SIGPwny

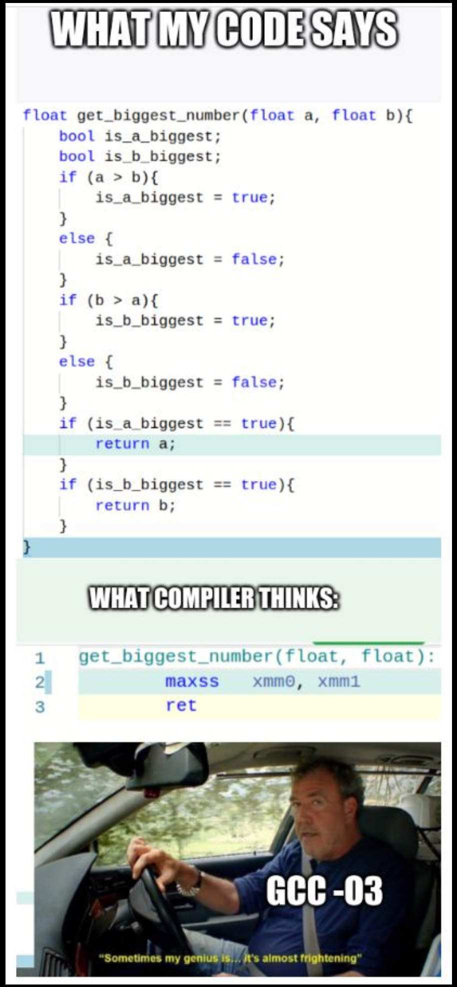
FA2024 Week 06 • 2024-10-10

Reverse Engineering II

Nikhil Date

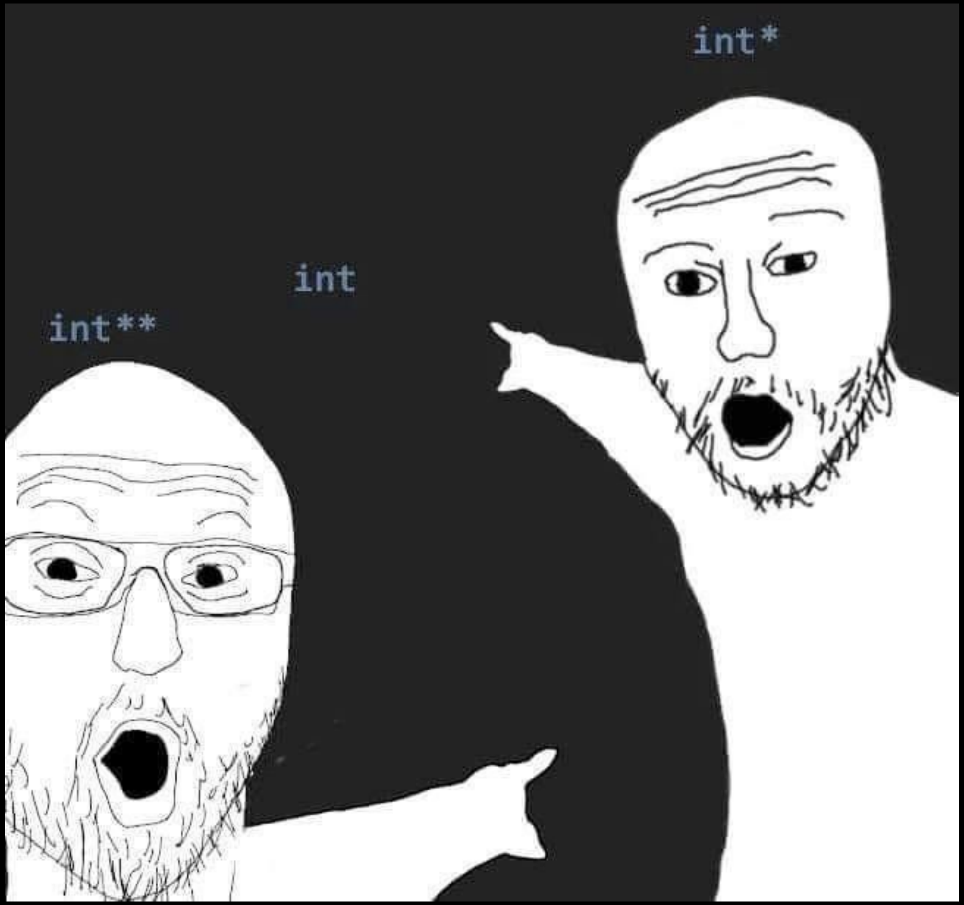
ctf.sigpwny.com

sigpwny{unrecovered_jumptable}



```
float get_biggest_number(float a, float b){
    bool is_a_biggest;
    bool is_b_biggest;
    if (a > b){
        is_a_biggest = true;
    }
    else {
        is_a_biggest = false;
    }
    if (b > a){
        is_b_biggest = true;
    }
    else {
        is_b_biggest = false;
    }
    if (is_a_biggest == true){
        return a;
    }
    if (is_b_biggest == true){
        return b;
    }
}
```

```
get_biggest_number(float, float):
        maxss   xmm0, xmm1
        ret
```



Setup

- If you haven’t installed Ghidra yet, start downloading it through the slides here: **sigpwny.com/rev_setup23**

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
  - more broadly: get useful information out of a program
- Why reverse engineering?
  - Solve reverse engineering CTF challenges and get flags
  - Find vulnerabilities in software
  - Makes you a better programmer
  - And more
- Two major (non-exclusive) techniques
  - Static analysis (today: **Ghidra**)
  - Dynamic analysis (today: **GDB**)

Recap: Assembly

Sam and Emma’s slides from Sunday

What is Assembly?

- A human-readable abstraction over CPU machine codes

```
010010000000010111011110110000000011011100010011

48 05 DE C0 37 13

add rax, 0x1337c0de
```

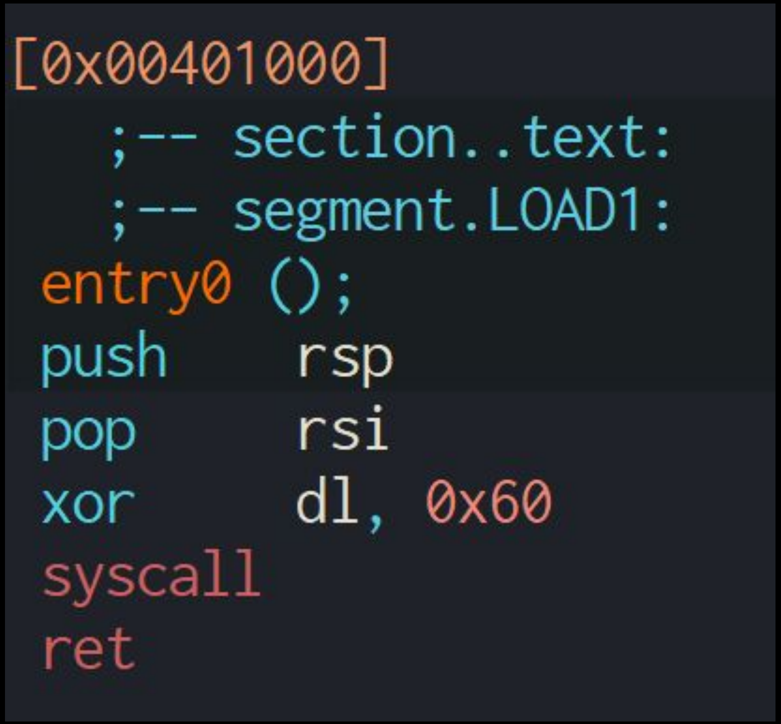
What is Assembly?

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

```asm
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
```

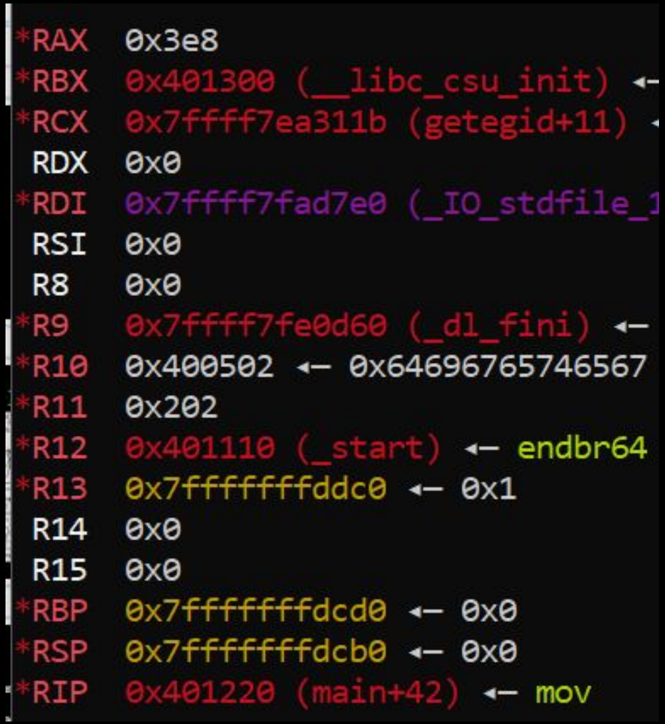
Basic CPU Structures

Instruction Memory



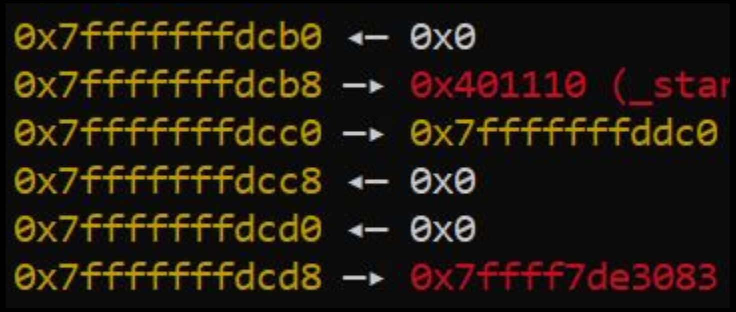
```
[0x00401000]
    ;-- section..text:
    ;-- segment.LOAD1:
 entry0 ();
 push      rsp
 pop       rsi
 xor       dl, 0x60
 syscall
 ret
```

