

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	Task 3: Encryption Mode - ECB vs. CBC	2
2.4	Task 4: Padding	3
2.5	Task 5: Error Propagation - Corrupted Cipher Text	3
2.6	Task 6: Initial Vector (IV)	5
2.7	Task 7: Programming using the Crypto Library	6
3	MD5 Collision Attack	7
3.1	Task 8: Generating Two Different Files with the Same MD5 Hash	7
3.2	Task 9: Understanding MD5's Property	7
3.3	Task 10: Generating Two Executable Files with the Same MD5 Hash	7
3.4	Task 11: Making the Two Programs Behave Differently	8
4	RSA Public-Key Encryption and Signature	8
4.1	BIGNUM APIs	8
4.2	A Complete Example	8
4.3	Task 12: Deriving the Private Key	8
4.4	Task 13: Encrypting a Message	10
4.5	Task 14: Decrypting a Message	11
4.6	Task 15: Signing a Message	11
4.7	Task 16: Verifying a Message	12
4.8	Task 17: Manually Verifying an X.509 Certificate	12
5	Pseudo Random Number Generation	13
5.1	Task 18: Generate Encryption Key in a Wrong Way	13
5.2	Task 19: Guessing the Key	14
5.3	Task 20: Measure the Entropy of Kernel	15
5.4	Task 21: Get Pseudo Random Numbers from /dev/random	15
5.5	Task 22: Get Random Numbers from /dev/urandom	16

1 Introduction

This is the solvement 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, I mainly conducted decryption tests through the statistical frequency of the main letters. After multiple rounds of letter deduction, as shown in Figure 1.

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

Figure 1: Execute Result

```
[03/01/24]seed@VM:~$ echo $foo  
tet string  
[03/01/24]seed@VM:~$
```

Figure 2: Execute Result

2.2 Environment variable and Set-UID Programs

```
1 $ openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_aes128cbc.bin -K  
   00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-128-cbc encryption  
2 $ openssl enc -aes-256-cbc -e -in plaintext.txt -out cipher_aes256cbc.bin -K  
   00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-256-cbc encryption  
3 $ openssl enc -bf-cbc -e -in plaintext.txt -out cipher_blowfishcbc.bin -K  
   00112233445566778889aabbccddeeff -iv 0102030405060708 # blowfish encryption  
4 $ openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher_aes128cfb.bin -K  
   00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-128-cfb encryption
```

Listing 1: TASK 2.2 Command Line

As shown in listing 1, I try to use 4 different encryption ciphers by openssl to encrypt the plaintext generated in Sec 2.1.

2.3 Task 3: Encryption Mode - ECB vs. CBC

I conduct the following command 2 to encrypt the original images to ECB and CBC respectively. It's not hard to seen that the ECB mode encrypted image have a characteristics of the original image, while the CBC doesn't, as shown in Figure 4.

```
1 $ openssl enc -aes-128-ecb -e -in pic_original.bmp -out encrypted_ecb.bmp -K  
   00112233445566778889aabbccddeeff # ECB Mode  
2 $ openssl enc -aes-128-cbc -e -in pic_original.bmp -out encrypted_cbc.bmp -K  
   00112233445566778889aabbccddeeff -iv 0102030405060708 # CBC Mode
```

