CSI4109 Assignment #5
Due: June 8th 2:59pm

Spawn A Shell

In this homework assignment you will write memory corruption exploits that spawn a shell. There are five challenges you'll have to tackle, and for each challenge you are given a binary which contains an input buffer overflow vulnerability to exploit.

Challenge 0. Warm-up You are given a binary `warmup` that has an input buffer overflow vulnerability in it. The buffer overflow vulnerability can be triggered when you enter on standard input an input larger than a certain size, as follows.

```
$ ./warmup
Enter name:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

Find an input that exploits the buffer overflow vulnerability present in the program and makes this given program open a shell. The code — `execve("/bin/sh", NULL, NULL)` — which opens a shell is contained in the binary, but this code, unfortunately, is not executed by default (unless a malicious input exploits the vulnerability). Save your input as a file called `warmup.exploit`, which can be used to open a shell as follows.

```
$ (cat warmup.exploit; cat) | ./warmup
Enter name:
ls /
bin  boot  dev  etc  home  init  lib  lib32  lib64 ...
```

What to submit.

- `warmup.exploit`

Challenge 1. Return-Oriented Programming You are given a binary called `exploitme` which has an input buffer overflow vulnerability in it. Unfortunately, the source code that this binary was compiled from does not have the exact statement `execve("/bin/sh", NULL, NULL)` in it. In class, we discussed how an attacker could still open a shell without the exact statement that opens a shell; one could stitch together multiple instructions that end with the `ret` instruction (called *ROP gadgets*) located here and there in the binary.

In this challenge, you will exploit the buffer overflow vulnerability present in the program, and make this given program open a shell using *Return-Oriented Programming (ROP)*. You are given a Python script `default_exploit.py`, which has two functions in it: `get_id` and `get_password`. You need to rewrite the body of these two functions such that they return inputs that successfully exploit the binary. These functions will be imported and called by `exploit.py`, which will be invoked as follows.

```
$ ./exploit.py --target exploitme
...
...
Enter id:
... (the value returned by get_id is shown here)
Enter password:
... (the value returned by get_password is shown here)
$ ls /
bin   dev  home  lib    lib64   media  opt   root  sbin  srv  tmp  var
boot  etc  init  lib32  libx32  mnt    proc  run   snap  sys  usr
$ touch exploitme.exploited
$ exit
```

What to submit.

- `default_exploit.py`

Challenge 2. Bypassing SafeStack In recognition of the risks posed by potential stack overflow exploits that overwrite return addresses, the defender decided to harden the program against such exploits by using the *Safe-Stack* protection (enabled via the compiler flag `-fsanitize=safe-stack`) discussed in class. The hardened binary is called `exploitme-safestack`, and it is much more difficult to corrupt return addresses using stack overflow vulnerabilities. In class, we discussed code pointers other than return addresses that can be corrupted.

In this challenge, you will exploit the buffer overflow vulnerability present in the program, and make this given program open a shell using *Call-Oriented Programming (COP)*. (Hint: Look for code pointers located on *Unsafe* stack.) For this challenge, you are given a Python script `safestack_exploit.py`, which has the same two functions defined. You need to rewrite the body of these two functions such that they return inputs that successfully exploit the binary. These functions will be used by `exploit.py` to open a shell as follows.

```
$ ./exploit.py --target exploitme-safestack
...
...
Enter id:
... (the value returned by get_id is shown here)
Enter password:
... (the value returned by get_password is shown here)
$ ls /
bin   dev  home  lib    lib64   media  opt   root  sbin  srv  tmp  var
boot  etc  init  lib32  libx32  mnt    proc  run   snap  sys  usr
```

```
$ touch exploitme-safestack.exploited
$ exit
```

What to submit.

- `safestack_exploit.py`

Challenge 3. Bypassing Control-Flow Integrity The defender decided to *further harden* (on top of Safe-Stack) the binary against COP exploits by employing indirect function call checking (enabled via the compiler flag `-fsanitize=cfi-icall`), a form of *Control-Flow Integrity (CFI)* discussed in class. This means that the functions that you can call in your COP exploit is limited to the set of functions that have the same type as the type of function pointer. This further hardened binary is called `exploitme-safestack-cfi`.

In this challenge, you will exploit the buffer overflow vulnerability present in the program, and make this given program open a shell bypassing Safe-Stack and Control-Flow Integrity. For this challenge, you are given a Python script `cfi_exploit.py`, which has the same two functions defined. You need to rewrite the body of these two functions such that they return inputs that successfully exploit the binary. These functions will be used by `exploit.py` to open a shell as follows.

```
$ ./exploit.py --target exploitme-safestack-cfi
...
...
Enter id:
... (the value returned by get_id is shown here)
Enter password:
... (the value returned by get_password is shown here)
$ ls /
bin   dev  home  lib    lib64   media  opt   root  sbin  srv  tmp  var
boot  etc  init  lib32  libx32  mnt    proc  run   snap  sys  usr
$ touch exploitme-safestack-cfi.exploited
$ exit
```

What to submit.

- `cfi_exploit.py`

Challenge 4. Bypassing ASLR The defender realized that the binary could still be exploited even with the presence of the Safe-Stack and the Control-Flow Integrity protection. To further raise the bar for attackers, the defender decided to enable *Address Space Layout Randomization (ASLR)* on top of the Safe-Stack and Control-Flow Integrity protection. This means that the function addresses are going to be different every time the binary is run; in other words, you can no longer hard-code function addresses in your exploit. This new hardened binary is called `exploitme-safestack-cfi-aslr`.

In this challenge, you will exploit the buffer overflow vulnerability present in the program, and make this given program open a shell bypassing Safe-Stack, Control-Flow Integrity, and ASLR. (Hint: Look for code pointer leak vulnerabilities.) For this challenge, you are given a Python script `aslr_exploit.py`, which has the same two functions defined. You need to rewrite the body of these two functions such that they return inputs that successfully exploit the binary. These functions will be used by `exploit.py` to open a shell as follows.

```
$ ./exploit.py --target exploitme-safestack-cfi-aslr
...
...
Enter id:
... (the value returned by get_id is shown here)
Enter password:
... (the value returned by get_password is shown here)
$ ls /
bin   dev  home  lib    lib64   media  opt   root  sbin  srv  tmp  var
boot  etc  init  lib32  libx32  mnt    proc  run   snap  sys  usr
$ touch exploitme-safestack-cfi-aslr.exploited
$ exit
```

What to submit.

- `aslr_exploit.py`

Submission Instructions

Submit on LearnUs (`ys.learnus.org`) your input (i.e., exploit payload) to each challenge:

- `warmup.exploit`
- `default_exploit.py`
- `safestack_exploit.py`
- `cfi_exploit.py`
- `aslr_exploit.py`

You must also submit `README`, which is in **plain text** and contains your name, student ID, and a description of how each exploit works.

Grading Rubric

Your exploit will be tested on Ubuntu 22.04 64-bit.

- If you successfully exploit `warmup`. (1 pt)
- If you successfully exploit `exploitme`. (2 pts)
- If you successfully exploit `exploitme-safestack`. (2 pts)
- If you successfully exploit `exploitme-safestack-cfi`. (2 pts)
- If you successfully exploit `exploitme-safestack-cfi-aslr`. (3 pts)