Binary Exploitation

Julian Gremminger | 11.05.2023

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

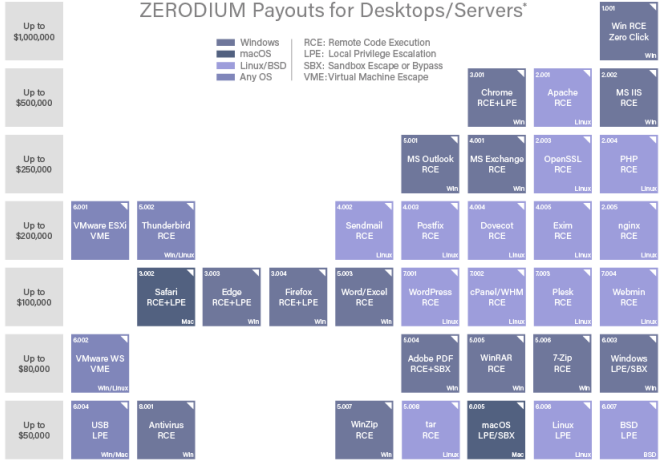
- Often C/C++ binaries written for the competition
- Sometimes real world targets with introduced bugs
  - Chrome: Google CTF 2021 Fullchain [1]
  - Firefox: 33c3 CTF Feuerfuchs [2]
- Objective: Remote Code Execution on challenge server
  - Linux: call system("/bin/sh")

```
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}$
```

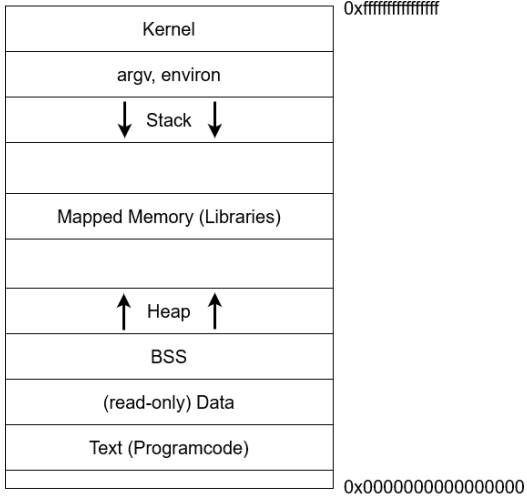
Binary Exploitation in the „Real World“

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

Binary Exploitation in the „Real World“



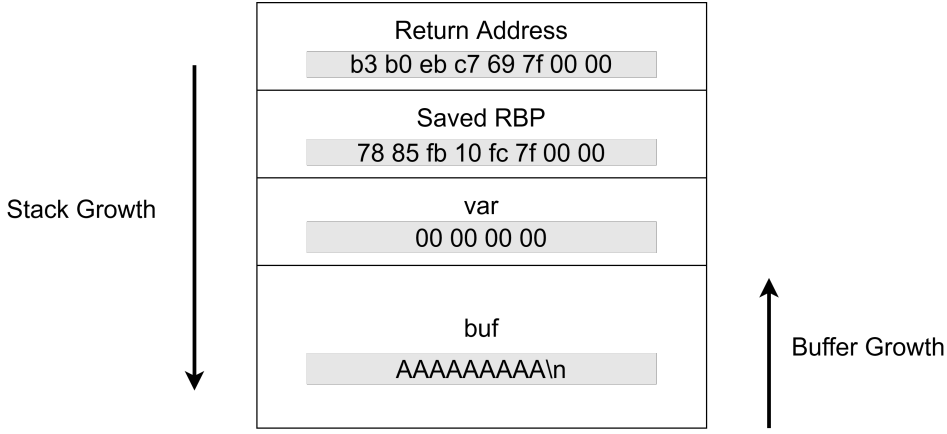
Linux Process Layout



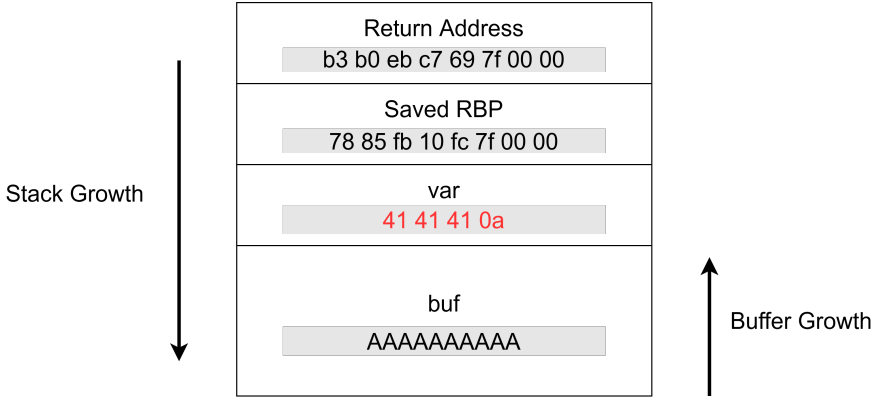
Buffer Overflows

```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;
}
```

