CISC 468 - P2PSecApp

Leyne, Aidan (20213321)
Liang, Sean (20221439)

ABSTRACT

This project explores, tests, and implements a secure peer-to-peer communication service through a command-line interface. The service is implemented and available in two languages, Python and JavaScript, and can be used in combination with each other. Messages or files sent between peers are protected against replay-attacks and users may regenerate and distribute new keys if needed. The service maintains message security in the even of a host becoming lost or stolen.

1 INTRODUCTION

This report describes the design and development of our secure messaging application. It details the technologies and methodologies we used to ensure that the application meets modern security standards as well as each specified requirement. The project goal was to create a system where messages between users are private, authenticated, and tamper-proof.

We selected our specific cryptography libraries and algorithms based on their proven reliability and effectiveness in ensuring data security, and made sure to keep cryptography best-practices in mind. Key lengths and other security parameters were chosen to balance performance with the level of security needed to protect against potential threats. The report also covers the protocols we developed for secure client communication, and how we've managed to maintain message integrity and confidentiality.

2 LANGUAGE SELECTION

The decision to use Python and JavaScript for our application was influenced largely by ease of development, as well as consideration of our project's goals.

2.1 Python and JavaScript

After weighing all options, we found it best to use Python and JavaScript for the following reasons:

2.1.1 Familiarity. Exploring a new project which encompasses multiple languages, new libraries and many uncertainties, so we looked to languages we have used in the past - languages we are syntactically-comfortable and languages for which we are able to debug.

2.1.2 Popularity and Documentation. The popularity and maturity of the two languages for security applications translates into a wealth of resources, including extensive documentation, community support, and a plethora of learning materials. Their respective cryptographic libraries also provide well-tested and widely scrutinized implementations of our needed algorithms and functions, which greatly reducing the risk of vulnerabilities associated with lesser-used crypto code.

2.1.3 Development Turnaround Time. Both languages are known for their concise syntax. This helped expedite the development process and troubleshooting.

2.1.4 Socket-level Capabilities. The real-time, bi-directional communication between clients and servers is streamlined through excellent socket-level capabilities in both languages. This feature is vital for implementing secure, efficient communication channels in our application.

2.2 Other Language Options

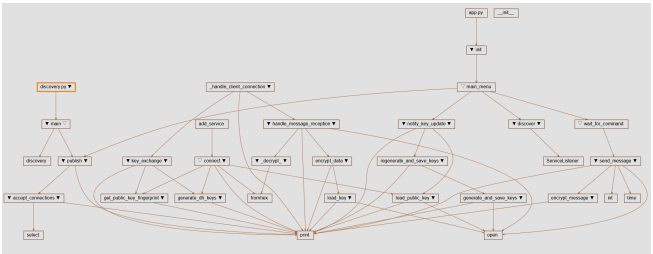
While other programming languages like Java, C#, and C were considered, they presented certain drawbacks in the context of our project needs:

2.2.1 Java and C#. Despite their powerful capabilities, these languages have relatively fewer dedicated cryptographic libraries compared to Python and JavaScript. Additionally, their setup requirements and verbose syntax could potentially slow development turnaround times. The complexities associated with socket-level communication further influenced our decision.

2.2.2 C. As a low-level programming language, C offers granular control over system operations. However, this comes with increased complexity regarding memory management, data types, and pointers, introducing a higher margin for error and potential security vulnerabilities.

3 SYSTEM DESIGN AND ARCHITECTURE

The architecture of our application is modular, comprising of several components that interact with each other to deliver secure messages. The application is a decentralized, peer-to-peer (P2P) system that eliminates the need for a centralized server for message routing and key management. The architecture can be visualized as follows:



Python-side architecture (Provided through an *Understand* Butterfly graph on discover.py)

3.1 User Interface

A Command-Line Interface (CLI) is used for user interaction, which provides a straightforward and accessible way for users to user the application. This lightweight interface operates seamlessly across different operating systems. The CLI prompts users with clear,

concise options for generating keys, regenerating keys, publishing services, discovering peers, sending messages, and sending files.

3.2 Key Generation Module

Automatically generates RSA key pairs for each user upon initialization. These keys serve two primary purposes: secure message encryption/decryption and identity verification through digital signatures.

3.3 Discovery Mechanism

Enables users to locate each other on the network. It facilitates the sharing of public keys and network addresses, which is essential for initiating direct communication. Handles the initial connection setup and subsequently manages encrypted communication using exchanged keys. Also directly handles the transmission and reception of messages between peers using the Messaging module.

