DAT510 - Assignment 2

Eirik Sakariassen

October 13, 2020

Contents

1 Abstract **1**

2 Introduction **2**
2.1 Diffie-Hellman 2
2.2 Blum Blum Shub 2
2.3 Advanced Encryption Standard 2
2.4 System 2

3 Design and Implementation **3**
3.1 Diffie-Hellman 3
3.2 Blum Blum Shub 3
3.3 Advanced Encryption Standard 4
3.4 Web Server 4

4 Test Results **6**

5 Discussion **8**

6 Conclusion **8**

1 Abstract

We will implement secure communication scenarios in this assignment by utilizing "simplified" cryptographic primitives. The assignment is divided into two parts, where we in part 1 will implement three cryptographic primitives. The first step is to use Diffie-Hellman (DH) key exchange[1] which enables two parts to exchange keys securely over a public network. We then use the shared key as seed in a cryptographically strong pseudo-random bit generator (CSPRBG) named the Blum Blum Shub (B.B.S)[2]. B.B.S generates a secure shared private key that in the end will be used to encrypt and decrypt messages over the network using AES ciphers[3]. We will in part 2 implement two web servers that communicates securely by using the program from part 1. The web server is

created as a chat app with live updated session information for demonstration purpuses only.

2 Introduction

2.1 Diffie-Hellman

The Diffie-Hellman key exchange allows two parties, which may be unknown to each other, to securely establish a shared key to be used for secure communication. This can all be done over an insecure channel. It was first formulated by Whitfield Diffie and Martin Hellman in 1976 [4]. The algorithm depends on the difficulty of computing discrete logarithms.

2.2 Blum Blum Shub

B.B.S is a cryptographically strong pseudo-random bit generator, meaning it passes the *next-bit test* [1]. Its cryptographic strength has strong public proof. The generator produces a sequence of bits by taking the least significant bit of x_{n+1} at each iteration. We choose two large prime numbers p and q such that

$$p = q = 3(mod4)$$

and our seed x_0 as the shared key from the DH-exchange. We define M as the product of p and q and can then iteratively produce x_{n+1} this way:

$$x_{n+1} = x_n^2(modM)$$

2.3 Advanced Encryption Standard

AES is a symmetric block cipher that will be used to encrypt and decrypt messages over the network. We will use it in Electronic Code Book (ECB) mode with the secret random key generated by B.B.S. ECB mode is not semantically secure and may leak information about the plaintext[5], so it should not be implemented in a real world application. We did however consider it "safe enough" for this short message chat app. We would ideally have used another mode where we can feed initialization vectors and nonces together with the content.

2.4 System

We can combine the three cryptographic primitives and create a secure communication protocol between two fictive characters named Alice and Bob. They will run a chat app on their own web servers which only has access to their own private key and the methods developed. Everything is implemented in python.

[1] https://en.wikipedia.org/wiki/Nextbit_test

3 Design and Implementation

3.1 Diffie-Hellman

We implement the Diffie-Hellman key exchange by a class which is initiated with a generator and a prime. The generator q is set as a 64-bit random prime and the prime p is chosen as the primitive root of q in order to prevent small subgroup attacks. The user who starts the chat session will propose p and q in the beginning to make sure that both parties agree on the same global parameters.

Once p and q is known, both parties starts of by generating their own private keys $(X_A < q)$ and $(X_B < q)$. These are used to calculate their public keys $(Y_A = p^{X_A} mod q)$ and $(Y_B = p^{X_B} mod q)$. Code block 1 shows the implementation of public key generation. They will then send their public keys in plaintext so that the receiving end can calculate a shared secret key K. Part A will use public key of B to generate K this way: $K = (Y_B)^{X_A} mod q$. Part B calculates K in a similar manner. This is shown in code block 2.

```
def get_public_key(self):
    return pow(self.prime, self.private_key, self.generator)
```

Listing 1: Generation of public keys

```
def get_session_key(self, public_key):
    return pow(public_key,self.private_key,self.generator)
```

