

Contents

1	Introduction	2
2	Environment Variable and Set-UID Program	2
2.1	Manipulating environment variables	2
2.2	Environment variable and Set-UID Programs	2
2.3	The PATH Environment variable and Set-UID Programs	3
2.4	The LD.PRELOAD environment variable and Set-UID Programs	3
2.5	Invoking external programs using system() versus execve()	4
2.6	Capability Leaking	5
3	Buffer Overflow Vulnerability	5
3.1	Task 7: Programming using the Crypto Library	5
4	MD5 Collision Attack	6
4.1	Task 8: Generating Two Different Files with the Same MD5 Hash	6
4.2	Task 9: Understanding MD5's Property	7
4.3	Task 10: Generating Two Executable Files with the Same MD5 Hash	7
4.4	Task 11: Making the Two Programs Behave Differently	8
5	RSA Public-Key Encryption and Signature	9
5.1	BIGNUM APIs	9
5.2	A Complete Example	9
5.3	Task 12: Deriving the Private Key	9
5.4	Task 13: Encrypting a Message	9
5.5	Task 14: Decrypting a Message	11
5.6	Task 15: Signing a Message	11
5.7	Task 16: Verifying a Message	12
5.8	Task 17: Manually Verifying an X.509 Certificate	12
6	Pseudo Random Number Generation	14
6.1	Task 18: Generate Encryption Key in a Wrong Way	14
6.2	Task 19: Guessing the Key	14
6.3	Task 20: Measure the Entropy of Kernel	15
6.4	Task 21: Get Pseudo Random Numbers from /dev/random	15
6.5	Task 22: Get Random Numbers from /dev/urandom	16

1 Introduction

This is the solvment for the CS5293 Assignment II. Each Task have a short statment for the result as well as the procedures or the screenshoot or code listing attached.

2 Environment Variable and Set-UID Program

2.1 Manipulating environment variables

For this question,

1. No output occurred when searching for `foo` initially, indicating the variable wasn't part of the environment.
 2. After setting `foo` with a string value, it still didn't appear in the environment, as assignment alone doesn't export it.
 3. Post `export foo`, the variable `foo` displayed with `printenv`, confirming its addition to the environment.
 4. Following `unset foo`, the variable `foo` ceased to appear, signifying its removal from the environment.
- 1b.

```
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$ echo $PWD  
/home/seed  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$ env | grep PWD  
PWD=/home/seed  
[03/01/24]seed@VM:~$
```

(a) printenv

```
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$ foo='test string'  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$ export foo  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
foo=test string  
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$ unset foo  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$
```

(b) set and unset env

Figure 1: Execute Result

Conclusion:

Variables must be exported to appear in the environment. The 'unset' command effectively removes them. This demonstrates the lifecycle of environment variables in Bash.

2.2 Environment variable and Set-UID Programs

Initially, `foo` was unset and, as expected, didn't appear in the output of the Set-UID program. Upon setting `foo` with a value but without exporting, `foo` still did not show up. This is because the Set-UID program inherits only exported environment variables. After exporting `foo`, it was then visible in the output, indicating that the Set-UID program did inherit `foo` from the user's process as 2.

```
[03/02/24]seed@VM:~/assignment2$ unset foo^C  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
[03/02/24]seed@VM:~/assignment2$ foo='test string'  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
[03/02/24]seed@VM:~/assignment2$ export foo  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
foo=test string  
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$
```

Figure 2: Execute Result

Conclusion:

The Set-UID programs inherit exported environment variables from the user's process. This demonstrates how users can influence Set-UID program behavior through the environment, emphasizing the need for careful security practices around such programs.

2.3 The PATH Environment variable and Set-UID Programs

The manipulations with the PATH variable and the Set-UID `ls` program demonstrate how the system's behavior changed. Initially, the custom `ls` program, when executed, listed the contents of the current directory, similar to the standard `/bin/ls` command. After modifying the PATH variable to include the current directory at the beginning and changing the ownership and permissions of the `ls` program to mimic a Set-UID program, the `ls` command should have executed the malicious program. However, the output indicates that the custom `ls` program printed a message and the user IDs, which were both 1000, meaning it did not run with root privileges.

```
[03/02/24]seed@VM:~$ touch myls.c
[03/02/24]seed@VM:~$ vim myls.c
[03/02/24]seed@VM:~$ gcc -o ls myls.c
mysls.c: In function 'main':
mysls.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
  system("ls");
  ^
[03/02/24]seed@VM:~$ ls
android      assignment2  Customization  Documents  examples.desktop  lib  Music  Pictures  source  Videos
assignment1  bin         Desktop        Downloads  get-pip.py        ls   myls.c  Public   Templates
[03/02/24]seed@VM:~$ ./ls
android      assignment2  Customization  Documents  examples.desktop  lib  Music  Pictures  source  Videos
assignment1  bin         Desktop        Downloads  get-pip.py        ls   myls.c  Public   Templates
[03/02/24]seed@VM:~$ export PATH=/home/seed:$PATH
[03/02/24]seed@VM:~$ ./ls
^C[03/02/24]seed@VM:~$ ls
^C[03/02/24]seed@VM:~$ sudo chown root ls
[03/02/24]seed@VM:~$ sudo chmod 4755 ls
[03/02/24]seed@VM:~$ ls
^C[03/02/24]seed@VM:~$ vim ls.c
[03/02/24]seed@VM:~$ gcc -o ls ls.c
ls.c: In function 'main':
ls.c:5:34: warning: implicit declaration of function 'getuid' [-Wimplicit-function-declaration]
  printf("\nMy real uid is: %d\n", getuid());
                                ^
ls.c:6:39: warning: implicit declaration of function 'geteuid' [-Wimplicit-function-declaration]
  printf("\nMy effective uid is: %d\n", geteuid());
                                ^
[03/02/24]seed@VM:~$ ls
This is my ls program
My real uid is: 1000
My effective uid is: 1000
[03/02/24]seed@VM:~$
```

Figure 3: Execute Result

```
[03/02/24]seed@VM:~$ sudo chown root ls
[03/02/24]seed@VM:~$ sudo chmod 4755 ls
[03/02/24]seed@VM:~$ export PATH=/home/seed:$PATH
[03/02/24]seed@VM:~$ ls
This is my ls program
My real uid is: 1000
My effective uid is: 0
[03/02/24]seed@VM:~$
```

Figure 4: Execute Result After Set-UID

Conclusion:

If the Set-UID program runs our code instead of the intended `/bin/ls`, the code will execute with root privileges because Set-UID programs run with the effective permissions of the file owner, which is root in this case. This demonstrates a significant security risk with using relative paths in Set-UID programs and highlights the importance of using absolute paths for system calls.

