Assignment 3
Buffer Overflow Attacks

Basic Setup

Exploiting buffer overflows requires **precise** control over the execution environment. A small change in the compiler, environment variables, or the way the program is executed can result in slightly different **memory layout** and **code structure**, thus requiring a different exploit. For this reason, this assignment uses a Virtual Machine (VM) to run the **vulnerable** web server code.

To start working on this assignment, you'll need **Software (Hypervisor)** that lets you run a virtual machine. For Linux users, we recommend running the **VM** over Kernel-based Virtual Machine (KVM), which is built into the Linux kernel. KVM should be available through your Ubuntu distribution, try `sudo apt-get install qemu-kvm`. KVM requires **hardware virtualization** support in your CPU, and you must enable this support in your BIOS (which is often, but not always, the default). If you have another virtual machine monitor installed on your machine (e.g., VMware), that virtual machine monitor may grab the hardware virtualization support exclusively and **prevent** KVM from working.

On Linux without KVM, use VMware Workstation from IS&T or, Oracle VirtualBox.

If you are using a computer with a **non-x86** CPU (such as a Mac with an **ARM M1 or M2** processor), running the VM locally on your computer may be **prohibitively slow**, because your VM will have to **emulate** x86 instructions instead of running them natively. Hence, it is strictly recommended for everyone to rent a Ubuntu machine from Baadal VM for this assignment, or perform a **local** setup on a x86 machine (preferrably **Ubuntu 22.04** or higher).

Setup IITD Proxy to make all private IP addresses accessible to you. Once you have hypervisor installed on your machine, you should download the VM image using the following commands.

```
wget --no-check-certificate  http://10.237.27.193/2402-SIL765.sh
wget --no-check-certificate  http://10.237.27.193/2402-SIL765.vmdk
```

This virtual machine contains an installation of **Ubuntu 22.04** Linux.

To start the VM using **VMware**, import `2402-SIL765.vmdk`. Go to File > New select "create a custom virtual machine", choose Linux > Debian 9.x 64-bit, choose Legacy BIOS, and use an existing virtual disk (and select the `2402-SIL765.vmdk` file, choosing the "Take this disk away" option). Finally, click Finish to complete the setup.

To start the VM with **KVM**, first make the shell script executable using the command `sudo chmod a+x 2402-SIL765.sh` and then run `sudo ./2402-SIL765.sh` from a terminal (`Ctrl+A x` to force quit). To prevent writing **sudo** while executing the script, try adding yourself to the **kvm group**.

You will use the **student** account in the VM for your work. The password for the student account is **student**. You can also get access to the **root** account in the VM using **sudo**; for example, you can install new software packages using `sudo apt-get install <pkgname>`.

You can either log into the virtual machine using its **console**, or use Secure Shell (SSH) to log into the virtual machine over the (virtual) network. The latter also lets you easily copy files into and out of the virtual machine with Secure Copy Protocol (SCP) or Remote Sync (rsync). How you access the virtual machine over the network depends on how you're running it. If you're using **VMWare**, you'll first have to find the virtual machine's **IP address**. To do so, log in on the console, run `ip addr show dev eth0`, and note the IP address listed beside **inet**.

With **KVM**, you can use **localhost** as the IP address for **SSH** and **HTTP**. You can now log in with ssh by running the following command from your host machine: `ssh -p 2222 student@<IPADDRESS>`.

For security reasons, SSH **does not** allow logging in over the network using a password (and, in this specific case, the password is known to everyone). To log in via SSH, you will need to set up an SSH Key.

You may also find it helpful to create a host alias for your VM in your **~/.ssh/config file**, so that you can simply run, for example, `ssh lab` or `scp -r ./lab/ lab:~`. To do this, add the following lines to your **~/.ssh/config file**:

```
Host lab
Hostname localhost
User student
Port 2222
```

Getting started

The files you will need for this lab are distributed using the Git version control system. You can also use Git to keep track of any changes you make to the initial source code. Here's an overview of Git and the Git user's manual, which you may find useful.

