****

# Artemis Financial Vulnerability Assessment Report

Table of Contents

[Document Revision History 3](#_Toc32574607)

[Client 3](#_Toc32574608)

[Instructions 3](#_Toc32574609)

[Developer 4](#_Toc32574610)

[1. Interpreting Client Needs 4](#_Toc32574611)

[2. Areas of Security 6](#_Toc32574612)

[3. Manual Review 7](#_Toc32574613)

[4. Static Testing 1](#_Toc32574614)0

[5. Mitigation Plan 1](#_Toc32574615)5

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **8/2/23** | **Kayleigh Kinsey** |  |

## Client



## Instructions

Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In the report, identify your findings of security vulnerabilities and provide recommendations for the next steps to remedy the issues you have found.

* Respond to the five 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 One Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

Kayleigh Kinsey

## Interpreting Client Needs

**What is the value of secure communications to the company?**

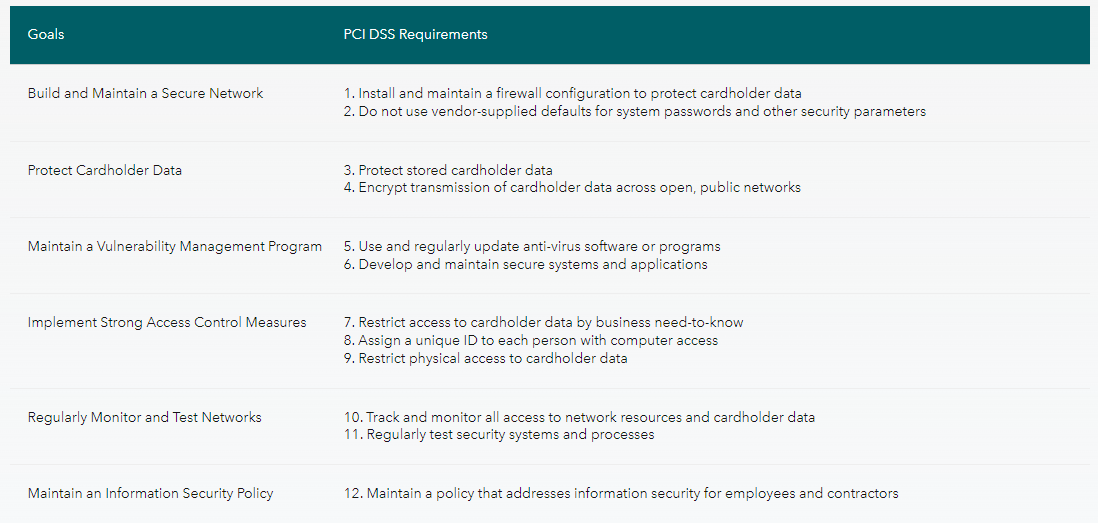
Secure communications are important in maintaining customer trust and Artemis Financial's image. If there were a data breach, the company would lose a lot of money, not just to the hackers, but to the government in fines for failing to protect the financial data of customers. Customers would no longer feel confident in the company's ability to protect their information. Ultimately, the entire company could collapse. Secure communications are essential in preventing this sort of disaster.

**Does the company make any international transactions?**

Being a personal financial consulting company, most of their transactions would be within the country. I can't think of any situations in which they would need to make international transactions. Their website will be accessible from around the world, though.

**Are there governmental restrictions about secure communications to consider?**

If they are taking credit card information, they will need to comply to PCI-DSS. The PCI-DSS requirements are outlined in the table below:



PCI Security Standards Council. (n.d.). [Table containing the PCI-DSS Goals and Requirements]. Retrieved July 11, 2023 from listings.pcisecuritystandards.org/pci\_security/maintaining\_payment\_security

Since they handle the financial information of their customers, they also need to comply to The Graham–Leach–Bliley Act, which dictates that they need to publicly disclose their information sharing practices and protect the data of their customers. That data cannot be shared with third parties without customer permission. (FTC, n.d.) If Artemis Financial does not comply to these restrictions, there will be hefty fines to pay.

**What external threats might be present now and in the immediate future?**

Hackers are highly motivated by money, so companies that deal in finances need to be extra careful with their security, because there are almost certainly going to be attempts at breaking in. Some common threats to be aware of are injection/remote code execution, DoS attacks, role escalation, and data leaks.

