White-Hat Summary

# **Introduction**

There are three key aspects of the SSL protocol: authentication, confidentiality, and integrity. We modeled our implementation after these aspects. For the authentication portion, public key ciphers sign and verify signatures for session keys. For the confidentiality portion, a symmetric key cipher encrypts and decrypts messages sent in the channel. For the integrity portion, a hash function and message authentication code algorithm both identify changes made to data in between the time it is sent and received.

Hence, those using our protocol can expect…

* their communication to be safely hidden from unintended parties
* to be communicating with their intended party, and not a pretender
* their communication to be untampered with

Our protocol implementation has an additional feature. Clients can choose which public key ciphers they wish to use with the server. This grants an extra level of independence from the protocol while still benefiting from the security it provides. Consider a client that does not think ElGamal is secure enough for the messages they want to send. They can opt to use RSA instead.

# **Authentication** - Asymmetric Key Ciphers

The possible asymmetric key ciphers the ATM and Bank can use include RSA and ElGamal. We use them for digitally signing keys. The keys can be employed in the algorithms for the symmetric cipher or for generating and verifying MACs.

## Rivest-Shamir-Adleman

On top of the *textbook* Rivest-Shamir-Adleman (RSA) cryptosystem, we augmented it by introducing random padding. Specifically, we implemented optimal asymmetric encryption padding (OAEP), as detailed in version 2.2 of Public-Key Cryptography Standards (PKCS#1 v2.2). This scheme allows us to

1) encode plaintext messages prior to encrypting them with RSA

2) decode ciphertext messages after decrypting them with RSA

To encode messages, we first combine them with some hash output, the number 1 represented as a byte, and a padding string. All these things get concatenated into a new string (DB).

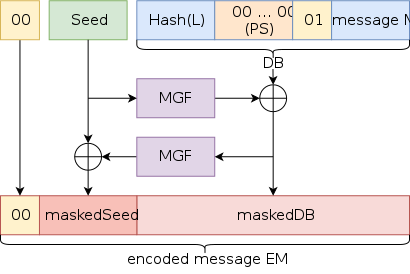
The hash output is, more precisely, the output of a chosen hash function (Hash) after passing it a string called label (L). The label is optional, so if one is not provided it will default to an empty string. In our implementation, we use this default.

This padding string (PS) is a string of null-bytes, with an exact length equal to k – mLen – (2\*hLen) – 2. Here, k equals the byte length of the modulus being used in the accompanying RSA algorithm, mLen equals the byte length of the message to encrypt, and hLen equals the byte length of any given output from Hash.

Next, we need to define a mask generation function. A mask generation function (MGF) is essentially a hash function, but it produces output of varying lengths. One generates a random value as the seed for the MGF, ensuring it has the same length as hLen. In our implementation, we utilize MGF1 which is a mask generation function defined in PKCS#1.

Next, DB is XOR’d with the output of the MGF. Note we are allowed to XOR these two because just like DB, the output also has k – hLen – 1 bytes. This acts as our way of “masking” DB. This result is now our new temporary seed for MGF. We use it to generate a string of length hLen. Now we can also mask the original random value seed by XOR’ing it with this newly generated string.

Finally, the encoding of the original message is the combination of the seed mask, DB mask, and an extra null-byte.



To decode messages, we basically reverse all the steps of the encoding algorithm. We want to ‘undo’ the masking that occurred by XOR’ing an already masked string and the mask output that was used to do the encoding. For example, the masking of DB = MGF(seed, k – hLen - 1) ^ DB. Thus, the demasking of this masking…

= masking ^ MGF(seed, k – hLen - 1)

= (MGF(seed, k – hLen - 1) ^ DB) ^ MGF(seed, k – hLen - 1) by substitution

= (MGF(seed, k – hLen - 1) ^ MGF(seed, k – hLen - 1)) ^ DB by associativity

= 0 ^ DB = DB. by identity

First, we compute the hash of the given label, which is an empty string. Then from the encoded message, extract the first byte as Y, the next hLen bytes as the masked seed, and the remaining k – hLen – 1 bytes as the masked DB. Compute MGF with masked DB as the input and a desired length of hLen. This is the mask that was used to create the masked seed value. Hence, we can determine the original seed by XOR’ing the masked seed and this newly found mask. Similarly, compute the DB mask by passing this original seed and a desired length of k – hLen - 1 into the MGF. Then we can also determine the original DB by XOR’ing the masked DB and this newly found mask.

Now that we have the original DB, we can extract the original message from the last hLen bytes.

