Pseudo-Random Number Generation Lab

Task 1: Generate Encryption Key in a Wrong Way

In this task I was experimenting with how PRNGs (pseudo-random number generators) operate. Since a computer program *can't* generate actual random numbers, we explore how they're considered *pseudo-random* numbers since they do not produce a pattern and cannot be predicted. They do this by using things such as the current time as a seed for the PRNG. Using the given code and executing it twice – once with the 'srand(time(NULL));' commented out (Figure 2) and once with the command uncommented (Figure 1), we can see the difference between the two pseudo-random numbers that are generated.

```
Terminal
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ gcc task1.c -o t1
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1
1570822921
3282b011621d5dd4687c59f1c2aaaf45
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1
1570822923
33c14bc458818a3a1213cf3281d3217f
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1
1570822930
2eafd25531ec556b1facbea84f09637b
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$
```

Figure 1

```
Terminal
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ gcc t1comment.c -o t1com
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1com
1570823013
67c6697351ff4aec29cdbaabf2fbe346
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1com
1570823014
67c6697351ff4aec29cdbaabf2fbe346
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ ./t1com
1570823015
67c6697351ff4aec29cdbaabf2fbe346
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$
```

Figure 2

As you can see, the program executed in figure 1 produces a completely different group of hex characters every time its executed since it uses the srand(time(NULL)); function. Comparitively, the program executed in figure 2 produces the same number every time.

Questions:

1) The top output value represents the system time and bottom number represents the random number that the program has generated.

2) Figure 1 shows the program using the time in the srand() function as the seed to produce random numbers. Since the time is constantly changing, you will never have the same random number generated.

3) Figure 2 shows what happens if the seed remains the same or isn't provided – the generated number will repeat. This is exactly what happened when we commented out the srand(time(NULL)) function which would use system time → time(NULL) as the seed.

Task 3: Measure the entropy of the Kernel

In this task we are exploring the different sources of randomness that programs may use to produce more unpredictability. Randomness is measured using something called entropy which is the number of bits of random numbers the system currently has. As shown in the screenshot below (Figure 3), we can use the command 'cat /proc/sys/kernel/random/entropy_avail' to display the entropy the kernel has available at that moment.