**What are the modernization requirements that you must consider? For example:**

**The role of open-source libraries**

**Evolving web application technologies**

It is important to keep up to date with the latest information on any libraries the project is using. This means updating them regularly and watching for new vulnerabilities that may be uncovered. These unresolved vulnerabilities need to be taken into account when implementing security. We need to remember that a library is not infallible just because it’s popular and open-source. They have security vulnerabilities too, which broaden the attack surface of an application and are harder to deal with than those in the application code.

We should use the latest versions of Java and HTTP as well, not using any retired terms and functions unless there's good reason to. The newer versions are almost always more secure. We also need to keep up with best security practices. This means following the latest security standard practices recommended by organizations like OWASP and also maintaining good, clean code. A relatively new web technology is cloud computing, which I would recommend to Artemis Financial since back-end security would be handled by the service provider.

## Areas of Security

These are the areas of security that apply to Artemis Financial’s project:

**- Input validation**

Users will need to interact with the application by entering information like a username and password. These and anything else coming from the client are untrusted data, and must be validated before they are used by the application. Otherwise, the system will be vulnerable to DoS attacks and code injection. Any application that takes input from the user (which is pretty much every application) needs to validate that input.

**- Client/Server**

Since this application will be web-based, it will involve some communication between client and server. Even if the brunt of the client/server code is left to another company, securely implementing client and server communication is still important. Otherwise, log information, user data, and other data on the inner workings of the application could leak out.

**- Code Error**

Errors and failures can provide hackers a window through which they can infiltrate the program. Exception messages may also be leaked and give information to hackers about the application’s components and functions. Any sort of program can have errors that are exploitable by those that may want to get in, and in the case of a company handling financial data, there are many such people. This is why the application code should have as few errors as possible.

**- Code Quality**

Code quality is important to any program. Having clean, easy to read code that follows best practices is a good base for security. In confusing spaghetti code, it’s hard to find vulnerabilities and fix them. Some may think this would make it harder for hackers to find the vulnerabilities, but that isn’t really true. It is much better to have clean code that is clearly secure.

**- Encapsulation**

Proper encapsulation is key to protecting an application. It makes it harder for attackers to access and manipulate the program, protecting it in the event that other security measures do not work. If encapsulation is utilized properly, then a hacker that has invaded one area of a program will be unable to access others, and the damage they can do is minimized.

## Manual Review

**Input validation**

Input is not validated in the GreetingController or CRUDController classes. The application takes strings directly from the URL pathway and uses them. This is really dangerous, because someone could write “/greeting” and then a really long name that causes a denial of service error, or they could input quotation marks and then some code, which would be automatically executed by the application. The CRUDController class allows access to the data of any business name put after “/read”, without validating the input or verifying whether or not the user has permission to view this information. This gives everyone access to the data of any business, which is probably not intended.

Public methods also don’t validate their input, which also makes them vulnerable to Dos attacks in the case that an attacker passes them very large arguments.

**Client/Server**

I didn't notice any client/server vulnerabilities in this code.

**Code Error**

When I ran the program, there were no compilation errors.

The DocData class’ constructor is empty, and currently doesn’t do anything. If a DocData instance is created and getId() is called, then the program will crash because id was never initialized and there is no way to set it.

When CRUDController tries to create a new DocData instance and convert it to a string, this will create an error because objects of type DocData cannot be converted into a string (at least, as far as I know).

The myDateTime class seems unfinished. Its attributes are never initialized, setMyDateTime() doesn't do anything, and retrieveDateTime() seems to return an empty array. This could lead to errors when myDateTime instances are in use.

**Code Quality**

Fine for the most part, but there are a few violations of best practices. Variables do not have a consistent name scheme, like the attributes of the customer class are named with underscores and the attributes of myDateTime use camelCase.

**Encapsulation**

The account\_balance attribute of the customer class is not declared as private. If someone were to gain access to the program, they could directly manipulate a user’s account balance. Obviously, this is not good.

The myDateTime class is not encapsulated at all. None of its attributes or methods are private or protected. Someone could inject code through the “/greeting” or “/read” pathway and access this class, assigning myDateTime’s attributes large numbers or invalid data types. This could cause the application to slow down or crash.

## Static Testing

Here are the vulnerabilities listed in my dependency report. My descriptions are mostly paraphrased from the report.

## CVE-2016-1000338 Bouncy Castle JCE is vulnerable to injection due to flaws in its signature validation. It seems that in new patches, the vulnerability has been fixed.

**CVE-2016-1000343**

Bouncy Castle JCE generates a weak private key by default. To prevent this from happening, the key pair generator needs to be passed explicit parameters. The problem has been fixed in patches.

