Binary Exploitation

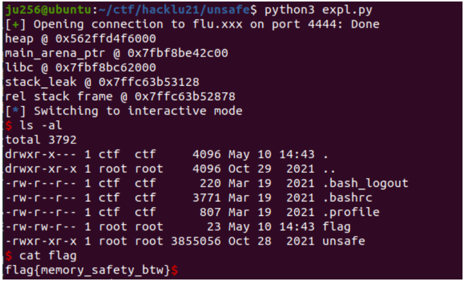
Julian Gremminger | 12.05.2022

Overview

- Finding and exploiting bugs in a binary/executable
- Programs written in low-level language
- Reverse engineering often mandatory first step
- Memory corruption vs logic bugs

Binary Exploitation in CTFs

- Often C/C++ binaries written for the competition
- Sometimes real world targets with introduced bugs
  - Chrome: Google CTF 2021 Fullchain [1]
  - Firefox: 33c3 CTF Feuerfuchs [2]
- Objective: Remote Code Execution on challenge server
  - Linux: call system("/bin/sh")

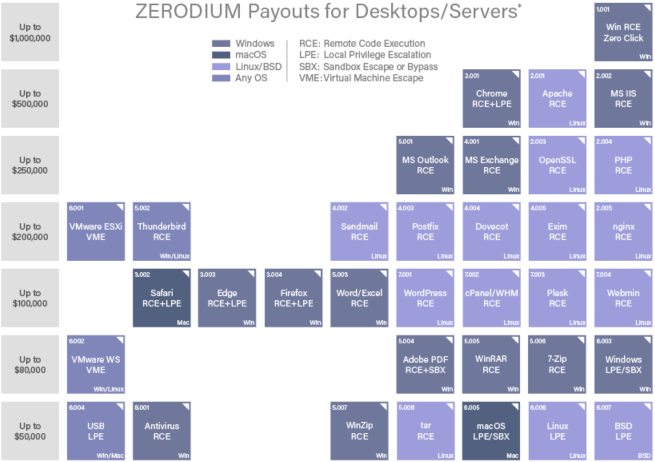


```
ju256@ubuntu:~/ctf/hacklu21/unsafe$ python3 expl.py
[+] Opening connection to flu.xxx on port 4444: Done
heap @ 0x562ffd4f6000
main_arena_ptr @ 0x7fbf8be42c00
libc @ 0x7fbf8bc62000
stack_leak @ 0x7ffc63b53128
rel stack frame @ 0x7ffc63b52878
[*] Switching to interactive mode
$ ls -al
total 3792
drwxr-x--- 1 ctf  ctf     4096 May 10 14:43 .
drwxr-xr-x 1 root root    4096 Oct 29  2021 ..
-rw-r--r-- 1 ctf  ctf      220 Mar 19  2021 .bash_logout
-rw-r--r-- 1 ctf  ctf     3771 Mar 19  2021 .bashrc
-rw-r--r-- 1 ctf  ctf      807 Mar 19  2021 .profile
-rw-rw-r-- 1 root root      23 May 10 14:43 flag
-rwxr-xr-x 1 root root 3855056 Oct 28  2021 unsafe
$ cat flag
flag{memory_safety_btw}$
```

Binary Exploitation in the „Real World“

- Memory-unsafe languages still widely used
  - Browsers
  - Hypervisors
  - Web servers
- Even the „best“ codebases contain exploitable bugs

Binary Exploitation in the „Real World“



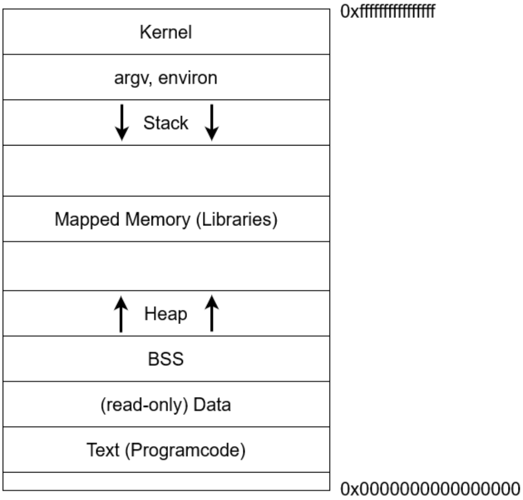
ZERODIUM Payouts for Desktops/Servers*

- ■ Windows
- ■ macOS
- ■ Linux/BSD
- ■ Any OS

RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass
VME: Virtual Machine Escape

