SEC670 I RED TEAMING TOOLS: DEVELOPING WINDOWS IMPLANTS, SHELLCODE, COMMAND AND CONTROL

670.1

Windows Tool Development

SANS | GIAC CERTIFICATIONS

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

SEC670.1

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Windows Tool Development

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.1
Welcome to Section 1 of 670. In this section, we will explore what it’s like to develop tools for the Windows platform.

Table of Contents (1)

This page intentionally left blank.

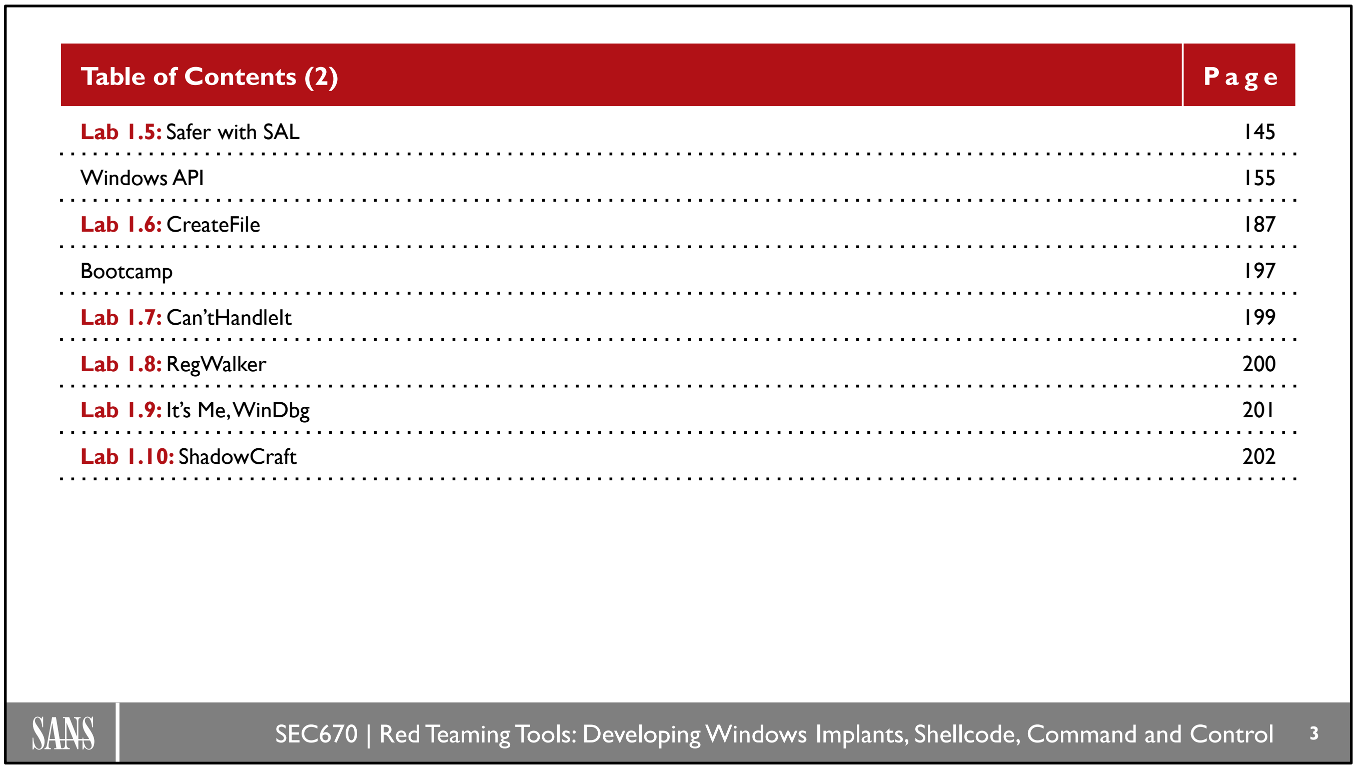
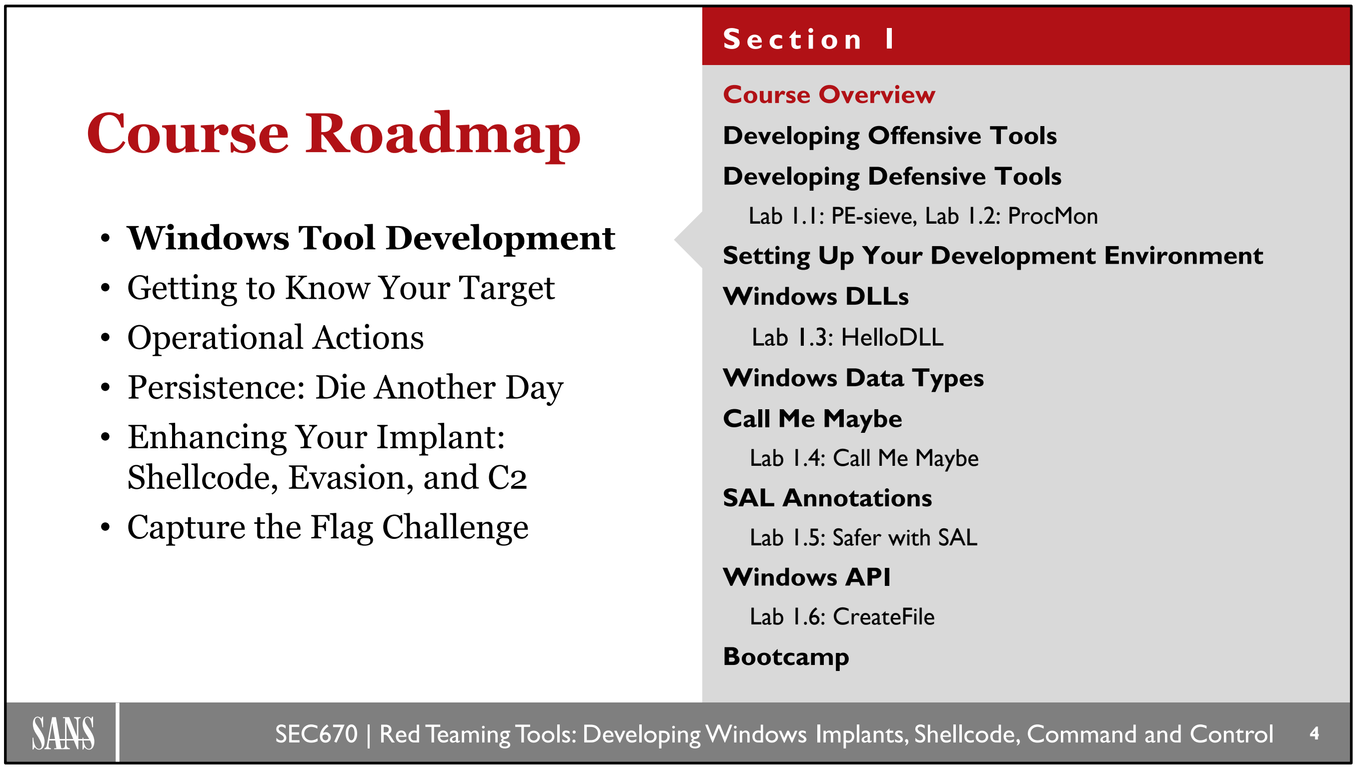


Table of Contents (2)

Page

This page intentionally left blank.



Course Roadmap

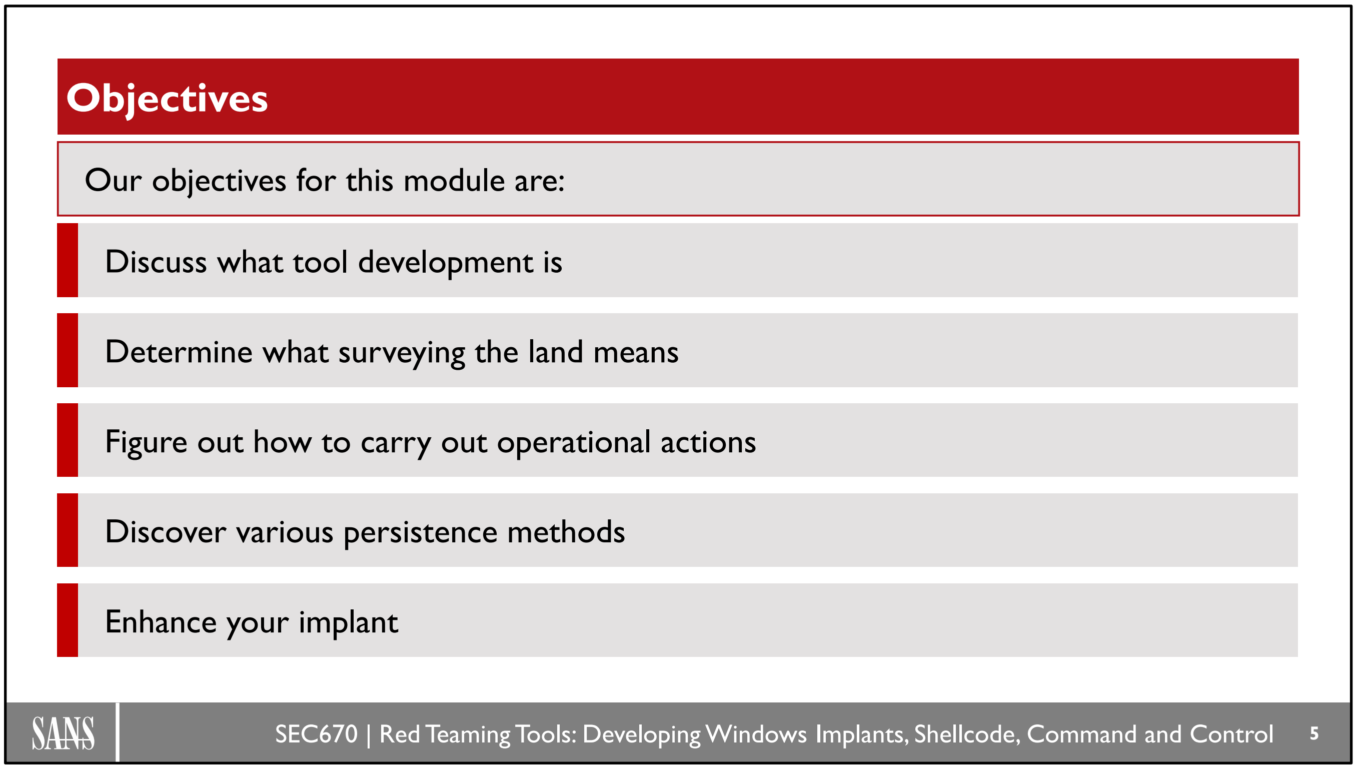
- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

SANS

SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 4

In this module, we will discuss at a high level what the course will be covering.

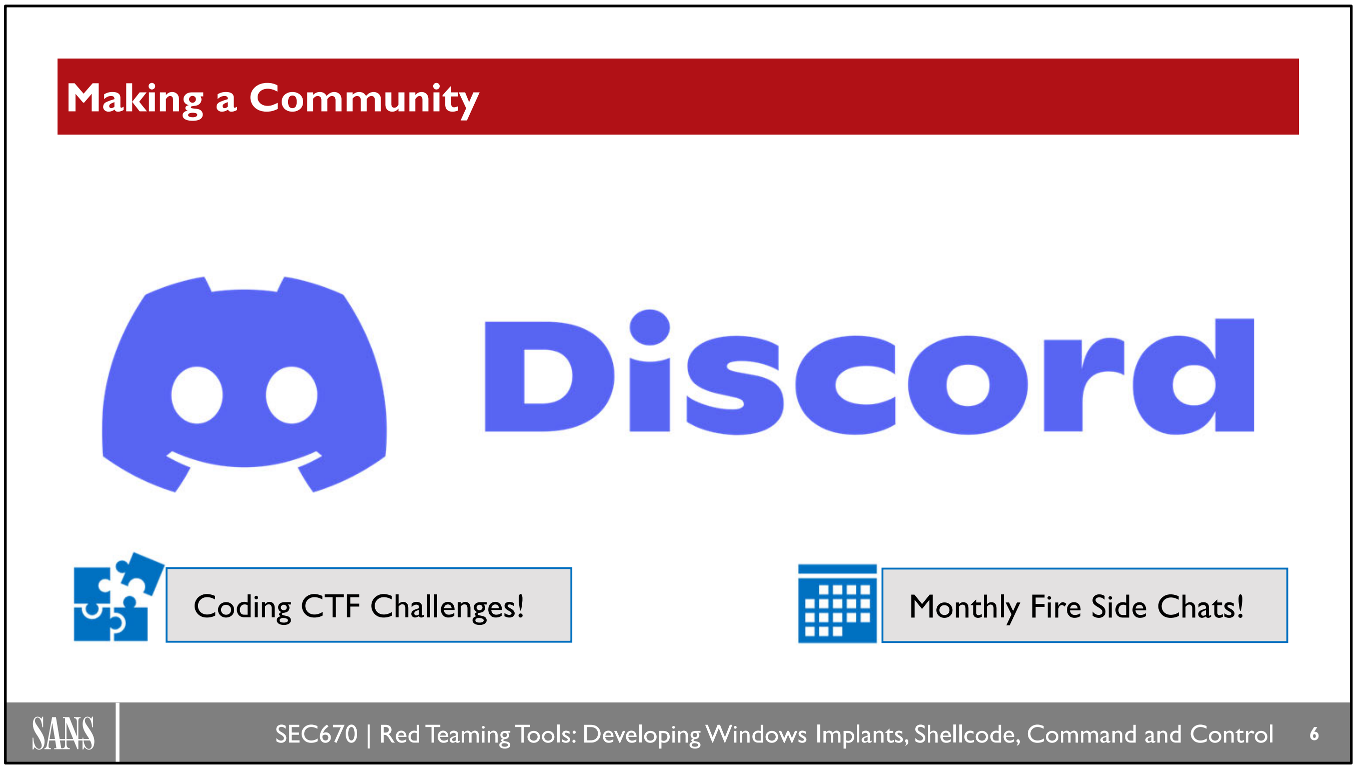


Objectives

Our objectives for this module are:

- Discuss what tool development is
- Determine what surveying the land means
- Figure out how to carry out operational actions
- Discover various persistence methods
- Enhance your implant

Objectives
The objectives for this course are straightforward. Each section will be talked about at a high level and then during that section's content, we will take a deep dive into what each one involves. We start off discussing tool development, what it is, how to do it, and different perspectives. Next, we will discuss surveying the land and how to develop custom recon tools. During operational actions, we will discuss what can be done after initial access is obtained: actions like ruining or corrupting data on the target, exfiltration, killing processes, etc. We will also go over several ways to keep your access to a target system. Finally, we will discuss how we can enhance the implant.



Making a Community

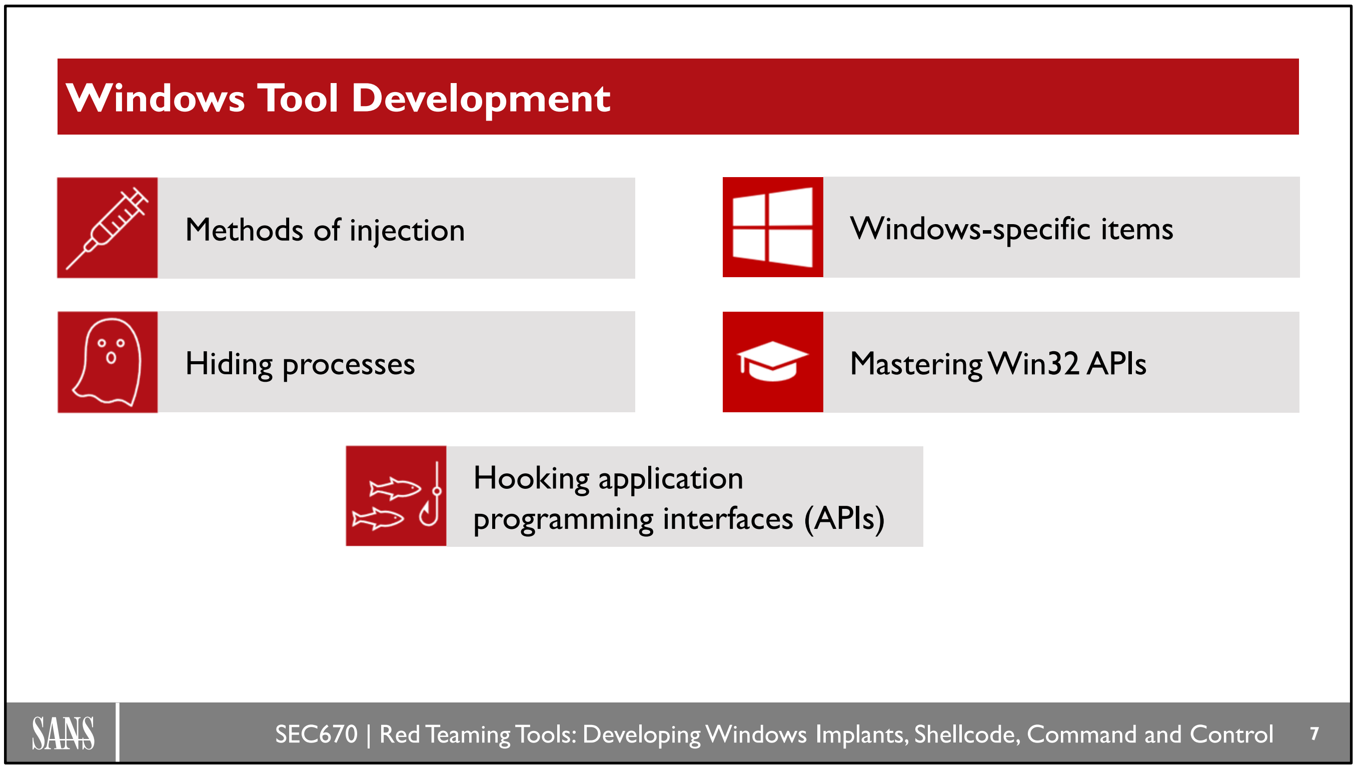
Discord

Coding CTF Challenges!

Monthly Fire Side Chats!

Objectives
This course has a custom Discord server, a community solely for Windows developers and those aspiring to become one. The server is open to anyone, but mainly created for students of this course. Private channels and special roles have been made only for you as a way for us to give back to you beyond the class. Please join the existing community of SEC670 Alum when you get a moment. See you there!

Reference:
https://discord.gg/offensivewindowsdev



Windows Tool Development

Methods of injection

Windows-specific items

Hiding processes

Mastering Win32 APIs

Hooking application programming interfaces (APIs)

Windows Tool Development
Windows tool development is more than just creating a console application that prints out "Hello, World!" Instead, we will focus on becoming familiar with Windows APIs that will leverage greater offensive capabilities for your tooling. In the world of offensive tools, it is hard to "see" the effectiveness of your tool. In fact, if you are really good at what you do, your tool will never be seen. Not really seeing a tool in action makes demos somewhat boring since there is no fancy GUI application that shows something taking place. Regardless, here is a short list of some items you will be able to take away from this course.

- Becoming familiar with Windows-specific items
- Mastering the Win32 APIs that leverage offensive ops
- Understanding various methods of injection
- Hooking APIs via various techniques
- Learning how to conduct research and create new techniques

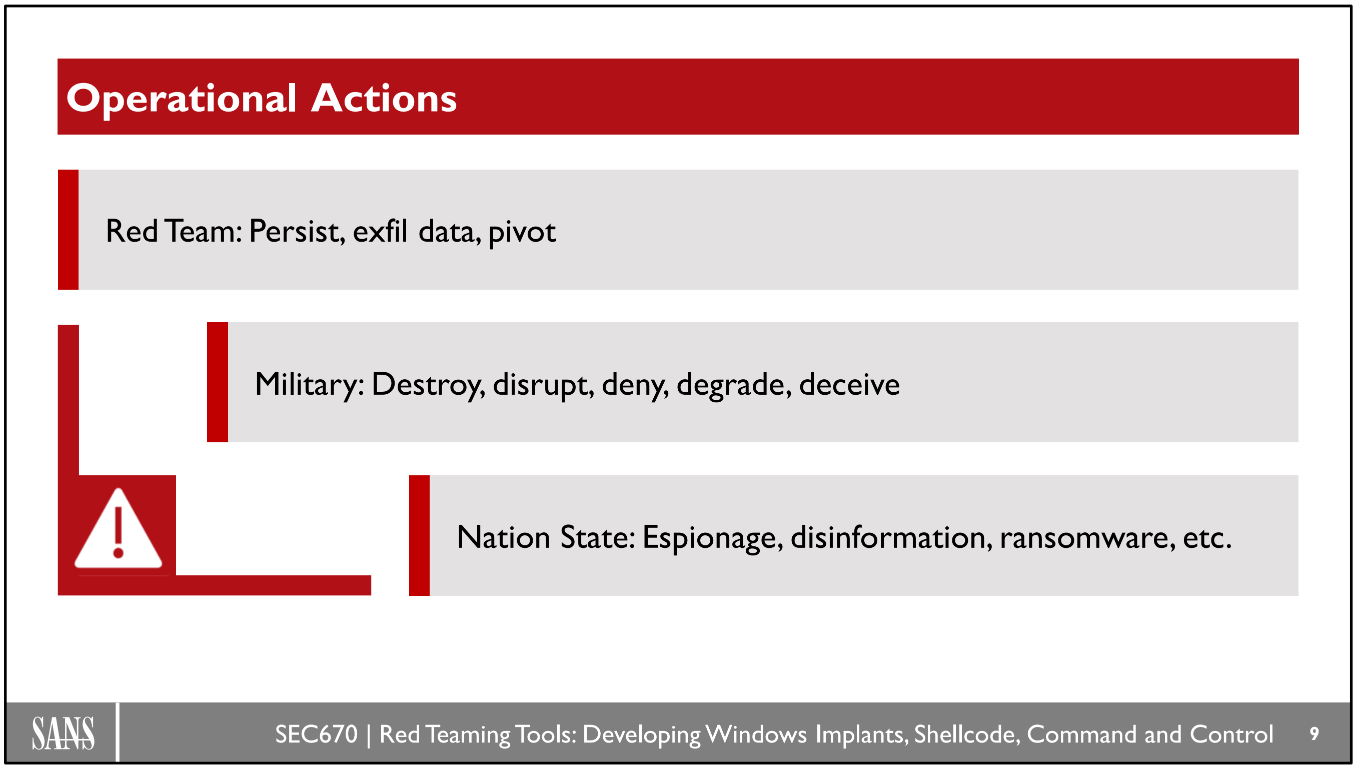
Getting to Know Your Target

What happens after you gain initial access to the target?

Typically, you are not the one who is creating the exploit that gives initial access to the operator. You are the one creating the tool that gets dropped after initial access. You aid in assisting the operator in determining the purpose of the target. Arguably, one of the best methods for accomplishing such a task is developing a host survey tool.

Getting to Know Your Target
This course is not aimed at gaining that initial access (that happens in SEC560), but rather what custom tools can be made to leverage that access. Since you are the developer, what actions are taken next are nearly limitless. You are no longer going to be bound by a current tool or C2 framework and you will not have to say, "I wish this tool allowed me to do this," or "I wish someone would create something that did this." Guess what—you are going to become that person who will create that missing thing. If you want a tool or a feature that enumerates specific registry keys, you can create that feature.

Perhaps one of the most important first tasks you perform with your initial access is determining the purpose of your target. Gathering as much information as possible will enable you to make a very accurate decision, and perhaps the most important decision to make from the gathered information is, "Do you continue?" Perhaps it would be better to clean off the target as best you can because you saw something in your gathered information that indicated the target was watched like a hawk or that the sysadmin was very tech savvy. Maybe you saw the installation of a new AV product you did not expect to see? The list can go on and on but in the end, creating and throwing down a host survey tool is a great first action.



Operational Actions

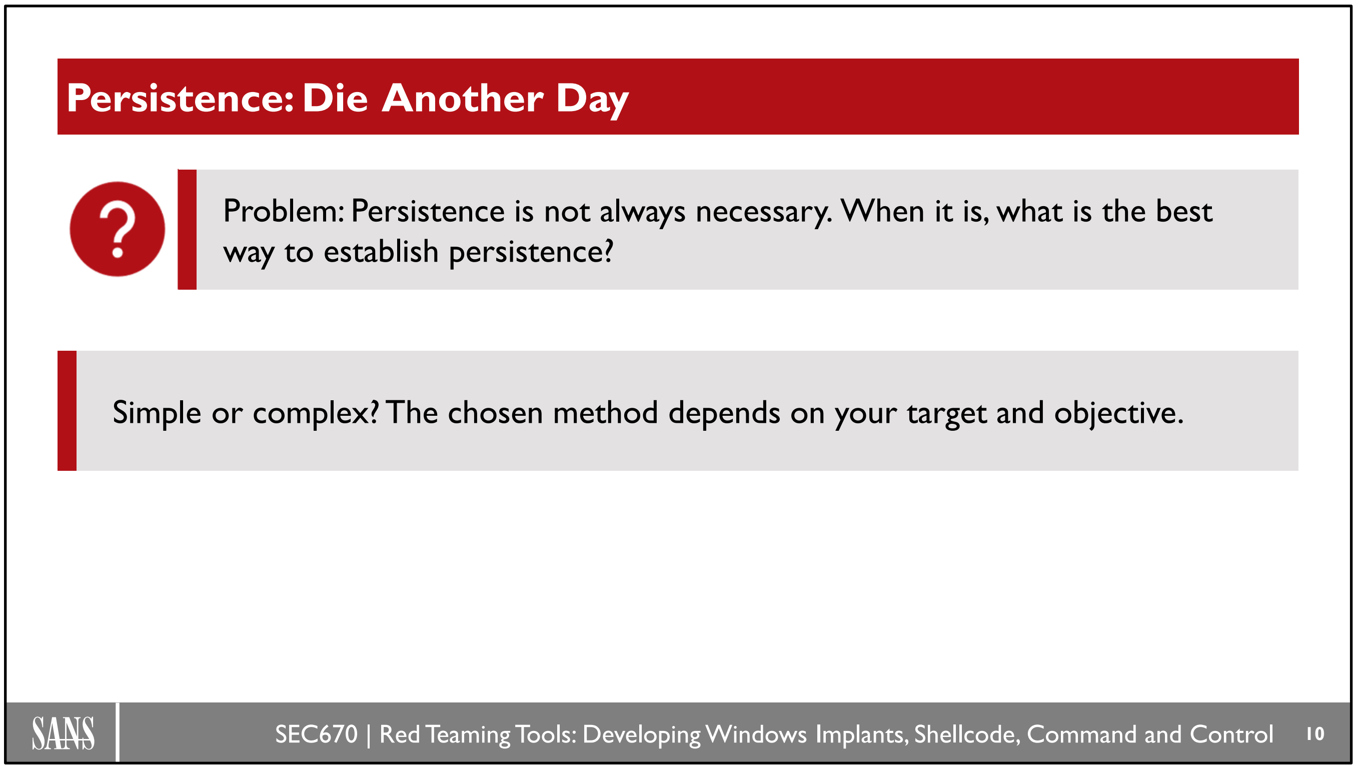
Red Team: Persist, exfil data, pivot

Military: Destroy, disrupt, deny, degrade, deceive

Nation State: Espionage, disinformation, ransomware, etc.

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 9

Operational Actions
After you have gotten to know your target a bit more, your objectives can be very different depending on the work role you are currently filling. If you are developing tools for a red team, you are probably not creating and releasing ransomware seeking out a payday. Instead, you might be working with the blue team to create a custom tool that is designed to test specific detections, which leads into purple teaming. If you are developing tools for your nation's military, then you probably have the authority to create tools that can destroy data on your target. The military typically has certain authorities granted by the government that authorize what operations can and cannot be executed, like dropping a missile on an ammo depot or wiping the computer of an enemy nation-state operator. If you are some rogue group actor or enemy nation state, then the sky seems to be the limit with what can be done. Countries and companies have suffered the effects of ransomware, espionage, data theft, and more.



Persistence: Die Another Day

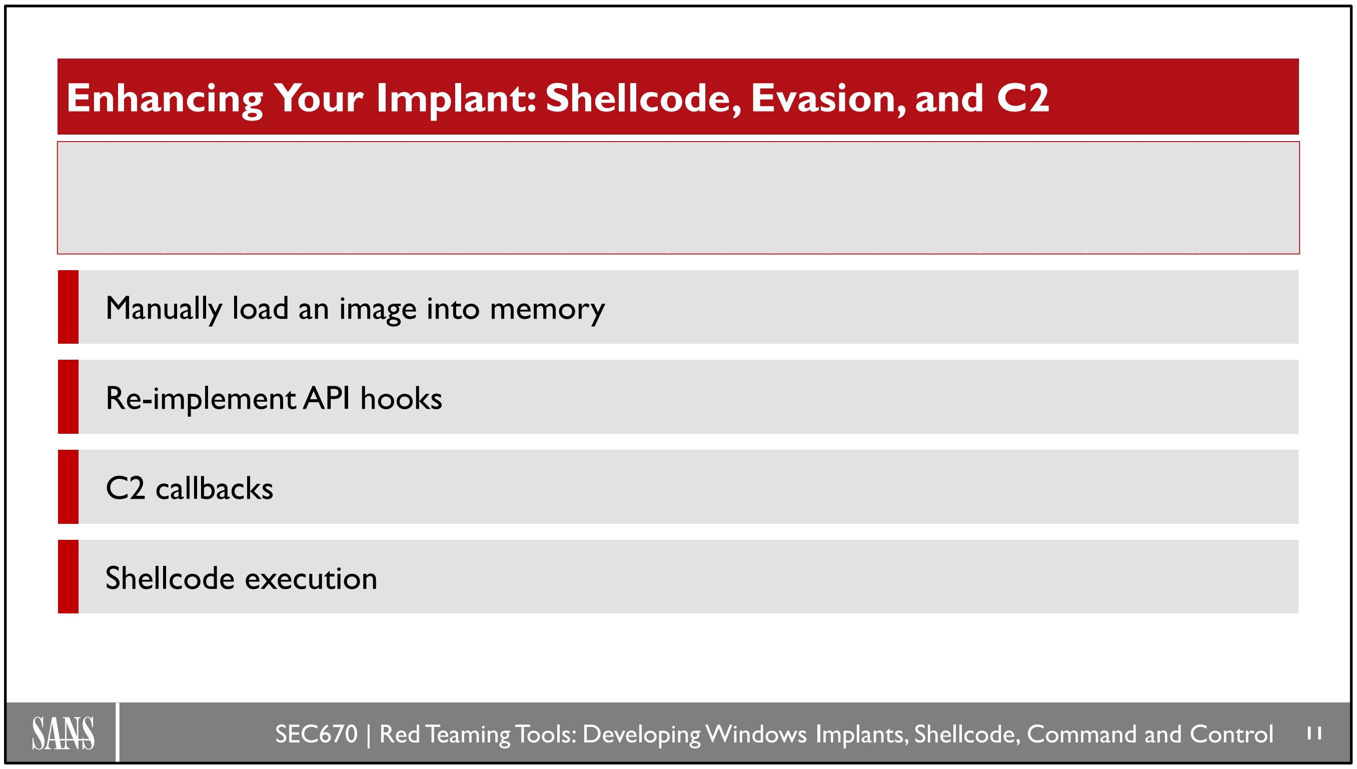
Problem: Persistence is not always necessary. When it is, what is the best way to establish persistence?

Simple or complex? The chosen method depends on your target and objective.

Persistence: Die Another Day
When a decision has been made to maintain access to a target, a red team operator or other operator position might want a few options for how to persist. An important question a red team operator should answer either before or during an op is whether persistence is required to successfully complete your objective(s). There is risk with establishing a foothold and that risk must be weighed against your target, your objective, and being caught. A lot of times, persistence can require pulling down another tool, your precious implant that your team of developers worked on for months on end. Are you okay with giving up that tool just to persist on that one target? If not, then complete your op and clean off. If you are, then determine what persistence method will work for your op and install your implant.

There is more than one method for persisting on your target and some methods might get you caught faster than others. The method chosen should match up with how important that target is to your op. Perhaps a simple modification to a Registry Run key is enough because nothing is in place to monitor it. Perhaps you need to be more creative and stealthier because the admin of that box is savvy and has tools in place to watch out for common locations like Run keys.

In the end, the lowest hanging fruit wins. Meaning, you should go after the easiest method that you know you can get away with on target.



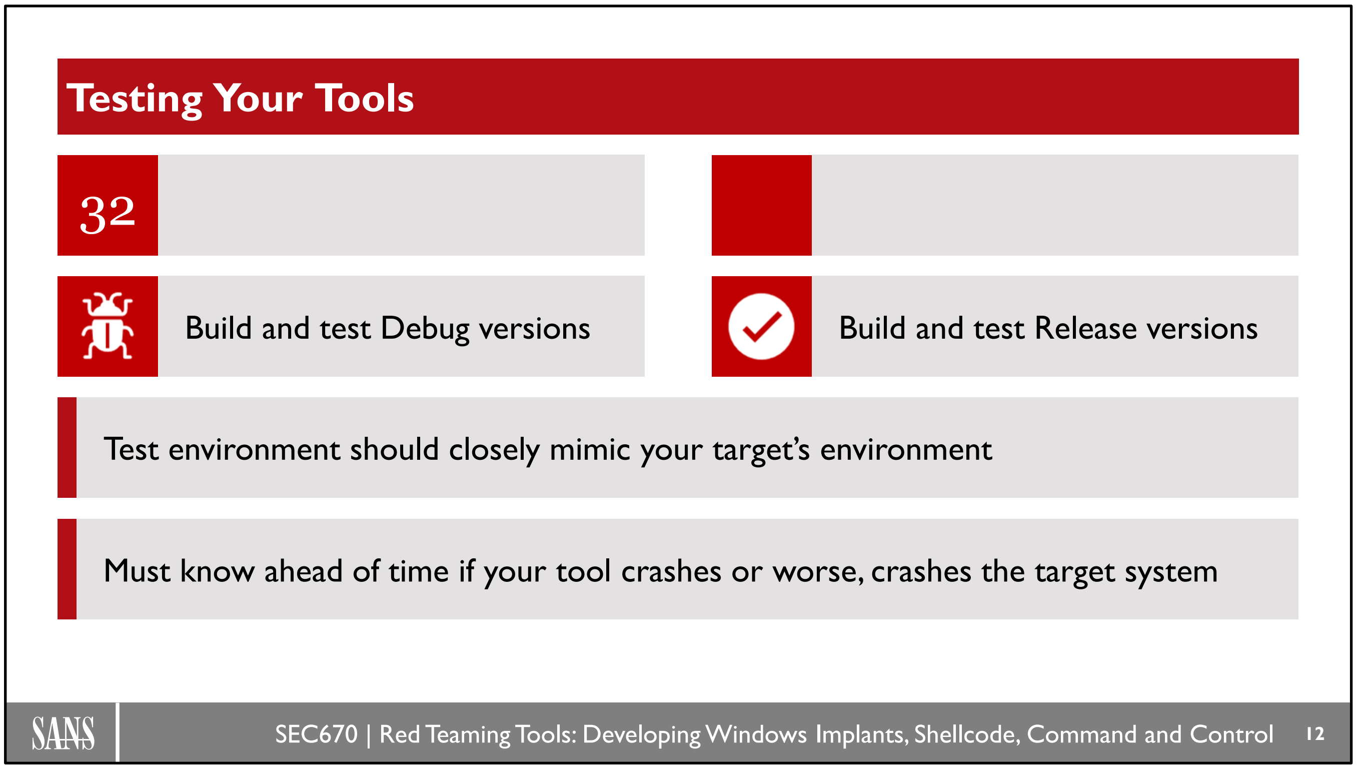
Enhancing Your Implant: Shellcode, Evasion, and C2

- Manually load an image into memory
- Re-implement API hooks
- C2 callbacks
- Shellcode execution

SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 11

Enhancing Your Implant: Shellcode, Evasion, and C2
Sometimes you will be forced to produce a more advanced tool to get the job done. These should be a last resort option because, depending on what capability you are developing, it can take a while to develop and release. Plus, do you want to spend a few months developing an advanced capability for an easy to knock over target? If there isn't a single AV product installed and the sysadmins are lazy, then why bother pulling down your most prized possession?

Many of the advanced techniques will require intimate knowledge of the PE32 and PE32+ format, process virtual address space, the registry, OS internals, etc. Such mastery comes after years of tool development and debugging undocumented APIs. Also, not much is going to happen if you cannot control your implant and give it taskings. For tasking to happen, it must have the ability to communicate out of the target network and to your C2 infrastructure. Also, having the ability to execute custom shellcode payloads is a great feature to implement.



Testing Your Tools

32

Build and test Debug versions

Build and test Release versions

Test environment should closely mimic your target’s environment

Must know ahead of time if your tool crashes or worse, crashes the target system

Testing Your Tools
Once your tool is finally developed and ready for operational use, you must resist the urge to use it anywhere and everywhere you can. One of the first things you should do before operational deployment is test it. Proper testing is extremely important if time permits because there are rare conditions that require modifications to already existing tools on the fly. You should have the skillset of being able to tear down your code too after compilation, meaning disassembly. Reversing your tool and stepping it through with a debugger will be not only valuable for fixing bugs but also for verifying your code. Did the compiler do what you were wanting it to do, or do you need to pass it some flag options to change things a bit? You can change certain compiler and linker options to shrink the overall size of the binary. You could also eliminate the dependency on the C runtime along with other dependencies. Eliminating external references, or XREFs as we call them, can aid in making your code position independent. Position independent code, or PIC, is something that we will dive into at the end of the course.

Another item you might want to take into consideration would be easily identifiable strings. Strings are always the easiest item to find thanks to tools like *strings.exe* from the Sysinternals Suite of tools, but they can also give away too much about what your tool is designed to do on target. If you can eliminate them altogether that would be best—the least you can do is obfuscate and/or encrypt your strings. Section 5 will lightly touch on how you can cut down on strings in your compiled binary.

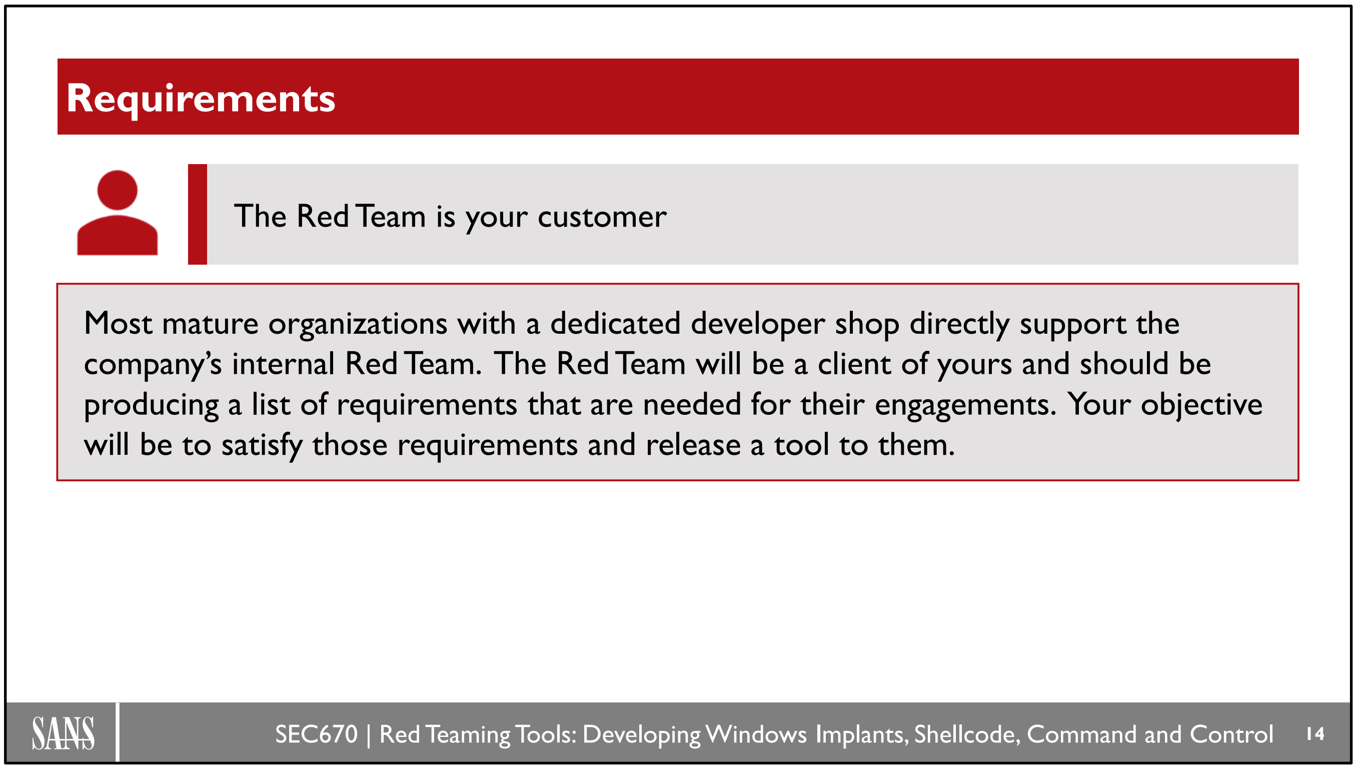
Be Creative

- Must learn to think outside the box; be creative
- Can assist with making your tool different from others; signature-wise
- Could also lead to discovery of new techniques or capabilities

Be Creative
Tool development can be greatly limited by your creativity and/or your knowledge of operating system internals. It is great to ask yourself questions like, "What happens if I implement this?" or "What if I trigger this event and do this?" Questions like that and many others are what pull out the creative side of your brain. This is how new capabilities are developed as well as how new vulnerabilities can be found. Creativeness, or newly found ideas, is not the best place for production code that is going to be used for a high value op. Explore and test what you found to explore and discover any edge cases.

One person online who does an amazing job at showing creative thinking is x86matthew. The URL for his blog post is down below, and in his posts, he will explore how to do basic, everyday tasks, like writing to process memory, in a different way. There are other, lesser-known APIs that can do the very same thing that the more commonly used APIs can do. When you can find a completely different API to accomplish the same thing, use it, because EDR solutions might not be looking out for those APIs.

Here is the URL for x86matthew: https://www.x86matthew.com.

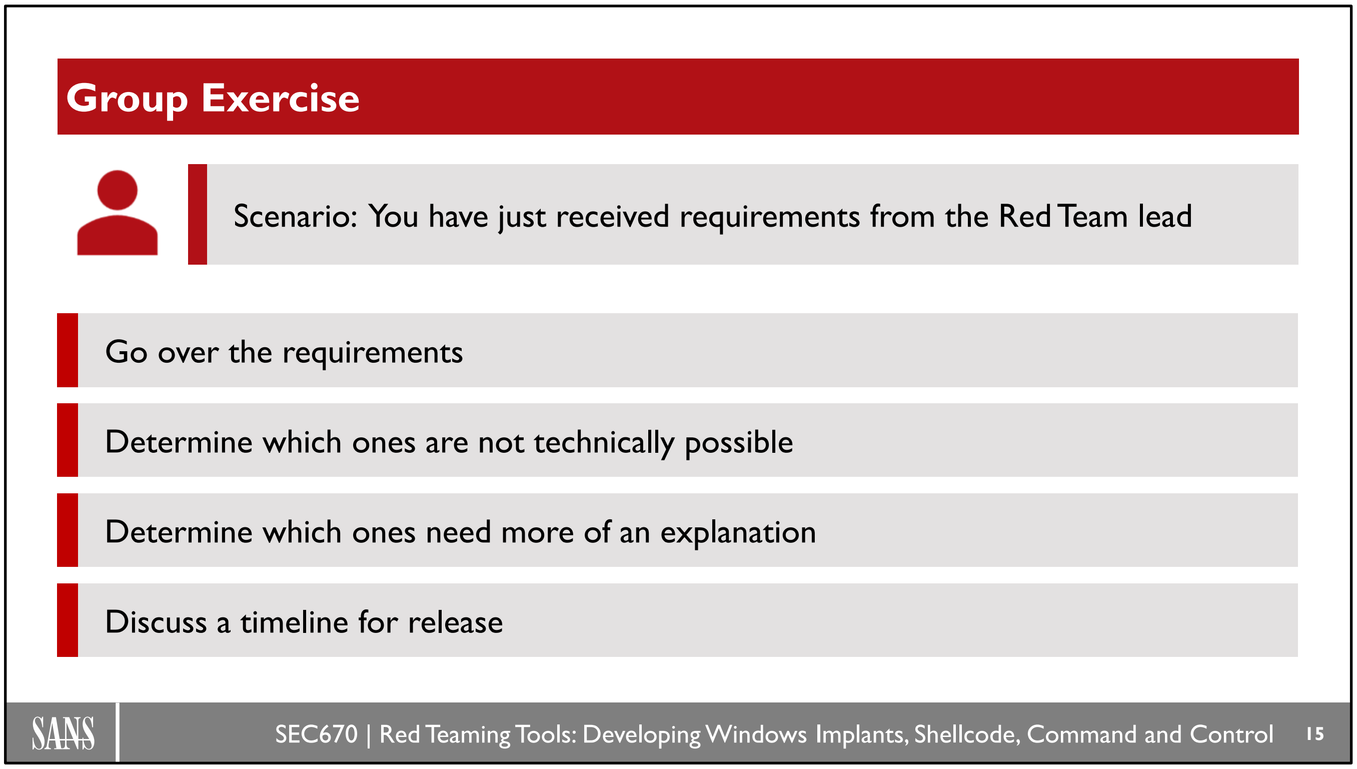


Requirements

The Red Team is your customer

Most mature organizations with a dedicated developer shop directly support the company's internal Red Team. The Red Team will be a client of yours and should be producing a list of requirements that are needed for their engagements. Your objective will be to satisfy those requirements and release a tool to them.

Requirements
When it comes to determining how to make a capability that does X, Y, and Z things, you need to think of the end user of your tool and how they might want it deployed in an engagement. When the roles are split out properly, meaning a company does not require their Red Team operators to be developers as well, they will have created and properly funded a development shop whose sole purpose is to support the Red Team or the Penetration Testing Team. There should be some formal system in place for how the Red Team gives the development shop a list of requirements for a feature to add to an existing tool or to make a brand-new tool from scratch. This could be a Jira ticket, a Word doc, an email, etc., something that has a paper trail and is formal. The development shop should then discuss those requirements to determine how long it might take to meet them and go over if some of them are not even feasible. Once a potential date is determined, the tool should be released and formally accepted by the Red Team.



Group Exercise

Scenario: You have just received requirements from the Red Team lead

Go over the requirements

Determine which ones are not technically possible

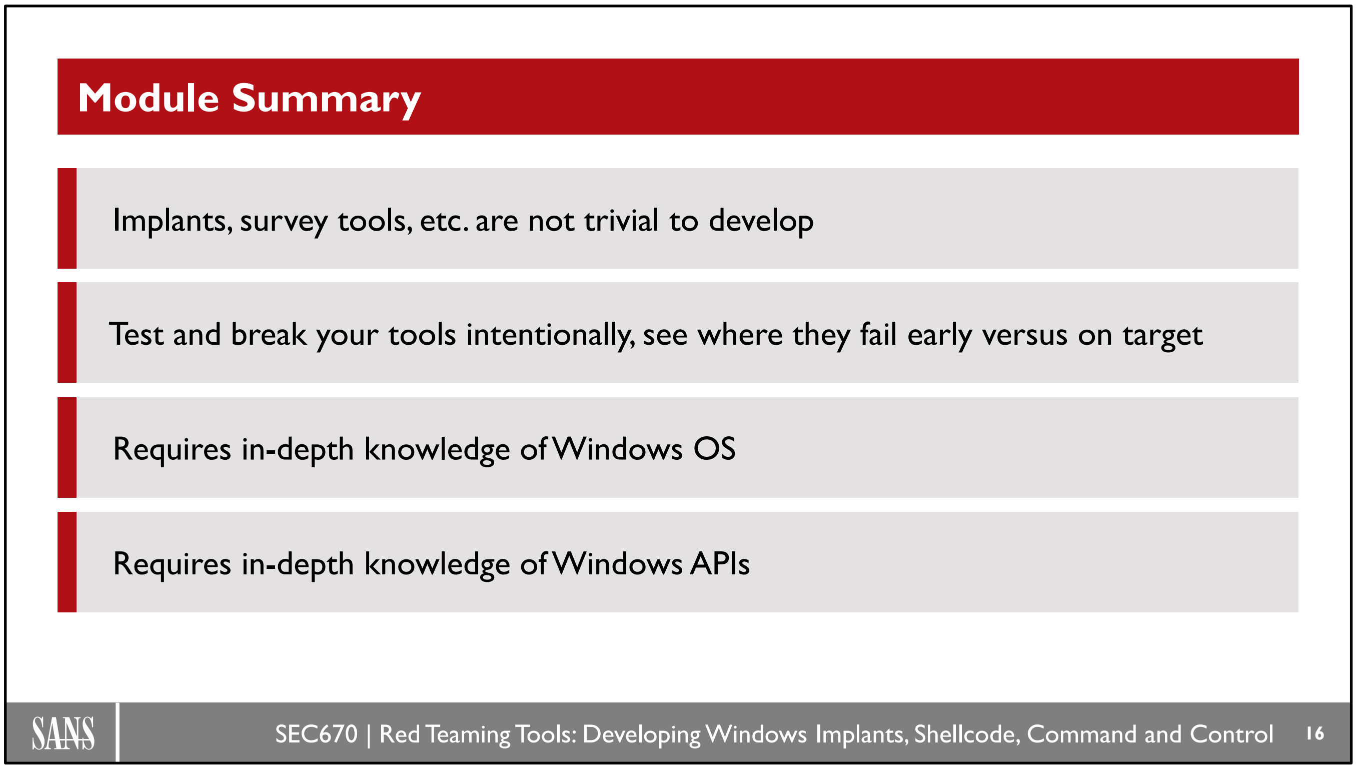
Determine which ones need more of an explanation

Discuss a timeline for release

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 15

Group Exercise
Let us pause for a moment and go over a scenario where you are the team lead for the development shop. The Red Team lead has just created a ticket that has a fairly large number of requirements in it for a new capability for an upcoming engagement they have. The Red Team will be facing a new security product they have yet to test against but have now required that from you. Here is the list of requirements the Red Team lead has given along with a timeline:

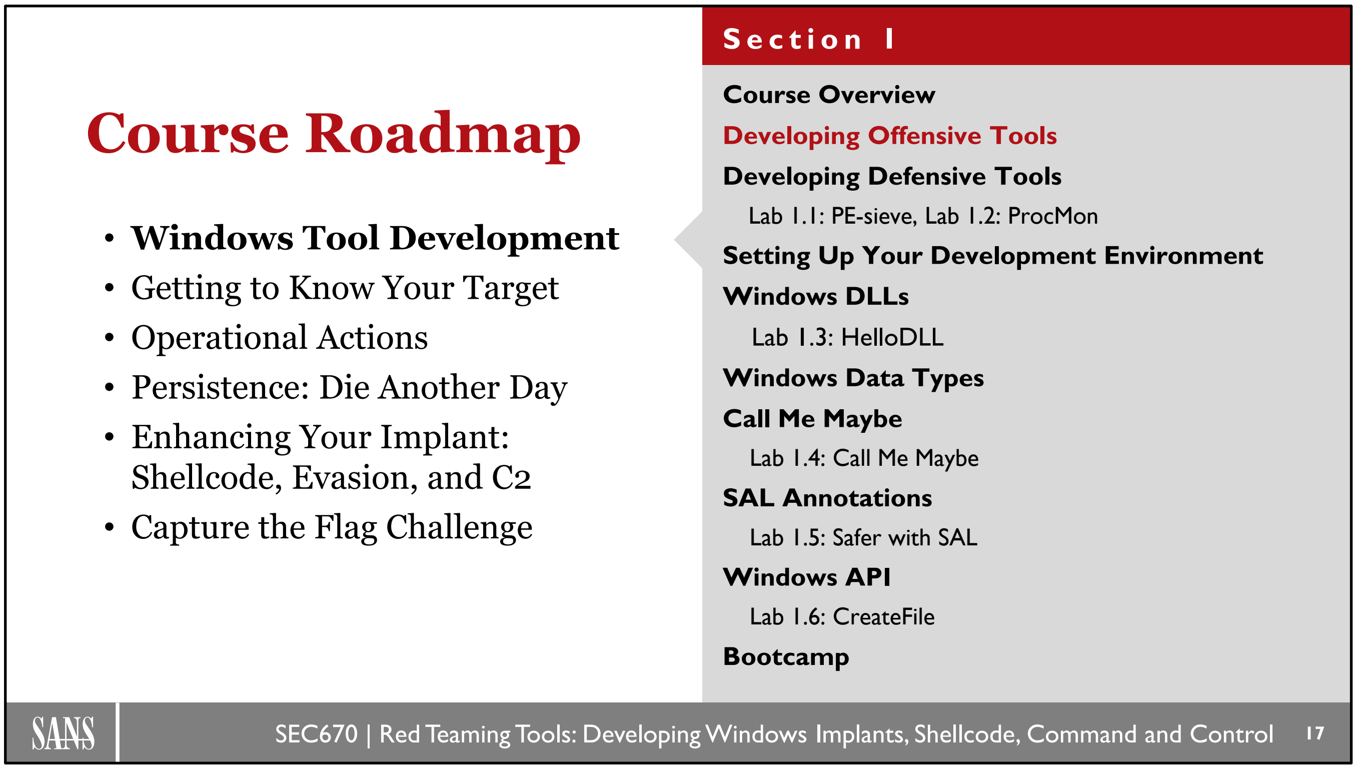
- Compiled 32-bit and 64-bit binaries
- Main program is an EXE
- Additional payloads in DLL format
- Must be no larger than 400kb
- Must perform recon
- Must be able to do code injection
- Must be able to persist
- Must be able to elevate privileges
- Must be able to bypass security products user-mode hooks
- Must be able to load additional payloads, e.g., DLLs, shellcode blobs, etc.
- Must communicate over HTTP with Python C2
- Must support download capabilities from target
- Time to release: 6 days



Module Summary

- Implants, survey tools, etc. are not trivial to develop
- Test and break your tools intentionally, see where they fail early versus on target
- Requires in-depth knowledge of Windows OS
- Requires in-depth knowledge of Windows APIs

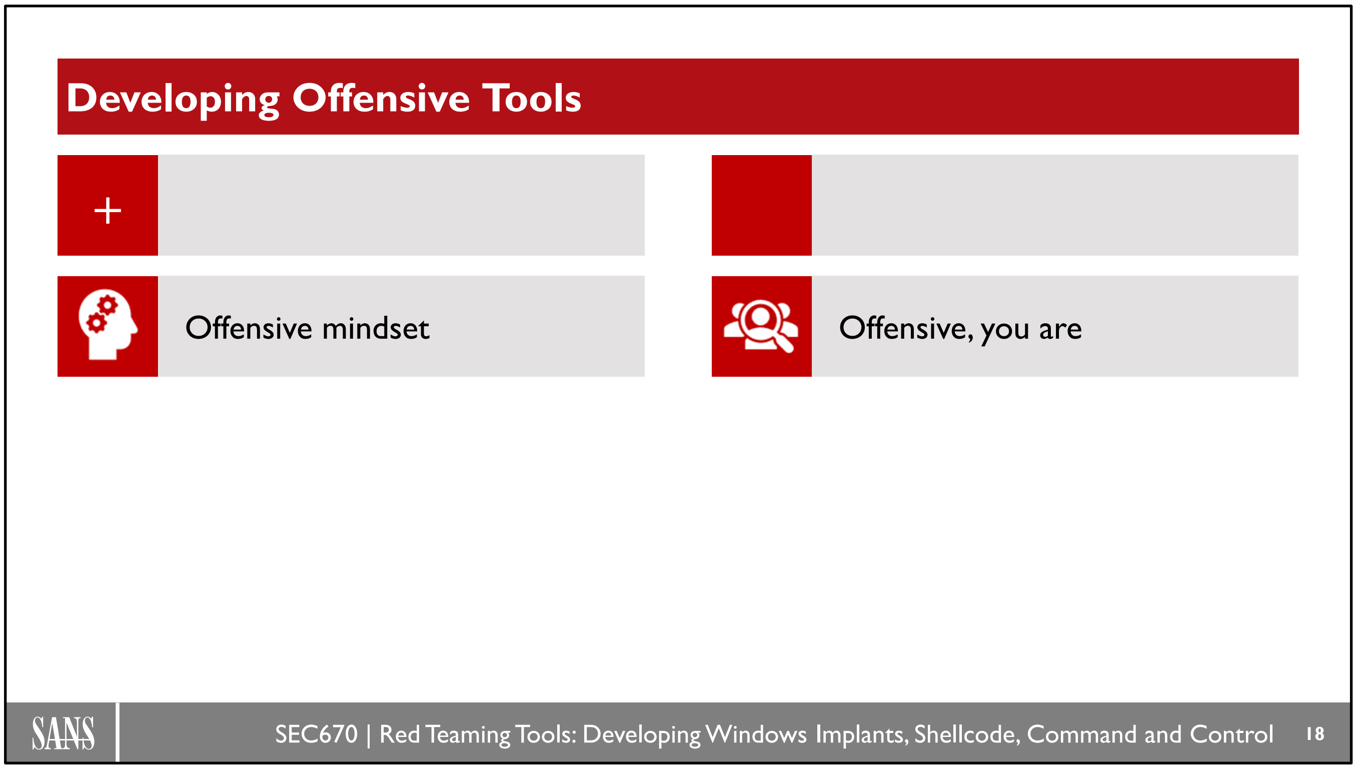
Module Summary
It is not easy to develop well written and well performing code, but this is a necessity. The tools that you develop must never crash your target and should never run with unexpected results. Your testing should tell you everything your tool will do and will not do, like what breaks it. Unleash your creative side and explore the realm of new possibilities.



