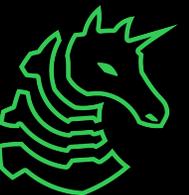FA2023 Week 06 • 2023-10-05

Reverse Engineering II

Richard, Pete, Henry

Announcements

- DRM circumvention with Ojas this sunday
- No CTF this weekend
- Fall CTF solutions will be released soon™
- Fuzzing team meeting after this at 8PM
  - Weekly time moved to Thursday
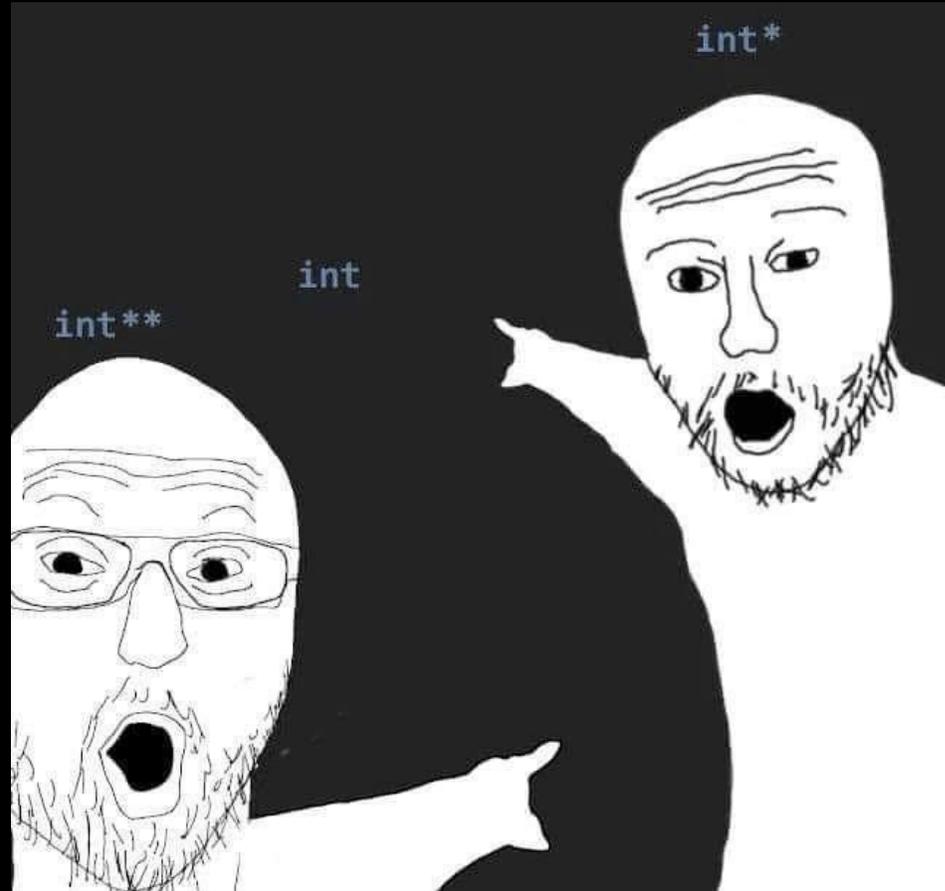
sigpwny{nsa_backdoor?}

# Table of Contents

- Leaderboard / helper callout
- RE
    - Recap: What is reverse engineering?
    - Recap: Assembly
    - Compilation / Decompilation
    - Static analysis & optimization
- Ghidra
    - Demo
- GDB
    - Demo

If you haven't installed Ghidra yet, start downloading it through the slides here!
sigpwny.com/rev_setup23

Top 10 - differences from Week 4
Scoreboard

1	ronanboyarski	+3k points	26410
2	NullPoExc	+3k points	24515
3	caasher	+3k points	17065
4	mgcsstywth	+8k points (up 2 places)	14650
5	CBCicada	+2k points	9320
6	EhWhoAmI	+.5k points	8645
7	aaronthewinner	+3k points (up 1 place)	7205
8	ilegosmaster	+4k points (NEW!)	6660
9	drizzle	+1.5k points	6175
10	SHAD0WV1RUS		5970
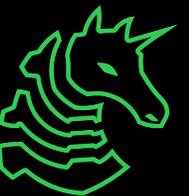
Want to be a helper?

