Introduction to Binary Exploitation

Robert Klink

StudSec | The Student CyberSecurity Association

Who?

- Robert Klink
    - **Rombertus**
- MSc Computer Security @ VU
- Loyal **StudSec** member
    - Got me started in CySec
    - Head of infrastructure
        - ~~blame me for whatever breaks~~

Overview

Stack

Buffer Overflow

- Basics
- Examples

(Bypassing) Protections

- ROP
- ASLR

Final remarks

Overview

Stack

Buffer

- Ba
- Ex

(Bypassing) Protections

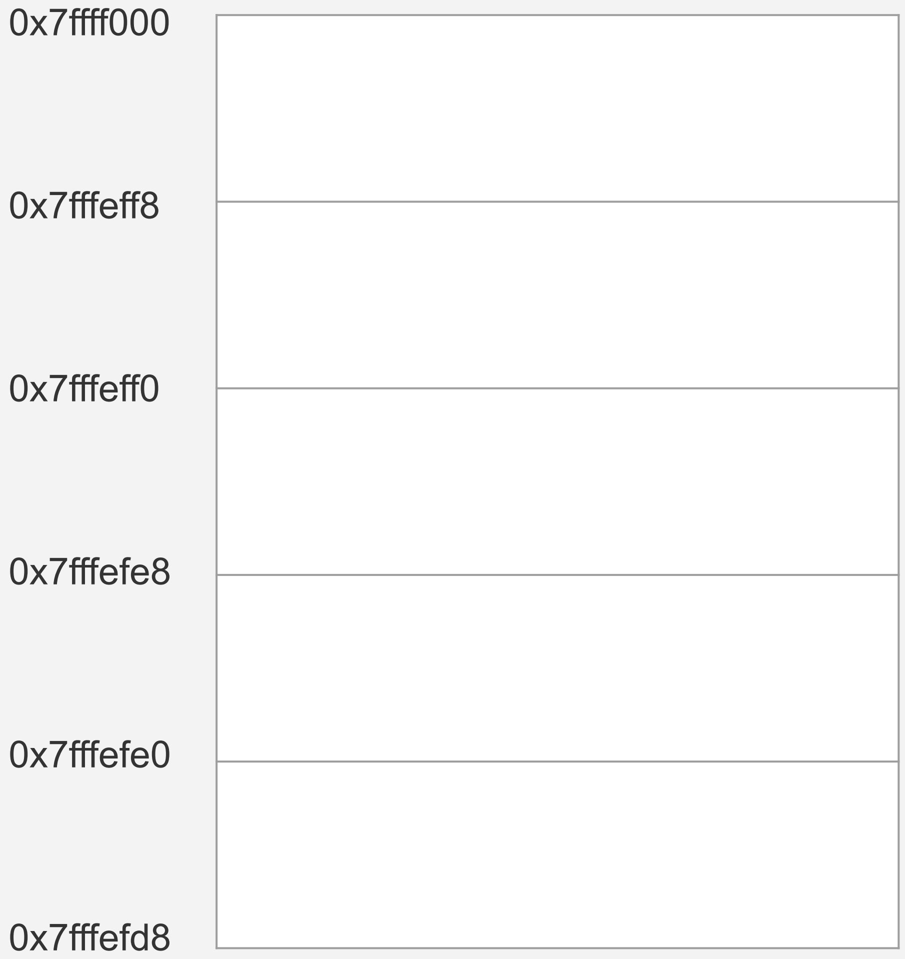
- ROP
- ASLR

Final remarks

Challenges!

Stack

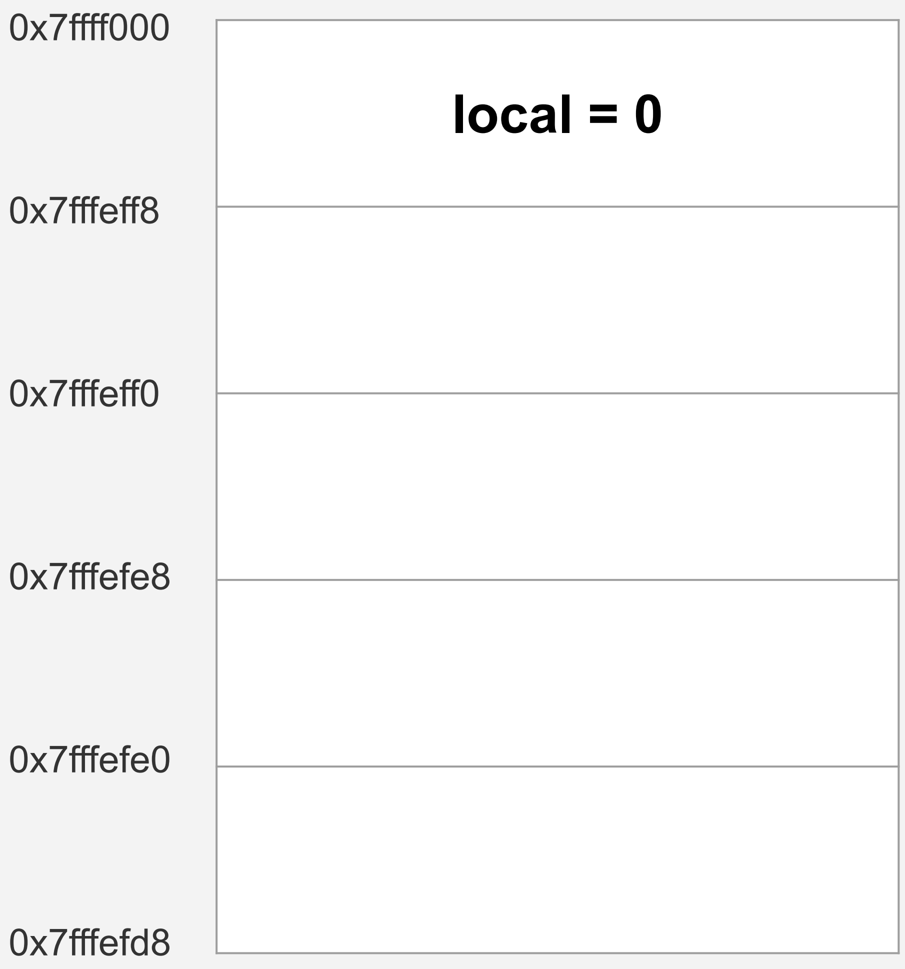
- Memory where execution data is stored
  - Grows from high address to low address
  - 8 byte sized (64 bits) for 64-bit arch



Stack

- Memory where **program data** is stored
  - **Locals**

```
int main(void) {
        long long int local = 0;
        return 0;
}
```