Listing 2: TASK 2.3 Command Lines

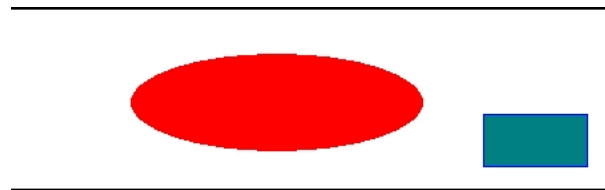


Figure 3: Original image

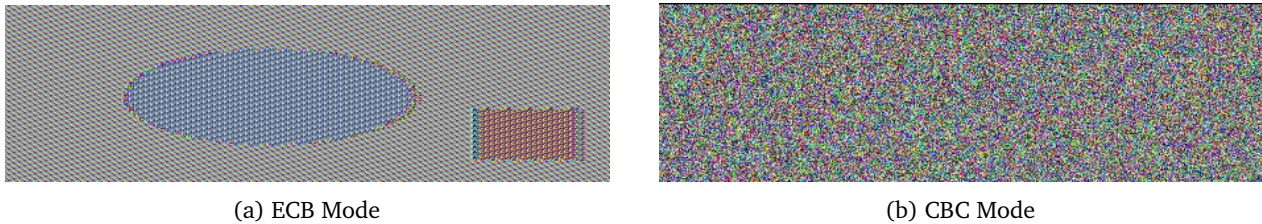


Figure 4: Encrypted images

2.4 Task 4: Padding

- **ECB, CBC:** Ciphertext is divided into blocks, so padding is needed.
 - **CFB, OFB:** Operate on smaller units (e.g., 8 bits), so padding isn't strictly required. They can still apply padding for compatibility or security reasons.
- The following listing shows the results.

```

1 # generate small files
2 $ echo -n "12345" > f1.txt
3 $ echo -n "1234567890" > f2.txt
4 $ echo -n "1234567890123456" > f3.txt
5
6 # encrypt the small files by aes-128-cbc
7 $ openssl enc -aes-128-cbc -e -in f1.txt -out f1_encrypted_cbc.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
8 $ openssl enc -aes-128-cbc -e -in f2.txt -out f2_encrypted_cbc.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
9 $ openssl enc -aes-128-cbc -e -in f3.txt -out f3_encrypted_cbc.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
10
11 # decrypt the small files by aes-128-cbc with -nopad
12 $ openssl enc -aes-128-cbc -d -nopad -in f1_encrypted_cbc.bin -out f1_decrypted_cbc.bin
   -K 00112233445566778889aabbccddeeff -iv 0102030405060708
13 $ openssl enc -aes-128-cbc -d -nopad -in f2_encrypted_cbc.bin -out f2_decrypted_cbc.bin
   -K 00112233445566778889aabbccddeeff -iv 0102030405060708
14 $ openssl enc -aes-128-cbc -d -nopad -in f3_encrypted_cbc.bin -out f3_decrypted_cbc.bin
   -K 00112233445566778889aabbccddeeff -iv 0102030405060708
15
16 # observe the results
17 $ hexdump -C f3_decrypted_cbc.bin
18 00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
19 00000010 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 |.....|
20 00000020
21 $ hexdump -C f3.txt
22 00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
23 00000010
24 $ hexdump -C f1.txt
25 00000000 31 32 33 34 35                                |12345|
26 00000005
27 $ hexdump -C f1_decrypted_cbc.bin
28 00000000 31 32 33 34 35 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b |12345.....|

```

Listing 3: TASK 2.4 Command Lines

2.5 Task 5: Error Propagation - Corrupted Cipher Text

Expected Behavior:

- In ECB mode, each block is encrypted independently of other blocks. A single bit error in one block should only affect that block during decryption, not propagating errors to subsequent blocks.

- CBC mode XORs each plaintext block with the ciphertext block from the previous iteration. A single bit error in one block will result in unpredictable errors in both the corresponding block and the following blocks during decryption.
- CFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.
- Similar to CFB, OFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.

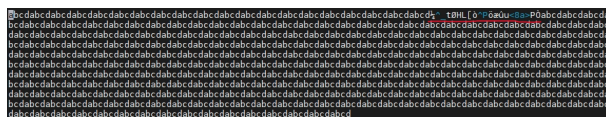
Conduct Experiments:

```

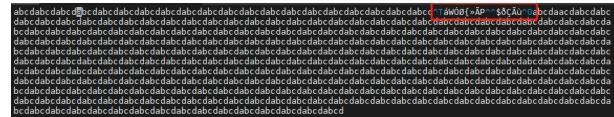
1 # generate a 1.2k bytes abcd repeating textual file
2 $ echo -n $(printf 'abcd%.0s' {1..300}) > plaintext.txt
3
4 # encrypt the file by different mode
5 $ openssl enc -aes-128-ecb -e -in plaintext.txt -out encrypted_aes_ecb.bin -K
   00112233445566778889aabbccddeeff
6 $ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_aes_cbc.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
7 $ openssl enc -aes-128-cfb -e -in plaintext.txt -out encrypted_aes_cfb.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
8 $ openssl enc -aes-128-ofb -e -in plaintext.txt -out encrypted_aes_ofb.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
9
10 # manually edit the bit of the 55th byte
11 $ bless encrypted_aes_ecb.bin # manually edit the bit of the 55th byte
12 $ bless encrypted_aes_cbc.bin # manually edit the bit of the 55th byte
13 $ bless encrypted_aes_cfb.bin # manually edit the bit of the 55th byte
14 $ bless encrypted_aes_ofb.bin # manually edit the bit of the 55th byte
15
16 # decrypt the file by different mode
17 $ openssl enc -aes-128-ecb -d -in encrypted_aes_ecb.bin -out decrypted_ecb.txt -K
   00112233445566778889aabbccddeeff
18 $ openssl enc -aes-128-cbc -d -in encrypted_aes_cbc.bin -out decrypted_cbc.txt -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
19 $ openssl enc -aes-128-cfb -d -in encrypted_aes_cfb.bin -out decrypted_cfb.txt -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
20 $ openssl enc -aes-128-ofb -d -in encrypted_aes_ofb.bin -out decrypted_ofb.txt -K
   00112233445566778889aabbccddeeff -iv 0102030405060708

