Hacettepe University

Computer Engineering Department

BBM465 Information Security Laboratory - 2024 Fall

Homework Assignment 2

November 10, 2024

Group ID: 28
Students name:
Mehmet Yiğit
Mehmet Oğuz Kocadere

Students Number:
b2210356159
b2210356021

1 Problem Definition

The goal is to understand basic cryptographic techniques through practical application. The assignment consists of two main parts:

1. **Creating Public/Private Key Pair** : The program generates a 2048 bit RSA public/private key pair and creates a self-signed X.509 public key certificate with the help of Java keytool. The private key is encrypted with AES, using an MD5 hash of a user-provided password as the AES key. The certificate and encrypted private key are stored and used for future registry and integrity verification processes.

2. **Creating Registry File** : The program records each file path and its hash value (MD5 or SHA-256) within the monitored directory in a registry file. After hashing each file, the registry file is signed using the private key to maintain data integrity. All actions, including added files and completion messages, are logged with timestamps.

3. **Checking Integrity** : The program verifies the registry file by checking the signature with the public certificate; if the verification fails, it logs an error and exits. If verification is successful, the program checks for any additions, deletions, or modifications within the monitored directory. All detected changes, or a "no changes detected" message, are logged with timestamps to document the directory's current state.

2 Creating Public/Private Key Pair Implementations

2.1 createKeyPair

```
1 //You can run this code from the console with this command
2 java ichecker createCert -k PriKey -c PubKeyCertificate
```

It first prompts the user to enter a password for encrypting the private key. This password is read from the console using a Scanner. It then generates a key pair (private and public keys) and a self-signed certificate using the keytool command-line tool. The key algorithm used is RSA with a key size of 2048 bits, and the generated certificate is valid for one year (365 days). The keystore, which stores the private key, is created at the path provided by the privateKeyPath argument, and the keystore password is provided via storePassword. The certificate is then exported from the keystore using another keytool command, and the exported certificate is saved at the location specified by certificatePath. After the key pair and certificate are successfully created, a line "PRIVATEKEY" is appended to the private key file to indicate that it contains the private key. The private key file is then encrypted using the password entered by the user (keyPassword) and the provided initialization vector (iv) to ensure the key is securely stored. The function utilizes helper methods, including executeCommand, to run the keytool commands and encryptPrivateKeyFile to perform the encryption of the private key file. Throughout the process, exceptions are caught, and errors are handled with appropriate messages, ensuring the user is notified if something goes wrong during key generation, certificate export, or file operations.

2.2 encryptPrivateKeyFile

The `encryptPrivateKeyFile` function encrypts a private key file using AES encryption. First, it creates a new file, `encryptedPrivateKeyFile`, to store the encrypted data alongside the original private key file. The function then reads the private key file into a byte array, `privateKeyData`. This data is encrypted using the `aesEncrypt` method, which applies AES encryption with a password (the `key`) and an initialization vector (IV) to provide security. The encrypted byte array is then written to `encryptedPrivateKeyFile`, effectively securing the original private key data in an encrypted form. If any issues arise during the process, the function throws an exception to handle errors.

2.3 aesEncrypt

The `aesEncrypt` function performs AES encryption on input data using a specified password and initialization vector (IV). First, it generates an encryption key by applying an MD5 hash on the password, creating a fixed-length key for AES encryption. Using the AES algorithm in CBC mode with PKCS5 padding, the function sets up a `Cipher` instance and initializes it with the encryption key and IV. The `cipher.doFinal(data)` method then encrypts the data, producing `encryptedData`. The function combines the IV and the encrypted data into a single byte array, `result`, to ensure the IV is stored with the encrypted content for decryption purposes. Finally, it returns this combined byte array as the encrypted result.

3 Creating Registry File

3.1 createRegistry

```
//You can run this code from the console with this command
java ichecker createReg -r RegFile -p Path -l LogFile -h Hash -k PriKey
```

This `createRegistry` function creates a registry file by hashing files in a directory and adding a digital signature. The function first prompts the user to input a password to decrypt a private key file, which is essential for signing the registry. The password is validated to ensure it's not empty and has a specific length. After verifying the password, the function decrypts the private key using AES decryption and saves it temporarily in a new file. Then, the function reads the decrypted private key and checks for the presence of a "PRIVATEKEY" label to ensure it's correct.