0x7ffff000

local = 0

0x7fffeff8

0x7fffeff0

0x7fffefe8

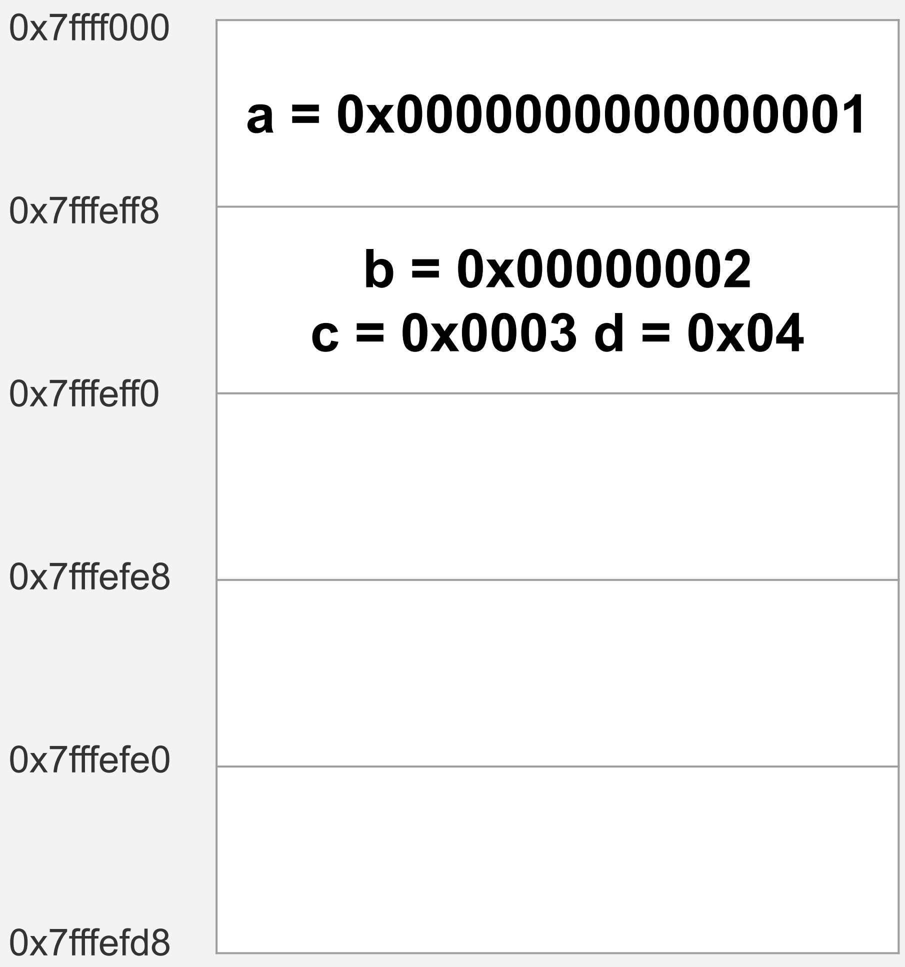
0x7fffefe0

0x7fffefd8

Stack

- Memory where **execution data** is stored
    - Locals
        - **Types**

```
int main(void) {
        long long int a = 1;
        int b = 2;
        short c = 3;
        char d = 4;
        return 0;
}
```



0x7ffff000

a = 0x0000000000000001

0x7fffeff8

b = 0x00000002
c = 0x0003 d = 0x04

0x7fffeff0

0x7fffefe8

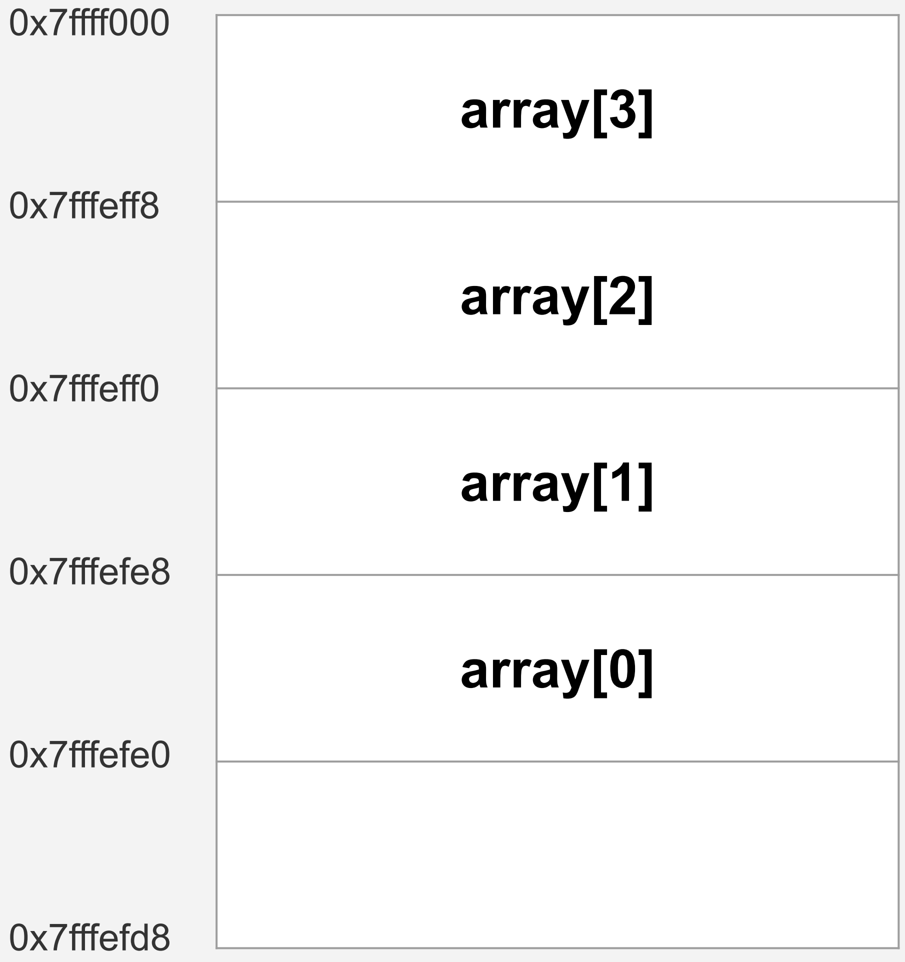
0x7fffefe0

0x7fffefd8

Stack

- Memory where **execution data** is stored
  - Locals
    - **Arrays** too!
    - Grow up

```
int main(void) {
        long long int array[4];
        return 0;
}
```



0x7ffff000

array[3]

0x7fffeff8

array[2]

0x7fffeff0

array[1]

0x7fffefe8

array[0]