Up to $1,000,000									1.001 Win RCE Zero Click Win
Up to $500,000							3.001 Chrome RCE+LPE Win	2.001 Apache RCE Linux	2.002 MS IIS RCE Win
Up to $250,000						5.001 MS Outlook RCE Win	4.001 MS Exchange RCE Win	2.003 OpenSSL RCE Linux	2.004 PHP RCE Linux
Up to $200,000	6.001 VMware ESXi VME	5.002 Thunderbird RCE Win/Linux			4.002 Sendmail RCE Linux	4.003 Postfix RCE Linux	4.004 Dovecot RCE Linux	4.005 Exim RCE Linux	2.005 nginx RCE Linux
Up to $100,000		3.002 Safari RCE+LPE Mac	3.003 Edge RCE+LPE Win	3.004 Firefox RCE+LPE Win	5.003 Word/Excel RCE Win	7.001 WordPress RCE Linux	7.002 cPanel/WHM RCE Linux	7.003 Plesk RCE Linux	7.004 Webmin RCE Linux
Up to $80,000	6.002 VMware WS VME Win/Linux					5.004 Adobe PDF RCE+SBX Win	5.005 WinRAR RCE Win	5.006 7-Zip RCE Win	6.003 Windows LPE/SBX Win
Up to $50,000	6.004 USB LPE Win/Mac	8.001 Antivirus RCE Win			5.007 WinZip RCE Win	5.008 tar RCE Linux	6.005 macOS LPE/SBX Mac	6.006 Linux LPE Linux	6.007 BSD LPE BSD

Linux Process Layout



0xffffffffffffffff

Kernel

argv, environ

Stack

Mapped Memory (Libraries)

Heap

BSS

(read-only) Data

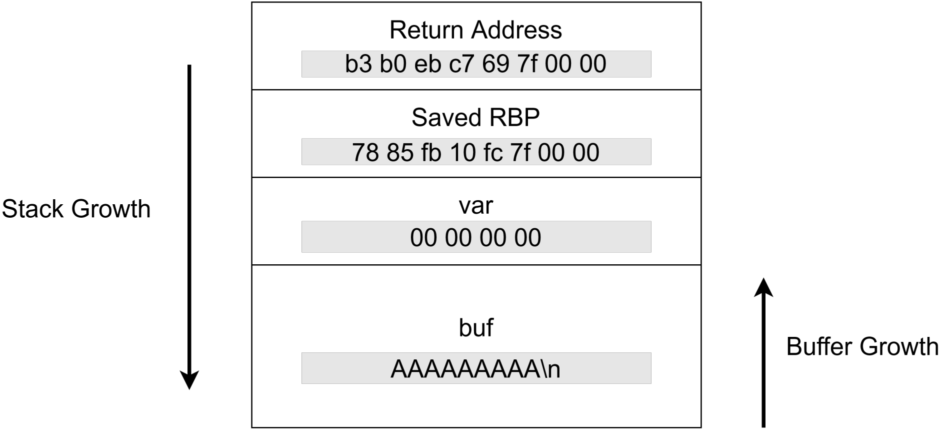
Text (Programcode)

