Emerging Topics in Cyber Security

Secure Coding Practices - JAVA

MS18908770

Lahiru Lakshan Hewawasam

M.Sc. in Information Technology

Specializing in Cyber Security

Table of Contents

Introduction

The following document focuses on secure coding practices which would require the developers to use when developing an application using Java. These best practices help developers build applications which are less prone to attacks and in turn make them secure.

The best practices looked at in this document include native vulnerabilities which reside in the programming language and also coding practices which open up vulnerabilities in the logic of the application or the mechanism where the data is processed.

Secure Coding Practices

Input Validation

Set input field limits

When creating input fields to allow users to input data, always maintain a limit to the number of character which can be inserted into a specific field by the user. This is to make sure that a manipulated input does not cause the buffer to overflow resulting in the application to crash or for the user to execute arbitrary code.

The field limits would be set according to the input which is expected from a specific field, making sure that these values are optimized to allow the maximum number of characters needed but does not in turn create an issue where the user cannot insert valid data.

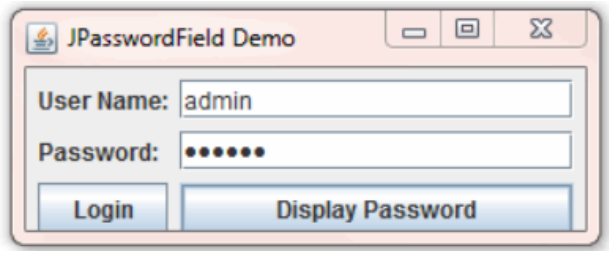
Validate field inputs before being processed

Fields which allow user inputs should be validated to check if there are characters which does not belong in a certain type of input, which could cause the application to fail due to an unexpected character being inserted by the user.

This can be accomplished at the time where the user inserts the data or when the data is being processed. When the characters which are allowed to be inserted into the field are defined the user will not have the chance to enter invalid characters, but it is important to check if this restriction is applicable if a user pastes values into a field. When the input is validated before being processed by a function, it should be checked against a pre-defined set of characters which are not allowed to be inserted, therefore reducing the chances of a user entering an invalid character which can cause the application function to fail.

Explicitly define the type of input for each input field

Explicitly defining input type for each field would ensure that the inputs are not inserted insecurely, this would affect the insertion of passwords into the application, where if the input type is not set accordingly, it would cause the password entered in the input field to remain in plain text. This issue should be handled by explicitly defining the password field to mask the entered characters as the user inserts the data to make sure the data entered is not visible.



Define wrappers around native methods

Using pure Java code is advisable since it is subjected to runtime checks which checks for criteria such as type, library usage and array bounds, but when using Native code these check do not take place. Therefore, Native code is not protected against attacks such as buffer overflow and other attacks.

It is recommended that these types of functions be declared private and be exposed through a public Java-based wrapper method which can provide the necessary input validation to prevent attacks. The wrapper would handle the errors which are thrown by the Native method safely thus not causing the application to misbehave or crash. (Oracle, 2019)

Database queries should not be vulnerable to injection attacks

User provided information should always be considered untrusted, constructing queries by using these inputs can result in the SQL query being susceptible to an injection attack. This is more pronounced when using string concatenations to insert the parameters into the SQL query. (SonarSource, 2019)

Instead of using string concatenations to insert the parameters into the query, it is recommended to use Java Prepared Statements which will ensure that the data provided by the user is properly escaped. (SonarSource, 2019)

```
String query = "SELECT * FROM users WHERE user = ? AND pass = ?"; // Safe

java.sql.PreparedStatement statement = connection.prepareStatement(query);
statement.setString(1, user); // Will be properly escaped
statement.setString(2, pass);
java.sql.ResultSet resultSet = statement.executeQuery();
return resultSet.next();
```

Unique names should be assigned to Struts validation forms

When using the Struts validator, it is important to have unique name for each form, since having two forms with the same name will cause the Struts validator to choose one form randomly and this would cause a logical fault in the application which the developer did not expect. (SonarSource, 2019)

XML parsers should not be vulnerable to external entity attacks

XML specification allows the use of Internal and external entities, meaning that these external entities are fetched from an external or untrusted network.

Allowing access to external entities in XML parsing could lead to vulnerabilities such as Server Side Request Forgeries and File disclosures.

To mitigate these types of vulnerabilities being brought in by allowing external entities during XML parsing, the following options are recommended to be set. (SonarSource, 2019)

- ACCESS_EXTERNAL_DTD is to be set to “”
- ACCESS_EXTERNAL_SCHEMA is to be set to “”
- ACCESS_EXTERNAL_STYLESHEET is to be set to “”

Settings these options will stop the parser looking up external entities during parsing.