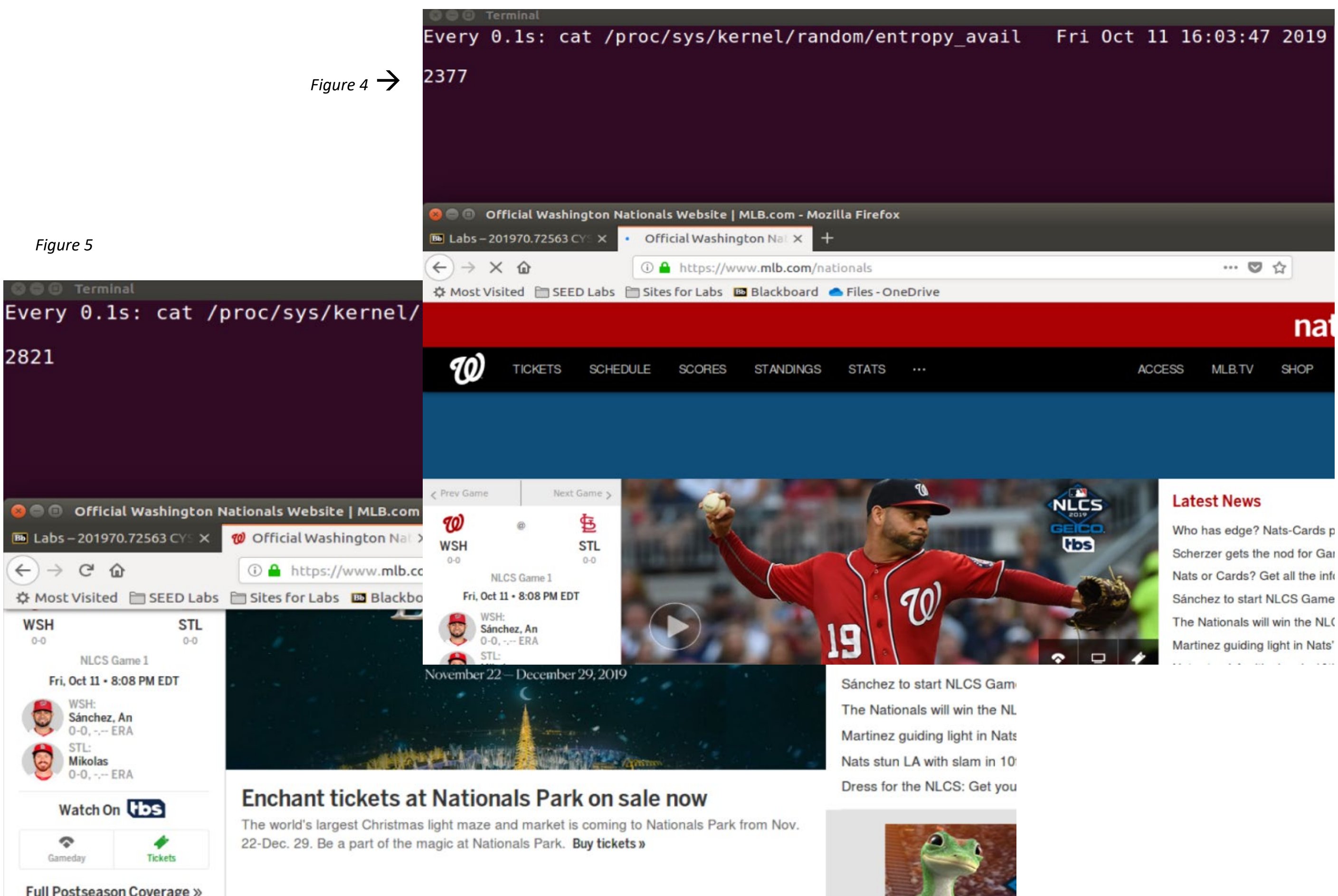
```
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ cat /proc/sys/kernel/random/entropy
_avail
1115
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$ cat /proc/sys/kernel/random/entropy
_avail
1127
[10/11/19]JCogswell@Attacker-131:~/.../PRNG$
```

Figure 3

Now, we can use the command 'watch -n .1 cat /proc/sys/kernel/random/entropy_avail' to run the same command every .1 seconds and show us how the entropy changes as we visit a website, use the keyboard, and click the mouse. Figure 4 shows the entropy after going to www.nationals.com and figure 5 shows the entropy after also clicking my mouse and keyboard buttons a lot. Take note of the increase in entropy as a result of using my keyboard and mouse as well as visiting a website.

Figure 4 →

Figure 5



Questions:

1) The entropy value increases by a lot the more I do things like use my mouse and keyboard, visit websites, and read a large file. Visiting several websites will increase the entropy since Linux gains randomness from these types of physical resources.
2) Other ways to generate and increase entropy were listed above, but for clarity, other ways can be reading a large file, typing, moving your mouse, and clicking your mouse.

Task 4: Get Pseudo Random Numbers from /dev/random

This task demonstrates how Linux uses two devices to turn randomness into pseudo random numbers. The device we're focusing on in T4 is /dev/random. In Figure 6 you can see two terminals. The top one is doing the same thing as task 3 – watching the entropy levels and displaying it to the screen every .1 seconds. What this is and does will be explained in the answers to the questions below.

