ERROR_INVALID_PARAMETER: SizePointer is NULL, Family was not a valid family option.
- ERROR_NOT_ENOUGH_MEMORY: Literally not enough memory to complete the function.
- ERROR_NO_DATA: No addresses found.

GetNumberOfInterfaces API

GetNumberOfInterfaces

Grabs the number of interfaces

Has DWORD return type

```
IPHLPAPI_DLL_LINKAGE
DWORD
GetNumberOfInterfaces(
 _Out_ PDWORD pdwNumIf
);

// example

DWORD dwCount = 0;
GetNumberOfInterfaces(&dwCount);

// error check
```


GetNumberOfInterfaces API

If you wanted to run something quick and easy, the ***GetNumberOfInterfaces*** function would be it. The only parameter you need to worry about is an **out** parameter. The function will write to it the number of interfaces that have been discovered on the local machine. The *pdwNumIf* parameter is of type PDWORD. All interfaces should be enumerated by the function, including the system's loopback adapter. If you do not care about the loopback interface, then you do not need to use this function. Other functions like ***GetAdaptersInfo*** and ***GetInterfaceInfo*** will not return information about the loopback interface. Also, the number returned might be higher than what you might be expecting as it does not directly relate to physical NICs on the target. Logical interfaces will be included in the count of interfaces, so if you were expecting to see 2 and got back something like 18, that would be the reason why.

Module Summary

Discussed how to gather information about the network

Discussed how to gather NIC information about the target

Module Summary
In this module, we discussed the why and how of gathering information about a target's NIC configuration, as well as any other information we can gather about the network overall. The information presented in this module can be the foundations for creating tools like arp, ipconfig, netstat, etc.

Unit Review Questions

What API will give you an IP address for a network adapter?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API will give you an IP address for a network adapter?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Unit Review Answers

What API will give you an IP address for a network adapter?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Answers
Q: What API will give you an IP address for a network adapter?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Unit Review Questions

What API includes logical interfaces in its results?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API includes logical interfaces in its results?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Unit Review Answers

What API includes logical interfaces in its results?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API includes logical interfaces in its results?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

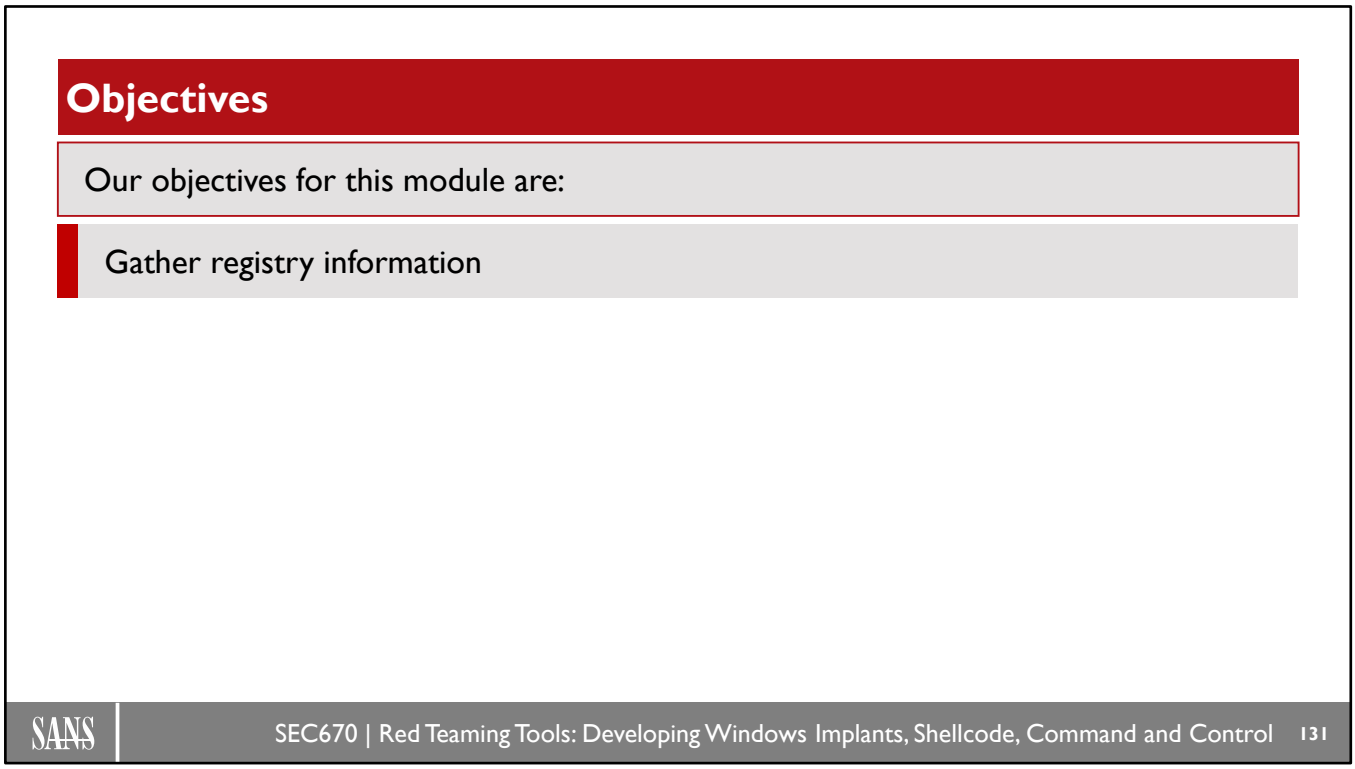
Services and Tasks

Network Information

Registry Information

Bootcamp

This module will discuss how to enumerate the Windows Registry to find critical information about the system.



Objectives
The objectives for this module are to understand what information can be found in the registry.

Registry Information

Troves of information

The Windows Registry contains troves of information that can arguably be deemed critical to your survey tool. The registry is so important that even the system itself relies on information found in the registry.

Registry Information

The registry was discussed in tremendous detail during Section 1, along with the APIs needed to enumerate practically everything in it. It is being included here during this day as a brief reminder that it should not be forgotten about when conducting your survey. The registry is an excellent source for collecting information from the target. Some sections you might not be able to query unless you have Administrator privileges or higher, but even still, you can collect useful information as a basic user.

The Registry (1)

The registry API family provides most functionality for registry interaction.