The OAEP scheme uses random oracles (RO) in the form of the mask generating and hash functions. We inject randomness via these random oracles. Such randomness turns the subsequent RSA encryption probabilistic. Hence, the implementation is **semantically secure** against chosen plaintext attacks (**IND-CPA**). Note that a cryptosystem which is semantically secure implies it is secure against the **ciphertext-only** attacker model. This is another benefit to making our public key cipher semantically secure.

By encoding a message before feeding it into RSA’s trapdoor permutation function, the overall encryption process becomes **plaintext aware**. To produce any ciphertext, one must know its associated plaintext.

Because our implementation is both semantically secure and plaintext aware, it is also secure against chosen ciphertext attacks (**IND-CCA1**). An attacker with access to encryption and decryption oracles cannot gain a meaningful advantage in deciding which plaintext corresponds to a selected (by a challenger) ciphertext. Note that in this context, the attacker loses their decryption oracle once the challenge is issued.

Using OAEP with RSA is also secure against adaptive chosen ciphertext attacks (**IND-CCA2**). This is essentially the same as IND-CCA1 except now the attacker can *still* use their decryption oracle even after the challenge ciphertext was declared, as long as they do not submit the challenge ciphertext itself.

# **Confidentiality** - Symmetric Key Ciphers

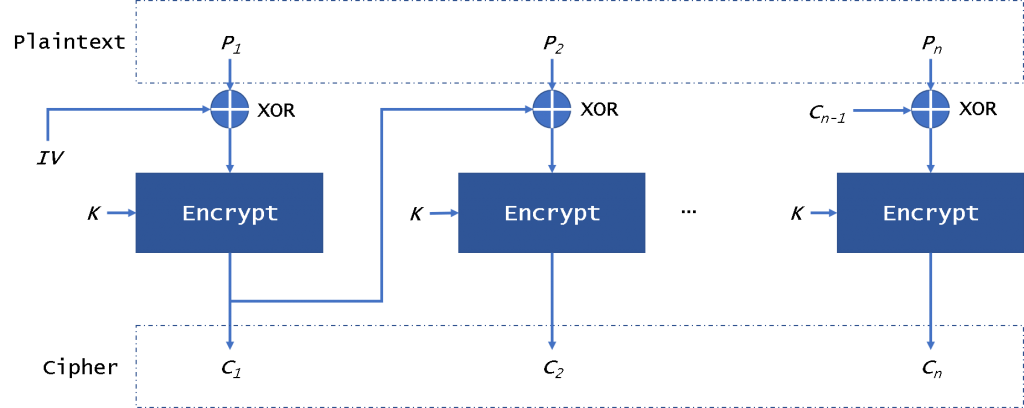
The ATM and Bank employ a symmetric key cipher called AES. It is used for encrypting and decrypting messages sent in a secured channel.

## Advanced Encryption Standard

We chose AES because of its status as the most widely used symmetric key cipher. It is endorsed by the NSA and multiple standards organizations.

AES has a variety of “modes” it can be utilized in. Electronic Code Book (ECB) encrypts all blocks of the plaintext in the same way, with the same key. It would have been the easiest mode to code but also would have been the easiest mode to take advantage of. The same plaintext leads to the same ciphertext, so patterns in the ciphertext reflect patterns in the plaintext. For example, if an attacker always saw ‘a’ followed by a varying number of ‘b’s, they might guess that ‘a’ is a 1 and ‘b’ is 0. They can reasonably guess the amount of money a user is depositing and withdrawing without ever looking at the commands in plaintext. Cipher feedback mode (CFB) first encrypts an initialization vector then XORs it with the first plaintext block. Then the resulting ciphertext block is encrypted and the result is XORed with the next plaintext block. This repeats until all the plaintext blocks are used. Counter (CTR) is a mode that encrypts some counter value and XORs the result with the plaintext. This mode requires the counter to be a value that is from a greater sequence where repeats are rare.

We implemented AES in cipher block chaining mode (CBC). Although similar to CFB, we felt CBC still had the easiest implementation with the least disadvantages when compared to other modes. It is more complex than ECB and can propagate errors just like CFB (making them equivalent enough). A major red flag for CBC mode, however, is its susceptibility to padding oracle attacks. This stems from the requirement that plaintext be padded to a length which is a multiple of 16. Introducing padding means introducing a new tool that attackers can use - padding oracles. These tell an attacker if the ciphertext they submitted is a valid padding or not. Hence, they can use the client or server as a padding oracle to gain information about parts of a ciphertext. Using these parts together, they may eventually decrypt the ciphertext entirely. We knew about this huge weakness in the mode but decided to implement it anyway.



