Reverse Engineering

Richard Liu

September 23, 2023

Contents

1 Start here **2**

1.1 General tips 2

1.2 A note about past meetings 2

2 Basics **2**

2.1 What it is 2

2.2 Main types of analysis 2

2.3 A word on abstractions 3

3 Tools **3**

3.1 Bytecode viewer 3

3.1.1 Installation 3

3.1.2 When to use 3

3.1.3 How to use 3

3.2 Ghidra 3

3.2.1 Installation 3

3.2.2 When to use 3

3.2.3 Interface 4

3.3 GDB 6

3.3.1 Installation 6

3.3.2 When to use 6

3.3.3 Basics 6

3.3.4 Commands 6

3.3.5 General workflow 7

1 Start here

1.1 General tips

- figure out what the goal is
  - there is usually a clear “win condition”, such as printing a flag
- figure out what the input is
  - some parts of the program don’t change depending on the input
  - it might not matter what the input is!
  - how does the input get used?

1.2 A note about past meetings

SIGPwny has already ran two meetings on this topic! Check out Reverse Engineering Setup and Reverse Engineering I. We have slides and recorded meeting presentations, which you may prefer more than these notes.

2 Basics

2.1 What it is

Reverse engineering is the process of understanding computer programs. The goal is to figure out what the program does. Usually, programs are difficult to understand, either intentionally or unintentionally.

2.2 Main types of analysis

- Static analysis: reading code, using tools to understand code *without running it*
  - Good place to start, not great if there’s a lot of code
- Dynamic analysis: running code, inspecting or modifying the program as it’s running
  - Generally faster, captures entire program environment

2.3 A word on abstractions

- Abstract (higher level) programs are easier to understand
- Languages like Python and JavaScript are higher level
- Languages like assembly and C are lower level
- As you modify a program to become more abstract (to better understand it), you lose some information in the process

3 Tools

3.1 Bytecode viewer

3.1.1 Installation

- see `https://github.com/Konloch/bytecode-viewer`

3.1.2 When to use

This program is used to decompile Java files, which usually have the .jar extension

3.1.3 How to use

Simply import the java jar program into the bytecode viewer and see the decompiled java code! This works by recovering the java code from the compiled java bytecode.

3.2 Ghidra

3.2.1 Installation

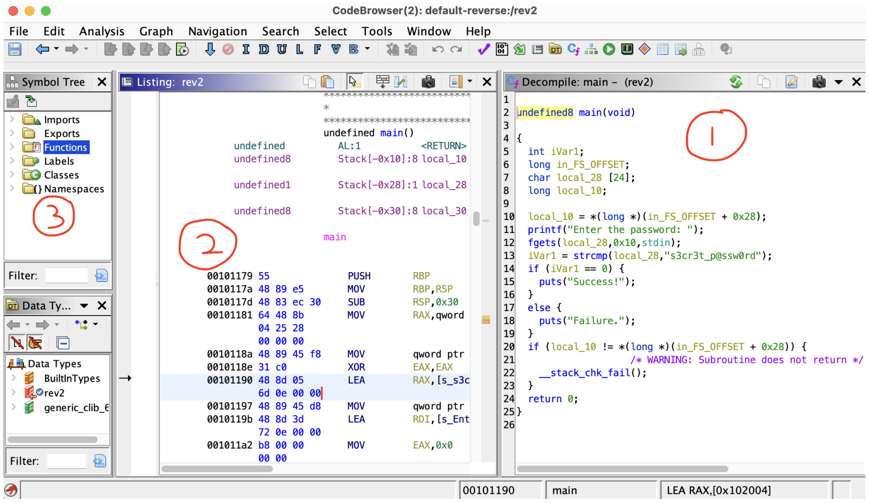
- see Reverse Engineering Setup
- or, just read the installation guide

3.2.2 When to use

Use this tool for binaries, not python scripts. Ghidra "decompiles", or simplifies, binary programs into more human-readable "pseudo-C" code.

Ghidra is a **static analysis** tool.