2.4 The LD_PRELOAD environment variable and Set-UID Programs

1. When `myprog` was a regular program, running it as a normal user resulted in the overridden `sleep` function being called, confirming that `LD_PRELOAD` influenced the linker to load `libmylib.so.1.0.1`

first.

2. After making myprog a Set-UID root program, running it as a normal user did not invoke the overridden sleep, indicating that the Set-UID program did not inherit the LD_PRELOAD variable from the user's environment, likely for security reasons.
3. Exporting LD_PRELOAD in the root account and then running the Set-UID root myprog resulted in the overridden sleep being called, suggesting that when the Set-UID program is run by root, it respects the LD_PRELOAD variable.
4. With myprog set as a Set-UID program owned by another user (user1) and LD_PRELOAD set in a non-root account, the overridden sleep was not called, similar to the second case, reinforcing the idea that Set-UID programs ignore LD_PRELOAD from non-owner environments.

```
[03/02/24]seed@VM:~/assignment2$ cat mylib.c
#include <stdio.h>
void sleep (int s)
{
/* If this is invoked by a privileged program,
you can do damages here! */
printf("I am not sleeping!\n");
}
[03/02/24]seed@VM:~/assignment2$ gcc -fPIC -g -c mylib.c
[03/02/24]seed@VM:~/assignment2$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[03/02/24]seed@VM:~/assignment2$ export LD_PRELOAD=./libmylib.so.1.0.1
[03/02/24]seed@VM:~/assignment2$ vim myprog.c
mylib.c myprog.c
[03/02/24]seed@VM:~/assignment2$ vim myprog.c
[03/02/24]seed@VM:~/assignment2$ gcc -o myprog myprog.c
myprog.c: In function 'main':
myprog.c:4:1: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
sleep(1);
^
[03/02/24]seed@VM:~/assignment2$ ./myprog
I am not sleeping!
[03/02/24]seed@VM:~/assignment2$ sudo chown root myprog
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 myprog
[03/02/24]seed@VM:~/assignment2$ ./myprog
[03/02/24]seed@VM:~/assignment2$ sudo export LD_PRELOAD=./libmylib.so.1.0.1
sudo: export: command not found
[03/02/24]seed@VM:~/assignment2$ sudo -i
root@VM:~# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:~# ./myprog
-bash: ./myprog: No such file or directory
root@VM:~# cd /eh^C
root@VM:~# cd /home/seed/assignment2
root@VM:/home/seed/assignment2# ./myprog
I am not sleeping!
root@VM:/home/seed/assignment2#
```

Figure 5: Execute Result

2.5 Invoking external programs using system() versus execve()

1.If Bob were to exploit the system() call with a command like “./25 /etc/passwd; rm -f /path/to/some/file”, as figured in 6a, the shell would execute the cat /etc/passwd command and then attempt the rm -f command, potentially allowing unauthorized file deletion if the syntax were correct and the shell executes the second command.

The use of system() in a Set-UID program poses a security risk due to its reliance on the shell, which can interpret additional commands and metacharacters. This risk is not present with execve() as it does not invoke a shell and executes the specified command directly. The observations suggest that the program is functioning with elevated privileges, but the specific access to /etc/shadow could not be confirmed from the provided output.

2.After recompiling and setting the program to use execve(), any attempts to use command chaining or injection as part of the input to the Set-UID program should fail, as figured in 6b, input such as filename; rm -f somefile would not cause the deletion of somefile because execve() would attempt to pass the entire string as a single argument to /bin/cat, which would then result in an error as it would be an invalid file name.


```

[03/02/24]seed@VM:~/assignment2$ sudo gcc -o 25 25.c
[03/02/24]seed@VM:~/assignment2$ sudo chown root 25
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 25
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ echo "123" > testfile
[03/02/24]seed@VM:~/assignment2$ sudo chmod 444 testfile
[03/02/24]seed@VM:~/assignment2$ cat testfile
123
[03/02/24]seed@VM:~/assignment2$ ./25 "testfile;rm testfile"
rm: remove write-protected regular file 'testfile'? y
[03/02/24]seed@VM:~/assignment2$ cat testfile
cat: testfile: No such file or directory
[03/02/24]seed@VM:~/assignment2$

```

Set-UID Program
add a read-only file
remove it by the Set-UID program

(a) Step 1 Execute Result

```

[03/02/24]seed@VM:~/assignment2$ vim 25.c
[03/02/24]seed@VM:~/assignment2$ sudo gcc -o 25 25.c
25.c: In function 'main':
25.c:18:2: warning: implicit declaration of function 'execve' [-Wimplicit-function-declaration]
execve(v[0], v, NULL);
^
[03/02/24]seed@VM:~/assignment2$ sudo chown root 25
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 25
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ echo "123" > testfile
[03/02/24]seed@VM:~/assignment2$ sudo chmod 444 testfile
[03/02/24]seed@VM:~/assignment2$ cat testfile
123
[03/02/24]seed@VM:~/assignment2$ ./25 "testfile;rm testfile"
/bin/cat: 'testfile;rm testfile': No such file or directory
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$

```

Not Worked

(b) Step 2 Execute Result

Figure 6: Execute Result

2.6 Capability Leaking

The program ./26 successfully wrote "Malicious Data" to /etc/zzz. Initially unable to open /etc/zzz, after setting correct permissions and ownership, the Set-UID program, running with root privileges, opened /etc/zzz. Upon dropping privileges with `setuid(getuid())`, the child process inherited the file descriptor with root access, leading to the capability leak which allowed writing to the file, even as a non-privileged user. This demonstrates the security risk of inheriting file descriptors from privileged processes.

```

[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ sudo chown root 26
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4775 26
[03/02/24]seed@VM:~/assignment2$ ./26
Cannot open /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo touch /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo chown root /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo chmod 0644 /etc/zzz
[03/02/24]seed@VM:~/assignment2$ ./26
[03/02/24]seed@VM:~/assignment2$ cat /etc/zzz
Malicious Data
[03/02/24]seed@VM:~/assignment2$

