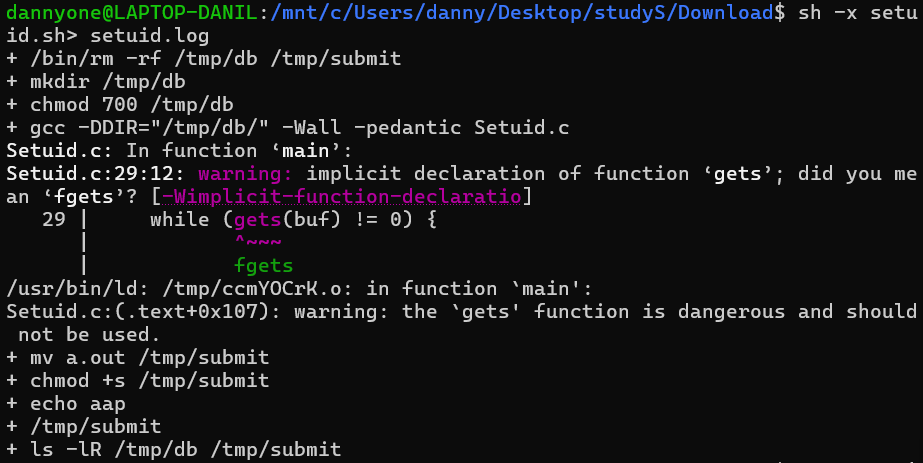
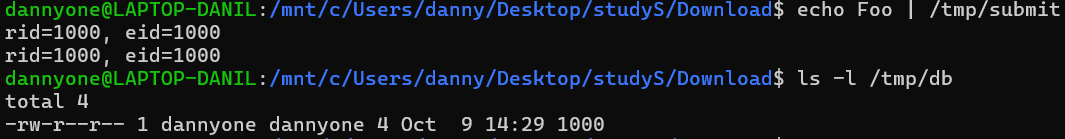
**Assignment: setiud.sh**

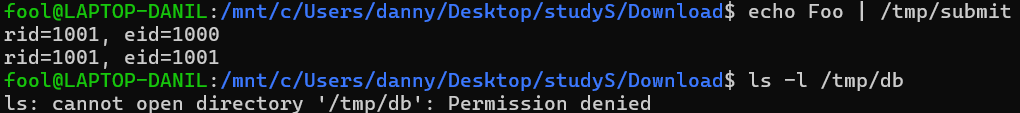
After running *setuid.sh*, I got this output:



After running given commands on my main account, I got the following result:



Then I created a new account, called *fool*, and run those commands again, now I got this output:



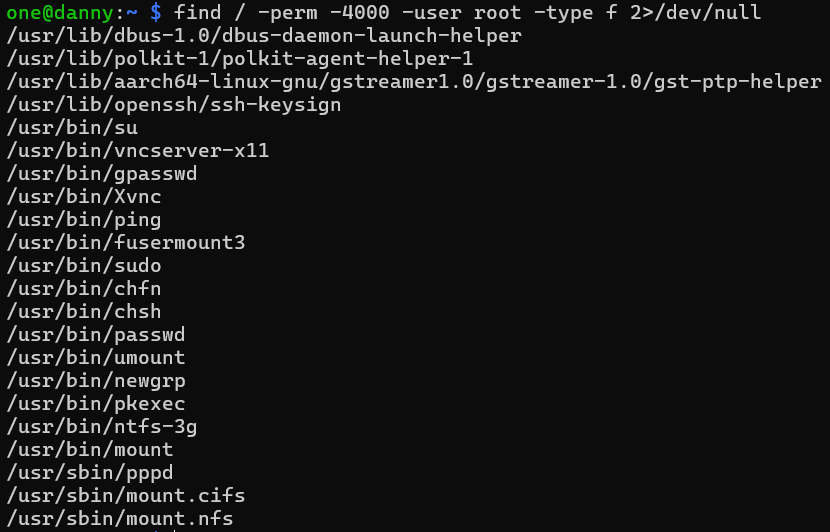
Both accounts could execute */tmp/submit*, but the output differs, since these accounts have difference Id’s.

