Homework #3: Software/System Hacking Competition

Due: Jun. 11, 11:59 PM
Responsible TA: Jaeho Bae (bjho@unist.ac.kr)

- **Updated notes.** The question in Problem 4-(b), "Is it enough to use your exploit (i.e., shellcode) in order to read the flag?", is asking whether the shellcode you wrote in Problem 3 would be sufficient to read the flag if it were executed via a control flow hijacking attack.
- **Environment.** We use Debian 10 for the CTF environment. You can easily access and use this environment by accessing to the server at `10.20.12.123` vis `ssh` with the account information provided to each student. Note that this server can only be accessed from the UNIST internal network, so make sure to use a VPN if you are connecting from outside.
- **Late submission policy.** Late submission will be assessed a penalty of 10% per day (We will only accept late submissions of up to 3 days).
- **Submission guidelines.**
  - You should submit both your report and flags.
    - Solve the problems and submit the flags to our homework webpage: `http://10.20.12.187:4000`. This server can only be accessed from the UNIST internal network. Use a VPN to access from outside.
    - You should upload a single PDF file on BlackBored. Your report must describe the answer to each question in this homework. Your report (`Your_ID-hw3.pdf`) can be written in either English or Korean.
  - The name of the PDF file should have the following format: `Your_ID-hw3.pdf`. For example, if your ID is 20231234, then you should submit a file named `20231234-hw3.pdf`.
  - If your solution includes some code (assembly, C, etc.), you should embed them in your PDF.
- **Capture The Flag (CTF) guidelines.**
  - You can find each problem on our homework webpage: `http://10.20.12.187:4000`
  - If you solve each challenge, you will be able to obtain a flag. Submit the found flag to the website. Each flag is in the following format: `flag{some_string}` (e.g., `flag{C0N9R@7u1aT1on!}`).
  - For your information, Section 7 offers instructions for crafting exploits and configuring GDB.
  - Your score in the CTF scoreboard is nothing to do with the actual score for your homework. The CTF score is just for fun. Your work will be evaluated according to *the quality of the submitted report*. Make sure to write the report as thoroughly and clearly as possible.
  - Do not attack the CTF environments, including web services!
  - If you think the services are not working properly or have any questions, please publicly upload your question on the BlackBored.

Problem 1. CTF Warm-up: Basic (10 points)

In this challenge, you need to donwload and analyze the provided binary.

(a) (3 points) Reverse engineer the main function of the provided binary (`basic`), and show the corresponding C code. What does this program do?

(b) (3 points) There is a function that is not affected by control flow. What is the name of that function?

(c) (4 points) Reverse enginner aforementioned function, show the corresponding C code, and get the flag.

Problem 2. RetFunc (13 points)

The program `retfunc` is running on a remote machine. It is running as an `xinetd` service. Download the binary `retfunc` from the CTF website, and answer the following questions.

(a) (5 points) Reverse engineer the main function of the provided binary (`retfunc`), and show the diagram of the stack layout after the execution of instruction of `0x8049201`.

(b) (3 points) There is a function that is not affected by control flow of the program. Reverse enginner this function, show the corresponding C code.

(c) (5 points) What is the requirement for your exploit in order to read the flag? Explain in detail how you exploited the program. If you used a script to exploit it, include the script in the writeup. If you show your work, you can get partial points even if you cannot get the flag.

Problem 3. ShellEval (16 points)

Shellcode is a small piece of code that runs a command-line shell such as `/bin/sh` on Linux or `cmd.exe` on Windows. Spawning a command-line shell is important in terms of exploitation because it allows an attacker to run any arbitrary code. In this problem we ask you to write your own shellcode.

(a) (1 point) Locate a file in the provided Linux environment that has a complete list of syscall numbers for Linux x86.

(b) (1 point) What is an ABI (Application Binary Interface)? Why is this related to syscalls?

(c) (1 point) Describe the syscal calling convention of Linux x86.

(d) (6 points) Write your own shellcode in 32-bit x86 assembly (i.e., .s file) that spawns a shell. Make sure that your shellcode meets the following conditions: (1) assembly code for your shellcode must include detailed comments; (2) when compiled, the binary representation of your shellcode should have size no larger than 100 bytes; (3) the binary representation of your shellcode should not contain any zero (NULL) and `\x0a` (new line) byte; and (4) your shellcode should be semantically equivalent to the following C code snippet: it invokes `setreuid` with `geteuid()` as first/second parameter, and then finally execute the command `/bin/sh` with `execve`.

```
\\ Shellcode written in C
# include <stdio.h>
# include <unistd.h>

void shellcode() {
    char* shell[] = {"/bin/sh", NULL};
    setreuid(geteuid(), geteuid());
    execve(shell[0], shell, 0);
}
```

(e) (5 points) Describe the meaning of the `setreuid(geteuid(), geteuid())`. If an attacker were to successfully inject this shell into a vulnerable program and run it, what privileges would they gain?

(f) (2 point) Inject your shellcode to the program `shelleval` and get the flag located in `/home/shelleval/flag.txt`.

Problem 4. RetShell (20 points)

In the forth CTF problem, you need to gain the local privilege escalation by exploiting a bug in a poorly written program called `retshell`.

To work on this problem, we recommend you to create your own directory at `/tmp`. Create a directory with a random name so that people cannot guess it easily. This is going to be your working directory. For example, you can write a script in the directory while exploiting the program, or you can put your GDB script inside the directory. Remember every student will use the same user ID, and therefore, they can see your files if they know the path to your working directory. This is the reason why you want to make your directory with a random string that cannot be guessed easily. For your information, this machine has GDB, VIM editor, Python, and Perl installed.

