CS5231 – Systems Security

Homework 2: Binary Exploitation

Due date and time: 23:59pm, October 9, 2023. This is a firm deadline. This homework MUST be finished independently.

1. Introduction

In this homework, you will get familiar with various types of memory corruption vulnerabilities and different ways to exploit them.
This assignment is divided into two parts --- PartA and PartB. The PartA consists of 7 custom programs with hand-crafted memory corruption bugs while the PartB consists of a real-world program (*sudo*) with inherent memory corruption vulnerabilities.

2. Environment Setup

2.1. Download VM

A Linux VM image is given to you which contains the PartA and PartB materials in their respective folder under **~/A2** directory.

If you use an Intel machine, you can download the VM from the following link:
https://drive.google.com/file/d/15drMziY3nONad0Xvuzc3QwWTRZHxzC6g/view?usp=sharing
If you use an Apple Silicon Macbook, you can download the UTM file via the following link:
https://drive.google.com/file/d/1qUSmF_eh_MRG9bwJ6vcTma6J7yeh5ISv/view?usp=sharing
Note that the provided UTM image is installed with non-GUI mode Debian machine. You can use the command `ifconfig` to figure its IP address, say 192.168.64.10. Then use the command `ssh student@192.168.64.10` from the terminal of MacOS to access the environment.

Logging in the VM. Both the user account and the password of the VM are `student`.

2.2. Generate File for Random Seed

After logging into the VM, please run the following command to generate SID file:

```
~/A2/gen_sid.sh
```

The vulnerable programs are parameterized by this SID file. The SID file will be checked during evaluation and should NOT be changed after generating. Please do not modify it once confirming its correctness.

You will be prompted for the NUSNET ID (*starting with e*) to generate SID file. Please be reminded that it is not the STUDENT ID. **Please make sure** that the information you provide is correct (please double check the SID file after running `gen_sid.sh` to make sure the content of this file is replaced by `exxxxxxx`: which is starting with your own NUSNET ID).

2.3. Disable ASLR

Address Space Layout Randomization (ASLR) is enabled by default on the VM which you received. In order to complete some of the assignments, ASLR will have to be disabled. To do so, you can execute the following commands as root:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Note that this option will be reverted every time the VM is restarted so do remember to disable ASLR again if the system is restarted.

Alternatively, you can execute the following command to make the ASLR setting persistent:

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

3. Part A [12 points]

What is given? For this section, we provide you with 7 vulnerable programs. The programs are located in `~/A2/PartA/programs` (`/home/student/A2/PartA/programs`).

Each of these programs has a memory safety bug in it. We have provided the skeleton for 6 exploits. The first five programs `program1.c`, `program2.c`, …, `program5.c` are compiled with an executable stack while `program6.c` and `program7.c` have a non-executable stack[1].

What should you do? Your task is to identify the type of vulnerability in the programs and write an exploit code for it. You need to work on **program1 (1 point)** and **program 7 (3 points)** and **choose 4 programs (2 points each) out of the rest 5 programs, program2 to program 6**. Each exploit code from 1 to 6 when executed should yield a shell (`/bin/sh`). If the vulnerable program has the *setuid* bit set to root it should yield a root shell (at least when `/bin/sh` is a symlink to older version of dash). You can use any of the attack techniques like arbitrary code injection, return address corruption, etc. discussed in the class to exploit `program1.c` to `program5.c`. Due to the non-executable stack found in `program6.c` and `program7.c`, **you are expected to use return-to-libc (for `program6.c`) and ROP (for `program7.c`) that can bypass the No-eXecute prevention.** In each of the exploit programs, you should use the actual stack location of the vulnerable program that is corrupted. Note: You

[1] The Makefile in **~/A2/PartA/programs** is used to compile the programs.

are not allowed to use `get_sp` function as explained in the reading material “Smashing the stack for fun and profit”.

The `program7.c`, when exploited, should print out the contents of `flag.txt` in `/home/victim` directory. Note that it takes input from keyboard (`stdin)` and reads the file `./exploit-file`, so you need to fill two files (`exploit7-file` and `exploit7-stdin`) with your exploit code in them and run make install to copy them to the `/tmp` directory. After that, you can execute `exploit7-run.sh` to run the vulnerable program7.
Note: Program6 should be exploited with return-to-libc and program7 should be exploited with ROP. For program7, you need to provide an explanation in `exploit7-explain.txt`.

3.1. Inputs: `/home/student/A2/PartA/Programs.`

You are given 7 vulnerable programs for the PartA in the `~/A2/PartA/programs` folder. You need to compile and install these programs in `/tmp` directory. Perform the following steps to compile your programs:

Generate vulnerable programs. You should have generated the SID file as shown in section 2.3 before this step. Then run this:

```
make generate
```

Compile. To compile the vulnerable programs, run this:

```
make
```

Installation. To install the vulnerable programs in `/tmp` directory:

```
make install
```

Every time you reboot the VM, you will need to install the vulnerable programs in `/tmp` directory by running `make install` in the `/home/student/A2/PartA/programs` folder.