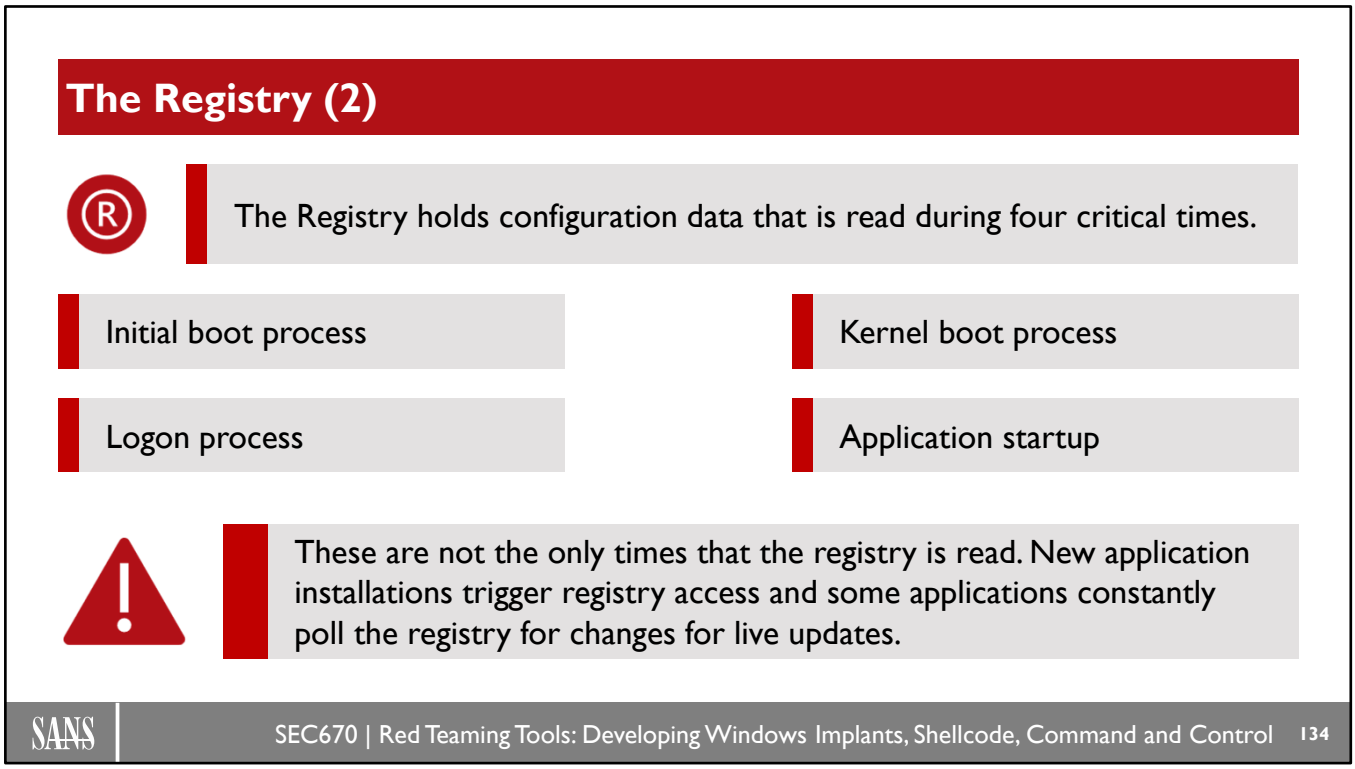
The Registry is a collection of five hives where each one exposes information, some critical to the functionality of the OS. The hives have keys, which then have subkeys with values that applications or services might need to query.

It is has become the go-to location for developers for storing application information.

The Registry (1)

If you are coming from a pure Linux world, the Registry could be quite foreign to you. The Registry is simply a collection of five hives with each one exposing various pieces of information. In all reality, there are only two true hives, **HKEY_USERS** and **HKEY_LOCAL_MACHINE**. If you fire up regedit.exe you would see the five hives, but the other three hives are simply a combination of some of the data that can be found in the two main hives. The hives have keys and subkeys that hold settings and configurations that can be specific to users or to the machine. For 64-bit Windows, there will exist a 32-bit portion of the registry that will store information specific for 32-bit applications installed on the system. There will be a virtual redirection for 32-bit apps implemented by a mechanism called registry virtualization. The 64-bit apps see no such portion of the Registry as there is no need for virtual redirection.

The Registry used to be the go-to location for developers to store information for the applications they would develop. Now, Microsoft would like it if you left the Registry well enough alone so that processes internal to Windows are the only ones interacting with it. There are other ways to store information for your application, such as creating configuration files like INI, YAML, JSON, DAT, XML, etc. For this class and for our purposes, we will not be listening to that recommendation.



The Registry (2)
The configuration data held in the Registry is typically read from at four critical times.

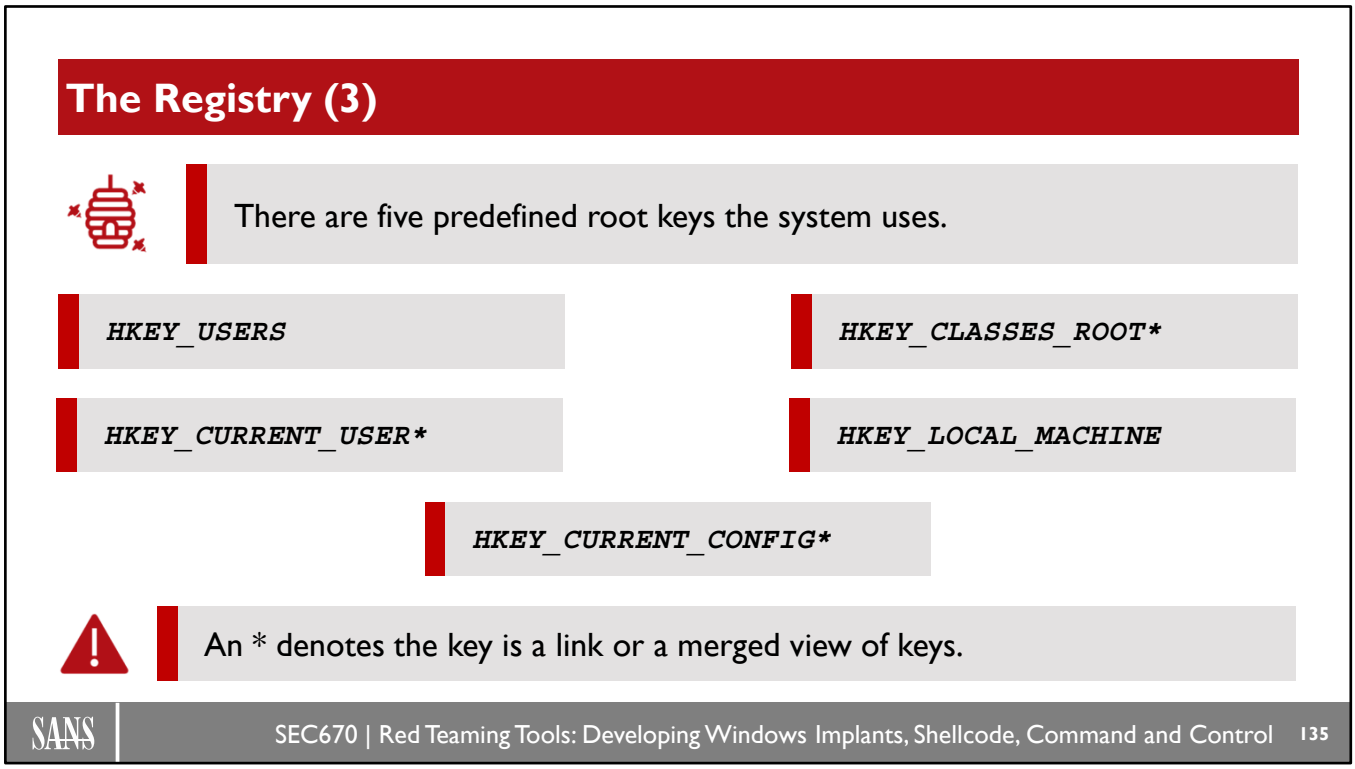
1. The initial boot process
2. The kernel boot process
3. The logon process
4. Application startup

The Boot Configuration Database (BCD) is stored in a registry hive and the boot loader must read the configuration data to retrieve a list of boot device drivers to load into memory. Once that happens, the kernel can start its initialization process. The kernel will load the appropriate device drivers and determine how some managers like the process manager and memory manager configure themselves. All of this is a way to fine tune how the computer will behave.

