Java SE Security
Lab 3

1 Lab setup

Make sure you have the following items installed

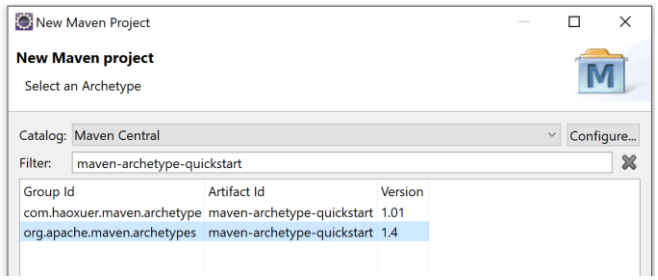
- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Eclipse Enterprise Edition or Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A suitable hex editor (HxD Hex Editor)
- A utility to extract zip files

2 MD5 and SHA hashes

The source code for this lab is in the `Hash-Basics-Demo` folder.

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter in the following details and click Finish.

Group id: `com.workshop.basics`
Artifact id: `HashBasicsDemo`
Version: `0.0.1-SNAPSHOT`
Package: `com.workshop.basics`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

In the package `com.workshop.basics`, place the following files:

```
BasicUtils
HashUtils
FileUtils
MD5SHADemo
```

Place this file in the project root folder:

```
plaintext.txt
```

An instance of the MessageDigest class is used for the various hashing algorithms to compute the hash on a byte sequence.

Run the app

The contents of the file `plaintext.txt` will be read and a hash computed on it. Copy the hash value (in the form of a Hex string) into a text editor such as NotePad++.

Open `plaintext.txt` and randomly modify a single character anywhere in the file. Save it, stop the app and rerun it again. Note down the new hash value computed and compare it with the previous hash value. Notice that just changing a single character in the message, regardless of the length of the message, results in a significant change in the hash computed. This is a key characteristic of all hashes which is very useful in helping ensure integrity of a message.

You can verify the MD5, SHA-256 and SHA-512 hashes generated in the code implementation using this online generator:
https://passwordsgenerator.net/md5-hash-generator/

Many open source projects provide hashes and/or signatures for the downloadable artifacts from their project website. This are typically bundled in the standard archive formats (*.zip for Windows, *.tar.gz for Linux). These hashes are usually use MD5 or a variant of SHA, while the signatures are

based on the PGP (Pretty Good Privacy) cryptography scheme. We will look at signatures in a later lab.

Navigate to the download page of Apache Commons Codec
https://commons.apache.org/proper/commons-codec/download_codec.cgi
Select any one of the binary or source distributions to download.
Open File Explorer in the project root folder and copy the downloaded file here.

Open the corresponding sha512 page for the file you downloaded and note down the Hex string sequence into a text editor such as NotePad++.

Provide the relevant details to the running program, for e.g.:

Name of file to compute hash on : `commons-codec-1.15-bin.zip`
Name of hash algorithm to use : `SHA-512`
Precomputed hash provided (enter for none) : `2495a5dcc2e572` ……

The program should verify integrity by confirming that both hashes (from the sha512 page and the one actually computed on the downloaded file) match. You can copy the computed hash and place it next to the provided hash and manually compare them character by character to determine that they are identical if you wish.

You can also verify the hashes on a file using an online generator:
https://md5file.com/calculator
https://emn178.github.io/online-tools/sha512_file_hash.html

The vast majority of Maven dependencies on the primary Maven repository will also have hashes to verify their integrity:
https://repo1.maven.org/maven2/

For e.g. if you navigate to:
https://repo1.maven.org/maven2/junit/junit/4.9/

You can download the various jar files and verify them using either a MD5 or SHA-1 hash. For e.g. you could download: `junit-4.9-javadoc.jar` and view its corresponding hashes at: `junit-4.9-javadoc.jar.md5` and `junit-4.9-javadoc.jar.sha1`

The rerun this program and verify it again by entering the appropriate details:

Name of file to compute hash on : `junit-4.9-javadoc.jar`
Name of hash algorithm to use : `MD5`
Precomputed hash provided (enter for none) : `6a3ffa9a70633bf723d3dc625b32824a`

You can repeat this for a few other random samples on the main Maven repository site or any one of the download pages for the Apache open source projects.

The last part of this program demonstrates that the size of the hash (in bytes) is fixed based on the specific hashing algorithm being used (MD5, SHA-256, SHA-512, etc), regardless of the length of the message being hashed.

From JDK 9 onwards, the latest SHA-3 family of hashing algorithms are supported and these are considered cryptographically more secure than SHA-2, although SHA-2 has yet to been broken in practice.

- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512

Keccak-256 is another popular SHA3-256 hashing algorithm. You can use this via the BouncyCastle provider:

```
Security.addProvider(new BouncyCastleProvider());
MessageDigest digest = MessageDigest.getInstance("Keccak-256");
byte[] hashSequence = digest.digest(originalString.getBytes());
```

Hashes are the foundation of blockchain technologies: for e.g. Bitcoin uses SHA-256 while Ethereum uses Keccak-256.

3 HMAC and Checksums

In the package `com.workshop.basics`, place the following files:

```
CryptoUtils
HMACCRCDemo
```

And make changes to the following files:

```
HashUtils-v2
```

Run the app.

We use the MAC class to compute the HMAC. Notice that the we use the same algorithm (HmacSHA512) for generating the key and initializing the MAC. This is not compulsory, but it is recommended for most cases.

The contents of the file `plaintext.txt` will be read and a CRC32 checksum computed on it. Copy the CRC value (in the form of Long number) into a text editor such as NotePad++.

Open `plaintext.txt` and randomly modify a single character anywhere in the file. Save it, stop the app and rerun it again. Note down the new checksum computed and compare it with the previous value. Notice that just like in the case of a hash, changing a single character in the message, results in a significant change in the checksum computed. This demonstrates how a checksum can be useful in helping ensure integrity of a message.

There is a second implementation of computeCRC32 which uses a CheckedInputStream and a byte array to read a certain amount of bytes from the file and process the computation of the checksum in a gradual manner. This is slightly more efficient than the first approach and is useful when memory is

limited and the file size is extremely large. The final computed checksum is the same regardless of the approach used.

4 Using hashes and KDFs (PBKDF2) as cryptographic keys

In the package `com.workshop.basics`, place the following files:

`HashesForKeysDemo`

We have seen that AES secret keys are random byte sequences of specific lengths: 128, 192 or 256 bits (16, 24 and 32 bytes, respectively). Since hashes produce random byte sequences from strings, we can use them in simple cases to create keys.

The DigestUtils class in `commons-codec` provides simplified methods to work with message digests so this is a valid alternative to the traditional MessageDigest class. We use some of these methods here:
https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/DigestUtils.html

However, the most secure and correct way to create a secret key from a string (such as a password) is using the Key Derivation Function (KDF), of which the most recent version is PBKDF2. The security of this secret key is achieved via configuration: bit length and the number of repetitions.

Run the app.

Notice there is a slight pause in the generation of the secret key using PBKDF2, compared to with standard key generation in the case of the MD5 and SHA256 hashes. This demonstrates the trade-off between increased security and decreased performance.

5 Secure password hashing

Since password tables (lists of usernames and passwords) are a prime attack target for hackers, passwords should never be stored in plaintext. Passwords are also not typically encrypted because the passwords will then be known to the possessor of the private key (usually the system administrator). This violates the confidentiality principle that requires the password be only known to the individual who created it or whom it was assigned to.

Passwords are conventionally stored as hashes, matched with a particular username or account name. This ensures that if the list of username/password hashes is compromised on a system, the actual password cannot be obtained since it should be impossible in theory to reverse a hash to obtain the original string.

A brute force attack would involve computing hashes on all combination of printable characters in different lengths and comparing these computed hashes with the known hash of an actual password. While such an approach would eventually succeed, it would consume an intractably long amount of time making it impractical.

A faster approach is to use a rainbow table. This is a precomputed dictionary of character strings and their corresponding hash values. This eliminates the additional computational cost of generating a hash in real-time: now, we simply need to check for the hash that we are attempting to crack to see whether it is stored in the table.

A comprehensive list of such tables can be found at:
https://project-rainbowcrack.com/table.htm

https://passwordsgenerator.net/md5-hash-generator/
Using the online hash generator above, generate the hashes using any algorithm of your choice (MD5, SHA1, SHA256, SHA512) for these simple words (or any other standard English words you can think of):

```
cat
donkey
superman
```

Take the generated hash and paste it into the password cracker below and see how long it takes to crack the hash:
https://crackstation.net/

Because these are words that can be found in the dictionary, they will be part of the rainbow table and the cracking is almost instantaneous.

Generate a random password using the same online site of 6 characters in length. Now with a password that is comprised of a random arrangement of characters that are not in the dictionary, compute a hash on it using the weakest algorithm (MD5). Try to crack the hash again. This time you will see that the hack attempt fails (at least in this instance, where the rainbow table size is still small).