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

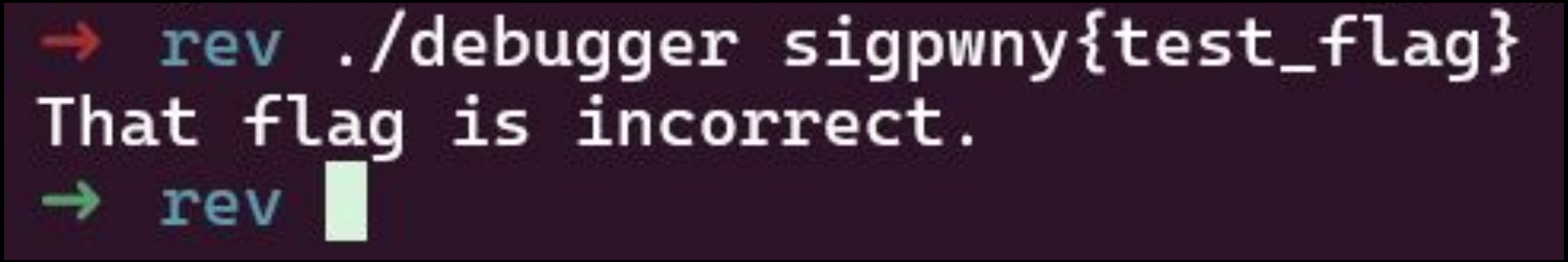


```
0x7fffffffdcb0 ◂— 0x0
0x7fffffffdcb8 —▸ 0x401110 (_star
0x7fffffffdcc0 —▸ 0x7fffffffddc0
0x7fffffffdcc8 ◂— 0x0
0x7fffffffdcd0 ◂— 0x0
0x7fffffffdcd8 —▸ 0x7ffff7de3083
```

What is this meeting about?

- Reverse engineering binaries
    - Compiled executables
    - All source information is usually (but not always) stripped
- What do we have to work with?
    - Machine code
    - Sometimes, some symbol names (like function names)
    - At minimum, only what the OS needs to execute the program

Running example: debugger

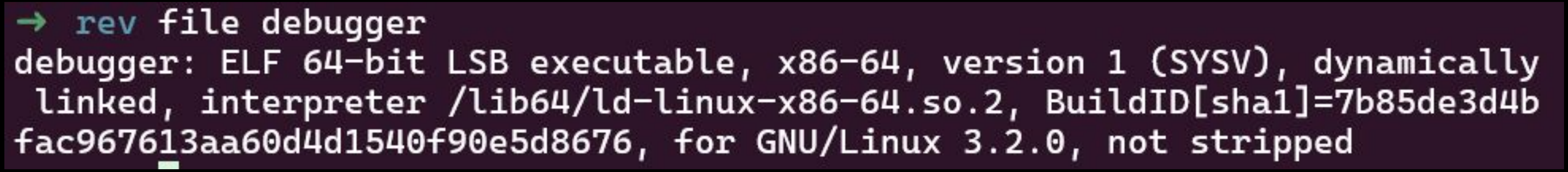


```
→  rev ./debugger sigpwny{test_flag}
That flag is incorrect.
→  rev 
```

- Challenge might feel completely opaque right now
- But we will be able to solve it by the end of the meeting
- Follow along!

The ELF Format

- What kind of file is debugger?
  - The more information you have about the program you are reversing, the easier it is
- Use Unix “file” utility

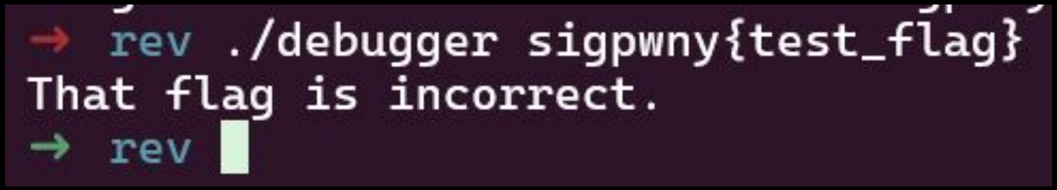


```
→ rev file debugger
debugger: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=7b85de3d4b
fac967613aa60d4d1540f90e5d8676, for GNU/Linux 3.2.0, not stripped
```

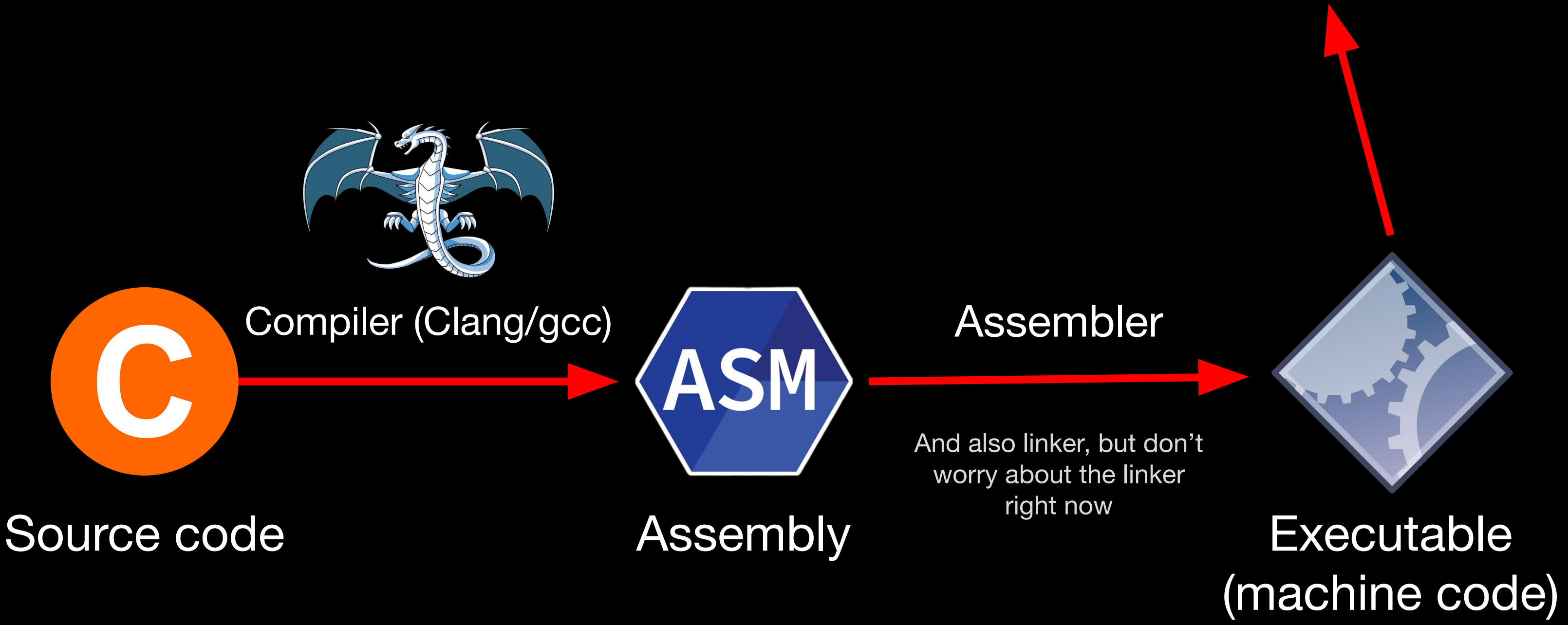
- ELF: Executable and Linkable Format
  - File format for **executables**, libraries, object files
  - Contains program code and data, plus metadata needed to execute program
  - Can also contain symbols (“not stripped”)
  - More info: https://github.com/corkami/pics/blob/28cb0226093ed57b348723bc473cea0162dad366/binary/elf101/elf101.pdf
  - Useful tool: readelf

Compilation

Or, how does source code become an executable



```
→ rev ./debugger sigpwny{test_flag}
That flag is incorrect.
→ rev
```



