Buffer Overflow Exploitation

Pedro Adão

September 2017

The goal of this session is to analyse and exploit buffer-overflow vulnerabilities. Buffer-overflow vulnerabilities usually occur when someone is allowed to write and/or to execute code in areas that one should not, and usually derives from the usage of unsafe function like `gets`.

Modern OS's and compilers already incorporate some security features that prevent these attacks such as *canaries*, *Executable space protection* (XP)/*Data Execution Prevention* (DEP), and *Address Space Layout Randomisation* (ASLR).

Canaries are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will be the canary, and a failed verification of the canary data is therefore an alert of an overflow; *Executable space protection* (XP) prevents certain memory sectors, e.g. the stack, from being executed; and *Address Space Layout Randomisation* (ASLR) is a technology used to help prevent shell-code from being successful. It does this by randomly offsetting the location of modules and certain in-memory structures.

Combining these techniques makes it very difficult to exploit vulnerabilities in applications using shell-code or return-oriented programming (ROP) techniques. In order for our attacks to succeed we'll thus need to disable them. For that we'll need to compile our programs using the following flags (we use here `vuln.c` as the program to be compiled, and `vuln` as the generated binary).

Disable canaries: `gcc vuln.c -o vuln -fno-stack-protector`

Disable XP: `gcc vuln.c -o vuln -z execstack`

To disable ASLR we do it for the whole system:

Disable ASLR: `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`

Disable ASLR now as we need it for all our exercises.

1 Setup

These exercises will run on the machine downloaded for the first lab. You may want to install some of the auxiliary tools by running `./basic-install.sh`.

Task 1 – Exercises with No Protection

0-Simple Overflow

Let us start with program `0-simple.c`. Compile this file with no-canaries. The goal of this attack is to print the message "YOU WIN!!!" in the screen. How can we de it?

- Recall how variables are recorded in the stack; can variable `buffer` interfere with variable `control`?
- You might want to use GDB to see where `buffer` and `control` are stored in memory.

1-Match an Exact Value

Now that you know how to overflow a buffer, can you do it with an exact value?. Compile `1-match.c` file with no-canaries. The goal of this attack is to print the "Congratulations" message in the screen. Can you do it?

- Notice that the argument is input in the command line, ie, `./1-match <string to write in buffer>`;
- Recall that `0x61626364` is the string `abcd`;
- Have you heard of *little-endian* and *big-endian*?

2-Calling Functions

Ok, we already know how to overflow a buffer in a controlled way and change variables. But can I call a function that is not called anywhere in our code? Compile `2-functions.c` file with no-canaries. The goal of this attack is to call function `win` and print the "Congratulations" message in the screen. Can you do it?

- Recall that the name of a function in C is the address where this function is written in the memory;
- Can `fp` be `win`?

3-Return Address

We now know everything about the stack and how to change its values, change the functions that are called and so on. But can I call a function *even if NO function is called* anywhere in our code? Compile `3-return.c` file with no-canaries. The goal of this attack is to call function `win` and print the "Congratulations" message in the screen. Can you do it?

- Recall that the name of a function in C is the address where this function is written in the memory;
- Is it true that no function is called in our program? How can I call `win`? Recall how the stack is organised and what are the values stored in the stack.

4-Unhauthorized Access

The goal of this attack is to gain access to the system without introducing the correct password *V3RY53cr37*. Can you do it? Compile `4-auth.c` file with no-canaries and elaborate on Exercise 1.

5-Unhauthorized Access

The goal of this attack is again to gain access to the system without introducing the correct password *V3RY53cr37*. But now, we have changed lines 11 and 12 of the previous program. Can you still do it? Compile `5-auth.c` file with no-canaries and elaborate on Exercise 3.

6-Running Arbitrary Code

Congratulations, You know now how to change the values of your stack variables, and to modify the expected flow of execution of a program. But can we do more? Can we exploit buffer-overflows to run arbitrary code?

The goal of this attack is to run several pieces of code that do nasty things. Compile `6-code.c` file with no-canaries and no XP. You should do it in `root` mode. Type

```
sudo gcc 6-code.c -o 6-code -fno-stack-protector -z execstack
```

To run the program with some *attack-string* that is stored in `file` run

```
cat <file> | sudo ./6-code
```

Shell-code The pieces of code we want to execute are available in the file `test-shellcode.c`. Without getting into much details the 4 pieces of code in this file are the low-level instructions of programs that:

1. Print "Hello world";
2. Print the file `/etc/passwd` on the screen;
3. Reboot our machine;
4. Open a connection that allows one to control this shell remotely.