Listing 2: Calculating shared secret key

3.2 Blum Blum Shub

The Blum Blum Shub generator class initializes two large prime numbers p and q by the internal methods *_generate_q* and *_generate_p*. The bit length of each prime is specified by a bitlen parameter in the init function which defaults to 1048 bits. The primes p and q can and also be explicitly specified if wanted (e.g. in test cases). The B.B.S initialization is shown in code block 3.

```
class BlumBlumShub:
    def __init__(self, q=None, p=None, bitlen=1024):
        self.bitlen = bitlen
        self.q = q if q is not None else self._generate_p()
        self.p = p if p is not None else self._generate_q()
        self.M = self.p*self.q
```

Listing 3: Initialization of B.B.S generator

We can generate keys with specific lengths once the object is initialized. The method *generate*, as shown in code block 4, takes a seed (from DH-exchange in our app) and the specified key length as parameters. The method iterates and computes bit for bit by the strategy described in subsection 2.2. It returns the generated secret key as a string of bits once the iteration is completed. The key length should match the symmetric cipher method it is meant to be used for. In our case with AES, bit lengths of 128, 192 or 256 would be fine.

```python
def generate(self, seed, key_len):
    bits = []
    for i in range(key_len):
        seed = pow(seed,2, self.M)
        bits.append(bin(seed)[-1])
    return ''.join(bits)
```

Listing 4: B.B.S generator

3.3 Advanced Encryption Standard

AES was not implemented as it was not a part of this assignment and we did already implement a symmetric cipher in Assignment 1. Instead we used two packages from the external library PyCryptodome[2], namely Crypto.Cipher and Crypto.Util.Padding. We created methods for encryption and decryption as shown in code blocks 5 and 6. Both methods converts the string of bits (key) to bytes by first converting it to a hexadecimal string with a helper method *to_hex*. We can then use the built in python method *bytes.fromhex* to get the key in correct format for AES. The cipher could then initiated in ECB mode with the key.

We had to add padding for plaintext and unpadding for ciphertext in the cases where they did not fit in the block size of AES (16 bytes). This was performed by the helper functions *pad_text* and *unpad_text* which used methods from the Crypto.Util.Padding package. The encryption method returned ciphertext as bytes while the decryption method decoded it back to a string in the return.

```python
def encrypt(plaintext, key):
    key = bytes.fromhex(to_hex(key))
    plaintext = pad_text(plaintext.encode())
    cipher = AES.new(key, AES.MODE_ECB)
    return cipher.encrypt(plaintext)
```

Listing 5: AES encryption

```python
def decrypt(ciphertext, key):
    key = bytes.fromhex(to_hex(key))
    cipher = AES.new(key, AES.MODE_ECB)
    plaintext = cipher.decrypt(ciphertext)
    return unpad_text(plaintext).decode()
```

Listing 6: AES decryption

3.4 Web Server

We developed two Flask web servers with the same logic. These were meant to communicate securely with each other by using the communication protocol we developed. The fictive clients Alice and Bob could connect to a chat app that enabled this secure

[2]https://pycryptodome.readthedocs.io/en/latest/index.html

communication between them. The application was written in simple basic HTML with Jinja2 templating.

Whenever both users are online, one of them can click on the "Establish new chat session" button to set up a secure channel between them. The button triggers a GET request to route */new_chat* on its own web server. Lets say Alice clicks on this button. Alice's web server will then clear the current session and set up a new DH-exchange object which is used to calculate a public key. Alice will then request Bob's public key by a POST request to */getpub* on Bob's server. Alice is sending along the calculated public key as well as the prime and generator used. Bob can in the other end set up its own DH-exchange object with the same parameters and generate a public key for Alice. Bob can then use Alice's public key to generate a session key that will be used as seed for B.B.S. Bob can now get a secret key to be used for AES. Bob has now set up everything and can send its own public key back to Alice so she can complete the last steps too. This setup between the two web server are illustrated in code blocks 7 and 8

```
@app.route('/new_chat')
def new_chat():
    clear_session()
    dh = DiffieHellman()
    public_key = dh.get_public_key() # Send to other part
    params = {'generator': dh.get_generator(), 'prime': dh.get_prime(), '
    key': public_key}
    url = session['endpoint'] + '/getpub'
    res = requests.post(url, json=params).json()
    received_public_key = res['key']
    session['key'] = dh.get_session_key(public_key=received_public_key)
    #CSPRNG
    bbs = BlumBlumShub()
    secret_key = bbs.generate(seed=session['key'], key_len=128)
    session['secret_key'] = secret_key
    session['active_chat'] = True
    return redirect(url_for('index'))
```

