Network and Telecommunications Systems Security

Name: Evangelos Siatiras
AM: EN2190001

Ex_1_i

In order to Create the file with size of 16 bytes and remove the newline character I used the following commands.

```
vim name.txt #Create and fill with vaggelissiatiras
bless name.txt #remove the newline character
ls -al name.txt #for evaluation of the size
```

The encryption key is in the file “key.bin” located in /security folder.By doing:

```
bless key.bin #Encryption Key: security_di
```

We get the 11 bytes encryption key.

The commands used in order to encrypt with AES-256 with the ECB,CBC and CTR modes of operation respectively are the following:

For Encrypt with ECB mode of operation and output of ciphertext to encrypted_ecb.bin:

```
openssl enc -aes-256-ecb -in name.txt -out encrypted_ecb.bin -nosalt -nopad -kfile
../key.bin
```

For Encrypt with CBC mode of operation and output of ciphertext to encrypted_cbc.bin:

```
openssl enc -aes-256-cbc -in name.txt -out encrypted_cbc.bin -nosalt -nopad -kfile
../key.bin -iv 00000000000000000000000000000000
```

For Encrypt with CTR mode of operation and output of ciphertext to encrypted_ctr.bin:

```
openssl enc -aes-256-ctr -in name.txt -out encrypted_ctr.bin -nosalt -nopad -kfile
../key.bin -iv 00000000000000000000000000000000
```

Note that, CBC and CTR mode of operations require and Initialization Vector (IV) whereas above is given one filled with Zero.

```
oot@ubuntu:~/security/ex_1_i# ll | grep -i bin
rw-r--r-- 1 root root   16 Jun 18 09:01 encrypted_cbc.bin
rw-r--r-- 1 root root   16 Jun 18 09:01 encrypted_ctr.bin
rw-r--r-- 1 root root   16 Jun 18 09:14 encrypted_ecb.bin
```

Ex_1_ii

By following the same procedure described above we create the file called repeated_name.txt and we evaluate it’s size. Also by following the same rational we encrypt it using AES-256 and the 3 different modes of operations and create the required ciphertext files with the following commands.

```
openssl enc -aes-256-ctr -in repeated_name.txt -out new_encrypted_ctr.bin -kfile
../key.bin -iv 00000000000000000000000000000000 -nosalt -nopad

openssl enc -aes-256-cbc -in repeated_name.txt -out new_encrypted_cbc.bin -kfile
../key.bin -iv 00000000000000000000000000000000 -nosalt -nopad

openssl enc -aes-256-ecb -in repeated_name.txt -out new_encrypted_ecb.bin -kfile
../key.bin  -nosalt -nopad
```

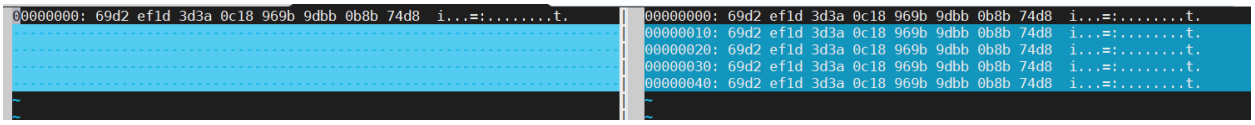
```
oot@ubuntu:~/security/ex_1_ii# ls -lrt | grep -i bin
rw-r--r-- 1 root root  80 Jun 18 09:07 new_encrypted_ctr.bin
rw-r--r-- 1 root root  80 Jun 18 09:07 new_encrypted_cbc.bin
rw-r--r-- 1 root root  80 Jun 18 09:08 new_encrypted_ecb.bin
```

The Resulting ciphertext from case i and case ii are binary files. In order to be able to compare them more efficiently we transforming them to their hex equivalents using the following commands:

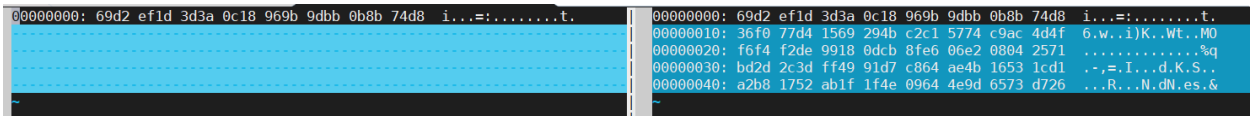
```
xxd encrypted_cbc.bin > encrypted_cbc.hex
xxd encrypted_ctr.bin > encrypted_ctr.hex
xxd encrypted_ecb.bin > encrypted_ecb.hex
xxd new_encrypted_cbc.bin > new_encrypted_cbc.hex
xxd new_encrypted_ctr.bin > new_encrypted_ctr.hex
xxd new_encrypted_ecb.bin > new_encrypted_ecb.hex
```

And we use vimdiff in order to perform diff between the resulted files per mode of operation as per below.

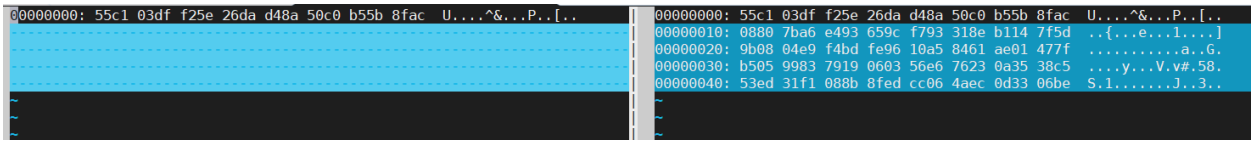
```
vimdiff  encrypted_ecb.hex ../ex_1_ii/new_encrypted_ecb.hex
```




```
vimdiff  encrypted_cbc.hex ../ex_1_ii/new_encrypted_cbc.hex
```