Since my main account (**dannyone**) has read access to */tmp/db*, it can list the contents. The file is likely associated with my user ID, on the other hand The **fool** account doesn’t have permission to read */tmp/db*, indicating that this directory may have restricted permissions.

**Assignment: setuid programs**

To get the number of programs I executed the following command: ***find / -perm -4000 -user root -type f 2>/dev/null***

I got the following output:



Programs with setuid root let any user run them with root privileges, that’s very risky because if one of this program has a vulnerability, it can let attackers gain root access. Moreover, user himself can misuse some of these programs to become root, giving access to system files or sensitive data. For this reason, It's best to keep these programs to a minimum and ensure they're essential and secure.

**Assignment: setuid on vfat**

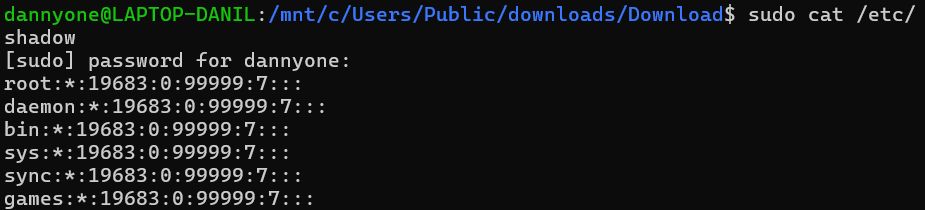
I could easily copy the binary (following the instuctions below), however the attempt to setuid root fails. It happens due to the limitations of the VFAT file system and security considerations. VFAT does not support Unix file permissions, including setuid, moreover, If setuid worked, anyone could run the binary with higher privileges, which could be dangerous.

Src: https://mangohost.net/blog/mounting-usb-drives-in-linux/

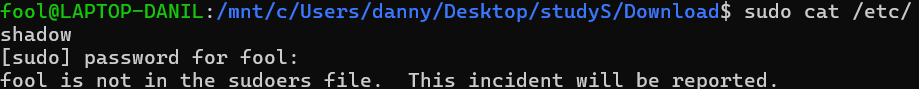
**Assignment: Shadow**

The */etc/shadow* file is restricted because it contains sensitive hashed passwords for all system users. Only root and members of the shadow group can read it to prevent unauthorized access. Allowing normal users to read this file would expose the system to serious risks, including data theft, unauthorized access, and potential full control by malicious users.

For this reason if I use my main account I get the access:

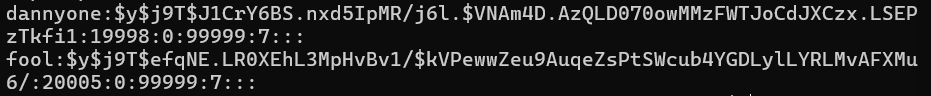


While newly created account cannot see this information



**Assignment: Hash algorithm**

I have the following hashes:

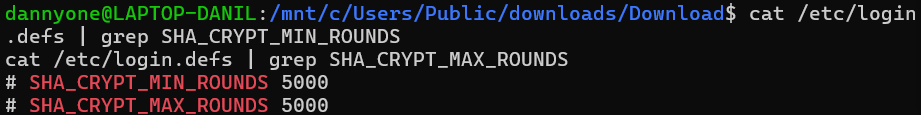


The presence of *$y$* indicates the use of the Yubikey password hashing scheme. To get the number of itterations I used the following commands:

*cat /etc/login.defs | grep SHA\_CRYPT\_MIN\_ROUNDS*

*cat /etc/login.defs | grep SHA\_CRYPT\_MAX\_ROUNDS*

I got the folowing result, so the number of itteration in my case is 5000



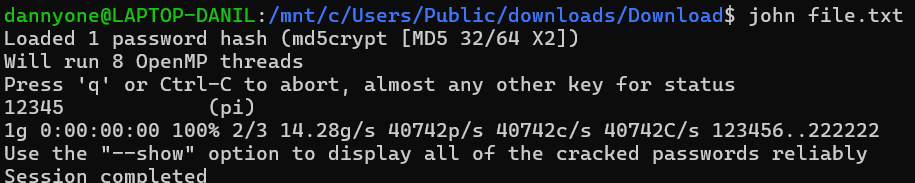
**Assignment: salts**

Salts play a crucial role in enhancing password security by making password cracking significantly harder. It adds a random value to user’s password, which is unique for this user. In this way even if two userss have the same password, their hashes will be different. Moreover, it makes life of the Attackers harder, since if he tries to brute force it, it will take much more computational effort to crack passwords, especially if the number of possible passwords is large. And since *salts* only change the length of the password, which is hashed, it doesn’t affect the number of itterations, so The hash algorithm needs to be invoked the same number of times for both salted and unsalted passwords, and it is based on the iteration count specified.

**Assignment: john**

To crack the password, I had to install john using the following command: *sudo apt install john*

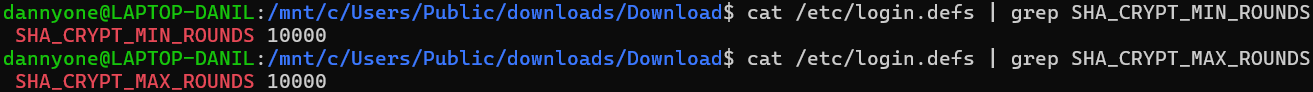
After than, I have tried to pass *passwd-old.txt* to john, but this folder was not seen, so I manualy created *file.txt* and copied there all the data from *passwd-old.txt* and then used john with the new file. This time the passowrd was cracked and I got the following output, so the password is 12345



**Assignment: Iteration count**

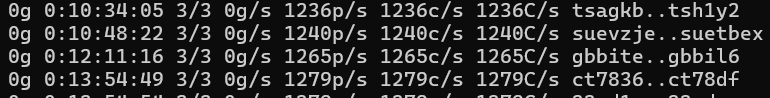
I used the following command to get access to the file: *sudo nano /etc/login.defs* In this file I found

grep SHA\_CRYPT\_MIN\_ROUNDS and grep SHA\_CRYPT\_MAX\_ROUNDS and replaced their values with 10 000. In this way I doubled the iteration count and therefore increased the security of my system:



**Assignment: Jumbo**

I installed jumbo following the guide below, and then tried to run it on *passwd-new.txt* I was wainting for 14 hours and then I understood that it will take forever if I try to crack this code in this way.

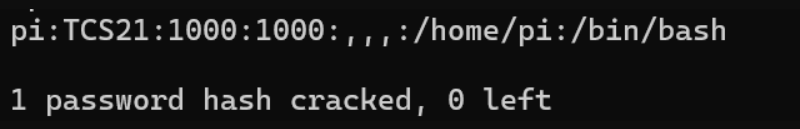


So I had to use mask, since the code had length of 5, I used the following command*:*

*john --mask= ?a?a?a?a?a passwd-new.txt*

The ***?a***means any character (lowercase, uppercase, digits, and special characters)

After that, I had to wait for another 2 hours, however this time the password was succesfully cracked and I found that the password was: **TCS21**

