Reverse Engineering: Cutter

Table of contents

1 Introduction

In this lab you will use Cutter to view and manipulate assembly to get the desired output from compiled code. It is recommended that you do this lab on your virtual machine. The files needed for this lab are located on canvas as `cutter-executables.zip`

2 Download and Install Cutter

Follow the appropriate set of instructions to install Cutter on your Virtual Machine.

2.1 x86-64

If your machine has an x86-64 architecture (host machine is Windows), follow these instructions:

1. Go to https://cutter.re and click download.

Troubleshooting Tip

If you are using firefox and have trouble visiting the download page, or if the download page doesn't autodetect that you're using Linux, download Chrome and install Chrome and try again.

2. Once downloaded, you will have a .AppImage file. Move this file to a location that you will remember it, perhaps a folder for this course.
3. Make the .AppImage file executable

```
chmod +x [filename].AppImage
```

4. Make sure you can open the file by either running it as an executable in the terminal or double clicking on it.

2.2 arm64

If your machine has arm64 architecture (ex: host machine is an m series Mac), you will need to build from source to install on your Virtual Machine.

1. Follow the instructions found here https://cutter.re/docs/building.html.
2. Move the file named `cutter` that was created in ./build to a convenient location, perhaps a folder for this course.
3. Make the file executable.

```
chmod +x cutter
```

3 File Manipulation

Dowload the `cutter-executables.zip` from Canvas onto your VM. When you unzip this file, you should find multiple folders inside indicating the architecture and OS for the machine that compiled the files. Note that the source code used to create these files is identical. When running the files, you should use the ones that correspond to your Virtual Machine's architecture.

3.1 File 1.exe

For file 1, our goal is to become familiar with various parts of the Cutter interface by modifying the output of the executable without the source code.

1. Before opening Cutter, execute `1.exe` to see what it does. Note that you may need to make it executable before running it using `chmod +x 1.exe`. Once you have run it, you should see that it simply prints out `Hello World`.
2. Now open Cutter and select `1.exe` as the file you want to work with using the `Select` button. Click Open once it has been selected.
3. After clicking Open, a window named `Load Options` will appear. You can leave the defaults and just click `Ok`.
4. Once you've opened the file, find the `Graph` tab at the bottom of the screen and select it.
5. Double click on `main` in the left navigation to view the assembly code associated with the main function.
6. At this point, if viewing the x86-64 version of `1.exe`, you should see some commands that are similar to those mentioned in class. If viewing the arm64 version of `1.exe`, they will look slightly different although you can probably guess the equivalent instructions for x86 for most of them.
7. Find `str.Hello_World` inside the assembly code while the Graph tab is active.
8. Navigate to the `Strings` tab.
9. Find the `Hello World` string. There is a filter input at the bottom of the screen to make it easier than just scrolling through.
10. Make a note the memory address to the left of the `Hello World` string. You might consider copying the address by right clicking on and selecting "Copy Address".
11. Next, navigate to the `Hexdump` tab found at the bottom of the screen. The hexdump shows us the bytes within the file. There are three areas of interest in the hexdump.

