Project 3.0 - Exploiting Buffer Overflow Vulnerabilities

Network Security - Spring 2025

Due Date: March 26 by 11:55 pm

CS473 Network Security Labs, LUMS © 2025 by Basit Shafiq and Abdul Rafay is licensed under Creative Commons Attribution-NonCommercial 4.0 International. To view a copy of this license, visit `https://creativecommons.org/licenses/by-nc/4.0/`

Contents

1 Introduction

This project is concerned with buffer overflow vulnerabilities and different ways to exploit them. A buffer overflow is a condition in which a program attempts to write data beyond the original bounds of a buffer, causing the data to overflow into adjacent memory. This can lead to a variety of security vulnerabilities, including the potential for an attacker to execute arbitrary code on the system. This situation can be particularly dangerous if the program is running with elevated privileges, such as root or administrator access.

Modern systems, however, come with a lot of mitigation techniques to prevent buffer overflow attacks. These include Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Stack Canaries. For this project, we will be disabling ASLR, DEP, and stack canaries to make the buffer overflow attacks easier to perform. For the next project, these mitigations will be enabled, and you will have to bypass them to exploit the vulnerabilities.

2 Getting Started

Before you get started, here's a few things you need to do to set up your environment.

1. Make sure that your VM instance (used for the previous project) is up and running. Previous projects' files are not necessarily required for this project. However, if you are using a new VM instance, make sure to do the setup first, as mentioned in Project 1.

2. Download the archive from LMS. Extract the contents of the archive, and copy it over to your VM, using `scp` or any other method that you have been using. Table 1 lists the files/contents that you will find in the archive.

3. The `setup.sh` script is provided to help you setup the environment, specific to this project. Update your roll number inside this script. Then, make it executable, and run it using the following command (on the VM, inside the correct directory):

```
$ chmod +x setup.sh && sudo ./setup.sh
```

4. The script performs the following setup steps, which you can also do manually if needed.

5. The script fetches the vulnerable binary (which is unique for each student), and places it in the `bin` directory. You will need to use this binary to perform the buffer overflow attacks.

6. Since you will be debugging a binary, you will need to install `gdb` on your VM. The script already does that for you, but you can install it directly via `apt` as well.

7. We need to make sure that ASLR is disabled, before starting the lab. The script should do that for you. It can be done manually via the following command:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

8. The script also sets up a virtual environment for the Python application and installs the required dependencies. You must activate the virtual environment using the following command, before launching the application.

```
$ source .venv/bin/activate
```

9. You can then start the application as usual, by first sourcing the `env.sh` file, and then launching the `fastapi` application.

10. Finally, you need to open port 17771 on your VM. You can do this by adding a new rule to the Security List, as done in project 1 to open ports 443 and 80. Note that this port is arbitrary, but we are choosing it for the sake of consistency.

File/Directory	Description
`app`	The FastAPI application that you will be attacking.
`practice`	Contains the exploit script, source code, makefile, and the binary for Task 1.
`ppm_images`	Some sample ppm images, useful for testing.
`shellcodebin`	Contains the shellcodes for Task 2 and Task 3 in a bin file. More details in 3.2.1 and 3.3.1.
`exploits`	Contains the exploit scripts for Task 2 and 3.
`env.sh`	ENV variables for the application. Doesn't require any change.
`setup.sh`	The setup script for the project. Add your roll number to this script.
`requirements.txt`	No change needed.

Table 1: Project files description

3 Project Tasks

After the security flaws were found and exploited in the previous project, the company was forced to fire the previous developer. A new firm was hired for a complete rewrite of the application, addressing the security issues and ensuring a more robust implementation. Additionally, new features were introduced such as the ability for users to view and update their profile, including their profile picture.

As a security consultant, your job is to test the new application for security vulnerabilities. You must assume the role of an adversary and exploit the new application to uncover flaws. Your objective is to identify vulnerabilities and report them so that the company can secure their application. The web vulnerabilities discovered in the previous project, can be assumed to be fixed and irrelevant for now.

For the profile section, the developer made an unusual choice by allowing users to upload images in the PPM format (specifically, P6 PPMs). More details about the PPM image format can be found at `https://netpbm.sourceforge.net/doc/ppm.html`. However, since browsers do not natively support PPM images, the application uses a C program called `converter`, located in the `app/bin` directory, to convert PPM images into BMP format first before storing them.

Since this binary processes raw user-supplied data, it may be vulnerable to buffer overflow attacks. Your task is to analyze the binary, understand how it works, and determine if it can be exploited to gain control over the application.

3.1 Task 1: Getting Familiar with Stack Smashing

Before working with the `converter` binary, you need to familiarize yourself with `gdb` and basic stack-smashing techniques. A vulnerable binary, `practice/vuln`, has been provided for this purpose. Try to exploit this binary using stack smashing.