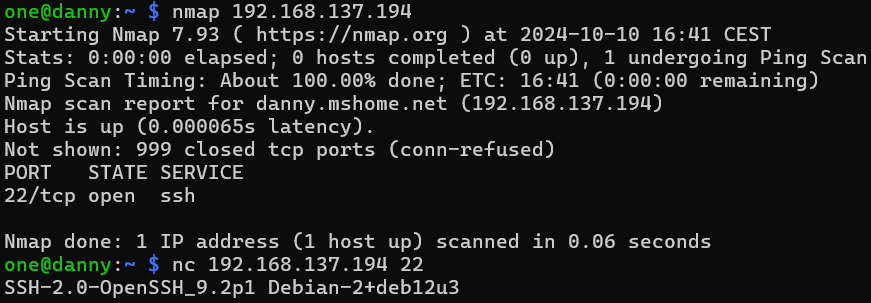


Src: https://github.com/openwall/john/blob/bleeding-jumbo/doc/INSTALL

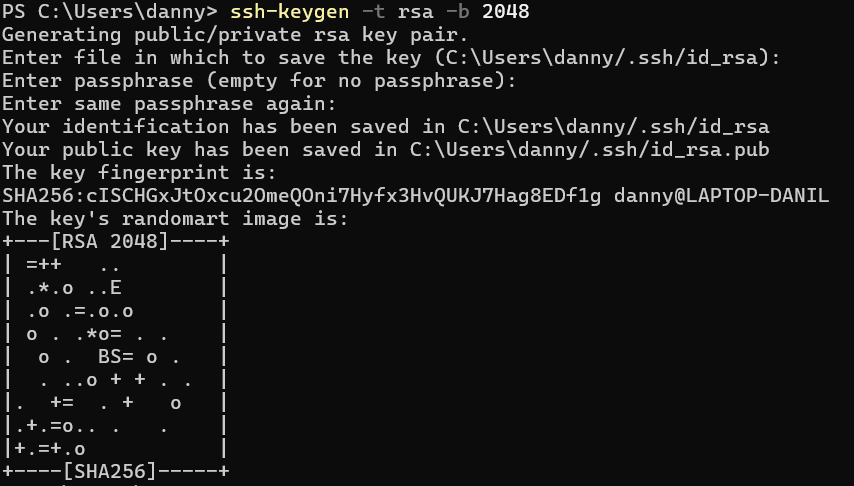
**Assignment: nmap**

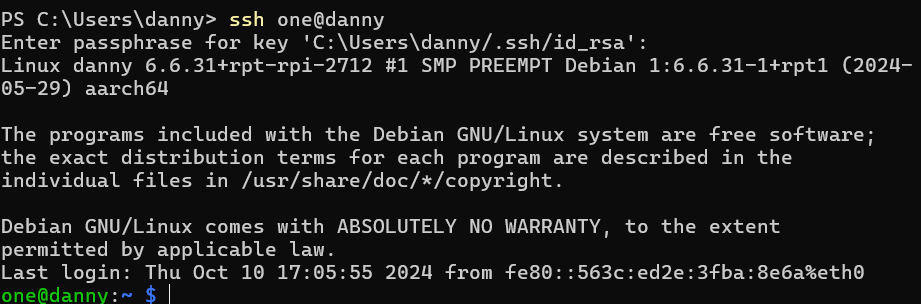
First of all, I used ***nmap*** command and specified my PI’s ip. From the result I found that my Raspberry Pi has port 22 open and is running an SSH server. After that I used ***nc*** command with my PI’s ip and a port number. I got this SSH bunner: *SSH-2.0-OpenSSH\_9.2p1 Debian-2+deb12u3*

It shows that Pi is running ***OpenSSH*** version *9.2p1.* which is part of *Debian-based distribution*



**Assignment: SSH public key authentication**

First of all I generated an SSH keypair on my laptop using the following command. 

After that I copied public key manyaly using scp (since the command, which copied the id\_rsa.pub wasn’t working) And from this moment I was able to use my passphrase instead of my PI’s password:

**Assignment: Password protected SSH keys**

Setting a passphrase for SSH keypair adds an important layer of security, since even if someone gains access to my private key file, they will also need the passphrase to use it. And eve if my laptop was stollen, it wouldn’t guarantee that an attacker can immediately log into my PI. Passphrase prevents *unauthorized Access* and *Key Theft*, providing an additional barrier against possible attackers.

**Assignment: Hardware tokens**

Advantages of Hardware Tokens for SSH Authentication:

* *Reduced Key Theft Risk*: Tokens securely store cryptographic keys, which never leave the device, minimizing the risk of theft, since physical device is much harder to steal than the digital one
* *Two-Factor Authentication (2FA)*: Many tokens support 2FA, adding an extra layer of security.
* *User Convenience*: Easy to use—simply insert the token to authenticate.
* *Durability*: Tokens are robust and can withstand physical wear, unlike software keys.

