Security Best Practices For JAVA Programming Language

Assignment 3

Student Number: MS19801896

Student Name: Janarthanan Krishnamoorthy

Subject: Emerging Topics in Cyber Security

Course: M.Sc. Information Technology (Cyber Security)

Table of Contents

Abstract

Java programming language is a one of the best programming language, which is used almost in all the software programs and applications. The Java platform and its third party libraries provided different types of security features to provide secure coding. Mishandling the features of the programs might lead to high risk factor to the application and systems

As per the research carried out, I found that programming challenges are typically identified with APIs or libraries, including the complicated cross-language data treatment of cryptography APIs, and the complex Java-based or XML-based approaches to configure Spring Security. Also, we revealed various security weaknesses in the recommended code of accepted answers on the Stack Overflow discussion. The weaknesses included incapacitating the default protection against Cross-Site Request Forgery (CSRF) attacks, breaking SSL/TLS security through bypassing certificate validation, and utilizing insecure cryptographic hash functions. Our discoveries uncover the insufficiency of secure coding assistance and documentation, just as the tremendous hole between security hypothesis and coding practices.

In this report it is discussed about different security practices that should take while developing an Java programs and applications.

Introduction

One of the principle design considerations for the Java platform is to give a restricted climate to executing code with various permission levels. Java comes with its own one of a kind arrangement of security challenges. While the Java security architecture can as a rule help to protect users and frameworks from antagonistic or getting rowdy code, it cannot guard against implementation bugs that occur in confided in code. Such bugs can unintentionally open the very openings that the security architecture was designed to contain. In serious cases local programs might be executed or Java security disabled. These bugs can conceivably be utilized to take confidential data from the machine and intranet, abuse framework resources, forestall valuable operation of the machine, help further attacks, and numerous other malicious activities.

The choice of language framework impacts the power of any software program. The Java language and virtual machine give numerous highlights to relieve common programming botches. The language is type-safe, and the runtime gives automatic memory management and bounds-checking on exhibits. Java programs and libraries check for unlawful state at the soonest opportunity. These highlights additionally make Java programs exceptionally impervious to the stack-smashing and buffer overflow attacks conceivable in the C and less significantly C++ programming languages. The explicit static composing of Java makes code straightforward (and facilitates static analysis), and the dynamic checks guarantee unexpected conditions bring about predictable conduct.

To limit the probability of security weaknesses caused by programmer error, Java engineers ought to cling to recommended coding guidelines. Existing publications, such as Effective Java, give excellent guidelines identified with Java software design. Others, such as Software Security: Building Security In, plot core values for software security. This document scaffolds such publications together and includes coverage of extra topics. It gives a more complete arrangement of security-specific coding guidelines focused at the Java programming language. These guidelines are important to all Java designers, regardless of whether they create confided in end-client applications, execute the internals of a security component, or create shared Java class libraries that perform common programming assignments. Any implementation bug can have genuine security ramifications and could show up in any layer of the software stack.

Use query parameterization to prevent injection

SQL injection appeared at the top of the list as the number one vulnerability in the OWASP top ten list. The following is an unsafe execution of SQL in Java, which can be used by an attacker to gain more information than otherwise intended.

```
public void selectExample(String parameter) throws SQLException {

   Connection connection = DriverManager.getConnection(DB_URL, USER,
PASS);

   String query = "SELECT * FROM USERS WHERE lastname = " + parameter;

   Statement statement = connection.createStatement();

   ResultSet result = statement.executeQuery(query);



   printResult(result);

}
```

If the parameter in this example is something like `" OR 1=1`, the result contains every single item in the table. This could be even more problematic if the database supports multiple queries and the parameter would be `"; UPDATE USERS SET lastname="`.

To avoid this in Java, we should parameterize the queries by using a prepared statement. This should be the only way to create database queries. By defining the full SQL code and passing in the parameters to the query later, the code is easier to understand. Most importantly, by distinguishing between the SQL code and the parameter data, the query can't be hijacked by malicious input.

```
public void prepStatmentExample(String parameter) throws SQLException
{
```

```
   Connection connection = DriverManager.getConnection(DB_URL, USER,
PASS);

   String query = "SELECT * FROM USERS WHERE lastname = ?";

   PreparedStatement statement = connection.prepareStatement(query);

   statement.setString(1, parameter);

   System.out.println(statement);

   ResultSet result = statement.executeQuery();



   printResult(result);

}
```

In the example above, the input binds to the type `String` and therefore is part of the query code. This technique prevents the parameter input from interfering with the SQL code.

