Binary Security - Exercise 6

Stephanie Briere Americo / 22579627

February 5, 2019

1 Basic Reverse Engineering (Aufgabe 1)

I used the 64-bits version to do this question.

1.1 Challenge 1

Using `strings` we can get several assembly and libc -related strings, which can be easily identified is most cases. When it is not, comparing both 32 and 64 -bit files we can eliminate the strings which differ, since we can kind of conclude that it is also architecture related.

With this, the help and results strings are discovered (e.g. "Congratulations..."). So we can find the password for challenge 1 is `p4ssw0rd`.

1.2 Challenge 2

After checking the assembly for the function `checkMedium` using `objdump`, we can find a sequence of operations with a value in the `ecx` register before calling `sprintf` and `strcmp`. Using `gdb` we can check that the value of the register `rcx` (altered by the operations on `ecx`) right after the arithmetical operations is `1337`, which turns out to be the password.

1.3 Challenge 3

By running the program with `gdb` and setting a breakpoint right before the `strcmp` call at address 0x400771, we can use `x/-22sg 0x7fffffffd950` to analyze the content of the stack (starting on `rbp` at address 0x7fffffffd950 until `rsp`). Analyzing the stack is worth since the `checkHard` function is recursive, as we can see at the assembly code for it.

The stack has the "1337" string for Challenge 2 and other bytes that do not behave like a string. But we can find `0x7fffffffd937: "zyvgjqpc"` on the stack. Trying `zyvgjqpc` we can find out that this is the password for the Challenge 3.

2 Basic Exploitation (Aufgabe 2)

2.1 Crash both programs

Using any string with 266 or more characters crashes the 32-bit version. Similarly, using any string with 274 or more characters crashes the 64-bit version. The strings used are in `input_crash_{32,64}` files. The `output_crash_{32,64}` files have the execution with the mentioned strings.

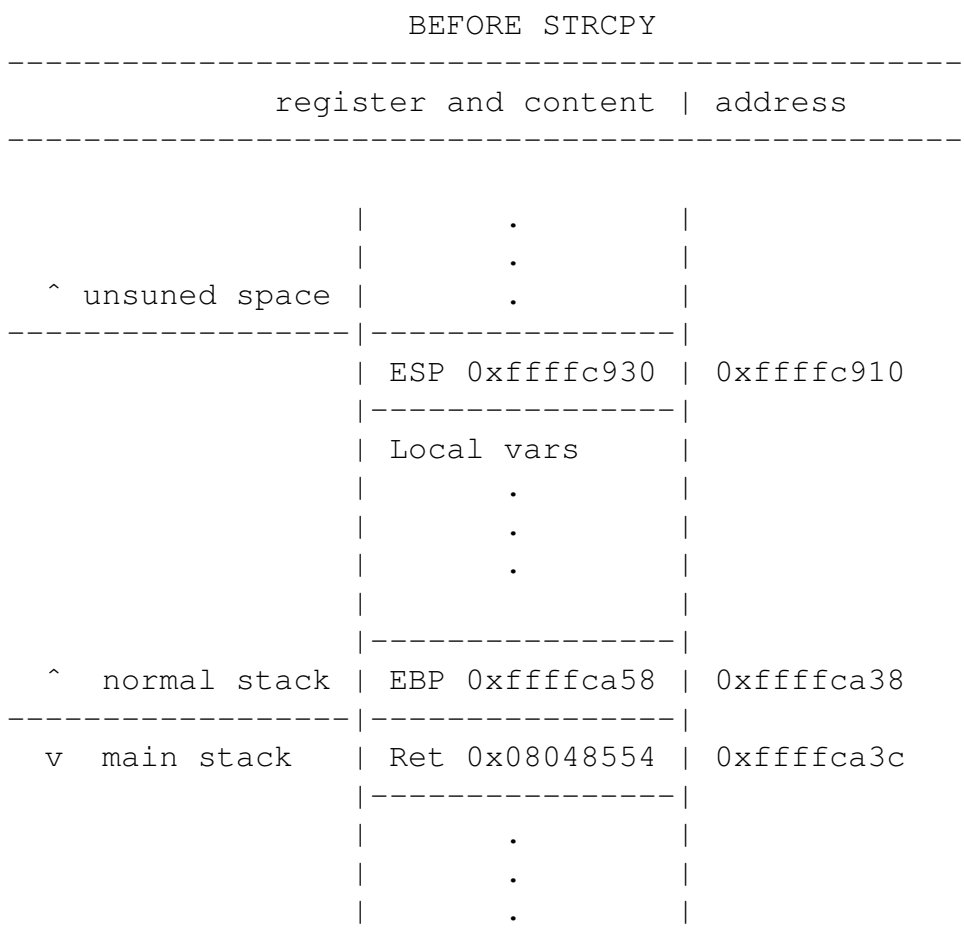
2.2 Explaining the crash

Analysis using the 32-bit version disassembly (available at `assembly32`):

The program calls a function called `normal`, which do the job of calculating the string length (with `strlen`) and print the output. Turns out that this function stores the string on a fixed offset on the stack. The address is passed to the `strcpy` call at `0x80484d2`:

```
80484d2:  e8 69 fe ff ff call 8048340 <strcpy@plt>
```

This is the function which copies the string to this fixed offset on the stack. If the string is big enough, it overwrites all the stack and also the memory beyond the stack "owned" by the function, as follows:



As we can see, it is possible to overwrite the EBP register (which can cause segfaults for accessing wrong memory regions at the function returning), and if the string is even longer, the return address pointer is also overwritten, which will direct the program flow (RIP) to other pieces of code at the function returning.