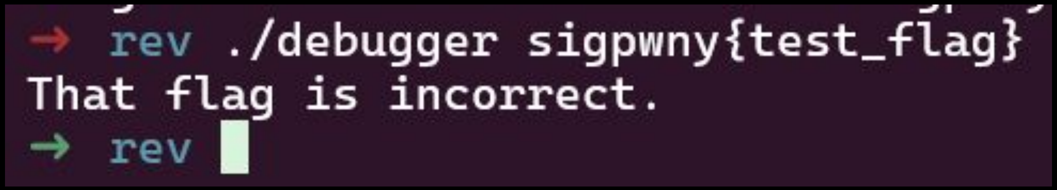
Source code

Assembly

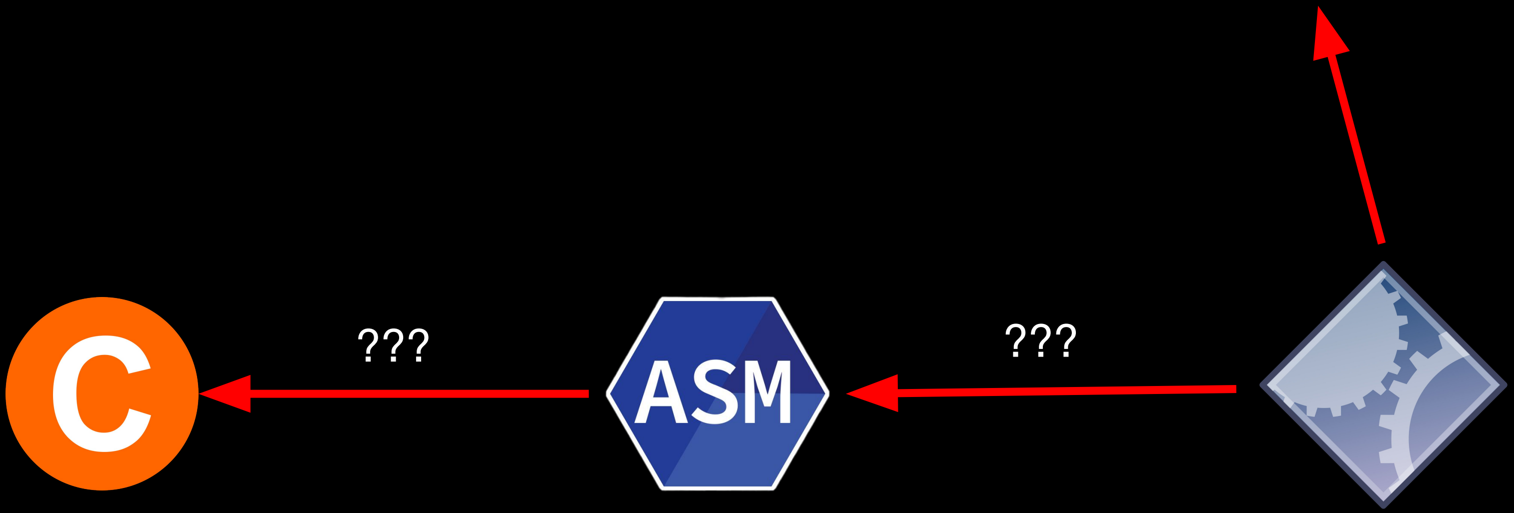
And also linker, but don't worry about the linker right now

Executable (machine code)

Can we go the other way?



```
→ rev ./debugger sigpwny{test_flag}
That flag is incorrect.
→ rev
```

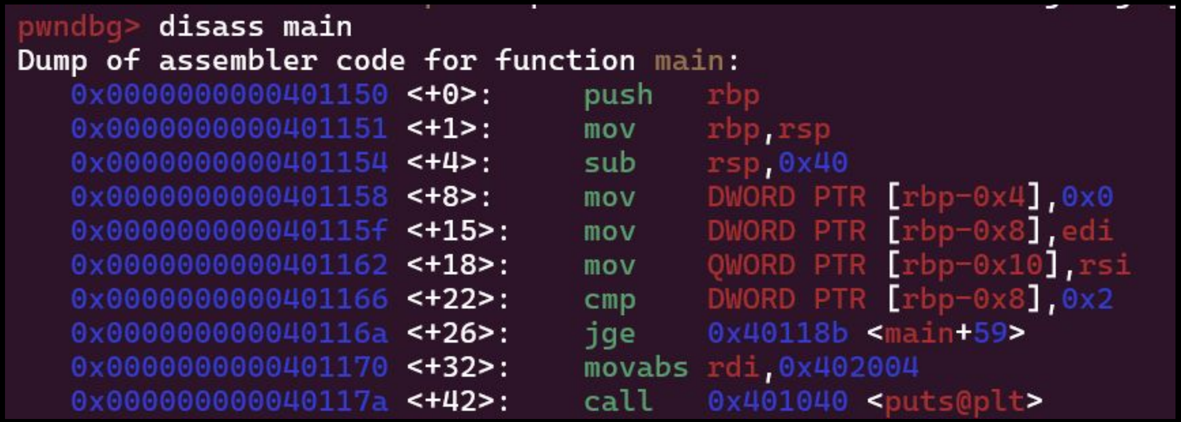


Source code

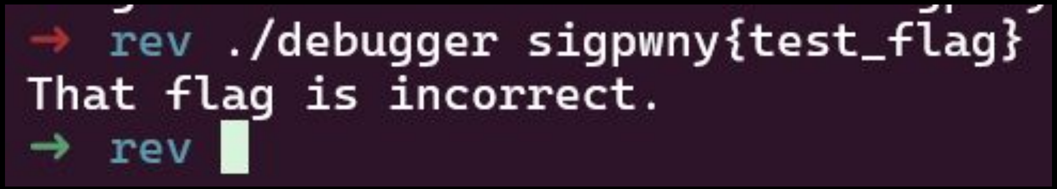
Assembly

Executable
(machine code)

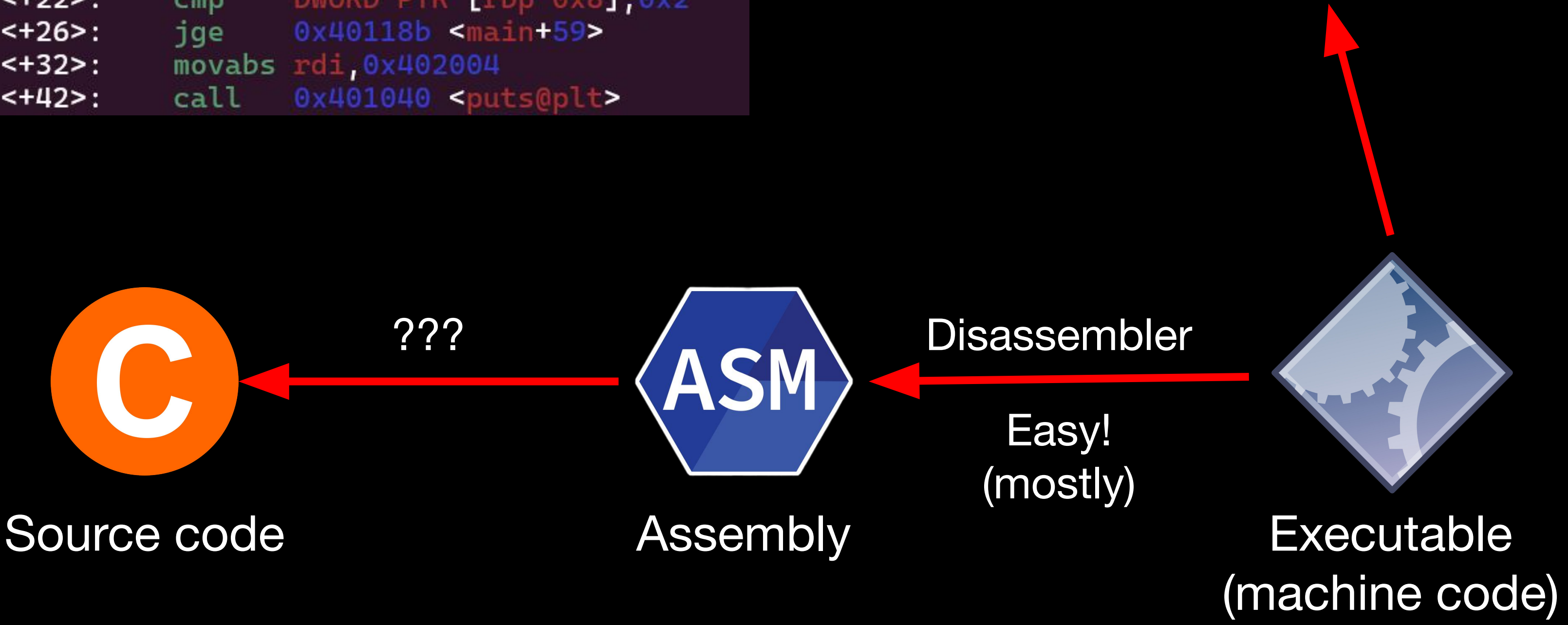
Can we go the other way?



```
pwndbg> disass main
Dump of assembler code for function main:
   0x0000000000401150 <+0>:     push   rbp
   0x0000000000401151 <+1>:     mov    rbp,rsp
   0x0000000000401154 <+4>:     sub    rsp,0x40
   0x0000000000401158 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x000000000040115f <+15>:    mov    DWORD PTR [rbp-0x8],edi
   0x0000000000401162 <+18>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000401166 <+22>:    cmp    DWORD PTR [rbp-0x8],0x2
   0x000000000040116a <+26>:    jge    0x40118b <main+59>
   0x0000000000401170 <+32>:    movabs rdi,0x402004
   0x000000000040117a <+42>:    call   0x401040 <puts@plt>
```



```
➜ rev ./debugger sigpwny{test_flag}
That flag is incorrect.
➜ rev
```



Source code

???