3.4 Messaging Module

Contains our encryption and decryption funcions. Employs RSA for secure key exchange and AES-256 in CBC mode for message content encryption.Ensures that messages are readable only by the intended recipient.

3.5 Storage Module

The Storage Module handles the secure management of cryptographic keys, as well as the messages. It uses AES-256-CBC for this purpose. Key management involves generating a unique 256-bit encryption key stored securely on the file system and only loaded when necessary, which minimizes exposure. Each encryption operation generates a new 16-byte initialization vector (IV) to produce distinct ciphertexts for similar data inputs. The decryption process involves extracting the IV from the encrypted payload and using the stored key to accurately reconstruct the original plaintext.

4 CRYPTOGRAPHIC LIBRARIES AND ALGORITHMS

This section reviews the cryptographic libraries implemented within the application, along with the reason for their selection. We will also explore the algorithms implemented to ensure the security properties.

4.1 Cryptographic Libraries

4.1.1 JavaScript Crypto Library. The *crypto* module, which is available within Node.js, serves as our JavaScript-side cryptographic library. It includes support for various encryption algorithms, hash functions, and secure random number generation.

4.1.2 Python Cryptography Library. We use the Python *cryptography* library for its extensive cryptographic functionalities, including key generation, encryption/decryption, and secure message handling. It provides a large suite of cryptographic primitives, making it an ideal choice for our security operations.

4.2 Socket and Connection Libraries

4.2.1 JavaScript bonjour & net. For the JavaScript side of the application, we used *Bonjour* (also known as zero-configuration networking), which enables automatic discovery of devices and services on a local network using industry standard IP protocols. We made this choice due to its simplicity in service advertisement and discovery over the network. It allows our application to automatically discover peers without the need for manual configuration, making the network communication user-friendly, and effectively removes any setup work for the user.

The *net* module, a core Node.js library, was the other library we used for establishing socket connections on this side. It provides an asynchronous network wrapper and provides the necessary functionality to establish a TCP server that listens for connections and to connect to peers as a client. This is crucial for the P2P nature of our application, where each node can dynamically switch roles between client and server based on the network topology and communication needs.

4.2.2 Python Zeroconf & socket. Zeroconf, the python equivalent of *Bonjour*, is a python library based on Paul Scott-Murphy's *pyzerconf*. It too provides zero-configuration networking utilities. It allowed us to create a usable network over which we communicate with the Python listener instance. We also use the built-in Python socket package in order to interact with ports on both the publisher and listener ends. Together, these packages

4.3 Key Algorithms and Their Roles

For this project, we used multiple algorithms based on the use-case. This is to ensure security while maintaining a well-performant application. We explain the reasons for each algorithm below.

4.3.1 Diffie-Hellman Key Exchange. Implemented on both sides of the application, the Diffie-Hellman algorithm enables the secure derivation of shared secrets over the public channel. This is crucial for establishing a secure communication channel without prior sharing of secret keys. The nature of the session keys being unique and temporary, ensures forward-secrecy.

4.3.2 RSA Algorithm. Both the Python and JavaScript components use the RSA algorithm for digital signatures and mutual authentication. It allows for the encrypted exchange of session keys by forcing the other user to prove their legitimacy, by using their private key to decrypt the intial message. A key length of 2048 bits was selected for its strong security while still maintaining acceptable performance levels.

4.3.3 AES Encryption. For message encryption, we used the AES-256 standard in Cipher Block Chaining (CBC) mode, which provides a high level of security and efficiency. The speed of AES-256 makes it more suitable than RSA for computing large volumes of data, which is necessary during encryption and decryption. This key size is widely-used and considered secure against all known attack vectors. CBC mode is used for its effectiveness in obscuring data patterns.

5 PEER COMMUNICATION

Peer-to-peer (P2P) communication forms the backbone of our application and enables users to interact directly without intermediary servers. From start to finish, our messaging process is the following:

(1) Initial Handshake with Diffie-Hellman (DH): Peers begin communication by exchanging DH parameters, creating a shared secret for encryption without exposing it over the network. This process effectively guards against eavesdropping.
(2) Public Key Exchange: Following the handshake, peers exchange RSA public keys, allowing them to encrypt messages in a way that only the intended recipient can decrypt. Public key integrity and authenticity are ensured through digital signatures, preventing man-in-the-middle attacks.
(3) AES Symmetric Encryption: The shared secret from the DH exchange is then used to derive a symmetric AES key.

