SE448: Information and Cyber Security	**Recitation 4**
Semester 2 5785	**10 April 2025**
Lecturer: Michael J. May	**Kinneret College**

Using Hashes and HMAC

We will start this week's recitation with some experiments on various file types to see how they interact with hash functions. Notice how with certain files there is no correlation between the viewable content and the hash value (*i.e.* two files with the same viewable user content have different hash digest values). We will then see how the Java library supports hashing and message authentication codes (MAC) in code.

1 Hashing via Java

In class we talked about hash functions: MD5, SHA-1, SHA-2, SHA-3. SHA-2 and SHA-3 are actually families of hash functions. For instance, SHA-2 includes SHA2-256 and SHA2-512. Java offers implementations of MD5, SHA-1, SHA-2, and SHA-3 for you to use to compute hashes. We delved into the details of SHA-1 to give a taste for how hash functions actually compute digests. You can learn more about SHA-3 at this link: `https://www.youtube.com/watch?v=JWskjzgiIa4`.

Hash verification is fairly simple - just examine whether the computed hash value is equal to the value it is supposed to have.

1.1 A Note on Versions

If you're using the Java 8 JDK or JRE, you're not going to get access to SHA-3 algorithms. They were added only later. For this recitation, therefore, you must use a modern version of the JDK and JRE. I recommend Java 17 or later.

2 Hashing and HMAC Tool

An empty command line program is provided as a starting point for your work. The program contains a helper method that parses command line arguments for use. The tool takes three arguments that must be provided with the following parameters:

operation "validate" or "compute" When requesting compute, the tool will output the digest for the file. When requesting validate, the tool will output "OK" or "Bad".

alg The algorithm to use for the computation or validation. Options should be MD5, SHA1, SHA-256, SHA-512, SHA3-256, and SHA3-512 for hash functions. For HMAC, it should support HmacSHA256, HmacSHA512, HmacSHA3-256, HmacSHA3-512.

inFile The file to process

key Used for HMAC processing. The key is a hexadecimal string.

inDigest Used for validate. The inDigest is a hexadecimal string. The tool compares the calculated digest with the one provided as input.

3 Hashing Files

When hashing a file, you must read it into the tool block by block and provide it to the `MessageDigest` object. Processes the file provided using the selected hash algorithm. The hash algorithms take a `byte[]`

to hash, so we can read the file one block at a time and provide it to the hash algorithm using the `update` function. Once we're done reading the file, we can use the `digest()` method to get the final digest.

Note: If you have a small file, you can read the entire file into a single `byte[]` using `Files.readAllBytes()` and then just the `digest(byte[])` method.

3.1 Verifying Hashing Files

The verification process for a file is almost identical to the process of computing a hash. Process the file as you would when hashing it, then compare the received digest with the one provided by the user from the command line (*i.e.* inDigest). If the digests are identical, output OK. Otherwise, output Bad.

4 HMAC Files

When the user requests that we use an Hmac algorithm, we must ensure that a key is provided. If we have one provided, the tool uses the `Mac` class to create the digest. Processes the file provided using the provided hexadecimal key and the message authentication code algorithm chosen. The HMAC algorithms (`Mac`) can take a `byte[]` to hash, so we can read the file one block at a time and provide it to the hash algorithm using the `update` function. Once we're done reading the file, we can use the `digest()` method to get the final digest.

`Note:` If you have a small file, you can read the entire file into a single `byte[]` using `Files.readAllBytes()` and then just the `digest(byte[])` method.

4.1 Verifying HMAC Files

The verification process for an HMAC on a file is almost identical to the process of computing an HMAC value. Process the file as you would when creating the HMAC code, then compare the received digest with the one provided by the user from the command line (*i.e.* inDigest). If the digests are identical, output OK. Otherwise, output Bad.

5 Byte Manipulation

Just like DES (and AES), the output of the hash functions is a byte array (`byte[]`) that can not be displayed as is. It is more common to display digests as hexadecimal strings, so I have written a simple function

```
namespace Hashing {
    class ByteManipulation {
    ...
    public static String bytesToHex (byte[] array) {
    ...
```

which takes a byte array and returns an equivalent hexadecimal string. For those interested, the output Java byte array stores the highest order byte in array entry 0. You can use the function I provided to format the `byte[]` output in a hexadecimal string.

