670.1

Windows Tool Development

SANS | GIAC CERTIFICATIONS

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

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

SEC670.1 | 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)

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

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

Objectives

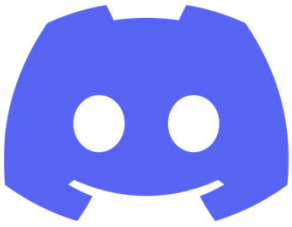
Our objectives for this module are:

- 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

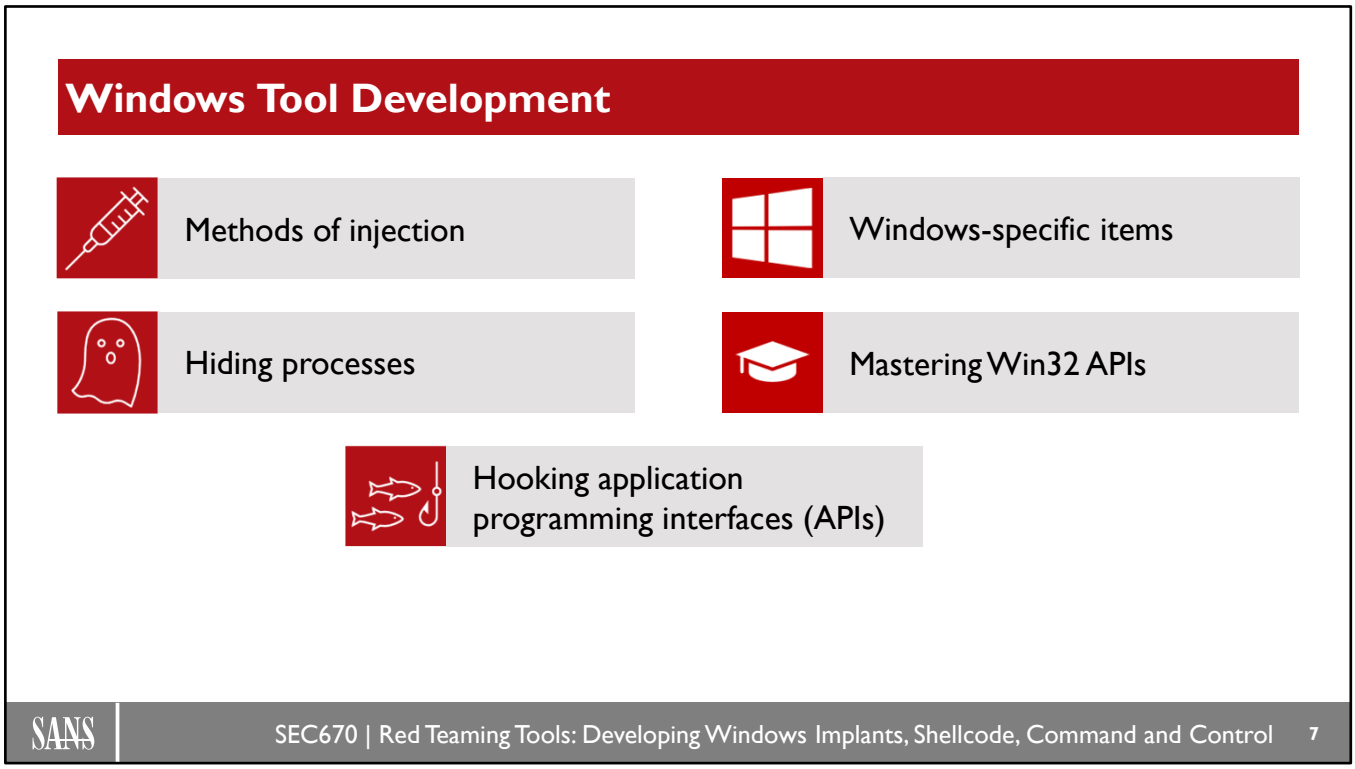
Coding CTF Challenges!

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

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

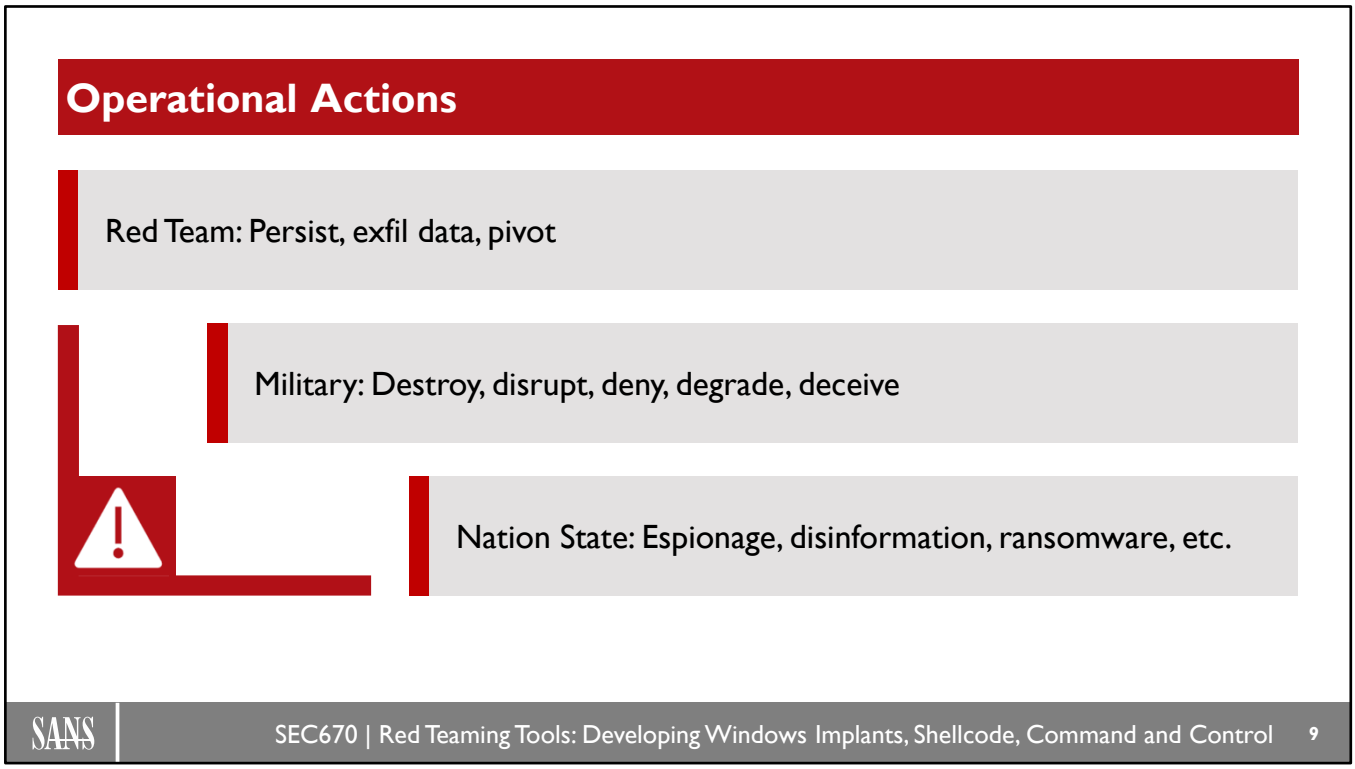
What happens after you gain initial access to the target?

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

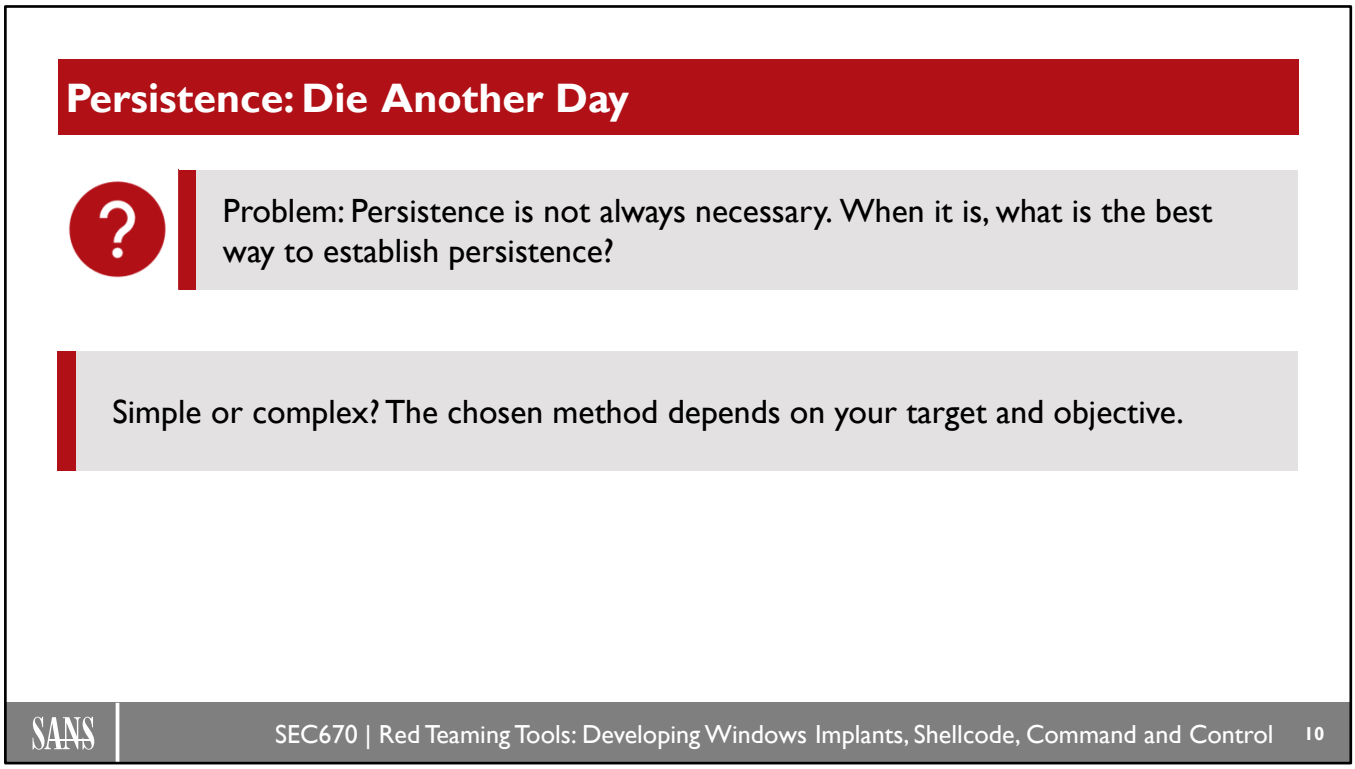
This course is not aimed at gaining that initial access (that happens in SEC560), but rather what custom tools can be made to leverage that access. Since you are the developer, what actions are taken next are nearly limitless. You are no longer going to be bound by a current tool or C2 framework and 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

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

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.

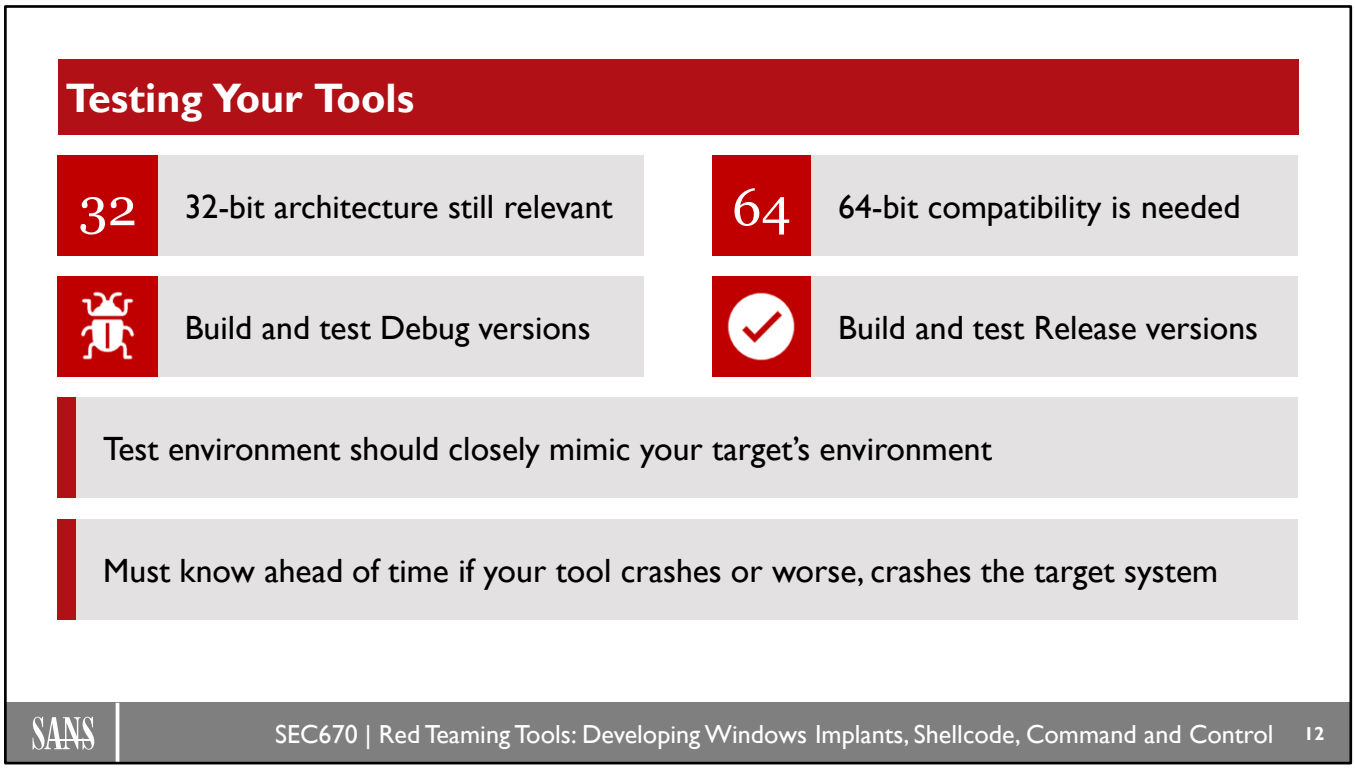
Enhancing Your Implant: Shellcode, Evasion, and C2

When you've met your match. When facing a tech savvy admin. When stealth is desired. When normal (basic) techniques fail.

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

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?

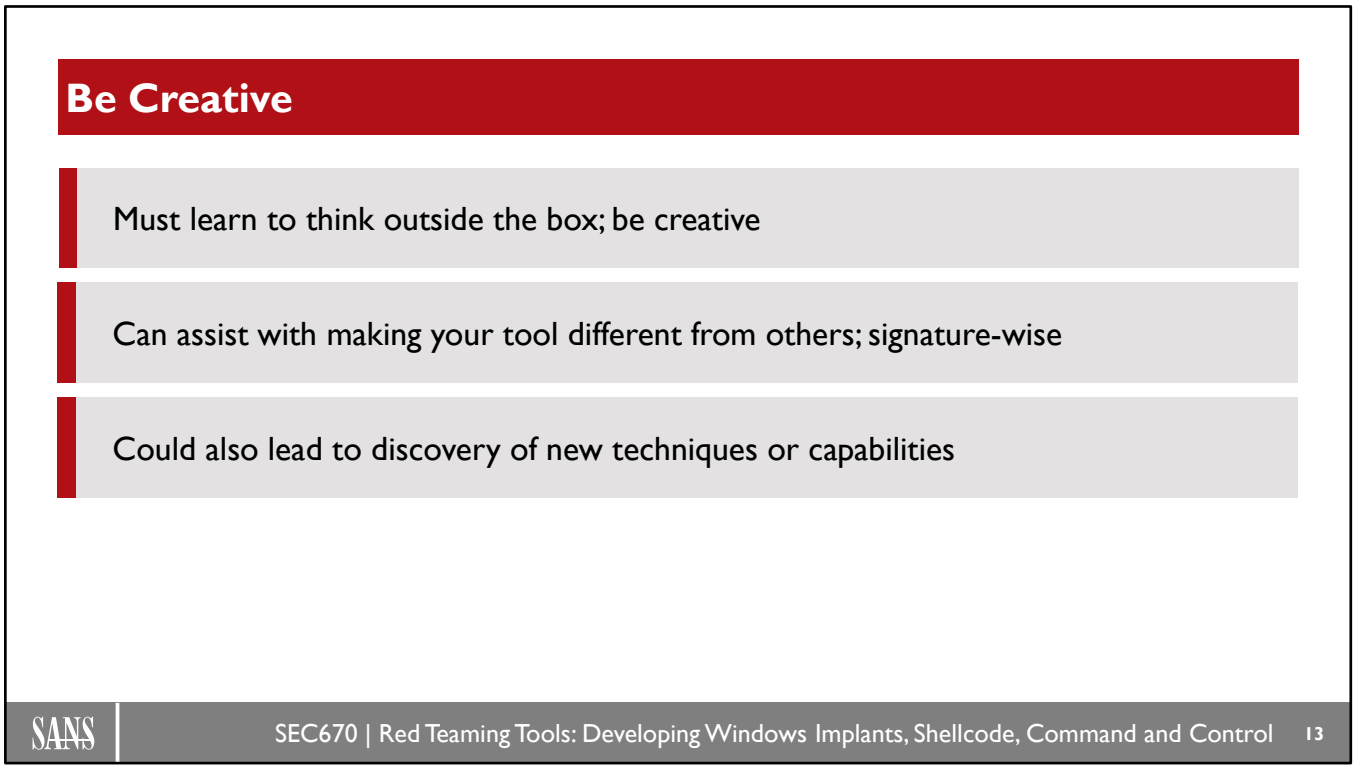
Many of the advanced techniques will require intimate knowledge of the PE32 and PE32+ format, process virtual address space, the registry, OS internals, etc. Such mastery comes after years of tool development and debugging undocumented APIs. Also, not much is going to happen if you cannot control your 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

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
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.

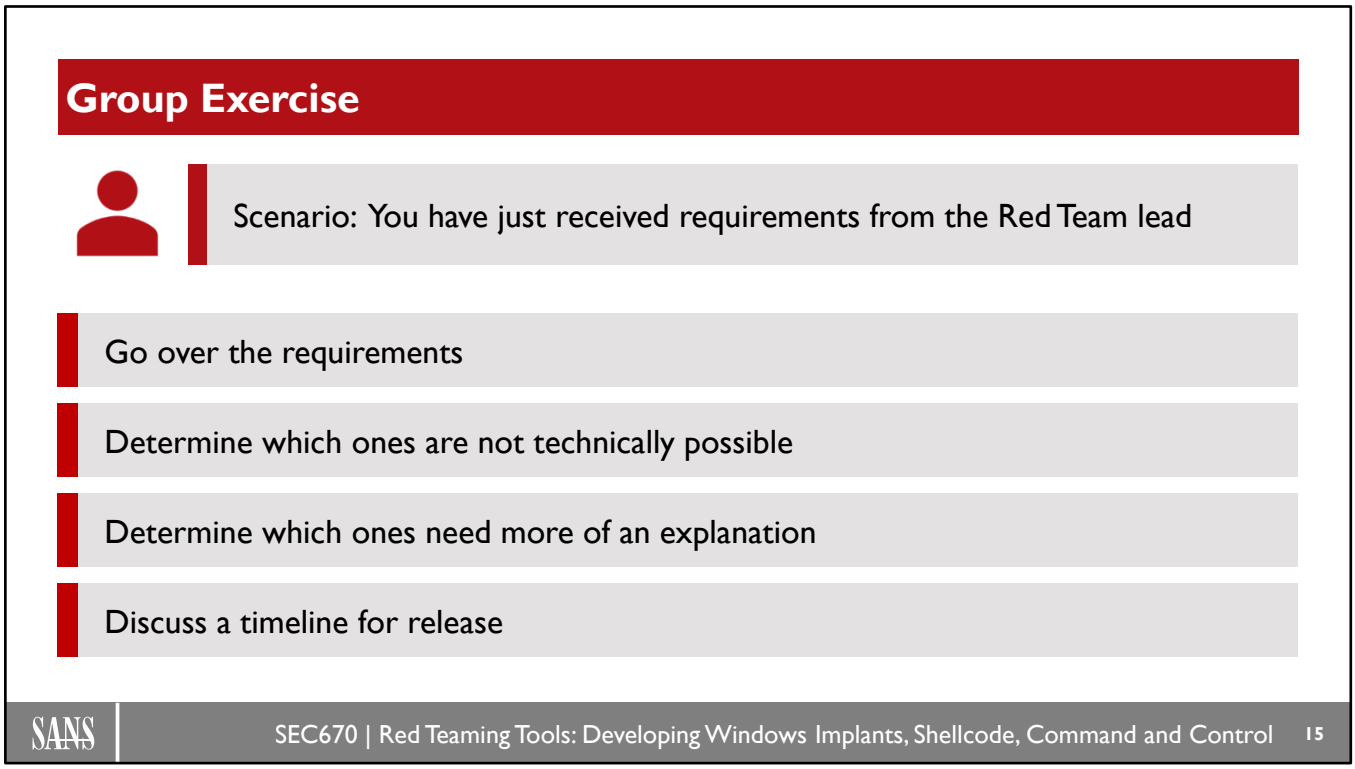
Requirements

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
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

Section 1

Course Overview

Developing Offensive Tools

Developing Defensive Tools

Lab 1.1: PE-sieve, Lab 1.2: ProcMon

Setting Up Your Development Environment

Windows DLLs

Lab 1.3: HelloDLL

Windows Data Types

Call Me Maybe

Lab 1.4: Call Me Maybe

SAL Annotations

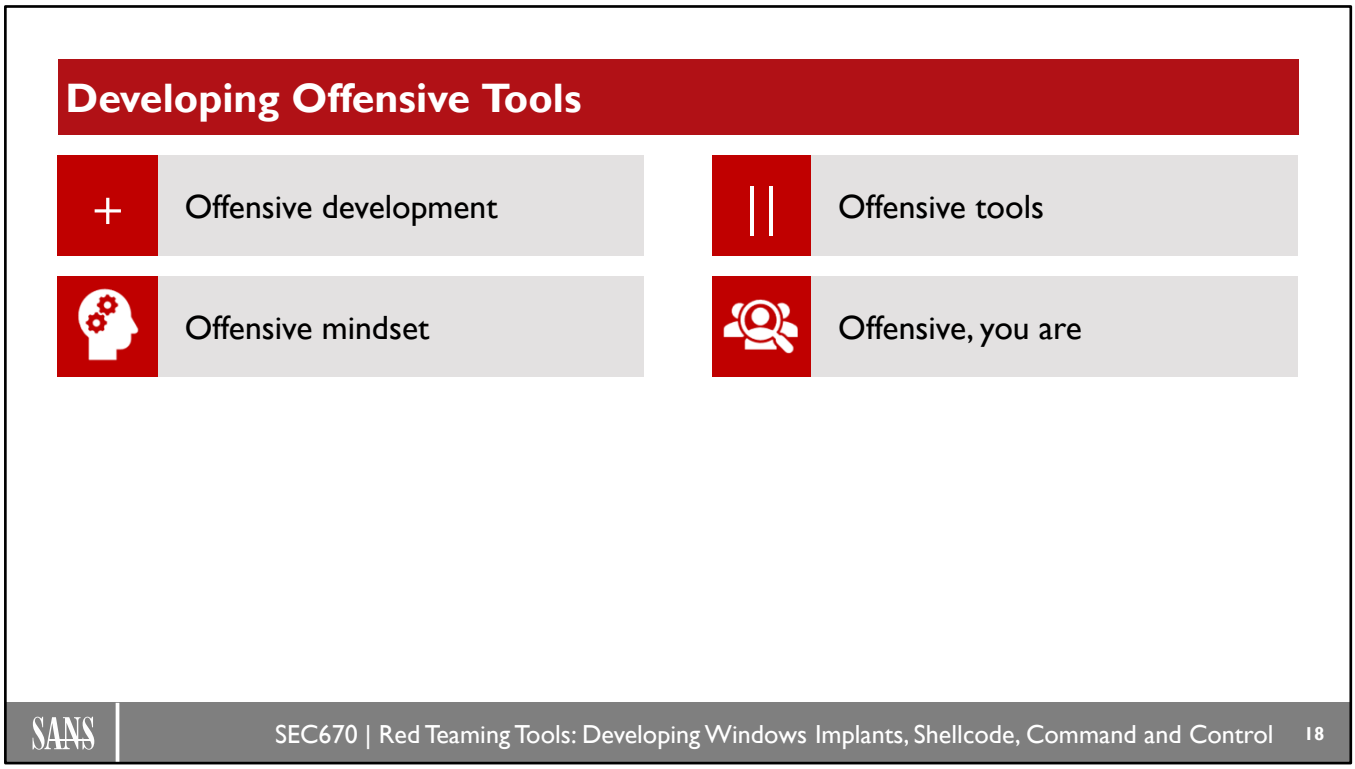
Lab 1.5: Safer with SAL

Windows API

Lab 1.6: CreateFile

Bootcamp

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



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

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*

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

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.