The logon process starts the per-user preferences. Explorer is one of many Windows components that must read the per-user profiles to set up various items like desktop wallpaper, screen saver, where desktop icons are placed, and what applications must be started. This is how each user on a local system can have different setups/preferences for their environment.

When the applications are being loaded by the system loader, they will read some systemwide settings as well as per-user preferences, like how the window layout is for an application like Word. If the user has a custom toolbar layout for an application, then that needs to be read and prepared when the application starts. Also, the most recently accessed documents will be retrieved and available to view.

New application installs are another moment when the registry is being read. The application will store various information in certain keys that are specific to it. Some of that data might be easily readable, but some might be obfuscated. Many AV products do this in an attempt to conceal some of their internal mechanisms, like how they communicate with their cloud engine when a suspicious file needs to be analyzed. Also, some applications can be overly aggressive with their registry access by polling for live changes made to its configuration data. This is not a best practice as idle systems should not have a lot of registry activity.



The Registry (3)
You might have noticed that each root key starts with an H. This is because the root key names are Windows handles (H) to keys (KEY); hence the name HKEY. The key names on the slide that are annotated with an asterisk "*" are links to other keys or a combination of two keys to provide a unique view of information. Here is a brief breakdown of the five root keys. For more detailed information, check out the *Windows Internals* books in addition to MSDN online documentation.

HKEY_USERS: Each time a new user logs in to the system, a new hive, called the user profile hive, will be created. Here, information about all system accounts is stored.

HKEY_CURRENT_USER: Data tied to the user who is currently logged on. This is a subkey under HKEY_USERS.

HKEY_CLASSES_ROOT: File association and Component Object Model (COM) object registrations.

HKEY_LOCAL_MACHINE: Systemwide settings.

HKEY_CURRENT_CONFIG: The current hardware profile.

The Registry (4)

Deep dive: HKEY_USER (HKU)

The HKU key will hold a subkey (HKCU) for each user profile on the local system. There is also a profile for the system that has its own subkey, *HKU\.Default*. Winlogon uses the system profile to determine various settings like the desktop background.

ProfilesDirectory

ProfileList

The Registry (4)

The HKU key holds a wealth of user information. In fact, it holds user information for every user that has logged on to the local machine, as long as the user does not have a roaming profile with Active Directory or similar. Each user will have their own specific subkey—even the system has its own subkey located at *HKU\.Default*. The system profile is used by the Winlogon process to read information like the desktop background to show to the user when logging into their account. The user profiles are stored in a location determined by the registry value held in *HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory*. If the value has not been modified, the default is *%SystemDrive%\Users*.

If you wanted to develop some kind of user enumeration tool, then a good place to enumerate a list of profiles that exist on the system would be the ProfileList key. From there, the subkey name will be the Security Identifier, or SID, of the account that is tied to it.

The Registry (5)

Deep dive: HKEY_CURRENT_USER (HKCU)*

This root key holds configuration information for the locally logged-on user regarding software configuration information and user preferences. They key points to the user profile, which is located at *\Users\<username>\Ntuser.dat*.

- Created for each new login
- Subkey under HKU
- Subkeys: console, software, control panel, identities, printers, keyboard layout, etc.

The Registry (5)

The HKCU key exists for each locally logged-on user. Each time a new user logs on, this key is created as a subkey under HKU. User-specific information is stored in this key located at \Users\<username>\Ntuser.dat. In addition, software configuration information is also stored in this key since different users might have different preferences for how their applications are laid out. In addition to this, different users could also have different user preferences that must be stored, and this is the perfect location to store that information. Also, any service processes that kick off under the context of a specific user will trigger their specific HKCU to be loaded.

There are several subkeys: AppEvents for sounds/events, Console for command window settings, Control Panel for screen saver information, Software for user-specific software preferences, Printers for printer connection settings, Keyboard layout for what country region layout should be (US or UK), or Identities for Windows Mail account information. This is not all the subkeys found under HKCU and to list them all, you can write an enumeration tool to gather that information.

The Registry (6)

Deep dive: HKEY_CLASSES_ROOT (HKCR)*

This root key holds three types of information: file extension associations, COM class registrations, and virtualized registry root for the UAC. Every registered file extension will have its own key that is typically the REG_SZ value type. Sometimes they simply point to another key that holds the needed information.

HKCU\SOFTWARE\Classes

HKLM\SOFTWARE\Classes

The combination of the above Classes keys make this root key.

The Registry (6)

This root key is not a genuine root key like HKU or HKLM, nor is it a link or subkey of another key. It is the combination of two Classes keys: HKCU\SOFTWARE\Classes and HKLM\SOFTWARE\Classes. It might not make sense to have separate locations for user and system registration, but users having roaming profiles that are configured differently creates the need for that separation.

This key also holds information regarding file extensions and what applications they are associated with. Each registered file extension will have its own key that is simply a link to another location. Of course, the location and layout of the Registry greatly depends on what version of Windows you are running.

The Registry (7)

Deep dive: HKEY_LOCAL_MACHINE (HKLM)

This root key holds vital information for the system. Some of the critical information like how the system boots is stored here. Other information is stored here, like systemwide software configurations, installed components, user passwords, and boot entries to name a few.

BCD: boot entries

SAM: account passwords

The Registry (7)
The HKLM key is the root key that holds all systemwide configuration subkeys.

- BCD00000000
- COMPONENTS
- HARDWARE
- SAM
- SECURITY
- SOFTWARE
- SYSTEM

BCD00000000 subkey is the Boot Configuration Data (BCD) information that is loaded as a registry hive. BCD is the replacements for the boot.ini file and it adds greater capabilities. There can be any number of entries in the BCD, like one for the Windows installation itself. The entries can be found in the subkey Objects.

COMPONENETS subkey stores information regarding the Component Based Servicing (CBS) stack. The Windows installation has various file and resources that make up the installation image. Depending on how many components are installed, this subkey can become very large and so it is loaded and unloaded dynamically, as necessary.

HARDWARE subkey holds information about the hardware in the system. This is read when the Device Manager user application is started.

SAM subkey holds user passwords, any group definitions, and associated domains. Because of the sensitive nature of the information stored in this subkey, the security descriptor does not even allow the Administrator account to access it.

SECURITY subkey holds policies that are systemwide; in addition, User-rights are also stored in this subkey.

SOFTWARE subkey holds systemwide configuration information that is not critical for booting the computer. Any third-party application installed on the system will also store their settings here.

SYSTEM subkey holds systemwide configuration information that is critical for booting the computer. Changing information in this subkey can render your system useless and for such occasions, Windows stores a last known good control set under this subkey. There, a lightweight copy of the subkey, called a maintenance copy, lets admins choose a working control set they know was good.

The Registry (8)

Deep dive: HKEY_CURRENT_CONFIG (HKCC)*

This root key is nothing more than a link to the hardware profile that is stored in HKLM: *HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current*. It's not prevalent today but Windows keeps it around in the name of backwards compatibility.

The Registry (8)
The HKEY_CURRENT_CONFIG, or HKCC for short, root key is one of the three links to other root keys. HKCC is formally linked to the HKLM root key and since this linked key points to whatever the current hardware profile is, the root key path is: HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Windows does not support hardware profiles anymore, but the key is still created all in the name of backwards combability with legacy applications that depend on it being there.

The Registry (9)

Deep dive: HKEY_PERFORMANCE_DATA (HKPD)

This root key is unique because it cannot be accessed directly via the Registry Editor. It must be accessed programmatically via the Registry APIs. In it you would find performance counters either from system components or server applications.