0x7fffefe0

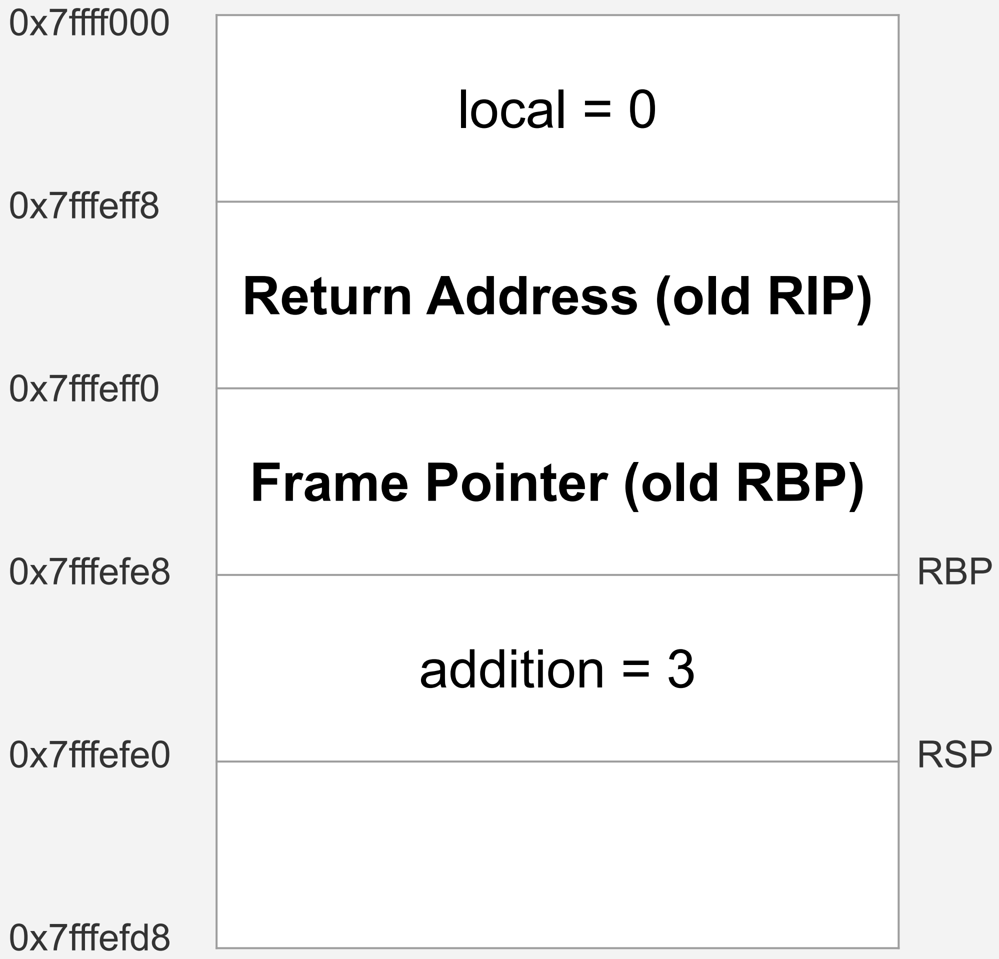
0x7fffefd8

Stack

- Memory where **execution data** is stored
    - Locals
    - **Stack frames**

```
void add(int lhs, int rhs) {
        long long int addition = lhs + rhs;
}

int main(void) {
        long long int local = 0;
        add(1, 2);
}
```



0x7ffff000

local = 0

0x7fffeff8

Return Address (old RIP)

0x7fffeff0

Frame Pointer (old RBP)

0x7fffefe8 RBP

addition = 3

0x7fffefe0 RSP

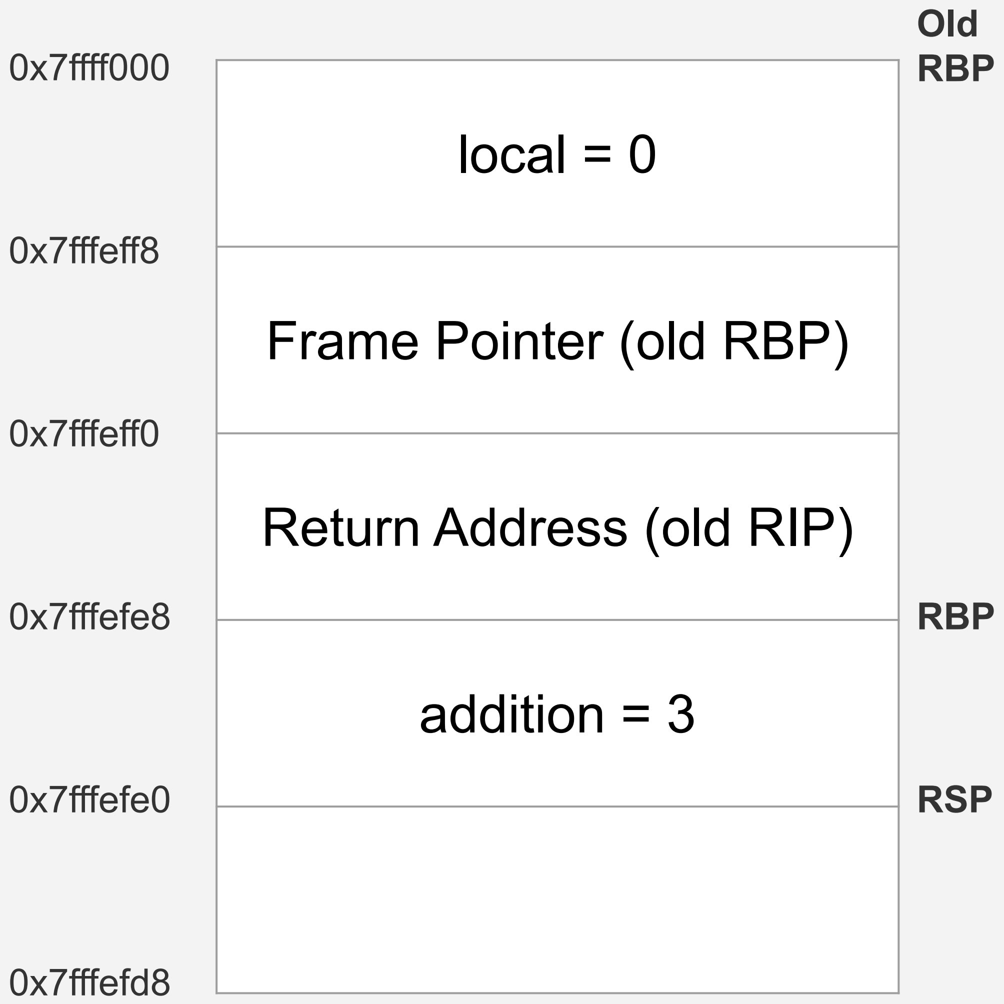
0x7fffefd8

Stack

- Registers?
  - RIP - **Instruction pointer**
  - RBP - **Frame pointer**
  - RSP - **Stack pointer**

```
void add(int lhs, int rhs) {
        long long int addition = lhs + rhs;
}           RIP

int main(void) {
        long long int local = 0;
        add(1, 2);
}           Old RIP
```