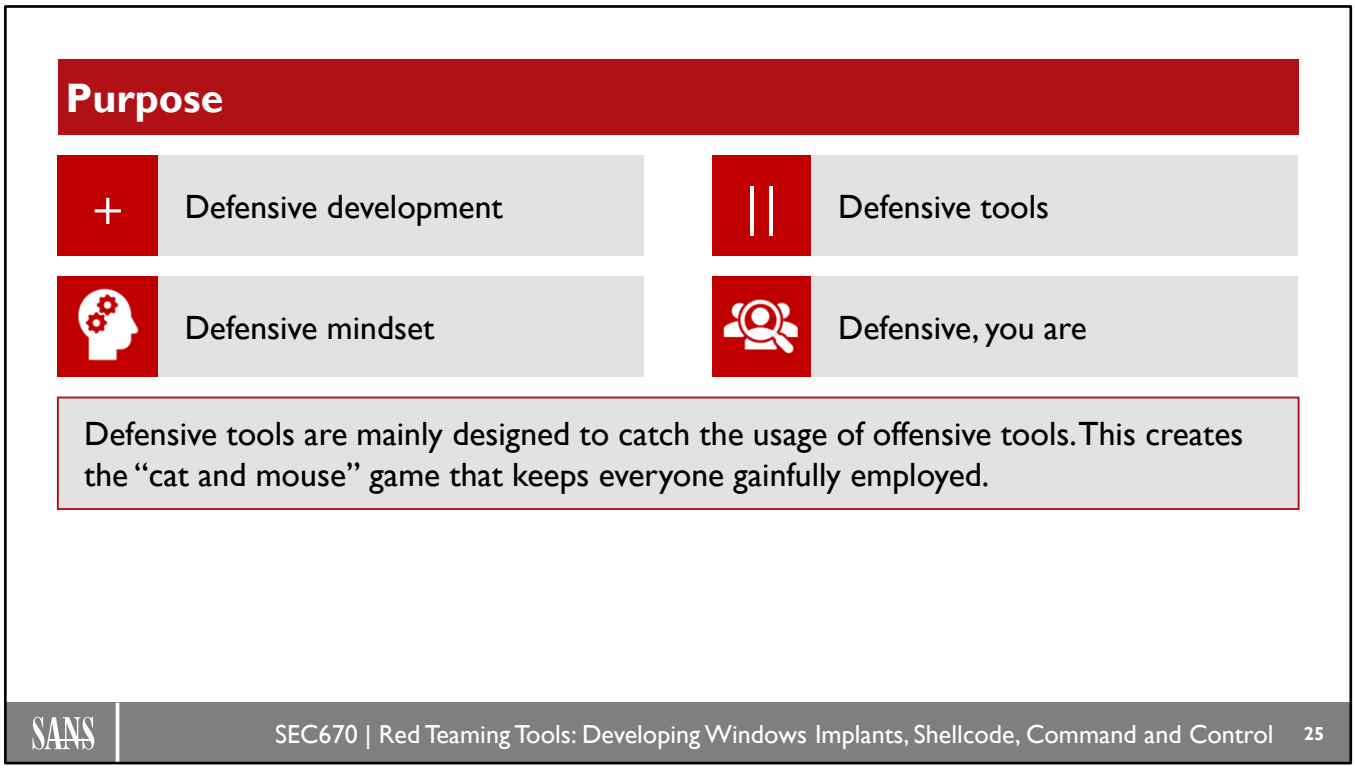
Course Roadmap

- **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 I

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

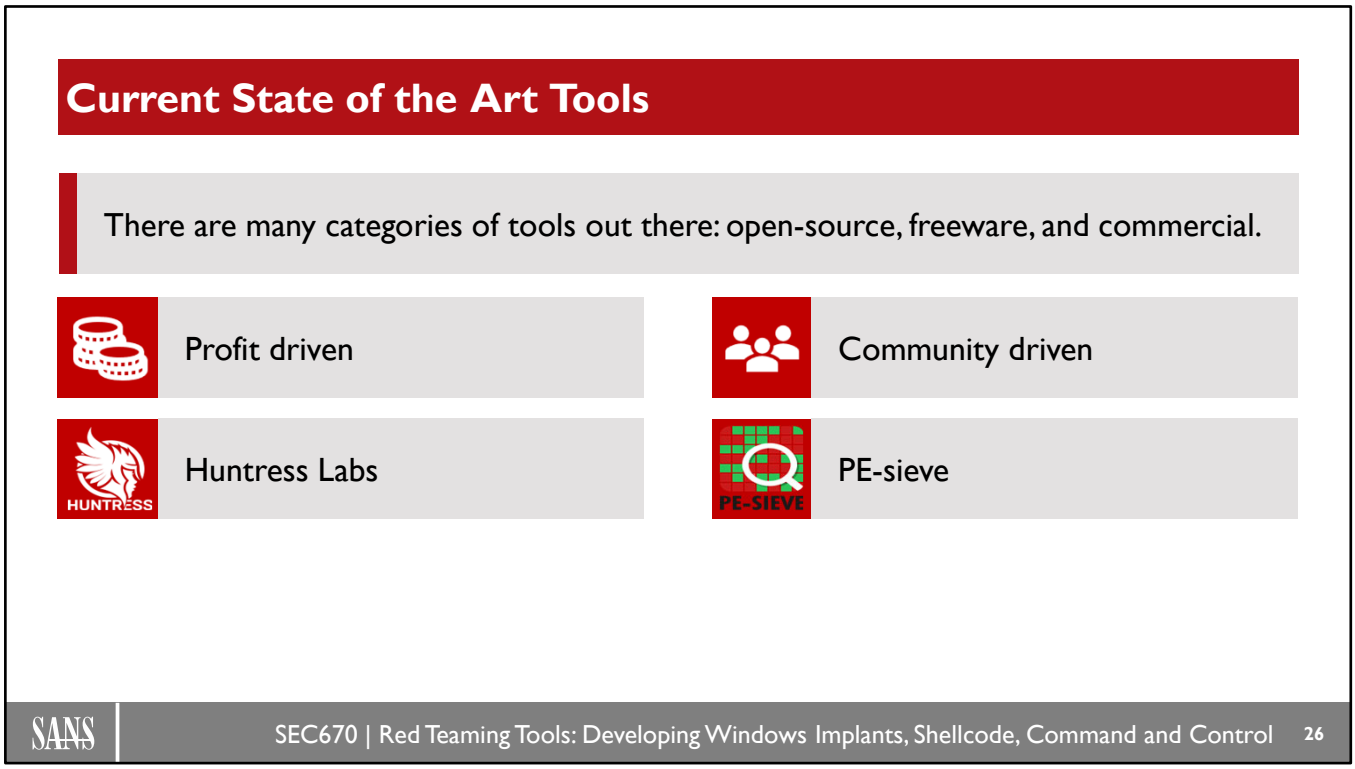
In this module, we will discuss the development of defensive tools, their purpose, state of the art tools, their future, and how you can contribute.



Purpose

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

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.

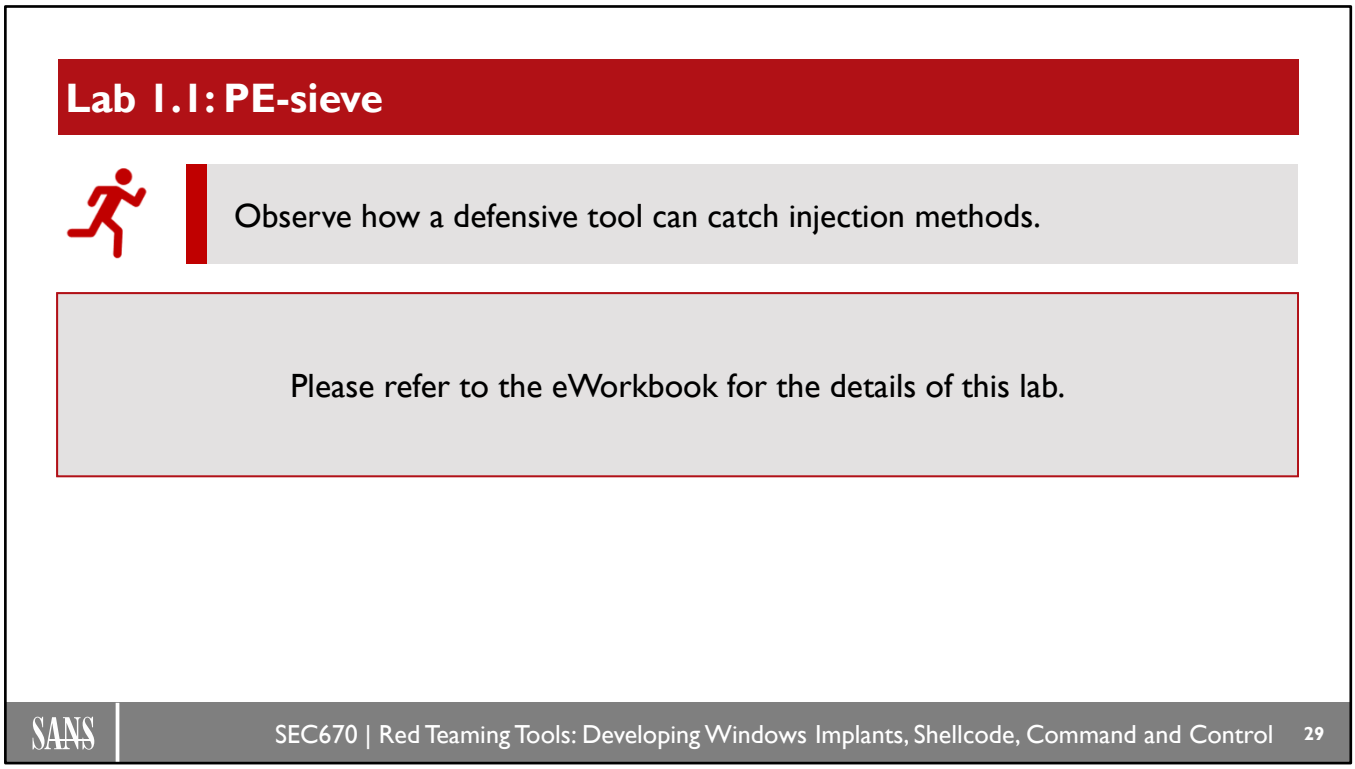
How You Can Contribute

- 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

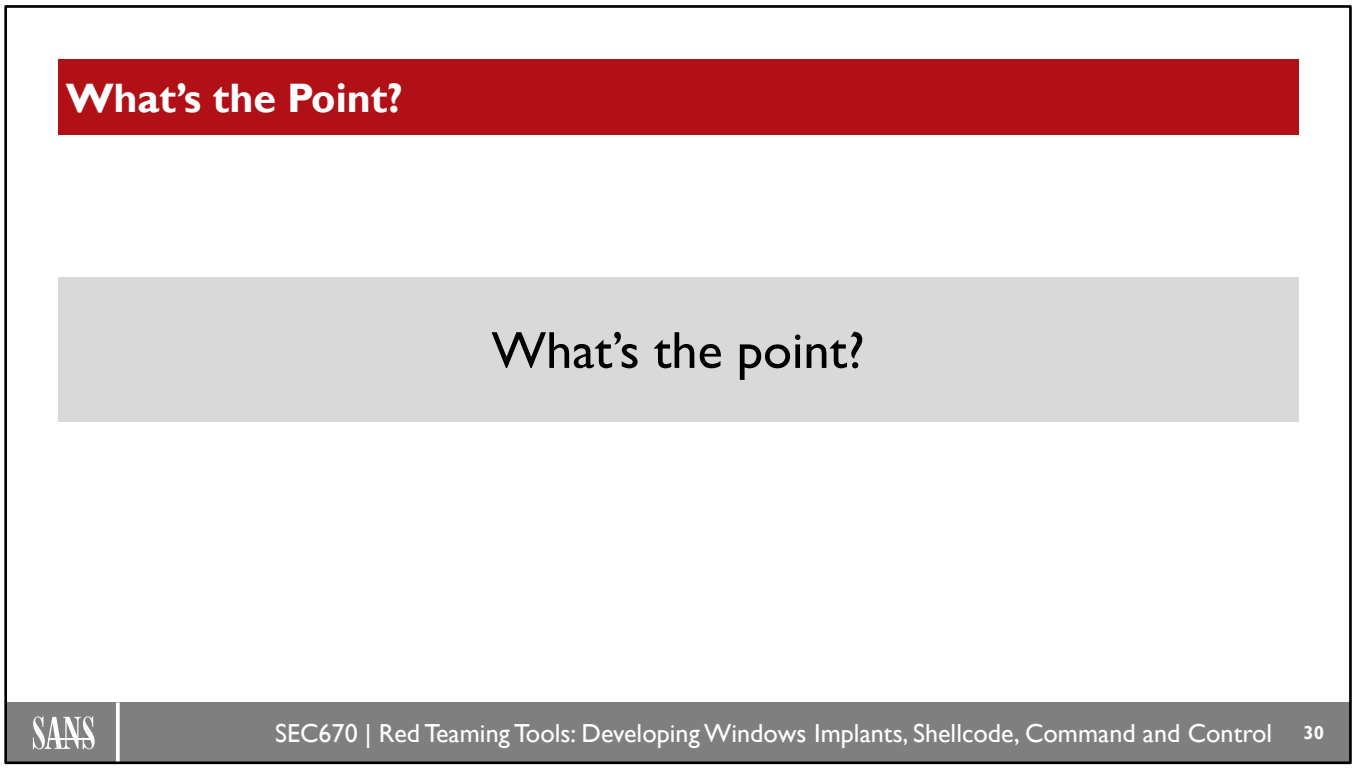
There are many ways you can contribute, and GitHub is just one of the many ways for you to share your code with the world. Once you make a few projects you can start to spread the word about it and share the link to your repos. Many developers will write a short blog about their tool, how to use it, 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.

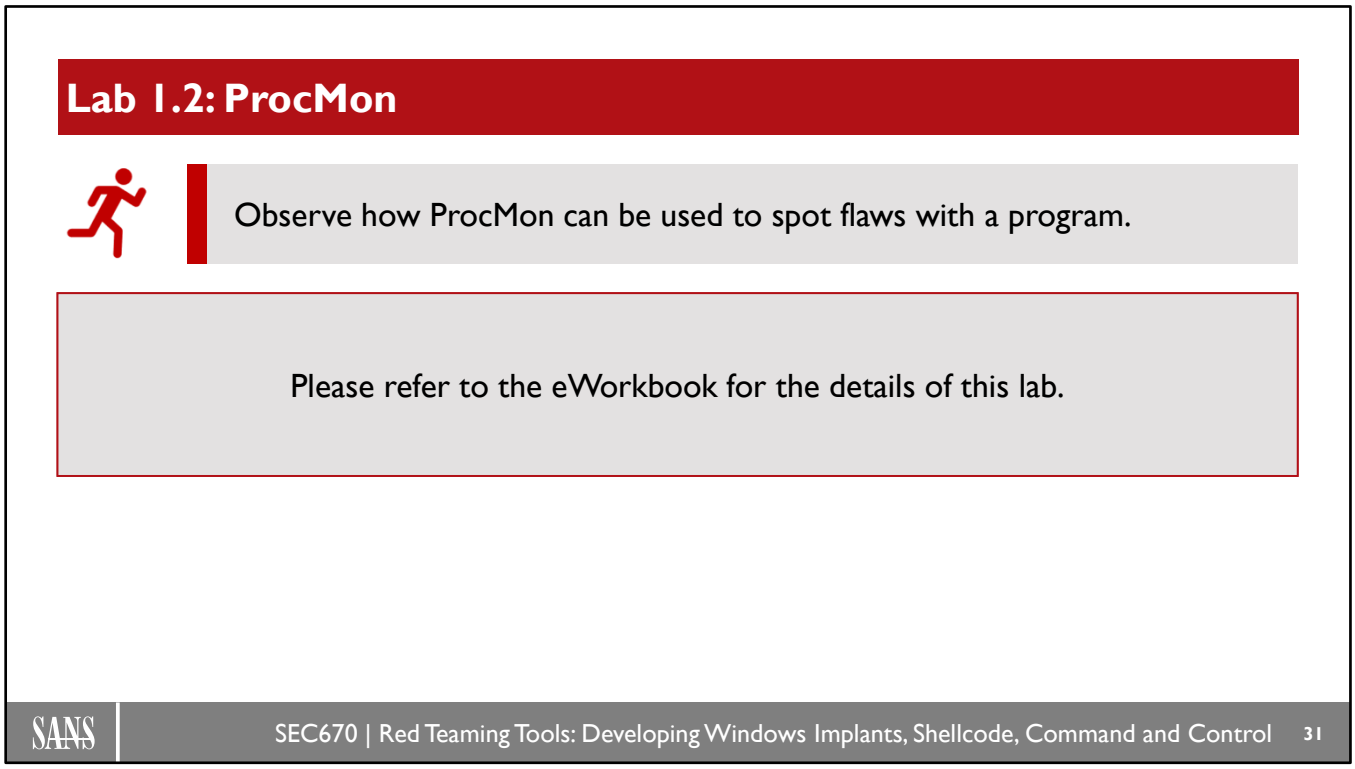


Lab 1.1: PE-sieve
PE-sieve, according to hasherezade's GitHub repo, "is a tool that helps detect malware running on the system, as well as to collect the potentially malicious material for further analysis." The tool is designed to scan a single process, but it does a great job at detecting various items 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?
The point of the lab was to become familiar with defensive tools that were made to detect the effects our offensive tools. PE-sieve is one of many tools that has this kind of capability.



Lab 1.2: ProcMon
Process Monitor is one of the tools that comes bundled with the Systinternals Suite and is great for seeing what a process is doing when it starts up. ProcMon also has the ability to monitor what is happening when the OS starts.

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

What's the Point?

What's the point?

What's the Point?
The point of 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

Module Summary
In this module, we discussed several reasons why it is necessary to develop defensive security tools. We also discussed a few tools like PE-sieve, the current state of the art tools, and the future of more advanced tools with better AI/ML integration.

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

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss how to properly create a development environment.

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 (1)
The VM you downloaded for this course should already have everything you need to develop some code and follow along with the labs in the course. Some of applications like prompting you for updates, but please do not install any of them. The labs were created and tested for what is already on the VMs, but if you would like to do so, you can take a lot of what is done in this course and apply it to older versions of Windows, like Windows XP.

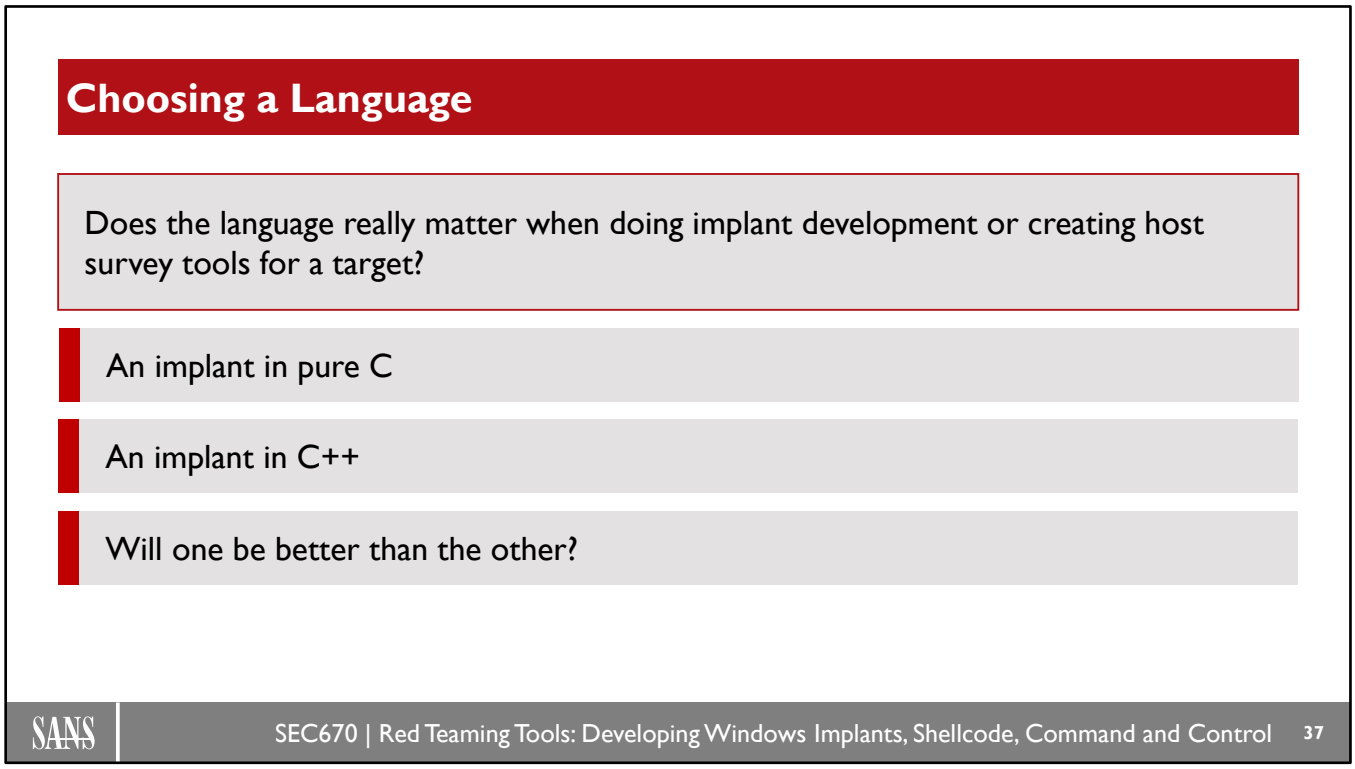
Setting Up Your Development Environment (2)

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

Setting Up Your Development Environment (2)
The environment you choose to develop your tools with is important. There are some who like to do Windows development on Ubuntu, but I would not recommend it. The Windows eco-system will be perfect for your Windows development needs. Getting started, you will obviously need some sort of IDE. This course is using Visual Studio Community as it is the Cadillac of IDEs. Also, it is very likely that your company will provide you with Visual Studio Professional or even Visual Studio Enterprise.

On top of this, you will need to install the Windows SDK and the C/C++ build tools at a minimum. Visual Studio has a very robust debugger built into it, but you can also install WinDbg, or WinDbg Preview, as it is the de facto standard for Windows debuggers. Another item for consideration would be to install the legacy WinDDK so that your tooling can target WinXP. Let's face it, WinXP is not going away anytime soon, so you might come across a WinXP target at some point.



Choosing a Language

Let us say for a moment that you do not want to use C to develop your Windows tools, which is fine as there are many compiled languages to choose from today. Many tool developers today are experimenting, and with great success, using Rust to create offensive tools and shellcode. Others out there are using C#, C++, Go, etc. How do you choose? Perhaps you do not have to choose just one and this enables tool diversity and having different signatures. This course will only focus on C/C++, but it would be a good idea branch out with other languages. A great way to have a diverse toolset is to use different languages. If your previous 10 tools have been done in C, think about changing it up a bit and use C++. Tool diversity can not only allow your tools to go undetected longer but also make them harder to signature and attribute back to you. On the note of signatures, having the same tool signature over and over is not good and it is a great way to get caught.

Having multiple options is great for implant development because it gives you the option to create the same functionality with those various languages. Should you choose C++, like this course does, you can create the same tool but use different compilers. The various compilers used can change the signature of your capability. Although using different compilers is great, this course will just use Microsoft's compiler that comes with Visual Studio community.

