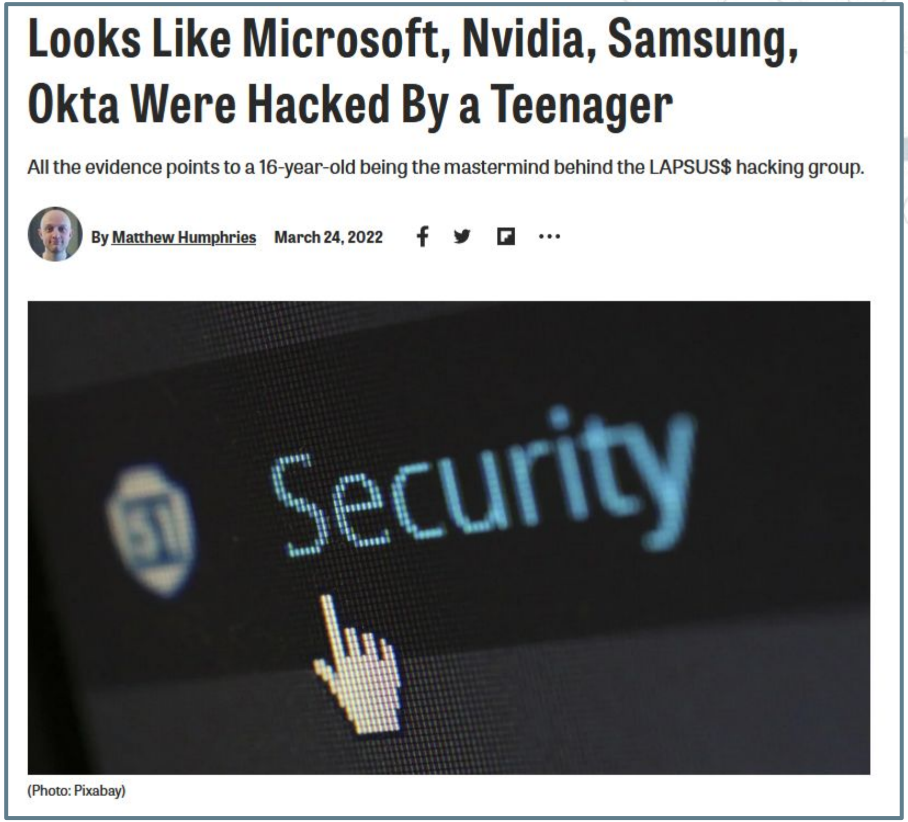
Patch Notes

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.

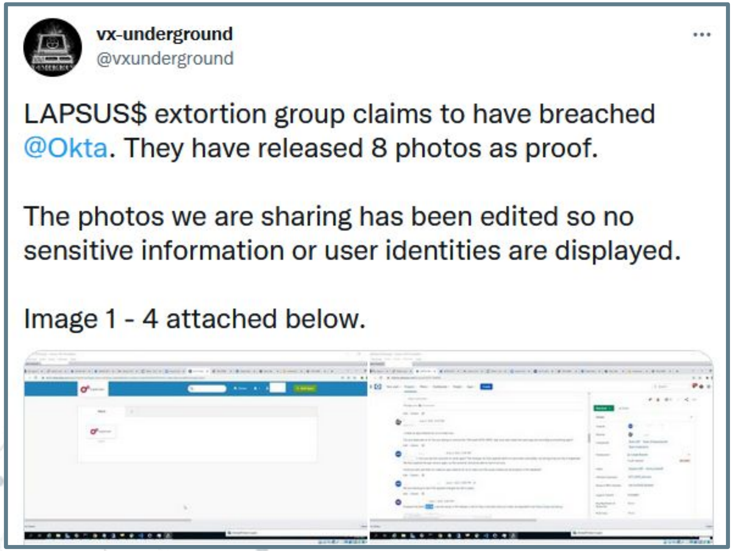
By Matthew Humphries March 24, 2022

(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



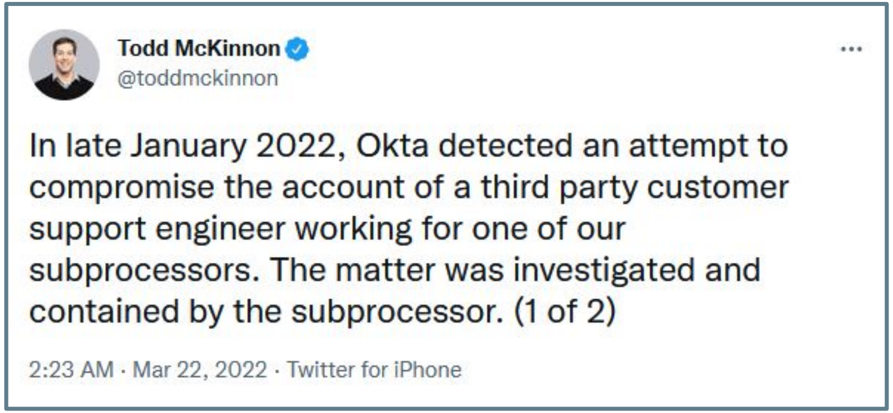
vx-underground
@vxunderground

LAPSUS$ extortion group claims to have breached @Okta. They have released 8 photos as proof.

The photos we are sharing has been edited so no sensitive information or user identities are displayed.

Image 1 - 4 attached below.

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



Todd McKinnon
@toddmckinnon

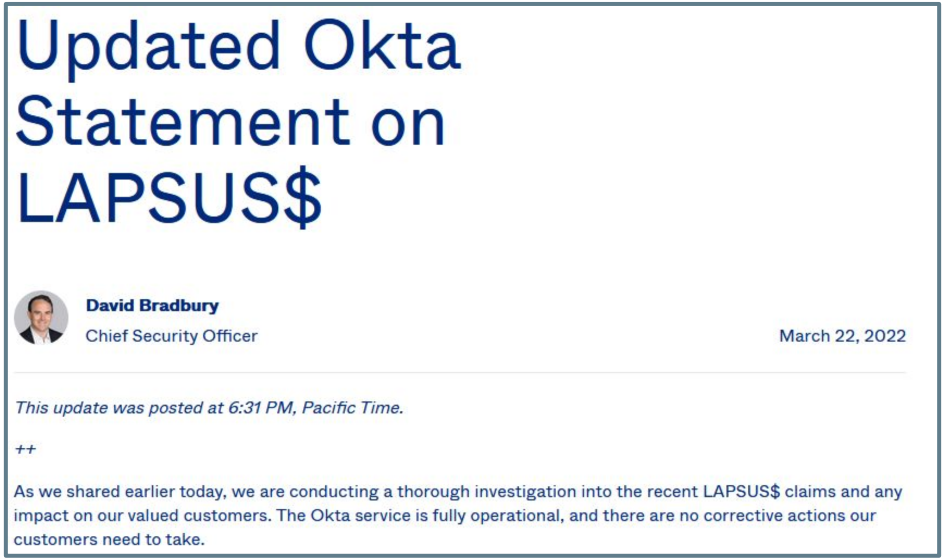
In late January 2022, Okta detected an attempt to compromise the account of a third party customer support engineer working for one of our subprocessors. The matter was investigated and contained by the subprocessor. (1 of 2)

2:23 AM · Mar 22, 2022 · Twitter for iPhone

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.

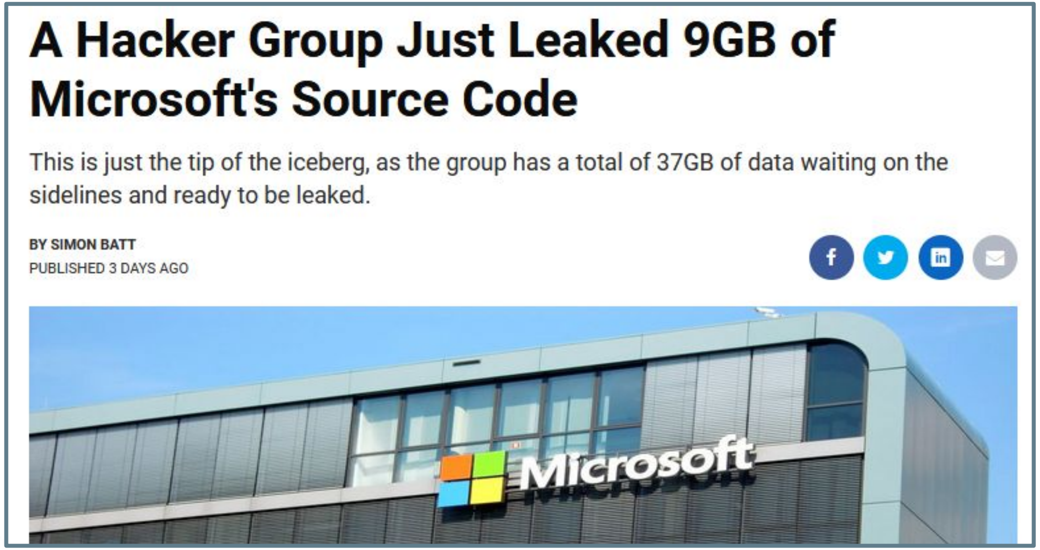
++

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



A Hacker Group Just Leaked 9GB of Microsoft's Source Code

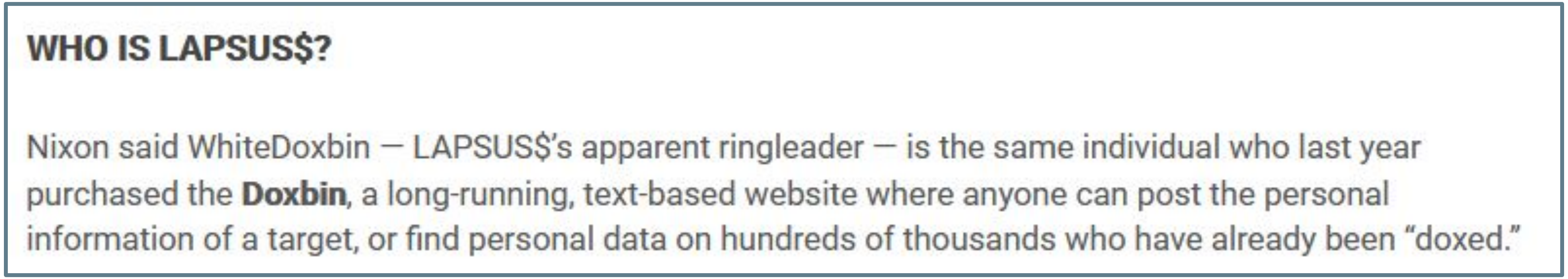
This is just the tip of the iceberg, as the group has a total of 37GB of data waiting on the sidelines and ready to be leaked.

BY SIMON BATT
PUBLISHED 3 DAYS AGO

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



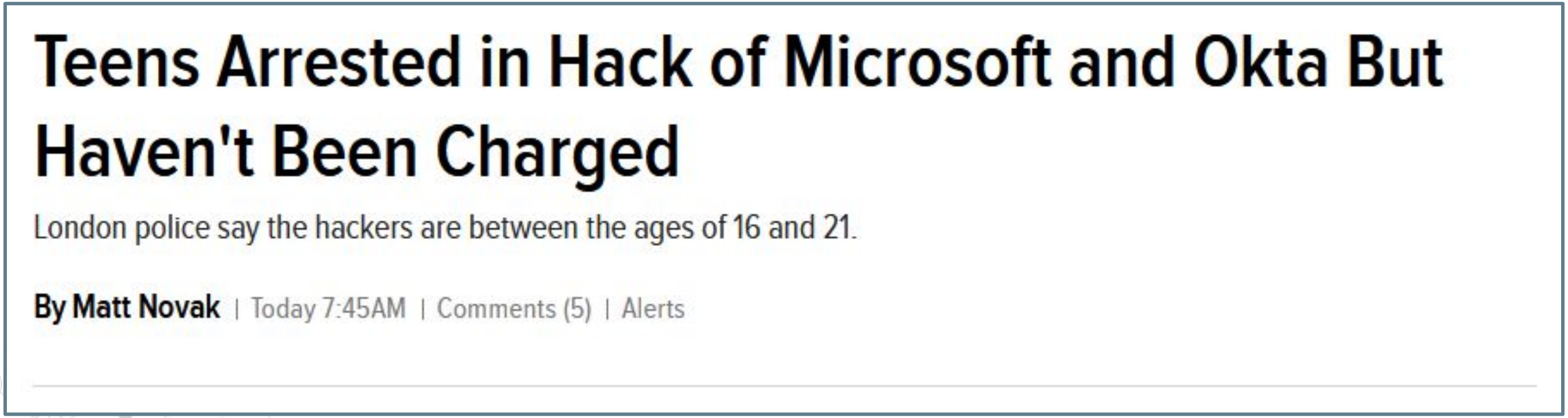
WHO IS LAPSUS$?

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.

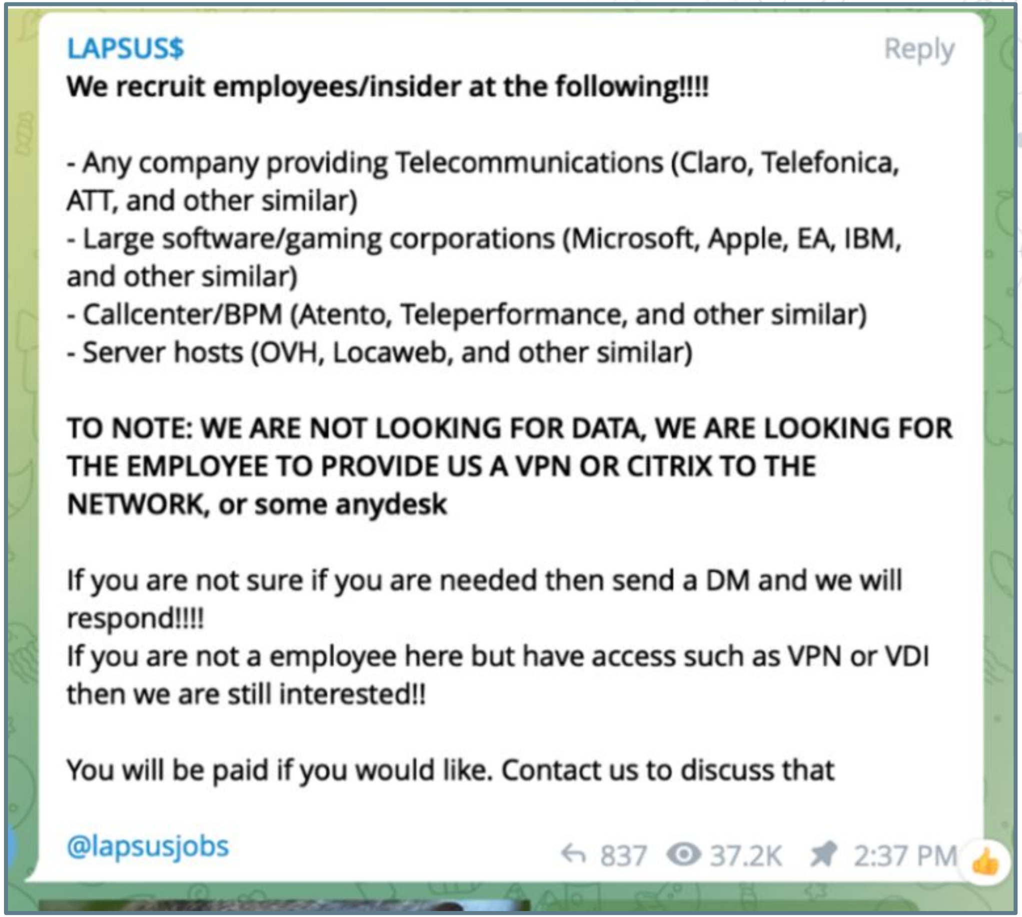
By Matt Novak | Today 7:45AM | Comments (5) | Alerts

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



LAPSUS$

We recruit employees/insider at the following!!!!

- Any company providing Telecommunications (Claro, Telefonica, ATT, and other similar)
- Large software/gaming corporations (Microsoft, Apple, EA, IBM, and other similar)
- Callcenter/BPM (Atento, Teleperformance, and other similar)
- Server hosts (OVH, Locaweb, and other similar)

TO NOTE: WE ARE NOT LOOKING FOR DATA, WE ARE LOOKING FOR THE EMPLOYEE TO PROVIDE US A VPN OR CITRIX TO THE NETWORK, or some anydesk

If you are not sure if you are needed then send a DM and we will respond!!!!
If you are not a employee here but have access such as VPN or VDI then we are still interested!!

You will be paid if you would like. Contact us to discuss that

@lapsusjobs

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

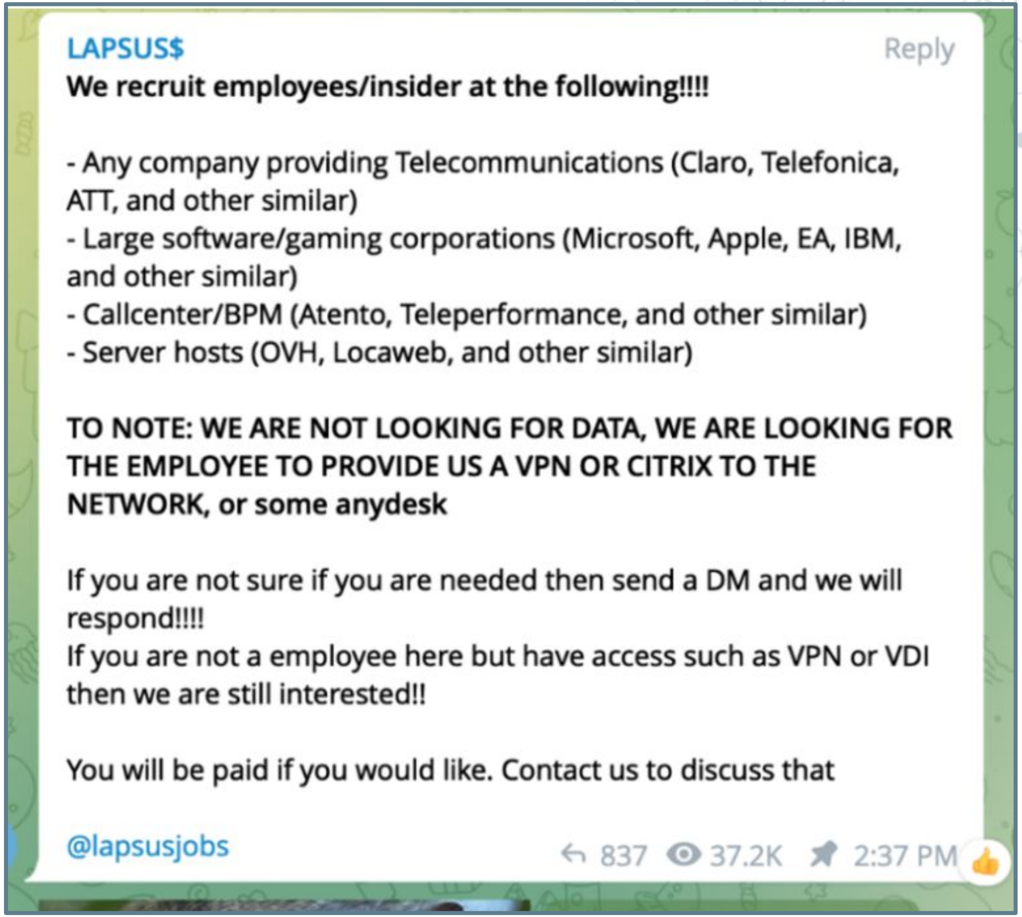
Patch Notes

How did they do it?

- Social engineering
- Paying insiders
- SIM swapping

Takeaway:

- Humans are always the weakest links



LAPSUS$

We recruit employees/insider at the following!!!!

- Any company providing Telecommunications (Claro, Telefonica, ATT, and other similar)
- Large software/gaming corporations (Microsoft, Apple, EA, IBM, and other similar)
- Callcenter/BPM (Atento, Teleperformance, and other similar)
- Server hosts (OVH, Locaweb, and other similar)

TO NOTE: WE ARE NOT LOOKING FOR DATA, WE ARE LOOKING FOR THE EMPLOYEE TO PROVIDE US A VPN OR CITRIX TO THE NETWORK, or some anydesk

If you are not sure if you are needed then send a DM and we will respond!!!!

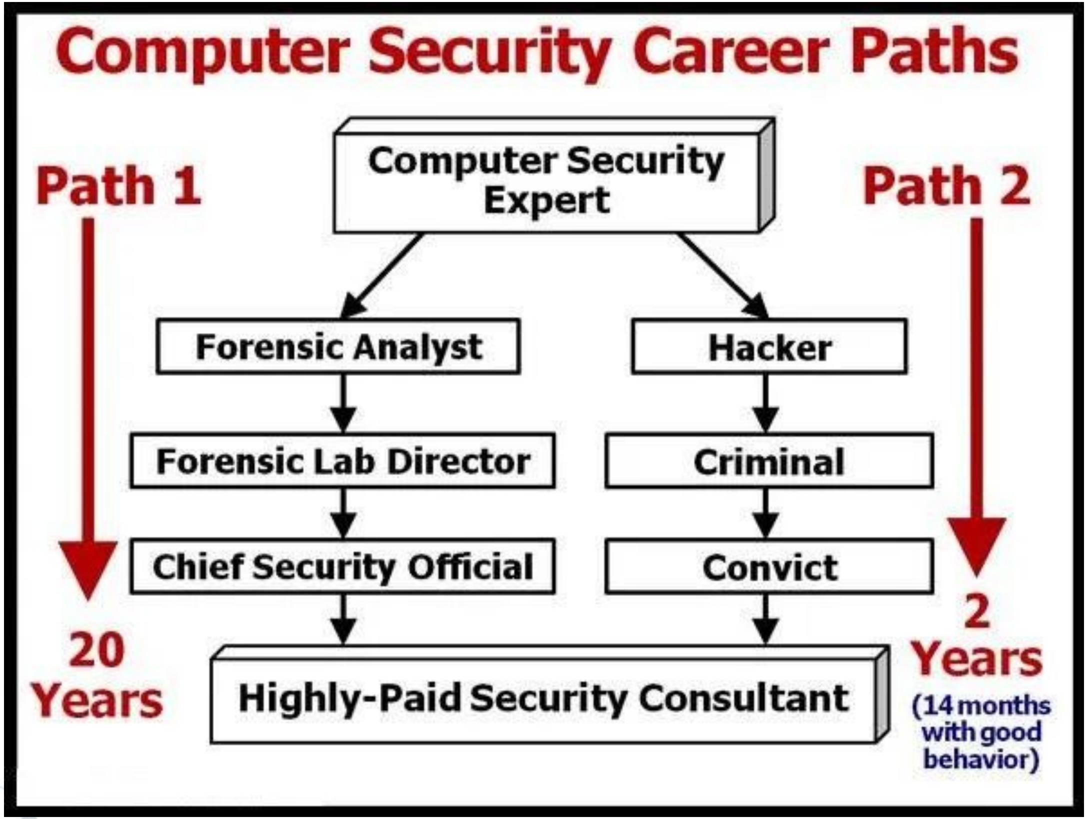
If you are not a employee here but have access such as VPN or VDI then we are still interested!!

You will be paid if you would like. Contact us to discuss that

@lapsusjobs

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

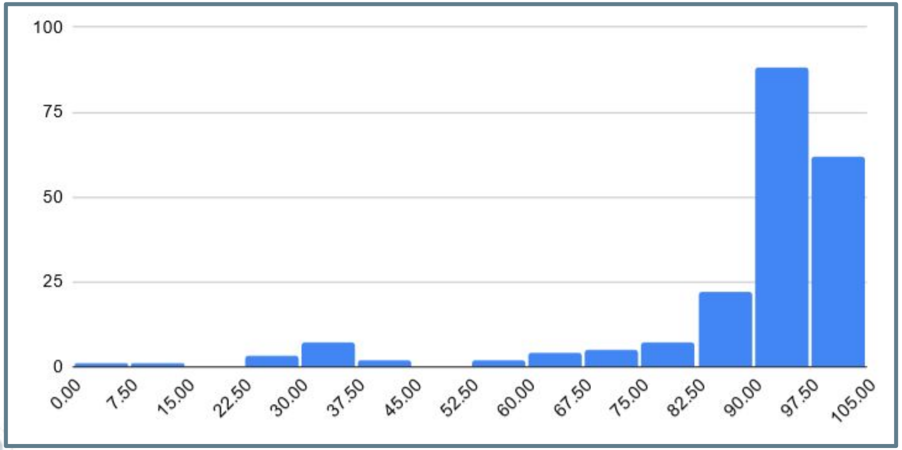
Patch Notes


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:

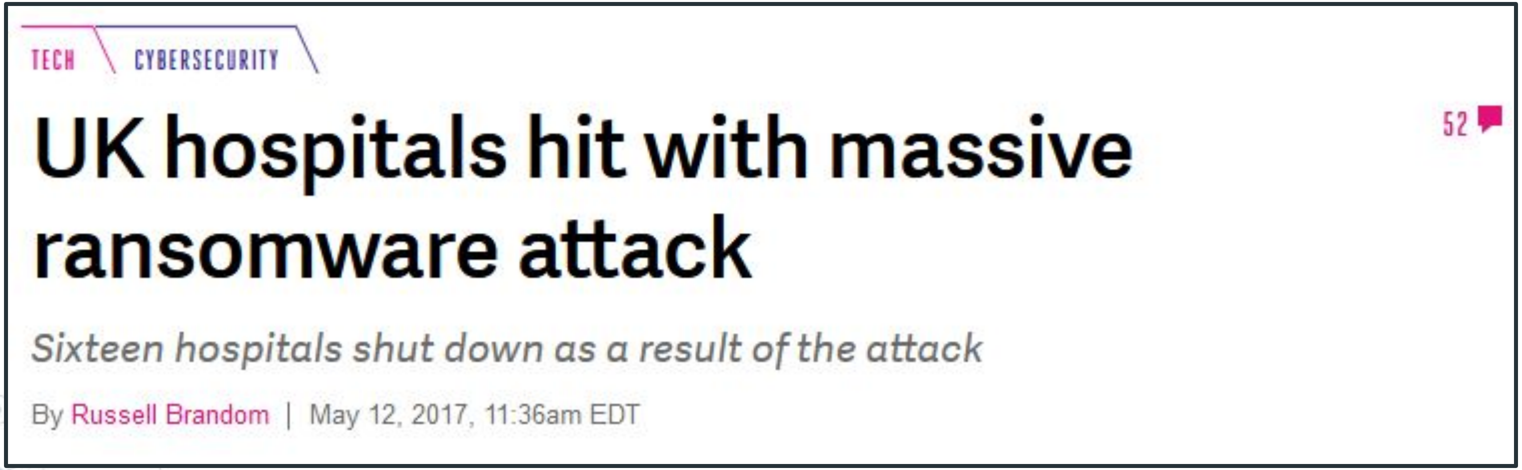
- Next two weeks: Program security
- Final two weeks: Miscellaneous

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

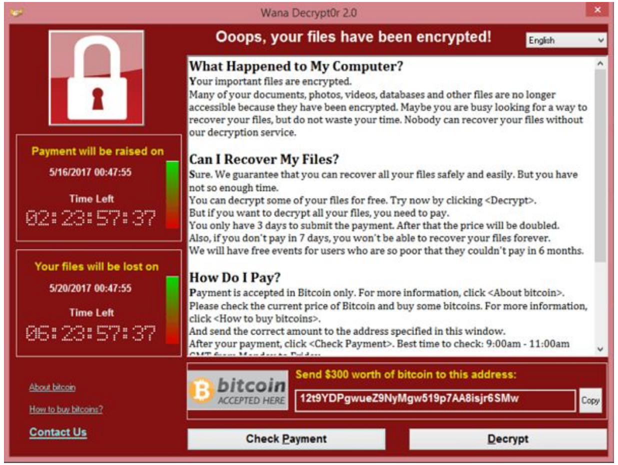
Sixteen hospitals shut down as a result of the attack

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



Ooops, your files have been encrypted!

Payment will be raised on
5/16/2017 00:47:55
Time Left
02:23:57:37

Your files will be lost on
5/20/2017 00:47:55
Time Left
06:23:57:37

What Happened to My Computer?

Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?

Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

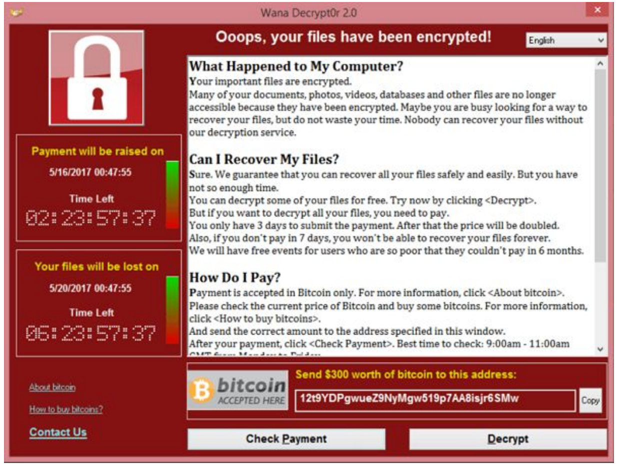
Send $300 worth of bitcoin to this address:
12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw

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**



Ooops, your files have been encrypted!