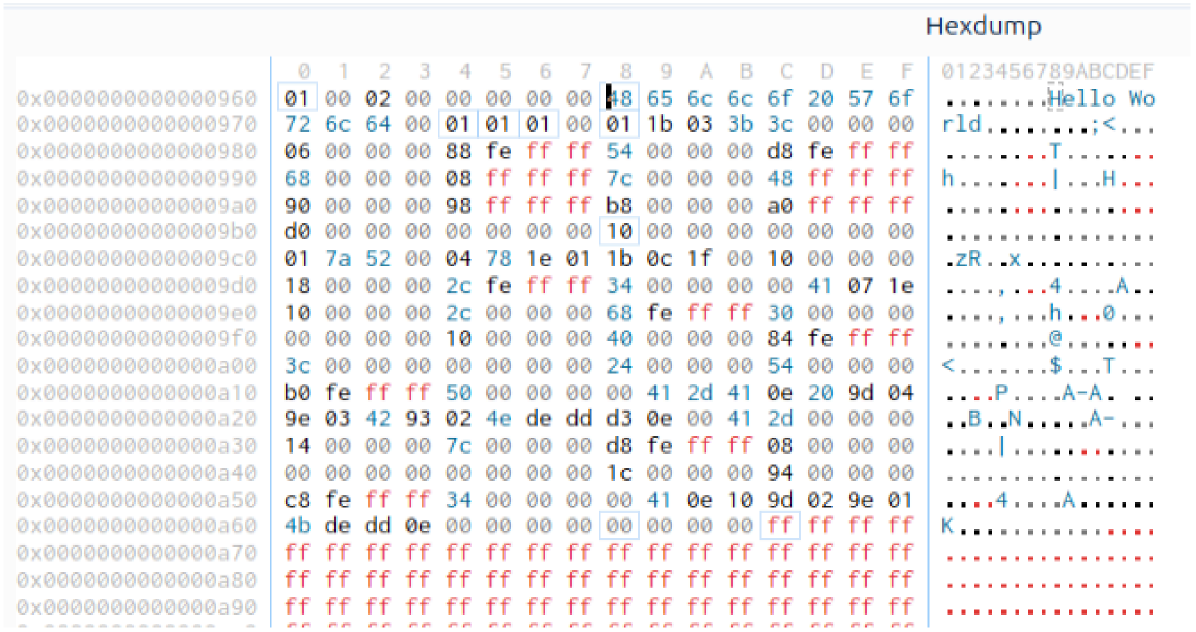
i. The leftmost area is one single column. It shows us a memory address with the least significant digit zeroed out (ex: 0x000008ac0).

ii. The center area has 16 columns, each corresponding to a hex digit. The hex digits are the headers for these columns running from 0 though F. This digit represents the least significant digit that is zeroed out in the leftmost column. The contents of this area are the contents of the compiled file represented as hex. Each grouping of 2 characters is 1 byte.

iii. The rightmost area shows us text characters that correspond to each byte in the center area.

12. Type (or paste) the memory address into the search bar at the top to find the bytes for the Hello World string. Note that the search bar shows `Type flag name or address here` if nothing is in it. Hit enter to commence the search.

13. Once found, the cursor in the middle section of the hexdump should indicate the memory location. Note that the cursor lands on the first byte of the Hello World message with the hex value 48. This value in decimal is 72. When referencing ASCII, 72 is the capital letter H. The other hex values for ASCII characters in `Hello World` follow it with `65 6c 6f 20 57 6f 72 6c 64`, and then a `00` since strings end with a null value. Note that the hex dump also shows you the character corresponding to each byte in the section on the right.



```
                                                                       Hexdump
                    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0x0000000000000960  01 00 02 00 00 00 00 00 48 65 6c 6c 6f 20 57 6f  ........Hello Wo
0x0000000000000970  72 6c 64 00 01 01 01 00 01 1b 03 3b 3c 00 00 00  rld........;<...
0x0000000000000980  06 00 00 00 88 fe ff ff 54 00 00 00 d8 fe ff ff  ........T.......
0x0000000000000990  68 00 00 00 08 ff ff ff 7c 00 00 00 48 ff ff ff  h.......|...H...
0x00000000000009a0  90 00 00 00 98 ff ff ff b8 00 00 00 a0 ff ff ff  ................
0x00000000000009b0  d0 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00  ................
0x00000000000009c0  01 7a 52 00 04 78 1e 01 1b 0c 1f 00 10 00 00 00  .zR..x..........
0x00000000000009d0  18 00 00 00 2c fe ff ff 34 00 00 00 00 41 07 1e  ....,...4....A..
0x00000000000009e0  10 00 00 00 2c 00 00 00 68 fe ff ff 30 00 00 00  ....,...h...0...
0x00000000000009f0  00 00 00 00 10 00 00 00 40 00 00 00 84 fe ff ff  ........@.......
0x0000000000000a00  3c 00 00 00 00 00 00 00 24 00 00 00 54 00 00 00  <.......$...T...
0x0000000000000a10  b0 fe ff ff 50 00 00 00 00 41 2d 41 0e 20 9d 04  ....P....A-A. ..
0x0000000000000a20  9e 03 42 93 02 4e de dd d3 0e 00 41 2d 00 00 00  ..B..N.....A-...
0x0000000000000a30  14 00 00 00 7c 00 00 00 d8 fe ff ff 08 00 00 00  ....|...........
0x0000000000000a40  00 00 00 00 00 00 00 00 1c 00 00 00 94 00 00 00  ................
0x0000000000000a50  c8 fe ff ff 34 00 00 00 00 41 0e 10 9d 02 9e 01  ....4....A......
0x0000000000000a60  4b de dd 0e 00 00 00 00 00 00 00 00 ff ff ff ff  K...............
0x0000000000000a70  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
0x0000000000000a80  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
0x0000000000000a90  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
```

