****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 5](#_Toc102040759)

[3. Deploy Cipher 6](#_Toc102040760)

[4. Secure Communications 7](#_Toc102040761)

[5. Secondary Testing 8](#_Toc102040762)

[6. Functional Testing 10](#_Toc102040763)

[7. Summary 10](#_Toc102040764)

[8. Industry Standard Best Practices 11](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **10/14/2023** | **Jacob Balaj** | **Initial Version** |

## Client



## Instructions

Submit this completed practices for secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

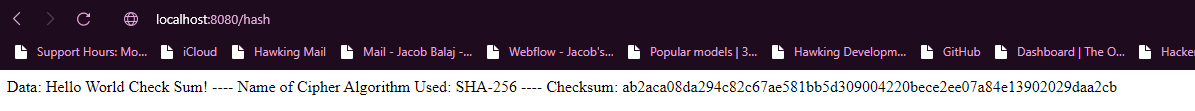
* Respond to the steps outlined below and include your findings.
* Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
* Refer to the Project Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

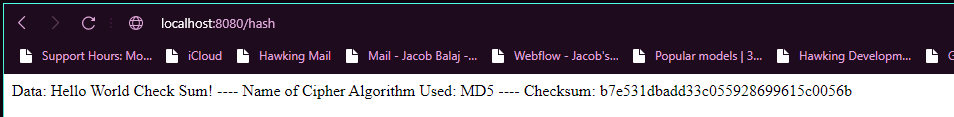
Jacob Balaj

## Algorithm Cipher

I have decided to use the SHA-256 cipher. It is a widely industry standard and in terms of security, I believe that Occam’s Razor should be applied. That is, I believe it will typically be better to use a solution that has been widely adopted and used in the industry rather than search for esoteric solutions or developing your own. SHA-256 is widely used enough that any serious concerns to the integrity of it as a hashing algorithm would quickly be brought to light and there would be an extensive professional community with a stake in solving the problem involved. It also has a great avalanche effect and has a 256-bit digest as an output. I believe that there is a benefit to creating larger digests when hashing since its more secure for two primary reasons. It will take more time for any threat actors to try and brute force it. Since it is unlikely that any threat actors would have the time or computing resources to even break a 128-bit output, the second reason would be for collision reduction. Whether through brute force or accident, a larger output means a smaller chance that there will be any collisions in the hashing function which allows for that function to be used in environments where the clusters of data being stored are significant. To demonstrate this, I am going to first show a screenshot of my SHA-256 checksum generation, which in this case is a conversion of my hash that has been formatted to show half of the characters.



I am now going to run the same input again, but this time using the MD5 algorithm and show the result in the image below.



At a practical level, even a 128-bit digest is going to be a monumental task for any threat actor to even attempt to break/reverse. I do think, however, that as developers and security professionals we should be taking wins whenever we can. The small amount of computing tradeoffs between running MD5 and SHA-256 aren’t comparable to the massive amount of collision risk buy-down you perform by selecting the solution that nets you twice the amount of space and even more increased security. For keystore we are using PKCS12 because during the creation of a JKS keystore file, I was warned that it has since been deprecated and replaced by PKCS12 at large by the community, and that I should consider changing its type over.

## Certificate Generation

I am first going to show a screenshot of the conditions I used to create my self-signed certificate. This will allow us to compare the input to the output, which would be a certificate attached to the web application once we run it.

A computer screen with white text

Description automatically generated

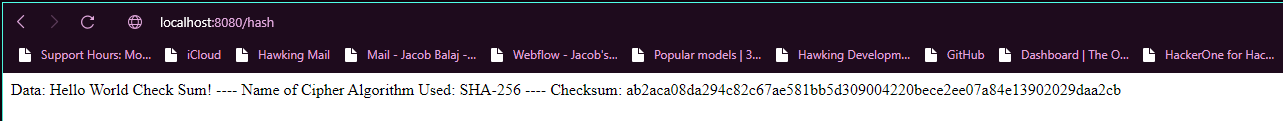
The actual contents of the certificate are going shown in the image below and is going to look like a whole bunch of scrambled nonsense which is great for preventing a reversal of the certification should someone get their hands on the certificate file (which would be unlikely since there would likely be many security measures taken to protect the certificate files and keystores/trust stores.

A screenshot of a computer screen

Description automatically generated

## Deploy Cipher

This is my initial run at deploying the algorithm which is verified through its checksum being printed out as well. To ensure that we are able to verify the avalanche effect of the function, we will have the initial static input and its checksum shown in the image below.



As explained previously, the avalanche effect is a critical factor in the strength of the SHA-256 algorithm. This means that when I do something as simple as add a single character, or remove a single character, that my resulting digest from the algorithm should be completely different from before so that no pattern could be established. Below is an image of running the algorithm again, but this time adding a letter to the word “Hellow”.

