Software Security

Overall Goal

The binary modification techniques you learned so far are pretty limited in their applicability. Hex editing is useful for small modifications, but you can't add much (if any) new code or data. LD_PRELOAD allows you to easily add new code, but you can use it only to modify library calls. In this project, we explore how to inject a completely new code section into an ELF binary.

For pedagogical reasons, the project is divided into a number of challenges in order to help you in your progress. You have carte blanche to organize your code in different files/scripts.

High-Level Overview

To add a new section to an ELF binary, you first require to inject the bytes that the section will contain by appending them to the end of the binary. Next, you create a section header, and a program header for the injected section.

As you may recall from our lectures, the program header table is usually located right after the executable header. Because of this, adding an extra program header would shift all of the sections and headers that come after it. To avoid the need for complex shifting, you can simply overwrite an existing one instead of adding a new program header.

You might wonder which headers you can safely overwrite without breaking the binary? One program header that you can always safely overwrite is the PT_NOTE header, which describes the PT_NOTE segment. The PT_NOTE segment encompasses sections that contain auxiliary information about the binary. For example, it may tell you that it's a GNU/Linux binary, what kernel version the binary expects, and so on. This trick is commonly used by malicious parasites to infect binaries, but it also works for benign modifications.

Environment setup

In the project, we will not reinvent the wheels and manually parse ELF files. Instead, you will rely on the libbfd library to read ELF files. In a nutshell, the Binary File Descriptor library (libbfd) provides a common interface to read and write binary files in many different object file formats. The libbfd library is part of the GNU project and often used by binutils tools, including readelf and objdump.

On your system, do the following:

- Create a gitlab project and add your supervisor to it.
- Install the latest release of libbfd library as well as the associated manual.
- Install the assembler nasm.
- Check that your Linux utilities are not PIE (i.e. their ELF type should be EXEC). If it is not the case, download the '*date*' binary from the Software Security website.
- Make a copy of the *date* binary into your project directly, and make sure that it still works.

Challenge 1: Initialize ELF file for reading

The major demanding task in this project is to develop a tool called **isos_inject** that takes 5 arguments:

1. The elf file that will be analyzed.
2. A binary file that contains the machine code to be injected.
3. The name of the newly created section.
4. The base address of the injected code.
5. A Boolean that indicates whether the entry function should be modified or not.

The 1st argument will always be the path to the copied 'date' binary. Regarding the 2nd one, we will not be concerned about the real content of the injected machine code until Challenge 7. Thus, just create a file with some arbitrary content. The 3rd argument is just an arbitrary string; be creative! The 4th is chosen, so that there is no overlap with other sections/segments. The 5th argument will control whether the address of the entry function should be modified.

Do the following:

- Create **isos_inject.c** and make it parse its 5 arguments. Use argp instead of getopt. Make sure to create --help option.
- Initialize the BFD library, and open the elf using libbfd.
- Make sure that the opened binary is of format elf, of architecture 64-bit and executable.

Challenge 2: Find the PT_NOTE segment header

After getting the executable header, **isos_inject** loops over all the program headers in the binary to check whether the binary has a PT_NOTE segment that's safe to overwrite.

To find the PT_NOTE segment, do the following:

- Open the binary again (not with libbfd) in reading and parse the executable header into a structure that you create.
- Look up the number of program headers that the binary contains.
- Inspect the p_type field of all program headers, and check whether the type is PT_NOTE.
- Store the index of the first program header of type PT_NOTE.

HINTS:
Within the project, you will implement various helper functions to parse ELF headers.
Clearly identify your need from the beginning would definitely help you.
Use **mmap** to simplify your implementations.

Challenge 3: Code injection

After locating the overwritable PT_NOTE segment, it's time to append the injected code to the binary.

- Create an assembly file that contains the following content, and then compile it with nasm. This will be the code to be injected into the elf file. You will complete it in Challenge 7.

```
BITS 64

SECTION .text
global main

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

    ; fill here later

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

    ; return
    ret
```

- Use standard C file operations to append the injection code to the elf file (i.e. the end of the binary). Save the byte offset where the code bytes were written.
- The ELF specification requires that the offset and address are congruent modulo 4096. To ensure correct alignment, compute the address, so that the difference with the file offset becomes zero modulo 4096.

Now that the code injection is done, all that remains is to update a section and program header (and optionally the binary entry point) to describe the new injected code section and ensure it gets loaded when the binary executes.

Challenge 4: Overwriting the concerned section header

In this project, we will (arbitrarily) choose to overwrite the header of the .note.ABI-tag section that is part of the PT_NOTE segment. This requires to modify the following fields in the section header:

- sh_type to SHT_PROGBITS (denoting a code section).
- sh_addr, sh_offset, sh_size to their new values.
- sh_addralign to 16 bytes (code properly aligned into memory).
- sh_flags by adding SHF_EXECINSTR flag to mark the section as executable.

Do the following:

- Get the index number of the section header describing the .shstrtab section.
- Loop overall section headers, inspecting each one as it goes along.

- Inside the loop, get the name of each iterated section header. Here, you will need to use the index of the .shstrtab section, and the index shdr.sh_name of the current section's name in the string table.
- If the name of the current section is .note.ABI-tag, note its index and overwrite the fields in the section header to turn it into a header describing the injected section.
- Once the header modifications are complete, write the modified section header back into the ELF binary file. More details are found below.

Writing the modified section header back to the binary.

- Compute the file offset at which to write the updated section header.
- Seek the file descriptor to that offset in the ELF file and write the updated header.