Stack

- Registers?
  - ○
  - ○
  - ○

Example 1

```
void add(
        long long int addition
}
```

RIP

```
int main(void) {
        long long int local = 0;
        add(1, 2);
}
```

Old RIP

Old RBP

0x7ffff000

local = 0

0x7fffeff8

RIP)

BP)

RBP

0x7fffefe0

RSP

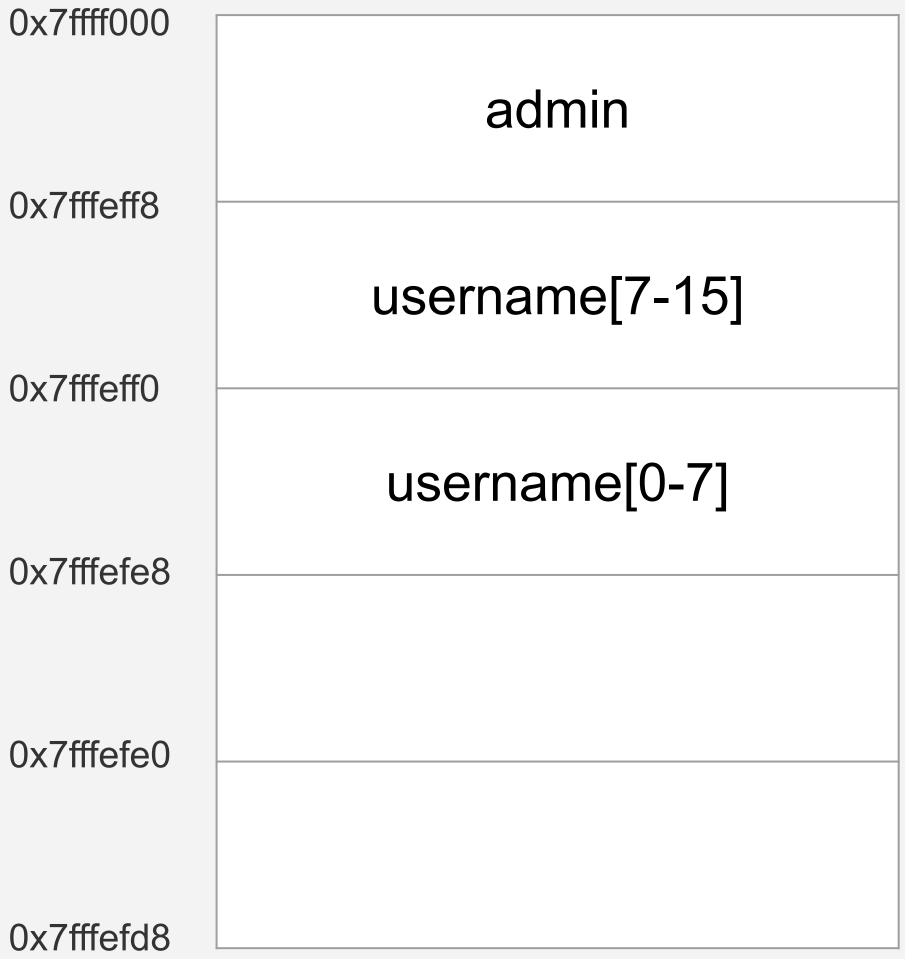
0x7fffefd8

Buffer Overflow

- What happens when a local is read into?
  - gets: read string until **EOF** or **newline**, append **null byte**

```
int main(void) {
        char username[16];
        int admin = 0;
        gets(username);

        if (admin > 0)
                printf(“win!\n”);
}
```



0x7ffff000

admin

0x7fffeff8

username[7-15]

0x7fffeff0

username[0-7]

0x7fffefe8

0x7fffefe0

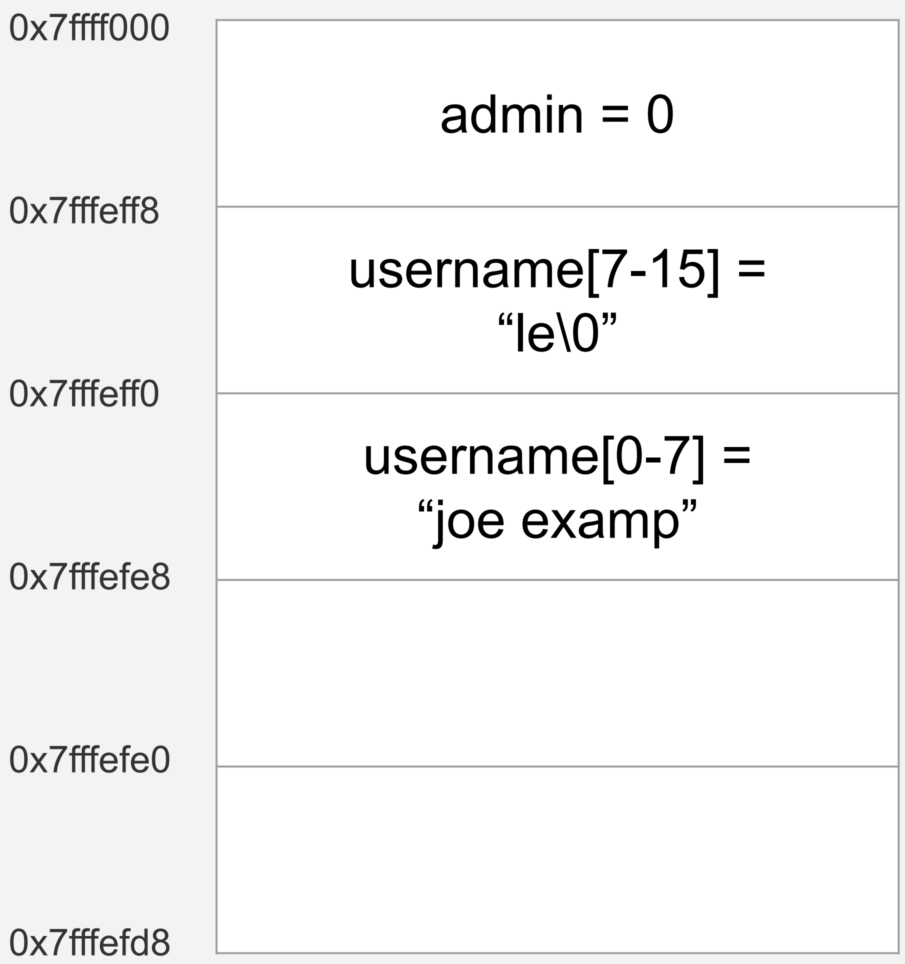
0x7fffefd8

Buffer Overflow

- What happens when a local is read into?
  - **Intended** usage:
    - “joe example”
    - **<** 16

```
int main(void) {
        char username[16];
        int admin = 0;
        gets(username);

        if (admin > 0)
                printf(“win!\n”);
}
```