```

Figure 7: Execute Result

3 Buffer Overflow Vulnerability

3.1 Task 7: Programming using the Crypto Library

```

1 #!/usr/bin/python3
2 from Crypto.Cipher import AES
3 from Crypto.Util.Padding import pad

```

```

4
5 plaintext = "This is a top secret."
6 ciphertext = "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
7 iv = "aabbccddeeff00998877665544332211"
8 plaindatabits = bytearray(plaintext, encoding='utf-8')
9 cipherdatabits = bytearray.fromhex(ciphertext)
10 ivdatabits = bytearray.fromhex(iv)
11
12 with open('./words.txt') as f:
13     keys = f.readlines()
14
15 for k in keys:
16     k = k.rstrip('\n')
17     if len(k) <= 16:
18         key = k + '#'*(16-len(k))
19         cipher = AES.new(key=bytearray(key, encoding='utf-8'), mode=AES.MODE_CBC, iv=
ivdatabits)
20         testbits = cipher.encrypt(pad(plaindatabits, 16))
21         if cipherdatabits == testbits:
22             print("find the key:",key)
23             exit(0)
24
25 print("cannot find the key!")

```

Listing 1: find key script in Python

The script use the Crypto to call the AES API in Python to encrypt the text, and simple loop to find a matched key in word list from a file.

After execute the script, the matched key is “Syracuse”.

4 MD5 Collision Attack

4.1 Task 8: Generating Two Different Files with the Same MD5 Hash

```

[01/27/24]seed@VM:~/.../task8$ hexdump -C out1.bin
00000000  54 68 69 73 20 69 73 20  61 20 74 6f 70 20 73 65  |This is a top se|
00000010  63 72 65 74 2e 00 00 00  00 00 00 00 00 00 00 00  |cret.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
00000040  07 ff c2 20 ad d2 ff 6f  b2 65 f4 db eb 1a 84 9b  |... ..o.e.....|
00000050  45 4b bc 03 4a a0 94 b0  11 0a 82 06 16 3a ab 08  |EK..J.....|
00000060  ff cb fc bd 78 ec 39 82  c7 07 47 61 de a5 c8 5e  |...x.9...Ga...^|
00000070  f5 9f 4c b2 82 9b 6c 81  1a ad 8b 78 61 a2 00 f8  |...l...xa...|
00000080  45 6f c7 cb 5a ac 46 b6  e1 82 b5 fe 7f b4 aa 56  |Eo..Z.F.....V|
00000090  96 1e 53 6a 25 e6 35 31  d1 5a 06 4b ec c7 01 70  |..Sj%.51.Z.K...p|
000000a0  e6 c3 3e 18 06 8b 04 d4  58 80 5d 2f a0 20 f2 ac  |..>....X.]/. ..|
000000b0  4c ca ee 7f d5 c9 28 a9  62 a0 62 52 ae 33 c4 b6  |L.....(.b.br.3..|
000000c0
[01/27/24]seed@VM:~/.../task8$ hexdump -C out2.bin
00000000  54 68 69 73 20 69 73 20  61 20 74 6f 70 20 73 65  |This is a top se|
00000010  63 72 65 74 2e 00 00 00  00 00 00 00 00 00 00 00  |cret.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
00000040  07 ff c2 20 ad d2 ff 6f  b2 65 f4 db eb 1a 84 9b  |... ..o.e.....|
00000050  45 4b bc 83 4a a0 94 b0  11 0a 82 06 16 3a ab 08  |EK..J.....|
00000060  ff cb fc bd 78 ec 39 82  c7 07 47 61 de 25 c9 5e  |...x.9...Ga...^|
00000070  f5 9f 4c b2 82 9b 6c 81  1a ad 8b f8 61 a2 00 f8  |...l...a...|
00000080  45 6f c7 cb 5a ac 46 b6  e1 82 b5 fe 7f b4 aa 56  |Eo..Z.F.....V|
00000090  96 1e 53 ea 25 e6 35 31  d1 5a 06 4b ec c7 01 70  |..S%.51.Z.K...p|
000000a0  e6 c3 3e 18 06 8b 04 d4  58 80 5d 2f a0 a0 f1 ac  |..>....X.]/. ...|
000000b0  4c ca ee 7f d5 c9 28 a9  62 a0 62 d2 ae 33 c4 b6  |L.....(.b.b.3..|
000000c0

```

Figure 8: MD5 Collision

Question:

1. Length Not Multiple of 64: md5collgen might pad the prefix with additional bytes to reach a multiple of 64, as MD5 operates on 64-byte blocks.
2. Prefix of Exactly 64 Bytes: No padding should be necessary, and the collision blocks will directly follow the prefix in both output files.

3. **Extent of Differences:** The number and distribution of differing bytes can vary depending on the specific collision generated.

4.2 Task 9: Understanding MD5's Property

```
1 # out1.bin and out2.bin are two files generated in Task8 with the same MD5 hash:
2 $ md5sum out1.bin
3 189b839e730f57d79a8b4a98aadb36ea out1.bin
4 $ md5sum out2.bin
5 189b839e730f57d79a8b4a98aadb36ea out2.bin
6 # generate a suffix
7 $ echo -n "Lanch a missile." > suffix.txt
8 # concat two files with the suffix
9 $ cat out1.bin suffix.txt > out3.bin
10 $ cat out2.bin suffix.txt > out4.bin
11 # observe the concated files
12 $ md5sum out3.bin
13 30c758787b6e481692efd404ccb5c1a4 out3.bin
14 $ md5sum out4.bin
15 30c758787b6e481692efd404ccb5c1a4 out4.bin
```

Listing 2: MD5 Property test script

Result:

The two concated files that with the same prefix MD5 and same suffix will result in the same result MD5 HASH value.