The course Git repository is available here. To get the lab code, log into the VM using the student account (by running `./2402-SIL765.sh` in a terminal and running the following commands in separate terminal). Setup IITD Proxy and GitHub Proxy in your remote machine and clone the source code for this assignment as follows:

```
student@65660-v23:~$ git clone https://github.com/prateekbhaisora/SIL765-
    Assignment-3.git lab
Cloning into 'lab'...
student@65660-v23:~$ cd lab
student@65660-v23:~/lab$
```

It's important that you clone the repository into the lab directory, because the length of pathnames will matter in this assignment. Alternatively, you can clone the repo in the **host** machine and scp the file/folder contents to **remote** machine, as mentioned above.

Before you proceed with this assignment, remove the old binaries (if any) and make sure you can compile the **zookws** web server:

```
student@65660-v23:~/lab$ sudo rm zookd zookd-exstack zookd-nxstack
student@65660-v23:~/lab$ make
cc zookd.c -c -o zookd.o -m64 -g -std=c99 -Wall -Wno-format-overflow -
    D_GNU_SOURCE -static -fno-stack-protector
cc http.c -c -o http.o -m64 -g -std=c99 -Wall -Wno-format-overflow -
    D_GNU_SOURCE -static -fno-stack-protector
cc -m64  zookd.o http.o   -o zookd
cc -m64 zookd.o http.o   -o zookd-exstack -z execstack
cc -m64 zookd.o http.o   -o zookd-nxstack
cc zookd.c -c -o zookd-withssp.o -m64 -g -std=c99 -Wall -Wno-format-
    overflow -D_GNU_SOURCE -static
cc http.c -c -o http-withssp.o -m64 -g -std=c99 -Wall -Wno-format-
    overflow -D_GNU_SOURCE -static
cc -m64  zookd-withssp.o http-withssp.o   -o zookd-withssp
cc -m64   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
cc run-shellcode.c -c -o run-shellcode.o -m64 -g -std=c99 -Wall -Wno-
    format-overflow -D_GNU_SOURCE -static -fno-stack-protector
cc -m64  run-shellcode.o   -o run-shellcode
student@65660-v23:~/lab$
```

The component of **zookws** that receives HTTP requests is **zookd**. It is written in **C** and serves static files and executes dynamic scripts. For this assignment, you don't have to understand the dynamic scripts; they are written in Python and the exploits in this assignment apply only to C code. The HTTP-related code is in **http.c**. Here is a tutorial about the HTTP protocol.

There are two versions of zookd you will be using:

- zookd-exstack
- zookd-nxstack

zookd-exstack has an **executable** stack, which makes it easy to inject executable code given a **stack buffer overflow vulnerability**. **zookd-nxstack** has a **non-executable stack**, and requires a more sophisticated attack to exploit stack buffer overflows. In order to run the web server in a **predictable** fashion — so that its stack and memory layout is the **same** every time — you will use the clean-env.sh script. This is the same way in which we will run the web server during grading, so make sure all of your exploits work on this configuration!

The reference binaries of **zookd** are provided in `bin.tar.gz`, which we will use for grading. Make sure your exploits work on those binaries. The `sudo sudo make check-lab1` command will always use both clean-env.sh and `bin.tar.gz` to check your submission.
Now, make sure you can run the **zookws** web server and access the **zoobar** web application from a browser running on your machine, as follows:

```
student@65660-v23:~/lab$ ./clean-env.sh ./zookd 8080
```

The clean-env.sh commands starts **zookd** on port **8080**. To open the zoobar application, open your browser and point it at the URL `http://<IPADDRESS>:8080/`, where <IPADDRESS> is the VM's IP address we found above.

Part 1: Finding buffer overflows

In the first part of this assignment, you will find buffer overflows in the provided web server. To do this assignment, you will need to understand the **basics** of buffer overflows. To help you get started with this, you should read Smashing the Stack in the 21st Century, which goes through the details of how buffer overflows work, and how they can be exploited.

Exercise 1

Study the web server's C code (in **zookd.c** and **http.c**), and find one example of code that allows an attacker to overwrite the **return address** of a function. *Hint:* Look for buffers allocated on the stack.

For your vulnerability, try to identify the buffer which may overflow, how you would structure the input to the web server (i.e., the **HTTP request**) to overflow the buffer and overwrite the return address, and the call stack that will trigger the buffer overflow (i.e., the chain of function calls starting from **process_client**).