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

In this module, we will discuss what it means to develop offensive tools.



Developing Offensive Tools

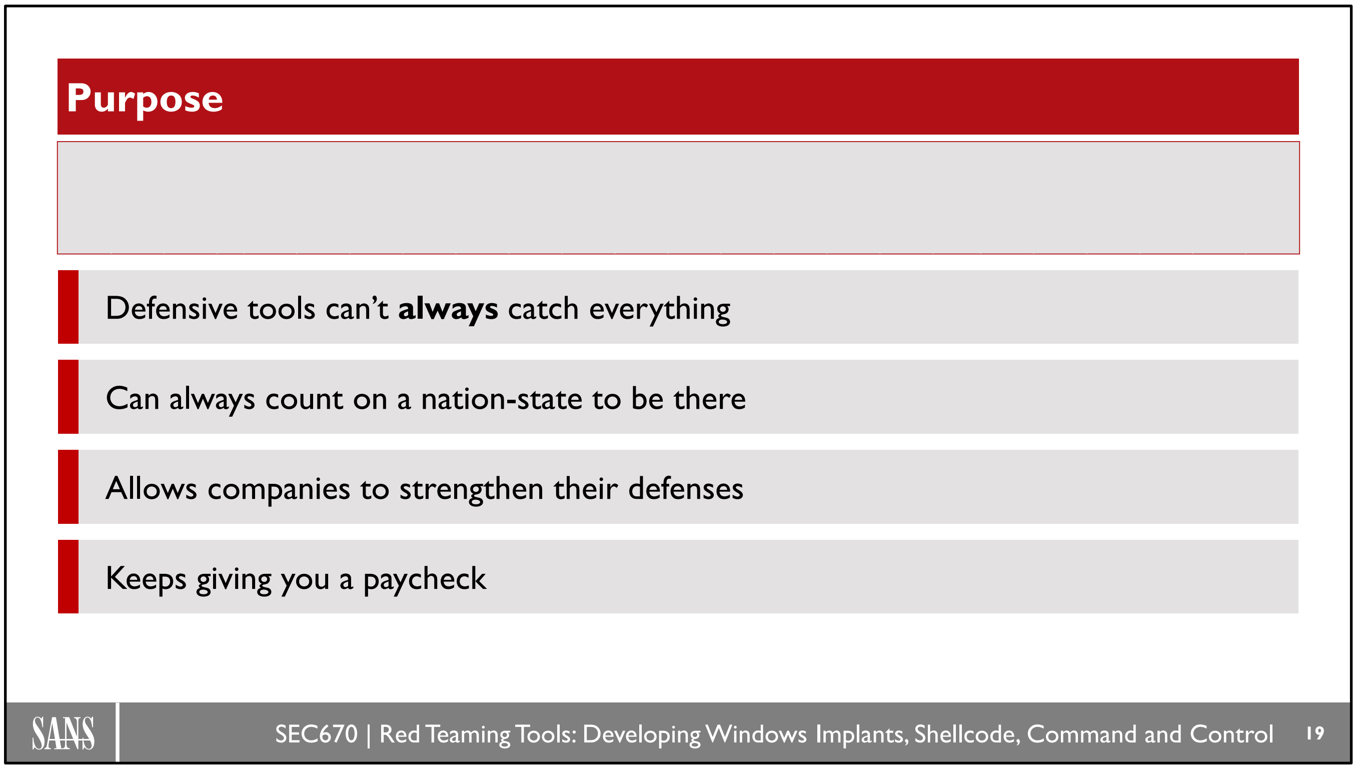
+

Offensive mindset

Offensive, you are

Developing Offensive Tools
Tim Medin, Founder and CEO of Red Siege, also a SANS Instructor/Author for SEC560, has a saying, "I am offensive." This saying is great because being offensive or having that offensive mindset is what will help you achieve your goals, and our main goal for this course is making you an offensive tool developer for Windows targets. The offensive mindset and action is all about taking advantage of weaknesses or vulnerabilities, exploiting built-in features to establish persistence, avoiding AV/EDR solutions, and out-maneuvering sysadmins.

You are going to be developing programs that do something nefarious. We will not be developing tools to detect malicious actions but to create the tool that puts those detection tools through the paces. If you are coming from the defensive side and have the defensive mindset, great! You can use that as an assist and eventually you can switch your mindset to more of an offensive one.

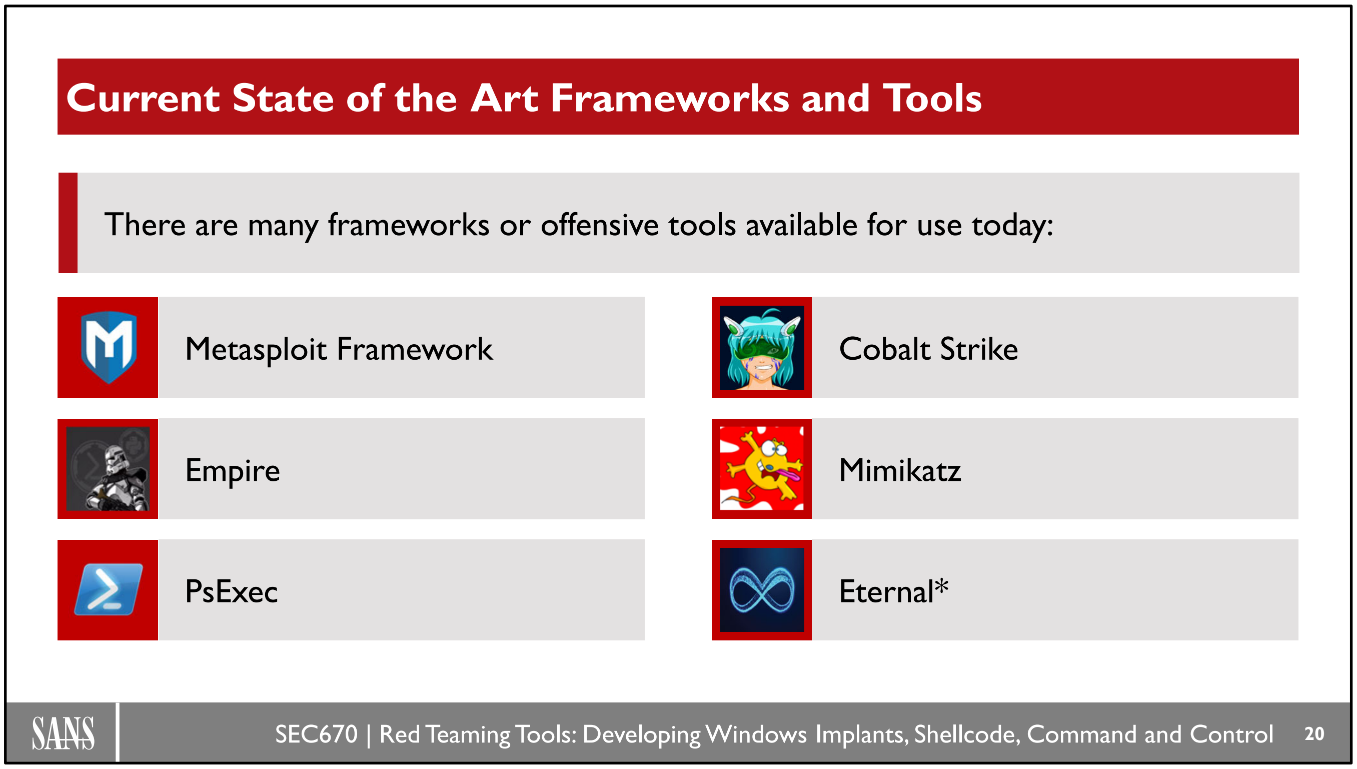


Purpose

- Defensive tools can't **always** catch everything
- Can always count on a nation-state to be there
- Allows companies to strengthen their defenses
- Keeps giving you a paycheck

Purpose
Offensive tooling is necessary for several reasons, the first one is that you probably would not have a job without it. Several organizations might even require external assessments to be conducted every so often. Financial institutions have audit requirements that must take place at certain intervals, and an external red team might serve that requirement. Another reason would be for national security because you do not want enemies of your country to gain unauthorized access to your network all while you are trying to get access to theirs. How do you think you get access to their network aside from human operations or something similar? The answer: offensive tools.

Perhaps another reason is so we can better defend our networks from enemy nation states and non-nation-state actors by having effective red team ops using similar tools. The more you can practice your defense the stronger your defenses become, but you cannot practice alone. You need someone to practice against, so that you can identify possible weak points in your playbook. Until we can rid the world of bad actors, we will always need offensive tools or capabilities.



Current State of the Art Frameworks and Tools

There are many frameworks or offensive tools available for use today:

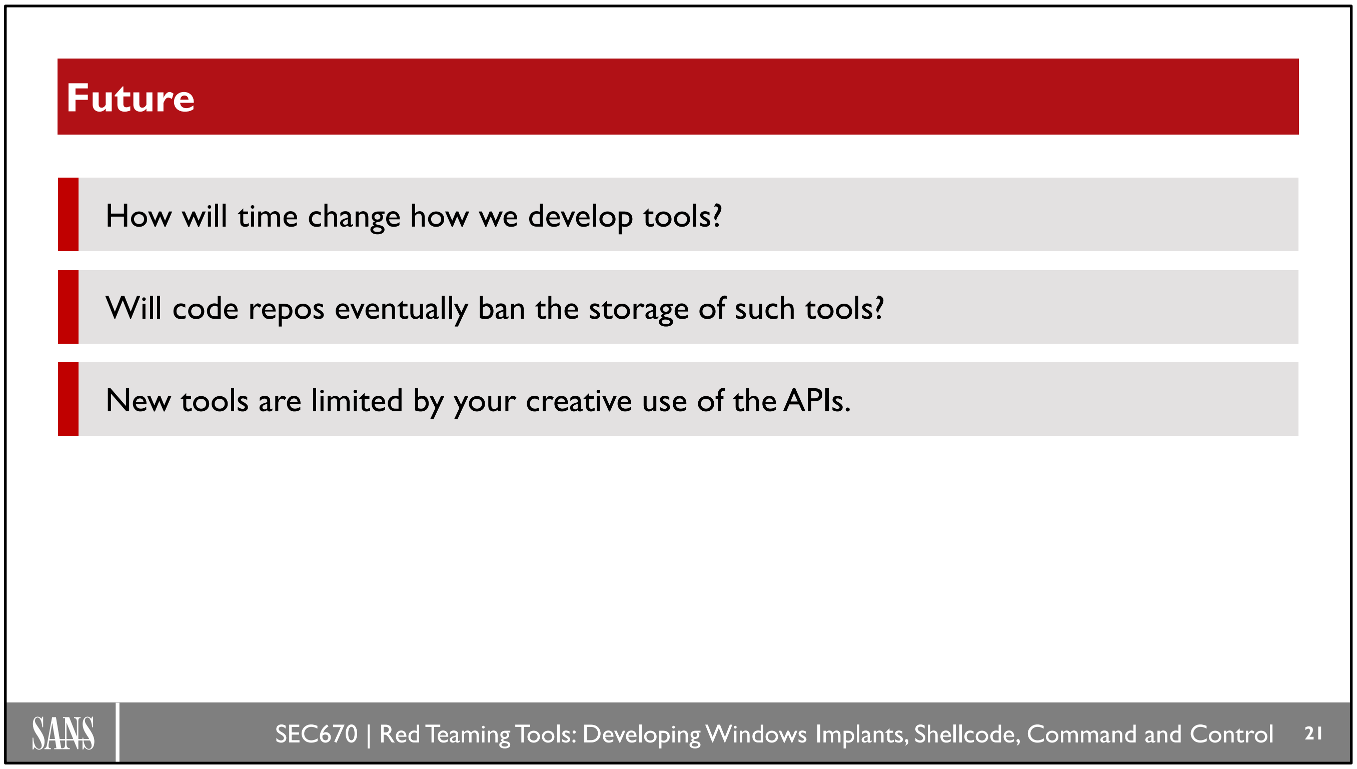
- Metasploit Framework
- Cobalt Strike
- Empire
- Mimikatz
- PsExec
- Eternal*

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 20

Current State of the Art Frameworks and Tools
Today, there are some amazing tools out there and many are free for anyone to grab and leverage. We see this happening very frequently and such actions have sparked intense discussions regarding open sourcing powerful tools and capabilities that nation state and non-nation-state actors can use against us. Regardless of where you stand in that debate, offensive tools will almost always be needed so we can sharpen up defenses.

In the open-source world, or the public community, the tooling is getting better as new techniques are discovered. Researchers and developers are getting more creative with their craft. Developers are discovering new methods to bypass AV/EDR solutions, or bypass UAC restrictions, or elevate privileges. Take #Zerologon, for example: at a high level, a researcher discovered that you could, with access to a domain controller, create an account in the domain with an empty password, which allowed for a complete takeover of the DC. Once a vulnerability is found then it is time to develop a tool to exploit it, which could then be added as a capability to an existing framework or a custom in-house tool.

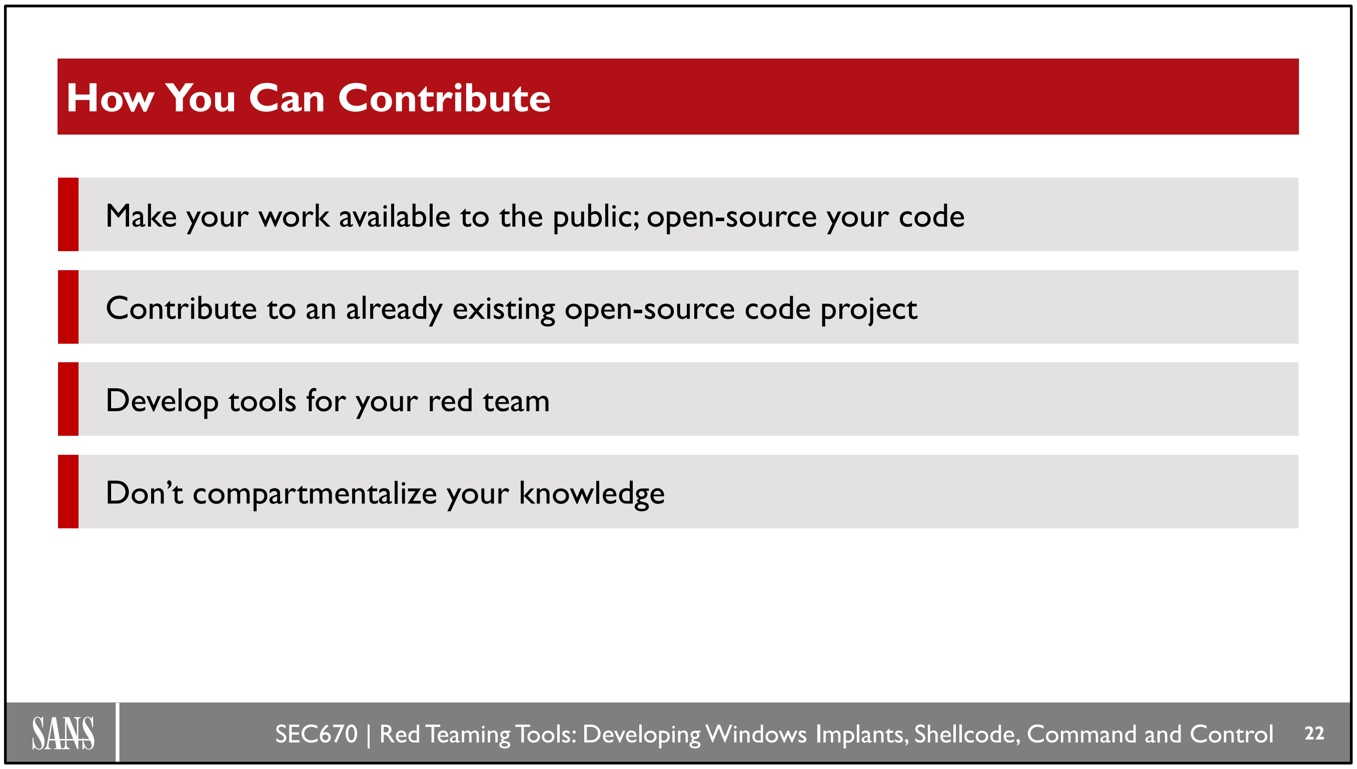
There are several free tools that offer great capabilities that also might have a commercially licensed version that offers more advanced capabilities. This is where the major differences are. When tools come with a heavier price tag, you should expect it to do a lot more for you out of the box, or at least allow you to fully customize certain parts of it. What is being used at your organization: commercial, open-source, or your own internal framework?



Future

- How will time change how we develop tools?
- Will code repos eventually ban the storage of such tools?
- New tools are limited by your creative use of the APIs.

Future
The future can be somewhat what we make it to be if you stop and think about it. If you close off all creativity, ban the posting on offensive security tools to GitHub and other similar sites then you could run the risk of not advancing by much. On the flip side, if your organization can invigorate and inspire creative thinking, then the future for developing new offensive tools and capabilities could be very bright.

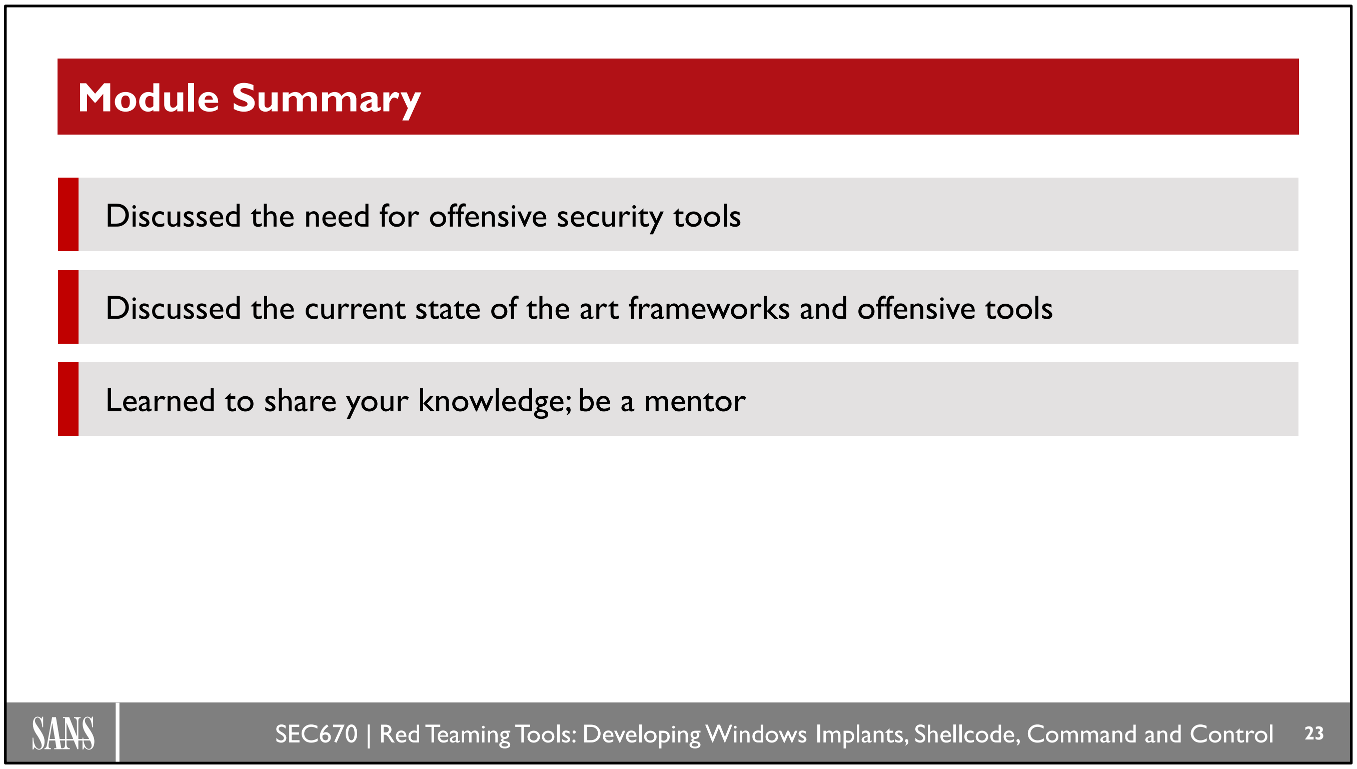


How You Can Contribute

- Make your work available to the public; open-source your code
- Contribute to an already existing open-source code project
- Develop tools for your red team
- Don't compartmentalize your knowledge

How You Can Contribute
Before you start publishing your offensive tools online, be sure to check with your current employer as to what their intellectual property terms are. You do not want to be in a situation where whatever you create during your own free time with your own equipment belongs to your employer. It could land you in some hot water with a legal team. If you are good to go on that front then by all means, get your work out there. Feel free to contribute to an existing project or framework like Metasploit or Sliver C2. It can be a great way for you to practice and sharpen your development skillset.

If you are not already on a red team, then perhaps you could approach your company's red team lead or director about creating a tool development section within the team, if they do not have one already. Today, more and more red teams are creating development roles to create in-house tools. Whatever you choose to do, share your knowledge with those around you, especially those who are trying to get into this line of work. Compartmentalization just creates a single point of failure and is not good for the overall success of a team.

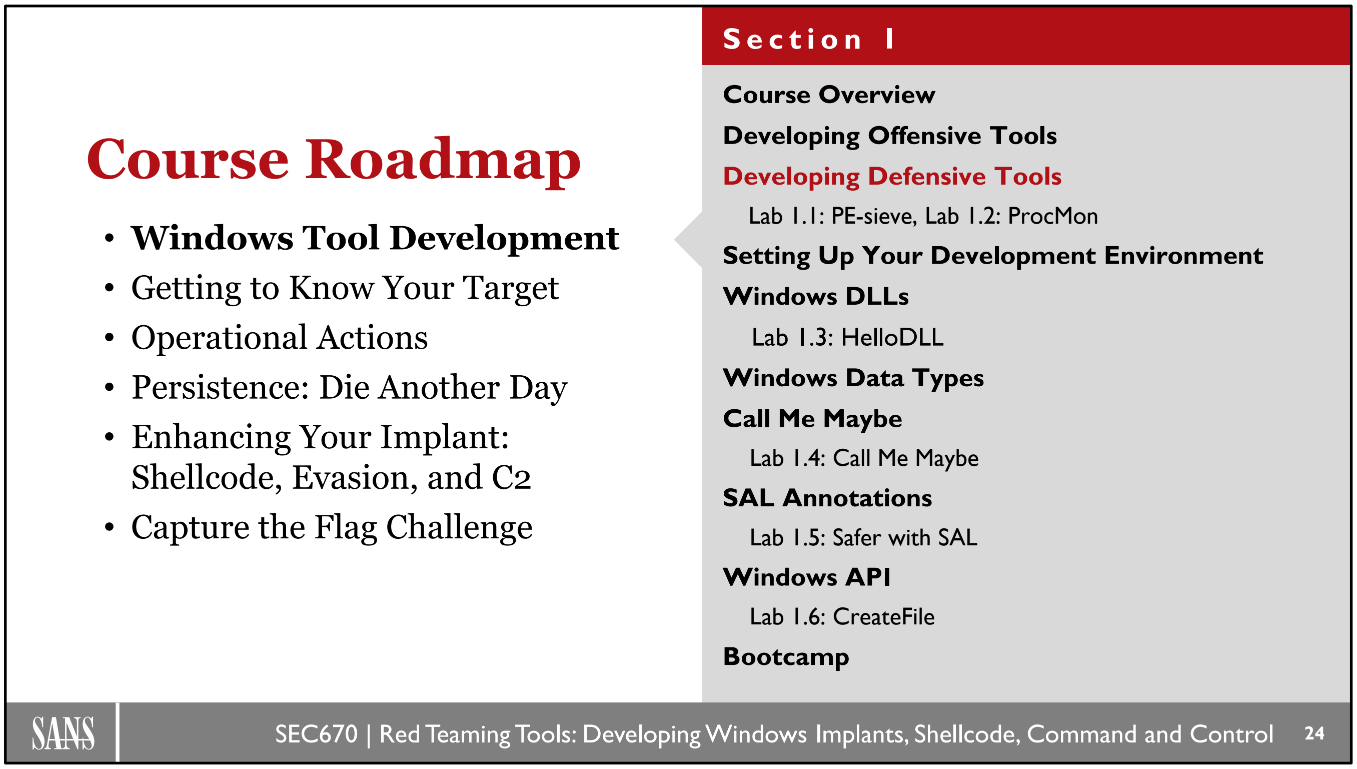


Module Summary

- Discussed the need for offensive security tools
- Discussed the current state of the art frameworks and offensive tools
- Learned to share your knowledge; be a mentor

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 23

Module Summary
In this module, we discussed several reasons why it is necessary to develop offensive security tools. We also discussed various state of the art frameworks and offensive tools, where we can take things from here, and not hoarding the knowledge and experience you might have gained over the years.



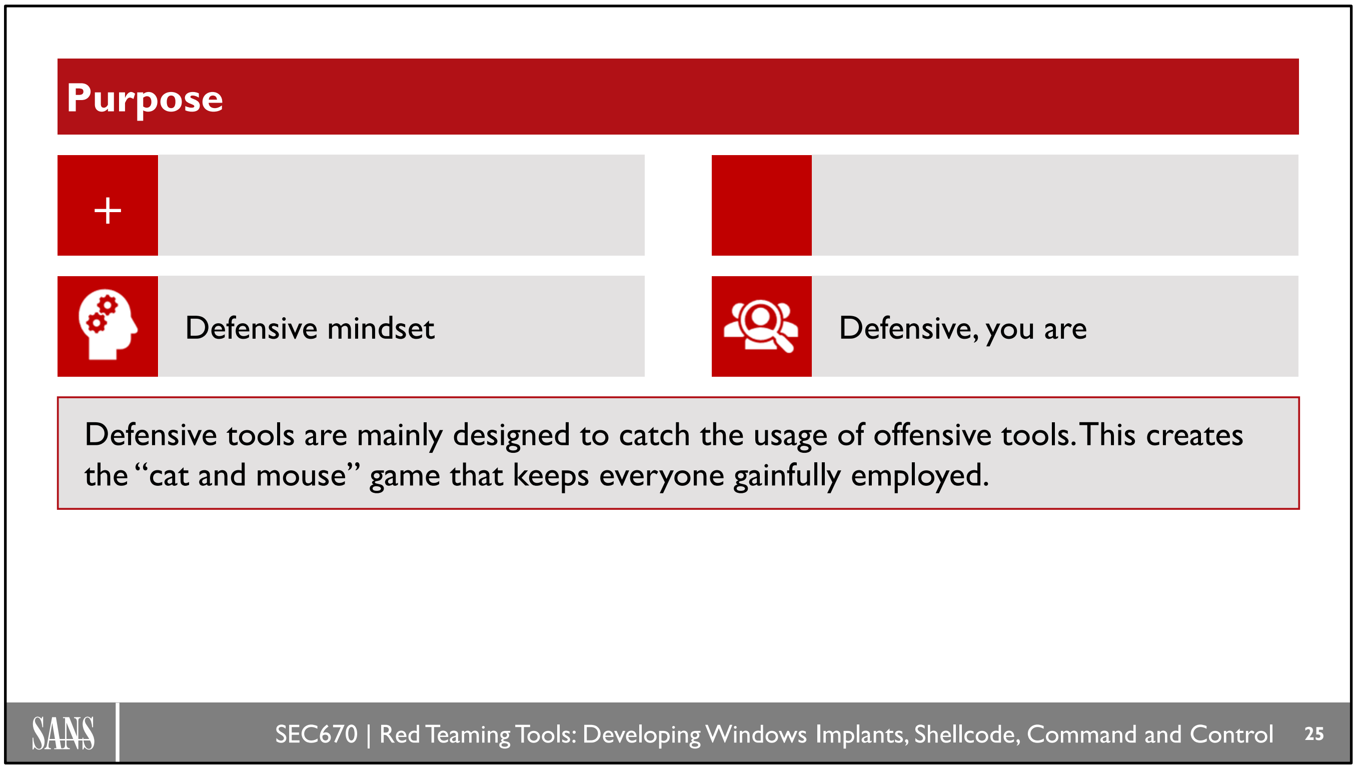
Course Roadmap

- **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 1

SANS SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 24

In this module, we will discuss the development of defensive tools, their purpose, state of the art tools, their future, and how you can contribute.



Purpose

+

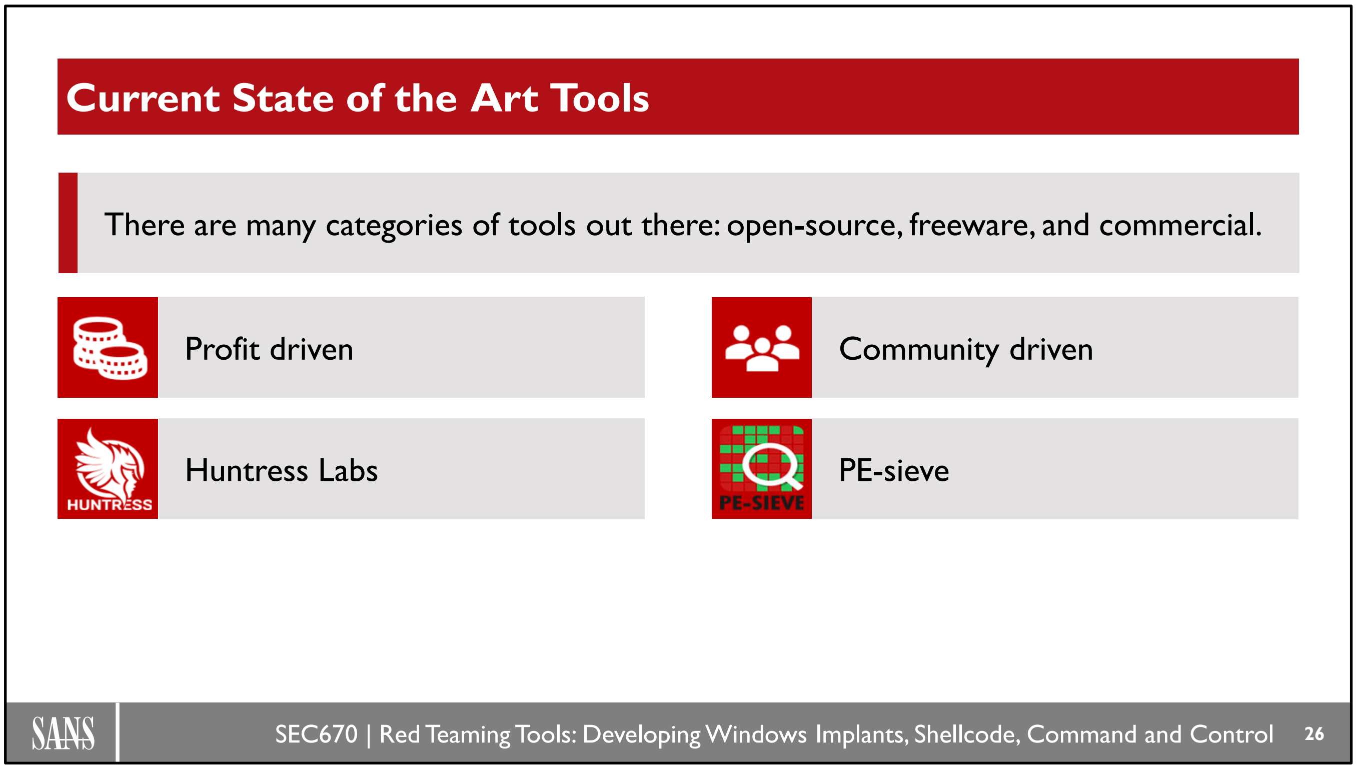
Defensive mindset

Defensive, you are

Defensive tools are mainly designed to catch the usage of offensive tools. This creates the "cat and mouse" game that keeps everyone gainfully employed.

Purpose
Just like offensive tools, there is a necessity for defensive tools. Many companies profit from selling their defensive tools and there is nothing wrong with that. There have been many defensive tools that have been open sourced and pushed to GitHub for all the world to see. Some are truly open source while others are simply freeware, and there is nothing wrong with that, either. Defensive tools, when operating at their best, can be used to detect the execution of your offensive tools. As new offensive tools are created and used, defensive tools must be modified to check for those new methods. This back and forth is a cat and mouse game that will be played for many years to come.

Programming securely can be often viewed as a defensive programming technique. When done correctly it makes the job of vulnerability researchers and exploit developers that much more difficult. If you do not check for logic bugs, memory leaks, overflows, etc. before you release your product, then rest assured that someone else will find it for you; it is only a matter of time. Solid defensive programming is incredibly difficult, and you have to get it right every single time because that one time you do not, there is either a red teamer or a bad actor that is ready and willing to exploit the flaw.



Current State of the Art Tools

There are many categories of tools out there: open-source, freeware, and commercial.

Profit driven	Community driven
Huntress Labs	PE-sieve

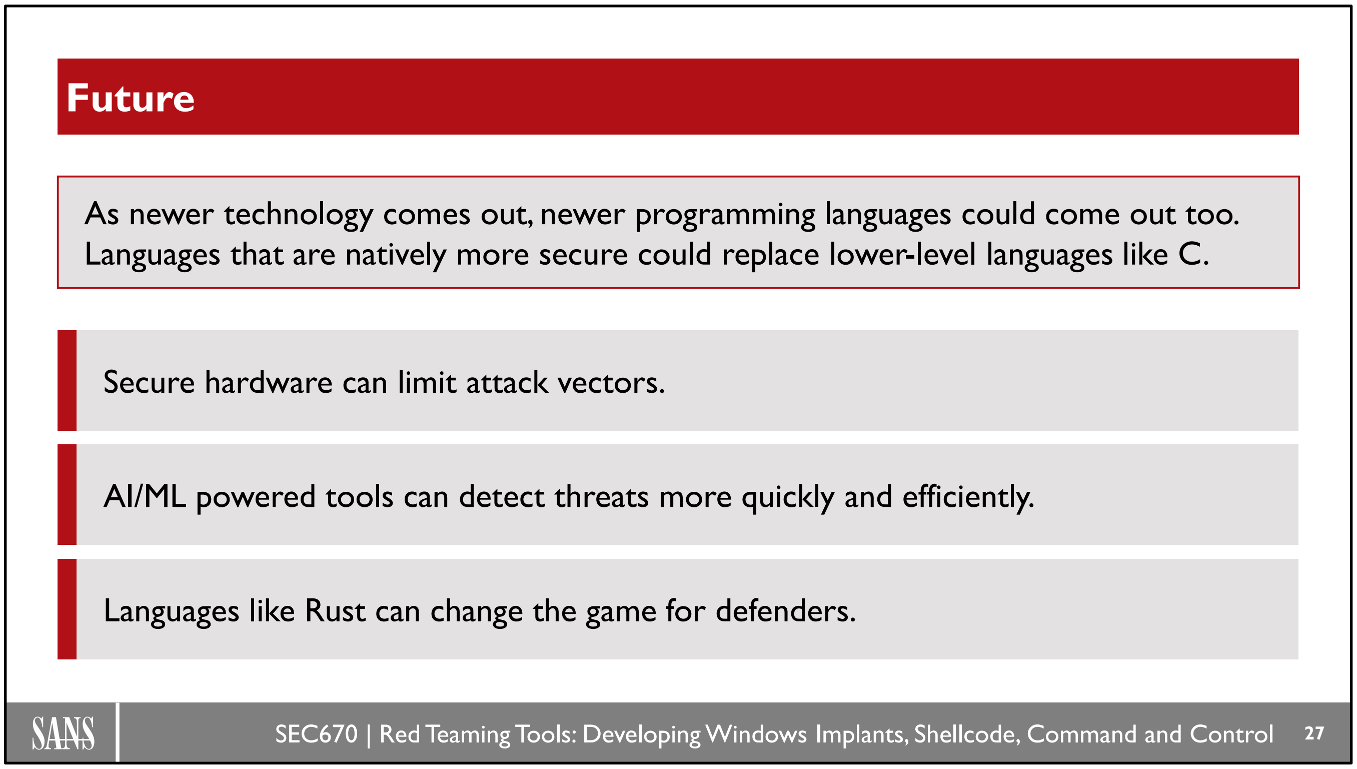
SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 26

Current State of the Art Tools
Open-source tools are great and sites like GitHub are full of them. There is nothing wrong with closed-source, commercial tools but when you can see the code like you can on GitHub, you can fork the project and modify it to fit your needs. Granted you will need to check the license the author put on it, but often the license is not that restrictive. Freeware tools are tools that are, well, free! One possible downside is that the code does not have to be made available to the public.

What about making a profit? There is nothing wrong with companies trying to make a profit, but commercial tools can sometimes come with a hefty price tag. The saying "You get what you pay for" can sometimes apply here and the more popular the company the more expensive the software. Amazing products can come from small to medium sized companies and one such company is Huntress Labs. Huntress Labs offers an amazing product that can complement your current security suite. If your organization is looking for something awesome, then check them out online and talk to Kyle Hanslovan.

PE-sieve is a great defensive tool aimed at finding malware. The author goes by the handle hasherezade and has published the tool on GitHub. The wiki describes the tool very well along with what the tool's capabilities are. For example, PE-sieve can easily dump implants for you that have been injected into process memory. It can also detect the various injection techniques that we will be learning about this week. Pro tip: If you want to see how good your tool is at evading detection, get it to bypass PE-sieve.

The Sysinternals Suite of tools are great too. For instance, ProcMon, with the right filters in place, can help identify if your software is looking for DLLs that are not on the system. Something like this can lead to DLL hijacking where an attacker drops a DLL that is being sought out by the target program. Sysmon is another powerful tool that operates as a system service that can monitor process creation, network connections, and several other items of interest.

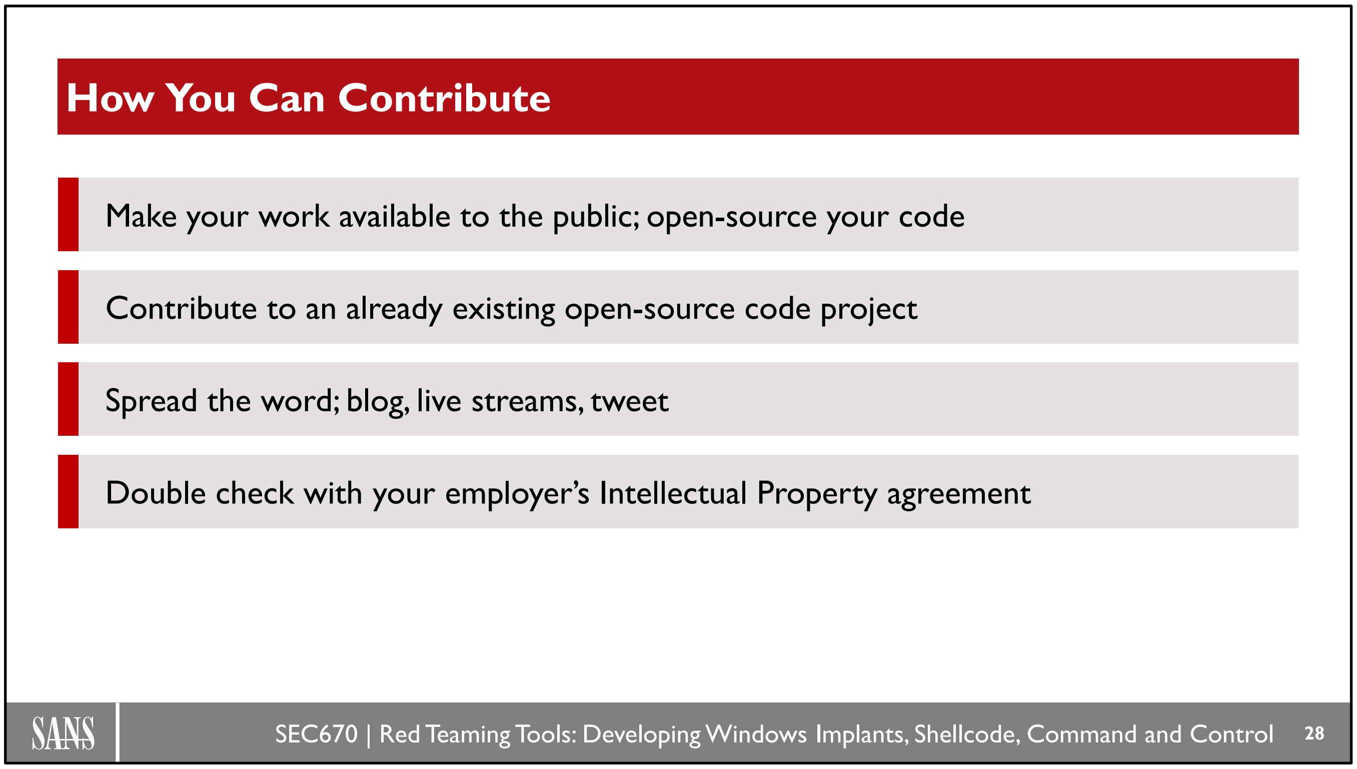


Future

As newer technology comes out, newer programming languages could come out too. Languages that are natively more secure could replace lower-level languages like C.

- Secure hardware can limit attack vectors.
- AI/ML powered tools can detect threats more quickly and efficiently.
- Languages like Rust can change the game for defenders.

Future
Just like the future of developing offensive tools, the future of defensive tools and programming depends heavily on creativity and attention to detail. Some languages like Rust could really change the game for defensive tools and secure programming. Windows is rewriting some of the kernel in Rust and thus this would increase the level of difficulty for exploitation. Check out Rust if you ever get some free time to see the differences for yourself. Fully automated systems are increasingly becoming more efficient and accurate with the advancements of modern-day computing and technology. That technology can be folded into defensive tools to make them more robust and take some of the fatigue away from the analyst.

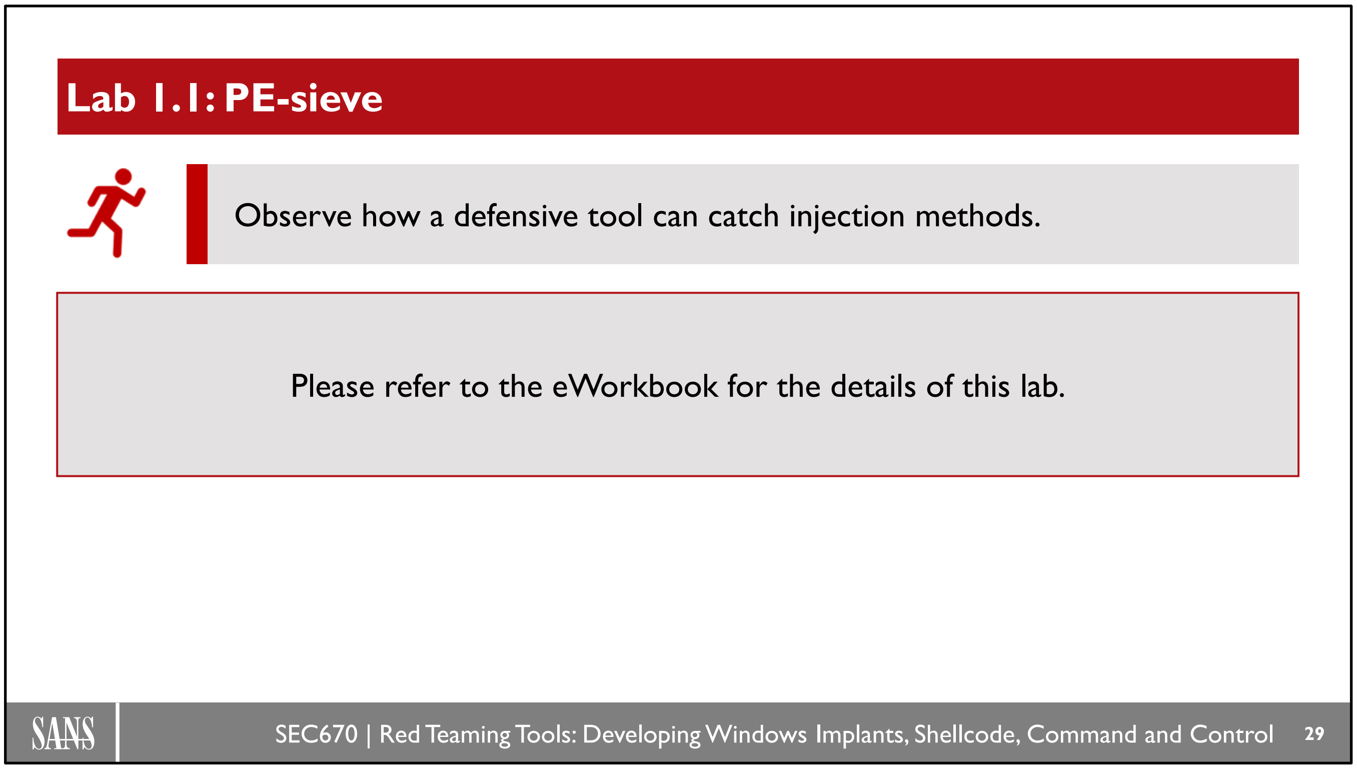


How You Can Contribute

- Make your work available to the public; open-source your code
- Contribute to an already existing open-source code project
- Spread the word; blog, live streams, tweet
- Double check with your employer's Intellectual Property agreement

How You Can Contribute
There are many ways you can contribute, and GitHub is just one of the many ways for you to share your code with the world. Once you make a few projects you can start to spread the word about it and share the link to your repos. Many developers will write a short blog about their tool, how to use it, and what inspired them to develop it in the first place. If you do not want to release your own projects, then perhaps you could contribute to some existing ones. Many of the major projects out there love it when people contribute and submit a pull request. Look around and see if there are any projects that interest you.

As always, be sure to check with your employer before doing so.



Lab 1.1: PE-sieve

Observe how a defensive tool can catch injection methods.

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

Lab 1.1: PE-sieve
PE-sieve, according to hasherezade's GitHub repo, "is a tool that helps detect malware running on the system, as well as to collect the potentially malicious material for further analysis." The tool is designed to scan a single process, but it does a great job at detecting various items like injected PEs and hooks. It also has the ability to dump an implant should one be discovered injected into a process.

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

What's the Point?

What's the point?

Lab 1.2: ProcMon

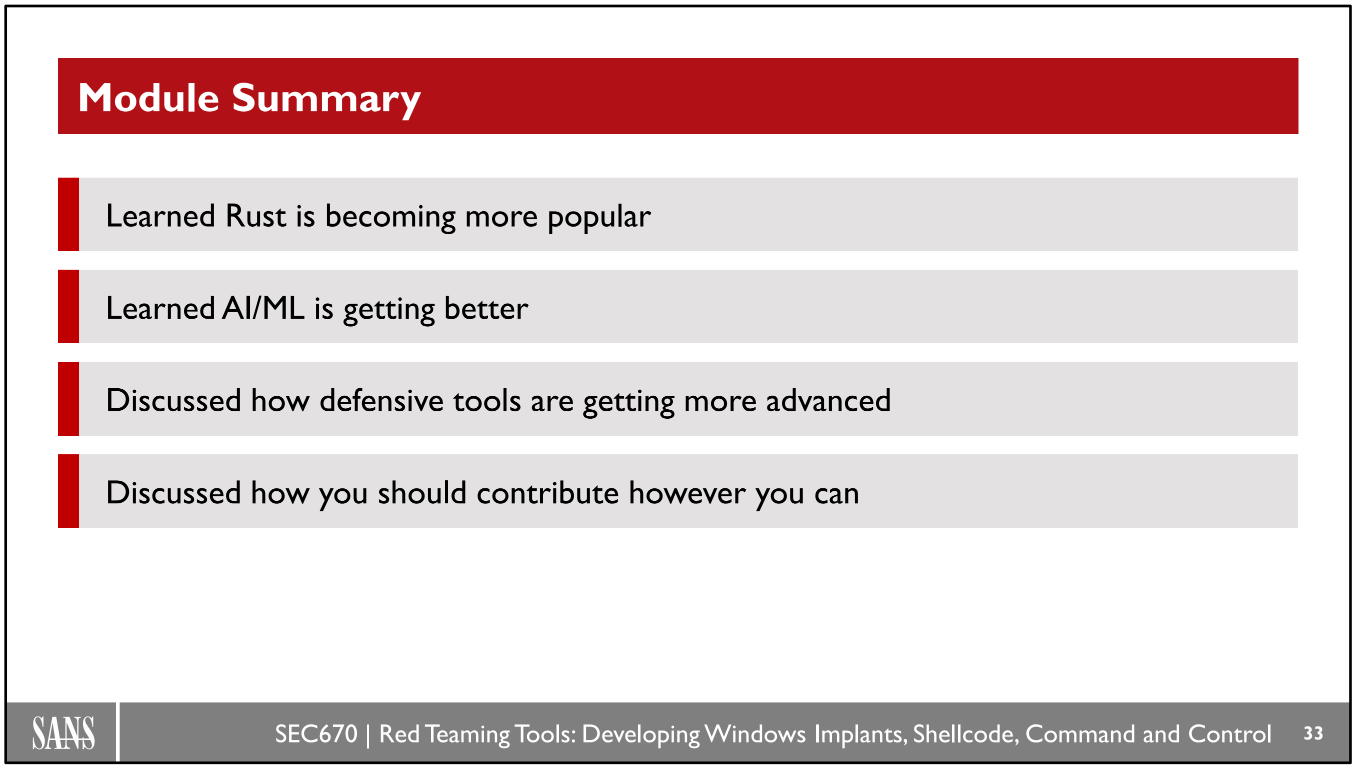
Observe how ProcMon can be used to spot flaws with a program.

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

What's the Point?

What's the point?

What's the Point?
The point of that lab was to explore how defenders can use Process Monitor to see what their application is doing. Maybe they spotted something that needs to be fixed and they are able to do so before it ever is made available to the user.



Module Summary

- Learned Rust is becoming more popular
- Learned AI/ML is getting better
- Discussed how defensive tools are getting more advanced
- Discussed how you should contribute however you can

Course Roadmap

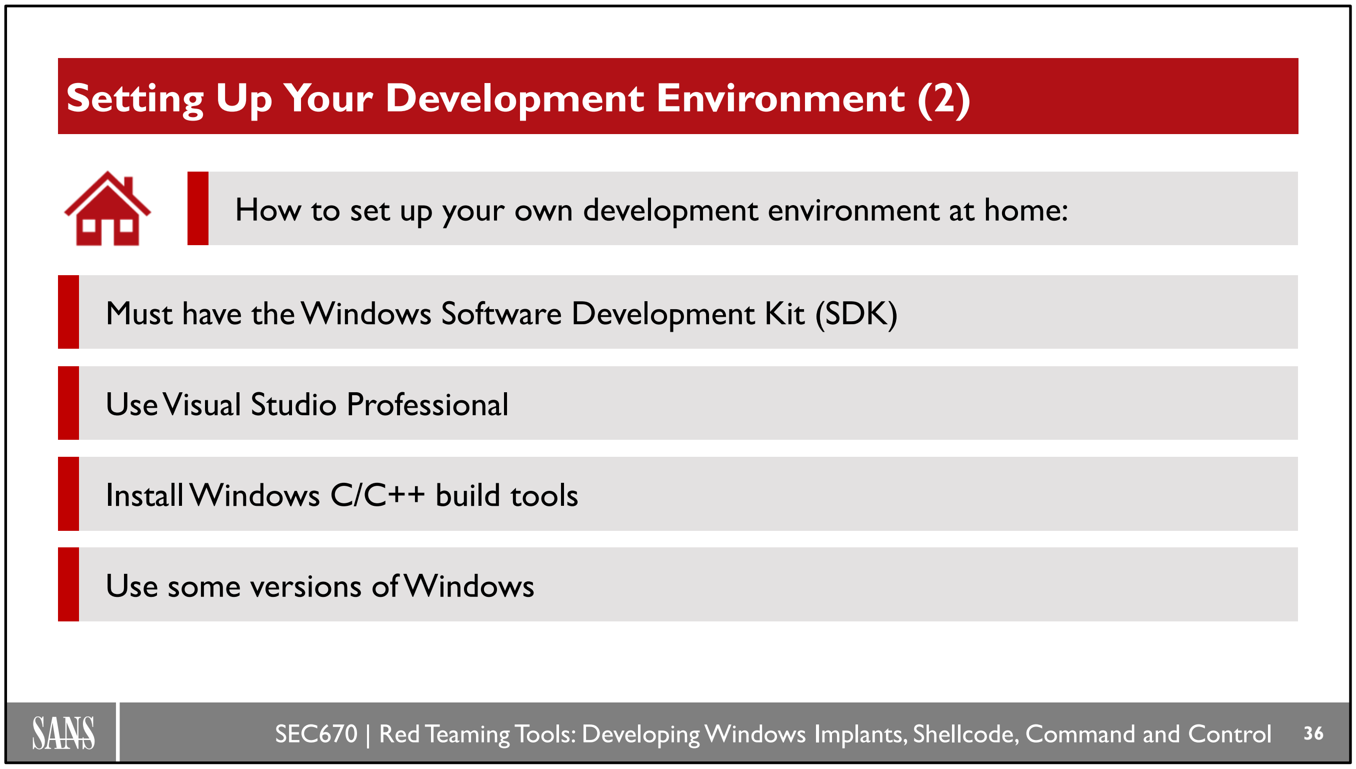
- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Setting Up Your Development Environment (1)

The virtual machines made available to you are already configured for this course. If you have your own development environment and would like to use it, that is fine.