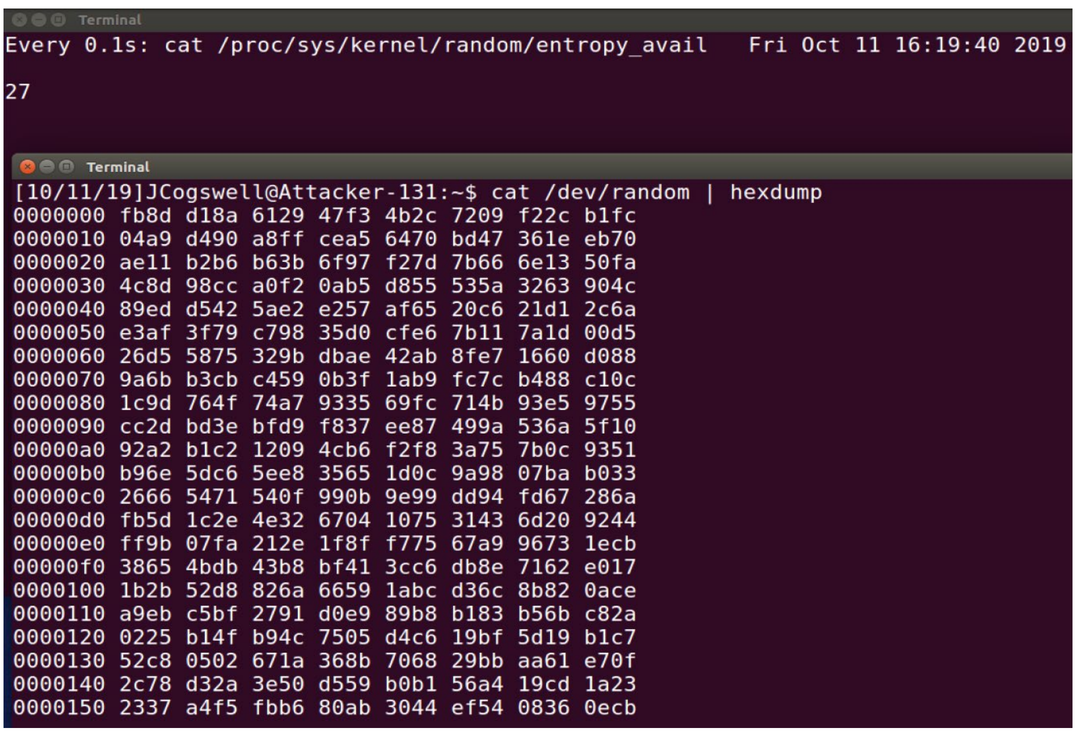


Figure 6

Questions

1) /dev/random is a blocking device so everytime a random number is given out by this device, the amount of entropy will decrease. When there is no more entropy, /dev/random will block and no more pseudo-random numbers will be generated until there is enough randomness available.

From what I could tell, the /dev/random blocks until it has 64 bits of entropy available because that's just about where the number resets back to zero every time.

2) If a server uses /dev/random to generate the random session key with a client, you can launch a DOS attack on that server. If you attempt to establish a lot of sessions between you and the server over and over, then the entropy pool will eventually be exhausted and will no longer be able to establish a session with anyone during the attack. This will result in denial of service.

3) It is preferred to use /dev/urandom over /dev/random since /dev/urandom does not block. Since /dev/urandom doesn't block, it isn't susceptible to DOS attacks. /dev/urandom will continue generating new numbers and will use whatever entropy is available in the pool as the seed and does not require a specific amount like /dev/random does.

RSA Public Key Encryption Lab

Task 1: Deriving the Private Key

In this task I derive the private key (d) given p, q, and e, each of these being prime numbers that are large, but much smaller than necessary so that the task is simpler. These numbers are 128 bits but in practice, these numbers will need to be at least 512 bits long. The public key we will use is (e, n).

p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3

Using the provided code, the decrypted key (d) is computed and displayed to the screen, as shown in Figure 7.

```
Terminal
[10/11/19]JCogswell@Attacker-131:~/.../PKC$ ./task1
d =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[10/11/19]JCogswell@Attacker-131:~/.../PKC$
```

Figure 7

Questions

1) The value of d can be verified by either using the program as described above or by calculating it using the following algorithm (from the Internet Security textbook):
To decrypt ciphertext C, use the modular exponentiation using d and n.

$$M = C^d \ mod \ n$$

Now we need to find out if $C^d \ mod \ n$ can get back M or not, so we need to prove that

$$M^{ed} \ mod \ n = M$$

Since we know that the public key exponent *e* and the private key exponent *d* are related, we can get rid of the modulo operator:

$$ed = k\emptyset(n) + 1$$

So,

$$M^{ed} \, mod \, n = M^{k\emptyset(n)+1} \, mod \, n = \; M^{k\emptyset(n)} * M \, mod \, n$$
$$= \left(M^{\emptyset(n)} \, mod \, n\right)^k * \; M \, mod \, n \quad \leftarrow \text{distributive rule}$$
$$= 1^k * M \, mod \, n \leftarrow \text{Euler's theorem}$$
$$= M$$

Now, this method will prove that *d* is the correct key if we get the correct value of M. If we don't, then we do not have the correct key.