5.1 Message Exchange Protocol

Once a secure channel is established, peers exchange messages encrypted with the AES symmetric key. Each message's confidentiality is maintained through encryption, and its integrity and source authenticity are ensured by including RSA-based digital signatures to authenticate the sender and verify the message has not been altered.

5.2 Messaging Format

The format of messages exchanged between peers includes necessary metadata, alongside the actual encrypted message content, such as a timestamp of when it was sent, which we validate during decryption by rejecting messages arriving outside an acceptable 5-minute time window. This prevents replay attacks. More specifically, our message structure is a JSON with the following fields:

```
{
  "sender": "Alice",
  "recipient": "Bob",
  "timestamp": "2024-02-25T15:00:00Z",
  "content": "Encrypted message here"
}
```

6 DEVELOPMENT CHALLENGES

Throughout the implementation of the project, we faced a number of challenges. In this section, we will review the issues for each of the languages respectively.

6.1 JavaScript Challenges

The most prominent issue we faced involved a discrepancy in the AES keys derived from DH shared secrets between the discovery and publish functionalities. After extensive logging, we identified that when we originally used *crypto.createDiffieHellman(2048)*, we were creating a new DH instance with a newly generated 2048-bit prime and generator each time. This meant that every instance of the application (e.g., the publish side and the discover side) could potentially use a different set of DH parameters.

The solution was specifying the predefined "modp14" parameter for the DH exchange, a 2048-bit MODP (modular exponential) Group. This consistency made sure both parties derive the exact same shared secret by forcing both sides to use the same prime and generator. This experience showed the importance of standardized security parameters.

6.2 Python Challenges

Throughout the Python implementation, we encountered a few main challenges.

(1) Understanding the JavaScript Architecture: As the Python implementation was completed after the JavaScript portion, understanding the architectural approach and implementation was a significant hurdle. This task necessitated time to understand the requirements on the Python-end
(2) One of the most time-consuming challenges, requiring more than 30 hours, arose from a persistent bug in the key exchange process. This issue underscored the critical importance of consistent encoding and decoding practices at both ends of the communication channel. Debugging this problem required meticulous examination of the data at every stage of transmission and processing. This debugging was done all the way down to the byte-level to find the issue in *discovery.py*'s *key_exchange()* method.
(3) Final implementation compatibility with JavaScript was another long task. Despite both sides adhering to the same cryptographic protocols, numerous unforeseen issues emerged, such as encoding and timeouts. This emphasized that the protocol agreement alone does not guarantee seamless interoperability.

Through these challenges, we learned valuable lessons to some of the certain intricacies with crytographic and socket implementations, along with the importance of precise data handling, and the complexities of creating compatible systems across different programming languages.

7 SECURITY ANALYSIS

Using Diffie-Hellman Key Exchange with a 2048-bit key to establish a shared secret, which is then used to exchange a 256-bit AES key, is computationally secure by current cryptographic standards and practices. This approach combines the strengths of DHKE for securely exchanging keys over an unsecured channel with the robust encryption provided by AES-256 for messaging once the connection is established.

7.1 Diffie-Hellman Key Exchange (2048-bit)

DHKE is widely used for securely exchanging cryptographic keys over a public channel. A 2048-bit key size for DHKE is considered strong and aligns with current recommendations for secure key exchange. The National Institute of Standards and Technology (NIST) suggests a minimum of 2048 bits for DH keys to ensure security against current computational capabilities. This recommendation is outlined in publications: "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography"

(NIST SP 800-56A Rev. 3). DHKE is used in this application to exchange and establish the AES-key used for message encryption and decryption.

7.2 AES-256-CBC Encryption

AES with a 256-bit key is among the most secure encryption algorithms available today. The key length of 256 bits provides a high level of security, making it resistant to brute-force attacks with current and foreseeable technology. Further, when generating the AES key, we use SHA256 in order to further protect against collision attacks. As a whole, this process is validated through the Cryptographic Algorithm Validation Program (CAVP) managed by NIST for its security properties.

8 CONCLUSION

Our secure messaging application leverages advanced cryptographic protocols and algorithms to build a robust peer-to-peer communication. Implementing the Diffie-Hellman Key Exchange with a 2048-bit key alongside AES-256-CBC encryption, we have established a secure framework that ensures confidentiality, integrity, and authenticity of messages exchanged between peers. The choice of Python and JavaScript for development not only facilitated a streamlined development process but also leveraged the extensive cryptographic capabilities and community support available for these languages. Through the challenges encountered and overcome, this project has not only served as a practical application of cryptographic principles but also as an exploration into the complexities of developing secure communication systems.