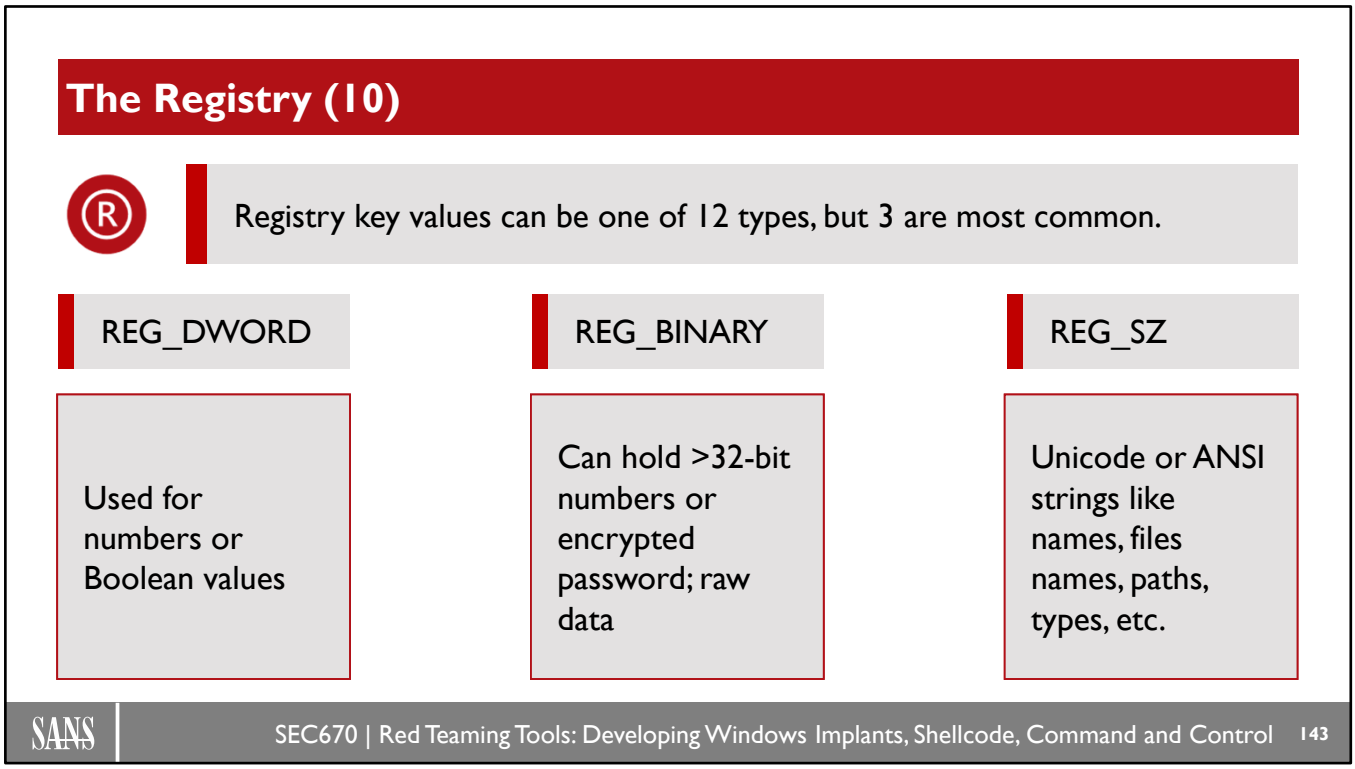
RegQueryValueEx

Technically not stored here

The Registry (9)

One might think that every registry key would be accessible for viewing via a tool like regedit.exe. Well, the HKEY_PERFORMANCE_DATA key is not one of those keys that regedit.exe can view. If you want to poll performance specific data, then you must do so using the appropriate Registry APIs like ***RegQueryValueEx***.

Another interesting tidbit about this key is that the performance counter information is not actually created/stored here but rather it comes from a provider. These performance data providers actually push the values to the key. One could, as another option for querying this information, use the Performance Data Helper API functions that are provided by the Pdh.dll, which would be the recommended choice to use.



The Registry (10)
A registry key's value can hold several different types, 12 types to be exact. Despite the number of value types, there are three that you are more likely to come across.

1. REG_DWORD
2. REG_BINARY
3. REG_SZ

If you remember from the Windows data types section earlier in this section, then the DWORD type should be familiar to you. REG_DWORD can hold 32-bit numbers or Boolean values to represent on/off or enabled/disabled flags.

REG_BINARY, as the name suggests, store raw binary information like encrypted passwords. It can also be used to store numbers that are larger than 32-bits.

The last of the common three types is REG_SZ. This is used to hold a string that has a null terminator. Perhaps the SZ means String Zero indicating String\0.

Registry Keys and Values (1)

RegOpenKeyEx

Return value is LSTATUS

Used to open a handle to a Registry key

```
//declared in Winreg.h

LSTATUS RegOpenKeyExW(
  _In_     HKEY  hKey,
  _In_opt_ LPWCSTR lpSubkey,
  _In_opt_ DWORD  ulOptions,
  _In_     REGSAM samDesired,
  _Out_    PHKEY  phkResult
);
```


Registry Keys and Values (1)
First and foremost, look at the function's return type: LSTATUS is a 32-bit signed integer, and any errors will be returned. The error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. No need to call ***GetLastError*** with NSTATUS return values. Let's break down the parameters in detail.

hKey requires an open handle to an open registry key. If there is no handle opened yet, then one of the predefined keys can be passed: HKEY_CLASSES_ROOT, HKEY_CURRENT_CONFIG, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS.

lpSubkey needs a pointer to the name of the registry subkey to open. Case sensitivity is ignored here. Because the SAL annotation contains the _opt_ suffix, the pointer is optional; nullptr. It is important to note that NULL can only be passed here if hKey is passed one of the predefined keys.

ulOptions should just pass NULL here. If not null, then certain options are to be applied when open the key. The only option noted by MSDN documentation is REG_OPTION_OPEN_LINK, meaning the key is simply a symbolic link.

samDesired is a mask indicating the desired access to the key.

phkResult, as annotated by the annotation _Out_, should be a pointer to a variable that will hold the handle to the opened key.

Example: RegOpenKeyEx

```
INT main(VOID)
{
    HKEY hHKCU = HKEY();
    RegOpenKeyExW(HKEY_CURRENT_USER, L"Console", NULL, KEY_READ, &hHKCU);
}
```


Example: RegOpenKeyEx
The example here initializes a variable of type HKEY that will be used to store the handle that the function gives upon success. Just like the function declares, the last parameter must be the address of the variable. Afterall, it is an _Out_ parameter so the user is responsible for making that available for the function to use.

The function doesn't "return" a handle because this function returns an LSTATUS value that could be used to determine why it may have failed.

Error handling is left out for brevity.

