# CS 305 Module Five Coding Assignment Checksum Verification

Sean Jetté

CS-305

Professor Norman

## Algorithm Cipher

For this assignment, we are to select an appropriate encryption algorithm cipher for preventing attacks and we need to produce a cryptographic hash function that is not susceptible to collisions; which is “when different data inputs result in the same hash after being processed by a hashing mechanism” (Fulber-Garcia, 2024). Thus, for this situation, the appropriate hashing algorithm would be SHA-256. A member of the SHA-2 family, this form of cryptographic operations is widely utilized and is resistant to known collisions. Additionally, it is supported by Java’s *java.security* package. According to NIST (n.d.), the security strength of the SHA-256 hash function has a collision resistance strength in bits of 128 and preimage resistance strength in bits of 256. Additionally, this form of hashing algorithm is appropriate for generating file checksums while ensuring the integrity of data during transmission and safeguarding against potential attacks.

## Justification

In order to make certain that customers receive unchanged information during the process of information sharing, it is vital to employ a cryptographic hash function. Within the *Java Security Standard Algorithm Names* file, we have an extensive array of security standards utilized for API requirements which lists standard names for algorithms, certificates, and keystore types. Among these, the SHA-256 cryptographic hash function is the most appropriate for the given scenario. This algorithm processes input data into a fixed 256-bit hash value which outputs a fixed-size string of random characters as 256 bits long, which ensures that the output is unique for every distinct input (Gitlan, 2024). A key advantage of the SHA-256 hashing algorithm, compared to its predecessors, is the inherent collision resistance. An example is within the MD5 hashing algorithm, where Chinese researchers demonstrated pseudo-collisions for the hashing algorithm MD5, which in only 15 seconds to five minutes, they were able to generate collisions that produced identical hash values (Wang et al., 2004). In essence, collision resistance is a critical property of hash functions’ security features, preventing two different inputs from producing the same hash value and ensuring data integrity. Where malicious data could otherwise impersonate legitimate data, the implementation of SHA-256 for its computational efficiency and widespread adoption would make this a practical adoption for checksum generation. Avoiding collisions is a crucial step for preserving the integrity of downloaded files and maintaining the credibility of the distributed data.

## Generate Checksum

A screenshot of a computer program

Description automatically generated

## Verification

A screenshot of a computer

Description automatically generated

***Code Base:***

package com.snhu.sslserver;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.GetMapping;

import java.security.MessageDigest;

*@SpringBootApplication*

public class ServerApplication {

public static void main(String[] args) {

SpringApplication.*run*(ServerApplication.class, args);

}

}

*@RestController*

class ServerController {

/\*\*

\* End point to generate a hash value for the given name.

\*

\* **@param** name The input name (default: "Sean Jette").

\* **@return** HTML response displaying the name, algorithm used, and checksum value.

\* **@throws** Exception If a hashing error occurs.

\*/

*@GetMapping*("/hash")

public String generateHash(*@RequestParam*(value = "name", defaultValue = "Sean Jette") String name) throws Exception {

// Call method to compute hash

String hashValue = computeHash(name);

// Build dynamic HTML response

StringBuilder response = new StringBuilder();

response.append("<p>Data: ").append(name).append("</p>");

response.append("<p>Name of Cipher Algorithm Used: SHA-256</p>");

response.append("<p>Checksum Value: ").append(hashValue).append("</p>");

return response.toString();

}

/\*\*

\* Helper method to compute the SHA-256 hash for a given string.

\*

\* **@param** data The input data to hash.

\* **@return** The hash value as a hex string.

\* **@throws** Exception If a hashing error occurs.

\*/

private String computeHash(String data) throws Exception {

// Initialize the MessageDigest object for SHA-256

MessageDigest md = MessageDigest.*getInstance*("SHA-256");

md.update(data.getBytes());

// Generate the digest as a byte array

byte[] digest = md.digest();

// Convert the byte array to a hexadecimal string

StringBuilder hexString = new StringBuilder();

for (byte b : digest) {

hexString.append(String.*format*("%02x", b)); // Zero-padded two-character hex

}

return hexString.toString();

}

}

References:

Fulber-Garcia, V. (2024, March 18). *Hash Collision: Weak and Strong Resistance*. Baeldung. https://www.baeldung.com/cs/hash-collision-weak-vs-strong-resistance

Gitlan, D. (2024, September 19). *What is the SHA-256 algorithm & how it works*. SSL Dragon. https://www.ssldragon.com/blog/sha-256-algorithm/

NIST. (n.d.). *Hash Functions | CSRC | CSRC*. https://csrc.nist.gov/projects/hash-functions