```

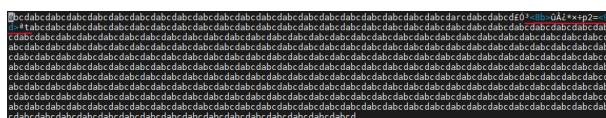
Listing 4: TASK 2.5 Command Lines



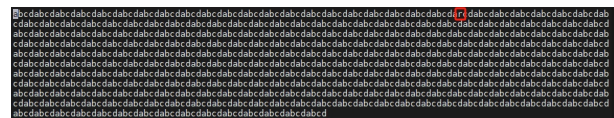
(a) ECB Mode



(b) CBC Mode



(c) CFB Mode



(d) OFB Mode

Figure 5: Error Propagation Observation

Justification:

- In ECB mode, As expected, errors in ECB mode do not propagate to subsequent blocks, and the corrupted block is isolated.
- Errors in CBC mode propagate to subsequent blocks due to the XOR operation with the previous ciphertext block.
- CFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.
- CFB and OFB: Errors in both CFB and OFB modes do not propagate to subsequent blocks, as they operate on the output of the encryption operation.

2.6 Task 6: Initial Vector (IV)

• Task 6.1

```
1 # generate a plain text
2 $ echo -n "1234567890123456123123" > plaintext.txt
3 # Encrypting with Two Different IVs:
4 $ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_iv1.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
5 $ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_iv2.bin -K
   00112233445566778889aabbccddeeff -iv 1122334455667788
6 # Encrypting with the Same IV:
7 $ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_same_iv.bin -K
   00112233445566778889aabbccddeeff -iv 0102030405060708
8 # Observations:
9 $ hexdump -C encrypted_iv1.bin
10 00000000 ec fb 05 81 98 77 71 c9 46 02 98 ba 32 85 a2 c2 |.....wq.F...2...|
11 00000010 3a 5a 5c 7f ac 22 10 62 b6 a8 6b 86 52 da 3a 24 |:Z\..".b..k.R.:$|
12 00000020
13 $ hexdump -C encrypted_iv2.bin
14 00000000 bb cf 5b 59 ae 45 65 1d 00 d5 9c ba ec 4e 13 9e |..[Y.Ee.....N..|
15 00000010 ef 6d 53 d9 63 8d 0b aa 48 ae 20 28 b9 2c 45 f8 |.mS.c...H. (.,E.|
16 00000020
17 $ hexdump -C encrypted_same_iv.bin
18 00000000 ec fb 05 81 98 77 71 c9 46 02 98 ba 32 85 a2 c2 |.....wq.F...2...|
19 00000010 3a 5a 5c 7f ac 22 10 62 b6 a8 6b 86 52 da 3a 24 |:Z\..".b..k.R.:$|
```

Listing 5: TASK 2.6.1 Command Lines

In the first case, where different IVs are used: encrypted_iv1.bin and encrypted_iv2.bin will be different. In the second case, where the same IV is used: encrypted_same_iv.bin will be the same as encrypted_iv1.bin because the same IV was used.

When different IVs are used, the resulting ciphertexts are different, even for the same plaintext and key. This is crucial for security because it ensures that patterns in the plaintext are not easily observable in the ciphertext.

