Software Security Report

Abdoulkader MOUSSA MOHAMED

April 2023

1 Introduction

The goal of this project is to explore how to inject a completely new code section into an ELF binary.

2 Get started

2.1 Run the program

Firstly, lets clone the repository :

```
$ git clone git@gitlab.istic.univ-rennes1.fr:amoussamoham/softwaresecurityproject.git

$ cd softwaresecurityproject/

$ git checkout challenge-7
```

Then lets read the `documentation` of our program :

```
$ make
..

$ ./isos_inject --help
Usage: isos_inject [OPTION...]
Example: cp date date_copy; ./isos_inject --path-to-elf=./date_copy
--path-to-code=./inject_code_for_got_plt_hijack --new-section-name='.inject'
--base-address='0x500000'

  -a, --base-address=ADDRESS Base address of the injected code(in hexadecimal)
  -c, --path-to-code=PATH    Path to binary file that contains the machine code
                             to inject
  -e, --path-to-elf=PATH     Path to ELF file to analyze
  -m, --modify-entry-function   Modify the address of the entry
                             function(default: False)
  -n, --new-section-name=NAME   Name of the newly created section
```

```
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Finally, our program can be launched as follows :

```
# For entry point modification
$ cp date date_copy; ./isos_inject --path-to-elf=./date_copy
--path-to-code=./inject_code_for_entry_point_modification --new-section-name='.inject'
--base-address='0x500000' --modify-entry-function

# For hijacking GOT Entries
$ cp date date_copy; ./isos_inject --path-to-elf=./date_copy
--path-to-code=./inject_code_for_got_plt_hijack --new-section-name='.inject'
--base-address='0x500000'
```

The names of options in our program are chosen to make it easily understandable.

2.2 Check warnings

We can compile all the program with the given commands (excluding one) in the subject to make sure that any warning is generated :

```
$ make check_warnings
```

The excluded command is `clang-tidy` which generates many warnings. Those warnings are not that significant :

```
$ clang-tidy isos_inject.c -checks=cert-*,clang-analyzer-*
```

2.3 Run tests

We have a test script that tests the executables generated with the commands provided in the subjects.

```
$ make test
```

For example, a test case verifies that if the binary file is not an ELF executable of the 64-bit architecture, the program should exit with a failure as expected.

3 How do we find the pt_note segment header ?

This function return the index of the first program header of type PT_NOTE :

```c
/**
 * Return the index of the first program header of type PT_NOTE
 */
int get_index_of_first_program_header(Elf64_Ehdr executable_header,
                                      char **addr) {

  int index_pt_note = -1;
  /* Initializing of program header number i */
  Elf64_Phdr *program_header =
      (Elf64_Phdr *)malloc(executable_header.e_phnum * sizeof(
    Elf64_Phdr));

  if (program_header == NULL) {
    errx(EXIT_FAILURE, "Error while calling malloc \n");
  }

  memcpy(program_header, *addr + executable_header.e_phoff,
         executable_header.e_phnum * sizeof(Elf64_Phdr));

  for (int i = 0; i < executable_header.e_phnum; i++) {

    if (program_header[i].p_type == PT_NOTE) {
      index_pt_note = i;
      break;
    }
  }
  free(program_header);

  return index_pt_note;
}
```

First, the program do a memory allocation to store the program headers. Then, it positions itself at the offset `e_phoff` and uses `memcpy` to copy the program headers into the allocated memory.

Next, the program iterates over all the program headers to find the one with the field `p_type` equal to `pt_note`.

Finally, we can find in our program that the index of first program header of type `pt_note` is 5.

4 Append inject code to ELF

This function append the injected code to the end of ELF and compute also the new base address (given by the user).