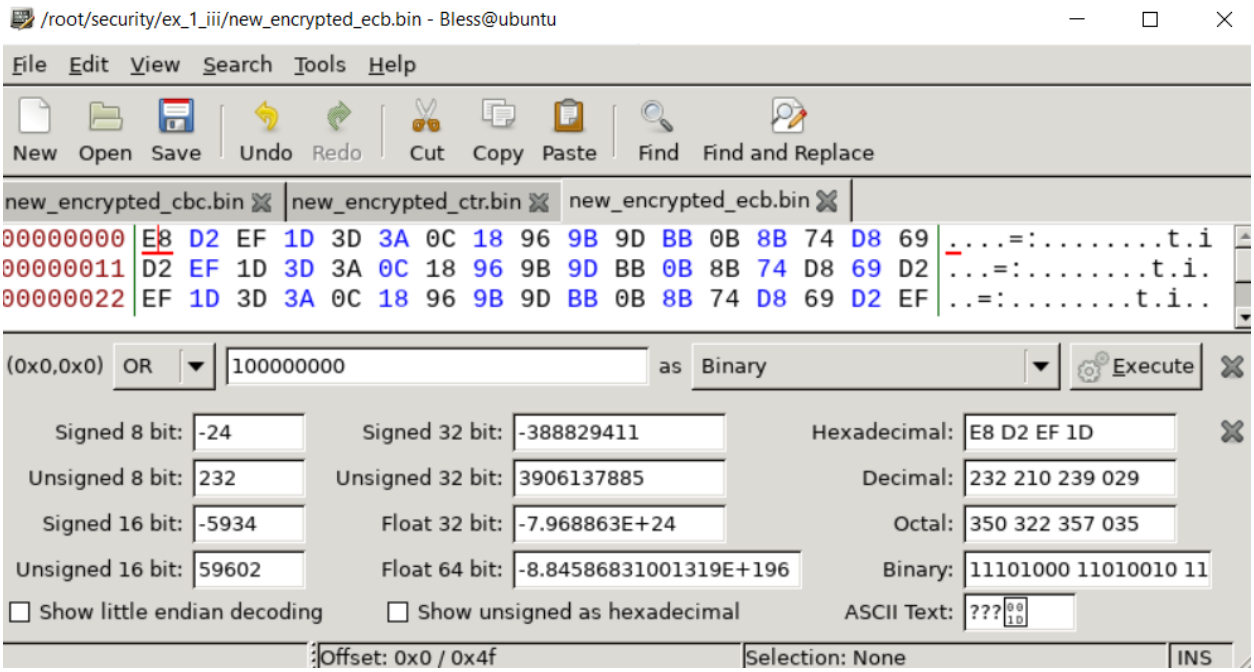

```
vimdiff  encrypted_ctr.hex ../ex_1_ii/new_encrypted_ctr.hex
```



At first in AES 256 encryption the block size is 128bit or 16 byte which is equal to the size of our initial plaintext in part i. Also these are the 16 repeating bytes in repeated_name.txt. Thus, without take into consideration the modes of operation in the encrypted*.hex files we can say that this is the result of AES-256 encryption. In the encrypted_cbc and encrypted_ecb the result is the same which is the result from encrypting one block of 16 bytes with AES Block Cipher without care about the mode of operation. Similarly, the result in encrypted_ctr is the one from encrypted with AES in Counter Mode. Generally, if we have to encrypt a number of bytes equal or less to the block size, we do not take into account the mode of operation. In the new_encrypted* files we have to encrypt a message in many blocks. **ECB** encrypts each block of data independently and the same plaintext block will result in the same ciphertext block. **CBC** has the above mentioned IV and thus inserts randomness every time a message is encrypted. Such randomness is evaluated from the ciphertext (above picture), after the first block. **CTR** mode makes block cipher way of working similar to a stream cipher. In this mode, subsequent values of an increasing counter are added to a *nonce* value (the nonce means a number that is unique: *number used once*) and the results are encrypted as usual.The nonce plays the same role as initialization vectors in the CBC mode. Similarly, the randomness is evaluated in the above picture.

Ex_1_iii

To modify the first bit of a ciphertext OR it with number 1 as per below for new_encrypted_ecb.bin and by the same way for the others.



To perform decryption with respect to mode of operation and produce the required plaintexts I use the following commands:

```
openssl enc -d  -aes-256-ctr -in new_encrypted_ctr.bin -out new_decrypted_ctr.bin -kfile
../key.bin  -nosalt -nopad -iv 00000000000000000000000000000000

openssl enc -d  -aes-256-cbc -in new_encrypted_cbc.bin -out new_decrypted_cbc.bin -kfile
../key.bin  -nosalt -nopad -iv 00000000000000000000000000000000

openssl enc -d  -aes-256-ecb -in new_encrypted_ecb.bin -out new_decrypted_ecb.bin -kfile
../key.bin  -nosalt -nopad
```

And the resulting plaintexts are:

CBC

```
2  ¢Æ~Þ@®[Á†KG▒▒FW÷aggelissiatirasvaggelissiatirasvaggelissiatirasvaggelissiatiras
```

If we take a closer look in the resulting plaintext there are two observations.

At first the resulting plaintext equal to the first block size (the first 16bytes) is fully affected and fully corrupted. Is the product of AES-256 decryption XORed with IV.

Regarding the second block we see that is affected only in the first byte corresponding to the modified byte place in the ciphertext.(First byte in the cyphertext from the first block, first byte in plaintext from the second block). This is evaluated from the fact that the ciphertext from the previous block (in our case the first block with one modified bit) is XORed with the decrypted product-plaintext from the second block.

$P_1 = Dec(C_1) \oplus IV$ and $P_i = Dec(C_i) \oplus C_{i-1}, 1 < i < N$ where N is the number of blocks

ECB

```
2  ¢Œ~Þ@®[ÁtKGBBIFWvaggelissiatirasvaggelissiatirasvaggelissiatirasvaggelissiatiras
```

As mentioned above **ECB** encrypts each block of data independently and the same plaintext block will result in the same ciphertext block. So the first block is corrupted due to the modification applied in the ciphertext but the subsequent ones remain intact.

CTR

```
+aggelissiatirasvaggelissiatirasvaggelissiatirasvaggelissiatirasvaggelissiatiras
```

CTR mode builds a pseudo-random keystream that is XORed against the message. That is for each bit M_i of the message: $C_i = M_i \oplus K_i$

If we XOR ciphertext bit C_i with some bit then the decryption procedure will result in:

$$M_i' = C_i' \oplus K_i = C_i \oplus bit \oplus K_i = M_i \oplus K_i \oplus bit \oplus K_i = M_i \oplus bit$$

So If one bit of a plaintext or ciphertext message is damaged, only one corresponding output bit is damaged as well evaluated the above result.

Ex_2

In the terms of this exercise I created a file named modified_message.txt and I diff them in order to be able to find out the differences. In the following picture is clear that only the last 4 bytes differ.