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

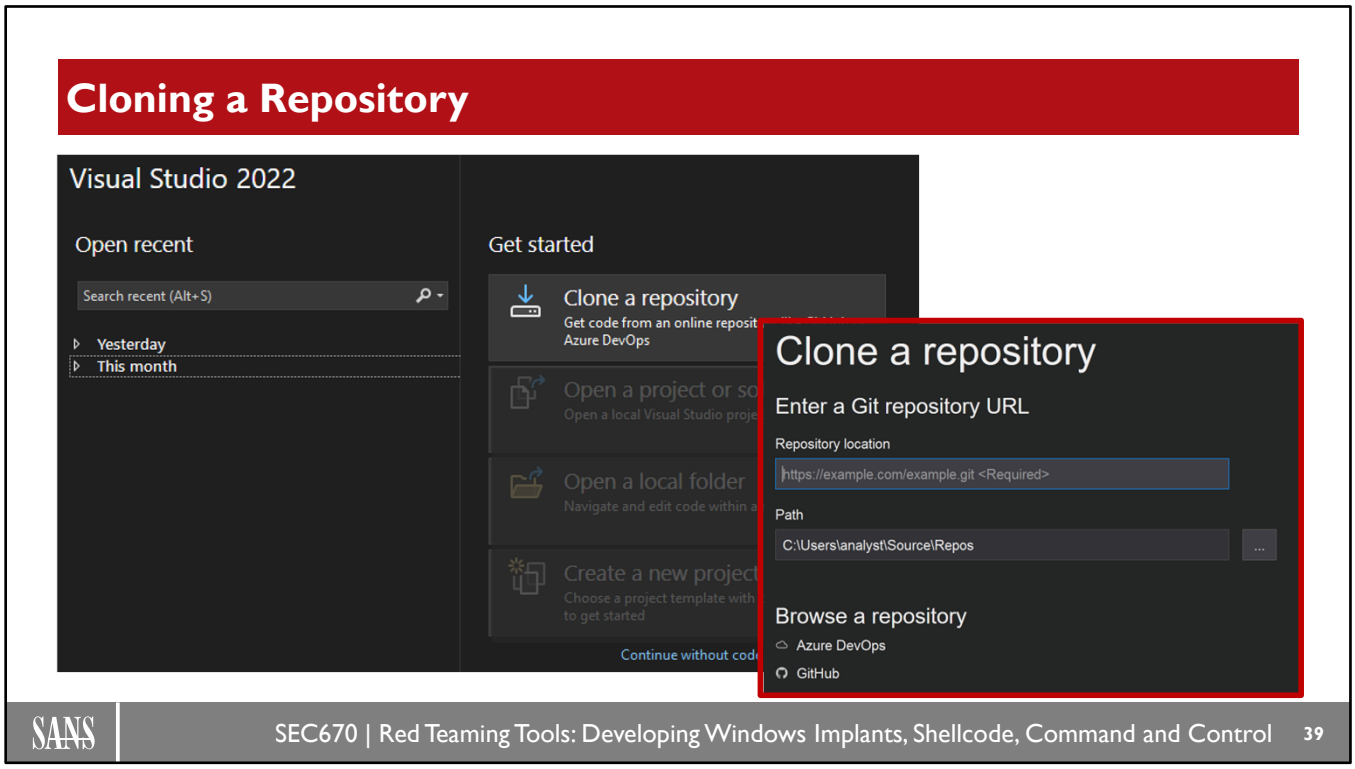
Choosing an IDE

If you thought choosing a language could be a hard choice, then choosing an IDE could prove to be even harder. Open-source IDEs are great, and they do have their place and purpose. Certain IDEs can be geared for a specific language in mind but will still support other languages and platforms. Xcode is one such IDE and it is amazing, but you must have a Mac. You can still develop in C/C++ with Xcode; however, the primary purpose of Xcode is for iOS, macOS, etc. development, so it's probably not an ideal choice for Windows development.

Code::Blocks and Visual Studio Code are open-source IDEs that are very good. Code::Blocks got an update in March of 2020 and is a decent choice if you are just getting started. Visual Studio Code is getting better with what seems like each passing month. There are many extensions that can be installed to support more languages and with some tweaking it can feel and behave very similar to Visual Studio Community.

This course will be using the Community version of Visual Studio. Visual Studio Community is free, and for the purposes of this course, it is just as powerful as the paid for version of Visual Studio Professional. As an individual developer, we can do just about anything the professional version can do. The time to move to the professional version would be when you are joining a team of developers that are working together to create a product that will be deployed on production systems. The previous statement is especially true from the perspective of offensive tool development. From here on out, this course will simply refer to offensive tool development as implant development, or implant dev for short.

As an individual developer, Community gives you the power to use intellisense, code completion, code analysis, memory layouts of structures, parallel threads stack viewing, and so much more!

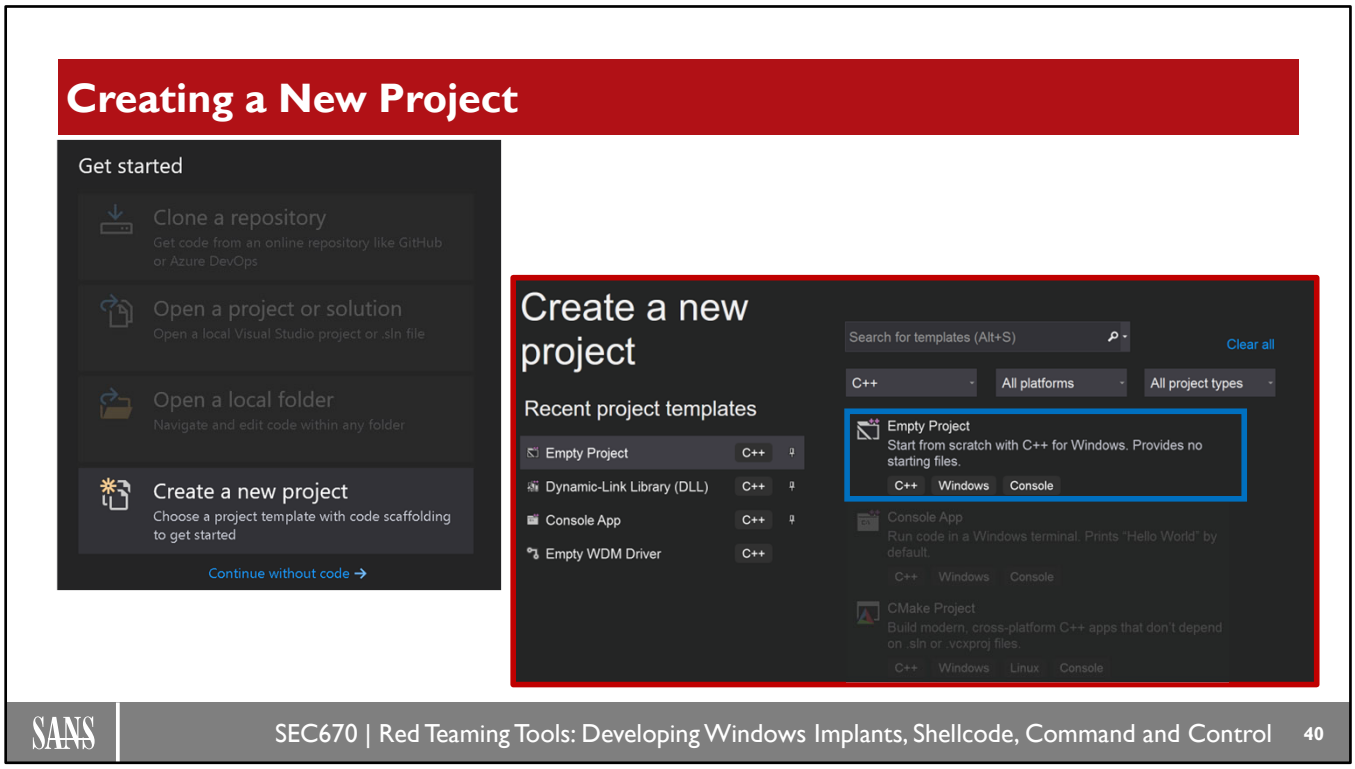


Cloning a Repository

This is one of the first windows you will see each time you open Visual Studio. If this is your first time opening Visual Studio on this system after a fresh installation, you will not see anything listed under "Open recent". As you begin to create new projects and tools, they will start to populate your recent items list, allowing you to quickly jump back into a project right where you left off previously.

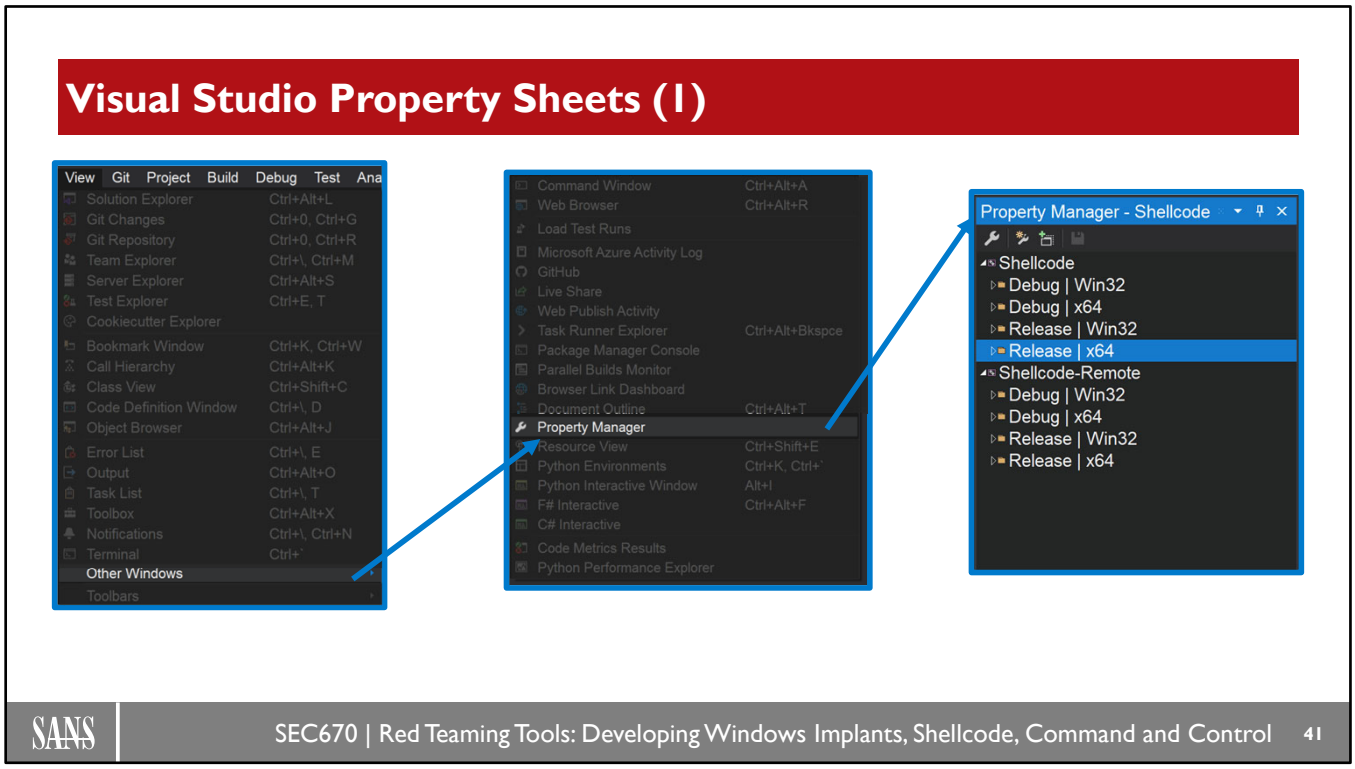
Since version control is built-in, you can clone a repository from either GitHub or Azure. This course will not use Azure but will have two (2) local git repositories on your Dev-VM: skeleton and main (please see your lab setup instructions for details). If you click on the "Clone a repository" button, you will be greeted with the next window that prompts you to enter a few details about the repository of interest. The first one is the link to the project to clone, and the second is the local path where you would like the cloned repository to reside on disk. This feature makes it so easy to clone a project shared by a friend or colleague and immediately begin working on it and make it your own.

The other options on this main menu page are "Open a project or solution," "Open a local folder," and "Create a new project." We are interested in creating a brand-new project, so let's go ahead and look at how to do that.



Creating a New Project

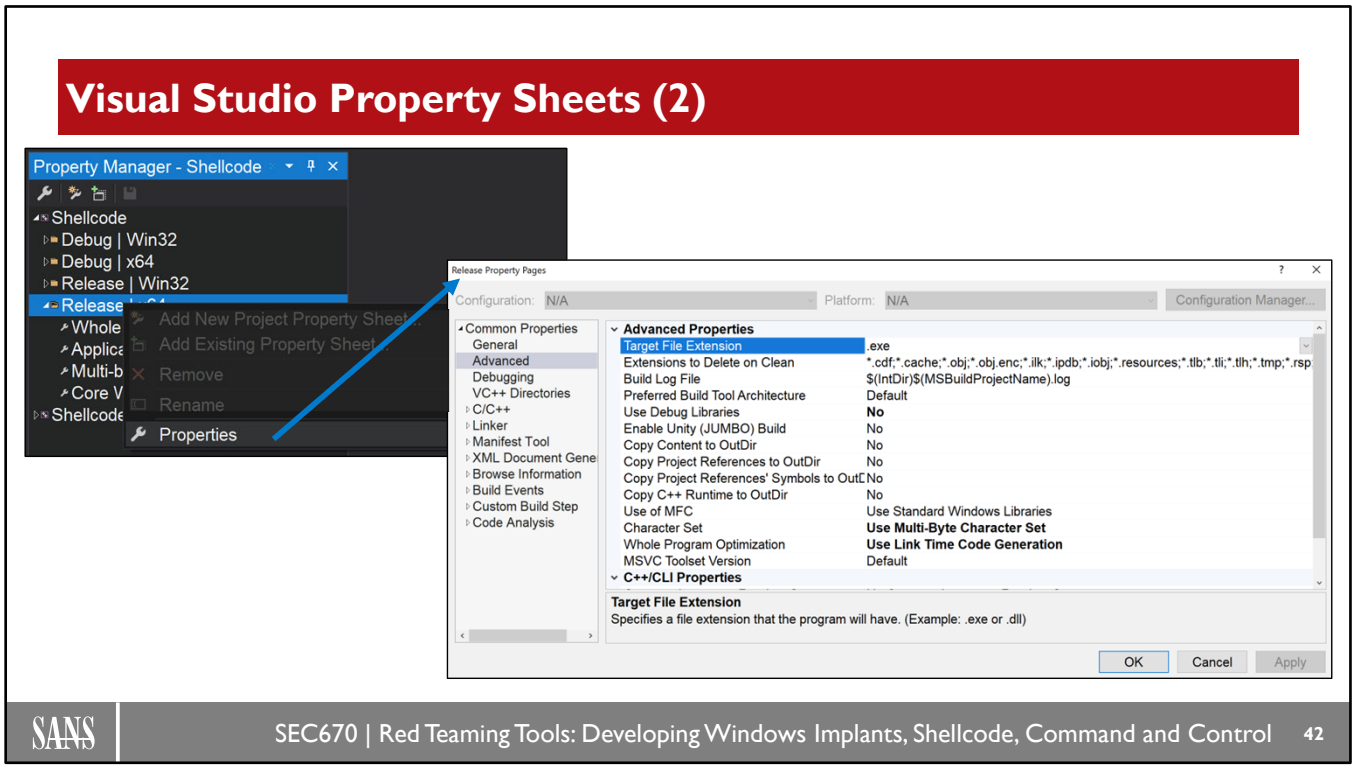
When you are first starting out, you will need to choose "Create a new project" from the menu. When you select it, you will be greeted with a new menu that will display several items to you. Anything you have worked on previously will show up under the "Recent project templates" area. One of the nice features about that area is that you can pin some templates there if you know you will be working with those often. For this course, your primary two (2) project templates will be the "Empty Project" and the "Dynamic-Link Library (DLL)" templates. You can optionally search for certain templates as well as filter on a specific programming language like C++. A search term you could enter could be "empty" if you wanted to see a list of project templates that have the word "empty" in them. Once you choose your project template another menu will be shown, and you will then choose the name of the project and where you would like the project to be saved.



Visual Studio Property Sheets (1)

One of the amazing features of Visual Studio is the ability to have common project settings that can be shared among several projects. You can also share your common project settings with team members who are working on a similar codebase. This feature is called property sheets, or prop sheets for short. Property sheets have a .props file extension and can be created for each project. To explore the property sheets for your current project, navigate to the "View" menu and from the drop down, look for the "Other Windows" option. Hover over "Other Windows" and wait for the next submenu to show. From there, look for the "Property Manager" option and select it. Once you do, the Property Manager window will show up somewhere. It might show up alongside your "Solution Explorer" window, but you can move it and snap it anywhere you would like. Looking at the new "Property Manager" window, you can observe the options that are available here. This screenshot is of the Shellcode project which is a lab under Section 5. The project you open does not matter; the prop sheets will be the same across the board at this moment.

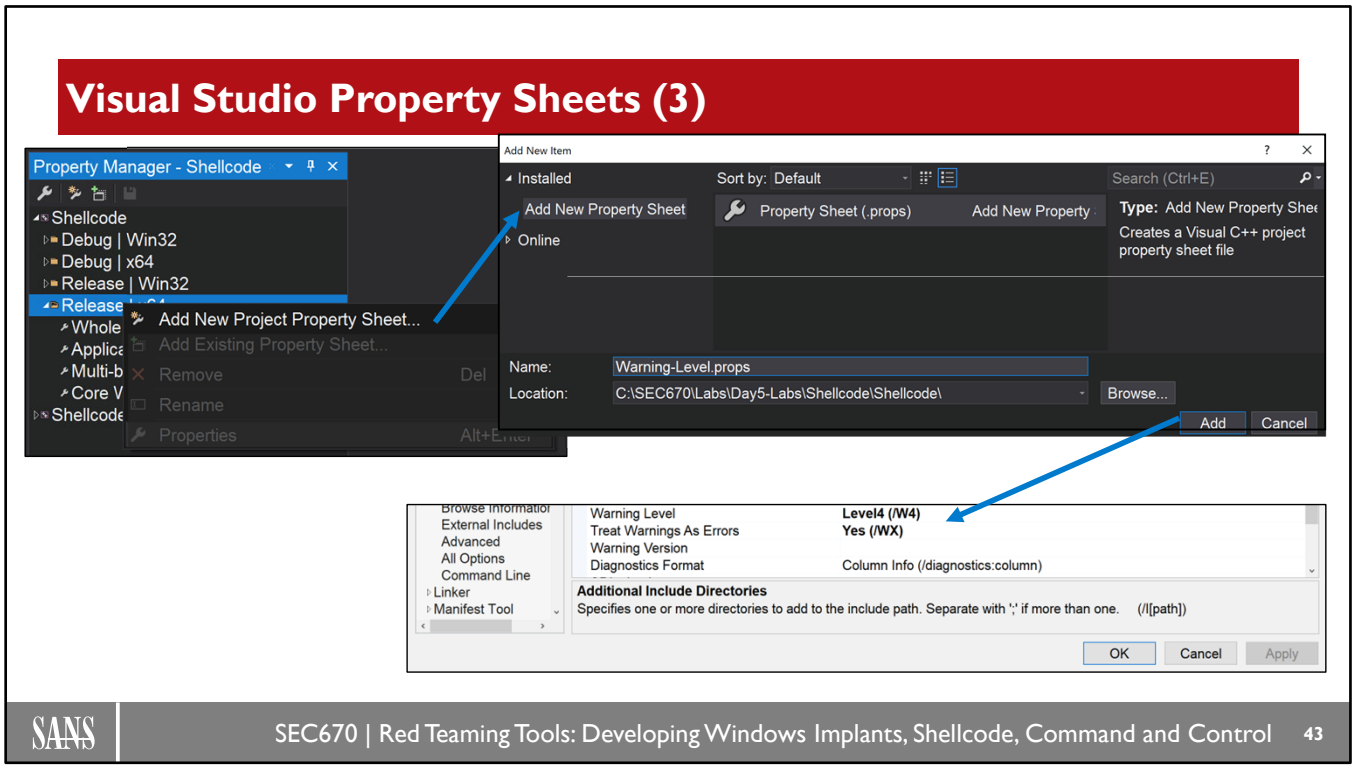
Expanding the drop-down arrow to the left of the project name, Shellcode in this instance, a listing of settings will be shown that are specific to the different builds you can have for your program. For example, you can select the "Release | x64" prop sheet to explore what settings have been configured for the Release and 64-bit platform. Once a specific prop sheet is expanded, you can look at what settings are currently configured.



Visual Studio Property Sheets (2)

When you are ready to start making changed to a project's configuration, you can navigate to the "Property Manager" window and right-click on any of the configuration nodes listed, like "Release | x64" in this example. Once you right-click, choose "Properties" from the submenu and the "Release Property Pages" will be displayed. It is here that you can start to change certain project settings. As you explore the "Release Property Pages", you might see some settings that are emphasized in bold. When you see something in bold, it means that specific configuration setting only applies to the current local context. Otherwise, it would be normal font indicating the setting is inherited from a parent project or is simply a default setting that is made when the project was initially created. Again, you can modify settings here but a more preferred way to take full advantage of prop sheets is to create a new prop sheet, and then make your configuration changes. To do that, you would choose "Add New Project Property Sheet" instead of choose "Properties".

The prop sheets will have a specific order to how each one is applied to a certain configuration node. Staying with the configuration node "Release | x64", you can see the existing prop sheets that Visual Studio created for you when the project was created. The order in which the prop sheets appear in the "Property Manager" windows is the order with which the compiler will evaluate them. So, prop sheets that appear later are evaluated later in the build and will then override any values the previously evaluated prop sheets may have set. If you do not like the order in which they appear, you can simply move the prop sheets around as you see fit. Moving prop sheets around is done by right-clicking on the prop sheet you want to move, and then choosing "Move Later In Evaluation" or "Move Earlier In Evaluation". Please keep in mind that the only prop sheets that Visual Studio will allow you to move around in the evaluation order are the custom ones that you make. The ones that are made by default by Visual Studio are permanently stuck in their existing order.



Visual Studio Property Sheets (3)

As a very quick example, say you wanted to make a prop sheet named "Warning-Levels" that changed a few settings regarding the warning level when the project is being built. From within the "Property Manager" window, right-click on the configuration node of your choosing, in this example, "Release | x64", then select "Add New Project Property Sheet". A new window will open, asking you to name your new prop sheet. It would be recommended to give it a meaningful name so you can quickly get an overall idea as to what settings might be configured. In this example, the prop sheet name will be "Warning-Levels". After you select "Add", a new window will open titled "Warning-Levels Property Pages". It is within these pages that you can navigate to the appropriate sections to make your configuration settings. Since this prop sheet is named "Warning-Levels", we will set the **Warning Level to Level4 (/W4)** along with the **Treat Warnings As Errors** to **Yes (/WX).** Let's break down what those settings do to the project.

Setting the warning level to four (4) will show all warnings at the levels <4 along with all the informational warnings at level four (4). This is a setting that is highly recommended, especially for brand-new projects since it can reduce the amount of time it might take to locate some annoying, hard-to-find bugs. The other setting is **Treat Warnings As Errors**. This one was set to **Yes (/WX)** because the first time we compile a brand-new project, we want to pinpoint any bugs in the code early on and with as much accuracy as possible. It might make building your projects a bit more annoying because you might have more errors popping up, but once you resolve them, the result should be a much more polished code base.

Now that you have a general idea of how to properly make prop sheets, you can make as many prop sheets as you would like for all your projects. Once you have them all created, bringing in existing prop sheets is as simple as clicking a button. The "Add Existing Property Sheet..." button does exactly what it sounds like. You can also share them with team members or anyone else using Visual Studio, which is extremely helpful when someone else is attempting to contribute to your code or if they are having issues compiling your codebase.

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.

Your Testing Environment

Testing your tools on the same system you use to develop them is not a good practice. Even if you were not developing offensive tools, it would not be a wise decision. Since this is an offensive development class, we want to make sure your testing environment is dedicated and not the same as your development environment. For this very reason, you were provided two primary Windows images: a development VM and a test VM. The other images will be used later for AV evasion.

With offensive tools, if you are working on a capability that modified sensitive or critical parts of the registry, then there is a chance you could place your machine in an unusable state, forcing you to restart or revert the VM. Even worse, if you are working on ransomware that you kicked off on your development image and it started to encrypt everything. Of course, you would have the key to decrypt it all, but it would be annoying. Using a virtual testing environment provides you with the ability to take a snapshot of it so you can revert to a known clean state before you would kick off another test. Pro tip: Remnants from other tests could possibly ruin future tests.

You should test your tool many times, hundreds or even thousands of times. You want to be sure it will work all the time and you also want to know what might break it. With testing, you must validate that your tool did what it was supposed to do. If your tool was supposed to create a user as a backdoor, create a registry key, drop a file, and delete a file, then how do you make sure it did those things? Validate your tool before you use it on an op. We will not be doing advanced testing here, but we will be testing the tools you develop during class on the testing VM provided to see if they work the way they should.