Please do not install any updates unless directed otherwise.

Copy over everything from the media downloads to your host machine.



Setting Up Your Development Environment (2)

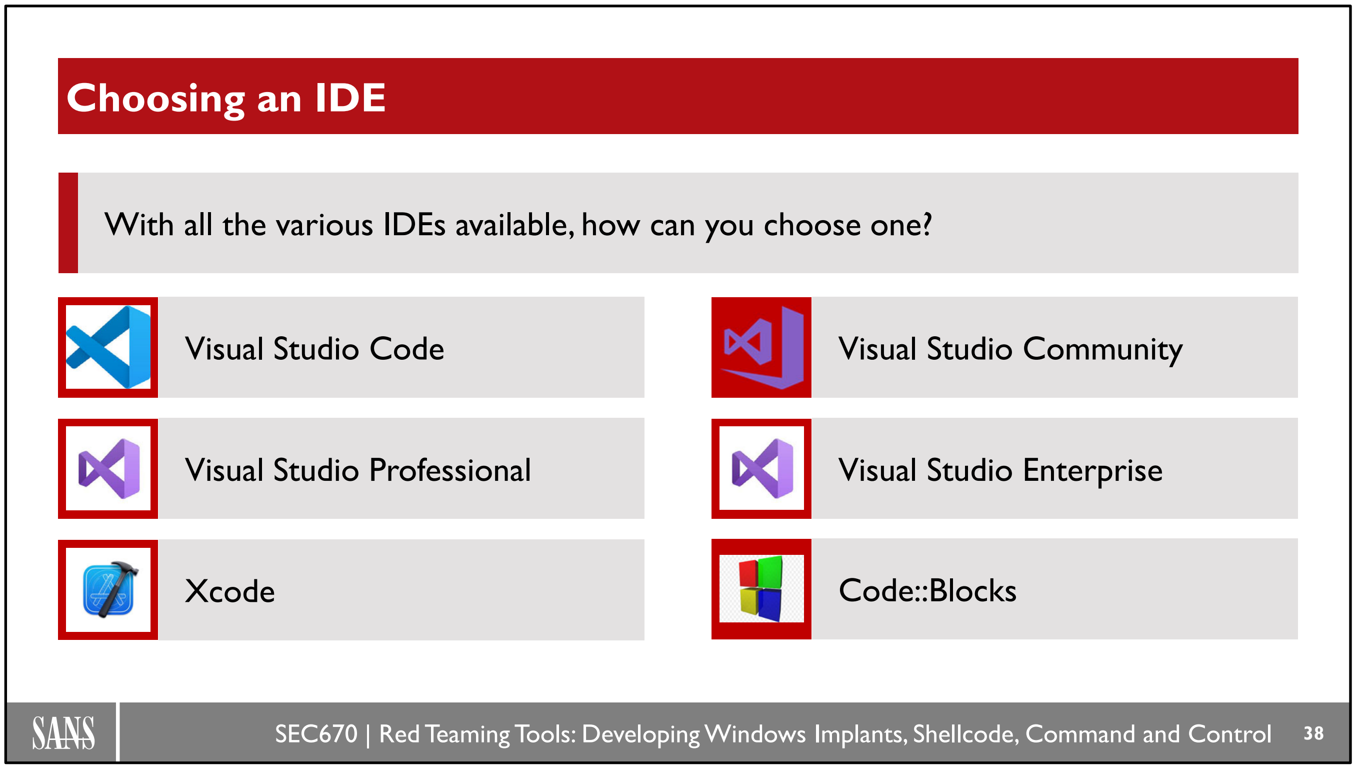
How to set up your own development environment at home:

- Must have the Windows Software Development Kit (SDK)
- Use Visual Studio Professional
- Install Windows C/C++ build tools
- Use some versions of Windows

Choosing a Language

Does the language really matter when doing implant development or creating host survey tools for a target?

- An implant in pure C
- An implant in C++
- Will one be better than the other?



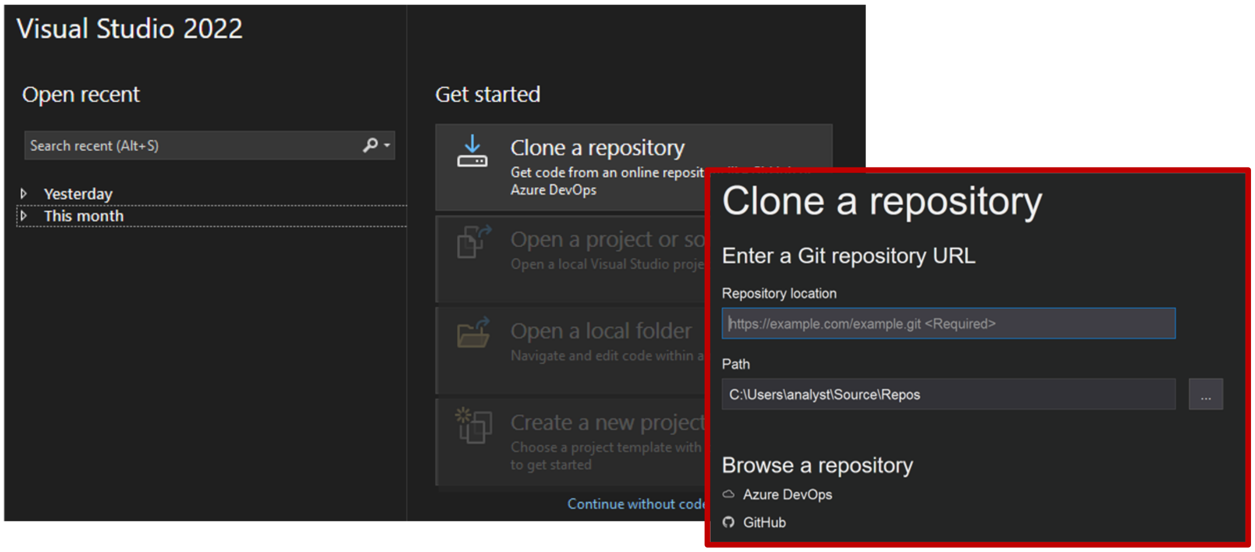
Choosing an IDE

With all the various IDEs available, how can you choose one?

- Visual Studio Code
- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise
- Xcode
- Code::Blocks

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 38

Cloning a Repository


Visual Studio 2022

Open recent

Search recent (Alt+S)

Yesterday

This month

Get started

Clone a repository

Get code from an online reposit

Azure DevOps

Open a project or so

Open a local Visual Studio proje

Open a local folder

Navigate and edit code within a

Create a new project

Choose a project template with

to get started

Continue without code

Clone a repository

Enter a Git repository URL

Repository location

https://example.com/example.git <Required>

Path

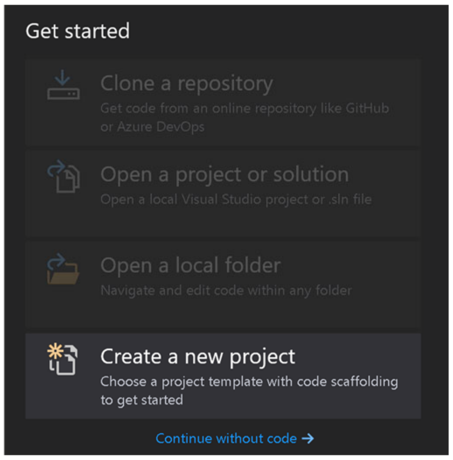
C:\Users\analyst\Source\Repos

Browse a repository

Azure DevOps

GitHub

Creating a New Project


Get started

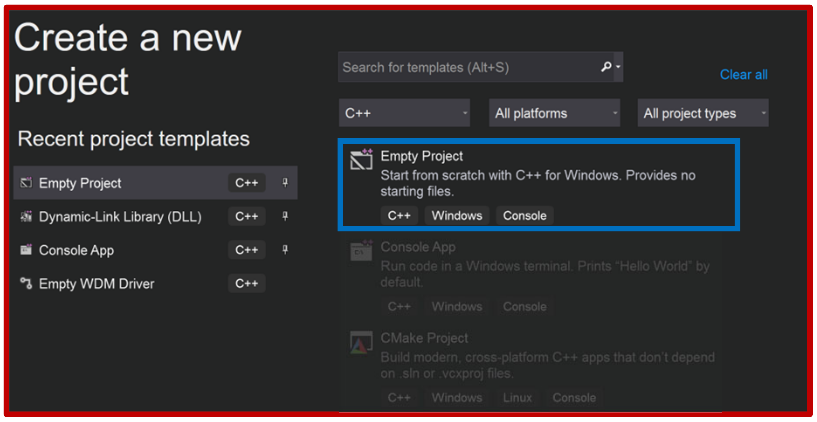
Clone a repository
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →


Create a new project

Recent project templates

Empty Project	C++
Dynamic-Link Library (DLL)	C++
Console App	C++
Empty WDM Driver	C++

Search for templates (Alt+S)

Clear all

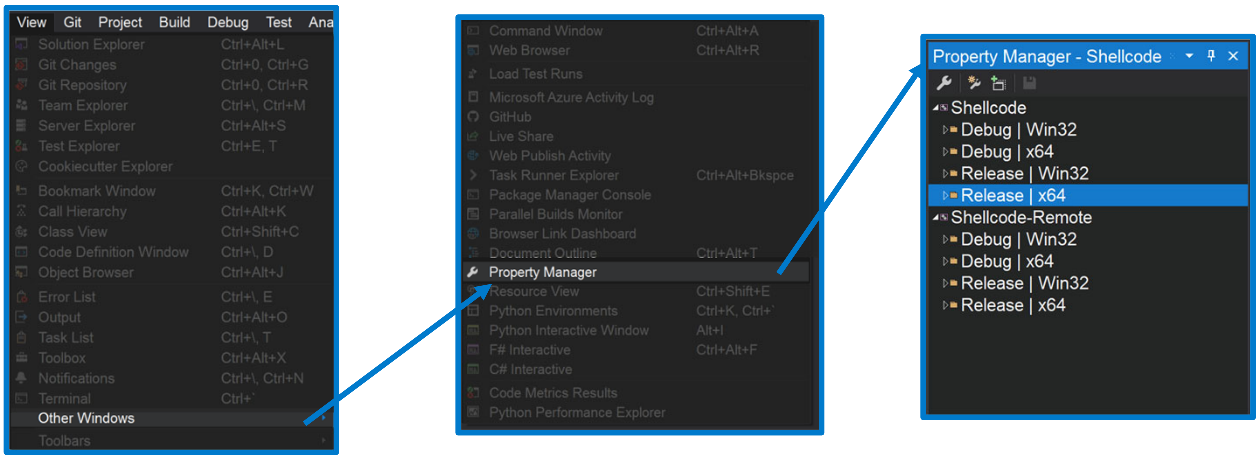
C++ | All platforms | All project types

Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ Windows Console

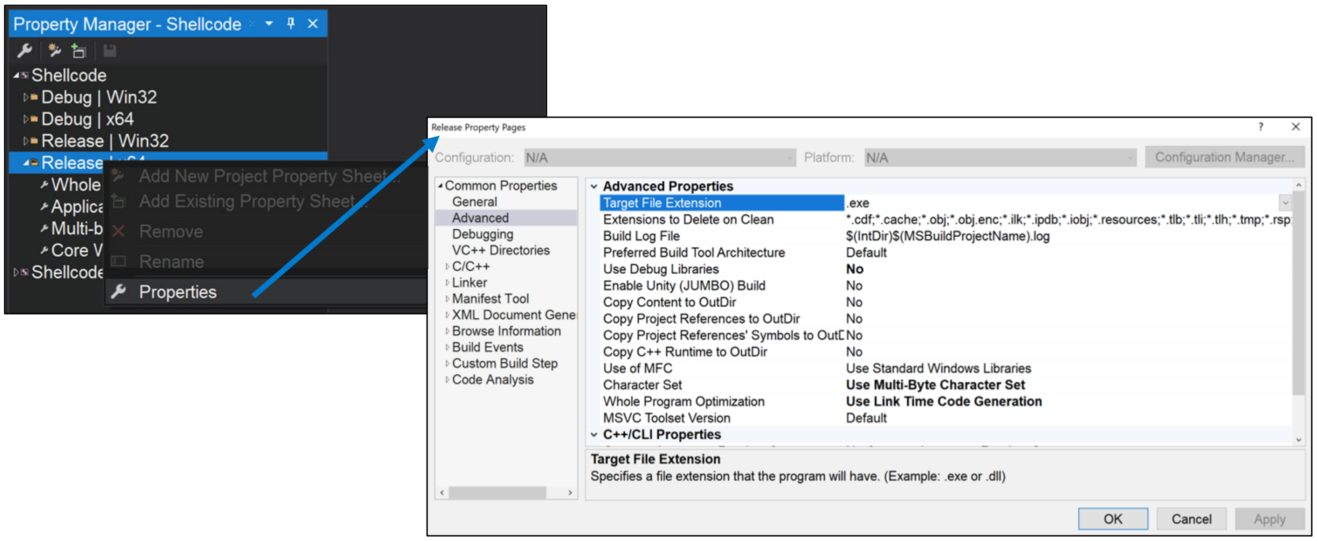
Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ Windows Console

CMake Project
Build modern, cross-platform C++ apps that don't depend on .sln or .vcxproj files.
C++ Windows Linux Console

Visual Studio Property Sheets (1)


View
Git
Project
Build
Debug
Test
Ana
Solution Explorer Ctrl+Alt+L
Git Changes Ctrl+0, Ctrl+G
Git Repository Ctrl+0, Ctrl+R
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
Test Explorer Ctrl+E, T
Cookiecutter Explorer
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
Output Ctrl+Alt+O
Task List Ctrl+\, T
Toolbox Ctrl+Alt+X
Notifications Ctrl+\, Ctrl+N
Terminal Ctrl+`
Other Windows
Toolbars
Command Window Ctrl+Alt+A
Web Browser Ctrl+Alt+R
Load Test Runs
Microsoft Azure Activity Log
GitHub
Live Share
Web Publish Activity
Task Runner Explorer Ctrl+Alt+Bkspce
Package Manager Console
Parallel Builds Monitor
Browser Link Dashboard
Document Outline Ctrl+Alt+T
Property Manager
Resource View Ctrl+Shift+E
Python Environments Ctrl+K, Ctrl+`
Python Interactive Window Alt+I
F# Interactive Ctrl+Alt+F
C# Interactive
Code Metrics Results
Python Performance Explorer
Property Manager - Shellcode
Shellcode
Debug | Win32
Debug | x64
Release | Win32
Release | x64
Shellcode-Remote
Debug | Win32
Debug | x64
Release | Win32
Release | x64

Visual Studio Property Sheets (2)



Configuration: N/A Platform: N/A

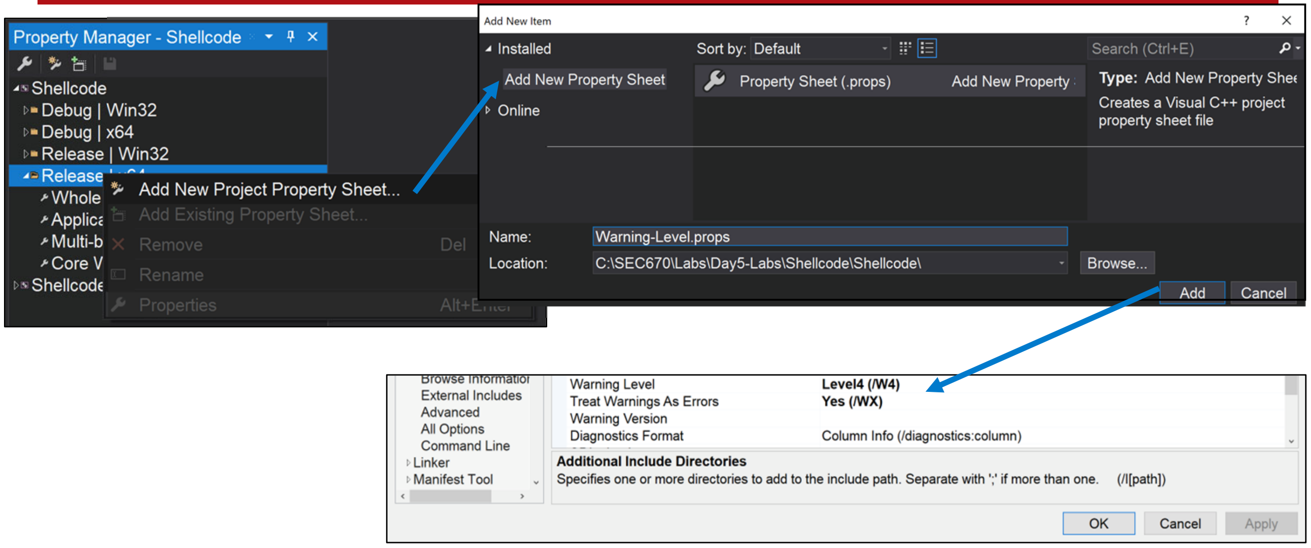
Advanced Properties

Target File Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*.iobj;*.resources;*.tlb;*.tli;*.tlh;*.tmp;*.rsp
Build Log File	$(IntDir)$(MSBuildProjectName).log
Preferred Build Tool Architecture	Default
Use Debug Libraries	**No**
Enable Unity (JUMBO) Build	No
Copy Content to OutDir	No
Copy Project References to OutDir	No
Copy Project References' Symbols to OutD	No
Copy C++ Runtime to OutDir	No
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Multi-Byte Character Set**
Whole Program Optimization	**Use Link Time Code Generation**
MSVC Toolset Version	Default

C++/CLI Properties

Target File Extension
Specifies a file extension that the program will have. (Example: .exe or .dll)

Visual Studio Property Sheets (3)



Name: Warning-Level.props
Location: C:\SEC670\Labs\Day5-Labs\Shellcode\Shellcode\

Warning Level	**Level4 (/W4)**
Treat Warnings As Errors	**Yes (/WX)**
Warning Version	
Diagnostics Format	Column Info (/diagnostics:column)

Additional Include Directories
Specifies one or more directories to add to the include path. Separate with ';' if more than one. (/I[path])

Your Testing Environment

Your testing environment is just for that, testing. It should be robust and capable of scalability to suit your needs.

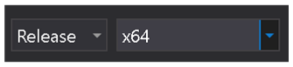
- It might be a good idea to test your tool with various version of Windows.
- Don't perform only a single test case and assume all is well—perform hundreds.
- Validate your tool before you put it live on a target.

Build Targets

Even if you develop on the most recent version of Windows, it does not mean you cannot build a version that targets an older version like Windows XP.

- Builds for a single target
- Builds for maximum effect and reach; multiple versions of Windows
- Your tool does not have to execute on every version of Windows

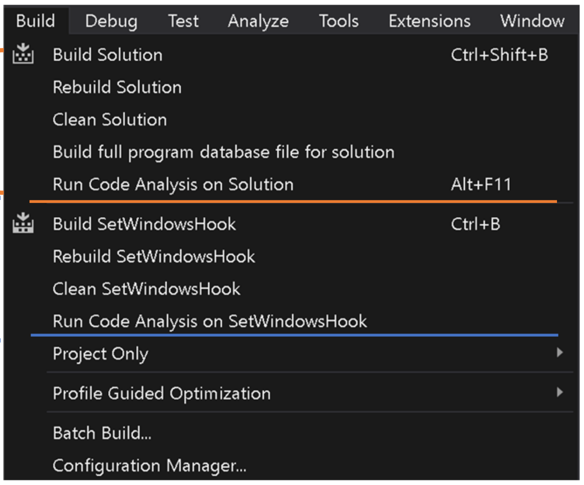
Build Visual Studio Solutions/Projects (1)



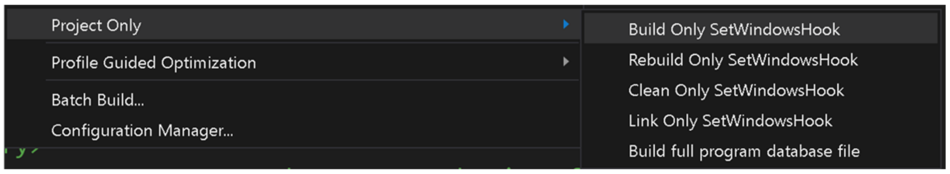
Choose configuration and architecture: Debug/Release, x86/x64

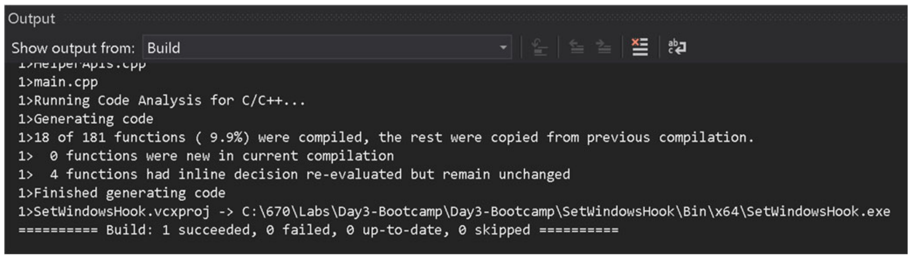
Build the entire solution?

Build only the project?


Build
Debug
Test
Analyze
Tools
Extensions
Window
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Build full program database file for solution
Run Code Analysis on Solution Alt+F11
Build SetWindowsHook Ctrl+B
Rebuild SetWindowsHook
Clean SetWindowsHook
Run Code Analysis on SetWindowsHook
Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...

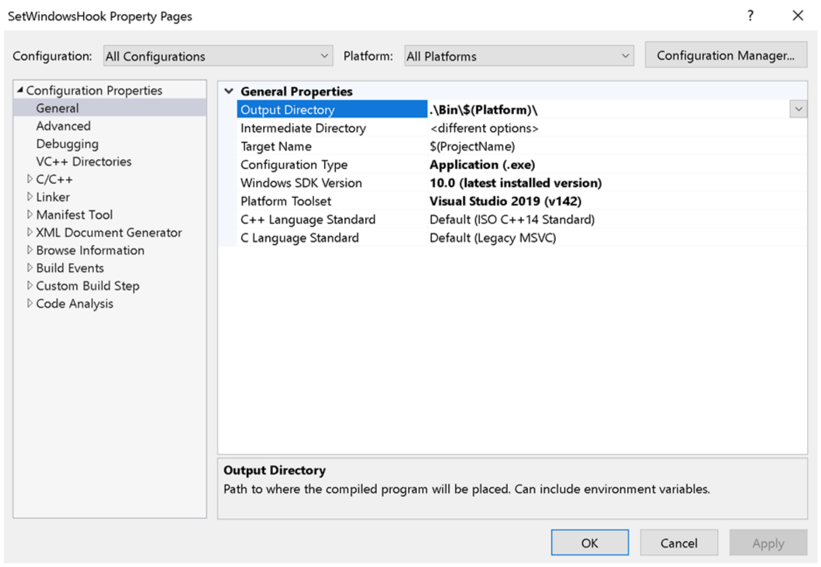
Build Visual Studio Solutions/Projects (2)


Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...
Build Only SetWindowsHook
Rebuild Only SetWindowsHook
Clean Only SetWindowsHook
Link Only SetWindowsHook
Build full program database file



```
1>HelperApis.cpp
1>main.cpp
1>Running Code Analysis for C/C++...
1>Generating code
1>18 of 181 functions ( 9.9%) were compiled, the rest were copied from previous compilation.
1>  0 functions were new in current compilation
1>  4 functions had inline decision re-evaluated but remain unchanged
1>Finished generating code
1>SetWindowsHook.vcxproj -> C:\670\Labs\Day3-Bootcamp\Day3-Bootcamp\SetWindowsHook\Bin\x64\SetWindowsHook.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Visual Studio Project Settings



General Properties

Output Directory	.\Bin\$(Platform)\
Intermediate Directory	<different options>
Target Name	$(ProjectName)
Configuration Type	**Application (.exe)**
Windows SDK Version	**10.0 (latest installed version)**
Platform Toolset	**Visual Studio 2019 (v142)**
C++ Language Standard	Default (ISO C++14 Standard)
C Language Standard	Default (Legacy MSVC)

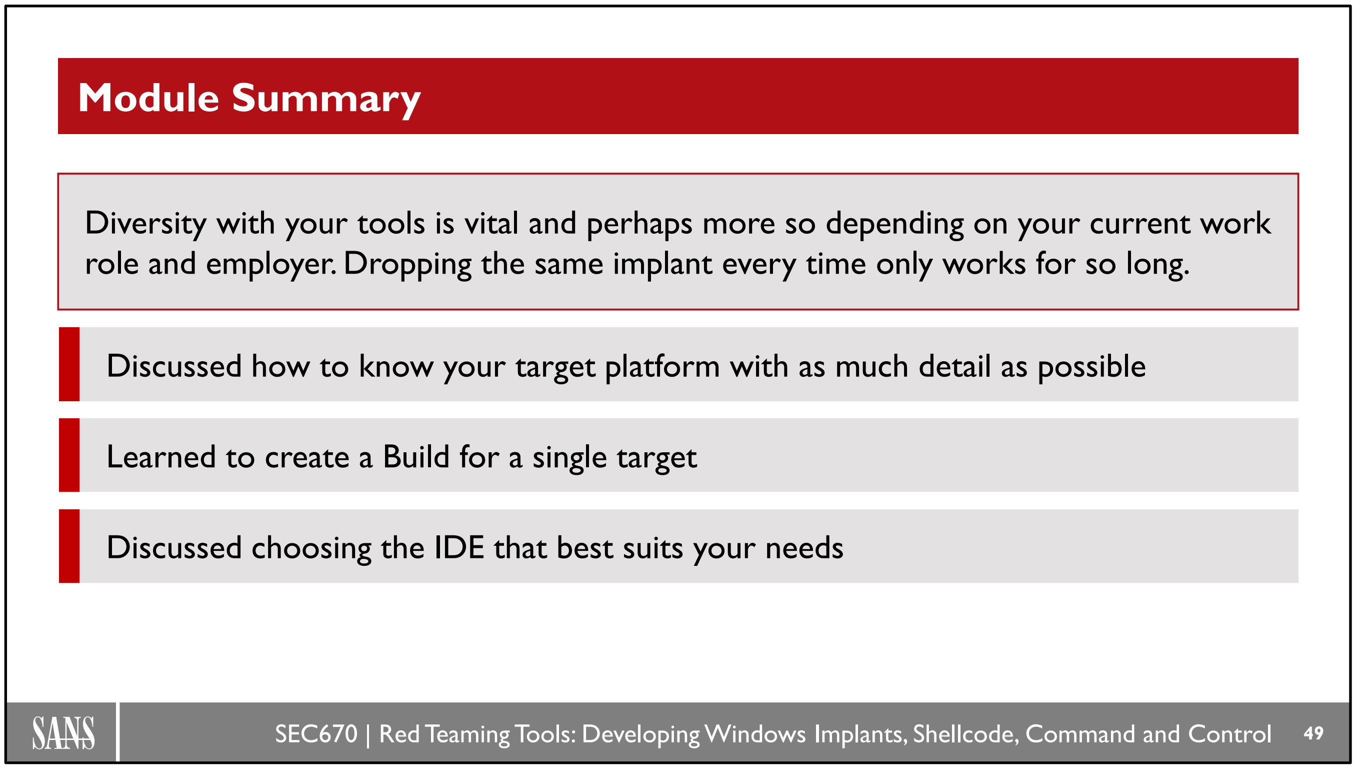
Output Directory
Path to where the compiled program will be placed. Can include environment variables.

Visual Studio Project Settings
We will not be going over every single setting in the Project Properties window, but it is important to know some basic items and options here. At the topmost portion of the window, you have Configuration and Platform. The Configuration choices are nothing new: Debug, Release, or All. The Platform can be one of the following options: Win32, x64, or All. In the General Properties, the Configuration Type can be selected: EXE, DLL, or LIB. If your system has more than one SDK available, you can select a specific version, like one for Windows XP.

There are other sections in the Property Pages window, such as C/C++ where code optimization can be tweaked, and Precompiled headers can be modified and selected. There are also a few other options here, such as Warning Level and Calling Convention. The Linker section is where you can specify any additional dependencies—think of libraries like kernel32.lib, etc. You can also choose any module definition files that might be of use, say if you are building a DLL. In addition, there are linker optimizations that can be tweaked as well as characteristics like Data Execution Prevention (DEP) and Dynamic Base (ASLR). This is where you can tweak the size of your binary, and in the world of implants, the smaller your implant is the better. 40MB is not a good size for an implant.

The last section covered here is Code Analysis, which is where you can enable Microsoft Code Analysis to run as you develop and build your program. Be sure this feature is enabled as it can save you from some future headaches. Code analysis does what it sounds like. It will perform a static analysis of your code and provide some detailed feedback should anything pop up that needs your attention. To directly quote MSDN documentation about the feature: “Common coding errors reported by the tool include buffer overruns, uninitialized memory, null pointer dereferences, and memory and resource leaks.” Code analysis improves when your source code uses source-code annotation language (SAL). SAL is something that is used heavily in this course and it even has its own module and lab later in this section. Look at the references below for more information about the code analysis for C/C++.

Reference:
https://learn.microsoft.com/en-us/cpp/code-quality/code-analysis-for-c-cpp-overview?view=msvc-170



Module Summary

Diversity with your tools is vital and perhaps more so depending on your current work role and employer. Dropping the same implant every time only works for so long.

- Discussed how to know your target platform with as much detail as possible
- Learned to create a Build for a single target
- Discussed choosing the IDE that best suits your needs

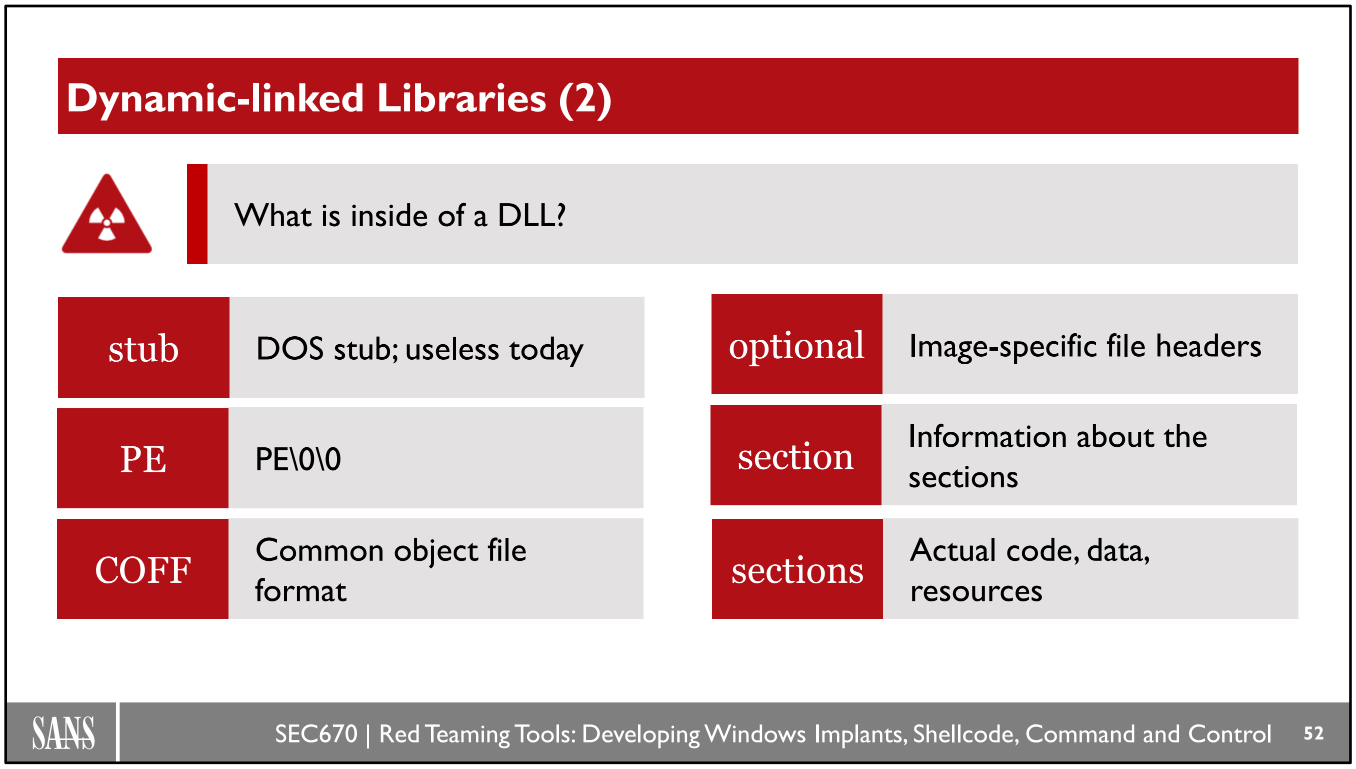
Course Roadmap

- **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

Dynamic-linked Libraries (1)

Dynamic-linked libraries (DLL) look just like a standard Windows executable file. The purpose is what separates the two. DLLs primarily supply a common functionality to applications that need it.

- PE32 / PE32+ format
- Designed to be shared
- Various extensions: .lib and .dll



Dynamic-linked Libraries (2)

What is inside of a DLL?

stub	DOS stub; useless today
PE	PE\0\0
COFF	Common object file format
optional	Image-specific file headers
section	Information about the sections
sections	Actual code, data, resources

Dynamic-linked Libraries (3)

How do you see what is inside of a DLL?

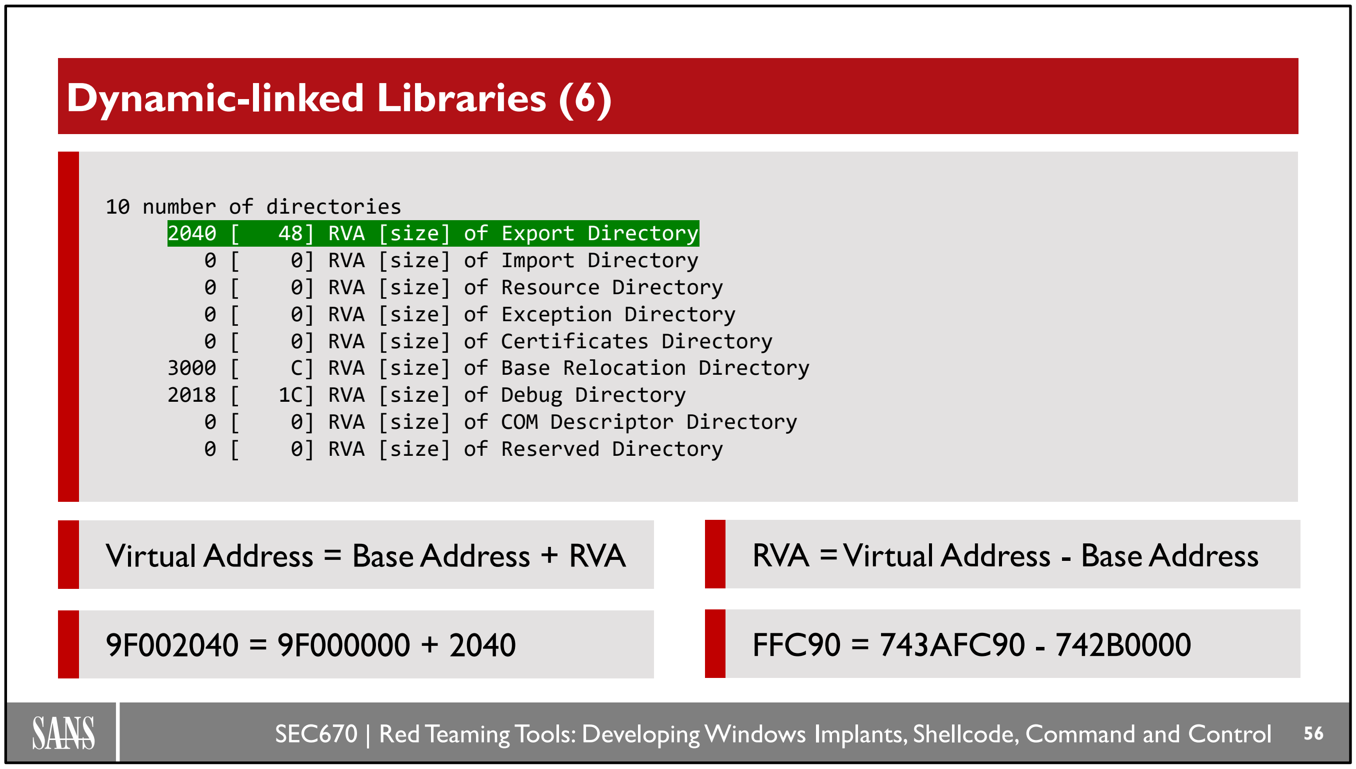
- Dumpbin utility
- PEview
- PE-bear
- CFF Explorer

Dynamic-linked Libraries (4)

```
// dumpbin /headers hello.dll
Dump of file hello.dll

PE signature found

File Type: DLL

FILE HEADER VALUES
             14C machine (x86)
               3 number of sections
        602A3E55 time date stamp Mon Feb 15 09:26:45 2021
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            2102 characteristics
                   Executable
                   32 bit word machine
                   DLL
```

Dynamic-linked Libraries (5)

```
OPTIONAL HEADER VALUES
             10B magic # (PE32)
           14.28 linker version
             200 size of code
             400 size of initialized data
               0 entry point
            1000 base of code
            2000 base of data
        10000000 image base (10000000 to 10003FFF)
            1000 section alignment
            4000 size of image
             400 size of headers
               0 checksum
               2 subsystem (Windows GUI)
             540 DLL characteristics (Dynamic base, NX compatible, No SEH)
          100000 size of stack reserve
            1000 size of stack commit
```



Dynamic-linked Libraries (6)

```
10 number of directories
     2040 [    48] RVA [size] of Export Directory
        0 [     0] RVA [size] of Import Directory
        0 [     0] RVA [size] of Resource Directory
        0 [     0] RVA [size] of Exception Directory
        0 [     0] RVA [size] of Certificates Directory
     3000 [     C] RVA [size] of Base Relocation Directory
     2018 [    1C] RVA [size] of Debug Directory
        0 [     0] RVA [size] of COM Descriptor Directory
        0 [     0] RVA [size] of Reserved Directory
```

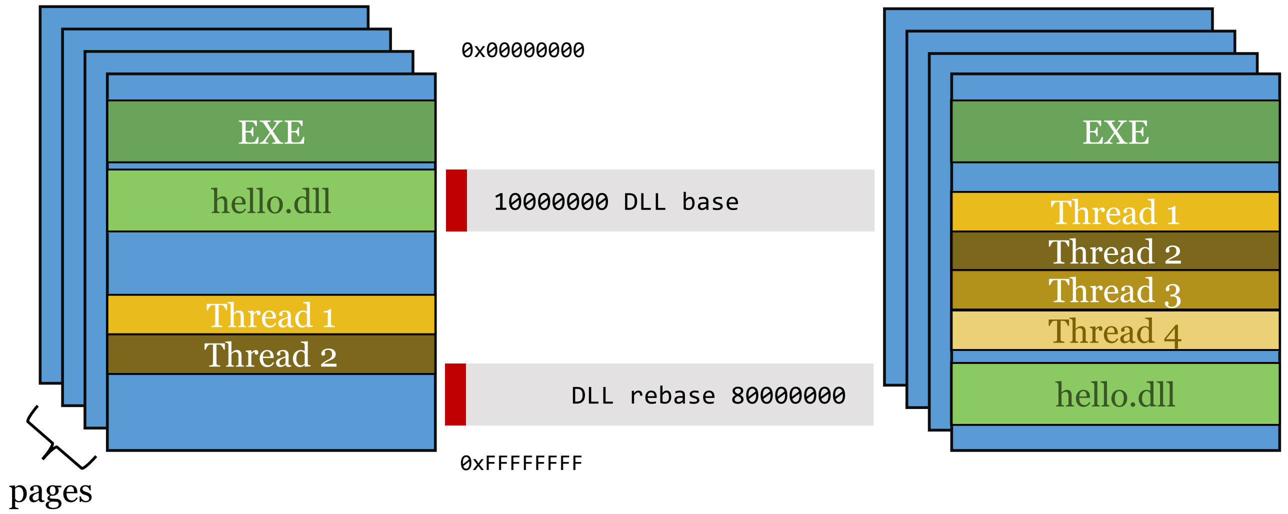
Virtual Address = Base Address + RVA

RVA = Virtual Address - Base Address

9F002040 = 9F000000 + 2040

FFC90 = 743AFC90 - 742B0000

Dynamic-linked Libraries (7)



Dynamic-linked Libraries (8)

```
SECTION HEADER #1
  .text name
        A virtual size
     1000 virtual address (10001000 to 10001009)
      200 size of raw data
      400 file pointer to raw data (00000400 to 000005FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
 60000020 flags
          Code
          Execute Read
```

.TEXT section

The executable code

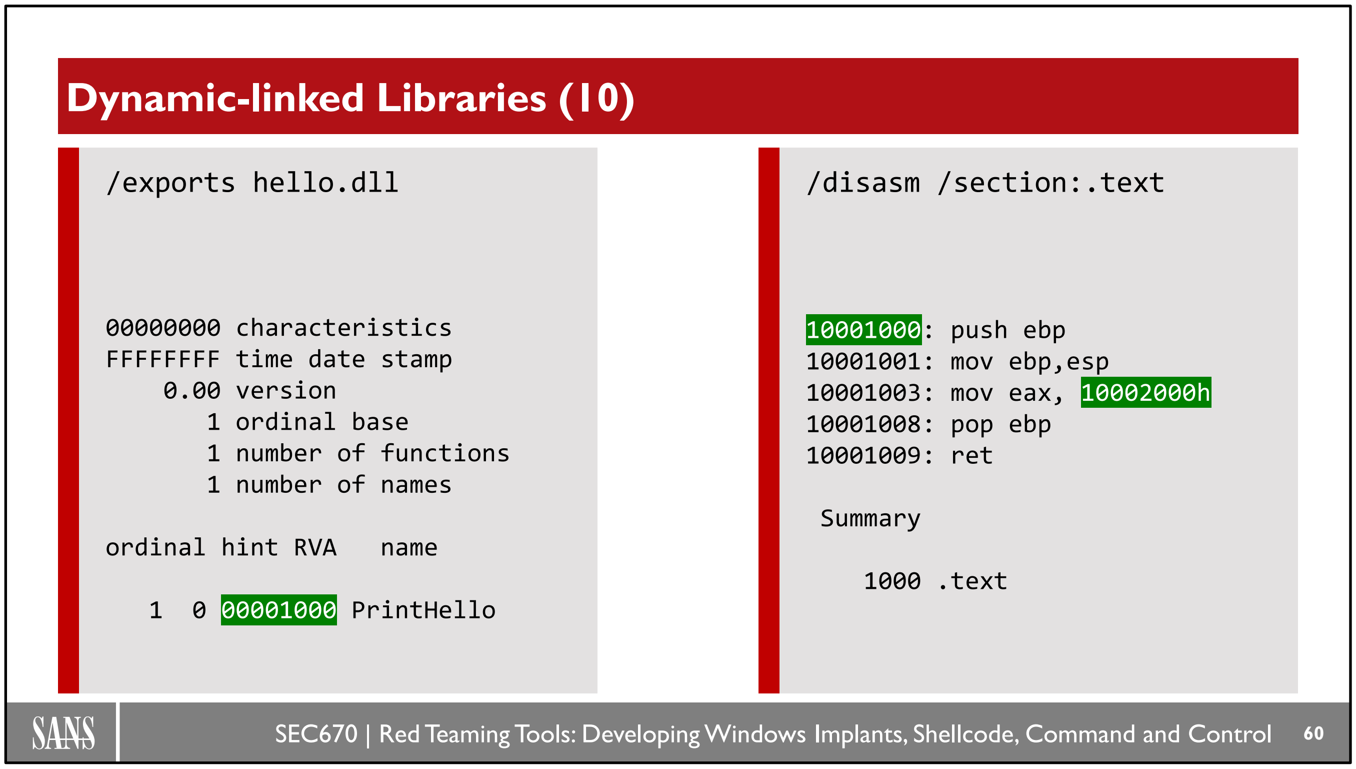
Dynamic-linked Libraries (9)

```
SECTION HEADER #2
 .rdata name
       D8 virtual size
     2000 virtual address (10002000 to 100020D7)
      200 size of raw data
      600 file pointer to raw data (00000600 to 000007FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
 40000040 flags
          Initialized Data
          Read Only
```

.RDATA section

Read only, initialized data

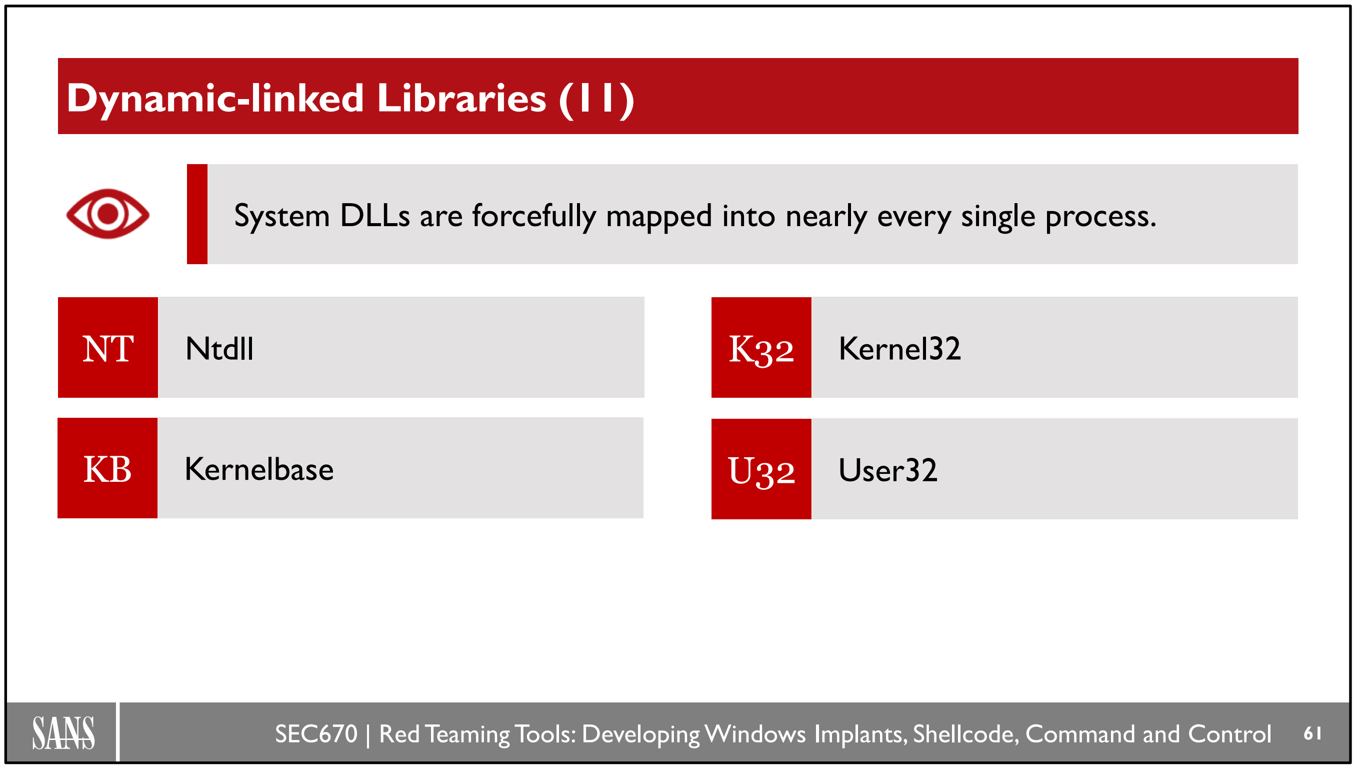
OPTIONAL_HEADER Directories
2040 [48] RVA [size] of Export Directory ← within .rdata
2018 [1C] RVA [size] of Debug Directory ← within .rdata



Dynamic-linked Libraries (10)

```
/exports hello.dll



00000000 characteristics
FFFFFFFF time date stamp
    0.00 version
       1 ordinal base
       1 number of functions
       1 number of names

ordinal hint RVA    name

   1   0 00001000 PrintHello
```

```
/disasm /section:.text



10001000: push ebp
10001001: mov ebp,esp
10001003: mov eax, 10002000h
10001008: pop ebp
10001009: ret

 Summary

    1000 .text
```



Dynamic-linked Libraries (11)

System DLLs are forcefully mapped into nearly every single process.

NT	Ntdll	K32	Kernel32
KB	Kernelbase	U32	User32

DLLs: Linking (1)

A DLL does nothing for you unless you link your program to it.

Implicit linking	Explicit linking
The OS is going to load the DLL right when that executable uses it. The exported functions can be called as if the functions were linked statically and resided within the executable itself.	The OS is going to load the DLL on demand right at runtime. The desired DLL must be **explicitly** *loaded* and *unloaded* by the client executable. Exported functions are called via pointers.

DLLs: Linking (2)

How to implicitly link a DLL.

- Must have header file; .h files
- Must have the actual DLL file
- Must have import library to link into the EXE
- Project Settings in Visual Studio

```
#pragma comment(lib, "hello")

//function from hello.lib
//PrintHello();

puts(PrintHello());
```

DLLs: Implicit Linking

```
// implicit_callhello.cpp
#include <stdio.h>

EXTERN_C PCHAR __cdecl PrintHello();

INT main(){
  puts(PrintHello());
}
```

Must be linked against the LIB file

```
link implicit_callhello.obj hello.lib
```

```
dumpbin /all hello.lib
```

```
5 public symbols
  [..snip..]
  62E _PrintHello
Archive member name at 62E:
  Type: code
  Symbol name: _PrintHello
  Name: PrintHello
```

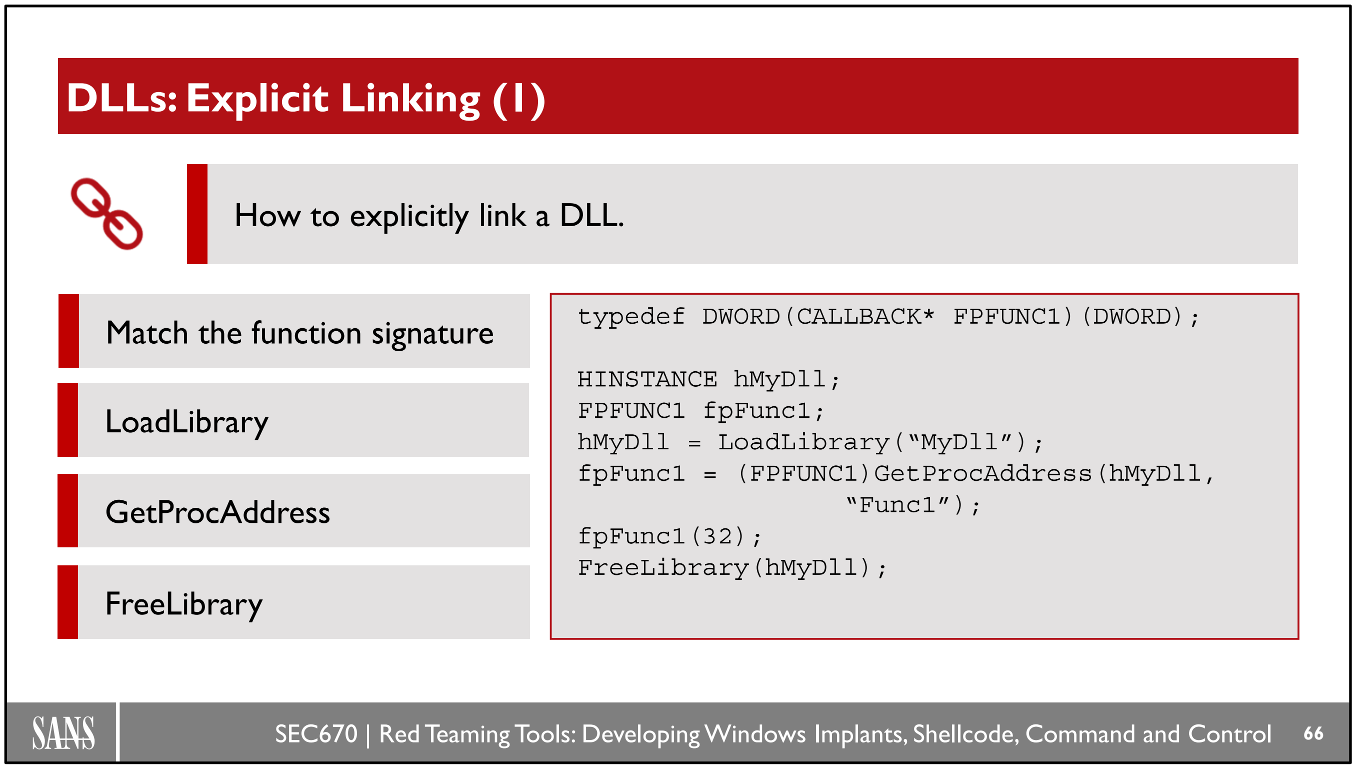
```
Image has the following dependencies:

  hello.dll
  KERNEL32.dll
```

dumpbin /exports hello.lib

```
File Type: LIBRARY

   Exports

    ordinal  name

         _PrintHello
```

dumpbin /all hello.lib

```
    5 public symbols

   18A __IMPORT_DESCRIPTOR_hello
   3AC __NULL_IMPORT_DESCRIPTOR
   4E0  hello_NULL_THUNK_DATA
   62E _PrintHello
   62E __imp__PrintHello

Archive member name at 62E: hello.dll/

 DLL name   : hello.dll
 Symbol name : _PrintHello
 Type      : code
 Name type  : no prefix
 Hint      : 0
 Name      : PrintHello
```



DLLs: Explicit Linking (1)

How to explicitly link a DLL.

- Match the function signature
- LoadLibrary
- GetProcAddress
- FreeLibrary

```
typedef DWORD(CALLBACK* FPFUNC1)(DWORD);