0x0000000000000000

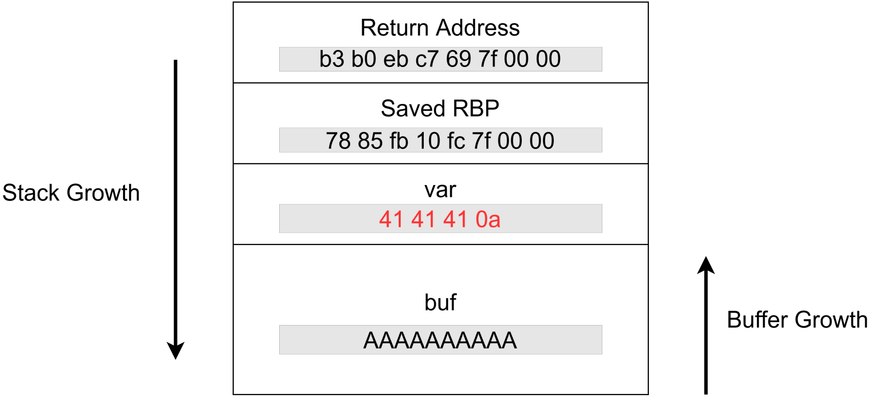
Buffer Overflows

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int var = 0;
    char buf[10];
    gets(buf);
    if (var != 0) {
        printf("%s", "success!");
    }
    return 0;
}
```

Stack Frames



Overflowing the Buffer



Stack Growth

Return Address

b3 b0 eb c7 69 7f 00 00

Saved RBP

78 85 fb 10 fc 7f 00 00

var

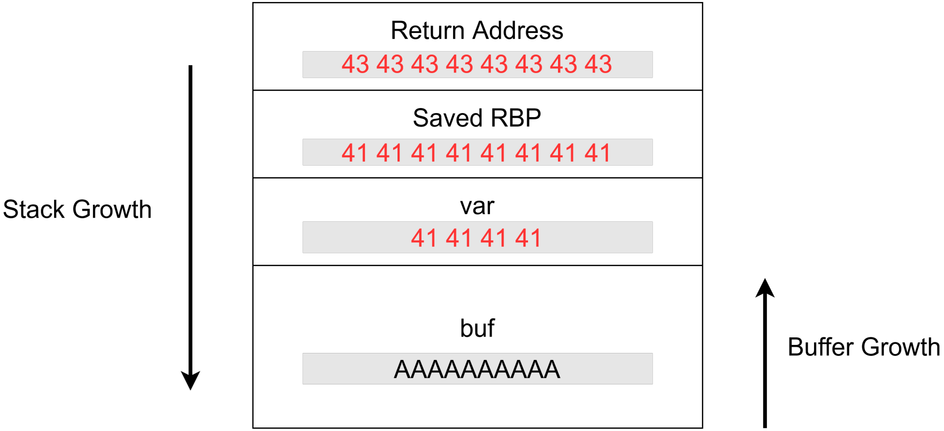
41 41 41 0a

buf

AAAAAAAAAA

Buffer Growth

RIP-Control?



Return Address

43 43 43 43 43 43 43 43

Saved RBP

41 41 41 41 41 41 41 41

var

41 41 41 41

buf

AAAAAAAAAA

Stack Growth

Buffer Growth

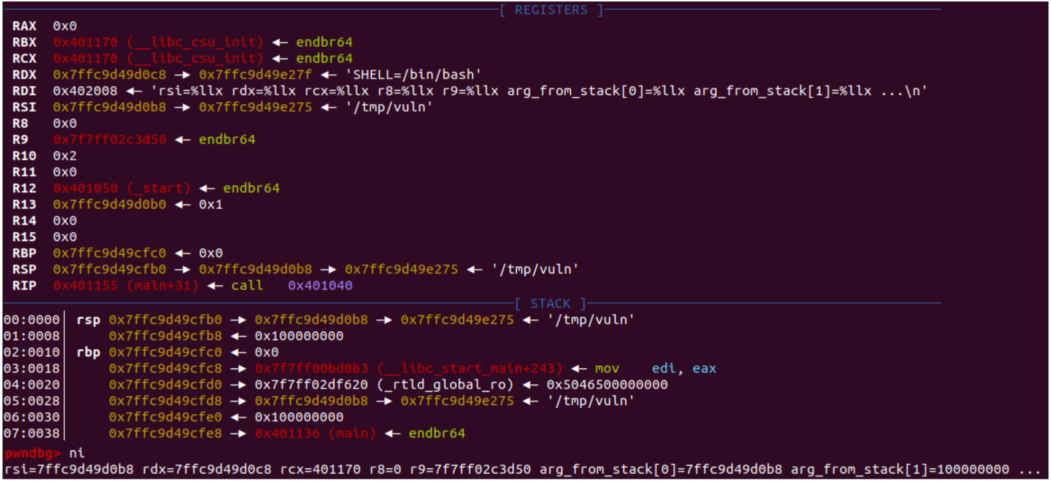
- RIP-Control after execution of ret instruction (RIP = 0x4343434343434343)

Format String Bugs

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("rsi=%llx rdx=%llx rcx=%llx r8=%llx r9=%llx"
           "arg_from_stack[0]=%llx arg_from_stack[1]=%llx ...\n");
}
```

- No arguments supplied to printf
- What happens?