1. Before starting, have a look at the `https://beej.us/guide/bggdb/` guide to get familiar with `gdb`, if you are not already. The GDB cheat sheet at `https://csapp.cs.cmu.edu/3e/docs/gdbnotes-x86-64.pdf` is a great resource to quickly look up GDB commands.

2. The `practice/vuln` binary is vulnerable to a buffer overflow attack since the application fills a buffer without checking the input length. The source code for the binary and the makefile is also included, so you can compile it yourself if needed (simply run `make` in the `practice` directory).

3. To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored on the stack. Run the binary in GDB and analyze the memory layout to determine the following:
   - Starting address of the buffer. You can note this by simply observing whatever address is being pushed to the stack, just before the call to the `gets` function. (remember that the function arguments are pushed on the stack, and `gets` only needs the buffer address as an argument).
   - Buffer offset from the frame pointer (not the return address).
   - Required return address (address of the `secret` function in this case).
4. Construct a payload to overwrite the return address with the address of the `secret` function. You can use the provided exploit script `exp-practice.py` to generate the payload, or you can do so manually.
5. `exp-practice.py` has variables for the buffer offset and the address of the `secret` function. You can update these variables with the values you found in the previous step.
6. You can then pass this payload to the `vuln` binary to execute the `secret` function, as follows (inside the `practice` directory):

```
$ python3 exp-practice.py | ./vuln
```

   Another method is to save the payload in some file, and then provide the input to the binary using the following command:

```
$ python3 exp-practice.py > payload
$ ./vuln < payload
```

   This is useful when you need to debug with `gdb` because you can provide the payload as input to the binary inside `gdb`. (`run < payload`)
7. If the application crashes, without printing the message, it indicates that the payload wasn't correctly constructed. You can debug the application in GDB to identify the issue.
8. Upon successful execution, the program will print a message indicating that the `secret` function has been executed.

3.1.1 Submission requirements

For this task, you are required to submit the following. Each of the following items is required and carries equal marks.

1. `exp-practice.py`: the exploit script, with the correct buffer offset and return address.
2. `task1.md`: A markdown document describing the following:
   - Buffer starting address.
   - Buffer offset from the frame pointer.
   - Return address of the secret function.
   - First assembly instruction present at the return address (should be the first instruction of the `secret` function).
   - A screenshot of the output of the binary, showing output of the `secret` function.

Put both items under a single directory named, **Task1**.

3.2 Task 2: Shellcode Execution

For the subsequent tasks, we will be using the `converter` binary, which is used by our web application. The goal is to exploit the binary, to run arbitrary code on the server, with the application privileges. Note that the binary is a `setuid` binary, and runs with the privileges of the `root` owner.

The authentication mechanism is currently disabled. You can directly access the `/dashboard` endpoint, once the app is up and running. Explore the profile section and test the profile picture upload feature using the provided PPM images (try `ppm_images/sign_1.ppm` e.g.). If you observe the logs, you'll see the output of the `converter` binary, which is converting the PPM image to a BMP image.

PPM format supports the inclusion of comments in the image files. Any line that starts with a `#` character is considered a comment and is ignored by the image processing software. For our purposes, we are assuming that the image can have only one comment, inside the PPM header (**open the `ppm_images/test.ppm` file, in a text editor, for reference, the comment is at line 4**). We can, therefore, construct a valid image file, with a comment line that can maybe used to exploit the binary.

1. Open the image file `test.ppm` in a text editor, increase the length of the comment line (append a bunch of `'a'`s to the comment) and save the file.
2. Run the `converter` binary with the modified `test.ppm` file as input. Observe if the binary crashes. If it does not, increase the size of the comment further and try again.
3. If the binary crashes, it indicates that the input is being processed incorrectly (most likely a buffer overrun in the program). You can debug the binary in GDB to identify the issue. Run the binary inside gdb, and provide the same image file as input.
4. Use the backtrace information, provided by gdb to figure out which function is causing a buffer overflow. You should identify that the function named `process_comment` inside the binary contains the buffer, which is overrun because of the use of the unsafe `strcpy` function. The buffer address, as well as the offset from the frame pointer, can be determined easily by looking at the disassembly of the function, `process_comment` and observing the arguments passed to the `strcpy` function. The application already prints the starting address of the buffer, but the offset from the frame pointer is something you need to determine.
5. Use the provided script `exp-shellcode.py` to generate the payload image. This generated payload image contains the payload inside its comment, which will be read by the `process_comment` function causing a buffer overflow, and shellcode execution.
6. Note that the addresses are different depending upon whether it's running inside GDB or not. Use GDB to determine the offset of the buffer, and note that value in the exploit script. Re-run the binary, outside gdb to determine the correct address of the buffer (which gets printed). Use this buffer address (found outside gdb) inside the exploit script.
7. Run the exploit script, to generate the `payload.ppm` image. Then, run the `converter` binary with the payload image as input. If the exploit is successful, the binary will execute the shellcode and spawn a shell.

```
$ python3 exp-shellcode.py
$ ./converter payload.ppm out.bmp
... (output skipped)
# whoami
root
```

3.2.1 Submission requirements

For this task, you are required to submit the following. Each of the following items is required, and carries equal marks.