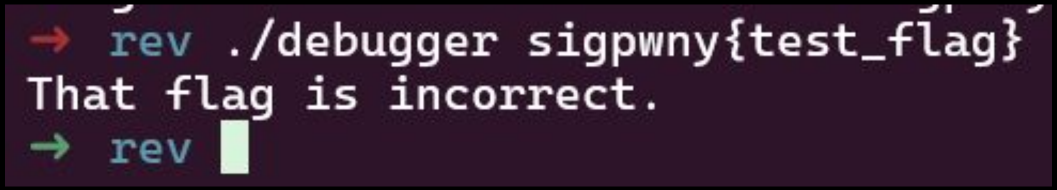
Assembly

Disassembler

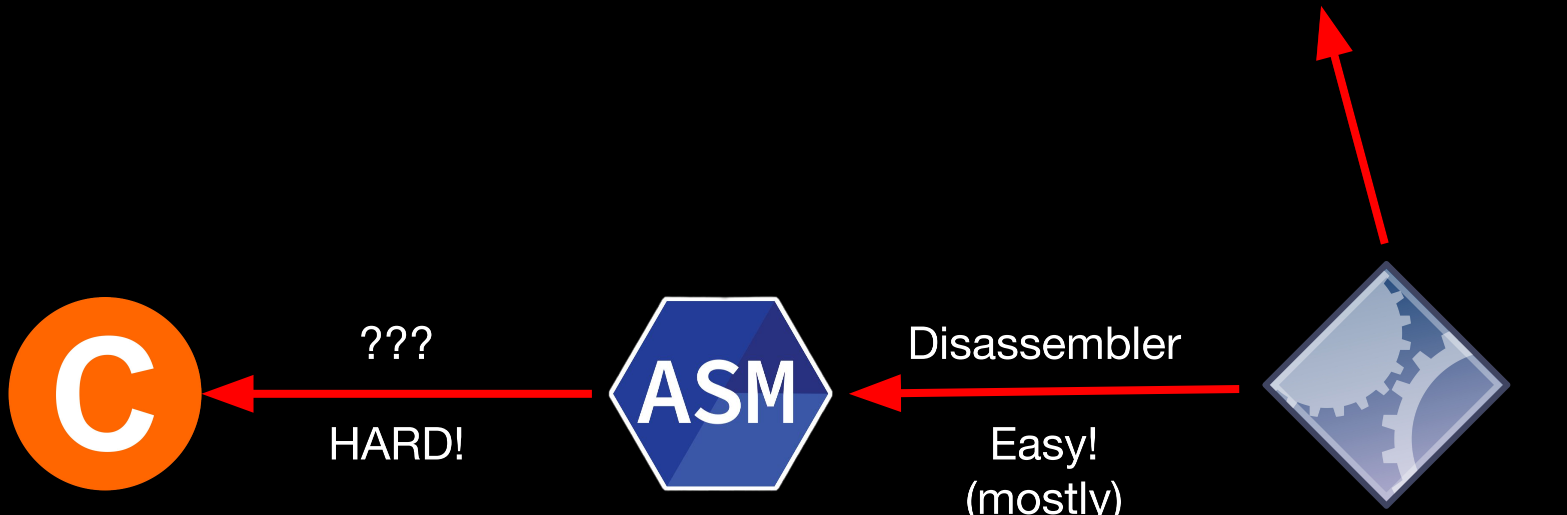
Easy!
(mostly)

Executable
(machine code)

Can we go the other way?



```
→ rev ./debugger sigpwny{test_flag}
That flag is incorrect.
→ rev
```



???

HARD!

Disassembler

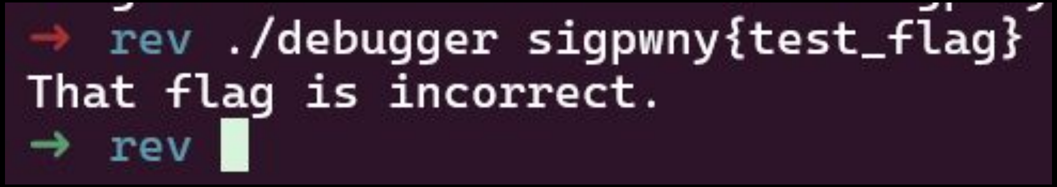
Easy!
(mostly)

Source code

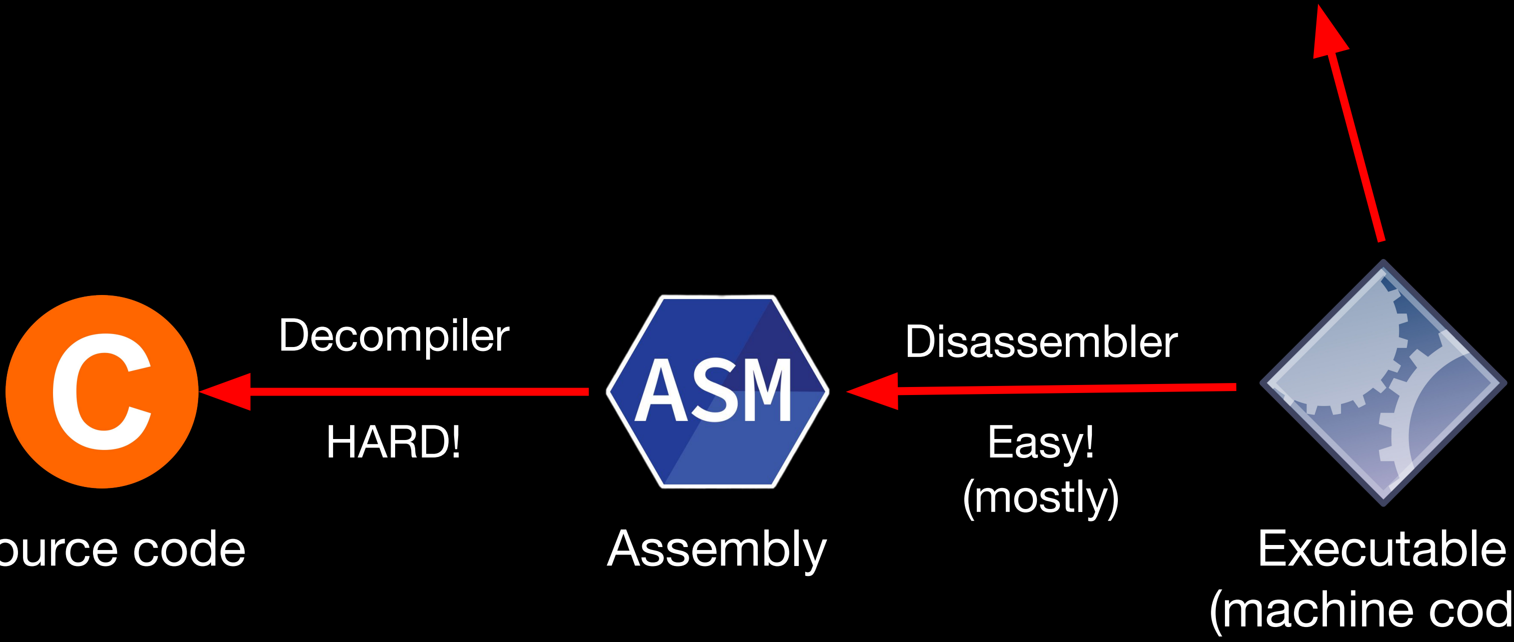
Assembly

Executable
(machine code)

Can we go the other way?



```
→ rev ./debugger sigpwny{test_flag}
That flag is incorrect.
→ rev
```



Decompiler

HARD!

Disassembler

Easy!
(mostly)

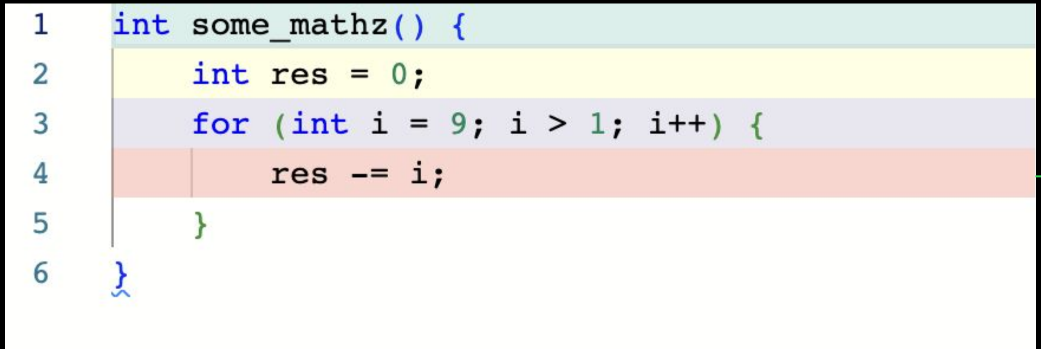
Source code

Assembly

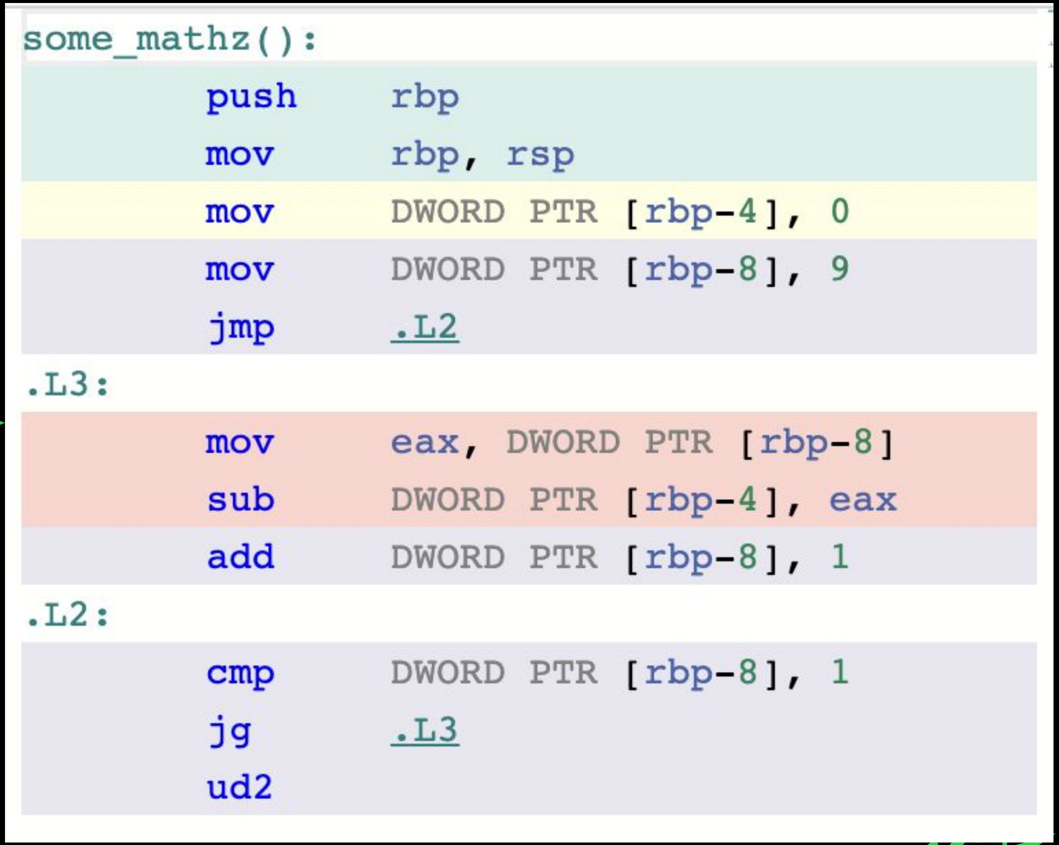
Executable
(machine code)