During the encryption process, we chose to do 14 rounds. This is the maximum number of rounds typically done. The more rounds there are, the more complex the ciphertext becomes. We also made sure to expand the key to 14 bits so that the corresponding key space an attacker would have to search through is larger. Our S-box is larger to account for more variety in the substitutions. There are more possibilities that a state byte can turn into, thus further complicating the encryption.

# **Integrity**

## Hash Function

The code implements the SHA-1 hashing algorithm, which is a cryptographic hash function that produces a 160-bit (20-byte) hash value. The SHA-1 algorithm operates on messages of up to 2^64 bits and processes the message in 512-bit blocks.

## Message Authentication Code

The code also implements the HMAC algorithm, which is a mechanism for message authentication using a cryptographic hash function in combination with a secret key. The code uses four constants, called K values, as inputs to the SHA-1 algorithm.

These K values are used in the calculation of the intermediate hash values and are specific to the SHA-1 algorithm. The code includes several helper functions, including a padding function that pads the message to be hashed to ensure it is a multiple of 512 bits, a function for circular left shift rotation, and a function that applies the SHA-1 algorithm to the padded message to generate the hash value.

Finally, the code includes a function for generating a secret key for use with the HMAC algorithm.

# **\*Homomorphic Cipher**

The ATM and Bank use .

# **\*Digital Banking -** Introduction

# **Digital Banking -** ATM

The code in atm.py defines a class called ATM. This class represents an Automated Teller Machine and allows a user to connect to a bank server via a socket and perform transactions on their bank account.

The code imports various libraries such as json, hash, socket, ast, secrets, and some classes from other files such as rsa, elgamal, and aes.

The constructor \_\_init\_\_ initializes various instance variables such as aeskey, mackey, p, prefs, counter, id\_num, and a socket object s. The s.connect(('127.0.0.1', 5432)) statement establishes a connection to a server with IP address 127.0.0.1 and port number 5432.

The countercheck function checks if a message is tampered with or if the counter is less than or equal to the current counter.

The post\_handshake function handles the exchange of messages between the client and server to establish a secure connection. It also authenticates the user by asking for their username and password, encrypting and hashing them, and sending the result to the server.

The ATM class provides a command-line interface to the user, where they can perform various transactions such as deposit, withdraw, and check balance.

The code uses various encryption and hashing techniques such as AES encryption, HMAC, and SHA1 hashing to ensure the security of the communication between the client and server.

# **Digital Banking -** Bank

The code in bank.py is an implementation of a banking server that listens for requests from ATM clients. The Bank class is the main component of the server, and it contains methods for handling client requests such as withdrawal, deposit, and balance checks.

To summarize, the Bank class initializes by reading two JSON files that contain user data: usertohashpass.txt and usertomoney.txt. It also sets up a list of available public key encryption methods (RSA and ElGamal), initializes some cryptographic variables, and creates a TCP socket to listen for incoming connections from ATM clients.

The countercheck method checks if the message received from the client has a counter value greater than the server's counter value. This is used to prevent replay attacks, where an attacker captures and resends a previously sent message.

The withdraw, deposit, and check methods handle client requests for withdrawing money, depositing money, and checking account balances respectively. Each method sends a response back to the client after encrypting the response and appending a HMAC (hash-based message authentication code) to ensure message integrity.

The post-handshake method is called after a client connects to the server and completes a handshake. This method exchanges a counter value with the client to ensure that both sides are synchronized. It also sets a flag to indicate that the client is now logged in.

The main loop of the Bank class listens for incoming commands from the client and dispatches them to the appropriate method. Each command is decrypted, verified for message integrity using the HMAC, and checked for replay attacks using the counter value. If any of these checks fail, an exception is raised.

# **Digital Banking -** Conclusion

The code consists of two parts, one for simulating an ATM and one for managing bank accounts. The ATM simulation allows the user to withdraw and deposit money, and also displays the account balance. It has built-in error handling for invalid inputs and insufficient funds. The bank account management code allows the user to create new accounts, deposit and withdraw money, and display the account information. It also has error handling for invalid inputs and negative balances.

Overall, the code provides basic functionality for a banking system, but it lacks more advanced features such as transaction history, interest rates, and user authentication. Additionally, the code could benefit from better organization and separation of concerns, such as creating separate classes for the ATM and the bank account management.