Build Targets

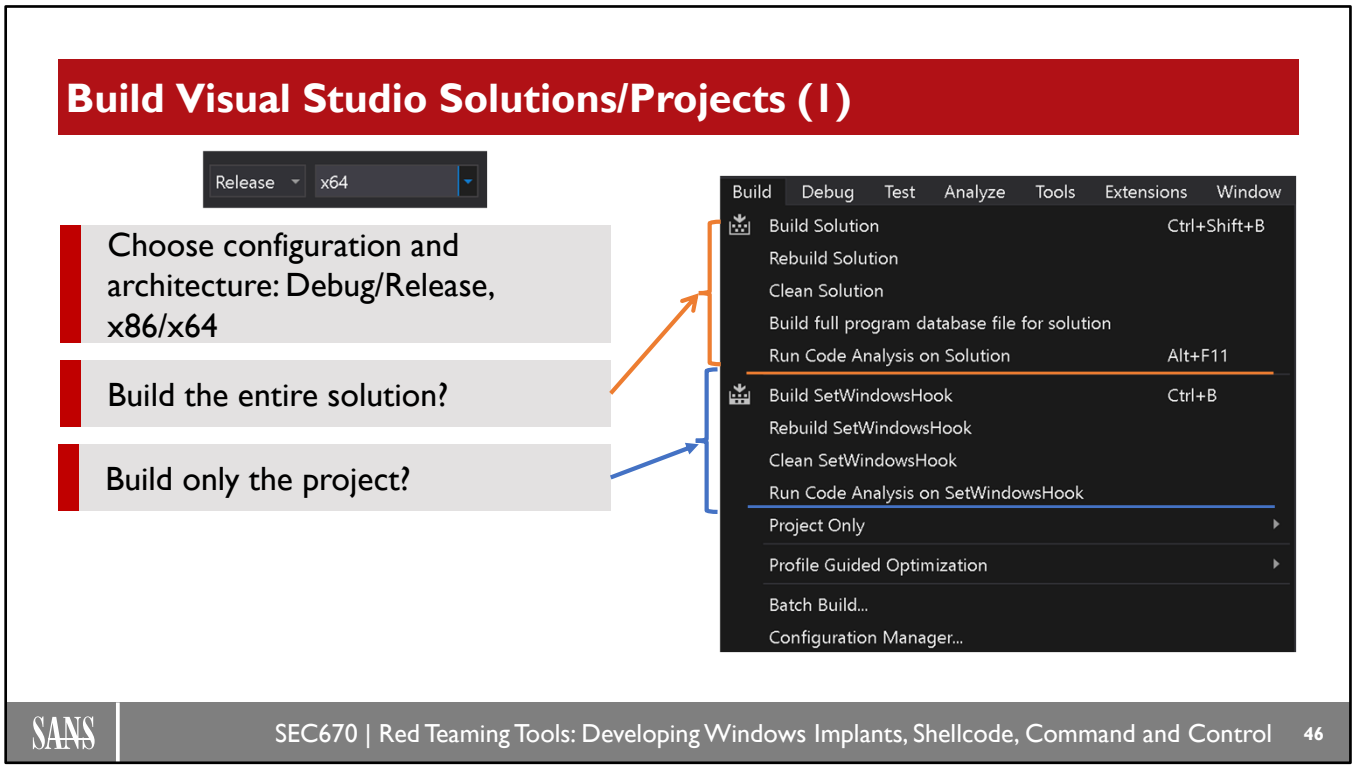
Even if you develop on the most recent version of Windows, it does not mean you cannot build a version that targets an older version like Windows XP.

- 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 Targets
This can kind of go along with testing since you would be testing your tool on every possible target in mind. If you are developing a capability that targets WinXP x86, then you better have that same image in your test environment. Perhaps you are hoping to develop a universal capability that can be used against almost every version of Windows from XP and beyond. Sometimes you want your tool to be as specific as possible so that maybe it only executes on your true target and not just any system your tool happens to be installed on.

When developing a tool that is expected to run on more legacy systems, care must be taken to not use APIs that would not have been available for that OS version. The most recent versions of Windows 10 will certainly have APIs that were not available for use on Windows XP. This must be thought out ahead of time when discussing potential targets.

When it comes to building your tool, there are several build settings for your Visual Studio project such as Debug and Release versions. The Debug and Release versions can be built for x86 and x64 as well.

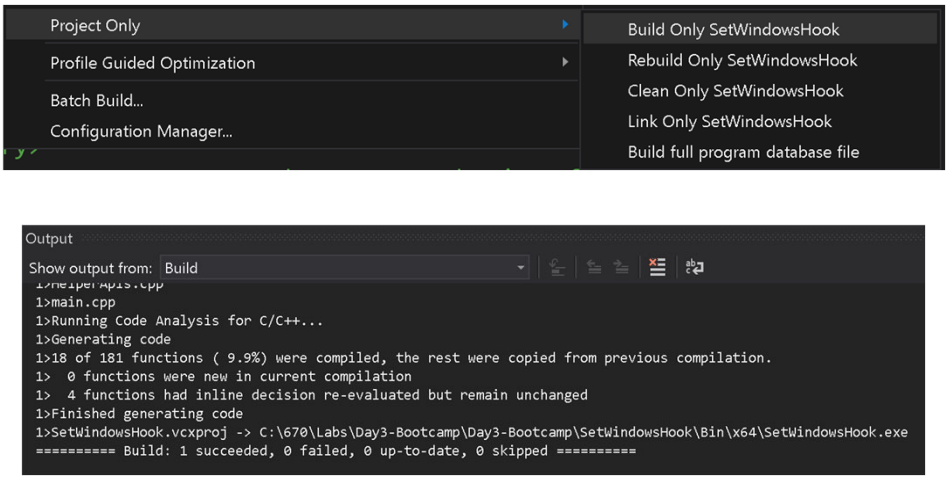


Build Visual Studio Solutions/Projects (1)
After you are done with your code development, you can build your solution or project. You can choose a configuration setting from the menu bar as either Debug or Release. Debug builds will be larger in size, but that is because they are built with more information to aid you in debugging any issues your program might have at run time. The Release configuration will be optimized, your program size should be smaller, and run more efficiently. By default, both configurations will create PDB (Program Database) files that hold debugging information about that program like its symbols. After that selection is made, the choice of x86 or x64 can be made. You might have a requirement to compile for both x86 and x64. There are quite the number of options when it comes to building your projects. We will get into some of those options later in the course, like how to cut down on size, and eliminate dependencies like the C runtime (CRT).

Within the Build dropdown menu, there are more options available to you. The first section is specific to the entire solution and a solution can be quite large, holding any number of projects. There have been some solutions that hold over 100 projects! Building the solution will build every project in the solution. If you are not interested in building each project in a solution, you can simply build a specific project. The project only also comes with its own set of options. The example on the slide shows a single solution that you can build but it also shows you the current project being worked on in that solution. The project here is called SetWindowsHook, which is a project you will be working on later in the course.

If you have ever messed around with Cmake or Makefiles, the clean solution and clean <project_name> might sound a bit familiar to you. The clean action in Visual Studio behaves very similarly to issuing the following make command: make clean. When you clean a solution or a project, any intermediate files that have been created will be wiped away. This will also include any compiled binaries like EXE, SYS, DLL, or LIB files, to name a few. The Rebuild option will not only build your solution or project, but it will clean everything first.

Build Visual Studio Solutions/Projects (2)



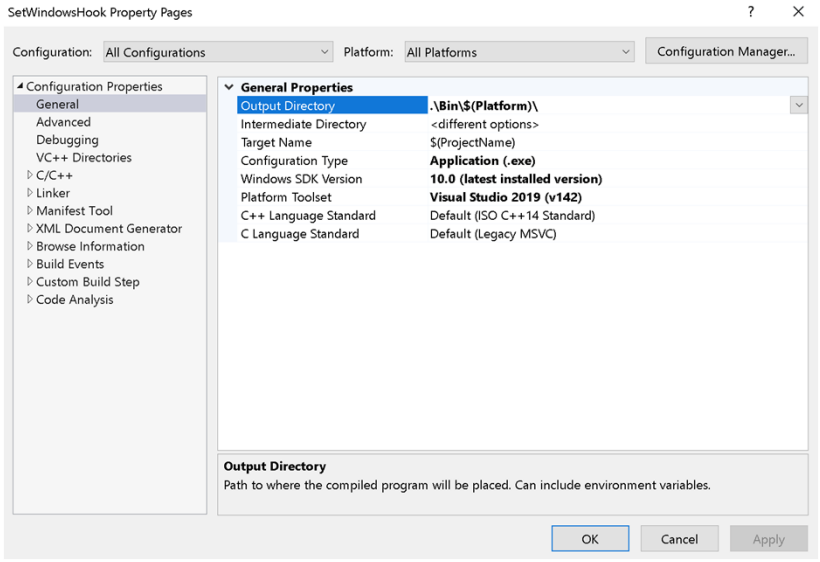
Build Visual Studio Solutions/Projects (2)

For Project Only, there are several choices like build, rebuild, clean, link, and build full program database file. Once you have made your choice of building the entire solution or just a project, the Output window will appear with some information from the Build process. The output in the screenshot on the slide shows the full compilation for an entire solution. The solution was comprised of a single project called SetWindowsHook, and it was built for the Release version for x64. When making subsequent builds, the output will be slightly different depending on what was changed. The output will indicate how many functions were compiled, how many failed, how many were up to date, or how many were skipped for some reason. The output in the screenshot can be broken down as follows:

The list of source code files that are being processed
- Code analysis being ran
- The number of functions (18 of 181) being compiled
    - If this was being done for the first time, you'd see all functions being compiled
- The output path where the compiled binary is located

The numbers to the left of each line hint at what project number is being built tied to threads/logical cores of your CPU. This example was just holding a single project, which is why you are only seeing the number 1 on the left of each line. More complicated solutions that house several projects will have other numbers appear.

Visual Studio Project Settings



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

Module Summary
In this module, we discussed the importance of having a diverse toolset, picking the best IDE for you, testing your tools, and knowing what kind of systems you would be targeting.

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

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss Windows DLLs and how to make them.

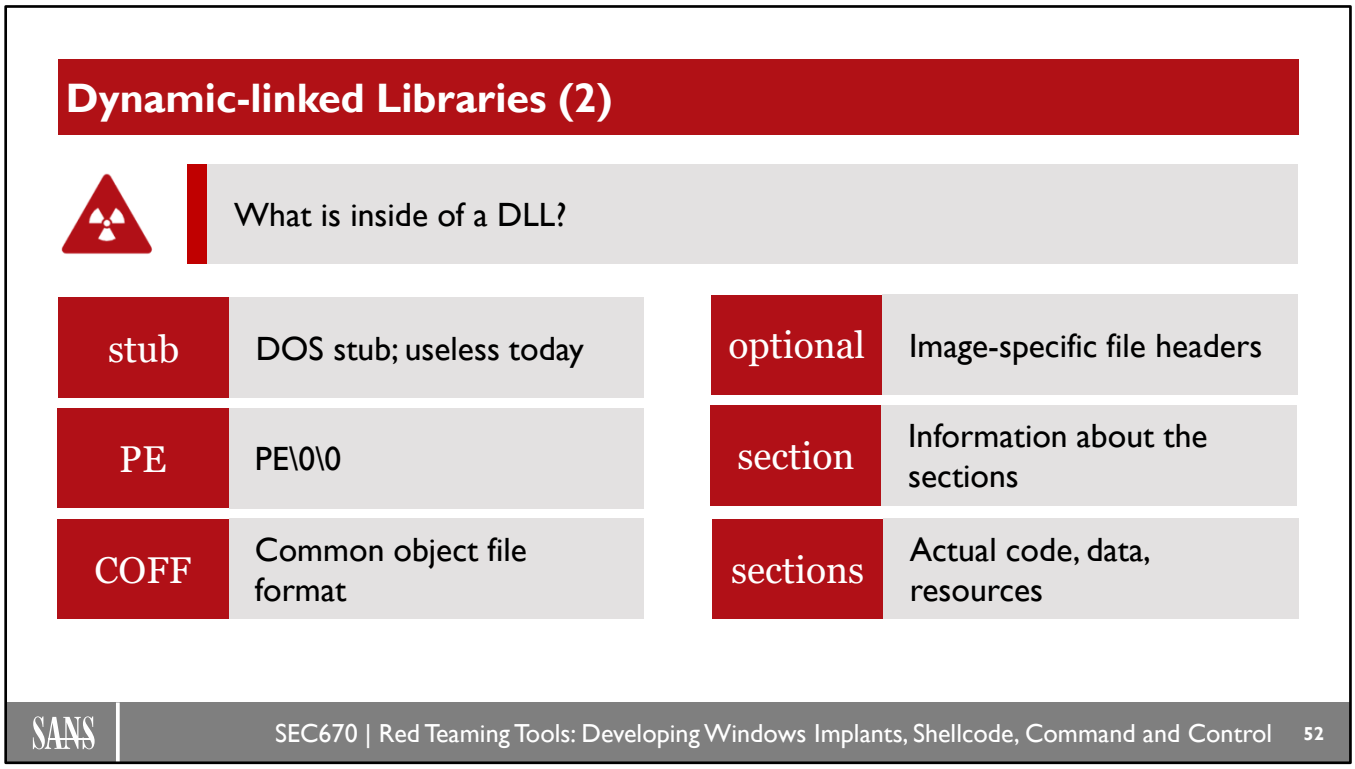
Dynamic-linked Libraries (1)

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 (1)
DLLs have been around in Windows for a very long time, just like Shared Objects (.so) files on Linux. Some of these DLLs are probably older than you are. At its most basic form, a DLL is just like an EXE in the sense that it's a **Portable Executable (PE)** file. DLLs can be shared with any application rather easily while just a single instance of the DLL is in RAM. Decades ago, a developer had to be extremely efficient with RAM because of its limited size. Nowadays, RAM size can be extremely large because it is cheaper to acquire, and more is better. The processes today can also be quite large and have many DLLs being used. When you build a DLL project in Visual Studio, there will also exist a .lib file. The .lib file is the actual import library that is generated by Visual Studio on your behalf and has two pieces of information in it: name of the DLL and its exported symbols. The format of DLLs and EXEs are standard across the board, just like how SOs and ELFs have a standard format on Linux. Later in this module, we will go through a lab that creates a DLL binary. Both the compiled results, .lib and .dll, can be explored using dumpbin, which is a command line utility that ships with the SDK. Your Windows development VM already has the SDK installed. All you would need to do is open a visual studio developer command prompt.

For any Linux users who are brand new to DLLs, you can tie the .lib and .dll Windows file types to .a and .so Linux file types, respectively.



Dynamic-linked Libraries (2)
DOS stub: This stub has been around since MS-DOS version 2 and the only reason it is still present is to alert the user that the program cannot be run in DOS mode. Simple!

PE signature: Simple marker of PE followed by 2 NULL bytes: PE\0\0.

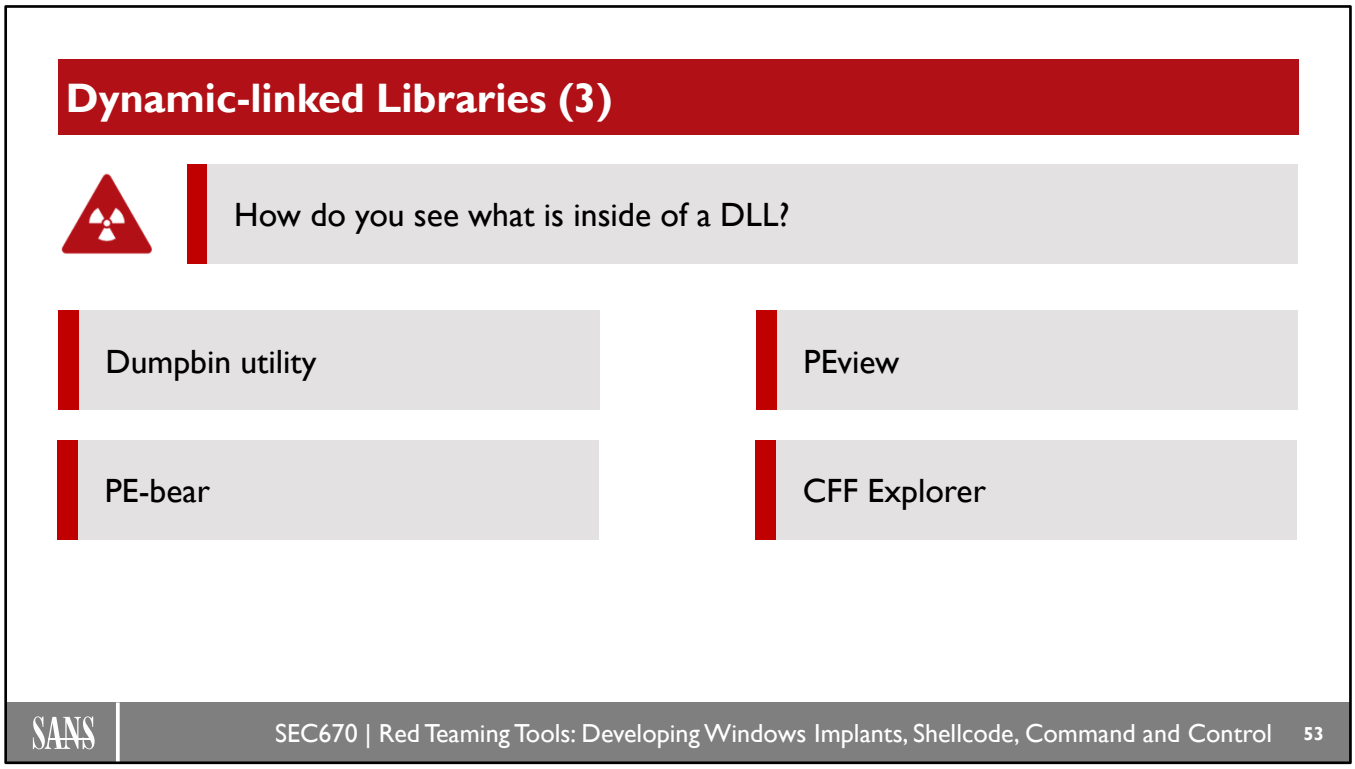
COFF: Common object file format. Used for holding various information like what machine types the program should execute, how many sections are in it, when it was created, pointer to the symbol table, how many entries are in the symbol table, what the size of the optional header is, and what characteristics the file has. Below is a short list of some characteristics a file could have.

- IMAGE_FILE_EXECUTABLE_IMAGE :: 0x0002 :: image file is valid and can be executed
- IMAGE_FILE_RELOCS_STRIPPED :: 0x0001 :: no base relocations. program must be loaded at preferred base address
- IMAGE_FILE_LARGE_ADDRESS_AWARE :: 0x0020 :: program can handle addresses >2GB
- IMAGE_FILE_DEBUG_STRIPPED :: 0x0200 :: debug info has been stripped
- IMAGE_FILE_32BIT_MACHINE :: 0x0100 :: based on 32-bit word architecture

Optional header: By no means is this optional. It is very much required since the loader depends on information from this section. The only file where this header is truly optional is for object files.

Section headers: This is a table filled of rows of section headers that immediately follows the optional header. The number of rows here is determined by the NumberOfSections field in the file header.

Sections: Where the code, data, and other resources are held.



Dynamic-linked Libraries (3)
There are a few tools available today that let you look at what is inside of a DLL. The tools do not just parse the structure of DLL files, but they can parse almost any type of PE file you throw at it. The dumpbin utility is a command-line tool that is typically available with a standard installation of Visual Studio. PEview is a GUI application with the bare necessities for viewing the file's structure. The headers are easily identified allowing for simple navigation through them. PE-bear is a rich GUI application that is full of great features. You can load several PE files at the same time and manually browse the file of interest. Tabs organize the structure breakdown and there is a nice high level navigation bar on the right-hand side that shows where you are in the file. CFF Explorer is yet another GUI application that offers similar features to that of PE-bear. Dumpbin and PE-bear will be used as the course progresses.

Over time, you will establish a preference as to what program you like most when diving into executable images, GUIs or command line. Another powerful utility not mentioned on the slide is WinDbg. WinDbg has a built-in command that will parse an executable image and dump the headers. It also allows you to manually parse the headers on your own with a combination of other built-in commands. Below is a snippet of executing the **!dh** command in WinDbg. The argument passed to **!dh** is the starting address of the **notepad.exe** image in memory.

!dh 00007ff6`29240000

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    8664 machine (X64)
       7 number of sections
D3FDE383 time date stamp Mon Sep 14 17:10:27 2082

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 (4)

As mentioned previously, the dumpbin utility ships with the SDK and as such, it is only available on your Dev-VM. Dumpbin offers a wide array of commands, everything from dumping headers to showing the disassembly of an image's section, like the TEXT section. Running the **dumpbin /headers** headers command will direct dumpbin to parse the entire image's PE header. The tool will provide verbose output similar to the output shown on the slide. The output shown indicates that it checks the validity of the image being parsed to ensure it knows the file format. If you were to give dumpbin a file it does not know the format of, like a Windows header file, it will return to you an error. The error typically seen is "**Invalid format file; ignored**". Once the PE signature is found, it will then check the file type. In this instance, the file type is a DLL. This flag comes from the characteristics field in the file header. It is simply a bit that can be set (1) or cleared (1). Then dumpbin will start parsing the file header, which is what is shown on the slide. There are a few important items highlighted in green: machine and number of sections. Here are what those fields indicate:

Machine – the architecture, rather the CPU type, the system needs to have. 14Ch is hex for IMAGE_FILE_MACHINE_I386, meaning 32-bit version of Windows. For 64-bit installations of Windows that might have an x86_64 CPU, you would see the value x8664 for IMAGE_FILE_MACHINE_AMD64. There are over two dozen other values that could be present here.

Number of sections: this value indicates how many basic units of code or data the PE image has. Sections can be .PDATA, .TEXT, .IDATA, etc.

Another important field, although it is not highlighted, is the size of optional header. The sizes of optional header is important because when you are creating your own PE parsing utility, you can use this value in a calculation to jump over the optional header and land right at the first section of the binary.

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 (5)
Note: Some output was removed to fit on the slide.

The two more important fields in the optional header are the magic number and the entry point. The magic number here will either indicate if the binary is a 32-bit compilation or a 64-bit compilation. The hex value 10Bh indicates 32-bit compilation, or simply PE32. For a 64-bit compilation, the hex value 20Bh would be shown, or simply PE32+. Again, the optional header is not optional for PE files because the loader depends on information found within this header. Import information it needs from this section is, for starters, the PE type or the optional header type. The slide output shows a PE32 file, meaning an x86 DLL that must be loaded by x86 clients. The entry point is also important since the value found here will redirect you to where the first executable instructions are for the program. You can loosely tie the entry point to the DLL's DllMain() function or a program's main() function. For this example, the entry point is 0 because it was explicitly passed /NOENTRY at the command line when being compiled. This indicates that there is no custom entry point for this DLL. NOENTRY is something that will be discussed in greater detail when we start to build shellcode.

Other fields to mention are the following:
The size of code – this is the size of the TEXT section where the executable code resides
Size of initialized data – any data, like variables or strings, that have been initialized with values will be here
Image base (preferred base address) – where the image would like to be loaded versus where the system loader loaded it
Section and file alignment – how the file is aligned on disk versus in memory with proper page alignment for each section
Size of image – the overall size of the entire image; useful when manually loading a DLL
Size of headers – the sum sizes of all headers that has been rounded up to the nearest file alignment size value

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

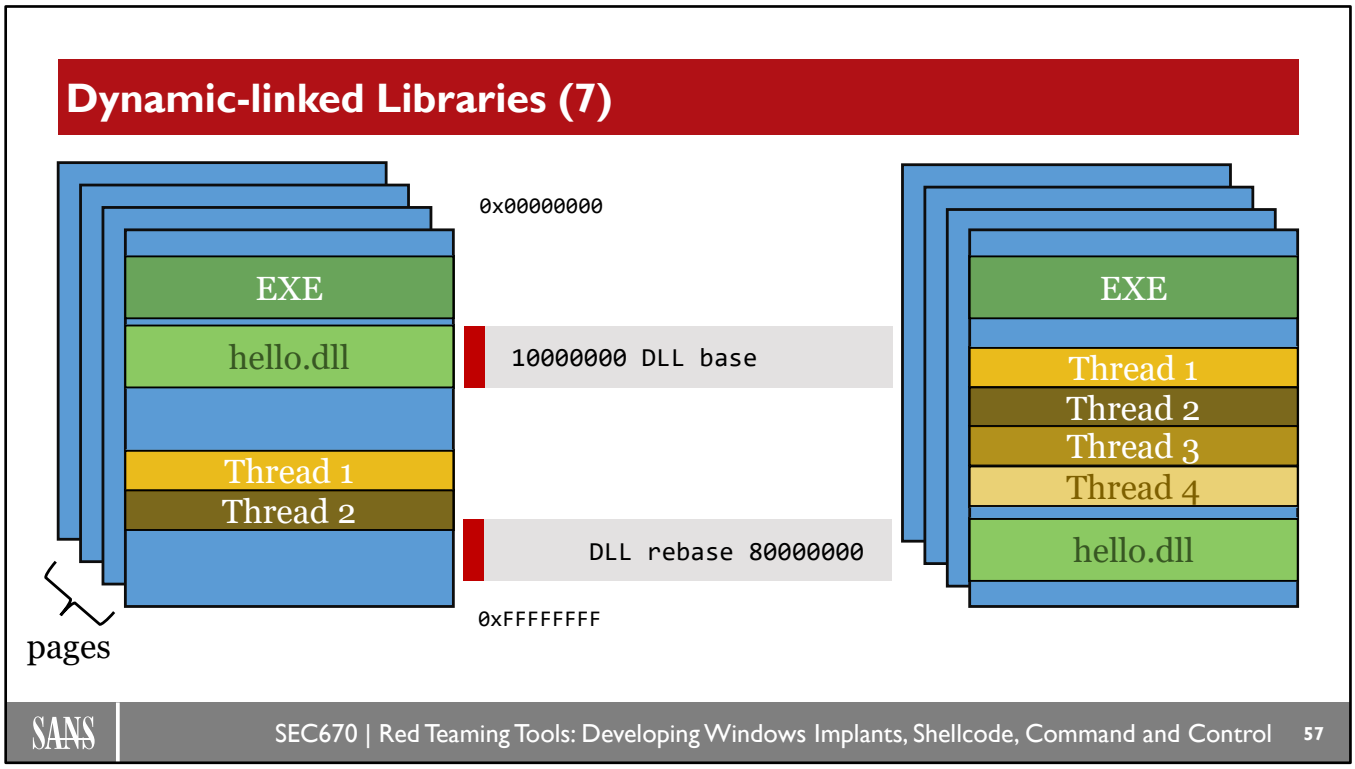
FFC90 = 743AFC90 - 742B0000

Dynamic-linked Libraries (6)

The last portion of the optional header is the number of directories. This is going to be important to understand as the course progresses because tools will be developed to parse through this section and modifications will be made here. Before we begin, let us first understand RVAs. RVAs are relative virtual addresses from the beginning of the file. In other words, they are simply offsets from the beginning of the file. Since DLLs can be loaded at any random address, by design, it is much easier to deal with offsets. Once a DLL is loaded, the virtual address in memory can be calculated with some simple math. Take the base address of the DLL and add the RVA to it. The result is the virtual address. The reverse is also true—given a base address and virtual address, the RVA can be determined.