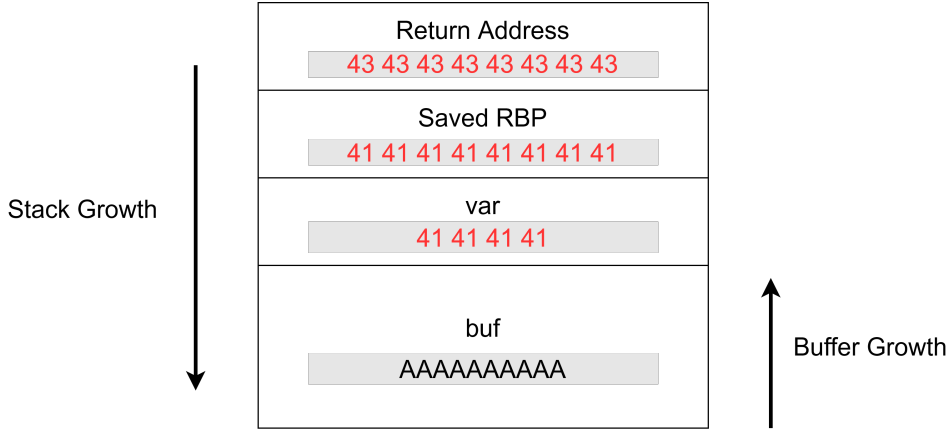
Stack Frames



Overflowing the Buffer



RIP-Control?