4.3 Task 10: Generating Two Executable Files with the Same MD5 Hash

```

1 # After Compiling the source code to Executable Program
2 # Cut the Executable Program to two parts
3 $ head -c 4224 out1 > prefix
4 $ tail -c +4287 out1 > suffix
5 # generated the same MD5 part of the program and join them to suffix
6 $ md5collgen -p prefix -o out1.bin out2.bin
7 $ cat out1.bin suffix > program1
8 $ cat out2.bin suffix > program2
9 # observe the MD5 of two Different program
10 $ md5sum program1
11 60824a7b99425740d90c0cfb41a34a1e  program1
12 $ md5sum program2
13 60824a7b99425740d90c0cfb41a34a1e  program2
14 $ chmod +x program1
15 $ chmod +x program2
16 $ ./program1
17 $ ./program2
18 # observe the execute result

```

Listing 3: MD5 Executable Program

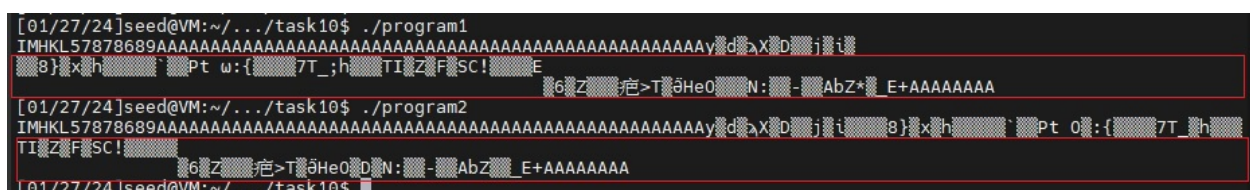


Figure 9: Execute Result

Obviously, they are not the same program according to the execute result but they have the same MD5 HASH value.

4.4 Task 11: Making the Two Programs Behave Differently

```

1 # The same operation as TASK10
2 $ head -c 4160 program > prefix
3 $ md5collgen -p prefix -o out1.bin out2.bin
4 $ tail -c +4288 program > suffix
5 $ cat out1.bin suffix > test1
6 $ cat out2.bin suffix > test2
7 # Use bless modify both Y Array to be same as the X Array of test1
8 $ bless test1
9 $ bless test2
10 $ ./test1 # the X array and Y array are the same in test1
11 This is a good program
12 $ ./test2 # the X array and Y array are different in test2
13 This is a bad program
14 # Observe the MD5 HASH, they are the same
15 $ md5sum test1
16 8022e63d3dba85eb0ba3278ff78485c6 test1
17 $ md5sum test2
18 8022e63d3dba85eb0ba3278ff78485c6 test2

```

Listing 4: Different Behavior Program

```

[01/27/24]seed@VM:~/.../task11$ history | grep echo^C
[01/27/24]seed@VM:~/.../task11$ ./test1
This is a good program
[01/27/24]seed@VM:~/.../task11$ ./test2
This is a bad program
[01/27/24]seed@VM:~/.../task11$ md5sum test1
8022e63d3dba85eb0ba3278ff78485c6 test1
[01/27/24]seed@VM:~/.../task11$ md5sum test2
8022e63d3dba85eb0ba3278ff78485c6 test2
[01/27/24]seed@VM:~/.../task11$
[01/27/24]seed@VM:~/.../task11$

```

Figure 10: Different Behavior

```

.n>&...L."..b...Q.....3...\R|Z..).
D...n-..._G...cm...o...`}.T...,...
N.....>...0...m...8E.'..M.IwGI...
e.3.....a.....?..=.u..)....w...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAB.....

```

```

.n>&...L."..b...Q.....>3...\R|Z..).
D...n-..._G...Xdm...o...`}.T...,"...
N.....>...0...m...8...'.M.IwGI...
e.3.....a.....?..=.u..)....|w...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAB.....

```

```

.n>&...L."..b...Q.....3...\R|Z..).
D...n-..._G...cm...o...`}.T...,...
N.....>...0...m...8E.'..M.IwGI...
e.3.....a.....?..=.u..)....w...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAB.....

```

```

.n>&...L."..b...Q.....3...\R|Z..).
D...n-..._G...cm...o...`}.T...,...
N.....>...0...m...8E.'..M.IwGI...
e.3.....a.....?..=.u..)....w...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAB.....

```

(a) same arrays in test1

(b) different arrays in test2

Figure 11: test program binary

5 RSA Public-Key Encryption and Signature

5.1 BIGNUM APIs

5.2 A Complete Example

The running example result is:

```
[01/28/24]seed@VM:~/assignment1$ gcc bn_sample.c -lcrypto -o bn_sample
[01/28/24]seed@VM:~/assignment1$ ./bn_sample
a * b = A53C7BED561E3D6823B62CF03C4D1200E14647131B3CDC39BC7B2D68CD6BCD63A9A2174424B31E1D2BB00DCBDAF78A1C
a^b mod n = 7572915C7FC40D423E4655C98515472D2312FD31AC507D4468DA7D324BC4BCD1
[01/28/24]seed@VM:~/assignment1$ ./bn_sample
a * b = BE7081A10CE8974FF302D5F62AF18098BE6FCD48BB01D86B3C40AE8C87FE9A81AE950A5404DE85ED78F10C844042471C
a^b mod n = 08A8A5502A73ED173E41AB4012993AE9058FB23BDD8AB99C357E32C3C7440A15
[01/28/24]seed@VM:~/assignment1$
```

Figure 12: BIGNUM API Example

5.3 Task 12: Deriving the Private Key

```
1 #include <stdio.h>
2 #include <openssl/bn.h>
3 int main() {
4     // Given values
5     char p_hex[] = "F7E75FDC469067FFDC4E847C51F452DF";
6     char q_hex[] = "E85CED54AF57E53E092113E62F436F4F";
7     char e_hex[] = "0D88C3";
8     // Convert hexadecimal strings to BIGNUM
9     BIGNUM *p = BN_new();
10    BIGNUM *q = BN_new();
11    BIGNUM *e = BN_new();
12    BN_hex2bn(&p, p_hex);
13    BN_hex2bn(&q, q_hex);
14    BN_hex2bn(&e, e_hex);
15    // Calculate phi(n) = (p-1)*(q-1)
16    BIGNUM *phi_n = BN_new();
17    BN_sub(phi_n, p, BN_value_one());
18    BN_sub_word(phi_n, 1);
19    BN_mul(phi_n, phi_n, q, BN_CTX_new());
20    BN_sub_word(phi_n, 1);
21    // Calculate d = e^-1 mod phi(n)
22    BIGNUM *d = BN_new();
23    BN_mod_inverse(d, e, phi_n, BN_CTX_new());
24    // Print the private key d
25    char *d_hex = BN_bn2hex(d);
26    printf("Private Key (d): %s\n", d_hex);
27    // Free allocated memory
28    return 0;
29 }
```