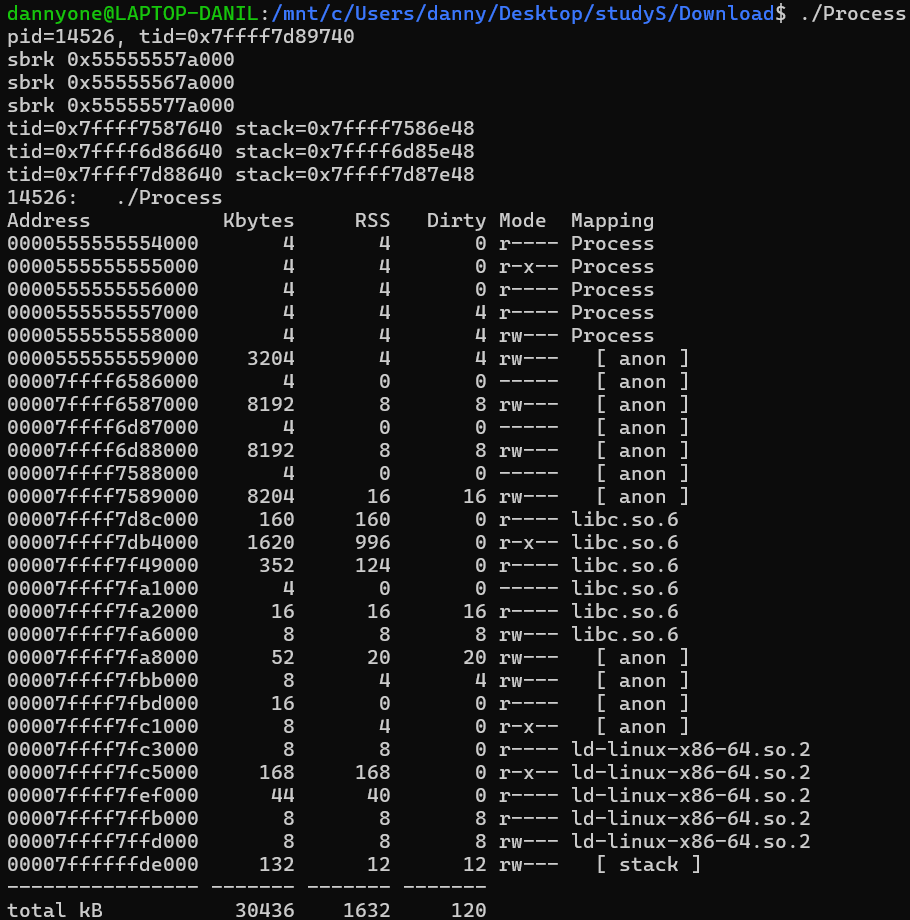
**Assignment: ProcessLayout reloaded**

I compiled and run *ProcessLayout.c* and from the output I found that stack is readable and writeable:



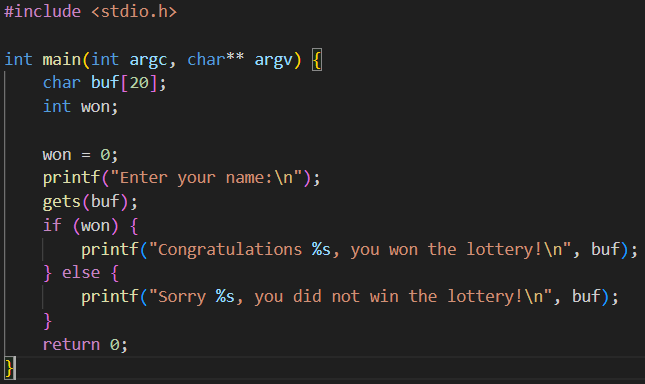
The stack holds dynamically allocated data for each function call, which needs to be frequently modified as the program runs. These modifications require both reading from and writing to the stack, so without these permisons, the stack has no sence. It is not executable, since it’s only purpose is to store data, so making it not executable allows us to separate code from data which is very good from a security point of view. Since if the stack is non-executable, attacks which involve writing malicious code to the stack, like ***stack-based buffer overflows*,** are significantly harder to perform.