Listing 7: New chat on Alice's server

```
@app.route('/getpub', methods=['GET','POST'])
def get_pub():
    if request.method == 'POST':
        # Calculate new session key
        clear_session()
        data = request.get_json(force=True)
        #Use same generator and prime as the user who initiated this
    session
        dh = DiffieHellman(generator=data['generator'], prime=data['prime
    '])
        public_key = dh.get_public_key() # Send to other part
        session['key'] = dh.get_session_key(data["key"])
        #CSPRNG
        bbs = BlumBlumShub()
        secret_key = bbs.generate(seed=session['key'], key_len=128)
        session['secret_key'] = secret_key
```

```
15          session['active_chat'] = True
16          return jsonify({'key': public_key})
17      else:
18          return session['key']
```

Listing 8: Getpub on Bob's server

The session information is displayed in a box that contains the session key (from DH), secret key (from B.B.S used in AES) and whether the chat is active. We can verify the implementation by looking at this info box on both sides. The session key and secret key should be the same on both apps. They can then start to send encrypted messages to each other once the chat is active. The endpoints */getmsg* [10] and *sendmsg* [9] takes care of this communication. Encryption happens when the user sends a message and decryption when it receives.

```
1  @app.route('/sendmsg', methods=['POST'])
2  def sendmsg():
3      if session['secret_key'] is not None:
4          msg = request.form['msg']
5          msg = encrypt(plaintext=msg, key=session['secret_key'])
6          msg = str(base64.encodebytes(msg), 'utf-8')
7          url = session['endpoint'] + '/getmsg'
8          res = requests.post(url, json={'msg': msg})
9      return redirect(url_for('index'))
```

Listing 9: Encrypt message and send it

```
1  @app.route('/getmsg', methods=['POST'])
2  def get_msg():
3      if request.method == 'POST':
4          data = request.get_json(force=True)
5          msg = base64.decodebytes(data['msg'].encode())
6          session['messages'].append({
7              'decrypted': decrypt(ciphertext=msg, key=session['secret_key'
       ]),
8              'encrypted': msg
9          })
10     return ''
```

Listing 10: Get encrypted message and decrypt it

The users can refresh its inbox to see new messages from the other part. The inbox shows both the encrypted message it received and the final decrypted message it got after using AES with the secret key. The decrypted message should then (obviously) be whatever the other user sent. We can see that the communication protocol worked as expected by figure 1 and figure 2.

4 Test Results

A test script named *test.py* would enable the user to run tests on the entire protocol or each of the sub parts (DH, BBS, AES). These tests used default parameters and