HINSTANCE hMyDll;
FPFUNC1 fpFunc1;
hMyDll = LoadLibrary("MyDll");
fpFunc1 = (FPFUNC1)GetProcAddress(hMyDll,
                  "Func1");
fpFunc1(32);
FreeLibrary(hMyDll);
```

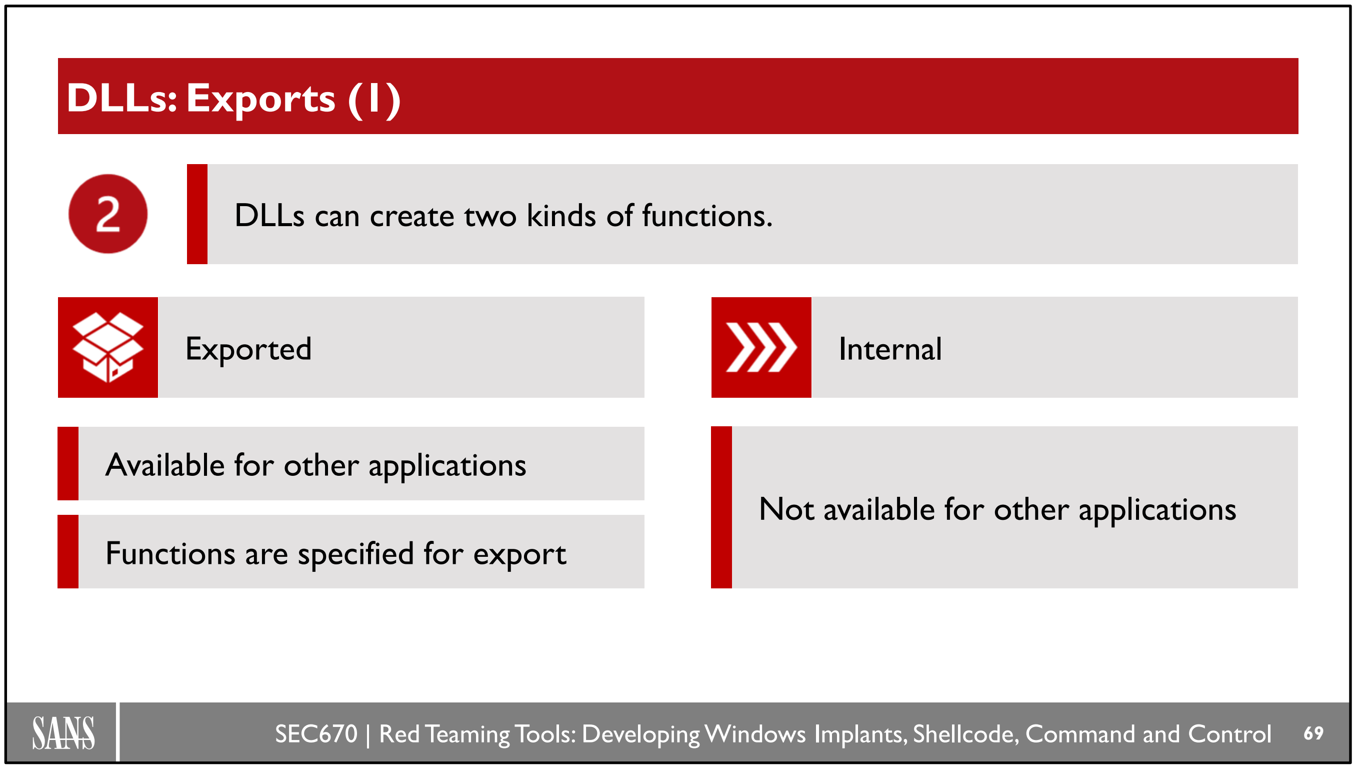
DLLs: Explicit Linking (2)

```
// callhello.cpp
#include <stdio.h>
#include <Windows.h>

INT main(){
  HMODULE helloDll = LoadLibraryExW(L"hello.dll", nullptr, 0);

  using t_PrintHello = PCSTR (__cdecl*)();
  t_PrintHello PrintHello = reinterpret_cast<t_PrintHello>(GetProcAddress(
                                                                 helloDll,
                                                                 "PrintHello"));

  puts(PrintHello());
  FreeLibrary(helloDll);

  return ERROR_SUCCESS;
}
```

DLLs: Explicit Linking (3)

```
dumpbin /dependents callhello.exe



Dump of file callhello.exe

File Type: EXECUTABLE IMAGE

 Image has the following dependencies:
    KERNEL32.dll

 Summary
    2000 .data
    6000 .rdata
    1000 .reloc
    D000 .text
```

```
dumpbin /imports callhello.exe



Section contains the following imports:

KERNEL32.dll
 40E000 Import Address Table
 4133EC Import Name Table
      0 time date stamp
      0 Index of first forwarder

    1AB FreeLibrary
    2AE GetProcAddress
    3C3 LoadLibraryExW
    [..snip..]
```



DLLs: Exports (1)

2 DLLs can create two kinds of functions.

Exported	Internal
Available for other applications	Not available for other applications
Functions are specified for export	

DLLs: Exports (2)

How to specify functions to be exported

__declspec()

__declspec(dllexport)

#define DLLEXPORT

Module definition file, .def

Contains a list of exported functions

Requires a certain syntax

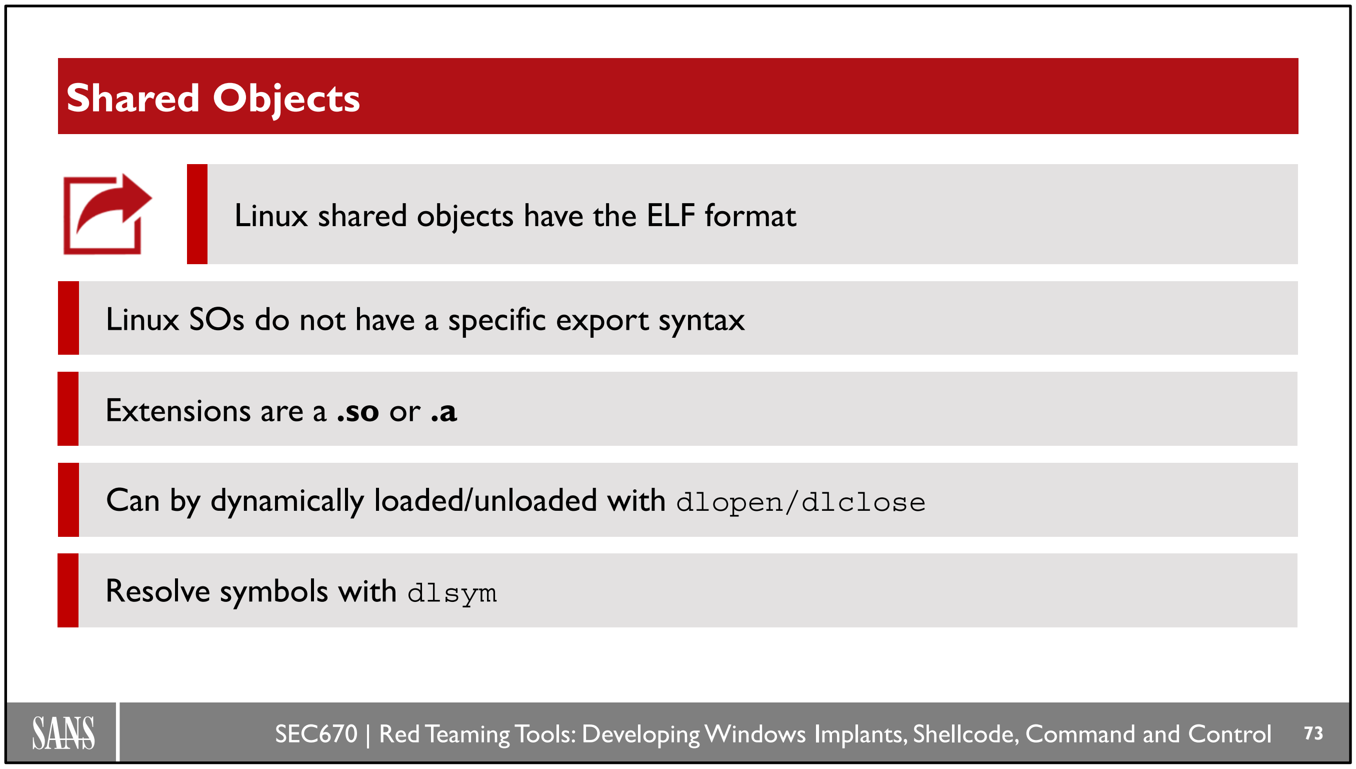
DLLs: Exports (3)

Example of a .def file

```
LIBRARY  Evil32
EXPORTS  ExecShellcode                @1       NONAME
         GetComputerName              @3       PRIVATE
         GetNetAdapter                @5
         GetSysInfo=another_dll.GetSystemInformation
VERSION 2.1
```

DLLs: Injection

- DLL injection: Forcing a process to load a specific DLL
- The code of the DLL will execute with the context of the target process
- Unlimited access to everything in the target process
- Injection does not always mean malicious activity; it is neither good nor bad



Shared Objects

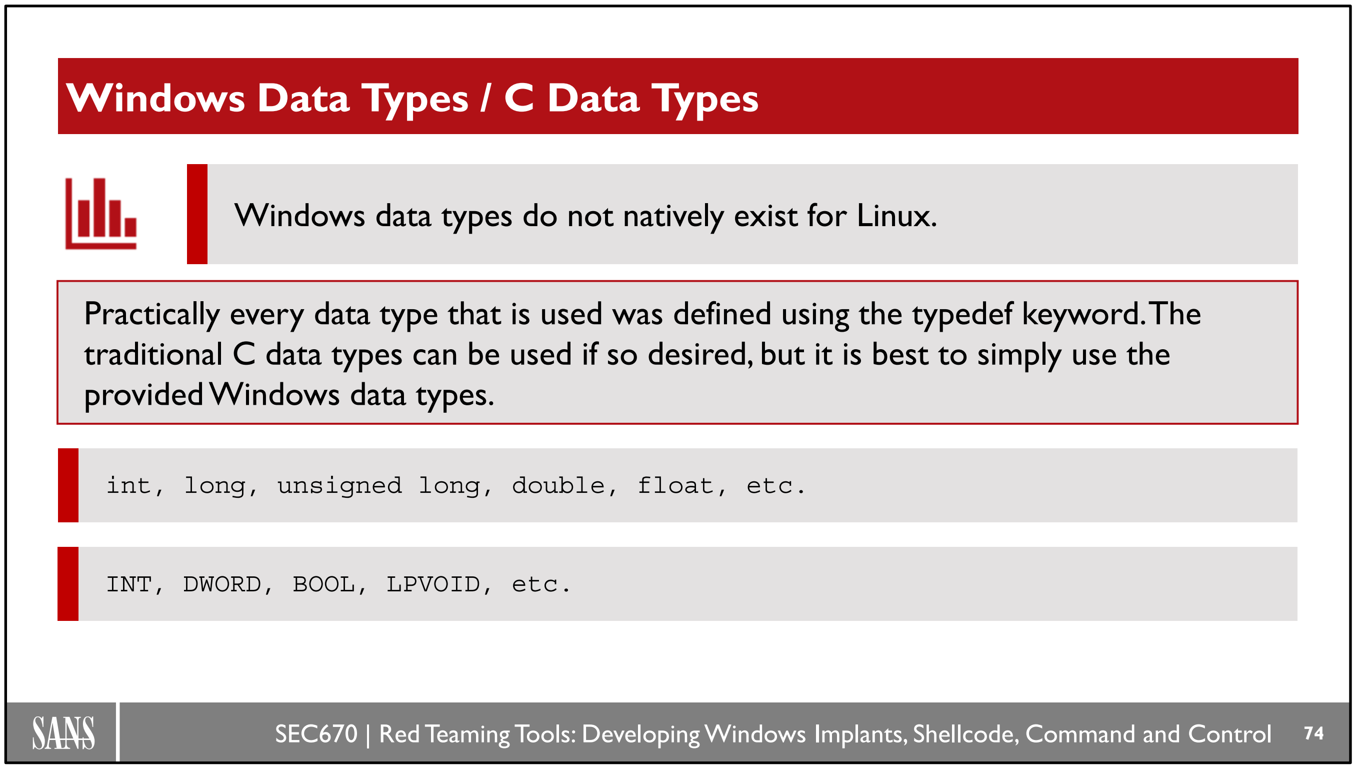
Linux shared objects have the ELF format

Linux SOs do not have a specific export syntax

Extensions are a **.so** or **.a**

Can by dynamically loaded/unloaded with `dlopen/dlclose`

Resolve symbols with `dlsym`



Windows Data Types / C Data Types

Windows data types do not natively exist for Linux.

Practically every data type that is used was defined using the typedef keyword. The traditional C data types can be used if so desired, but it is best to simply use the provided Windows data types.

```
int, long, unsigned long, double, float, etc.
```

```
INT, DWORD, BOOL, LPVOID, etc.
```

Header Files

Also known as include files: `#include <stdio.h>`

Header files, or include files, are the same thing you might already be accustomed to with your Linux development.

Windows created their own header files.

Windows.h | Windef.h | WinNt.h | WinReg.h | WinSvc.h

Windows APIs

Windows APIs and their calling conventions

This is a Windows-specific calling convention that behaves slightly differently from __cdecl convention.

API names can be very descriptive and lengthy.

Critical functionality is provided via these APIs.

Standard C Functions

- C standard library of functions are still available.
- memcpy, printf, fopen, etc.
- In the end, just use the Windows APIs.

Lab 1.3: HelloDLL

Learn how to build a simple DLL.

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

What's the Point?

What's the point?

Module Summary

Discussed how DLLs are just like SOs

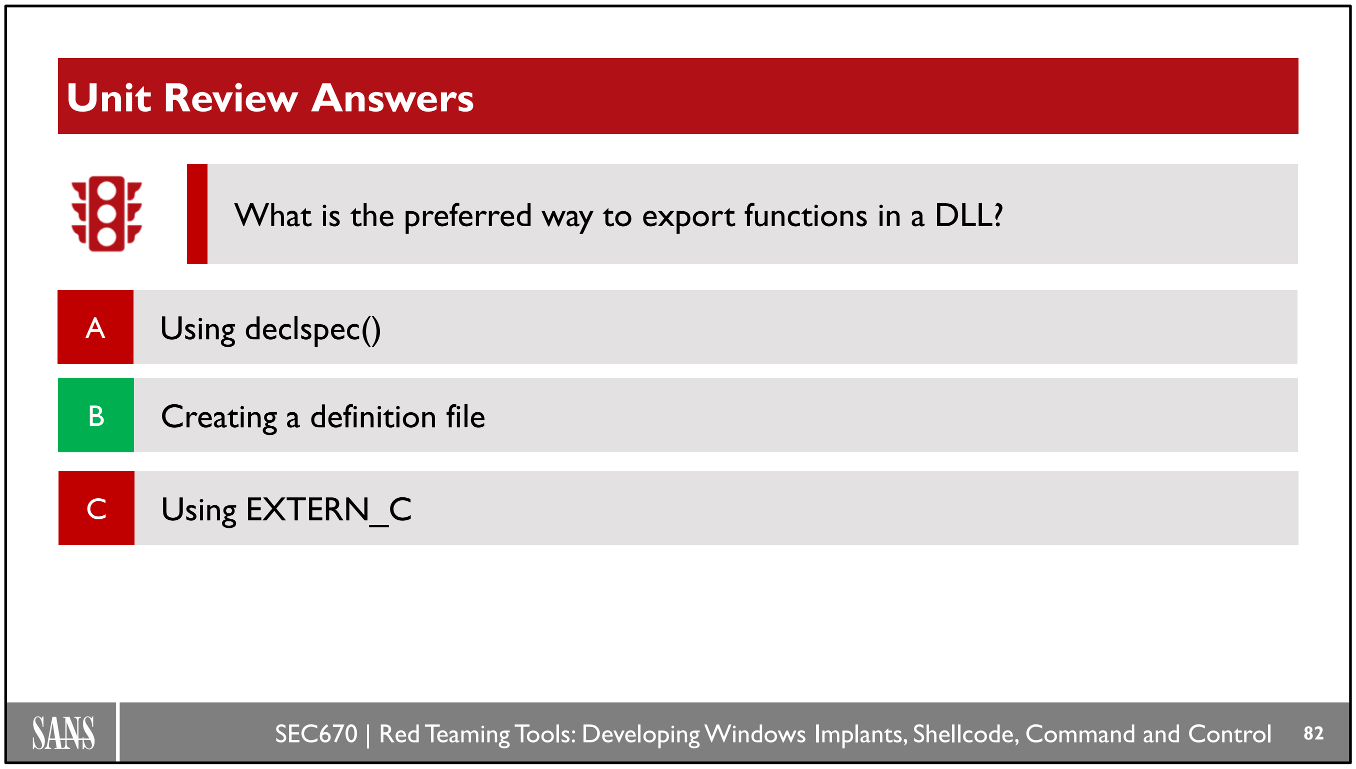
Covered how new header files bring new APIs for new functionality

Discussed a best practice: Use Windows APIs whenever possible

Unit Review Questions

What is the preferred way to export functions in a DLL?

A	Using declspec()
B	Creating a definition file
C	Using EXTERN_C



Unit Review Answers

What is the preferred way to export functions in a DLL?

A Using declspec()

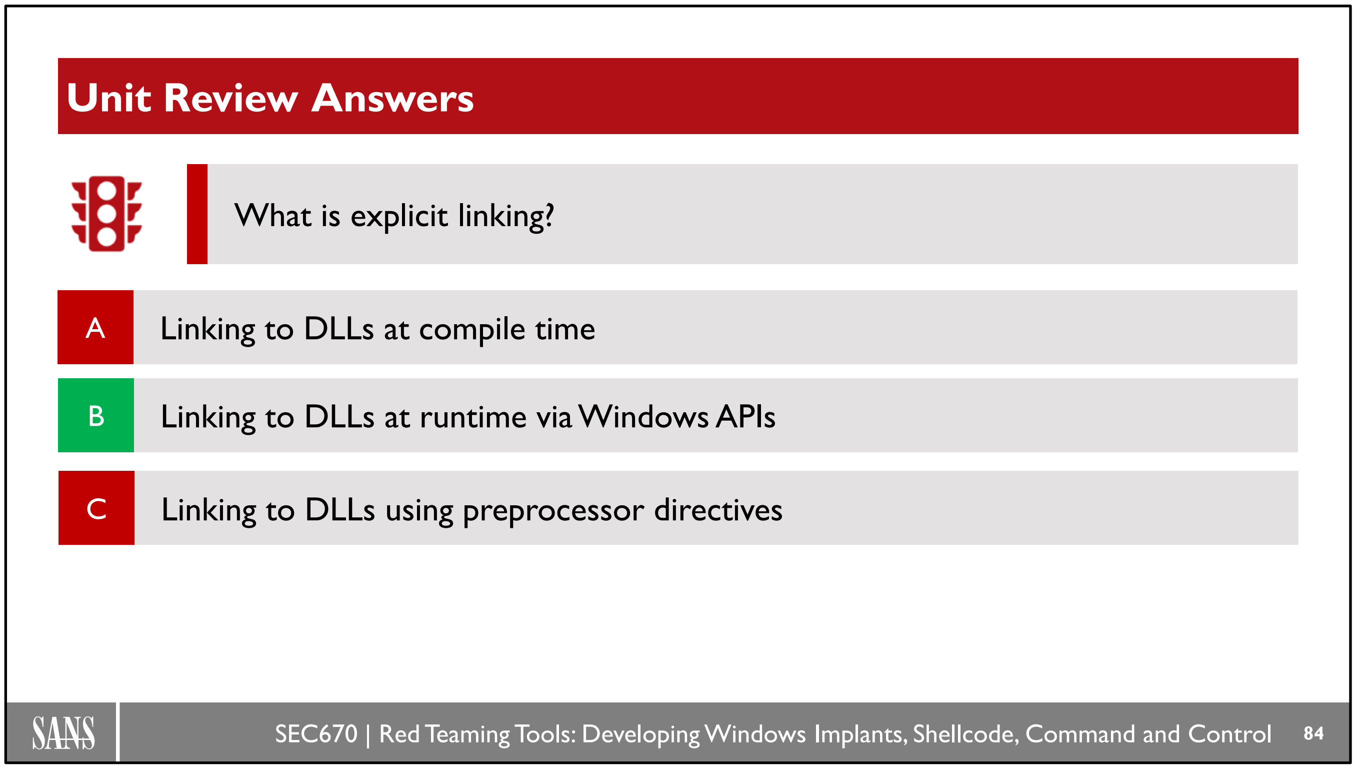
[✓ B] Creating a definition file

C Using EXTERN_C

Unit Review Questions

What is explicit linking?

A	Linking to DLLs at compile time
B	Linking to DLLs at runtime via Windows APIs
C	Linking to DLLs using preprocessor directives



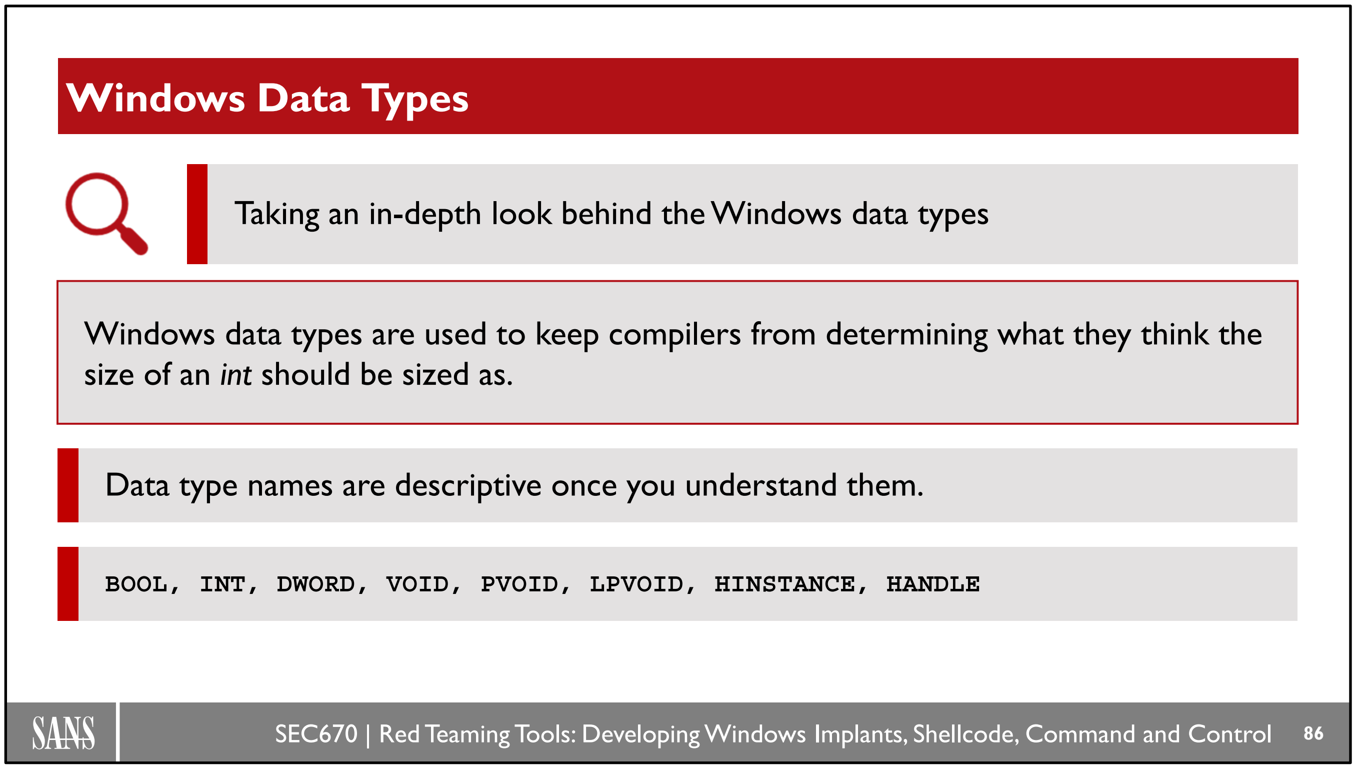
Unit Review Answers

What is explicit linking?

A	Linking to DLLs at compile time
🟩 B	Linking to DLLs at runtime via Windows APIs
C	Linking to DLLs using preprocessor directives

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



Windows Data Types

Taking an in-depth look behind the Windows data types

Windows data types are used to keep compilers from determining what they think the size of an *int* should be sized as.

Data type names are descriptive once you understand them.

```
BOOL, INT, DWORD, VOID, PVOID, LPVOID, HINSTANCE, HANDLE
```

BOOL/BOOLEAN

`BOOL` and `BOOLEAN` appear the same, but are quite different internally

Either on or off

Functions can return TRUE/FALSE

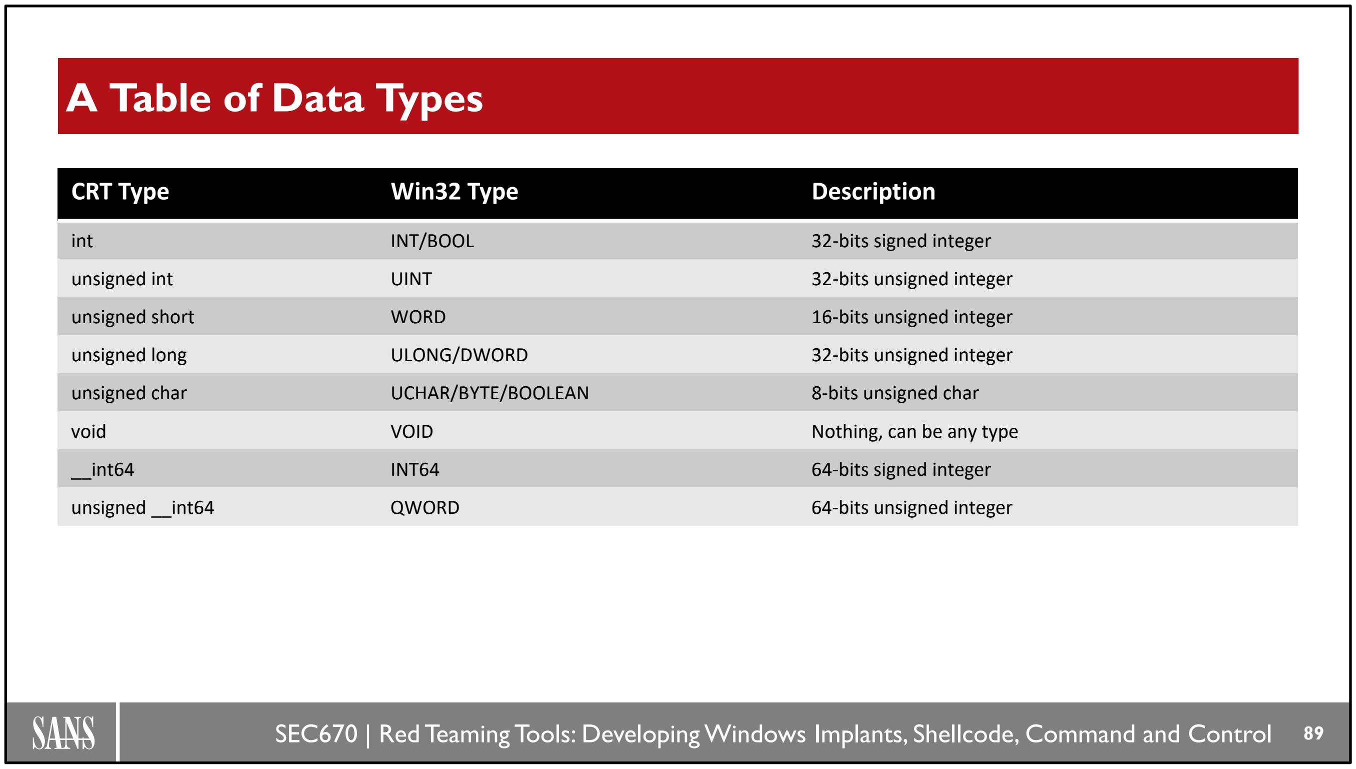
Variable can hold TRUE/FALSE values

```
// WinDef.h
typedef int BOOL;

// WinNT.h
typedef BYTE BOOLEAN;

BOOL IsRunning = TRUE;
BOOLEAN IsElevated = TRUE;
```

Example: BOOL

BOOL example

```
BOOL EnableDebug(
  HANDLE Token,
  LPCTSTR Privilege,
  BOOL EnablePrivilege
)
{
  //adjust token privileges
  //attributes to
  //SE_PRIVILEGE_ENABLED

  if (!AdjustTokenPrivileges())
    return FALSE;

  return TRUE;
}
```



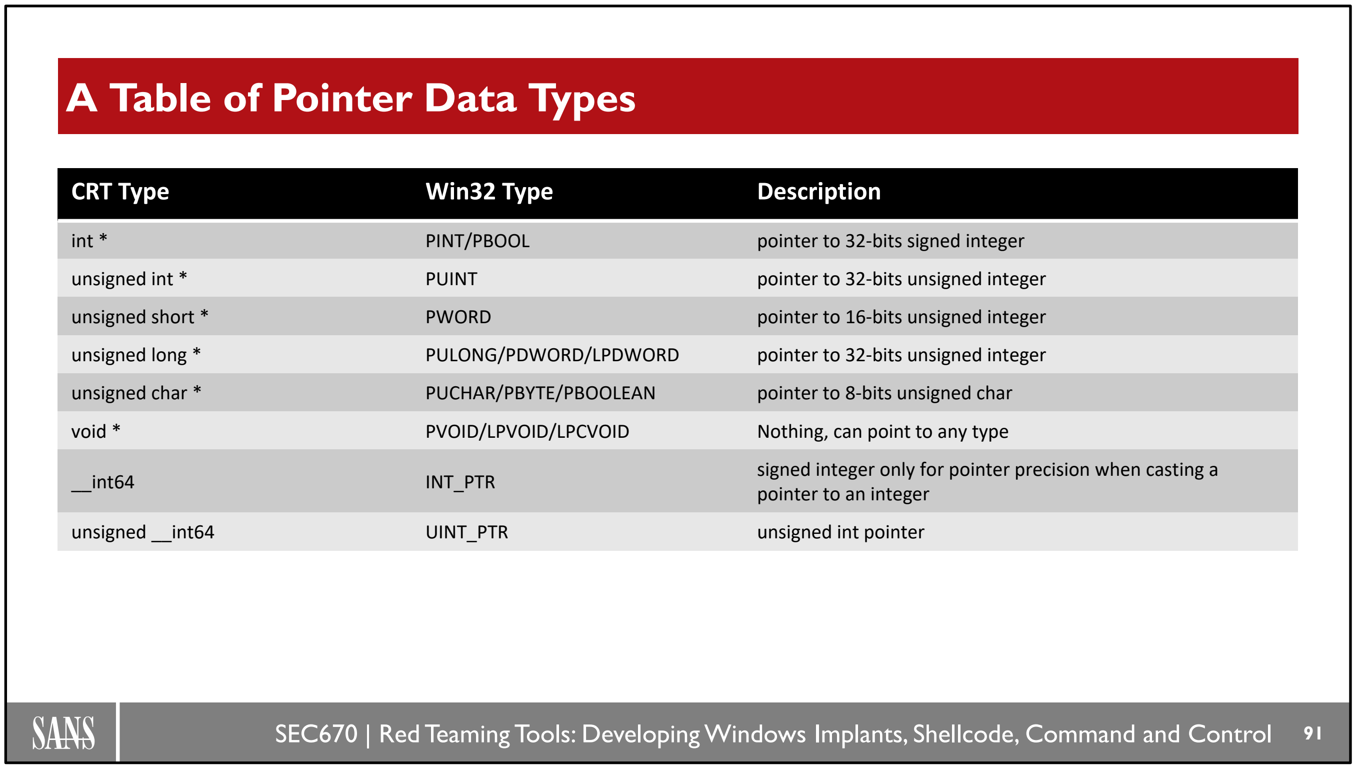
A Table of Data Types

CRT Type	Win32 Type	Description
int	INT/BOOL	32-bits signed integer
unsigned int	UINT	32-bits unsigned integer
unsigned short	WORD	16-bits unsigned integer
unsigned long	ULONG/DWORD	32-bits unsigned integer
unsigned char	UCHAR/BYTE/BOOLEAN	8-bits unsigned char
void	VOID	Nothing, can be any type
__int64	INT64	64-bits signed integer
unsigned __int64	QWORD	64-bits unsigned integer

A Table of Data Types
The table here is a nice snapshot of a small subset of the data types you will come across during your time doing Windows implant development. Many of the Windows types are typed to other Windows types, which are then typed to the standard c-runtime type. You can also see in the table that Windows can have several types that all boil down to the same CRT type. For example, UCHAR, BYTE and BOOLEAN all boil down to unsigned char, which is a single byte in memory.

Example: WORD, DWORD, LPDWORD, QWORD

```
DWORD ProcessId = GetCurrentProcessId();

PDWORD pProcessId = &ProcessId;

LPDWORD lpProcessId = &ProcessId;

DWORD pPeb32 = __readfsdword(0x30);

QWORD pPeb64 = __readgsqword(0x60);
```



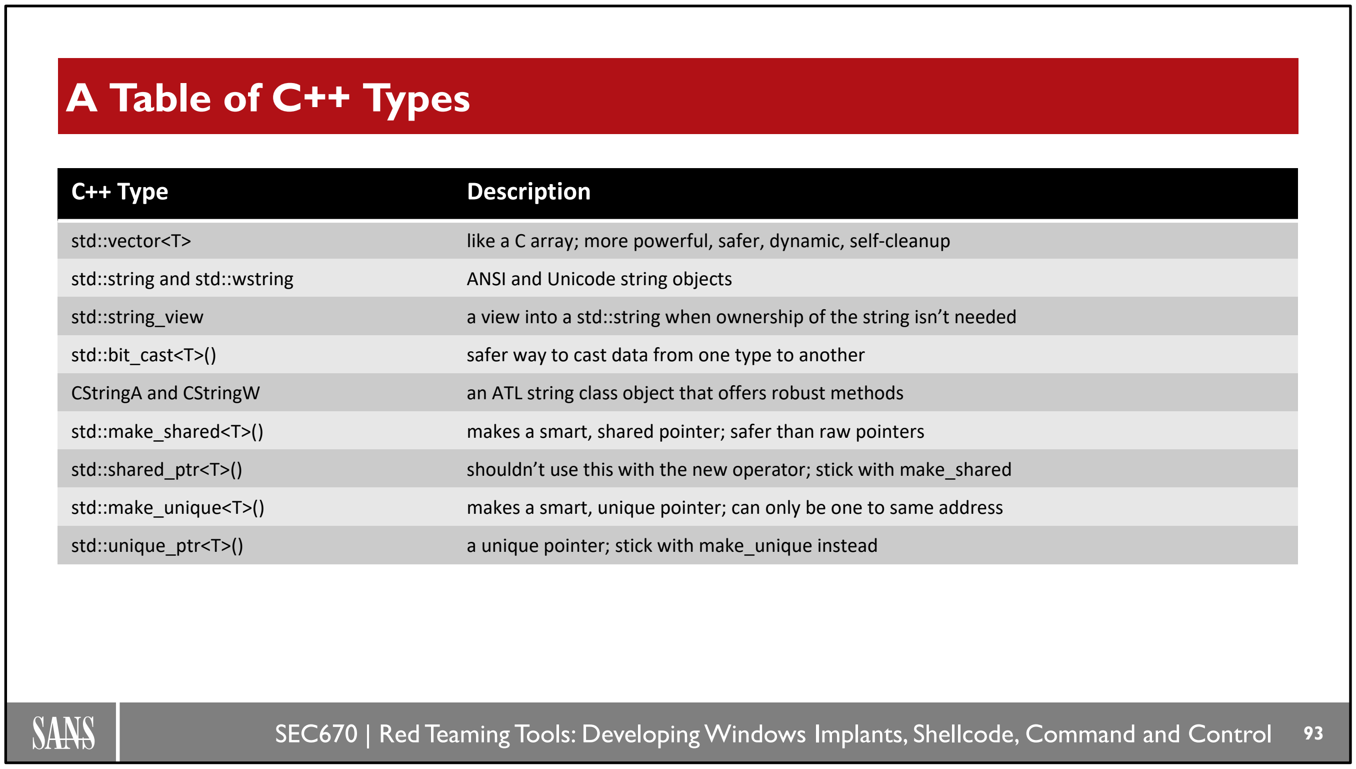
A Table of Pointer Data Types

CRT Type	Win32 Type	Description
int *	PINT/PBOOL	pointer to 32-bits signed integer
unsigned int *	PUINT	pointer to 32-bits unsigned integer
unsigned short *	PWORD	pointer to 16-bits unsigned integer
unsigned long *	PULONG/PDWORD/LPDWORD	pointer to 32-bits unsigned integer
unsigned char *	PUCHAR/PBYTE/PBOOLEAN	pointer to 8-bits unsigned char
void *	PVOID/LPVOID/LPCVOID	Nothing, can point to any type
__int64	INT_PTR	signed integer only for pointer precision when casting a pointer to an integer
unsigned __int64	UINT_PTR	unsigned int pointer

A Table of Pointer Data Types
The table here is a nice snapshot of a small subset of the pointer data types you will come across during your time doing Windows implant development. You will notice that some of the types have the letter L in front. This used to indicate the pointer is a long pointer to some data type. Long pointers are not relevant anymore and as such, they can be used interchangeably with traditional pointer types like so: LPVOID and PVOID. One thing of importance is with the PVOID data type, you cannot perform pointer arithmetic with it. Should you try to do so, Visual Studio should give it a red underline telling you it cannot be done. You should not even be able to build your project. If you are using a different compiler that lets you get away with that operation, the behavior will be unpredictable. The industry classifies that behavior as undefined behavior.

Example: VOID, PVOID, LPVOID, LPCVOID

```
VOID GiveGreeting()
{
 printf("I don't return anything. Muah ah ah!\n");
}

PVOID baseEntry = (PVOID)pimgOptionalHeader->AddressOfEntryPoint;

LPCVOID pConstant = &SomeConstant;

VOID OverflowMe(PCHAR Buffer)
{
  strcpy(Buffer, "Buffer overflows are in SEC660!");
}
```



A Table of C++ Types

C++ Type	Description
std::vector<T>	like a C array; more powerful, safer, dynamic, self-cleanup
std::string and std::wstring	ANSI and Unicode string objects
std::string_view	a view into a std::string when ownership of the string isn't needed
std::bit_cast<T>()	safer way to cast data from one type to another
CStringA and CStringW	an ATL string class object that offers robust methods
std::make_shared<T>()	makes a smart, shared pointer; safer than raw pointers
std::shared_ptr<T>()	shouldn't use this with the new operator; stick with make_shared
std::make_unique<T>()	makes a smart, unique pointer; can only be one to same address
std::unique_ptr<T>()	a unique pointer; stick with make_unique instead

A Table of C++ Types
The here captures some of the more commonly used C++ types that you'll find in this course and outside this course. Many come from the Standard Template Library (STL) and others come from the standard library. Of course, each one will have their respective header files to include, unless you are using C++ 23, which utilizes header units. If this is your first time seeing some of these data types, take your time with them as they will come with a bit of a learning curve.

The std::vector<T> is by far one of the most common and the most versatile data type. You will see this being used in the course to act as a buffer. For example, it can be used to hold the contents of a file that you read into memory.

The std::string and std::wstring are a massive enhancement from a traditional C-style string, which are just CHAR arrays. They offer many methods for comparing strings, appending, updating, etc. These are much, much easier to use than the C-style strings too. One nice small feature of std::strings is the number of characters in it. Please note that this is a character count and not a byte count, which can be two different values.

The std::string_view is a reference to a constant string. It is the perfect type for when ownership is not needed, like when a function will not be modifying the contents of the string. This means that a string will not have to be copied each time it is being passed to another function. This would increase efficiency in larger applications. The string could be anything from a C-style string to a std::string. The string_view object could not care one bit what it views.

The std::bit_cast<T>() is a safe way to type cast bits from one object to another object of type T. It is best to use this when both types are of the same size and padding, otherwise you might introduce some undefined behavior. This method is extremely safer than the popular reinterpret_cast<T> you may have used or have seen. The reinterpret_cas<T> can also introduce undefined behavior. So, with C++ 20, use bit_cast<T>.

CStringA and CStringW are part of a class under the Active Template Library, or ATL. These objects are very powerful and feature rich. They give std::strings a run for their money because of their safety, features, and ease of use.

The std::make_shared<T>() and std::make_unique<T>(). These are both used to make smart pointers. Smart pointers are much preferred over raw pointers, or traditional C-pointers. Smart pointers imitate raw pointers in a sense because they contain an address to something, and you can generally use them the same way. The best thing about smart pointers is they free you of the requirement to call delete or delete[] operators when freeing memory. Smart pointers will be automatically released when they fall out of scope and are no longer needed. This eliminates memory leaks, dangling pointers, etc. Amazing isn't it?

Example of C++ Data Types

```
std::vector<BYTE> fileContents{}; /* and empty vector */

std::string filename{}; /* an empty string */
std::string filename{ R"(c:\flag.txt)"}; /* raw literal string; escapes not needed */

std::string_view svFileName{ filename }; /* a view of filename */

/* cast the address of CloseHandle to the CloseHandle type */
/* this is a function pointer to CloseHandle */
auto pfnCloseHandle{ std::bit_cast<decltype(CloseHandle)*>(GetProcessAddress()) };

CStringA csMessage{ "" };

auto pSmart{ std::make_shared<DWORD>( 42UL ) };

auto pBestClass{ std::make_unique<DWORD>( 670UL ) }; /* unique DWORD pointer */
*pBestClass = 770UL; /* dereference the smart pointer */
auto pRawPtr = pBestClass.get(); /* obtain a raw pointer */
```

Handle Data Types

Handles are used to reference system objects.

A user application cannot directly access system objects, so they retrieve a handle to the object. The application can use the handle to modify the object. All handles are entered into an internal handle table that hold the addresses of the resources and the resource type.

- `LPHANDLE`
- `HRSRC`
- `HKEY`
- `HINSTANCE`

Handle Data Types Defined

Handles are defined in the WinNt.h header file.

```
typedef PVOID HANDLE;

typedef HANDLE* LPHANDLE;

typedef HANDLE HRSRC;

typedef HANDLE HKEY;

typedef HANDLE HINSTANCE;

typedef HINSTANCE HMODULE;
```

WINAPI, APIENTRY, CALLBACK

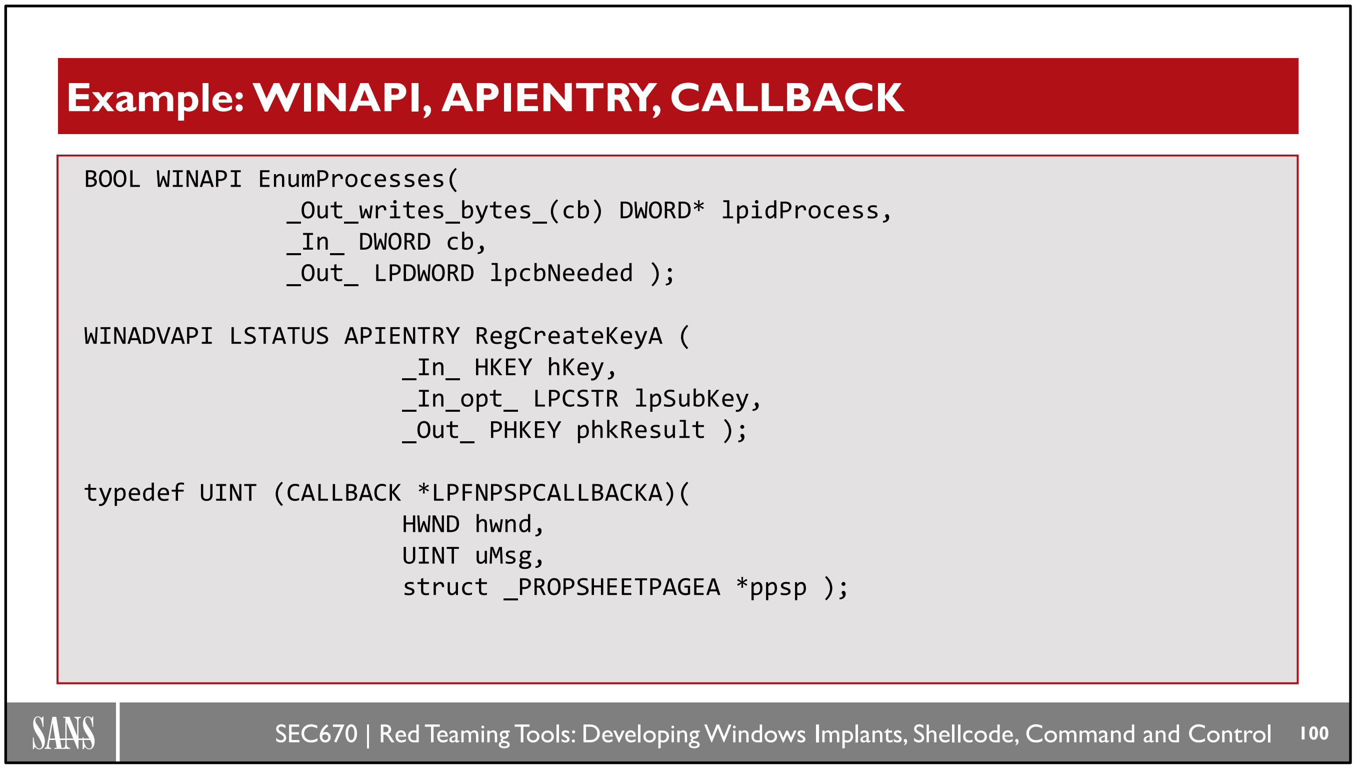
These data types are used for decorating functions that use `__stdcall`

WINAPI, APIENTRY, and CALLBACK are utilized when defining a function. Looking at some of the Windows header files, you will see these types scattered around right before a function body.

WINAPI, APIENTRY, CALLBACK Defined

WINAPI, APIENTRY, and CALLBACK are defined in WinDef.h header file

```
#define WINAPI __stdcall

#define APIENTRY WINAPI

#define CALLBACK __stdcall
```



Example: WINAPI, APIENTRY, CALLBACK

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

WINADVAPI LSTATUS APIENTRY RegCreateKeyA (
                      _In_ HKEY hKey,
                      _In_opt_ LPCSTR lpSubKey,
                      _Out_ PHKEY phkResult );

typedef UINT (CALLBACK *LPFNPSPCALLBACKA)(
                      HWND hwnd,
                      UINT uMsg,
                      struct _PROPSHEETPAGEA *ppsp );
```

Module Summary

- Learned you must become familiar with the Windows data types
- Discussed how Windows data types could prevent future headaches
- Learned some data types just stem from others; HINSTANCE and HMODULE
- Discussed how seeing the definitions can help understand something better
- Learned some data types are interchangeable; HINSTANCE and HMODULE

Unit Review Questions

What does the type WINAPI expand as?

A	__stdcall
B	__thiscall
C	__cdecl

Unit Review Answers

What does the type WINAPI expand as?

A 🟩	__stdcall
B 🟥	__thiscall
C 🟥	__cdecl

Unit Review Questions

What do the types HKEY, HINSTANCE, HRSRC have in common?

A	Nothing.
B	They are all of type HANDLE.
C	They all refer to GUI applications.

Unit Review Answers

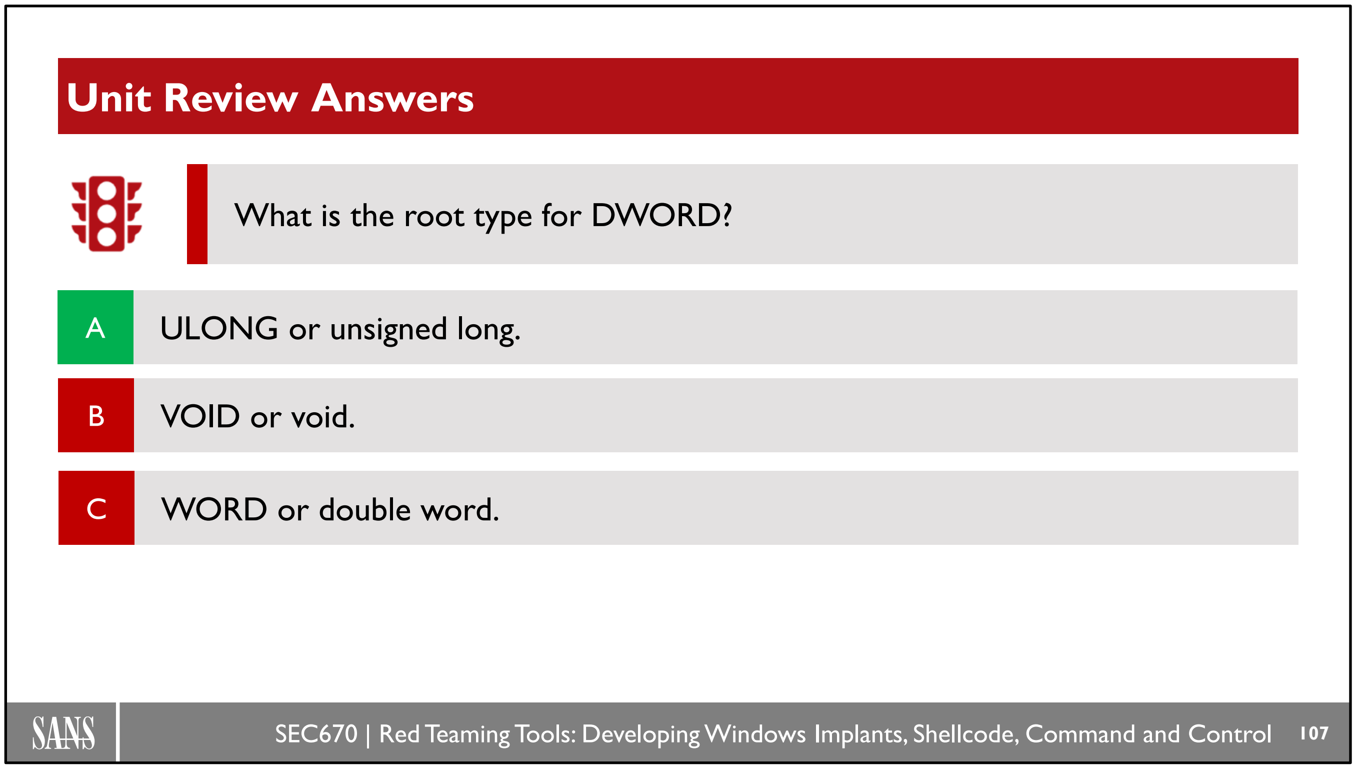
What do the types HKEY, HINSTANCE, HRSRC have in common?

A	Nothing.
🟩 B	They are all of type HANDLE.
C	They all refer to GUI applications.

Unit Review Questions

What is the root type for DWORD?

A	ULONG or unsigned long.
B	VOID or void.
C	WORD or double word.



Unit Review Answers

What is the root type for DWORD?

- 🟩 A ULONG or unsigned long.
- 🟥 B VOID or void.
- 🟥 C WORD or double word.

Unit Review Questions

On modern systems, what is the difference between LPVOID and PVOID?

A There is no difference.

B LPVOID is a long pointer, PVOID is not.

C LPVOID is not a pointer.

Unit Review Answers

On modern systems, what is the difference between LPVOID and PVOID?

A	There is no difference.
B	LPVOID is a long pointer, PVOID is not.
C	LPVOID is not a pointer.