Now, let use talk about the directories. There are 0x10, or 16 in decimal, directories with most of them being empty or NULL. The export directory, at offset 2040, is used to hold an array of export entries. This table holds information about the functions that the DLL exports. More on that later.

The next item that has data is the base relocation directory. This is in case the DLL cannot be loaded into its preferred base address, shown earlier. When the image is given a new base address, items will have to be relocated and fixed up when the DLL is loaded into memory.



Dynamic-linked Libraries (7)

It was already discussed that EXE and DLL files have a base value that indicates its preferred base address. There are times when the preferred base address for a DLL can be given by the loader when it is loaded, but the more common case is that it will not get its preferred base address. If you look at the graphic on the slide, the stack of pages on the left side of the slide shows the address space along with several items, like threads and the image itself, already occupying some memory regions. It just so happens that nothing is mapped at the DLL's preferred base address. When the loader is mapping it in and it sees this, it will happily map it there.

The stack of pages on the right of the slide shows what happens when there isn't room because something is already mapped to that address, like more thread stacks. The module will have to be move around and given a different base address. When this happens, the loader will fix up everything in the DLL, like pointers to various items. The action of relocating an image is why understanding RVAs is important.

Now, in the interest of security, we want to prevent attackers from knowing where DLLs will be loaded ahead of time. Malware authors must now do a bit more work to find critical modules, like kernel32, because the loader will be choosing a random address to map modules into. The base address for application modules are randomized each time the process is mapped into memory and system modules are randomized with each boot.

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 (8)

The section headers should immediately follow the optional header and the number of sections can be found in the file header. The first one, TEXT, is where the executable code is located. Please pay special attention to the permissions of this section and you might notice how the Write permission flag is missing. The TEXT section is only Execute and Read because the processor must be allowed to read the instructions and execute them. If the section had the Write permission, then an attacker would be free to make changes to program code. The loader will make sure this section is mapped into a page of memory with only PAGE_EXECUTE_READ permissions set.

The virtual size is 10 bytes. The RVA is 0x1000, which can be located easily once the base address is known.

Using the */rawdata /section:.text* will show the data in hex, but you can also look at the disassembly of it too by using */disasm /section:.text.*

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 (9)
The *.rdata* section is a Read only section that holds any data that has been initialized, like setting the variable *best_sans_class* to 670. The other interesting tidbit here is that the directories that were noted in the optional header reside specifically within the .rdata section. This can be observed and verified by looking at the RVAs listed for the Export and Debug directories. The RVAs are clearly within the .rdata section.

Just like with the .text section, the raw data can be viewed to see what is actually being held in the .rdata section. After all, it should have some information because initialized data should be here.

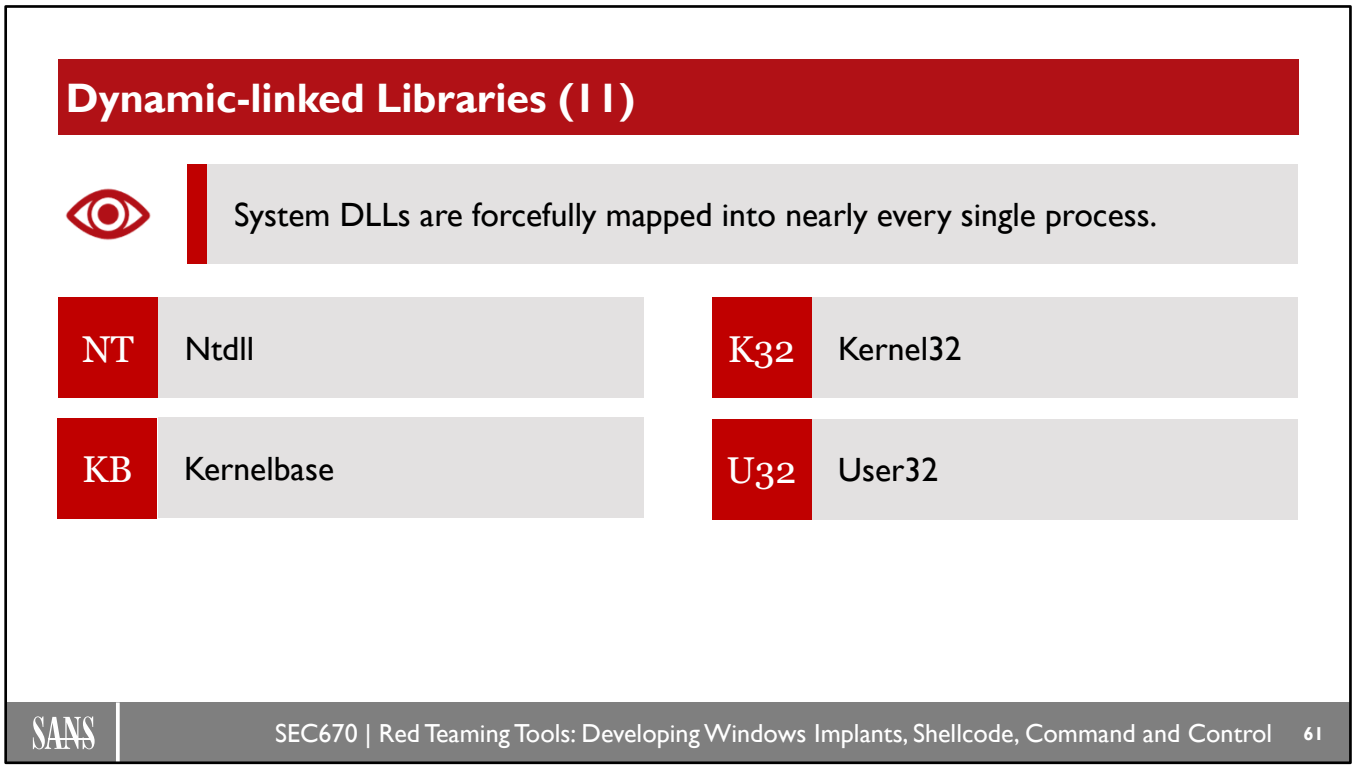
Dynamic-linked Libraries (10)

```
/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 (10)

It was discovered on the previous slide that the RVA for the PrintHello function was 1000. The output from the */exports* switch verified what was found with a manual lookup. Checking the output from the */disasm* switch along with */section:.text*, the code for the function can be seen. Understanding the assembly for the function is simple because the function only does one thing—return the pointer to a string. The __cdecl calling convention specified for this function uses the EAX register to hold the return values. Here, EAX will end up holding the RVA 2000. The data at RVA 2000 was previously seen already and is indeed the address to the string "Welcome to DLL Hell!"



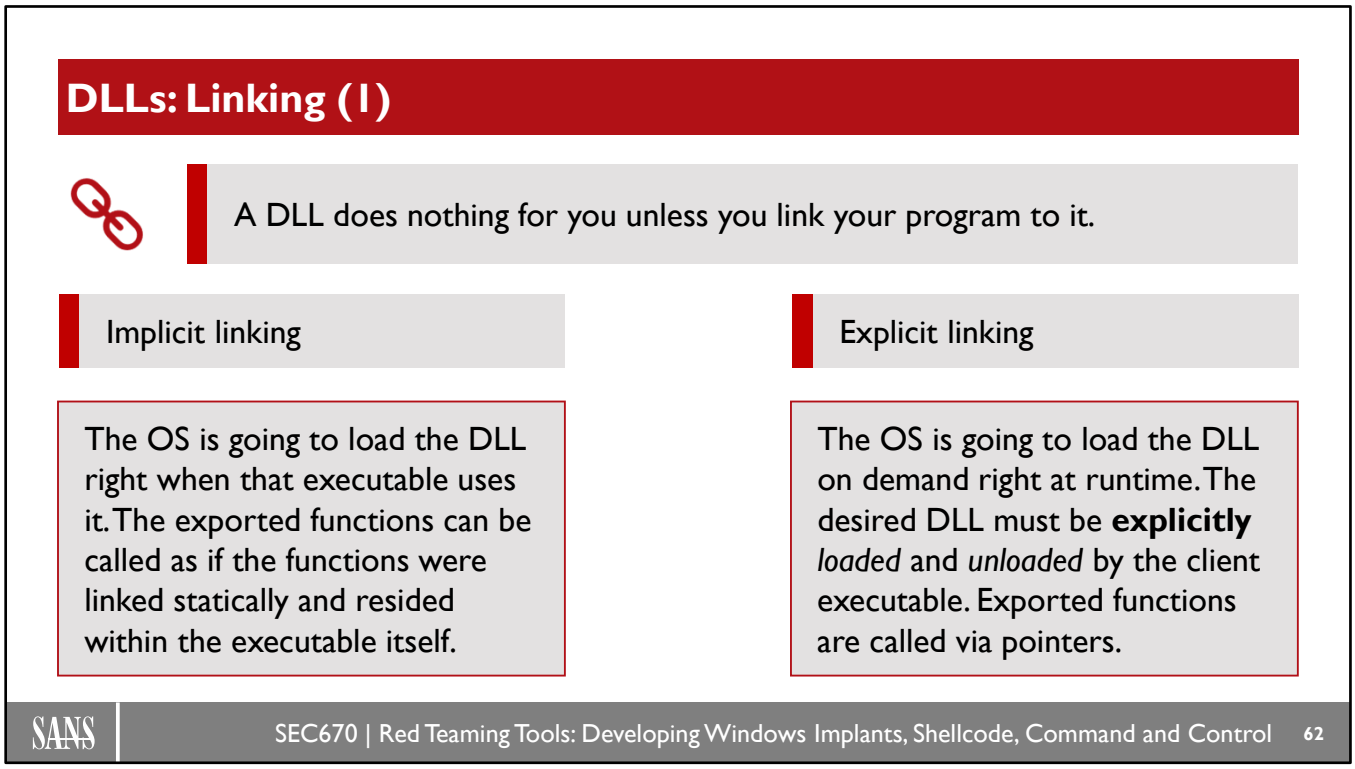
Dynamic-linked Libraries (11)
There are several system DLLs that will be mapped into almost every process: Ntdll.dll, Kernel32.dll, and Kernelbase.dll. Despite them practically always being mapped, Ntdll.dll is the only one required, but the OS will take care of that for you. NTDLL exports many functions that act as a gateway of sorts before making the jump into kernel land.

KERNEL32 also exports many functions and a large number of which are simply re-exported functions from NTDLL. Some functions might not have any code in them at all but are simply jumps or forwarders to a function in NTDLL.

USER32 is a primary component for GUI applications as it holds various functions for creating graphical boxes and windows.

KERNELBASE came around with Windows 7 and basically replaces KERNEL32. KERNELBASE brought with it more and more functionality with each new version. The file size is a quick way to determine if more functionality has been added to a DLL. For example, with its original debut in Windows 7, it was around 288.2KB in size. For Windows 10 it was around 1.5MB

There is at least one project on GitHub that experimented with unloading all system DLLs, even Ntdll.dll. The idea was to bypass 32-bit User-mode hooks by unloading every DLL including the WoW64 DLLs. We will be talking about hooks later in the course.



DLLs: Linking (1)
DLLs must be linked to in some way, shape, or form if you are going to utilize them in your program. The two ways they can be linked to is implicitly or explicitly. The example DLL used for breaking down the export directory structure had a client application that used explicit linking. It explicitly loaded the hello.dll into the program via *LoadLibraryEx* and resolved the address of the PrintHello function using the *GetProcAddress* function. When the program was finished, it freed the library by calling *FreeLibrary*. Any time you see those functions being called; it is a sure indicator that a library is being explicitly linked/loaded.

Implicit linking is nearly the opposite as you rely on the OS to do the heavy lifting for you. Not much to worry about with implicit linking other than maybe ensuring the program has the proper code to bring in the needed LIB file. A great benefit of implicit linking is there is no need to resolve function addresses programmatically. The functions can be called as if they were defined directly in your program, and this method is the de facto standard in most code.

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: Linking (2)

As mentioned previously, implicit linking is by far the easiest method to implement and can be done a few ways. Thanks to recent enhancements to Visual Studio, you can modify the settings for the project and identify the additional dependencies or references. You can access this configuration under Configuration Properties | Linker | Input | Additional Dependencies. While browsing this area, you might also see dependencies already listed, like kernel32.lib. I do not recommend messing around with what is already listed unless you are comfortable dealing with the ramifications.

The other method for implicit linking is to use C/C++ code to add the additional dependency. This is implemented using the #pragma comment directive so that the executable or the object file will contain a comment record. You must know where the lib file resides on the file system, of course, but there are other configurations you can make that would make the search easier. An example could be *#pragma comment(lib, "../relative/path/to/ws2_32.lib")*.

With implicit linking, if a DLL is not found, the process will be terminated with a code execution error.

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
```


DLLs: Implicit Linking

Implicit linking gives your program the freedom to simply call a function as needed without any extra steps. As long as you know the function name and what parameters it has, you can call it as if it were any other function defined locally in your source code. If you would compile the above program using cl /c implicit_callhello.cpp, you could then link it against the hello.lib library like so: ***link implicit_callhello.obj*** hello.lib. Now your program depends on the hello.dll module and it will not run if that module were to be deleted or somehow not found by the loader. Running ***dumpbin /dependents implicit_callhello.exe*** will show what modules the program depends on to run. More details about the hello.lib library can be shown by running ***dumpin /all hello.lib***. The library only has five public symbols so finding the symbol for PrintHello won't be too hard. Scroll through its output until the PrintHello function is found. The output shown indicates what type it is (code, in this case), what the symbol name is (_PrintHello), and what the function name is (PrintHello). As an alternative, run the same dumpbin command against the kernel32.lib located here: C:\Program Files (x86)\Windows Kits\10\Lib\10.0.18362.0\um\x86\kernel32.lib. I recommend redirecting the output to a TXT file as its output is quite large.

Below is the full output from the dumpbin commands.

dumpbin /dependents implicit_callhello.exe
File Type: EXECUTABLE IMAGE

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 (1)

Explicit linking is quite different in many ways, one of them being that you do not add any **#include** directives as that will only generate "unresolved external" errors. Another difference is how you call the functions exported from your target DLL. To get started, you must call the ***LoadLibrary*** or the ***LoadLibraryEx*** function passing in the name of the DLL to load. Either function will do what it sounds like—it will load a given library into your process returning a handle to it. The great thing here is that if the DLL is not found for any reason, the process will not be terminated. Instead, NULL is returned, and your code can account for this possibility to take corrective action. The calling of ***LoadLibrary*** is you being explicit with the DLL resource you are "linking against" and loading as a dependency. This still does nothing for your program because you have no idea where a function is located inside that DLL, and to figure that out, you must use ***GetProcAddress*** or one of its variants. Upon success, the returned value is the address of the function you need to use. Let's bring back the example used near the start of this section on the next slide.

Side note: The combination of ***LoadLibrary*** and ***GetProcAddress*** are common APIs that are looked for by AV/EDR products and we will discuss "workarounds" later in course like creating your own versions of them.

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 (2)

There are no include statements that bring in the function PrintHello. Any attempt to call PrintHello directly would cause errors. The hello.dll is linked to explicitly by calling the ***LoadLibraryExW*** function and it is also freed later using ***FreeLibrary***. The PrintHello function can be called once the proper declaration has been made for it.

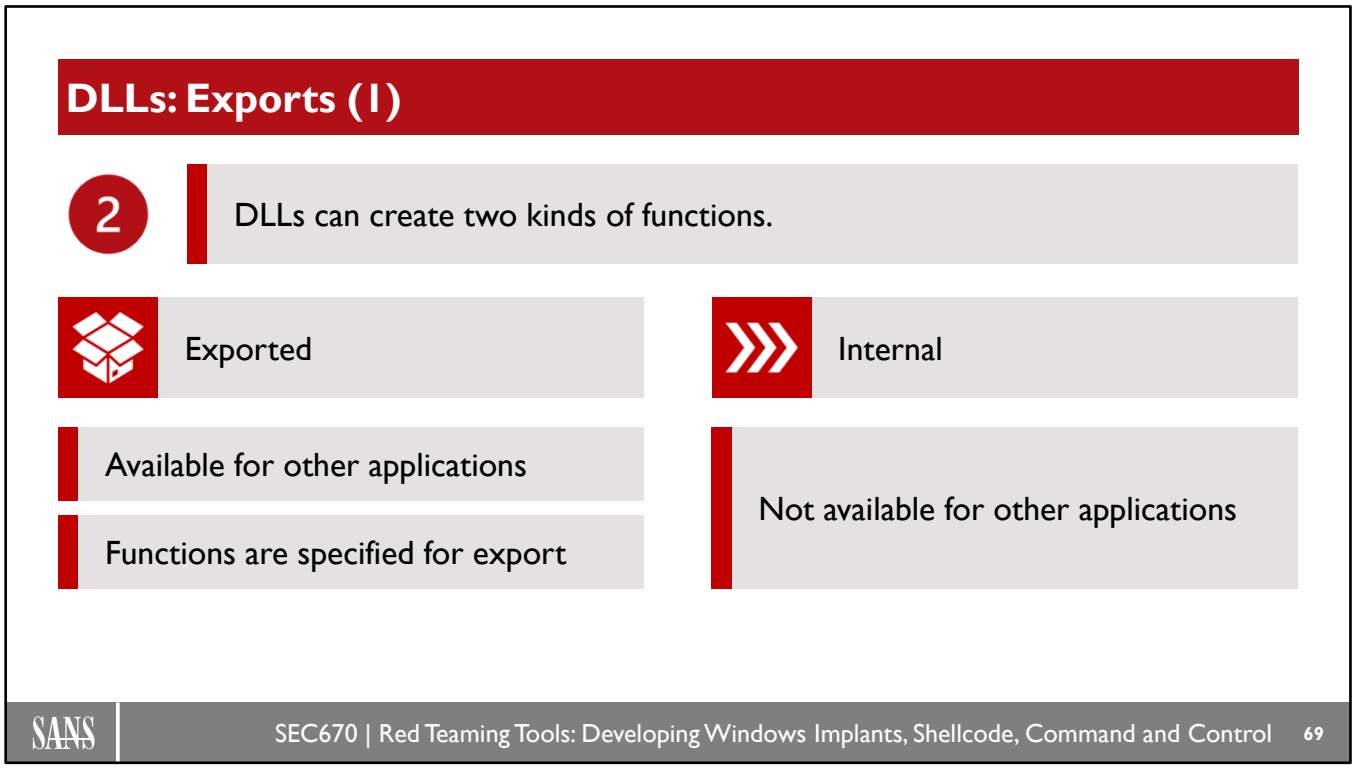
DLLs: Explicit Linking (3)

```
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: Explicit Linking (3)

Using the /dependents switch with dumpbin, any module that it depends on will be listed. Notice that hello.dll is not listed in the output. This is expected since it was not implicitly linked. The callhello.exe application only depends on the KERNEL32 DLL. The same goes for the imports. It makes sense that there are no imported function from hello.dll because we are not formally importing any functions that hello.dll exports. In fact, we did not import any at all, but we still call them. How can that even be possible? How can we call a function that has not been imported? The only functions that are being imported are primarily from KERNEL32.dll: ***FreeLibrary, GetProcAddress***, and ***LoadLibraryExW***. The primary function that assists us with explicitly linking against DLLs is ***LoadLibraryExW***. We can now use that function to bring in the hello.dll into our process address space. This is needed because we cannot call any of its functions until it gets loaded into memory. Once it is in memory, we can find one of its exported functions and then call that function. The API that let us obtain a function's address is ***GetProcAddress***. Most of this was already discussed a few slides ago. To summarize, the output here is just highlighting that functions you explicitly use from a DLL will not show up in your import table. Later in the course, we will go even further by eliminating the import entries for ***GetProcAddress*** and ***LoadLibraryExW*** by implementing those APIs manually.



DLLs: Exports (1)
DLLs that do not export functions are rather useless, most of the time, but how does that happen? Also, how does the compiler distinguish an exported function from a private one that's only for internal use by that DLL? According to Microsoft, there are four methods to export a function, or a definition. They have a list ordered by recommendation of use. The next several slides will go into the following recommendations.

1. __declspec(dllexport)
2. EXPORTS statement created in a .DEF file
3. Linker option /EXPORT
4. A comment in the source code: #pragma comment(linker, "/export: definition ")

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 (2)
The *__declspec* keyword is an extended attribute that is used to indicate that a specific instance and type should be stored using a Microsoft-specific storage-class attribute. There are many Microsoft-specific attributes available for use like *dllimport*, but the one we care about in this instance is *dllexport*. Be aware, though, that you must use *__declspec(dllexport)* for each function you want to export. If your DLL is going to export hundreds of functions, then perhaps you would prefer to use a module definition file.

A module definition file has a .def extension and is simply a text file containing special syntax to describe the attributes of the DLL. Using a definition file is much simpler and faster when your DLL has hundreds, if not thousands, of functions to export.

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: Exports (3)
Module-Definition files serve the purpose of giving the linker various information, such as exports, attributes, heap size, or other program information that is to be linked. Within a DEF, there are various statements, and each statement has rules that must be followed. Some statement examples are:

EXPORTS: The statement that marks a section of function names to be exported to include the ordinal of the function.

LIBRARY: This indicates to the linker program that a DLL is to be created while, at the same time, create an import library. This is going to be the name of the DLL. When building the DLL, be sure to use the /DLL linker option either in the project settings or at the command line.

VERSION: This is a simple way to tell the linker program to put a number in the header of the DLL. Default is 0.0 if not specified in the .def file.

*** NONAME = export ordinal only
*** PRIVATE = prevents that name from being an entry in the imports table

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

DLLs: Injection

When you hear the phrase DLL injection, you might think it is a bad thing. Well, it is not. There are several legitimate reasons for injecting DLLs into processes and AV solutions would serve as just one example. The badness, if you will, comes from the DLL's code. If the DLL's code is malicious then obviously you would not want injection to take place. If you are installing AV software, then you would want the injection to happen and you would want it to hook certain APIs because it must do its job protecting your system and the processes on it.

DLL injection is a forceful technique so there's nothing nice about it. You are forcing a process to execute the code of your DLL in its context. Meaning, the DLL's code will have access to everything in the virtual address space of the process, which is of grave concern when the code is malicious. Otherwise, inject away.

There are several popular DLL injection methods out there today and we will explore a few of them later in the course.

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`

Shared Objects
For those who are coming over from the *Nix side of C/C++ development, Linux shared objects can be viewed the same as Windows DLLs. Both provide functionality that can be imported and utilized in your programs. Both are a form of executable, though the file format is specific to the platform: Windows PE or Linux ELF.

Both provide symbols for programs to utilize and just like on Windows, they can be loaded and unloaded dynamically (explicit linking) using functions dlopen (LoadLibrary), dlclose (FreeLibrary). Resolution can be done just like Windows as well using dlsym (GetProcAddress). As mentioned previously, the resolution of a symbol's address is required since the library was loaded on demand.

One difference is that Windows DLLs must specifically export their routines using a special statement; __declspec(dllexport). Also, as mentioned in the previous slides, you could create a DEF document that lists the functions to be exported. Shared objects have no such statement and as such, every function can be used by a process.

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.
```


Windows Data Types / C Data Types

Windows data types will be discussed in greater detail in the next section, but they are mentioned here because they are obviously not readily available to use on Linux. If you desired, you could create them in a header file and include that in your programs, but I have never seen that done by any Linux developers I know. The standard C data types can be used on Windows, but it is not recommended, which will also be explained in detail during the next section. In the end, the Windows data types are nothing more than defined types, or type defs. You will see this in code as the typedef keyword, one you will be using a lot.

