University of Rennes - Cyberschool

Master 1 - Cybersecurity

Report - software-security-project

Code injection

AUTHOR:
Faical Sid Ahmed RAHMANI

2022 - 2023

Contents

1 Introduction

The objective of this project is to demonstrate the ability to modify an ELF file by injecting and executing new code. The project is divided into seven challenges, each of which presents a different task to accomplish. I have defined a separate function for each challenge, which will be called in the main function. By breaking down the project into these smaller challenges, we can focus on each task and ensure that we are moving towards our ultimate objective of injecting and executing new code in the ELF file.

My code is organized into two files: The first file, `my_elf.h`, contains the necessary structures and definitions for working with ELF files, such as the `Elf64_Ehdr`, `Elf64_Phdr`, and `Elf64_Shdr` structures. The second file, `isos_inject.c`, contains the functions necessary for executing the seven challenges we'll be tackling. Additionally, there are two assembly files, `entry-point.asm` and `hijack-got.asm`, which contain the code to be injected if we want to modify the entry point or hijack the Global Offset Table respectively.

2 Challenge (1): Initialize ELF file for reading

In this Challenge, i use the `BFD (Binary File Descriptor)` library to analyze the ELF file and verify its validity using `check_elf` function that checks if the given ELF file is a valid 64-bit executable. It initializes the BFD library, opens the ELF file, and checks if the binary is of format ELF, of architecture 64-bit, and executable. Finally, it closes the file.

I defines a struct `arguments` to store the values of the command-line arguments, including the name of the ELF file, a binary file to be injected, the name of the newly section, the base address of the injected code, and a Boolean that indicates whether the entry function should be modified or not. I also define a `parse_opt` function that is called for each command-line option or non-option argument. The function handles different options and arguments using a switch statement and stores the values of the arguments in the `arguments` struct that i created before. My program defines an instance of the `argp` structure that specifies the parameters for the command-line argument parser. It also specifies the options supported by the program, the arguments documentation, and the program documentation.

3 Challenge (2): Find the PT_NOTE segment header

This challenge provides a function called `check_ptnote` that takes an elf file as input and checks whether the given file has a `PT_NOTE` segment.

My function first opens the file using the `open` system call and then retrieves information about the file using the `fstat` system call. It then maps the entire file into a readable memory region using the `mmap` system call, which allows us to access the content of the file as if it were in memory. After that, the function retrieves a pointer to the ELF header `(Elf64_Ehdr)` from the mapped memory region, and reads the number of program headers `(e_phnum)` from the header. Next, the function also retrieves a pointer to the first program header `(Elf64_-Phdr)` from the mapped memory region using the following formula:

```
......
Elf64_Phdr *phdr = (Elf64_Phdr *)(void*)((char*)map_addr + ehdr->e_phoff);
......
```

The code first casts the `map_addr` pointer to a `char` pointer and adds the value of `e _phoff` to it. This creates a pointer to the start of the program header table in the mapped memory region. Then, it casts this pointer to a pointer to `Elf64_Phdr` to indicate that the memory region starting from this address contains program header structures.

The function then iterates over the program headers to find the first program header of type `PT _NOTE`. If such a program header is found, i store its index, otherwise, i set the index to -1. Finally, my function `unmaps` the memory region using the `munmap` system call and closes the file using the `close` system call. It returns the index of the first program header of type `PT _NOTE`, or -1 if not found.

4 Challenge (3): Code injection

In order to append the injected code to the binary, i defined a function called `inject_code` that takes three arguments: `elf_file, binary_file, and base_address` where the injected code should be placed in the ELF file.

The function first opens the ELF file in binary append mode and the injected binary file for reading in binary mode using `fopen` system call. it then uses `fseek` to move the file pointer to the end of the injected binary file to get its size using `ftell` and then set the file pointer back to the beginning of the file using `rewind` system call. The function saves the byte offset where the injected code will be written to the ELF file using `ftell` on the ELF file pointer and assigns the result to the global variable `injected_offset`.

My function then allocates a buffer to store the injected code from the binary file, and puts the code to be injected into the buffer using `fread` system call. Then it writes the code from the buffer to the ELF file using `fwrite`. Next, The function frees the buffer using `free` and closes the ELF file and the binary file using `fclose` system call.

Finally, the function calculates the difference between the injected offset and the base address modulo 4096 using the following formula:

```
......
long diff = (injected_offset % 4096) - (base_address % 4096);
......
```

The formula calculates the difference between the injected offset and the base address modulo 4096, ensuring that the difference is a multiple of 4096. If the difference is not a multiple of 4096, it means that the injected code is not properly aligned, and the formula returns a value that can be used to adjust the address so that it is properly aligned.

5 Challenge (4): Overwriting the section header

For this part of the project, we will overwrite the header of the `.note.ABI-tag` section that is part of the `PT_NOTE` segment. To do this, i defined a function that takes as input: a pointer to the ELF header `ehdr`, a pointer to the section header table `shdr_tab`, a pointer to the string table section `shstrtab` containing section names, and the `base_address` where the injected section will be loaded.