Registry Keys and Values (2)

RegQueryValueEx

Return value is LSTATUS

Used to read the type and data of a Registry key value

```
//declared in Winreg.h

LSTATUS RegQueryValueExW(
  _In_         HKEY    hKey,
  _In_opt_     LPCWSTR lpValueName,
  _Reserved_   LPDWORD lpReserved,
  _Out_opt_    LPDWORD lpType,
  _Out_        LPBYTE  lpData,
  _Inout_opt_ LPDWORD lpcbData
);
```


Registry Keys and Values (2)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. Let's understand the function's parameters in detail.

hKey is a handle to a key, just like in the ***RegOpenKey*** function.

lpValueName is a pointer to the string holding the name of the registry value. Passing NULL or "" will make the function grab the type and data for the key's default or unnamed value. If the *lpValueName* cannot be found, or it does not exist, the function will return ERROR_FILE_NOT_FOUND.

lpReserved must be NULL since it is reserved for internal purposes only.

lpType is a pointer to a variable that will hold a code that indicates the type of data being stored in the mentioned value. MSDN documents the possible types that are acceptable to pass here, but some are REG_BINARY, REG_DWORD, and REG_SZ.

lpData is a pointer to a buffer that will hold the value's data. If there is no requirement for data, then simply pass NULL here. This buffer can be too small to accept the data and if that is the case, the function will return ERROR_MORE_DATA.

lpcbData is a pointer to a variable that indicates the buffer size in bytes pointed to by *lpData*. Upon return, the variable will hold the size of the copied data. *lpcbData* is allowed to be NULL but only when *lpData* is also NULL.

Example: RegQueryValueEx

```
INT main(VOID)
{
    HKEY hHKCU = HKEY();
    DWORD dwType = 0;
    DWORD dwSize = 0;

    RegOpenKeyExW(HKEY_CURRENT_USER, L"Console", NULL, KEY_READ, &dwSize);
    auto Value = std::make_unique<BYTE[]>(dwSize);
    ReqQueryValueExW(hHKCU, L"FaceName", NULL, &dwType, Value.get(), &dwSize);

    // do something with the queried value ...

    RegCloseKey(hHKCU);

    return ERROR_SUCCESS;
}
```


Example: RegQueryValueEx
The example here builds from the first snippet on the previous slide showing ***RegOpenKeyExW*** where the handle to the key was obtained. That key handle is needed for future calls that are specific to that key. ***RegQueryValueExW*** accepts that key handle as its first parameter. You might question the need for consecutive calls to ***RegQueryValueExW*** and the reason for this is because at first, we do not know the size of the data being held in the FaceName key. To determine the proper size needed, the function allows us to pass in NULL for the lpData parameter and a non-NULL value for the lpcbData parameter. When the function does its parameter checking, it will see this and store the size of the data, in bytes, in the variable pointed to by lpcbData. In our case that would be ***size***. Before we get to the second call, we take the size of the data and use that to create a byte buffer using make_unique. Here, I am calling that ***value*** since it will hold the key's value when done.

Fully equipped with the information needed, the second call can be made. A buffer can now be passed into the lpData parameter, and the size of that buffer can be passed into the lpcbData parameter.

Error handling is left out for brevity.

Walking the Registry (1)

RegEnumKeyEx

Return value is LSTATUS

Used to enumerate subkeys under a specific key

```
//declared in Winreg.h

LSTATUS RegEnumKeyEx(
  _In_         HKEY    hKey,
  _In_         DWORD   dwIndex,
  _Out_        LPTSTR  lpName,
  _Inout_      LPDWORD lpcchName,
  _Reserved_   LPDWORD lpReserved,
  _Out_        LPTSTR  lpClass,
  _Inout_opt_  LPDWORD lpcchClass,
  _Out_opt_    PFILETIME lpftLastWriteTime
);
```


Walking the Registry (1)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. This function is great for enumerating a particular key's subkeys. Let's break down the function's parameters.

hKey is a handle to a key, just like in the RegOpenKey function, but this time the handle must have been open with the KEY_ENUMERATE_SUB_KEYS access mask or you will be denied access when you make this call.

dwIndex will be used to keep track of loop iterations. Starting with a value of 0, we can increment this value each iteration until we get a return value of ERROR_NO_MORE_ITEMS.

lpName is just the simple key's name and not the root key name. If the buffer for this parameter is not large enough to hold the key's name, then the function will fail with ERROR_MORE_DATA. If this happens, don't expect lpName to have anything in it, it should be NULL.

lpcchName is the size of the lpName and should be set to the maximum number of characters that a buffer can hold. This includes the NULL byte. The function will modify this value to reflect the correct number of characters written, but not include the NULL byte. FYI, the max key name length is 255.

That is the bulk of the function. If a developer wished, they could grab the class name, hardly ever used or needed, along with the last time the key was modified. The latter part could be interesting if you are looking for something in particular.

Walking the Registry (2)

RegEnumValue

Return value is LSTATUS

Used to enumerate a key's value

```
//declared in Winreg.h

WINADVAPI
LSTATUS
APIENTRY
RegEnumValueW(
  _In_  HKEY  hKey,
  _In_  DWORD dwIndex,
  _Out_writes_to_opt_(*lpcchValueName,*lpcchValueName +
1) LPWSTR lpValueName,
  _Inout_   LPDWORD lpcchValueName,
  _Reserved_ LPDWORD lpReserved,
  _Out_opt_  LPDWORD lpType,
  _Out_writes_bytes_to_opt_(*lpcbData, *lpcbData)
__out_data_source(REGISTRY) LPBYTE lpData,
  _Inout_opt_ LPDWORD lpcbData
  );
```


Walking the Registry (2)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error.

Another observation you may have had is the addition of two SAL annotations that are describing the function. WINADVAPI is defined as DECLSPEC_IMPORT, which is defined as __declspec(dllimport). The APIENTRY type is one we have seen before earlier today. To refresh your memory, it's defined as WINAPI, which is defined as __stdcall. Now let's understand a few of the function's parameters in detail.

hKey is one we've talked about enough.

dwIndex is new and is used when doing a for loop while iterating over the values. When the function returns **ERROR_NO_MORE_ITEMS**, you can break out of the loop and continue.

Just like ***RegEnumKeyEx***, the buffers that you make for holding the ValueName and Data need to be large enough. If you want, you could initialize them to the values that are returned from the ***RegQueryInfoKey*** function described on the next slide.

lpType doesn't need to be described again. lpData will hold the values and lpcbData is the size of it—again, just like the other functions mentioned already.

If you run into any issues using this function, like getting **ERROR_ACCESS_DENIED**, then you should check the permissions you requested when you opened the handle to the root key. You might need **KEY_QUERY_VALUE** to have success.

Example: RegEnumValue

```
for (DWORD index = 0; ; index++)
{
    DWORD cbSize = cbMaxValueLen;
    DWORD cchName = cbMaxValueName + 1;

    auto status = RegEnumValueW(
        hKey,               // hKey
        index,              // dwIndex
        keyName.get(),      // lpValueName
        &cchName,           // lpcchValueName
        NULL,               // lpReserved
        &type,              // lpType
        keyValue.get(),     // lpData
        &cbSize             // lpcbData
    );

    if (status == ERROR_NO_MORE_ITEMS)
        break;
```