It is worth taking your time on this exercise and familiarizing yourself with the code, because your next job is to exploit the vulnerability you just identified. In fact, you may want to go back and forth between this exercise and later exercises, as you work out the details and document them. That is, if you find a buffer overflow that you think can be exploited, you can use later exercises to figure out if it, indeed, can be exploited. It will be helpful to draw a stack diagram (on paper) like the figures in Smashing the Stack in the 21st Century.

Now, you will start developing exploits to take advantage of the buffer overflows you have found above. We have provided template Python code for an exploit in **/home/student/lab/exploit-template.py**, which issues an HTTP request. The exploit template takes **two** arguments, the **server name** and **port number**, so you might run it as follows to issue a request to **zookws** running on **localhost**:

```
student@65660-v23:~/lab$ ./clean-env.sh ./zookd-exstack 8080 &
[1] 2676
student@65660-v23:~/lab$ ./exploit-template.py localhost 8080
HTTP request:
b'GET / HTTP/1.0\r\n\r\n'
...
student@65660-v23:~/lab$
```

You are **free** to use this template, or write your **own** exploit code from scratch. Note, however, that if you choose to write your own exploit, the exploit **must** run correctly inside the provided virtual machine.

You may find GNU Debugger (gdb) useful in building your exploits. As **zookd forks** off many processes (one for each client), it can be difficult to debug the correct one. The easiest way to do this is to run the web server ahead of time with `clean-env.sh` and then attaching **gdb** to an already-running process with the **-p** flag. You can find the **PID** of a process by using **pgrep**; for example, to attach to **zookd-exstack**, start the server and, in another shell, run:

```
student@65660-v23:~/lab$ gdb -p $(pgrep zookd-)
...
(gdb) break your-breakpoint
Breakpoint 1 at 0x1234567: file zookd.c, line 999.
(gdb) continue
Continuing.
```

Keep in mind that a process being debugged by gdb will **not** get killed even if you terminate the parent **zookd** process using **CTRL + C**. If you are having trouble restarting the web server, check for **leftover** processes from the previous run, or be sure to exit **gdb** before restarting **zookd**. You can also save yourself some typing by using **b** instead of break, and **c** instead of continue.

When a process being debugged by **gdb forks**, by default **gdb** continues to debug the parent process and **does not** attach to the child. Since **zookd** forks a child process to service **each** request, you may find it helpful to have **gdb** attach to the child on **fork**, using the command `set follow-fork-mode child`. By default, **gdb** will detach from the parent, but it can be convenient to have it remain attached, so you can repeatedly try exploits without **quitting** and **re-starting** (and setting up again) the debugger. This is done using the command `set detach-on-fork off`. See the reference on inferiors (the name GDB uses for processes) for how to switch between parent and child. When the child exists, you can use **inferior 1** to switch back to the parent. We have added the two settings to **home/student/lab/.gdbinit**, which will take effect if you start gdb in **that** directory.

As you develop your exploit, you may discover that it causes the server to **hang** as opposed to **crash**, depending on what buffer overflow you are trying to take advantage of and what data you are **overwriting** in the running server. You can **dig** into the details of why the hang happens, to understand how you are affecting the server's execution, in order to make your exploit **avoid** the hang and instead crash the server. Or you can choose to exploit a different buffer overflow that avoids the hanging behavior.

For this and subsequent exercises, you may need to encode your attack payload in different ways, depending on which vulnerability you are exploiting. In some cases, you may need to make sure that your attack payload is **URL-encoded**; that is, use + instead of space and % 2b instead of +. Here is a URL encoding reference and a handy conversion tool. You can also use quoting functions in the python **urllib** module to URL-encode bytes (in particular, urllib.parse.quote_from_bytes, followed by **.encode('ascii')** to get the bytes from the string). In other cases, you may need to include binary values into your payload. The Python struct module can help you do that. For example, **struct.pack("<Q", x)** will produce an 8-byte (64-bit) binary encoding of the integer **x**.

Exercise 2

Write an exploit that uses a buffer overflow to **crash** the web server (or **one** of the processes it creates). You do not need to **inject** code at this point. Verify that your exploit crashes the server by checking the last few lines of `sudo dmesg | tail`, using **gdb**, or **observing** that the web server crashes (i.e., it will print Child process 9999 terminated incorrectly, receiving signal 11).

Provide the code for the exploit in a file called **exploit-2.py**.

The vulnerability you found in **Exercise 1** may be too hard to exploit. Feel **free** to find and exploit a **different** vulnerability.