**Assignment: ASLR**

Address Space Layout Randomization (ASLR) is a security feature that randomizes the memory addresses used by various components of a program each time it runs. It makes lifes of the attacker much harder, since due to ASLR it is not so easy to predict where specific code or data is stored. To disable this feature I used the following command: *sudo bash -c 'echo 0 > /proc/sys/kernel/randomize\_va\_space'* After that I run ProcessLayout several times and got the same result:  


That means that memory addresses are not randomized anymore and that makes my life as an atacker easier.

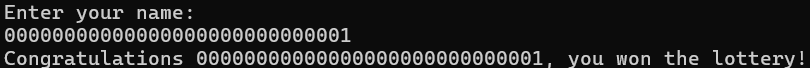
**Assignment: Lottery**

****

Obviously, as we can see in the code, no input can let you win the lottery, if we are trying to do it fair, however, by overflowing the buffer (20 bytes), I can rewrite the variable responsible for winning the lottery. So, first of all, I have tried value, which is more than 20 bytes, and then increased its size, untill I got this error:

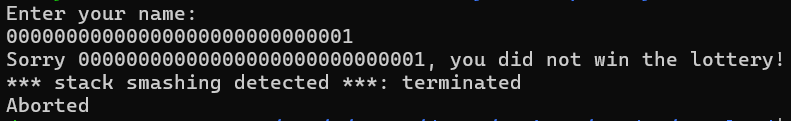


This is another security feature, so I had to disable it by adding a flag to gcc command : ***gcc -o lottery -fno-stack-protector lottery.c*** After that the error disappeared, so I have tried several more inputs and found this one: *00000000000000000000000000001* :



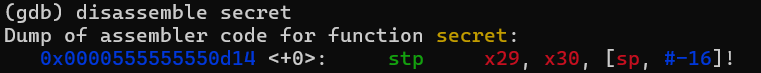
The length of this input is **29**. It takes **20** bytes to fill the buffer + **4** bytes of padding + **4** bytes for *int won* and the last **29th** byte overwrites the value of the *won* variable, allowing me to win this lottery!

**Assignment: Stack protector**

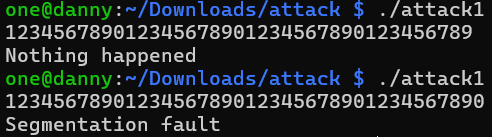
After recompiling *lottery.c* with this flag: ***-fstack-protector-all*** I enabled stack canaries again. And now the program detects stack smashing and for this reason my previous input doesn’t work anymore: 

**Assignment: Attack1**

To get the address of ***secret*** function I used gdb and disassemble this function, using the following command: *disassembe secret* I got the following address: **0x0000555555550d14**



**Assignment: Attack1 return pointer**

To find the number of bytes which can overwrite return address, I have tried inputs of different length. When the length reached **40**, I got *Segmentation fault*. 