You can find it and some other useful hexadecimal functions such as `byte[] hexToBytes(String s)` in the `ByteManipulation` class provided.

6 Using Hashes and HMAC

The interface for the hash functions is straightforward, you just need to use `MessageDigest` to request one of the algorithms by name using the `getInstance()` static method. Once you have an instance, you can compute the digest using the `update(byte[])` and `digest()/digest(byte[])` methods.

For a text string you must first convert it to a byte array. You can use the `String.getBytes()` or `String.getBytes(Charset)` methods.

For a file you must first build a `FileInputStream` object based on the provided file name and then provide it to the `MessageDigest` instance in chunks.

Note: Don't forget to close the `FileInputStream` when you are done using it or put it in a try-with clause.

For HMAC, you must use `Mac` class to request one of the algorithms by name using the `getInstance()` static method. Once you have an instance, you need to initialize it with the secret you want to use using the `init()` method. Like `Cipher`, `Mac.init()` needs a key. You can make one using a `SecretKeySpec`, just like we did in the previous recitation:

```
hmac.init(new SecretKeySpec(keyBytes, "HmacSHA256"));
```

Creates a key for SHA256. Change the algorithm name if you are working with a different algorithm.

Once you have configured the secret using `init`, you can compute the digest using the `update` and `digest` methods mentioned above.

7 What to do

Your job is to use the Java objects to make the tool perform the following:

1. Compute a hash using MD5, SHA1, SHA2–256, SHA2-512, SHA3–256, and SHA3–512 on a file
2. Verify the hash using MD5, SHA1, SHA2–256, SHA2-512, SHA3–256, and SHA3–512 on a file
3. Compute a message authentication code on a file using HMAC-SHA-256, HMAC-SHA-512, HMAC-SHA3-256, HMAC-SHA3-512.
4. Verify a message authentication code on a file using HMAC-SHA-256, HMAC-SHA-512, HMAC-SHA3-256, HMAC-SHA3-512.

Since Java provides implementations of SHA1, MD5, and SHA-256, SHA-512, SHA3-256, and SHA3-512 using a single class (`MessageDigest`), your task is quite simple. The implementations of HMAC-SHA-256 and HMAC-SHA-512 can be accessed using the `Mac` class. You'll need to use the following official algorithm names:

- MD5
- SHA1
- SHA-256
- SHA-512
- SHA3-256
- SHA3-512
- HmacSHA256
- HmacSHA512

- HmacSHA3-256
- HmacSHA3-512

7.1 Verifying your output

You can check that your tool works correctly by comparing the results you get against the expected values found in the Hash Testing Vectors file and associated test files on Moodle.

You can also find some online test vectors at websites such as `https://www.di-mgt.com.au/sha_testvectors.html`.

8 Experiments

Once you have completed your hashing tool and you verified its output, use it to perform the following experiments:

8.1 Experiment 1: Simple Hashing

1. Create a file `hf1.txt` with the text "Octogenarian tadpoles" and compute its digest using SHA3-256, SHA3-512, SHA2-512, SHA2-256, SHA1, and MD5. Make sure it verifies with all of them.
2. Delete the last character ("s") in the file. Compute the digest using SHA3-256, SHA3-512, SHA2-512, SHA2-256, SHA1, and MD5. Compare your results from the previous experiment. How much of the string changed with SHA3? SHA2-512? SHA2-256? How much with SHA1? How much changed with MD5?

What does that tell you about the sensitivity of the algorithms to small changes in the input?

8.2 Experiment 2: Files and Strings

1. Put back the last "s" you erased in the previous experiment. Create a second file `hf2.txt` with the same text, but force its encoding format to be Unicode. You can do this easily using Notepad++.
2. Try the previous experiment (computing the hash on the file). Is the output identical between the two files?

What does that tell you about the sensitivity of the algorithms to the file system and its default encodings?

8.3 Experiment 3: HMAC and Files

1. Use the file `hf1.txt` that you created before and use the following hexadecimal password: `ABadDadFed0123456789`. Compute the MAC value for it using HmacSHA256. What do you get as the message authentication code?
2. Change the code to the following `ABadDadFed0123456780` (change the last 9 to a 0). Recompute the MAC value using HmacSHA256. What do you get as the message authentication code? How much of the output changed?