If the memory pointed by this register is not a valid region of code, the program will also crash. This is what happens on the following items: by modifying the return pointer to call the function `secret`, we destroy the stack, and when the `secret` function is returning, it access bad memory regions and crashes the program.

2.3 Running `secret` in 32-bits

Using the assembly dumped before, we can search for the label `secret`.

Using the above information about the stack, return address pointer and `secret` address, we can build a string that contains the right bytes at its end, overwriting the return address pointer, pointing it to the address of the `secret` function. (script at `secret/secret.32` build this string and calls the `hack-me.32` binary)

2.4 Running `secret` in 64-bits

Following the same idea from the previous item, the script at `secret/secret.64` were built. It generates the right string based on the `secret` function address and call the `hack-me.64` binary.

3 Game Labrys (Aufgabe 3)

3.1 Cracking

3.1.1 Serial

By using `gdb`, `PEDA`[1] (a GDB enhancer for assembly-related tasks) and `radare2`[2] (a suite of reverse engineering tools) I was able to better visualize the workflow of the 64-bits assembly code. Then I could got some insights that special chars like `/` and `@` and numbers and uppercase letters were somehow involved in the first loop that checks the key.

To find the key, the process was basically (0) setting a key with unique values at each char (after discovering that it should contain 29 chars), (1) setting a breakpoint before a jump to the `licenseerror` function, (2) check the registers and compare which key char was being compared and with what value, (3) check the operations that culminated on the target value (in some cases there were XORs, ORs and ANDs involved), (4) adjust the key to fulfill the requirements for the current block of checking, (5) if the block of code was successfully passed, then we can go to the next block and check another portion of the key.

This way, I could reach a valid key, stored on `cracking/license.key` file.

3.1.2 Keygen

Using the analysis of the previous item, I was able to check that the key is made by five blocks of uppercase letters or numbers. Then, the first block should be equal to `A` when all the chars are `XOR`ed. The second block have the last char equal to the `XOR` of the first three chars, and the fourth char is free. The third block must have the last char equal to the `AND` of the other four. The fourth block is checked this way: first, third and fifth chars are `OR`ed and the result is `AND`ed with `0xf`; this should be equal to the `XOR` of the second and fourth chars. The last block should end with an `X`, the second char must be the first plus one, and the fourth must be the third minus one.

The script generates all the blocks of five letters/numbers (five lists of blocks) and do the checks to see if it is a valid block. Then, with all the blocks calculated (it uses 250Mb RAM) it builds a string with random blocks of each list.

3.1.3 Patch

Using `radare2` I could change binary (64 bits) so the key was not necessary. Since we cannot change the file size due to the static and relative addresses inside the binary, my solution was to change a instruction at the beginning of the binary. The instruction

`40500e:  75 16 jne 405026 <main+0x58>`,

responsible for jumping the `exit` call if there is no license key file, was changed to

`40500e:  e9 33 02 00 00 jmpq 405246 <main+0x278>`,

so it skips all the license verification.

Unfortunately, I could not find out 2 problems on exiting the game: sometimes it crashes with segfault and sometimes it crashes with an illegal instruction error. This was the best modification I could do, other changes on the beginning of the main function were also tried, but they crashed (with segfault) randomly even before the game start, so I chose this version (available at `cracking/labrys.64` (patch for `bspatch` on `cracking/patch.64`).

[1] `https://github.com/longld/peda`

[2] `https://github.com/radare/radare2`

3.2 Cheating

The patches and binaries (64 bits) for this cheats (separately) are on the `cheating/` folder.

3.2.1 Wallhack

On the `Wall::collide` method the instruction

```
414215:  73 5c jae 414273 <_ZN4Wall7collideEPfi+0x1af>
```

was changed to

```
414215:  73 63 jae 41427a <_ZN4Wall7collideEPfi+0x1b6>
```

so we can walk through walls, but not grounds, stairs, artifacts and so on.

3.2.2 Flyhack

On the `Player::save_state` method the instruction

```
41d4a5:  74 09 je 41d4b0 <_ZN6Player10save_stateEv+0x66>
```

was changed to

```
41d4a5:  75 09 jne 41d4b0 <_ZN6Player10save_stateEv+0x66>
```

so the character can fly.

3.2.3 Speedhack

On the `Player::save_state` method the instruction

```
41d4c8:  74 13 je 41d4dd <_ZN6Player10save_stateEv+0x93>
```

was changed to

```
41d4c8:  75 13 jne 41d4dd <_ZN6Player10save_stateEv+0x93>
```

so the character speed was increased.

3.3 Exploitation

3.3.1 Shellcode Injection

Using the slides and [1], I build a file with a big `NOP` sled, the shellcode (from slides for 64 bits and from [1] for 32 bits), and after running it several times, I could drop a shell in some of the executions. Inside `gdb`, however, it is pretty much easier to get the jump to the `NOP` sled. (files `exploitation/labyrinth.{32,64}`)

[1] `https://dhavalkapil.com/blogs/Shellcode-Injection/`

3.3.2 Return Oriented Programming

Using the files for the previous item (which have the `NOP` sled and string of `A`s to overflow the stack), I just changed the `NOP`s to `A`s since we don't want to execute the stack anymore, and changed the very end of the file, which has the address to overwrite the return pointer address.

Using `ropper --chain "execve cmd=/bin/sh" --badbytes 000a --file labrys.64` I got a python script (at `exploitation/ropchain64.py` which generates the chain containing the successive addresses to jump, which will result in the shellcode to be executed (the chain alone is at `exploitation/ropchain.64`. By concatenating the `A` string with the chain generated by the python script, the file `exploitation/labyrinth_ropchain.64` was generated. When using this file as the `labyrinth` file of level 5, we can drop a shell successfully.