If the same IV is reused, an observer might notice patterns or repetitions in the ciphertext, which can lead to vulnerabilities and compromise the security of the encryption. Using unique IVs helps in preventing such patterns and adds an additional layer of security.

• Task 6.2

- (1) Yes, they can potentially decrypt all subsequent messages encrypted with the same key and IV.
- (2) Yes, I can likely decrypt P2 using a known-plaintext attack.

$$\begin{aligned} output_stream &= C1 \oplus P1 \\ P2 &= C2 \oplus output_stream \end{aligned} \quad (1)$$

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main(int argc, char *argv[]) {
5     // Check arguments
6     if (argc != 4) {
7         printf("Usage: %s <plaintext> <ciphertext1> <ciphertext2>\n", argv[0]);
8         return 1;
9     }
10    // Convert arguments to byte arrays
11    size_t len1 = strlen(argv[1]);
12    unsigned char plaintext[len1 + 1];
13    memcpy(plaintext, argv[1], len1);
14    plaintext[len1] = '\0';
15    size_t len2 = strlen(argv[2]);
16    if (len2 % 2 != 0) {
17        printf("Error: ciphertext1 must be in hexadecimal format\n");
18        return 1;
19    }
20    size_t ciphertext1_len = len2 / 2;
21    unsigned char ciphertext1[ciphertext1_len];
22    for (int i = 0; i < ciphertext1_len; i++) {
23        sscanf(argv[2] + 2 * i, "%2hhx", &ciphertext1[i]);
24    }
25    size_t len3 = strlen(argv[3]);
26    if (len3 % 2 != 0) {
```

```

27     printf("Error: ciphertext2 must be in hexadecimal format\n");
28     return 1;
29 }
30 size_t ciphertext2_len = len3 / 2;
31 if (ciphertext1_len != ciphertext2_len) {
32     printf("Error: ciphertexts must be of equal length\n");
33     return 1;
34 }
35 unsigned char ciphertext2[ciphertext2_len];
36 for (int i = 0; i < ciphertext2_len; i++) {
37     sscanf(argv[3] + 2 * i, "%2hhx", &ciphertext2[i]);
38 }
39 // Perform XOR operation and decode
40 unsigned char recovered_text[ciphertext1_len];
41 for (int i = 0; i < ciphertext1_len; i++) {
42     recovered_text[i] = plaintext[i] ^ ciphertext1[i] ^ ciphertext2[i];
43 }
44 // Print recovered plaintext
45 printf("Recovered plaintext: %s\n", recovered_text);
46 return 0;
47 }

```

Listing 6: known-plaintext-attack in C

After execute the program by `./attack P1 C1 C2`, the result is Recovered plaintext: "Order: Launch a missile!"

(3) Only the first block of P2 can be confidently revealed in CFB mode.

- **Task 6.3** Here's how we can exploit predictable IVs to determine the content of P1:

Construct P2:

Eve creates a message P2 identical to P1 except for the first character: If she suspects P1 is "Yes", P2 will be "Xes". If she suspects P1 is "No", P2 will be "Wo".

Request Encryption:

Eve sends P2 to Bob for encryption. Bob, unaware of the attack, encrypts P2 using AES-128 in CBC mode with the predictable IV "1234567890123457".

Analyze Ciphertext C2:

Bob returns the ciphertext C2 to Eve.

Eve examines the first block of C2:

If the first block is "bef65565572ccee2", it strongly suggests P1 was "Yes". If the first block is anything else, it indicates P1 was likely "No".

2.7 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  plaintextbits = 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(plaintextbits, 16))
21         if cipherdatabits == testbits:
22             print("find the key:", key)
23             exit(0)
24
25 print("cannot find the key!")

```

Listing 7: 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”.

3 MD5 Collision Attack

3.1 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...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 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 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 25 c9 5e |...x.9...Ga...%|^|
00000070 f5 9f 4c b2 82 9b 6c 81 1a ad 8b 78 61 a2 00 f8 |..L...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 6: 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.

3.2 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 8: 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.

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


```

Sharing violation on path /home/seed/.config/bless/preferences.
Document does not have a root element.
[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 8: 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...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAB.....