Format String Bugs



```
─────────────────────────────────────────[ REGISTERS ]─────────────────────────────────────────
 RAX  0x0
 RBX  0x401170 (__libc_csu_init) ◂— endbr64
 RCX  0x401170 (__libc_csu_init) ◂— endbr64
 RDX  0x7ffc9d49d0c8 —▸ 0x7ffc9d49e27f ◂— 'SHELL=/bin/bash'
 RDI  0x402008 ◂— 'rsi=%llx rdx=%llx rcx=%llx r8=%llx r9=%llx arg_from_stack[0]=%llx arg_from_stack[1]=%llx ...\n'
 RSI  0x7ffc9d49d0b8 —▸ 0x7ffc9d49e275 ◂— '/tmp/vuln'
 R8   0x0
 R9   0x7f7ff02c3d50 ◂— endbr64
 R10  0x2
 R11  0x0
 R12  0x401050 (_start) ◂— endbr64
 R13  0x7ffc9d49d0b0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7ffc9d49cfc0 ◂— 0x0
 RSP  0x7ffc9d49cfb0 —▸ 0x7ffc9d49d0b8 —▸ 0x7ffc9d49e275 ◂— '/tmp/vuln'
 RIP  0x401155 (main+31) ◂— call   0x401040
───────────────────────────────────────────[ STACK ]───────────────────────────────────────────
00:0000│ rsp 0x7ffc9d49cfb0 —▸ 0x7ffc9d49d0b8 —▸ 0x7ffc9d49e275 ◂— '/tmp/vuln'
01:0008│     0x7ffc9d49cfb8 ◂— 0x100000000
02:0010│ rbp 0x7ffc9d49cfc0 ◂— 0x0
03:0018│     0x7ffc9d49cfc8 —▸ 0x7f7ff00bd0b3 (__libc_start_main+243) ◂— mov    edi, eax
04:0020│     0x7ffc9d49cfd0 —▸ 0x7f7ff02df620 (_rtld_global_ro) ◂— 0x5046500000000
05:0028│     0x7ffc9d49cfd8 —▸ 0x7ffc9d49d0b8 —▸ 0x7ffc9d49e275 ◂— '/tmp/vuln'
06:0030│     0x7ffc9d49cfe0 ◂— 0x100000000
07:0038│     0x7ffc9d49cfe8 —▸ 0x401136 (main) ◂— endbr64
pwndbg> ni
rsi=7ffc9d49d0b8 rdx=7ffc9d49d0c8 rcx=401170 r8=0 r9=7f7ff02c3d50 arg_from_stack[0]=7ffc9d49d0b8 arg_from_stack[1]=100000000 ...
```

Format String Bugs

```c
#include <stdio.h>

#define SIZE 0x100

int main(int argc, char* argv[]) {
    char buf[SIZE];
    fgets(buf, SIZE, stdin);
    printf(buf);
    return 0;
}
```

- User-controlled format string
- Can we exploit this?

Format String Exploitation Building Blocks

- `%n` Write amount of already printed bytes to an address
- This address will be taken from the „argument stream“
  - If our buffer resides on the stack we can choose this address (put address in the format string)
  - There might be interesting pointers on the stack already
- Writes of different sizes possible
  - `%n` => *(int *) write
  - `%hn` => *(short int *) write
  - `%hhn` => *(char *) write

Format String Exploitation Building Blocks

- Meaningful stuff in „already printed bytes“?
- `printf(”AAAAAAAA%hhn”)` results in `*(char *)$rsi = 0x8`
- Shortcut for setting „already printed bytes“: `%<Padding>c`
  - `printf(”%255c%hhn”)` results in `*(char *)$rsi = 0xff`