0x7ffff000

admin = 0

0x7fffeff8

username[7-15] =
“le\0”

0x7fffeff0

username[0-7] =
“joe examp”

0x7fffefe8

0x7fffefe0

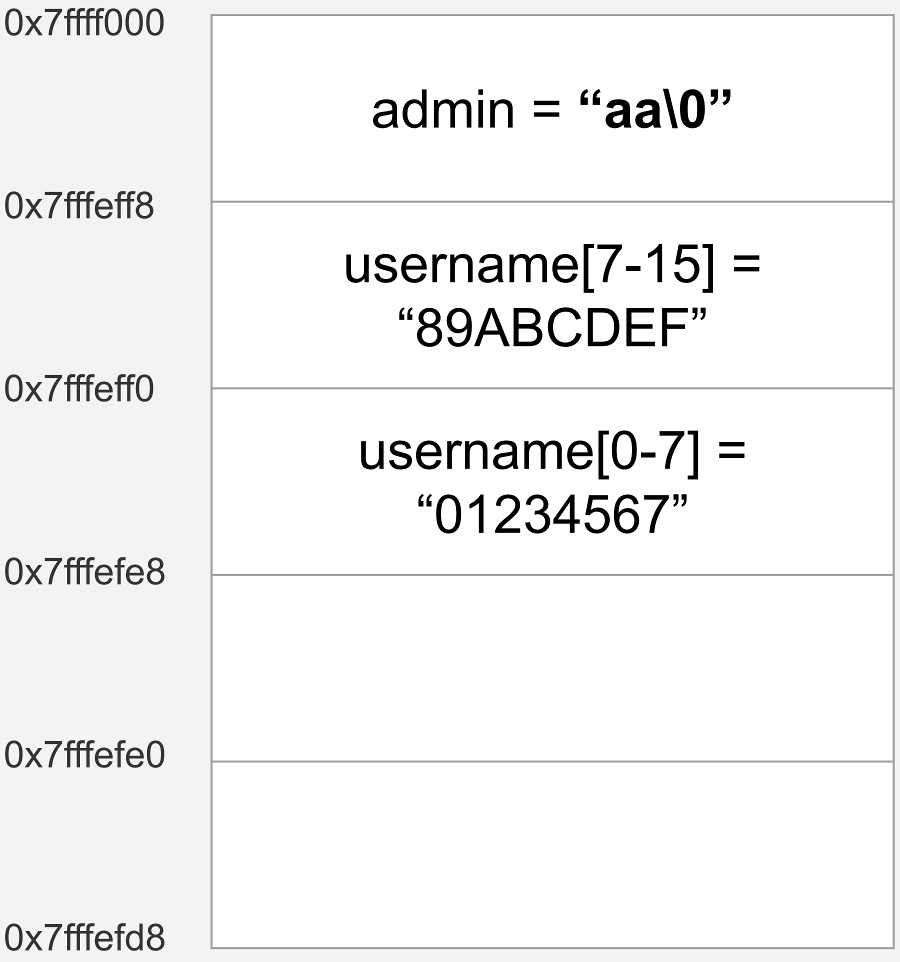
0x7fffefd8

Buffer Overflow

- What happens when a local is read into?
    - **Malicious** usage
        - “0123456789ABCDEF**aa**”
        - input beyond 16 is **overflow**

```
int main(void) {
        char username[16];
        int admin = 0;
        gets(username);

        if (admin > 0)
                printf(“win!\n”);
}
```



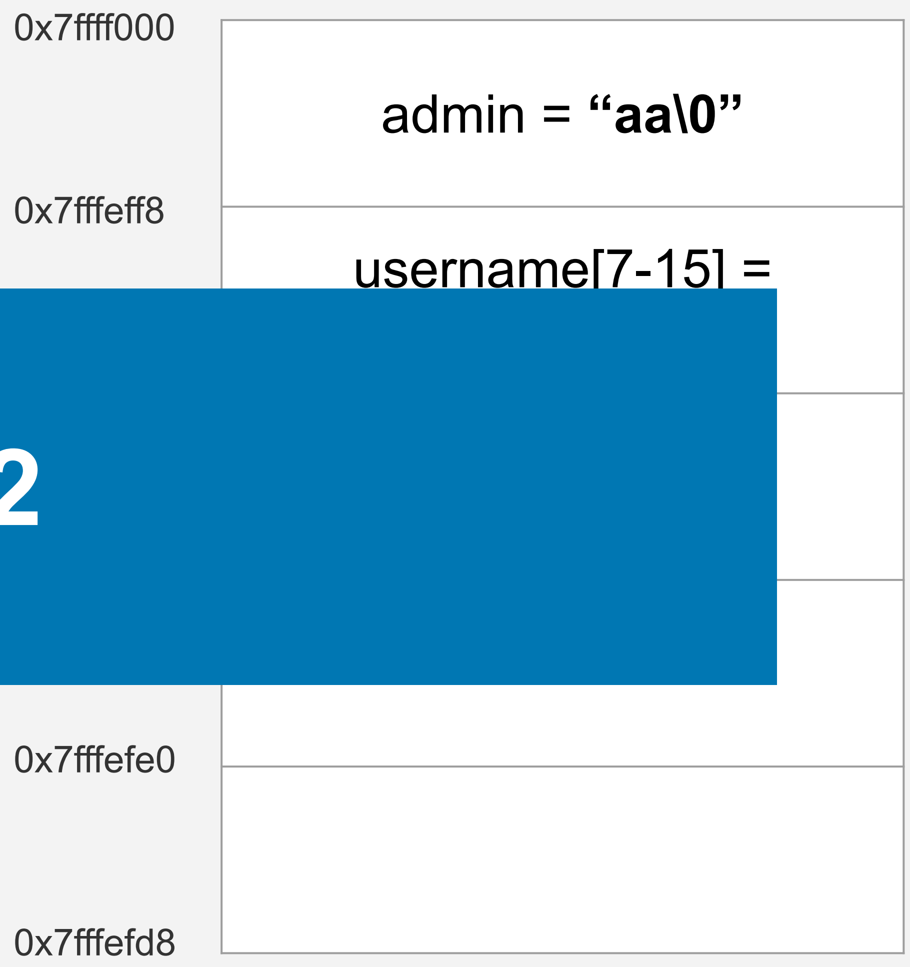
Buffer Overflow

- What happens when a local is read into?
  -

Example 2

```
int main(v
        char username[16];
        int admin = 0;
        gets(username);

        if (admin > 0)
                printf("win!\n");
}
```



