lab3: Software security

You will be experiencing some conventional ways to *hack* programs using their bugs. These techniques don't work on modern software systems due to the defence mechanisms that we've talked earlier, but experience of conducting these old-fashioned attacks often help you become familiar with what an *attack* look like. Many defence machanisms have been developed to prevent these and modern offensive techniques are built on top of these.

The offensive procedure is very tedious and labor-intensive in general, so we use severl tools that white hacker, or the CTF players have developed.

- pwntool
- peda
- ghidra

pwntool: a python framework for exploit develpment. This lets you to write a python script that runs a program, talks with the program and gives you the *privileged shell* that we want. The goal of each challenge is to write a python script and its description.

`pwntool` is already installed in the VM.

peda: a gdb plugin. In developing your exploit you may want to observe how the target program behaves and `gdb` is the tool that you should use for this. `peda` is a gdb plugin that enriches the gdb environment.

You need to manually install `peda` from the url above, which is as ease as runnig two shell commands.

ghidra is a disassembler and decompiler that helps you to understand the structure of the target program. This is not essential, but could help some of you. My recommendation is to install this locally on you computer (outside the lab VM) and open the challenge binaries. To make this step easy, all challenge binaries are compiled statically.

For each challenge, you will be writing one python script that looks like this.

```
#!/usr/bin/env python3

from pwntool import *

'''
To reproduce the exploit,
1. check if we are the main module
2. if so, print the string.
'''

'''
```

```
Check if we are main
'''
if __name__ == '__main__':
    '''
    print the string
    '''
    print('Hello World')
```

Essentially, your script starts with *she-bang* (`#!/usr/bin/env python3`) followed by a write-up in the form of multi-line comment describing minimal details required for reproducing your exploit script. Afterwards, your exploit should be commented per-line.

For each lab, your goal is to trick the program to give you a shell on bahalf of a diffent user. For each challenge, there is a user (e.g., `chal0`). The goal is to read the flag file for each challenge (e.g., `/challenge/flag0` for `chal0`), which is owned by `chal0:chal0` and not readable by others. Your chance is that each challenge binary is *setgid* binary, owned by `vagrant:chal0` for `chal0`. That is, you can run the program, and the process can read the flag file, if you can trick the process to do it.

Each challenge has pretty trivial bug that allows you to do so.

Challenge 0 (15 points)

This challenge is for you to become familiar with the `pwntool`. Reading the program either using `ghidra` or `objdump`, you will see that you can very easily trick the program to run another shell program. Once the program executes shell, you can just `cat flag`.

Challenge 1 (55 points)

This is the first challenge in which you will be exploiting a real bug: buffer overflow. Reading the program, you will again see the intermediate goal that you should achieve to let the program execute a shell for you.

Challenge 2 (55 points)

This challenge is not as kind as the others. There is a function giving you a shell, but is not being called. How can you call it?

Challenge 3 (55 points)

This one is pretty similar to the challenge 2 although you may not agree. To have an idea, please read the program either using `objdump` of `ghidra`. If you still are lost, write this C program and run it.

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    system("sh");
}
```

Got the idea? To lessen your effort, there is "sh" manually injected. You can make use of this