Payment will be raised on
5/16/2017 00:47:55
Time Left
02:23:57:37

Your files will be lost on
5/20/2017 00:47:55
Time Left
06:23:57:37

What Happened to My Computer?

Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?

Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Send $300 worth of bitcoin to this address:

12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw

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

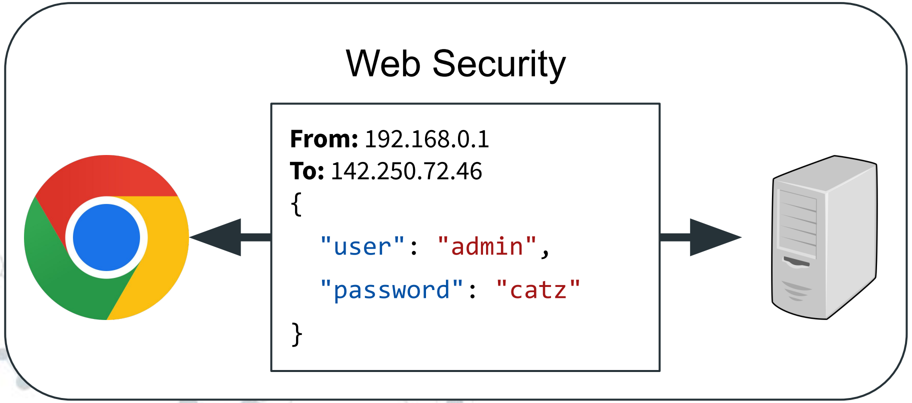
- We will focus on compiled binary programs such as web browsers, operating systems, etc

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

Application Security

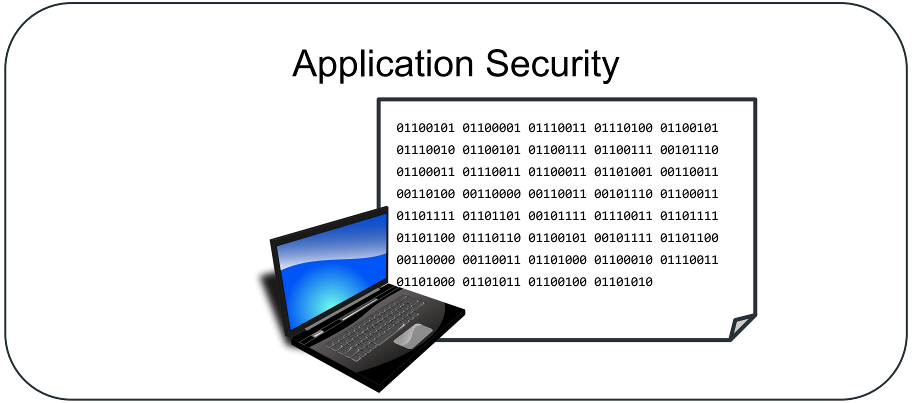
Major Differences with Web Security

👍Less focus on networks



Web Security

```
From: 192.168.0.1
To: 142.250.72.46
{
  "user": "admin",
  "password": "catz"
}
```



Application Security

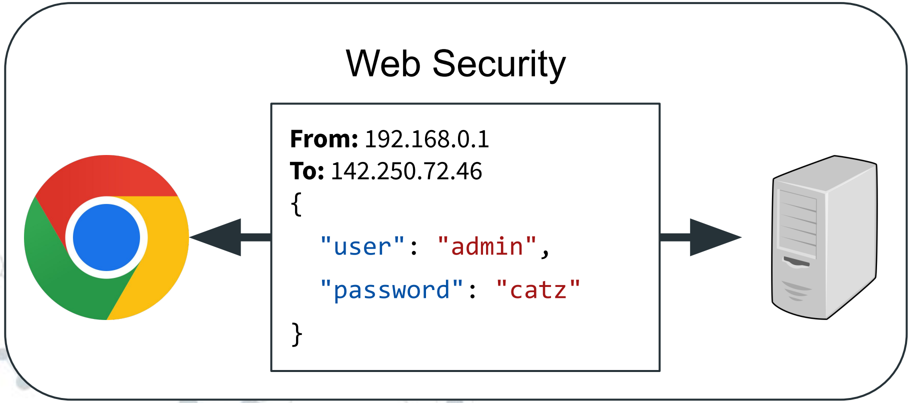
```
01100101 01100001 01110011 01110100 01100101
01110010 01100101 01100111 01100111 00101110
01100011 01110011 01100011 01101001 00110011
00110100 00110000 00110011 00101110 01100011
01101111 01101101 00101111 01110011 01101111
01101100 01110110 01100101 00101111 01101100
00110000 00110011 01101000 01100010 01110011
01101000 01101011 01100100 01101010
```

Application Security

Major Differences with Web Security

👍 Less focus on networks

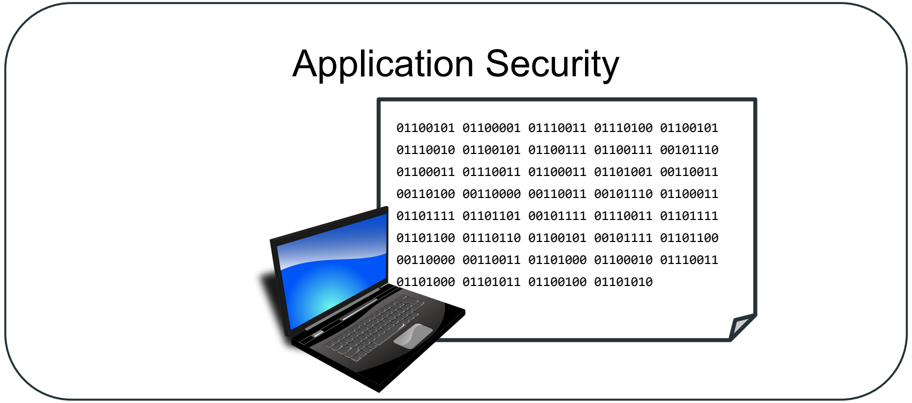
👎 No sandboxing: The entire computer is vulnerable



Web Security

From: 192.168.0.1
To: 142.250.72.46

```
{
  "user": "admin",
  "password": "catz"
}
```



Application Security

```
01100101 01100001 01110011 01110100 01100101
01110010 01100101 01100111 01100111 00101110
01100011 01110011 01100011 01101001 00110011
00110100 00110000 00110011 00101110 01100011
01101111 01101101 00101111 01110011 01101111
01101100 01110110 01100101 00101111 01101100
00110000 00110011 01101000 01100010 01110011
01101000 01101011 01100100 01101010
```

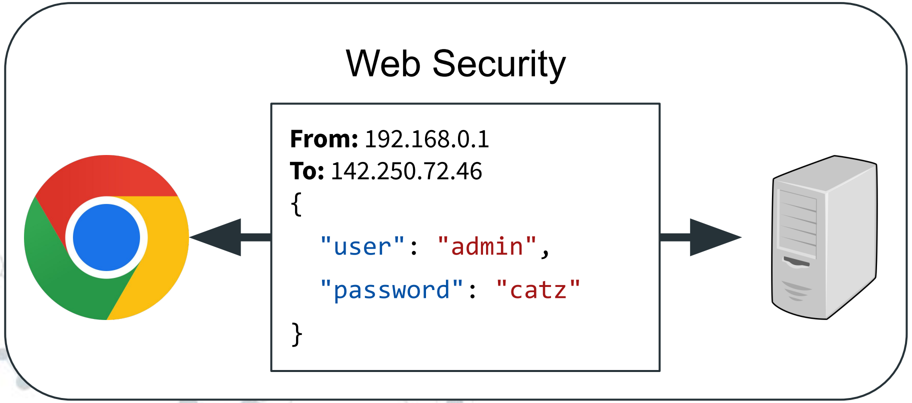
Application Security

Major Differences with Web Security

👍 Less focus on networks

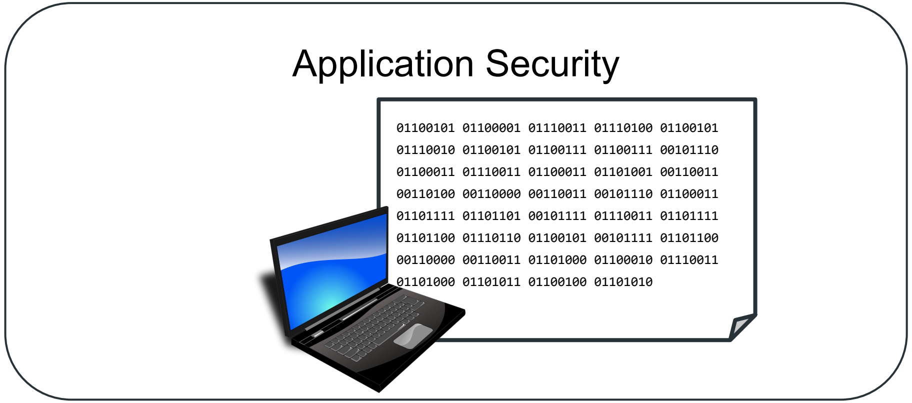
👎 No sandboxing: The entire computer is vulnerable

👎 Code is provided as a compiled binary



Web Security

```
From: 192.168.0.1
To: 142.250.72.46
{
  "user": "admin",
  "password": "catz"
}
```



Application Security

```
01100101 01100001 01110011 01110100 01100101
01110010 01100101 01100111 01100111 00101110
01100011 01110011 01100011 01101001 00110011
00110100 00110000 00110011 00101110 01100011
01101111 01101101 00101111 01110011 01101111
01101100 01110110 01100101 00101111 01101100
00110000 00110011 01101000 01100010 01110011
01101000 01101011 01100100 01101010
```

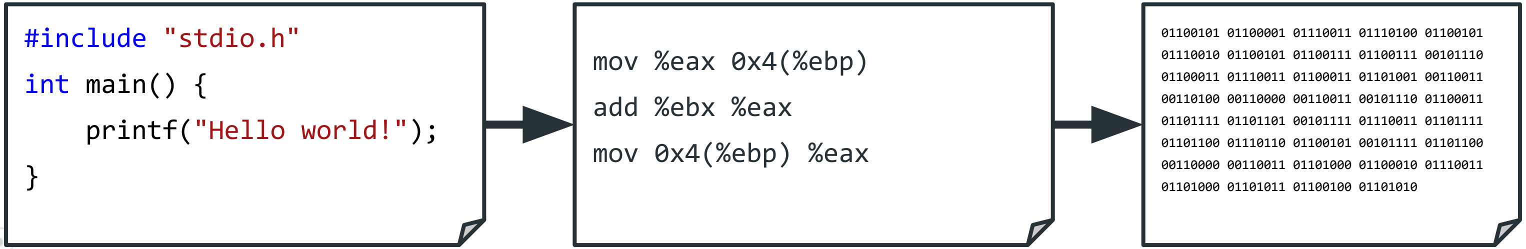
Recap of how Binary Files work

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



```
#include "stdio.h"
int main() {
    printf("Hello world!");
}
```

```
mov %eax 0x4(%ebp)
add %ebx %eax
mov 0x4(%ebp) %eax
```

```
01100101 01100001 01110011 01110100 01100101
01110010 01100101 01100111 01100111 00101110
01100011 01110011 01100011 01101001 00110011
00110100 00110000 00110011 00101110 01100011
01101111 01101101 00101111 01110011 01101111
01101100 01110110 01100101 00101111 01101100
00110000 00110011 01101000 01100010 01110011
01101000 01101011 01100100 01101010
```

Application Basics

Registers:

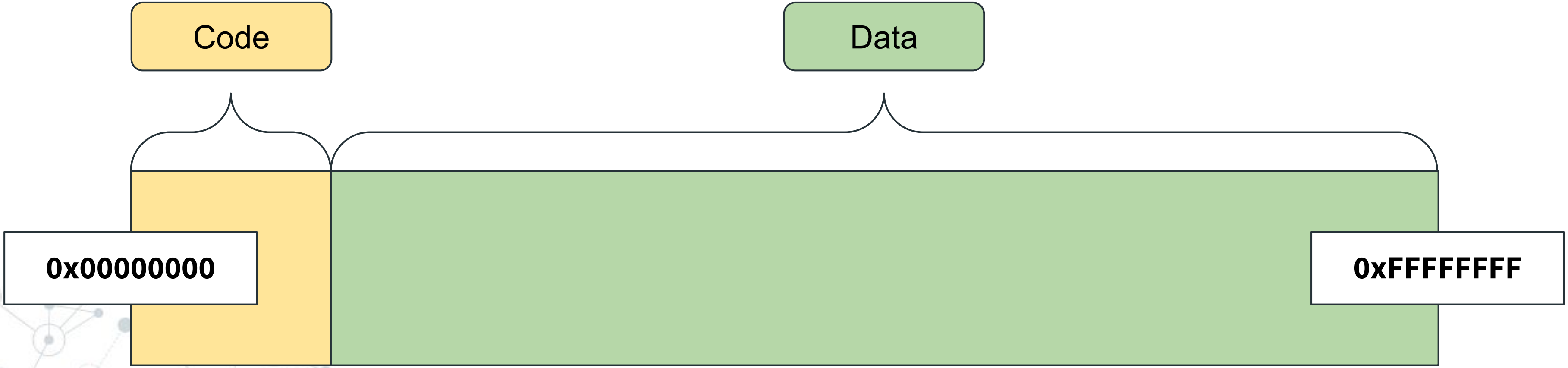
- Binary instructions are run on the CPU
- Instructions are run on small CPU **registers**

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:

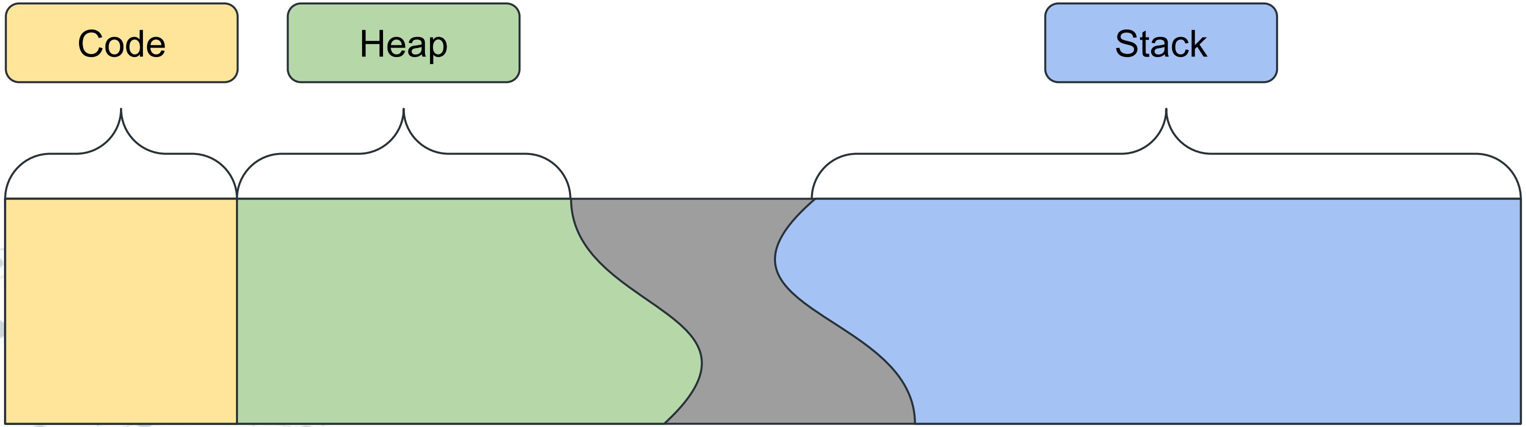
- Registers only hold a few bytes. Most code and data is stored in byte-addressable **memory**



Application Basics

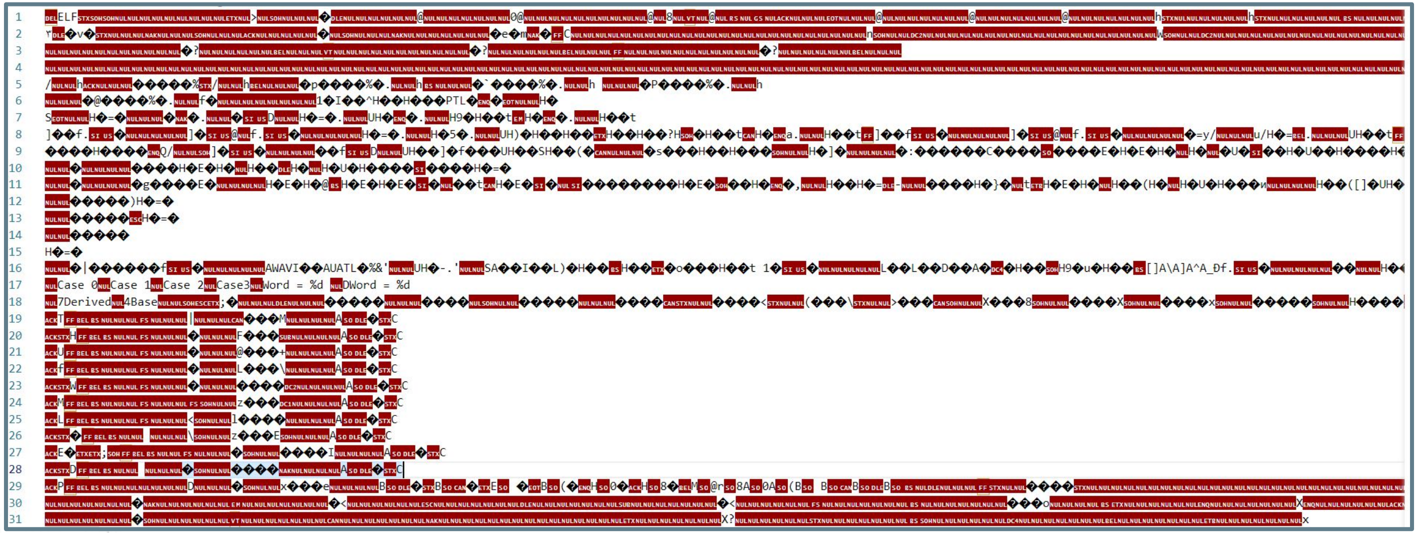
Memory Layout:

- 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?

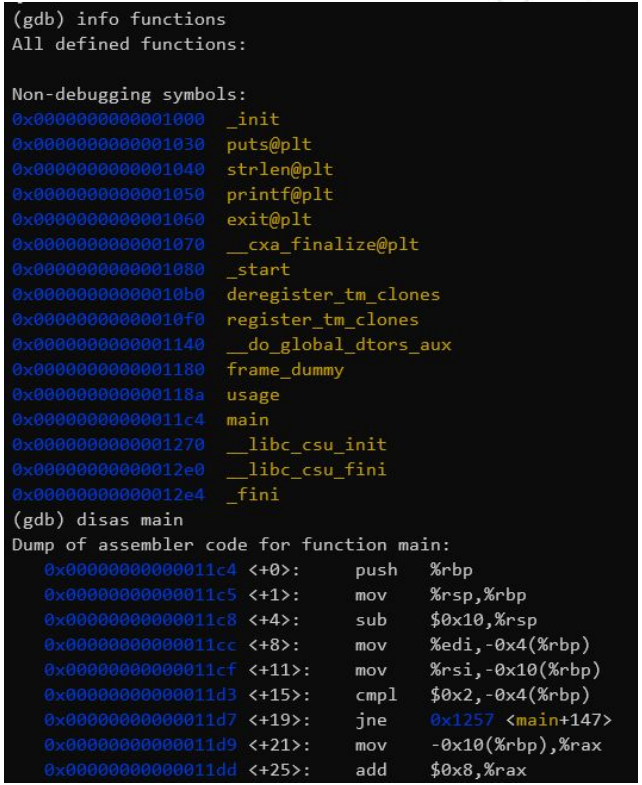


Application Basics

Option 1: Binary debuggers:

Can read and analyze binary code and memory

- GDB
- x64dbg
- Radare
- Binary Ninja



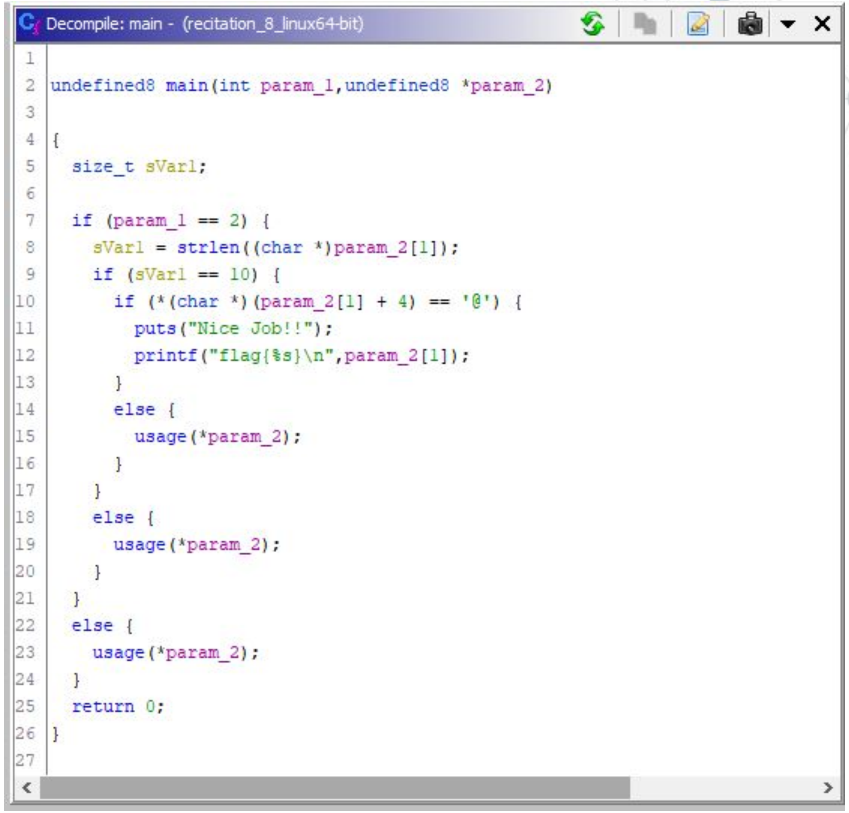
```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  strlen@plt
0x0000000000001050  printf@plt
0x0000000000001060  exit@plt
0x0000000000001070  __cxa_finalize@plt
0x0000000000001080  _start
0x00000000000010b0  deregister_tm_clones
0x00000000000010f0  register_tm_clones
0x0000000000001140  __do_global_dtors_aux
0x0000000000001180  frame_dummy
0x000000000000118a  usage
0x00000000000011c4  main
0x0000000000001270  __libc_csu_init
0x00000000000012e0  __libc_csu_fini
0x00000000000012e4  _fini
(gdb) disas main
Dump of assembler code for function main:
   0x00000000000011c4 <+0>:     push   %rbp
   0x00000000000011c5 <+1>:     mov    %rsp,%rbp
   0x00000000000011c8 <+4>:     sub    $0x10,%rsp
   0x00000000000011cc <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000000011cf <+11>:    mov    %rsi,-0x10(%rbp)
   0x00000000000011d3 <+15>:    cmpl   $0x2,-0x4(%rbp)
   0x00000000000011d7 <+19>:    jne    0x1257 <main+147>
   0x00000000000011d9 <+21>:    mov    -0x10(%rbp),%rax
   0x00000000000011dd <+25>:    add    $0x8,%rax
```

Application Basics

Option 2: Decompilers:

Attempt to reverse assembly back into source code

- Ghidra
- IDA Pro



```

undefined8 main(int param_1,undefined8 *param_2)

{
  size_t sVar1;

  if (param_1 == 2) {
    sVar1 = strlen((char *)param_2[1]);
    if (sVar1 == 10) {
      if (*(char *)(param_2[1] + 4) == '@') {
        puts("Nice Job!!");
        printf("flag{%s}\n",param_2[1]);
      }
      else {
        usage(*param_2);
      }
    }
    else {
      usage(*param_2);
    }
  }
  else {
    usage(*param_2);
  }
  return 0;
}

```

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

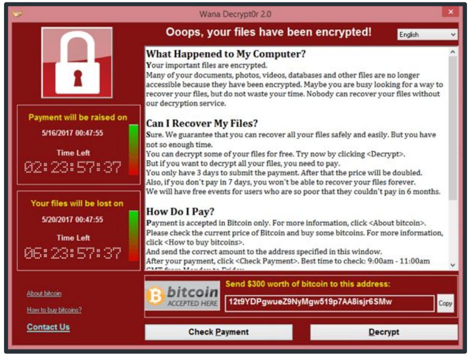
???: The NSA discovered a logic bug that can take over Windows computers

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



Payment will be raised on
5/16/2017 00:47:55
Time Left
02:23:57:37

Your files will be lost on
5/20/2017 00:47:55
Time Left
06:23:57:37

What Happened to My Computer?

Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?

Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Send $300 worth of bitcoin to this address:
12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw

Application Basics

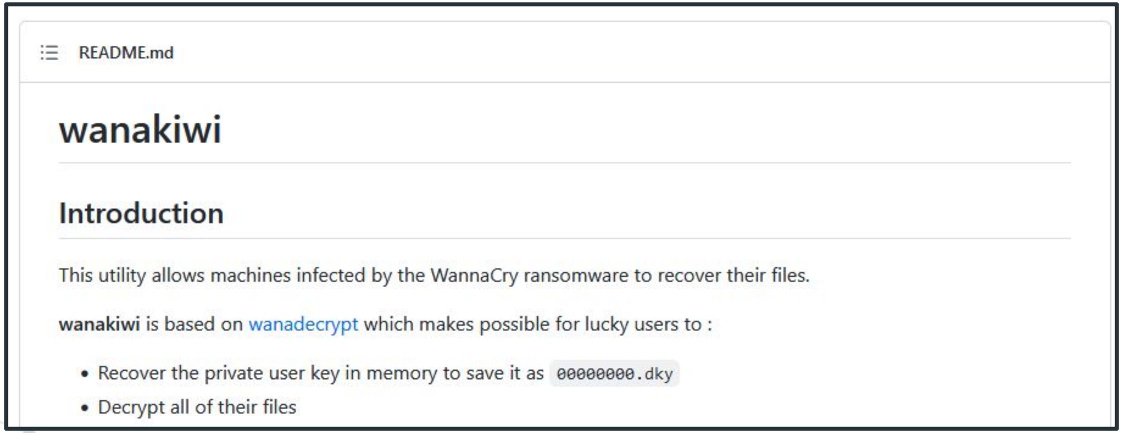
Case study: EternalBlue

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



README.md

wanakiwi

Introduction

This utility allows machines infected by the WannaCry ransomware to recover their files.

wanakiwi is based on wanadecrypt which makes possible for lucky users to :

- Recover the private user key in memory to save it as `00000000.dky`
- Decrypt all of their 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

