# 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 && 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.

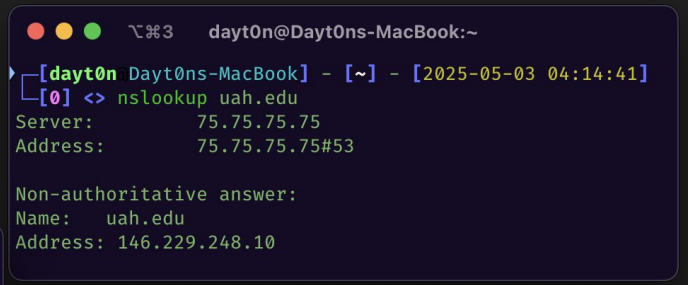
# Challenge Types

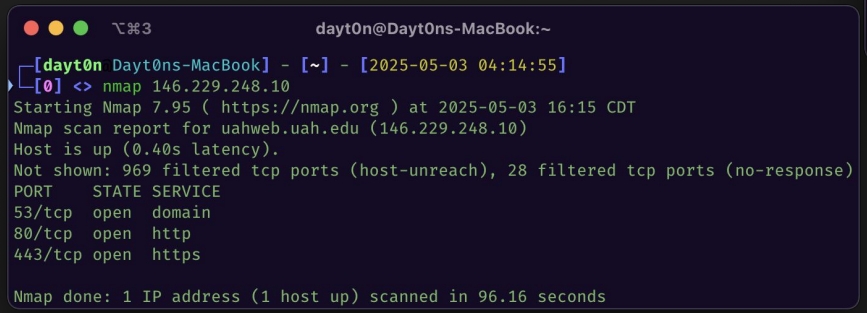
# Web exploitation

- 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



# 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

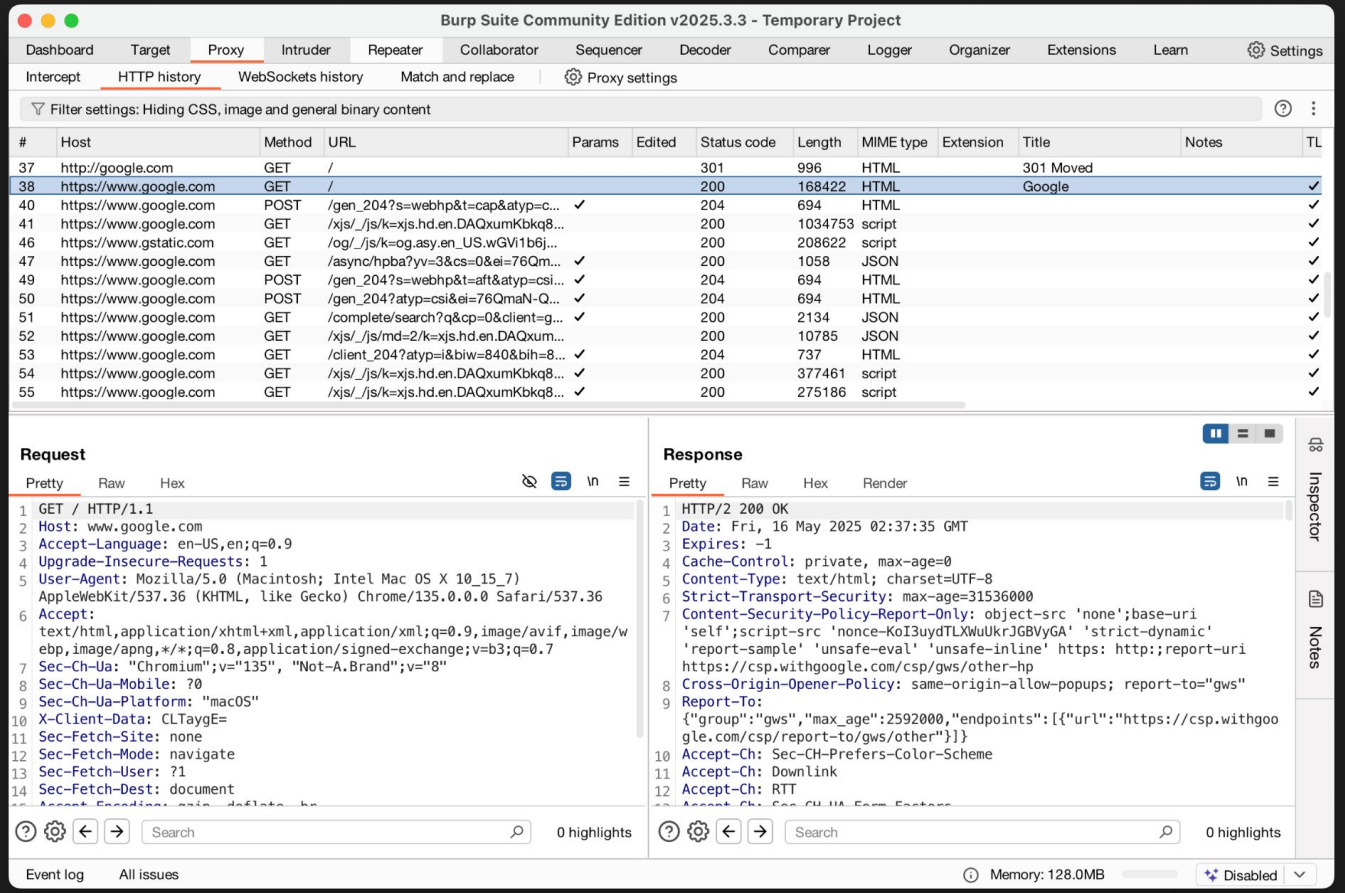
# Common web exploits

- 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/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



# 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

Source code

```