Congratulate yourself - you made it to week 6 of meetings 😎😎😎😎

SIGPwny has a flipped leadership model - you get *invited* to become a helper

Some things we look for

- You frequently attend meetings and are actively engaged with the meeting content
- You interact with other club members
- You have a learning/teaching-focused mindset

You demonstrate an interest in improving the club. This can be shown in various ways, such as contributing to **ongoing projects**, sharing your cybersecurity knowledge by **running a meeting / creating challenges / participating in CTFs**, or expressing **interest in {design, branding, outreach, or marketing}**

- talk to an admin / send a message on discord to let us know you want to help!
- See sigpwny.com/faq for more details
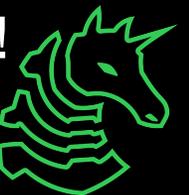
Recap: Reverse Engineering

- Reverse Engineering: Figure out how a program works

- Two major (non-exclusive) techniques
  - Static analysis (today: **Ghidra**)
  - Dynamic analysis (today: **GDB**)

- Different strategies for RE
  - Today: C / C++ on Linux ("ELF binaries")
  - Later: Java Rev, Rev III (Side channels, VMs, Symbolic execution)
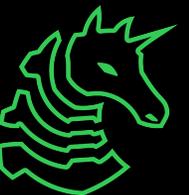
Compilation

(base) nathan@desktop:~/Documents/sigpwny/re3/pres$ ./my_compiled_program
Hello world!

Compiler
(Clang)

Executable
(assembly code)

Source code
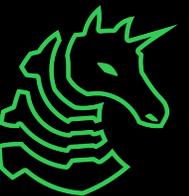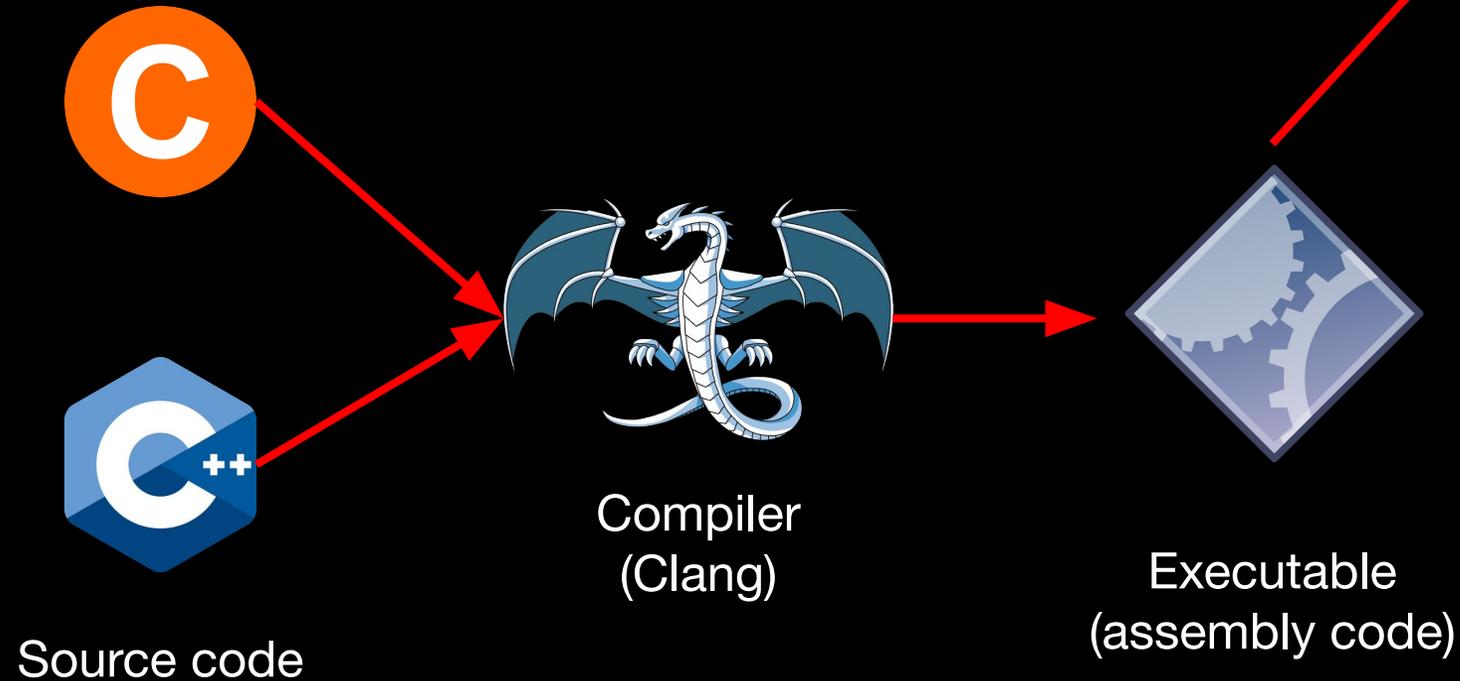
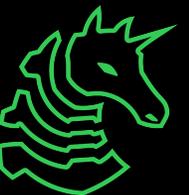# Recap: Assembly

Sam's slides from Sunday

What is Assembly?