Once the private key is verified, the function scans the specified directory, hashing each file using either MD5 or SHA-256, as specified by the `hashAlgorithm` parameter, and records each file path with its hash value in a `registry` map. This registry data is converted to a string format for writing to a file. The function then uses the decrypted private key to sign the registry content with an RSA-based SHA-256 signature. Finally, it writes both the registry data and the encoded digital signature to the registry file, logging each file added to the registry and confirming completion. If any errors occur during the process, appropriate error messages are printed, and the function may exit in some cases.

3.2 aesDecrypt

The `aesDecrypt` function decrypts data encrypted with AES using a specified password. It first generates a 128-bit key by hashing the password with MD5. The initialization vector (IV), which is the first 16 bytes of the data, is extracted separately from the encrypted data. The rest of the data (excluding the IV) is the actual encrypted content. A `Cipher` object is created to perform AES decryption in CBC mode with PKCS5 padding. The cipher is initialized with the generated key and IV, and then the encrypted data is decrypted using the `doFinal` method. Finally, the decrypted byte array is returned. If an error occurs during the decryption process, an exception is thrown.

4 Checking Integrity

4.1 checkIntegrity

```
6 //You can run this code from the console with this command
7 java ichecker check -r RegFile -p Path -l LogFile -h Hash -c PubKeyCertificate
```

The `checkIntegrity` function verifies the integrity of files in a specified directory by comparing their hashes to those stored in a registry file. First, it checks the validity of the registry file by verifying its signature against a provided certificate using the `verifySignature` method. If the registry signature is invalid, an error message is logged, and the program terminates. Next, the function reads the registry file into a list of lines, parses each line to build a map (`registry`) where the file paths are mapped to their corresponding hash values.

The function then walks through the files in the given directory, computes their hashes using the specified hash algorithm (MD5 or SHA-256), and stores the results in a `currentFiles` map, with file paths as keys and their corresponding hash values as values. Afterward, it compares the file hashes in `registry` with those in `currentFiles`. If a file is missing, altered, or newly created, it adds a message to the `changes` list, indicating the type of change (e.g., "deleted," "altered," or "created").

Finally, if any changes are detected, the function logs them in the specified log file and prints a message to the console. If no changes are found, it logs a message indicating that the integrity check was successful with no modifications.

4.2 verifySignature

The `verifySignature` function verifies the authenticity of a registry file by checking its digital signature using a specified certificate. It starts by reading the contents of the registry file and separating the actual content from the signature, which is stored in the last line. The signature is extracted and decoded from Base64 format.

Next, the function loads the certificate from the provided file path (`certificatePath`) using the `CertificateFactory` class and retrieves the public key from the `X509Certificate`. Using this public key, a `Signature` object is initialized with the "SHA256withRSA" algorithm to verify the signature.

The content of the registry (excluding the signature) is updated in the `Signature` object, and the `verify()` method is called with the decoded signature. If the signature is valid, the function returns `true`; otherwise, it catches a `SignatureException` and returns `false`. This function ensures that the integrity of the registry file is intact and that it hasn't been tampered with.

5 Helper Functions

5.1 hashFile

The `hashFile` function computes the hash of a file using a specified hash algorithm, such as MD5 or SHA-256. It begins by selecting the appropriate `MessageDigest` instance based on the provided algorithm (either "MD5" or "SHA-256"). The function then opens the file at the given `filePath` using a `FileInputStream` and reads the file in chunks of 8192 bytes. For each chunk, the data is passed to the `hasher.update()` method, which continuously updates the hash value as it processes the file. After reading the entire file, the hash is finalized by calling `hasher.digest()`. The resulting hash is then encoded in Base64 format and returned as a string. If an error occurs during any part of the hashing process, an exception is thrown.

5.2 logMessage

The `logMessage` function logs a given message to a specified log file, prefixed with a timestamp. It begins by generating a timestamp using the `SimpleDateFormat` class, formatted as "dd-MM-yyyy HH:mm:ss" to represent the current date and time. The function then opens the log file for appending by creating a `BufferedWriter` wrapped around a `FileWriter` in append mode. The log entry, consisting of the timestamp followed by the provided message, is written to the log file. After writing the message, the `BufferedWriter` is automatically closed due to the use of a try-with-resources statement, ensuring proper resource management. If any file operation errors occur during the logging process, an `IOException` is thrown.