Setuid root and replace symlink of /bin/sh. After you confirm that your exploit can yield a shell, you can perform the steps below to check if it can yield root shell when setuid bit of the binary is set to root. This is because setuid root will make it difficult to debug the programs. However, just setuid root might not be sufficient if `/bin/sh` is a symlink to newer bash or dash implementation that will drop privileges if the shell is started with the effective user (group) I not equal to the real user (group) id.

You can run the following commands as root to test your exploits with older dash implementation:

```
make setuid
rm /bin/sh && cp -P /bin/s1 /bin/sh
```

You can also recover the default /bin/sh by running:

```
rm /bin/sh && cp -P /bin/s0 /bin/sh
```

3.2. Output: `/home/student/A2/PartA/exploits`

This directory contains the skeleton solutions (e.g., `exploit1.c`, `exploit2.c`, ...) used to generate exploit for their respective programs (e.g., `program1.c`, `program2.c`, ...). The purpose of the files found in this directory is as follows.

`exploitsX.c` (X = 1,2,...6). You are given 6 skeleton exploit programs in the `PartA/exploits` folder. You need to write the code in these skeleton programs to exploit the corresponding vulnerable program. This program executes the corresponding vulnerable program (e.g.: exploit1.c will execute program1.c) using `execve` and passes the argument from the exploit code.

`shellcode.h`. Included in this directory is `shellcode.h` file which contains a copy of Aleph One's shellcode from "Smashing the stack for fun and profit". You can use the shellcode by including `shellcode.h` in your exploit program. For program7 you are not given a skeletal exploit program.

Makefile. The `Makefile` contains instructions used by make to compile the exploit programs.

4. Part B [8 points]

This section contains a real-world program --- *sudo* that has memory corruption vulnerabilities. **NOTE: You should not recompile the program from the source code. Recompiling will change the binary and it will no longer match the binary in the grading system.** The source code is only for you to analyze the vulnerabilities. The compiled vulnerable program is already installed.

SUDO: `/home/student/A2/PartB/sudo`

The VM has a vulnerable *sudo* program placed at `/usr/local/libexec/sudo`. The source code for this program is present in this directory. Students are expected to perform a data-oriented attack on this program under two different scenarios --- ASLR enabled and ASLR disabled.

What is the vulnerability? The *sudo* program has a format string vulnerability in the sudo debug function found in `sudo/src/sudo.c` as shown below.

```
void sudo_debug(int level, const char *fmt, ...)
{ va_list ap;
  char *fmt2;
  if (level > debug_level)
    return;
  /* Backet fmt with program name and a newline to make it a single write */
  easprintf(&fmt2, "%s: %s\n", getprogname(), fmt); <---- the bug is here
  va_start(ap, fmt);
  vfprintf(stderr, fmt2, ap);
  va_end(ap);
  efree(fmt2);}
```

The vulnerable function is found in line 8. Specifically, getprogname() returns an attacker-controlled string which is used as part of the format string. The following figure shows a simple proof of concept that you can use as a starting point for your exploit. In the final solution, you are required to replace the %n in YOUR_SOLUTION with the actual format string to be used in the exploit. During evaluation, your ./sudo_exploit1.sh will be executed. The content of the file ./sudo_exploit1.sh is:

```
#!/bin/bash
.....
cp -f ./test.sh /tmp/test.sh
YOUR_SOLUTION="%n"

ln -s /usr/local/libexec/sudo $YOUR_SOLUTION
env -i SUDO_ASKPASS=/tmp/test.sh ./$YOUR_SOLUTION -D9 -A ls
```

Attack Explanation. We now provide the explanation of the various steps and parameters used in the above proof of concept that will help you in understanding the attack.

- Creating a symlink to the vulnerable *sudo* program installed in /usr/local/libexec/sudo allows the attacker to specify an arbitrary string to be used as the program name. Since the program name is used to create a format string, this allows the attacker to create any arbitrary format string.
- The -D9 option will call the sudo_debug function triggering the code path that exhibits the vulnerability.
- The -A option executes a script pointed by the SUDO_ASKPASS environment variable. We have set SUDO_ASKPASS=/tmp/test.sh where /tmp/test.sh is copied from ./test.sh. This /tmp/test.sh reads /etc/shadow and write to /tmp/test.log.

Goal. The goal is to use this format string vulnerability to corrupt the data in the variable user_details.uid in sudo.c. You need to create a format string to replace YOUR_SOLUTION in the given PoC such that the value of user_details.uid is corrupted to 0. This will grant root privileges to the process in execution. Finally, executing the exploit script should copy the /etc/shadow and write to /tmp/test.log.

SUBPART 1 [4 points]. For the first subpart, you need to perform the above attack with ASLR disabled. The solution for this should be provided in the sudo_exploit/sudo_exploit1.sh file. We have provided skeleton exploit code in this file.

