Software Security Project - Report

RIQUIER Erwan

Contents

1 Introduction

Elf file is something really important and can be found almost everywhere so we will try to do this project to learn more about them and manipulate it using the C language. For that the goal is to inject a small piece of assembly code modify the current section header and to finally be able to hijack the got entry allowing us to execute our code on a binary elf file.

2 Initialize ELF FILE for reading

I used the C function getopt to create an arguments parser so that the program must take 5argument

- -r nameOfElfFile
- -b nameOfBinaryToBeInjected
- -c newSectionName
- -a AddressOfInjectedCode
- -e

The option -e will decide wheter we want to overwrite the entry point or not.
Then we will use for the rest of the project the libelf to be able to get the information of an elf file. With the function getElf i will be able to get the elf file of the argument passed in command line. An elf file contains an executable header that can be get from the libelf function elf64_getehdr and if the result of e_ident[4] is 2 then that means that it is a 64bits executable.

3 Find the PT_NOTE segment header

The libelf function elf_getphdrnum will give us the number of program header than can also be found in the executable header.
Then the program header can be get through the function elf64_getphdr and because the program header is an array of program headers we can iterate over this array and inspect the p_type to check if the name is PT_NOTE wich is for auxiliray information. then we will store the index when we found the first PT_NOTE.

4 Code injection

Now after habing the PT_NOTE segment we will append our injected code to the elf binary for that as I previously said, we will create an assembly code and we will compile it using nasm which is an assembler for x86 CPU architecture. This task is really simple, we just have to open a file descriptor for thoses 2file and use the function fseek to position ourself inside the specific file so we will fseek at the end of the elf file and do an fwrite to write the injected binary at the end of the elf file.

.

Now we have injected our binary but the Elf specification requires that the offset and address are congruent modulo 4096. To ensure correct alignment we will modify the address so that the difference with the file offset becomes zero modulo 4096. To do that we have to create a delta integer.

$$\sigma = 4096 - ((elfSize - argumentAddress) \mod 4096) \tag{1}$$

and replace our base address by adding the difference.
argumentAddress += 4096 - σ

Then our address is align with 4096 so that

$$((argumentAddress - elfSize) \mod 4096) \iff 0 \tag{2}$$

5 Overwriting the concerned section header

Now we have to get the number of section header in the .shstrtab section that can be get using the libelf function elf_getshdrstrndx then we will loop over all sections headers to find the .note.ABI-tag to do that the liblef function elf_nextscn can be use to get the section header descriptor to loop until we find NULL then we will use elf64_getshdr to get the section header depending of the scn with the section header we can get the name using elf_strptr and if we find the section .note.ABI-tag we will modify it.

- sh_type = SHT_PROGBITS Because our code is a program data
- sh_flags = SHF_EXECINSTR | SHF_ALLOC Because our code must be run
- sh_addr = BaseAddress
- sh_offset = ElfSize Where the code has been injected
- sh_size = BinaryInjectedSize
- sh_type = 16

and we can just write back using the same fwrite and fseek method as the codeInjection. We will write back the section depending on the index of .note.ABI-tag inside the section header.

6 section headers calibration

We will now reorder the sections header by section address for that we can create an array that will contains every section header during the loop in the previous task then we will pass it through the function reoder and it will check wheter the new address of .note.ABI-tag have to be swapped left or right. with a simple loop over the left or right section of .note.ABI-tag and a simple comparison between their address. To swap the section header inside the array we will just create a temporary memory that will get a section header then to a simple swap.

When everything is done we will loop over the array and write back each section header to the elf binary.

Now we have to set the name that was passed through the command line argument
We can just get the .shstrtab which is the string table in ELF. The string table is used for all other references so each symbols from an ELF objectf has a member called st_name which is an index into this string table. then the field sh_name is an index inside the .shstrtab so we can use it to create an offset.

$$offsetShstrtabAbiTag = shstrtab- > shOffset + shdrAbitag- > shName \tag{3}$$

When we have the offset of the section abiTag in our string table we can just overwrite it with a simple fwrite.

Be aware that before changing the namen we have to check that the new name length is lower than .note.ABI-tag else we will overwrite another section name.

7 Overwriting the PT_NOTE program header

Before trying to execure our injected code, we now have to modify the program header PT_NOTE index

- phdr[index_PT_NOTE].p_type = PT_LOAD Allow the segment to be loadable
- phdr[index_PT_NOTE].p_offset = elfSize
- phdr[index_PT_NOTE].p_vaddr = baseAddress
- phdr[index_PT_NOTE].p_paddr = baseAddress
- phdr[index_PT_NOTE].p_filesz = injectedSize
- phdr[index_PT_NOTE].p_memsz = injectedSize
- phdr[index_PT_NOTE].p_flags = PF_X | PF_R for readable and executable
- phdr[index_PT_NOTE].p_align = 4096

And as simple as it is we just have to write it back to the elf binary.

8 Execute the injected code

Now that we have done all this. We can now try to execute our injected code to print a string on the output. 2 method.

8.1 Entry Point Modification

If we pass the option -e so we will modify the entry point for that we just modify the executable header e_entry by changing it to our baseAddress. then we also have to modify the assembly code to jump back to the old entry point. to jump back we can just do that

```

        mov rax, 0x4022e0          ;0x4022e0 is the old entry point
        jmp rax

```

at the end of our assembly code that means we will jump at the address content of the rax register.

8.2 Hijacking GOT Entries

But if we don't have the -e option we have to overwrite the got entry.

PLT is the Procedure Linkage Table which is used to call external functions whose address isn't known when linking and is done at run time by the dynamic linker.
GOT is the Global Offsets Table that will be used to resolve addresses.

Now we have to find a function that will be used to hijack the program. For that the command ltrace.
Ltrace intercepts and records the dynamic library callswhich are called by the executed process and the signals which are received by that process. It can also intercept and print the system calls executed by the program. cf ltrace manual page
so we can ltrace our date program and see a function getenv that will be used to overwrite for that we can just objdump the .plt section and search the getenv address we check the jump address and we have

```

        0000000000401700 <getenv@plt>:
        401700: ff 25 22 e9 20 00     jmpq   *0x20e922(%rip)        # 610028 <
    gmon_start@plt+0x20e4c8>
        401706: 68 02 00 00 00        pushq  $0x2
        40170b: e9 c0 ff ff ff        jmpq   4016d0 <__ctype_toupper_loc@plt-0x10>

```

We can try to modify the just that is done in 610028 in the got.plt section by our injectedCode address so that when the program run getenv function the jump performed point now to our program instead of the libc function getenv.

So we just have to saerch over the section to get the .got.plt section then we overwrite it at offset + 0x28 and run the program then we will have our string with the date print on the output.

We have been able to hijack the GOT entries successfully.

9 complementary information and conclusion

To run this project I choose to use 2assembly code depending on whether we want to modify the entry point or to the GOT entries so the program launch test is different.
Moreover because the binary is always modified when we run each program we are force to keep a backup of our date binary.

to launch the program please refer to the Makefile on the git associated to this report.

To conclusion we have learned a lot of things on the elf binary and have been able to hijack it to inject our code and execute it we learned the way to use the C language to manipulate every elf content. But there's still more to learn with dynamic binary !