Decompilation

We can go from C code to assembly…



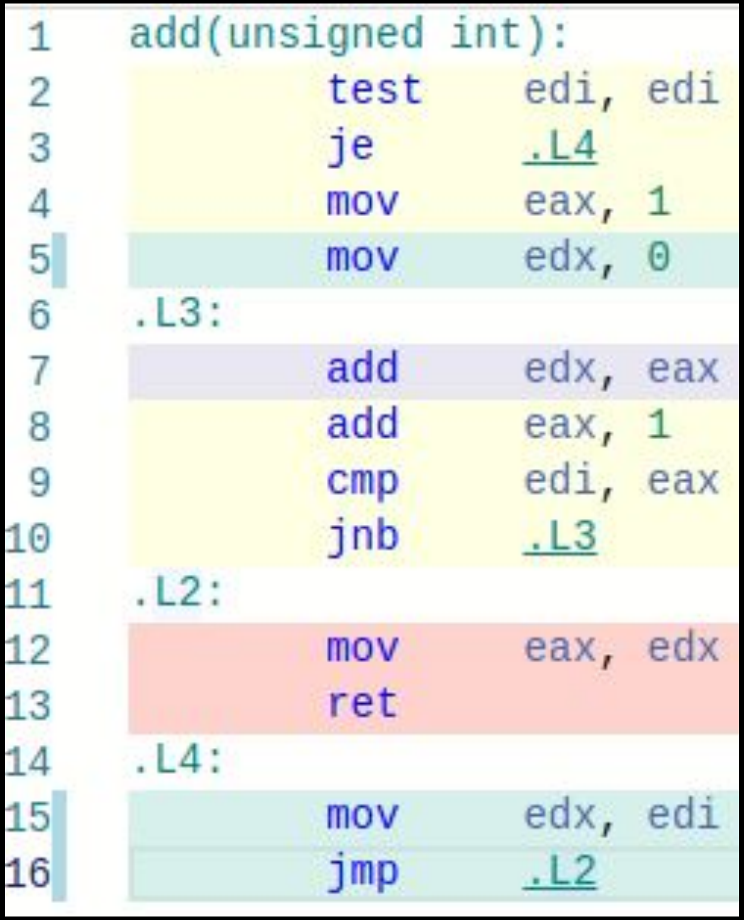
```c
int some_mathz() {
    int res = 0;
    for (int i = 9; i > 1; i++) {
        res -= i;
    }
}
```



```
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/

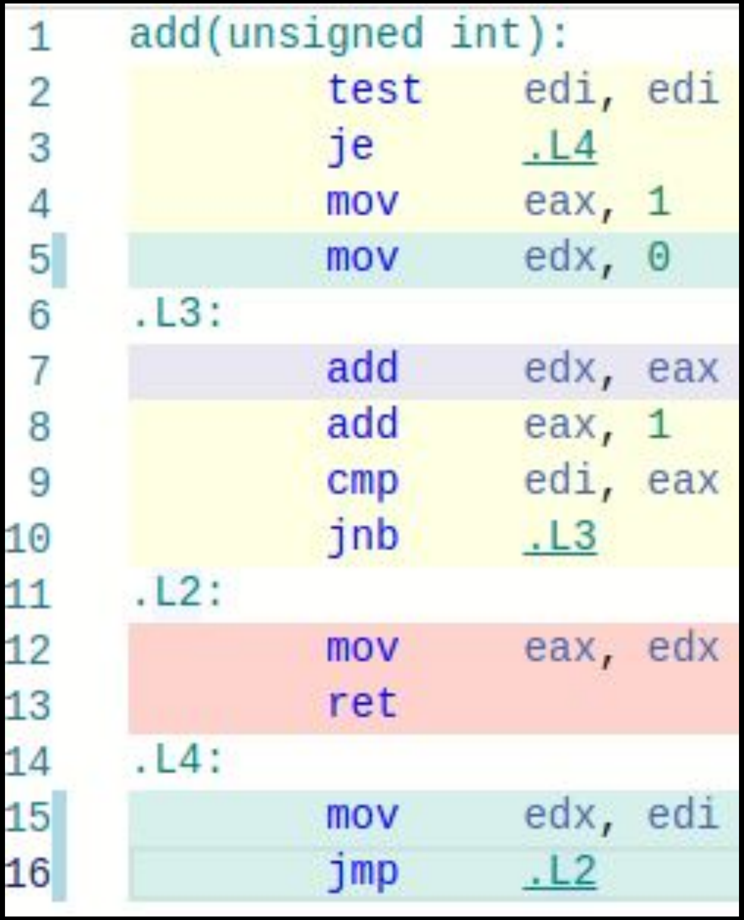
Now go from assembly to C code 😈



```
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
```

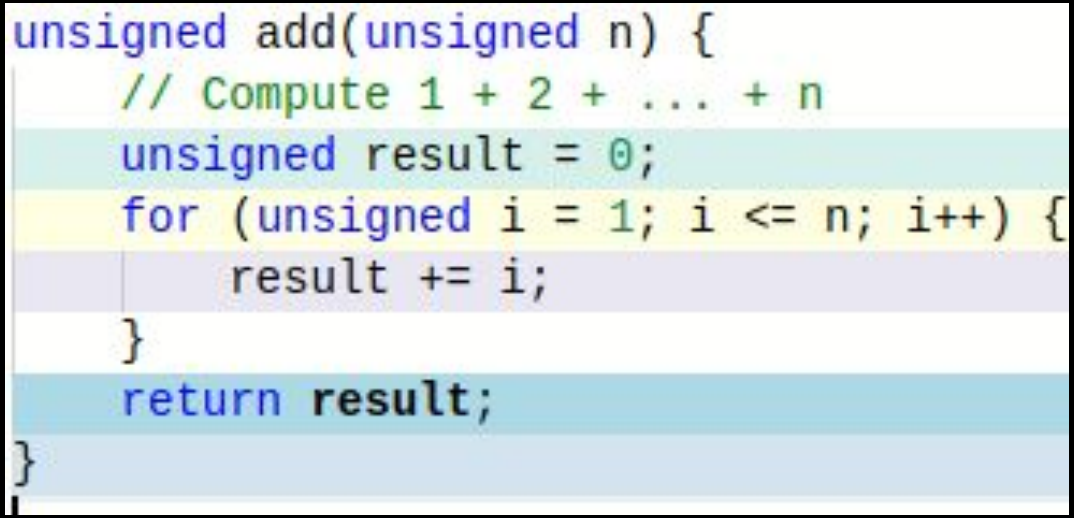
Challenge: What does this do?

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?

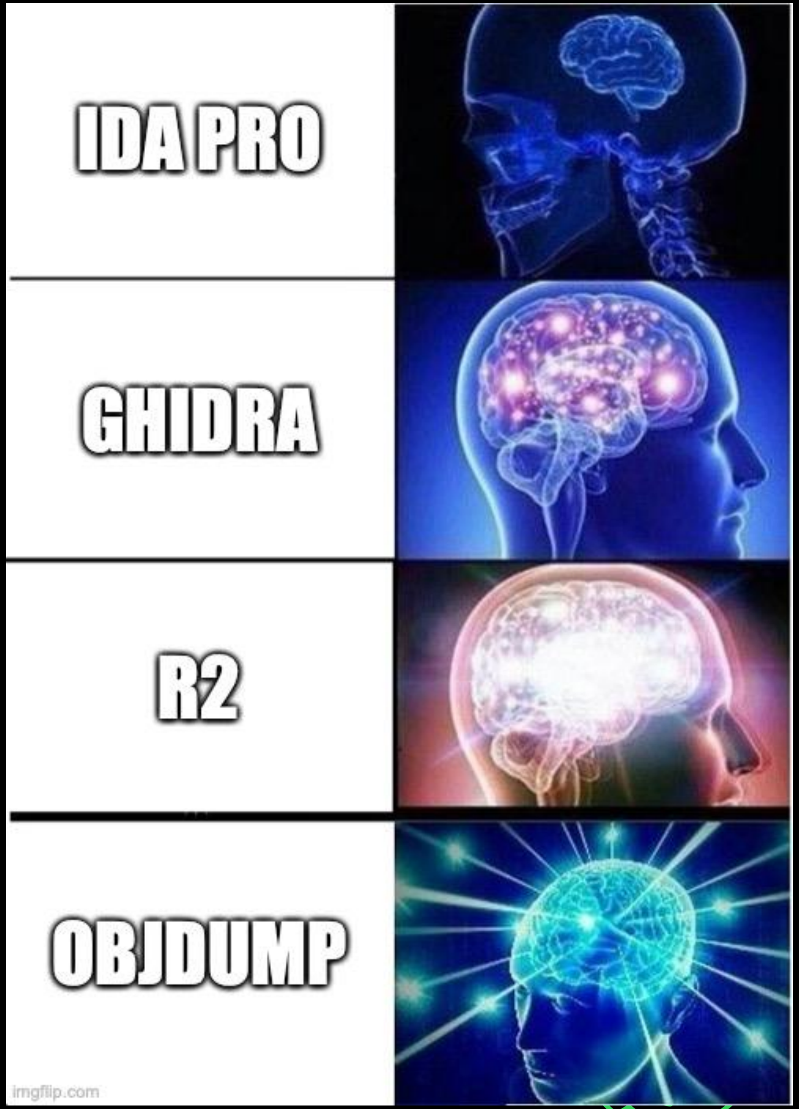