It is also recommended to not use FEATURE_SECURE_PROCESSING feature to protect against External Entity attacks since depending on the implementation of the application, this feature will not guarantee that the proper mitigation controls would be put in place. (SonarSource, 2019)

Password Management

Reading password from console

Reading sensitive information from the console inputs will cause these inputs to be stored in the application until a garbage collector recollects the value which was stored, thus leaving sensitive information such as the password in plaintext for anyone with the proper permissions on the application to gain unauthorized access to it.

Therefore, it is not recommended to use the ReadLine function to collect the password being entered by the user but instead use the ReadPassword method which is a much secure alternative. (GIANCHANDANI, 2018)

Password field should always be obscured to the user

This is similar to defining the type of the field to make sure that its content is treated accordingly. The password field in an application should always show its values in an obscured format where it is not possible for any user to read this information in a plaintext format.

This ensures that the passwords entered into a field cannot be seen by any other user who might be using a social engineering attack such as shoulder surfing to gain access to sensitive information.

"javax.crypto.NullCipher" should not be used in production applications

The NullCipher class is to used only for testing purposes. When invoked it does not change or encrypt the plaintext in anyway thus leaving the result the same as the plaintext. (SonarSource, 2019)

Using this function in a production application can leave sensitive information unencrypted and susceptible to hijacking or theft.

Authentication should not rely on insecure "PasswordEncoder"

When storing users' passwords in a database It is not recommended to store them in plaintext, instead it is required to store it in a hashed format where the plaintext of the password can never be recovered even if the database containing the passwords are compromised.

In a Java spring application, the PasswordEncoder method can be used to create the hash of the password when it is required to store or to authenticate a valid user into an application. (SonarSource, 2019)

When using the PasswordEncoder method it is only recommended to use the following modes, since the rest of the ciphers are considered to be deprecated and unsecure. (SonarSource, 2019)

- org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
- org.springframework.security.crypto.password.Pbkdf2PasswordEncoder

Caching results of potential privileged operations

Any result which is generated by a privileged operation should not be cached into a context which does not have the relevant permissions to generate that same result. This would lead to the application leaking sensitive information into lower privileged contexts. (Oracle, 2019)

To check if a particular context has the required permissions the Java AccessController API can be used to enforce the constraints on the privileges being set within the application.

Cryptographic functions should be carried out in the server

When cryptographic functions are used in an application, all its functions should take place within the server; this includes all the operations including but not limited to the encryption and decryption of sensitive data. (GIANCHANDANI, 2018)

The reason behind this being that the client is more susceptible to attacks and therefore if decompiled the attackers will get to know about the internal algorithms running on the application.

Output encoding

Output from functions which take in user specified characters as their input should always be encoded before being sent to other functions for processing. (GIANCHANDANI, 2018)

This is to make sure that in the exception that specific hazardous characters need to be accepted and used, the encoding process will make it so that these characters no longer bring in the same level of risk into the application rather when processing the RAW characters.

Use of Java hash code

The Java hashcode function can be used to create hashes of objects within the Java application, even though this can be used, it is not recommended considering the output length of the hash being produced by the function.

The java hashcode function only creates a 32-bit unsigned integer, which approximately contains 4 billion unique values, since this is by far inferior to other functions or dedicated algorithms available. It is recommended that a superior and industry recognized algorithm such as SHA-256 be used for the purpose of hashing within the application. (Oracle, 2018)

Use of strong cryptographic algorithms

When using cryptographic algorithms within the application, it is recommended to use a strong cryptographic algorithm such as the AES-256 algorithm. (SonarSource, 2019)

Other algorithms such as the DES, 3DES, RC2, RC4 and Blowfish algorithms have been deemed unsecure since they do not require a lot of computational effort to break them. (SonarSource, 2019)

Therefore, it is recommended to use a strong cryptographic algorithm which is set by the industry standards.

Using AES algorithm in a secure mode

Using a secure cryptographic cipher such as the AES-256 algorithm is recommended. However, these ciphers needs to be implemented in a secure mode where additional vulnerabilities do not get introduced to the application. (SonarSource, 2019)

Implementing the AES algorithm in Electronic Codebook mode or Cipher Block Chaining mode is not recommended since these modes make susceptible to multiple known attacks. (SonarSource, 2019)

Instead using the Galois/Counter mode with no padding is preferred.