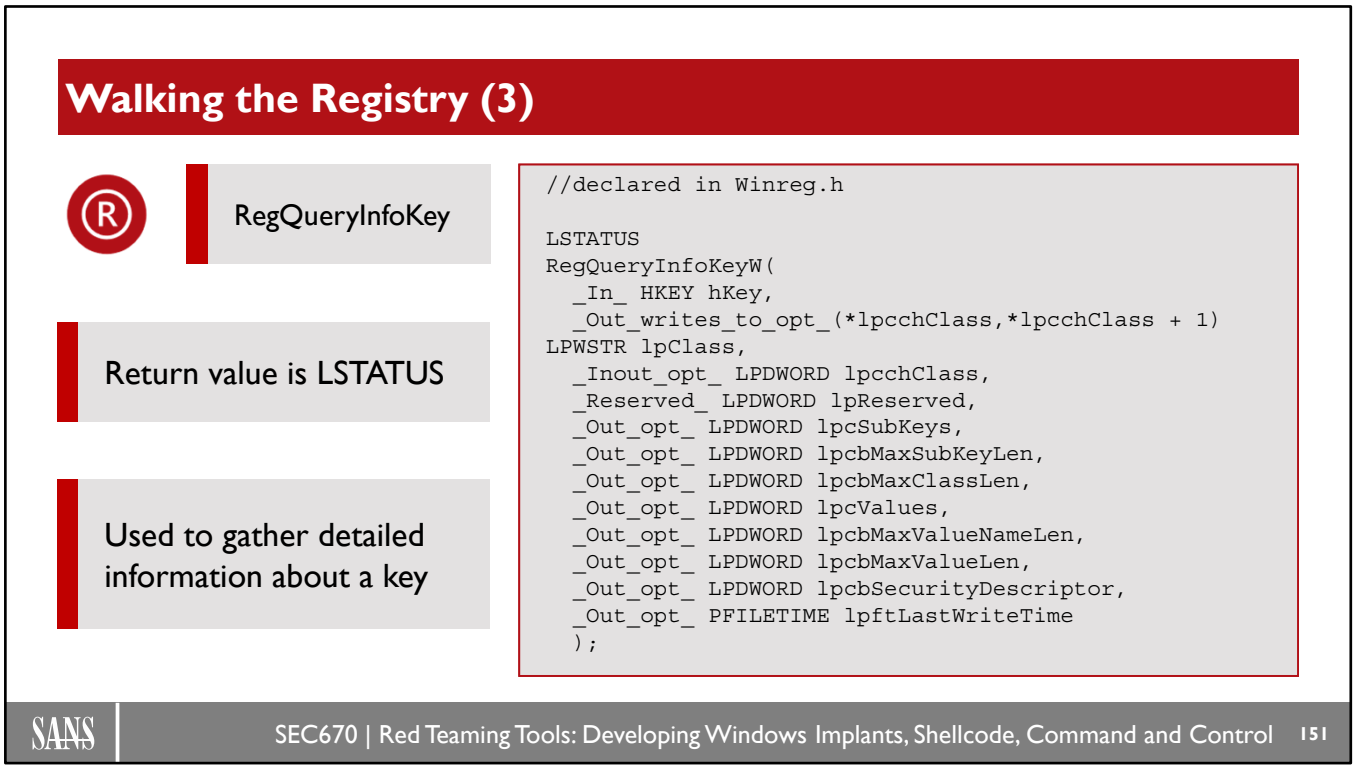

Example: RegEnumValue
The example here shows the function being used in a loop with the *index* variable being used as the loop iterator value, or counter. The key handle is passed followed by the dwIndex value, which is an index of the value that is to be retrieved. It is a good idea to have this value be NULL on the first iteration. Incrementing the value is fine for subsequent calls. The name of the key will be stored in the *keyName* buffer for the lpValueName parameter.

The lpcchValueName parameter, upon return, will hold the character count stored in the buffer. Note, the count does not take into account the NULL terminating character, so you need to do that in your program. lpReserved is NULL followed by the address of the variable to hold the type of data being stored. The lpData parameter was passed the address to the keyValue buffer that will be used to store the key's value. The lpcbData parameter is the size of the buffer but then after the function returns, will indicate the number of bytes written to the buffer.

The loop will continue until the function indicates there are no more items by returned the **ERROR_NO_MORE_ITEMS** error code.

The comments are provided for easy reference with the parameter names.

Error handling is left out for brevity.



Walking the Registry (3)

This RegQueryInfoKey API has many parameters to understand. It might look like a complex function, but it's not that difficult to use compared to some of the others we have discussed today. Looking at the data types for the parameters you can see that most of them are LPDWORD, which are simply DWORD*. Almost all of them are optional too, so that makes it nice to use. Depending on your needs, you could make the API call like this:

RegQueryInfoKeyW(hKey, NULL, NULL, NULL, &nKeys, NULL, NULL, &nValues, &nMaxValNameLen, &nMaxValSize, NULL, NULL)

That small example has seven NULL parameters, leaving just five parameters for you to prep for use. Now we can look at some of the parameters.

lpcSubKeys will hold the number of subkeys, if any, that are found to be under the queried key.

lpcValues will hold the number of—you guessed it—values that are found under the subkey. If you do not care about this, then just pass NULL here. It is optional after all.

lpcbMaxValueNameLen will hold the size of the key's longest value name, but it does not consider the NULL terminating byte, so your code will have to account for that one.

lpcbMaxValueLen will hold the size of the longest data component in bytes.

For failures, unlike previous functions, the function will return a system error code that is defined in WinError.h header file. You could also browse to the following URL for easy lookups: https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes.

Example: RegQueryInfoKey

```
DWORD retCode;
retCode = RegQueryInfoKeyW(
    hKey,                       //
    NULL, NULL, NULL,           //
    &cSubKeys,                  //
    &cbMaxSubKey,               //
    &cbMaxClass,                //
    &cValues,                   //
    &cbMaxValueName,            //
    &cbMaxValueLen,             //
    &cbSecurityDescriptor,      //
    &modified                   //
);
```


Example: RegQueryInfoKey

The example here shows how ***RegQueryInfoKey*** could be called. A standard handle to a key is given followed by three NULL values: lpClass, lpcClass, and lpReserved. Next, the address of the *cSubKeys* variable is passed so the function can write the number of subkeys contained by that key. Next, the address of the *cbMaxSubKey* variable is passed so the function can write the size of the key with the longest name, minus the NULL terminating character of course. Next, the address of the *cbMaxClass* variable is passed to receive the size of the longest string for a subkey class. The address of the *cValues* variable will hold the count of values a key might have. The address of the *cbMaxValueName* variable will hold the size of the key's longest value name. The address of the *cbMaxValueLen* variable will hold the byte count of the size of the longest data component. The address of the *cbSecurityDescriptor* variable will hold the size of the key's security descriptor. The last one is the address of the ***modified*** variable that is a FILETIME structure to hold the last write time.

The comments indicating the parameter names are left out for you to fill them in later.

Error handling is left out for brevity.

Registry Watch Dogs (1)

You can be notified about changes in the Registry.

Perhaps it might be necessary for your code to be notified as soon as a change in the Registry happens. Maybe you want to know if an antivirus product was just installed after your implant was dropped. There could be several reasons you determine.

Don't poll too often

Choose your trigger

Registry Watch Dogs (1)

At times, it might serve your program well to be notified when certain changes happen in the Registry. As mentioned earlier, third party developers are being recommended to move away from storing items in the Registry, but it still happens. Malware is still using the Registry too, and your implants might as well as you develop them.

From an offensive perspective, you might want to watch certain existing keys for modification. Perhaps your own program creates a few keys and then sets up a watch dog to ensure keys aren't deleted. You also might want to check a certain key's value to see if your tool needs to delete itself or take some other action. It's up to you since you are the developer.

Taking a defensive and different angle, perhaps you drop your tool on a target that doesn't have any AV products just yet, but later the user installs one that creates a few registry keys. You might want to know this, so it doesn't alert the user to your implant. Seeing that a new key was created gives you the situational awareness you might need for your survival.

Typically, applications don't need to poll the Registry too often. The ***RegNotifyChangeKey*** is perfect for notifications. As mentioned earlier, an idle system shouldn't have any registry interaction. You can be too aggressive with polling for changes and there's just no need for that. Repetitive access to the same keys and or values could tip your hand to the user or Admin.