Listing 5: C Program Code for Deriving the Private Key

```
1 $ gcc main.c -lcrypto -o main
2 $ ./main
3 Private Key (d): 9B740D556F9080815E14B6633E9BCC3C87EAA0F0AD699E0A7E0719A725A94AA7
```

Listing 6: Result of the Task 12

5.4 Task 13: Encrypting a Message

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <openssl/bn.h>
4 int main() {
```

```

5 // Given public key values
6 char n_hex[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4DOCB81629242FB1A5";
7 char e_hex[] = "010001";
8 // Given message
9 char message_hex[] = "4120746f702073656372657421"; // Hex representation of "A
top secret!"
10 // Convert public key values to BIGNUM
11 BIGNUM *n = BN_new();
12 BIGNUM *e_value = BN_new();
13 BN_hex2bn(&n, n_hex);
14 BN_hex2bn(&e_value, e_hex);
15 // Convert the message from hex to BIGNUM
16 BIGNUM *message_bn = BN_new();
17 BN_hex2bn(&message_bn, message_hex);
18 // Allocate memory for the result
19 BIGNUM *result = BN_new();
20 // Perform encryption: ciphertext = message^e mod n
21 BN_mod_exp(result, message_bn, e_value, n, BN_CTX_new());
22 // Print the encrypted result
23 char *result_hex = BN_bn2hex(result);
24 printf("Encrypted Result: %s\n", result_hex);
25 // Free allocated memory
26 return 0;
27 }

```

Listing 7: C Program Code for Encrypt

```

1 #include <stdio.h>
2 #include <openssl/bn.h>
3 int main() {
4 // Given private key values
5 char n_hex[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4DOCB81629242FB1A5";
6 char e_hex[] = "010001";
7 char d_hex[] = "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D";
8 // Given ciphertext
9 char ciphertext_hex[] = "6
FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC";
10 // Convert private key values to BIGNUM
11 BIGNUM *n = BN_new();
12 BIGNUM *e_value = BN_new();
13 BIGNUM *d = BN_new();
14 BN_hex2bn(&n, n_hex);
15 BN_hex2bn(&e_value, e_hex);
16 BN_hex2bn(&d, d_hex);
17 // Convert ciphertext from hex to BIGNUM
18 BIGNUM *ciphertext_bn = BN_new();
19 BN_hex2bn(&ciphertext_bn, ciphertext_hex);
20 // Allocate memory for the result
21 BIGNUM *result = BN_new();
22 // Perform decryption: message = ciphertext^d mod n
23 BN_mod_exp(result, ciphertext_bn, d, n, BN_CTX_new());
24 // Print the decrypted result
25 char *result_hex = BN_bn2hex(result);
26 printf("Decrypted Result: %s\n", result_hex);
27 // Free allocated memory
28 return 0;
29 }

```

Listing 8: C Program Code for Evaluation

```

1 $ ./main
2 Encrypted Result: 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
3 $ ./decrypt
4 Decrypted Result: 4120746F702073656372657421
5 $ python -c 'print("4120746F702073656372657421".decode("hex"))'
6 A top secret!

```

Listing 9: Result of the Task 13

5.5 Task 14: Decrypting a Message

This Task is a lot of same with the Task 13. Just change the ciphertext value.

```
1 //same with the Task 13 evaluation
2 #include <stdio.h>
3 #include <openssl/bn.h>
4 int main() {
5     // Given private key values
6     // Given ciphertext
7     char ciphertext_hex[] = "8
COF971DF2F3672B28811407E2DABBE1DA0FEBBDFC7DCB67396567EA1E2493F";
8     // Convert private key values to BIGNUM
9     //...
10    // Free allocated memory
11    return 0;
12 }
```

Listing 10: C Program Code for Decryption

```
1 $ gcc decrypt.c -lcrypto -o decrypt
2 $ ./decrypt
3 Decrypted Result: 50617373776F72642069732064656573
4 $ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
5 Password is dees
```

Listing 11: Result of the Task 14

5.6 Task 15: Signing a Message

Signature the message is using $S = m^d \bmod n$.

```
1 $ python -c 'print("I owe you $2000.".encode("hex"))'
2 49206f776520796f752024323030302e # covert the text to hex
3 $ python -c 'print("I owe you $3000.".encode("hex"))'
4 49206f776520796f752024333030302e # covert the text to hex
5 $ gcc sign.c -lcrypto -o sign # sign.c is listing as Listing 20
6 $ ./sign
7 Signing: I owe you $2000.
8 Signature Result: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
9 Signing: I owe you $3000.
10 Signature Result: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

Listing 12: Result of the Task 15

```
1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <string.h>
4 int main() {
5     //signing the first message
6     printf("Signing: I owe you $2000.\n");
7     BIGNUM *m = BN_new();
8     BN_hex2bn(&m, "49206f776520796f752024323030302e");
9     BIGNUM *d = BN_new();
10    BIGNUM *n = BN_new();
11    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
12    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
13    BIGNUM *signature = BN_new();
14    //s = m ^ d mod n
15    BN_mod_exp(signature, m, d, n, BN_CTX_new());
16    char *signature_hex = BN_bn2hex(signature);
17    printf("Signature Result: %s\n", signature_hex);
18    //signing another message
19    printf("Signing: I owe you $3000.\n");
20    BIGNUM *m2 = BN_new();
21    BN_hex2bn(&m2, "49206f776520796f752024333030302e");
22    BN_mod_exp(signature, m2, d, n, BN_CTX_new());
```

```

23 //...
24 // Free BIGNUMs
25 return 0;
26 }

```

Listing 13: C Program Code for Signature

Compare both signatures, obviously the message has only 1 bit change while the signature is totally different.