The function loops through all the section headers to find the section header with the name `.note.ABI- tag`. Once it finds the concerned section header, it saves its index and modifies its fields to describe the injected section. The modified fields include the type of the section, the virtual address where the section will be loaded `(sh_addr)`, the offset of the section in the file `(sh_offset)`, the size of the section `(sh_size)`, and the alignment of the section `(sh_addralign)`. It also sets the `SHF_EXECINSTR` flag in the section header to indicate that the section contains executable instructions.

Before calling this function and the functions in challenges 5, 6, and 7, i first mapped the entire ELF file into a readable and writable and executable memory region using the `mmap` function:

```
......
void *map_addr = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED, fd_elf
    , 0);
if (map_addr == MAP_FAILED) {
  errx(EXIT_FAILURE, "Failed to map the file");
}
......
```

The type of mapping is `MAP_SHARED` which specifies that the changes made to the mapped memory will be visible to other processes that map the same file, and will also be written back to the file automatically.

6 Challenge (5): section headers calibration

The goal of this challenge is to reorder section headers by section address and set the name of the injected section

6.1 Reorder section headers by section address

To complete this challenge, i defined two functions: `swap_section_headers` and `reorder _- section_headers`. The `swap_section_headers` function takes two pointers to the section headers and swaps their contents. `The reorder_section_headers` function reorders the section headers of the ELF binary. It takes as input, a pointer to the ELF header structure, a pointer to the section header table, and the index of the injected section in the section header table.

The function first stores the old index of the injected section. It then compares the address of the injected section with the address of its direct neighbors to decide whether it

should be moved right or left. If the injected section is less than its left neighbor, it needs to be moved to the left. If it is greater than its right neighbor, it needs to be moved to the right. The function then swaps the injected section header with its neighbors until it is in the correct position. It updates the `sh_link` field of all section headers based on the direction of the swap:

```
......
  if (right) {
    for (int i = 0; i < ehdr->e_shnum; i++) {
      if (shdr_tab[i].sh_link != 0 && (int)shdr_tab[i].sh_link > old_injected_sh_index &&
                                      (int)shdr_tab[i].sh_link <= injected_sh_index) {
        shdr_tab[i].sh_link--;
      }
    }
  }

  if (left) {
    for (int i = 0; i < ehdr->e_shnum; i++) {
      if (shdr_tab[i].sh_link != 0 && (int)shdr_tab[i].sh_link >= injected_sh_index &&
                                      (int)shdr_tab[i].sh_link < old_injected_sh_index) {
        shdr_tab[i].sh_link++;
      }
    }
  }
......
```

According to the direction of our swap, the function decrements or increments the `sh_link` field of all section headers whose value falls within the shift interval.

Finally, the function returns the new index of the injected section after reordering.

6.2 Set the name of the injected section

I defined a function that sets the name of the injected section. It takes four arguments: the file descriptor of the ELF file to modify, a pointer to the section header string table, a pointer to the contents of the string table, and the new name to give to the injected section.

First, the function gets the size of the new section name and the size of the `.note.ABI-tag` string and then checks that the new section name has a smaller length than `.note.ABI-tag`. Otherwise it returns -1. Next, the function searches for the offset of `.note.ABI-tag` into the string table:

```
......
  char *note_abi_tag_address = shstrtab;
  while (strcmp(note_abi_tag_address, ".note.ABI-tag") != 0) {
    note_abi_tag_address += strlen(note_abi_tag_address) + 1;
  }
  int note_abi_tag_offset = note_abi_tag_address - shstrtab;
......
```

The function iterates over the contents of the string table until the string `.note.ABI-tag` is found. The offset of the string is then calculated as the difference between the address of the string and the address of the start of the string table. The function then gets the offset to the start of the string table, and it calculates the offset at which to write the new section name by adding the offset of `.note.ABI-tag` in the string table to the offset of the string table

in the ELF file. It then seeks the file descriptor to that offset and writes the new section name.

Finally, the function returns 0 if the section name was successfully set, or -1 if the new section name is larger or equal to the old one.

7 Challenge (6): overwriting PT_NOTE header

For this challenge, i defined a function named `modify_ptnote` that modifies the program header fields of the `PT_NOTE` segment specified by the `ptnote_index` parameter.

The `p_type` field is set to `PT_LOAD`, to indicate that the segment should be loaded into memory during execution. The `p_offset`, `p_vaddr`, and `p_paddr` fields are set to `injected_offset` and `base_address`, respectively, specifying the file offset and virtual and physical addresses of the segment. The `p_filesz` and `p_memsz` fields are set to `injected_size`, indicating the size of the injected section in the file and in memory. The `p _flags` field has the `PF_X` flag, indicating that the segment is executable. The `p_align` field is set to 0x1000, specifying the memory alignment of the segment.

8 Challenge (7): Execute the injected code

For this part, i used `hijack` function to modify the ELF binary file by either changing its entry point or hijacking a GOT entry.