All of this is great and all, but one major drawback to this function is that it doesn't show you exactly what changed, only that a change has occurred. It is up to you to determine the finer details. Thankfully, there is alternative that will be discussed later in the course.

Registry Watch Dogs (2)

RegNotifyChangeKeyValue

Return value is LSTATUS

Used to be notified when specific changes happen

```
// defined in winreg.h

WINADVAPI
LSTATUS
APIENTRY
RegNotifyChangeKeyValue(
  _In_    HKEY hKey,
  _In_    BOOL bWatchSubtree,
  _In_    DWORD dwNotifyFilter,
  _In_opt_ HANDLE hEvent,
  _In_    BOOL fAsynchronous
  );
```


Registry Watch Dogs (2)
hKey is the key to be watched and the handle should have the REG_NOTIFY access mask when the handle is being obtained.

bWatchSubtree is a simple Boolean flag to indicate if just the one specified key should be watched (FALSE) or if the entire tree of keys under it need to be watched (TRUE).

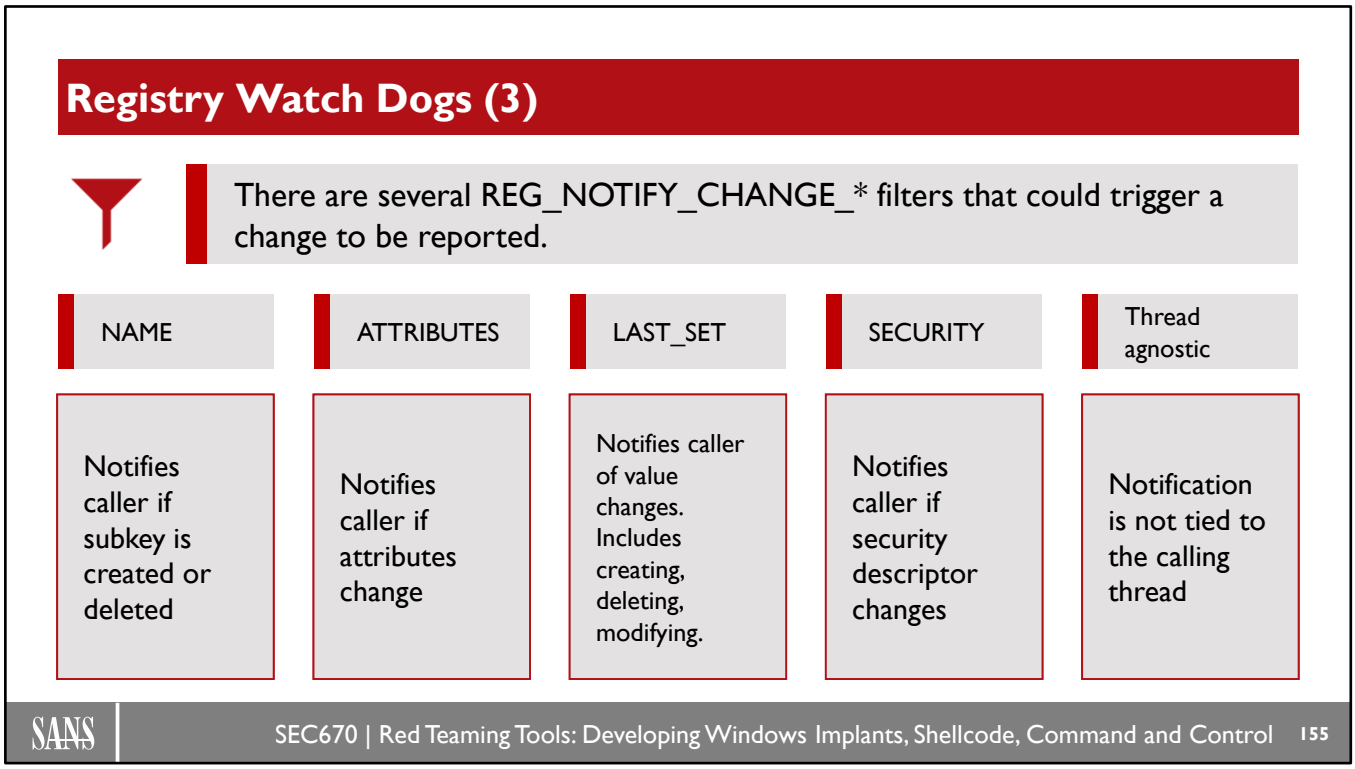
dwNotifyFilter specifies the filter that triggers the notification to the caller. There are several to choose from, like watching for name changes. You can also combine flags to suit your needs using the pipe '|' or the OR operation. The filters are discussed in greater detail on the next slide.

hEvent is just a handle to an event. Depending on if the fAsynchronous flag is set, the function will return straight away, and the change is reported by the event being signaled. If it is not set, then the function will not return until some change takes place.

fAsynchronous is a flag to indicate when the function should return and signal an event. If it is set, TRUE, the function returns immediately and signals the specified event. If FALSE, the function waits until a change is made.

The function will return ERROR_SUCCESS upon success and will return a nonzero value upon failure. The code can be queried using ***FormatMessage*** if desired.

One last note about this function: it will only wait and watch for a single change. When that change happens, it has done its job, so if you want to keep being notified of changes then you must call the function again. Perhaps infinite loops in a new thread could serve well in your program.



Registry Watch Dogs (3)
REG_NOTIFY_CHANGE_NAME if specified, will alert the caller that a subkey has been added or deleted.

REG_NOTIFY_CHANGE_ATTRIBUTES filter will notify the caller if the key's attributes have changed. One of the attributes that could change is the security descriptor information.

REG_NOTIFY_CHANGE_LAST_SET filter will notify the caller if the value of the key has changed. Adding, deleting, and even modifying an existing key's value will trigger the alert.

REG_NOTIFY_CHANGE_SECURITY filter will notify the caller if the security descriptor of the key changes.

REG_NOTIFY_THREAD_AGNOSTIC removes the notification tie to the thread that called the function. This is great for when you want to spin up a new thread for the registration. Also, if that thread terminates, the registration won't die with it because the lifetime of the registration is not directly tied to the lifetime of the calling thread.

Module Summary

Discussed the registry and information found within it.

Module Summary
In this module, we discussed what the registry is, many of the keys, and some of the information that can be found within the registry.

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 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