1. `exp-shellcode.py`: The exploit script, with the correct buffer offset and starting address.
2. `converter`: Your converter binary inside the `app/bin` directory.
3. `task2.md`: A markdown document describing the following:
   - Buffer starting address.
   - Buffer offset from the frame pointer.
   - The return address at which we jumped to.
   - First assembly instruction present at the return address.
   - A screenshot of the output of the binary, showing access to the root shell, as shown above.
   - A brief description of the shellcode used. The description involves writing a brief explanation of what each assembly instruction is doing. Note that the shellcode is already present in the exploit script. But for disassembly purposes, the shellcode in a bin file is also provided (inside the `shellcodebin` directory). However, you need to use `objdump` to disassemble this shellcode. Disassemble it using the following command (assuming you are inside the `shellcodebin` directory):

     ```
     $ objdump -D -b binary -mi386 ./shellcode.bin
     ```

     This will give you the disassembled version of the shellcode. The assembly instructions are simple and trivial. If a string is being pushed to the stack, don't just give the hex representation, but also describe what string is being pushed. Similarly, if you encounter interrupt instructions, provide what syscall is being called.

Put all these items under a single directory called **Task2**.

3.3 Task 3: Reverse Shell

In the previous task, you were able to spawn a root shell on the server. However, the shell spawned was not interactive, as you need terminal access on the server to interact with the shell. You had that access, however, if we were to exploit this binary only via uploading the image from the web application, we would not have any terminal access. In that case, we need a reverse shell. A reverse shell is a shell session, with the shell's `stdin` and `stdout` redirected to a TCP connection so that we can interact with the spawned shell remotely. In this section, we'll try to first spawn a reverse shell directly on the server, as done in the previous task. Then, we'll use the same principle to exploit the application remotely, by uploading a malicious file.

1. The `exp-revshell.py` script is provided to generate the payload image for the reverse shell. This shellcode listens on port `17771` for connections, which was opened in the section 2. Make sure that port is open for inbound and outbound traffic, from the VM's security list settings.
2. The script generates the payload image, which contains the reverse shell shellcode. You must however provide the correct buffer address, as well as the buffer offset from the frame pointer, to get the correct payload.
3. Run the exploit script, to generate the `payload.ppm` image. Then, run the `converter` binary with the payload image as input. If the exploit is successful, the binary will execute the shellcode and spawn a reverse shell.

```
$ ./converter payload.ppm out.bmp
&buff: 0xffffc680
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::17771
Ncat: Listening on 0.0.0.0:17771
```

4. You can then connect to the reverse shell using `nc` from your local machine. You can also demonstrate this attack, by running `netcat` directly on the VM as well. In that case, the `VM_IP` is simply `localhost`.

```
$ nc <VM_IP> 17771
```

5. You should now have a shell on the server, which you can interact with. You won't see a prompt, so directly run commands like `whoami`, `ls`, etc. to verify that you have root access.

6. Once verified, use this payload image to exploit the application in a more realistic scenario. Make sure the app is up and running, then proceed to the profile section, and upload this generated ppm payload as a new picture. The application should be stuck on that page, as the web server is waiting for the response from the `converter` binary. You can then connect to the reverse shell, as mentioned above, using `nc <VM_IP> 17771` command.

3.3.1 Submission requirements

For this task, you are required to submit the following. Each of the following items is required, and carries equal marks.

1. `exp-revshell.py`: The exploit script, with the correct buffer offset and starting address.
2. `converter`: Your converter binary inside the `app/bin` directory.
3. `task3.md`: A markdown document describing the following:
   - Buffer starting address.
   - Buffer offset from the frame pointer.
   - The return address at which we jumped to.
   - First assembly instruction present at the return address.
   - A screenshot of the output of the binary, clearly showing it has started a `netcat` server, as shown above.
   - A brief description of the shellcode used in this exploit. Disassemble the reverse shell shellcode using the following command (assuming you are inside the `shellcodebin` directory):

     ```
     $ objdump -D -b binary -mi386 ./revshell.bin
     ```

     This will give you the disassembled version of the shellcode. The assembly instructions are simple and trivial. If a string is being pushed to the stack, don't just give the hex representation, but also describe what string is being pushed. Similarly, if you encounter interrupt instructions, provide what syscall is being called.

Put all these items under a single directory called **Task3**.

4 Submission

You are required to submit the following directories:

1. `Task1`: Submission files for Task1, as instructed in section 3.1.1.
2. `Task2`: Submission files for Task2, as instructed in section 3.2.1.
3. `Task3`: Submission files for Task3, as instructed in section 3.3.1.

Create an archive named `s_<your_rollnumber>.zip`, containing these three directories, and submit it via LMS.