```c
int append_inject_code_to_elf_and_compute_new_base_address(
    int fd, struct arguments *arguments) {

..
  int end_position_elf = lseek(fd, 0, SEEK_END); /* At the end of
    the binary */

```

```c
  /* Buffer containing the injection code bytes */
  char *buffer = malloc(fstat_inject.st_size * sizeof(char));

  /* Code injection at the end of the binary */
  read(inject_file_fd, buffer, fstat_inject.st_size);
  write(fd, buffer, fstat_inject.st_size);
  free(buffer);
  close(inject_file_fd);

  /* Computing base address so that the difference with the offset become
    zero modulo 4096.*/
  arguments->injected_code_base_address +=
      (end_position_elf - arguments->injected_code_base_address) % ALIGNMENT;

  printf("Computed base address: %d\n", arguments->injected_code_base_address);

  return end_position_elf;
}
```

Firstly, the program positions itself at the end of the ELF file, and uses `read` system call to read the file to inject and put it inside a buffer. Then it uses `write` syscall to append the content of the buffer to the ELF file.

Next, it computes again the address given by the user so that the difference with the ELF offset become zero modulo 4096. This is done to ensure that the injected code is properly aligned in memory,

5 Values inserted inside the section header

Once we found the index of the header of the .note.ABI-tag section, we needed to modify the following fields :

```
.sh_type = SHT_PROGBITS;
.sh_addralign = ADDR_ALIGN;
.sh_flags |= SHF_EXECINSTR;

.sh_addr =
  arguments.injected_code_base_address; # Base address address recomputed
.sh_offset = fstat_structure.st_size; # Size of the ELF which is where the injected code is added
.sh_size = inject_code_size; # Size of the injected code
```

We also overwrited the fields of the `pt_note` program header. They are almost the same as the ones of the .note.ABI-tag section :

```
pt_note_ph->p_type = PT_LOAD;
pt_note_ph->p_flags |= PF_X;
```

```
pt_note_ph->p_align = 0x1000;

# For the sake of readability, x represents section_headers[index_of_note_abi_tag].
pt_note_ph->p_offset = x.sh_offset;
pt_note_ph->p_paddr = x.sh_addr;
pt_note_ph->p_vaddr = x.sh_addr;
pt_note_ph->p_filesz = x.sh_size;
pt_note_ph->p_memsz = x.sh_size;
```

Once we modified the fields, we use standart C file operation : `lseek, write` to save the update to the ELF.

6 Sort section headers

Before sorting the section headers, we know that all the sections are sorted regarding their address except the one that we modified. So to sort the section headers, we follow the steps mentionned below :

— We look at the left and right neighboor of the section that we modified to `check` that he is at the good position and in this case, it's already sorted.

```
if ((section_headers[*index_of_note_abi_tag - 1].sh_addr <
       section_headers[*index_of_note_abi_tag].sh_addr) &&
      (section_headers[*index_of_note_abi_tag].sh_addr <
       section_headers[*index_of_note_abi_tag + 1].sh_addr)) {
    /* Already sorted, no need to move */
    return false;
  }
```

— We look again to the left and right neighboor in order to know whether it should be moved right or left.

— Then, we shift the concerned sections to left or right respectively if the section that we modified must be moved to right or left.

— We place the section at it right place

The sections headers are sorted again but with `readelf`, we can notice some warnings. So we update the field `sh_link` of each section in order to avoid the warning.

We can check that sections are well sorted with a base address of `0x500000` :

```
$ readelf --sections --wide date_copy
Il y a 29 en-têtes de section, débutant à l'adresse de décalage 0x10430 :
```

```
En-têtes de section :
  [Nr] Nom               Type            Adr               Décala.Taille ES Fan LN Inf Al
  ..
  [16] .eh_frame_hdr     PROGBITS        000000000040e514 00e514 00034c 00   A  0   0  4
  [17] .eh_frame         PROGBITS        000000000040e860 00e860 00143c 00   A  0   0  8
  [18] .inject           PROGBITS        0000000000500b70 010b70 010b70 00  AX  0   0 16
  [19] .init_array       INIT_ARRAY      000000000060fe10 00fe10 000008 00  WA  0   0  8
  ..
```

7 Entry Point Modification

To change the entry point, here is our code :

```
executable_header.e_entry = arguments.injected_code_base_address;
lseek(fd, 0, SEEK_SET);
write(fd, &executable_header, sizeof(Elf64_Ehdr));

printf("-- Entry point changed successfully -- \n");
```

Firstly, we change the `e_entry` field of the executable header, then the porgram positions itself to the begining of the ELF and we finally use `write` to save the update to the ELF.

