

The objectives are to:

- (a) Optimize/prepare shellcode to exploit buffer overflow (BOF) vulnerabilities
- (b) Analyze potential buffer overflow vulnerabilities in source code
- (c) Exploit buffer overflow vulnerabilities using different techniques



Never use any of the provided code on a network connected to the Internet.

1. Prerequisites

- (a) Disable address space randomization

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

- (b) Running the vulnerable C program

```
$ gcc -m32 -z execstack -fno-stack-protector -o prog prog.c
```

- (c) Creating a new payload (a startup Python script is provided)

```
$ python3 <code>.py
```

- (d) Install ropper

Visit the ropper repository for installation steps: <https://github.com/sashs/Ropper>

Task 1: Optimizing the Shellcode [20%]

The bytecode produced from the `labsh.asm` program (used in Lab 02) will not be useful in exploiting BOF vulnerabilities. Your **task** is to prepare a valid bytecode that can be used to exploit a BOF vulnerability and to minimize the size of the resulting bytecode as much as you can!

You can do this by writing an optimized Assembly code (call it `labsh_opt.asm`), and use the resulting bytecode in the following tasks. You can inspect the bytecode by using `objdump` or `gdb`.

Questions

- Inspect the provided `vuln_1.c` program. Why cannot the bytecode of `labsh.asm` be used to exploit the BOF vulnerability in `vuln_1.c`?
- Explain every optimization technique you performed in `labsh_opt.asm`.
- What is the size of your bytecode (in bytes)?

Task 2: Exploiting a BOF vulnerability -- Jump to Shellcode [40%]

The provided `vuln_1.c` program has a BOF vulnerability. The program reads a text file called `shellcode_1` and copies its content to a buffer that we plan to overflow (buffer in function `bof`).

```
int bof(char *str)
{
    char buffer[BUF_SIZE];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
```

Your **task** is to generate the contents of the `shellcode_1` file to exploit the buffer overflow vulnerability. In particular, your shellcode from Task 1 should be used in the payload. If the attack succeeds, a new shell will be spawned during the normal flow of the program.

In this exploit, you need to jump to the shellcode and use NOP sleds.

- You should set the `BUF_SIZE` to be $27+x$, where x is the least significant two digits in your SFU ID. If these two digits are zeros, choose the next significant two digits. Different numbers will result in different stack layouts. E.g., if your SFU ID is 400508678, the buffer size should be $(27+78)$. If your SFU ID is 400508600, the buffer size should be $(27+86)$.
- We provided a simple Python script (`generate_payload_1.py`) that generates the shellcode.
- In `generate_payload_1.py`, complete the missing parts in the new file:
 - Fill in the shellcode from Task 1
 - How the `shellcode_1` file is initialized
 - The offset value `offset`, and
 - The return address at `content[offset+0]...content[offset+3]`.

Note that, for the purpose of this task, the shellcode needs to be copied to *the end* of the generated payload.

Questions

- What is the effect of `BUF_SIZE` on your solution?
- Explain (with screenshots) the steps you made to calculate the offset value `offset` and the return address.
- If the address space randomization is enabled, suggest a strategy to exploit the buffer overflow vulnerability for this program.

Task 3: Exploiting a BOF vulnerability -- Jump to Register [40%]

Your **task** is to exploit the buffer overflow vulnerability in `vuln_2.c` by jumping to a general-purpose register (i.e., not `esp`).

This task has two subtasks.

Subtask 1. Inspect the `vuln_2.c` program, and answer the following questions.

- Explain whether you can perform the jump-to-register technique for the `vuln_2.c` program. Support your answer with proper screenshots from `gdb`.
- If you cannot, modify the `bof` function only in `vuln_2.c` to enable exploiting BOF using jump-to-register. In `vuln_2.c`, the `BUF_SIZE` should be identical to the one in Task 2.
- What register can be used to perform jump-to-register? Why? Support your answer with proper screenshots from `gdb`.

Subtask 2. In this subtask, you need to generate the contents of the `shellcode_2` file (using `generate_payload_2.py`) to exploit the buffer overflow vulnerability. To calculate the return address, you need to find the *absolute address* of the jump-to-register pattern (or gadget) in one of the loaded libraries in your program. So, you need to perform the following:

- Find the *base address* of a loaded library using `gdb`. When running the program in `gdb`, you can inspect the outputs of `vmmap` to check the base address of loaded libraries. You may choose any loaded library.
- Use `ropper` to find the gadget *offset* inside a library. A gadget is the pattern that you want to look for inside a given library, e.g., `jmp <reg>`.

The following figure shows a typical usage of `ropper` to find the gadget offset.

```
(ropper)> file <Library Path>
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
[INFO] > search /1/ <pattern>
[INFO] Searching for gadgets: 
[INFO] File: 
Offset :
```

Once you find the *base address* of the library and gadget *offset*, you can calculate the *absolute address* of the gadget as *base address + offset*.

Questions

- (a) Explain (with screenshots) the steps you made to calculate the offset value `offset` in `generate_payload_2.py` and the return address.
- (b) Does the exploit work if you copy the shellcode to the end of the payload? Explain.

3. Submission

You need to submit:

- (1) All source code files that you developed.
- (2) A detailed lab report. The PDF file should be named `FirstName_LastName.pdf`

The files should be compressed in a single (.zip) archive with the name `FirstName_LastName.zip`. The code should compile and run without any errors.