- A human-readable abstraction over CPU machine codes

0100100000000010111011110110000000011011100010011
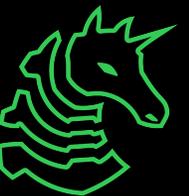
48 05 DE C0 37 13

add rax, 0x1337c0de

What is Assembly?

```
int method(int a){
    int b = 6;
    char c = 'c';
    return a+b;
}
```

```
method:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 6
        mov     BYTE PTR [rbp-5], 99
        mov     edx, DWORD PTR [rbp-20]
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        pop     rbp
        ret
```
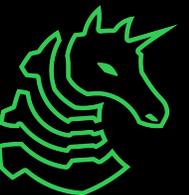
Basic CPU Structures

Instruction Memory

```
[0x00401000]
    ;-- section..text:
    ;-- segment.LOAD1:
entry0 ();
push      rsp
pop       rsi
xor       dl, 0x60
syscall
ret
```

Registers
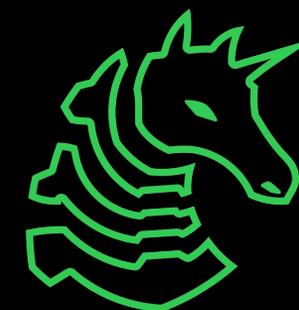
```
*RAX  0x3e8
*RBX  0x401300 (__libc_csu_init) <-
*RCX  0x7ffff7ea311b (getegid+11) <
 RDX  0x0
*RDI  0x7ffff7fad7e0 (_IO_stdfile_1
 RSI  0x0
 R8   0x0
*R9   0x7ffff7fe0d60 (_dl_fini) <-
*R10  0x400502 <- 0x64696765746567
*R11  0x202
*R12  0x401110 (_start) <- endbr64
*R13  0x7fffffffddc0 <- 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdcd0 <- 0x0
*RSP  0x7fffffffdcb0 <- 0x0
*RIP  0x401220 (main+42) <- mov
```

Stack

```
0x7fffffffdcb0 <- 0x0
0x7fffffffdcb8 -> 0x401110 (_star
0x7fffffffdcc0 -> 0x7fffffffddc0
0x7fffffffdcc8 <- 0x0
0x7fffffffdcd0 <- 0x0
0x7fffffffdcd8 -> 0x7ffff7de3083
```

Compilation / Decompilation

We can go from C code to assembly...

```c
1  int some_mathz() {
2      int res = 0;
3      for (int i = 9; i > 1; i++) {
4          res -= i;
5      }
6  }
```

```asm
some_mathz():
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 9
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-8]
        sub     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-8], 1
.L2:
        cmp     DWORD PTR [rbp-8], 1
        jg      .L3
        ud2
```
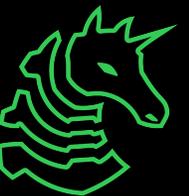
https://godbolt.org/

Now go from assembly to C code 😈

```
add(unsigned int):
        test    edi, edi
        je      .L4
        mov     eax, 1
        mov     edx, 0
.L3:
        add     edx, eax
        add     eax, 1
        cmp     edi, eax
        jnb     .L3
.L2:
        mov     eax, edx
        ret
.L4:
        mov     edx, edi
        jmp     .L2
```