main.c:

#include <stdio.h>
int main() {
    puts("hello world!");
    return 0;
}
```

Assembly

```
.LC0:
        .string "hello world!"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        pop     rbp
        ret
```

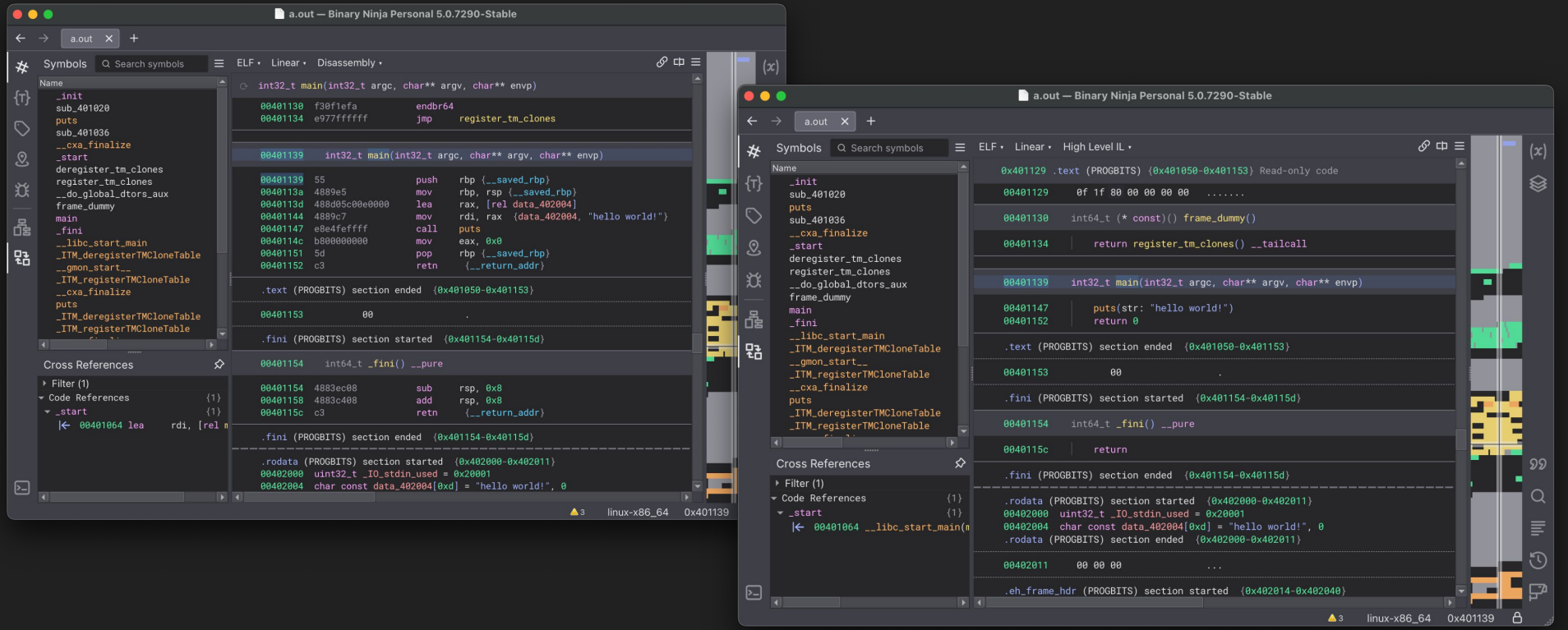
Machine code

```
7f45 4c46 0201 0100 0000 0000 0000 0000
0300 3e00 0100 0000 5010 0000 0000 0000
4000 0000 0000 0000 9036 0000 0000 0000
0000 0000 4000 3800 0d00 4000 1f00 1e00
0600 0000 0400 0000 4000 0000 0000 0000
4000 0000 0000 0000 4000 0000 0000 0000
d802 0000 0000 0000 d802 0000 0000 0000
0800 0000 0000 0000 0300 0000 0400 0000
```