```
unsigned add(unsigned n) {
    // Compute 1 + 2 + ... + n
    unsigned result = 0;
    for (unsigned i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}
```

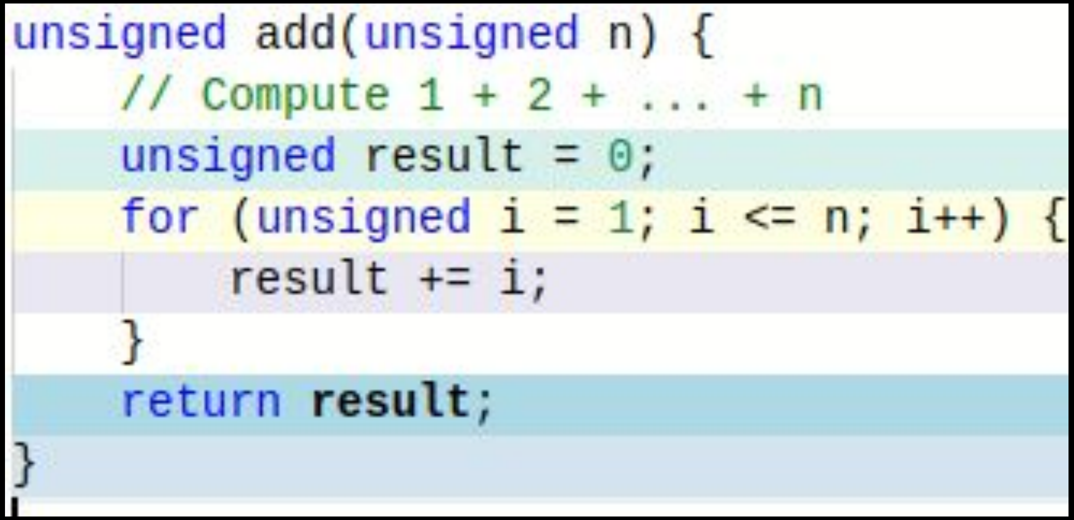
Ghidra to the rescue!

- Open source disassembler/decompiler/”reverse engineering framework”
  - **Disassembler**: binary machine code to assembly
  - **Decompiler**: assembly to pseudo-C
  - Reverse engineering framework: control flow graph recovery, cross-references, binary similarity/diffing, and more!

- Written by the NSA 😳

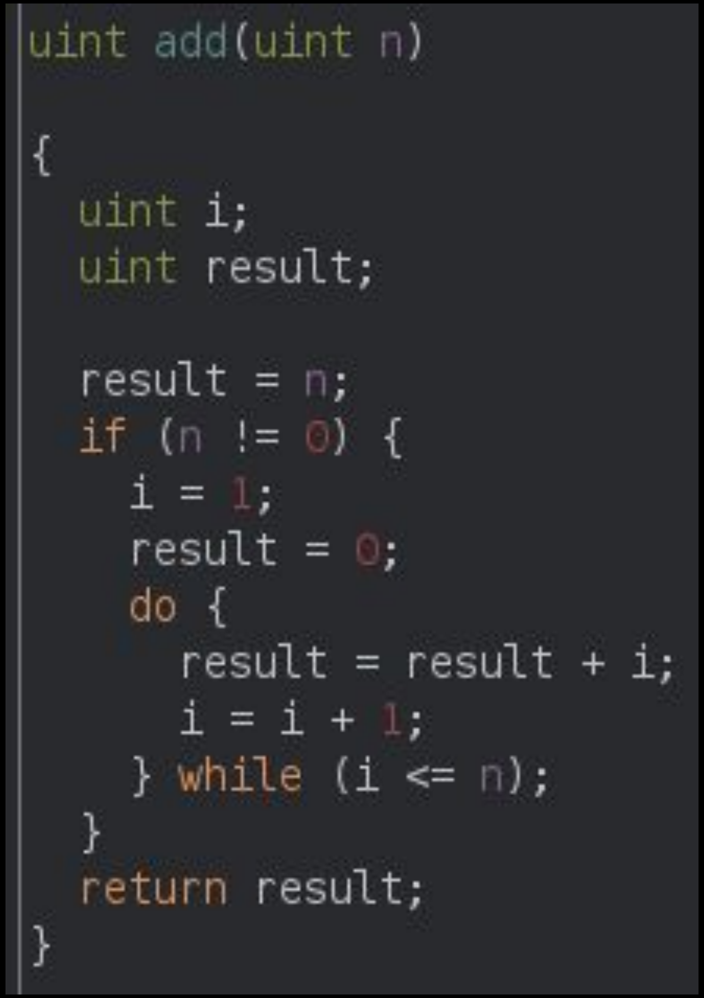


Ghidra caveats



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

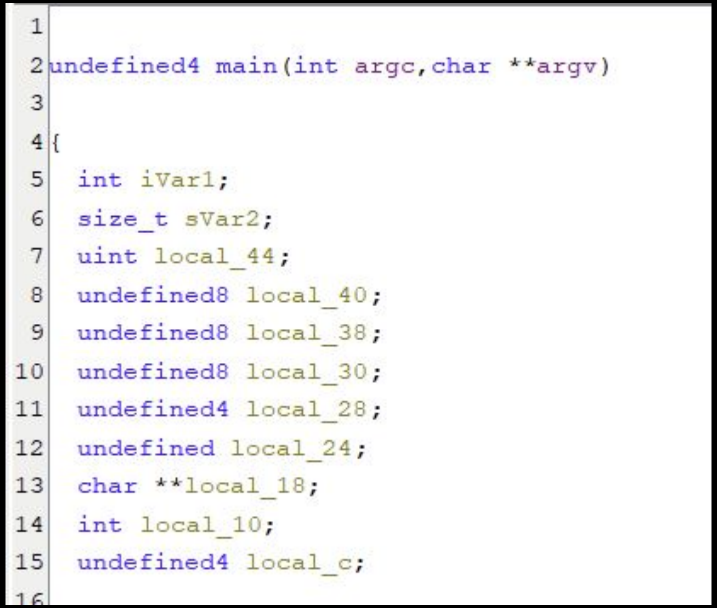
Decompilation not always the same! Many ways to write equivalent code



```c
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;
}
```

Ghidra caveats

- Ghidra output is not meant to be recompilable
  - It’s meant to be human-readable
- Decompilation is a best guess
  - But not all information (e.g. types) is always recovered

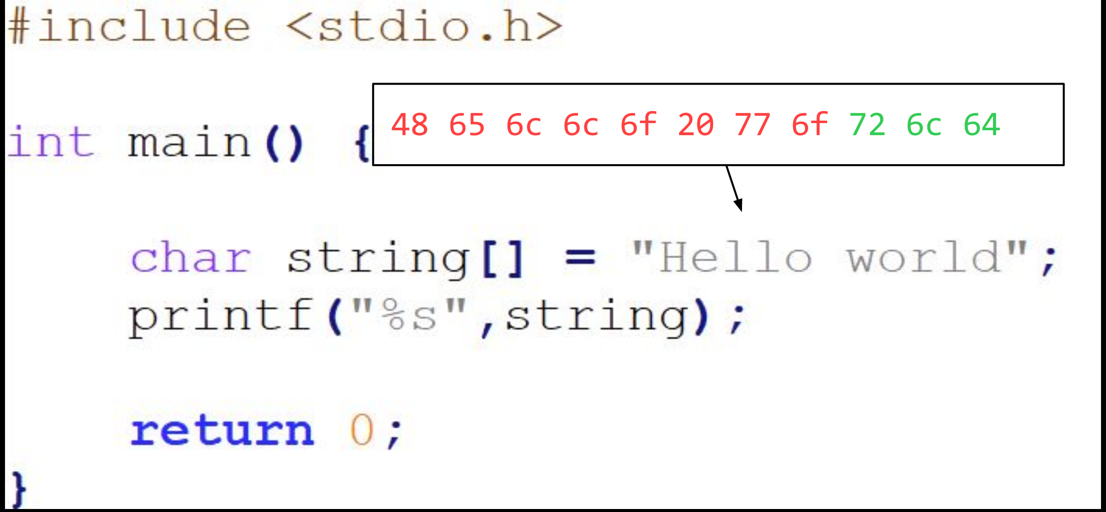


```

undefined4 main(int argc,char **argv)

{
  int iVar1;
  size_t sVar2;
  uint local_44;
  undefined8 local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined4 local_28;
  undefined local_24;
  char **local_18;
  int local_10;
  undefined4 local_c;
```

Common Optimizations

Loading an array with bytes

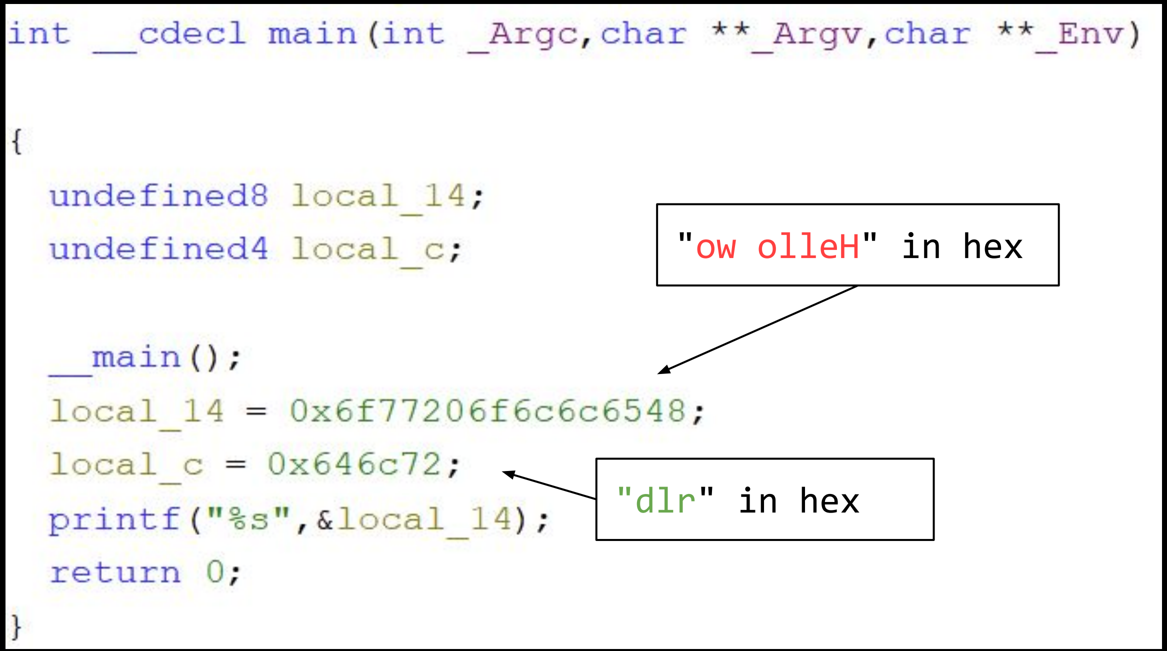
- Loading first 8 bytes simultaneously into stack (in one instruction)



