TP Class Assignment - 29/Apr/2019

Each group must answer the questions of the following exercises in the Github area of their group until the end of 10/May/2019. For each day of delay, 0.15 points will be deducted from the grade of this assignment.

Note that these exercises should be done in the virtual machine provided.

Exercises

1. *Buffer Overflow*

Instructions for anyone who is not using the virtual machine provided:

1. For this exercises you need to install gdb and GDB-Peda in the user account *user* in the virtual machine. It is suggested that you perform the following command sequence:

```
sudo apt-get install gdb

cd ~/

git clone https://github.com/longld/peda.git ~/bin/peda

echo 'source ~/bin/peda/peda.py' | sudo tee --append ~/.gdbinit > /dev/null
```

2. Additionally, save the files in the directory Aula 11.a to your local machine in the directory /home/user/Aulas/Aula11.a

Experience 1.1 - Organization of program memory

Use the Unix `size` command to see the size, in bytes, of the text, data, and bss segment of the programs size1.c, size2.c, size3.c, size4.c, and size5.c, by following these steps:

1. Compile the various files. Ex: `gcc -o size1 size1.c`
2. Execute the `size` command for each of the executable files. Ex: `size size1`
3. Check and explain the value differences in the various text, data, and bss segments.

Experience 1.2 - Organization of program memory

Use the program memoryLayout.c to check the layout of the program memory. Make sure the results are the expected ones (note that compiler optimizations and machine architecture can lead to slightly different results).

Note: You will have to compile and run the program.

Experience 1.3 - Buffer overflow in multiple languages

Check what happens in the same program written in Java (LOverflow.java), Python (LOverflow.py) and C++ (LOverflow.cpp), by running them.

Explain the behavior of programs.

Question P1.1 - Buffer overflow in multiple languages

Check what happens in the same program written in Java (LOverflow2.java), Python (LOverflow2.py), and C++ (LOverflow2.cpp) by executing them.

Explain the behavior of programs.

Question P1.2 - Buffer overflow in multiple languages

Check what happens in the same program written in Java (LOverflow3.java), Python (LOverflow3.py), and C++ (LOverflow3.cpp) by executing them.

Explain the behavior of programs.

Experience 1.4 - Buffer overflow in multiple languages

Check what happens in the same program written in Java (ReadTemps.java) and Python (ReadTemps.py) by running them.

- Test with temps.txt with the values: 30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9
- Test with temps.txt with the values: 30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9 31.0 31.2
- Change the programs to solve the problem without changing the size of the array.

Question P1.3 - Buffer overflow

Review and test the programs written in C RootExploit.c and 0-simple.c.

- Indicate which vulnerability of *Buffer Overflow* exists and what you have to do (and why) to exploit it, and (i) get confirmation that you have been given root/admin permissions, without using the correct *password*, (ii) get the message "YOU WIN !!!".

Experience 1.5 - Formatted I/O

Formatting I/O in C can make all the difference.

Analyze and test the program IOformatado.c and see the difference between the segura() and vulneravel() function.

Note: Remember that in C there are formatting directives, such as %d, %s, ...

Question P1.4 - Read overflow

Analyze and test the program written in C Read Overflow.c.

- What can you conclude?

Experience 1.6 - Heap Buffer overflow

The example of *Heap Buffer overflow* seen in the theoretical class is found in the overflowHeap.1.c and overflowHeap.2.c files.

- Test and change the output of the *readonly* variable.

Experience 1.7 - Stack Buffer overflow

Buffer overflow vulnerabilities occur when it is possible to write and / or execute code in memory areas that normally would not be possible, and is usually derived from the use of unsafe functions.

Most current operating systems and compilers already incorporate some security features that prevent such attacks, such as:

- *canaries* - values placed in the *stack* (by the compiler) between a buffer and control data, in order to monitor *buffer overflows* during program execution. If there is a *buffer overflow*, the first data to be corrupted/changed is the canary, and a failed verification of the canary data is an alert for the existence of a *overflow*;
- *Executable space protection* (XP) / *Data Execution Prevention* (DEP) - prevents some sectors of memory (for example, *stack*) from being executed (i.e., the program can not execute code that is in those sectors of memory);
- *Address Space Layout Randomisation* (ASLR) - technique used to prevent the execution of code injected into the program by randomly placing modules and data structures in memory space.