**CVE-2016-1000344**

Bouncy Castle JCE allowed the use of ECB mode. This mode is regarded as dangerous and is no longer supported by the developer.

**CVE-2016-1000341**

Bouncy Castle JCE is vulnerable to timing attacks, through which data can be leaked . In new patches, this problem has been addressed.

**CVE-2016-1000345**

Because of the previously mentioned vulnerability, Bouncy Castle JCE is vulnerable to a padding oracle attack, which can reveal more vulnerabilities to an attacker.

**CVE-2017-13098**

Using a vulnerability called “ROBOT”, an attacker can obtain an application’s private encryption key. This issue has been fixed in new patches.

**CVE-2020-0187**

In engineSetMode of BaseBlockCipher.java, there is a possible cryptographic error that could leak information. The problem seems to apply only to android operating systems and has been patched out by Google.

**CVE-2016-1000339**

Bouncy Castle’s AESFastEngine was at risk of leaking the AES encryption key. AES engine has been changed to fix the problem and use of AESFastEngine is generally not recommended anymore.

**CVE-2020-26939**

By sending invalid ciphertext to Bouncy Castle BC and observing its response, it is possible to figure out the private exponent of the encryption key. This problem can be solved by performing a length check and ensuring the input is valid before encrypting it.

**CVE-2023-33201**

Bouncy Castle is vulnerable to an LDAP injection attack during certificate validation. This problem has been fixed in newer versions.

**CVE-2015-7940**

Older versions of the Bouncy Castle library are vulnerable to an “invalid curve attack”, which can be used to uncover private keys. This issue has been addressed in new patches.

**CVE-2018-5382**

In older versions of Bouncy Castle, the BKS keystore used an HMAC that was 16 bits long. This was vulnerable to attack, so new patches use a 160 bit HMAC. The 16 bit version is available, but it is highly insecure and not recommended.

**CVE-2016-1000346**

Due to a validation error, older versions of Bouncy Castle JCE could be fed invalid keys, and the responses could be used to find private key information. It has been fixed in new versions of the library.

**CVE-2015-6644**

On Android devices, it was possible for malicious applications to access sensetive information through Bouncy Castle, but Android updates have fixed this issue.

**CVE-2020-10693**

There is a flaw in Hibernate that allows attackers to bypass sanitization. It does seem like the developer is aware of it and has dealt with it in updated versions.

**CVE-2020-25649**

The FasterXML Jackson Databind is vulnerable to XXE attacks because entity expansion is not secure. This problem seems to have been patched out of newer versions.

**CVE-2020-36518**

Older versions of jackson-databind are vulnerable to DoS attacks via nested objects.

**CVE-2021-46877**

Older versions of jackson-databind are vulnerable to DoS attacks via JsonNode JDK serialization. This seems to have been a very small issue that required specific conditions to pose a problem, but it was still patched out quickly.

**CVE-2022-42004**

Old versions of FasterXML jackson-databind are vulnerable to DoS attacks via nested objects, but this was only applicable to applications with certain custom deserialization settings.

**CVE-2023-35116**

Older versions of jackson-databind are vulnerable to DoS attacks via objects with cyclic dependencies. This vulnerability has been disputed though, and there are few sources related to it. The developer’s opinion is that jackson-databind is not intended for use with untrusted data, and it is the responsibility of library users to validate untrusted data before using it with jackson-databind. This is important to remember when working with this library.

**CVE-2020-9488**

The Apache Log4j SMTP appender is vulnerable to a man-in-the-middle attack due to improper certificate validation. This could allow someone to access leaked log messages. This issue has been addressed in new versions.

**CVE-2021-42550**

In older logback versions, an attacker could access configuration files and create malicious configuration code. This issue was resolved so that only someone with proper permission can access the files.

**CVE-2022-1471**

SnakeYaml versions before 2.0 had an improperly secured constructor. Because of this, deserializing untrusted yaml files could lead to remote code execution. Snake Yaml has a SafeConstructor that does not have this problem, and the newer versions are more secure.

**CVE-2017-18640**

SnakeYaml’s alias feature allowed entity expansion, which put it at risk. This problem has been fixed in newer versions.

**CVE-2022-25857**

Older versions of SnakeYaml are vulnerable to DoS attacks because they do not have depth limitations for nested objects. This problem has been resolved in newer versions.

**CVE-2022-38749**

If SnakeYaml is used to parse untrusted YAML files, it could put the application at risk of a DoS attack. This has been patched out.