Challenge: What does this do?

```c
unsigned add(unsigned n) {
    // Compute 1 + 2 + ... + n
    unsigned result = 0;
    for (unsigned i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

Not perfect!

Ghidra to the rescue!

- Open source disassembler/decompiler
  - **Disassembler**: binary machine code to assembly
  - **Decompiler**: assembly to pseudo-C

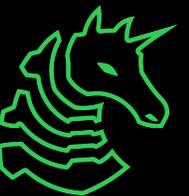- Written by the NSA 😳
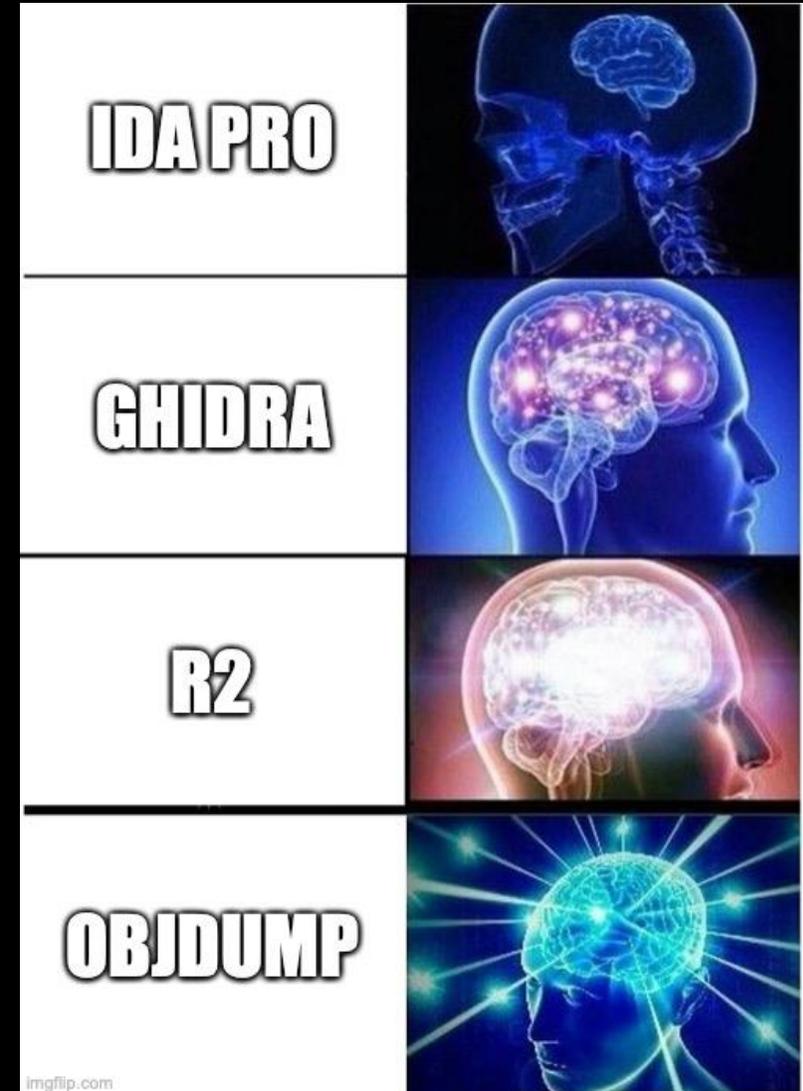
Ghidra to the rescue!

```
unsigned add(unsigned n) {
    // Compute 1 + 2 + ... + n
    unsigned result = 0;
    for (unsigned i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

Decompilation not always the same! Many ways to write equivalent code

```
uint add(uint n)

{
    uint i;
    uint result;

    result = n;
    if (n != 0) {
        i = 1;
        result = 0;
        do {
            result = result + i;
            i = i + 1;
        } while (i <= n);
    }
    return result;
}
```

Common Optimizations

Loading an array with bytes
-   Loading first 8 bytes simultaneously into stack (in one instruction)

```c
#include <stdio.h>

int main() {
    48 65 6c 6c 6f 20 77 6f 72 6c 64

    char string[] = "Hello world";
    printf("%s",string);

    return 0;
}
```

Challenge: why is the text of the decoded number backwards?

```c
int __cdecl main(int _Argc,char **_Argv,char **_Env)

{
    undefined8 local_14;
    undefined4 local_c;

    __main();
    local_14 = 0x6f77206f6c6c6548;     "ow olleH" in hex
    local_c = 0x646c72;                "dlr" in hex
    printf("%s",&local_14);
    return 0;
}
```

Common Optimizations (Cont.)