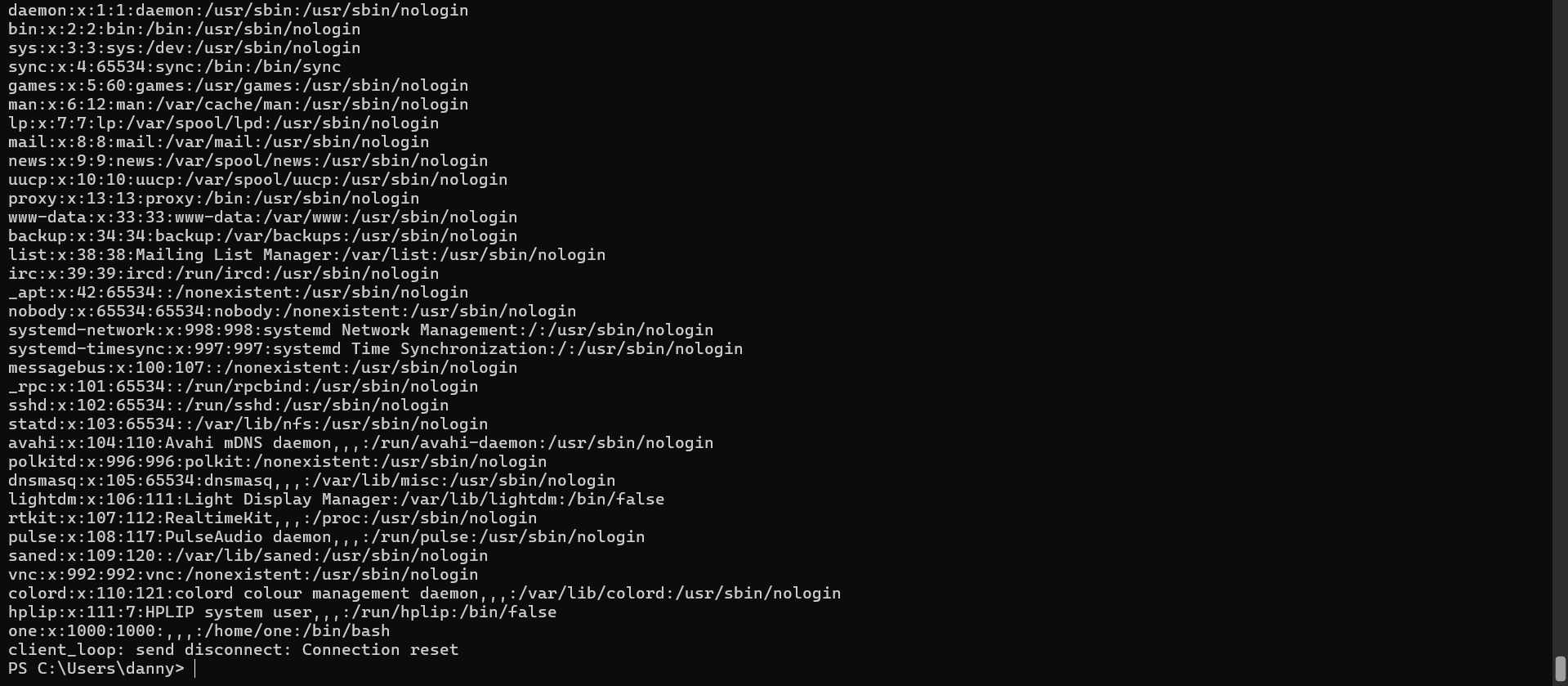
Thit means that the first byte of the return address was overwritten. From the previous exercise, I have found that the length of the return address on is 8 bytes. So in total I need **47** bytes in order to overwrite the entire return address.

**Assignment: Attack 1 exploit**

If I need to execute *secret* I have to overwrite return address into address of the *secret* function, which I found in previous exercise (**0x0000555555550d14**). Since I already know that to overwrite the entire return address I need to enter 47 bytes of data and the data starting from **40th** byte will be “new return address”, I have to put the address of the *secret* function between 40th and 47th byte. Moreover, stack is **LIFO**, for this reason I have to enter the address in the reverse order. I used the following command for that:



I got the following result, that’s the result of the *secret* function execution and the ouput was so long that I had to disconnect my PI to stop it.



**Assignment: Attack 1 with your own code**

If I overwrite the return address with the address of the first instruction of my code, then the program will jump to this address instead of the address of the previous function. After that the program will start executing my code without any questions. This is a classic technique used in buffer overflow exploits, known as **shellcode injection**.

Src: <https://www.crow.rip/crows-nest/mal/dev/inject/shellcode-injection>

**Assignment: Attack1 with stack canaries**

My previous input doesn’t work, because with stack protection enabled, a *canary value* is added before the return address on the stack. By overflowing the buffer, I overwrite the canary value, however programs cannot find this *canary value* before jumping to the return address, so it aborts, preventing the exploit. Of course, my input can be addopted for this binary as well just by placing this *canary value* right before address of the *secret* function. However it is almost imposible to find this value, so I couldn’t do it, nevertheless it is possible.

**Assignment: Attack2 exploit**

**Assignment: Attack2 with stack canaries exploit**

**Assignment: ASLR enabled**

Address Space Layout Randomization (ASLR) is a security feature that randomizes the memory addresses used by various components of a program each time it runs. When I enable it back, it randomize the memory and since my exploit relies on fixed addresses (*return address*) , it fails because I can no longer reliably predict the memory addresses needed for the overflow to work.

Danil Badarev

S 3210928