# 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



# 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


⌥⌘1
docker run --rm -it -v `pwd`:/app pwndbg gdb /app/heapedit
Breakpoint 1, 0x000000000040080b in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
RAX 0x400807 (main) ◂— push rbp
RBX 0
RCX 0x400a80 (__libc_csu_init) ◂— push r15
RDX 0x7fffffffece8 —▸ 0x7fffffffeed3 ◂— 'HOSTNAME=7cbad70c0177'
RDI 1
RSI 0x7fffffffecd8 —▸ 0x7fffffffeec5 ◂— '/app/heapedit'
R8 0x7ffff7e1bf10 (initial+16) ◂— 4
R9 0x7ffff7fc9040 (_dl_fini) ◂— endbr64
R10 0x7ffff7fc3908 ◂— 0xd00120000000e
R11 0x7ffff7fde660 (_dl_audit_preinit) ◂— endbr64
R12 0x7fffffffecd8 —▸ 0x7fffffffeec5 ◂— '/app/heapedit'
R13 0x400807 (main) ◂— push rbp
R14 0
R15 0x7ffff7ffd040 (_rtld_global) —▸ 0x7ffff7ffe2e0 ◂— 0
RBP 0x7fffffffebc0 ◂— 1
RSP 0x7fffffffebc0 ◂— 1
RIP 0x40080b (main+4) ◂— sub rsp, 0xc0
[ DISASM / x86-64 / set emulate on ]
► 0x40080b <main+4> sub rsp, 0xc0 RSP => 0x7fffffffeb00 (0x7fffffffebc0 - 0xc0)
0x400812 <main+11> mov dword ptr [rbp - 0xb4], edi [0x7fffffffeb0c] <= 1
0x400818 <main+17> mov qword ptr [rbp - 0xc0], rsi [0x7fffffffeb00] <= 0x7fffffffecd8 —▸ 0x7fffffffeec5 ◂— '/app/heapedit'
0x40081f <main+24> mov rax, qword ptr fs:[0x28] RAX, [0x7ffff7fb0768] => 0x2d25fd82feb64f00
0x400828 <main+33> mov qword ptr [rbp - 8], rax [0x7fffffffebb8] <= 0x2d25fd82feb64f00
0x40082c <main+37> xor eax, eax EAX => 0
0x40082e <main+39> mov rax, qword ptr [rip + 0x200843] RAX, [stdout@@GLIBC_2.2.5] => 0x7ffff7e1b780 (_IO_2_1_stdout_) ◂— 0xfbad2084
0x400835 <main+46> mov esi, 0 ESI => 0
0x40083a <main+51> mov rdi, rax RDI => 0x7ffff7e1b780 (_IO_2_1_stdout_) ◂— 0xfbad2084
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 0x7fffffffebc0 ◂— 1
01:0008 +008 0x7fffffffebc8 —▸ 0x7ffff7c29d90 (__libc_start_call_main+128) ◂— mov edi, eax
02:0010 +010 0x7fffffffebd0 ◂— 0
03:0018 +018 0x7fffffffebd8 —▸ 0x400807 (main) ◂— push rbp
04:0020 +020 0x7fffffffebe0 ◂— 0x1ffffecc0
05:0028 +028 0x7fffffffebe8 —▸ 0x7fffffffecd8 —▸ 0x7fffffffeec5 ◂— '/app/heapedit'
06:0030 +030 0x7fffffffebf0 ◂— 0
07:0038 +038 0x7fffffffebf8 ◂— 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 challs

- 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

# Aside: SMT Solvers

- Remember these?