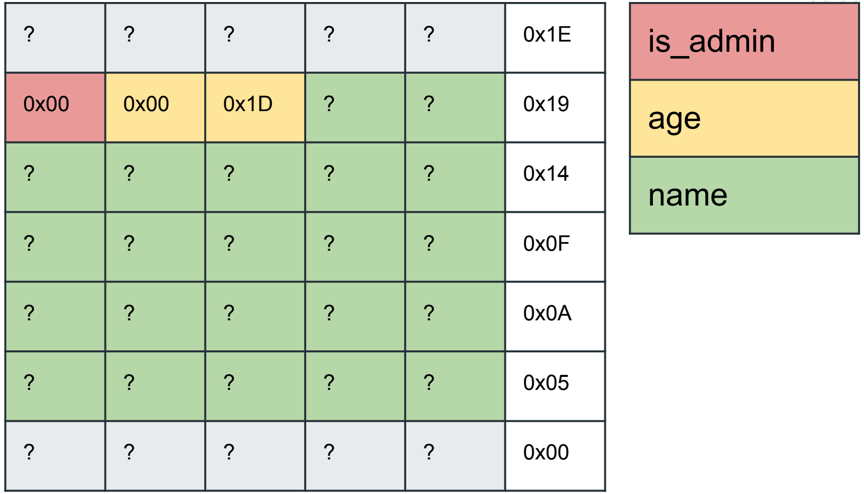
Buffer overflows: Modifying program behavior by writing outside an intended region of memory

```
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

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!

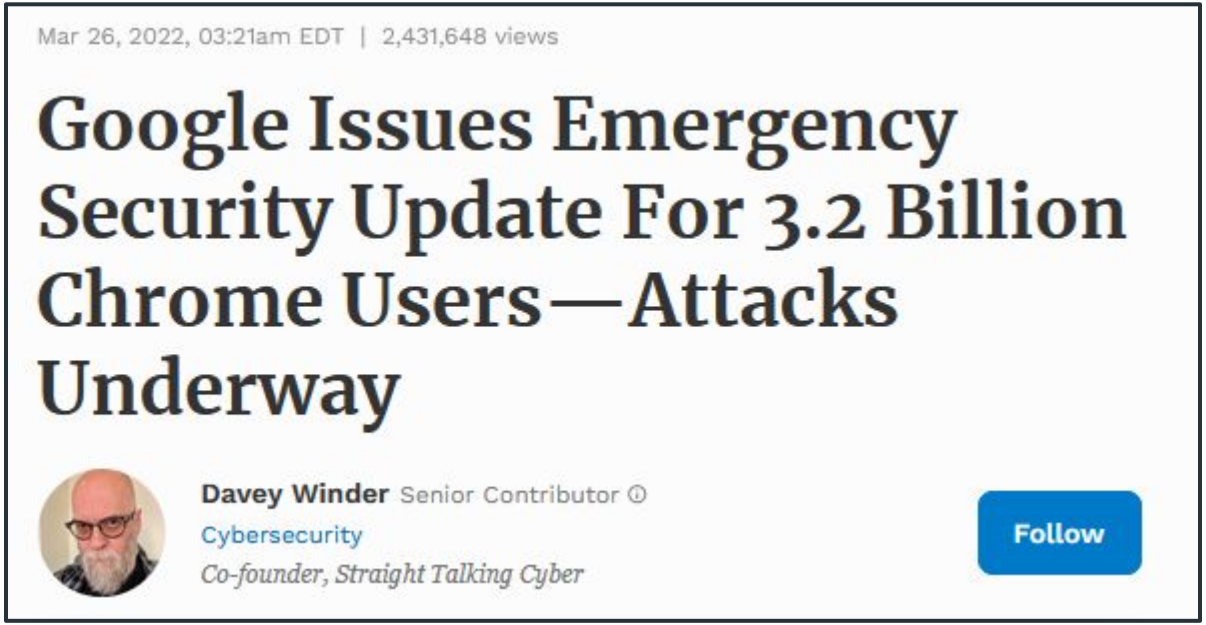
Buffer Overflows

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

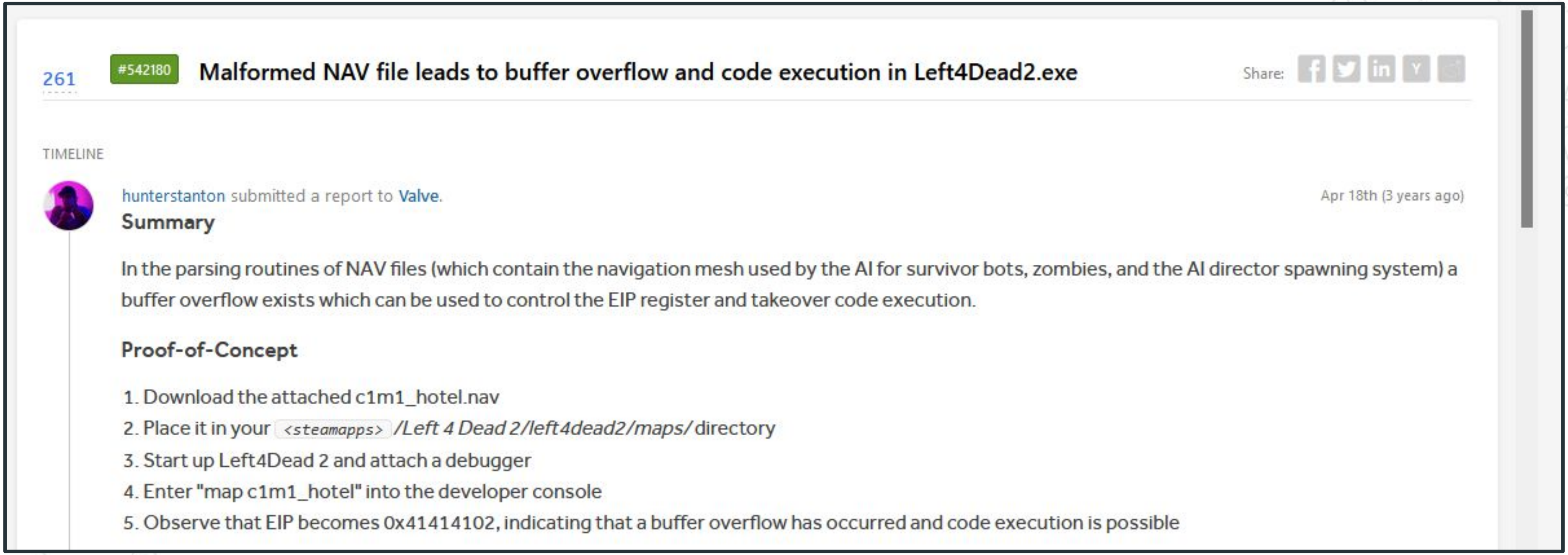
Google Issues Emergency Security Update For 3.2 Billion Chrome Users—Attacks Underway

Davey Winder Senior Contributor ⓘ

Cybersecurity

Co-founder, Straight Talking Cyber

Recent examples



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

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:

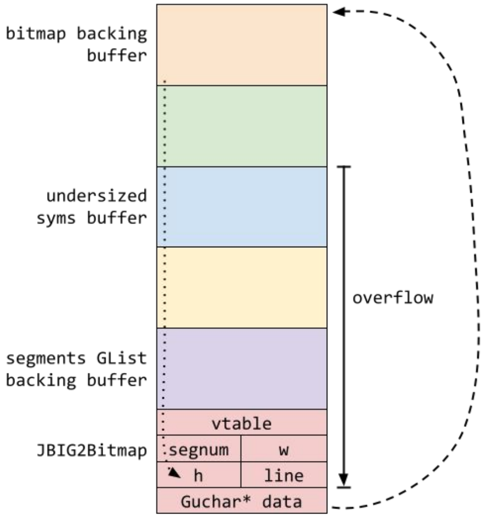
FORCEDENTRY (2021)

FORCEDENTRY

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

FORCEDENTRY

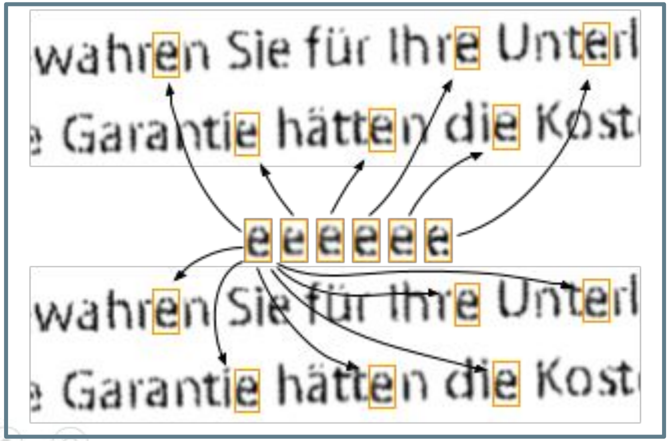
A buffer overflow allows arbitrary memory access…

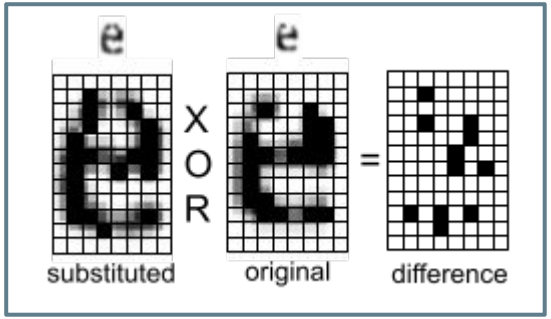


https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

FORCEDENTRY

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

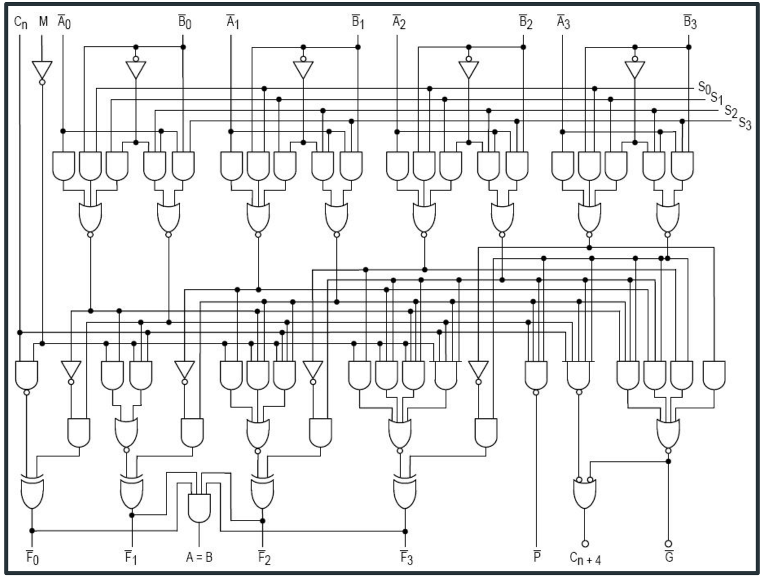




https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

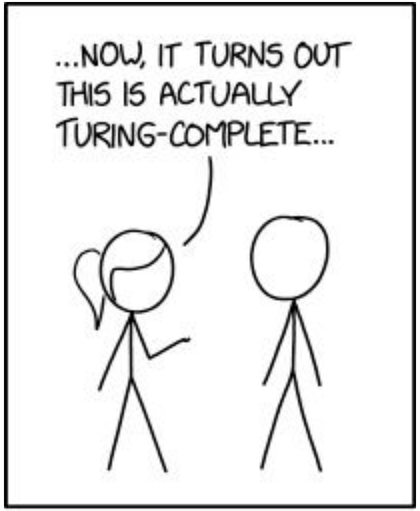
FORCEDENTRY

Easy, just build a CPU from scratch!



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

FORCEDENTRY



THIS PHRASE EITHER MEANS SOMEONE SPENT SIX MONTHS GETTING A DISHWASHER TO PLAY MARIO OR YOU'RE UNDER ATTACK BY A NATION-STATE.

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

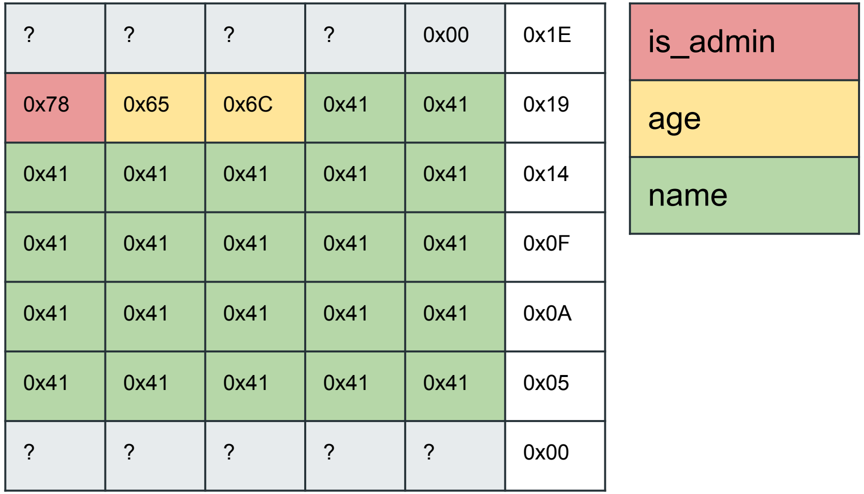
Patch Notes

- **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:

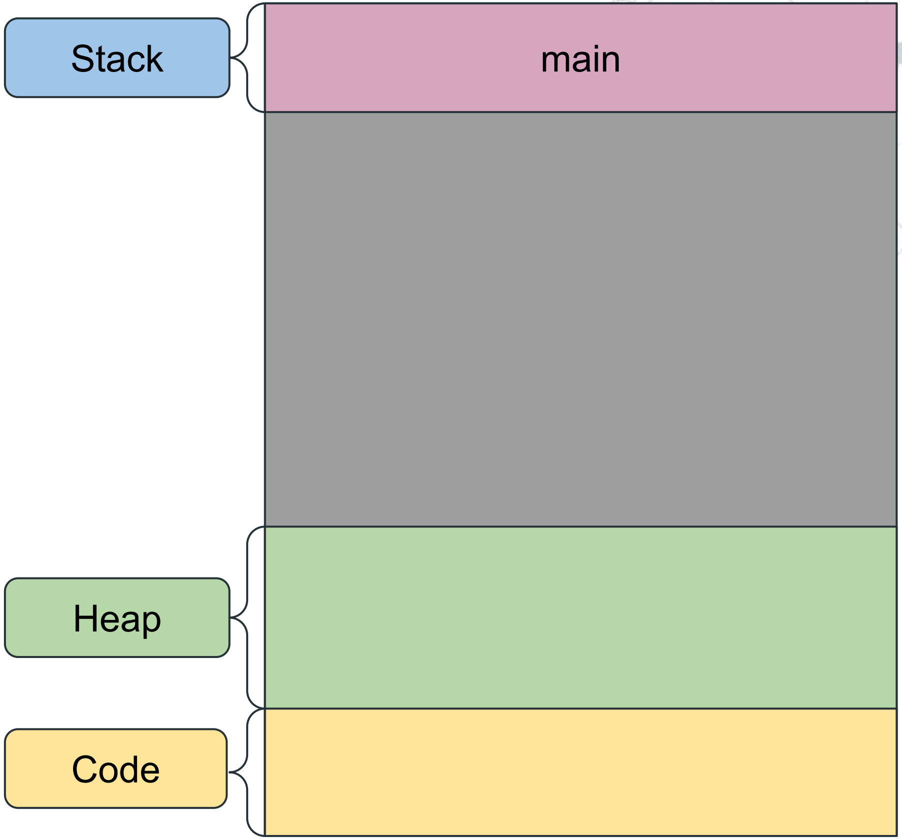
- `gets(buf)`
  - *Writes any amount of user input to buf*
- `strcpy(str1, str2)`
  - *Copies any length of str1 to str2*
- `strcat(str1, str2)`
  - *Concatenates any length of str2 to str1*

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);
}
```

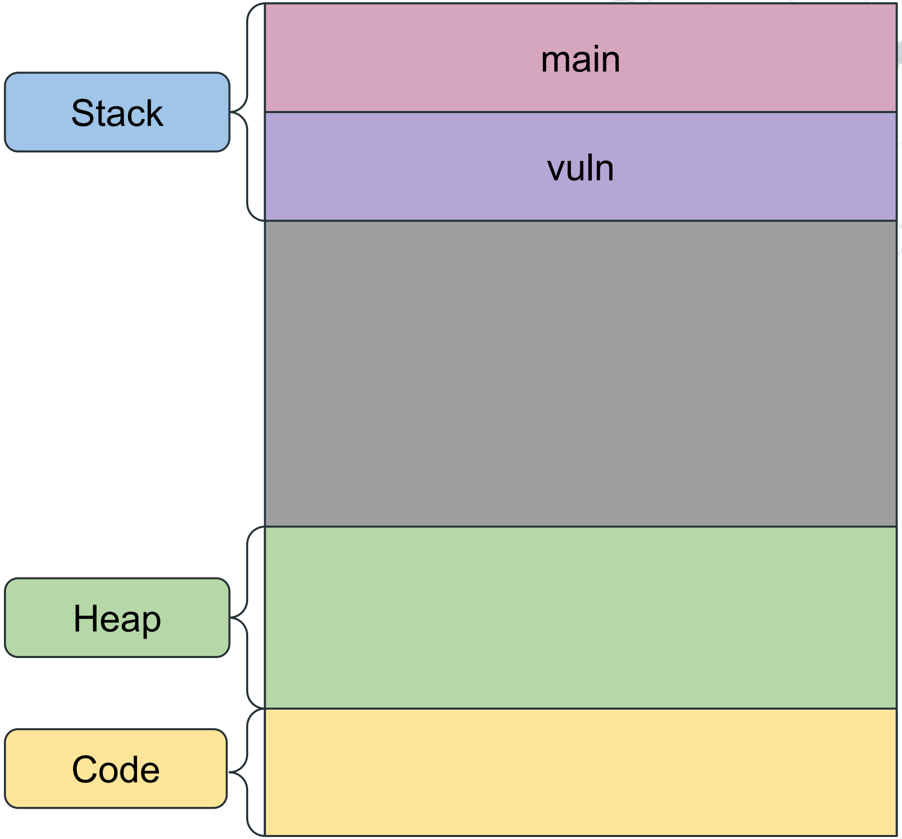


Buffer Overflows

Memory Layout:

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

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



Stack

main

vuln

Heap

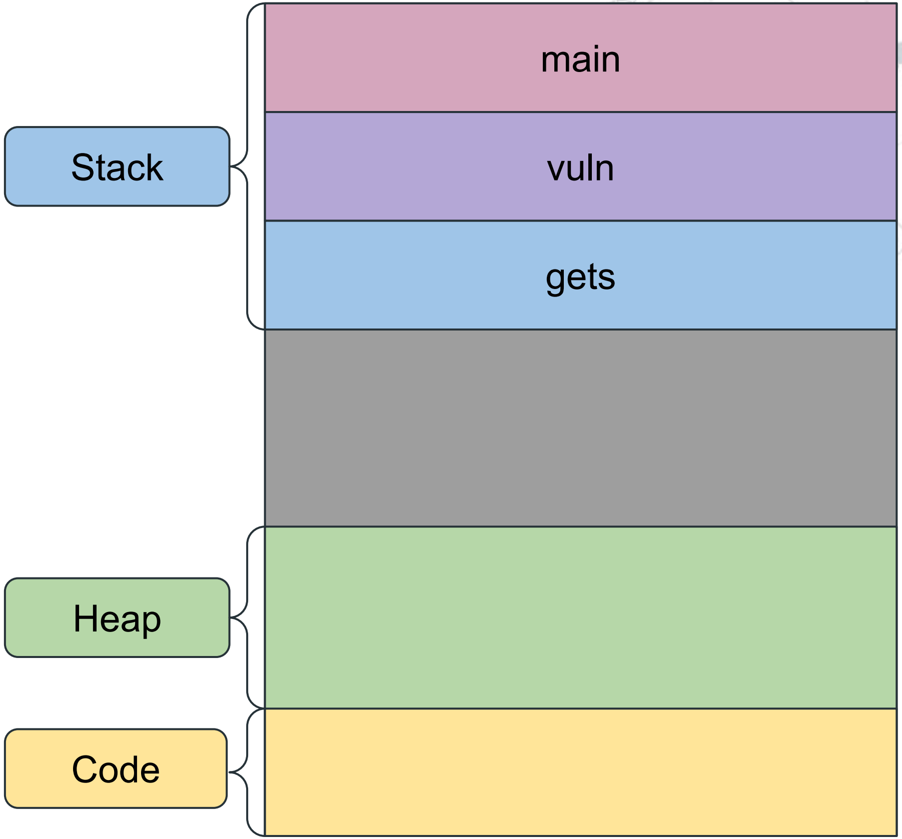
Code

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);
}
```

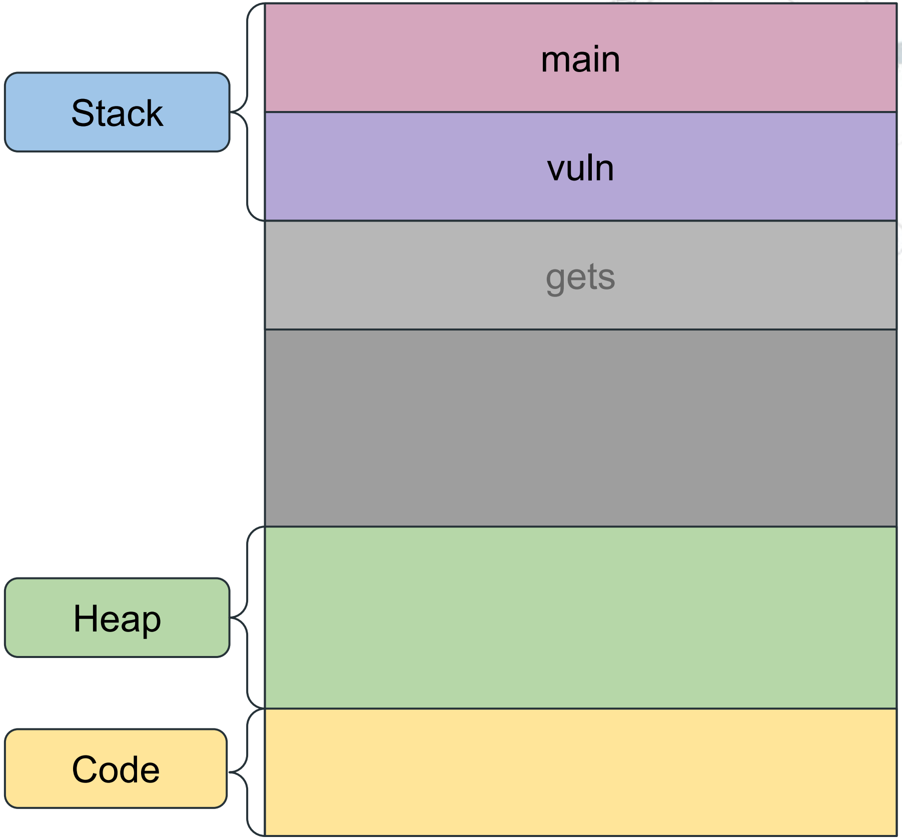


Buffer Overflows

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);
}
```