- 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

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

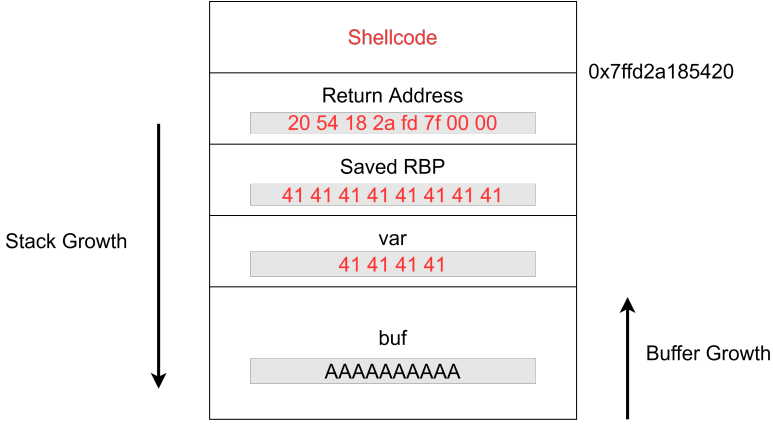
```c
#include <stdio.h>

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

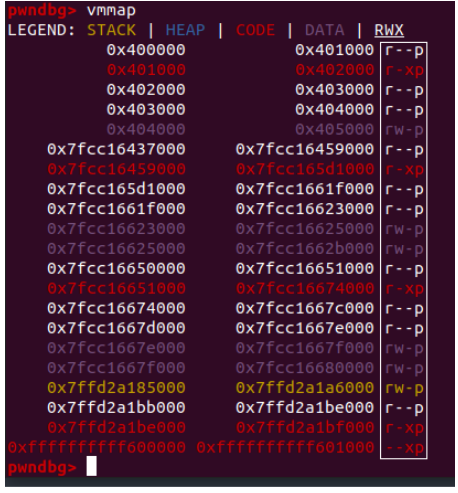
RIP-control to shell

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



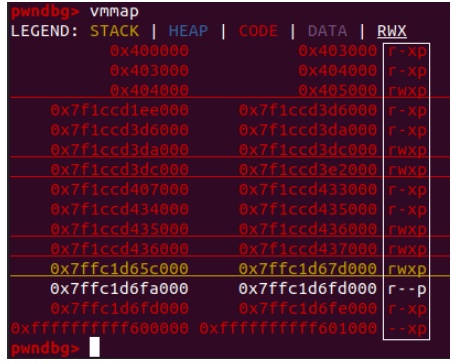
What’s the catch?

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



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

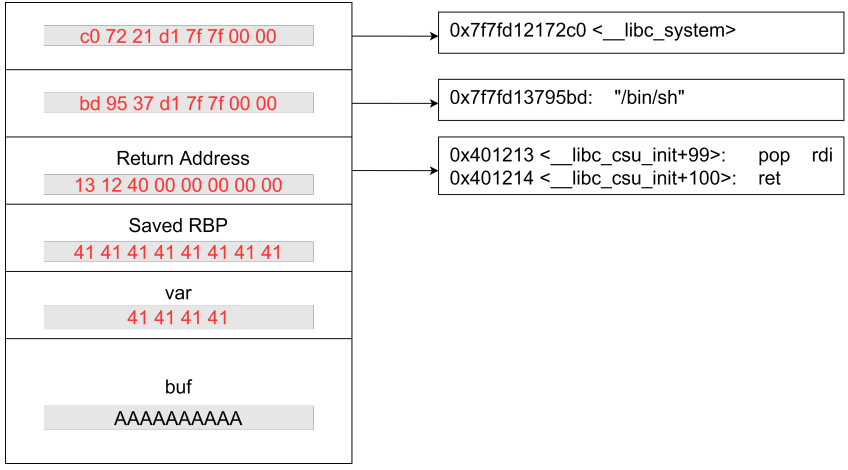


No need for own code[1] (Code Reuse Attacks)

- 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

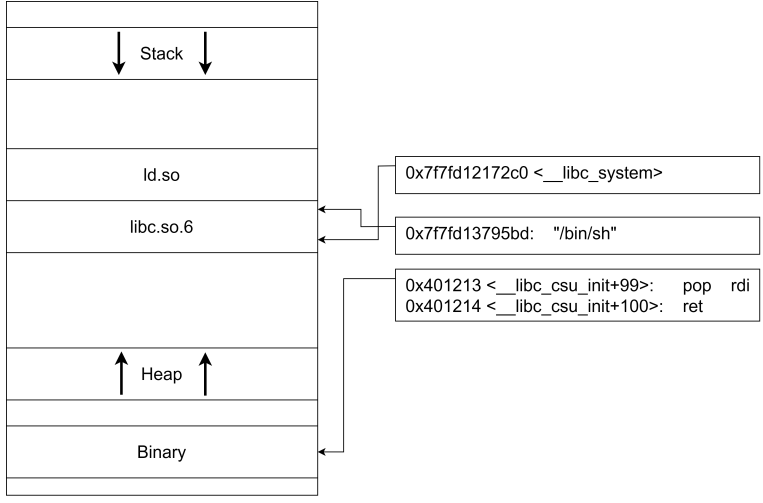
ROP



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

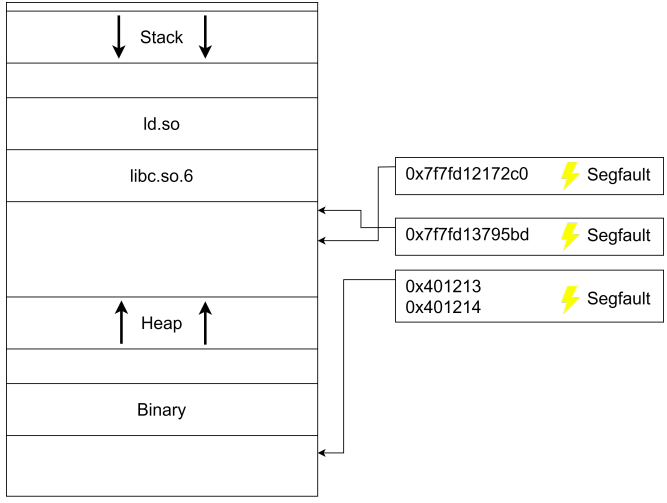
Mitigate Code Reuse Attacks

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



Mitigate Code Reuse Attacks

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



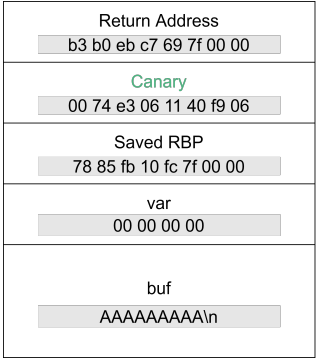
ASLR and PIE

- **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

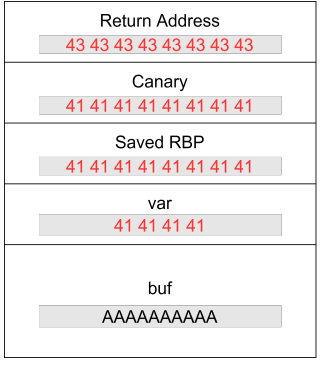
Canaries

- 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




```
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




```
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
```

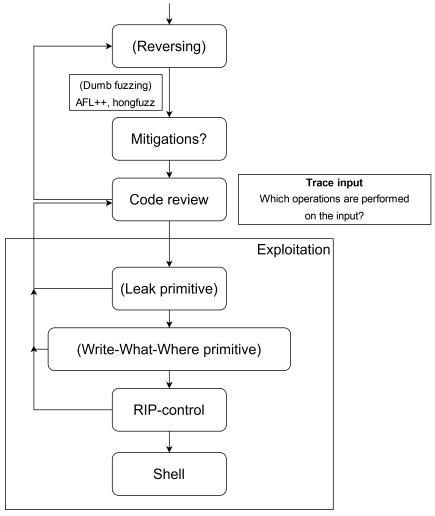
- Canary leak necessary
- Overwrite with correct value possible with leak

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



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.