Modulo replaced with mask
- % 4 replaced with & 0b11 (Taking the last two bits of unsigned int)

```c
#include <stdio.h>

int main() {

    unsigned int A,B;
    scanf("%u",&A);
    B = A % 4;
    printf("%u",B);

    return 0;
}
```

```c
int __cdecl main(int _Argc,char **_Argv,char **_Env)

{
    uint A;
    uint B;

    __main();
    scanf("%u",&A);
    B = A & 0b00000011;
    printf("%u",(ulonglong)B);
    return 0;
}
```

Ghidra Follow Along

Open Ghidra!

sigpwny.com/rev_setup23

Download "debugger" from https://ctf.sigpwny.com/challenges

Ghidra Cheat Sheet

- Get started:
    - View all functions in list on left side of screen inside "Symbol Tree". Double click **main** to decompile main
- Decompiler:
    - Middle click a variable to highlight all instances in decompilation
    - Type "L" to rename variable (after clicking on it)
    - "Ctrl+L" to retype a variable (type your type in the box)
    - Type ";" to add an inline comment on the decompilation and assembly
    - Alt+Left Arrow to navigate back to previous function
- General:
    - Double click an XREF to navigate there
    - Search -> For Strings -> Search to find all strings (and XREFs)
    - Choose Window -> Function Graph for a graph view of disassembly
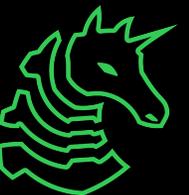
GDB (Dynamic Analysis)

-   Able to inspect a program's variables & state as it runs
-   Set breakpoints, step through, try various inputs
-   Debugging analogy: print statements after running
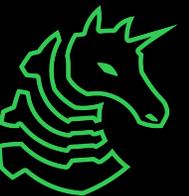
Dynamic Analysis with GDB

- Run program, with the ability to pause and resume execution
- View registers, stack, heap
- Steep learning curve
- `chmod +x ./chal` to make executable

GDB Follow Along

Same file as Ghidra follow along (debugger)
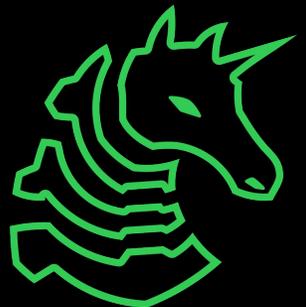
GDB Cheat Sheet   [gdb](gdb)   pwndbg

- `b main` - Set a breakpoint on the main function
  - `b *main+10` - Set a breakpoint a couple instructions into main
- `r` - run
  - `r arg1 arg2` - Run program with arg1 and arg2 as command line arguments. Same as `./prog arg1 arg2`
 - `r < myfile` - Run program and supply contents of myfile.txt to stdin
- `c` - continue
- `si` - step instruction (steps into function calls)
- `ni` - next instruction (steps over function calls) (`finish` to return to caller function)
- `x/32xb 0x5555555551b8` - Display 32 hex bytes at address 0x5555555551b8
  - `x/4xg addr` - Display 4 hex "giants" (8 byte numbers) at addr
  - `x/16i $pc` - Display next 16 instructions at $rip
  - `x/s addr` - Display a string at address
  - `x/4gx {void*}$rcx` - Dereference pointer at $rcx, display 4 QWORDs
  - `p/d {int*}{int*}$rcx` - Dereference pointer to pointer at $rcx as decimal
- `info registers` - Display registers (shorthand: `i r`)
- x86 Linux calling convention* ("System V ABI"): RDI, RSI, RDX, RCX, R8, R9

*syscall calling convention is RDI, RSI, RDX, **R10**, R8, R9

Pwndbg cheat sheet

- `emulate #` - Emulate the next # instructions
- `stack #` - Print # values on the stack
- `vmmap` - Print memory segments (use `-x` flag to show only executable segments)
- `nearpc` - Disassemble near the PC
- `tel <ptr>` - Recursively dereferences <ptr>
- `regs` - Use instead of `info reg` (gdb's register viewing)
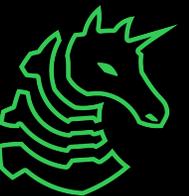
Go try for yourself!

- https://ctf.sigpwny.com
- Start with Crackme 0
- Practice practice practice! Ask for help!
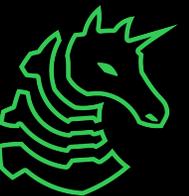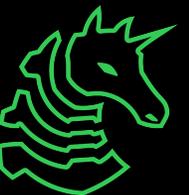
Next Meetings

2023-10-08 - **This Sunday**

- DRM circumvention with Ojas this sunday

2023-10-12 - **Next Thursday**

- Crypto I
- Learn the fundamentals of cryptography and encryption!

ctf.sigpwny.com

sigpwny{nsa_backdoor?}

Thanks for listening!

SIGPwny