**CVE-2022-27772**

Old spring-boot versions are vulnerable to temporary directory hijacking. The versions with this problem are no longer supported by the developer, so it is best to just update if they’re in use.

**CVE-2023-20883**

In older versions of spring-boot, there is a risk of DoS attacks if both Spring MVC and a reverse proxy are in use. The problem has been addressed in new versions.

**CVE-2022-22965**

Spring MVC and Spring WebFlux might be at risk of injection through data binding. This vulnerability is for specific instances where the application is running on Tomcat as a WAR deployment. An application using the default Spring deployment is not vulnerable, but otherwise the vulnerability could possibly apply.

**CVE-2021-22118**

Webflux applications using older versions of Spring are vulnerable to vertical privelage escalation. This is resolved in new versions, and the recommended mitigation technique is just to upgrade.

**CVE-2020-5421**

Old versions of Spring have a vulnerability in their defense against RFD attacks. An attacker can bypass this in some browsers using a jsessionid path parameter. This problem has been fixed in new versions.

**CVE-2022-22950**

In old versions of Spring, an attacker could create a SPeL expression that causes denial of service. This is resolved in new versions, and the recommended mitigation technique is just to upgrade.

**CVE-2022-22971**

In old Spring Framework versions, WebSocket endpoints using a STOMP are vulnerable to a DoS attack. This problem has been patched in newer versions.

**CVE-2022-22968**

Old Spring Framework versions makes disallowed fields case-sensetive, which makes it difficult to fully protect a field. This is resolved in new versions, and the recommended mitigation technique is just to upgrade.

**CVE-2022-22970**

Older versions of the Spring Framework are vulnerable to a DoS attack through specific circumstances involving MultipartFile or javax.servlet.Part. They should not use data binding to set these files to a field in a model object.

**CVE-2021-22060**

In old Spring Framework versions, a user could cause the creation of extra log entries. This is resolved in new versions, and the recommended mitigation technique is just to upgrade.

**CVE-2016-1000027**

Pivotal Spring Framework is at risk of remote code execution when used to deserialize untrusted data. Some measures have been taken to deal with this issue, but in general the developer has said that it is up to users of Spring to be careful deserializing untrusted data.

**CVE-2020-1938**

Apache Tomcat implicitly trusts Apache JServ Protocol (AJP) more than other connections, which opens up opportunities for attackers to get in and execute code. This is only possible if untrusted users have access to the AJP. Upgraded versions of Tomcat are protected against this threat, but if an old version is already in use, it may require rewriting code. Since this project doesn’t make use of Tomcat yet, it can be upgraded without a problem.

**CVE-2020-11996**

Older versions of Apache Tomcat can be used to cause a denial of service via a series malicious HTTP/2 sequences. It has been resolved in new versions.

**CVE-2020-13934**

Old versions of Apache Tomcat could suffer an OutOfMemory exception if too many HTTP/1.1 requests were made. This issue has been fixed in new versions.

**CVE-2020-13935**

In older versions of Apache Tomcat, WebSocket frame payloads were not properly validated for length, and this put it at risk of DoS attacks using invalid lengths.

**CVE-2020-17527**

Old versions of Apache Tomcat can reuse HTTP headers from previous streams. In most cases, this would just lead to an error and not be exploitable, but it is possible that data could leak. The problem has been resolved in updated versions, and the recommended mitigation method is simply upgrading.

**CVE-2021-25122**

In older Apache Tomcat versions, sometimes the header and some of the body of a request header would be copied across h2c connection requests. This could cause one user to receive the results of another user’s request. The problem has been patched out in new versions.

**CVE-2021-41079**

Older versions of Apache Tomcat did not validate TLS packets properly. Under certain conditions, an attacker could create a TLS packet that triggers an infinite loop, resulting in a denial of service.

**CVE-2022-29885**

The documentation for Apache Tomcat implies that EncryptInterceptor made it safe to run on an untrusted network. Although it is helpful for this, it doesn’t offer full protection. This only applies to the documentation for older versions of Tomcat.

**CVE-2022-42252**

Old Apache Tomcat versions were vulnerable to request smuggling if configured to ignore invalid HTTP headers. This problem was resolved with later versions, and can otherwise be mitigated bu setting rejectIllegalHeader to true.

**CVE-2020-9484**

With old versions of Apache Tomcat, an attacker can access the files on a server and reconfigure the server to execute code remotely. There are many steps involved and it is a very specific process.