Use OpenID Connect with 2FA

Identity management and access control is difficult and the reason for data breaches are because of the broken authentication. There are many things you should take into account when creating authentication yourself: secure storage of passwords, strong encryption, retrieval of credentials etc. In many cases it is much easier and safer to use exciting solutions like OpenID Connect. OpenID Connect (OIDC) enables you to authenticate users across websites and apps. This eliminates the need to own and manage password files. OpenID Connect is an OAuth 2.0 extension that provides user information. It adds an ID token in addition to an access token, as well as a **`/userinfo`** endpoint where you get additional information. It also adds an endpoint discovery feature and dynamic client registration. [1]

Scan your dependencies for known vulnerabilities

There's a good chance you don't know how many direct dependencies your application uses. It's also extremely likely you don't know how many transitive dependencies your application uses either. This is often true, despite dependencies making up for the majority of your overall application. Attackers target open-source dependencies more and more, as their reuse provides a malicious attacker with many victims. It's important to ensure there are no known vulnerabilities in the entire dependency tree of your application. [1]

Benefits of performing dependencies scanning:

- Ensure your application does not use dependencies with known vulnerabilities.
- Test your app dependencies for known vulnerabilities.
- Automatically fix any existing issues.
- Continuously monitor your projects for new vulnerabilities over time.

Handle sensitive data with care

Exposing sensitive data, like personal data or credit card numbers of your client, can be harmful. But even a more subtle case than this can be equally harmful. For example, the exposure of unique identifiers in your system is harmful if that identifier can be used in another call to retrieve additional data. First of all, you need to look closely at the design of your application and determine if you really need the data. On top of that, make sure that you don't expose sensitive data, perhaps via logging, auto completion, transmitting data etc.

An easy way to prevent sensitive data from ending up in your logs, is to sanitize the `toString()` methods of your domain entities. This way you can't print the sensitive fields by accident. If you need to send sensitive data to other services, encrypt it properly and ensure that your connection is secured with HTTPS, for instance. [1]

Sanitize all input

Cross-site scripting (XSS) is a well-known issue and mostly utilized in JavaScript applications. However, Java is not immune to this. XSS is nothing more than an injection of JavaScript code that's executed remotely. Rule #0 for preventing XSS, according to OWASP, is "Never insert untrusted data except in allowed locations". Although this is not an XSS attack, it is a good example of why you have to sanitize all input. Every input is potentially malicious and should be sanitized accordingly. [1]

Configure your XML-parsers to prevent XXE

With XML eXternal Entity (XXE) enabled, it is possible to create a malicious XML, as seen below, and read the content of an arbitrary file on the machine. It's not a surprise that XXE attacks are part of the OWASP Top 10 vulnerabilities.Java XML libraries are particularly vulnerable to XXE injection because most XML parsers have external entities by default enabled.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!DOCTYPE bar [

       <!ENTITY xxe SYSTEM "file:///etc/passwd">]>

<song>

   <artist>&xxe;</artist>

   <title>Bohemian Rhapsody</title>

   <album>A Night at the Opera</album>

</song>
```

Changing the default settings to disallow external entities and doctypes for xerces1 or xerces2, respectively, prevents these kinds of attacks.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
SAXParser saxParser = factory.newSAXParser();

factory.setFeature("https://xml.org/sax/features/external-general-
entities", false);

saxParser.getXMLReader().setFeature("https://xml.org/sax/features/exte
rnal-general-entities", false);

factory.setFeature("https://apache.org/xml/features/disallow-doctype-
decl", true);
```

Avoid Java serialization

Serialization in Java allows us to transform an object to a byte stream. This byte stream is either saved to disk or transported to another system. The other way around, a byte stream can be deserialized and allows us to recreate the original object. The deserializing part looks something like this.

```
ObjectInputStream in = new ObjectInputStream( inputStream );

return (Data)in.readObject();
```

There's no way to know what you're deserializing before you decoded it. It is possible that an attacker serializes a malicious object and send them to your application. Once you call `readObject()`, the malicious objects have already been instantiated. You might believe that these kinds of attacks are impossible because you need to have a vulnerable class on you classpath. However, if you consider the amount of class on your classpath that includes your own code, Java libraries, third-party libraries and frameworks— it is very likely that there is a vulnerable class available. If you need to implement serializable on your domain entities, it is best to implement its own `readObject()`, as seen below. This prevents deserialization. [1]

```
private final void readObject(ObjectInputStream in) throws
java.io.IOException {

throw new java.io.IOException("Deserialized not allowed");
```

```
}
```

Use strong encryption and hashing algorithms.