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

Header Files
Header files are, for the most part, the same. You can still include stdio.h, iostream, etc. just like you could on Linux. The main difference here is Windows will have their own header files that you must include in your projects because they give access to the Win32 APIs and the Windows data types. The windows.h header file will arguably be included the most in a large majority of your projects.

As you become more familiar with developing Windows programs, you will know what header files will need to be included to give your program the functionality you desire.

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.

Windows APIs
The Windows APIs will be very new to those coming over from Linux. The naming conventions are very detailed and sometimes rather lengthy. I believe the longest public API is still *RtlWriteDecodedUcsDataIntoSmartLBlobUcsWritingContext*. Aside from lengthy function names, the Windows API also introduced a few Windows-specific calling conventions like stdcall that will be discussed in their own section later on.

Standard C Functions

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

Standard C Functions
The basic C functions like memcpy, printf, strcpy, etc. can still be used, but Windows has created APIs that wrap most of them like RtlCopyMemory, which behaves just like memcpy.

If you wanted, you could absolutely use the standard C functions. However, it is best to stick with the Windows APIs for nearly every type of functionality in your program. Use Windows-specific macros whenever possible too, because if Windows decides to change anything then your program should automatically adjust accordingly, if using the macros.

Lab 1.3: HelloDLL

Learn how to build a simple DLL.

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

Lab 1.3: HelloDLL
This lab has you building out your first DLL. We will be using Visual Studio for this and will be creating .def files and using exports. An EXE program can be used to use your DLL. This will also be a perfect time to explore using rundll32.exe to execute your DLL.

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

What's the Point?

What's the point?

What's the Point?
The point of the lab was to become familiar with building a DLL and how to export functions. DLLs can be very beneficial and are great for injecting into processes.

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

Module Summary

In this module, we covered a few development differences between Linux and Windows. We also covered some of the ways that development is similar. C is still C but the power and main functionality for Windows comes from using the Windows APIs.

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 Questions
Q: 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 Answers
Q: 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 Questions
Q: 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

Unit Review Answers
Q: 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

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will introduce you to the Windows data types. Most of them will probably seem strange at first, but once we look at how they are defined any confusion should be gone.

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
```


Windows Data Types

You might be reading this slide and thinking to yourself, "Aren't data types on Windows just the same as they are for Linux?" Not really. Back in the day, when Windows was a true 16-bit OS and people were developing 16-bit applications, they were using 16-bit ints. When the upgrade to a 32-bit OS came around, you would have to manually change every single instance where you used an int because you needed to swap it out with a new data type that was 32-bit, which could be quite tedious depending on how large your project was.

The solution? Windows data types. If you were using the Windows data types when these changes were rolling out, then you did not have to worry about a thing. Imagine how much easier it would be to change one spot in your program versus finding every instance where you used a traditional data type. Microsoft did not want the compiler to be the one determining what the size of an int should be when an application was compiled. The technique allows types like DWORDs (double WORD) to always be an unsigned 32-bit value regardless of the target system: 16-bit, 32-bit, or 64-bit.

The bottom line here is just stick to the Windows data types for best practices and better portability.

BOOL/BOOLEAN

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

1010
1010

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;
```

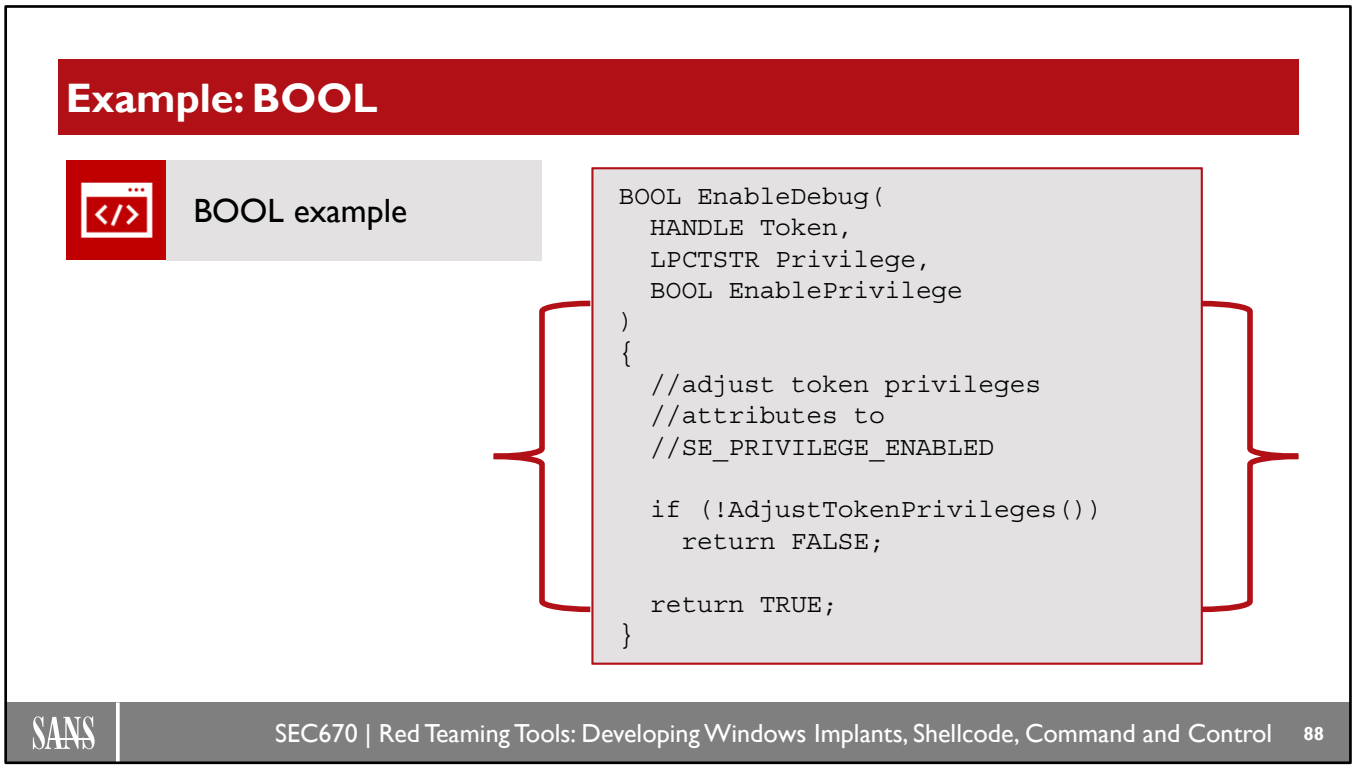

BOOL

BOOL is short for Boolean where the value can only be set (1) for TRUE or clear (0) for FALSE. On the back end, BOOLs are 32-bit values because the root type is an int. If you are really only needing to test a single byte then there is no need for the other three bytes. The other bytes are just wasted space that could be used for something else. Now take the type BOOLEAN.

BOOLEAN is rather interesting because the surface, its root type is a BYTE, but notice that it is in all caps. This means this is another Windows data type. The root type for a BYTE is an unsigned char. When you are looking at BOOLEAN typed variables in memory, they will only occupy one byte, so there is no wasted space. Some code examples use these types interchangeably and that might be fine at first, but it is not recommended.

It is also worth mentioning the definitions for *TRUE* and *FALSE* are 1 and 0, respectively. Some of you might be familiar with seeing or using *while*(1) to implement a while loop but here we could implement that like this:

```
while (TRUE)
{
  // stuff
}
```



Example: BOOL

This example here is showing how a function can be called to indicate whether debug privileges were successfully given to the process. If the function cannot adjust the privileges for the token, then it will return FALSE. When it can successfully do so, it will return TRUE, and your main program logic can check this condition and proceed on or take another action, like exiting.

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);
```


Example: WORD, DWORD, LPDWORD, QWORD
The first example on the slide is assigning the value returned from calling the ***GetCurrentProcessID*** API to the DWORD variable dwProcId.

Next, a pointer to the dwProcId variable is created and assigned to it is the pdwProcId variable.

Finally, a long pointer to the dwProcId variable is created and assigned to it is the lpProcId variable.

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!");
}
```


Example: VOID, PVOID, LPVOID, LPCVOID
The several examples here are not complicated but still, let's break them down one at a time.

The first is simply stating the function will not return a value so it is void. It also does not take any arguments and void could be used there as well.

The second one is a void pointer to a variable that could be pointing to the *AddressOfEntryPoint.*

The third one is declaring a const void pointer to the *SomeConstant* variable. This could also be something that could also be a constant value like a pointer to a base address as returned from ***MapViewOfFile*** or ***VirtualAlloc***. An LPCVOID variable could be passed into a function like the ***UnmapViewOfFile*** function. You could also use this type to cast the address of a wide character string that is to be passed to the ***WriteFile*** function as its second parameter takes in an LPCVOID buffer pointer.

The last one is just showing another VOID function. The big difference with this one is the argument, which is a pointer to a destination buffer for the ***strcpy*** function to copy over the string into it.

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?

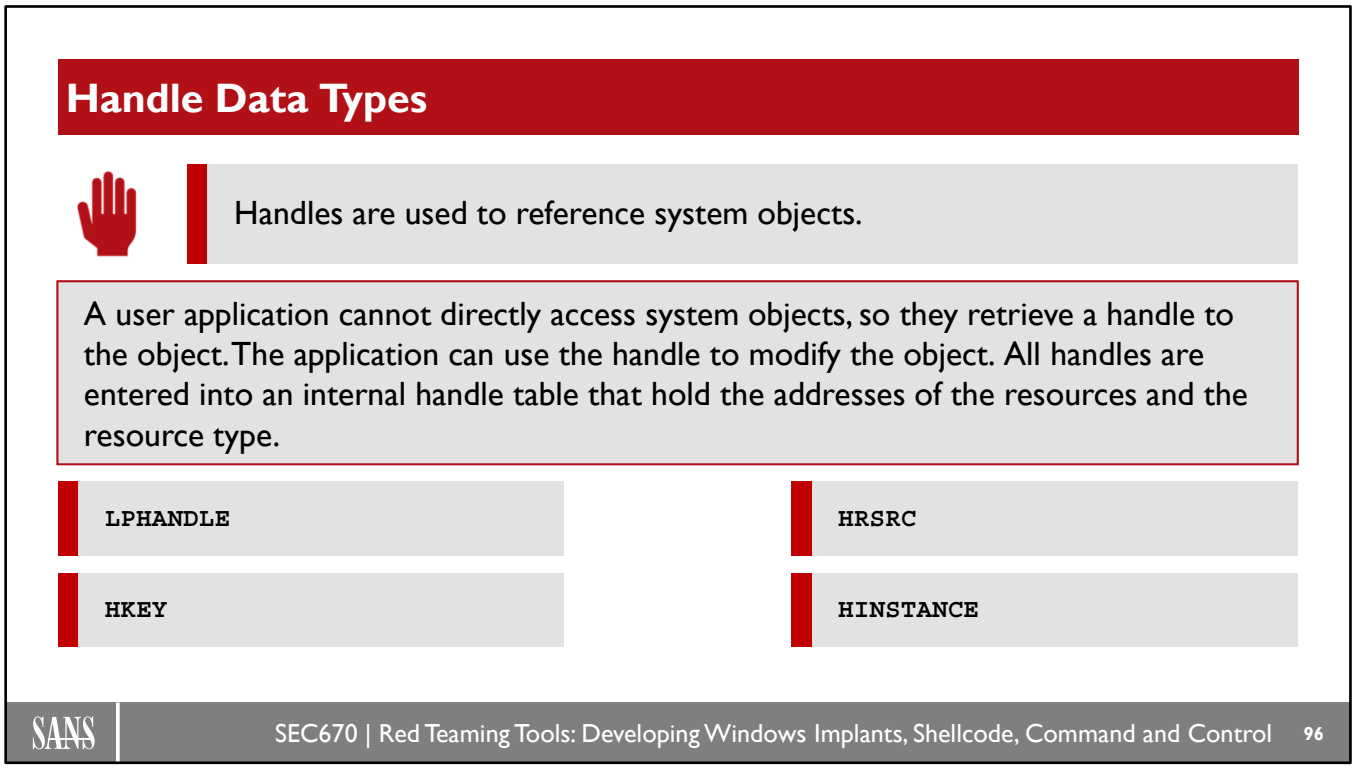
Example of C++ Data Types

```
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 */
```


Example of C++ Data Types

The examples above are trivial ones, but ones that you might be able to use in your code base. Of the ones above, the more important ones to note are the smart pointers. Note that when making them, there is no usage of the *new* operator. As noted previously, smart pointers can be treated similarly to raw pointers in the sense you can dereference them. That example is on the slide. The smart pointer *pBestClass* is dereferenced and given a new value. One can also access the get() method to make a raw pointer. This will return the address the smart pointer is pointing to. This could now lead to a dangling pointer if you are not careful. You might be forced to use *get()* because some functions will require a raw pointer. There are so many more things that smart pointers can do, and you will see a few of them being used in this course.

With modern C++, specifically C++ 20 and beyond, you should ***NEVER*** be using the new operator. Ever. Period!



Handle Data Types
The types listed here are all handles of some kind. Again, it can be seen how a lot of these data types really do build off each other and knowing this can allow you to use some interchangeably since you know they are the same. You can also interpret clearly what is happening. What is meant by that is HKEY would be a handle to a registry key, HMODULE would be a handle to a module, and HINSTANCE would be a handle to an instance. Both the HINSTANCE and HMODULE are just the base address of a module.

For example, if a Win32 API requires a parameter of type *HINSTANCE* but you only have a variable of type *HMODULE*, then you can simply pass that *HMODULE* in because they are really the same type in the end. However, this was not always the case. Let's go back again to 16-bit Windows where HINSTANCE and HMODULE were not the same. The module would be the file that was loaded into memory. If you would run two copies of Notepad on 16-bit Windows, you would have two instances of notepad.exe with each instance having its own set of variables.

Also, you may have noticed by now that the naming conventions used are very indicative as to what their type is. Here, the types are basically all HANDLEs, and the names start with an "H" to indicate this is a handle to an object.

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;
```


Handle Data Types Defined
The Winnt header file is filled with definitions, the ones listed on the slide are just a small subset of them. If you take a look at the first typedef, you can see that a HANDLE is really just a PVOID, which is simply a void*. The remaining typedefs are using the HANDLE definition as the base. Typically, whenever you see a handle type being used, you could assume that it's simply a void*, or PVOID.

Although it is not recommended, something like this could be done and might not cause any issues: PVOID hKey = (HKEY)HKEY_LOCAL_MACHINE;

The very last typedef is an interesting one. Even though it was mentioned on the previous slide, it is worth mentioning again due to its importance and sometimes being a driving source of confusion for beginner developers. Today, HINSTANCE and HMODULE types are completely interchangeable, although it is best practice to not do so. You should absolutely keep your types specific to what they are needed to be used for.

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
All three of these types are used for functions that will follow the stdcall calling convention, which is something that we will be discussing in the next section. When declaring a function in a header file you can use one of these to force it to use the *stdcall* convention.

This is a very simple example, but it does get the point across.

```
BOOL
WINAPI
IsDigit(_In_ DWORD dwDigit);
```

WINAPI, APIENTRY, CALLBACK Defined

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

```
#define WINAPI __stdcall

#define APIENTRY WINAPI

#define CALLBACK __stdcall
```


WINAPI, APIENTRY, CALLBACK Defined

The typedef examples here on the slide are riddled throughout a large number of header files. They are defined in the WinDef header file, but if you take a look inside the winnt or winreg header files, you will see how often they are used. As an example, the winreg header file shows a large number of the Reg* functions declared with APIENTRY and WINADVAPI (DECLSPEC_IMPORT). The functions with WINADVAPI indicates the function is an imported function from some other DLL. The APIENTRY indicates the function will fuse the __stdcall calling convention. The CALLBACK definition is also the __stdcall calling convention and is used typically when typecasting or creating a function pointer with a specific signature.

The list on the slide is not an exhaustive list as there are many other defined types like NTSYSAPI (DECLSPEC_IMPORT), NTAPI (__stdcall), along with many others.

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 );
```


Example: WINAPI, APIENTRY, CALLBACK
These are many types that, for the most part, mean the exact same thing. The EnumProcesses function has WINAPI, which indicates the function uses the __stdcall calling convention. The function itself writes the PIDs out to an array via the lpidProcess parameter. The RegCreateKeyA function has APIENTRY and WINADVAPI. From the previous slide, it is known that the function is using the __stdcall calling convention and is also an imported function using the __declspec(import) attribute. The last type, CALLBACK, is more of the same—it simply indicates the function will be using the __stdcall calling convention.

Again, Windows has many ways to indicate that a function will use the __stdcall calling convention or that one is an imported function. Try to not be confused when you seem them in header files. Despite there being many types that mean the same thing, it does look cleaner seeing the types versus seeing __stdcall and __declspec() all over the place.

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

Module Summary
In this module, we discussed many of the data types you will come across during your Windows development and during the rest of this course. We saw how many of those data types mean the same thing and thus, are interchangeable. We also saw how using types can make code look cleaner and more readable. I strongly recommend to use the Windows data types in your tools.

The more you practice coding various capabilities, the more familiar and comfortable you will become with the Windows data types.

Unit Review Questions

What does the type WINAPI expand as?

A __stdcall

B __thiscall

C __cdecl

Unit Review Questions
Q: 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 Answers
Q: 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 Questions
Q: 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 Answers
Q: 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 Questions
Q: 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 Answers
Q: 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 Questions
Q: 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.

Unit Review Answers
Q: 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.

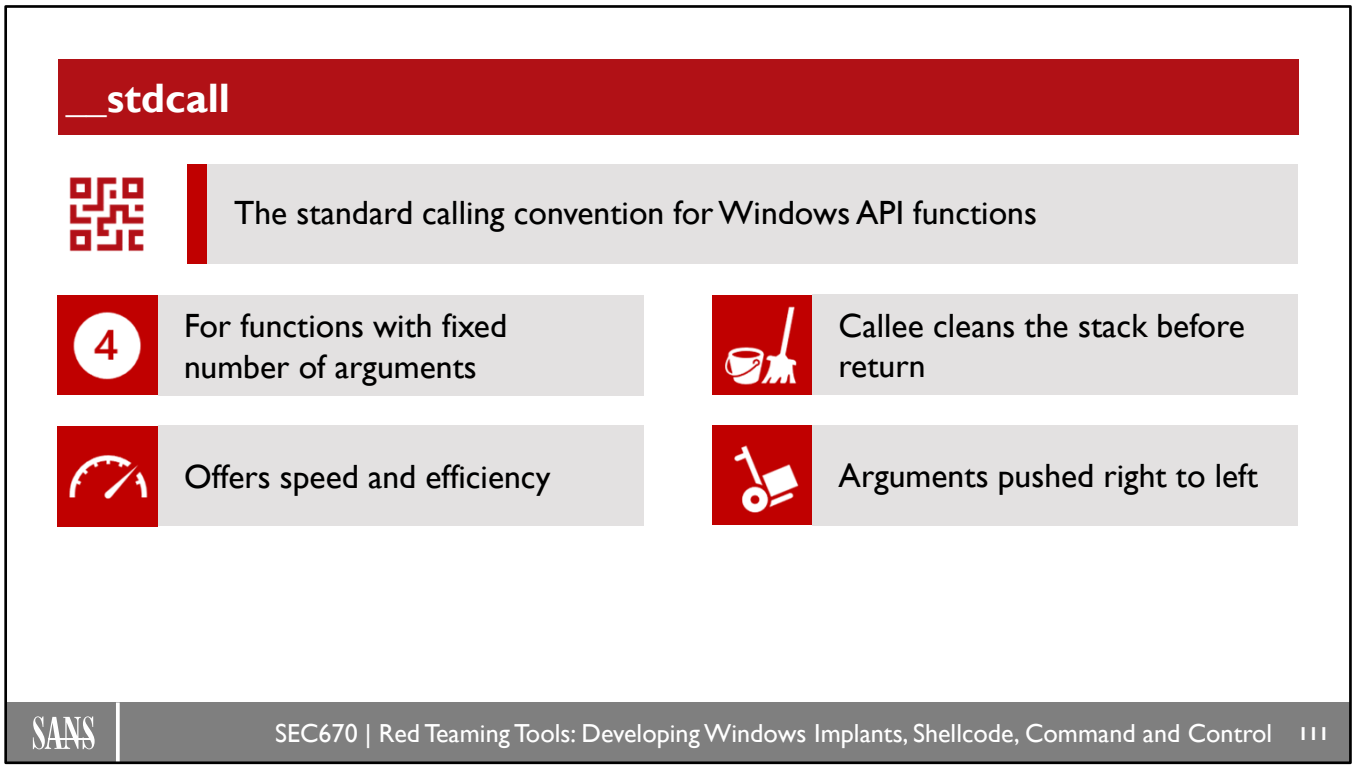
Course Roadmap

- **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

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss several calling conventions and which ones are specific to Windows.



__stdcall

The stdcall calling convention is used heavily for the Windows APIs since almost all of them have a fixed number of arguments. This is great because the function calling a stdcall function does not have to do stack cleanup when it returns. The called function is the one that will handle stack cleanup. As we discussed already, there are a number of data types defined around __stdcall.

When it comes to arguments, for x86, they are pushed onto the stack in a right to left manner. This ensures that the first argument is right before the function call. For x64, registers come into play. The first argument used is moved into RCX, the second into RDX, the third into R8, and the fourth into R9. Any additional arguments are then placed on the stack. Let's check out a simple example using CreateFile.

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: __stdcall (x86)
Look at a simple example using **CreateFileW** compiled for x86. The very first argument for CreateFileW is directly above it. Each additional argument falls into place one after another on the stack. Another way to look at this from a source code point of view is to annotate the stack offsets directly above the function's parameters. Since the arguments are referenced as offsets from EBP, the following can be made as a visualization. Also, note how the code makes the call to the Unicode version of the function. This is because Unicode is supported by default in Visual Studio projects, and the CreateFileW used in the source code is just a macro that is swapped out for the Unicode version.