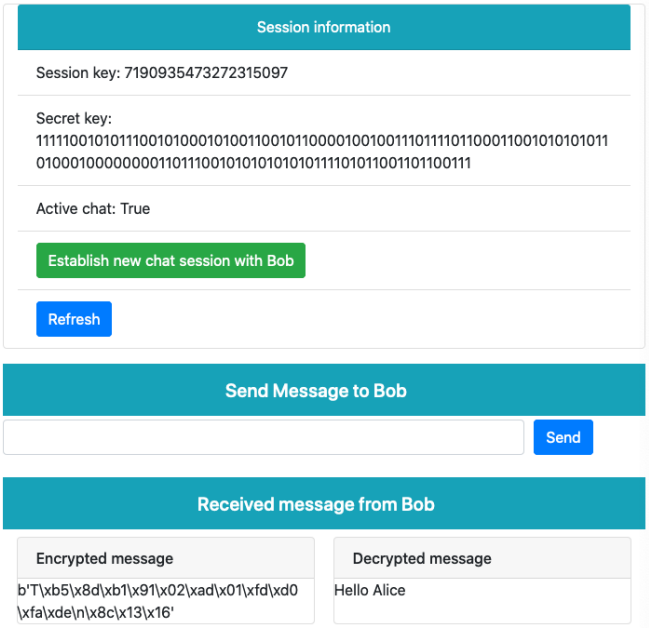

Figure 1: Alice's chat app

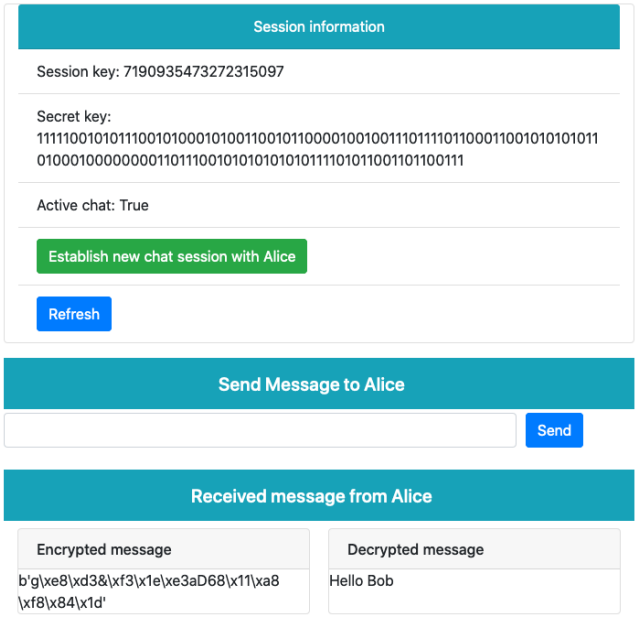

Figure 2: Bob's chat app

pre-defined messages only. If the user wanted to specify parameters such as primes, bit lengths and messages to be encrypted, another script named *demo.py* would enable the user to do so. The program would display all generated keys and parameters as well as the encrypted and decrypted message. The web application discussed in 3.4 could also be used to test the protocol as it provided a decent user interface with most of the same opportunities.

The program worked fine with all of the parameters described in this report. Trying different primes and bit lengths could break the program in some cases. The reason for this was usually that the program froze due to a too long bit lengths (e.g. over 5000 for BBS) or the encryption/decryption got errors when converting the key hexadecimal and then bytes.

5 Discussion

The encryption/decryption was a bit tricky since it contained a lot of converting for both text and the key. As named in 4 would the program crash in some cases due to byte and hex convertion of the key. The program would also need to decode and encode a lot back and forth in the web server since bytes weren't serializable in JSON. Smoother ways of solving this should be implemented in the future.

The web server defined p and q from static pre-generated files to ensure both parties used the same parameters. This was because these parameters could not be sent securely over the network during the setup phase between the parties. We could try to fix this issue by letting the first party generate $M = p * q$ and encrypt it with their shared key using AES. Bob could then decrypt it and generate BBS based on that.

6 Conclusion

We have in this assignment implemented three cryptographic primitives that put together created a secure communication channel between two parties. In order to demonstrate the "end-to-end" encryption scheme, a chat application was made on top of the protocol. Two fictive parties was then able to exchange their keys by public key cryptography, further strengthen it by pseudo random number generator, and then send messages encrypted by a symmetric cipher. The protocol and application was made as a "proof of concept" only, and should not be used for any purposes in a real application. We have discussed how we could improve some parts in order to use it in such a case. For instance should AES be used in other modes than ECB. We could also try the Elliptic Curve Diffie-Hellman combined with some other CSPRBG.

References

[1] Wikipedia contributors. Diffie–hellman key exchange — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Diffie%E2%80%`

93Hellman_key_exchange&oldid=983043554, 2020. [Online; accessed 13-October-2020].

[2] Wikipedia contributors. Blum blum shub — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Blum_Blum_Shub&oldid=974843571, 2020. [Online; accessed 13-October-2020].

[3] Wikipedia contributors. Advanced encryption standard — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=982883791, 2020. [Online; accessed 13-October-2020].

[4] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[5] Dan Boneh, David Brumley, and Shaoquan Jian. Block cipher.