3.2.3 Interface



```
                               **************************
                               *
                               **************************
                               undefined main()
          undefined              AL:1           <RETURN>
          undefined8             Stack[-0x10]:8 local_10

          undefined1             Stack[-0x28]:1 local_28

          undefined8             Stack[-0x30]:8 local_30

                               main

      00101179 55                 PUSH       RBP
      0010117a 48 89 e5           MOV        RBP,RSP
      0010117d 48 83 ec 30        SUB        RSP,0x30
      00101181 64 48 8b           MOV        RAX,qword
               04 25 28
               00 00 00
      0010118a 48 89 45 f8        MOV        qword ptr
      0010118e 31 c0              XOR        EAX,EAX
      00101190 48 8d 05           LEA        RAX,[s_s3c
               6d 0e 00 00
      00101197 48 89 45 d8        MOV        qword ptr
      0010119b 48 8d 3d           LEA        RDI,[s_Ent
               72 0e 00 00
      001011a2 b8 00 00           MOV        EAX,0x0
               00 00
```

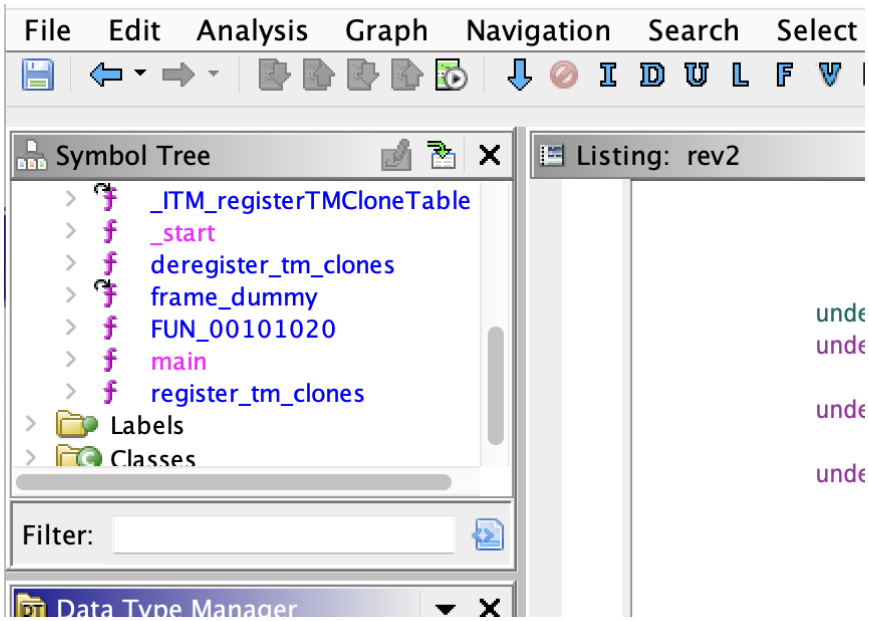
```

undefined8 main(void)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter the password: ");
  fgets(local_28,0x10,stdin);
  iVar1 = strcmp(local_28,"s3cr3t_p@ssw0rd");
  if (iVar1 == 0) {
    puts("Success!");
  }
  else {
    puts("Failure.");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}

```

Once you open a program in Ghidra, click “OK” for all the auto analyze popups (there should be several). Now, the interface should look like the above image.

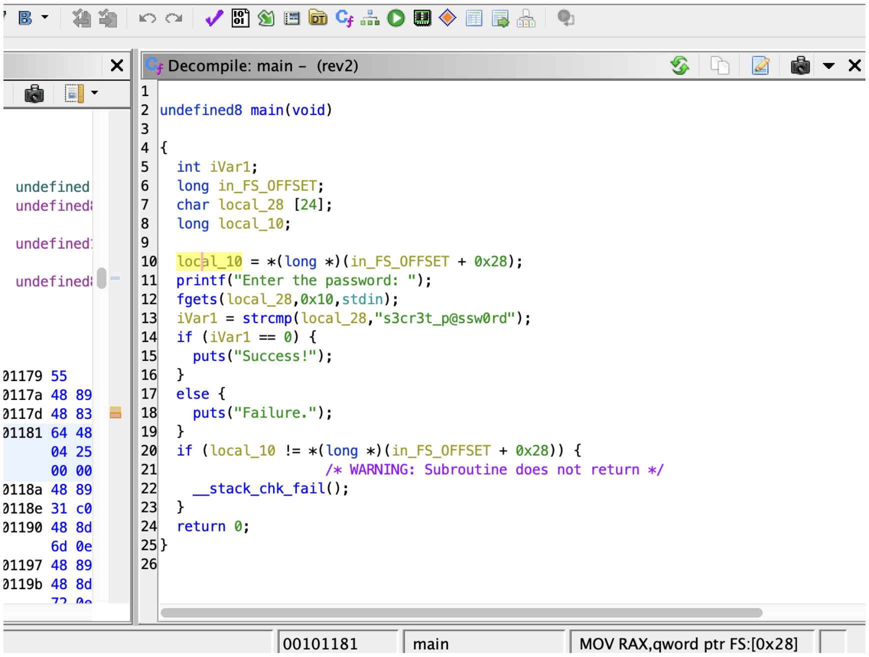
(1) is the decompiled code output. This is what you will be looking at for the most part. You can rename variables by clicking a variable and pressing `L`. Change the type by right clicking and selecting `Retype Variable`.

