Network and Internet Security Assignment

Nkosie Gumede, Cilliers Pretorius, Minenlhe Sithole

May 2019

1 Introduction

For this assignment, we were tasked with creating a client-server application that implement security using the Pretty Good Privacy (PGP) algorithm. We took an existing application that served as a client-server chat application and implemented the PGP algorithm to compress, encrypt, and authenticate messages that are sent.

2 Implementation

2.1 Language Choice

The original chat application was built in Java to take advantage of the native libraries that dealt with networking and sockets. Similarly, we used Java for the encryption to take advantage of the libraries that exist, thereby minimising the amount of code we have to write and test before the actual PGP algorithm can be implemented.

2.2 Connectivity and Communication

The server-client system we repurposed uses Java sockets, which are Transmission Control Protocol (TCP) by default. Using TCP as a transport layer allows for greater security and privacy compared to UDP. Reliability is a major factor to keep in mind when dealing with encrypted and compressed messages, to ensure that decryption is and decompressing is possible on the receiver side. Since TCP provides inherent reliability, we don't have to ensure reliability from an application point of view.

The server is considered 'always on'. Clients can log in to the server. When the client logs in, the server opens a port and socket for the client alone. A relatively secure TCP connection is established when the client logs in, and that connection is used for the duration of the session. When a client wants to send a message to another client, the first client (client A) sends the encrypted message to the server. The server then acts as client B and decrypts the message using the PGP algorithm.

2.3 Encryption Algorithms

We used the PGP algorithm to encrypt the message. PGP utilises both public-private key pair encryption and shared key encryption. We will explain the sequence of events that occur when sending a message to describe the algorithm.

Say client A (Alice) wants to send a message to client B (Bob). Alice will type a message (call the message M for now) and compute a hash of M using the Secure Hash Algorithm with a digest of 512 bits (SHA-512). Alice then encrypts this hash using her private key, and the encrypted hash is appended to M. This combined message is then compressed using the GZIP algorithm. A one time session key (to use in the Advanced Encryption Standard (AES) algorithm) is generated by Alice and used to encrypt the combined message and hash using the AES algorithm. Lastly, the session key is encrypted using Bob's public key and appended to the encrypted message and hash. This combination is then sent to Bob.

When Bob receives the message, the first step for him is to decrypt the session key using his private key. The session key is used to decrypt the combined message and hash, before the two parts are decompressed using the same algorithm that Alice used to compress it. Bob then calculates the hash value of the message using the same SHA-512 algorithm that Alice used. Lastly, this hash and the decrypted hash sent by Alice is compared. The two hash values are used to ensure that the message is authentic as sent by Alice. If the hash values are not identical, then the message was changed since Alice calculated the hash.

In our application, the server takes the role of Bob.

2.4 Key Management

When the clients log on, they generate their public/private key pair. The public key is sent to the server. At the same time, the client gets the server's public key from the server. The client is the only one with access to their private key. Similarly, the server is the only one with access to the server private key. The session key is determined for each message.

2.5 Procedure and Testing

Our first step was to improve the server-client app to work as intended. From there, we tried to create a PGP implementation from scratch, utilising native Java libraries and Bouncy Castle. However, this proved quite difficult and time-consuming. In earlier research to understand PGP, we came across Tejaas Solanki's tutorial on the subject (See acknowledgements below). We then adapted that code to work for our server-client application.

Testing consisted of trace statements that showed each step in the algorithm described above. A few test messages was sent by the client and the server output was compared to the initial message for correctness. Screenshots of one test run can be found in the appendix.

3 Work Allocation

Nkosie Gumede did most of the work on the server-client application itself, and assisted with integrating the encryption protocol into the application. Cilliers Pretorius did the writeup, and assisted in testing and debugging of the final product. Minenhle Sithole did the encryption implementation, as well as much of the debugging and testing of the final product.

It must be noted that we lost the fourth member of our group only two weeks before the assignment was due, while we were on the research phase and our work timeline was created with four group members in mind.

4 Acknowledgements

The server-client application was created by Dillon Woodman, Ntuthuko Mthiyane, and Nkosie Gumede. We improved on it and implemented encryption using the PGP algorithm. The code required for the encryption was adapted and modified from Tejaas Solanki's code to suit our application.[1] This code is freely available on Medium.com's freeCodeCamp platform.

5 Appendix

5.1 Example of message in PGP algorithm

Screenshots of a test run of the encryption is provided in sequence.