```c
#include <stdio.h>

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

    return 0;
}
```

48 65 6c 6c 6f 20 77 6f 72 6c 64

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;
  local_c = 0x646c72;
  printf("%s",&local_14);
  return 0;
}
```

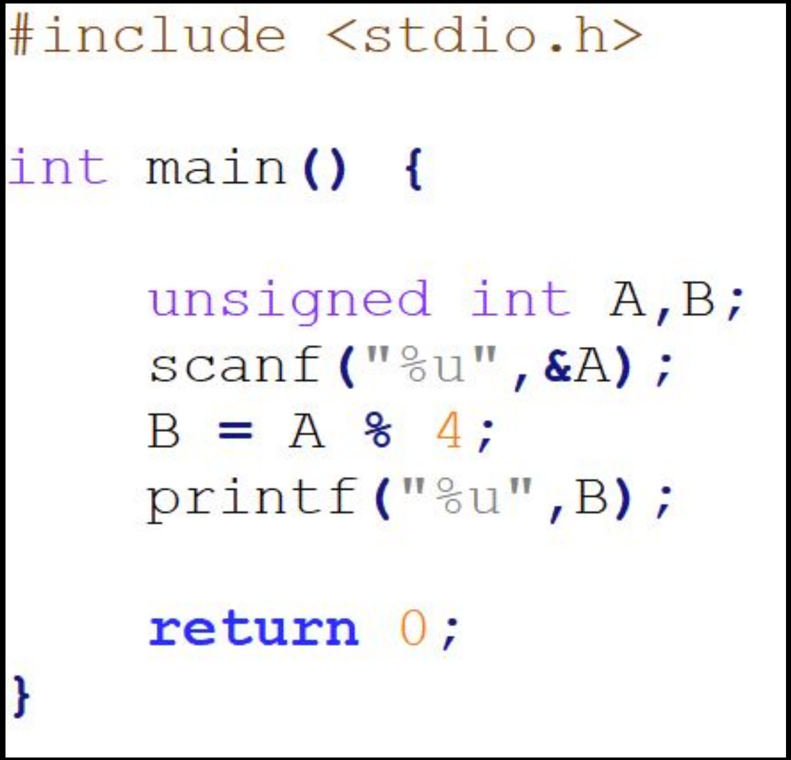
"ow olleH" in hex

"dlr" in hex

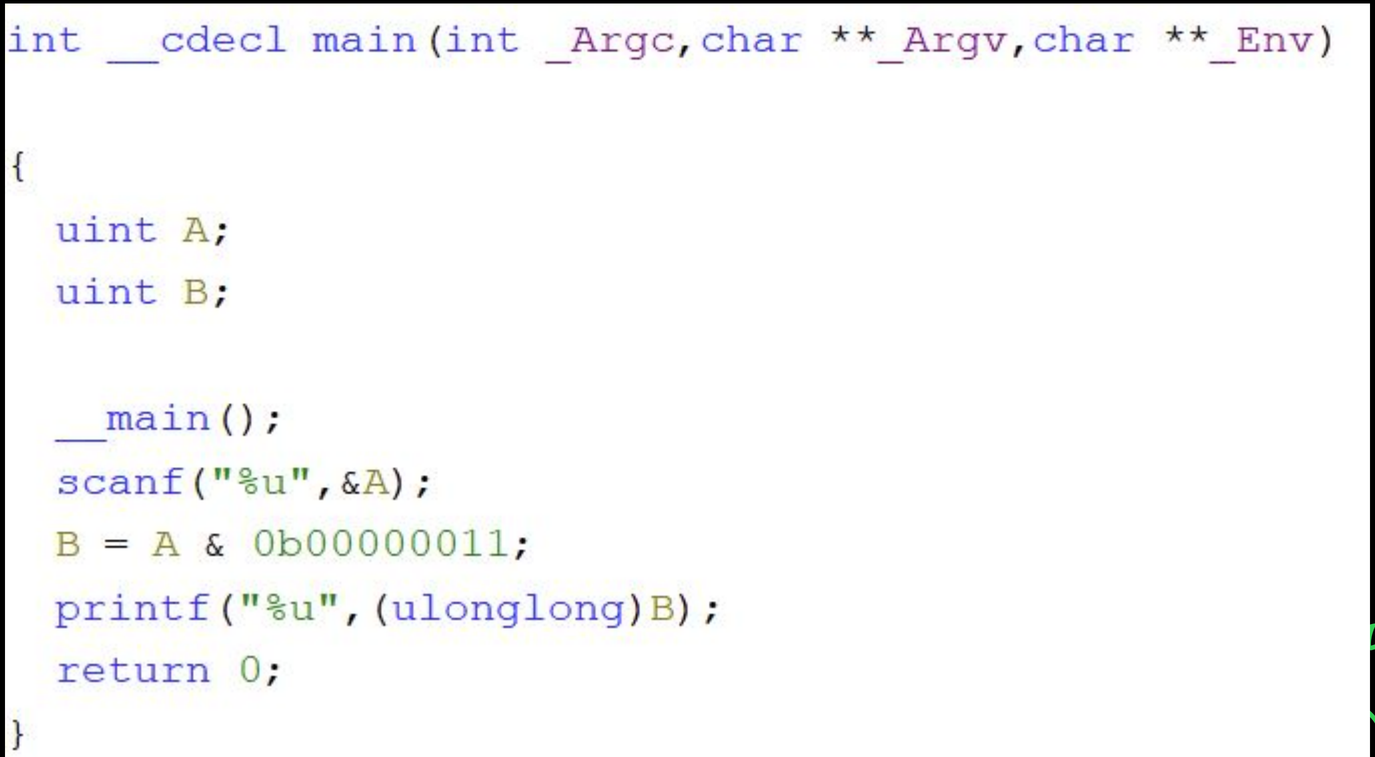
Common Optimizations (Cont.)

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

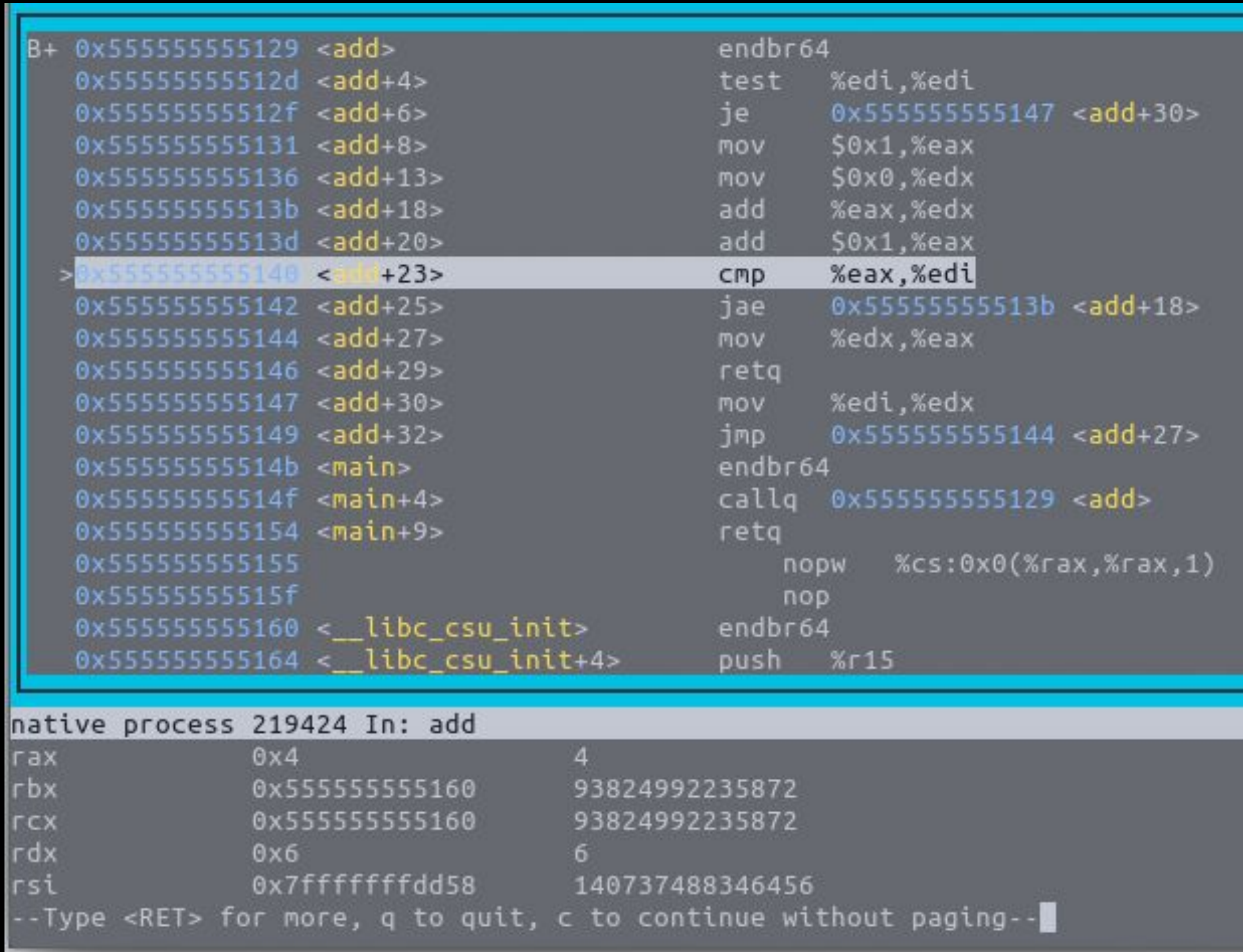
- Get started:
    - View all functions in list on left side of screen inside “Symbol Tree”. Double click **main** to decompile main
- Decompiler:
    - Middle click a variable to highlight all instances in decompilation
    - Type “L” to rename variable (after clicking on it)
    - “Ctrl+L” to retype 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