Stack

main

vuln

gets

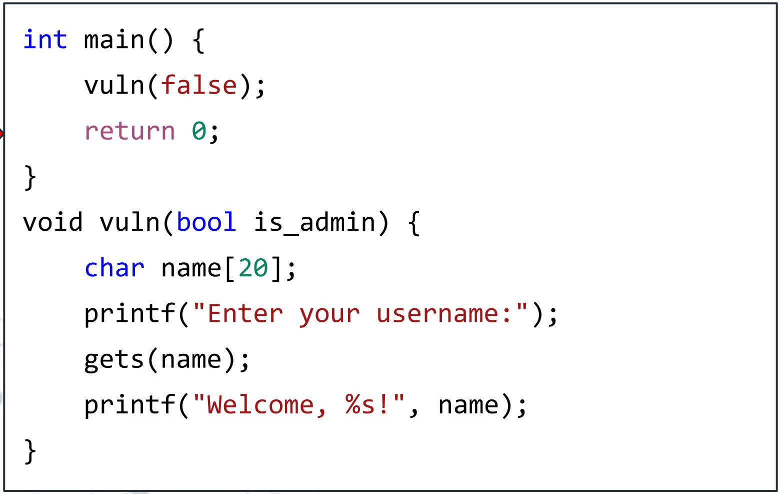
Heap

Code

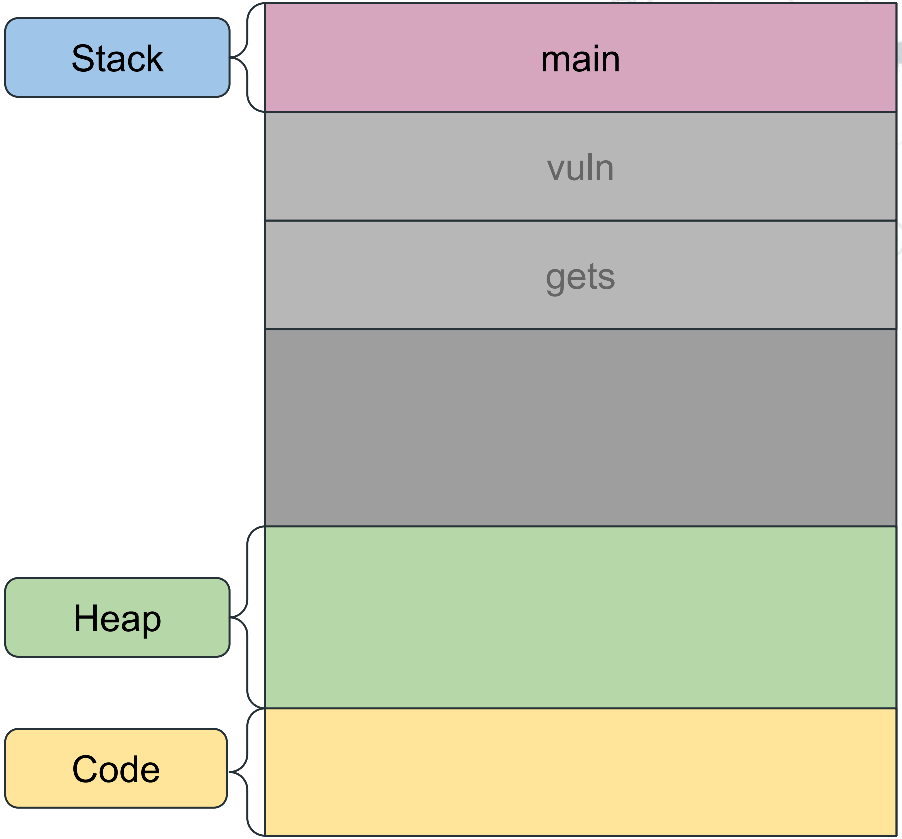
Buffer Overflows

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);
}
```

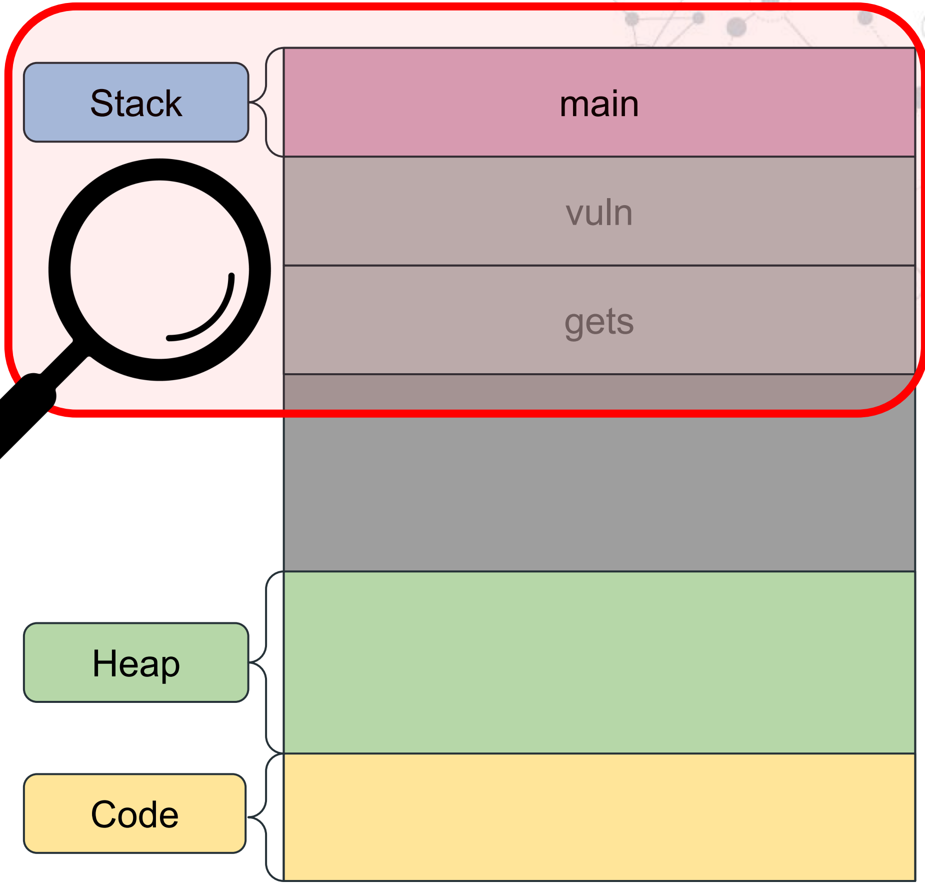


Buffer Overflows

Memory Layout:

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

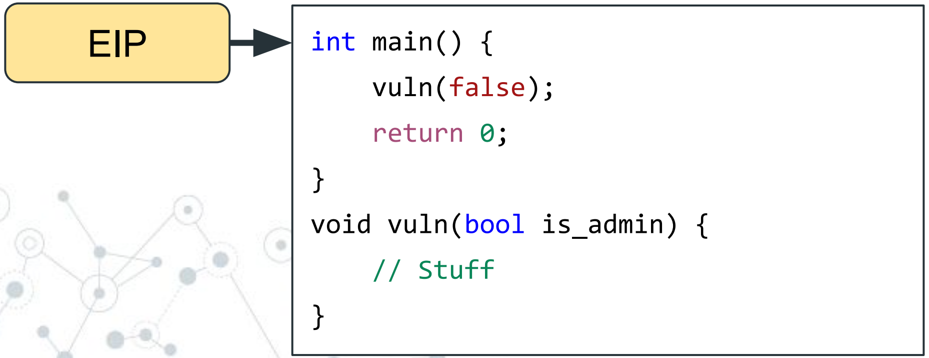
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```



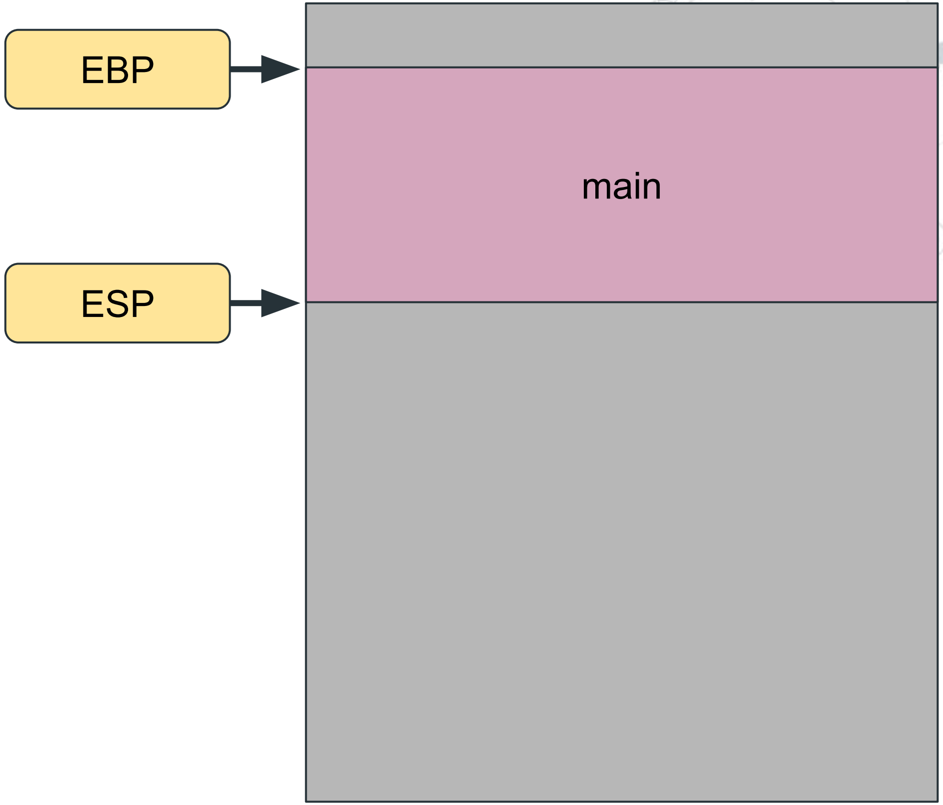
Buffer Overflows

Important registers:

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



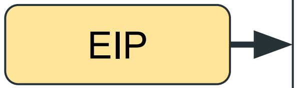
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```



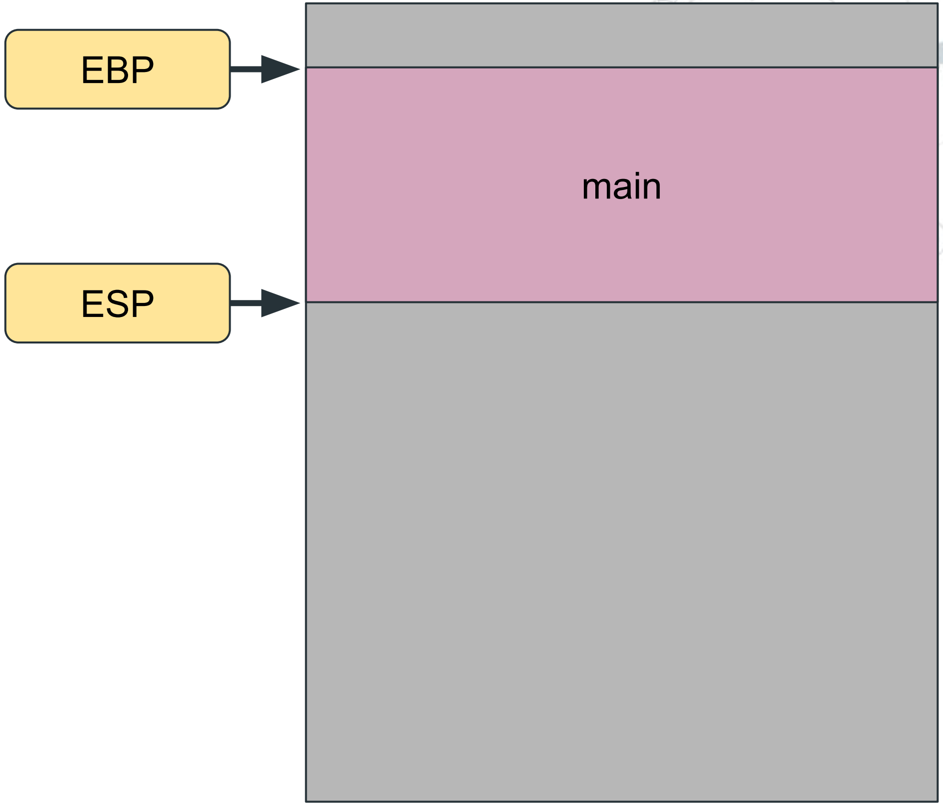
Buffer Overflows

Important registers:

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



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

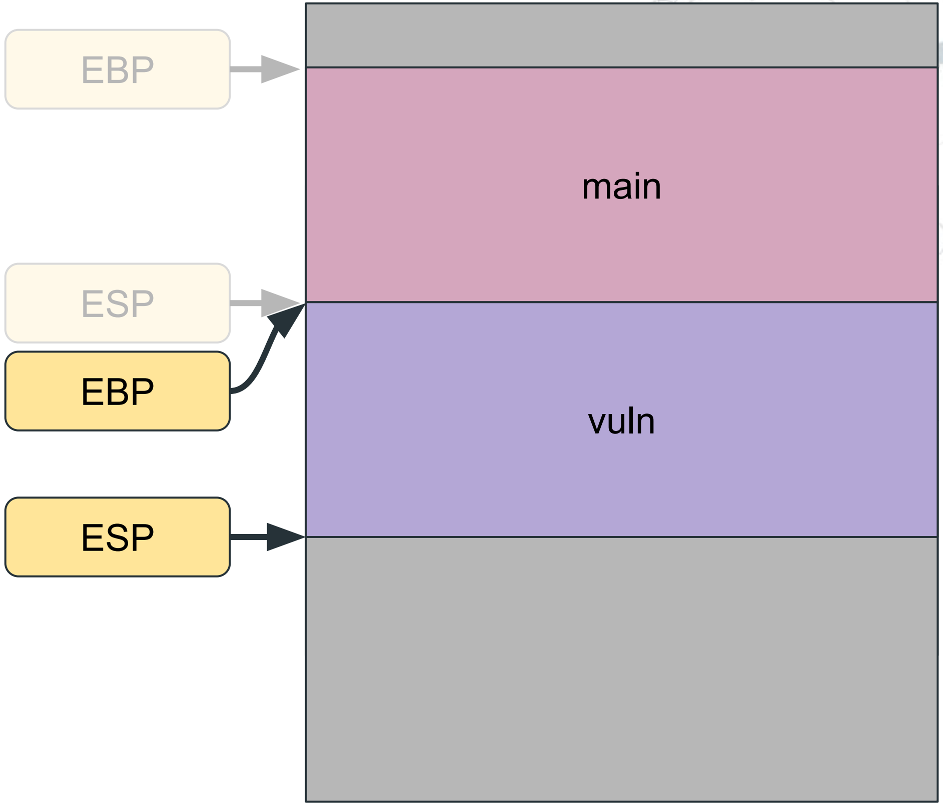


Buffer Overflows

Important registers:

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

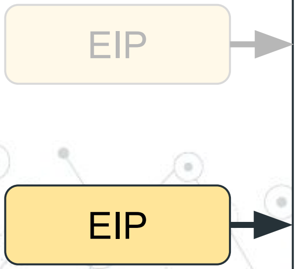
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```



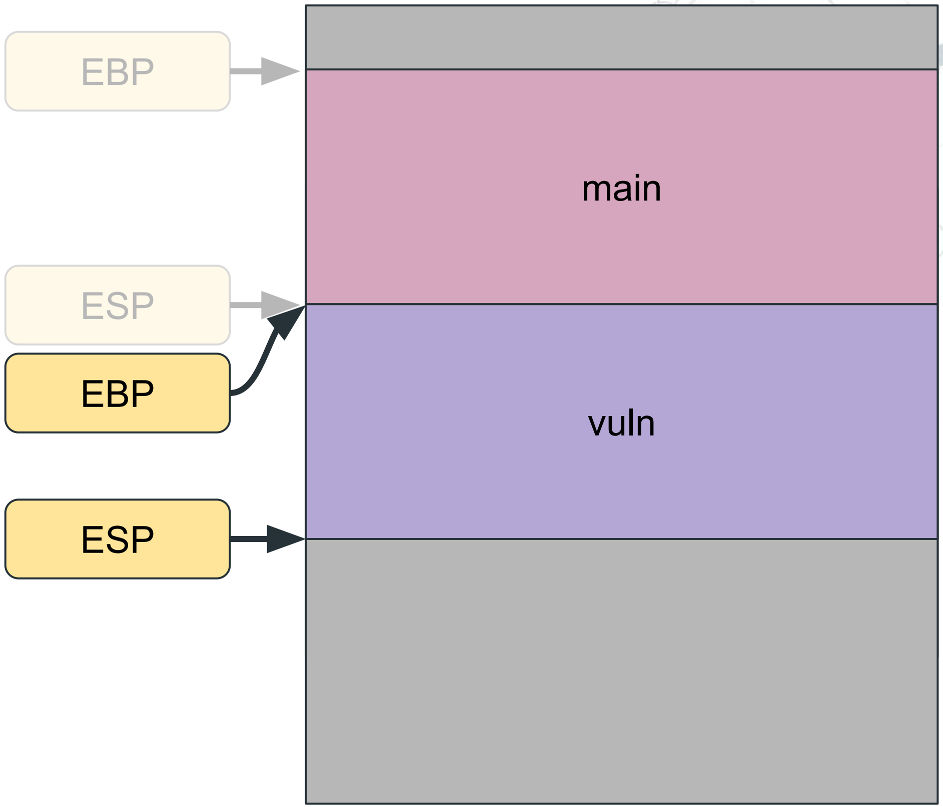
Buffer Overflows

Important registers:

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



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

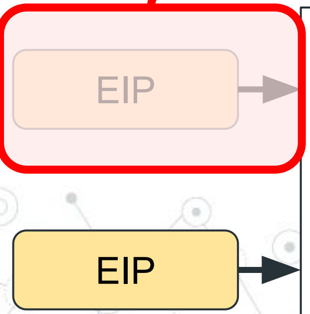


Buffer Overflows

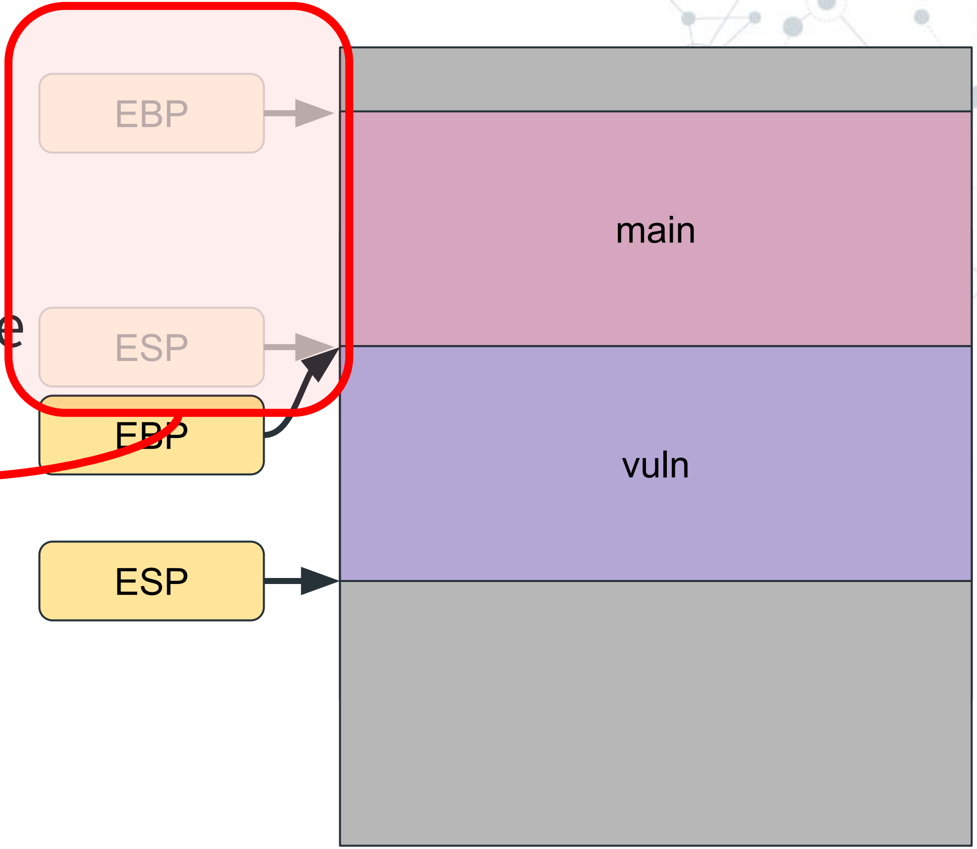
Important registers:

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

We need to store these!



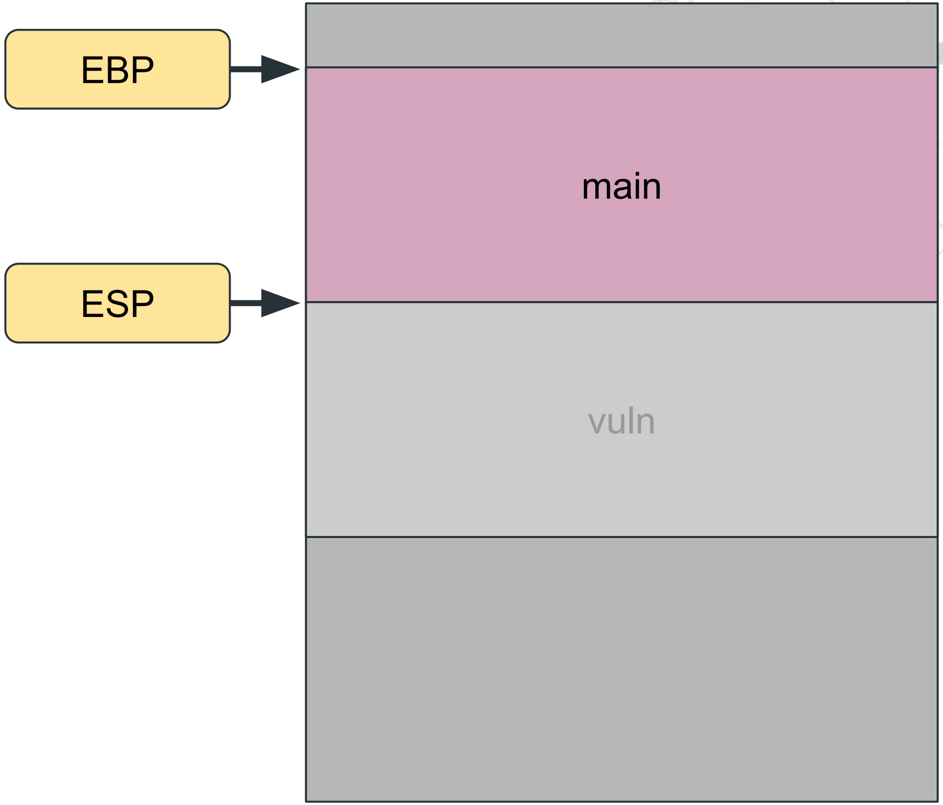
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

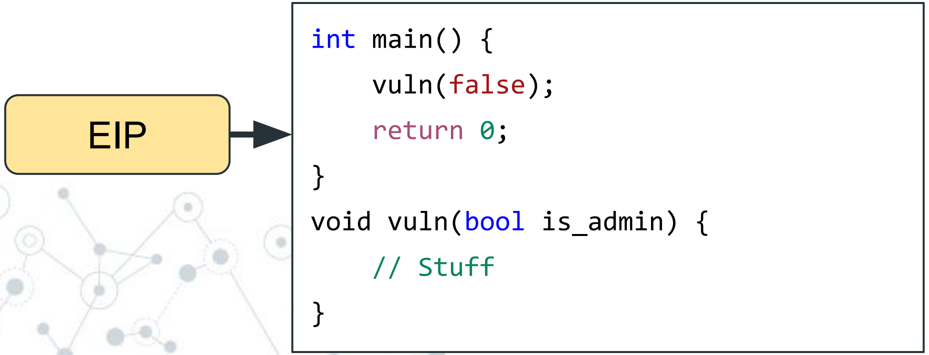


Buffer Overflows

Important registers:

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

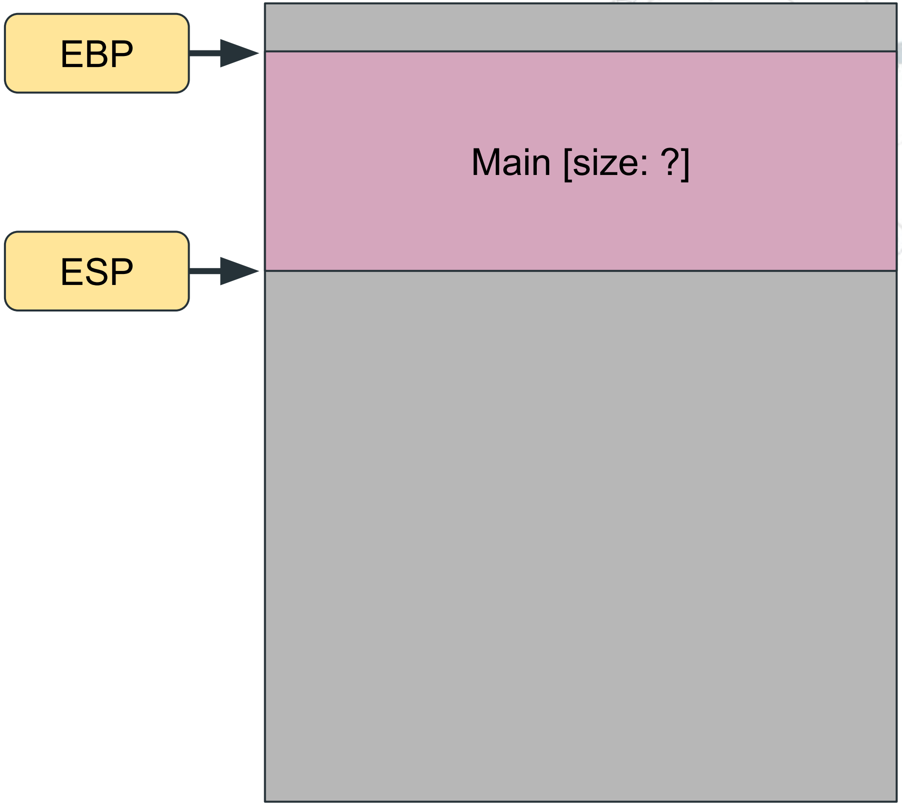




```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    // Stuff
}
```

Buffer Overflows

On each function call…

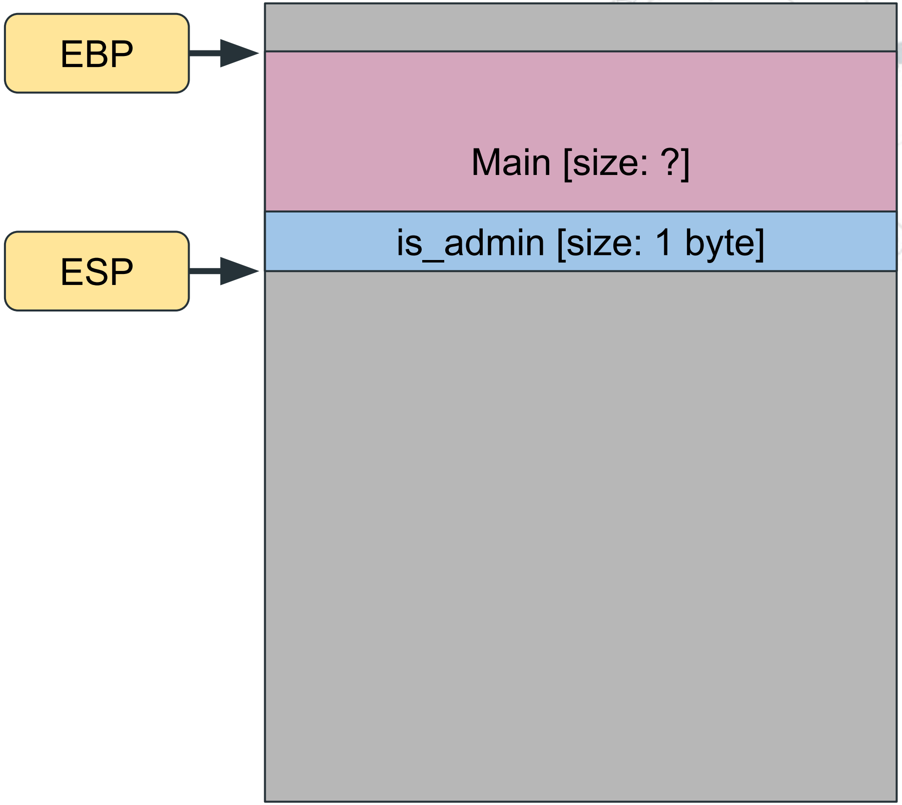


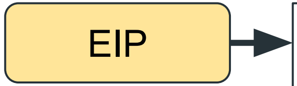
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

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

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



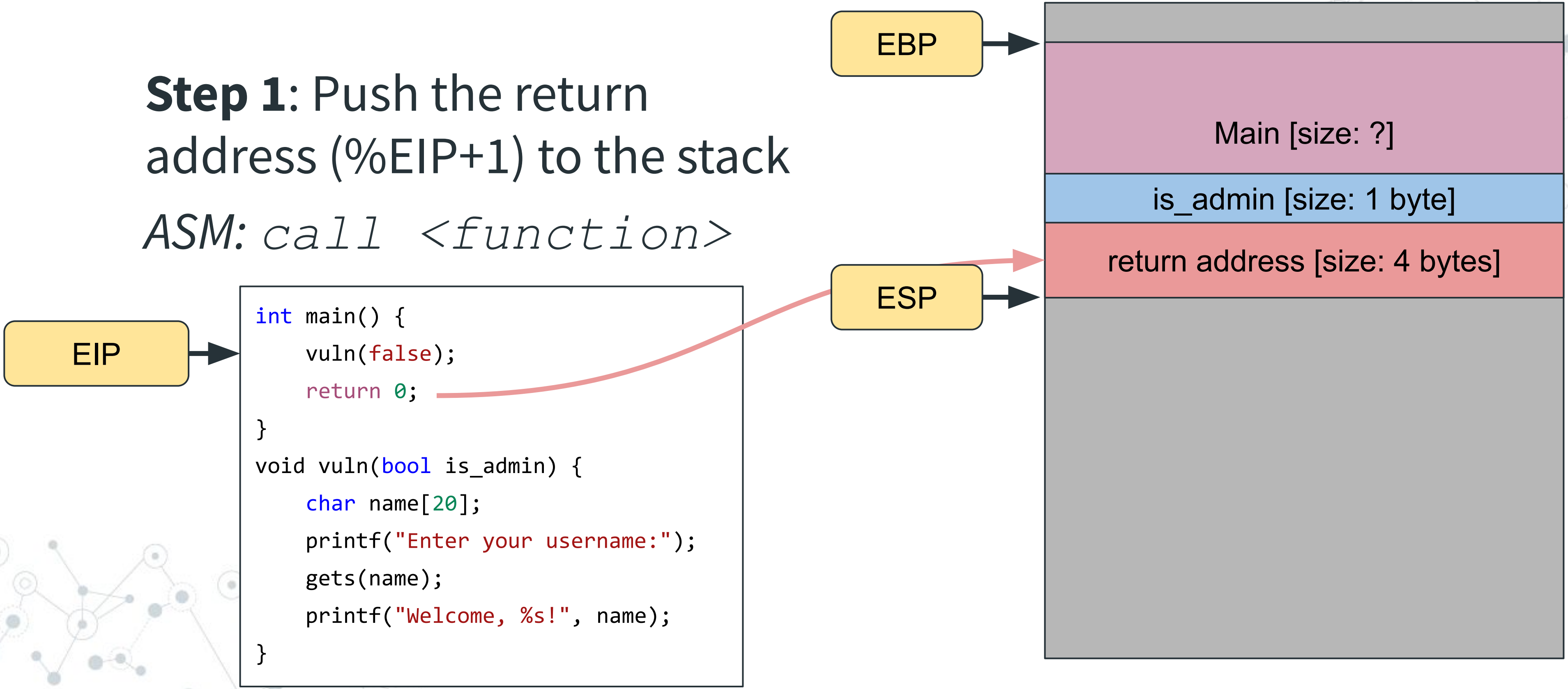


```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

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

ASM: `call <function>`

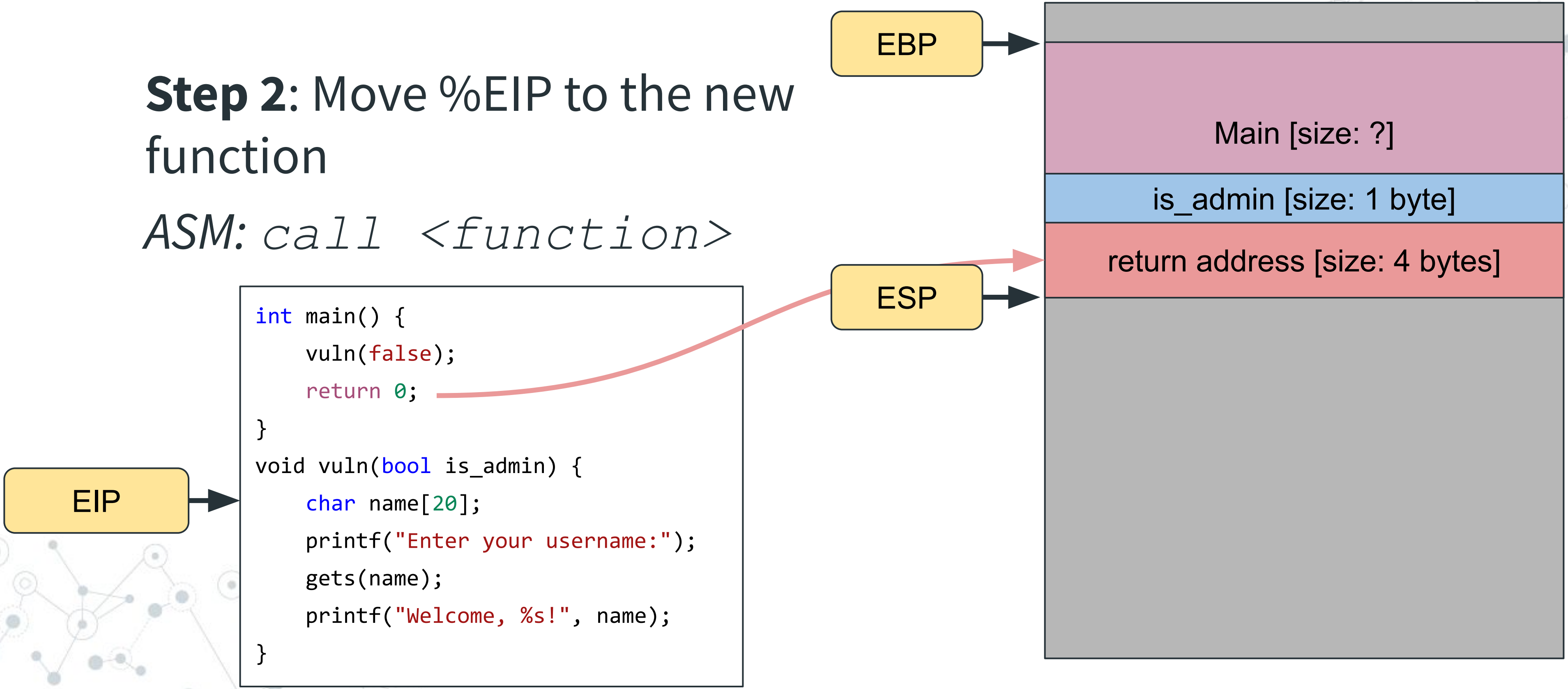


```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

Step 2: Move %EIP to the new function

ASM: `call <function>`

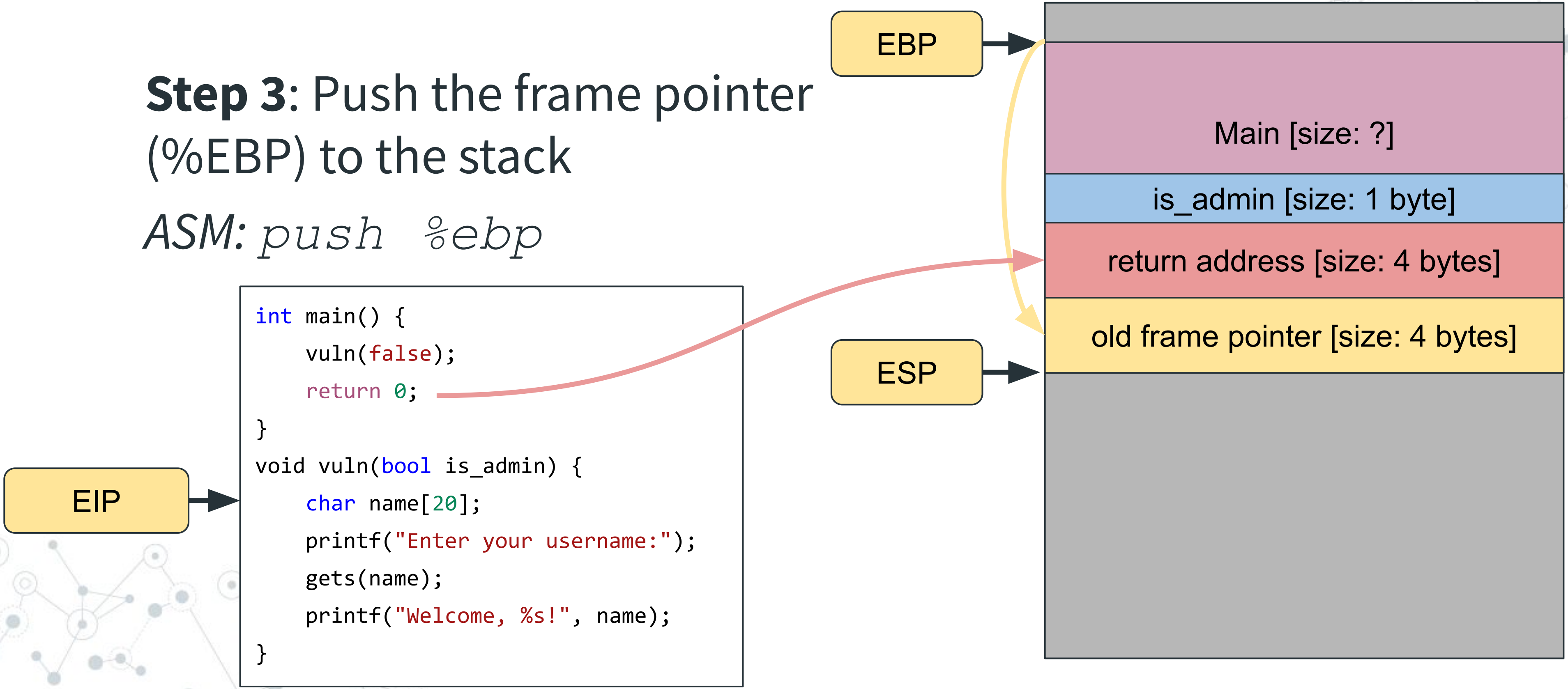