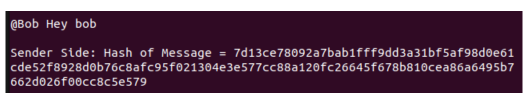

Figure 1: Alice's message and the hash she calculated

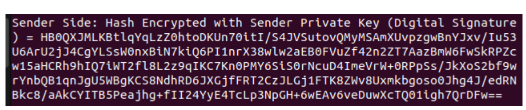

Figure 2: The message with the hash encrypted using Alice's private key

[1] https://medium.freecodecamp.org/understanding-pgp-by-simulating-it-79248891325f

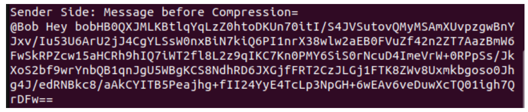

Figure 3: The message and appended encrypted hash

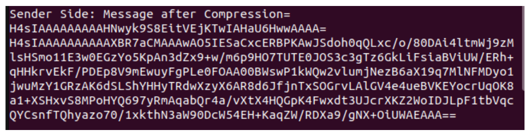

Figure 4: The message and appended encrypted hash after compression

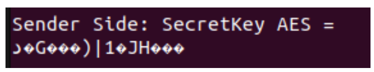

Figure 5: The session/message key generated by Alice

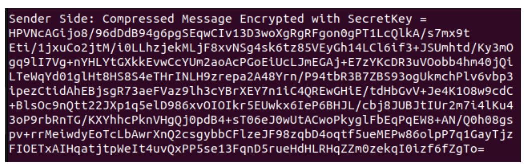

Figure 6: The message and hash encrypted using the message key

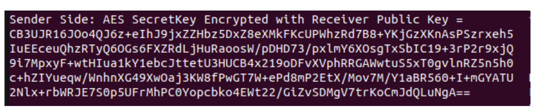

Figure 7: The message key encrypted using Bob's public key

Figure 8: The final encrypted message sent to Bob

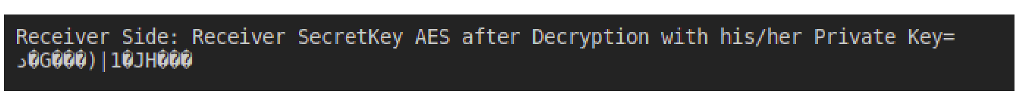

Figure 9: The secret key after being decrypted using Bob's private key

```
Receiver Side: Message After Decryption with SecretKey=
H4sIAAAAAAAAAHNwyk9S8EitVEjKTwIAHaU6HwwAAAA=
H4sIAAAAAAAAAXBR7aCMAAAwAO5IESaCxcERBPKAwJSdoh0qQLxc/o/80DAi4ltmWj9zMlsHSmo1lE3w0EGzYo5KpAn3dZx9+w/m6p9HO7T
aBViUW/ERh+qHHkrvEkF/PDEp8V9mEwuyFgPLe0FOAA00BWswP1kWQw2vlumjNezB6aX19q7MlNFMDyo1jwuMzY1GRzAK6dSLShYHHyTRdwX
AlGV4e4ueBVKEYocrUqOK8a1+XSHxvS8MPoHYQ697yRmAqabQr4a/vXtX4HQGpK4Fwxdt3UJcrXKZ2WoIDJLpF1tbVqcQYCsnfTQhyazo70/
qZW/RDXa9/gNX+OiUWAEAAA==
```

Figure 10: The compressed message after decryption using the secret key

```
Receiver Side: UnZipped Message=
@Bob Hey bob
HB0QXJMLKBtlqYqLzZ0htoDKUn70itI/S4JVSutovQMyMSAmXUvpzgwBnYJxv/Iu53U6ArU2jJ4CgYLSsW0nxBiN7kiQ6PI1nrX38wlw2aEB
wSkRPZcw15aHCRh9hIQ7iWT2fl8L2z9qIKC7Kn0PMY6SiS0rNcuD4ImeVrW+0RPpSs/JkXoS2bf9wrYnbQB1qnJgU5WBgKCS8NdhRD6JXGjf
mkbgoso0Jhg4J/edRNBkc8/aAkCYITB5Peajhg+fII24YyE4TcLp3NpGH+6wEAv6veDuwXcTQ01igh7QrDFw==
```

Figure 11: The message and encrypted hash after being decompressed

```
Receiver Side: Received Hash=
7d13ce78092a7bab1fff9dd3a31bf5af98d0e61cde52f8928d0b76c8afc95f021304e3e577cc88a120fc26645f678b810cea86a6495b
```

Figure 12: The hash received decrypted using Alice's public key

```
Receiver Side: Calculated Hash by Receiver=
7d13ce78092a7bab1fff9dd3a31bf5af98d0e61cde52f8928d0b76c8afc95f021304e3e577cc88a120fc26645f678b810cea86a6495b
```

Figure 13: The hash of the received message calculated by Bob

```
Received Hash == Calculated Hash
Thus, Confidentiality and Authentication both are achieved
Successful PGP Simulation
```

Figure 14: Confidentiality is assured by having both the received and calculated hashes being identical