SUBPART 2 [4 points].

a) Elaborate the steps to achieve the above exploit, even when your exploit is not functional.

b) When ASLR is turned on, describe how it affects the exploit. (sudo_exploit/sudo_exploit2.sh is provided in case you'd like to try, but the exploit code is not graded).

5. Online Judge Setup

We are using an automated Online Judge system, which allows to use GitHub repositories for assignment submissions. Please follow the steps below for setting it up.

1. Create a private GitHub repository with name CS5231_hw2_A0xxxxxxx (your STUDENT ID)
2. Setup SSH key pair inside the VM to login GitHub: https://www.inmotionhosting.com/support/server/ssh/how-to-add-ssh-keys-to-your-github-account/

3. Invite the TA's GitHub ID `icegrave0391` as a collaborator.
4. Submit your repository URL on Canvas in the quiz "CS5231 -> Quizzes".
5. Change directory to **~/A2**, and then run **~/A2/setup_repo.sh** to initialize your **~/A2** folder and link to your private GitHub repository.

Note: If you are using the **SOC network**, you may encounter problems connecting to GitHub via SSH. You may refer to this link for further help: https://docs.github.com/en/enterprise-cloud@latest/authentication/troubleshooting-ssh/using-ssh-over-the-https-port

Note: To ensure a uniform environment for the assignment and avoid compatibility issues, please refrain from using other systems. You may install new software on the VM if needed, however, the Online Judge will evaluate your submission in the VM handed out to you (i.e., Your submission should not rely on further packages or assumptions).

6. Request for Evaluation and Final Submission

The Online Judge system allows you to request automated evaluations at any time and get feedback in a few minutes. To request an evaluation, simply include the keyword `judge` (case insensitive) in the commit message of the commit that you want to be evaluated and push it to your GitHub repository. Note that the commit needs to be reachable from your main branch (**not** other branches such as `master`, etc.). Then, the Online Judge system will automatically pull your commit and evaluate your submission for correctness of all solutions and will push the results back to your repository on another branch named `eval-results`. **We will discuss the submission process in the coming lecture and then open the evaluation system.**

The initialization process (see Section 3) should have already initialized a repository and set up the remote repository URL in `~/A2`. All the files that you need to submit have also been whitelisted in `.gitignore`. You may directly start using `git add -A`, `git commit` and `git push` in that directory to make your submissions. For example, you may use the following commands to make a submission:

```
cd ~/A2
git add -A
git status
git commit -m 'judge'
git push
```

and you may replace `judge` with other comments to push your repository without requesting an evaluation. After 3 ~ 10 minutes, you should be able to see the auto evaluation results.

Important Notes:

1. Your last commit that contains the keyword `judge` will be considered as you Final Submission. The system will stop pulling from your repository automatically upon deadline.

2. The Online Judge system will not detect cheating. It will only detect if you successfully start `/bin/sh` or other signs of a successful exploit. **Automated inspection to detect cheating and additional constraints is performed after the deadline**. The following will be seen as cheating:

(a) Passing the Online Judge evaluation without exploiting the target program (such as run/bin/sh directly).
(b) We have conclusive evidence that you are copying other's solution or explanation.

3. Some tasks require specific techniques and constraints. We will not give you the mark if you failed to solve it under the specific constraints or the explanation is unreasonable. A summary of those special cases are:

(a) **program6**: Use return-to-libc
(b) **program7**: Use ROP and write an explanation
(c) ***sudo* subpart2**: Write an explanation for part 1.

4. When an evaluation is triggered, the system will also save a snapshot of your files that are being evaluated and keep the latest. Only the latest snapshot will be used for cheating detection and manual inspection after the deadline, not the repo or earlier snapshots.

Compatibility. The Online Judge server is running on the same VM handed out to you. It is, for that matter, useful for you to ensure validity/correctness of your solutions on your VM, before submitting them (pushing with `judge` keyword in commit message).

Size Limit. Please keep the total size of your repository within 20 MB (You should NOT include any compiled files in your git repo).

Time Limit. We limit the execution time of each exploits to 5 seconds (for PartA) or 10 seconds (for PartB).

7. Reading Materials

Suggested reading in Phrack, www.phrack.org (You don't need to read all of them):

Aleph One, "Smashing the Stack for Fun and Profit," Phrack 49 #14.

klog, "The Frame Pointer Overwrite," Phrack 55 #08.

Bulba and Kil3r, "Bypassing StackGuard and StackShield," Phrack 56 #0x05.

Silvio Cesare, "Shared Library Call Redirection via ELF PLT Infection," Phrack 56#0x07.

Michel Kaempf, "Vudo - An Object Superstitiously Believed to Embody Magical Pow-ers," Phrack 57 #0x08.

Anonymous, "Once Upon a free()...," Phrack 57 #0x09.

Nergal, "The Advanced Return-into-lib(c) Exploits: PaX Case Study," Phrack 58 #0x04.

Revision History:
V1. September 15, 2023. Initial version.