HINTS:
The executable header contains valuable information.
Do not forget to manage your previously file descriptors.

BONUS:
Can you do this challenge using libbfd API: bfd_get_section_by_name_if, bfd_set_section_size, bfd_set_section_flags, etc?

Challenge 5: section headers calibration

By this point, the new code bytes have been injected at the end of the ELF binary and there's a new code section that contains those bytes, but this section doesn't yet have a meaningful name in the string table, or a meaningful order. Let's update all that.

Reorder section headers by section address

Sections headers are sorted regarding their address. Since the overwritten section header has a new address, it should be placed elsewhere.

- Compare with the direct neighbors of the overwritten section in order to decide whether it should be moved right or left.
- Determine how far should it be moved in each direction.
- Swap the injected section, so that the sections headers are sorted again.

HINTS:
You start from a sorted list.

Set the name of the injected section:

- Check that the given name in argument has smaller length than the length of the string ".note.ABI-tag". Do you know why?
- Get (again) the string table section index.
- Get the offset of ".note.ABI-tag" into shstrtab.
- Get the offset to start of .shstrtab.
- Compute the file offset at which to write the new section name.

- Write the new section name to the ELF binary at the just-computed offset. Use standard C operations for file writing.

Challenge 6: overwriting the PT_NOTE program header

To recap, the ELF binary now contains the injected code, an overwritten section header, and a proper name for the injected section. The next step is to overwrite a PT_NOTE program header, creating a loadable segment that contains the injected section.

As you may remember, you have already located and saved the PT_NOTE program header to overwrite. All that's left to do is to overwrite the relevant program header fields and save the updated program header to file. This requires to modify the following fields in the section header:

- p_type to PT_LOAD (denoting a loadable segment).
- p_offset_p_vaddr, p_paddr, p_filesz, and p_memsz.
- p_flags by marking the segment as executable (instead of just readable).
- p_align to 0x1000 (a page of 4096 bytes).

Then, given a pointer to the program header table, all that remains now is to seek to the correct file offset, and write the updated program header there. That completes all the mandatory steps for injecting a new code section into an ELF binary! The remaining step is to execute the injected code.

Challenge 7: execute the injected code

Assembly instruction for injected code.
Complete the assembly given in Challenge 3 in order to invoke the 'write' syscall to print "Je suis trop un hacker". To make the code suitable for injection, you need to assemble it into a raw binary file that contains nothing more than the binary encodings of the assembly instructions and data. This because you don't want to create a full-fledged ELF binary that contains headers and other overhead not needed for the inject. To assemble into a raw binary, you can use the nasm assembler's '-f bin' option.

Entry Point Modification
To get the new code to execute, you require to call it through the ELF entry point, causing the new code to run as soon as the loader transfers control to the binary. However, you need not disturb the execution. Therefore, you must modify the assembly code in order to end by calling the original entry point function. If the option is chosen, update the e_entry field in the ELF executable header. Then, write the modified executable header back to the ELF file.

Hijacking GOT Entries
The entry point modification technique allows the injected code to run only once at startup of the binary. What if you want to invoke the injected function repeatedly, for instance, to replace an existing library function? Let's end this project and hijack a GOT entry to replace a library call with an injected function. Overwriting GOT entries essentially gives you the same level of control as the LD_PRELOAD technique but without the need for an external library containing the new function, allowing you to keep the binary self-contained. Moreover, it is a suitable technique for persistent binary modification.

First, modify the injected assembly because there is no longer need to transfer control back to the original implementation when the injected code completes. Thus, your code will not contain any hard-coded address to which it transfers control at the end. Instead, it simply ends with a normal return.

Second, call 'ltrace' and note the called functions. Pick one and modify its GOT (i.e. .got.plt) entry to call your injected code. If everything went right you would get a sweet "Je suis trop un hacker" each time you run 'date' to recall you of your great achievement.
Congratulations! Your journey of static binary modifications ends here. The teaching team hopes that you enjoy it. Looking forward to meeting you again in another journey with dynamic binary modifications leveraging 'ptrace'. Until then, keep hacking!

Elements of evaluation

- You have to do challenges in order: from number 1 to number 7.
- Challenge N gives you (N - 1) points. For instance, challenge 4 gives 3 points. Solving all the challenges gives you 21 points.
- A video (or written) report should be handed at the end of the project. The report does not give you any positive point. A good report gives you 0 (zero) point, and a very bad report will give you -4 (negative) points. However, the absence of report will give -5 (negative) points.
- High-quality code gives you 0 (zero) point, and ill-written code will give you -5 (negative) points. Please consider commenting your code, and organizing your program into small chunks with specific goals. No warnings should be issued when you compile with:
    - clang -fsyntax-only -Wall -Wextra -Wuninitialized -Wpointer-arith -Wcast-qual -Wcast-align
    - gcc -O2 -Warray-bounds -Wsequence-point -Walloc-zero -Wnull-dereference -Wpointer-arith -Wcast-qual -Wcast-align=strict
    - gcc -fanalyzer
    - Run the tool clang-tidy.
    - Run your code with clang sanitization options (address, memory, ub), and make sure that your program does not crash.
- Late challenges lose points. You get -1 (negative) point for each week after the deadline.
    - Week 1 starts on March 6th.
    - You should complete Task N before the beginning of Week N + 2.
    - Deadline to validate technical challenges is the end of week 7.

- Any plagiarism will be sanctioned by the disciplinary commission of Rennes 1.