5.7 Task 16: Verifying a Message

Signature the message is using $m' = s^e \bmod n$. And then compare the whether $m' = m$?

```

1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <string.h>
4 int main() {
5     // Convert signature to BIGNUM
6     BIGNUM *signature = BN_new();
7     BN_hex2bn(&signature, "643
D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
8     // Encrypt the message using Alice's public key
9     BIGNUM *e = BN_new();
10    BIGNUM *n = BN_new();
11    BIGNUM *message = BN_new();
12    BN_hex2bn(&e, "010001");
13    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
14    BN_hex2bn(&message, "4c61756e63682061206d697373696c652e");
15    BIGNUM *test_message = BN_new();
16    //m' = s ^ e mod n
17    BN_mod_exp(test_message, signature, e, n, BN_CTX_new());
18    // Compare the m with m'
19    if (BN_cmp(message, test_message) == 0) {
20        printf("Signature is valid.\n");
21    } else {
22        printf("Signature is invalid.\n");
23    }
24    // Free BIGNUMs
25 }

```

Listing 14: C Program Code for Verifying

```

[01/28/24]seed@VM:~/../task15$ ./verify
Signature is valid.
[01/28/24]seed@VM:~/../task15$ ./corrupted
Signature is invalid.
[01/28/24]seed@VM:~/../task15$

```

Figure 13: Verification Result

Figure 13 show the result of the right signature and the corrupted one, respectively. The modified signature (with 3F) won't match the encrypted message generated using Alice's public key. And RSA signature verification is highly sensitive to even minor changes in the signature or message.

5.8 Task 17: Manually Verifying an X.509 Certificate

According to Step 1 to Step 4, the X.509 certificate detailed is:
And then, a program to verify the information as follows:

```

1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <string.h>
4 int main()
5 {

```



```
[01/28/24]seed@VM:~/.../task17$ openssl x509 -in c1.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDBC90A20EDDFE3C4D09E9AA97A1D8288E51156
DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA25037B9A52416610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED5751
7426E27DC7CA622E131B7F238825536FC13458008B84FF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED37E236B4824862B5499AEC
C767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329AE
FB494FA415B9CEF25C19576D6B79A72BA2272013B5D03D40D321300793EA99F5
```

(a) Modulus

```
bb:b4:8e:27:87:27:c5:2b:74:d4:a8:d6:97:de:c3:
64:f9:ca:ce:53:a2:56:bc:78:17:8e:49:03:29:ae:
fb:49:4f:a4:15:b9:ce:f2:5c:19:57:6d:6b:79:a7:
2b:a2:27:20:13:b5:d0:3d:40:d3:21:30:07:93:ea:
99:f5
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints: critical
CA:TRUE, pathlen:0
X509v3 Subject Key Identifier:
B7:6B:A2:EA:A8:AA:84:8C:79:EA:B4:DA:0F:98:B2:C5:95:76:B9:F4
X509v3 Authority Key Identifier:
keyid:03:DE:50:35:56:D1:4C:BB:66:F0:A3:E2:1B:C3:97:B2:3D:D1:55
```

(b) Exponent

```
[01/28/24]seed@VM:~/.../task17$ cat signature | tr -d '[:space:]':
59e44ad8a982ba9a4af1630c6d762675b33c74bec5f73da79192f8cf062d5810edf3b8d6fc6cfff139632cd4fe98724850b74a2c2f60ff5a7d8
7d768aeee9c9582b6e006fb9cd24eac442c54c16859d34613923bfc68e95c984a9b2e5410f4478d795b9cfd974bf584fe716ff7c4030c46c4e
224dcb83673a93bf2bc5c59c1af243a1253b84f6f7536ea885aede14749130060df207d4c408ba4364c5e23fdaacc541afa37e8427674f713
bb4a7d3659819bc744df8973b93342e860c24d615d125a10f6efff33891450e8d69fc6b95c2b35dbadeddd36b625f2958aac693f9afe1af815
286dea185ac2d26218af4078b5fa5e098f53f9ccf823a1833123f4c6[01/28/24]seed@VM:~/.../task17$
```

(c) Signature

```
[01/28/24]seed@VM:~/.../task17$ sha256sum c0_body.bin
bbc2a75949c896bd66db4e636aab8b2cbaa970bc8302d8d02c99104ab04f4dd6 c0_body.bin
```

(d) Body HASH

Figure 14: Task 17 Information

```
6 char bodyhash[] = "BBC2A75949C896BD66DB4E...302D8D02C99104AB04F4DD6"; # the Body
HASH
7 BIGNUM *n = BN_new();
8 BIGNUM *e = BN_new();
9 BIGNUM *s = BN_new();
10 BIGNUM *m = BN_new();
11 BN_hex2bn(&n, "C14BB3654770BC...F5"); # the Modulus
12 BN_hex2bn(&e, "010001"); # the Exponent
13 BN_hex2bn(&s, "59e44ad8a982ba9a4af163...23f4c6"); # the Signature
14 BN_mod_exp(m, s, e, n, BN_CTX_new());
15 char *message_hex = BN_bn2hex(m);
16 size_t length = strlen(message_hex);
17 //compare the last 64 bytes
18 if (length > 64) {
19     // Move the pointer to the start of the last 64 bytes
20     char *substring = message_hex + (length - 64);
21     // Print or manipulate the substring as needed
22     int result = strcmp(substring, bodyhash);
23     if (result == 0) {
24         printf("Verification PASS.\n");
25     } else {
26         printf("Verification Failed\n");
27     }
28 } else {
29     // If the string is shorter than 64 bytes, handle it accordingly
30     printf("Something Wrong");
31 }
32 return 0;
33 }
```

Listing 15: C Program Code for Manually Verifying

```
[01/28/24]seed@VM:~/.../task17$ ./verify
Verification PASS.
[01/28/24]seed@VM:~/.../task17$ █
```

Figure 15: Verification Result

6 Pseudo Random Number Generation

6.1 Task 18: Generate Encryption Key in a Wrong Way

```
[01/28/24]seed@VM:~/.../task18$ gcc main.c -o main
[01/28/24]seed@VM:~/.../task18$ ./main
1706437608
e55b043016622014733f232fbbde0a05
[01/28/24]seed@VM:~/.../task18$ ./main
1706437609
2b288a091b1f7b6bbd5f4e9c788dcc6a
[01/28/24]seed@VM:~/.../task18$ ./main
1706437611
d868effed1acea8dad7179d7c91f6a64
[01/28/24]seed@VM:~/.../task18$ ./main
1706437612
09792625e4c9efbb6139de5c54fb6b48
[01/28/24]seed@VM:~/.../task18$ ./main
1706437613
5ec6ca03156988f27d0ecdc8b52c9e5f
[01/28/24]seed@VM:~/.../task18$ ./main
1706437614
25b64d1fc6f0ecadcd81f119bcc6ca81
[01/28/24]seed@VM:~/.../task18$ vim main.c
[01/28/24]seed@VM:~/.../task18$ gcc main.c -o main
[01/28/24]seed@VM:~/.../task18$ ./main
1706437638
67c6697351ff4aec29cdbaabf2fbe346
[01/28/24]seed@VM:~/.../task18$ ./main
1706437639
67c6697351ff4aec29cdbaabf2fbe346
[01/28/24]seed@VM:~/.../task18$ ./main
1706437639
67c6697351ff4aec29cdbaabf2fbe346
[01/28/24]seed@VM:~/.../task18$ ./main
1706437646
67c6697351ff4aec29cdbaabf2fbe346
```

Before comment

After comment

Figure 16: Result Observation

The `time(NULL)` function returns the current time as the number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC). It prints this value to the console.