Then I create a copy of cyphertext1.bin named as modified_cyphertext1.bin with the command:

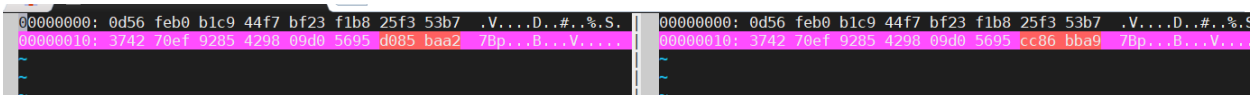
```
cp cyphertext1.bin modified_cyphertext1.bin
```

Now based on the assumption mentioned above that If one bit of a plaintext or ciphertext message is changed, only one corresponding output bit is changed and the changed bit is the same in order as the input. So our goal is to transform:

PlainText	CipherText
5 -> 1	d0 -> cc (d0 -4 = cc)
0 -> 1	85 -> 86 (85 +1 = 86)
8 -> 9	ba -> bb (ba+1 = bb)
1 -> 8	a2 -> a9 (a2 + 7 = a9)

Note that 1 byte in hex is a 2 digit representation. For the transformations I used bless.(Open the file and replace the required bytes).
Then to be able to diff them I create their hex equivalents.

```
xxd ciphertext1.bin > ciphertext1.hex
xxd modified_ciphertext1.bin > modified_ciphertext1.hex
vimdiff  ciphertext1.hex modified_ciphertext1.hex
```



Now as bob knows the secret key, when he will decrypt the modified_ciphertext1 he will get the modified bank account without being able to validate the integrity of the message and the authenticity of the message.
Alice can prevent this by using a message authentication system like HMAC or, even better, an authenticated encryption mode like GCM, which functions like CTR mode but also authenticates the data and thereby makes it impossible to change the ciphertext without being detected.

Ex_3_i

To create an RSA pair, with N = 3072, I executed the following commands:

Create the private key and store it to privatekey-mine.pem
```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:3072  -out privatekey-mine.pem
```
Create the public key from private key and store it to publickey-mine.pem (pubout is specified as we want the public key for the output)
```
openssl pkey -in privatekey-mine.pem -out publickey-mine.pem -pubout
```
Print Private Key
```
openssl pkey -in privatekey-mine.pem -text
```
```
RSA Private-Key: (3072 bit, 2 primes)
```

Print Public Key
```
openssl pkey -in publickey-mine.pem -pubin -text
```
```
RSA Public-Key: (3072 bit)
```

Ex_3_ii

With the following command we creating a random symmetric key of size 128 bits and save it in a file " key.bin". The bytes are generated by using linux urandom as input to rand, as we need random data used to seed the random number generator.
```
openssl rand -rand /dev/urandom 128 > key.bin
```

With the following command I encrypt with AES-CTR and zero IV the file name.txt using the key in the file key.bin and the produced ciphertext will be stored to encrypted_name.bin.
```
openssl enc -aes-256-ctr -nosalt -nopad -in name.txt -out encrypted_name.bin -pass file:key.bin -iv 00000000000000000000000000000000
```

Ex_3_iii

In order to appropriately encrypt key.bin such in a way that it will decrypted successfully only from the receiver, I will encrypt it using receivers public key and this is performed using rsautl as per the following command.

```
openssl rsautl -encrypt -inkey publickey-tutor.pem -pubin -in key.bin -out encrypted_key.bin
```

Then the encrypted_key.bin along with the encrypted_name.bin are sent to the receiver. Then he decrypts the encrypted_key.bin by using his private rsa key and then with the recovered key performs AES-CTR decryption to recover the plaintext.

Ex_4_i

According to the documentation referred the salt has the following format idsalt$encrypted where "id" identifies the encryption method."salt" stands for the up to 16 characters and the "encrypted" has a fixed string according to the hash function specified by the "id".In case of SHA-512 is 86 bytes long.In the shadow file we extract that the "id" equals to 6 in both passwords so the hash function has been used is SHA-512.

Ex_4_ii

Further to the password recovery I implemented the following program in C.

```
int
main(int argc, char* argv[])
{
    if (argc != 3)
    {
        printf("1st Arg should be the encrypted passwd from the shadowfile \n");
        printf("2st Arg should be the one possible password \n");
        return 1;
    }


    char *enc_passwd  = argv[1]; //Gets second value as hashed password
    char *toCrack = argv[2]; //Gets third value as hashed password
    char** tokens;

    printf("Encrypted Passwd: %s\n", enc_passwd);
    printf("Guessed Passwd:   %s\n", toCrack);

    char *password = malloc(sizeof(char*) * strlen(enc_passwd));

    memcpy(password,enc_passwd,strlen(enc_passwd));

    tokens = str_split(password, '$');

    assert(tokens && "Check Input Format");


    printf("id = %s salt = %s encrypted = %s \n", *(tokens),  *(tokens + 1),  *(tokens + 2));

     char salt [16];

     memset (salt , 0 ,strlen(salt));

    strcat (salt, "$");
    strcat (salt, *(tokens));
    strcat (salt, "$");
    strcat (salt, *(tokens + 1));
    strcat (salt, "$");
    printf("Salt = %s \n", salt);

    printf("\n");

    char *toCrackCiph = malloc(sizeof(char*) * strlen(enc_passwd));
    strcpy(toCrackCiph, crypt(toCrack, salt));

    printf("Encrypted Passwd:  %s\n", enc_passwd);
    printf("Guessed Enc Passwd:%s\n", toCrackCiph);

    if(strlen(toCrackCiph) != strlen(enc_passwd)) {

        printf("Warning Check Salt \n");
    }

    if ( 0 == strcmp(enc_passwd, toCrackCiph)){
        printf("Success Passwords are Equal \n");
    } else{
        printf("Warning Passwords are NOT Equal. Please reRun with another Guess\n");
    }

    return 0;
}
```

The program requires as arguments the salted hashed value of the password as exists in the shadow file followed by a password guess. Then it splits the salted hash value to the “id”,”salt” and “encrypted” and prints the result. Then it computes the salted hashed value of our guessed password and it compares it with the input. If both of them are equal, Success message is printed otherwise a warning is appearing as per below.