2) The BIGNUM API is one of several libraries that can perform arithmetic operations on integers of arbitrary size (as opposed to primitive data types with specific sizes). We need to use this for large numbers because we can't use simple arithmetic operators on very large numbers in programs. The necessary operators can only operate on primitive data types, like 32-bit and 64-bit long integers.

<u>Task 2: Encrypting a Message</u>

In this task I will use a public key (e, n) to encrypt an ASCII message "A top secret!" into a hex string, and then convert the hex string into a BIGNUM using the hex-to-binary API BN_hex2bn(). We will use the python command "python -c 'print("A top secret!".encode("hex"))' " which will return the value "4120746f782073616372657421". We are given:

n= DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e= 010001 (this hex value equals to decimal 65537)
M= A top secret!
d = 74D806F9F3A62BAE331FFE3FOA68AFE35B3D2E4794148AACBC26AA381C07D30D

Figure 8 shows the encrypted message and the decrypted message after executing the given program.

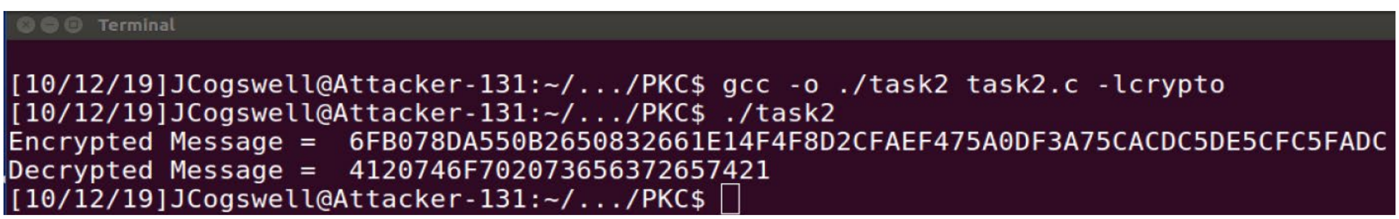


Figure 8

Questions

1) If I'm understanding this question correctly, I'm supposed to provide the hex value for the ASCII character "space" that appears in the plaintext message. The *encoded* (not encrypted) hex values for the spaces of the text message are 20.
41**20**746f78**20**73616372657421

2) The reason the encrypted message is not the same size as the original message is because RSA encryption uses padding in its proccess, although not directly. According to the Internet Security textbook, we don't directly encrypt a long plaintext using RSA. We instead use a hybrid approach bu using a content key and a secret-key incryption algorth to encrypt the plaintext, and then use RSA to encrypt the content key. Therefore, for large numbers, we use the content key as a number and raise it to the power of *e mod n*, which will generate a different-sized encrypted message compared to the original.

Task 3: Decrypting a Message:

In this task, we will use the same public and private keys from Task 2 to decrypt a given ciphertext *C*, and then convert it back to a plain ASCII string. We will use the python command "$ python -c 'print("50617373776F72642069732064656573".encode("hex"))' " to convert the hex string back to a plain ASCII string.

Given: *C* = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F

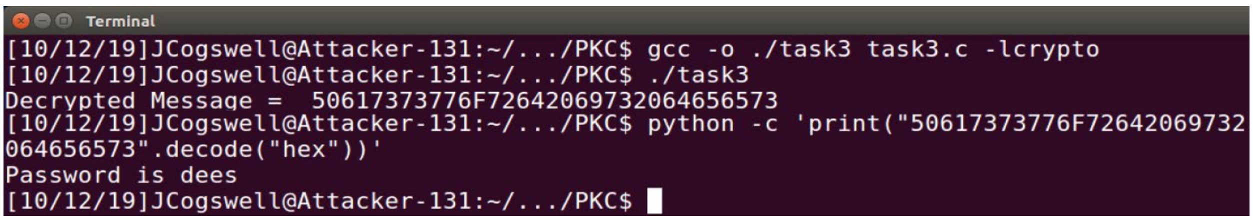


Figure 9

In Figure 9 shown above, we can see the decrypted message that the program calculated as well as the decoded hex to produce the ASCII string.

Questions

1) The password is **dees**.