Name: Bror Hansen

Student initials: broh

Course name: Introduction to Security

Date: 10/07/25

Mandatory Hand-in 1

**Introduction**

This report covers my solutions for Mandatory Hand-in 1 in the course **Introduction to Security**. The assignment is divided into three parts. In the first part, I use brute force to find the private key and decrypt a student number encrypted with the ElGamal scheme. In the second part, I modify an intercepted encrypted message so that it decrypts to my own student number without knowing the private key. In the third part, I create a simple secure client-server communication setup using public key cryptography and a self-signed certificate to ensure confidentiality and integrity.

**Part 1**

***Brute forcing x***

To recover the server’s private key I perform a straightforward brute-force search for the discrete logarithm . Rather than recomputing from scratch at every step, I keep an accumulator that holds the current power of . Starting with the value , the loop repeatedly multiplies the accumulator by the shared base and reduces until the accumulator equals the public key. This avoids calling an expensive modular-exponentiation routine inside the loop.[[1]](#footnote-1) Each iteration does a single modular multiplication, so the work per iteration is minimal.

***Finding the student number***

To reconstruct the key number, we just need to decrypt the intercepted ElGamal ciphertext to recover the student number m. The decryption process uses the fact that the shared secret s=c1x mod p was used during encryption to hide the original message. To retrieve the plaintext, we compute the modular inverse of this shared secret and multiply it by ​. In the code, this is done with, which efficiently calculates the modular inverse of under the prime . The result gives , which is then multiplied by c2 and reduced modulo p to obtain the original message . This final value represents the decrypted student number.

**Part 2**

In part 2 I exploit ElGamal’s multiplicative property to change an intercepted ciphertext so that, when the server decrypts it, the plaintext becomes a chosen value (my student number) even though I do not know the server’s private key. ElGamal ciphertexts have the following form

Where is the original plaintext and the public key, is .

In order to modify the intercepted encrypted message to be my chosen student id, I compute a multiplier , that transforms the original message to my desired target . Algebraically it looks like the following:

We then divide both sides by and get

where is the multiplicative inverse of modulo . In other words: is the value that, when multiplied with the original plaintext under the modulus, results in the chosen target.

In the code this is implemented as:

orig = original\_student\_number % p

target = my\_student\_number % p

factor = (target \* pow(orig, p - 2, p)) % p

c2\_new = (c2 \* factor) % p

Where orig and target are just the original and target student number reduced modulo so calculations are easier. The pow(orig, p - 2, p) essentially finds the modular inverse of orig (Fermat’s little theorem) and then multiplying that with the target gives the exact multiplier t (stored in factor) needed to make the decrypted message equal my student number. Then we just multiply this factor with which will result in the new that when decrypted, results in my student number.

Replacing with into the original ciphertext we get the following:

And when decrypted (removing ),

Because is left unchanged, the shared secret remains the same and the modification or “attack” is successful without knowing the private key.

**Part 3**

In part 3 I had to “Improve the system by ensuring that the communication between SAP and the grading system is confidential and ensures integrity for the messages”. I did this by implementing a simple server and a client that can communicate using TLS using a self-signed certificate. I created the self-signed certificate using the python SSL library. TLS ensures that data sent between the client and server is encrypted, authenticated and protected against interception or man in the middle tampering during transmission.

The server runs on a fixed port (7007) and uses a self-signed certificate (server\_cert.pem) along with its private key (server\_key.pem) to authenticate itself to connecting clients. Using Python’s ssl module, I created a secure SSL context based on ssl.PROTOCOL\_TLS\_SERVER, which enables the server to perform encrypted communication over a TCP socket.

The key initialization part of the server code looks like this:

context = ssl.SSLContext(ssl.PROTOCOL\_TLS\_SERVER)

context.load\_cert\_chain(certfile='server\_cert.pem', keyfile='server\_key.pem')

Once the server is up and running, it listens for incoming connections and then wraps each accepted socket in the TLS context. When a client connects, the server performs a TLS handshake, negotiates encryption parameters, and establishes a secure channel. It then prints information about the connection and securely receives and acknowledges the client’s message.

An example of the server’s output when a client connects can be seen below:

py server.py

Starting TLS server on port 7007

Connection from ('127.0.0.1', 55331) TLS: TLSv1.3

Received: test message

This confirms that a secure TLS session was successfully established and that all transmitted data is encrypted and integrity-protected. The TLS layer automatically handles key exchange, authentication, and message integrity checks using symmetric encryption and MACs negotiated during the handshake.

The client connects to the server using a similar TLS setup. It verifies the server’s certificate against the trusted server\_cert.pem, ensuring that the client only sends data to the legitimate server. After establishing a secure connection, the client transmits a message (for example, a student number update) and waits for the server’s acknowledgment (ACK). All the heavy lifting is done behind the scenes using the SSL library.

An example of a client connecting to the server and receiving an ACK in terminal can be seen here:

py client.py "test message" localhost 7007

Connected, TLS: TLSv1.3

Server reply: ACK

**CONCLUSION**

To very briefly conclude, this report demonstrates the use of cryptographic principles through ElGamal encryption and also TLS implementation.

1. Spanning Tree, *“Diffie-Hellman Key Exchange: How to Share a Secret”*, YouTube, uploaded May 27, 2024. Available at: <https://www.youtube.com/watch?v=yjHcQ2c1wCk> [↑](#footnote-ref-1)