```
# Before modification
$ readelf -h date
..
Adresse du point d'entrée:         0x4022e0

# After modification
$ readelf -h date_copy
..
Adresse du point d'entrée:         0x500b70
```

We had to update our assembly code in order to invoke the 'write' syscall to print "Je suis trop un hacker" and then calling the original entry.

```
  ..

  ; write
  mov rax, 1              ; System call number (write)
  mov rdi, 1              ; stdout
  lea rsi, [rel message]  ; message to display
  mov rdx, [rel len]      ; message length
  syscall                 ; kernel call
```

```
  ; load context
  ..

  ; call original entry point
  push 0x4022e0
  ret


message: db "Je suis trop un hacker", 0xA, 0x0
len: dd $-message
```

Our program runs successfully, and it produces the expected output :

```
$ ./date_copy
Je suis trop un hacker
sam. 22 avril 2023 01:48:52 CEST
```

8 Hijacking GOT Entries

With `ltrace`, among the called functions, we choosed localtime :

```
$ ltrace ./date
..
localtime(0x7fff7d2e3e20)           = 0x7f0e30e3d640
..
```

Then we need to know where is located the GOT entry it uses :

```
$ objdump -d date |less
..
0000000000401720 <localtime@plt>:
  401720:       ff 25 12 e9 20 00       jmp    *0x20e912(%rip)        # 610038 <__gmon_start
  401726:       68 04 00 00 00          push   $0x4
  40172b:       e9 a0 ff ff ff          jmp    4016d0 <__ctype_toupper_loc@plt-0x10>
..
```

So 610038 is the address where the GOT entry is located. We put it in a macro inside our program

```
#define HIJACK_ADDRESS_GOT_ENTRY 0x610038 /* Localtime is overwrited */
```

Using objdump, we can display the content of '.got.plt' section :

```
$ objdump -s --section=.got.plt date
Contenu de la section .got.plt :
..
 610030 16174000 00000000 26174000 00000000  ..@.....&.@.....
..
```

Our goal is to replace `26174000` with `efbeadde`(the address of the function we want to overwrite).

To do it, we iterate over all the section headers to find the index of `.got.plt`. we do some math to determine the appropriate position, and then we use lseek and write to hijack.

```
int got_start_real_address = section_headers[index_got_plt].sh_offset;
int got_start_virtual_address = section_headers[index_got_plt].sh_addr;

lseek(fd, got_start_real_address +
          (HIJACK_ADDRESS_GOT_ENTRY - got_start_virtual_address), SEEK_SET);
write(fd, &arguments.injected_code_base_address,
      sizeof(arguments.injected_code_base_address));

printf("-- .got.plt overwrited successfully -- \n");
}
```

Finally, we update our assembly code in order to invoke the 'write' syscall to print "Je suis trop un hacker" and then invoke 'exit' syscall because we no longer need to transfer control back to the original implementation.

```
..
; write
mov rax, 1              ; System call number (write)
mov rdi, 1              ; stdout
lea rsi, [rel message]  ; message to display
mov rdx, [rel len]      ; message length
syscall                 ; kernel call

; exit
mov rax, 60 ; System call number (exit)
mov rdi, 0  ; success status code
syscall     ; kernel call
..
```

Our program runs successfully, and it produces the expected output :

```
$ ./date_copy
Je suis trop un hacker
```

We can check again with `objdump` to ensure that our hijack address 0x500b70(700b5000 in little indian) is correctly located :

```
$ objdump -s --section=.got.plt date_copy
..
Contenu de la section .got.plt :
```

```
..
 610030 16174000 00000000 700b5000 00000000  ..@.....p.P.....
..
```