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

- y > 6
- y < 100
- y * 2 < 32
- y + 5 < 12

# 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.

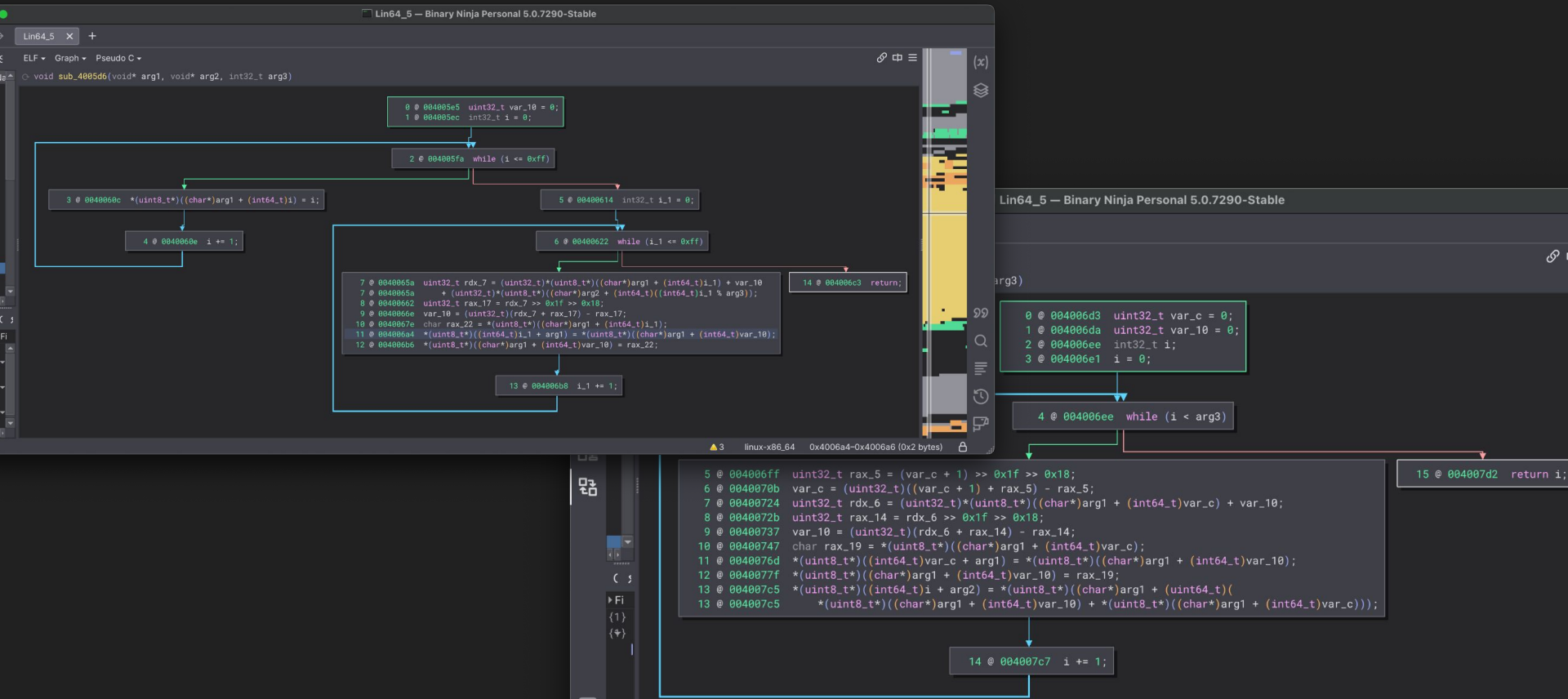
# Aside: Symbolic Execution

- 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
Lin64_5
ELF
Graph
Pseudo C
void sub_4005d6(void* arg1, void* arg2, int32_t arg3)
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 @ 0040065a uint32_t rdx_7 = (uint32_t)*(uint8_t*)((char*)arg1 + (int64_t)i_1) + var_10
7 @ 0040065a + (uint32_t)*(uint8_t*)((char*)arg2 + (int64_t)((int64_t)i_1 % arg3));
8 @ 00400662 uint32_t rax_17 = rdx_7 >> 0x1f >> 0x18;
9 @ 0040066e var_10 = (uint32_t)(rdx_7 + rax_17) - rax_17;
10 @ 0040067e char rax_22 = *(uint8_t*)((char*)arg1 + (int64_t)i_1);