```

```

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

```

```

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

```

```

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

```

(a) same arrays in test1

(b) different arrays in test2

Figure 9: test program binary

```

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 11: C Program Code for Deriving the Private Key

```

1 $ gcc main.c -lcrypto -o main
2 $ ./main

```

```
[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 10: BIGNUM API Example

```
3 Private Key (d): 9B740D556F9080815E14B6633E9BCC3C87EAA0F0AD699E0A7E0719A725A94AA7
```

Listing 12: Result of the Task 12

4.4 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[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5";
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 13: 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[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5";
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 14: 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 15: Result of the Task 13

4.5 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
8     COF971DF2F3672B28811407E2DABBE1DA0FEBBDFC7DCB67396567EA1E2493F";
9     // Convert private key values to BIGNUM
10    //...
11    // Free allocated memory
12    return 0;
13 }

```

Listing 16: 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 17: Result of the Task 14

4.6 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 18: 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, "49206f776520796f75202433030302e");
22 BN_mod_exp(signature, m2, d, n, BN_CTX_new());
23 //...
24 // Free BIGNUMs
25 return 0;
26 }

```

Listing 19: C Program Code for Signature

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

4.7 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, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
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 20: 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 11: Verification Result

Figure 11 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.

4.8 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 {
6     char bodyhash[] = "BBC2A75949C896BD66DB4E...302D8D02C99104AB04F4DD6"; # the Body HASH
7     BIGNUM *n = BN_new();

```

```
[01/28/24]seed@VM:~/.../task17$ openssl x509 -in c1.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDB90A20EDDFE3C4D09E9AA97A1D8288E51156
DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA25037B9A52416610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED5751
7426E27DC7CA622E131B7F238825536FC13458008B84FF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED37E236B4824862B5499AEC
C767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329AE
FB494FA415B9CEF25C19576D6B79A72BA2272013B5D03D40D321300793EA99F5
```

(a) Modulus

```
bb:b4:8e:27:87:27:c5:2b:74:d4:a8:d6:97:de:c3:
64:f9:ce:ce:53:a2:56:bc:78:17:8e:49:03:29:ae:
fb:49:4f:a4:15:b2:ce:f2:5c:19:57:ed:8b: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:AB:AA:B4:8C:79:EA:B4:DA:0F:98:B2:C5:95:76:B9:F4
X509v3 Authority Key Identifier:
keyid:03:DE:150:35:50:D1:4C:BB:66:F0:A3:E2:1B:1B:C3:97:B2:30:D1:55
```

(b) Exponent

```
[01/28/24]seed@VM:~/.../task17$ cat signature | tr -d '[:space:]'
59e44ad8a982ba9a4af1630c6d762675b33c74bec5f73da79192f8cf062d5810edf3b8d6fc6cff139632cd4fe98724850b74a2c2f60ff5a7d8
7d768aaee9c9582b6e006fb9cd24ec442c54c16859d34613923bfc68e95c984a9b2e5410f4478d795b9cfd974bf584fe716ff7c4030c46c4e
224dcb83673a93bf2bc5c59c1af243a1253b84f6f7536ea885aede14749130060df207d4c408ba4364c5e23fdaacc541afa437e8427674f713
bb4a7d3659819bc744df8973b93342e860c24d615d125a10f6eff33891450e8d69fc6b95c2b35dbadeddd36b625f2958aac693f9afe1af815
286dea185ac2d26218af4078b5fa5e98f53f9ccf823a1833123f4c6[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 12: Task 17 Information

```
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 21: C Program Code for Manually Verifying

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

Figure 13: Verification Result

5 Pseudo Random Number Generation

5.1 Task 18: Generate Encryption Key in a Wrong Way

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


```

[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 14: Result Observation

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.