Format String Exploitation Building Blocks

- How to access supplied addresses in the format string?
- Positional parameters: `%_$`
  - `%4$x` will access the same value as the last %x in `%x%x%x%x`
- Full arbitrary 8-byte write to given address:

```
%{short_write_val}c%10$hn%11$hn%12$hn%13$hn {addr + 0}{addr + 2}{addr + 4}{addr + 6}

With addr = 0x4141414141414141 and short_write_val = 0x7777
%30551c%10$hn%11$hn%12$hn%13$hn AAAAAAAACAAAAAAAEAAAAAAAGAAAAAAA

*(short int*)(0x4141414141414141 + 0) = 0x7777
*(short int*)(0x4141414141414141 + 2) = 0x7777
*(short int*)(0x4141414141414141 + 4) = 0x7777
*(short int*)(0x4141414141414141 + 6) = 0x7777
```

Integer Bugs

- Overflows and Underflows
  - 2147483647 + 1 == -2147483648
  - -2147483648 - 1 == 2147483647
- Comparison bugs
  - Explicit or implicit casts of values can lead to unexpected behavior

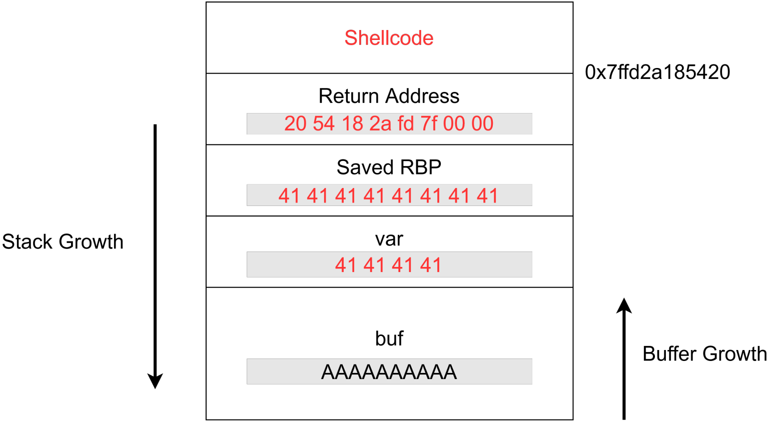
```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    char buf[0xff];
    int size = 0;

    scanf("%d", &size);
    if (size < 0xff) {
        read(0, &buf, size);
    } else {
        puts("Invalid size");
    }
    return 0;
}
```

Use-after-free

- Pointer to memory not cleared after free => Dangling pointer
- If this memory gets reallocated type confusions might occur
- Heap metadata corruption

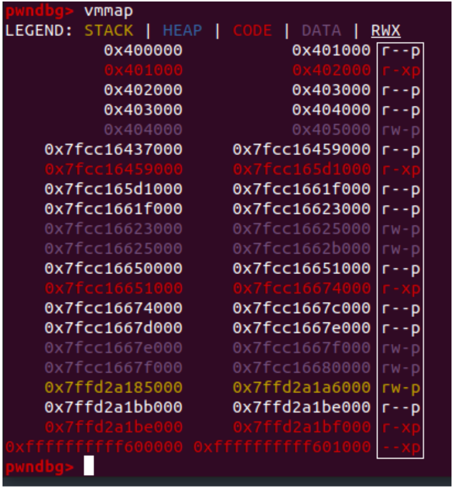
RIP-control to shell

- Shellcode: Inject our own code into memory and jump to it
  - Shellcode collection: http://shell-storm.org/



What’s the catch?

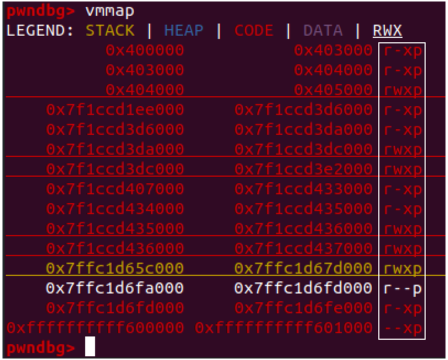
- **Mitigations**
- NX-Bit (No eXecute) / DEP
  - Page is writable XOR executable
  - Consequently stack not executable
  - Injected shellcode can’t be executed



```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX
          0x400000           0x401000 r--p
          0x401000           0x402000 r-xp
          0x402000           0x403000 r--p
          0x403000           0x404000 r--p
          0x404000           0x405000 rw-p
    0x7fcc16437000     0x7fcc16459000 r--p
    0x7fcc16459000     0x7fcc165d1000 r-xp
    0x7fcc165d1000     0x7fcc1661f000 r--p
    0x7fcc1661f000     0x7fcc16623000 r--p
    0x7fcc16623000     0x7fcc16625000 rw-p
    0x7fcc16625000     0x7fcc1662b000 rw-p
    0x7fcc16650000     0x7fcc16651000 r--p
    0x7fcc16651000     0x7fcc16674000 r-xp
    0x7fcc16674000     0x7fcc1667c000 r--p
    0x7fcc1667d000     0x7fcc1667e000 r--p
    0x7fcc1667e000     0x7fcc1667f000 rw-p
    0x7fcc1667f000     0x7fcc16680000 rw-p
    0x7ffd2a185000     0x7ffd2a1a6000 rw-p
    0x7ffd2a1bb000     0x7ffd2a1be000 r--p
    0x7ffd2a1be000     0x7ffd2a1bf000 r-xp
0xffffffffff600000 0xffffffffff601000 --xp
pwndbg> 
```