This experiment and the next questions will require ASLR disabled, so pelase run the following command:

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

The *Stack Buffer overflow* example seen in the theoretical class is found in the overflowStack.1.c and overflowStack.2.c files.

We will use Gnu debugger (GDB) to get the information needed to exploit the vulnerability without changing the program overflowStack.1.c.

To do this, execute the following commands:

- compile overflowStack.1.c with debug information, to be used by GDB;

```
gcc overflowStack.1.c -g -o overflowStack.1
```

- start debugging the program with gdb

```
gdb overflowStack.1
```

- in gdb, put *breakpoint* in the store() function

```
b store
```

- in gdb, run the program until *breakpoint*

```
r
```

- in gdb, get information about the current *frame* of program execution

```
info f
```

- in the last line it is indicated in which memory address is stored the base pointer (%rbp) and the return address (%rip). Since we will want to rewrite the contents of the return address in order to direct it to the debug() function, save the memory address where the %rip (emr) is stored
- the next step is to obtain the memory address where the debug() function is stored. To do this, in gdb perform the command

```
p debug
```

- save the address that is presented to you, since it is this address that you want to rewrite in emr.
- next, to be sure how many bytes you have to write in the buf array to rewrite the %rip, see what address the buf (eia) array starts in.

```
p &buf
```

- The number of bytes to write to the beginning of the emr is emr - eia. Use a hexadecimal calculator to get the result (for example, use http://www.calculator.net/hex-calculator.html)
- you have all the data that you need from gdb, so you can exit gdb with

```
quit
```

- Run the overflowStack.1 program with the required parameter to get the data hidden in the debug().

Question P1.5

Now that you have experience with _ buffer overflow_ (see question P1.3), can you do the same if you need an exact value?

Compile and run the 1-match.c program, and get the "Congratulations" message on the screen. + have you heard of *little-endian* and *big-endian*?

Please indicate the steps you took to exploit this vulnerability.

Experience 1.8

Compile and run the 2-functions.c program, and get the "Congratulations" message on the screen. + remember that the name of a function in C is equivalent to the address where it is written in memory; + may fp be win?

Please indicate the steps you took to exploit the vulnerability.

Experience 1.9

Compile and run the 3-return.c program, and get the "Congratulations" message on the screen.

Please indicate the steps you took to exploit the vulnerability.

Final Note: To re-enable the ASLR that you disabled in Experience 1.7, run the following command:

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

---

2. Integer vulnerability

Save the files in the directory Aula 11.c to your local machine in the directory /home/user/Aulas/Aula11.c.

Experience 2.1

Use the IntegerCheck.java program to check the maximum and minimum values of the various integers on your machine.

- How many bits does each of the integers have?

Experience 2.2

Test the IntegerError.java program and check what happens in *integer overflow* situation.

- Test with other integer types: byte, short, long
- What if you need to use larger integers?

Experience 2.3

Test the IntegerCheck2.java program and enter correct and incorrect values.

- What happens when you read a long and assign it to another type of integer?
- What would happen if I used the scan for the correct type in each case (nextByte (), nextInt (), ...)?

Question P2.1

Review the overflow.c program.

1. What is the vulnerability that exists in the *vulneravel()* function and what are the effects of it?
2. Complete *main()* to demonstrate this vulnerability.
3. When running does it give some error? Which?

Question P2.2

Review the underflow.c program.

1. What is the vulnerability that exists in the *vulneravel()* function and what are the effects of it?
2. Complete *main()* to demonstrate this vulnerability.
3. When running does it give some error? Which?

Experience 2.4

Review the erro_sinal.c program.

1. What is the vulnerability that exists in the *vulneravel()* function and what are the effects of it?
2. Complete *main()* to demonstrate this vulnerability.
3. When running does it give some error? Which?