`srand(time(NULL))` seeds the pseudo-random number generator (`rand()`) with the current time. The purpose of seeding is to initialize the random number generator with a somewhat unpredictable value, ensuring that subsequent calls to `rand()` produce different sequences of pseudo-random numbers.

When comment out the line `srand(time(NULL))` (Line 13) and rerun the program, the pseudo-random numbers generated by `rand()` will be the same every time when running the program because the generator is not reseeded.

In summary, the `srand(time(NULL))` line is responsible for seeding the pseudo-random number generator based on the current time, introducing unpredictability and ensuring a different sequence of pseudo-random numbers each time the program is executed.

6.2 Task 19: Guessing the Key

Firstly, I program a C program to generate all potential key from time to time accordingly.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define KEYSIZE 16
5 void main() {
6     long long int timestamp;
7     for(timestamp = 1523920129; timestamp <= 1523920129+2*24*60*60; timestamp++) {
8         char key[KEYSIZE];
9         srand(timestamp);
10        for (int i = 0; i < KEYSIZE; i++){
11            key[i] = rand()%256;
12            printf("%.2x", (unsigned char)key[i]);
13        }
14    }
```

```

14     printf("\n");
15 }
16 }

```

Listing 16: C Program Code for Generating Keys

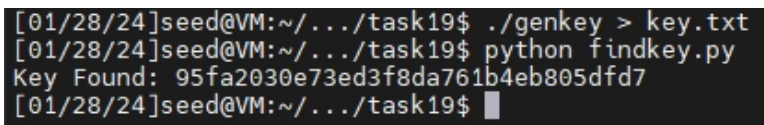
And using `genkey > key.txt` to generate a key files, after generating, then I use a Python script to find the key.

```

1 import binascii
2 from Crypto.Cipher import AES
3 # Read keys from the file
4 with open('./key.txt') as fp:
5     keys = fp.readlines()
6 # Iterate through each key in the file
7 for keyhex in keys:
8     # Remove trailing newline characters
9     keyhex = keyhex.rstrip()
10    # Convert IV, key, and plaintext from hex to bytes
11    iv = binascii.unhexlify('09080706050403020100A2B2C2D2E2F2'.lower())
12    key = binascii.unhexlify(keyhex.lower())
13    plaintext = binascii.unhexlify('255044462d312e350a25d0d4c5d80a34'.lower())
14    # Create AES encryptor object
15    encryptor = AES.new(key, AES.MODE_CBC, iv)
16    # Encrypt the plaintext
17    ciphertext = encryptor.encrypt(plaintext)
18    # Check if the ciphertext matches the known value
19    if ciphertext == binascii.unhexlify('d06bf9d0dab8e8ef880660d2af65aa82'.lower()):
20        print("Key Found: " + binascii.hexlify(key))

```

Listing 17: Python Script for Finding the Key