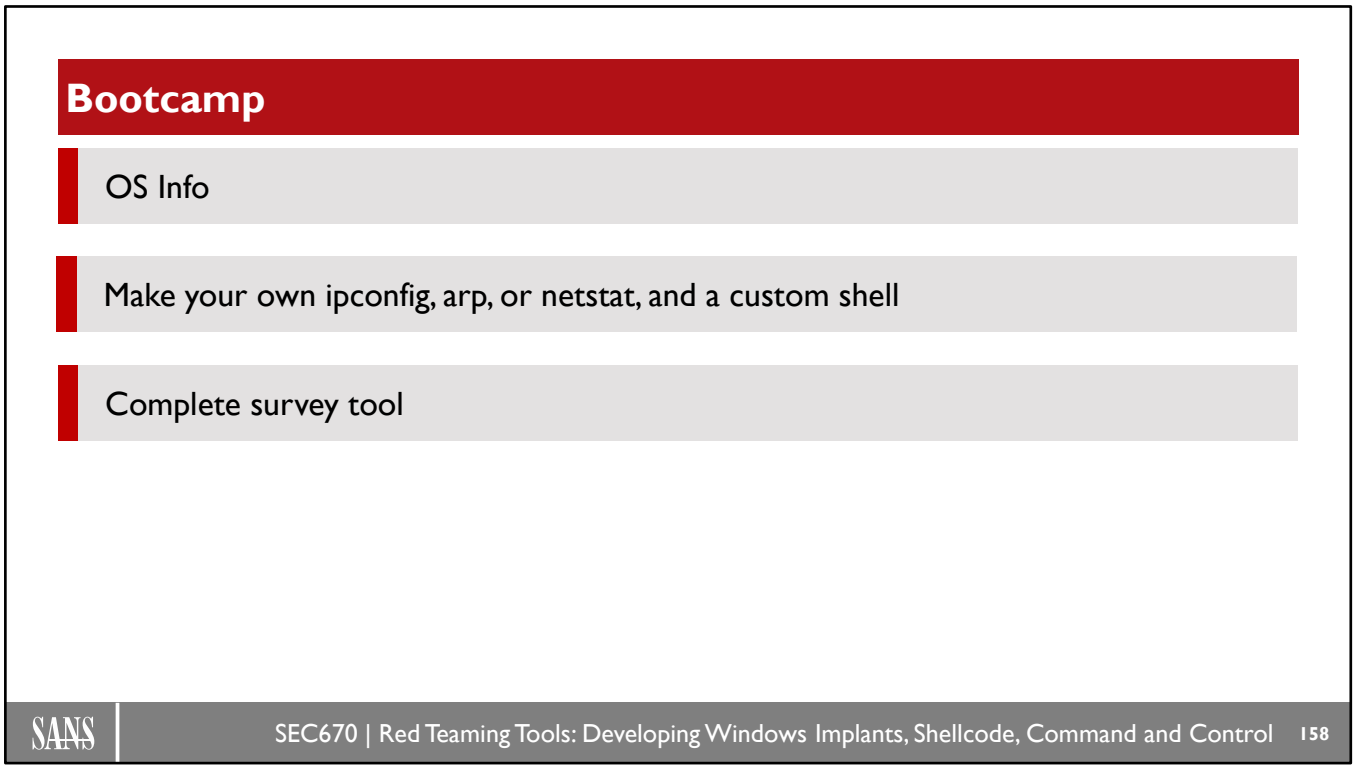
Services and Tasks

Network Information

Registry Information

Bootcamp

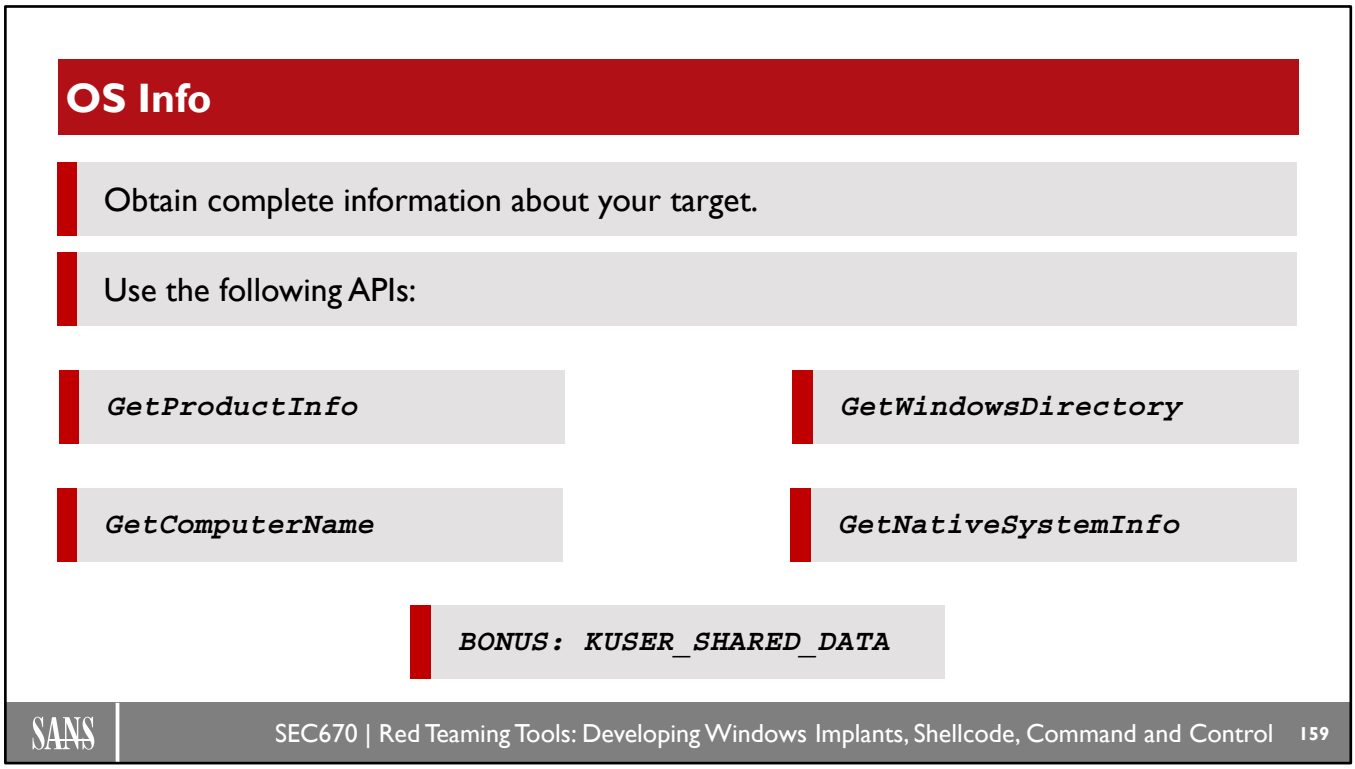
This is your time to go back and complete previous labs or move forward and complete the bootcamp challenges.



Bootcamp
The bootcamp challenges will have varying degrees of difficulty. OS Info brings back something you already learned but also tosses in some new items to see how well you can look at MSDN documentation to learn how to implement new APIs you have not seen before.

The second challenge is to recreate one of the following utilities: ipconfig, arp, or netstat. If you have time, then you can complete all three.

The last challenge is to complete a thorough host survey tool (CustomShell) that would enumerate all of the information that was discussed today. In addition, combine what was learned during Section 1 and create a log file that stores the enumerated system information.



OS Info

This bootcamp challenge is about leveraging a familiar API that you learned about earlier to gather some information about your target. The real challenge comes in with the introduction of several new APIs that can be called to help gather system information. You will have to teach yourself how to use these ones. As a bonus, if you finish the challenge and want to take on another challenge, get as much information as you can from the KUSER_SHARED_DATA struct that is available in almost every single user mode process. You can have all of the output print out to the terminal window to make it easier for testing and debugging. When it is all said and done, you can create a log file with all of the information in it.

Have fun!

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 2

Lab 2.6: Ipconfig
Lab 2.7: Arp
Lab 2.8: Netstat
Lab 2.9: ShadowCraft

This is your time to go back and complete previous labs or move forward and complete the bootcamp challenges.

Lab 2.6: Ipconfig

- Create your own version of ipconfig
- Can add optional arguments
- Can make it fancy with colored output

Lab 2.6: Ipconfig
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 2.7: Arp

- Create your own version of arp.
- Implement arguments like –a and –n.

Lab 2.7: Arp
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 2.8: Netstat

Create your own version of netstat.

Implement arguments like –a, –n, and –t.

Lab 2.8: Netstat
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 2.9: ShadowCraft

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

Lab 2.9: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.