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

```

Listing 22: 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 23: 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 15: Result

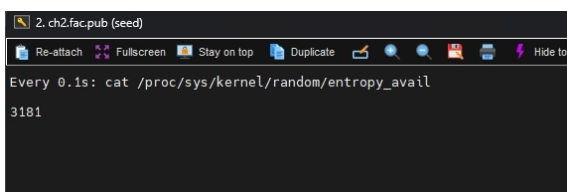
And finally find the key: 95fa2030e73ed3f8da761b4eb805dfd7

5.3 Task 20: Measure the Entropy of Kernel

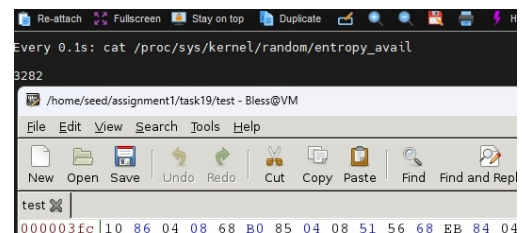
Since my Seed lab is running in a remote VM, I try to measure the entropy by such:

- 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.



(a) Background Entropy



(b) Activity Entropy

Figure 16: Task 20 Observation

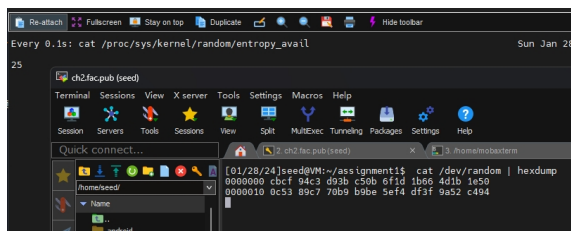
5.4 Task 21: Get Pseudo Random Numbers from /dev/random

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 17(b),

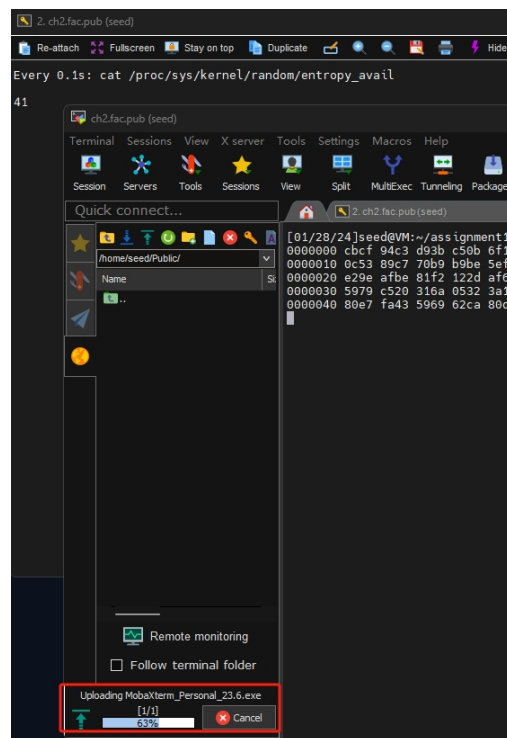
Q: Launching a DoS Attack:

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.



(a) Background Entropy



(b) Upload a file for increase the Entropy

Figure 17: Task 21 Observation

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.

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

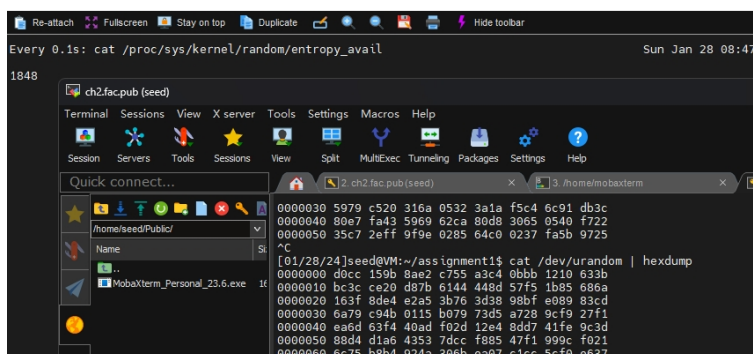


Figure 18: Not block random number generating by `/dev/urandom`

Behavior of `/dev/urandom`:

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

```
[01/28/24]seed@VM:~/assignment1$ head -c 1M /dev/urandom > output.bin
[01/28/24]seed@VM:~/assignment1$ ent output.bin
Entropy = 7.999847 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 222.59, and randomly
would exceed this value 92.94 percent of the times.

Arithmetic mean value of data bytes is 127.5297 (127.5 = random).
Monte Carlo value for Pi is 3.143108914 (error 0.05 percent).
Serial correlation coefficient is -0.000978 (totally uncorrelated = 0.0).
```

Figure 19: Measurement of the random number generating by /dev/urandom

```
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 24: C Program Code for Generating Keys by /dev/urandom

```
[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 20: Generating Key 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.