```

[01/28/24]seed@VM:~/.../task19$ ./genkey > key.txt
[01/28/24]seed@VM:~/.../task19$ python findkey.py
Key Found: 95fa2030e73ed3f8da761b4eb805dfd7
[01/28/24]seed@VM:~/.../task19$

```

Figure 17: Result

And finally find the key: 95fa2030e73ed3f8da761b4eb805dfd7

6.3 Task 20: Measure the Entropy of Kernel

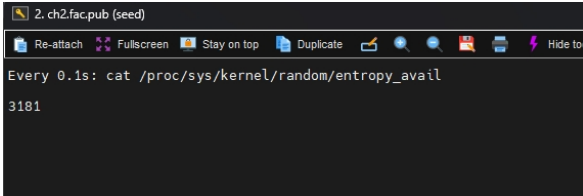
Since my Seed lab is running in a remote VM, I try to measure the entropy by such:

- User input: Typing rapidly and randomly on the keyboard introduces unpredictable timing between key presses, boosting entropy.
- Moving the mouse remotely in X11 erratically and clicking randomly generates unpredictable mouse events, adding to entropy.
- Disk activity: Reading large files from a hard drive or SSD creates variable disk access times due to seek times and read speeds, contributing to entropy.
- Network activity: Visiting the server by ssh involves network interactions that introduce variability in packet arrival times and server responses, increasing entropy. Downloading large files also involves unpredictable network delays and data transfer patterns.

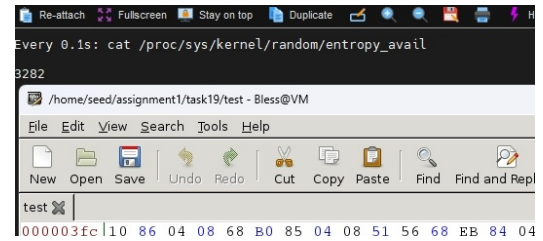
These activities significantly increase entropy.

6.4 Task 21: Get Pseudo Random Numbers from `/dev/random`

In the task, when execute the command, `cat /dev/random | hexdump`, the entropy will decrease significantly, when the entropy approaching 0, the command will stop outputting any information. And I try to increase the entropy by upload a file to the server, shown as 19(b),

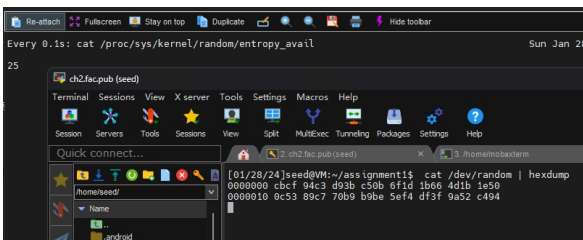


(a) Background Entropy

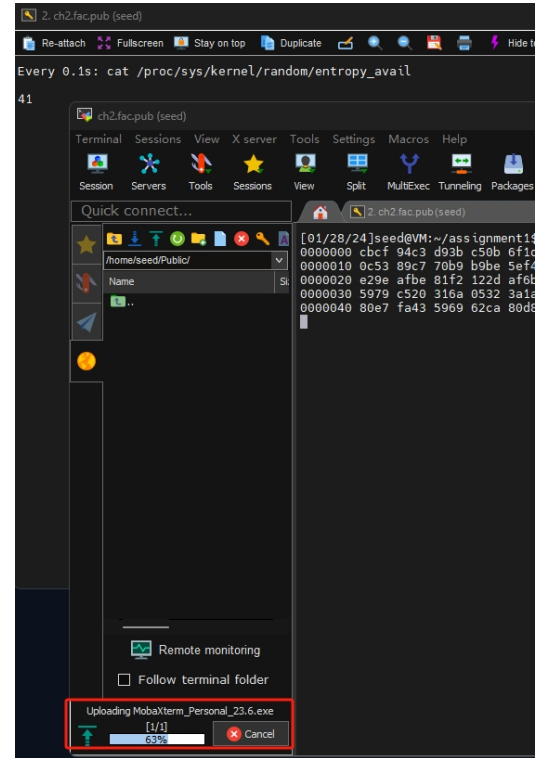


(b) Activity Entropy

Figure 18: Task 20 Observation



(a) Background Entropy



(b) Upload a file for increasin the Entropy

Figure 19: Task 21 Observation

Q: Launching a DoS Attack:

Exhaust Entropy Pool: Flood the server with requests that require random numbers from `/dev/random`. This depletes the entropy pool faster than it can be replenished, causing `/dev/random` to block.

Prevent Entropy Refilling: Avoid activities that generate entropy, like mouse movements, keyboard input, or disk I/O. This prevents the server from replenishing the entropy pool and resuming normal operations.

Impact: The server will become unresponsive to new requests that require random numbers, effectively denying service.

Prevention: Use `/dev/urandom` for non-critical randomness: `/dev/urandom` doesn't block and is suitable for most applications. Consider hardware RNGs: Hardware random number generators provide a more resilient source of entropy. Implement rate limiting: Restrict the rate of requests that use `/dev/random` to mitigate depletion attacks. Monitor entropy levels: Set up alerts for low entropy conditions to allow for proactive measures.

6.5 Task 22: Get Random Numbers from `/dev/urandom`

Behavior of `/dev/urandom`:

No blocking: Unlike `/dev/random`, `/dev/urandom` won't block, even when the entropy pool is low. It

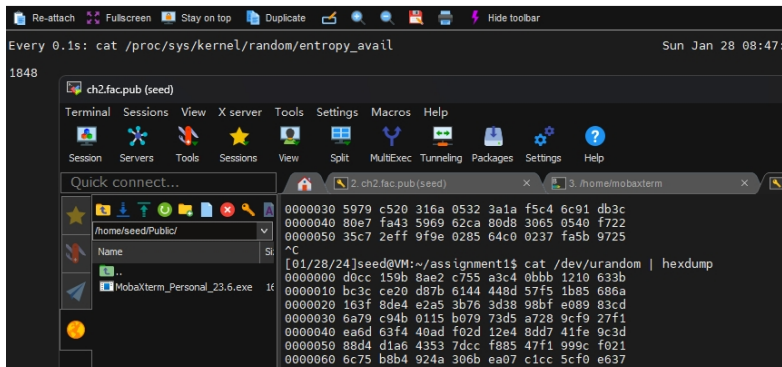


Figure 20: Not block random number generating by /dev/urandom

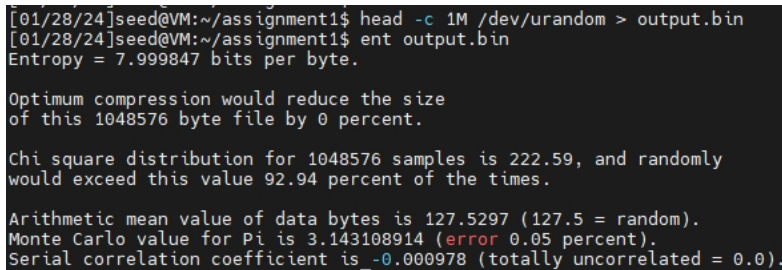


Figure 21: Measurement of the random number generating by /dev/urandom

will continue generating pseudorandom numbers using the available seed.

Network workload: won't have a noticeable effect on the output of `cat /dev/urandom | hexdump`. This is because `/dev/urandom` doesn't directly rely on real-time entropy input.

Quality of Random Numbers: Testing with `ent`: The `ent` tool evaluates randomness based on statistical tests. Good results typically indicate: Entropy estimates close to 8 bits per byte. Passing most or all statistical tests. No significant patterns or correlations in the data.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #define KEY_LEN 32 // 256 bits
4 int main() {
5     unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * KEY_LEN);
6     FILE *random = fopen("/dev/urandom", "r");
7     if (random == NULL) {
8         perror("Error opening /dev/urandom");
9         return 1;
10    }
11    fread(key, sizeof(unsigned char) * KEY_LEN, 1, random);
12    fclose(random);
13    // Print the generated key (replace with secure handling for actual use):
14    printf("Generated 256-bit encryption key: ");
15    for (int i = 0; i < KEY_LEN; i++) {
16        printf("%02x", key[i]);
17    }
18    printf("\n");
19    free(key);
20    return 0;
21 }

```

Listing 18: C Program Code for Generating Keys by /dev/urandom

This code generates a 256-bit encryption key using `/dev/urandom`. It allocates memory for the key, opens `/dev/urandom`, reads random bytes into the key, prints the key in hexadecimal format, and then frees the allocated memory. Compile and run this code to obtain your 256-bit encryption key.

```
[01/28/24]seed@VM:~/.../task22$ gcc main.c -o main
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 0e21b839ec74024298fe257960a275c330135a32b84d77b7dceadd16f117e4b8
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 1f5f073ac04245b6ca2d7a60d8bd6cec8fe588baebba7f1c2125b29efaf3df51
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 8804308c13342292a5b33da41167018078217d3034fc031e09aa5aabb6d41b7d
[01/28/24]seed@VM:~/.../task22$ █
```

Figure 22: Generating Key by /dev/urandom