What’s the catch?

- **Mitigations**
- NX-Bit (No eXecute) / DEP
  - Page is writable XOR executable
  - Consequently stack not executable
  - Injected shellcode can’t be executed



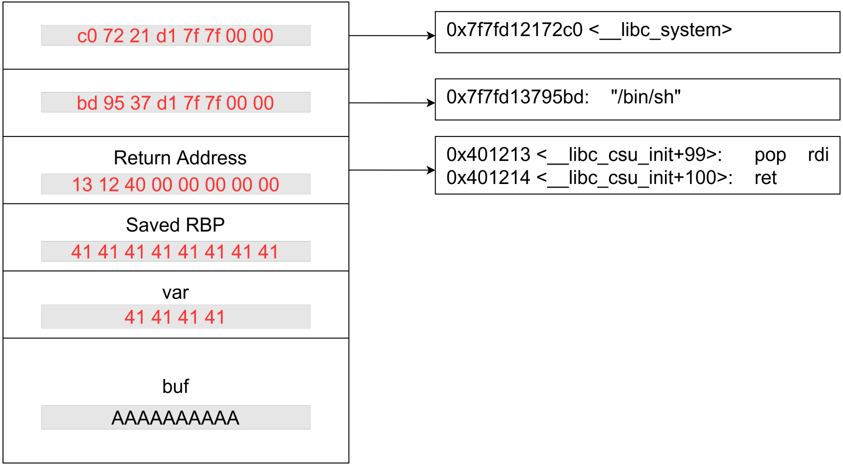
```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX
          0x400000           0x403000 r-xp
          0x403000           0x404000 r-xp
          0x404000           0x405000 rwxp
    0x7f1ccd1ee000     0x7f1ccd3d6000 r-xp
    0x7f1ccd3d6000     0x7f1ccd3da000 r-xp
    0x7f1ccd3da000     0x7f1ccd3dc000 rwxp
    0x7f1ccd3dc000     0x7f1ccd3e2000 rwxp
    0x7f1ccd407000     0x7f1ccd433000 r-xp
    0x7f1ccd434000     0x7f1ccd435000 r-xp
    0x7f1ccd435000     0x7f1ccd436000 rwxp
    0x7f1ccd436000     0x7f1ccd437000 rwxp
    0x7ffc1d65c000     0x7ffc1d67d000 rwxp
    0x7ffc1d6fa000     0x7ffc1d6fd000 r--p
    0x7ffc1d6fd000     0x7ffc1d6fe000 r-xp
0xffffffffff600000 0xffffffffff601000 --xp
pwndbg> 
```

No need for own code[1] (Code Reuse Attacks)

- Instead of injecting own code, use existing code
- Reuse code in binary or libraries
- For stack-based buffer overflow example:
  - Overwrite return address with pointer to existing code snippet („gadget“)
  - Gadgets can be chained together if they end in `ret`
    => Return-oriented programming (ROP)
- ropper [3] and ROPGadget [4] find gadgets and can even build full ROP-chains

[1] Requirements: Gadget addresses need to be known and useful gadgets have to exist