Figure 1: Hexdump

11. Put Cutter into Write mode by selecting `File > Set mode > Write Mode`
12. Select the first byte of the string `Hello World` where you see 48 by clicking on it (be sure it is the 48 that corresponds to the string and not one of the other values of 48 as that value corresponds to an instruction as well, you can be certain by keeping an eye on the right most column). Right click on the hex value 48 and go to `Edit > Write String`.
13. In the dialogue box that opens, write `Bye` and confirm it. You will notice the first three bytes of the string will have changed.
14. Click and drag to highlight the remaining bytes of the `Hello World`.
15. Right click on the highlighted area and select `Edit > Write Zeroes`
16. In order to run the modified program, we will need to put Cutter back into `Read Only` mode. Click `File > Set mode > Read Only Mode` to do so.
17. Open a terminal at the location of `1.exe` and run the file to see that you have changed the message. The executable should now output `Bye` instead of `Hello World`.
18. When you close cutter, you do not need to save the project. Note that this will not reset the `1.exe` file back to how it was. It is now permanently modified.

3.2 File 2.exe

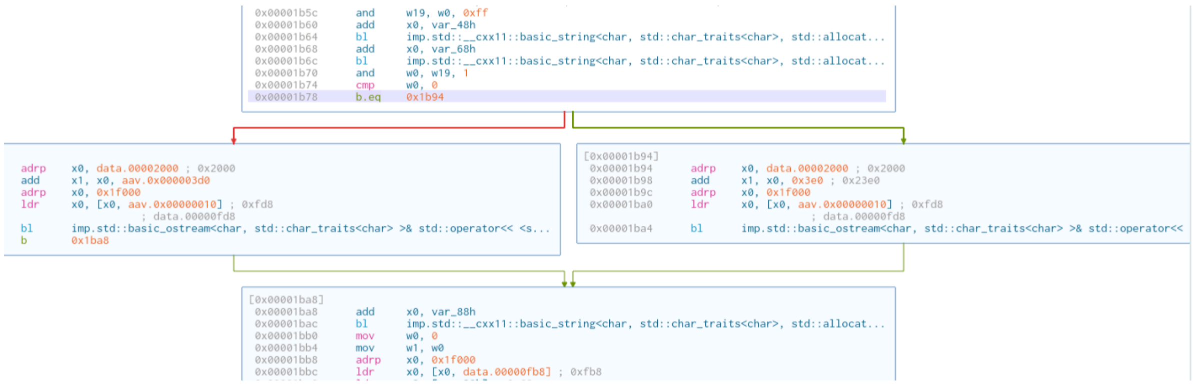
Now that you've gotten a brief tour of Cutter, you are left to explore file 2.exe with the goal of getting the SUCCESS message instead of the FAILURE message. Some guidance follows.

1. Run it to see what it does. You should see that it requires a password. An incorrect password resutls in a FAILURE message.
2. Open the file in Cutter to find the password. Hint: It is `obvious` and stored as a string that isn't quite as obvious.
3. Once you have found the password, see if it works by running the file again to get past it. If correct, you should get a success message.
4. Now, modify the password to be something else using the processes learned from File 1 and test it.