See the description in the CTF web page and answer the following questions.

(a) (1 point) Who is the owner of the flag located at `/home/retshell/flag.txt`? What kinds of permission does the flag have?

(b) (1 point) Is it enough to use your exploit (i.e., shellcode) in order to read the flag? Explain.

(c) (10 points) Reverse engineer the function `printer`, and show the corresponding C code. What does this program do? What kind of vulnerability does this program have?

(d) (8 points) Exploit the program (`retshell`) in order to read the flag. Make the best use of your shellcode you wrote in the previous problem. Explain in detail how you exploited the program. If you used a script to exploit it, include the script in the writeup. If you show your work, you can get partial points even if you cannot get the flag.

Problem 5. BadFormat (20 points)

In this CTF problem, you need to gain local privilege escalation by exploiting a bug in a poorly written program called `badformat`. Use `/tmp` directory as you did in Problem 4.

(a) (10 points) Reverse engineer the program (`badformat`), and show the corresponding C code. What does this program do? What kind of vulnerability does this program have?

(b) (10 points) Exploit the program (`badformat`) in order to read the flag. Explain in detail how you exploited the program. If you used a script to exploit it, include the script in the writeup. If you show your work, you can get partial points even if you cannot get the flag.

Problem 6. RetRet (35 points)

> Turn on the Address Space Layout Randomization (ASLR) protection for your environment in order to work on this problem.

The program `retret` is running on a remote machine. It is running as an `xinetd` service. Download the binary `retret` from the CTF website, and answer the following questions. To run this binary locally, you can either set up the `xinetd` service, or you can simply use `netcat` as follows:

```
nc -l -p 33333 -e ./retret
```

Notice there are two different versions of netcat on Debian/Ubuntu. You can install the correct version by typing:

```
apt-get install netcat-traditional
```

(a) (1 point) Check whether the binary is protected with non-executable stack (NX).

(b) (5 points) Reverse engineer the `whatisthis` function, and show the corresponding C code. What does `whatisthis` function do? Can you name a `libc` function that has the same semantics as `whatisthis`?

(c) (3 points) Do you think ASLR can be applied to the code section of this binary? Why or why not?

(d) (8 points) Pinpoint a vulnerability in this program that allows a control-flow hijack exploit. Explain how you can exploit this vulnerability at a high level.

(e) (10 points) Enumerate ROP gadgets in the program that you are going to use, and explain in detail why you chose such gadgets. (Hint: you should remember a way to invoke syscalls)

(f) (8 points) Show your final exploit script, and explain in detail how you obtained the flag.

7 Appendix

Here, we offer instructions for crafting exploits and configuring GDB. These guidelines are intended to lower the entry barrier for participating in our CTF and are not mandatory to follow (There are numerous ways to generating exploits and configuring GDB).

7.1 Crafting Exploits using Python3

It is recommended to use a programming language like Python or Perl to create your own exploit. Writing a single line of code is more efficient than entering 'A' (`0x41`) 400 times. We provide guidelines for crafting exploits using `Python 3`.

Writing byte sequences to stardard output. In this example, we describe how to write byte sequences to standard output using Python 3. Let's assume our script is named "`exploit.py`".

```
1  import sys
2  payload = b`'
3  payload += b`A' * 400   \\ Add \x41 for 400 times
4  payload += b`\xef\xf9\x12\xea'  \\ Add 0xea12f9ef to the payload in
5                                  \\ little-endian manner
6  sys.stdout.buffer.write(payload)  \\ write bytes in standard output
7  print() \\ recommended to use for the nc-based exploitation
8          \\ to flush the remote server's stdin buffer
```

Feed your exploit to the program. If you want to inject byte sequences as input into your program using the "`exploit.py`"script, use the following commands.

```
\\ Feed the exploit via standard input
$ [Binary Program] < <(python3 exploit.py)
$ nc [Target IP] [Port No] < <(python3 exploit.py)

\\ Feed the exploit via standard input and interact with the spawned shell
$ [Binary Program] < <(python3 exploit.py; cat)
$ (python3 exploit.py; cat) | nc [Target IP] [Port No]

\\ Feed the exploit via system argument
$ [Binary program] `python3 exploit.py`
```

7.2 GDB Setting

As we discussed in class, the buffer address identified through GDB is not the same as without GDB in normal situations. This discrepancy arises because GDB adds extra environment variables, such as `LINES` and `COLUMNS`, and there may be other differing environment variables as well, i.e., _ variable. To resolve this issue, we need to enter a specific set of commands in GDB. For example, in Problem 6, you can address this problem by following these commands.

```
1  $ gdb badformat
2  (gdb) unset env LINES
3  (gdb) unset env COLUMNS
4  (gdb) set env _=/home/badformat/badformat
5  (gdb) r < <(python3 exploit.py)  \\ Run badformat with your exploit in GDB
6  Starting program: /home/badformat/badformat \\ Run badformat with an absolute path
```

For problems that need to be solved on our server via `ssh` access (e.g. Problem 4), note that we have pre-entered the above command into `.gdbinit`. Additionally, note that, as you can see in Line 6, GDB runs the program with an absolute path. To ensure reliable exploit results, we recommend running the program with an absolute path even when you are not using GDB.