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

__stdcall

The standard calling convention for Windows API functions

For functions with fixed number of arguments

Callee cleans the stack before return

Offers speed and efficiency

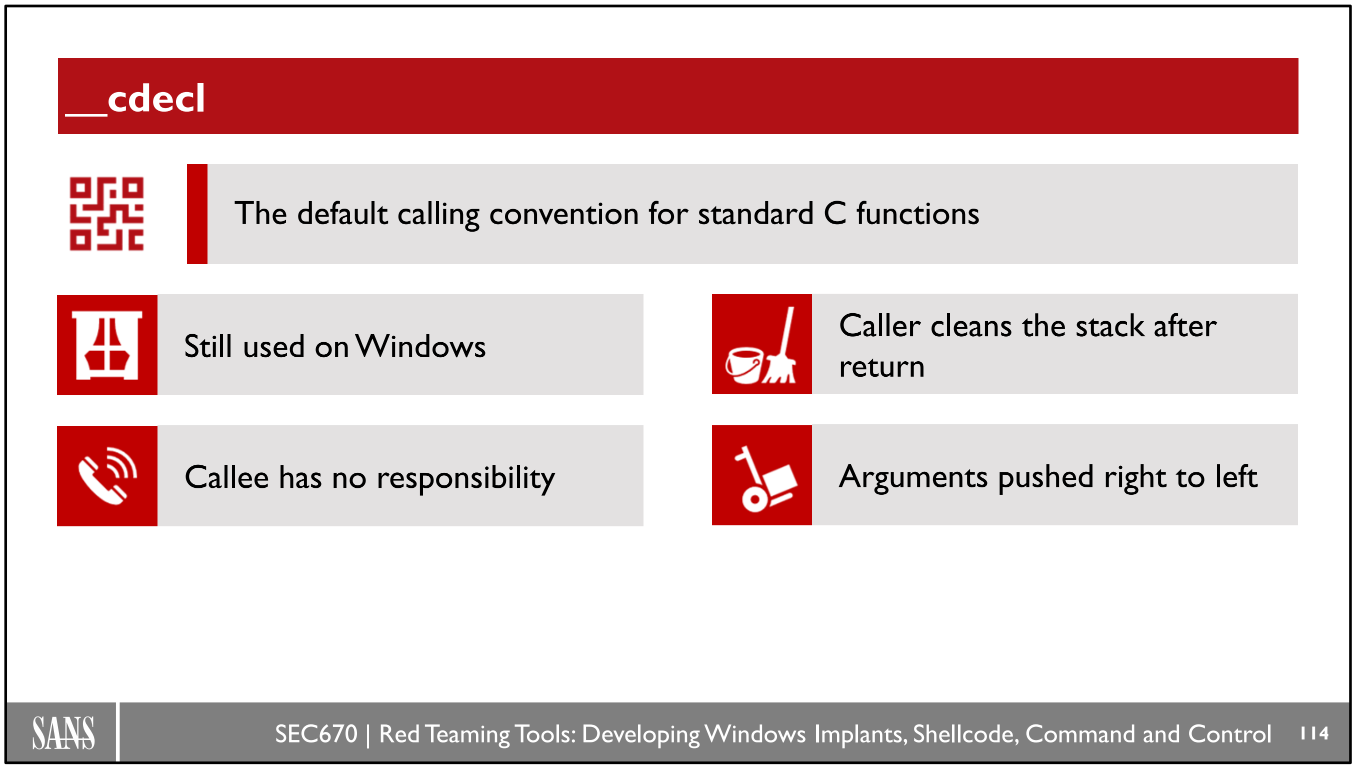
Arguments pushed right to left

Example: __stdcall (x86)

```
CreateFileW( L”hello.txt”, GENERIC_WRITE, 0, NULL, CREATE_NEW,
       FILE_ATTRIBUTE_NORMAL, NULL );

push  0                      ; hTemplateFile
push  80h                    ; dwFlagsAndAttributes
push  1                      ; dwCreationDisposition
push  0                      ; lpSecurityAttributes
push  0                      ; dwShareMode
push  40000000h                       ; dwDesiredAccess
push  offset FileName                 ; "hello.txt"
call  ds:__imp__CreateFileW@28        ; CreateFileW(x,x,x,x,x,x,x)
mov   edi, eax                        ; caller does no clean up
```

Example: (x64)

```
CreateFileW( L"hello.txt", GENERIC_WRITE, 0, NULL, CREATE_NEW,
       FILE_ATTRIBUTE_NORMAL, NULL );

mov      qword ptr ss:[rsp+30], 0            ; hTemplateFile on stack
lea      rcx, qword ptr ds:[7FF633682230]    ; ptr to L"hello.txt"
mov      dword ptr ss:[rsp+28], 80           ; FILE_ATTRIBUTE_NORMAL on stack
xor      r9d, r9d                            ; 0 lpSecurityAttributes
xor      r8d, r8d                            ; 0 dwSharedMode
mov      dword ptr ss:[rsp+20], 1            ; CREATE_NEW on stack
mov      edx, 40000000                       ; GENERIC_WRITE
call     qword ptr ds:[<&CreateFileW>]
```



__cdecl

The default calling convention for standard C functions

Still used on Windows

Caller cleans the stack after return

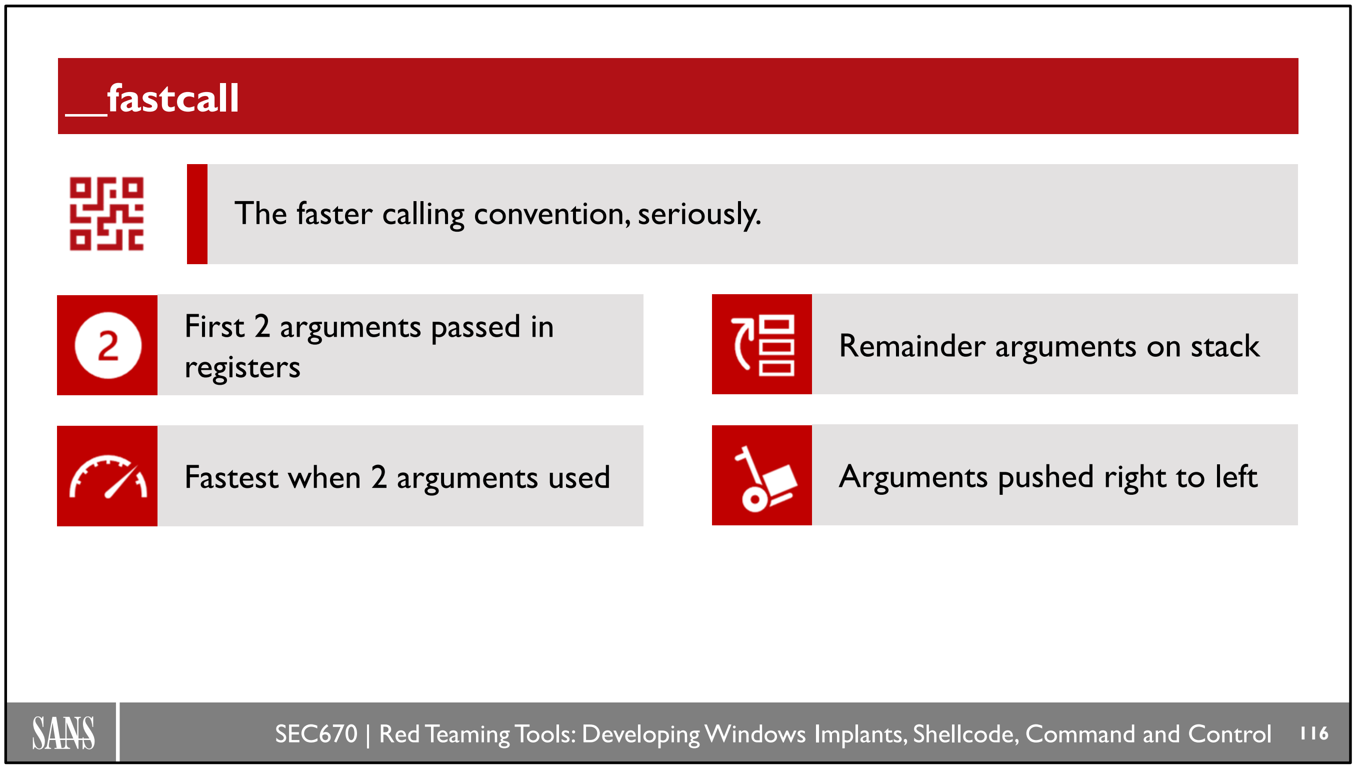
Callee has no responsibility

Arguments pushed right to left

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 114

Example: __cdecl

```
printf( “Score: %d, Avg: %d, High: %d, Low: %d\n”, 670, 92, 670, 660 );

push  294h           ; 660
push  29Eh           ; 670
push  5Ch            ; 92
push  29Eh           ; 670
push  offset _Format ; "Score: %d, Avg: %d, High: %d, Low: %d\n"
call  _printf
add   esp, 14h       ; caller cleans up the stack
```



__fastcall

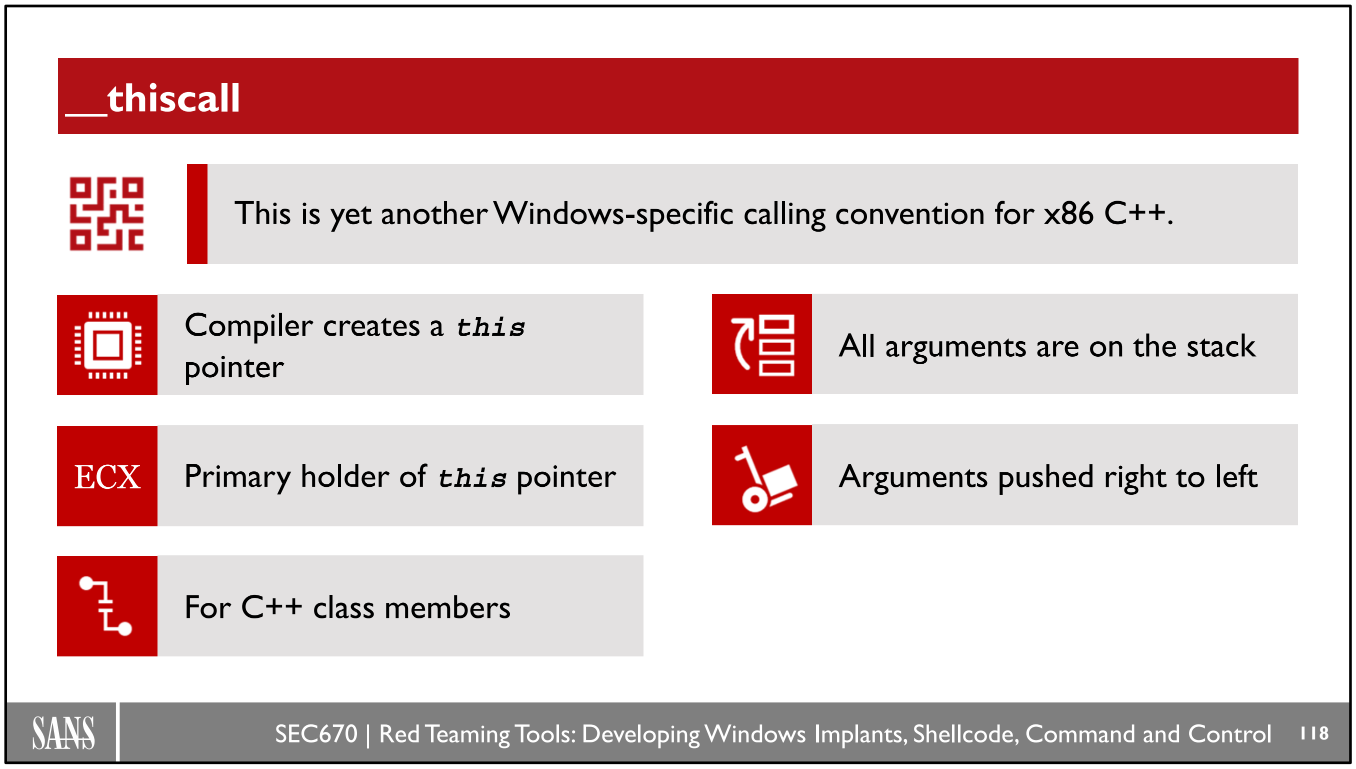
The faster calling convention, seriously.

- First 2 arguments passed in registers
- Remainder arguments on stack
- Fastest when 2 arguments used
- Arguments pushed right to left

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 116

Example: __fastcall

```
INT FASTCALL SomeFastFunction( PDWORD, PDWORD, BOOL);

------Main------
push  1                     ; Passed
lea   edx, [ebp+dwAge]      ; Age
lea   ecx, [ebp+dwDollars]  ; Dollars
call  @SomeFastFunction@12  ; SomeFastFunction(x,x,x)
mov   [ebp+Age], eax        ; no clean up

-------SomeFastFunction-------
[..snip..]

retn  4            ; cleaning up the stack
```



__thiscall

This is yet another Windows-specific calling convention for x86 C++.

- Compiler creates a ***this*** pointer
- ECX — Primary holder of ***this*** pointer
- For C++ class members
- All arguments are on the stack
- Arguments pushed right to left

Example: __thiscall

```
class MyClass { public: void PrintMsg( char* msg ); };

// this is really the same as above
struct TheClass {void TheMessage( char* msg ); };

// defined separately with the scope resolution operator ::
void MyClass::PrintMsg( char* msg ){ printf( msg ); };
```

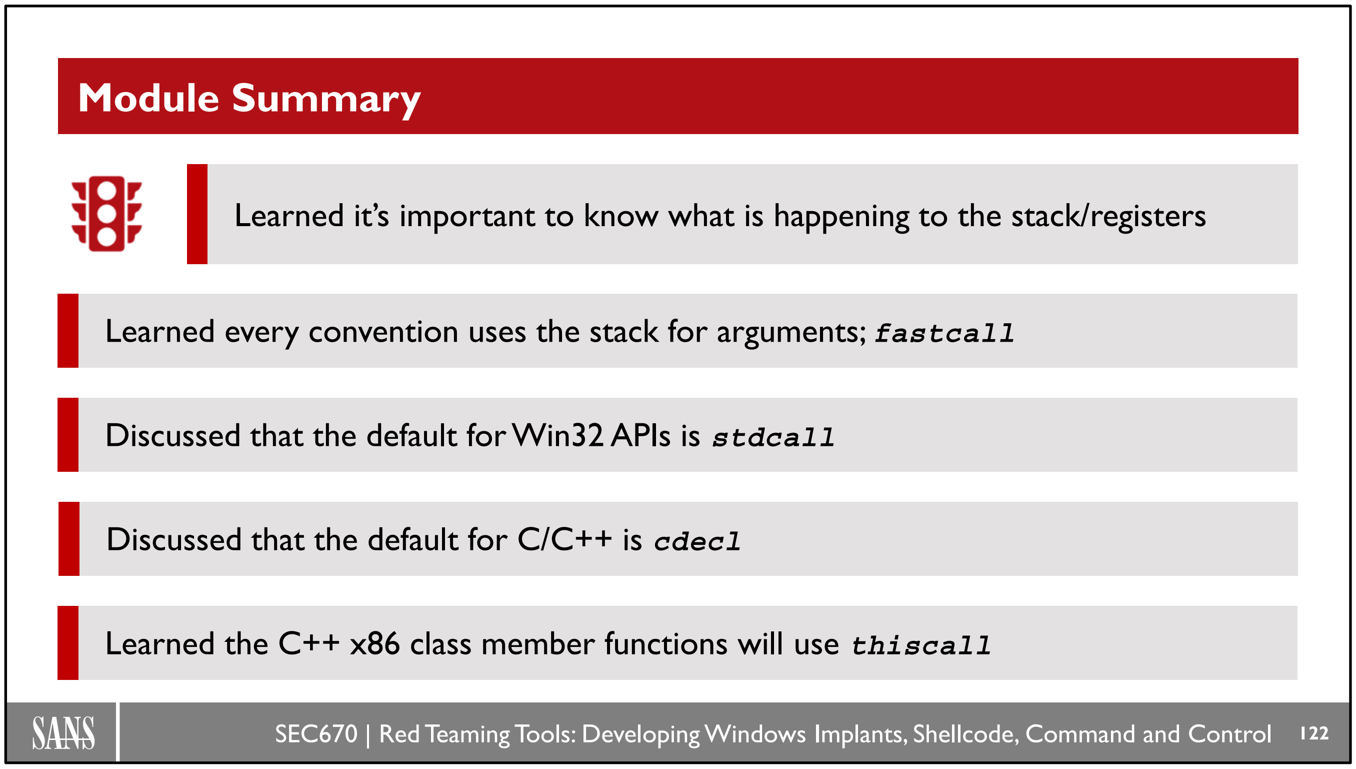
Lab 1.4: Call Me Maybe

Learn how to use the various calling conventions.

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

What's the Point?

What's the point?



Module Summary

Learned it's important to know what is happening to the stack/registers

Learned every convention uses the stack for arguments; ***fastcall***

Discussed that the default for Win32 APIs is ***stdcall***

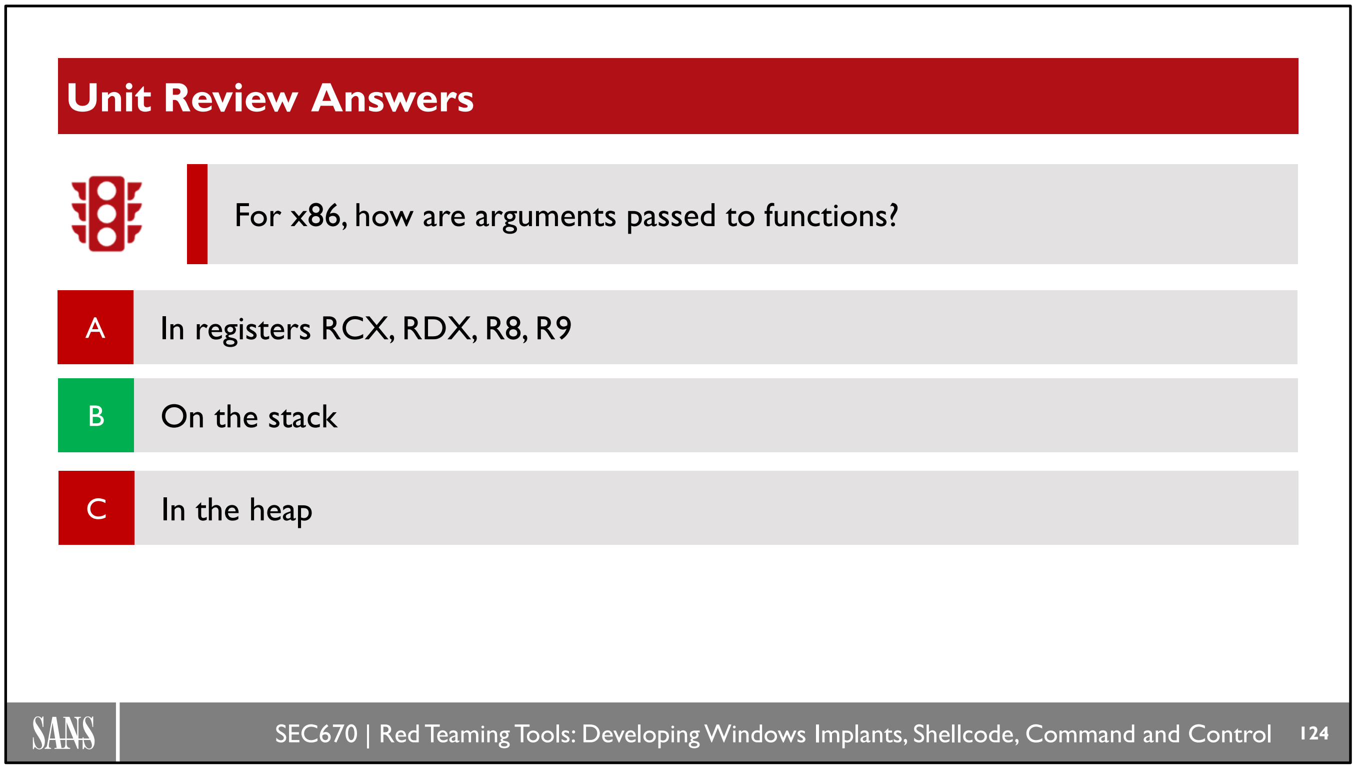
Discussed that the default for C/C++ is ***cdecl***

Learned the C++ x86 class member functions will use ***thiscall***

Unit Review Questions

For x86, how are arguments passed to functions?

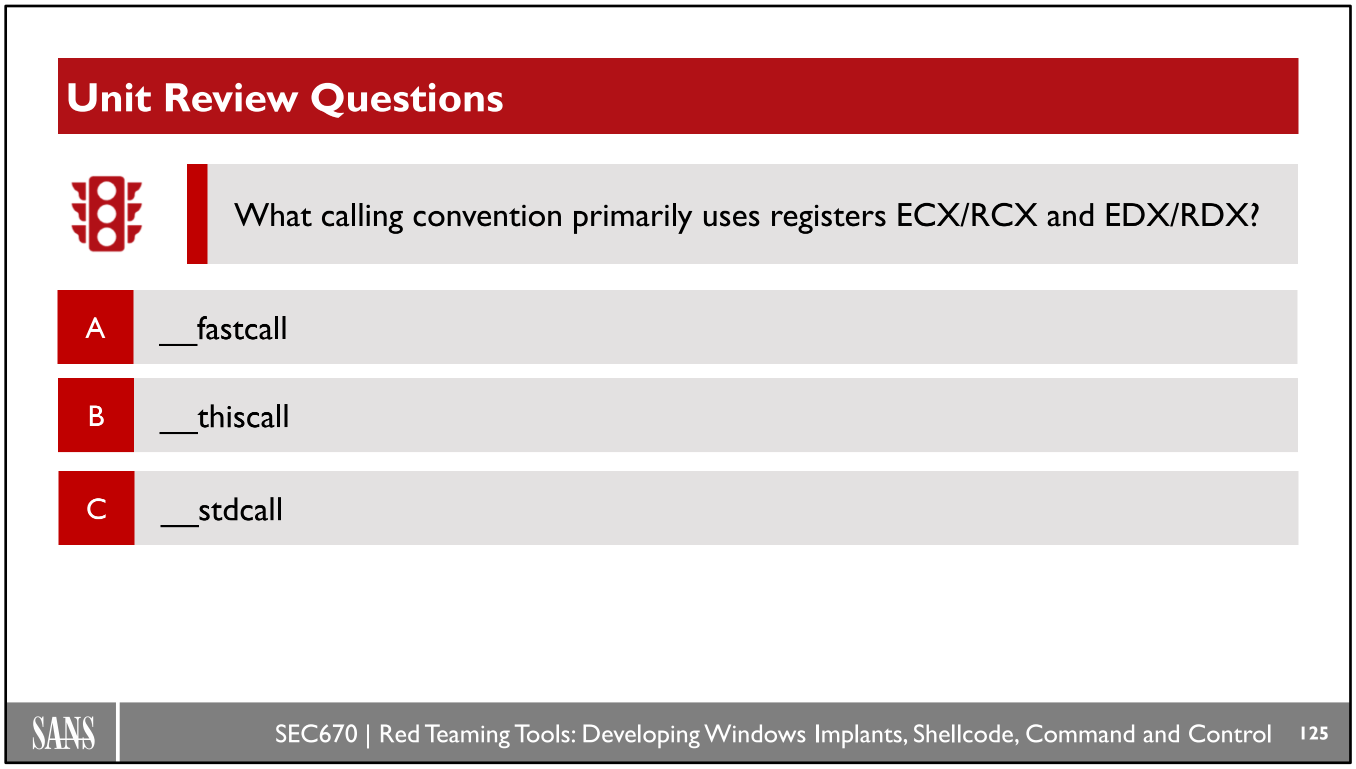
A	In registers RCX, RDX, R8, R9
B	On the stack
C	In the heap



Unit Review Answers

For x86, how are arguments passed to functions?

🟥 A	In registers RCX, RDX, R8, R9
🟩 B	On the stack
🟥 C	In the heap



Unit Review Questions

What calling convention primarily uses registers ECX/RCX and EDX/RDX?

A __fastcall

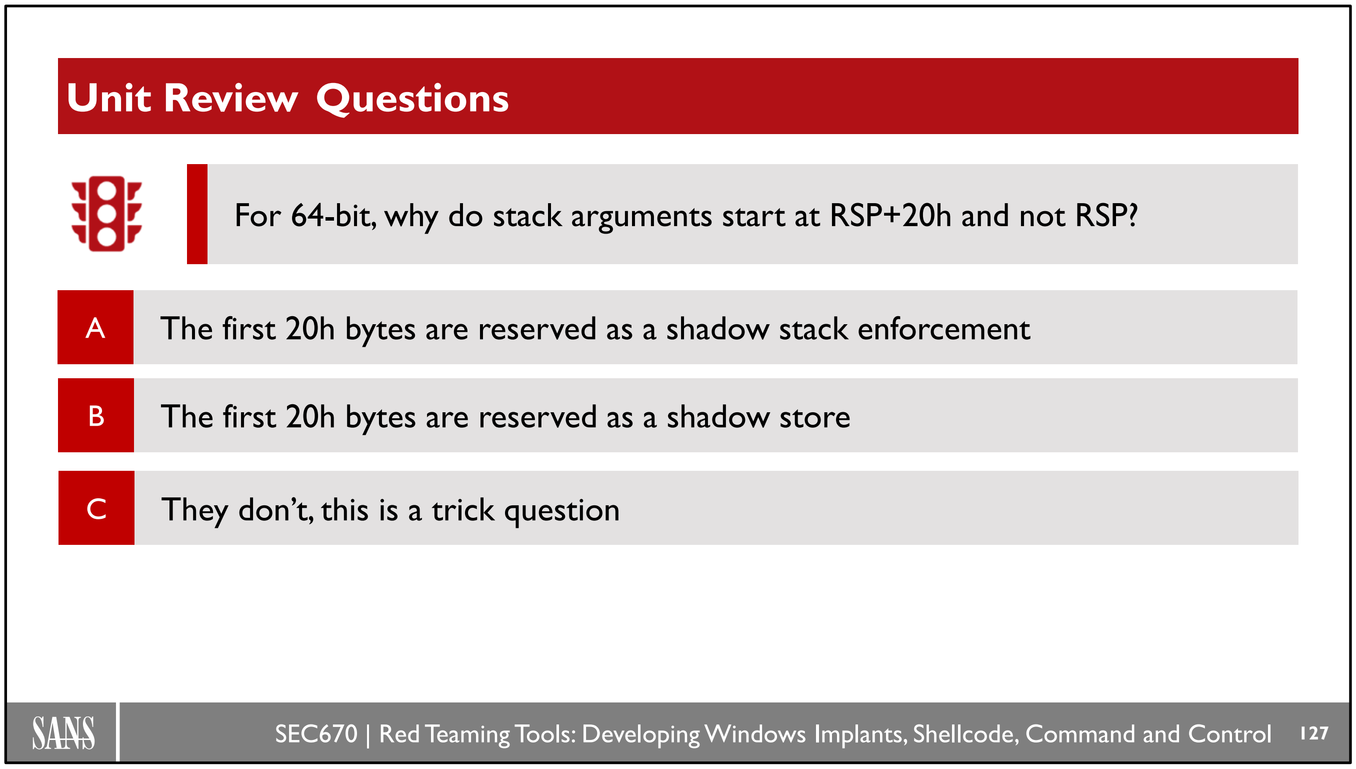
B __thiscall

C __stdcall

Unit Review Answers

What calling convention primarily uses registers ECX/RCX and EDX/RDX?

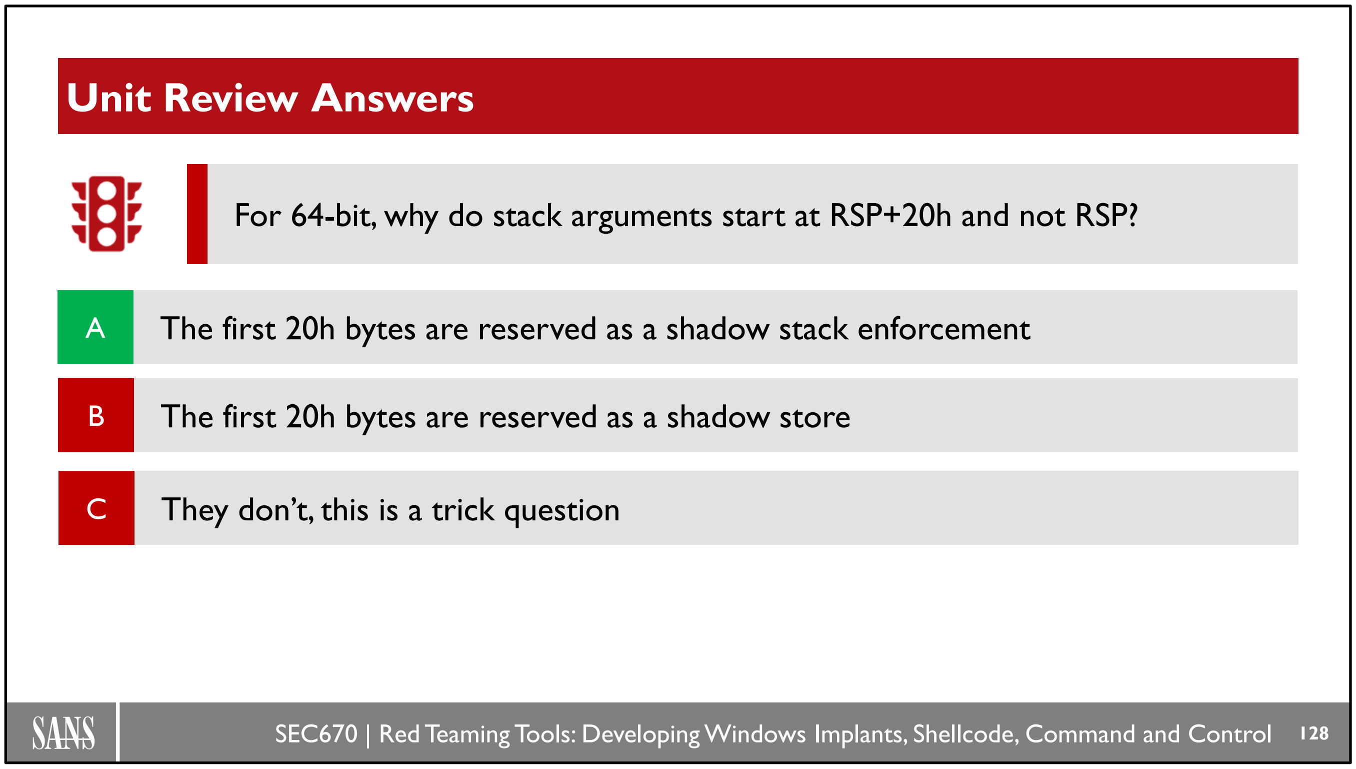
A	__fastcall
B	__thiscall
C	__stdcall



Unit Review Questions

For 64-bit, why do stack arguments start at RSP+20h and not RSP?

A	The first 20h bytes are reserved as a shadow stack enforcement
B	The first 20h bytes are reserved as a shadow store
C	They don't, this is a trick question



Unit Review Answers

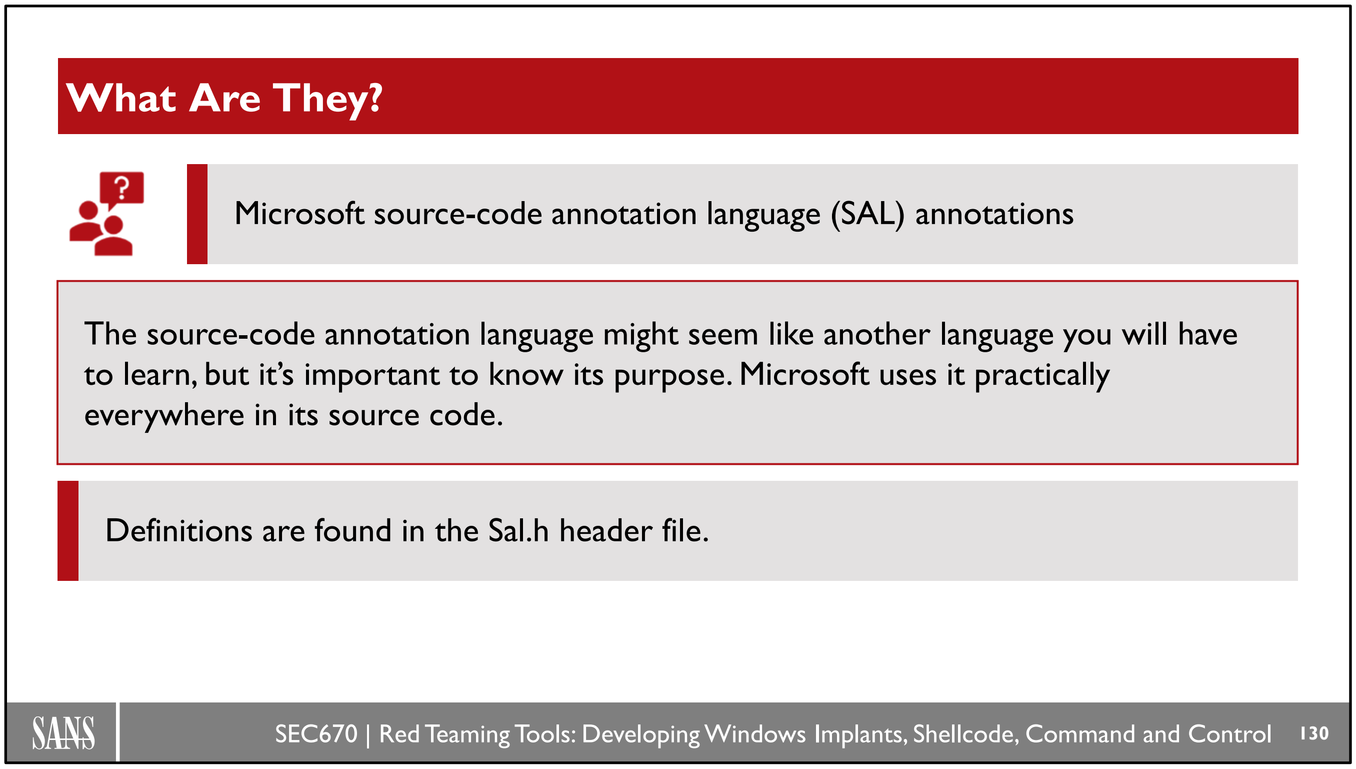
For 64-bit, why do stack arguments start at RSP+20h and not RSP?

A	The first 20h bytes are reserved as a shadow stack enforcement
B	The first 20h bytes are reserved as a shadow store
C	They don't, this is a trick question

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 1



What Are They?

Microsoft source-code annotation language (SAL) annotations

The source-code annotation language might seem like another language you will have to learn, but it's important to know its purpose. Microsoft uses it practically everywhere in its source code.

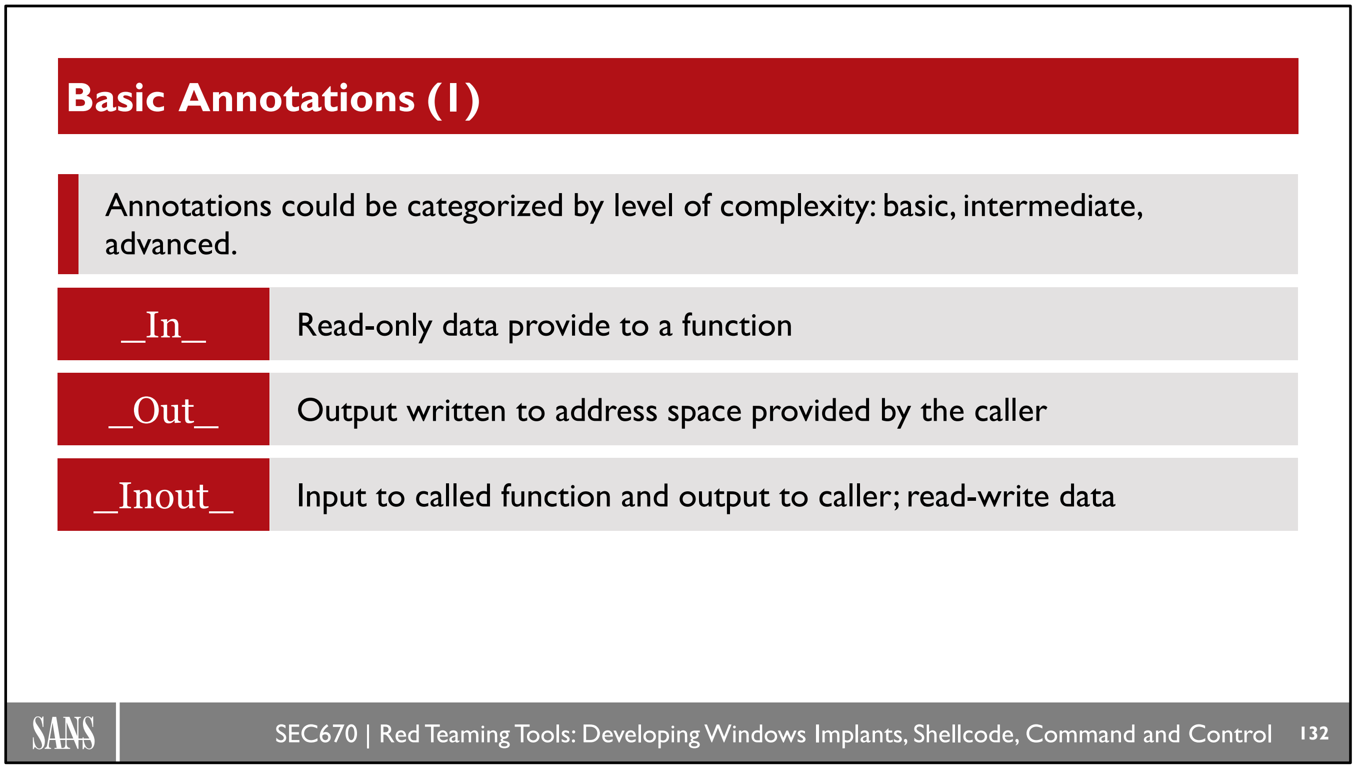
Definitions are found in the Sal.h header file.

Why Use Them?

Automatic static code analysis built into Visual Studio

SAL annotations make the intended use of function parameters clear

Very inexpensive compiler check for your code



Basic Annotations (1)

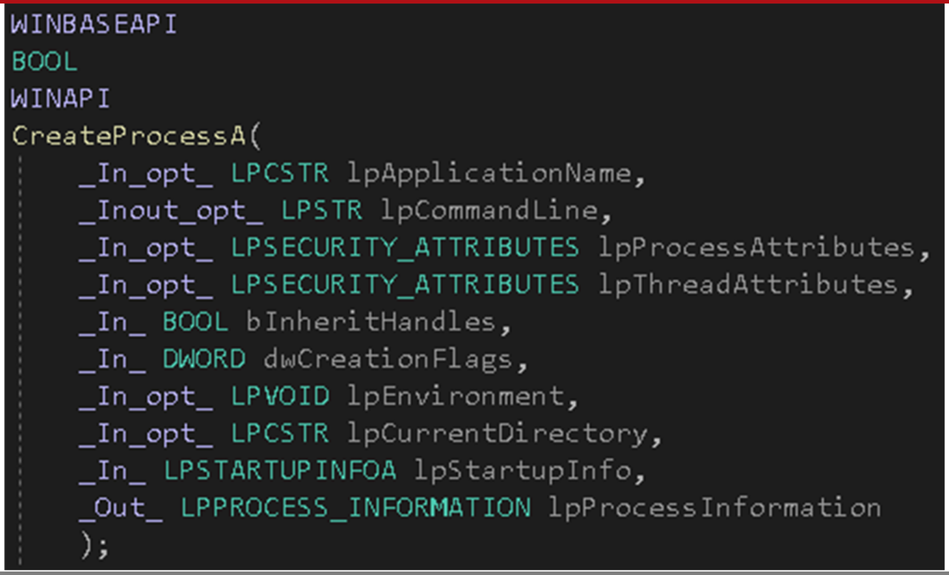
Annotations could be categorized by level of complexity: basic, intermediate, advanced.

_In_	Read-only data provide to a function
Out	Output written to address space provided by the caller
Inout	Input to called function and output to caller; read-write data

Basic Annotations (2)

_In_opt_	Inbound pointer that could also be NULL
_Out_opt_	Output to the caller that is optional
_Ret_z_	The function will return a null-terminated value, like a string

Examples: Basic Annotations



```
WINBASEAPI
BOOL
WINAPI
CreateProcessA(
    _In_opt_ LPCSTR lpApplicationName,
    _Inout_opt_ LPSTR lpCommandLine,
    _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ BOOL bInheritHandles,
    _In_ DWORD dwCreationFlags,
    _In_opt_ LPVOID lpEnvironment,
    _In_opt_ LPCSTR lpCurrentDirectory,
    _In_ LPSTARTUPINFOA lpStartupInfo,
    _Out_ LPPROCESS_INFORMATION lpProcessInformation
    );
```

Intermediate Annotations (1)

_In_reads_(s)
_In_reads_bytes_(s)
	When the parameter could have 1+ elements to read where (s) is the number of elements or bytes when _bytes_ is used

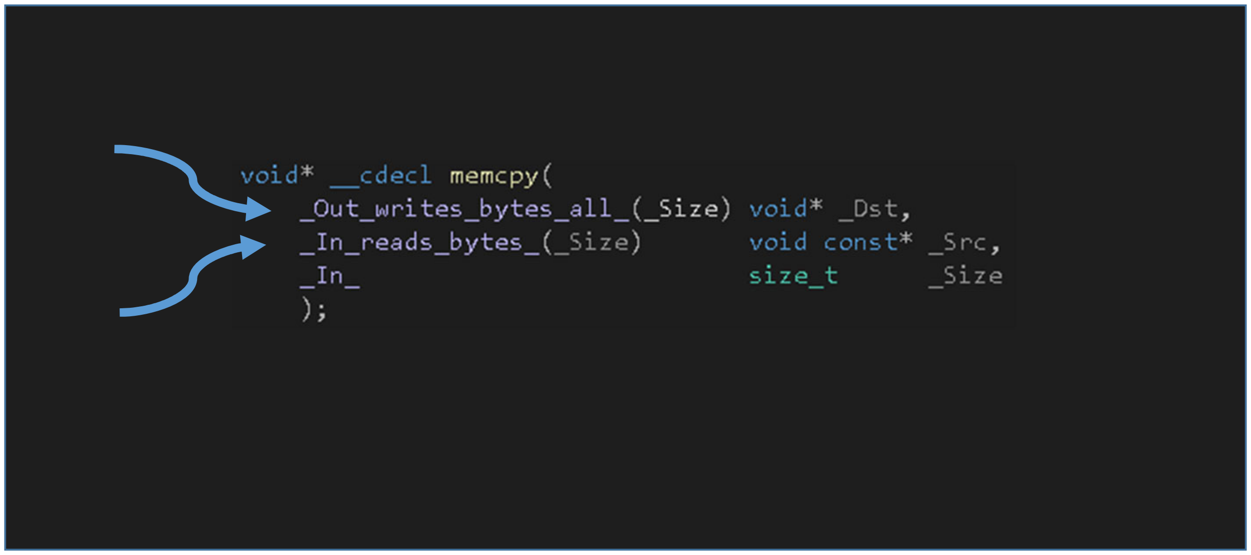
_In_reads_z_(s)
_In_reads_or_z_(s)
	The parameter is null-terminated and possibly 1+ elements should be read

_Out_writes_(s)
_Out_writes_bytes_(s)
	An out parameter that will write 1+ elements or bytes if _bytes_ is used

Intermediate Annotations (2)

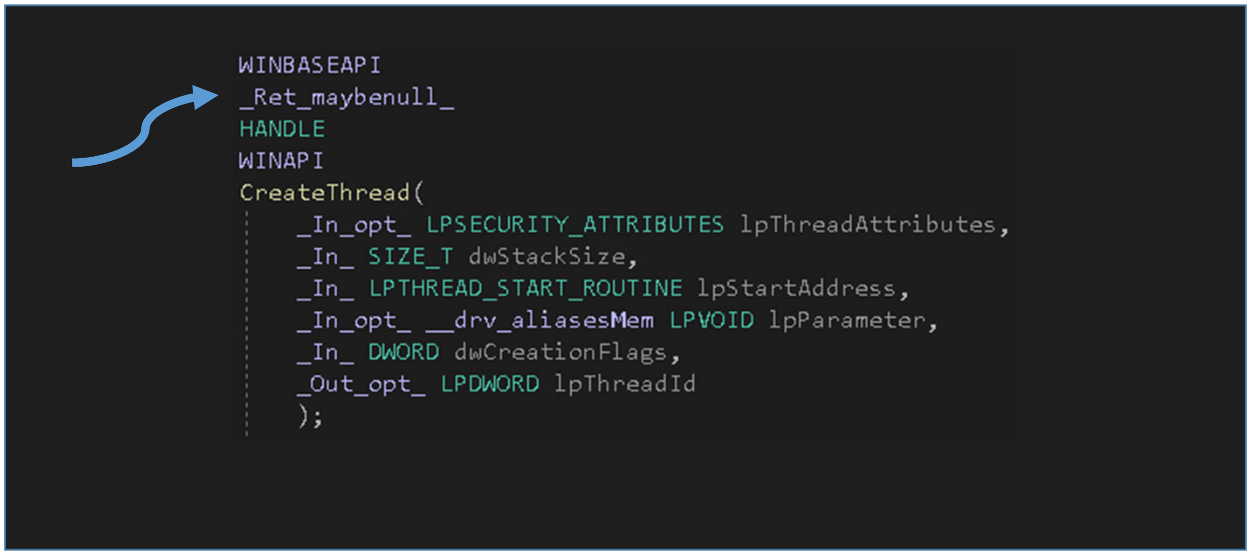
_Out_writes_bytes_all_(s)	All of 's' bytes will be written out
_Ret_maybenull_	Function's return value will, could, or won't be a nullptr

Example: Intermediate Annotations (1)



```
void* __cdecl memcpy(
    _Out_writes_bytes_all_(_Size) void* _Dst,
    _In_reads_bytes_(_Size)       void const* _Src,
    _In_                          size_t      _Size
    );
```

Example: Intermediate Annotations (2)

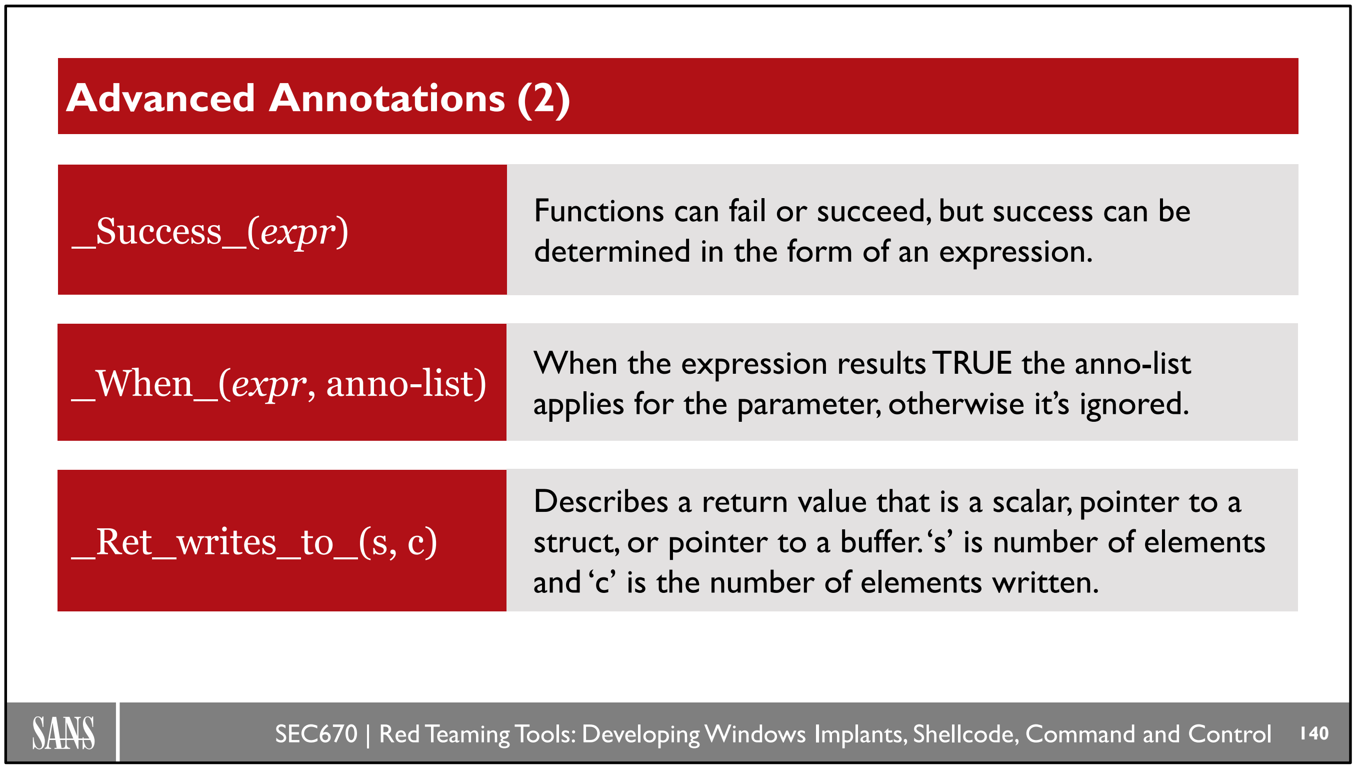


```
WINBASEAPI
_Ret_maybenull_
HANDLE
WINAPI
CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ SIZE_T dwStackSize,
    _In_ LPTHREAD_START_ROUTINE lpStartAddress,
    _In_opt_ __drv_aliasesMem LPVOID lpParameter,
    _In_ DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId
    );
```

Advanced Annotations (1)

_Inout_updates_(s)
_Inout_updates_bytes_(s)
	Accepts a pointer to an array of elements read and written to by the function.

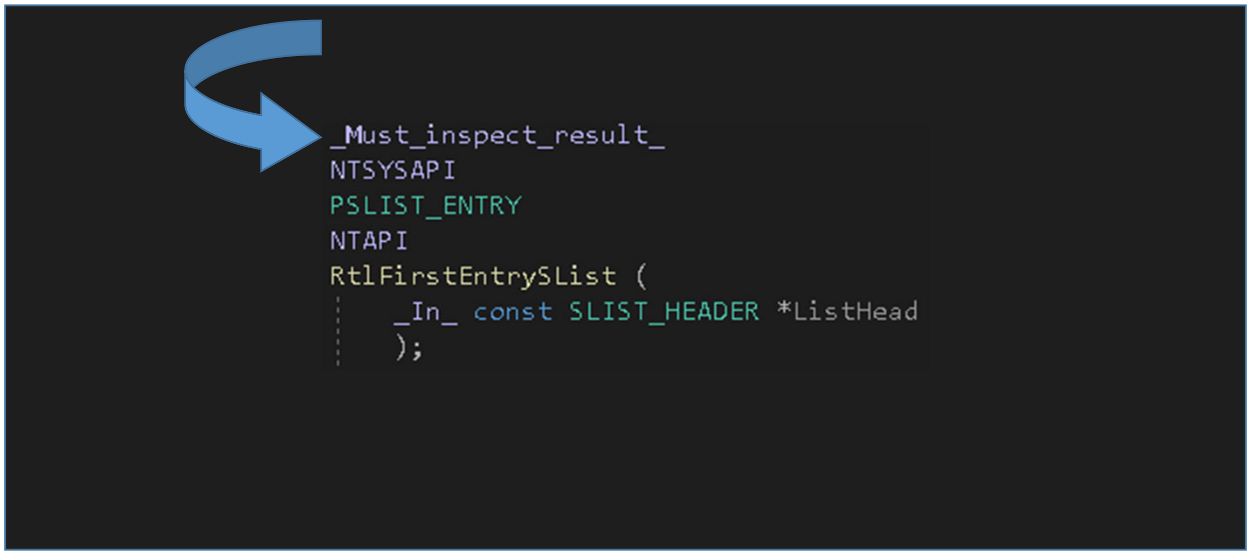
_Check_return_ _Must_inspect_result_	The caller must check the return value of the function.
_Use_decl_annotations_	Forces the reuse of a function's annotations for a function's declaration to avoid the duplication of them.



Advanced Annotations (2)

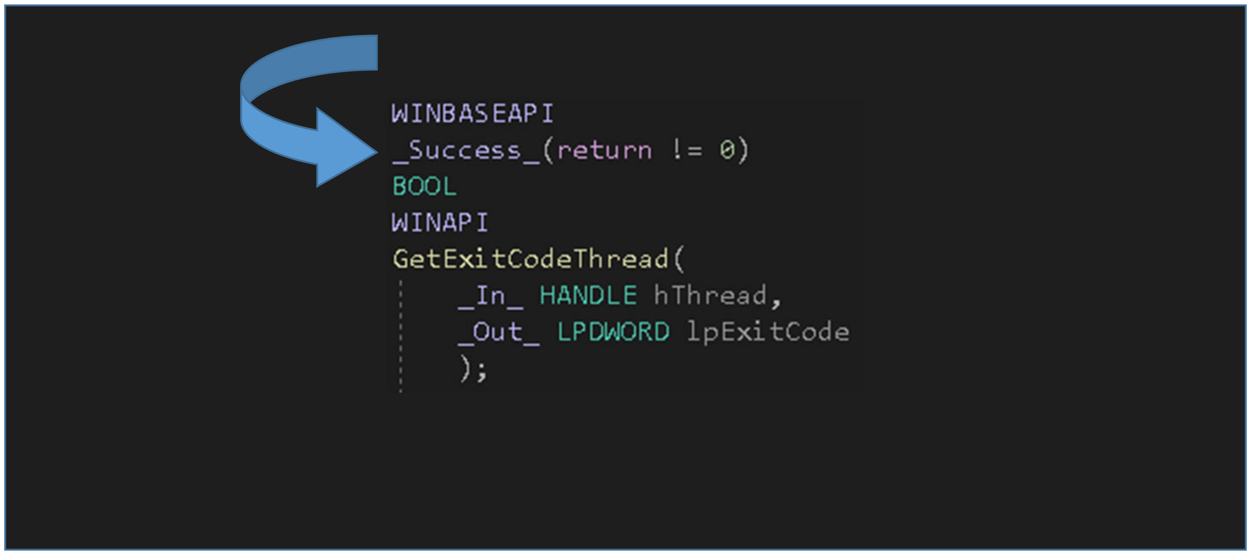
Success(*expr*)	Functions can fail or succeed, but success can be determined in the form of an expression.
When(*expr*, anno-list)	When the expression results TRUE the anno-list applies for the parameter, otherwise it's ignored.
_Ret_writes_to_(s, c)	Describes a return value that is a scalar, pointer to a struct, or pointer to a buffer. 's' is number of elements and 'c' is the number of elements written.

Example: Advanced Annotations (1)



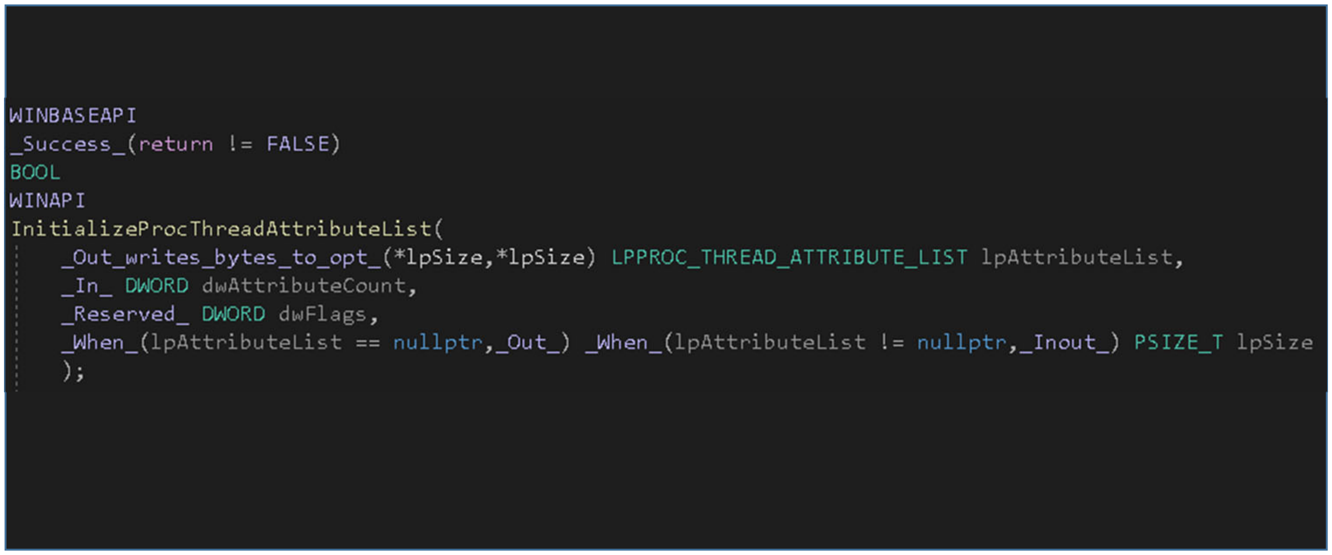
```
_Must_inspect_result_
NTSYSAPI
PSLIST_ENTRY
NTAPI
RtlFirstEntrySList (
    _In_ const SLIST_HEADER *ListHead
    );
```

Example: Advanced Annotations (2)



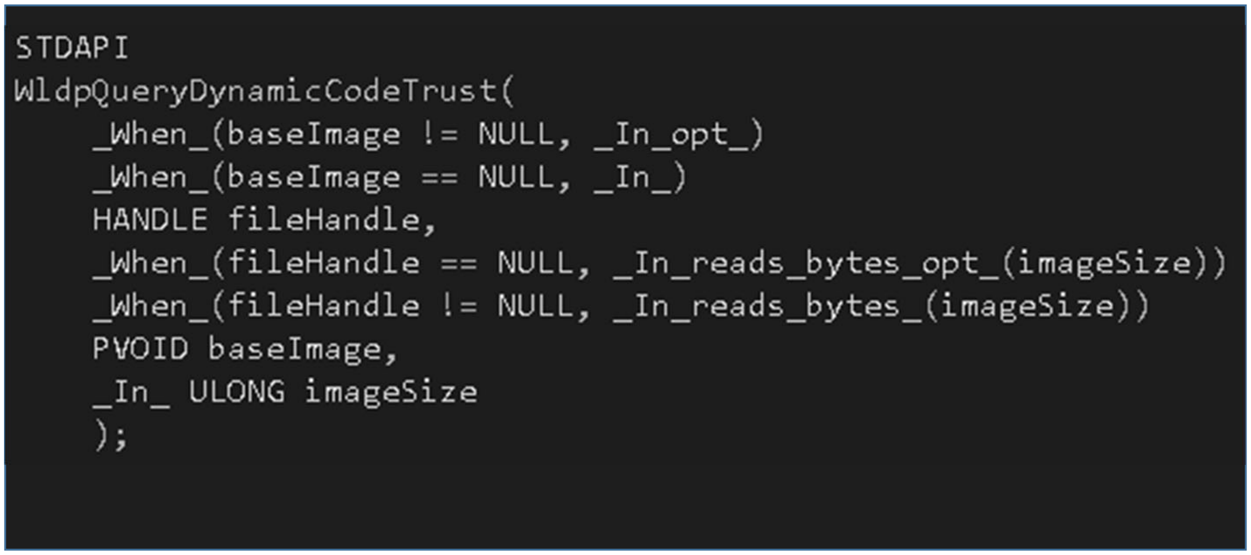
```
WINBASEAPI
_Success_(return != 0)
BOOL
WINAPI
GetExitCodeThread(
    _In_ HANDLE hThread,
    _Out_ LPDWORD lpExitCode
    );
```

Example: Advanced Annotations (3)



```
WINBASEAPI
_Success_(return != FALSE)
BOOL
WINAPI
InitializeProcThreadAttributeList(
    _Out_writes_bytes_to_opt_(*lpSize,*lpSize) LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
    _In_ DWORD dwAttributeCount,
    _Reserved_ DWORD dwFlags,
    _When_(lpAttributeList == nullptr,_Out_) _When_(lpAttributeList != nullptr,_Inout_) PSIZE_T lpSize
    );
```

Example: Advanced Annotations (4)



```
STDAPI
WldpQueryDynamicCodeTrust(
    _When_(baseImage != NULL, _In_opt_)
    _When_(baseImage == NULL, _In_)
    HANDLE fileHandle,
    _When_(fileHandle == NULL, _In_reads_bytes_opt_(imageSize))
    _When_(fileHandle != NULL, _In_reads_bytes_(imageSize))
    PVOID baseImage,
    _In_ ULONG imageSize
    );
```

Lab 1.5: Safer with SAL

Using SAL annotations makes your code more understandable.