Let’s make one guess for Bob’s password. Let’s assume password is “Athens".

```
./a.out
'$6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.' 'Athens'
```

```
root@ubuntu:~/security/ex_4# ./a.out '$6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.' 'Athens'
Encrypted Passwd: $6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Guessed Passwd:   Athens
id = 6 salt = Dnm6ouc0NPBkd45h encrypted = uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Salt = $6$Dnm6ouc0NPBkd45h$

Encrypted Passwd:  $6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Guessed Enc Passwd:$6$Dnm6ouc0NPBkd45h$Upm9KVypZbt0Hp4Nr5S2kfZUeIaCPV15CSUuolIbdiHzlTLtu0JbQ5FBd9kh5ACmu86lKGHi6xgD80skuSYfb1
Warning Passwords are NOT Equal. Please reRun with another Guess
```

Now let's rerun by assuming that the password is "pinkfloyd"

```
./a.out
'$6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.' 'pinkfloyd'
```

```
root@ubuntu:~/security/ex_4# ./a.out '$6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.' 'pinkfloyd'
Encrypted Passwd: $6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Guessed Passwd:   pinkfloyd
id = 6 salt = Dnm6ouc0NPBkd45h encrypted = uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Salt = $6$Dnm6ouc0NPBkd45h$

Encrypted Passwd:  $6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Guessed Enc Passwd:$6$Dnm6ouc0NPBkd45h$uD343vhwOMWSgf1Coep8OqUqU5R4FwLzX2YQEbniNkWCfUL6.PYXt/o16blw41h/RIKmjxY/67JS4gb.cIqBy.
Success Passwords are Equal
```

And we get the corresponding Success message confirms that our guess is correct. The assumption for the correct guess is based on the comparison of the product of crypt function `toCrackCiph` and the salted hashed value `enc_passwd`.

```
char *toCrackCiph = malloc(sizeof(char*) * strlen(enc_passwd));
strcpy(toCrackCiph, crypt(toCrack, salt));

printf("Encrypted Passwd:  %s\n", enc_passwd);
printf("Guessed Enc Passwd:%s\n", toCrackCiph);

if(strlen(toCrackCiph) != strlen(enc_passwd)) {

    printf("Warning Check Salt \n");
}

if ( 0 == strcmp(enc_passwd, toCrackCiph)){
    printf("Success Passwords are Equal \n");
} else{
    printf("Warning Passwords are NOT Equal. Please reRun with another Guess\n");
}
```

Similarly, for Alice's password let's try with the guessed password "zka5231"

```
./a.out
'$6$QNy/phwS$Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0' 'zka5231'
```

```
root@ubuntu:~/security/ex_4# ./a.out '$6$QNy/phwS$Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0' 'zka5231'
Encrypted Passwd: $6$QNy/phwS$Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0
Guessed Passwd:   zka5231
id = 6 salt = QNy/phwS encrypted = Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0
Salt = $6$QNy/phwS$

Encrypted Passwd:  $6$QNy/phwS$Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0
Guessed Enc Passwd:$6$QNy/phwS$Le4wVYuK8PiBOBijX975KGa/dE3YpfdTPceV2KH2lSHYetCAYGct8q1P.62pO8UPbkFbLIY2Br9S1TSmVmucd0
Success Passwords are Equal
```

The corresponding message confirms that our Guess was Correct.

Ex_4_iii

In order to create the SHA-512 hash value of the two above passwords we will use the following online tool in the link.

The SHA-512 of "pinkfloyd" is :

6e490e8ba10659b2d267e56cdb0c382b77ea7012396110109dfdb86cf723db578a56690124d354c516e4194ff65c4795637242455137a57a08461b9271d59247

And the SHA-512 of "zka5231" is:

7469ed64b3dc48ebccba2d9b8b11ce44a34ee582ebcdd6a87836b7e7f376d6824302e38e7d436eaaf72abaeeb7a91f849f312c158312b26d6486c6ef4a47c1c6

If we insert the two above hash values to the required tool and we get the following result, respectively.

Hash	Type	Result
6e490e8ba10659b2d267e56cdb0c382b77ea7012396110109dfdb86cf723db57 8a56690124d354c516e4194ff65c4795637242455137a57a08461b9271d59247	sha512	pinkfloyd

Hash	Type	Result
7469ed64b3dc48ebccba2d9b8b11ce44a34ee582ebcdd6a87836b7e7f376d682 4302e38e7d436eaaf72abaeeb7a91f849f312c158312b26d6486c6ef4a47c1c6	Unknown	Not found.

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

In the second case when we requested from the tool to perform a rainbow attack using the above hash value for "zka5231" we get the result not found. It is extremely difficult to create and maintain a lookup table with hashes from values such as plate numbers. It costs a lot of resources because of their tremendous number of possible combinations. Therefore, a rainbow attack is not effective for such kind of passwords.

Ex_5

As a first step we create a public and a private key using the following command:

```
gpg --gen-key
```

Then let's print our newly created key using the command:

```
gpg --list-keys
```

The output is in the following picture.

```
/root/.gnupg/pubring.kbx
------------------------
pub   rsa3072 2020-06-25 [SC] [expires: 2022-06-25]
      3B25276CE331FADBB0AFBAD29E8362B45CFEBD4B
uid           [ultimate] Evangelos Siatiras <EN2190001@di.uoa.gr>
sub   rsa3072 2020-06-25 [E] [expires: 2022-06-25]

root@ubuntu:~/security#
```

Then we are importing the recipient's provided public key with the following command:

```
gpg --import AA77401D3F369DB1B1E6AC64BEFB73866CD8DBE8.asc
```

and let's print all the keys in order to evaluate our import.

```
pub   rsa3072 2020-06-25 [SC] [expires: 2022-06-25]
      3B25276CE331FADBB0AFBAD29E8362B45CFEBD4B
uid           [ultimate] Evangelos Siatiras <EN2190001@di.uoa.gr>
sub   rsa3072 2020-06-25 [E] [expires: 2022-06-25]

pub   rsa2048 2020-05-09 [SCE]
      AA77401D3F369DB1B1E6AC64BEFB73866CD8DBE8
uid           [ unknown] Konstantinos Limniotis <klimn@di.uoa.gr>
```

The key has been imported successfully. Then we are encrypting name.txt using recipient's public key as only he will be able to decrypt it, using the following command and the resulting ciphertext is stored in encrypted_name.gpg :

```
gpg -trust-model always --output encrypted_name.gpg --recipient 'Konstantinos Limniotis'
--encrypt name.txt
```

Then we are creating a signature of name.txt using the following command

```
gpg --default-key 'Evangelos Siatiras'  --output name.sig --sign name.txt
```

After that we prompted for the passphrase we used when generating the gpg key and a file name.sig is created.