This is why you will see the standard advice on creating strong passwords typically stress on length, using random combination of a wide range of characters and avoiding dictionary words.

https://blog.avast.com/strong-password-ideas
https://www.howtogeek.com/195430/how-to-create-a-strong-password-and-remember-it/

The recommendation to change passwords frequently is also based on the concept that if a hacker manages to gain access to the list of password hashes and attempts a brute force attack to crack them (which will eventually succeed given enough time), by the time he succeeds, the password for that account would have changed. The drawback to changing passwords frequently is people are not capable of creating and remembering good passwords, and the net result may be the deterioration of the password strength over multiple changes.

https://www.howtogeek.com/187645/htg-explains-should-you-regularly-change-your-passwords/
https://www.maytech.net/blog/why-regularly-changing-password-puts-you-at-risk-of-attack

To further increase the strength of password hashes, the conventional approach accepted in the security community is to:

- Add random salt (a nonce) to the password before hashing it

- Use a secure password hashing algorithm (PBKDF2, BCrypt or SCrypt) along with HMAC-SHA-256 as the core hashing algorithm
- Increase the number of iterations on the algorithm (80,000 or more is recommended). The higher number of iterations significantly slows down the time to perform a single hash computation, which makes a brute force attack or the computation of rainbow tables for attacks much more time consuming.

In the package `com.workshop.basics`, place the following files:

```
PasswordHashDemo
PasswordEntry
```

Run the app.

Here, we simulate the process of registering an account and password that is typically encountered on many websites. The focus here is on showing the mechanism of hashing, storing and verifying the passwords, so we use a simple command line approach rather than a typical web approach which would require a Spring MVC app to implement properly. The PasswordEntry class represents a single entry in the password list, so we could annotate it with the @Entity JPA annotation if we wished to store these entries in a proper backend database using Spring Data JPA (or some other suitable JPA related technology).

6 Working with digital signatures

A digital signature is simply the hash of a message encrypted using the private key of a key pair. The message, the signature, the corresponding public key, and the algorithm are sent to a receiver. This is classified as a message with its digital signature. To check the digital signature, the message receiver generates a new hash from the received message, decrypts the received encrypted hash using the public key, and compares them. If they match, the signature is said to be verified.

In this process, the message is sent in clear text form, so signature by themselves only guarantee integrity of the signed message as well authentication the person who signed it. Signatures DO NOT provide confidentiality. In order to accomplish that, we can optionally encrypt the message that we are sending to the receiver if wish. Typically this is done using a session key that is negotiated between the sender and receiver once the receiver has received the public key of the sender via a certificate.

In the package `com.workshop.basics`, place the following files:

```
PasswordHashDemo
PasswordEntry
```

There are two ways to compute a digital signature and verify it.

We can do it manually by first producing a hash on the message to transmit, and then encrypt the hash with the private key of a key pair. For verification on the receiving end, we compute a hash on the message received and then decrypt the encrypted hash using the public key of the same key pair. Then we compare the decrypted hash and the newly compute hash: if they match, the signature is verified it.

Alternatively, we can initialize the Signature class with the message and private key in order to generate a signature. Then to verify, we initialize the class with the message, private key and generated signature. The actual hashing and encryption/decryption operation is encapsulated in this class, so we don't have to explicitly do it ourselves. This is the recommended approach.

Run the app.

Notice that the encrypted hash of the first approach differs from the signature of the second approach, even though they are both functionally identical. This is because the Signature class performs some additional encoding of the encrypted hash internally before it produces the final signature. Therefore, we should stick to only one of these approaches and use it consistently on both the sending and receiving ends.

Rerun the app.

After both signature and encrypted hash have been written to files and there is pause in program execution, change either the contents of the plaintext file or the contents of the signature files. This time the app will flag an unsuccessful signature verification.
You can even try changing the contents of the plaintext file and replacing the signature file contents with a hash computed on the new plaintext contents using an online SHA-256 generator:
https://md5file.com/calculator
https://emn178.github.io/online-tools/sha512_file_hash.html
Even with this attack, the verification process still fails because the hash is encrypted, and not in cleartext form.

7 Session key exchange with RSA

A key feature of asymmetric encryption is that it is used to solve the primary problem of symmetric encryption: key distribution. This concerns how to transport the secret key of symmetric encryption securely between two parties in such a manner so that it cannot be copied or stolen by a third party adversary, who can then subsequently decrypt all communications between the original two parties.

Although we can use the public key of a key pair to encrypt communication so that only the holder of the private key can decrypt it, encryption in this fashion is much slower compared to standard symmetric encryption. Therefore, the conventional approach for two parties A and B to communicate is for A to generate a symmetric secret key (or some random byte sequence that can be used to construct a secret key) and use B's public key to encrypt this secret key. The encrypted key is then transmitted to B. Since only B's private key can decrypt it, there is no way for anyone else to gain knowledge of that key. Once B has decrypted the secret key with his private key, this key is then used to encrypt all further communication between A and B. The shared secret key known only to A and B is known as the session key. Using the session key instead of the public key for all future encrypted communication between A and B encryption permits much faster and efficient encryption and is the basic underlying idea behind the SSL / TLS handshake sequence.

In this program, we simulate the creation and use of this session key.

In the package `com.workshop.basics`, place the following files:

`SessionKeyDemo`

Run the app. Trace through the sequence of actions to understand clearly what happens at each stage of establishment and use of the session key.