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

What's the Point?

What's the point?

Additional Information

Microsoft is your best source for understanding SAL annotations.

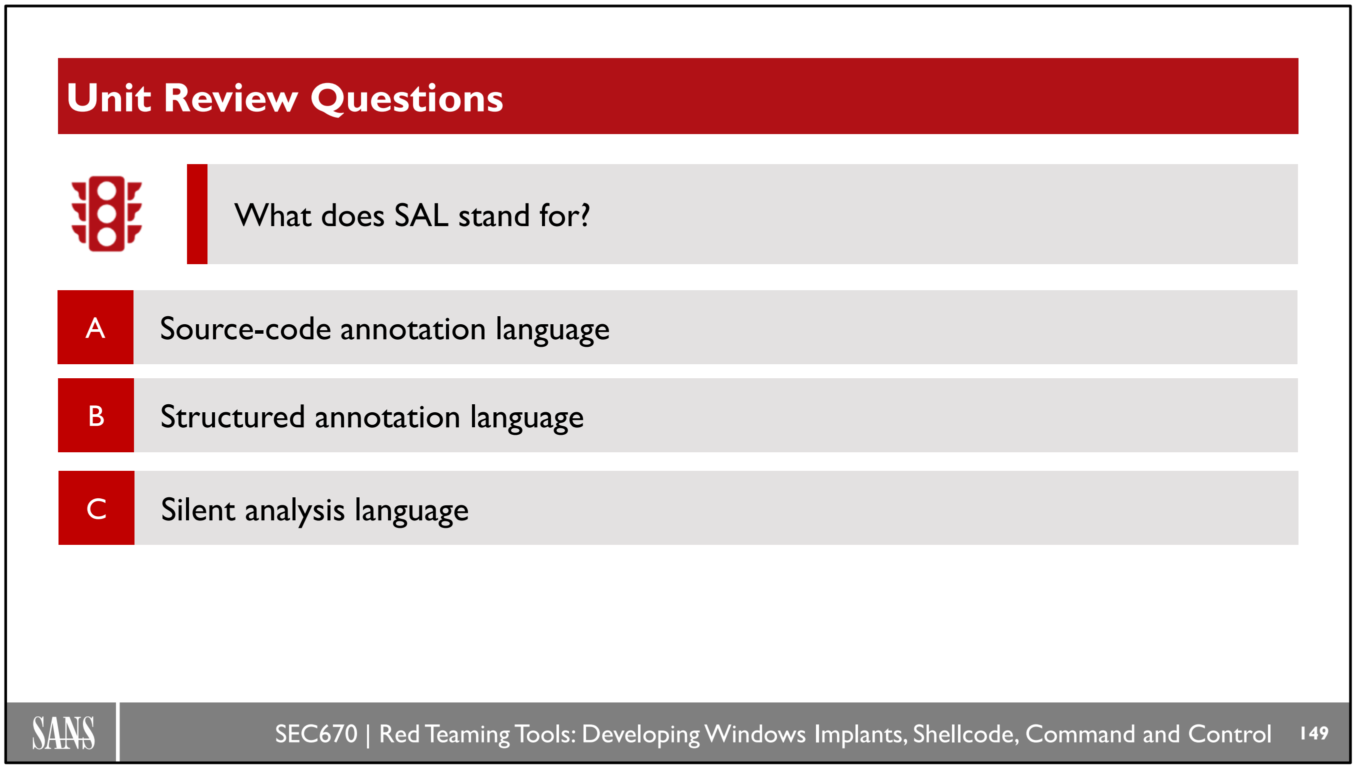
Header files can serve to be a great example of how to use certain annotations.

You can use PowerShell to search for specific annotations.

```
Select-String -Pattern "_When_\(" -Path . \*.h -CaseSensitive
```

Module Summary

- Discussed how SAL annotations can really assist with making code safer
- Learned some annotations can become complex very quickly
- Learned some annotations only apply for a function
- Covered how Visual Studio uses it for static code analysis



Unit Review Questions

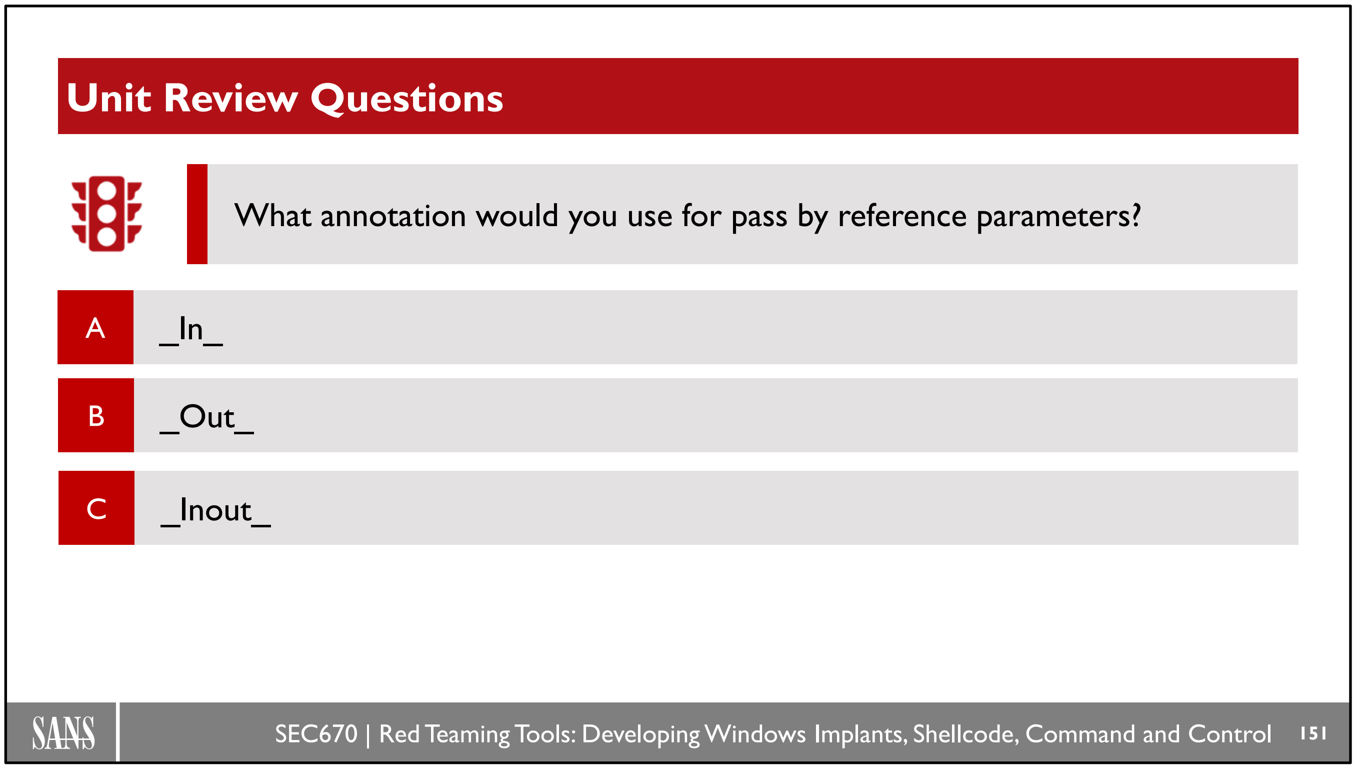
What does SAL stand for?

A	Source-code annotation language
B	Structured annotation language
C	Silent analysis language

Unit Review Answers

What does SAL stand for?

A	Source-code annotation language
B	Structured annotation language
C	Silent analysis language



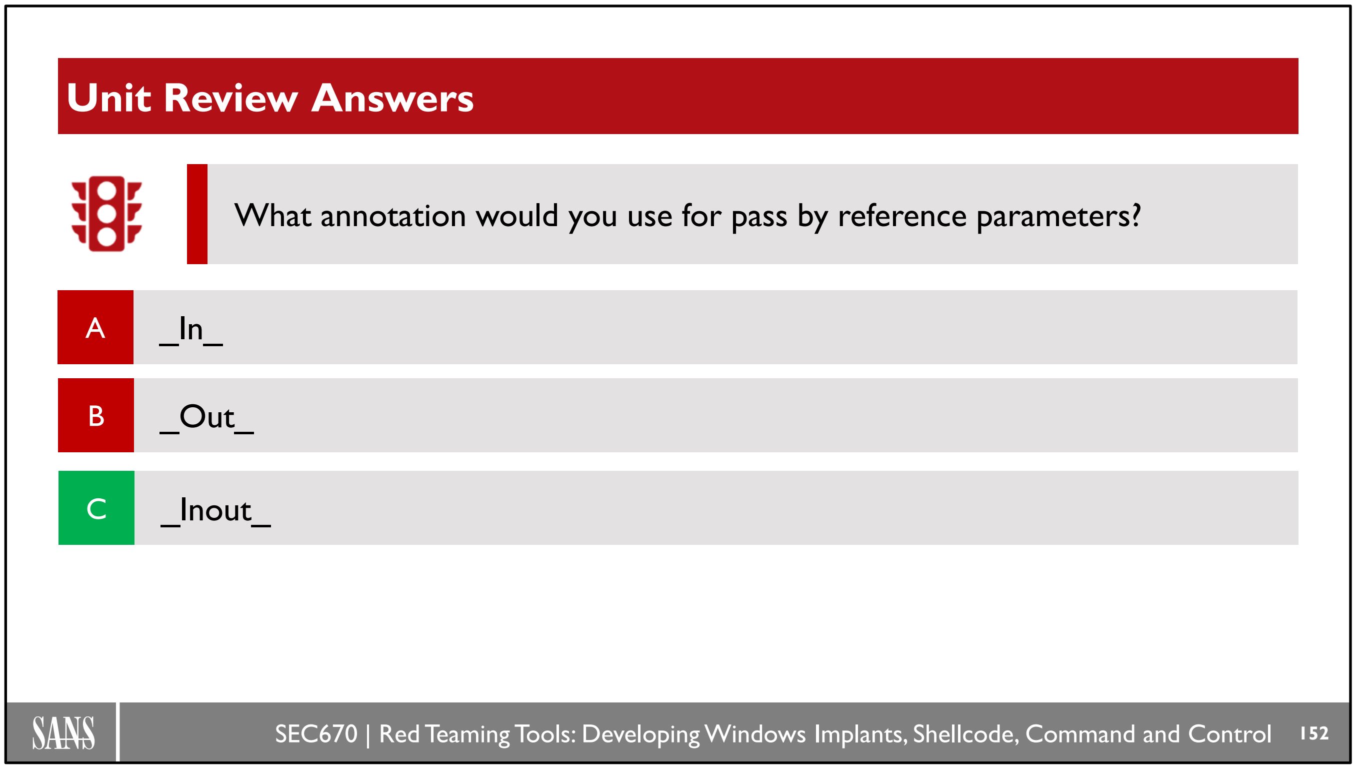
Unit Review Questions

What annotation would you use for pass by reference parameters?

A _In_

B _Out_

C _Inout_



Unit Review Answers

What annotation would you use for pass by reference parameters?

A _In_

B _Out_

[✓ C] _Inout_

Unit Review Questions

Using the _When_ annotation, make a pointer parameter become _Out_.

A	_When_(nullptr, _Out_)
B	_When_(_Out_, lpPointer == LPVOID())
C	_When_(lpPointer == nullptr, _Out_)

Unit Review Answers

Using the _When_ annotation, make a pointer parameter become _Out_.

- A _When_(nullptr, _Out_)
- B _When_(_Out_, lpPointer == LPVOID())
- [✓ C] _When_(lpPointer == nullptr, _Out_)

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 1

Windows API

The Windows Application Programming Interface

Provides nearly all functionality for the Windows OS to include userland and kernel land functionality. API names are very descriptive and can be very lengthy too. Heavily modified/updated APIs have "Ex" appended to the name.

APIs make creating shellcode more complex.

PrefixVerbTarget[Ex] / VerbTarget[Ex]

RegCreateKeyEx

Create APIs (1)

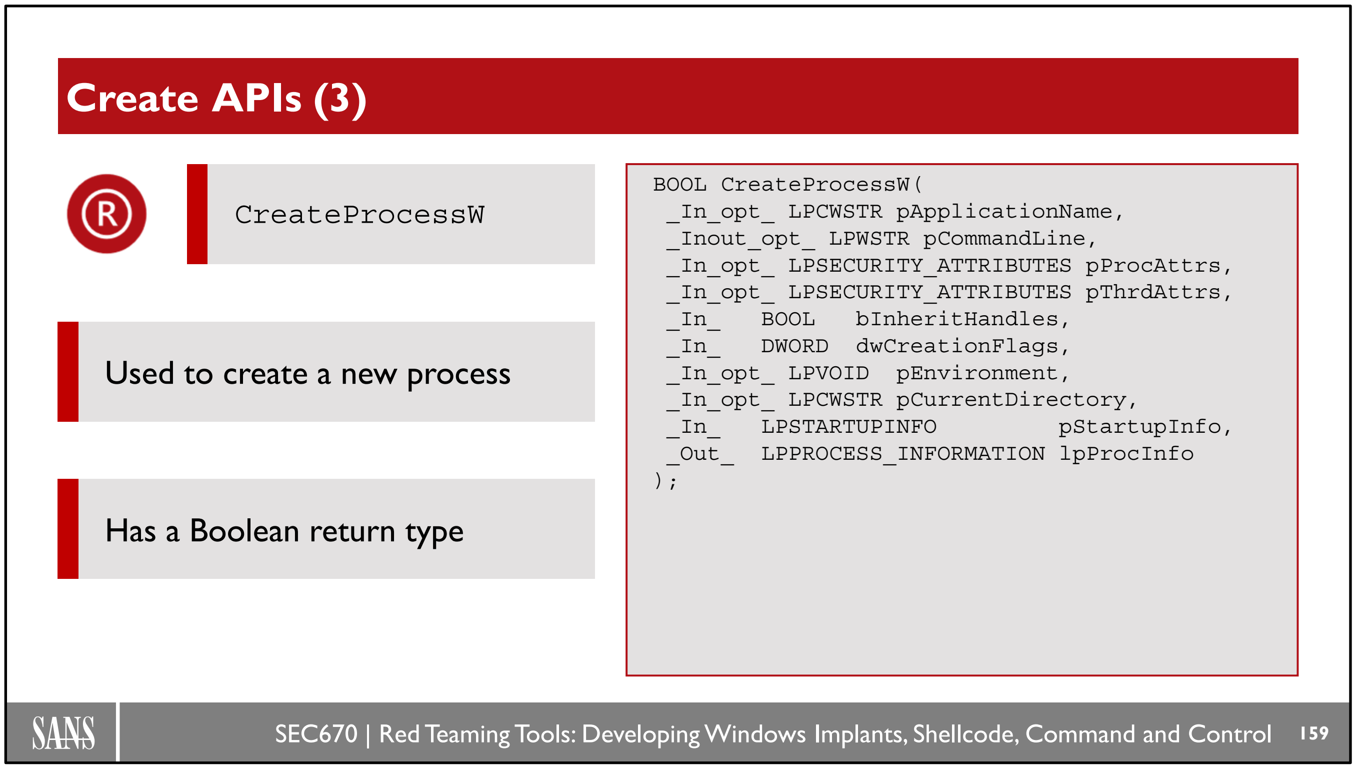
Objects and handles make the world go around.

There are at least 96 Windows APIs that start with "Create." Their main purpose is to help a user application create a system object. A Handle is typically returned that references the newly created object.

CreateProcess is one exception; its return value is of type BOOL

Create APIs (2)

- `CreateFile`
- `CreateFileMapping`
- `CreateDirectory`
- `CreateTimerQueue`
- `CreateMutex`
- `CreateEvent`
- `CreateThread`
- `CreateToolhelp32Snapshot`



Create APIs (3)

CreateProcessW

Used to create a new process

Has a Boolean return type

```
BOOL CreateProcessW(
 _In_opt_ LPCWSTR pApplicationName,
 _Inout_opt_ LPWSTR pCommandLine,
 _In_opt_ LPSECURITY_ATTRIBUTES pProcAttrs,
 _In_opt_ LPSECURITY_ATTRIBUTES pThrdAttrs,
 _In_   BOOL   bInheritHandles,
 _In_   DWORD  dwCreationFlags,
 _In_opt_ LPVOID  pEnvironment,
 _In_opt_ LPCWSTR pCurrentDirectory,
 _In_   LPSTARTUPINFO         pStartupInfo,
 _Out_  LPPROCESS_INFORMATION lpProcInfo
);
```

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 159

Just so you don't have to flip back several pages, here is the search order again in order.

1. Directory of the calling executable
2. Current directory of the process
3. System directory (this is returned by ***GetSystemDirectory***)
4. Windows directory (this is returned by ***GetWindowsDirectory***)
5. Directories found in %PATH%

A noticeable difference between the first two parameters is the type. *lpCommandLine* is of type LPWSTR, which is not a constant like for *lpApplicationName*, LPCWSTR. Constant are treated as read-only and because of this, the *CommandLine* must be available to modification by the function. In other words, the function will read and possibly write to the buffer that holds the image name. So, be sure to place the string in dynamic memory or on the stack, which should always be read/write.

lpProcessAttributes and *lpThreadAttributes* are the same type, which is a pointer to a SECURITY_ATTRIBUTES structure. NULL is typically passed here, but if you want the handle to have the inheritable attribute, then they cannot be NULL.

bInheritHandles is just a flag to indicate that inheritable handles will be inherited by a child process if one is created.

dwCreationFlags describe how the process is to be created. There are over a dozen flags that can be used in combination with others. One example is the CREATE_SUSPENDED flag that will place the primary thread in a suspended state that will not run until your code calls ***ResumeThread***. This technique will be explored later in the course.

lpEnvironment is a pointer to the environment block the new process should use. NULL is just fine here so the process can use the environment of the process that called it.

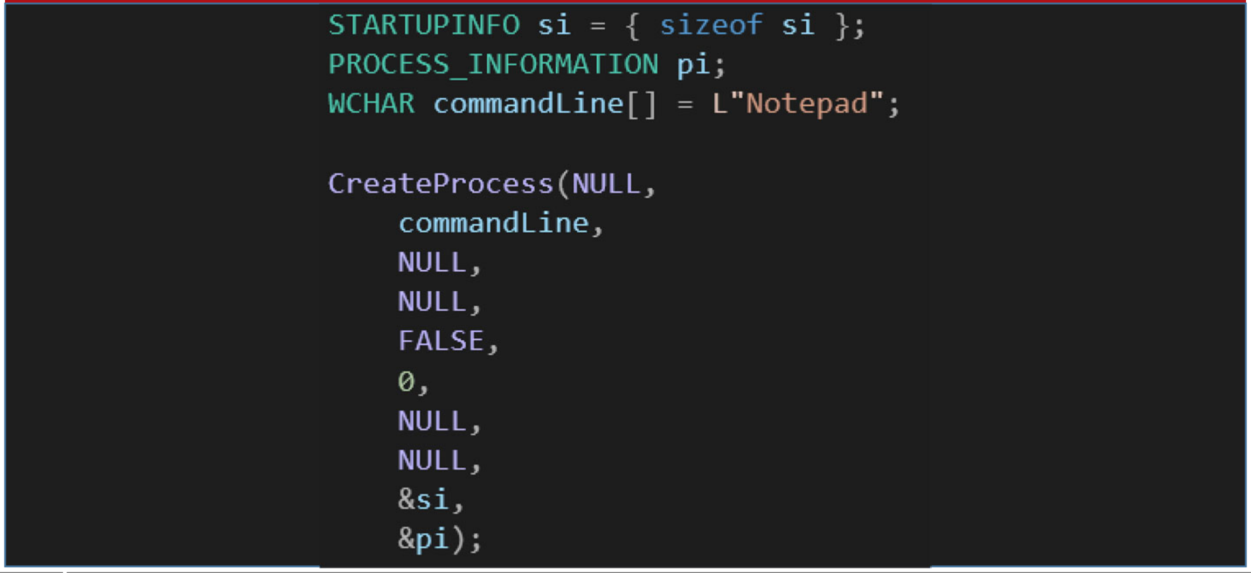
lpCurrentDirectory is the full path of the current directory and it can also be a UNC path. NULL is fine here too, and the new process will use the same drive and directory as the calling one.

lpStartupInfo is a pointer to one of two structures: STARTUPINFO or STARTUPINFOEX, which is just an extended version of the former.

lpProcessInformation is a pointer to a PROCESS_INFORMATION structure that will be filled out with information about the process once it's created. The handles that are in this structure need to be closed once no longer needed by your program, CloseHandle.

As mentioned previously, the function can return before the primary thread, or process, has had a chance to fully initialize itself. So, be sure to take this into account when you are developing your programs.

Example: CreateProcess



```
STARTUPINFO si = { sizeof si };
PROCESS_INFORMATION pi;
WCHAR commandLine[] = L"Notepad";

CreateProcess(NULL,
    commandLine,
    NULL,
    NULL,
    FALSE,
    0,
    NULL,
    NULL,
    &si,
    &pi);
```

Create APIs (4)

PROCESS_INFORMATION

Holds information about the new process and its primary thread

```
typedef struct
_PROCESS_INFORMATION {
 HANDLE hProcess;
 HANDLE hThread;
 DWORD  dwProcessId;
 DWORD  dwThreadId;
} PROCESS_INFORMATION,
*PPROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```

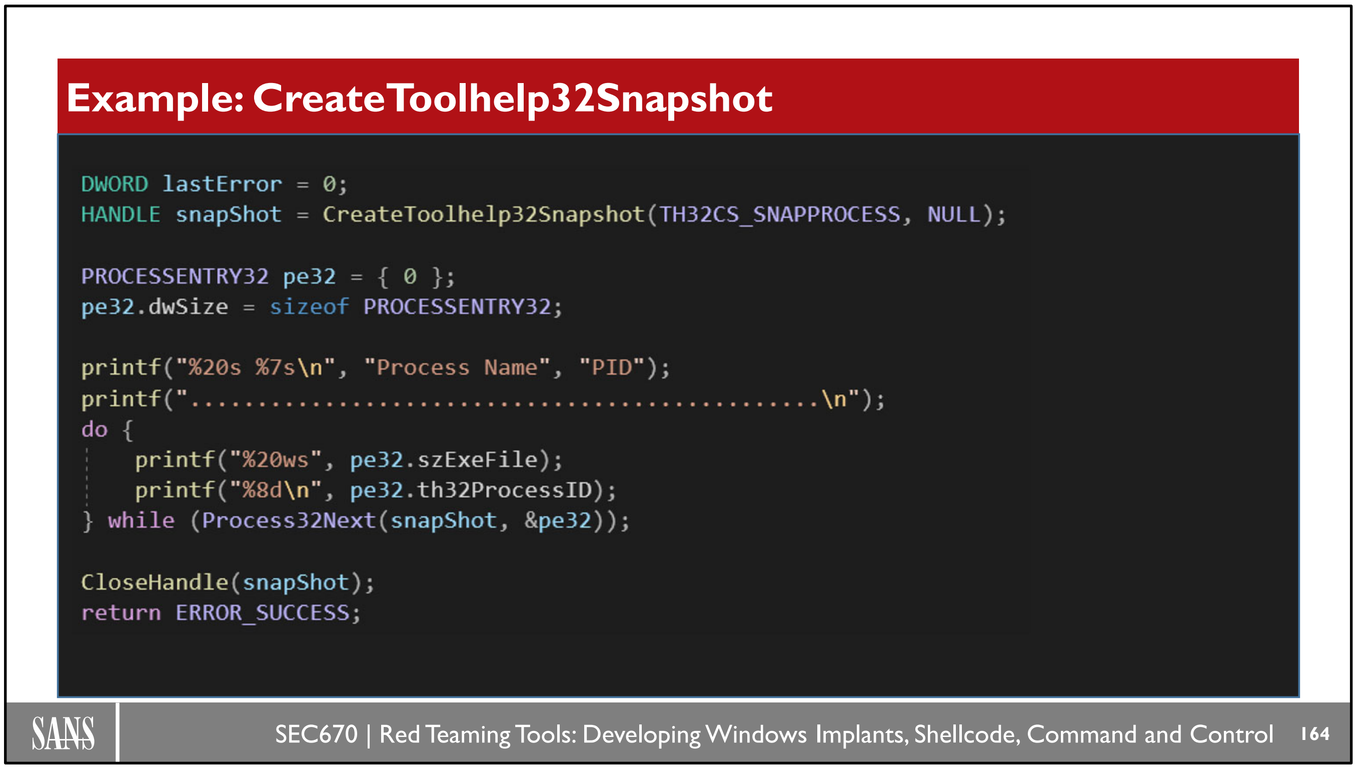
Create APIs (5)

`CreateToolhelp32Snapshot`

Used to create a snapshot of a process

Also, can be used for heaps, threads, and loaded modules

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

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

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



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 due to size limitations on the slide. Regardless, the main points are represented starting with the call to the ***CreateToolhelp32Snapshot*** API on the second line. We are only interested in capturing processes in the snapshot and we are not specifying a process ID as indicated by NULL (0). The function returns a handle value so that is saved off in the snapShot variable. In the full code, this would be checked against **INVALID_HANDLE_VALUE** to ensure the snapshot was successfully created. 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 is the do while loop. What is not shown is a check for the function call ***Process32First***(snapShot, &pe32) to make sure that the first process in the snapshot can be retrieved. After that succeeds the loop can be entered. The meat of the loop is simply printing out the name of the executable and the corresponding process ID. The exit condition of the loop is met when the ***Process32Next*** API 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.

Windows Objects (1)

System resources represented as data structures in system address space

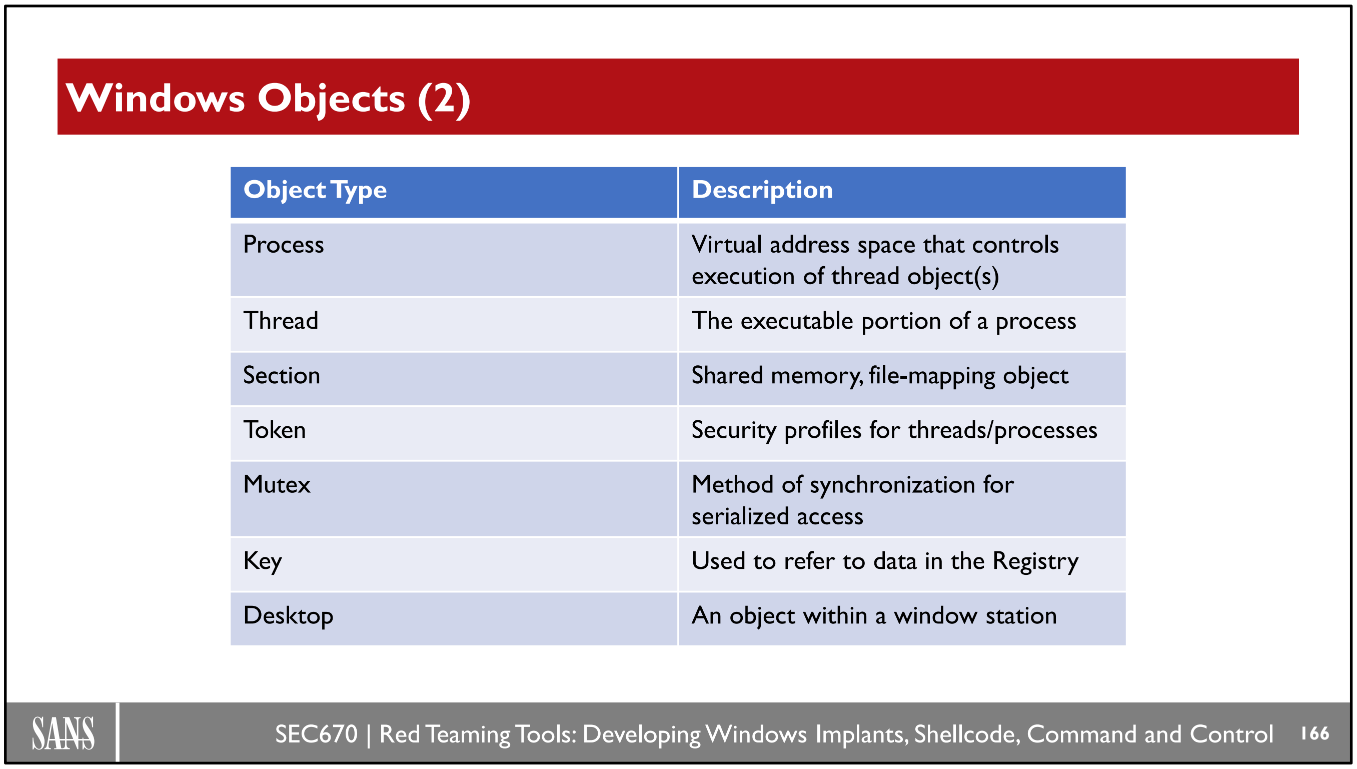
The Windows kernel does the hard work creating an object often at the request of a user application. There are over 4,000 object types that the Windows executive implements, some of which are not accessible via Windows APIs.

Files, images, threads, registry keys, and processes are several object types.

Windows Objects (1)
Objects in Windows are simply the system's way of representing system resources via a structured method. System resources should not be directly accessible by anyone, especially anyone from user space, thus secure access to them is maintained and enforced. To help the system with creating, deleting, and managing objects, there is the object manager, which is an executive module. When a user mode application calls a function like ***CreateFile*** or ***CreateThread***, a request is actually being made to create an object, or a system resource that represents that file or thread. The object manager gives the system a common interface for utilizing those created system resources. Objects should not be destroyed until all processes are done using it, and the object manager helps with that by providing rules that determine how long objects are retained.

There are officially three kinds of objects: kernel objects, executive objects, and user objects. Various executive components like the memory manger or the I/O manager will implement executive objects. The kernel objects are implemented by none other than the kernel itself, and as such, they are not available to user mode applications, only the executive.

Reference:
Mark Russinovich, David A. Solomon, and Alex Ionescu, *Windows® Internals, Part 1*, *6th Edition* (Washington, Microsoft Press, 2012), chap. 3.

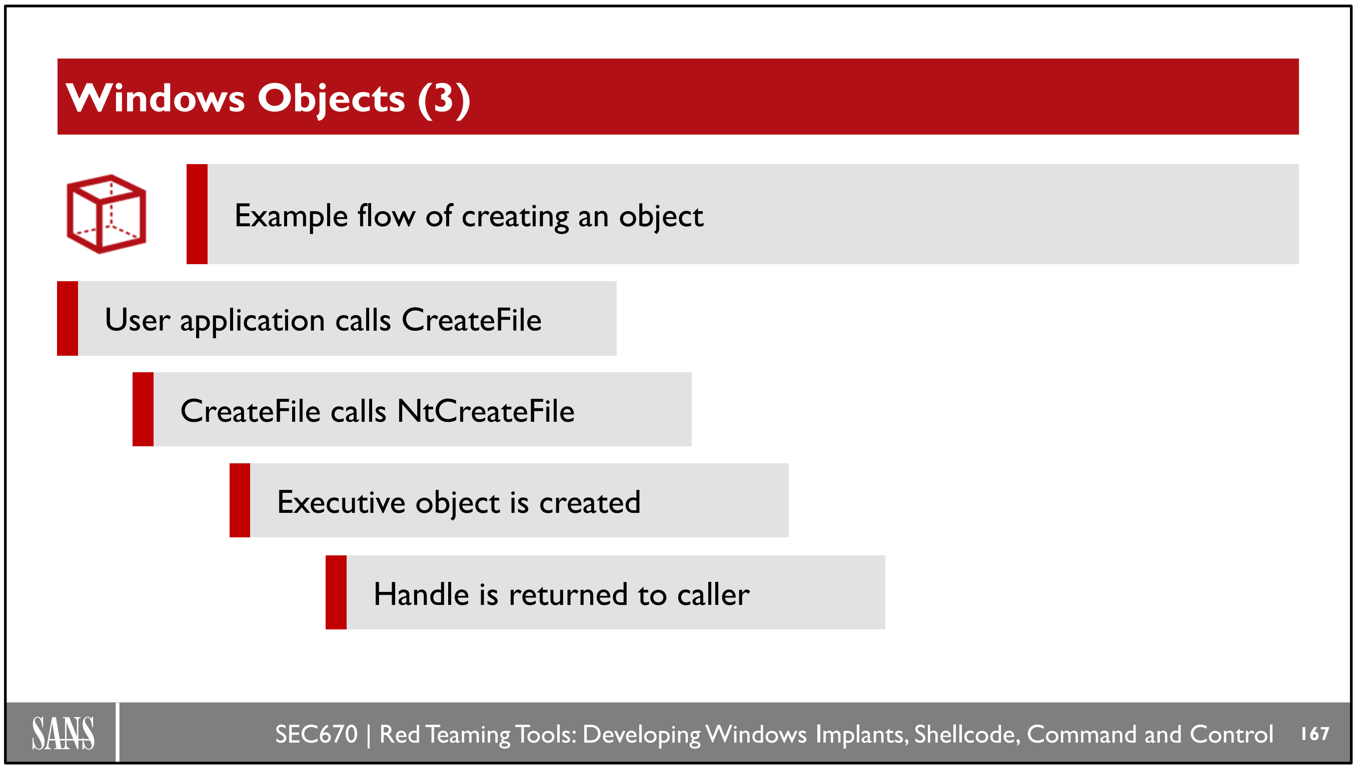


Windows Objects (2)

Object Type	Description
Process	Virtual address space that controls execution of thread object(s)
Thread	The executable portion of a process
Section	Shared memory, file-mapping object
Token	Security profiles for threads/processes
Mutex	Method of synchronization for serialized access
Key	Used to refer to data in the Registry
Desktop	An object within a window station

Windows Objects (2)
There are many types of objects on a Windows system, and the types listed on the slide are a small subset of the types of objects. The full listing can be found by running the WinObj utility from Sysinternals and browsing the ObjectTypes folder. For example, there are kernel objects, such as device and driver objects, that are utilized when developing kernel drivers. Also listed are process, job, and thread object types that are typically created at the request of user mode applications. Taking an example from the slide, the process object type represents a virtually contained address space for the execution of thread objects.

Reference:
Mark Russinovich, David A. Solomon, and Alex Ionescu, *Windows® Internals, Part 1*, *6th Edition* (Washington, Microsoft Press, 2012), chap. 3.

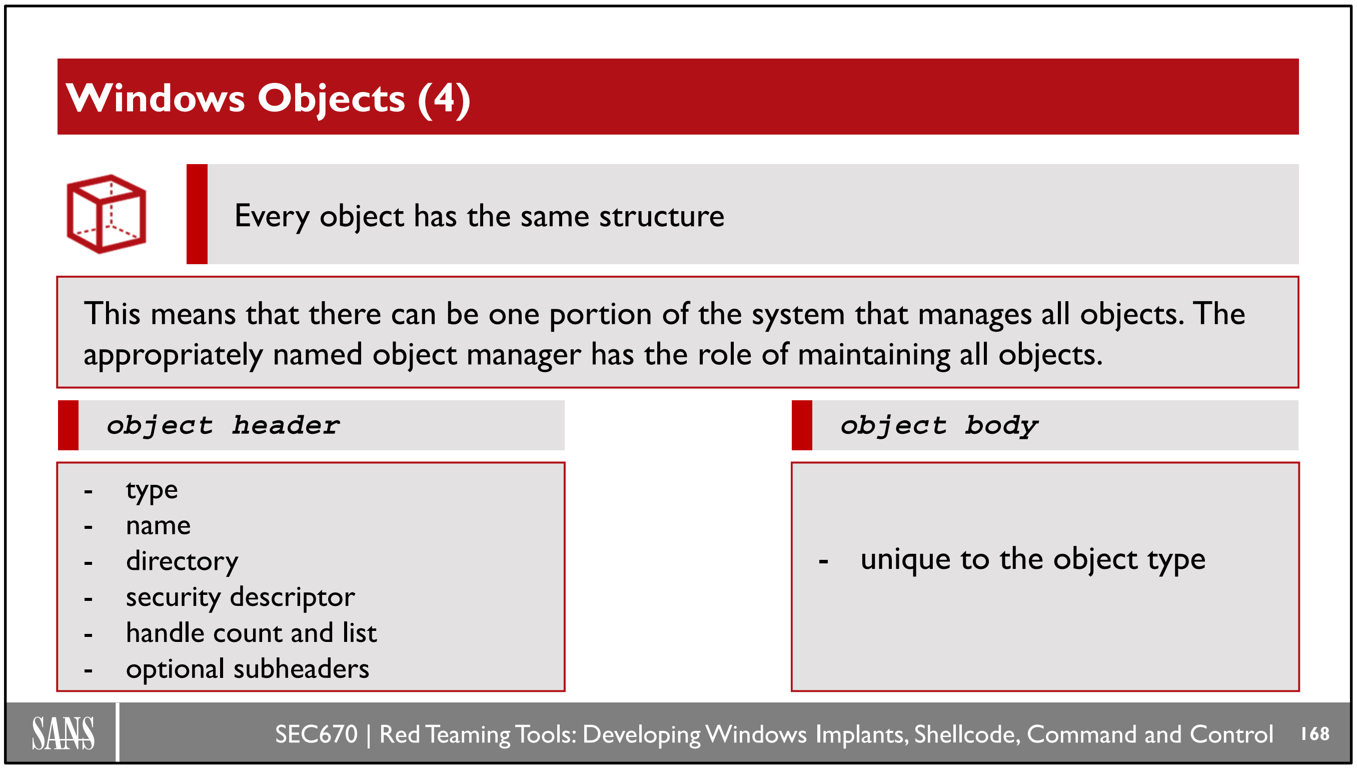


Windows Objects (3)

Example flow of creating an object

- User application calls CreateFile
  - CreateFile calls NtCreateFile
    - Executive object is created
      - Handle is returned to caller

Windows Objects (3)
There are many reasons why a user mode application might want to create a file. Perhaps the program is creating an error log file to maintain a record of any errors that are encountered during its execution time frame. The application will need an object representing the file that it requested the kernel create for it. The first thing that happens is the application must make a call to the ***CreateFile*** API. ***CreateFile*** is implemented from Kernelbase.dll as one part of the Windows subsystem and once everything checks out, ***NtCreateFile*** will be called. The kernel will eventually be invoked via a system call, and it will create the file object, specifically an executive file object, and return a handle to that object. The handle should also be placed as an entry in the process' handle table.



Windows Objects (4)

Every object has the same structure

This means that there can be one portion of the system that manages all objects. The appropriately named object manager has the role of maintaining all objects.

`object header`

- type
- name
- directory
- security descriptor
- handle count and list
- optional subheaders

`object body`

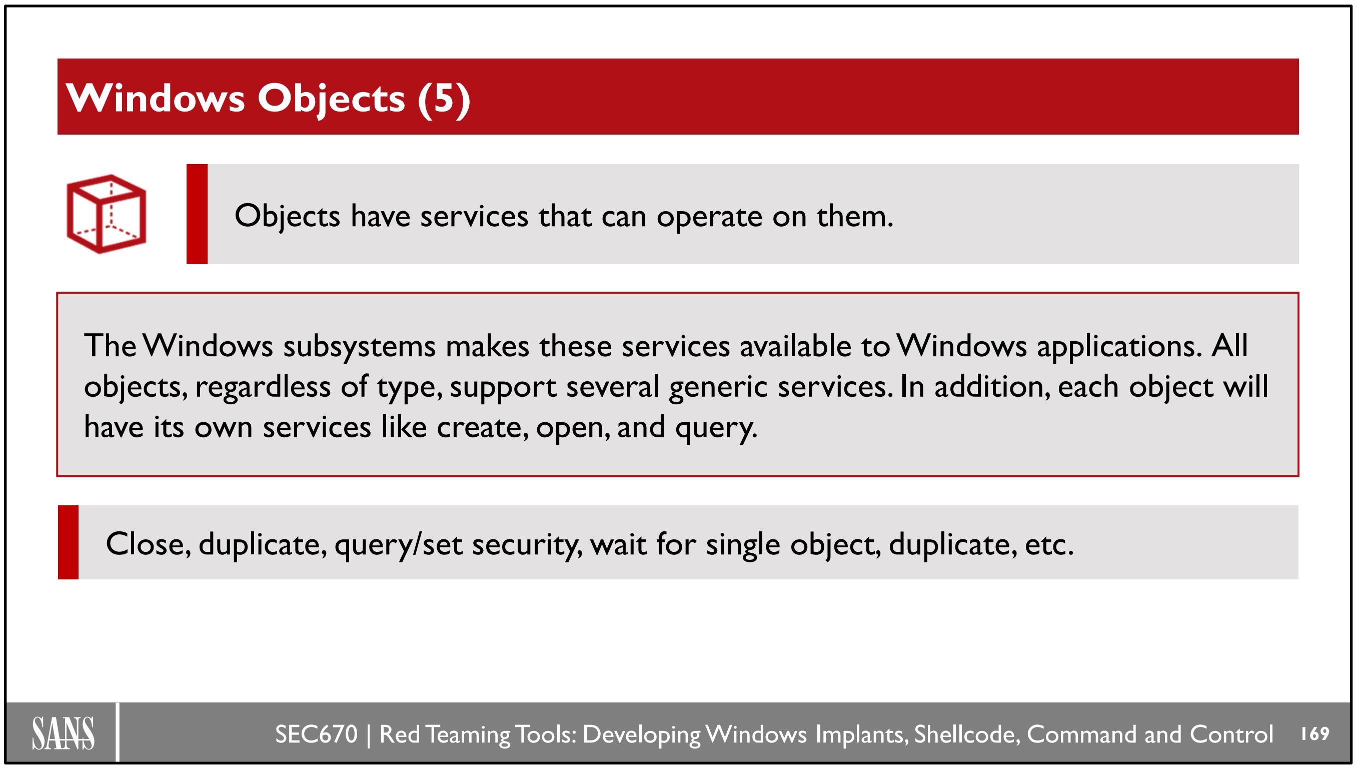
- unique to the object type

Windows Objects (4)
The object manager can perform several tasks, such as following:
- Create objects and validate that a process has the rights to use that object.
- Create the object handles and return them to the caller.
- Create duplicates of existing handles.
- Close handles to objects.

Each object is going to have a header and a body. The object header is controlled via the object manager and has a static size. The object body is controlled via the owning executive components and the size will vary based on the object type.

The structure of the object's header has more information than what fits in the slide, so the important portions are noted. Other items to note are the handle count (list) and type. The system needs to keep count of how many handles are currently open to the object, so it knows when to destroy it. The type is simply an index of object types found in a table named *ObTypeIndexTable*. Although each object header is structured the same way, they will undoubtedly contain different information with each instance of that object. The object name and security descriptors would be the most noticeable differences. The handle list is simply a list of open handles to that object. There could be several processes that are using that object via a handle.

Unlike the object header, the body of an object must be unique because there are several object types. However, objects that are of the same type will have the same object body format. If you wanted to peek into the internals of the object headers and type objects, they can be viewed using a kernel debugger.



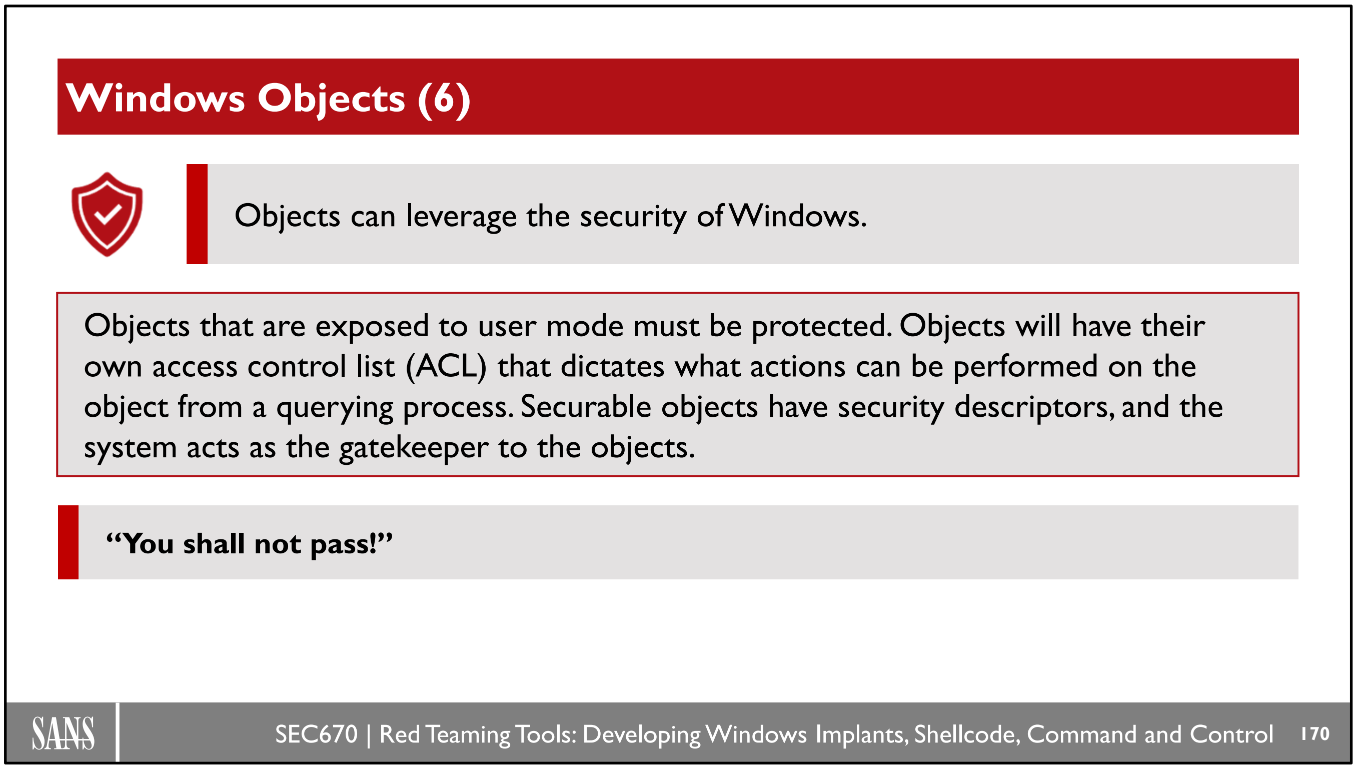
Windows Objects (5)

Objects have services that can operate on them.

The Windows subsystems makes these services available to Windows applications. All objects, regardless of type, support several generic services. In addition, each object will have its own services like create, open, and query.

Close, duplicate, query/set security, wait for single object, duplicate, etc.

Windows Objects (5)
With the standardization of object headers and sub headers, the object manager can provide applications with a small number of generic services. The generic services, being applied to each object regardless of type, can be executed with the corresponding APIs such as ***CloseHandle***. In addition, each object will have its own services like create, open, and query. We can go with one example for the creation of a file. We saw the overall flow of the process from a few slides back but diving into it a bit more here, the I/O system implements a service that allows an application to create a file object. The same thing applies for the creation of process objects. The process manager, another executive component, implements its create process service that allows an application to create a process object. Even though objects might be similar to each other, the routines used to create them are vastly different. This makes sense because the process of creating a file is vastly different from the process of creating a process. The object headers would be structured the same, but definitely not the object bodies. Furthermore, it would not make sense to suspend or terminate file objects the same way that you can terminate thread and process objects.



Windows Objects (6)

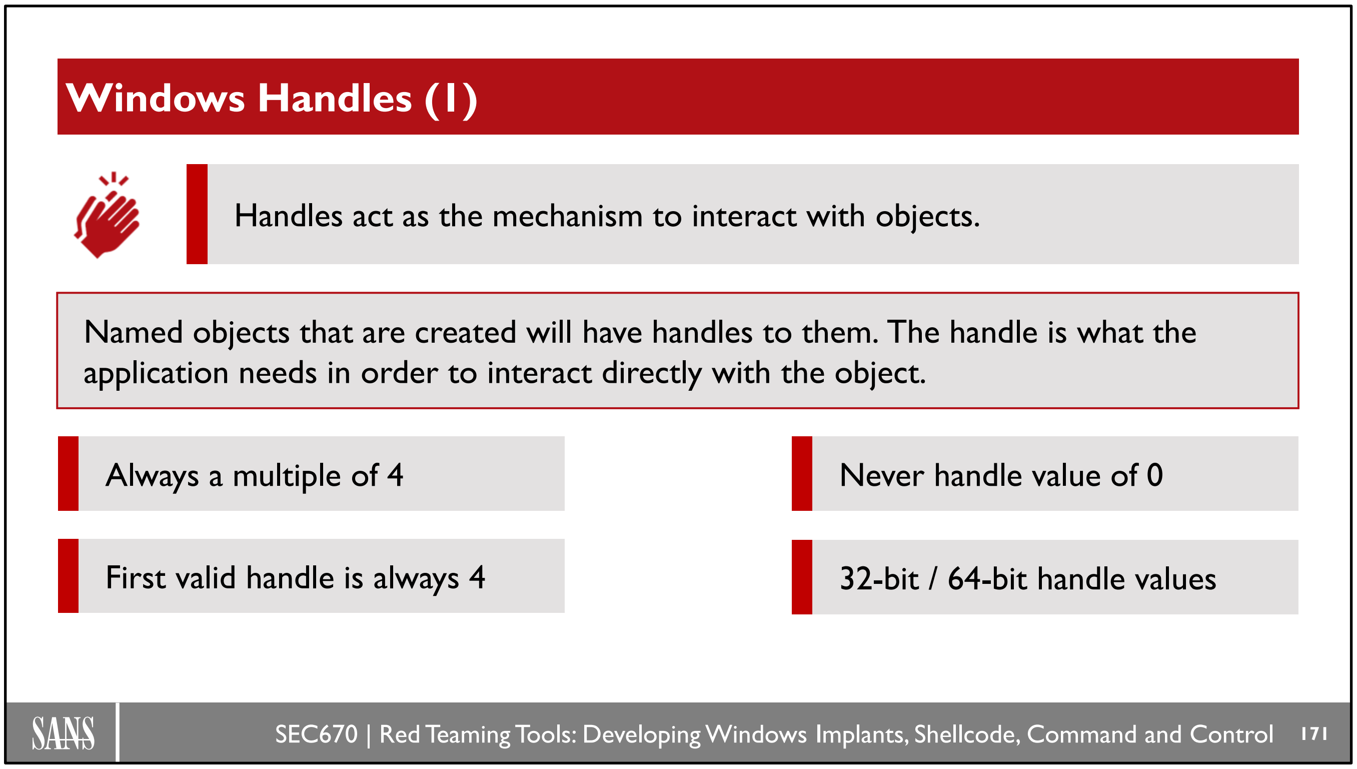
Objects can leverage the security of Windows.

Objects that are exposed to user mode must be protected. Objects will have their own access control list (ACL) that dictates what actions can be performed on the object from a querying process. Securable objects have security descriptors, and the system acts as the gatekeeper to the objects.

"You shall not pass!"

Windows Objects (6)
Objects must be secured or protected from malicious abuse or unauthorized access. Whenever an object is exposed directly to the user, it must be protected. Objects that are not directly exposed do not needed protections like kernel objects (driver objects). In a similar fashion to how files are protected on the file system, objects are protected too, but via security descriptors. The object manager acts as the gatekeeper to objects and controls who accesses them. When a process wishes to obtain a handle to an object, say for read and write access, the object manager steps in to see what is happening. The object manager, with the help of the kernel, determines what access rights will be granted to that object. The desired accesses are requested by the user mode application via flags like GENERIC_READ or GENERIC_WRITE.

A process' handles to objects can be viewed using Process Explorer from Sysinternals. Once a process is selected, the handles can be observed, but if nothing is showing, be sure to unhide the lower pane from the toolbar. A handle, or object, can be selected and the properties can be observed after right-clicking on it.



Windows Handles (1)

Handles act as the mechanism to interact with objects.

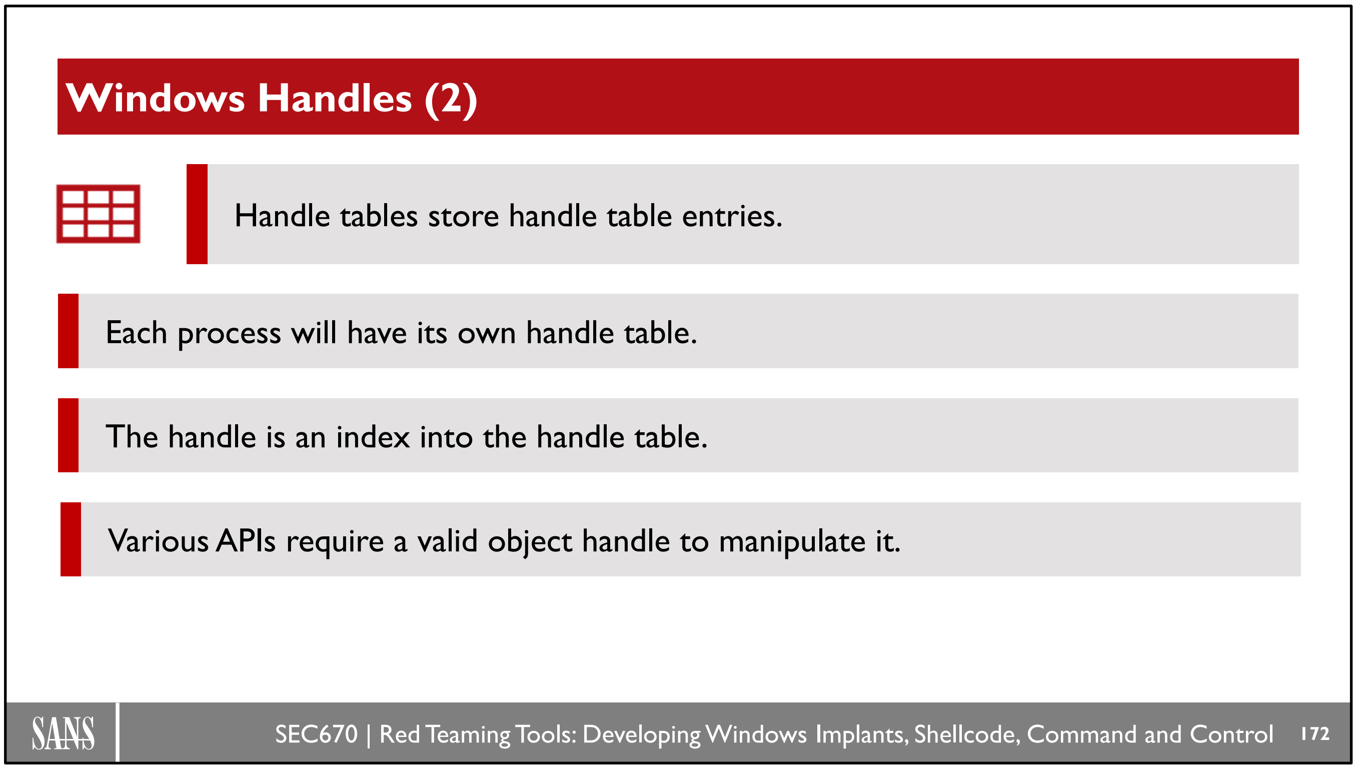
Named objects that are created will have handles to them. The handle is what the application needs in order to interact directly with the object.

- Always a multiple of 4
- Never handle value of 0
- First valid handle is always 4
- 32-bit / 64-bit handle values

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 171

Windows Handles (1)
Handles are the mechanism in place that allow a user mode application to interact with an object. When an object is created with a name, a named object, the object manager will provide the requester with a handle to it. All handles are the same regardless of the type of object they have a handle against. For example, a handle to a process is the same as a handle to a registry key or thread or file. This makes the job of reference counting easier for the object manager.

When it comes to handle values, they will always be some multiple of 4 and because of this, one thing you will never see is a handle value of 0. Therefore, the first valid handle value could only ever be 4. Handles will either be 32-bit values on 32-bit Windows and 64-bit values on 64-bit Windows. Throughout the remainder of this course, you will become extremely familiar with handles and how to pass them around to other functions that require them.



Windows Handles (2)

Handle tables store handle table entries.

Each process will have its own handle table.

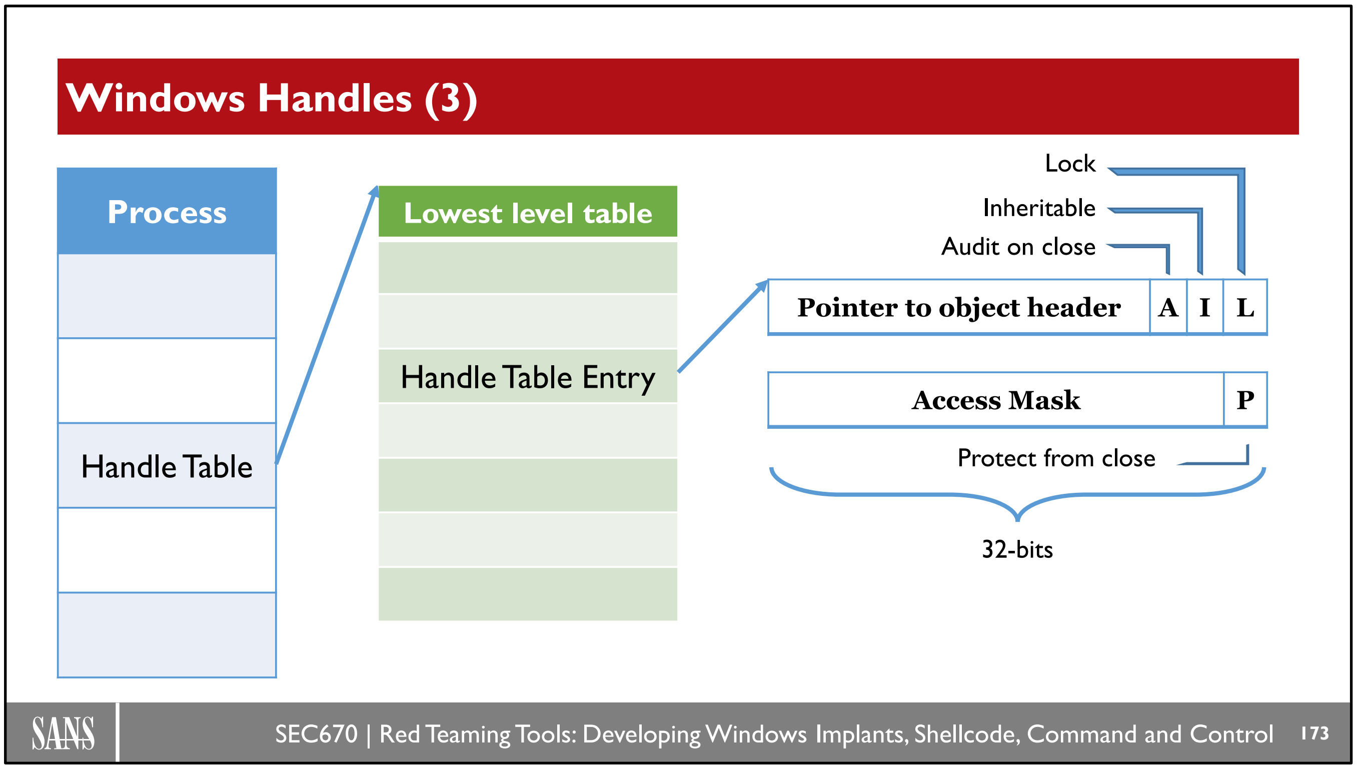
The handle is an index into the handle table.

Various APIs require a valid object handle to manipulate it.

Windows Handles (2)
As mentioned previously, handles are the direct result of the creation of named objects. There are several APIs that can create objects and handles (they were also mentioned earlier), but one such API is ***CreateProcess***. When a process is created, the system will create a handle table unique to that process. The table stores handles that reference various objects like threads or other objects that threads could open handles against. As far as the application is concerned, the handle table is opaque.

Once the correct handle table entry is located, the pointer to the object can be followed. Along with the pointer to the object, the access is also stored to protect the object. If a process or thread attempted a write operation to an object, but the handle was opened with only read access, the operation would be happily denied by the object manager. It is best to open a handle with the minimum access required to do the job. As an example, calling `OpenProcess(PROCESS_ALL_ACCESS, TRUE, GetCurrentProcessId())` would return a handle with all access to the current process. Even worse, the handle can be inherited by any child process.

No matter how hard you might try, handles cannot be directly created with an API, but they can be closed (***CloseHandle***), compared (***CompareObjectHandles***), or duplicated (***DuplicateHandle***). Also, the information about a handle can be retrieved via ***GetHandleInformation***. Certain handle properties can also be modified via ***SetHandleInformation***.

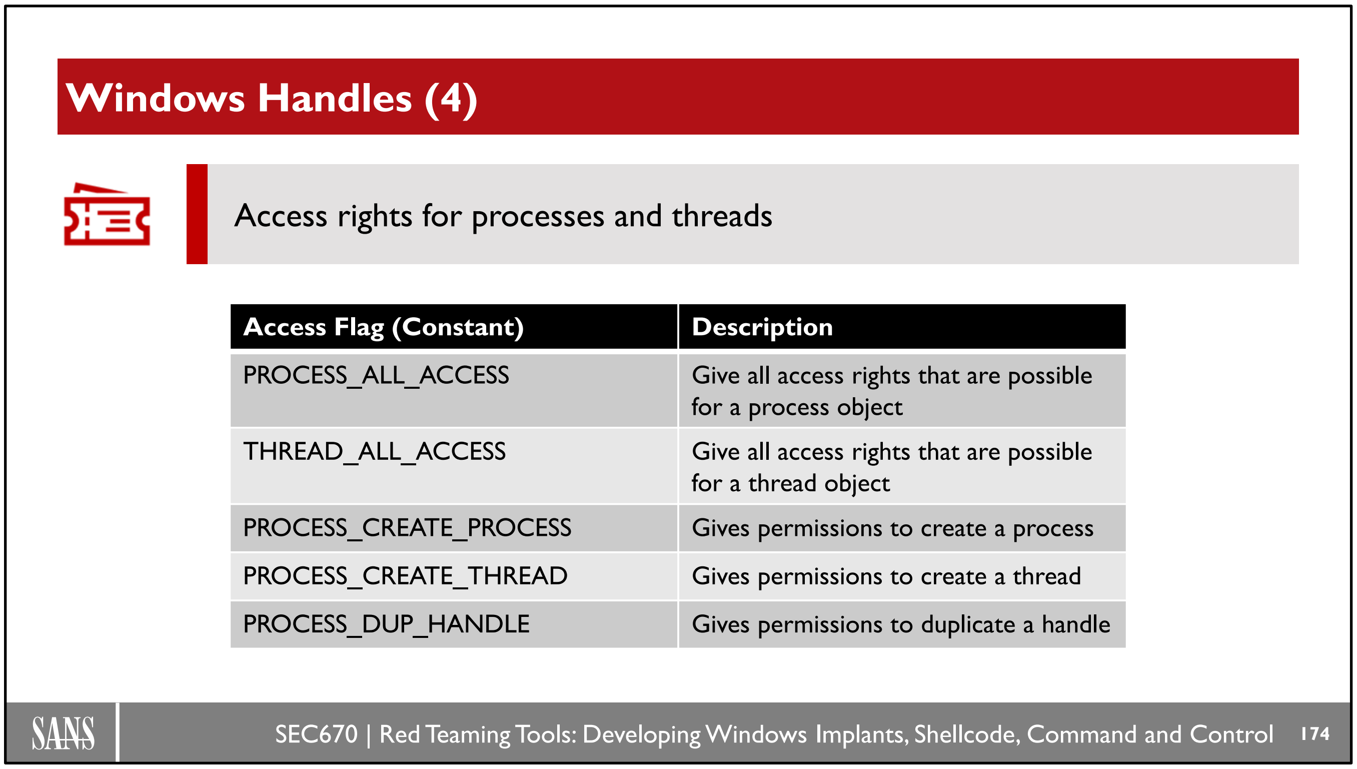


Windows Handles (3)

Windows Handles (3)
Somewhere in the process' virtual address space will be a pointer to its handle table. Just like how there are multiple tables involved with the translation of virtual addresses to physical addresses, there are multiple tables for handles. This is how a process can have a large number of handles, upwards of 16,000,000. Not all of the three tables are shown on this slide due to space limitations, but the lowest level table is the only one created right when a new process is initialized. The other table levels are made when there is a need.

The number of entries is dependent on the architecture being used. As an example, x86 machines have a page size of 4096 bytes. The size of a handle table entry is 8 bytes (32-bit pointer). Dividing the size of a page by the size of the table entry would yield 512 - 1 = 511. Each handle table entry is a structure and on x86 systems, both members of the structure are 32-bits in size. For x64, the handle table entry is 12 bytes in size, and it will have a 64-bit pointer followed by a 32-bit access mask.

The structure has a pointer to the object's header along with four flags: lock, inheritable, audit on close, and protect from close. The lock bit indicates if the entry is currently in use. The inheritance flag indicates that child processes will inherit the handle and be stored in its process handle table. The third flag, audit on close, indicates if an audit message should be created when the object is closed. The last flag, protect from close, indicates if the caller should be allowed to close the handle. The protect from close flag can be set with the ***NtSetInformationObject*** call.

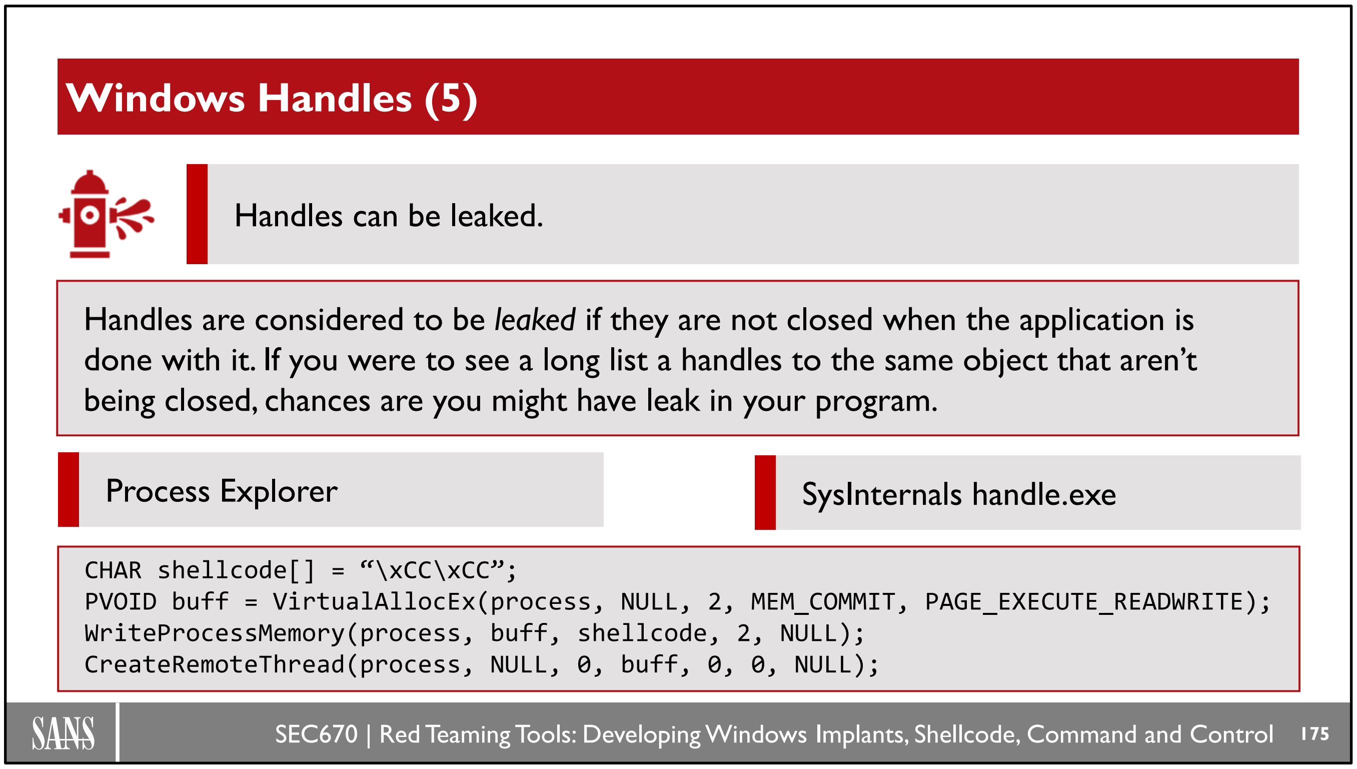


Windows Handles (4)

Access rights for processes and threads

Access Flag (Constant)	Description
PROCESS_ALL_ACCESS	Give all access rights that are possible for a process object
THREAD_ALL_ACCESS	Give all access rights that are possible for a thread object
PROCESS_CREATE_PROCESS	Gives permissions to create a process
PROCESS_CREATE_THREAD	Gives permissions to create a thread
PROCESS_DUP_HANDLE	Gives permissions to duplicate a handle

Windows Handles (4)
Handles can have various permissions and some of them can be great for abuse when the right conditions are set. The table of access rights on the slide is not an exhaustive list by any means, but some were noteworthy to mention. Handles that have ***_ALL_ACCESS** are needed for a process or thread to act upon itself via a pseudo-handle to itself. For example, ***GetCurrentProcess*** does not return a real handle, but rather a pseudo-handle with a value of (HANDLE)-1. ***GetCurrentThread*** also returns a pseudo-handle to itself, (HANDLE)-2. The kernel does some checks and will grant the appropriate access, **PROCESS_ALL_ACCESS** or **THREAD_ALL_ACCESS**. The ***_CREATE_*** access flags allow you to create a process or a thread. The **PROCESS_DUP_HANDLE** will allow you to duplicate a handle. A common example of duplicating handles is duplicating STD handles into a child process. Say, the parent process is a service that cannot print anything to console handles like STDOUT so it can create a child process and duplicate those handles into it.



Windows Handles (5)

Handles can be leaked.

Handles are considered to be *leaked* if they are not closed when the application is done with it. If you were to see a long list a handles to the same object that aren't being closed, chances are you might have leak in your program.

Process Explorer

SysInternals handle.exe

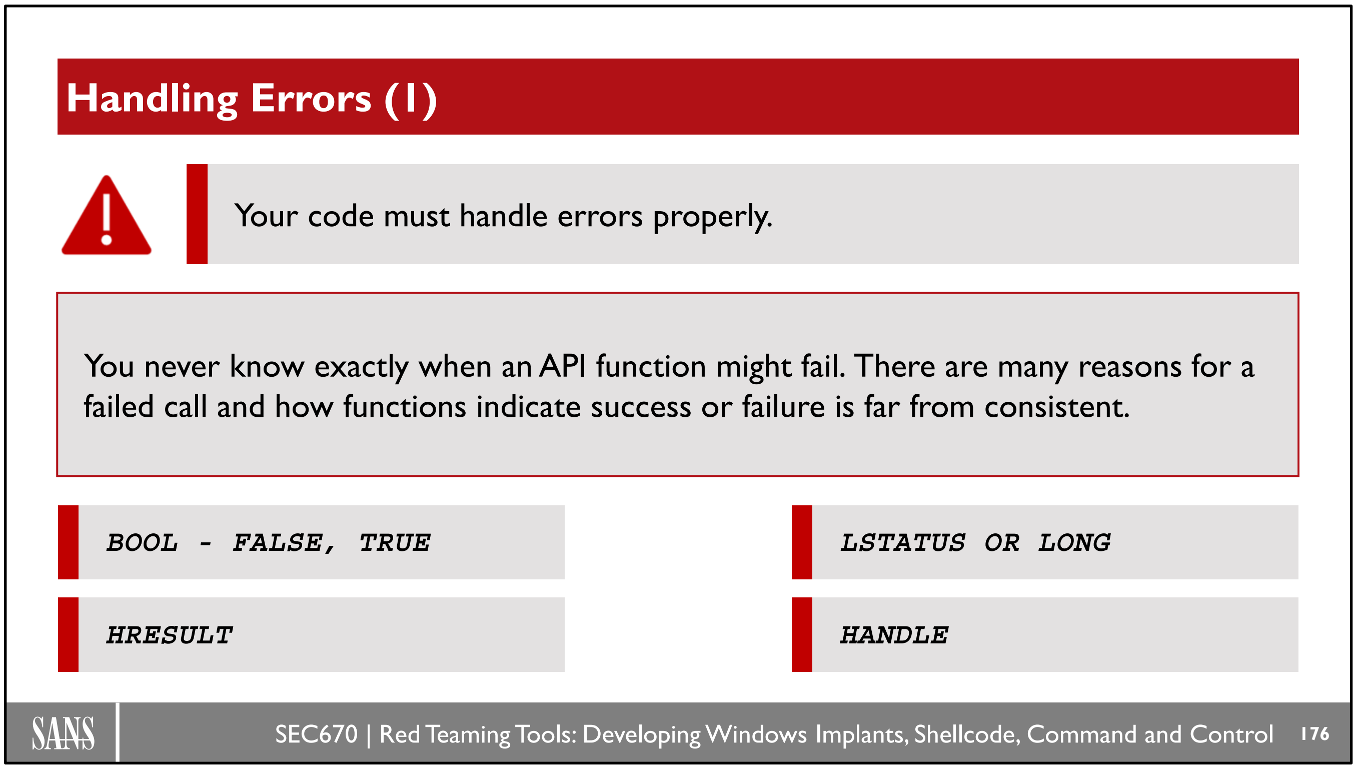
```
CHAR shellcode[] = "\xCC\xCC";
PVOID buff = VirtualAllocEx(process, NULL, 2, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(process, buff, shellcode, 2, NULL);
CreateRemoteThread(process, NULL, 0, buff, 0, 0, NULL);
```

SANS SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 175

Windows Handles (5)
When a developer is not careful to promptly close handles when they are no longer needed, they are considered to be leaked handles. There are various tools out there that let you view open handles and Process Explorer is one such tool. If you notice a large list of handles to the same object that have not been closed, then chances are pretty high that you are leaking handles. It is important to identify and correct your own leaked handles before someone like James Forshaw does and further proceeds to create a novel way to exploit it.

The highest access that could come with a handle would be specified with a flag like **PROCESS_ALL_ACCESS** or **THREAD_ALL_ACCESS**. If you created a SYSTEM level service or anything higher than a standard user application, and you called ***OpenProcess(*****PROCESS_ALL_ACCESS, TRUE, *GetCurrentProcessId())***, you just created a possible vulnerability to be exploited. The all access will literally give you an all-access pass to do pretty much whatever you want in that process' address space; you could create your own thread and execute shellcode!

There is also a command-line tool from SysInternals called "handle.exe" that shows the same information. You can run the handle utility on a single process and whatever open handles it has will be printed to the terminal window. It is also not that difficult to create your own version of the Sysinternals handles.exe utility.



Handling Errors (1)

Your code must handle errors properly.

You never know exactly when an API function might fail. There are many reasons for a failed call and how functions indicate success or failure is far from consistent.

BOOL - FALSE, TRUE

LSTATUS OR LONG

HRESULT

HANDLE

SANS SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 176

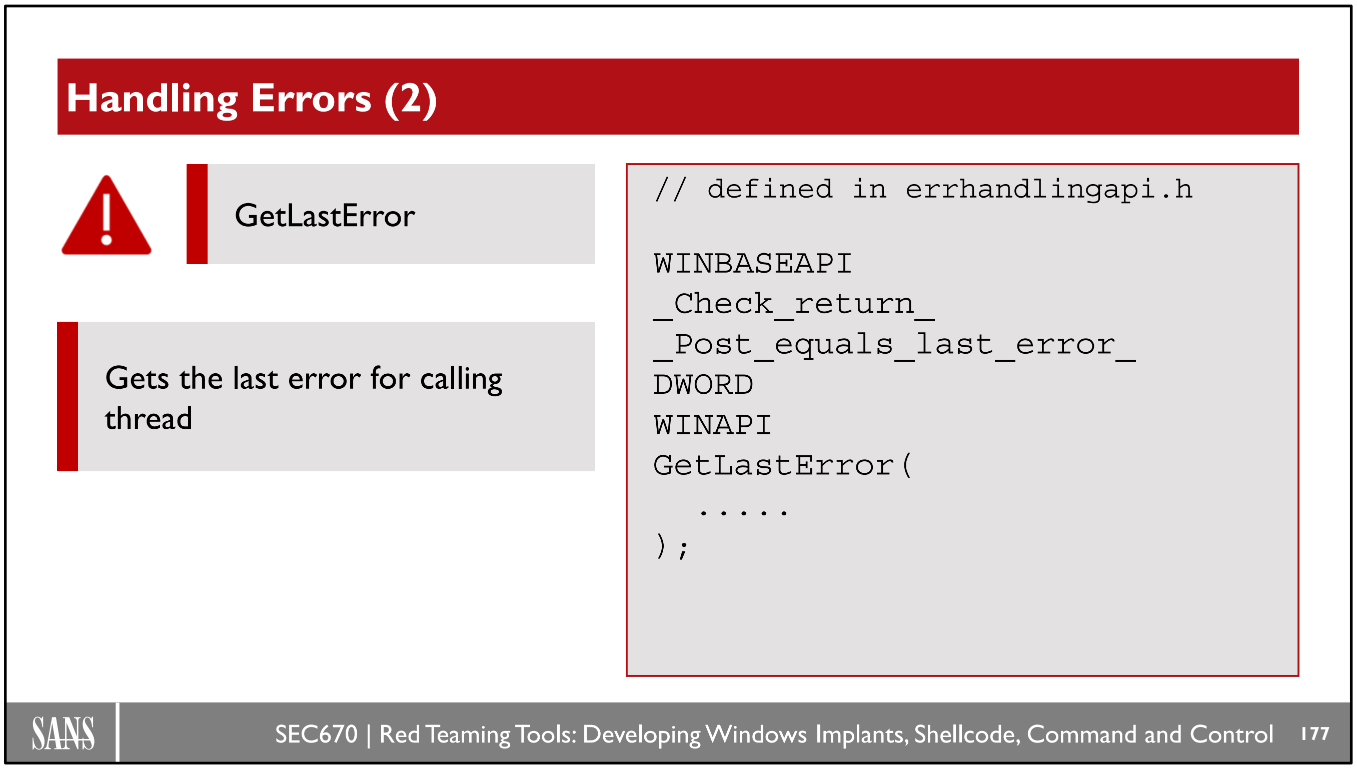
Handling Errors (1)
API functions can fail at any point in their execution and for a variety of reasons. Perhaps you messed up and passed in an incorrect parameter, or maybe a buffer you needed to have allocated to receive from an OUT parameter was not large enough. Something could also happen internally with the system that would cause your function to fail. The more common failures are either from access permissions or improper use of the function.

Here are some common return types for functions and how they could indicate failure. BOOL functions can indicate failure by having a zero (0) or FALSE be returned. The opposite, not zero, (TRUE (1)) would indicate that the function executed successfully. For BOOL type functions, you should call ***GetLastError*** for more details.

LSTATUS return type functions will return the error number itself back to the caller. This allows for easy lookup and avoids having to call ***GetLastError*** to determine what happened. We have seen LSTATUS types many times today. Anything other than ERROR_SUCCESS (0) is failure.

HRESULT return type functions will indicate success with something that is zero (0) or even something that is greater than zero. Sometimes you might see S_OK (0) in code. A failure is indicated with a negative value, so checking if the value is less than zero in your code should catch any kind of error. The number itself is also the error number.

HANDLE return type functions will indicate success with either not NULL (0) or not INVALID_HANDLE_VALUE (-1). ***GetLastError*** would be useful for this return type.



Handling Errors (2)

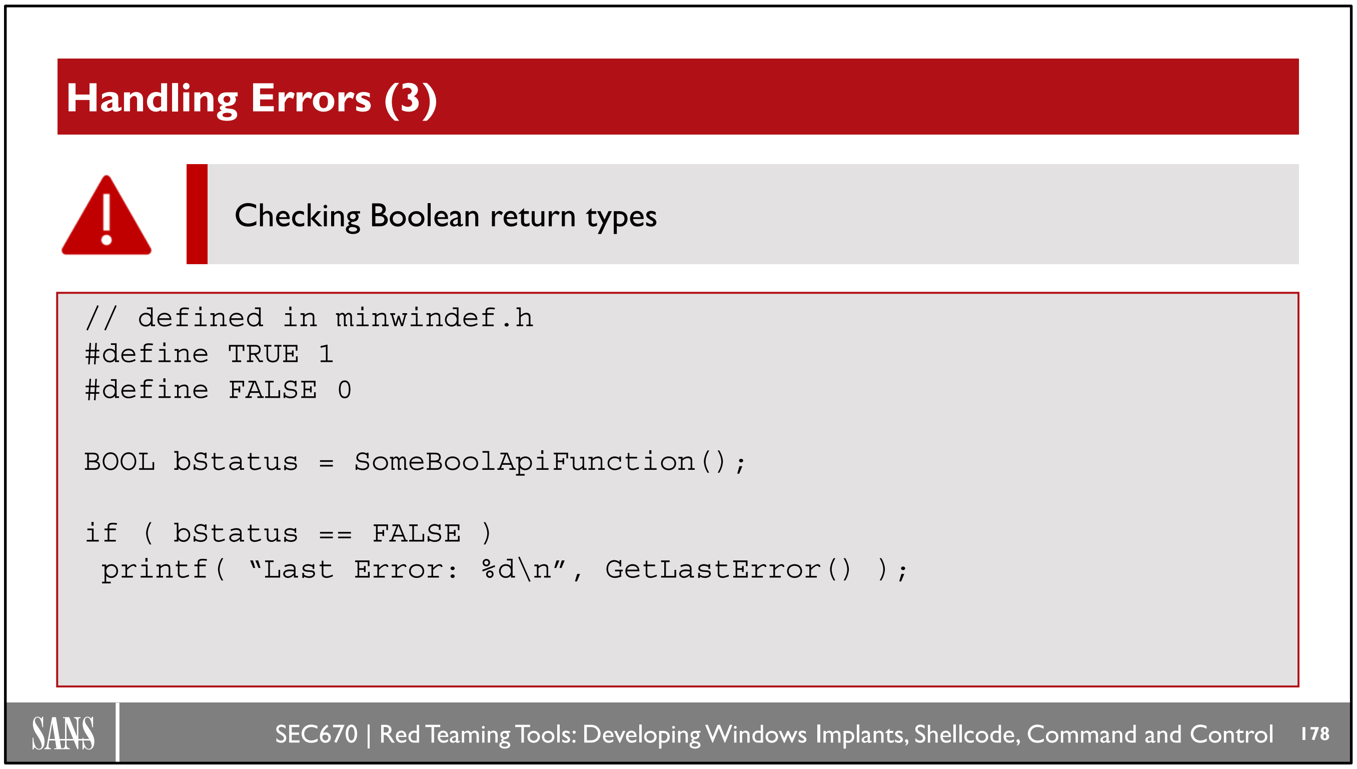
GetLastError

Gets the last error for calling thread