Let's verify the signature of name.sig using the following command:

```
gpg --verify name.sig
```

```
root@ubuntu:~/security/ex_5# gpg --verify name.sig
gpg: Signature made Thu 25 Jun 2020 07:57:12 AM PDT
gpg:                using RSA key 3B25276CE331FADBB0AFBAD29E8362B45CFEBD4B
gpg: Good signature from "Evangelos Siatiras <EN2190001@di.uoa.gr>" [ultimate]
```

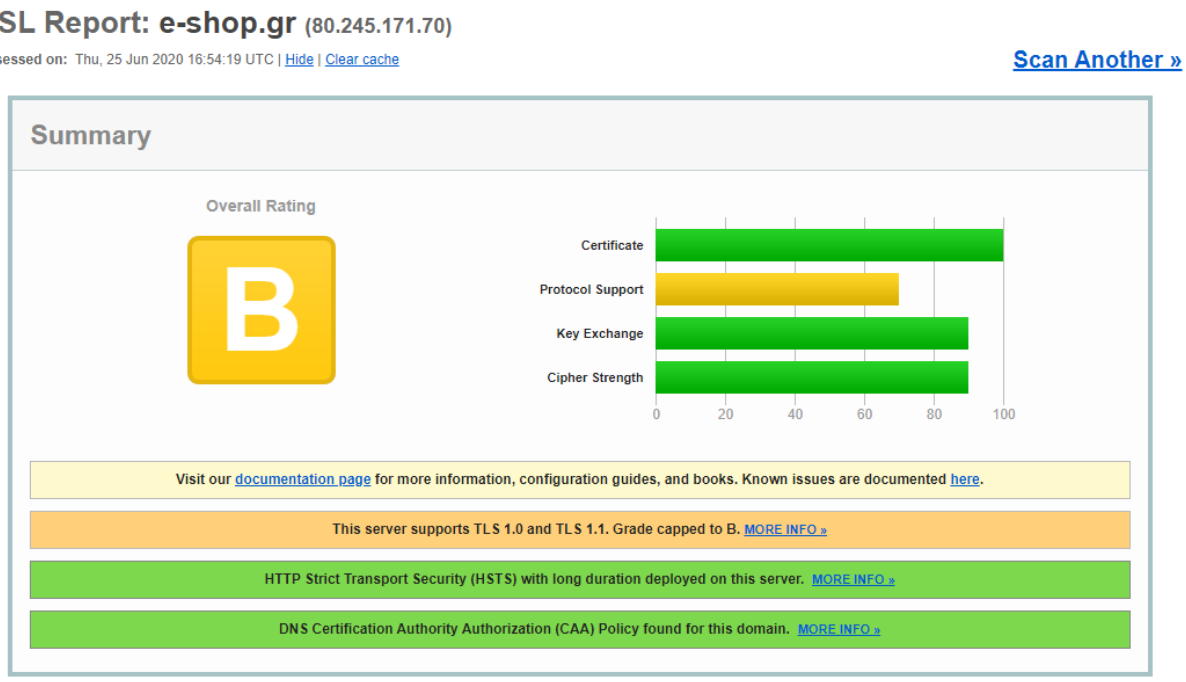
Now in order for the recipient to be able to decrypt the message and validate it's signature he needs the senders public key and the digital signature of the file (name.sig).The message is compressed before signed and the output is in binary format. The receiver given a signed document, can either check the signature or check the signature and recover the original document. To check the signature, use the --verify option. To verify the signature and extract the document use the --decrypt option.

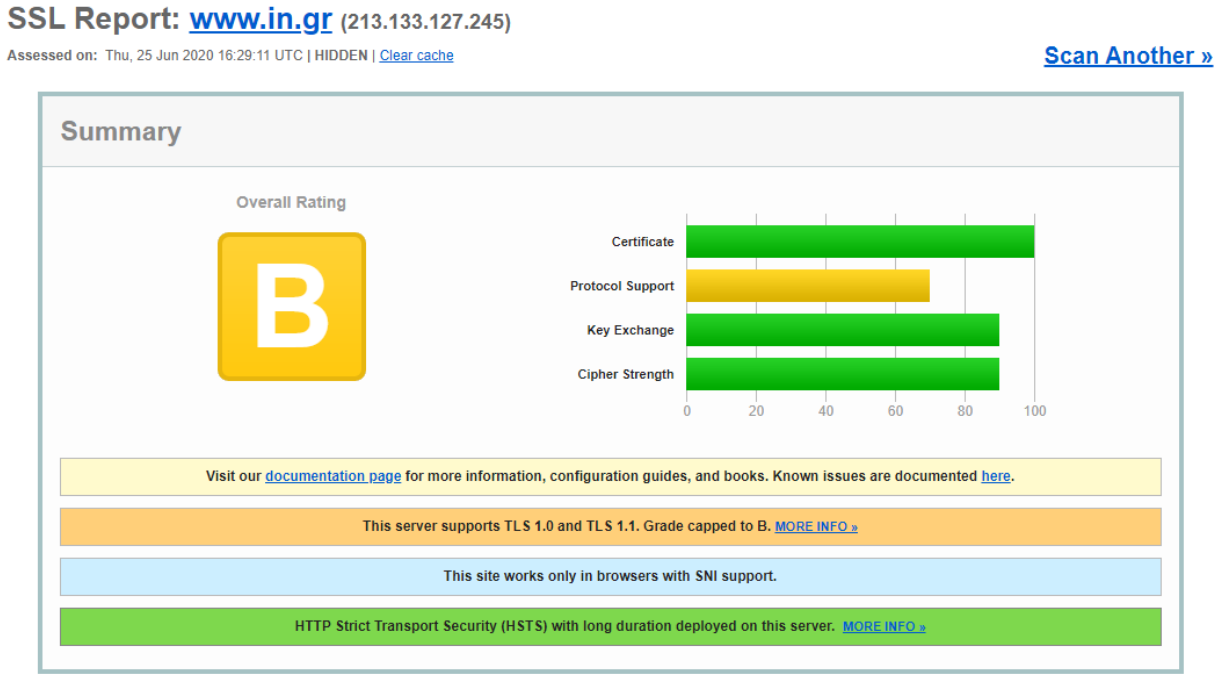
To export the senders public key, we use the following command:

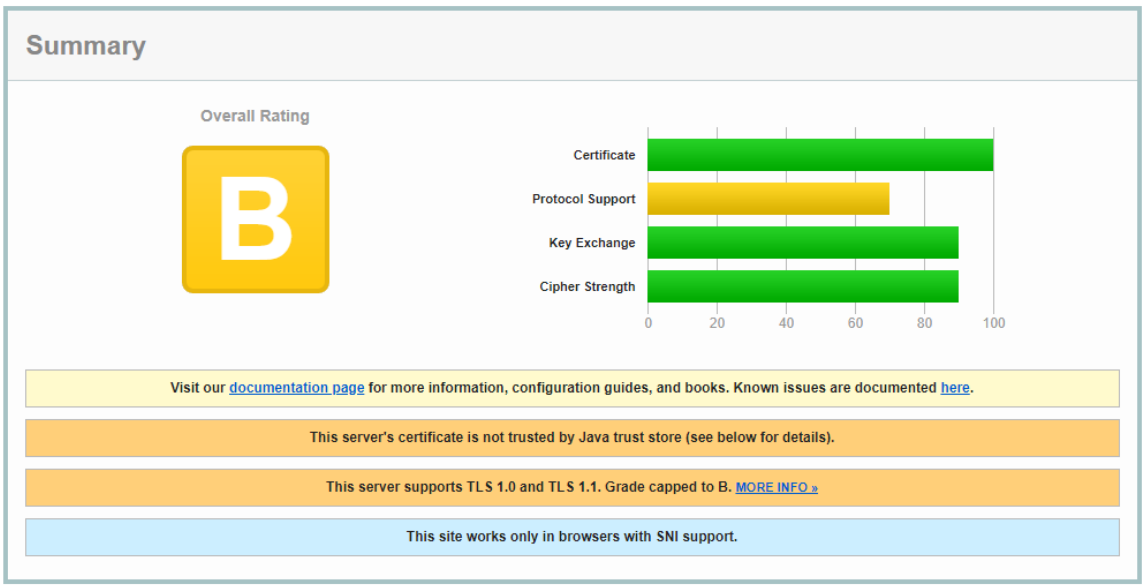
```
gpg --armor --export 'Evangelos Siatiras' > evangelos_siatiras_pub.asc
```

And we sent evangelos_siatiras_pub.asc and name.sig to the receiver and he has everything he need to validate and retrieve the plaintext.

Ex_6_a





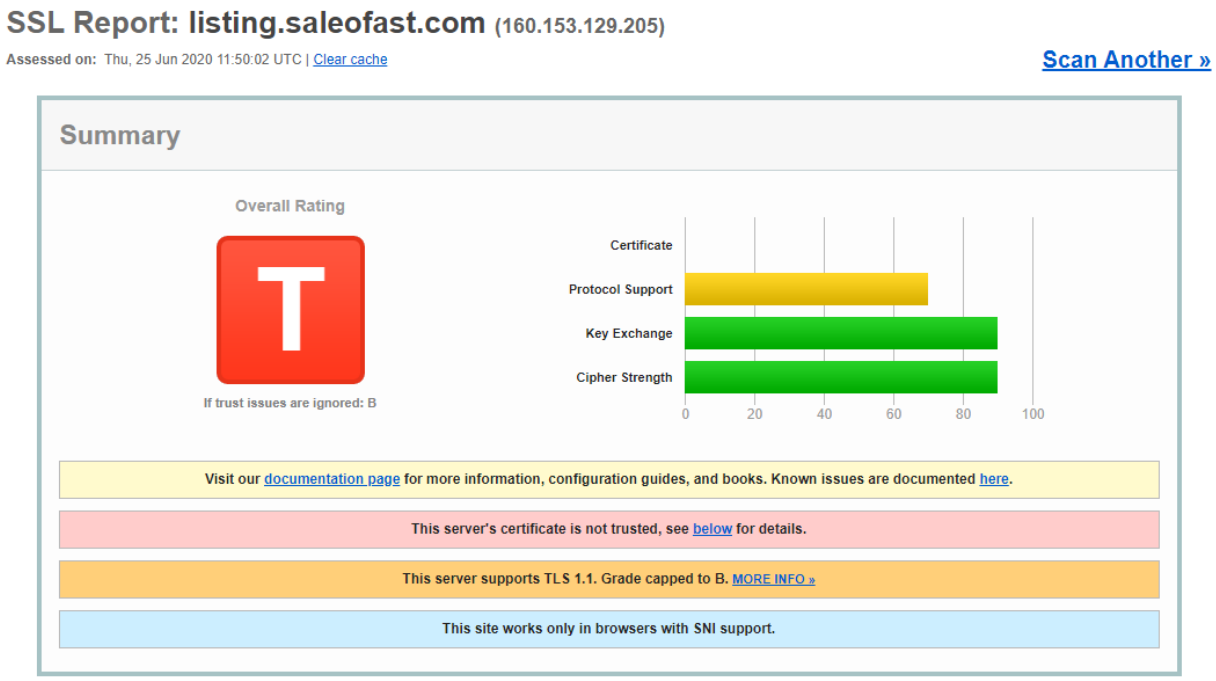


In the terms of the selection of a TLS Scanner tool I used the one proposed in the lecture which is in the following link and the results for 3 websites randomly selected are in the above pictures. The overall rating in the three websites is B due to they are still supporting TLS 1.0 and TLS 1.1.To encourage users to migrate to protocol TLS 1.2+ and remove protocol TLS 1.1 and TLS 1.0 from servers, SSL Labs lower the grade for SSL/TLS servers which use TLS 1.1 and TLS 1.0. Over the last few years, several serious attacks on TLS have emerged, including attacks on its most commonly used cipher suites and their modes of operation. For example The BEAST attack [BEAST] uses issues with the TLS 1.0 implementation of Cipher Block Chaining (CBC) (that is, the predictable initialization vector) to decrypt parts of a packet, and specifically to decrypt HTTP cookies when HTTP is run over TLS. Also there have been several practical attacks on TLS when used with RSA certificates (the most common use). These include [Bleichenbacher98] and [Klima03]. While the Bleichenbacher attack has been mitigated in TLS 1.0, the Klima attack, which relies on a version-check oracle, is only mitigated by TLS 1.1. Furthermore, both the AES-CBC [RFC3602] and RC4 [RFC7465] encryption algorithms, which together have been the most widely deployed ciphers, have been attacked in the context of TLS. Implementations SHOULD NOT negotiate TLS version 1.0 [RFC2246]; the only exception is when no higher version is available in the negotiation. Rationale: TLS 1.0 (published in

1999) does not support many modern, strong cipher suites. In addition, TLS 1.0 lacks a per- record Initialization Vector (IV) for CBC-based cipher suites and does not warn against common padding errors. Also implementations SHOULD NOT negotiate TLS version 1.1 [RFC4346]; the only exception is when no higher version is available in the negotiation. Rationale: TLS 1.1 (published in 2006) is a security improvement over TLS 1.0 but still does not support certain stronger cipher suites. o Implementations MUST support TLS 1.2 [RFC5246] and MUST prefer to negotiate TLS version 1.2 over earlier versions of TLS. Rationale: Several stronger cipher suites are available only with TLS 1.2 (published in 2008). In fact, the cipher suites recommended by this document (Section 4.2 below) are only available in TLS 1.2.

