Reverse Engineering March, 2023

Lab session 0x03

In this lab session, we will see some assembly code and disassemble a few ELF binaries.

1 Lab files

The files for this lab session are available at https://pwnthybytes.ro/unibuc_re/03-lab-files.zip and the password for the zip file is *infected*.

2 Tools we use (Windows and Linux)

Today, all the work will be done in the Windows and Linux environments. Make sure you have *IDA* installed for Windows.

2.1 Tasks: basic disassembly

The tasks today will make use of the compiler explorer Godbolt[1]. Using the gcc compiler write short sequences of code and check the resulting disassembly for:

1. Write a basic "hello world" C program. Compile it to the *hello* binary and then:
   (a) Run *readelf –program-headers hello* to see the ELF program headers.
   (b) Run *readelf –section-headers hello* to see the ELF program sections.
   (c) Pay attention to the following sections:
   - **.text** - containing the majority of code, both user-written code and boilerplate generated by the compiler.
   - **.init** - containing code usually generated by the compiler that is supposed to run before main() has been called.
   - **.fini** - containing code usually generated by the compiler that is supposed to run after main() has been called.
   - **.plt** - containing code generated by the compiler in order to call functions from libraries.
   - **.rodata** - containing Read Only Data used by the program (strings, constants, etc.).
   - **.data** - containing Read/Write Data, used for initialized variables, mutable strings, etc.
   - **.bss** - containing Read/Write Data, used for uninitialized global variables.
   - **.got / .got.plt** - (Global Offset Table) containing pointers used in library call resolution.
   - **.init_array** / **.fini_array** - containing pointers used in the code from the .init / .fini sections.

   (d) Compile and disassemble the *hello* binary using IDA in the following cases:
   - compile with debugging symbols (-g flag for gcc). Note especially the binary organization (code, data, relocations) and the IDA features (tabs, disassembly, graph view, navbar, xrefs, decompilation, symbols).
   - compile without debugging symbols (-s and/or -S flags for gcc). Note again the IDA features which are available now.
   - try to understand in IDA what is going on with the binaries *obscure* and *crackme* from Lab session 0x01.

[1] https://godbolt.org/

3 IDA cheat sheet

In Class 0x03 you have several references to IDA tutorials. Here we provide a brief overview of the main functionality that you will need.

Navigation

- To go into another function, double-click the function name either in the function sidebar (left pane) or in the IDA-View or Pseudocode view.
- Switching between IDA-View and Pseudocode: Press **Tab** to go to the exact assembly instruction for the current position. Alternatively, press **F5** to see pseudocode without pinpointing the above.
- Switching between Linear-View and Graph-View: Press **Space**.
- Press **Esc** to go to the previous view.
- To find usages of the current function/variable/item, right-click and choose **Jump to xref..** or press **x**.

Renaming/Redeclaring

- Changing the signature of a function: Right-Click the signature in the IDA-View or Pseudocode View and click **Set item type**. Keyboard shortcut: **y**.
- Changing the type of a variable: Right-Click the variable in the IDA-View or Pseudocode View and click **Set lvar type**. Keyboard shortcut: **y**.
- Changing the name of a function/variable: Right-Click the function/variable in the IDA-View or Pseudocode View and click **Rename global/lvar item**. Keyboard shortcut: **n**.

Reorganizing the stack variables

- To change a stack variable into something else (smaller, bigger, structure, turn into an array) first double-click on the variable to go into the Stack frame of that specific function. Observe how much space you have for your desired actions. Right-click on the variable and click **Set type**.
- Note that when turning into an array it is ideal to first change the variable into the array unit (e.g., if you want to change a stack space into int v[30], and v is currently char, first turn v into an int) and then right-click and choose **Array**. It will be possible to now see some suggestions from IDA regarding the ideal/maximum array size.

When in doubt Right Click!(or hover)

4 Lab tasks: disassemble with IDA

4.1 Reverse engineering with spoilers

Usually, when reverse engineering, all we have is a binary. Starting from it, we need to reconstruct (mainly through guessing/inferences) what the function names could be, what the variables are used for, and what the program does as a whole.

You have a binary, *task1*, and also its corresponding source code, *task1.c*. Using the stripped binary, you will simulate normal reverse engineering by using the source code (instead of guessing).

Your task is to create a near-original replica of the original source in the IDA interface by:

1. Renaming/retyping the 4 functions in the source code (aside from **main()**) (3p)
2. Renaming/retyping the stack variables in **setup()** and **main()**. (1p)
3. Renaming/retyping the stack variables (including the arrays) in **chance()** and **gen_rand_string()**. (2p)

4.2 Statically linked *crackme* - graybox analysis (dynamic + static)

In this task, you will learn to navigate through functions in a statically linked and stripped *crackme*.

Since the binary has a whopping 783 functions detected, you do not have the time or motivation to go through all of them. As such, you need to approach the problem in a clever and elegant way:

1. Run the program once and take note of any strings. Go to the **.rodata** segment (**Ctrl-s**) and find any/all of the strings. Using the **xref** functionality, determine where the **main()** function is. (1p)
2. Rename all the functions in **main()** and determine the password-checking function. (1p)
3. In the password-checking function, observe how the correct password is generated; we want to make this function more readable.
4. Go to the location of any **word_.....** variable in IDA-view and find the location of the start of the alphabet and redeclare that address as a wide C string (**Edit→Strings→Unicode**).
5. Again, in the password checking function, observe how the right-hand side looks now. Redeclare the alphabet with the "const" modifier at the beginning. This should collapse the function and reveal the correct password. Finally, check that the password is accepted. (2p)

4.3 Data Structures

In this task, you will learn to use the Structures functionality of IDA.

Only the simplest programs are written without any sort of data structure in mind. Even basic OOP features are implemented using structures; classes themselves are also compiled as structures. However, after compilation, structure and type information is lost (if we do not have debugging symbols) but we can still observe repeated access patterns and infer what various structures might have looked like.

Look at the code in **main()** and the **password checking function**, analyze the access patterns, and verify that it matches the linked list structure below.

Perform and check the following tasks:

1. Use the **Structures** tab and create the following list structure (also declare field_8_next as a struc_1* pointer) (2p)

```
00000000 struc_1            struc ; (sizeof=0x10, mappedto_8)
00000000 field_0_idx        dd ?
00000004 field_4            db ?
00000005                    db ? ; undefined
00000006                    db ? ; undefined
00000007                    db ? ; undefined
00000008 field_8_next       dq ?                    ; offset
00000010 struc_1            ends
```

2. In **main()**, cast the buffer returned from malloc and the head of the list to this struct type and propagate in the password checking function, renaming and retyping where necessary. (2p)
3. Describe what the code does and figure out the correct password. (2p)