(2) is the assembly instructions. This won’t be very helpful if you don’t know assembly, and can be mostly ignored for the challenges at Fall CTF.

(3) is the “symbol tree”. This shows you different named values that are present in the file. Click `Functions` and scroll down to select the `main` function. This shows you the first function that runs.



Here we can see the main function in the symbol tree. If there is no main, click _start and see what that function calls.



```

undefined8 main(void)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter the password: ");
  fgets(local_28,0x10,stdin);
  iVar1 = strcmp(local_28,"s3cr3t_p@ssw0rd");
  if (iVar1 == 0) {
    puts("Success!");
  }
  else {
    puts("Failure.");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}

```

Above is a picture of the decompilation (disclaimer: this is not a challenge from Fall CTF). Almost every function you see will have an if statement with `__stack_chk_fail` at the bottom. This is a check for the "stack canary", which is not relevant to any challenges here. It may be of more interest in pwn challenge. The `local_10 = *(long *)(in_FS_OFFSET + 0x28);` line at the top sets up the stack canary and can also be ignored.

Note that the variables are named with undescriptive names, such as `iVar1` and `local_28`. This is because the decompiler does not know the details of variables in the original function. As a result, it has to generate variable names.

3.3 GDB

3.3.1 Installation

- see Reverse Engineering Setup

3.3.2 When to use

Similarly to Ghidra, use this tool for binaries, not python scripts. GDB is a debugger that runs programs, giving you the ability to stop, inspect, and modify code as it is executing.

GDB is a **dynamic analysis** tool.

3.3.3 Basics

Run `gdb ./chal` on the command line, where `chal` is the name of the program. Note that you must be on Linux (WSL works too). This will not work for Apple Silicon Mac users.

GDB will launch you into a program with a different terminal prompt, where each line starts with `(gdb)`. You interact with the program by typing in commands

3.3.4 Commands

- misc
  - `help <command>`: get help about any of the commands listed here
- running
  - `run`: run the program from the start

  - `quit`: exit GDB
  - `start`: start the program and break on the `main` function

- breakpoints

  - `break <func>+<offset>`: set a breakpoint at the function `<func>` with an offset `<offset>`. Useful to get the offset from the `disas` command

- inspecting program

  - `disas <func>`: disassemble the `<func>` function
  - `info reg`: print all the registers
  - `x`: print data (see `help x` for more info)
    - `x/4gx 0x1234`: print 4 QWORDS (64-bit values) in hex starting at address `0x1234`
    - `x/10i $rip`: print 10 instructions starting at `$rip` (current instruction pointer)
    - `x/7wx $rsp`: print 7 WORDS (32-bit values) in hex starting at `$rsp` (stack pointer)
    - `x/8bd $rdi`: print 8 bytes in decimal starting at the address in `$rdi`
  - `set`: set values
    - `set $rax=23`: sets `$rax` to 23
    - `set $rip+=4`: adds 4 to `$rip`
      - this skips the current instruction, if it is 4 bytes long

3.3.5 General workflow

- first, identify interesting places to set a breakpoint in Ghidra

- use the assembly instructions window in Ghidra to see the offset to break at

- run the program in GDB and set a breakpoint

- modify or print values as desired

- repeat until solved