Reverse Engineering March, 2023

Lab session 0x04

In this lab session, we will analyze processes by using debugging techniques.

1 Lab files

The files for this lab session are available at https://pwnthybytes.ro/unibuc_re/04-lab-files.zip and the password for the zip file is *infected*.

2 Tools we use (Windows)

In the Windows environment, you need to download and install x64dbg[1].

2.1 Tasks: dynamic analysis with x64gdb

To find out what an unknown executable (malware or otherwise) does, we plan to use x64dbg on Windows and analyze the actual instructions executed by the CPU. Since there is a GUI interface, it is much easier to navigate (if you know what you are looking for).

In the main window you can see:

- tabs;
- CPU registers;
- current parameters (when calling another function);
- stack dump;
- memory dump.

The x64dbg tabs are (only the ones we need so far):

- CPU: shows the assembly code in a linear fashion. This can be changed:
  - To graph view: pressing **g**
  - To linear view again: pressing: **alt-c**
- Graph: shows the graph view above;
- Breakpoints: shows various breakpoints currently set and allows you to add/delete or enable/disable them;
- Memory map: shows the current virtual address space and mappings in the program. You can search for byte patterns or strings here;
- Call stack: shows how the program arrived at this particular instruction (the sequence of function calls).

The x64dbg basic commands are:

- Clicking in the CPU tab and then pressing **ctrl-g** allows you to navigate to arbitrary addresses in the memory;

[1] https://x64dbg.com

- After navigation, you can return to the current instruction by pressing * or right-clicking and selecting **Go to→Origin**;
- You can set a breakpoint anywhere by pressing **F2** at that address or right-clicking and selecting **Breakpoint→Toggle**;
- To see where a register points to, right-click it and select **Follow in Dump**. The memory contents will appear in the lower window titled **Dump**;
- To edit a register, double-click it. To toggle a CPU flag, double-click it;
- To move the current program counter to another instruction, right-click it and select **Set New Origin here**.

3 Tools we use (Linux)

In the Linux environment, you need to install gdb and peda using the following:

```
$ apt-get install gdb git
$ cd
$ git clone https://github.com/longld/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

3.1 Tasks: dynamic analysis with gdb/peda

We will use the following gdb commands.

- Starting the executable

```
(gdb) run
Starting program: /tmp/example/ret
```

- Breakpoint on address 0x80484f1

```
(gdb) b *0x80484f1
Breakpoint 3 at 0x80484f1: file ret.c, line 10.
```

- Deleting all breakpoints

```
(gdb) delete
Delete all breakpoints? (y or n) y
```

- Viewing all breakpoints

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
4       breakpoint     keep y   0x080484f1 in main at ret.c:10
```

- Deleting a specific breakpoint

```
(gdb) info break
Num     Type           Disp Enb Address    What
4       breakpoint     keep y   0x080484f1 in main at ret.c:10
5       breakpoint     keep y   0x080484c9 in main at ret.c:7
6       breakpoint     keep y   0x080484c9 in main at ret.c:8
(gdb) delete 5
(gdb) info break
```

```
Num     Type           Disp Enb Address    What
4       breakpoint     keep y   0x080484f1 in main at ret.c:10
6       breakpoint     keep y   0x080484c9 in main at ret.c:8
```

- Running until the next breakpoint: **continue**
- Running until the return of the current function: **finish**
- Stepping to the next instruction: **step**
- Stepping to the next instruction without entering functions: **next**
- Viewing registers: **context reg** or **print $rax**
- Dumping memory:

```
gdb-peda$ dump memory blabla.out 0x00400000 0x00400020
```

- Examining a fixed number of bytes from memory from a given location:

```
gdb-peda$ hexdump 0x00400000 20
0x00400000 : 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00    .ELF............
0x00400010 : 02 00 3e 00                                        ..>.
```

4 Lab tasks: debugging processes

4.1 Task: debugging in Windows

We will continue the puzzle-based tasks with a Windows PE (exe) file. The executable asks for a password and after doing some checks on it, alerts you if you are correct or wrong. However, the correct password is not in the binary such that you will have to investigate how the password-checking works.

It is possible to completely reverse the program using static analysis alone. However, there are three complex functions that would normally take additional time. As we want to illustrate how dynamic analysis can speed up static analysis, you are not allowed to look into these three functions at all:

- sub_1400011D0
- sub_140002C60
- sub_140001010

Tasks:

- Open the binary in IDA and identify the password checking function (same procedure as in lab session 0x03) and the final **if** condition that verifies whether the password is good or not. Also, figure out which function is sprintf; (2p)
- Open the binary in x64dbg and set a breakpoint at the function call in the **if** condition:
  - Note that after starting, x64dbg will set some standard breakpoints which you probably do not need. Note the state of the program (Paused/Running) in the lower-left corner;
  - Also note that on Windows, the calling convention is different; see the call window on the right;

  - To do this, copy the address from IDA and navigate to it in x64dbg after the program has started. See **x64dbg basic commands** above;
  - Identify which parameter is the result from user input and what it is compared against. (1p)
- Using **Set New Origin here** or by modifying the corresponding CPU flag manually, make the program branch into the "Correct password" part; (2p)
- Find out what the three restricted functions mentioned above do by treating them as a black box. Use dynamic analysis and: (2p)
  - enter "password" into the text field (we want to find out what the transformation does by serving it a common input);
  - getting the function output from the debugger;
  - searching on the Internet for that hex string.
- Figure out the correct input. (2p)

4.2 Task: debugging in Linux

The Linux binary for today is a bit more convoluted as it has some anti-debugging and anti-disassembly mechanisms which you will learn to bypass.

Approaches we learned so far:

- Run the program once, feed it a random input and take note of any strings for later use;
- Try the approach in lab session 0x03 by looking for xrefs to the strings. What do you observe? What might be the cause?
- Now try the approach in lab session 0x01 by running with ltrace. Does it work?

We need a different approach:

- Find the anti-debugging mechanism by searching for the **ptrace call** in IDA. Notice the condition for program exit;
- Then, in gdb/peda, set a **breakpoint** on the address right after the call;
- When the debugger stops there, modify the corresponding register such that when continuing execution under the debugger, the program does not exit. (2p)

You have successfully bypassed the anti-debugging mechanism! Continue the analysis:

- Using IDA, analyze the main function:
  - **scanf** gets the user input;
  - The third function is called with the user input as its parameter but going into it we see it is just garbage code, impossible to analyze in its current state;
  - The second function actually decrypts the code for that function.
- Go into the decryption function and pay attention to the **for loop**. Determine the start address and the end address for the decryption process;
- Then, in gdb, set a **breakpoint** after the decryption finishes (right before the decrypted function is called) and **dump** the decrypted memory. (2p)

You now have the third function decrypted, but in binary form. For the following, if you do not have IDAPython (e.g., IDA Trial), use this IDC guide[2].

- Using get_byte and patch_byte[3] in the Python scripting interface (File→Script Command with Scripting Language set to Python), decrypt the bytes of the function. You can either use:
  - Only **patch_byte** with the contents of the dumped memory;
  - Or **get_byte**, replicate the decryption and then **patch_byte**.
- The end result should be fully decompiled. (3p)

[2]https://www.hex-rays.com/products/ida/support/tutorials/idc/decrypt/
[3]https://www.hex-rays.com/products/ida/support/idapython_docs/ida_bytes.html