```
// defined in errhandlingapi.h

WINBASEAPI
_Check_return_
_Post_equals_last_error_
DWORD
WINAPI
GetLastError(
   .....
);
```

Handling Errors (2)
The ***GetLastError*** API function does not take a single parameter, as you can see with VOID being specified inside the parentheses. Typically, the best and perhaps only times to call this function are when using functions that have a BOOL return type like ***CreateProcess***. If you are checking to see if a Boolean function failed or succeeded based solely on TRUE / FALSE, you might not get the information you were hoping.

An error will be set when your chosen Windows API function is executing. If you ever get bored and want to explore your reverse engineering side of things, you can reverse some Windows API functions and try to identify the code flow for when they fail. While doing so, you might see calls to ***SetLastError*** or similar. The last error being set is the one you need to get before any other API call is made. If your ***GetLastError*** call is after two consecutive calls, but you were hoping to check the status of the first call, well then too bad. If you want to get the true last error from a Win32 API, then ***GetLastError*** must be called immediately after your API function is called.



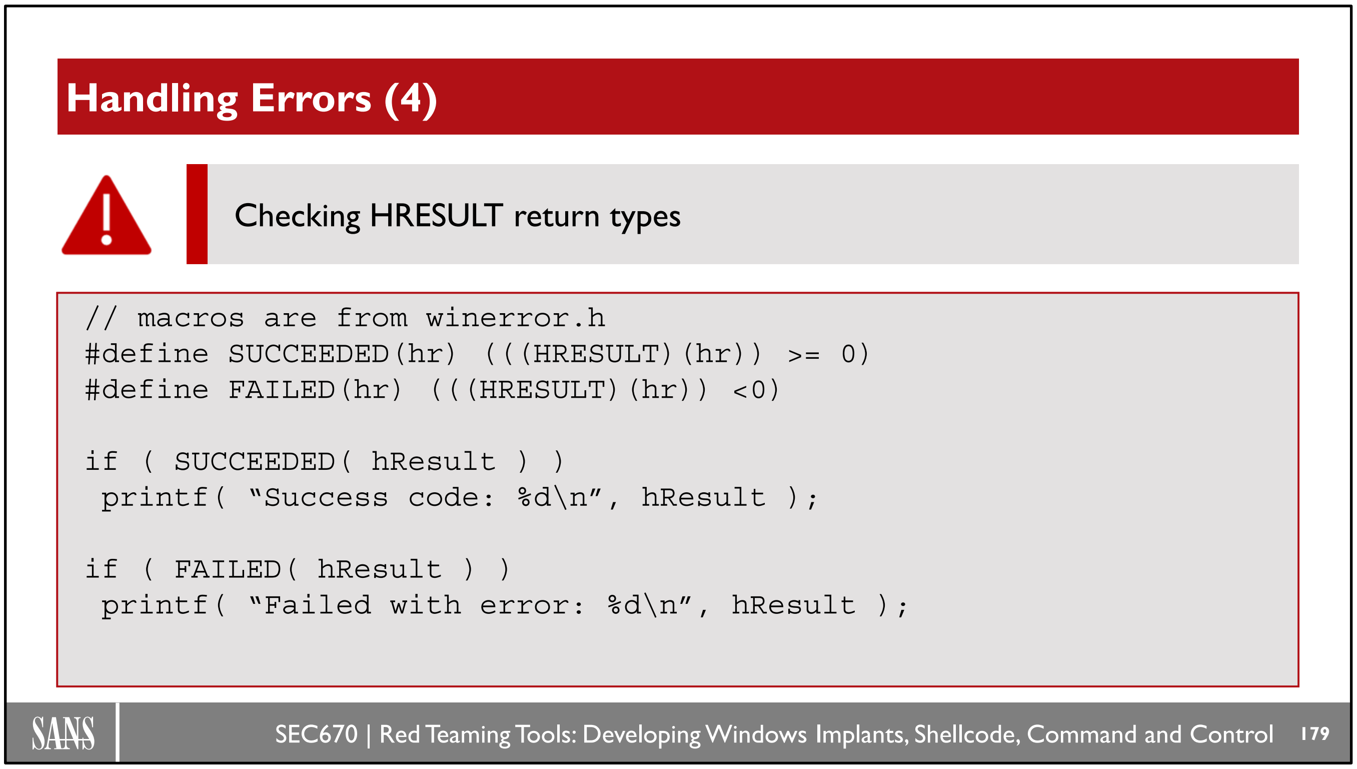
Handling Errors (3)

Checking Boolean return types

```
// defined in minwindef.h
#define TRUE 1
#define FALSE 0

BOOL bStatus = SomeBoolApiFunction();

if ( bStatus == FALSE )
 printf( “Last Error: %d\n”, GetLastError() );
```

Handling Errors (3)
Boolean return values could arguably be the easiest return value to check. A simple TRUE / FALSE check can let the operator of the tool know if a function has failed or errored out. In the end, almost every test condition is simply looking for NULL or non-NULL values. True could be indicated by any value that is not NULL (0). However, we will save that for other types because here, we are only concerned with 0 or 1.

If you do not know what your function should return, you could use BOOL instead of VOID. BOOL could be used to indicate that the function successfully executed and return TRUE. Any error that might happen during its execution can return FALSE where the “error” occurred.



Handling Errors (4)

Checking HRESULT return types

```
// macros are from winerror.h
#define SUCCEEDED(hr) (((HRESULT)(hr)) >= 0)
#define FAILED(hr) (((HRESULT)(hr)) <0)

if ( SUCCEEDED( hResult ) )
 printf( “Success code: %d\n”, hResult );

if ( FAILED( hResult ) )
 printf( “Failed with error: %d\n”, hResult );
```

Handling Errors (4)
When checking the return values of functions that have a HRESULT type, you can use the SUCCEEDED and FAILED macros that are defined in the winerror.h header file. The main task these macros are doing is determining if the result is less than or equal to zero for success. If not, then the function failed with a negative, or less than zero value.

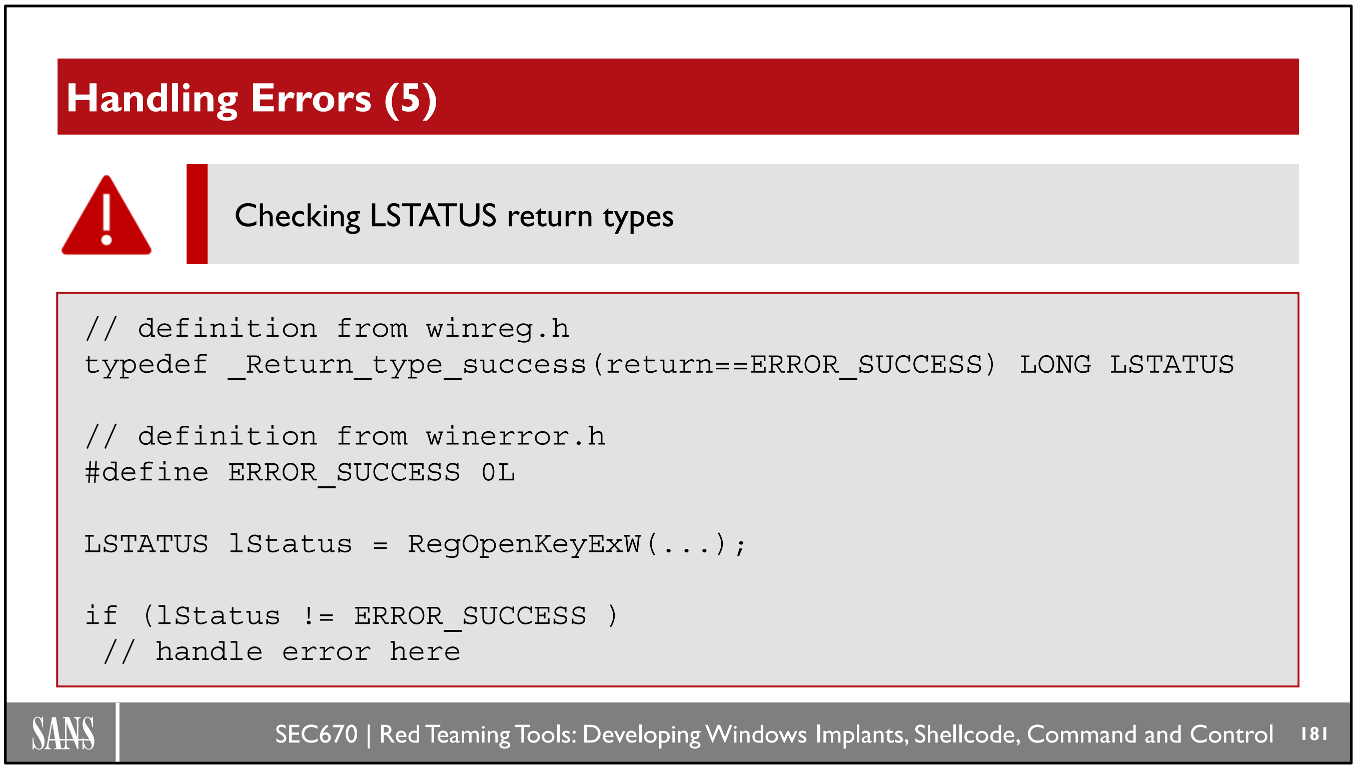
The interesting part about HRESULT types is that they are made up of four fields:
1. A 1-bit code will indicate the severity; zero is success and one is failure
2. A 4-bit value that is reserved for internal purposes only
3. A 11-bit code to indicate if it is a warning or error, which is also known as the facility code
4. A 16-bit code that describes the warning or error

Windows also provided macros that can interpret the one of the four fields of a HRESULT. For example, if one wanted to pull out the facility, they could use the HRESULT_FACILITY(hr) macro. To determine the severity, one could use the HRESULT_SEVERITY(hr) macro.

The following is a copy/paste from the winerror.h header file that explains the breakdown of the 32-bit value layout.

References:
Microsoft header file winerror.h
https://docs.microsoft.com/en-us/office/client-developer/outlook/mapi/hresult
https://docs.microsoft.com/en-us/windows/win32/seccrypto/common-hresult-values

```
// HRESULTs are 32-bit values laid out as follows:
//
//  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
// +-+-+-+-+-+---------------------+-------------------------------+
// |S|R|C|N|r|  Facility     |        Code      |
// +-+-+-+-+-+---------------------+-------------------------------+
//
// where
//
//   S - Severity - indicates success/fail
//
//     0 - Success
//     1 - Fail (COERROR)
//
//   R - reserved portion of the facility code, corresponds to NT's
//       second severity bit.
//
//   C - reserved portion of the facility code, corresponds to NT's
//       C field.
//
//   N - reserved portion of the facility code. Used to indicate a
//       mapped NT status value.
//
//   r - reserved portion of the facility code. Reserved for internal
//       use. Used to indicate HRESULT values that are not status
//       values but are instead message ids for display strings.
//
//   Facility - is the facility code
//
//   Code - is the facility's status code
```



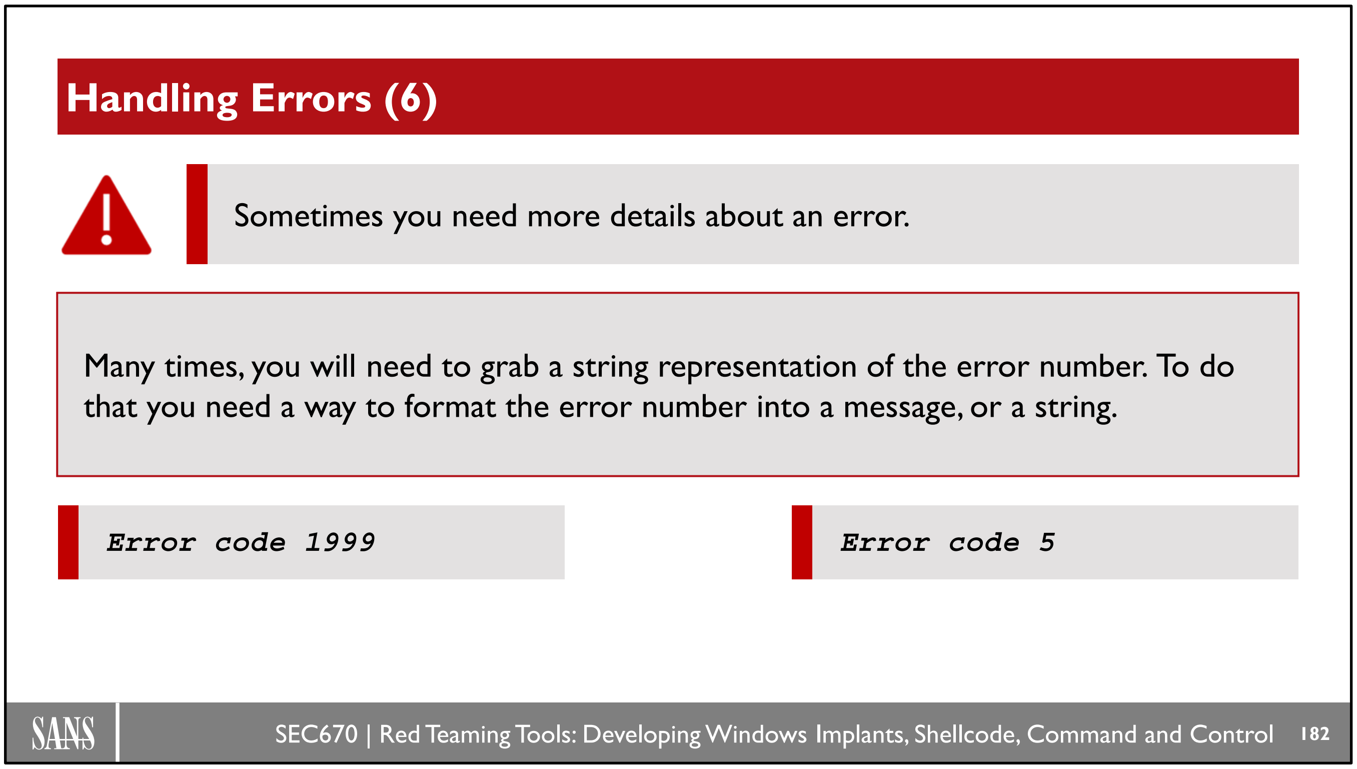
Handling Errors (5)

Checking LSTATUS return types

```
// definition from winreg.h
typedef _Return_type_success(return==ERROR_SUCCESS) LONG LSTATUS

// definition from winerror.h
#define ERROR_SUCCESS 0L

LSTATUS lStatus = RegOpenKeyExW(...);

if (lStatus != ERROR_SUCCESS )
 // handle error here
```

Handling Errors (5)
Most of the Reg* family of APIs are defined with LSTATUS return types, so it is important to know how to check them for errors and success. The winreg.h header file defines most of the Reg* APIs but also has a definition for LSTATUS. Here, you will find that it is simply a typedef as a LONG. While highlighted over LSTATUS, you can press F12, and the header file should open that contains its definition. Here is the full definition for LSTATUS. typedef _Return_type_success(return == ERROR_SUCCESS) LONG LSTATUS.

Knowing the return type and the ERROR_SUCCESS macro value, we can easily create a test condition to see if an error occurred. If success is indicated by a NULL value, then it does not matter if the returned value is negative or positive. ***GetLastError***, or similar, could be used to determine what the error code was and what it means. It would be best to have any errors logged in an error log file instead of to the terminal window. A simple description of the error could be given to the operator running to tool and could tell them that the log file would contain more details about the error.



Handling Errors (6)

Sometimes you need more details about an error.

Many times, you will need to grab a string representation of the error number. To do that you need a way to format the error number into a message, or a string.

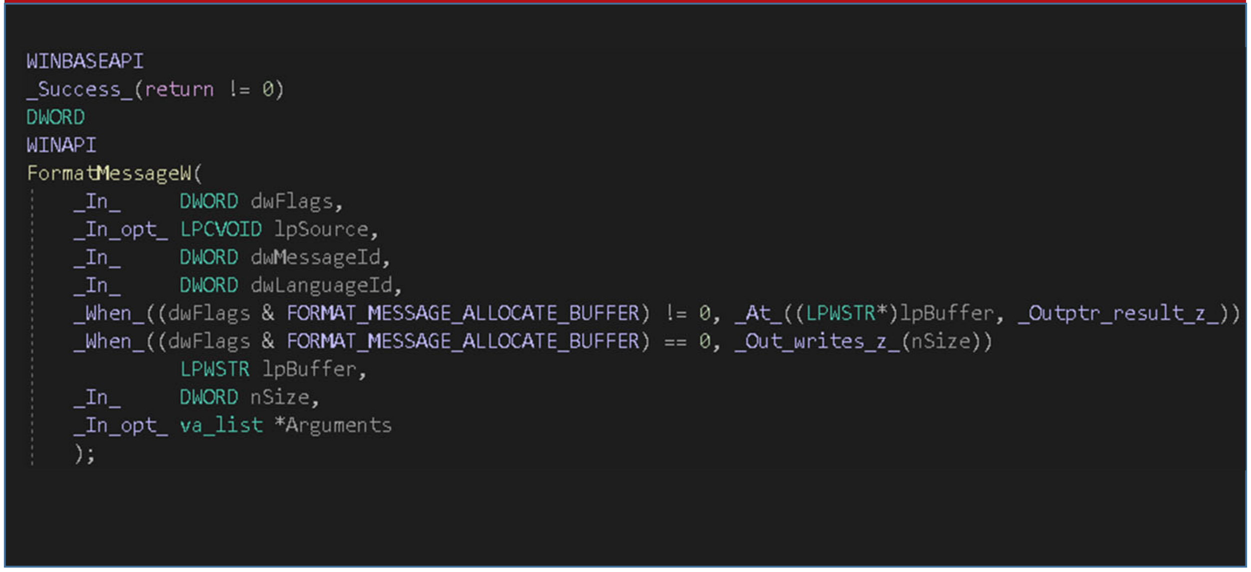
```
Error code 1999
```

```
Error code 5
```

Handling Errors (6)
Seeing your program or function fail with error number 5 might not mean anything at all to you. You would probably have to search online what the error code means, but you do not always have a means to do that lookup. Thankfully, this error lookup can be done programmatically with the help of the ***FormatMessage*** function.

The ***FormatMessage*** function is a very powerful function. Depending on a flag passed, it will allocate the buffer to hold the string on its own or simply let the caller give it a buffer large enough to hold the returning string. The latter option has a bit more manual effort and might be more error prone since it is up to you, the developer, to allocate the correct size buffer. It could be best to just let the function do the allocation for you.

Handling Errors (7)



```
WINBASEAPI
_Success_(return != 0)
DWORD
WINAPI
FormatMessageW(
    _In_     DWORD dwFlags,
    _In_opt_ LPCVOID lpSource,
    _In_     DWORD dwMessageId,
    _In_     DWORD dwLanguageId,
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) != 0, _At_((LPWSTR*)lpBuffer, _Outptr_result_z_))
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) == 0, _Out_writes_z_(nSize))
             LPWSTR lpBuffer,
    _In_     DWORD nSize,
    _In_opt_ va_list *Arguments
    );
```

Handling Errors (7)
At first glance, this looks like a highly complicated function, but in the end, it is not that bad. The SAL annotations make it look more complex than it is, but now that you know how to interpret SAL annotations, the function and its parameters should be easier to figure out. To make it even more understandable, we can break down the parameters and take a quick look at an example of calling the function.

The return type is a DWORD, but that is not terribly important with this function. Also of note, you can see several annotations used to describe the function itself. Success for this function is when the return value does not equal zero (0). Now let's dive into the parameters.

dwFlags can be almost any combination of flags. You can pass it FORMAT_MESSAGE_ALLOCATE_BUFFER to indicate that you want the function to allocate the message buffer on your behalf. This ensures the correct size buffer will be allocated. Passing FORMAT_MESSAGE_FROM_SYSTEM indicates that you want the function to search its message-table resources for the message. The result from ***GetLastError*** can also be passed in when this flag is used. Passing FORMAT_MESSAGE_IGNORE_INSERTS indicates that insert sequences like %1 should be ignored. You will pass that flag if you want to format your message later.

lpSource is where the message is defined. It's best to simply pass NULL here.

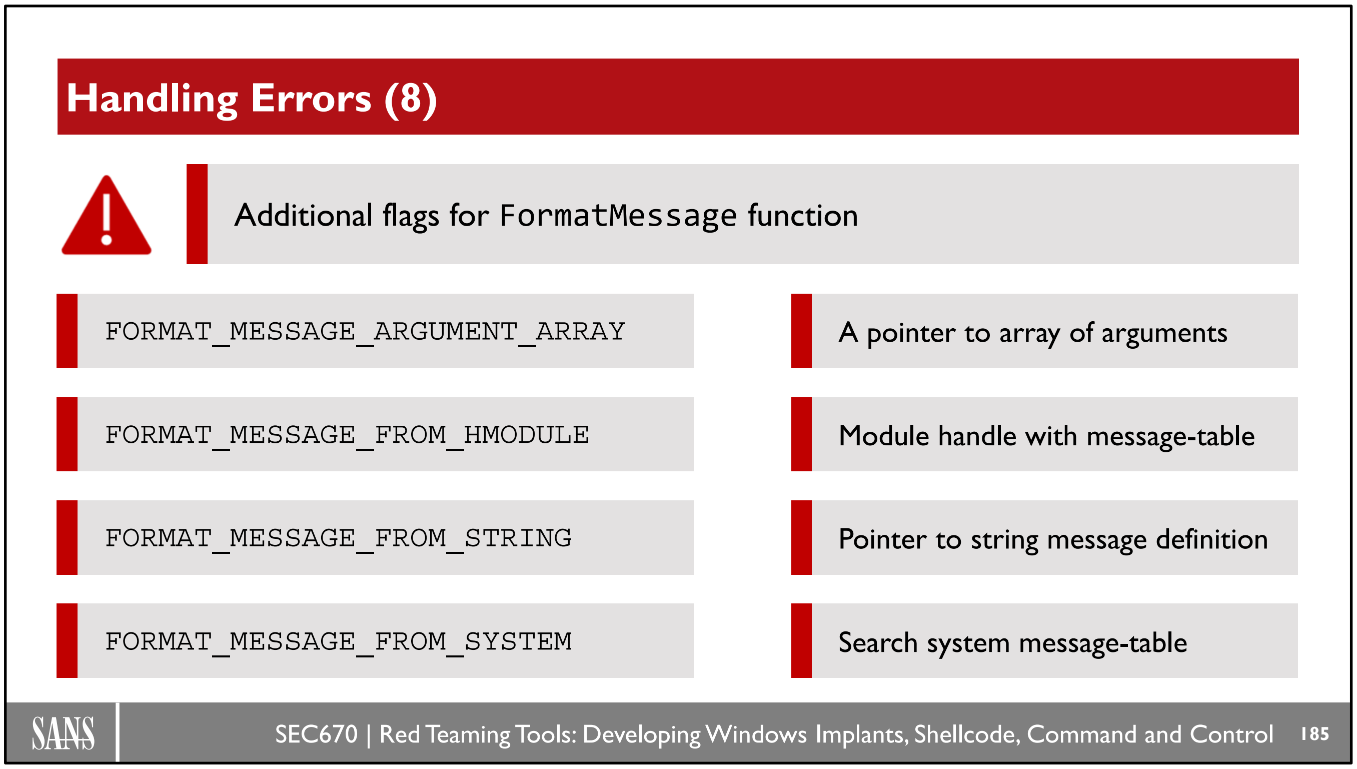
dwMessageId is the error number that is being queried. Be sure to cast your values to the correct type.

dwLangaugeId is the language identifier for the message. Passing NULL here is fine too and the function will do an internal search for the message with the proper LANGID.

Now comes the more interesting part. All this boils down to whether the function is to allocate a buffer, or the user is passing one.

nSize is the minimum size of the buffer to allocate if the allocate buffer flag is passed. If the flag is not set, then this should be the size of the receiving buffer in TCHARS.

**Arguments* are for insert values for a formatted message. %1 is the first argument, %2 would be the second, and so on. Most of the time you can just pass NULL here.



Handling Errors (8)

Additional flags for `FormatMessage` function

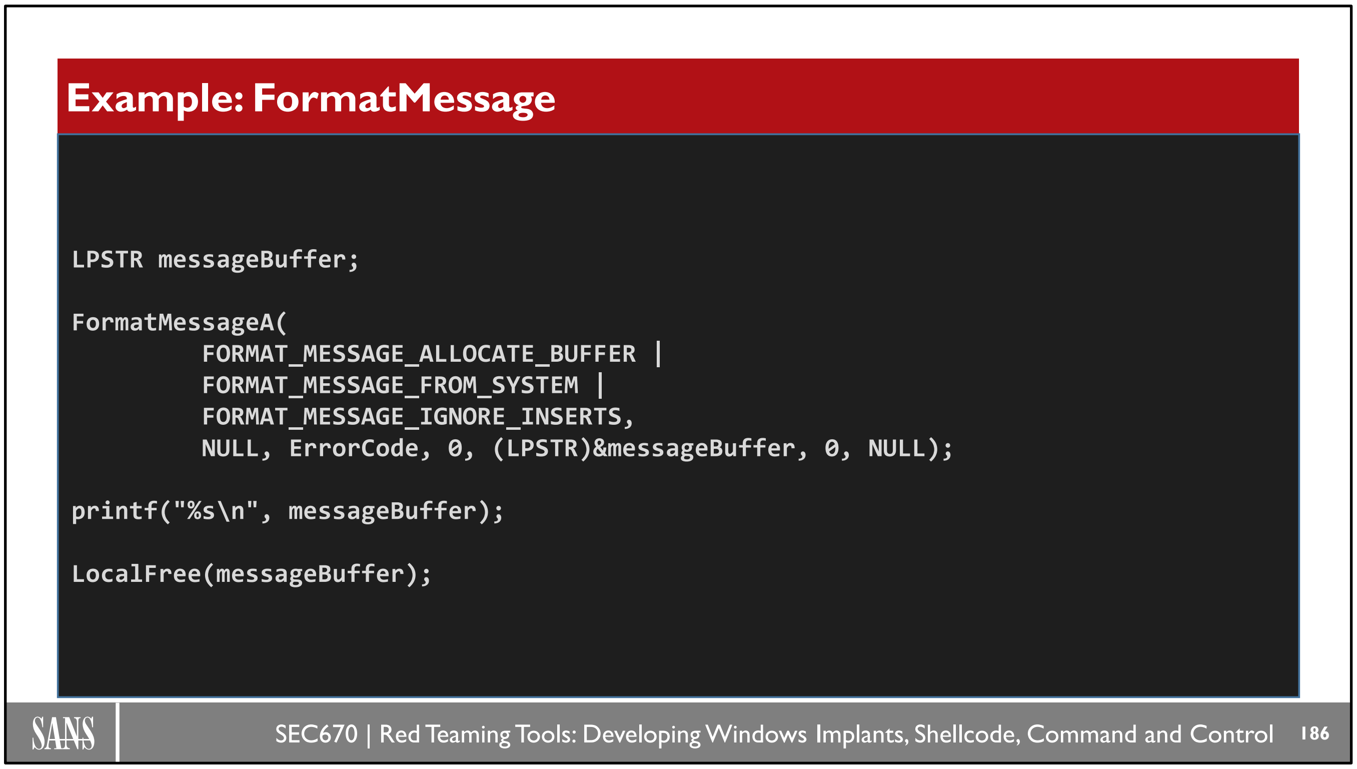
`FORMAT_MESSAGE_ARGUMENT_ARRAY`	A pointer to array of arguments
`FORMAT_MESSAGE_FROM_HMODULE`	Module handle with message-table
`FORMAT_MESSAGE_FROM_STRING`	Pointer to string message definition
`FORMAT_MESSAGE_FROM_SYSTEM`	Search system message-table

Handling Errors (8)
The *dwFlags* parameter can hold a single flag or even a combination of almost any flag. Some flags that cannot be used together are FORMAT_MESSAGE_FROM_STRING and FORMAT_MESSAGE_FROM_HMODULE. You can feel free to experiment with each of these to get a solid understanding of how they are used and what they provide. Here is a brief breakdown of the flags listed on the slide. FORMAT_MESSAGE_ARGUMENT_ARRAY indicates that the argument passed in for *Arguments* is a pointer to an array of arguments, similar to how arguments are passed into a formatted string message with ***printf()***.

The FORMAT_MESSAGE_FROM_HMODULE indicates that the *lpSource* parameter holds a handle to a module where the message-table can be searched. It is possible to create your own message-table and use it as a resource for making a resource only DLL. The Microsoft docs have a writeup of how to do that here: https://docs.microsoft.com/en-us/windows/win32/eventlog/message-text-files.

The FORMAT_MESSAGE_FROM_STRING flags indicates that the lpSource parameter is a string pointer that has the message definition in it. The string might also have insertions like %1.

The FORMAT_MESSAGE_FROM_SYSTEM flag has already been discussed but is provided here for easy reference. By passing this flag, it enables the caller to pass in the value from the ***GetLastError*** function as the dwMesageId.



Example: FormatMessage