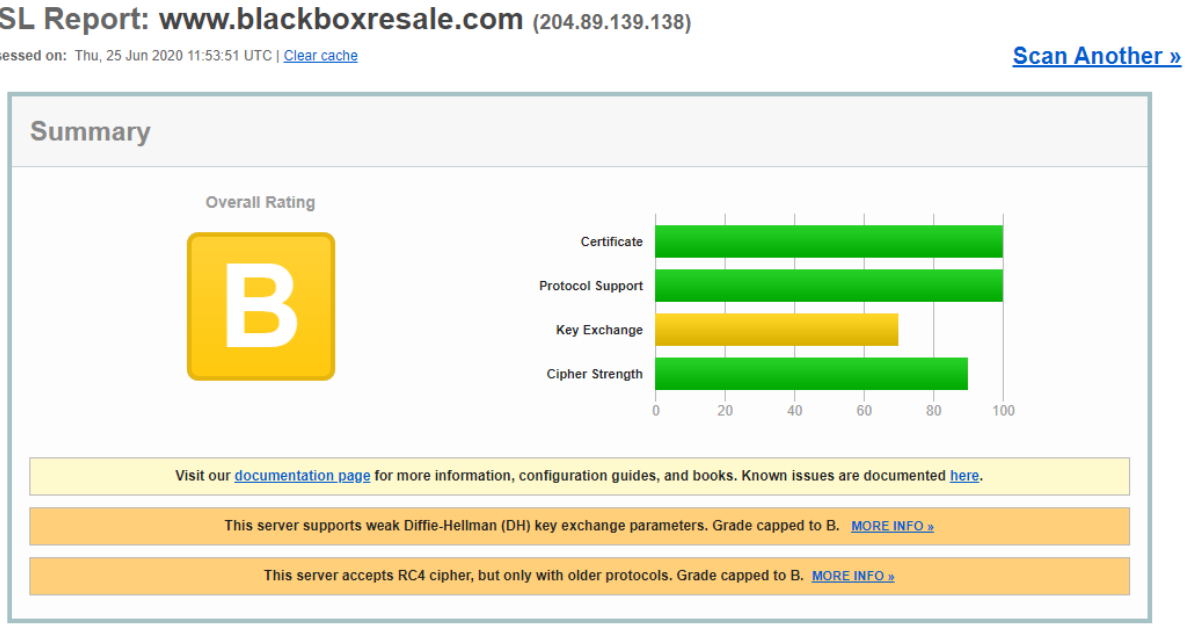
Ex_6_b

The overall rating from scanning the website mentioned below is particularly low.



Except the support of TLS 1.1 where it's importance explained above there is one major issue as the server's certificate is not trusted. Server certificate is often the weakest point of an SSL server configuration. A certificate that is not trusted (i.e., is not ultimately signed by a well-known certificate authority) fails to prevent man-in-the-middle (MITM) attacks and renders SSL effectively useless. A certificate that is incorrect in some other way (e.g., a certificate that has expired) erodes trust and, in the long term, jeopardizes the security of the Internet as a whole.

Ex_6_c



At first regarding the downgrade due to RC4 is because RC4 is demonstrably broken and unsafe to use in TLS as currently implemented. The difficulty is that, for public web sites that need to support a wide user base, there is practically nothing 100% secure they can use to replace RC4.

RC4 has long been considered problematic, but until very recently there was no known way to exploit the weaknesses. After the BEAST attack was disclosed in 2011, we—grudgingly—started using RC4 in order to avoid the vulnerable CBC suites in TLS 1.0 and earlier. This caused the usage of RC4 to increase, and some say that it now accounts for about 50% of all TLS traffic.At the moment, the attack is not yet practical because it requires access to millions and possibly billions of copies of the same data encrypted using different keys. A browser would have to make that many connections to a server to give the attacker enough data. A possible exploitation path is to somehow instrument the browser to make a large number of connections, while a man in the middle is observing and recording the traffic.

Regarding the downgrade due to weak Diffie-Hellman key exchange is a popular cryptographic algorithm that allows Internet protocols to agree on a shared key and negotiate a secure connection. It is fundamental to many protocols including HTTPS, SSH, IPsec, SMTPS, and protocols that rely on TLS. In the other hand there are several weaknesses in how Diffie-Hellman key exchange has been deployed.Some of them have been reported in the below attacks:

1. **Logjam attack against the TLS protocol.** The Logjam attack allows a man-in-the-middle attacker to downgrade vulnerable TLS connections to 512-bit export-grade cryptography. This allows the attacker to read and modify any data passed over the connection. The attack is reminiscent of the FREAK attack, but is due to a flaw in the TLS protocol rather than an implementation vulnerability, and attacks a Diffie-Hellman key exchange rather than an RSA key exchange. The attack affects any server that supports DHE_EXPORT ciphers, and affects all modern web browsers. 8.4% of the Top 1 Million domains were initially vulnerable.

2. **Threats from state-level adversaries.** Millions of HTTPS, SSH, and VPN servers all use the same prime numbers for Diffie-Hellman key exchange. Practitioners believed this was safe as long as new key exchange messages were generated for every connection. However, the first step in the number field sieve—the most efficient algorithm for breaking a Diffie-Hellman connection—is dependent only on this prime. After this first step, an attacker can quickly break individual connections.

Ex_7_A_A

Password *sniffing* is an attempt to intercept passwords passing over a computer network. Typically software programs are used to capture packets on the network. The attacker then analyzes the packets to determine which ones contain passwords. Encryption provides the best protection from sniffing attacks. Technologies such as SSL, SSH, and IPSEC provide a level of protection beyond traditional network layout and design countermeasures. IPSec is a network-level protocol that incorporates security into IPv4 and Ipv6 protocols directly at the packet level by extending the Ip packet Header. This allows the ability to encrypt any higher layer protocol. Password sniffing is inefficient in IPSec technology as the initial IP Packet is encapsulated to another IP header thus the payload (initial IP packet) is protected- encrypted.