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
```

Storing hard coded credentials in the code

Credentials should never be stored in the application code. This brings in security as well as operational issues. This will allow anyone with the source code to view the password and also will not allow for the password stored in the code to be changed without rolling out a patch for the application. (Kiuwan, 2019)

This will cause multiple issues with the fundamental maintenance of the application.

Clearing data which is only used temporarily in the application

Values such as credentials should never be stored in the application or on the JVM for a longer period of time, instead they should be cleared out of both the application and the JVM once they are used for their intended purpose.

This process should be handled manually and should never be kept until the garbage collector clears them. This is to minimize the risk of the application leaking sensitive information.

Using outdated java libraries

Using outdated java libraries will bring in vulnerabilities to the application via the many components that the library brings in. This could vary from being an authentication bypass vulnerability to all the way to an information exfiltration vulnerability.

Therefore, it is recommended to only use trusted libraries and to always keep these libraries up-to-date keeping the entire application's security intact.

Refrain from using serialization

Serialization is used in most cases to save storage space or to send the data as a part of the communication. Even though this method has its advantages, the vulnerabilities that this brings in are far superior. (SonarSource, 2019)

Successful exploitation of deserialization can lead to remote code execution of the target system or even privilege escalation attacks by manipulating the object which is being deserialized. (SonarSource, 2019)

Therefore, it is not recommended to use serialization on an application and to especially serialize data from untrusted sources such as data retrieved from user inputs

Variable Declarations

Declare public static fields final

Callers can access and modify public variables which have not been set as final static fields. Neither modifications or accesses can be guarded against this, and newly set values cannot be validated. (Oracle, 2019)

Therefore, it is recommended that all public static fields be set as final.

```
public class Files {
    public static final String separator = "/";
    public static final String pathSeparator = ":";
}
```

Error Handling

Refrain from displaying raw error messages

RAW error messages being generated from the application or the programming language can give a lot of information to an attacker about the inner-workings of the application and the libraries being used.

Therefore, proper error handling mechanisms should be built into the application to catch any error that can be caused by anything ranging from a misconfigured business logic to a manipulated request by an attacker.

The error message should only display generic error messages such as “User Login Failure” and must never disclose the exact reason behind the error. In specific exceptions, the application may display more information but it should never disclose any versions or system generated error messages.

Logging

Implement application logging functions

Implementing proper logging mechanisms in the application is vital, and the java Logger function can be used to deploy this function. (SonarSource, 2019)

It is important to configure this function accordingly since misconfigurations can lead to sensitive information being stored in the logs. (SonarSource, 2019)

Therefore, it is recommended to set the log level to only capture Warnings, info and error messages as debug level logging would store sensitive information within the logs. it is also important to include the precise time of the events and hostnames when applicable.

Additionally, these logs should never be saved in the same location as the application, instead they should be stored in a secure location where only system administrators or other authorized personal have access to them. (SonarSource, 2019)

References

GIANCHANDANI, P., 2018. *Developing Secure Java Code - Best Practices For A Team*.

Kiuwan, 2019. *OWASP Top 10: How To Find Vulnerabilities In Your Java Applications*. [online] Kiuwan. Available at: <https://www.kiuwan.com/blog/owasp-vulnerabilities-java-applications/>.

Oracle, 2018. *Object (Java Platform SE 7)*. [online] Docs.oracle.com. Available at: <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()>.

Oracle, 2019. *Secure Coding Guidelines For Java SE*. [online] Oracle.com. Available at: <https://www.oracle.com/technetwork/java/seccodeguide-139067.html#1>.

SonarSource, 2019. *Java Vulnerabilities: AES Encryption Algorithm Should Be Used With Secured Mode*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-4432>.

SonarSource, 2019. *Java Vulnerabilities: Configuring Loggers Is Security-Sensitive*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/tag/owasp/RSPEC-4792>.

SonarSource, 2019. *Java Vulnerabilities: XML Parsers Should Not Be Vulnerable To XXE Attacks*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-2755>.

SonarSource, 2019. *Java Vulnerability: "Javax.Crypto.Nullcipher" Should Not Be Used For Anything Other Than Testing*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-2258>.

SonarSource, 2019. *Java Vulnerability: Authentication Should Not Rely On Insecure "Passwordencoder"*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-5344>.

SonarSource, 2019. *Java Vulnerability: Cipher Algorithms Should Be Robust*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-5547>.

SonarSource, 2019. *Java Vulnerability: Database Queries Should Not Be Vulnerable To Injection Attacks*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-3649>.

SonarSource, 2019. *Java Vulnerability: Deserialization Should Not Be Vulnerable To Injection Attacks*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-5135>.

SonarSource, 2019. *Java Vulnerability: Struts Validation Forms Should Have Unique Names*. [online] Rules.sonarsource.com. Available at: <https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-3374>.