5.3 getPrivateKey

This function, `getPrivateKey`, retrieves a private key from a keystore stored in PKCS12 format. It begins by initializing a `KeyStore` instance that uses the "PKCS12" type, which is a commonly used format for storing private keys. The function then opens the specified file path (`privateKeyPath`) containing the keystore using a `FileInputStream` and loads the keystore into the `keyStore` object using a password (`storePassword`) to access it. After loading, it retrieves the private key from the keystore using the alias `"ichecker-cert"` and the same `storePassword` for decryption. If any errors occur during this process, such as issues with loading the keystore or accessing the private key, an exception is thrown. Finally, the private key is returned as a `PrivateKey` object.

5.4 md5Hash

This function generates an MD5 hash of the given password by using the `MessageDigest` class from Java's security package. It first obtains an instance of the MD5 hashing algorithm using `MessageDigest.getInstance("MD5")`. Then, the password is converted to a byte array using the UTF-8 character encoding and passed to the `digest()` method, which returns the MD5 hash as a byte array. If the MD5 algorithm is unavailable, a `NoSuchAlgorithmException` is thrown. The resulting hash can be used for secure password storage or comparison.

6 ichecker.main

The `main` method is the entry point for executing various commands in the `ichecker` program. It first initializes a 16-byte initialization vector (`iv`) for cryptographic operations, generated using `SecureRandom`. If the command-line arguments are insufficient (less than 2), it prints a usage message and exits the program. The first argument, `command`, is extracted, and based on its value, the program enters one of several `switch` cases to execute a specific functionality.

For each command (like `createCert`, `createReg`, or `check`), the method parses the additional command-line arguments to obtain necessary paths (e.g., private key, certificate, registry file, directory) and other parameters (e.g., hash algorithm). Depending on the command, it calls different methods such as `createKeyPair`, `createRegistry`, or `checkIntegrity`, passing the extracted arguments and the initialization vector (`iv`) for cryptographic operations.

Before executing the commands, the method prompts the user for a password (used for keystore encryption) and stores it in a variable. A `Scanner` is used to read the password from the user input. If any exceptions occur during the execution, such as unknown commands or errors in argument parsing, the program catches them and prints a generic error message ("Unknown command.") to the console.

In summary, the `main` method orchestrates the execution of different commands based on the input arguments, handles user input for passwords, and ensures proper error handling.

Example how to include and reference a figure: Fig. 1.

References

- BBM465 Lecture 6 Digital Signatures, Public Key Certificates, X509

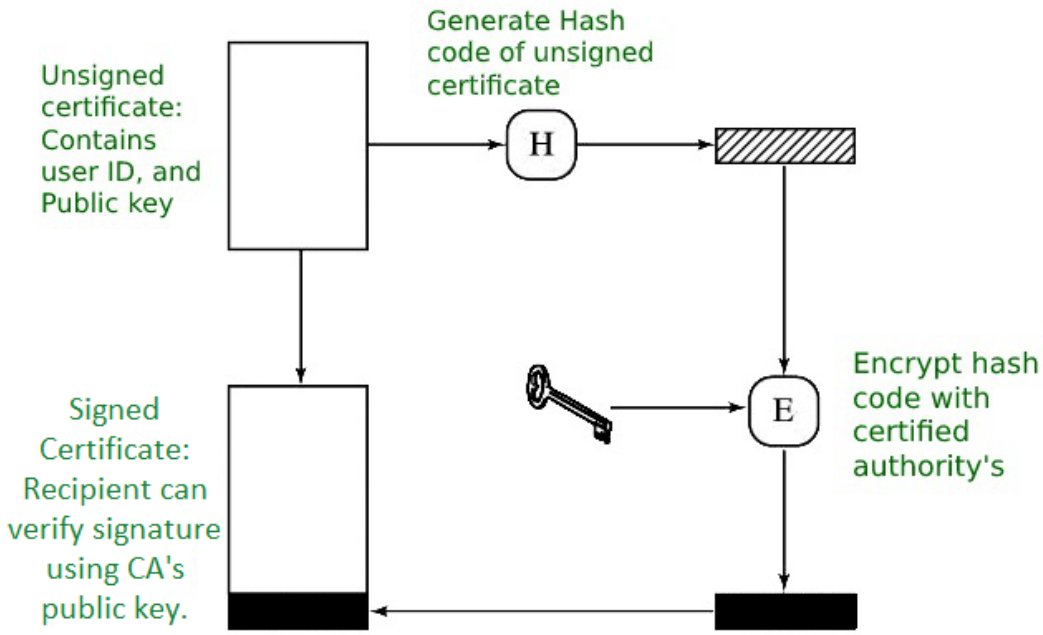

Figure 1: Symbolic and conceptual representation.