Buffer Overflow

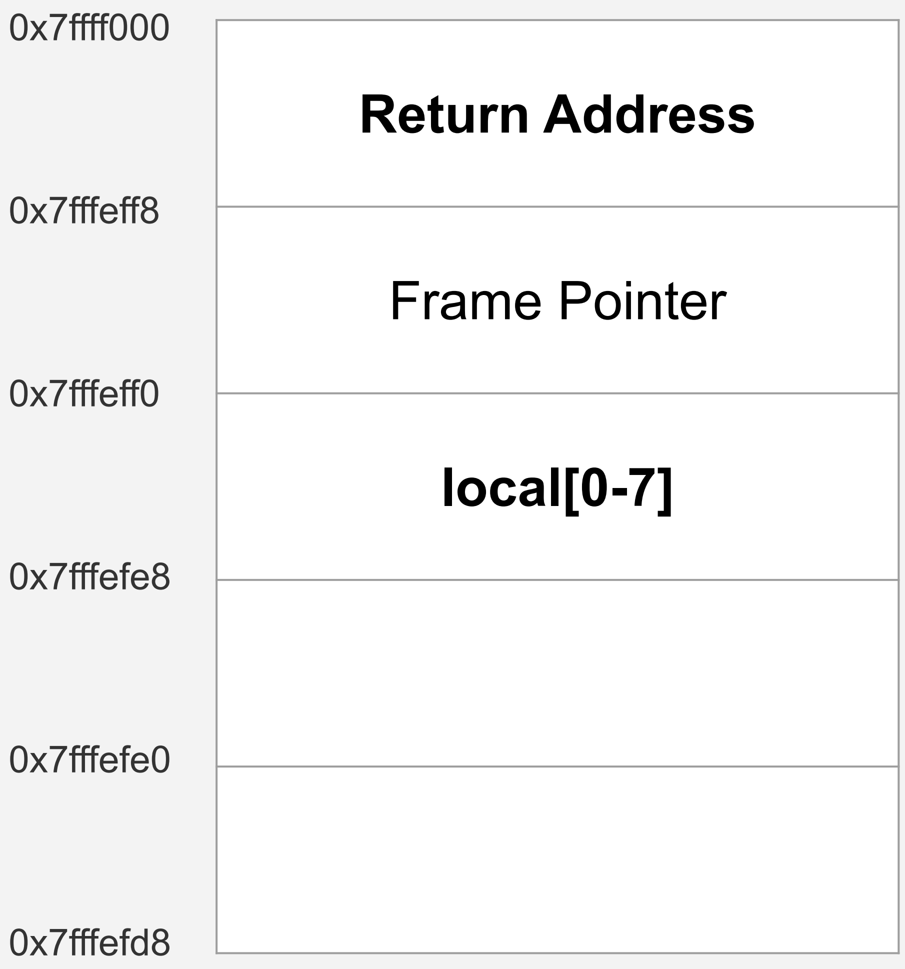
- That's all fun & games, but what if there is no such variable?

Buffer Overflow

- That's all fun & games, but what if there is no such variable?
  - **Return address**
    - Where to get?

```
void win(void) { printf(“win!\n”); }

int main(void) {
        char local[8];
        gets(local);
}
```



0x7ffff000

Return Address

0x7fffeff8

Frame Pointer

0x7fffeff0

local[0-7]

0x7fffefe8

0x7fffefe0

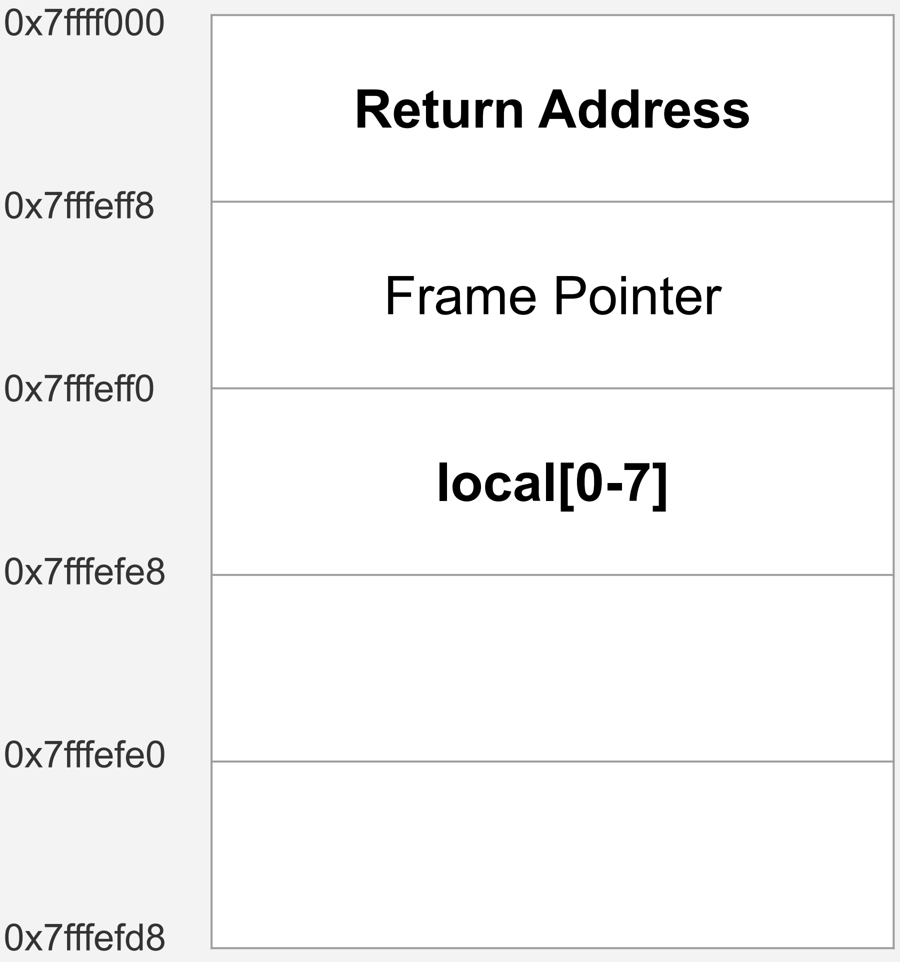
0x7fffefd8

Buffer Overflow

- That's all fun & games, but what if there is no such **variables**?
    - Return address
        - **readelf -s**
        - **pwntools**

```
void win(void) { printf(“win!\n”); }

int main(void) {
        char local[8];
        gets(local);
}
```



0x7ffff000

Return Address

0x7fffeff8

Frame Pointer

0x7fffeff0

local[0-7]