A computer screen shot

Description automatically generated

The avalanche effect is working great and I am including a table below for easy comparison. I am confident in the functionality of this cipher.

|  |  |
| --- | --- |
| Data Input | Generated Digest Checksum |
| Hello World Check Sum! | ab2aca08da294c82c67ae581bb5d309004220bece2ee07a84e13902029daa2cb |
| Hellow World Check Sum! | 6d31e7eae9cbdcc4c45df5af53a6c92fa863119de979ae90b4d6ef56bcc33e23 |

## Secure Communications

Once the key was created, I had to refactor my application file with the information needed to access the key. After refactoring my code, I should only be able to access my local server using a secure request and below is an image of me attempting to access the server with only a standard HTTP request.

A screenshot of a computer

Description automatically generated

By changing my request to a secure request, I am able to open the server and verify that the certificate I created earlier is being used for the application. You will notice that my browser displays a “Not Secure” tag and this is a result of me using a self signed certificate, which is not a big deal in a development or test environment but very much of an issue if being used for a production web application. My application is still secure but it is going to be flagged by my browser as a potential threat.

A screenshot of a computer

Description automatically generated

## Secondary Testing

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

As was just shown, my code does not have any errors in it. There are some warnings since this project was originally built for the JRE 1.8 environment, which is not something that I keep on my local machine. I always write in the most recent, however I did not want to change this since it did not affect my execution. The program ran just fine and I was able to successfully experiment with different hashing algorithms and certificate settings without any issues being introduced. I then ran the dependency check to the following result.

A screenshot of a computer

Description automatically generated

All of the dependency vulnerabilities that cropped up were as a result of the versions used of the dependencies that came bundled with the project and nothing new that arose as a result of my code. I also suspect that is due to the fact that I am only using algorithms and libraries that have been brought into the standard environment.

## Functional Testing

I do not believe that there is anything reversable in my code as a result of my code during runtime. If this was closer to production code and I would actually be taking in an input from a user, I would definitely make input sanitation my first effort. I currently have my inputs stored to string fields so that I can easily swap values in and out for testing, but this information would likely be thrown into a hash table. I am also storing my server and certificate information in the same project folder, which works for this project, but is not something I would do in production. I would also not publish my keystore certificate and its public certification in the source folder for my code. Again, these are necessities for the current project and not an issue in a local development environment, but I believe the security benefits of encapsulation extend to file structures in projects and not just programming. The image below is a screenshot of my code running successfully without errors as is shown by the console output. If there were any breaking errors, it would have shown up in the console output and prevented the sever from running on port 8080 since the code would still have to compile through the JVM before being spun up.

A screenshot of a computer

Description automatically generated

## Summary

The code has been refactored by having a secure hashing algorithm and checksum output verification method implemented. We have also included the keystore information required for our application to start in a secure environment by looking for a security certificate when attempting to establish a secure connection. I believe my code is secure because it is simple in its implementation. I create a digest object in order to take my inputs and put it through the SHA-256 algorithm. After that gets created, I call the next method which creates a string of converted hex values of my hash and outputs the string so it can be used a checksum of the hash. There is no state change in between the two methods meaning that there isn’t a way to exploit the space and time in between these two methods. My API is secure since it requires the self-signed certificate to open and there isn’t any front end input at the moment that could expose any potential flaws, and I am confident that if I were to implement a front end for user input that I would be able to sanitize my inputs.

## Industry Standard Best Practices

Industry standard best practices should always be at the core of any work that we do. That is why one of the greatest rules of security is to never write your own algorithms. Be it for hashing, encrypting, serializing, etc., etc. This is because there are entire teams of security researchers that are comprised of mathematicians and computer scientists whose sole goal is to develop and maintain security standards to the highest levels. I can definitely write my own hashing algorithm, but there isn’t a practical reality in which a hashing algorithm I create over the span of a few weeks will be able to beat out SHA-256 in terms of security. The industry is filled with professionals who are financially and/or personally invested in the security integrity of the algorithms being widely used, meaning that potential exploits will be quickly discovered and patched. It also means that there are usually people who are actively trying to break these algorithms and systems specifically so that they discover a flaw that can be fixed, and you will not get that kind of support on esoteric solutions or self-written solutions. That is why I have used SHA-256 for my encryption algorithm. It is currently the most widely used hashing algorithm meaning that we are getting a proven solution that while being a large target for threat actors, also imposes some of the steepest barriers. I have also used the PKCS12 archive file format to store my certificate information and its key since JKS is deprecated as the standard keystore file format to be supported by Java.