EBP +8 +Ch +10h +14h +18h +1Ch +20h
CreateFileW(L"hello.txt", GENERIC_WRITE, NULL, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

The following would be the disassembly, same as what is on the slide.

```
push 0
push 80h                    ;FILE_ATTRIBUTE_NORMAL
push 1                      ;CREATE_NEW
push 0
push 0
push 40000000h             ;GENERIC_WRITE
push FileName
call CreateFileW
```

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>]
```

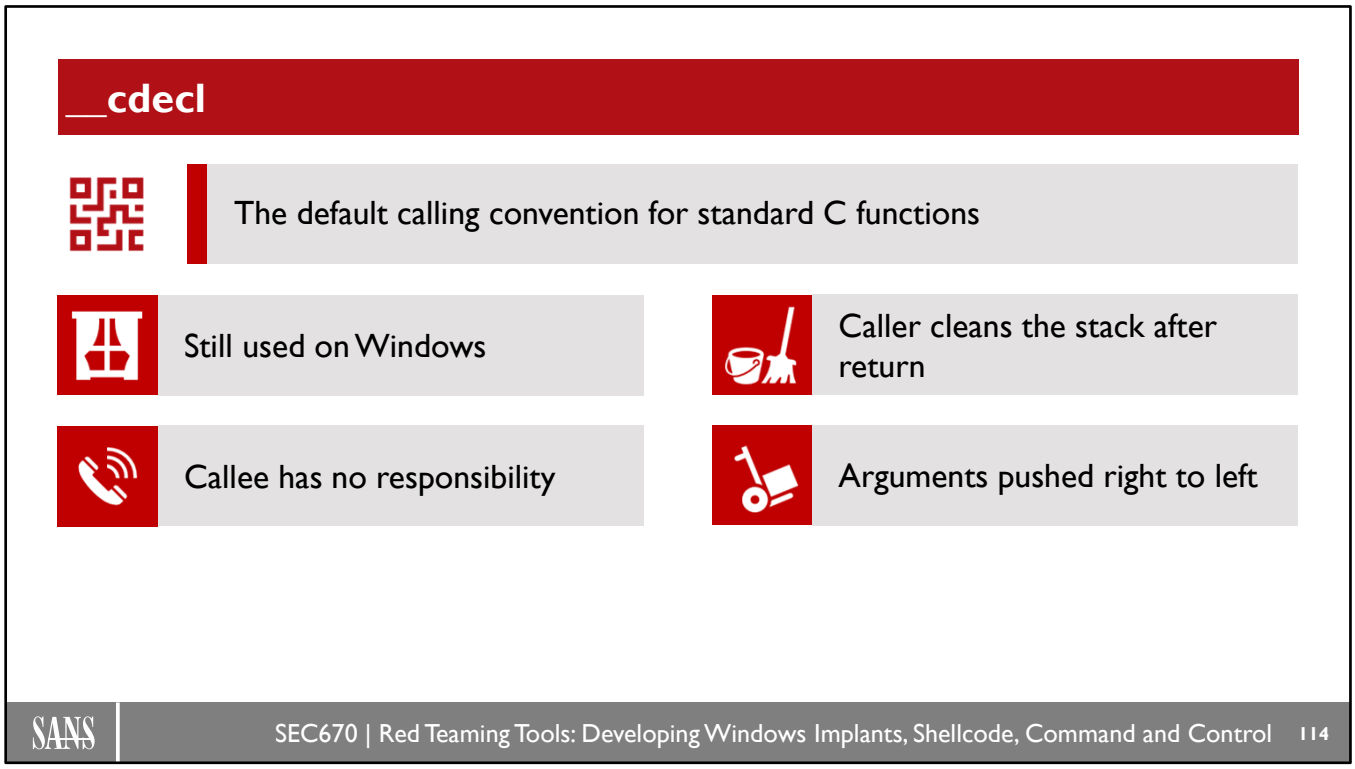

Example: (x64)

When we look at this example compiled for x64, the arguments look a bit different and out of order than what you might think. There is no order in terms of how it looks in assembly—all that matters is that the registers are loaded with the proper values. Since there are seven arguments total, the first four will be in registers, and the remainder will be placed on the stack. Here is one way this can be visualized. Placing the registers above the arguments can help understand where things will fall into place when the call is ready to be made.

RCX RDX R8 R9 stack stack stack

CreateFileW(L"hello.txt", GENERIC_WRITE, NULL, 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

This convention is the default and the main convention used for Linux C/C++ programs, but not only just Linux programs. It is also here in Windows too and is the default for CRT functions like ***printf().*** One great benefit of ***__cdecl*** is that functions can accept a varied number of arguments. You may have heard functions like these also be referred to as VARARG or variadic functions because they are so flexible with how many arguments they can accept. This process of handling a variadic number of arguments is enabled because the calling function will be the one to clean up the arguments off the stack. A perfect example of a variadic function is the ***printf()*** function. It can be passed one argument like a string to print, or any number of arguments. If you have, say, a formatted string that will ingest any number of formatted values into it like the string "Hello, %s, the class you are taking is %d\n", the function does not know ahead of its call exactly how many arguments there are, and it does not need to know that since it is not responsible for cleaning up the stack. Also, just like ***__stdcall***, arguments are "pushed" on the stack in a right to left manner so that the first argument will be right "above" the called function.

A small downside to having ***__cdecl*** functions in your code is that it can increase the size of your binary a decent amount. The size increase is because each ***__cdecl*** function must have the code to perform stack cleanup. If you can get rid of the CRT dependency in your toolset, then do it.

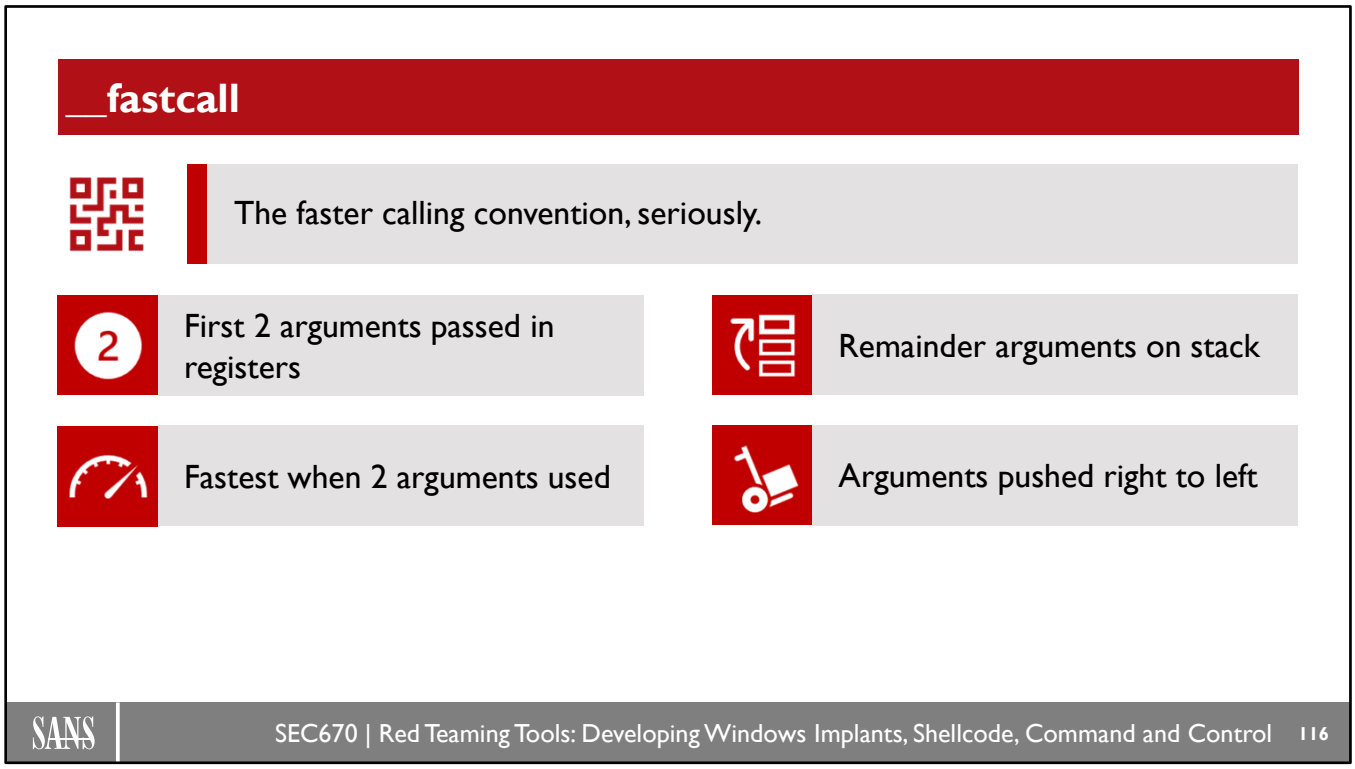
Example: __cdecl

```
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
```


Example: __cdecl

The example here shows the printf function, which uses the __cdecl format on both Windows and Linux. The format string takes in four formatters, or values to be put into the format specifiers in the string. These are treated as arguments to the printf function as can be seen via the disassembly. Just like __stdcall, the arguments are pushed right to left including the address of the string itself. Everything is in hexadecimal formatting, but comments have been made to make it easier to follow what arguments are where on the stack.

```
push  660
push  670
push  92
push  670
push  &theFormatString
call   printf
```



__fastcall

This convention is "fast" because of the heavy usage of the CPU's registers. The ***__fastcall*** convention is only available for x86 CPUs, so do not look for this in x86_64 code/binaries. When it comes to passing in the arguments, if the size of the first two arguments are the size of a DWORD (unsigned long) or smaller, then ECX and EDX will hold the first and second arguments, respectively. Remember, a DWORD is 4 bytes. Any additional arguments will be "pushed" onto the stack in a right to left manner just like how it is done for ***__stdcall*** and ***__cdecl***. Now when it comes time to debug your program and dive into the assembly of it all, it is nice to know what the calling convention is. For ***__fastcall***, you will see an "@" symbol as a prefix to the name of the function. This prefix is in addition to that "@" symbol that will be present at the end of the function name. After the trailing "@" symbol will be a decimal number indicating the total number of bytes the arguments occupy. This way, the called function can properly clean up before it returns to the caller. See the next slide for a detailed look.

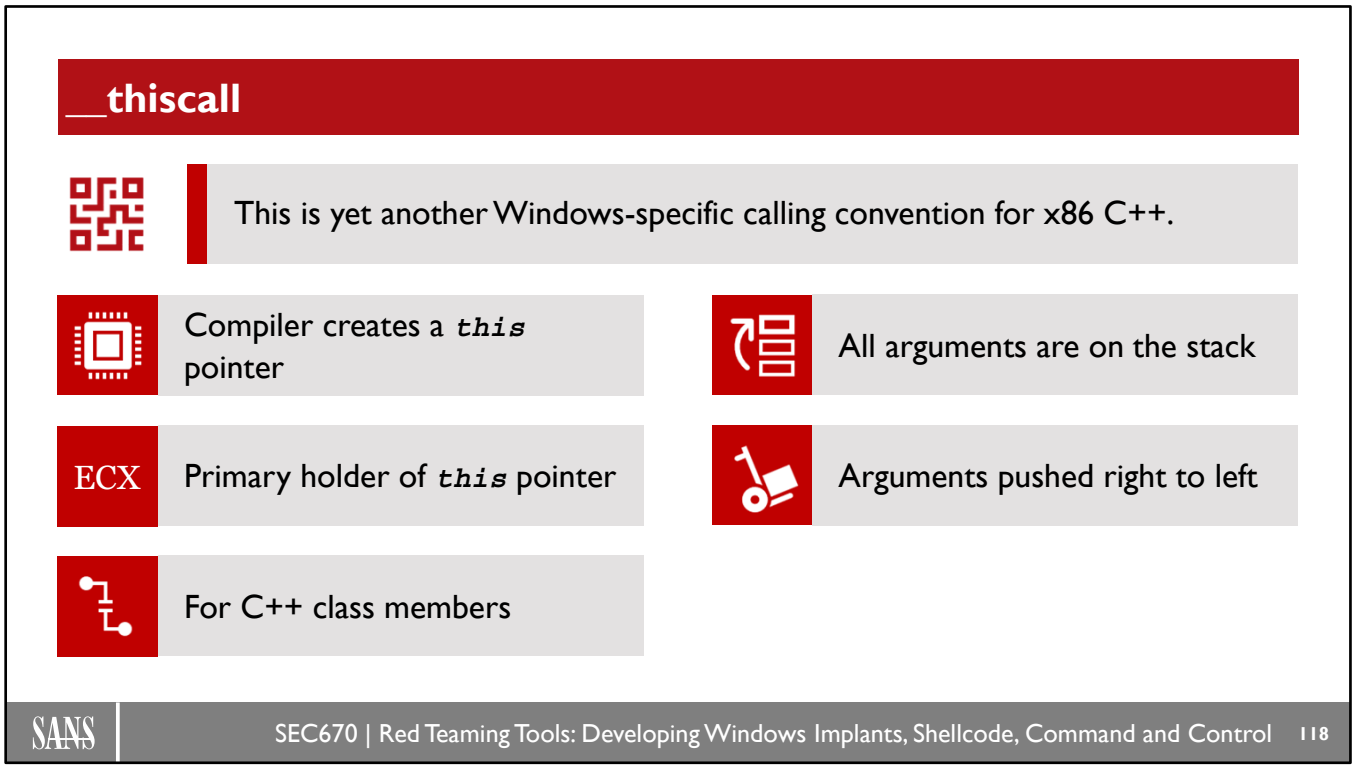
In the end, if you are creating an x86 capability, consider specifying ***__fastcall*** for your custom functions that only accept two arguments. Otherwise, it might be better to use ***__stdcall*** for all things Windows.

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
```


Example: __fastcall

There are several items on the slide that indicate that __fastcall calling convention is being used. Aside from the obvious being the function declaration, the first indicator is the use of the ECX and EDX registers. As mentioned on the previous slide, the ECX and EDX registers are heavily used for this calling convention. The second indicator is the name of the function itself as it has some strange additions to it, like the "@" before and after the function name. The "@" before the function name indicates __fastcall and the "@" followed by a number indicates the size of bytes in arguments. With these being 32-bit values, 12 means there are three arguments to SomeFastFunction.



__thiscall
Another calling convention that is specific to Microsoft and built for x86 C++ class member functions is thiscall. Just like cdecl and stdcall, all arguments are pushed on the stack in a right to left manner. The new item here is that the compiler will sneak in a pointer. This pointer points to a table that is indexed appropriately for the member function being called. The pointer is called this pointer and ECX is primarily used to hold it.

A quick refresher on class member functions is on the following slide. The class, MyClass, is defined as a class that will have one function available for public use; PrintMsg. The method is not defined inside the class but instead is done outside using the scope resolution operator.

There is a somewhat shorthand method of defining a class with the struct keyword. Methods defined this way default to public methods, so no reason to use public: like done in the class.

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 ); };
```


Example: __thiscall

If you are not familiar with C++ classes, they can get complicated rather quickly. To keep things as simple as possible just to serve a point, the example on the slide creates a basic class called MyClass. The class has a single public class method called PrintMsg that accepts a pointer to char, probably a pointer to a variable holding a string or something similar. One nice thing about C++ is you do not always have to use the class keyword. Creating a struct and adding in the member functions is the same thing. Also notice the lack of the public keyword in the struct. This is because public is the default unless you specify private or something else. If you do not want your member functions or variables to be used outside of your class definition, specify private. The last item on the slide is how a member function can be defined outside of the main class body. You can also define them in a source file that includes your class header file.

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

struct TheClass {void TheMessage( char* msg ); };    // same as above with public
being default for TheMessage

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

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.

Lab 1.4: Call Me Maybe
This lab has multiple objectives. The primary one is to create a __stdcall decorated function and a __cdecl decorated function. Once done, we will toss your compiled binaries into IDA Pro so we can visually see the difference between calling conventions.

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

What's the Point?

What's the point?

What's the Point?
The point of the lab was multi-purpose: it allowed you to explore calling conventions from a source code perspective as well as see it from an RE perspective. It is very interesting to get down to the low-level stuff.

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`

Module Summary
In this module, we discussed many calling conventions that could be used in your programs. Some of the conventions can offer speed benefits like fastcall depending on how many arguments are accepted.

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 Questions
Q: 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 Answers
Q: 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 Questions
Q: 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 Answers
Q: 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 Questions
Q: For 64-bit, why do stack arguments start at RSP+20 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

Unit Review Answers
Q: For 64-bit, why do stack arguments start at RSP+20 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

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will introduce the Microsoft source-code annotation language.

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.

What Are They?
SAL stands for the Microsoft source-code annotation language and can be used in your code. The primary purpose is to describe just how function parameters will be used by a function. You can look at the Sal.h header where all of them are defined.

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

Why Use Them?
If you have never done development on Windows before, then SAL markups will look very odd to you at first. Using them will make your code so much clearer and explicit. The parameters for your functions on how they are used will be easily interpreted.

Aside from making your code easier to interpret, they can help make your code safer. The compiler will check your code for you in a very inexpensive way. When we start diving into the Windows APIs, you will see a large variety of SALs being used. This really makes interpreting WinAPIs much easier and unfortunately, a lot of online blogs do not include SALs when describing a particular API.

If you would like to browse the header file now without spinning up your Windows VM, then check out this GitHub repo that's hosting a version of it:
https://github.com/dotnet/corert/blob/master/src/Native/inc/unix/sal.h.

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 (1)
The listed annotations on the slide are deemed to be basic annotations because they can be read and interpreted quickly. They are also very easy to implement in your code as a junior Windows developer. As you develop and use the annotations more and more, you will start to understand where you can include more advanced annotations in your functions.

_In_ is for data that will be treated as read-only and if your function attempts to modify that data, then the code analysis should catch that before your program is executed.

Out is for output that will be written to an address space provided by the caller. This is in addition to the function's returned value, so be sure to keep them separate. Some functions might just return a Boolean type but have an _Out_ parameter that holds the real or the needed information you were wanting.

Inout indicates input to the called function and output to the caller. This also indicates that the data being passed in could also be modified.

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

Basic Annotations (2)

_In_opt_ indicates that a pointer will be passed as an input parameter but also that the pointer could be null.

_Out_opt_ allows you to still interpret that data is being passed back to the caller but is optional.

_Ret_z_ would indicate that a null-terminated value will be returned. This one is different than the rest of the annotations listed on the slide because it is dealing with the function itself instead of the function's parameters.

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
    );
```


Examples: Basic Annotations
The *CreateProcessA* API is a great example for some of the basic annotations. You can see many of the basic annotations that were listed on the previous slide are utilized to mark up this function's parameters. When you see the annotations in a real example it becomes clearer how they are used. Let's break down the annotations in more detail.

_In_opt_ indicates the *lpApplicationName* does not have to be supplied. *nullptr* can be passed here just fine without causing any warnings or errors. This is because the ***CreateProcess*** function will be checking to see if the parameter is NULL or not.

Moving on to the lpComandLine, the _Inout_opt_ indicates that this parameter is optional as well. On top of that, the _Inout_ portion indicates that the value could be modified by the function. This makes sense since the data type used here is LPSTR and not LPCSTR, which would indicate a constant value. Constant values cannot be modified and attempts to do so will generate an access violation.

Next is bInheritHandles. The _In_ indicates input, read-only data that is of type BOOL. A one (1) or a zero (0) will be passed for this parameter.

Finally, we come to lpProcessInformation. _Out_ indicates that the address will have data written to it by the function and the caller is responsible for creating that space. Since the parameter is of type LPPROCESS_INFORMATION, the caller must create a variable of type PROCESS_INFORMATION and pass the address to it. Notice this is not an optional OUT parameter. The function must have a place to store the newly created process' information like its ProcessId, among other items.

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 (1)
All the annotations on these two slides have one parameter, except the last one: _Ret_maybenull_. The parameter can be one of two values, a constant or the name of a parameter, like a count or a size parameter. The choice would be yours in the end, of course.

_In_reads_(s) should be used where the parameter could have more than one element to be read. The annotation's parameter 's' will be used to describe the number of elements. The addition of _bytes_(s) to this annotation will read in just that—bytes. The 's' parameter is used the same way but just indicates the number of bytes being read instead of the number of elements. This is a small difference, so be sure to use the proper annotation.

_In_reads_z_(s) indicates that the parameter is a null-terminated one with the potential to have more than one element. When you use the _or_z_(s) option, you are indicating that the parameter's elements will be read up to 's' elements. Of course, it won't read beyond hitting a null terminator, so whichever one comes first; 's' elements or null.

_Out_writes_(s) is for an out parameter that will write one or more elements. Like the other annotations on this slide, the 's' parameter will dictate the number of elements to write. The extension of _bytes_(s) is just like _In_reads_bytes_(s) where the number of bytes will be written instead of the number of elements.

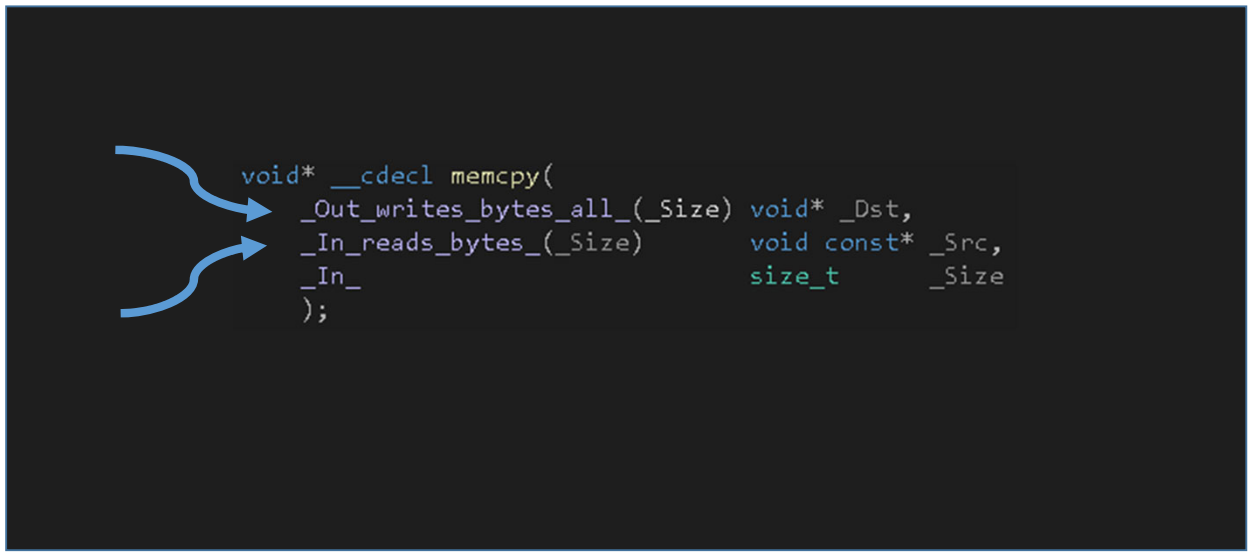
Intermediate Annotations (2)

_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

Intermediate Annotations (2)
_Out_writes_bytes_all_(s) is simply a variant of the _Out_writes_all_ annotation, but the 's' parameter indicates the number of bytes instead of elements.

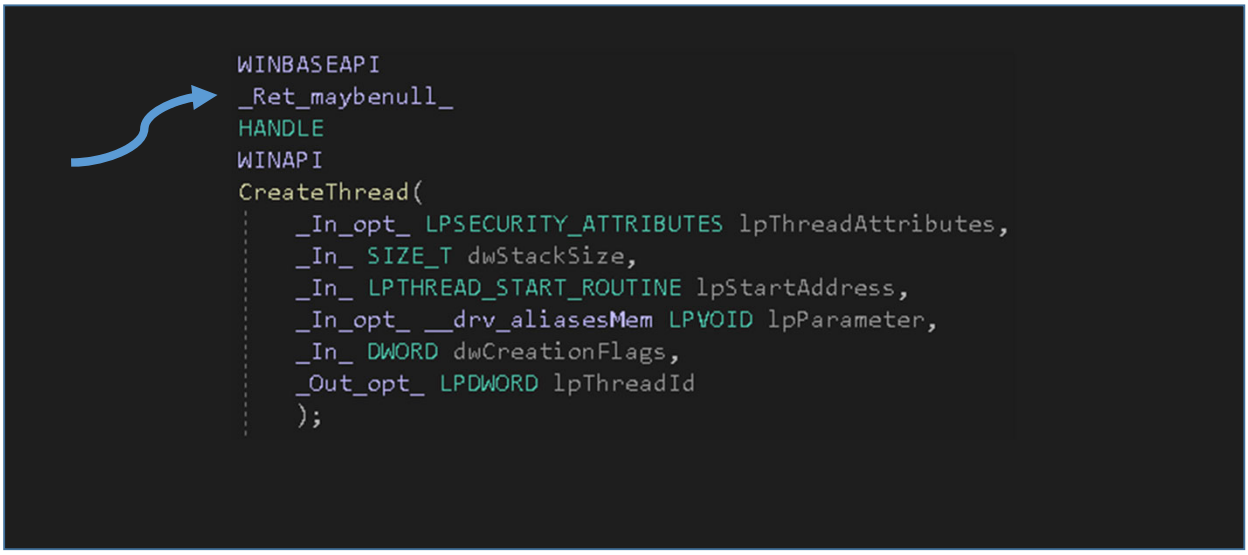
_Ret_maybenull_ is straightforward. The value that is to be returned could be a nullptr, will be a nullptr, or will not be a nullptr.

Example: Intermediate Annotations (1)



Example: Intermediate Annotations (1)
The first example on the slide is the memcpy declaration. You might already be familiar with this function, but perhaps not with SAL annotations. The annotations make it much more legible, and it is clear how the function intends to use its parameters. The first parameter, _Dst, is annotated with _Out_writes_bytes_all_() with the size field being passed as an argument. This indicates that all bytes up to size will be written to the buffer pointed to by the _Dst parameter.