What does that tell you about the sensitivity of HMAC-SHA2-256 to changes in the secret?

8.4 Experiment 4: Web Site Files

1. Go to the VirtualBox download page (`https://www.virtualbox.org/wiki/Downloads`) and choose to download one of the files (either for Windows hosts or Linux distributions).

2. The download will take a little while, but while you are waiting, go to the SHA256 checksums page (`https://www.virtualbox.org/download/hashes/7.1.6/SHA256SUMS`). The page has SHA2-256 digests for all of the files you can download.
3. After your download finishes, compute the SHA2-256 digest of the file you downloaded using the tool. Does the digest match the checksum posted online?
4. Go to the Apache Kafka download page (`http://kafka.apache.org/downloads`) and download the source for Apache Kafka 4.0.0 from one of the mirrors.
5. After the file downloads, check its checksum against the sha512 digest for it (`https://downloads.apache.org/kafka/4.0.0/kafka-4.0.0-src.tgz.sha512`). Does it match?

Why do VirtualBox and Apache include digests for the files on their websites?

9 Hashing and Invisible Changes

As discussed in class, hash digests can be used to create a message authentication code or a digital signature of data that is resistent to attackers. For text files and data structures that requires, of course, that the encoding used for interpreting the data be fixed. An interesting problem with creating such codes and signatures on files is that even seemingly non-changes to a file can cause them to fail. Try the following experiments to get a feeling for how sensitive hashing files is.

9.1 Experiment 1: Text Files

1. Create a new text file called "hash1.txt". Use Notepad to enter the text "This is a file to hash."
2. Save the file and close Notepad.
3. Use the hashing tool we developed to compute the SHA2-256 digest of the file.
4. After computing the SHA1 hash, reopen "hash1.txt" in Notepad and add a space at the end of the file so that it now reads "This is a file to hash. "
5. Save the file and close Notepad.
6. Go back to the hashing tool and click on the Verify button. What is the result?
7. Reopen the file "hash1.txt" in Notepad. Remove the trailing space, so it now reads "This is a file to hash."
8. Save the file and close Notepad.
9. Go back to the hashing tool and click on the Verify button. What is the result?

What have you learned about the storage of text files in Windows?

9.2 Experiment 2: MS Word Files

1. Create a new MS Word file called "hash1.docx". Use MS Word to enter the text "This is a file to hash."
2. Save the file and close MS Word.
3. Use the hashing tool we developed to compute the SHA2-256 digest of the file.
4. After computing the SHA1 digest, reopen "hash1.docx" in MS Word and add a space at the end of the file so that it now reads "This is a file to hash. "
5. Save the file and close MS Word.
6. Go back to the hashing tool and click on "Verify Hash". What is the result?
7. Reopen the file "hash1.docx" in MS Word. Remove the trailing space, so it now reads "This is a file to hash."
8. Save the file and close MS Word.
9. Go back to the hashing tool and click on "Verify Hash". What is the result?
10. Click on "Compute Hash" again to recompute the hash on the file "hash1.docx".
11. Reopen the file "hash1.docx" in MS Word. Make no changes to the document. Save the document and close MS Word.
12. Go back to the hashing tool and click on the Verify button. What is the result?

13. Reopen the file “hash1.docx” in MS Word. Add a space to the end of the file so that it reads “This is a file to hash. ”. Then delete the trailing space so that it once again reads “This is a file to hash.”
14. Save the file and close MS Word.
15. Go back to the hashing tool and click on the Verify button. What is the result?

What have you learned about MS Word files?

9.3 Experiment 3: MS Powerpoint Files

1. Download the file “hash1.pptx” from the course web page (it’s inside the ZIP file with the applications). Use the hashing tool to calculate the SHA2-256 hash of the file.
2. Open the file “hash1.pptx’ in MS Powerpoint. Start the slideshow. End the slideshow.
3. Save the file and close MS Powerpoint.
4. Go back to the hashing tool and click on the Verify button. What is the result?
5. Click on “Compute Hash” again to recompute the hash on the file “hash1.pptx”.
6. Reopen the file “hash1.pptx” in MS Powerpoint. Make no changes to the document. Save the document and close MS Powerpoint.
7. Go back to the hashing tool and click on the Verify button. What is the result?

What have you learned about MS Powerpoint files?