8.1 Assembly instruction for injected code

In my assembly code, I added a section called `.data`, which is used to define a string variable called `msg` that contains the message `"Je suis trop un hacker"`. I also defined a constant called `msg_len` that contains the length of the `msg` string.

```
......
SECTION .data
  msg db "Je suis trop un hacker", 10, 0
  msg_len equ $ - msg
......
```

Finally, I added code to invoke the `write` system call. This code loads the length of the `msg` string, and the address of the `msg` variable, the value 1 `(stdout)`, and the `syscall` number for `write` to indicate that the write system call should be invoked, which causes the message to be printed.

```
........
    ; write syscall
    mov rdx, msg_len ; length of msg
    lea rsi, [rel msg] ; message to display
    mov rdi, 1 ; stdout
    mov rax, 1 ; syscall number for write
    syscall ; invoke syscall
........
```

8.2 Entry Point Modification

If the `modify_entry` parameter is `true`, `hijack` function modify the entry point of the ELF file by setting the `e_entry` field of the `ELF` header structure to the specified `base_address`.

In order to end by calling the original entry point function, i used the `readelf` command to identify the address of the original entry point in the executable:

```
$ readelf -h date
........
  Type:                              EXEC (fichier exécutable)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Adresse du point d'entrée:                 0x4022e0
.......
```

Then, i wrote assembly code line that pushes the address of the original entry point onto the stack and uses the ret instruction to return to that address:

```
........
    ; return
    push 0x4022e0 ; pushes the address of the original entry
    ret
.......
```

8.3 Hijacking GOT Entries

In order to hijack the Global Offset Table `(GOT)` of the `date` program, i did the following steps:

First, I used the `ltrace` command to run the date program and identify which function to hijack, and i choosed to hijack `fputc`.

Then, i used the `objdump` command to disassemble the `date` binary and search for the call to **fputc@plt**.

```
$ objdump -d date | /usr/bin/less
......
00000000004018d0 <fputc@plt>:
  4018d0:       ff 25 3a e8 20 00       jmp    *0x20e83a(%rip)        # 610110 <
    __gmon_start__@plt+0x20e5b0>
  4018d6:       68 1f 00 00 00          push   $0x1f
  4018db:       e9 f0 fd ff ff          jmp    4016d0 <__ctype_toupper_loc@plt-0x10>
......
```

We can notice that `fputc@plt` jump to the address in the `.got.plt` section corresponding to this entry, which is `610110`.

In order to locate the offset of this entry in the `.got.plt` section, i used the `objdump` command again with the `-s` option and specified the `.got.plt` section.

```
$ objdump -s date --section=.got.plt

date:     format de fichier elf64-x86-64
Contenu de la section .got.plt
610000 28fe6000 00000000 00000000 00000000  (.`.............
610010 00000000 00000000 e6164000 00000000  ..........@.....
610020 f6164000 00000000 06174000 00000000  ..@.......@.....
610030 16174000 00000000 26174000 00000000  ..@.....&.@.....
......
```

We can notice that the `.got.plt` section starts at address `610000`, so the offset of the `fputc` entry in `.got.plt` is:

$$610110 - 610000 = 110.$$

Finally, we look up the offset of the `.got.plt` section in the `date` binary using the `readelf` command.

```
$ readelf -S date --wide
......
[24] .got.plt          PROGBITS        0000000000610000 010000 000258 08  WA  0   0  8
......
```

I found that the offset is `10000`. Adding the offset of the `fputc` entry (110) to the offset of the `.got.plt` section (10000) give us the final offset of the fputc entry in the `date` binary, which is 10110.

If the `modify_entry` parameter is `false`, the function will hijack a GOT entry in the ELF file. It will first seek to the offset of the `GOT` entry that has already been computed, and then write the base address of the injected code to that offset using the `write` system call.

9 Execution

In order to test the functionality of my code, I will show how to change the behavior of the program in two different ways: by modifying its entry point and by hijacking the `fputc` function in its Global Offset Table.

To modify the entry point, I will use the assembly code in the file `entry-point`, which simply displays the message "Je suis trop un hacker" to the console and then jumps to the original entry point of the program:

```
$ ./isos_inject date entry-point .rahmani 493445 true
$ ./date
Je suis trop un hacker
ven. 28 avril 2023 16:18:34 CEST
```

Next, I will modify the program's behavior by hijacking the `fputc` function in its GOT. I will replace the `fputc` function in the GOT with my own function in the file `hijack-got`, which will print the message "Je suis trop un hacker" each time the program tries to call `fputc` function:

```
$ make update-date
$ ./isos_inject date hijack-got .rahmani 493445 false
$ ./date
Je suis trop un hacker
Je suis trop un hacker
Je suis trop un hacker
Je suis trop un hacker
Je suis trop un hacker
Je suis trop un hacker
Je suis trop un hacker
ven.28avril2023162142CEST
```

10 Conclusion

Injecting code into an ELF file involves several steps, including finding the relevant program header and section header, overwriting the relevant section, and updating the ELF header to reflect the changes made. Through this project, I was able to understand the details of the ELF file format and how to manipulate it to inject code. I was also able to implement two different types of code injection: modifying the entry point and hijacking the GOT entry. The choice of which method to use depends on the specific goal of the code injection. Overall, this project allowed me to gain a deeper understanding of the inner workings of executable files and how to modify them for my purposes.