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
    );
```


Example: Intermediate Annotations (2)

The second example is the *CreateThread* API. If you were reading the declaration too quickly, you might have missed the annotation that describes the function itself. The _Ret_maybenull_ is used here to indicate that the return value might not be a handle to the thread. Something could go wrong and prevent the function from succeeding, so the caller must have some type of error checking in place just in case NULL is returned.

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 (1)
There are way more annotations that could be deemed as advanced, but these were the ones worth noting.

_Inout_updates_(s) takes in a pointer to some array of elements that will be read from and written to by the function. The 's' parameter is—you guessed it—the number of elements the array holds. Of course, the _bytes_ versions of this annotation is used when expressing the number of elements does not make sense, so the number of bytes is passed instead. A good time to use _bytes_ is when dealing with CHAR data type.

_Check_return_ is for the return value of a function that must be checked by the caller. An error will be generated if the function is called in a void context because you can't check the return value of a void function. Best to use this annotation when the function will return an error code or even a success code. A variant is _Must_inspect_result_.

_Use_decl_annotations is an annotation that must only be used for a function declaration. Do not attempt to use this to describe a function's parameter. This annotation will force the reuse of the annotations for a function's declaration as to avoid the duplication of that function's annotations. This will not work if the declaration is not in scope, and you don't add any additional annotations in the definition of the function. Remember, the declaration is not the same thing as the definition.

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.

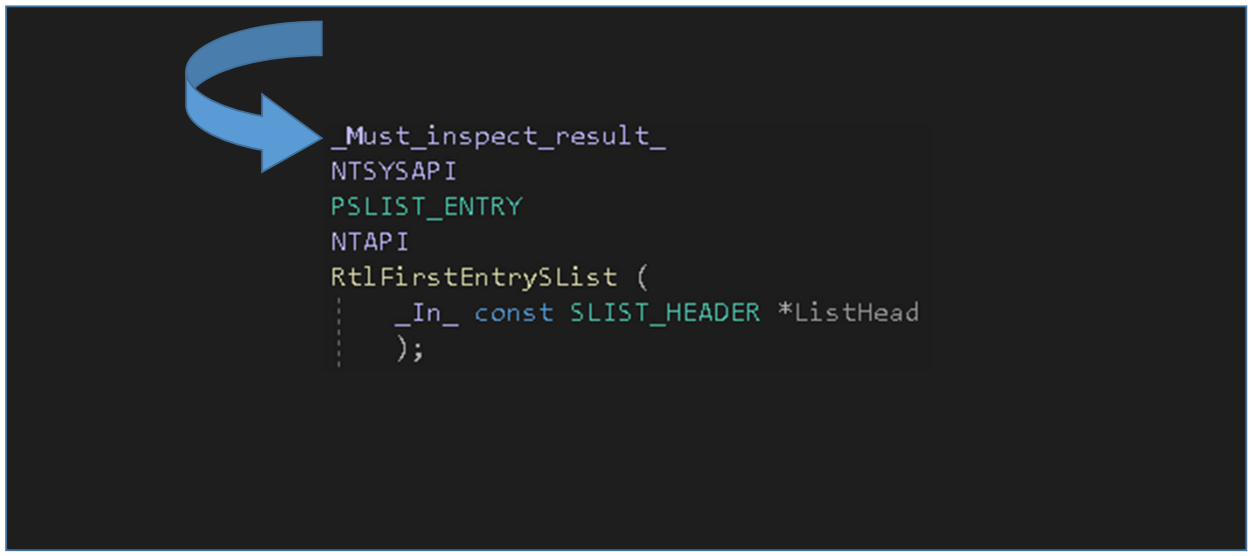
Advanced Annotations (2)

Success(expr) since functions can fail, care must be taken to know what truly indicated success or failure for that function. The expr is an expression that will generate the return value and there can only be one such annotation for the function's declaration and definition. This annotation cannot be used for a function's parameters.

When(expr, anno-list) can be pretty complex depending on the expression created for it. If there is a second parameter passed to it, then it will only apply when the expression passed for the first parameter results in a nonzero value. In other words, the expression must evaluate to a bool or be converted to one. Another way to think about this one is an if statement. If the expr is TRUE (1), then whatever is passed for anno-list will become applicable.

_Ret_writes_to_(s, c) is perfect to describe the return value when it is a scalar, a pointer to a struct, or some pointer to a buffer. The 's' parameter is no different from what we have seen so far, it's just the size of the array. The 'c' parameter is for the number of elements that have been written. This annotation can be tricky and takes some practice to fully understand it.

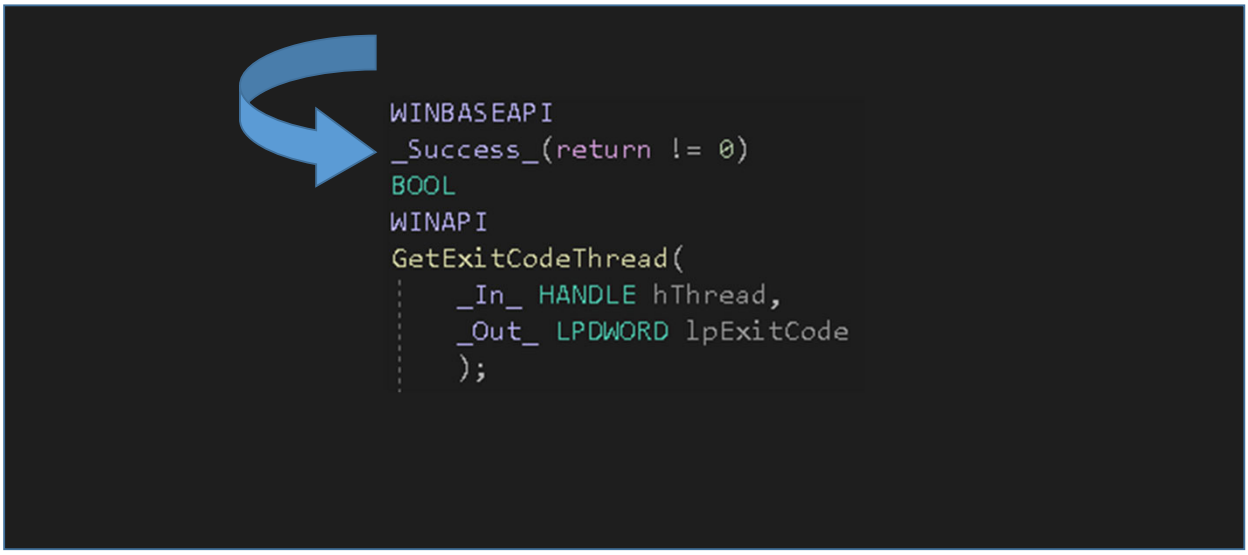
Example: Advanced Annotations (1)



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


Examples: Advanced Annotations (1)
The first example here is of the function **RtlFirstEntrySList**, which will grab the first entry in a singly linked list. The only item you need to provide is the address of the ListHead. The item of interest here is the annotation being used to describe the function, _Must_inspect_result_. This is really just a variant of _Check_return_, but both annotations indicate that the caller must have code in place that checks the return value of the function.

Example: Advanced Annotations (2)



Example: Advanced Annotations (2)
The second example is of the API ***GetExitCodeThread***, which is used to get a thread's termination status. Side note here: notice how the function is declared with a BOOL return type. This means that the function will not be returning an exit code but simply a TRUE/FALSE status if the function succeeded or not. Notice the _Out_ parameter—this is where the exit code will be going.

Anyway, notice the _Success_() annotation describing the function. This indicates that a successful return value for this function will not be equal to 0. Any nonzero would indicate success in this case and only 0, or NULL, would indicate failure.

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 (3)
The example here is of the ***InitializeProcThreadAttributeList*** API, which is used to initialize a list of attributes to be applied to a thread or a process. The function declaration can seem complex because of the annotations used, but let's break it down. We already went over the _Success_ annotation, so let's skip straight to:

_Out_writes_bytes_to_opt_()

The last annotation to interpret is _When_(<expression>, <anno-list>). Looking at the first _When_(), you can see that when the *lpAttributeList* has a nullptr passed in, then it will be treated as an _Out_. The second _When_() is indicating that if there is a pointer then it will be _Inout_. In other words, when there is a pointer to an attribute list, then the size will be the number of bytes of the attribute list buffer on input. For output, the parameter will receive the size in bytes of the initialized attributes list. If the attribute list is NULL, lpSize will simply receive the required buffer size in bytes.

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
    );
```


Example: Advanced Annotations (4)
The example here is of the ***WldpQueryDynamicCodeTrust*** API, which is used to grab a value to figure out if the specified in-memory dynamic code is trusted by Device Guard policy. It's a pretty cool function! Anyway, let's tackle some of these _When_() expressions.

At a high level, this is saying that if the *fileHandle* is not NULL, then *baseImage* better be NULL. The opposite is also true: if the *baseImage* is not NULL, then the *fileHandle* better be NULL. Okay, on to the expressions.

The first one indicates that the *fileHandle* will be optional when the *baseImage* is not NULL. The second one indicates that the *fileHandle* will be read-only and required when the *baseImage* is NULL.

The third annotation indicates that when the *fileHandle* is NULL, then reading in *imageSize* bytes is optional. When the *fileHandle* is not NULL, then the reading in of *imageSize* bytes is required.

Notice that *imageSize* is a required _In_ parameter no matter what.

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.

Lab 1.5: Safer with SAL
This lab is to get you familiar with using SAL annotations. Depending on your comfort level with SAL annotations, you can work on the easy, intermediate, or advanced versions of this lab. If you have time, you can work on all three. If you run out of time, you can work on them again during the bootcamp.

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

What's the Point?

What's the point?

What's the Point?
The point of this lab was the explore the various SAL annotations and how they could be used to describe a function's parameters or the function itself. Sadly, many sample projects you might find online do not use annotations and they lack proper descriptions of the function's parameters. The assumption must be that any experienced developer would be able to figure it out.

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
```


Additional Information
SAL annotations can be extremely complicated but once you use them more and more, you will begin to interpret them rather quickly. The MSFT docs are a great place to look at the annotations, but perhaps an even better place is inside the header files themselves. The header files will have annotations used in the declarations of most, if not all, of their functions. If you are looking for an example of a certain annotation, then you can leverage PowerShell to locate exactly where it is used. Use the **Select-String** cmdlet to perform your search. Here is an example that searches for _When_:

First, change directories to where the header files are located. One location could be: C:\Program Files (x86)\Windows Kits\10\Include\10.0.18362.0\um.

*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

Module Summary
In this module, we discussed many annotations that can be considered as basic, intermediate, and advanced. Eventually, all of them will become easy and basic for you the more often you use them. It is important to know what annotations can only be used to describe a function instead of the function's parameters.

Unit Review Questions

What does SAL stand for?

A Source-code annotation language

B Structured annotation language

C Silent analysis language

Unit Review Questions
Q: 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 Answers
Q: 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 Questions
Q: 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 Answers
Q: 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 Questions
Q: 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_)

Unit Review Answers
Q: 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

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will introduce some essential subject matter, concepts, and introductory topics required to perform advanced penetration testing and to proceed through this course.

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

Windows API

Becoming extremely familiar with the Windows API is a hard requirement for developing Windows tools. Your program would be rather bare and lack robust capabilities if it did not include Windows APIs. The Windows APIs are what provide a major part of functionality for Windows. If you wish to create a file, a new registry key, socket, a directory, etc. you will be using a Windows API to make it happen. Back in the day, before 64-bit versions came around, this was the Win32 API for Windows 32-bit versions because there were still 16-bit versions. Today, there is no need to make that distinction. Any reference to Win32 will apply to 32-bit and 64-bit versions.

The high dependency on Windows APIs is what makes creating shellcode so much more complicated than creating shellcode for Linux. Your shellcode will most likely need to call APIs to get its job done, so you need to push the arguments to the stack, and in the correct order. For these reasons, Windows shellcode is typically much larger than Linux shellcode. If you have taken SEC660 then you have direct experience with this hurdle since you spend an entire day with Linux and an entire day with Windows.

Create APIs (1)

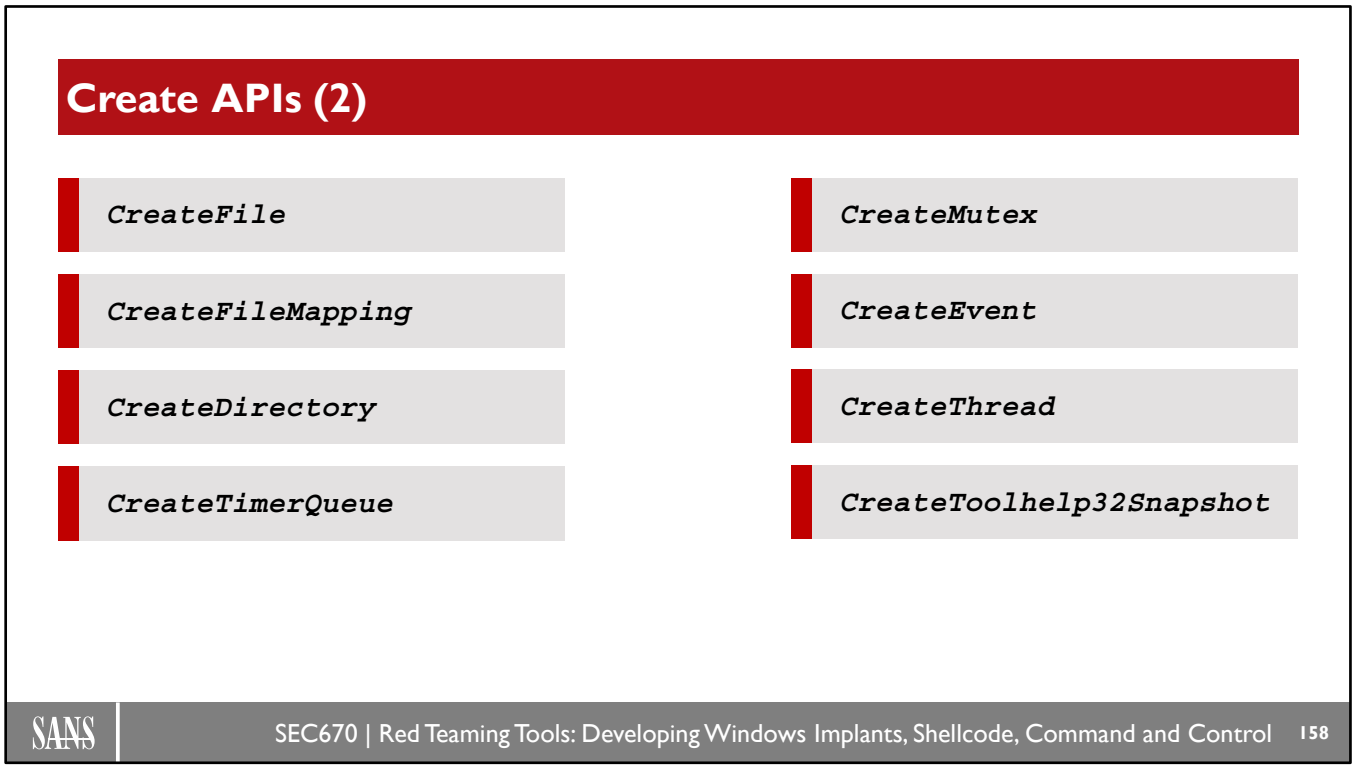
Objects and handles make the world go around.

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 (1)
There are many Create* APIs, and if you check out the Microsoft docs you would find at least 96 APIs that start with Create. This list is a little old so they might be more than what they have listed. The general theme with the Create family of APIs is they will create an object and return a handle to said object.

In all reality, the function is not creating the object—the kernel is the one creating the actual object and the objects live in kernel space. The handles that represent the objects are what get returned and they can be used in user space. The handles must be in user address space because your program must have a way to reference the object and take action against it. One could imagine how serious of a security issue it might be to have direct access to all kernel objects that reside in system space. Therefore, handles to those objects exist as liaisons, if you will.



Create APIs (2)

The functions listed on the slide are just a very small subset of the *Create** family of functions. As mentioned previously, there are almost 100 *Create** functions to use. For most of them, it is easy to understand what their intended purpose is due to their descriptive names, like *CreateFile*. You can expect a handle to the newly created object to be returned for most of these, and it is this handle that allows you to interact with the object in system space. All objects are created in system space and user space is given handles to them. You will be using many of these throughout the remainder of the course. On your own time, you can read about them in great detail on MSDN before we start using them for labs.

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
);
```


Create APIs (3)

Creating processes on a target system can be very useful in accomplishing your objectives. The ***CreateProcessW*** API must be given a valid executable image, or it will fail. It is not as fancy as what Explorer uses behind the scenes that will determine the file type and choose the appropriate executable. If you were to double-click on a TXT file, Explorer uses the ***ShellExecuteEx***, or similar, to search through the Registry and pick the application based on the file extension type association found for it.

CreateProcessW takes nine parameters and will return TRUE upon success. The return value can be misleading because the process can still fail to start but the function returned TRUE anyway. When the kernel creates the new primary thread of the process, its job is done and returns. The real proof of success is what is returned via the last parameter: *lpProcessInformation*. Let's back up to the first parameter.

pApplicationName is a pointer to the name of the module you want to execute, and it must be a valid Windows application. You can pass in the full path or just a partial name because the function will use whatever the current drive and directory is to complete it. The search path is not used, and you must provide the extension since a default .exe is not used. You can also choose not to pass anything for this parameter, which is pretty common practice. The second parameter offers more benefits and is more heavily utilized by developers.

pCommandLine has a primary purpose of passing the command-line arguments to the image specified for lpApplicationName, but when the first parameter is NULL, you can pass in the application name here. The advantage of using this parameter for the application name is that you don't have to provide the extension because the function will add the EXE for you and start its search. That brings us another benefit. If the full path is not passed in, then the system will perform a search for it just like how the system searches for DLLs to load.

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);
```


Example: CreateProcess
For this example of how to use ***CreateProcess***, we are simply creating the notepad process with bare minimum effort, meaning, we are not taking full advantage of what the ***CreateProcess*** function has to offer. The previous slide went over the parameters so there is no need to go over them again here, rather the relevant ones will be discussed.

First up, there is some standard housekeeping that must be done by the way of creating some variables with specific structure types: **STARTUPINFO** and **PROCESS_INFORMATION**. Each of those structures will have information filled out after ***CreateProcess*** returns. There is the commandLine variable that is simply holding the name of the process we wish to create, and the address is passed to the ***CreateProcess*** function. The last two parameters are for the addresses of the two structures we made.

This is intentionally omitting error handling and the closing of the returned handles due to space limitations.

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 (4)

The **PROCESS_INFORMATION** structure is where you can find more details about your newly created process. This is the proof as to whether the ***CreateProcess*** function really worked. There are two handles in here that are created when the process and thread have spun up. Remember to close these handles when they are not needed any longer. The other housekeeping information are the IDs of the process and the thread. If you are checking to see if the process was truly created successfully, you could do so with the hProcess handle that should have been given to you. Since the CreateProcess API is of type BOOL, it can return TRUE upon success, but the process could fail to be created due to an unknown exception or error.

You can point to this structure using **PPROCESS_INFORMATION** or **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
);
```


Create APIs (5)

The ***CreateToolhelp32Snapshot*** API can be used to aid in enumerating processes on the local system. It will create a snapshot of the process and return a handle to that snapshot. You can use the handle to perform your queries and extract the information you are looking for, like a specific process name or module name the process loaded. The function is relatively easy to call since it only takes two parameters of the same type. The dwFlags parameter is the most important as it dictates what data should be collected in the snapshot. There are seven flags that can be passed here, but the most interesting flag for us is **TH32CS_SNAPPROCESS** because, as the name implies, all processes in the system will be included. There are other flags that can be used to capture modules and threads—those will be explored later during labs and extended hours.

Enumeration can be performed using the *Process32First* function. Again, process enumeration will be covered in detail later in the course.

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.

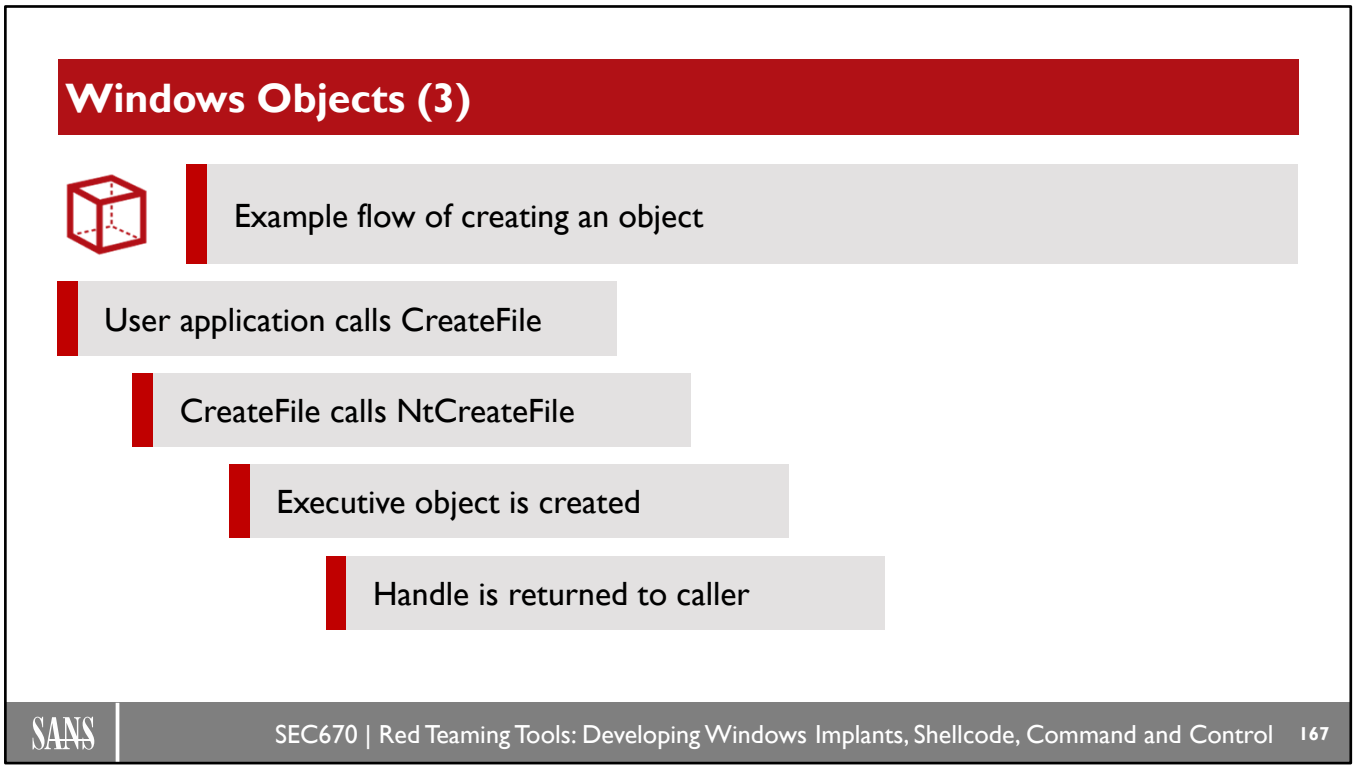
Reference:
Mark Russinovich, David A. Solomon, and Alex Ionescu, *Windows® Internals, Part 1*, *6th Edition* (Washington, Microsoft Press, 2012), chap. 3.

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)

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

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.

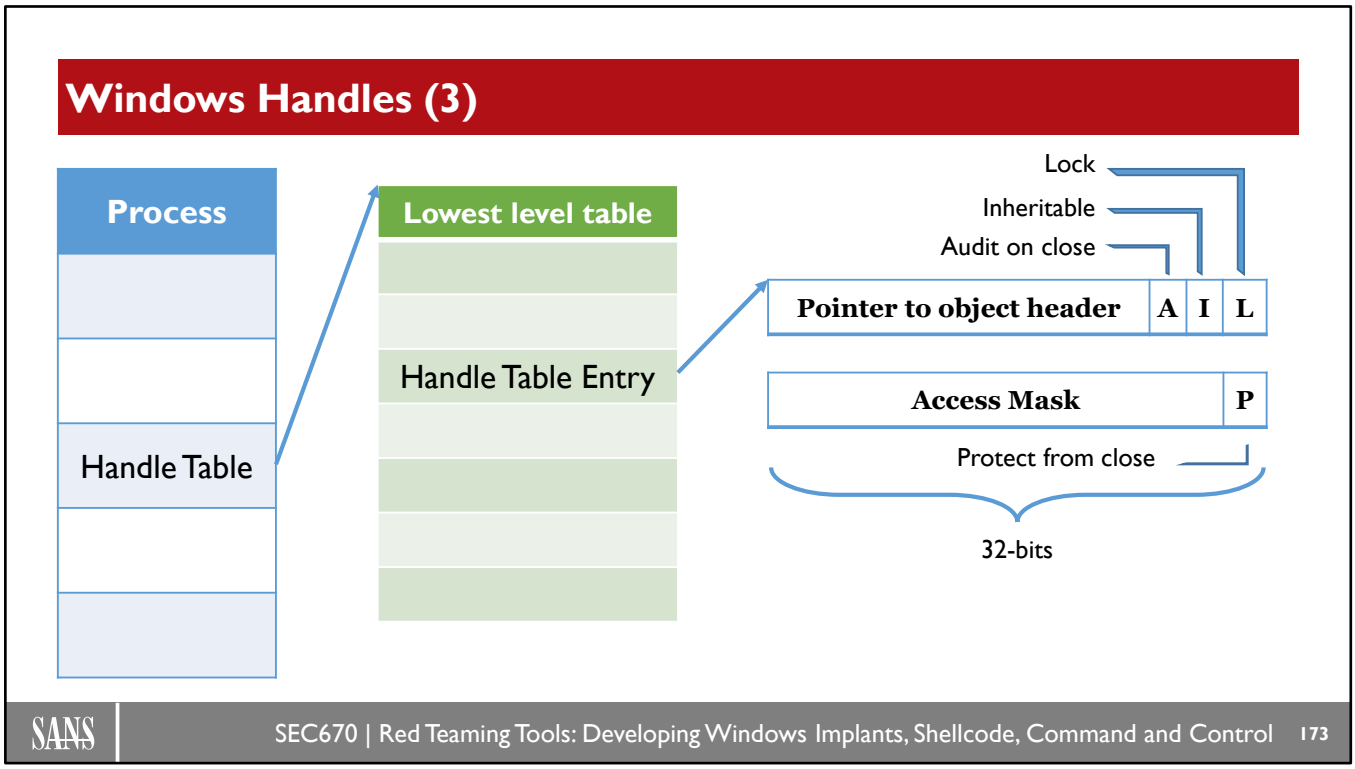
Various APIs require a valid object handle to manipulate it.

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)

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);
```


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***

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);
```


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?

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

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

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

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

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 I

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

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

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.