ISOS

Ali BA FAQAS

May 2024

1 Challenge 7

In this report, I will showcase the final step of this project and present the end result.

1.1 Assembly instruction for injected code.

The initial step involved modifying the assembly code injected in Challenge 3. The objective was to invoke the write system call to print the message "Je suis trop un hacker". To make the code suitable for injection, it was assembled into a raw binary file using the -f bin option of the nasm assembler. This process resulted in a file that contains only the binary encodings of the assembly instructions and data.

```
BITS 64

SECTION .data
    msg db 'Je suis trop un hacker', 0
SECTION .text
global main

main:
    ;save context
    push rax
    push rcx
    push rdx
    push rsi
    push rdi
    push r11

    ;write syscall
    mov rax, 1 ; syscall number (sys_write)
    mov rdi, 1 ; file descriptor (stdout)
    lea rsi, [rel msg] ; pointer to message to write
    mov rdx, 22 ; message length
    syscall

    ;load context
    pop r11
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rax

    ret
```

The assembly code was then transformed into a raw binary file using the nasm assembler's -f bin option.

1.2 Entry Point Modification

To execute the newly modified injected code, it was necessary to invoke it through the ELF entry point. This would ensure that the new code runs as soon as the loader transfers control to the binary. To accomplish this, the e_entry field in the ELF executable header was modified to point to the injected code. The modified ELF header was then written back into the ELF file.

```
 // Modify the entry point (optional)
  if (arguments.modify_entry) {

    // Modify the entry point of the ELF file to point to the injected code
    header->e_entry = arguments.base_address;
    // Write the modified ELF header back into the ELF file
    lseek(fd6, 0, SEEK_SET);
    write(fd6, header, sizeof(Elf64_Ehdr));
  }
```

Subsequently, to ensure the normal execution flow wasn't disturbed, the assembly code was modified to end by calling the original entry point function. The address of the original entry point was found in the ELF header. To prevent any segmentation fault, the address was moved to the rax register before jumping.

The updated assembly code is as follows:

```
BITS 64

SECTION .data
    msg db 'Je suis trop un hacker', 0
SECTION .text
global main

main:
    ;save context
    push rax
    push rcx
    push rdx
    push rsi
    push rdi
    push r11

    ;write syscall
    mov rax, 1 ; syscall number (sys_write)
    mov rdi, 1 ; file descriptor (stdout)
    lea rsi, [rel msg] ; pointer to message to write
    mov rdx, 22 ; message length
    syscall

    ;load context
    pop r11
    pop rdi
    pop rsi
    pop rdx
```

```
    pop rcx
    pop rax

    mov rax, 0x4022e0; original entry point
    jump to original entry point
    jmp rax
```

1.3 Hijacking GOT Entries

The modification of the entry point allows the injected code to run only once at the startup of the binary. To invoke the injected function repeatedly, a GOT entry was hijacked to replace a library call with the injected function. This was achieved by finding the .got.plt section and calculating the address of the GOT entry for fseek. The GOT entry was then overwritten with the address of the injected code.

The address of fseek was obtained using the readelf -r command.
to get the address of fseek we use readelf -r command.

```
$ readelf -r date

Relocation section '.rela.dyn' at offset 0xf30 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000060fff8  002700000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000610300  004a00000005 R_X86_64_COPY     0000000000610300 __progname@GLIBC_2.2.5 + 0

....
....

0000006101f8  003d00000007 R_X86_64_JUMP_SLO 0000000000000000 fseeko@GLIBC_2.2.5 + 0
```

After obtaining the address of fseek, the GOT part of the code was ready to be written.

```
// Hijack the GOT entry
  // Find the .got.plt section
  shdr = (Elf64_Shdr *)((uintptr_t)header + header->e_shoff);
  strtab = (char *)header + shdr[header->e_shstrndx].sh_offset;
  Elf64_Shdr *gotplt_shdr = NULL;
  for (int i = 0; i < header->e_shnum; i++) {
    if (strcmp(strtab + shdr[i].sh_name, ".got.plt") == 0) {
      gotplt_shdr = &shdr[i];
      break;
    }
  }

  if (!gotplt_shdr) {
```

```
    fprintf(stderr, "Failed to find .got.plt section\n");
    exit(-1);
  }

  // Calculate the address of the GOT entry for fseeko
  Elf64_Addr fseeko_got_entry = 0x6101f8 - gotplt_shdr->sh_addr;

  // Overwrite the GOT entry with the address of the injected code
  *(Elf64_Addr *)((uintptr_t)map_start6 + fseeko_got_entry) =
      arguments.base_address;
```

The jump to the original entry point was removed from the assembly code, as there was no longer a need to transfer control back to the original implementation when the injected code completes.

The final version of the assembly code is as follows:

```
BITS 64

SECTION .data
    msg db 'Je suis trop un hacker', 0
SECTION .text
global main

main:
    ;save context
    push rax
    push rcx
    push rdx
    push rsi
    push rdi
    push r11

    ;write syscall
    mov rax, 1 ; syscall number (sys_write)
    mov rdi, 1 ; file descriptor (stdout)
    lea rsi, [rel msg] ; pointer to message to write
    mov rdx, 22 ; message length
    syscall

    ;load context
    pop r11
    pop rdi
    pop rsi
    pop rdx
    pop rcx
```

```
    pop rax

    ret
```

After this step, the program was ready to be executed. The following results were obtained:

```
$ make src
make -C src all
make[1]: Entering directory '/home/aloosh/Desktop/S2/Software Security/isos-project/src'
gcc -o isos_inject isos_inject.c -lbfd
mv isos_inject ../
rm -f *.o
make[1]: Leaving directory '/home/aloosh/Desktop/S2/Software Security/isos-project/src'
cp dateOriginal ./date
./isos_inject --elf_file=date --machine_code=inject --section_name=new --base_address=0x5000
ELF_FILE = date
MACHINE_CODE = inject
section_name = new
BASE_ADDRESS = 5242880
MODIFY_ENTRY = 14
Found PT_NOTE at index 5
offset before 68464
mod  0
chmod +x date
./date
Je suis trop un hackerSun May 12 07:00:50 PM CEST 2024
```

As you can see, the output includes "Je suis trop un hacker" followed by the date, and the program ends without any segmentation fault.