```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

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

ASM: `push %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
EBP
ESP
Main [size: ?]
is_admin [size: 1 byte]
return address [size: 4 bytes]
old frame pointer [size: 4 bytes]

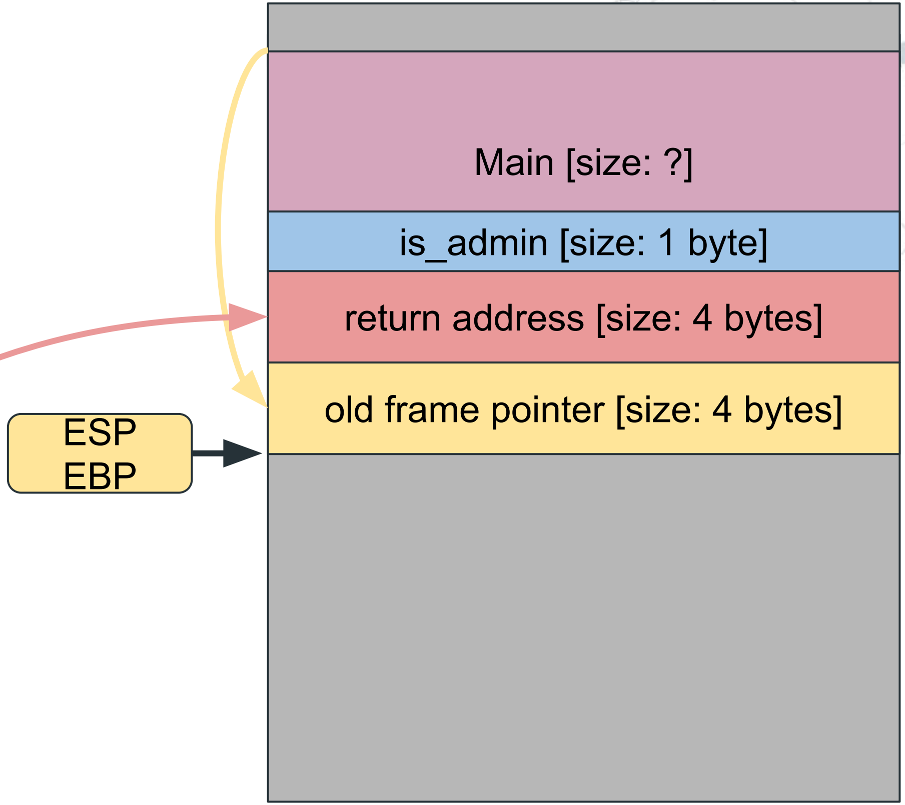
Buffer Overflows

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

ASM: `mov %esp,%ebp`

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);
}
```

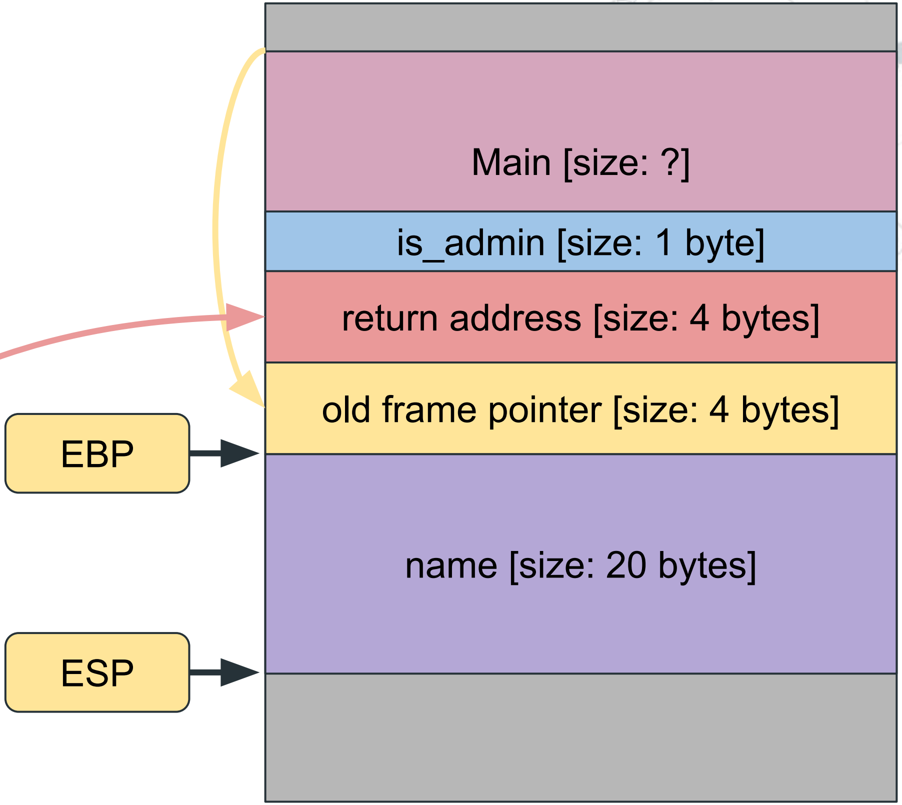

Main [size: ?]
is_admin [size: 1 byte]
return address [size: 4 bytes]
old frame pointer [size: 4 bytes]
ESP
EBP

Buffer Overflows

Step 5: Subtract space for variables from %ESP

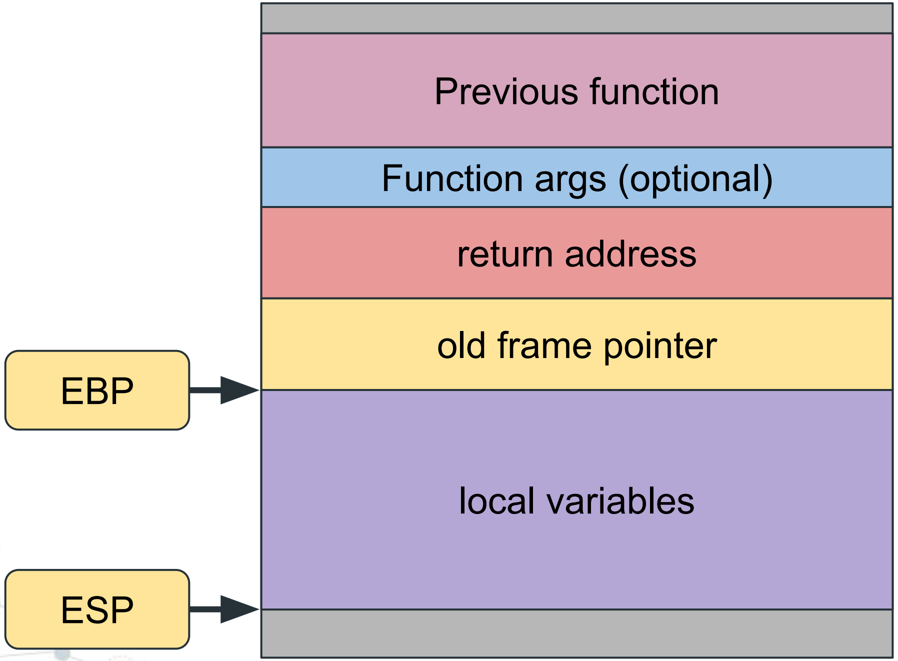
ASM: `sub $0x4,%esp`

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



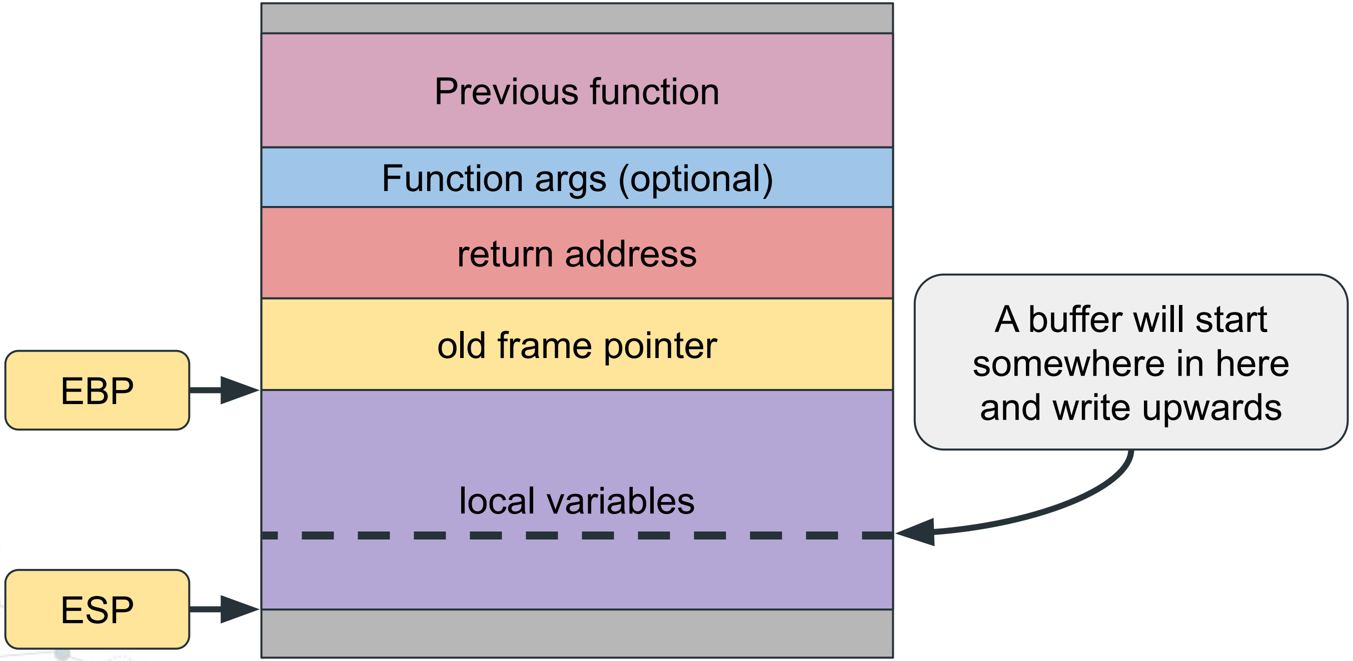
Buffer Overflows

Memory layout for each function



Buffer Overflows

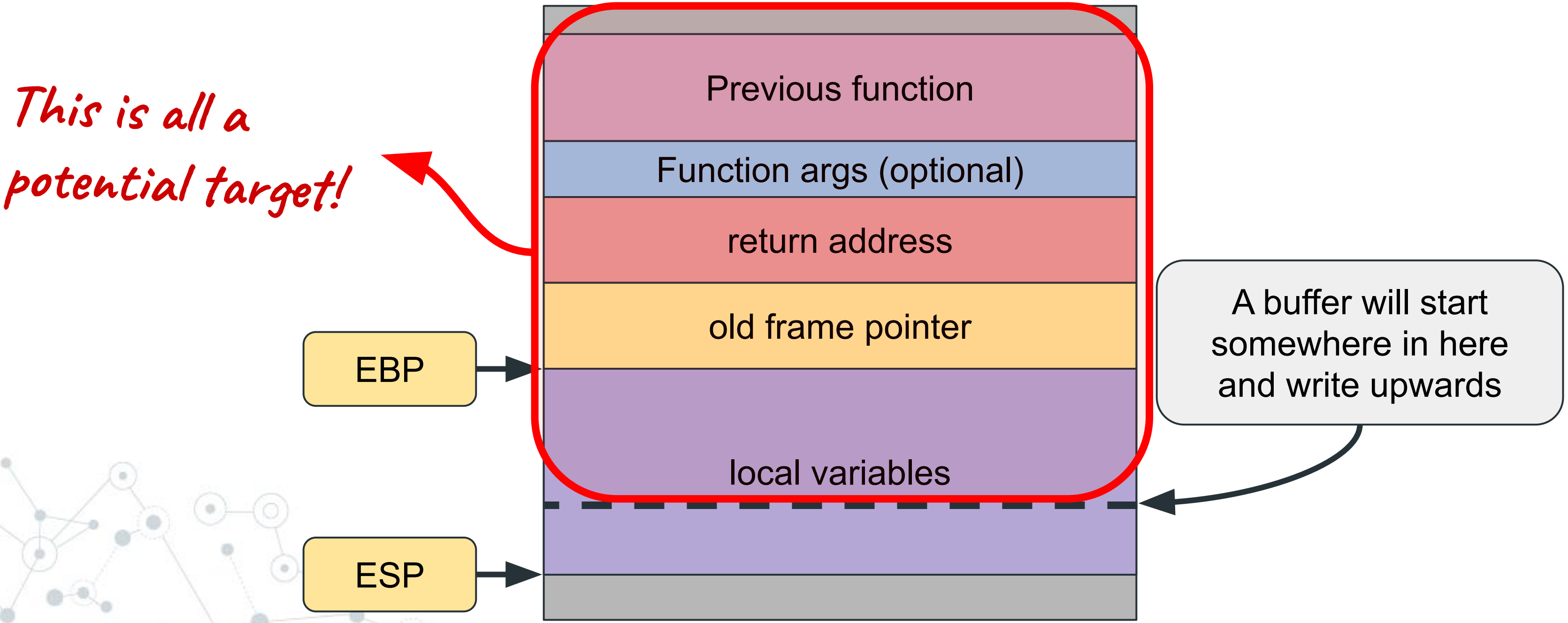
Memory layout for each function



A buffer will start somewhere in here and write upwards

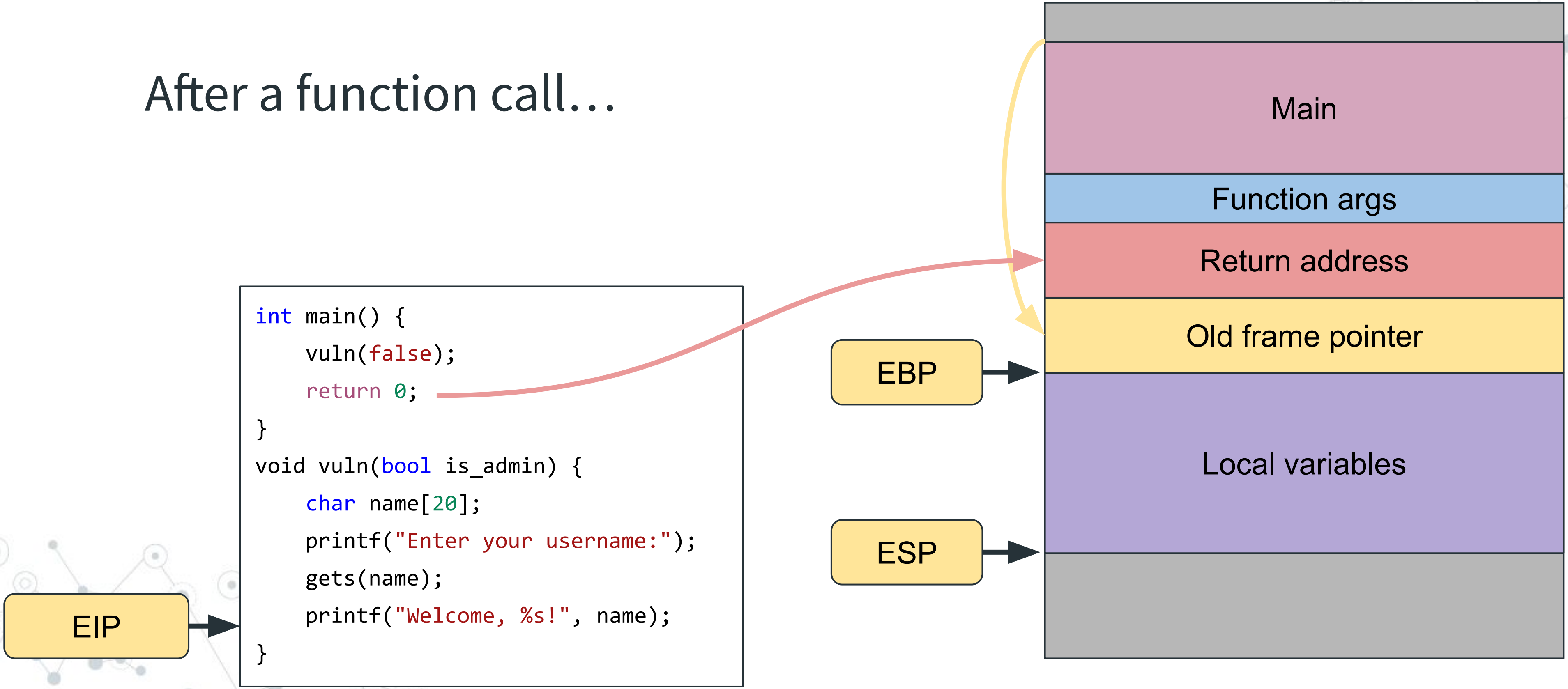
Buffer Overflows

Memory layout for each function



Buffer Overflows

After a function call…



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

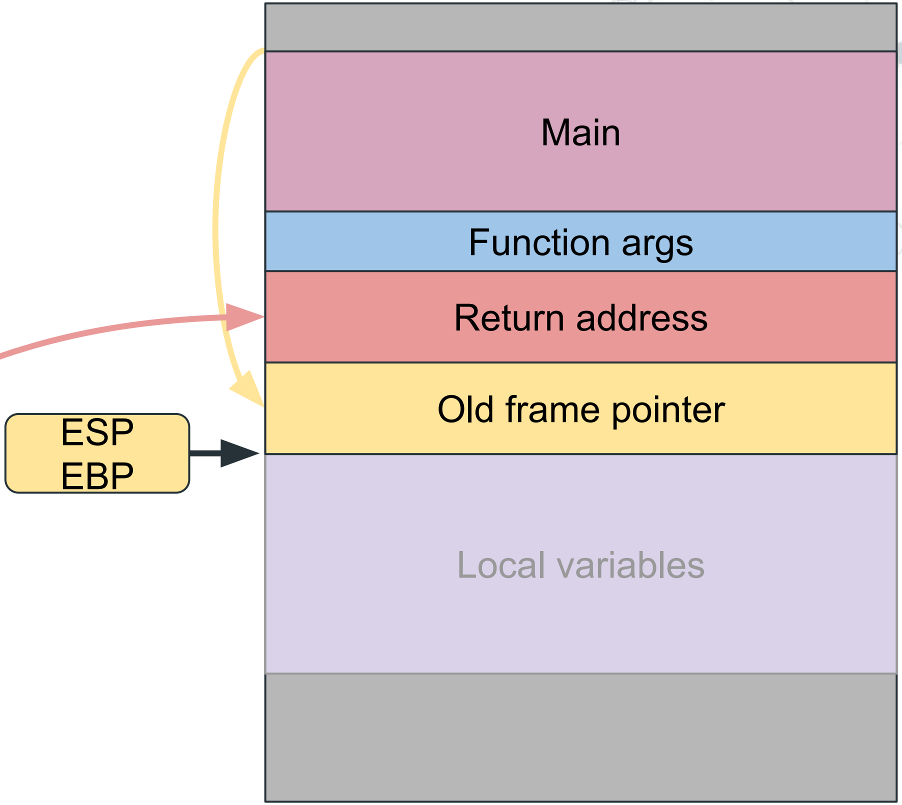
Buffer Overflows

Step 1: 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);
}
```

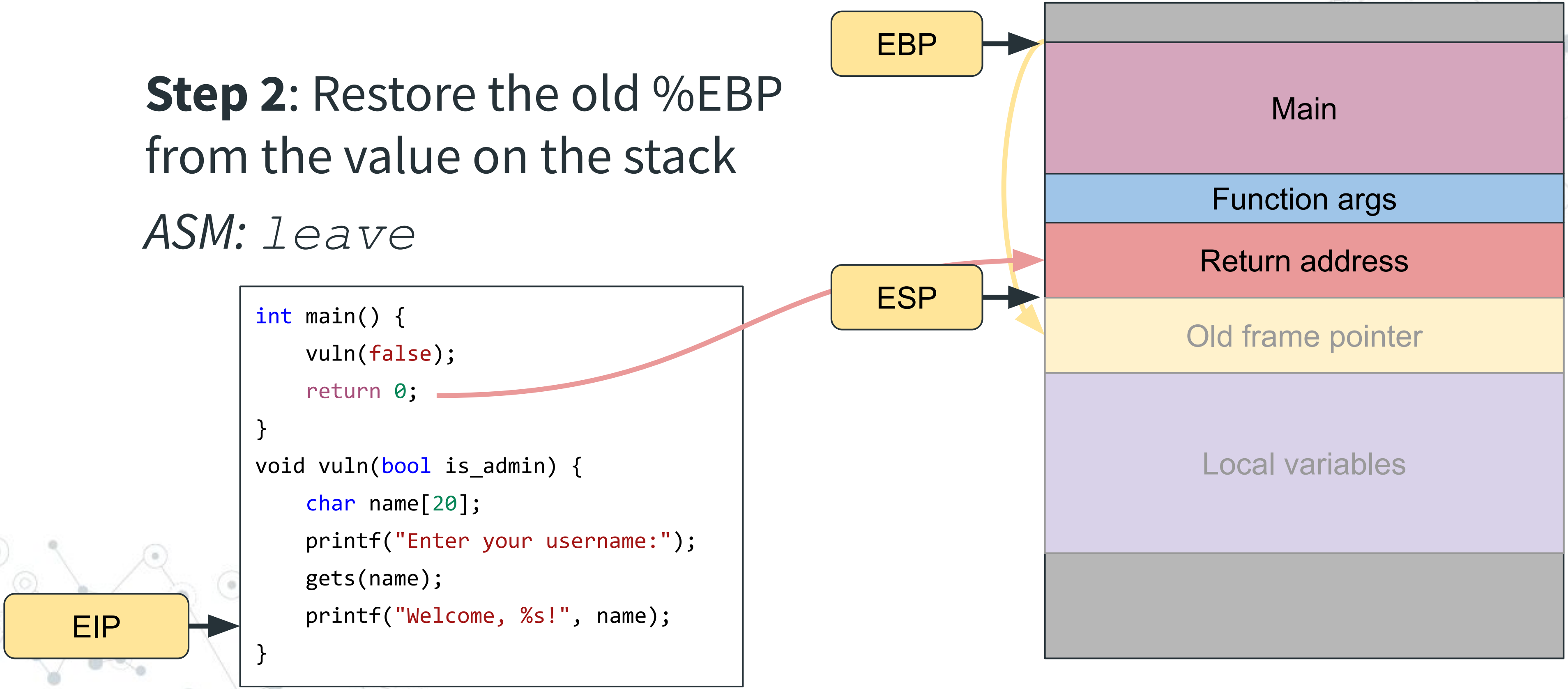
EIP



Buffer Overflows

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

ASM: `leave`

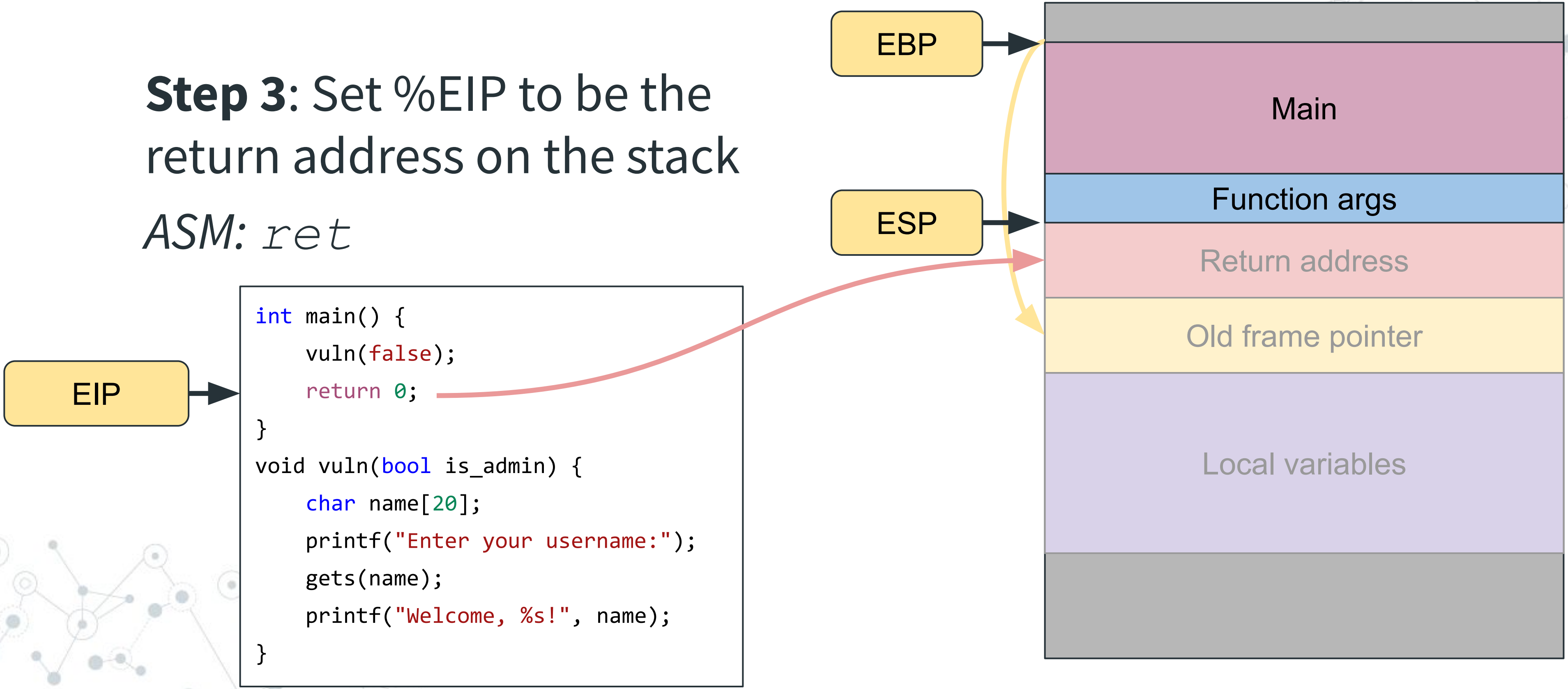


```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

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

ASM: `ret`

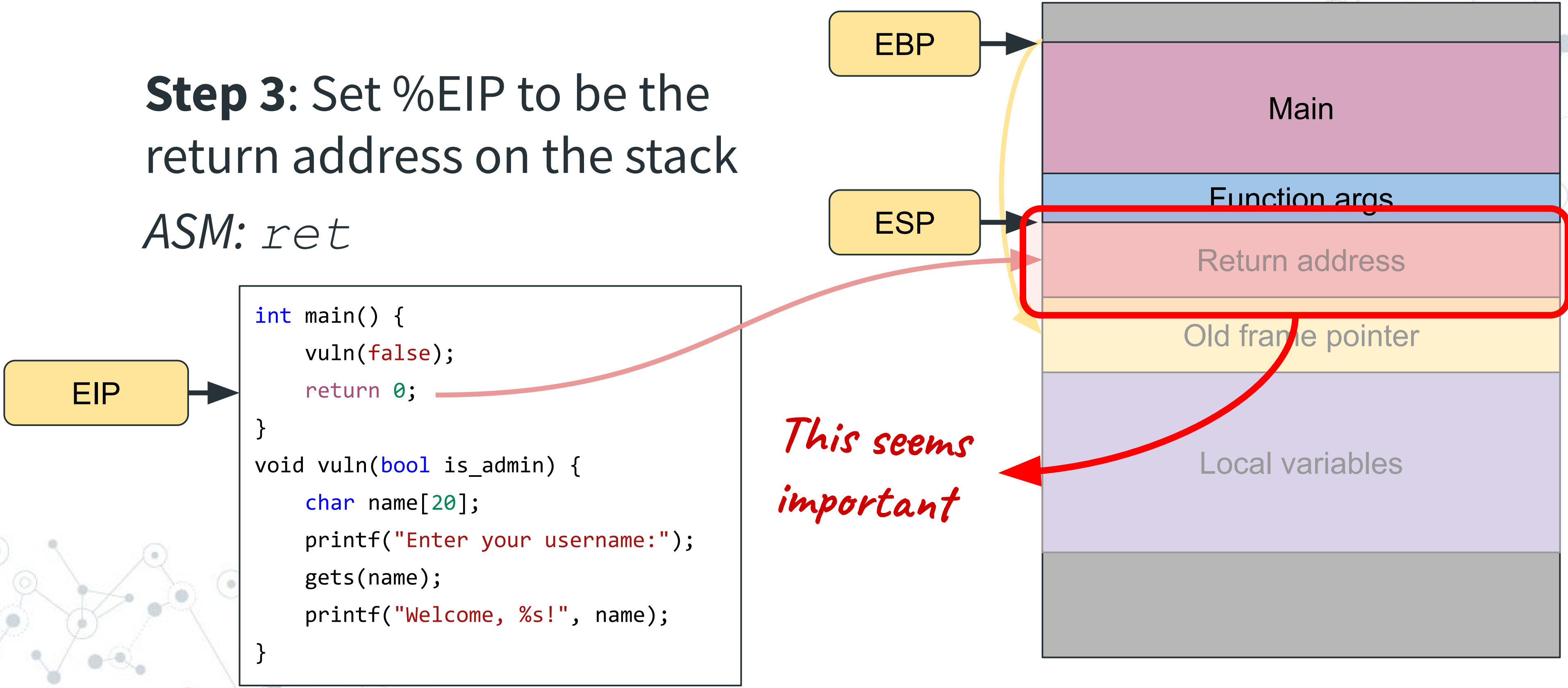