11 @ 004006a4 *(uint8_t*)((int64_t)i_1 + arg1) = *(uint8_t*)((char*)arg1 + (int64_t)var_10);
12 @ 004006b6 *(uint8_t*)((char*)arg1 + (int64_t)var_10) = rax_22;
13 @ 004006b8 i_1 += 1;
14 @ 004006c3 return;
3
linux-x86_64
0x4006a4-0x4006a6 (0x2 bytes)
Lin64_5 — Binary Ninja Personal 5.0.7290-Stable
arg3)
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)
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 + (uint64_t)(
13 @ 004007c5 *(uint8_t*)((char*)arg1 + (int64_t)var_10) + *(uint8_t*)((char*)arg1 + (int64_t)var_c)));
14 @ 004007c7 i += 1;
15 @ 004007d2 return i;


```
00400a62  bf220b4000        mov     edi, 0x400b22  {"[+] You achieved level 5!\r\
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.
00400a7d  b800000000        mov     eax, 0x0
00400a82  e809faffff        call    printf
00400a87  b801000000        mov     eax, 0x1
```

```
⌥⌘2        dayt0n@Dayt0ns-MacBook:~/s
[dayt0n@Dayt0ns-MacBook] - [~/sharedKali/us
[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?

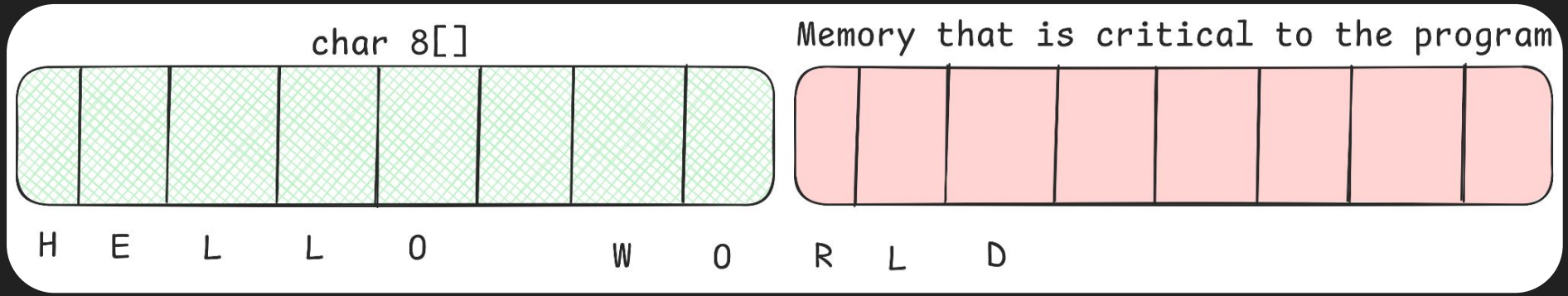
Yeah, pretty much.

# 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

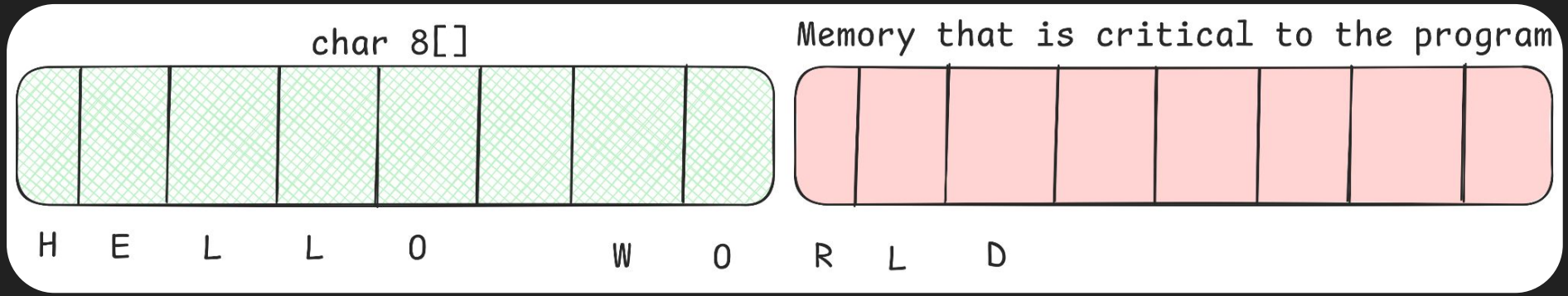
# Types of vulnerabilities

- Overflows
  - Stack-based
  - Heap-based



# Types of vulnerabilities

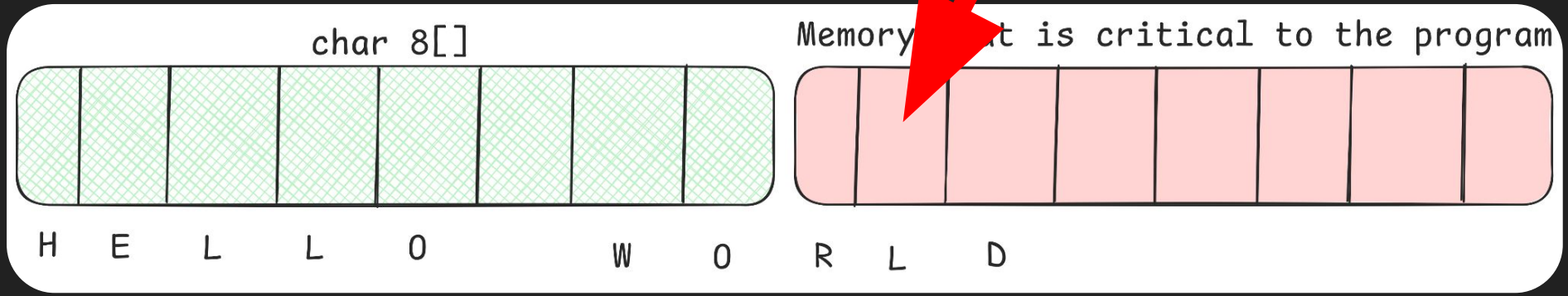
- Overflows
  - Stack-based
  - Heap-based



# Types of vulnerabilities

- Overflows
  - Stack-based
  - Heap-based

Address stating where the CPU should jump after this instruction completes

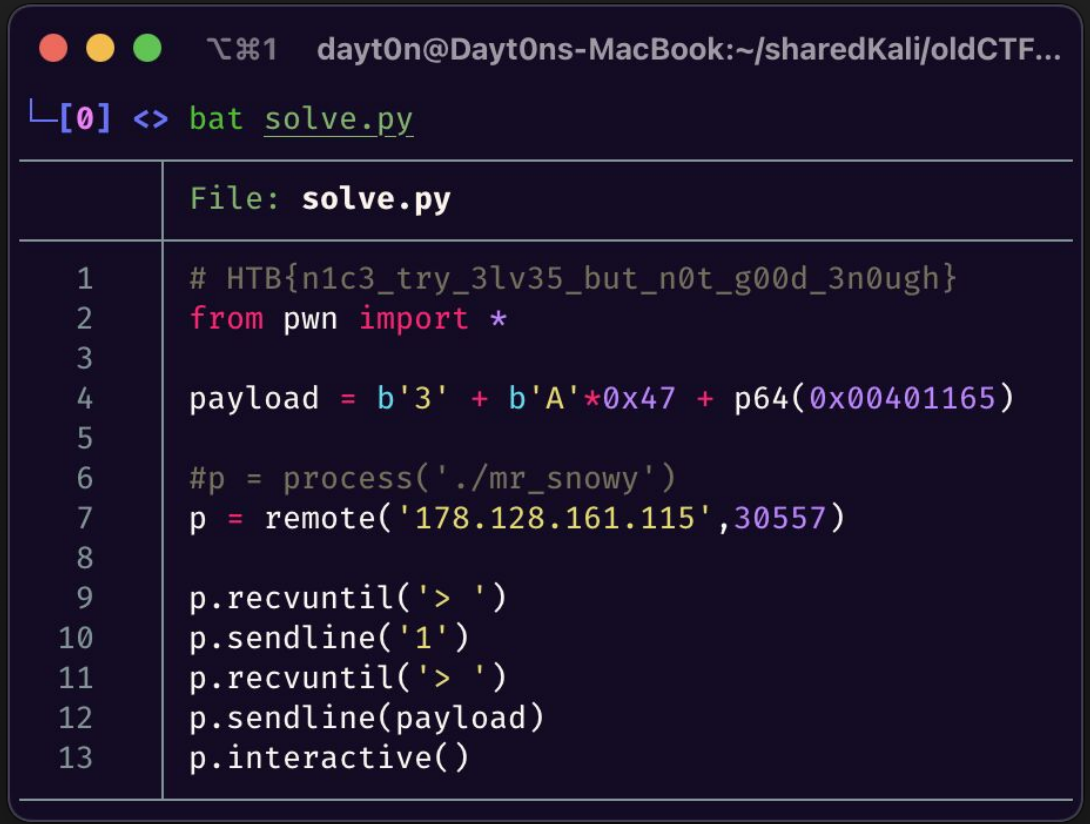


