ECE458/750: Computer Security
Assignment 3

Prepared by Kami Vaniea and Vinayak Sharma*

July 10, 2025

Due Date: 29th July, 2025 at 11:30 pm.

Lab VM Setup Requirements: This assignment is closely related to the SEED Labs. Please ensure that you run the commands specified in this assignment within the VM, not on your local computer. You can run the VM either on your local computer or in the cloud.
SEED Lab VM Setup: `https://seedsecuritylabs.org/labsetup.html`
Creating a SEED VM on the Cloud: `https://github.com/seed-labs/seed-labs/blob/master/manuals/cloud/seedvm-cloud.md`

Setup Files Requirements: For this assignment, you will need the setup files for the *Buffer-Overflow Attack Lab (Set-UID Version)*, available at: `https://seedsecuritylabs.org/Labs_20.04/Software/Buffer_Overflow_Setuid/`

Additional Reference: It is highly recommended to read the Buffer-Overflow Attack chapter up to section 4.5 in the SEED Labs reference book. This chapter is available at: `https://www.handsonsecurity.net/files/chapters/edition3/sample-buffer-overflow.pdf`

Working in Groups: You may complete this assignment either individually or in a group of two students. We recommend that at least one group member is comfortable with configuring and working with the SEED Lab Virtual Machine (VM), as it is essential for completing this assignment.

Questions: Use Piazza for questions and concerns.

Submission: Use Crowdmark for submission.

*Originally developed in part by Mohammadtaghi Badakhshan.

Question 1. User ID [3 marks] Please answer the following questions regarding the Linux User ID Model.

(a) Briefly explain why having a User ID Model in operating systems, such as Linux, is important for securing the operating system. You may look up information on the Internet to find your answer.

(b) Enter the following command in the VM's terminal: `seed@seed-labs:~$ cat /etc/passwd`. Briefly explain the output. What does it show? How is it formatted? You may look it up on the Internet.

(c) According to the previous question find and write down the rows corresponding to the users `seed` and `root`. Specifiy each user's UID. (You may use `grep` command. For example: `cat /etc/passwd | grep seed`)

Question 2. Process User ID [6 marks] Please answer the following questions about how UIDs are assigned to a process.

(a) Briefly explain what a Process User ID (UID) is and how User IDs are assigned to a process.

(b) Research `Set-UID` programs and briefly explain what they are. What are the differences between normal programs and `Set-UID` programs?

(c) What are the security considerations associated with `Set-UID` programs?

Question 3. Privilege Escalation [8 marks] In this question, we aim to escalate privileges from the user `seed` to `root` by running a Set-UID program. Follow these steps and report your observations:

(a) Make sure that you run all of the commands under the user `seed`. Run the following commands and turn in the output by copy pasting it.

   i. `$ id -u`

   ii. `$ ls /root`

(b) Run the following command to link `/bin/sh` to `/bin/zsh`, as `zsh` lacks security mechanisms that could prevent its execution in a `Set-UID` process.
`$ sudo ln -sf /bin/zsh /bin/sh`
(Note: Ensure that `zsh` is installed on your VM.)

(c) Change your directory to `Labsetup/shellcode/` and execute `make setuid` to compile `call_shellcode.c` as a `Set-UID` program. Then, run `ls -al` and copy paste the output.

(d) Execute either `./a32.out` or `./a64.out`. In the opened shell, enter the two commands in step (a) again and copy paste the output and explain why you are getting different outputs.

Question 4. Vulnerable Program [10 marks] Examine the code snippet below and respond to each question [1].

```c
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

(a) Explain the functionality of the `function` and `main` functions.

(b) The code has a vulnerability. Can you identify it and draw the memory stack to illustrate the issue?

(c) Why does the vulnerability occur in this code? In other words, what was the programmer's mistake?

(d) Suggest a solution to fix the vulnerability in the code.

(e) What can be the consequences of this vulnerability in a real-world application?

Question 5. Exploit [18 marks] In this question, we are going to exploit the buffer-overflow vulnerability in the `stack.c` program. The program will be executed as a root-owned `Set-UID` program. We aim to inject the shellcode from Question 3 into the vulnerable program execution.

(a) What is the vulnerability in `stack.c`?

(b) Compile `stack.c` using the `Makefile` with the `make` command. Explain how different values of `L` can change the output file.

(c) (In `Makefile`) Which security protections do these flags `-z execstack -fno-stack-protector` disable? How can disabling them make the code vulnerable? Why do we need to disable them to exploit the vulnerability in `stack.c`?

(d) Before starting the attack steps, we need to turn off the VM operating system security mechanisms:

- **Address Space Randomization**: To disable address space randomization, enter the following command:
  `$ sudo sysctl -w kernel.randomize_va_space=0`
- **Configuring /bin/sh**: Link `/bin/sh` to `/bin/zsh`.
  `$ sudo ln -sf /bin/zsh /bin/sh`
  (Note: Ensure that `zsh` is installed on your VM.)

Make sure you have successfully disabled the security countermeasures. Then, briefly explain the *Address Space Randomization* mechanism.

(e) To complete our attack, we need to overwrite the memory location on the stack that contains the return address from `bof` to `dummy_function`. Can you explain why?

[1]Source: https://insecure.org/stf/smashstack.html

(f) Use `gdb` to determine the address of the memory location on the stack that contains the return address from `bof` to `dummy_function`. Use `gdb-peda$ p $ebp` to print the current value of the base pointer ($ebp).
(Hint: Follow the instructions in Section 5.1 of "Buffer_Overflow_Setuid.pdf" and explain each step you took to determine the memory address. Also see Section 4.5.3 in the Buffer-Overflow Attack chapter in the SEED Labs reference book.)

(g) Use `gdb` to determine where the buffer starts in the memory, and what the distance is between the buffer's starting point and the return address field? Do this for all four buffer size levels (i.e., `stack-L1-dbg`, `stack-L2-dbg`, `stack-L3-dbg`, and `stack-L4-dbg`). Report your observation.

(h) In this step, you should generate the content of the `badfile` to inject the malicious assembly shellcode into memory and execute it by exploiting the vulnerability. You should complete and use `exploit.py`. Report how you completed `exploit.py` for `L1`, `L2`, `L3`, `L4`.

(i) Exploit the vulnerability for all four buffer size levels (i.e., `./stack-L1`, `./stack-L2`, `./stack-L3`, and `./stack-L4`) and report your observations.

Note: (In `Makefile`), `-g` is employed to embed debugging information into the executable file produced by the compiler. This debugging data enables tools like the GNU Debugger (GDB) to effectively analyze and debug the program[2].

Question 6. Self-Reflection [5 marks] The three questions below must be answered individually by **each group member**. If you are working alone, you should still complete this section.

(a) What did you learn from this assignment?

(b) Which part was the most challenging or interesting for you, and why?

(c) If you worked in a group, how did you divide the work?

Include all self-reflection answers in your final report. Clearly label each reflection with the contributor's full name and Waterloo ID.

Submission Instructions:

Submit a single PDF with answers to all the above questions on Crowdmark.

[2]For more information: https://dmalcolm.fedorapeople.org/gcc/2015-08-31/rst-experiment/options-for-debugging-your-program-or-gcc.html