```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

Buffer Overflows

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

ASM: `ret`



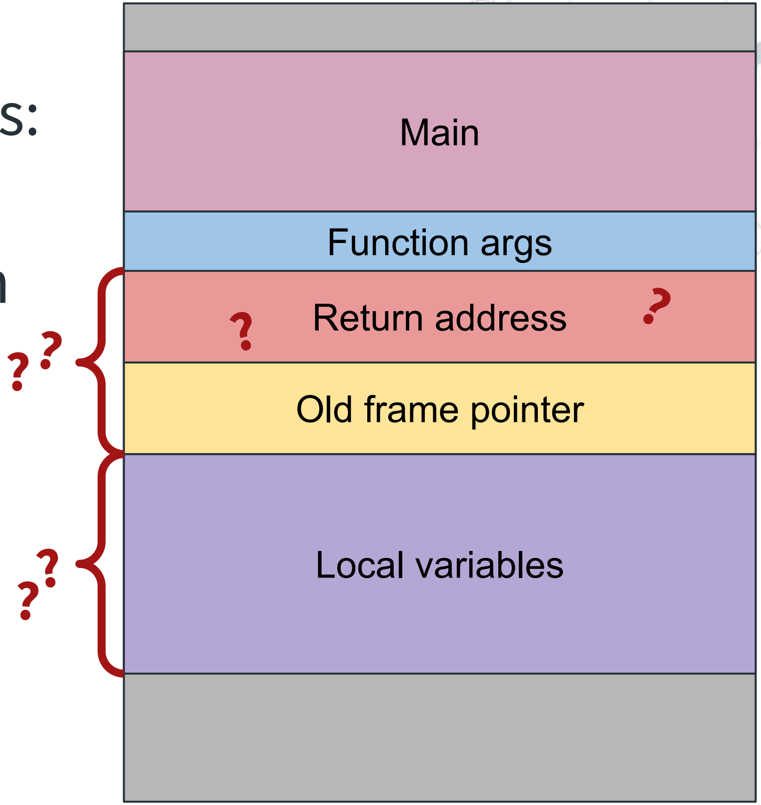
```
int main() {
    vuln(false);
    return 0;
}
void vuln(bool is_admin) {
    char name[20];
    printf("Enter your username:");
    gets(name);
    printf("Welcome, %s!", name);
}
```

This seems important

Buffer Overflows

An attacker needs to know two things:

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

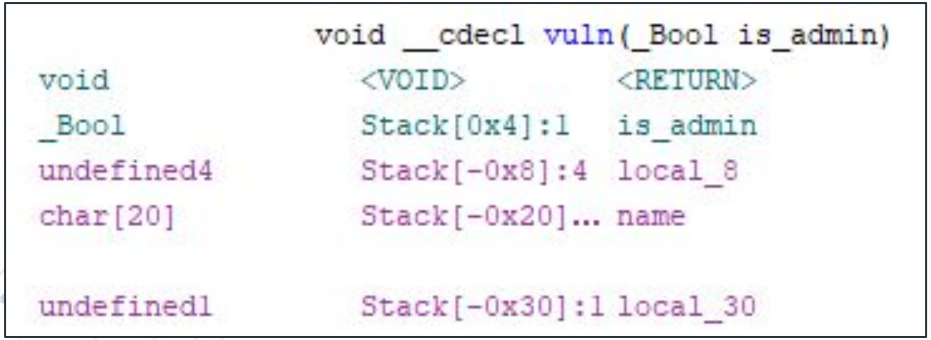


Buffer Overflows

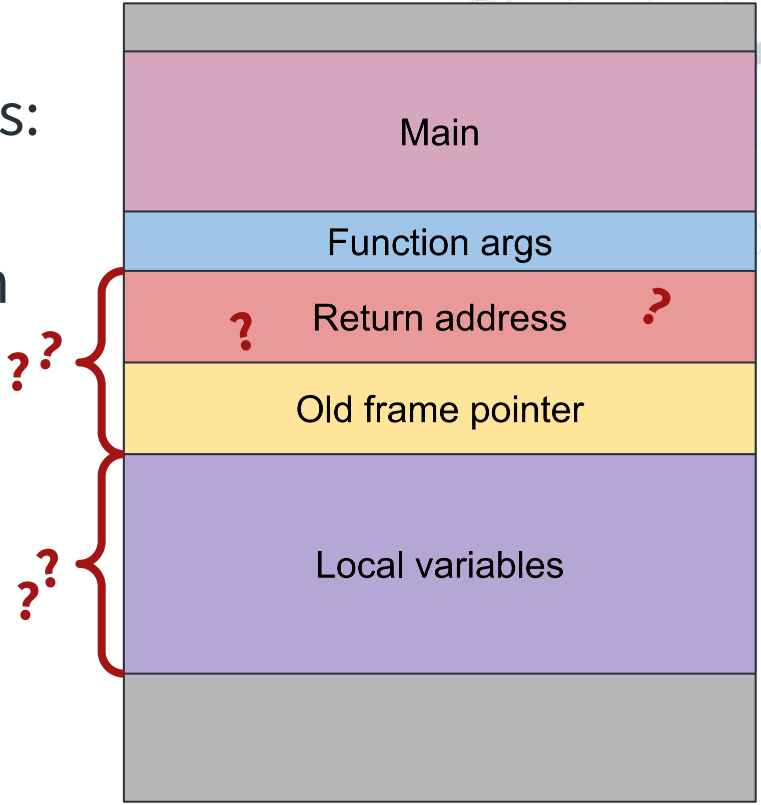
An attacker needs to know two things:

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
```



Main

Function args

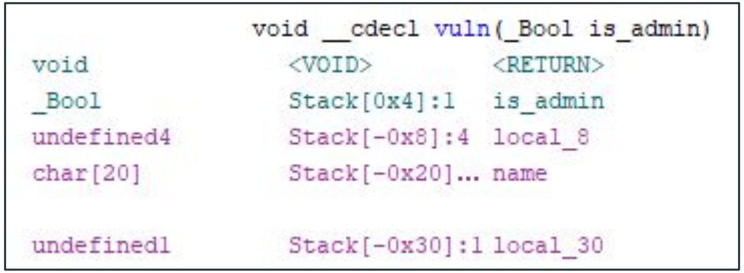
Return address

Old frame pointer

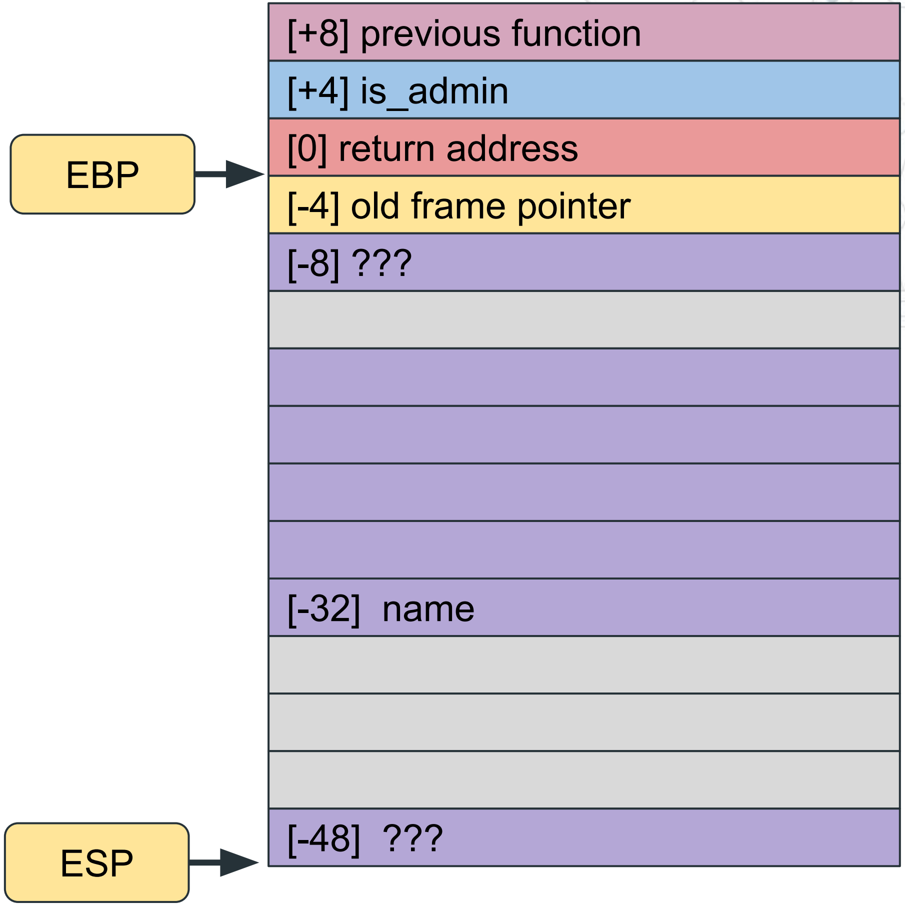
Local variables

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
```



[+8] previous function

[+4] is_admin

[0] return address

[-4] old frame pointer

[-8] ???

[-32] name

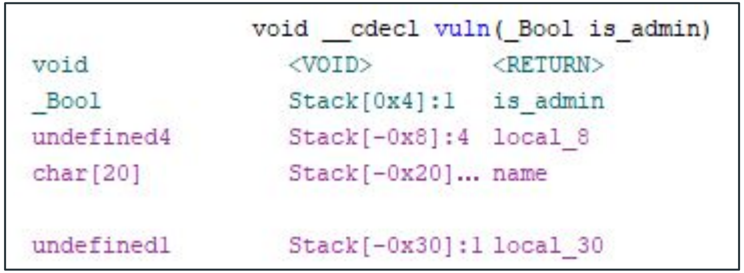
[-48] ???

EBP

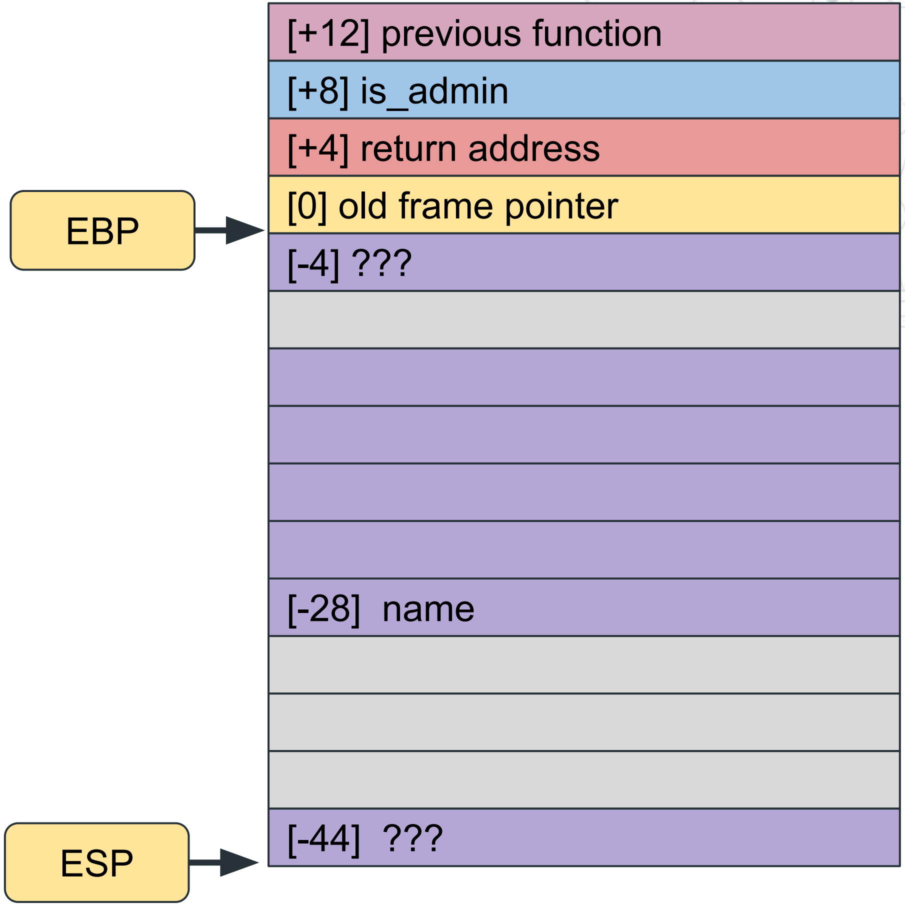
ESP

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
```



[+12] previous function

[+8] is_admin

[+4] return address

[0] old frame pointer

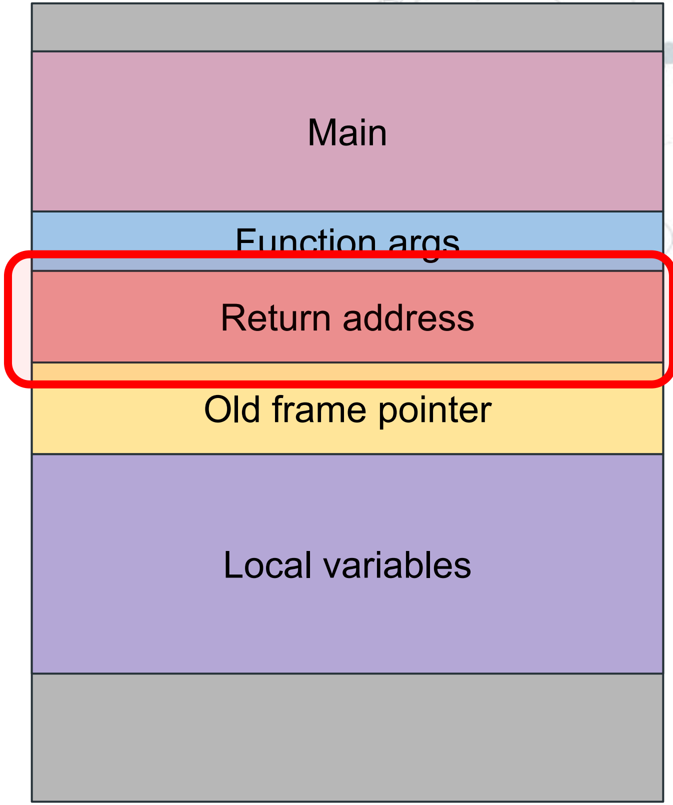
[-4] ???

[-28] name

[-44] ???

Buffer Overflows

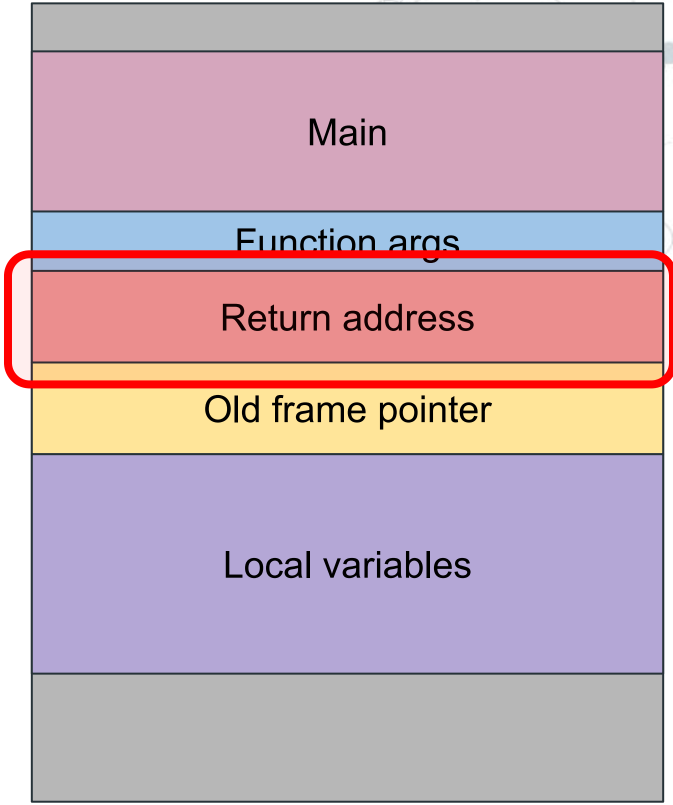
What can overwriting the return address do?



Buffer Overflows

What can overwriting the return address do?

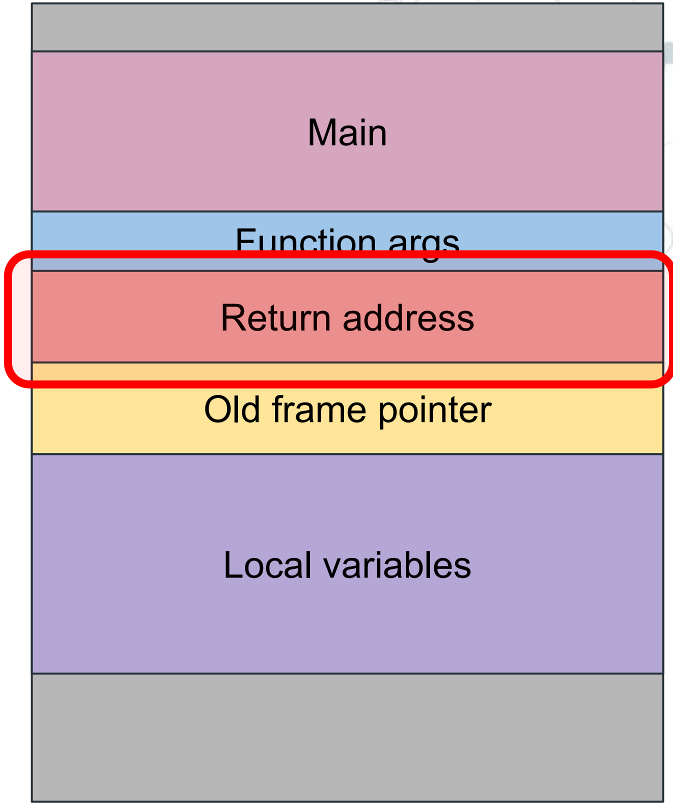
- Crash the program



Buffer Overflows

What can overwriting the return address do?

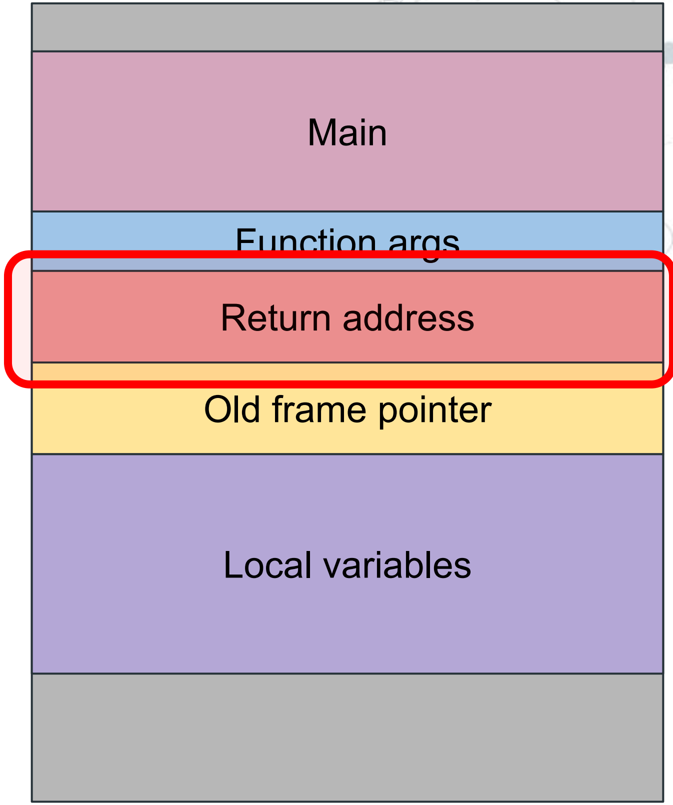
- Crash the program
- Return to the wrong function



Buffer Overflows

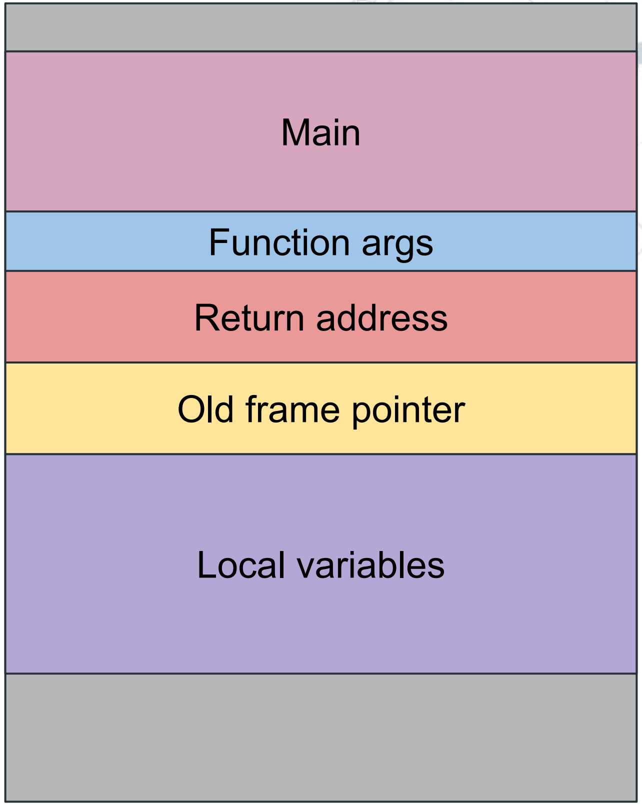
What can overwriting the return address do?

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



Buffer Overflows

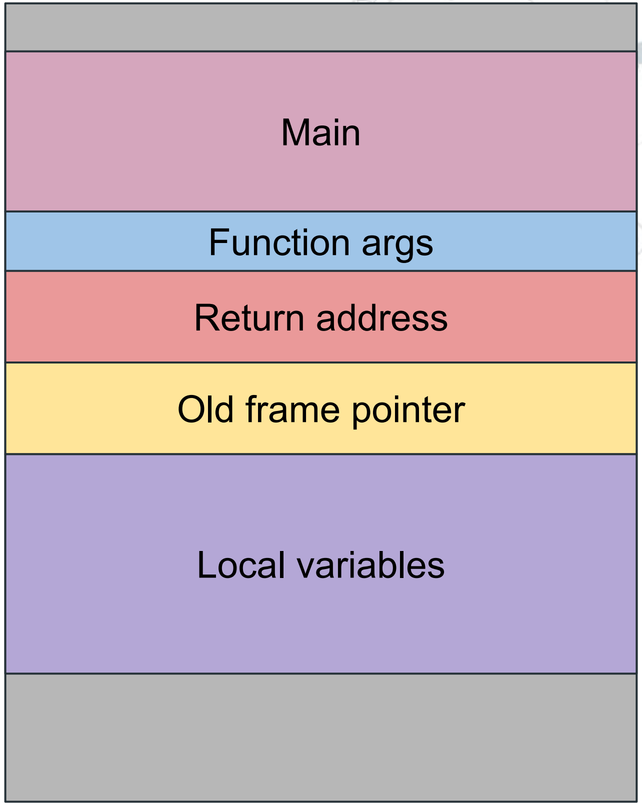
Shellcode: Attacker-supplied code they are trying to run.



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.

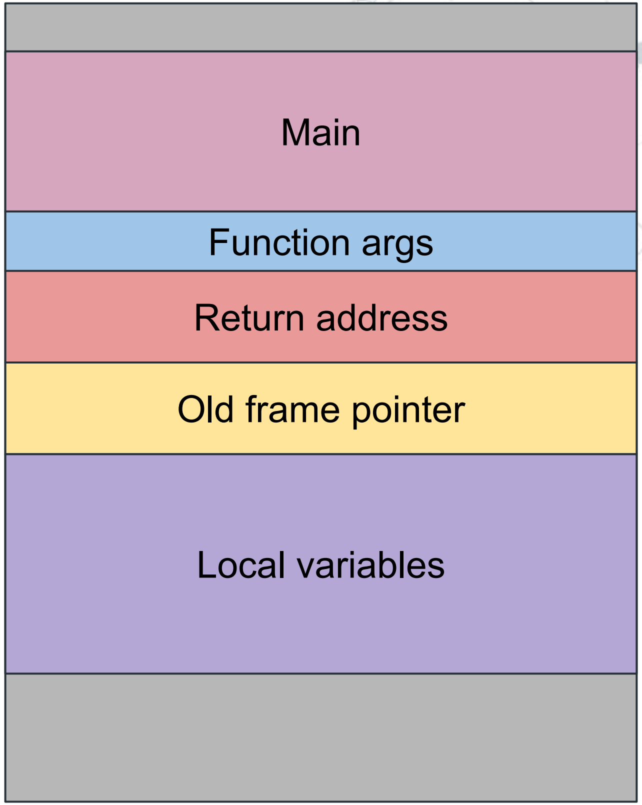


Buffer Overflows

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

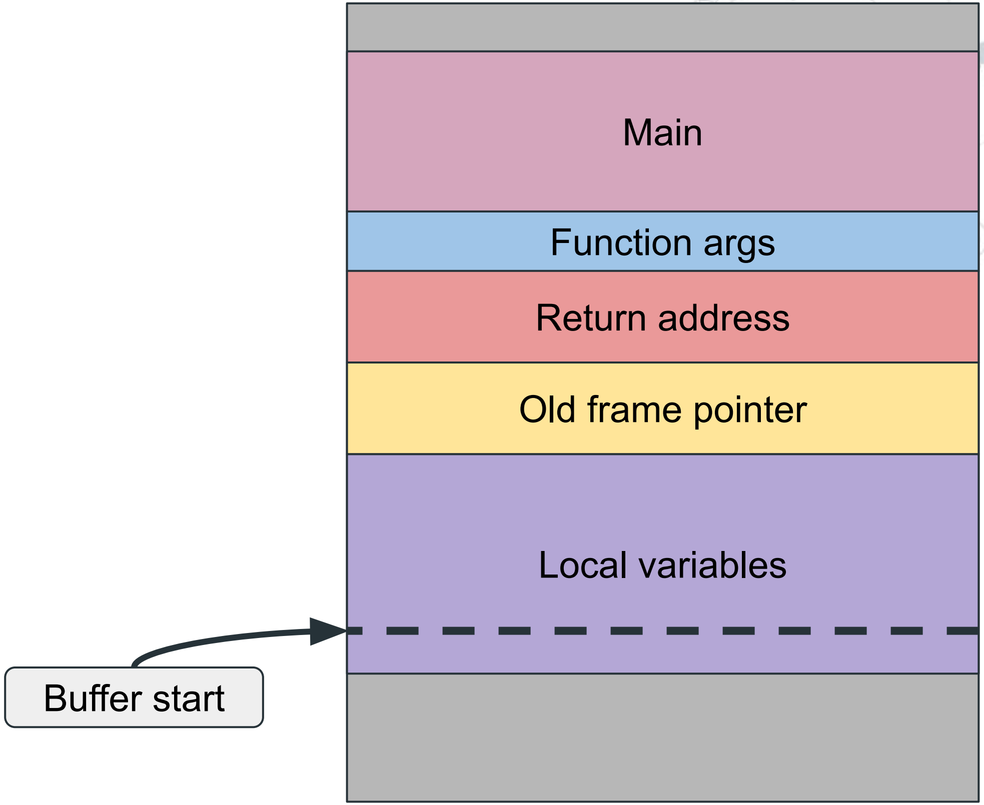
- Why call it “shellcode”? Easiest payload just opens a shell to allow for follow-up commands.

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



Buffer Overflows

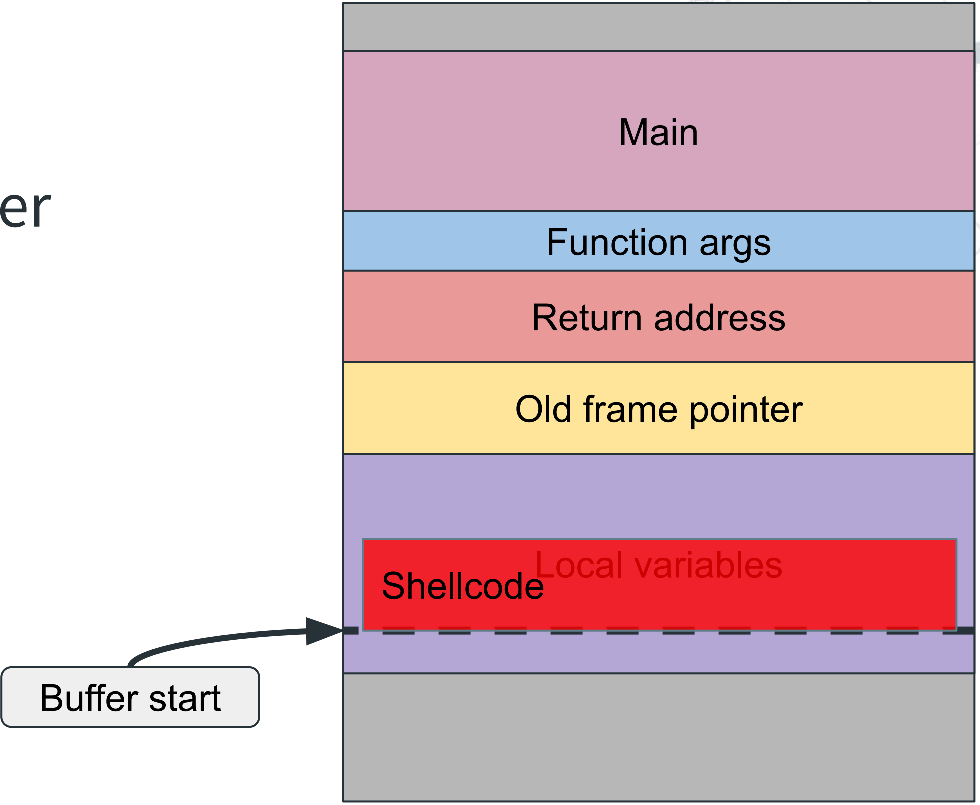
Shellcode injection:



Buffer Overflows

Shellcode injection:

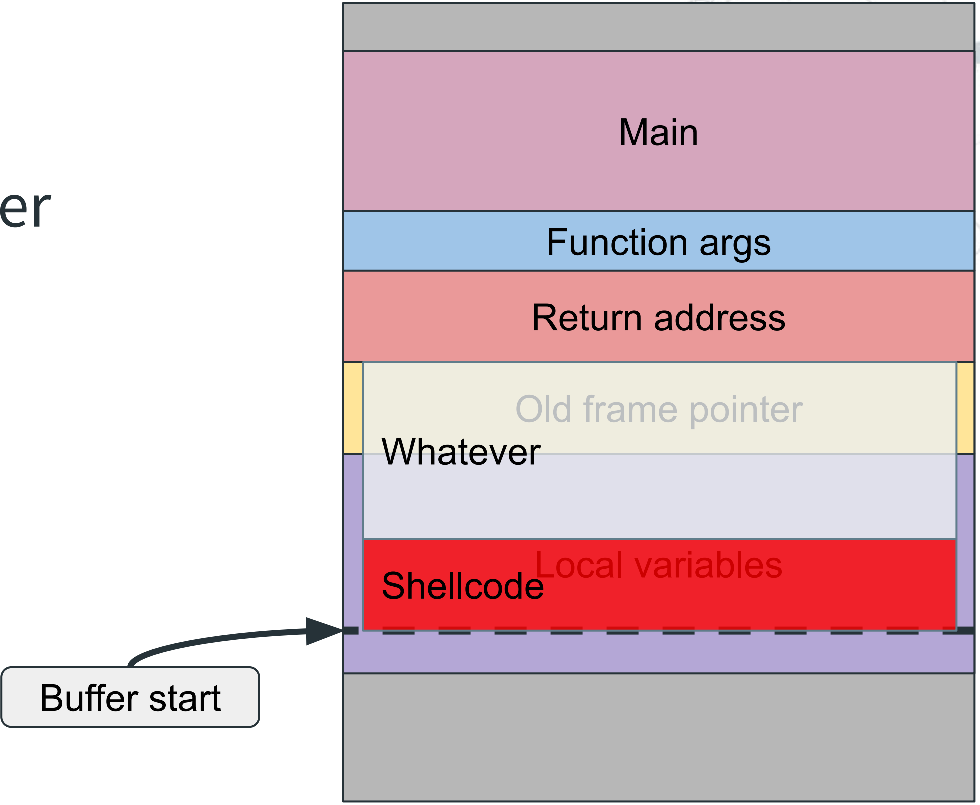
1. Code gets written to buffer



Buffer Overflows

Shellcode injection:

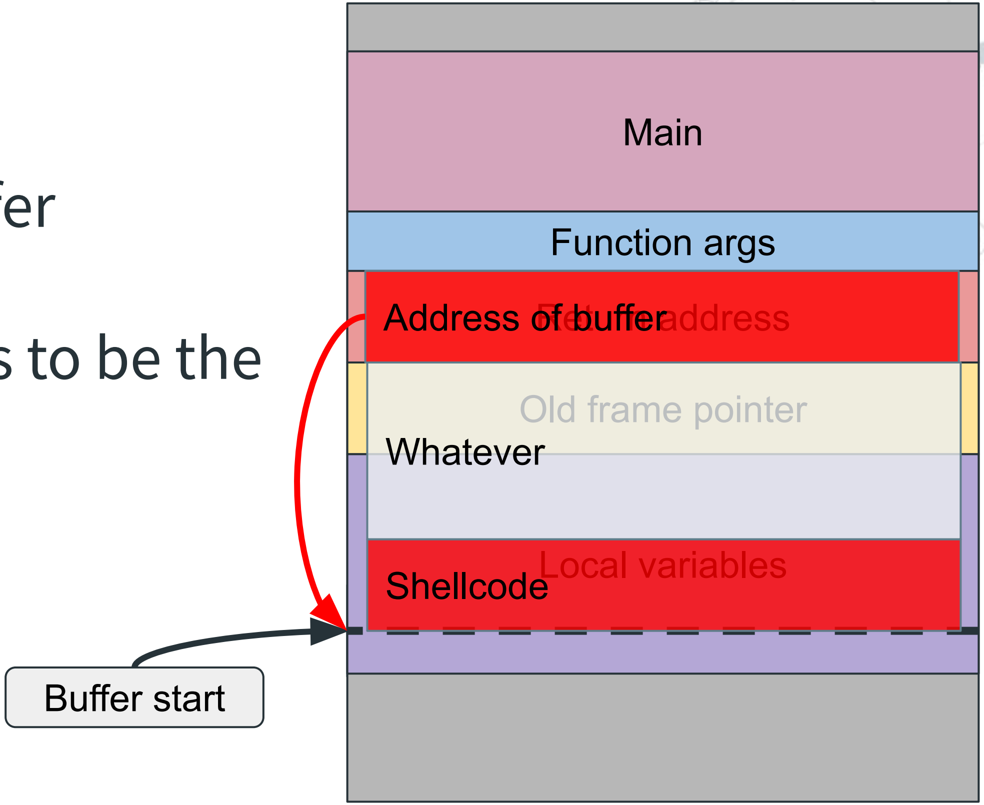
1. Code gets written to buffer
2. Rest of the buffer is filled



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

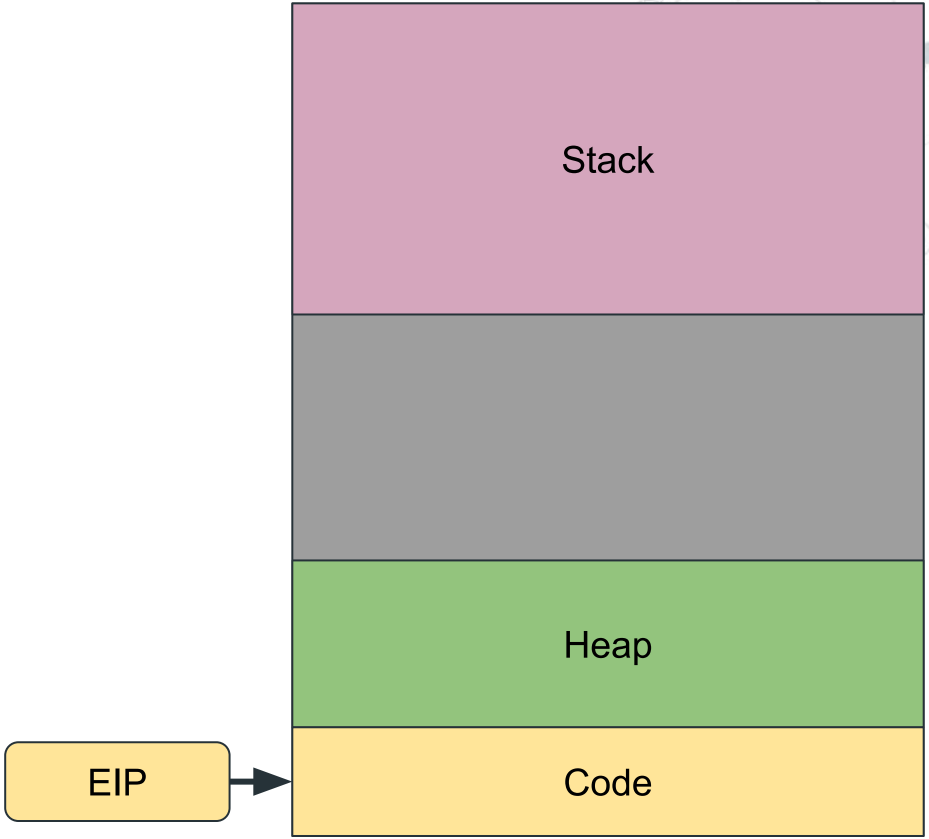


Demo!

Buffer Overflows

Note this moves the instruction register!

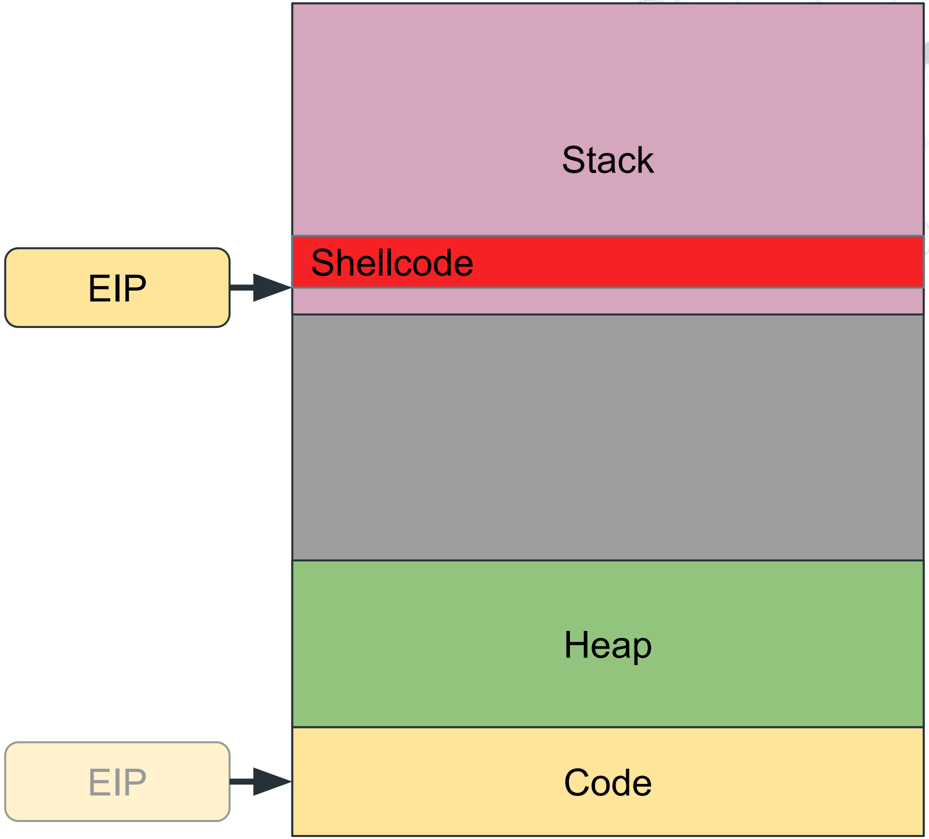
- Before attack: Only in the code section



Buffer Overflows

Note this moves the instruction register!

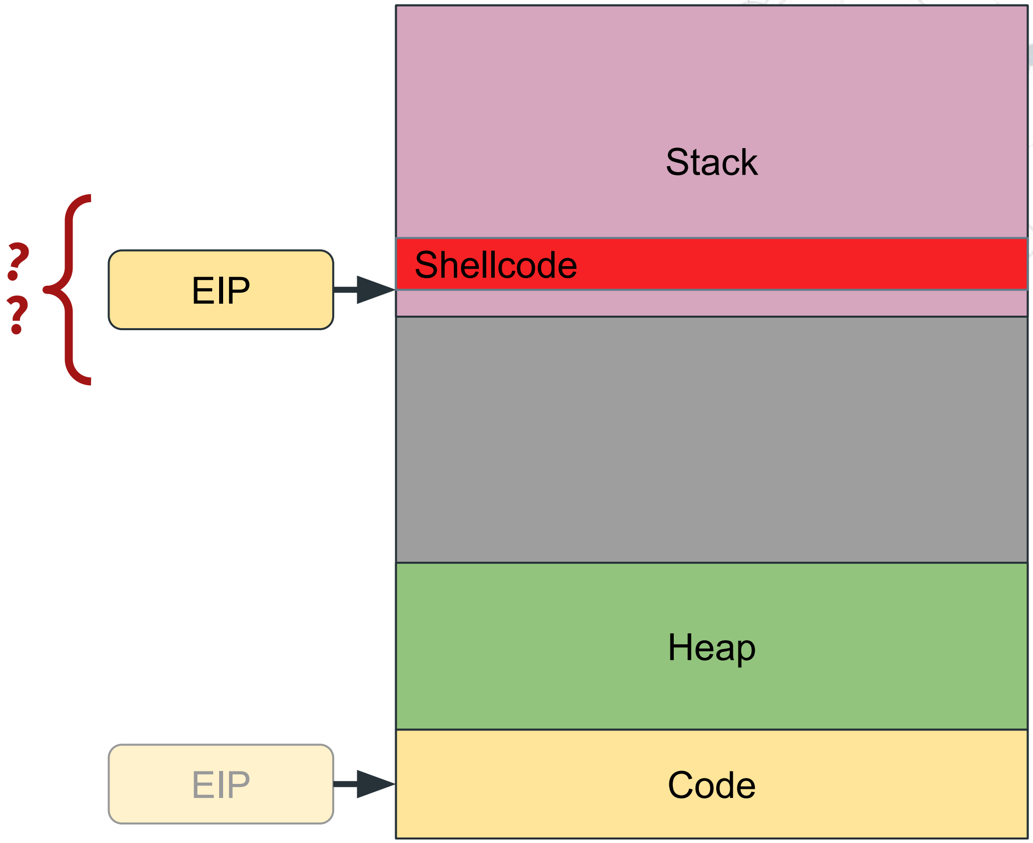
- Before attack: Only in the code section
- After attack: Goes to shellcode on the stack



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);
}
```