You can check whether your exploits crash the server as follows:

```
student@65660-v23:~/lab$ sudo make check-crash
```

Part 2: Code injection

In this part, you will use your buffer overflow exploits to **inject** code into the web server. The goal of the injected code will be to **unlink (remove)** a sensitive file on the server, namely **/home/student/grades.txt**. Use **zookd-exstack**, since it has an **executable stack** that makes it easier to inject code. The **zookws** web server should be started as follows.

```
student@65660-v23:~/lab$ ./clean-env.sh ./zookd-exstack 8080
```

You can build the exploit in two steps. First, write the **shell code** that unlinks the sensitive file, namely **/home/student/grades.txt**. Second, embed the compiled shell code in an HTTP request that triggers the buffer overflow in the web **server.a**.

When writing shell code, it is often easier to use **assembly** language rather than higher-level languages, such as **C**. This is because the exploit usually needs fine control over the **stack layout, register values** and **code size**. The **C compiler** will generate additional **function preludes** and perform various **optimizations**, which makes the compiled binary code **unpredictable**.

We have provided **shell code** for you to use in **/home/student/lab/shellcode.S**, along with Makefile rules that produce **/home/student/lab/shellcode.bin**, a compiled version of the shell code, when you run `make`. The provided shell code is intended to exploit **setuid-root binaries**, and thus it runs a shell. You will need to modify this shell code to instead unlink **/home/student/grades.txt**.

To help you **develop** your shell code for this exercise, we have provided a program called **run-shellcode** that will run your binary shell code, as if you correctly jumped to its starting point. For example, running it on the provided shell code will cause the program to **execve("/bin/sh")**, thereby giving you another shell prompt:

```
student@65660-v23:~/lab$ ./run-shellcode shellcode.bin
```

Exercise 3 (warm-up)

Modify shellcode.S to unlink **/home/student/grades.txt**. Your assembly code can either invoke the **SYS_unlink** system call, or call the **unlink()** library function.

To test whether the shell code does its job, run the following commands:

```
student@65660-v23:~/lab$ make
student@65660-v23:~/lab$ touch ~/grades.txt
student@65660-v23:~/lab$ ./run-shellcode shellcode.bin
# Make sure /home/student/grades.txt is gone
student@65660-v23:~/lab$ ls ~/grades.txt
ls: cannot access /home/student/grades.txt: No such file or directory
```

You may find strace useful when trying to figure out what system calls your shellcode is making. Much like with **gdb**, you **attach** strace to a running program:

```
student@65660-v23:~/lab$ strace -f -p $(pgrep zookd-)
```

It will then print **all** of the system calls that program makes. If your shell code isn't working, try looking for the system call you think your shell code **should** be executing (i.e., unlink), and see whether it has the **right** arguments.

Next, we construct a **malicious** HTTP request that injects the **compiled** byte code to the web server, and **hijack** the server's control flow to run the injected code. When developing an exploit, you will have to think about **what** values are on the stack, so that you can modify them accordingly.

When you're constructing an exploit, you will often need to know the addresses of **specific** stack locations, or **specific** functions, in a particular program. One way to do this is to add **printf()** statements to the function in question. For example, you can use `printf("Pointer:`

%p \n", &x); to print the address of variable **x** or function **x**. However, this approach requires some care: you need to make sure that your added statements are not themselves changing the stack layout or code layout. We (and sudo make check-lab1) will be grading the lab without any **printf** statements you may have added.

A more fool-proof approach to determine addresses is to use **gdb**. For example, suppose you want to know the stack address of the pn[] array in the **http_serve** function in **zookd-exstack**, and the address of its saved return pointer. You can obtain them using **gdb** by first starting the web server (remember **clean-env!**), and then attaching **gdb** to it:

```
student@65660-v23:~/lab$ gdb -p $(pgrep zookd-)
...
(gdb) break http_serve
Breakpoint 1 at 0x5555555561c4: file http.c, line 275.
(gdb) continue
Continuing.
```

Be sure to run **gdb** from the **~/lab** directory, so that it picks up the set follow-fork-mode child command from **~/lab/.gdbinit**. Now you can issue an HTTP request to the web server, so that it triggers the breakpoint, and so that you can examine the stack of **http_serve**.

```
student@65660-v23:~/lab$ curl localhost:8080
```