ROP



c0 72 21 d1 7f 7f 00 00

bd 95 37 d1 7f 7f 00 00

Return Address
13 12 40 00 00 00 00 00

Saved RBP
41 41 41 41 41 41 41 41

var
41 41 41 41

buf
AAAAAAAAAA

```
0x7f7fd12172c0 <__libc_system>
```

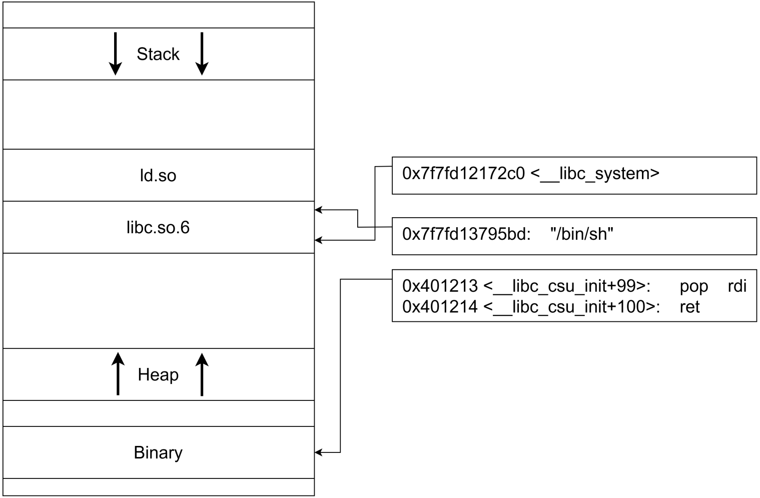
```
0x7f7fd13795bd:    "/bin/sh"
```

```
0x401213 <__libc_csu_init+99>:     pop    rdi
0x401214 <__libc_csu_init+100>:    ret
```

- Executed ROP-chain leads to call to system(”/bin/sh”)

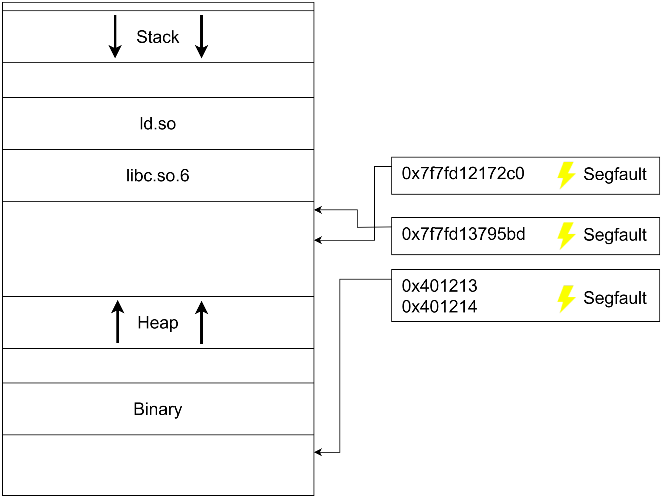
Mitigate Code Reuse Attacks

- So far we assumed we know addresses of gadgets, functions, libraries and stack


Stack
ld.so
libc.so.6
Heap
Binary
0x7f7fd12172c0 <__libc_system>
0x7f7fd13795bd: "/bin/sh"
0x401213 <__libc_csu_init+99>: pop rdi
0x401214 <__libc_csu_init+100>: ret

Mitigate Code Reuse Attacks

- So far we assumed we know addresses of gadgets, functions, libraries and stack
- Breaking this assumption breaks our attack



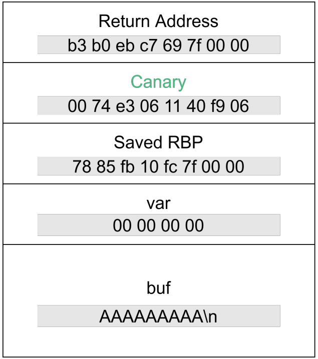
ASLR and PIE

- **A**ddress **S**pace **L**ayout **R**andomization
- Randomize memory layout on every execution
- Linux ASLR is based on 5 randomized (base) addresses
  - Stack, Heap, mmap-Base, vdso
  - Random base address for executable only if PIE is enabled

- Leak of **1** library address derandomizes all libraries
- Leak of **1** address in our binary breaks PIE
- Forked processes share layout with parent

Canaries

- Prevent stack-based buffer overflows
- 7 random bytes with least significant byte zero
- Set up in function prologue and verified in epilogue
- Invalid canary value leads to SIGABRT