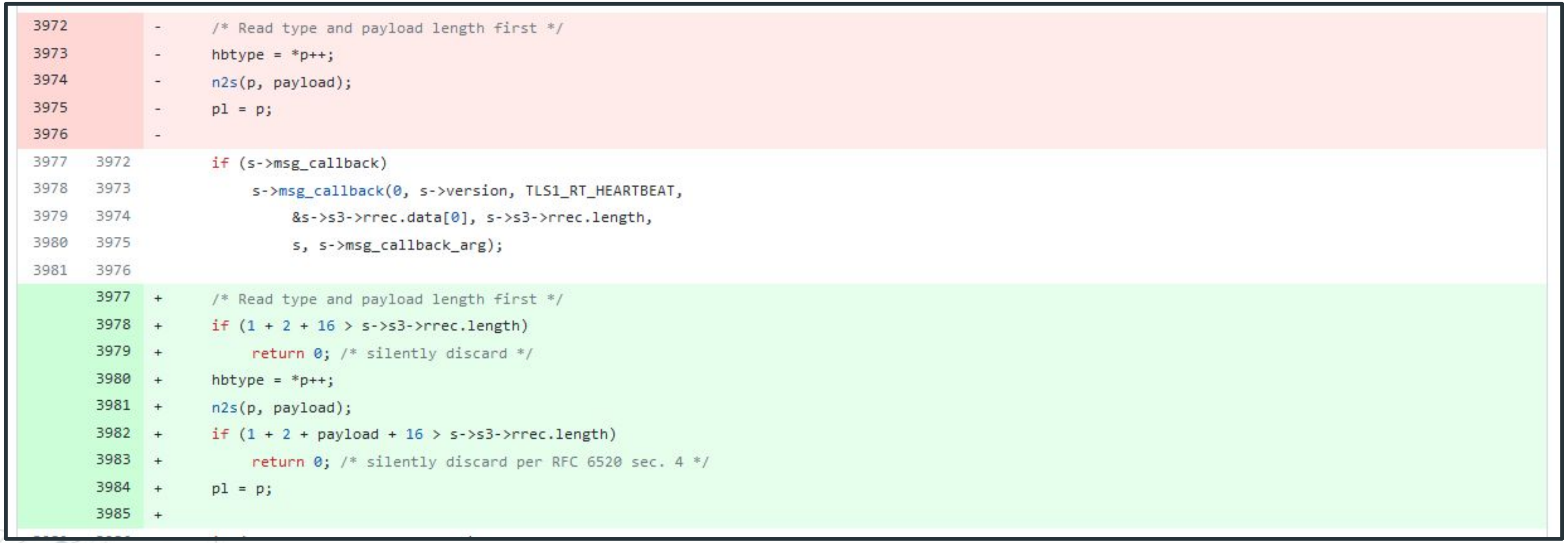
Safe functions:

```
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)



```diff
-	/* Read type and payload length first */
-	hbtype = *p++;
-	n2s(p, payload);
-	pl = p;
-
 	if (s->msg_callback)
 		s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
 			&s->s3->rrec.data[0], s->s3->rrec.length,
 			s, s->msg_callback_arg);
 