This will cause **gdb** to hit the breakpoint you set and **halt** execution, and give you an opportunity to ask **gdb** for addresses you are interested in:

```
Thread 2.1 "zookd-exstack" hit Breakpoint 1, http_serve (fd=4, name=0
    x55555575fcec "/") at http.c:275
275           void (*handler)(int, const char *) = http_serve_none;
(gdb) print &pn
$1 = (char (*)[2048]) 0x7fffffffd4a0
(gdb) info frame
Stack level 0, frame at 0x7fffffffdcd0:
 rip = 0x5555555561c4 in http_serve (http.c:275); saved rip = 0
     x55555555587b
 called by frame at 0x7fffffffed00
 source language c.
 Arglist at 0x7fffffffdcc0, args: fd=4, name=0x55555575fcec "/"
 Locals at 0x7fffffffdcc0, Previous frame's sp is 0x7fffffffdcd0
 Saved registers:
  rbx at 0x7fffffffdcb8, rbp at 0x7fffffffdcc0, rip at 0x7fffffffdcc8
(gdb)
```

From this, you can tell that, at least for this invocation of **http_serve**, the pn[] buffer on the stack lives at address **0x7fffffffd4a0**, and the saved value of %rip (the **return address** in other words) is at **0x7fffffffdcc8**. If you want to see register contents, you can also use info registers.

Now it's your turn to develop an exploit.

```
student@65660-v23:~/lab$ ./run-shellcode shellcode.bin
```

Exercise 4

Starting from one of your exploits from **Exercise 2**, construct an exploit that hijacks the control flow of the web server and **unlinks /home/student/grades.txt**. Save this exploit in a file called **exploit-4.py**.

Verify that your exploit works; you will need to **re-create /home/student/grades.txt** after each successful exploit run.

To do this exercise, first focus on obtaining **control** of the program counter. Draw (on paper) the **stack layout** that you expect the program to have at the point when you overflow the buffer, and use **gdb** to verify that your **overflow** data ends up where you expect it to. Step through the **execution** of the function to the return instruction to make sure you can control what address the program **returns** to. The **next**, **stepi**, and **x** commands in gdb should prove helpful.

Once you can reliably **hijack** the control flow of the program, find a **suitable** address that will contain the code you want to execute, and focus on placing the **correct** code at that address — e.g. a derivative of the provided shell code.

You can check whether your exploit works as follows:

```
student@65660-v23:~/lab$ sudo make check-exstack
```

The test either prints **“PASS”** or **“FAIL”**. We will grade your exploits in this way. Do **not** change the Makefile.

The standard C compiler used on Linux, GNU Compiler Collection (GCC), implements a version of **stack canaries** (called **SSP**). You can explore whether GCC’s version of stack canaries would or would **not prevent** a given vulnerability by using the SSP-enabled versions of **zookd: zookd-withssp**.

Part 3: Return-to-libc attacks

Many modern operating systems **mark** the stack **non-executable** in an attempt to make it **more difficult** to exploit buffer overflows. In this part, you will explore how this protection mechanism can be **circumvented**. Run the web server configured with binaries that have a **non-executable stack**, as follows:

```
student@65660-v23:~/lab$ ./clean-env.sh ./zookd-nxstack 8080
```

The **key** observation to exploiting buffer overflows with a **non-executable stack** is that you still control the **program counter**, after a **ret** instruction jumps to an address that you placed on the stack. Even though you cannot jump to the address of the overflowed buffer (it will not be **executable**), there's usually **enough** code in the vulnerable server's address space to perform the operation you want.

Thus, to **bypass** a **non-executable stack**, you need to first **find** the code you want to execute. This is often a **function** in the standard library, called **libc**, such as **execve**, **system**, or **unlink**. Then, you need to arrange for the stack and registers to be in a state **consistent** with calling that function with the desired arguments. Finally, you need to arrange for the **ret** instruction to jump to the function you found in the first step. This attack is often called a **return-to-libc** attack.

One **challenge** with **return-to-libc** attacks is that you need to pass arguments to the **libc function** that you want to invoke. The x86-64 calling conventions make this a challenge because the first 6 arguments are passed in registers. For example, the first argument must be in register %rdi (see `man 2 syscall`, which documents the calling convention). So, you need an instruction that loads the first argument into %rdi. In **Exercise 3**, you could have put that instruction in the buffer that your exploit overflows. But, in this part of the lab, the stack is **marked non-executable**, so executing the instruction would **crash** the server, but wouldn't execute the instruction.