If you need to store sensitive data in your system, you have to be sure that you have proper encryption in place. First of all you need to decide what kind of encryption you need —for instance, symmetric or asymmetric. Also, you need to choose how secure it needs to be. Stronger encryption takes more time and consumes more CPU. The most important part is that you don't need to implement the encryption algorithms yourself. Encryption is hard and a trusted library solves encryption for you. For example if we want to encrypt something like credit card details, we probably need a symmetric algorithm, because we need to be able to retrieve the original number. Say we use the Advanced Encryption Standard (AES), which is currently the standard symmetric encryption algorithm for US federal organizations. Below, example shows how to use Authenticated Encryption with Associated Data (AEAD) with AES. This allows us to encrypt plaintext and provide associated data that should be authenticated but not encrypted. [1]

```
private void run() throws GeneralSecurityException {

   AeadConfig.register();

   KeysetHandle keysetHandle =
KeysetHandle.generateNew(AeadKeyTemplates.AES256_GCM);



   String plaintext = "I want to break free!";

   String aad = "Queen";



   Aead aead = keysetHandle.getPrimitive(Aead.class);

   byte[] ciphertext = aead.encrypt(plaintext.getBytes(),
aad.getBytes());
```

```
   String encr = Base64.getEncoder().encodeToString(ciphertext);

   System.out.println(encr);

   byte[] decrypted = aead.decrypt(Base64.getDecoder().decode(encr),
aad.getBytes());

   String decr = new String(decrypted);

   System.out.println(decr);

}
```

If we want to encrypt passwords, it is safer to use asymmetric encryptions, as we don't need to retrieve the original passwords but just match the hashes. BCrypt and, his succor, SCrypt are the most suitable for this job. Both are cryptographic hashes (one-way functions) and computationally difficult algorithms that consume a lot of time. This is exactly what you want, because brute force attacks take ages this way. What is a strong encryption algorithm today may be a weak algorithm a year from now. Therefore, encryption needs to be reviewed regularly to make sure you use the right algorithm for the job. [1]

Enable the Java Security Manager

By default, a Java process has no restrictions. It can access all sorts of resources, such as the file system, network, eternal processes and more. There is, however, a mechanism that controls all of these permissions, the Java Security Manager. By default, the Java Security Manager is not active and the JVM has unlimited power over the machine. Although we probably do not want the JVM to access certain parts of the system, it does have access. More importantly, Java provides APIs that can do nasty and unexpected things. Activating the Java Security Manager is easy. By starting Java with an extra parameter, you activate the security manager with the default policy `java -Djava.security.manager`.

However, the default policy it's likely not to fit entirely to the purpose of your system. You might need to create your own custom policy and supply that to the JVM. `java -Djava.security.manager -Djava.security.policy==/foo/bar/custom.policy.`

Centralize logging and monitoring

Security is not just about prevention. You also need to be able to detect when something goes wrong so you can act accordingly. It doesn't really matter which logging library you use. In general, everything that could be an auditable event should be logged. Things like Exceptions, logins and failed logins might be obvious, but you probably want to log every single incoming request including its origin. At least you know what, when, and how something happened, in case you are hacked. It is advisable that you use a mechanism to centralize logging. For example with tools like Kibana, all logs from all servers or systems can be made accessible and searchable for investigation. [1]

Next to logging, you should actively monitor your systems and store these values centralized and easily accessible. Things like CPU spikes or an enormous load from a single IP address might indicate a problem or an attack. Combine centralized logging and live monitoring with alerting so you get pinged when something strange happens. [1]

Conclusion

In this report, it is discussed about ten security best practices in the JAVA programming and how can we protect the application and systems from attackers and malware attacks. Most of the discussed attacks are already been in the OWASP top ten vulnerable list. These vulnerabilities and attacks leads to serious problems for any organizations and even individual persons. So it is necessary to keep in mind while developing any applications or system using JAVA programming languages and make sure that the programs and systems are being tested thoroughly before it is deployed in the production environment or expose to the outside world.

References

[1] J. M. Brian Vermeer, "snyk," 16 September 2019. [Online]. Available: https://snyk.io/blog/10-java-security-best-practices/. [Accessed 1 December 2020].