Return Address
b3 b0 eb c7 69 7f 00 00

Canary
00 74 e3 06 11 40 f9 06

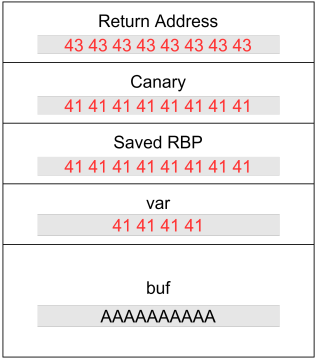
Saved RBP
78 85 fb 10 fc 7f 00 00

var
00 00 00 00

buf
AAAAAAAAA\n

```
0x401189 <+19>:   mov    rax,QWORD PTR fs:0x28
0x401192 <+28>:   mov    QWORD PTR [rbp-0x8],rax
...
0x4011d3 <+93>:   mov    rdx,QWORD PTR [rbp-0x8]
0x4011d7 <+97>:   sub    rdx,QWORD PTR fs:0x28
0x4011e0 <+106>:  je     0x4011e7
0x4011e2 <+108>:  call   0x401060 <__stack_chk_fail@plt>
0x4011e7 <+113>:  leave
0x4011e8 <+114>:  ret
```

Canaries



Return Address

43 43 43 43 43 43 43 43

Canary

41 41 41 41 41 41 41 41

Saved RBP

41 41 41 41 41 41 41 41

var

41 41 41 41

buf

AAAAAAAAAA

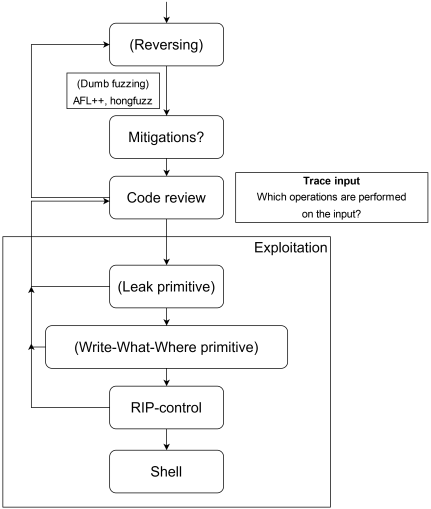
```
0x401189 <+19>:   mov    rax,QWORD PTR fs:0x28
0x401192 <+28>:   mov    QWORD PTR [rbp-0x8],rax
...
0x4011d3 <+93>:   mov    rdx,QWORD PTR [rbp-0x8]
0x4011d7 <+97>:   sub    rdx,QWORD PTR fs:0x28
0x4011e0 <+106>:  je     0x4011e7
0x4011e2 <+108>:  call   0x401060 <__stack_chk_fail@plt>
0x4011e7 <+113>:  leave
0x4011e8 <+114>:  ret
```

- Canary leak necessary
- Overwrite with correct value possible with leak

Heap Exploitation

- Overflows and other bugs not bound to stack
- Some heap specific bugs exist (e.g. double free)
- General approach
  - Use bug to abuse allocator behavior (metadata corruption)
  - Use bug to corrupt objects on the heap
- glibc-heap exploitation techniques: how2heap [5]

Combining everything



Trace input
Which operations are performed on the input?

Tools

- gdb
  - pwndbg [6]
- python
  - pwntools [7]
- checksec [8]

Exercises

- https://github.com/kitctf/www/tree/master/files/pwn.zip

- http://overthewire.org/wargames/narnia/
- https://picoctf.com/
- https://exploit.education/protostar/
- https://pwnable.kr/
- https://pwnable.tw/

References

[1] `https://github.com/google/google-ctf/tree/master/2021/quals/pwn-fullchain/challenge`.

[2] `https://archive.aachen.ccc.de/33c3ctf.ccc.ac/challenges/index.html`.

[3] `https://github.com/sashs/Ropper`.

[4] `https://github.com/JonathanSalwan/ROPgadget`.

[5] `https://github.com/shellphish/how2heap`.

[6] `https://github.com/pwndbg/pwndbg`.

[7] `https://docs.pwntools.com/en/stable/`.

[8] `https://github.com/slimm609/checksec.sh`.