The solution to this problem is to **find** a piece of code in the **server** that loads an address into %rdi. Such a piece of code is referred to as a **"borrowed code chunk"**, or more generally as a rop gadget, because it is a tool for **return-oriented programming (rop)**. Luckily, **zookd.c** accidentally has a useful gadget: see the function **accidentally()**.

Exercise 5

Starting from your exploit in **Exercises 2 and 4**, construct an exploit that **unlinks /home/student/grades.txt** when run on the binaries that have a **non-executable stack**. Name this new exploit **exploit-5.py**.

In this attack you are going to take **control** of the server over the network **without** injecting any code into the server. You should use a **return-to-libc** attack where you **redirect** control flow to code that already existed before your attack. The outline of the attack is to perform a buffer overflow that:

1. causes the argument to the chosen libc function to be on stack
2. then causes accidentally to run so that argument ends up in %rdi
3. and then causes accidentally to return to the chosen libc function

To do this exercise, it is essential for you to draw a stack diagram like the figures in Smashing the Stack in the 21st Century at (1) the point that the buffer overflows and (2) at the point that accidentally runs.

You can test your exploits as follows:

```
student@65660-v23:~/lab$ sudo make check-libc
```

Exercise 6

Challenge! The **accidentally** function is a bit artificial. Figure out how to perform the return-to-libc attack without relying on that function (delete it and find another way to make your exploit work). Provide your attack in **exploit-challenge.py**. Briefly explain the attack and provide **ROP gadgets** you use in a comment in **exploit-challenge.py**.

You will need to find another chunk of code to reuse that gives you control over %rdi. You can read through the **disassembly** (e.g. using **objdump**) to look for useful **ROP gadgets**.

Because of the nature of **x86/x86-64**, you can use another technique to find sequences of instructions that don't even appear in the disassembly! Instructions are variable-length (from 1 to 15 bytes), and by causing a misaligned parse (by jumping into the middle of an intended instruction), you can cause a sequence of machine code to be misinterpreted. For example, the instruction sequence pop %r15; **ret** corresponds to the machine code 41 5F C3. But instead of executing from the start of this instruction stream, if you jump 1 byte in, the machine code 5F C3 corresponds to the assembly pop %rdi; **ret**.

Automated tools such as **ROPgadget.py** can assist you in searching for **ROP gadgets**, even finding gadgets that arise from misaligned parses. The `2402-SIL765` VM already has **ROP gadgets** installed.

You may find it useful to search for **ROP gadgets** not just in the **zookd** binary but in other libraries that zookd loads at runtime. To see these libraries, and the addresses at which they are loaded, you can run (`ulimit -s unlimited && setarch -R ldd zookd-nxstack`). The **ulimit** and **setarch** commands set up the same environment used by **clean-env.sh**, so that **ldd** prints the same addresses that will be used at runtime.

Part 4: Fixing buffer overflows and other bugs

Finally, you will fix the vulnerabilities that you have exploited in this lab assignment.

Exercise 7

For each buffer overflow vulnerability you have exploited in Exercises **2, 4, and 5**, **fix** the web server's code to prevent the vulnerability in the first place. **Do not** rely on compile-time or runtime mechanisms such as **stack canaries**, removing --fno-stack-protector, **baggy bounds checking**, etc.

Make sure that your code **actually stops** your exploits from working. Use `sudo make check-fixed` to run your exploits against your **modified** source code (as opposed to the provided reference binaries from `bin.tar.gz`). These checks should report **FAIL** (i.e., exploit no longer works). If they report **PASS**, this means the exploit **still** works, and you **did not** correctly fix the vulnerability.

Note that your submission should not make changes to the **Makefile** and other **grading scripts**. We will use our **unmodified version** during grading, anyways.

You should also make sure your code still passes all tests using `sudo make check-lab1`, which uses the unmodified lab binaries.

You are done! Submit your answers to the lab assignment by running `make handin.zip` and upload the resulting **handin.zip** file to the Gradescope submission web site. Also, document your process, with relevant screenshots, in a file named **README.pdf**.