**CVE-2021-25329**

The original fix for the previous vulnerability was not enough, and Apache Tomcat was still vulnerable to the same attack. The attack is even more difficult to carry out though. This issue was dealt with in another update.

**CVE-2021-30640**

In older versions of Apache Tomcat, specifically the JNDI Realm, an attacker could authenticate using variations of a user name. This could allow them access to information they are not allowed.

**CVE-2022-34305**

Old versions of Apache Tomcat had a form authentication flaw that made them vulnerable to XSS. User data was used directly without being properly validated.

**CVE-2021-24122**

Under certain conditions, Apache Tomcat using the NTFS file system could leak source code. This was due to the Windows API acting unexpectedly. This problem only applies to older versions of Tomcat.

**CVE-2021-33037**

Under certain conditions, old Apache Tomcat versions didn’t parse HTTP transfer-encoding request headers, which made them vulnerable to request smuggling. It is advised that Tomcat be updated to mitigate this issue.

**CVE-2019-17569**

In some Tomcat updates, Transfer-Encoding headers were not dealt with properly, which introduced a vulnerability to request smuggling. An attacker would need to use a reverse proxy with a very particular functionality in order to exploit this vulnerability, so the risk is quite low if we’re using one of the vulnerable versions.

**CVE-2020-1935**

With older Apache Tomcat versions, the HTTP header parser sometimes read invalid headers as valid. This makes these versions vulnerable to request smuggling, but only if an attacker had a reverse proxy with very particular functionality.

**CVE-2020-13943**

For older versions of Apache Tomcat, if an HTTP/2 client has too many concurrent streams, enough HTTP requests could be made that users begin seeing unexpected responses.

**CVE-2023-28708**

Under a set of specific conditions, older versions of Apache Tomcat can end up sending cookies over insecure channels.

**CVE-2021-43980**

In some versions of Apache Tomcat, there is a very complicated bug that is hard to exploit. When triggered, it could cause clients to share a processor and potentially receive each others’ responses. It seems like the problem was fixed in later Tomcat releases.

## Mitigation Plan

**Input Validation**

It is most important to validate all untrusted data before using it in the program.

Unless all users should be able to read the data of any business, we should implement some form of role system that can check user permission before responding to "/read".

Validate input to public methods. If an attacker were able to get into the program, they could feed invalid input to a public method. Ensure arguments are valid length.

**Code Error**

Complete DocData class' constructor.

In the DocData class, overwrite the toString() method so that DocData objects can be properly converted to a string.

Finish the myDateTime class.

**Code Quality**

Ensure variables use a consistent name scheme.

**Encapsulation**

Declare customer class' account\_balance as private and create an accessor method so other parts of the program can still access it. This is a very important attribute that needs to be secured.

Encapsulate myDateTime class by declaring its fields as private, as well as any methods that don’t need to be accessed outside of the class.

**CVE-2016-1000339**

Do not use Bouncy Castle's AESFastEngine and instead use the newest version of AES engine. AESFastEngine has vulnerabilities and isn’t recommended anymore.

**CVE-2020-26939**

Validate all input for length before encrypting it. Even if new updates try to address this issue, it is general good practice that can defend against DoS attacks.

**CVE-2018-5382**

Do not use the 16 bit HMAC unless it is necessary, which it probably won't be for this particular project. It’s mostly used with legacy programs now.

**CVE-2023-35116**

Do not use untrusted data with jackson-databind. It wasn’t meant to handle untrusted data, and the burden is on users of jackson-databind to validate the data themselves.

**CVE-2022-22965**

Be careful deploying a spring application that doesn't use the default deployment configurations. Although this issue has been addressed, it may be applicable to certain deployments still. The vulnerability is very difficult to exploit, and it is unknown what deployment configurations are vulnerable, so we don’t need to worry too much. I do recommend we keep an eye on this vulnerability in case there are new related developments.

**CVE-2016-1000027**

Be careful deserializing untrusted data. This applies to a few other vulnerabilities as well. In general, deserializing untrusted data is risky, and should be done securely regardless of the libraries in use.

In general, the best mitigation plan for dealing with these vulnerabilities is just keeping all libraries up to date. Almost all of the vulnerabilites have been resolved by updates, and so long as we are using the latest stable library versions, the problems in the dependency report are of little concern.

Works Cited

FTC. (n.d.) *Gramm-Leach-Bliley Act*. Ftc legal library. www.ftc.gov/legal-library/browse/statutes/gramm-leach-bliley-act