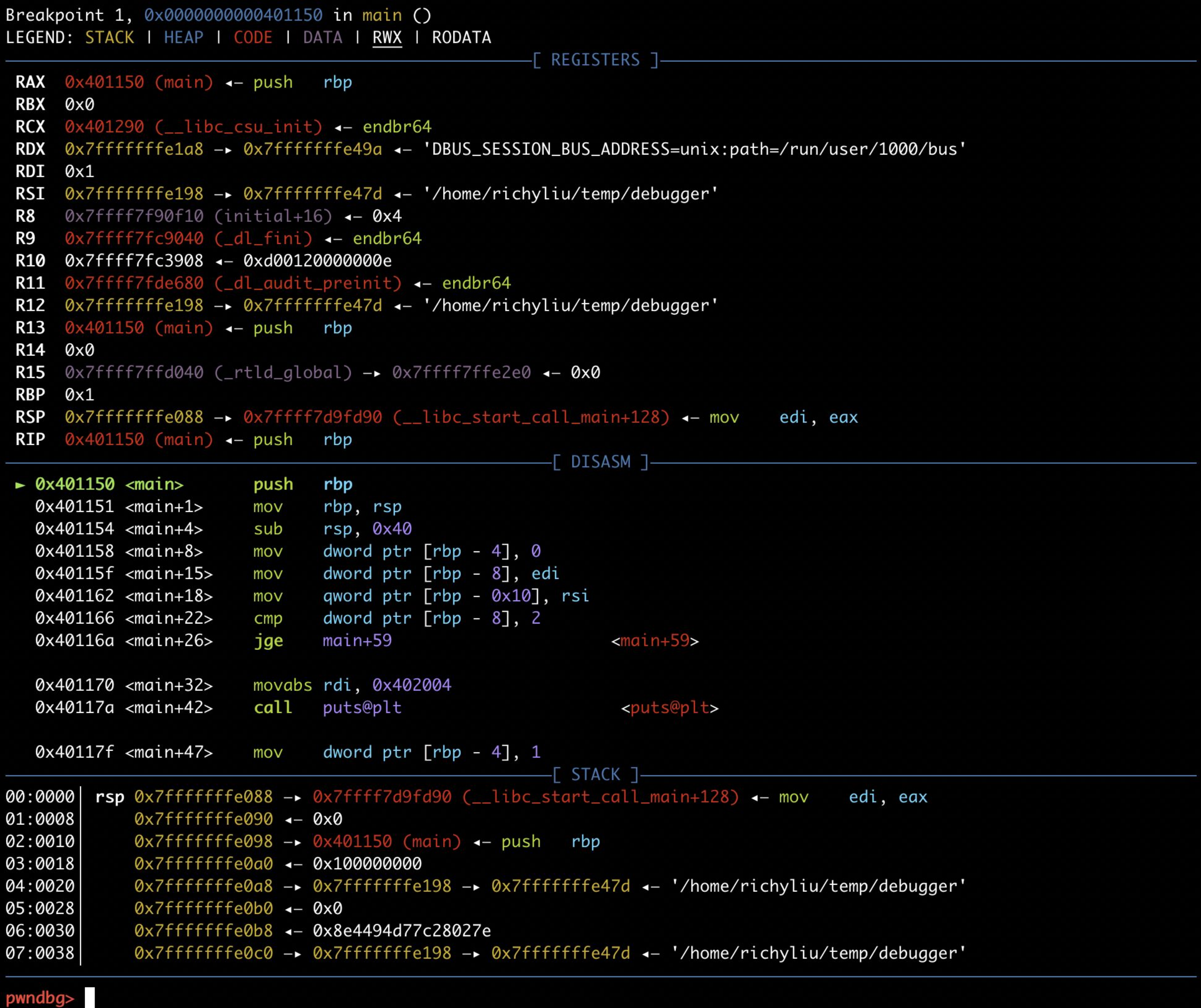
- Run program, with the ability to pause and resume execution
- View registers, stack, heap
- Steep learning curve
- `chmod +x ./chal` to make executable



```
B+ 0x555555555129 <add>                    endbr64
   0x55555555512d <add+4>                  test   %edi,%edi
   0x55555555512f <add+6>                  je     0x555555555147 <add+30>
   0x555555555131 <add+8>                  mov    $0x1,%eax
   0x555555555136 <add+13>                 mov    $0x0,%edx
   0x55555555513b <add+18>                 add    %eax,%edx
   0x55555555513d <add+20>                 add    $0x1,%eax
  >0x555555555140 <add+23>                 cmp    %eax,%edi
   0x555555555142 <add+25>                 jae    0x55555555513b <add+18>
   0x555555555144 <add+27>                 mov    %edx,%eax
   0x555555555146 <add+29>                 retq
   0x555555555147 <add+30>                 mov    %edi,%edx
   0x555555555149 <add+32>                 jmp    0x555555555144 <add+27>
   0x55555555514b <main>                   endbr64
   0x55555555514f <main+4>                 callq  0x555555555129 <add>
   0x555555555154 <main+9>                 retq
   0x555555555155                              nopw   %cs:0x0(%rax,%rax,1)
   0x55555555515f                              nop
   0x555555555160 <__libc_csu_init>        endbr64
   0x555555555164 <__libc_csu_init+4>      push   %r15
native process 219424 In: add
rax            0x4                 4
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x6                 6
rsi            0x7fffffffdd58      140737488346456
--Type <RET> for more, q to quit, c to continue without paging--
```

pwndbg

```
git clone
https://github.com
/pwndbg/pwndbg

cd pwndbg

./setup.sh
```



```
Breakpoint 1, 0x0000000000401150 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────[ REGISTERS ]──────────────────────────────────────────
 RAX  0x401150 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x401290 (__libc_csu_init) ◂— endbr64
 RDX  0x7fffffffe1a8 —▸ 0x7fffffffe49a ◂— 'DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus'
 RDI  0x1
 RSI  0x7fffffffe198 —▸ 0x7fffffffe47d ◂— '/home/richyliu/temp/debugger'
 R8   0x7ffff7f90f10 (initial+16) ◂— 0x4
 R9   0x7ffff7fc9040 (_dl_fini) ◂— endbr64
 R10  0x7ffff7fc3908 ◂— 0xd00120000000e
 R11  0x7ffff7fde680 (_dl_audit_preinit) ◂— endbr64
 R12  0x7fffffffe198 —▸ 0x7fffffffe47d ◂— '/home/richyliu/temp/debugger'
 R13  0x401150 (main) ◂— push   rbp
 R14  0x0
 R15  0x7ffff7ffd040 (_rtld_global) —▸ 0x7ffff7ffe2e0 ◂— 0x0
 RBP  0x1
 RSP  0x7fffffffe088 —▸ 0x7ffff7d9fd90 (__libc_start_call_main+128) ◂— mov    edi, eax
 RIP  0x401150 (main) ◂— push   rbp
───────────────────────────────────────────[ DISASM ]────────────────────────────────────────────
 ► 0x401150 <main>       push   rbp
   0x401151 <main+1>     mov    rbp, rsp
   0x401154 <main+4>     sub    rsp, 0x40
   0x401158 <main+8>     mov    dword ptr [rbp - 4], 0
   0x40115f <main+15>    mov    dword ptr [rbp - 8], edi
   0x401162 <main+18>    mov    qword ptr [rbp - 0x10], rsi
   0x401166 <main+22>    cmp    dword ptr [rbp - 8], 2
   0x40116a <main+26>    jge    main+59                       <main+59>

   0x401170 <main+32>    movabs rdi, 0x402004
   0x40117a <main+42>    call   puts@plt                       <puts@plt>

   0x40117f <main+47>    mov    dword ptr [rbp - 4], 1
───────────────────────────────────────────[ STACK ]─────────────────────────────────────────────
00:0000│ rsp 0x7fffffffe088 —▸ 0x7ffff7d9fd90 (__libc_start_call_main+128) ◂— mov    edi, eax
01:0008│     0x7fffffffe090 ◂— 0x0
02:0010│     0x7fffffffe098 —▸ 0x401150 (main) ◂— push   rbp
03:0018│     0x7fffffffe0a0 ◂— 0x100000000
04:0020│     0x7fffffffe0a8 —▸ 0x7fffffffe198 —▸ 0x7fffffffe47d ◂— '/home/richyliu/temp/debugger'
05:0028│     0x7fffffffe0b0 ◂— 0x0
06:0030│     0x7fffffffe0b8 ◂— 0x8e4494d77c28027e
07:0038│     0x7fffffffe0c0 —▸ 0x7fffffffe198 —▸ 0x7fffffffe47d ◂— '/home/richyliu/temp/debugger'
─────────────────────────────────────────────────────────────────────────────────────────────────
pwndbg> █
```

Windows users - WSL
m1 mac users - pwn-docker

GDB Follow Along

Same file as Ghidra follow along (debugger)

GDB Cheat Sheet

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!

Going Further

- Side channels: e.g. instruction counting
- Symbolic/concolic execution
- Ghidra scripts
- Z3 and constraint solvers
- Emulation for dynamic analysis
- Taint analysis
- and more!
- Many of these will be covered in Rev III

Next Meetings

`2024-10-13` **- This Sunday**

- Operational Security I with Minh and Sagnik
- Protect your digital footprint (and finally learn what passkeys are)

`2024-10-17` **- Next Thursday**

- Physical Security and Lockpicking with Emma
- Learn how people break into buildings and pick locks for flags!

ctf.sigpwny.com

sigpwny{unrecovered_jumptable}

Thanks for listening!

SIGPwny