+	/* Read type and payload length first */
+	if (1 + 2 + 16 > s->s3->rrec.length)
+		return 0; /* silently discard */
+	hbtype = *p++;
+	n2s(p, payload);
+	if (1 + 2 + payload + 16 > s->s3->rrec.length)
+		return 0; /* silently discard per RFC 6520 sec. 4 */
+	pl = p;
+
```

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

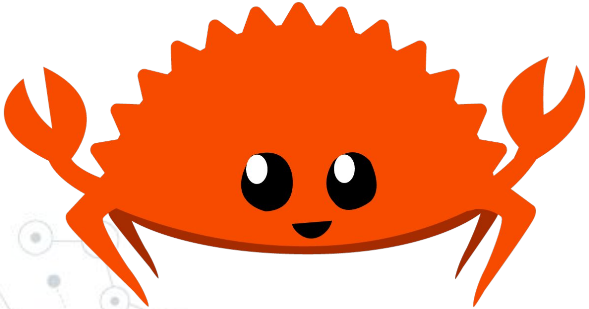
Mitigations

Maybe we could use a better language…?

Mitigations

Maybe we could use a better language…?

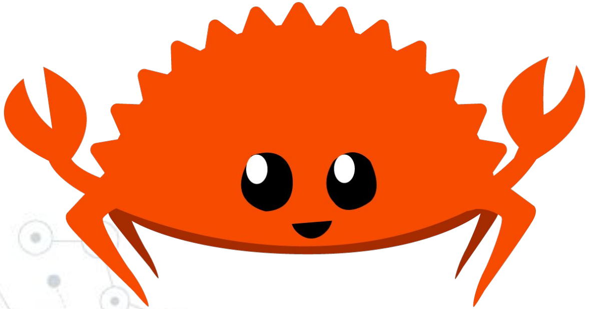
Like Rust. Rust is nice.

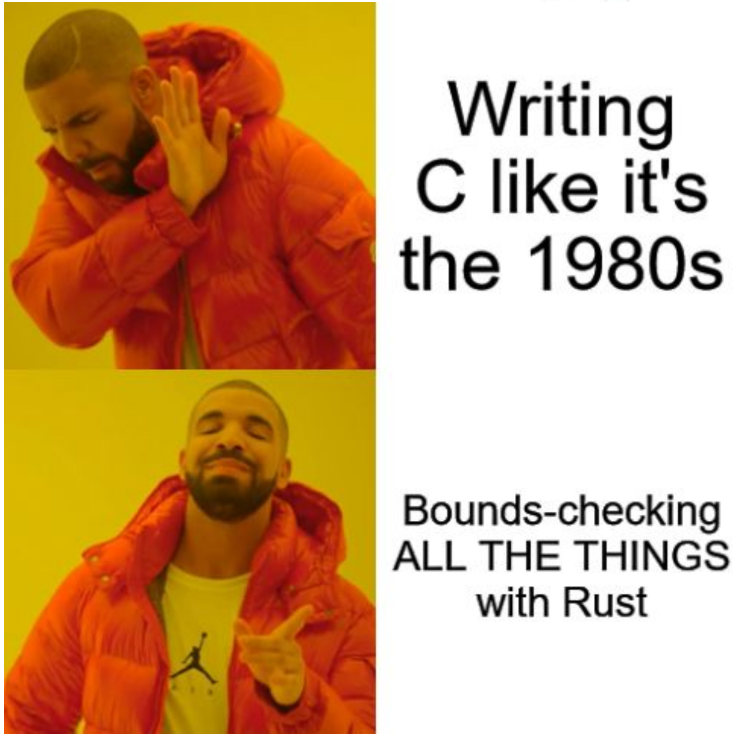


Mitigations

Maybe we could use a better language…?

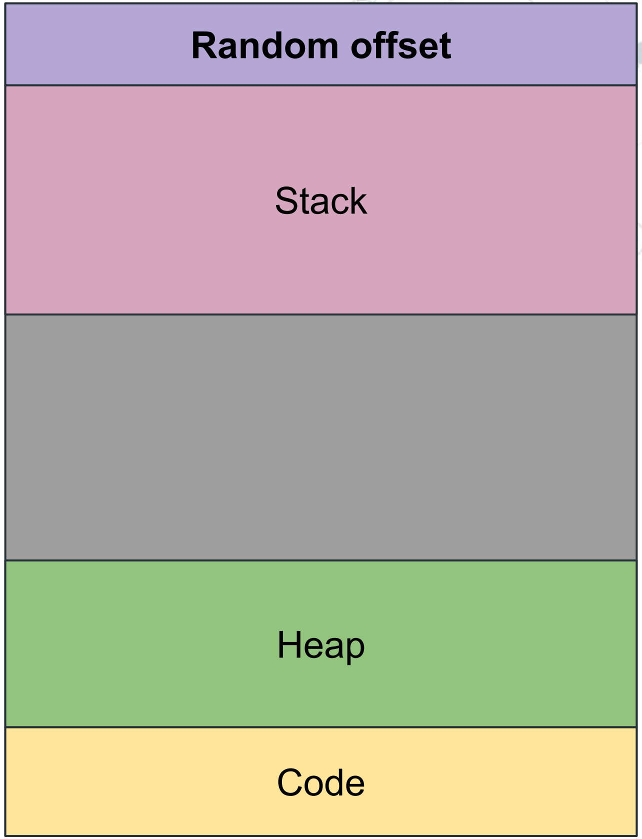
Like Rust. Rust is nice.




Writing C like it's the 1980s
Bounds-checking ALL THE THINGS with Rust

Mitigations

Option 2: Randomize Stack



Random offset

Stack

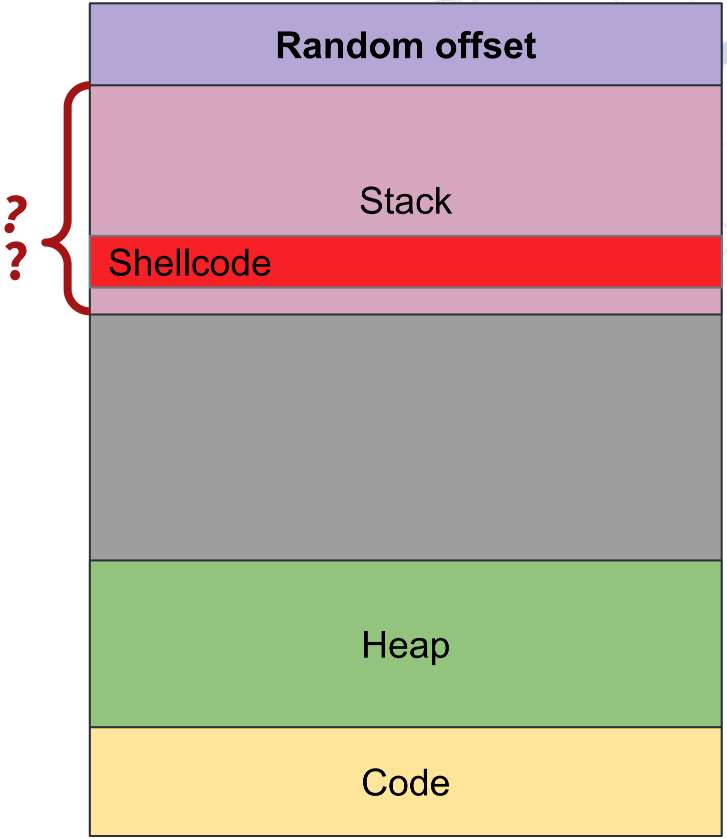
Heap

Code

Mitigations

Option 2: Randomize Stack

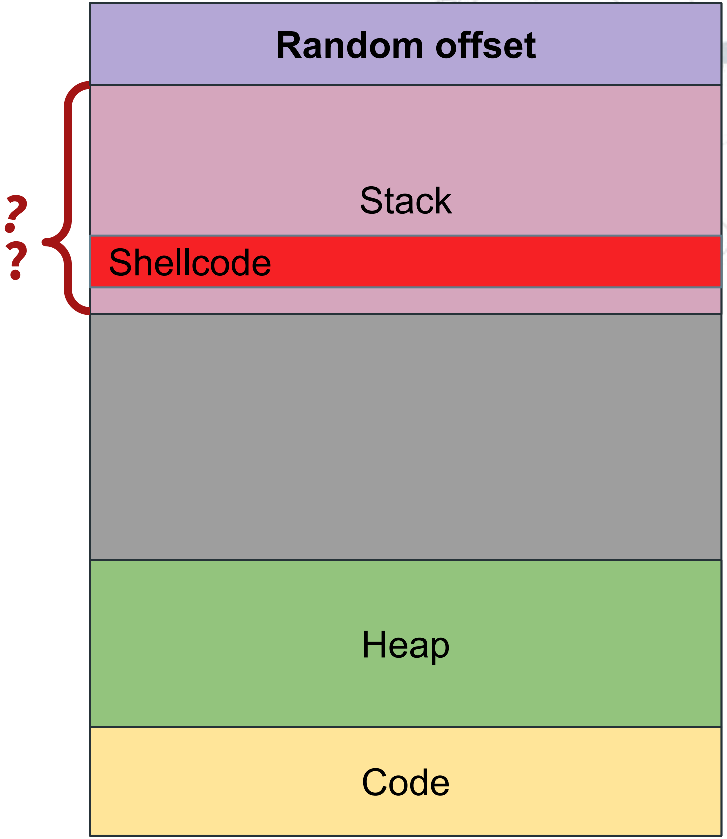
- **Pros:** Makes it harder to return to shellcode if the address is random



Mitigations

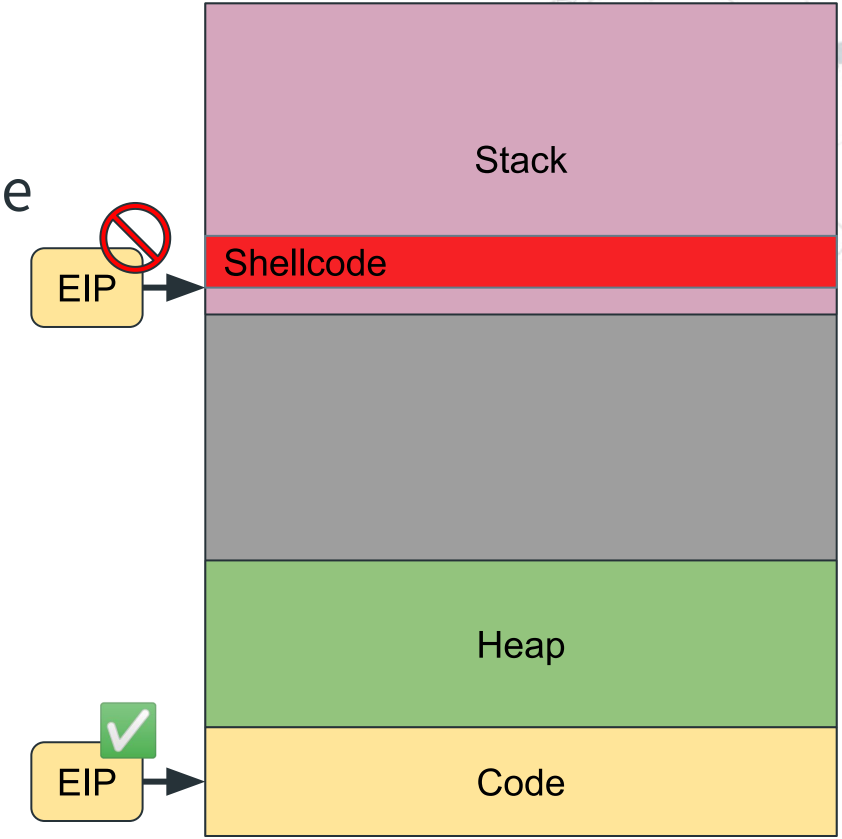
Option 2: Randomize Stack

- **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

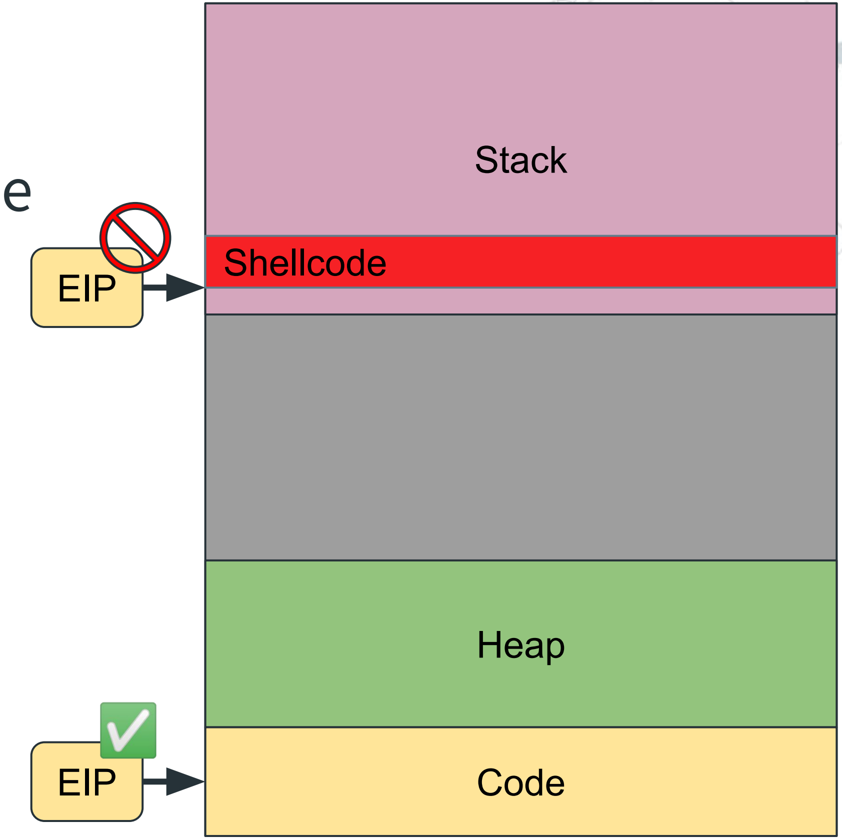
Option 3: Kill the program if the instruction pointer leaves the code section (non-executable stack)



Mitigations

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

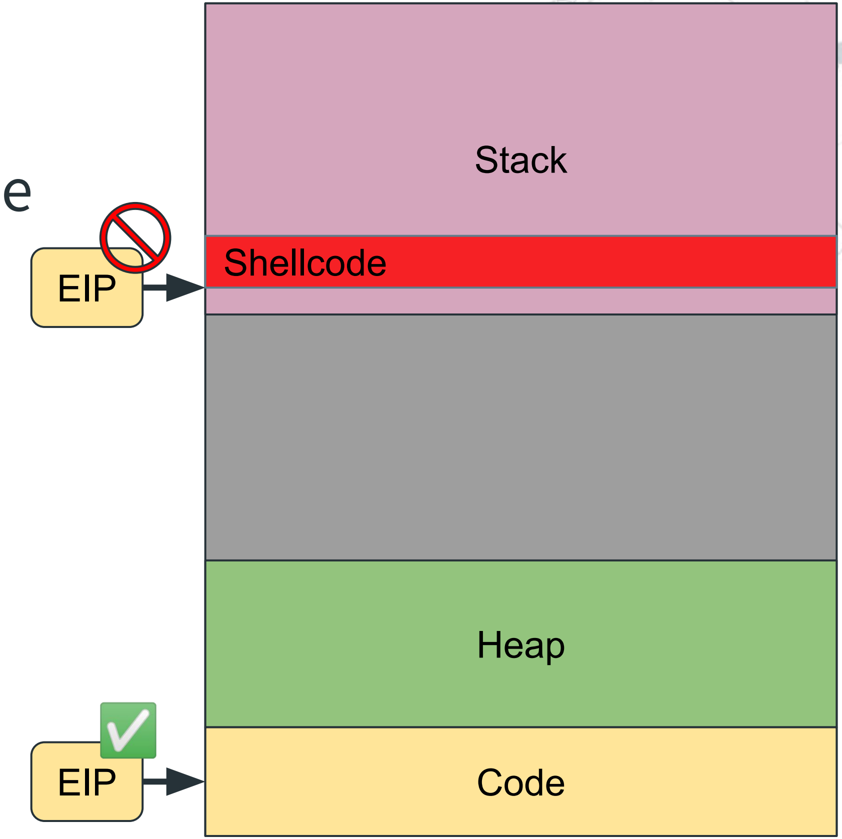
- **Pros**: Totally removes stack-based shellcode



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

- **Honeypots**: Remember to turn off droplets when finished so you do not get charged

Patch Notes

- **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)

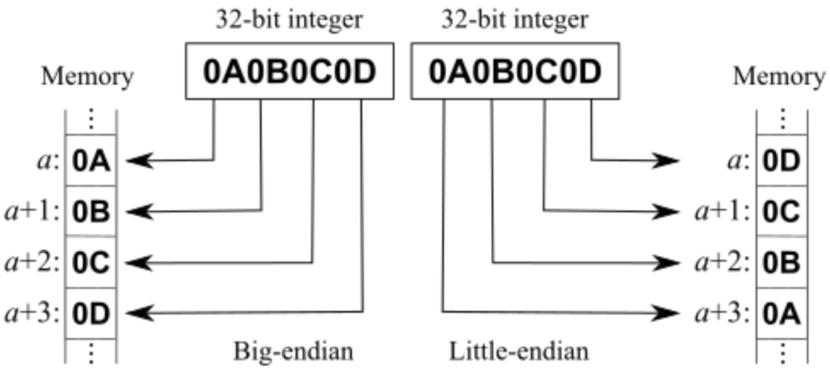
Last Episode

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)

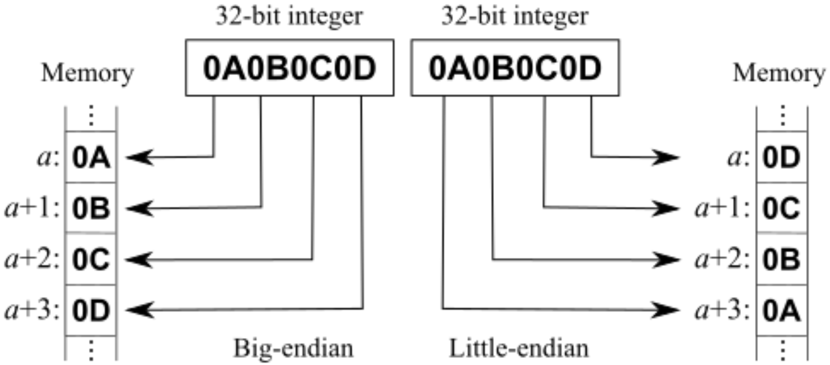


Last Episode

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

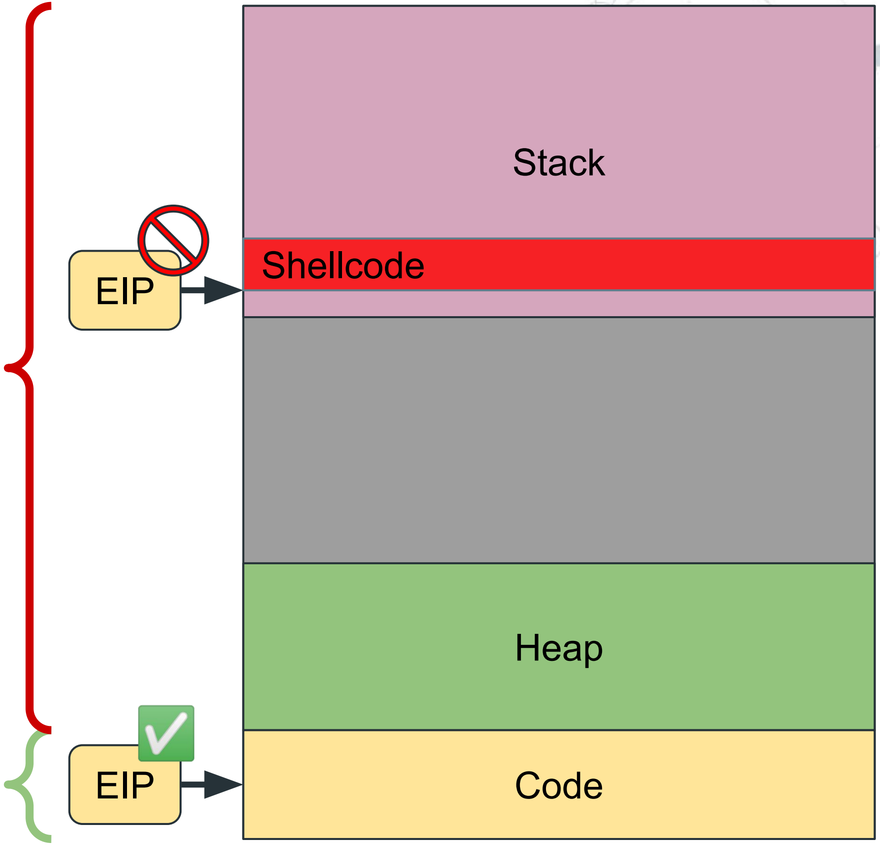
A: Endianness! Bytes are written up the stack from low to high addresses, but read from high to low (little-endian)

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



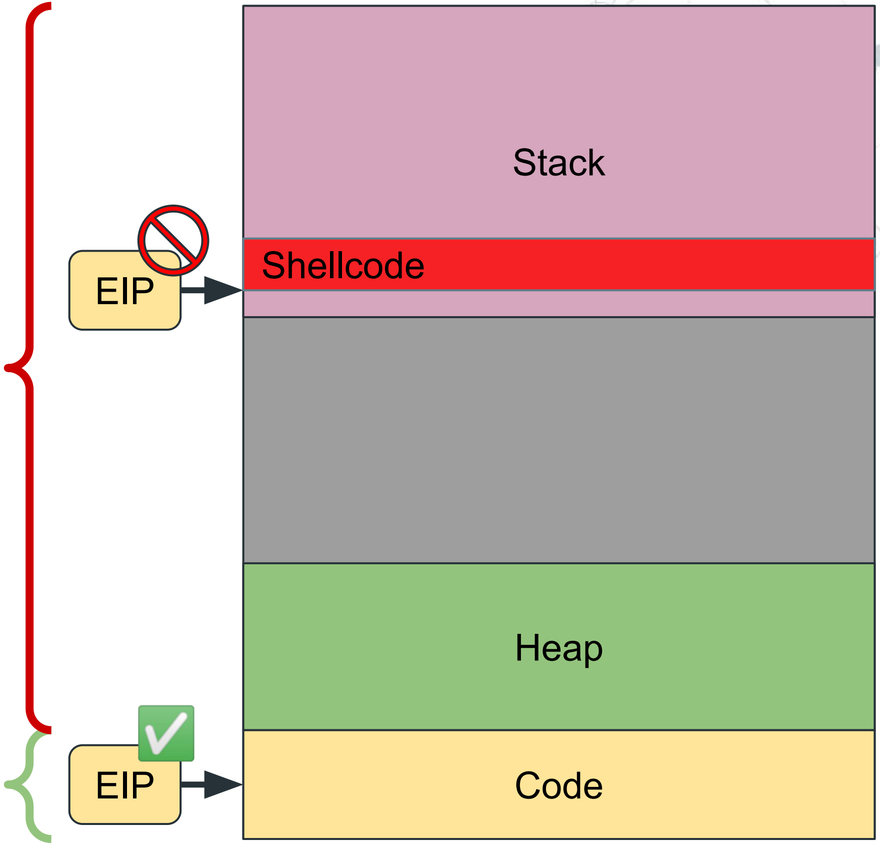
Last Episode

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



Last Episode

Q: Is this used everywhere?

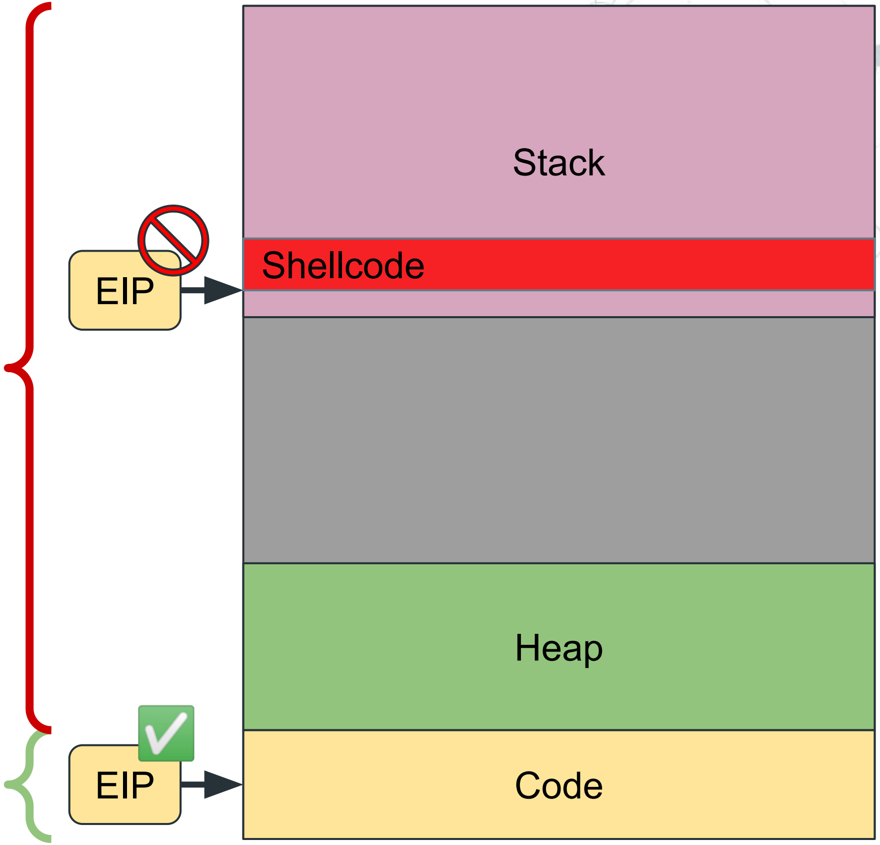


Last Episode

Q: Is this used everywhere?

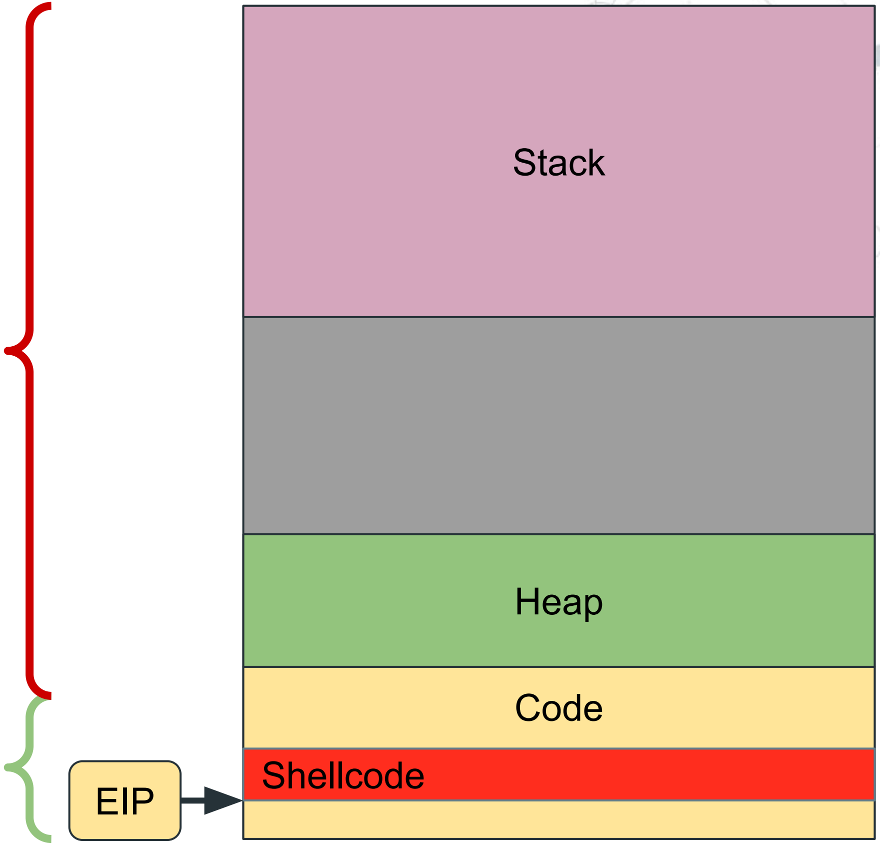
A: Not exactly:

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



Last Episode

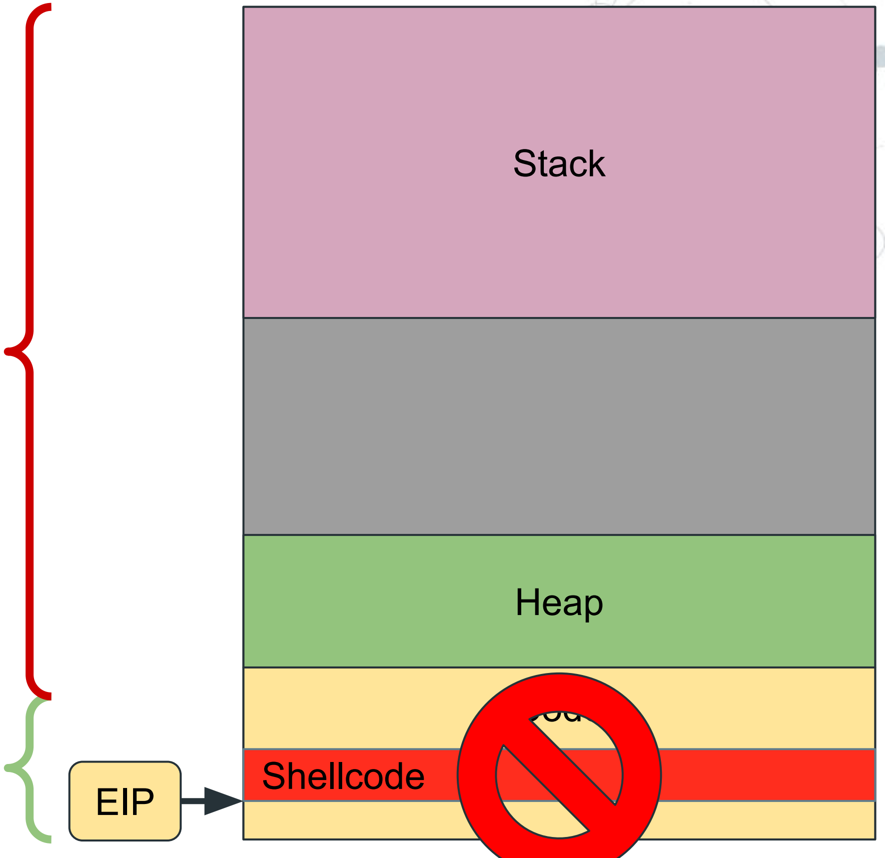
Q: Can they inject shellcode into the executable section?



Last Episode

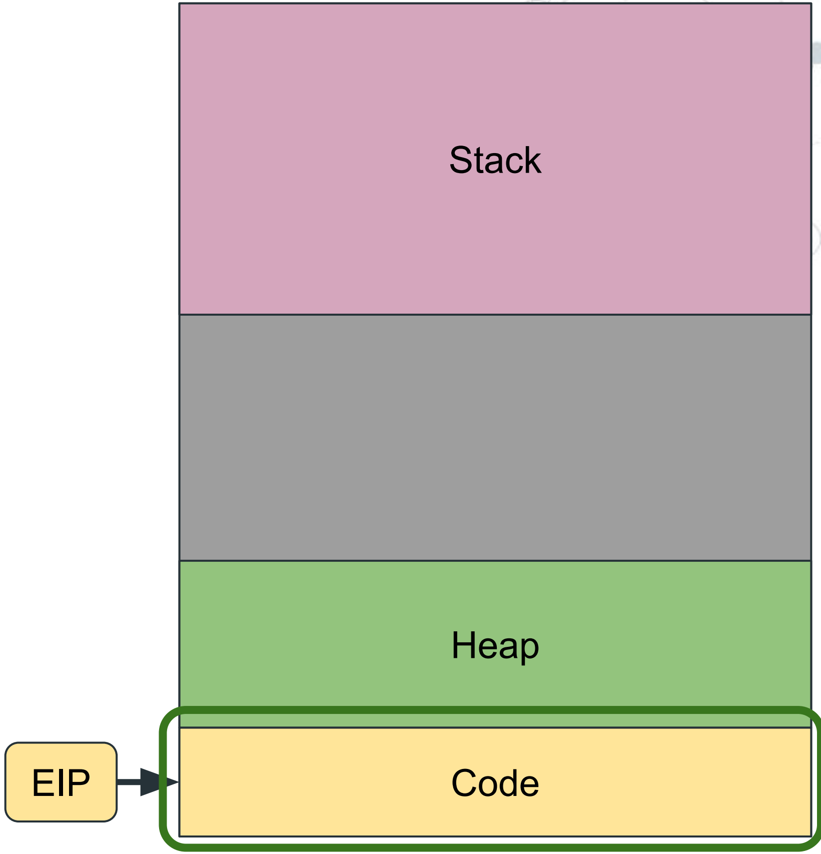
Q: Can they inject shellcode into the executable section?

A: No, all user data is written to the stack or heap!



Return-Oriented Programming

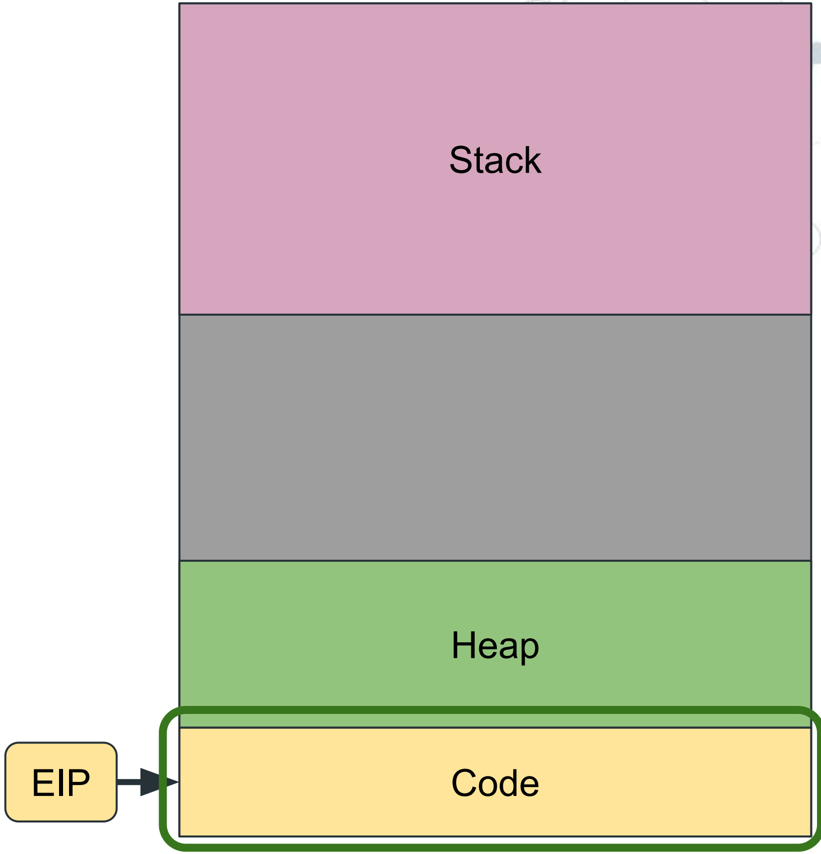
Return-Oriented Programming (ROP): Return to the existing code in unintended ways



Return-Oriented Programming

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

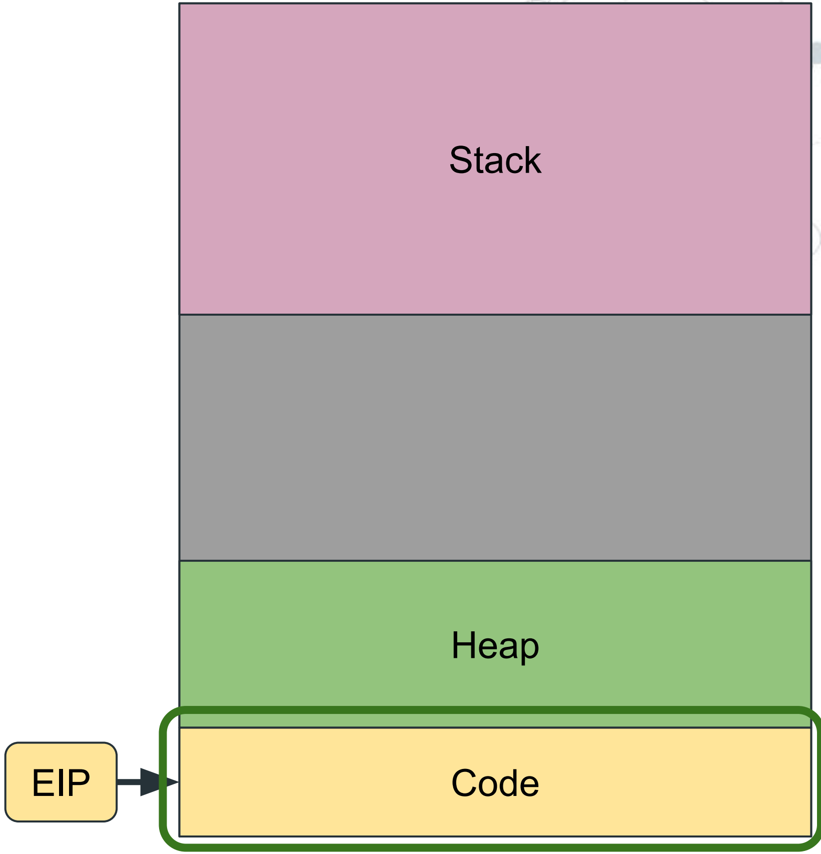
- Always running data from the Code section



Return-Oriented Programming

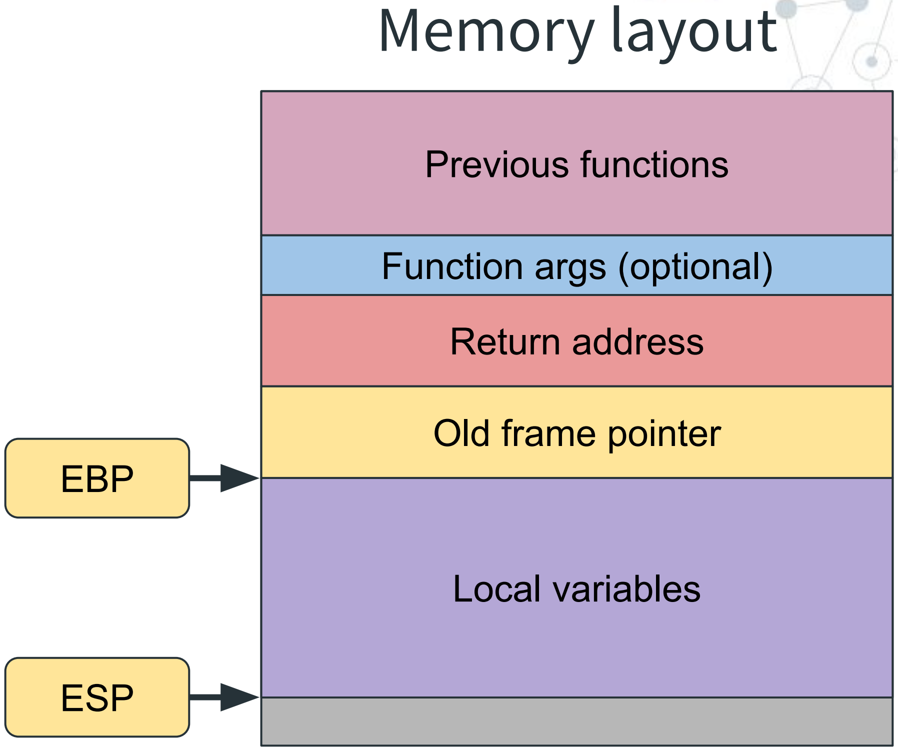
Return-To-LibC: Call a library function which is not normally called

- We did something similar yesterday



Return-Oriented Programming

Memory layout



Memory layout
Previous functions
Function args (optional)
Return address
Old frame pointer
Local variables
EBP
ESP