If you want to try these pieces in isolation just uncomment one at a time and run

```
gcc test-shellcode.c -o test-shellcode -fno-stack-protector -z execstack; ./test-shellcode
```

Some important notes

2. Change the permissions of `/etc/passwd` so that only root can read and write to this file.

   ```
   sudo chmod 600 /etc/passwd
   ```

   Can you still run this code and get the output? No?

   Good! We'll explore this later in our examples. **For now let's revert it otherwise you might not be able to login again :-)**

   ```
   sudo chmod 644 /etc/passwd
   ```

3. Be carefull and save all your open files before running this one! **Also, when you boot your machine up again, do not forget to disable ASLR again.**
4. Read the instructions and follow the steps indicated in the `test-shellcode.c` file for this case. What happens when you do `ls`? And `ls /root`? And if you do `sudo ls /root`?

Injecting Shell-code Now that you have experimented the 4-pieces of code and that you know that they do what they are supposed to do (in your machine) when you have the correct privileges, can you do it in another machine/process? Saying it differently, can you execute them without running `./test-shellcode`?

2. Notice that `6-code` is running as `root`. Can you make it show the `/etc/passwd` file when you change its permissions to 600? **Never forget to revert the permissions of `/etc/passwd` back to 644 afterwards!!!**
4. Supposing that `6-code` is running on a different machine (as root), can you read the content of its directories from the second terminal? Can you `ls /root` now from the second terminal? And from the first terminal?

7-Environment Variables

Good, we already know how to run arbitrary code. but what happens when the buffer we can inject is to small? The goal of this attack is to run the same pieces of code as in the previous exercise but when the buffer is much smaller (128 to 16). Compile `7-environment.c` file with no-canaries and no XP.

- Have you heard off environment variables?
- `export MYVAR = ....  ; echo $MYVAR`?
- Try to find this variable's address in order to run the attacks.

Task 2 – Enabling ASLR

Now, we turn on the Ubuntus address randomisation. We run the same attacks developed previously. Can you get perform them? If not, what is the problem? How does the address randomisation make your attacks difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomisation:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the result, how about running it for many times? Can you attack this way?

Task 3 – Enabling Canaries

Before working on this task, remember to turn off the address randomisation first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the Stack Guard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the `-fno-stack-protector` option. For this task, you will recompile the vulnerable programs, execute Task 1 again, and report your observations. You may report any error messages you observe.

Task 4 – Enabling Non-executable Stack

Before working on this task, remember to turn off the address randomisation first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable programs using the `-z noexecstack` option, and repeat the attack in Task 1. Does it work? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your lab report.

It should be noted that non-executable stack only makes it impossible to run shell-code on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The return-to-libc attack is an example.

Task 5 – Testing and Fixing

Use the tool `flawfinder` to detect possible bugs in our programs.

```
flawfinder file
```

Based on the reports, can you fix the original programs (even with the protections disabled)? You should describe your observations and potential corrections in your lab report.

References

[SEEDLabs] `http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/Buffer_Overflow.pdf`

Extras – Strings with non-printable chars

To perform these attacks you may need to input chars that are non-printable. The easier way to do it is to write such input string to a file, and then use this file as input to the program. For this, do the following:

```
python -c 'print string1 + string2 + ...  + stringn' > input-file.txt
```

```
./program < input-file.txt
```

Example:

```
python -c 'print "\x48\x31\xc0\x48" + "\x90" * 30 + "\x7f\x00\x00"' > input-exerciseX.txt
./file < input-exerciseX.txt
```

Extras – Some GDB commands

`gdb ./file` — open GDB to debug `file`

`disas function` — disassemble function `function`

`disas address` — disassemble function at this `address`

`b address` — inserts a breakpoint at address `address`

`r < file` — run the program with input `file`

`s` — execute next step of program

`c` — execute until the next breakpoint of the program

`stack n` — show the `n` registers after the stack pointer

`x/nx $rsp` — show the `n` registers after the register `$rsp`

`x/nx address` — show the `n` registers after the address `address`