0x7fffefe8

0x7fffefe0

0x7fffefd8

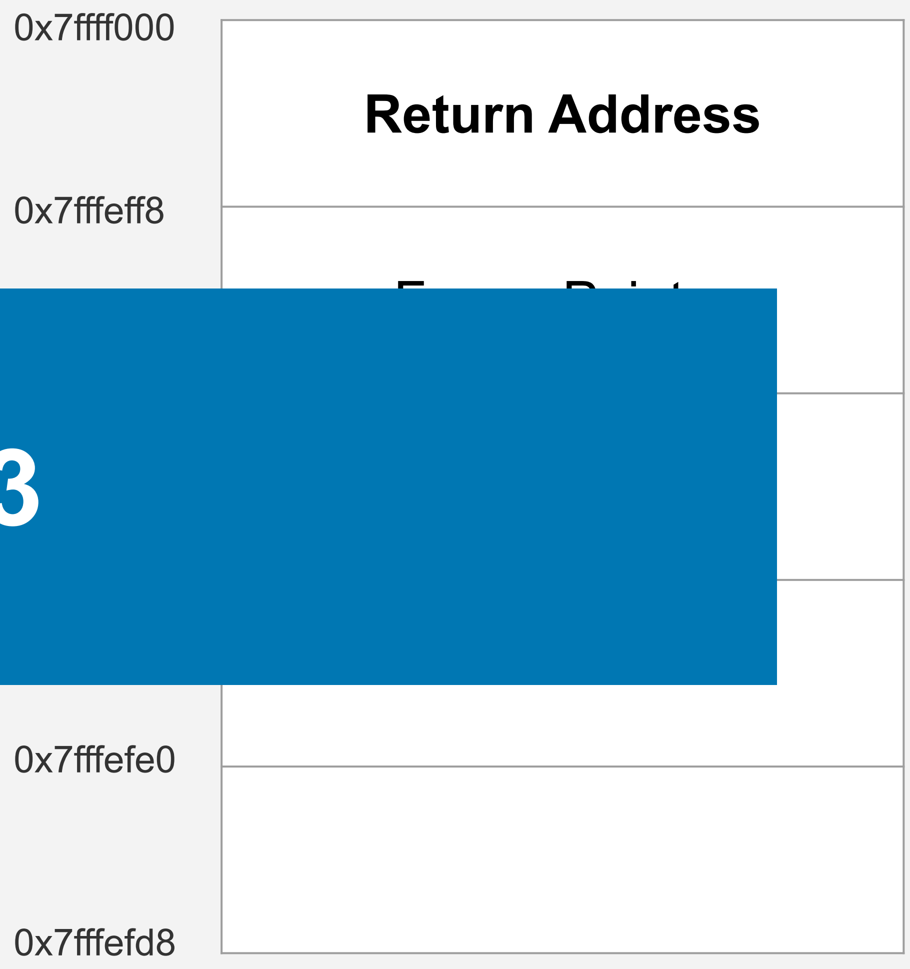
Buffer Overflow

- That's all fun & games, but what if there is no
  - ○

Example 3

```
void win(void) { printf(“win!\n”); }

int main(void) {
        char local[8];
        gets(local);
}
```



Buffer Overflow

- That's all fun & games, but what if there is no such variable and no **functions**?

Buffer Overflow

- That's all fun & games, but what if there is no such variable and no functions?
    - **Shellcode**
        - Write your own asm that calls a **shell**

```
int main(void) {
        char shell[80];
        printf(“%lX\n”, &shell);
        gets(local);
}
```

Buffer Overflow

- That's all fun & games, but what if there is no
    ○

Example 4

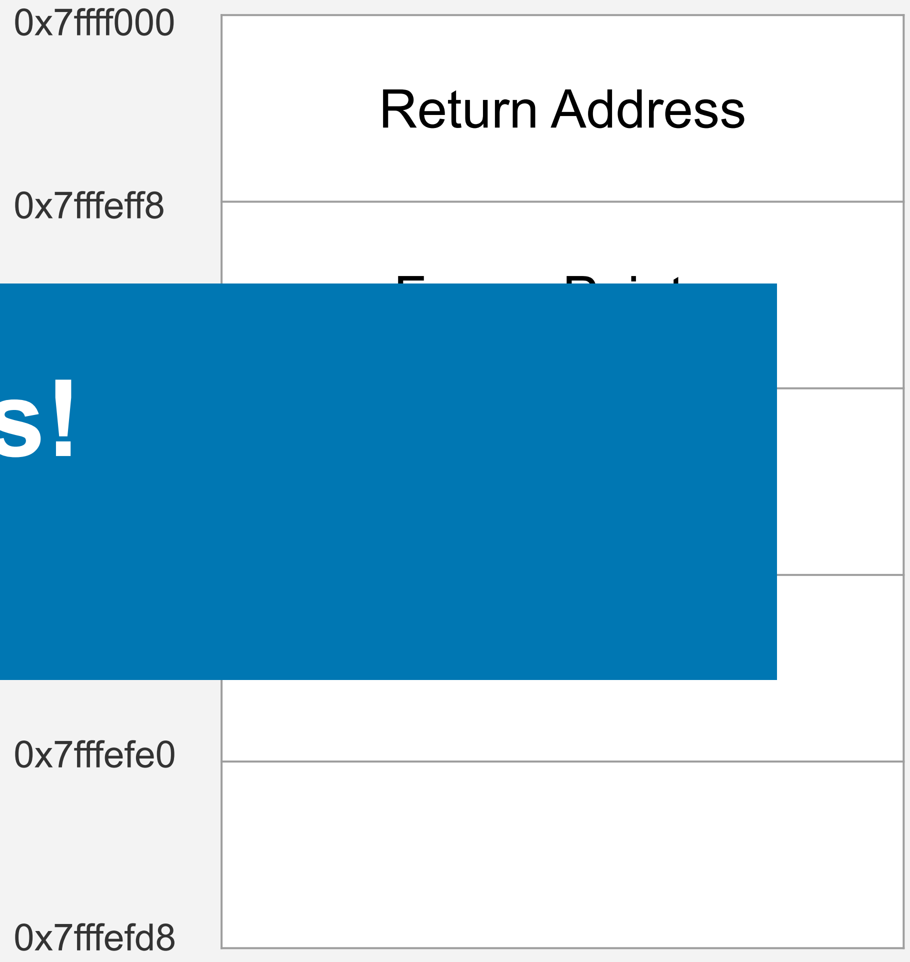
```
int main(void) {
        char shell[80];
        printf(“%lX\n”, &shell);
        gets(local);
}
```

Buffer Overflow

- That's all fun & games, but what if there is no
  - ○

Challenges!
(part 1)

```
int main(void) {
        char local[8];
        gets(local);
}
```



(Bypassing) Protections

- This sounds bad…
  - **Execute** memory
  - **Static** function locations

(Bypassing) Protections

- This ~~sounds~~ is bad!
- **NX / DEP**
  - **Executable** (but unwritable) or **writable** (but unexecutable)
  - No more shellcode

