Secure Coding Practices in Java: Challenges and Vulnerabilities

Ryan Fasco, Abhilash Chauhan, John Oh, Sara Grammer

Introduction

- Common Vulnerability
  - Caused by small # of programming errors
  - Logic bomb
  - Buffer overflow

- Secure Coding
  - Reduces/ eliminates vulnerabilities
  - Protect from cyber attacks/ exploitation

Secure Coding in Java

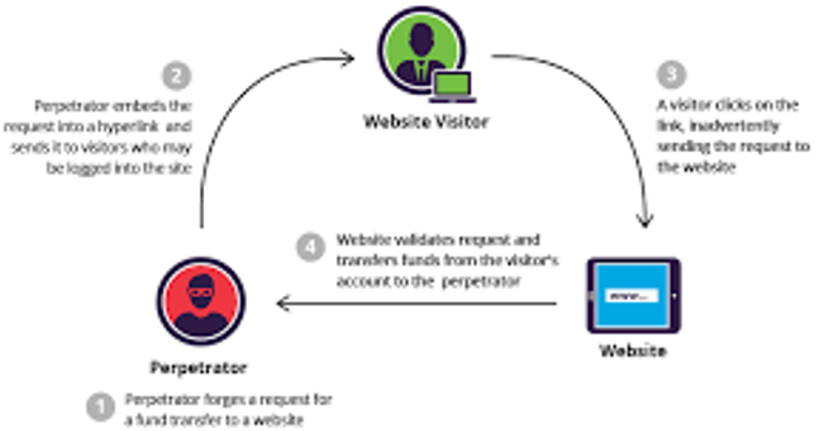
- Simple Java code is secure Java code
- Restrict privileges on the code
- Trust boundaries
- Design APIs with security in mind
- Only use trusted libraries
- Avoid sterilization
- Be aware of what stores sensitive information in Java
  - Errors
    - Use generic screen messages and helpful log messages
  - Exceptions
    - Filter sensitive information
  - Logging
    - Only log what is necessary and nothing too sensitive

Challenges

- Authentication
  - Integration between different types of applications
  - Implementation of security configurations
  - Conversion between configurations
- Cryptography
  - Vague error messages
  - Difficulties between different languages
  - Implicit constraints on API usage
- Java EE security
- Access control
  - Program context
  - Executable environment
- Secure communication
  - Validating SSL certificates
  - Establishing secure connection

Vulnerabilities

- Cross-site forgery
- SSL/TLS
- Password Hashing
- Misinformation



stackoverflow

What are the solutions?

1. Use query parameterization

Use prepared statements in Java to parameterize your SQL statements.

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

✅
```
String query = "SELECT * FROM USERS WHERE
lastname = ?";
   PreparedStatement statement =
connection.prepareStatement(query);
   statement.setString(1, parameter);
```

2. Use OpenID Connect with 2FA

OpenID Connect (OIDC) provides user information via an ID token in addition to an access token. Query the /userinfo endpoint for additional user information.

3. Scan your dependencies for known vulnerabilities

Ensure your application does not use dependencies with known vulnerabilities. Use a tool like Snyk to:

- Test your app dependencies for known vulnerabilities
- Automatically fix any existing issues
- Continuously monitor your projects for new vulnerabilities over time

4. Handle sensitive data with care

Sanitize the toString() methods of your domain entities.

If using Lombok, annotate sensitive classes. `@ToString.Exclude`

Use `@JsonIgnore and @JsonIgnoreProperties` to prevent sensitive properties from being serialized or deserialized.

5. Sanitize all input

Consider using the OWASP Java encoding library to sanitize input.

Assume all input is potentially malicious, and check for inappropriate characters (whitelist preferable).

6. Configure your XML parsers to prevent XXE

Disable features that allow XXE on your SAXParserFactory and SAXParser, or equivalent.

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

factory.setFeature("http://xml.org/sax/features/
external-general-entities", false);
saxParser.getXMLReader().setFea-
ture("http://xml.org/sax/fea-
tures/external-general-entities", false);
factory.setFeature("http://apache.org/xml/
features/disallow-doctype-decl", true);
```

7. Avoid Java serialization

If you must implement the serialization interface, override the readObject method to throw an exception.

```
private final void readObject(ObjectInputStream in)
throws java.io.IOException {
   throw new java.io.IOException("Not allowed");
}
```

If you have to deserialize, use the ValidatingObjectInputStream from Apache Commons IO to add some safety checks.

```
FileInputStream fileInput = new FileInputStream
(fileName);
ValidatingObjectInputStream in = new Validatin
gObjectInputStream(fileInput);
in.accept(Foo.class);

Foo foo_ = (Foo) in.readObject();
```

8. Use strong encryption and hashing algorithms

Always use existing encryption libraries, such as Google Tink, rather than doing it yourself.

For password hashing, consider using BCrypt or SCrypt. If using Spring, you can use it's built-in BCryptPasswordEncoder and SCryptPasswordEncoder for your hashing needs.

9. Enable the Java security manager

Enable via JVM properties on startup:

```
-Djava.security.manager
```

Create a policy that you use for your applications:

```
-Djava.security.policy==/my/custom.policy
```

10. Centralize logging and monitoring

Log auditable events, such as exceptions, logins and failed logins with useful information including their origin.

Centralize logs from multiple servers with tools like Kibana.

Monitor key system resources that indicate attack spikes or load from specific IP addresses.

Question

How can we enforce secure coding at VT?

References

https://www.whitehatsec.com/glossary/content/secure-coding

https://goldskysecurity.com/security-by-design-the-advantages-of-secure-coding-best-practices/

https://medium.com/@dinukadilshanfernando/what-is-secure-coding-ca56e36ce774

https://coralogix.com/blog/best-practices-for-writing-secure-java-code/

https://www.oracle.com/java/technologies/javase/seccodeguide.html