```
LPSTR messageBuffer;

FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL, ErrorCode, 0, (LPSTR)&messageBuffer, 0, NULL);

printf("%s\n", messageBuffer);

LocalFree(messageBuffer);
```

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 186

Example: FormatMessge
When doing a system error lookup, say from some LRESULT function, you need to make sure you call this function correctly like in the example code above. Since you do not control a system message, you must pass in the FORMAT_MESSAGE_IGNORE_INSERTS flag. The reason for this is the fact that ***FormatMessage*** can expect insertions in the message and when one is found, it will take whatever the first argument in the argument list is; and attempt to insert it into the message. Since we are only dealing with error codes, there will not be one and the function would fail with error 87: ERROR_INVALID_PARAMETER. Also, the way this function is being called, we are asking the system to allocate a buffer on our behalf by passing in FORMAT_MESSAGE_ALLOCATE_BUFFER. When using the FORMAT_MESSAGE_ALLOCATE_BUFFER flag, you must not forget to free that allocated buffer. To do so, just call ***LocalFree*** when you are done with it. This should be done with any API that dynamically allocates chunks of memory on your behalf.

Bottom line: when using ***FormatMessage*** to perform system error code lookups, you must pass both flags FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS.

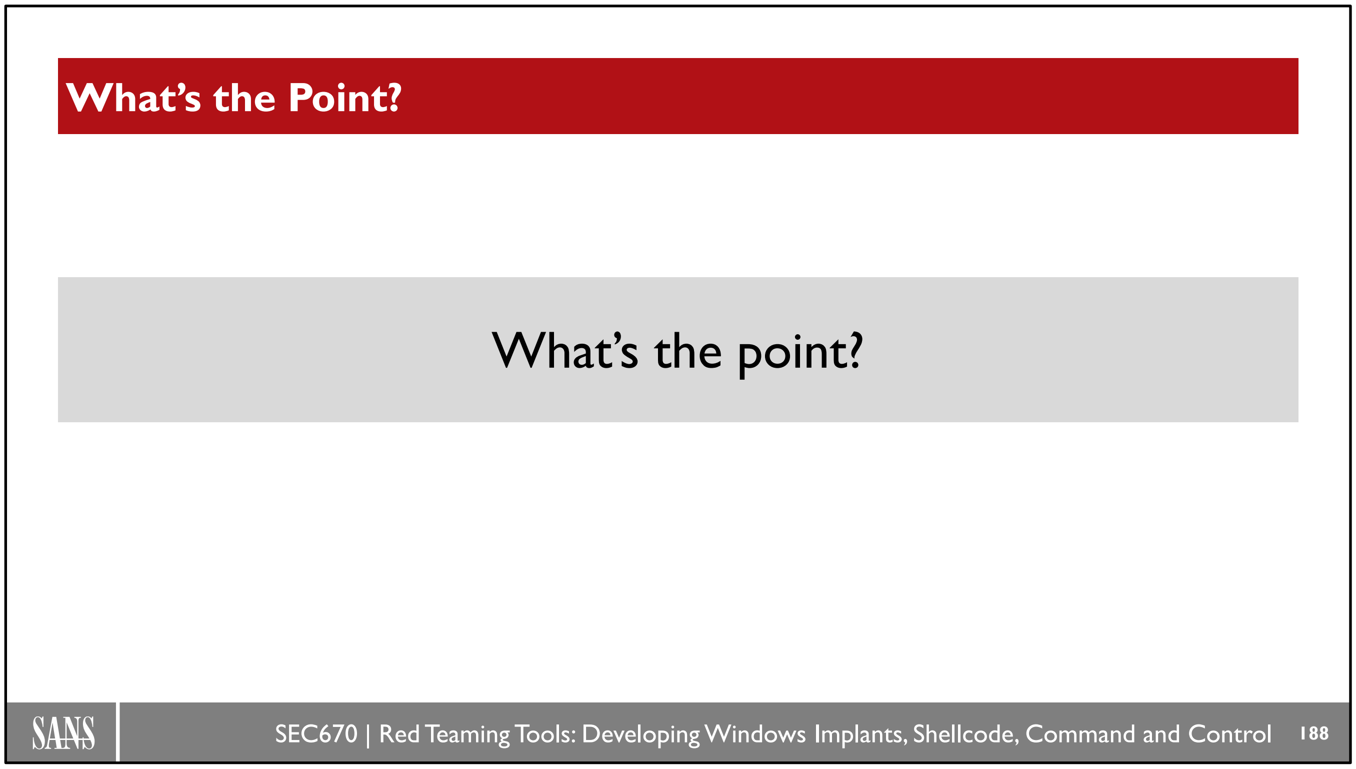
Lab 1.6: CreateFile

Using the CreateFile function create a file and write data to it.

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

Lab 1.6: CreateFile
This exercise is to get you familiar with the ***CreateFile*** API.

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

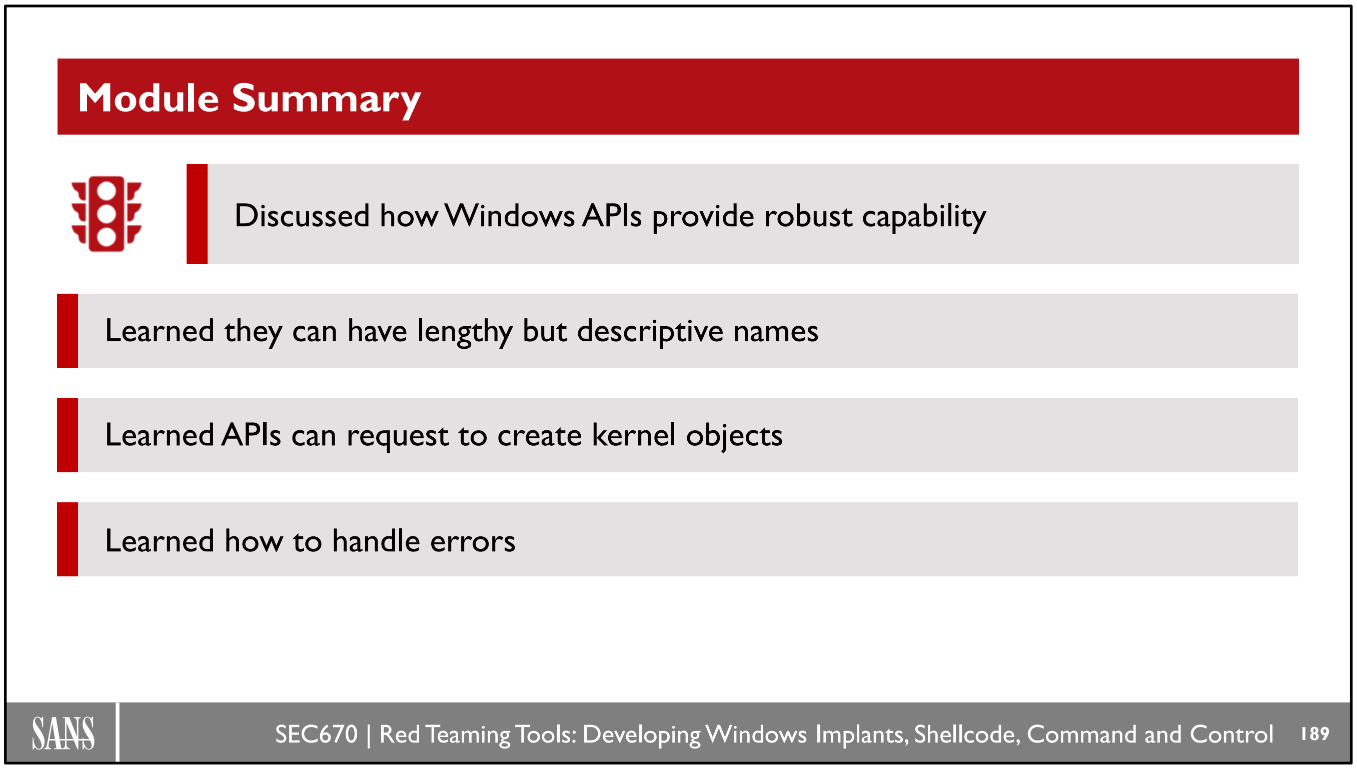


What’s the Point?

What’s the point?

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 188

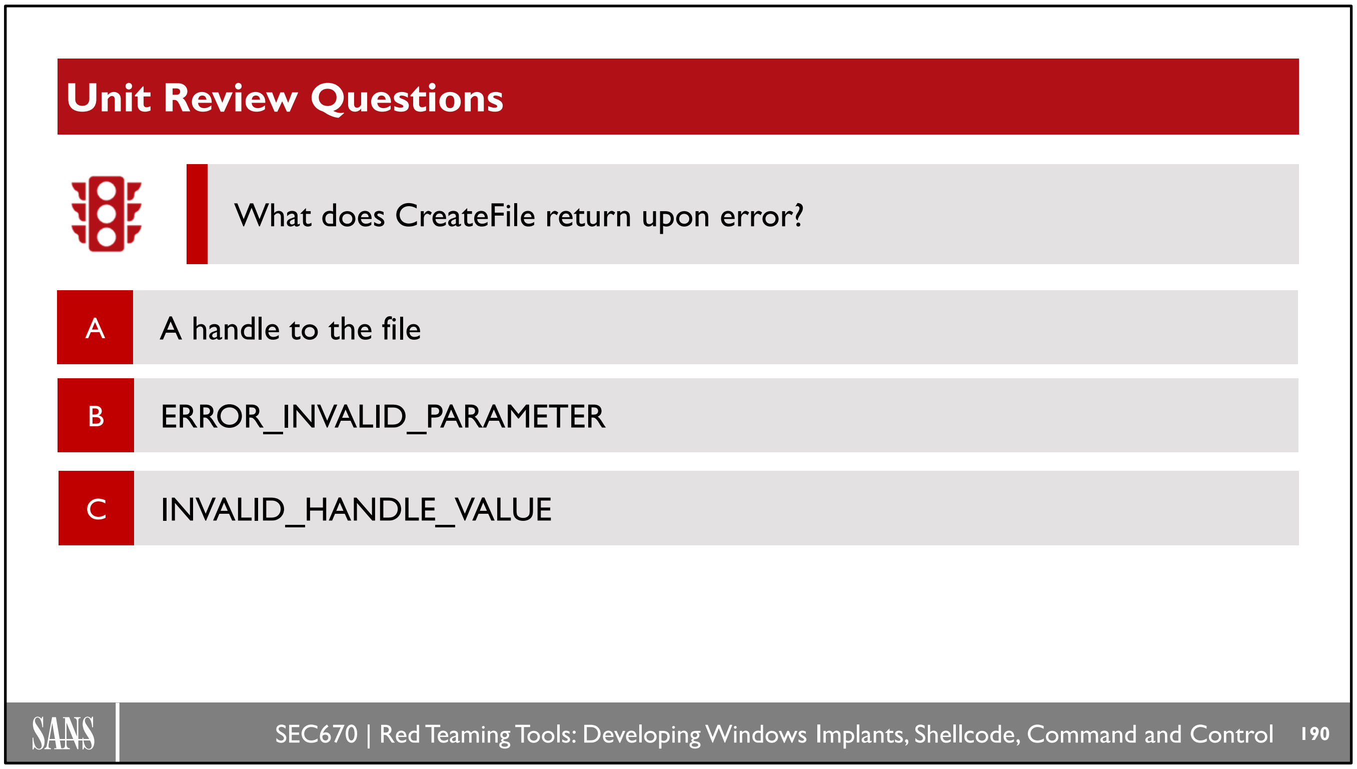
What’s the Point?
The point of this lab was to get you familiar with the various parameters of the ***CreateFile*** API. As you may have noticed, there are several parameters that can really change the behavior of the function. The other function that was used was ***WriteFile***, which was not discussed previously. This was done intentionally to force you to look at the parameters either online from MSDN or from the function’s declaration in the header file.



Module Summary

- Discussed how Windows APIs provide robust capability
- Learned they can have lengthy but descriptive names
- Learned APIs can request to create kernel objects
- Learned how to handle errors

Module Summary
In this module, we discussed the importance of using the Windows API in your programs because they provide your program with such a robust capability. The API names can be quite lengthy, but they are very descriptive with what they are attempting to do, like ***CreateProcess***. There are many APIs that will make requests to the kernel to create objects or modify existing objects with handles returned from the create functions.



Unit Review Questions

What does CreateFile return upon error?

A A handle to the file

B ERROR_INVALID_PARAMETER

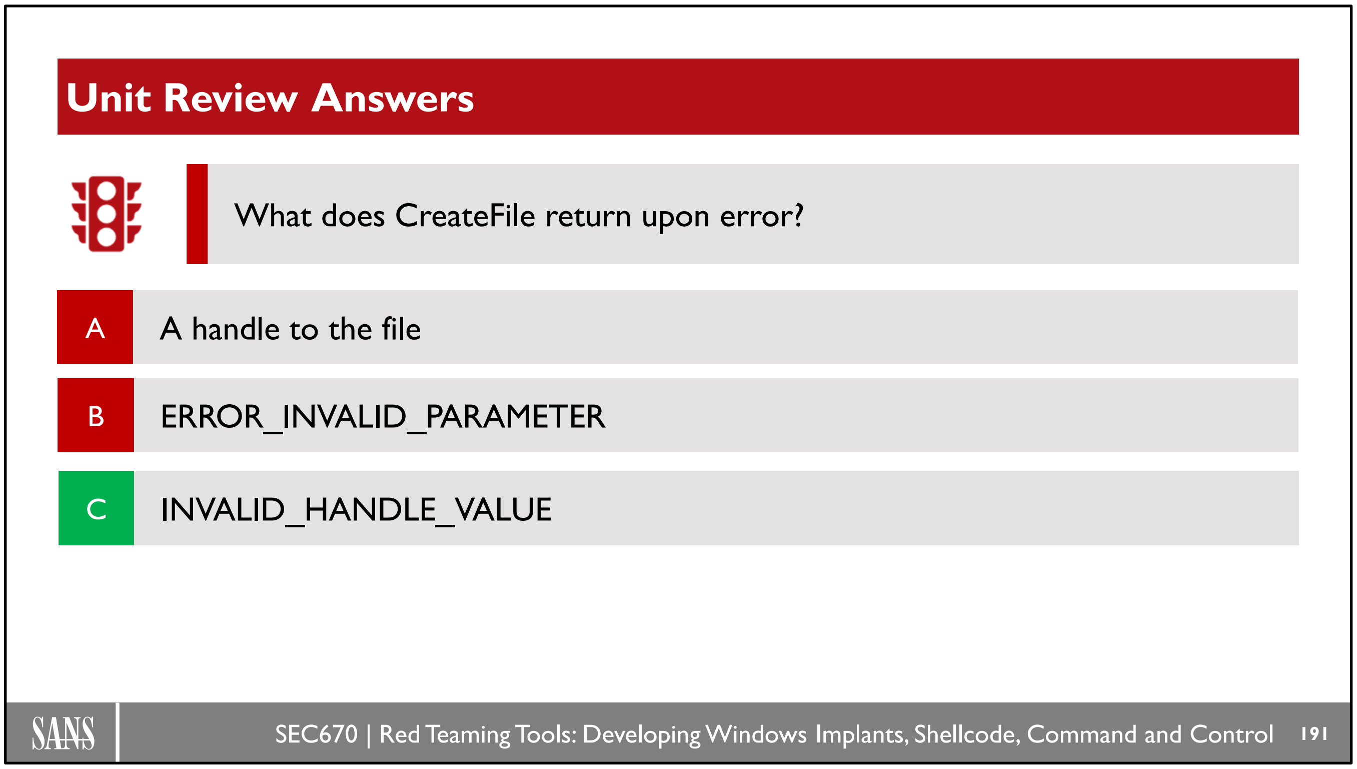
C INVALID_HANDLE_VALUE

Unit Review Questions
Q: What does CreateFile return upon error?

A: A handle to the file

B: ERROR_INVALID_PARAMETER

C: INVALID_HANDLE_VALUE



Unit Review Answers

What does CreateFile return upon error?

A A handle to the file

B ERROR_INVALID_PARAMETER

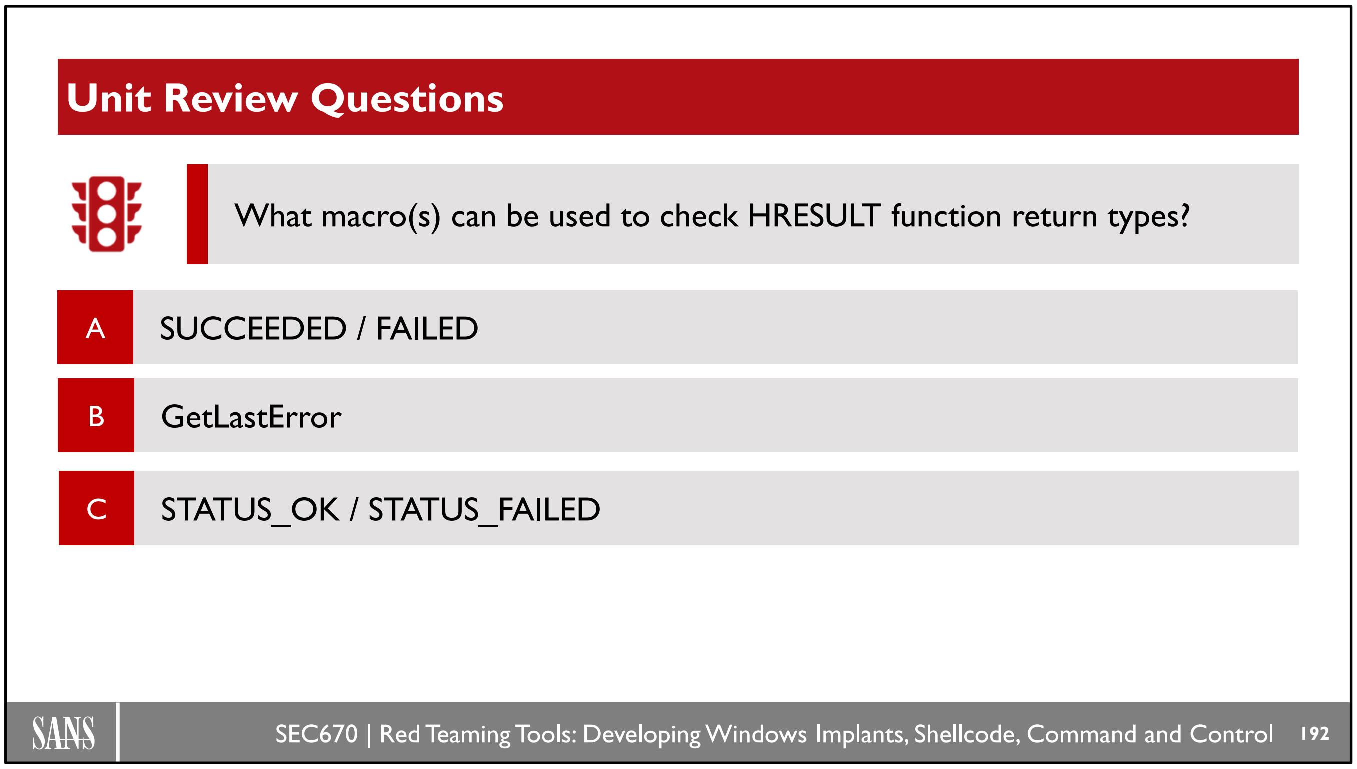
C INVALID_HANDLE_VALUE

Unit Review Answers
Q: What does CreateFile return upon error?

A: A handle to the file

B: ERROR_INVALID_PARAMETER

C: INVALID_HANDLE_VALUE



Unit Review Questions

What macro(s) can be used to check HRESULT function return types?

A SUCCEEDED / FAILED

B GetLastError

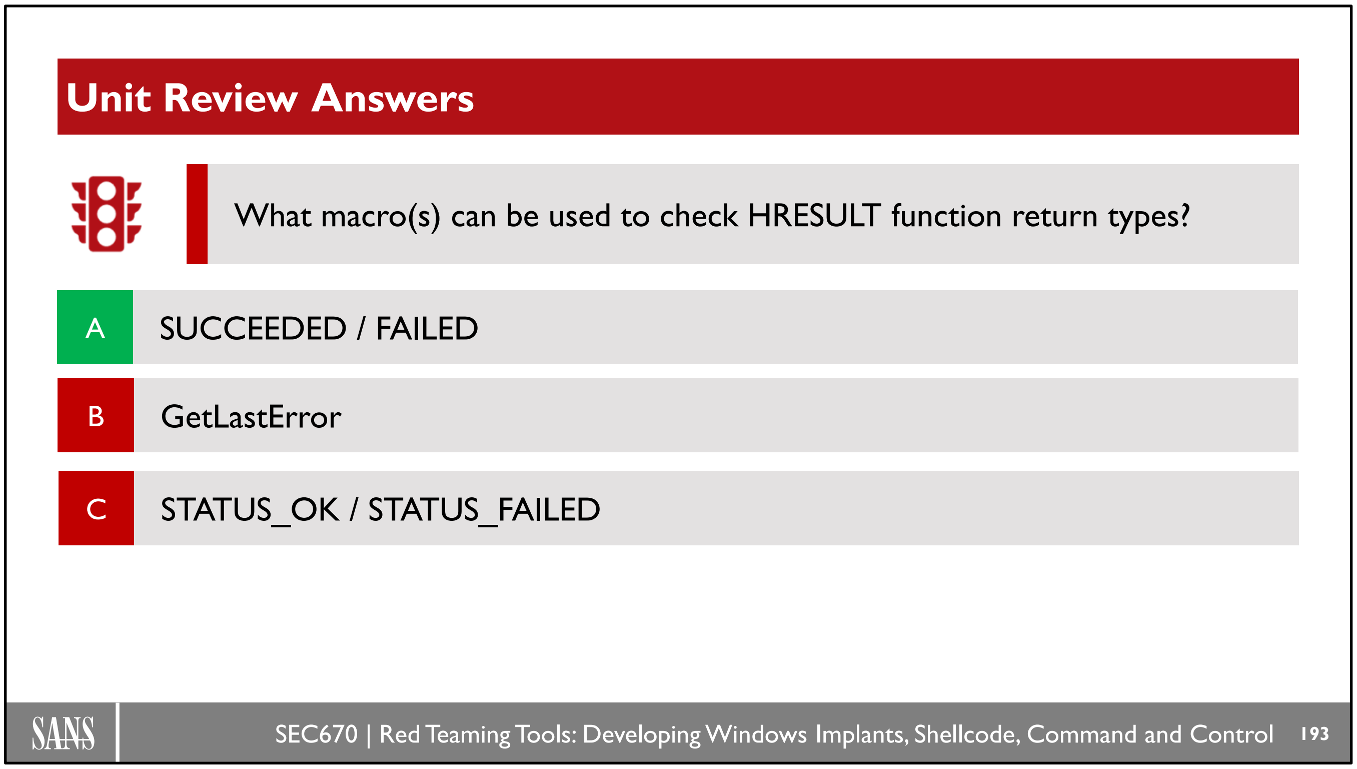
C STATUS_OK / STATUS_FAILED

Unit Review Questions
Q: What macro(s) can be used to check HRESULT function return types?

A: SUCCEEDED / FAILED

B: GetLastError

C: STATUS_OK / STATUS_FAILED



Unit Review Answers

What macro(s) can be used to check HRESULT function return types?

A	SUCCEEDED / FAILED
B	GetLastError
C	STATUS_OK / STATUS_FAILED

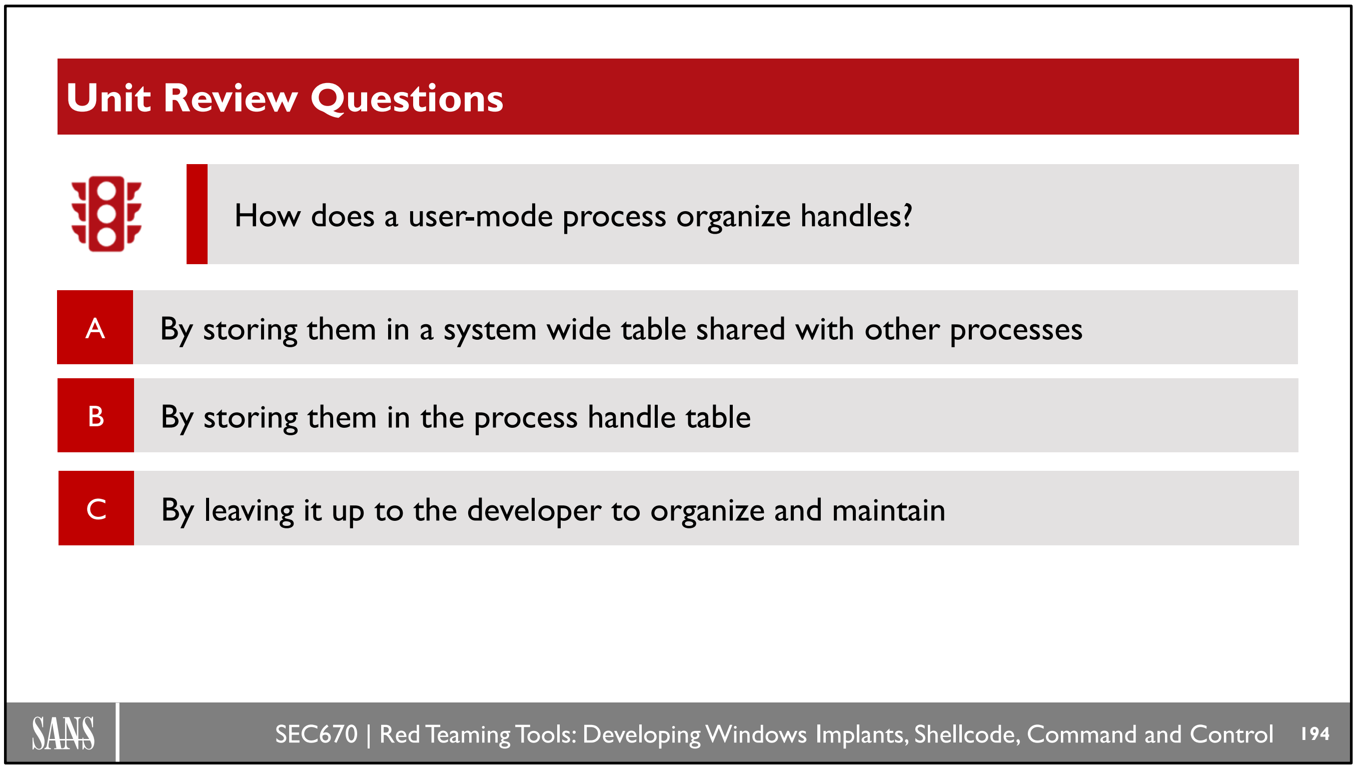
SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 193

Unit Review Answers
Q: What macro(s) can be used to check HRESULT function return types?

A: SUCCEEDED / FAILED

B: GetLastError

C: STATUS_OK / STATUS_FAILED



Unit Review Questions

How does a user-mode process organize handles?

A By storing them in a system wide table shared with other processes

B By storing them in the process handle table

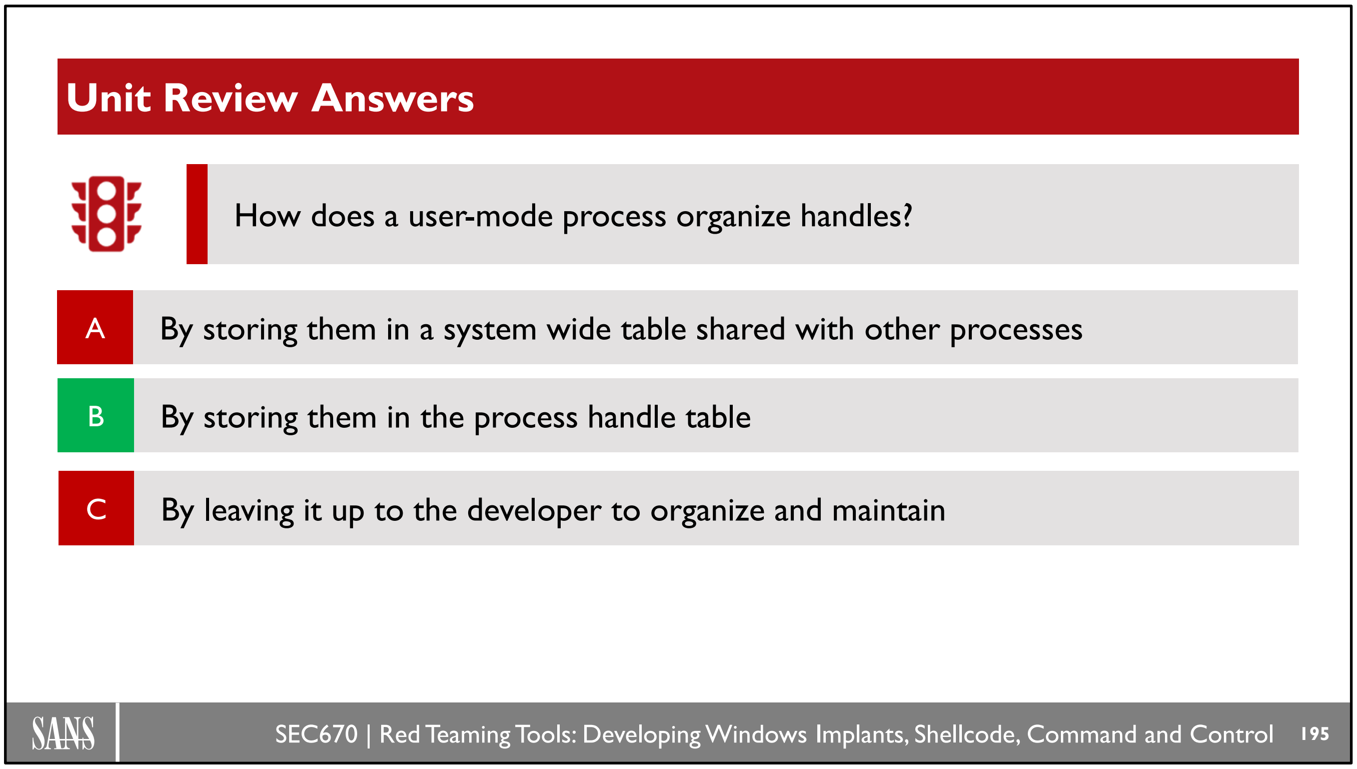
C By leaving it up to the developer to organize and maintain

Unit Review Questions
Q: How does a user-mode process organize handles?

A: By storing them in a system wide table shared with other processes

B: By storing them in the process handle table

C: By leaving it up to the developer to organize and maintain



Unit Review Answers

How does a user-mode process organize handles?

A By storing them in a system wide table shared with other processes

B By storing them in the process handle table

C By leaving it up to the developer to organize and maintain

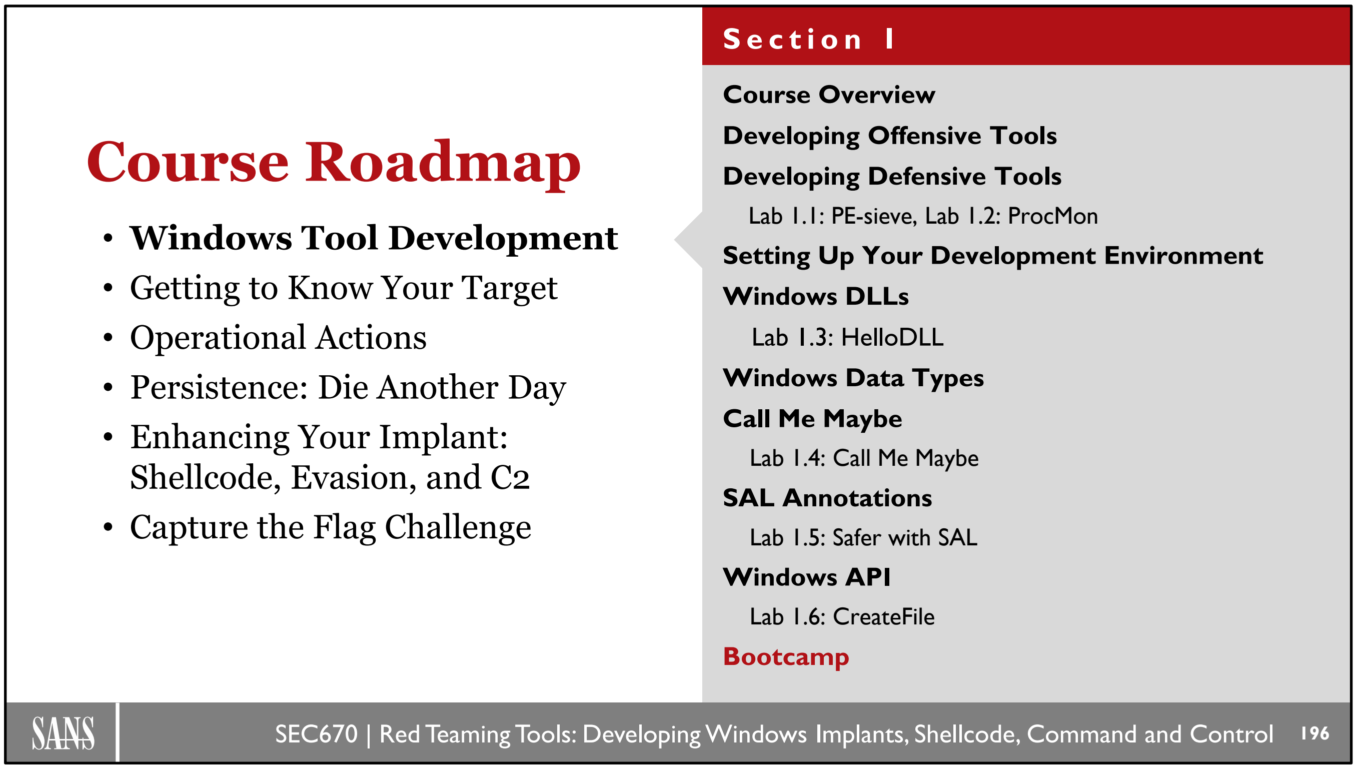
SANS SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 195

Unit Review Answers
Q: How does a user-mode process organize handles?

A: By storing them in a system wide table shared with other processes

B: By storing them in the process handle table

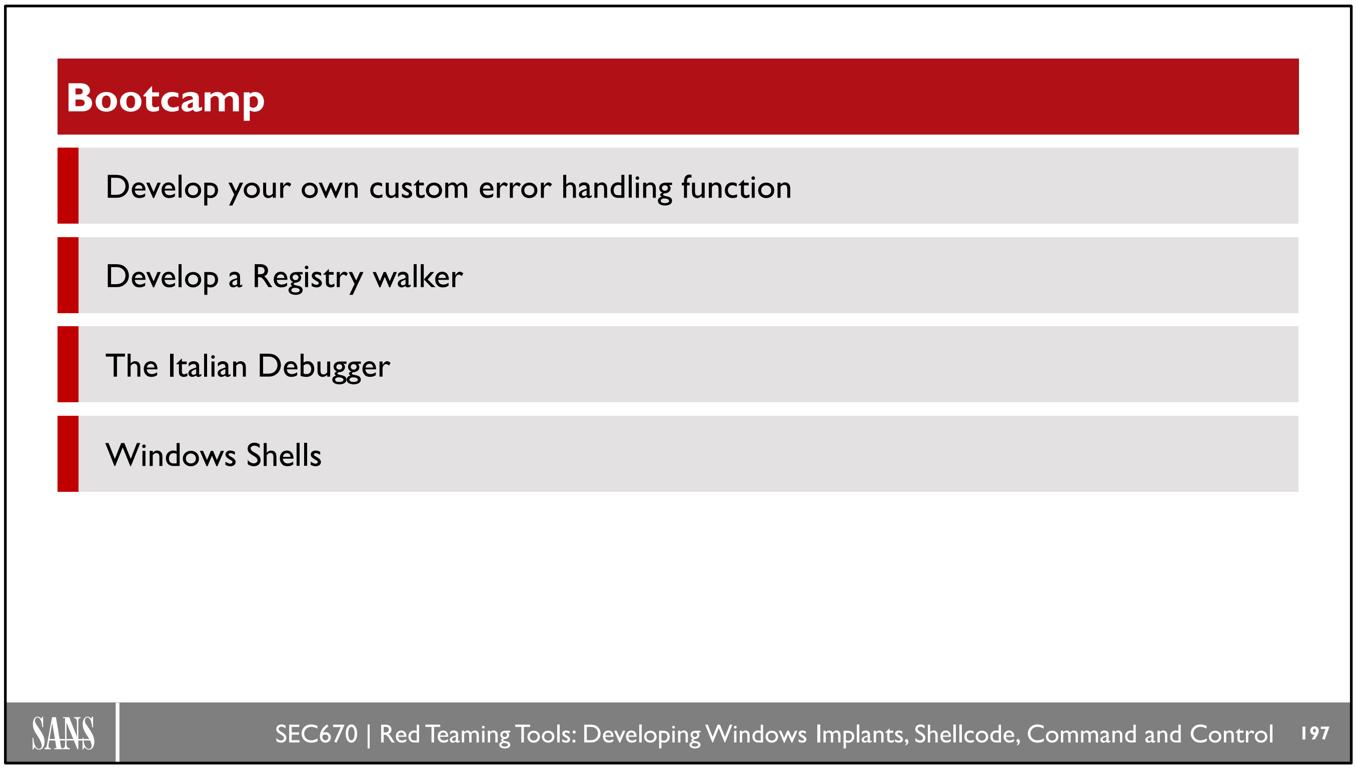
C: By leaving it up to the developer to organize and maintain



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

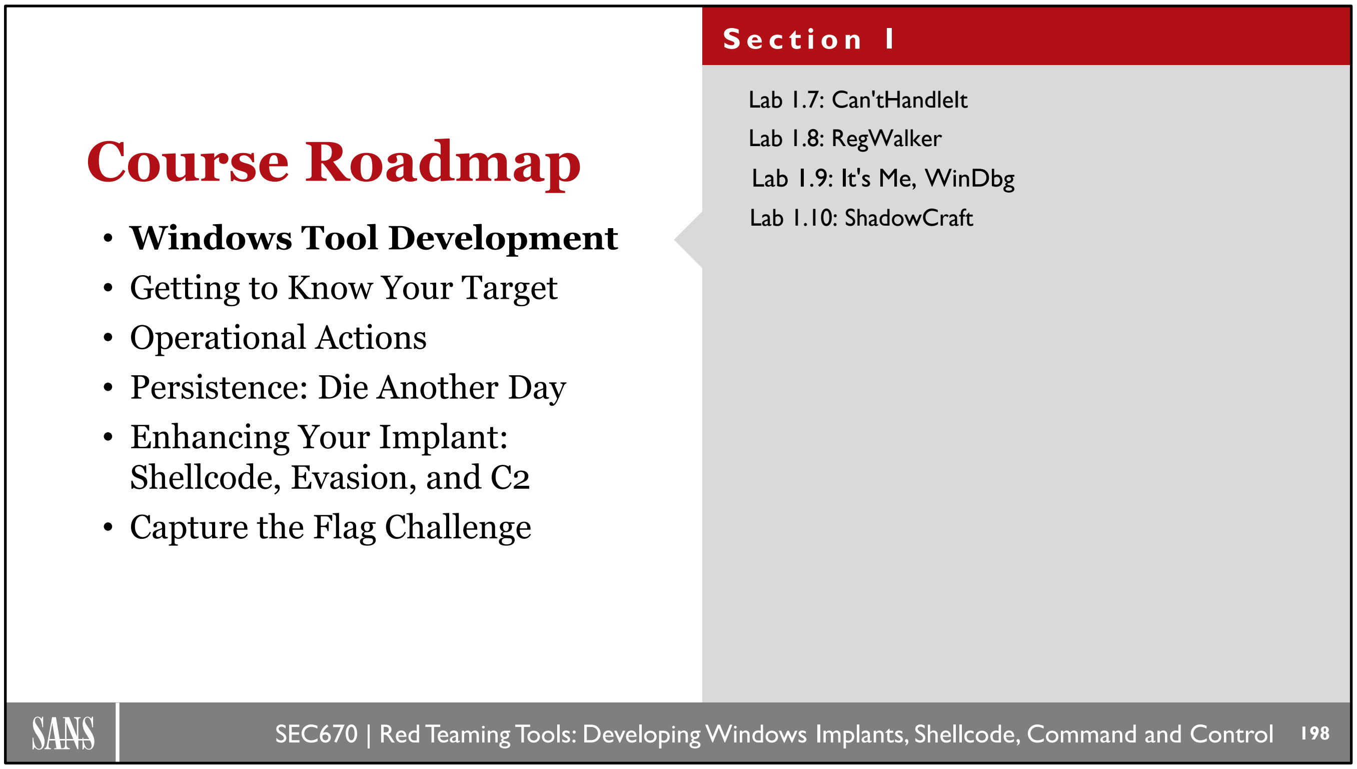
The bootcamp portion of the class is really extended class hours but without the lecture. This will give you lecture-free time for you to go back and practice labs again or take on a few of the challenges listed on the next slide.



Bootcamp

- Develop your own custom error handling function
- Develop a Registry walker
- The Italian Debugger
- Windows Shells

Bootcamp
For this bootcamp, this is the time to put the concepts learned from this section into practice. There are several bootcamp challenges and all of them are optional for you to complete. The challenges for this bootcamp can be done in any order since they do not depend on each other. The first challenge is to create a custom error handling function that makes use of ***GetLastError*** and ***FormatMessage***. The second challenge is to create a program that can recursively walk a given Registry key and dump the key's values, if any. The debugger one is not really a challenge per se, but more of a guide to get deeply acquainted with WinDbg Preview and remote kernel debugging. The last one is about creating custom shells that can eventually be run remotely across systems.



Course Roadmap

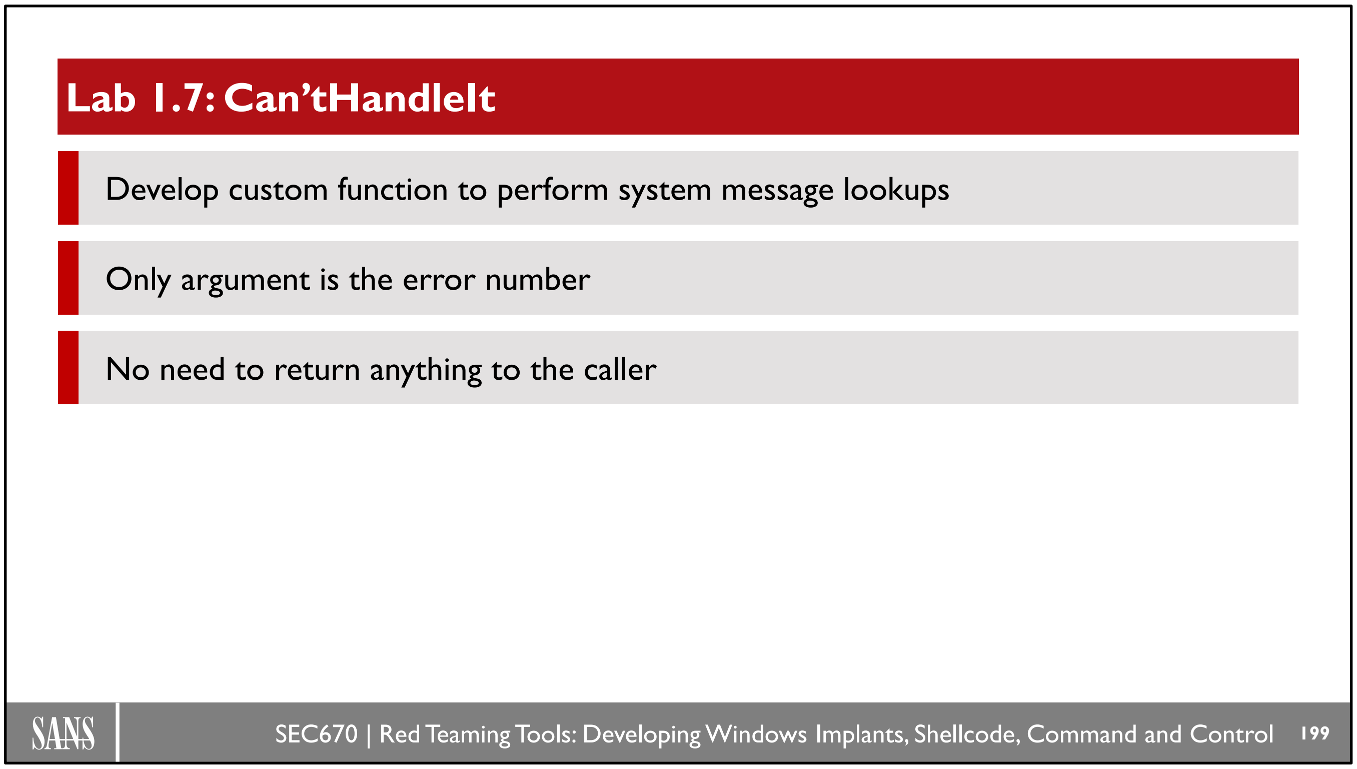
- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Lab 1.7: Can'tHandleIt
Lab 1.8: RegWalker
Lab 1.9: It's Me, WinDbg
Lab 1.10: ShadowCraft

SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 198

The bootcamp portion of the class is really extended class hours but without the lecture. This will give you lecture-free time for you to go back and practice labs again or take on a few of the challenges listed on the next slide.

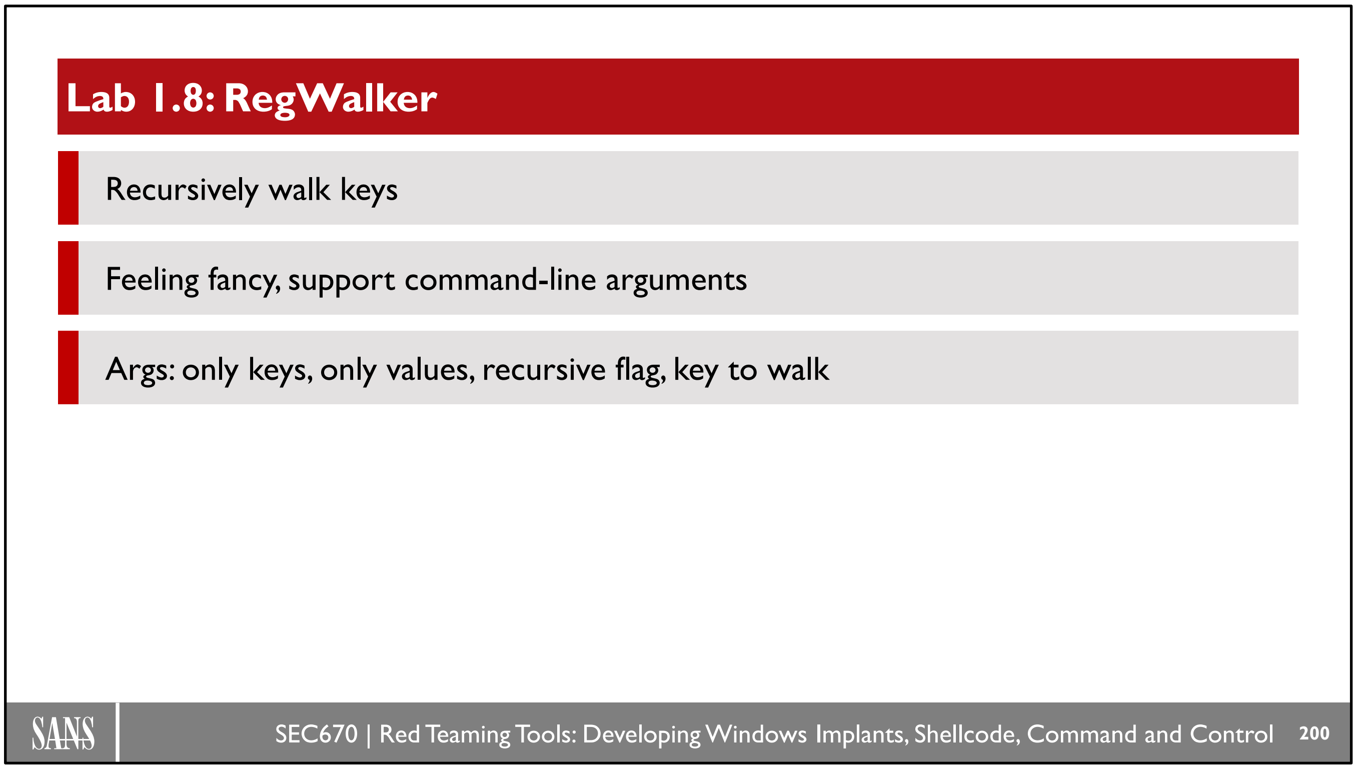


Lab 1.7: Can’tHandleIt

- Develop custom function to perform system message lookups
- Only argument is the error number
- No need to return anything to the caller

SANS SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 199

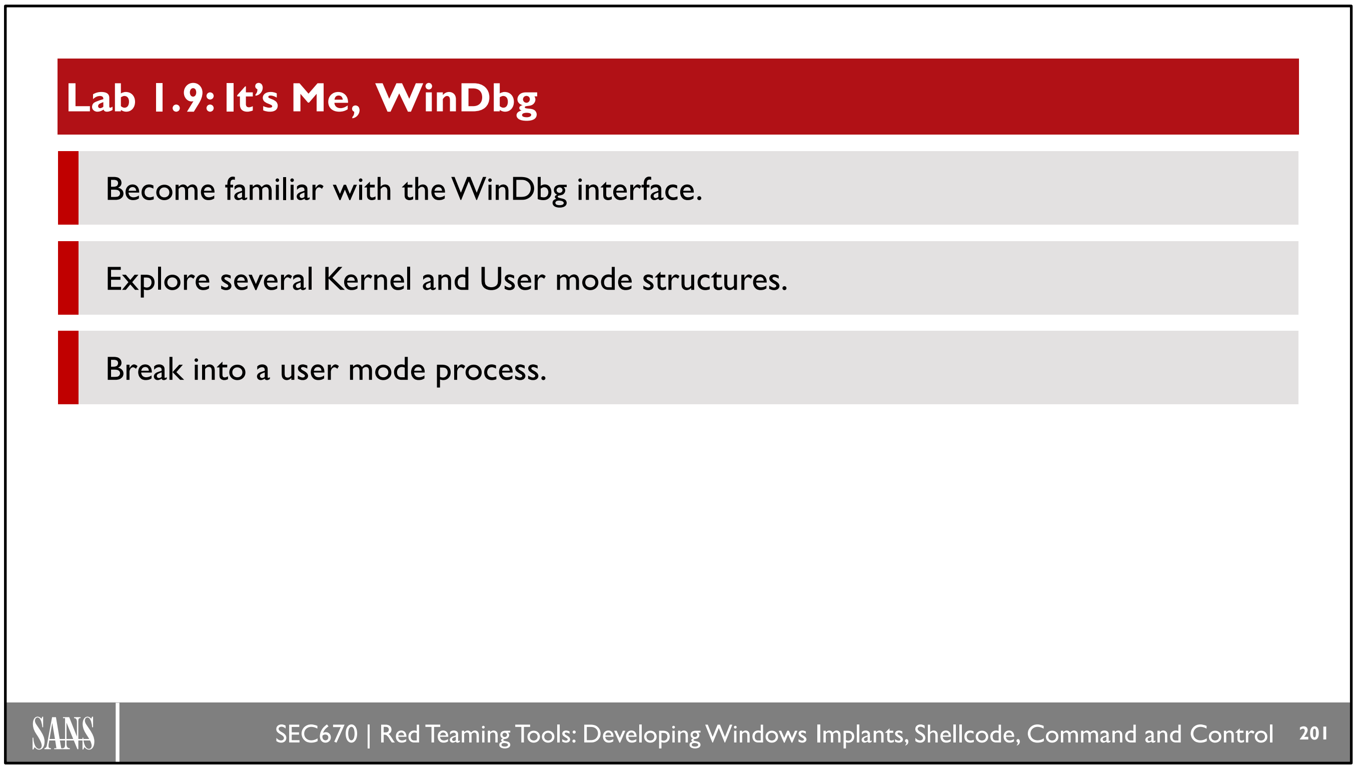
Lab 1.7: Can’tHandleIt
Please refer to the eWorkbook for the details of this bootcamp challenge.



Lab 1.8: RegWalker

- Recursively walk keys
- Feeling fancy, support command-line arguments
- Args: only keys, only values, recursive flag, key to walk

Lab 1.8: RegWalker
Please refer to the eWorkbook for the details of this bootcamp challenge.



Lab 1.9: It's Me, WinDbg

- Become familiar with the WinDbg interface.
- Explore several Kernel and User mode structures.
- Break into a user mode process.

Lab 1.9: It's Me, WinDbg
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 1.10: ShadowCraft

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

Lab 1.10: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.