(Bypassing) Protections

- This ~~sounds~~ is bad!
- NX / DEP
  - Executable (but unwritable) or writable (but unexecutable)
  - No more shellcode
- **PIE / ASLR**
  - Randomize where **stack, heap, program** is
  - No more statically known addresses

(Bypassing) Protections

- This ~~sounds~~ is bad!
- NX / DEP
  - Executable (but unwritable) or writable (but unexecutable)
  - No more shellcode
- PIE / ASLR
  - Randomize where **stack, heap, program** is
  - No more statically known addresses
- **Stack Canary**
  - Entry on the stack between **locals** and **stack frame data** that **halts** program if changed
  - Harder to overwrite **return address**
- **Defaults!**

(Bypassing) Protections

- NX / DEP
  - No more shellcode…
  - …**written by us!**
  - Program has plenty of instructions **available**

(Bypassing) Protections

- NX / DEP
  - No more shellcode…
  - …written by us!
  - Program has plenty of instructions available
    - Chain instructions with **returns**
    - **Return Oriented Programming** (ROP)

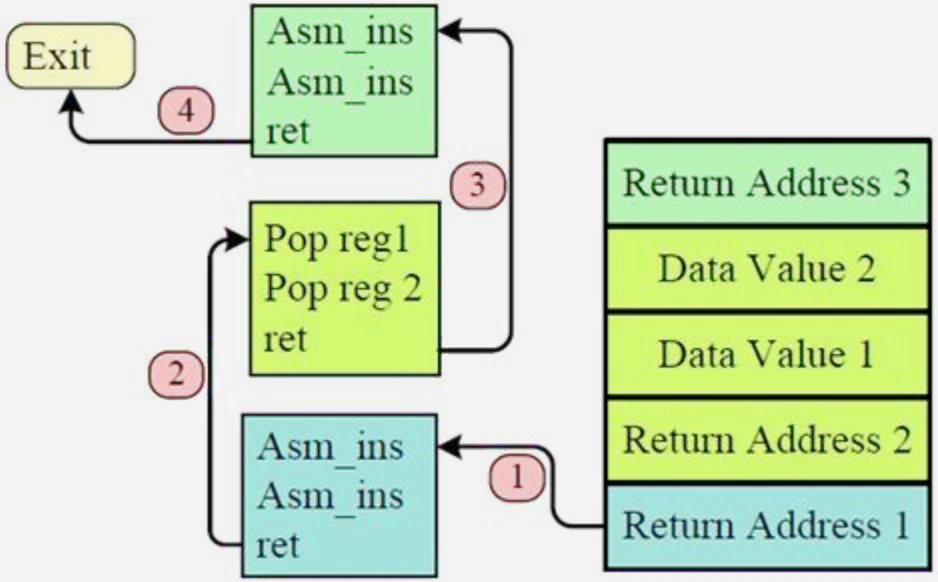
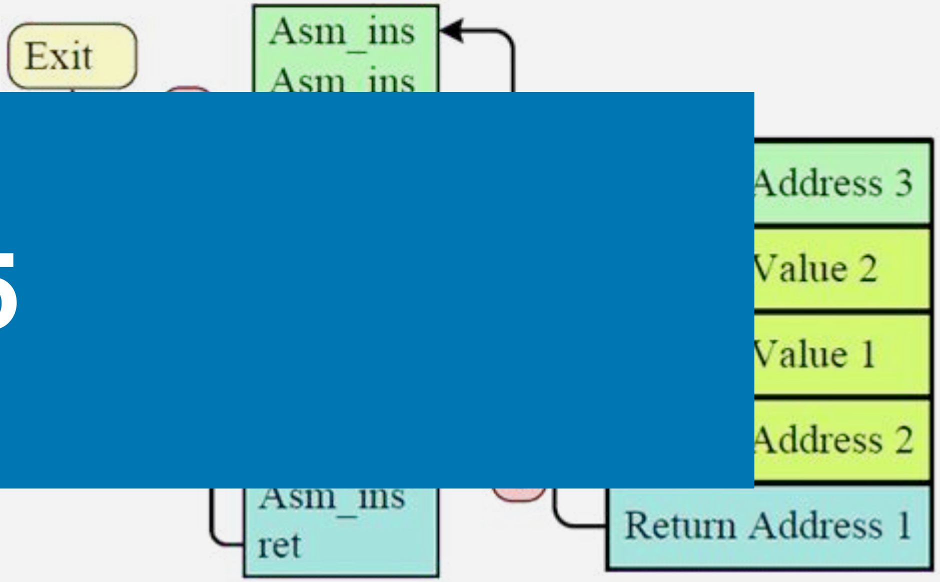


Image source

(Bypassing) Protections

- NX / DEP



Example 5

Image source

(Bypassing) Protections

- PIE / ASLR
  - No more statically known addresses…
  - …but **offsets** between functions are the same!
  - Just need to **leak one function**

(Bypassing) Protections

- PIE / ASLR
  - No more statically known addresses…
  - …but offsets between functions are the same!
  - Just need to leak one function
    - **Format string** attack
      - PIE

(Bypassing) Protections

- PIE / ASLR

Example 6

(Bypassing) Protections

- PIE / ASLR
    - No more statically known addresses…
    - …but offsets between functions are the same!
    - ~~Just need to leak one function~~
        - ~~Format string attack~~
            - ~~PIE~~
    - No such **function in binary**?
        - Return to **libc**

(Bypassing) Protections

- PIE / ASLR
  - No more statically known addresses…
  - …but offsets between functions are the same!
  - No such function in binary?
    - Return to **libc**
      - ASLR…
    - Return to **PLT**

(Bypassing) Protections

- PIE / ASLR

Example 7

    - Return to **PLT**

(Bypassing) Protections

- **Stack Canary**
    - Harder to overwrite return address…
    - …but not **impossible**!
    - **Need** to **somehow** keep the canary the **same**

(Bypassing) Protections

- **Stack Canary**
    - Harder to overwrite return address…
    - …but not impossible!
    - Need to somehow keep the canary the same
    - **Leak** the canary
        - **Format string** attack or **overread**
    - **Write directly** to return address
        - Control over **index**
    - **Bruteforce**
        - …but only in **specific** cases (e.g. **fork**)

Final Remarks

- Use **manpages** or **cppreference** (for c too)
  - Usually lists if function is “unsafe”
- Use **pwntools** and **pwndbg**
  - Good for exploiting and investigating respectively
- Lots of useful online sources
  - ir0nstone notes
  - CTF 101
- Practice makes **perfect**
  - pwn.college
  - studsec ;)

Final Remarks

Understand **C** and **Assembly**…

Challenges!
(part 2)

Feedback:

https://url.studsec.nl/feedback

https://ctf.studsec.nl/intro

Become a member

https://studsec.nl/signup