3.3 File 3.exe

File 2 showed one reason for why it is good to test passwords against hashes instead of storing them as plain text. File 3 instead, takes the input provided by the user and hashes it and compares it against the hash. Your goal in this file is to get past the password protection, but by modifying the program instructions instead of the string data. The instructions to do so follow.

1. Run it to see what it does. It should ask for a password. If you get it wrong, then you get a FAILURE message. If you somehow get it correct, then you should get a SUCCESS message.
2. Open the file in Cutter to find the hash of the password. Do not change it once you find it.
3. You may have noticed a very long string with a lot of the character `f`, this password is hashed and any password we provide will be hashed before we compare it. We can find another way around the password protection by modifying the jump condition.
4. Go to the Graph view of the program and be sure the `main` function is selected.
5. Notice that the program can take two separate paths. This indicates a jump (or branch for arm). One of those paths leads to a piece of the program that has a Success message located inside of it and the other has a Failure message inside. Note that in the `arm64` version of the program, you might have to chase the memory addresses indicated by the the add commands to find the actual text of the strings.



```
0x00001b5c      and     w19, w0, 0xff
0x00001b60      add     x0, var_48h
0x00001b64      bl      imp.std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat...
0x00001b68      add     x0, var_68h
0x00001b6c      bl      imp.std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat...
0x00001b70      and     w0, w19, 1
0x00001b74      cmp     w0, 0
0x00001b78      b.eq    0x1b94
```

```
adrp    x0, data.00002000 ; 0x2000
add     x1, x0, aav.0x000003d0
adrp    x0, 0x1f000
ldr     x0, [x0, aav.0x00000010] ; 0xfd8
                   ; data.00000fd8
bl      imp.std::basic_ostream<char, std::char_traits<char> >& std::operator<< <s...
b       0x1ba8
```

```
[0x00001b94]
 0x00001b94      adrp    x0, data.00002000 ; 0x2000
 0x00001b98      add     x1, x0, 0x3e0 ; 0x23e0
 0x00001b9c      adrp    x0, 0x1f000
 0x00001ba0      ldr     x0, [x0, aav.0x00000010] ; 0xfd8
                                 ; data.00000fd8
 0x00001ba4      bl      imp.std::basic_ostream<char, std::char_traits<char> >& std::operator<<
```

```
[0x00001ba8]
 0x00001ba8      add     x0, var_88h
 0x00001bac      bl      imp.std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat...
 0x00001bb0      mov     w0, 0
 0x00001bb4      mov     w1, w0
 0x00001bb8      adrp    x0, 0x1f000
 0x00001bbc      ldr     x0, [x0, data.00000fb8] ; 0xfb8
```

Figure 2: A jump depicted in graph view with arm64 assembly. Note that the SUCCESS message is located in memory at 0x23d0 and the FAILURE message is located at 0x23e0

5. Identify the line before the jump that is responsible for the jump. In `x86` assembly you should see `je` for jump if equal or `jne` for jump if not equal. In `arm` assembly you'll see `b.eq` for branch if equal or `b.ne` for branch if not equal.

6. Put cutter into write mode.
7. Right click on the jump instruction and click `Edit > Instruction` and change the current instruction to its opposite. Thus, for x86 machines, if it shows `je` change it to `jne`. For arm machines, if it shows `b.eq` change it to `b.ne`. Leave everything else about the instruction in tact.
8. Put cutter into Read Only mode.
9. Execute the file and provide an incorrect password. You should recieve the SUCCESS message this time. While you still don't know the password, you can still bypass it.

3.4 File 4.exe

File 4 is left for you to explore. Run the file to see that you get a countdown. Your goal is to get to the end of the countdown by either shortening the countdown by modifying the start time or by skipping the countdown all together. It could be helpful to know that `2,000,000` is `1E8480` in hex.

Jumps for arm

Note that for arm machines `b.gt` is for branch if greater than, `b.ge` is branch if greater than or equal to, `b.lt` is branch if less than, and `b.le` is for branch if less than or equal to.

Jumps for x86

Note that for arm machines `jg` is for jump if greater than, `jge` is jump if greater than or equal to, `jl` is jump if less than, and `jle` is for jump if less than or equal to.