Ex_7_A_B

IPSec does not prevent SYN flooding. IPSec is a VPN at the network layer (i.e. IP) and not the transport layer (TCP, UDP). It will transport any IP packets and not look deeper. But since SYN is a capability of TCP and SYN flooding an attack thus at the transport layer a pure IPSec VPN will not detect SYN flooding.Of course an IPSec VPN could be augmented with additional software or hardware to detect such attacks the same way as you could augment plain (non-VPN) IP networks this way. Though it will not be possible to detect such attacks within the encrypted IPSec traffic but only in the decrypted traffic.

Ex_7_A_C

DNS poisoning or DNS spoofing is a potential threat to DDoS attack and this has been witnessed in several case scenarios. One defense against the attacks would be to encrypt your sensitive traffic using an encrypting protocol such as SSH or IPsec. Middle boxes, e.g., NATs, complicate the scenario but do not necessarily prevent the attacks. As long as the attacker manages to spoof the DNS response or is able to manipulate the DNS servers, the attack will succeed equally well from outside the NAT. In a DNS-spoofing attack, the wrong IPsec policy is applied to the packets because the attacker controls the binding of DNS names, which are the identifiers used by the application, to IP addresses, which are the identifiers used in the security policy. It is therefore tempting to blame the insecure name resolution mechanism for the attacks. This logically leads to the idea of introducing a secure name service.

Ex_7_B_A

How the Attack is done.
If a machine “A” wishes to communicate with a machine “B”, machine “A” needs an IP address of machine “B”. However, “A” has only the name for “B”. So, what happens, “A” will use protocol DNS to get the IP address of “B” from the name.
A DNS request is then sent to a DNS server, declared at the level “A”, requesting the resolution of the name of “B” at its IP address. To identify this request, an identification number is assigned to it. Thus, the DNS server will send the response to this request with the same identification number.
The attack will consist of recovering this identification number by sniffing when the attack is carried out on the same physical network, or by using a flaw of the operating systems or the DNS servers which make these numbers predictable for the ability to send a falsified response before the DNS server.
Thus, machine “A” will use, without knowing it, the IP address of the hacker and not that of the machine “B” initially requested.

Ex_7_B_B

The Authentication header can provide the integrity service to the data packet, but cannot offer confidentiality to data packets like ESP. Thus, AH does not encrypt the traffic when used alone. So someone with access to the client system (either directly, or over the network) could re-configure one host to use a weak cipher that could be easily cracked (it would not take long to crack a 40-bit export-grade cipher with modern equipment).

Ex_8_a

The rational for the implementation I followed is exactly the one mentioned in the lecture presentation.

First, for generating a key pair, the user chooses two random strings x0 and x1; these constitute the user's private key, whilst the corresponding public key is the pair of hash values (h(x0), h(x1))

```
for(int i = 0;i<length;i++){
    private_key->X0[i] = (uint32_t) rand();
    private_key->X1[i] = (uint32_t) rand();

    public_key->X0[i] = secure_hash_function(private_key->X0[i]);
    public_key->X1[i] = secure_hash_function(private_key->X1[i]);
}
```

In terms of hash function I choose the one described in this Link .It is a very useful and enough optimal in the terms of performance as hashing with unsigned numbers is taking place and according to the author the "magic number" was calculated using a special multi-threaded test program that ran for many hours, which calculates the avalanche effect (the number of output bits that change if a single input bit is changed; should be nearly 16 on average), independence of output bit changes (output bits should not depend on each other), and the probability of a change in each output bit if any input bit is changed.

```
uint32_t secure_hash_function(uint32_t x){
    // From https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    // The second line further mixes the bits. Using just one multiplication isn't as good
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = (x >> 16) ^ x;
    return x;
}
```

The signature process works in case that there are only two possible messages, let' say "0" and "1". If the user wants to sign the message "0", then she reveals x0; otherwise, she reveals x1.

```
void sign(struct key* private_key, uint32_t message, uint32_t* signature){
    // There are only two possible messages, let' say "0" and "1".
    for(int i=0; i< private_key->length_t; i++){
        signature[i] = ((message >> i) & 1) ? private_key->X0[i] : private_key->X1[i];
    }
}
```

The verifier computes h(x0) (respectively, h(x1)) and checks whether the result coincides with the first (respectively, second) half of the public key.

```
int verify(uint32_t message, uint32_t* signature, struct key* public_key){
    for(int i = 0; i< public_key->length_t; i++){
        //check if i-th value of signature matches hash of public key (index determined by message-bit at position i)
        if((message >> i) & 1){
            if(secure_hash_function(signature[i]) != public_key->X0[i]){
                printf("\nFailed to verify signature at %dth bit. Excpected %u, Got %u\n",i, signature[i], public_key->X0[i]);
                return 0;
            }
        }else{
            if(secure_hash_function(signature[i]) != public_key->X1[i]){
                printf("\nFailed to verify signature at %dth bit: Excpected %u, Got %u\n",i, signature[i], public_key->X1[i]);
                return 0;
            }
        }
    }
    return 1;
}
```

Ex_8_b

So before runtime the user has to put as arguments the message he wants to sign followed by the key size.

Then the signature is being printed along with the verification result.

By following the methodology, I choose key size 2*m to sign a m-bit message.

The output given the message m1 and key size as :

```
./a.out '1001' 8
```

```
root@ubuntu:~/security/ex_8# ./a.out '1001' 8
Message: 9
Message in Binary: 00001001
Key Length: 8
Signature: 1033444424 1245913725 906500653 67763610 1365381887 1470871250 2061099077 200009660
 Success Signature is Valid
```

Ex_8_c

The output given the message m2 kai key length as :

```
root@ubuntu:~/security/ex_8# ./a.out '0010' 8
Message: 2
Message in Binary: 00000010
Key Length: 8
Signature: 674332273 1957729361 687629891 729600319 1918044876 1655876350 1937430229 742621022
 Success Signature is Valid
```

Ex_8_d

Now if we attempt to change the private key sign and then verify the signature using the previous public key,for the message m2 we get the following result:

```
**** Signing again with different private key ****
Signature: 674332273 1957729361 687629891 729600319 1126270887 1655876350 1937430229 742621022
Failed to verify signature at 4th bit: Excpected 1126270887, Got 3438799042

 Error Signature is NOT Valid
```

Note that prior publishing the signature of the message, no one else knows the 2×8 random numbers in the private key. Thus, no one else can create the proper list of 8 random numbers for the signature. And after the signature published, others still do not know the other 8 random numbers and thus cannot create signatures that fit other message hashes. The above scenario is valid by considering that the chosen has function is enough secure.