# Types of vulnerabilities

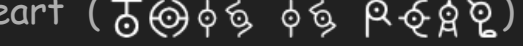
- Logic bugs
- Format string (https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf)
- Integer under/overflows
- Use-after-frees

# Pwntools

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



# Crypto(graphy)

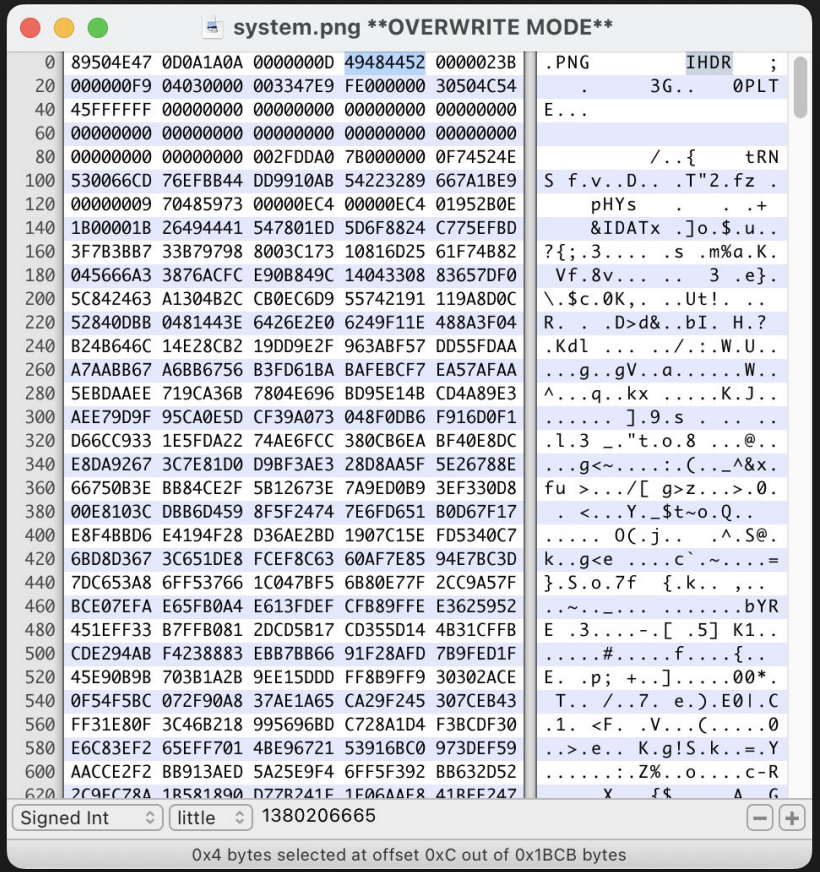
- My least-favorite category
-  ??????? what ???
- Things you'll learn:
  - Math
  - A lot more math
  - How one weak component in an algorithm can completely reveal secret data
  - The Unown Pokemon alphabet by heart (  )
- Tools
  - RsaCtfTool (https://github.com/RsaCtfTool/RsaCtfTool)
  - Sage (https://doc.sagemath.org/html/en/index.html)
  - 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)


```
⌥⌘1  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