10 Java security best practices

In this cheat sheet edition, we’re going to focus on ten Java security best practices for both open source maintainers and developers. This cheat sheet is a collaboration between Brian Vermeer, Developer Advocate for Snyk and Jim Manico, Java Champion and founder of [Manicode](https://manicode.com/) Security.  
  
We recommend you [print out the cheat sheet](https://res.cloudinary.com/snyk/raw/upload/v1568658651/Cheat_Sheet-_10_Java_Security_Best_Practices.pdf) and also read more about each of the 10 Java security tips, which we discuss more in depth below.

So let’s get started!

Java security issues

Although we all aim to write great code, Java security is not always part of a developer’s mindset. However, preventing Java security issues should be just as important as making your Java application performant, scalable, and maintainable.  
In this cheatsheet, we discuss 10 common Java security issues. We will give you some pragmatic guidance and examples on how to prevent these common Java security vulnerabilities in the applications you write.

1. Use query parameterization to prevent injection

In the 2017 version of the [OWASP Top 10 vulnerabilities](https://snyk.io/learn/owasp-top-10-vulnerabilities/), injection appeared at the top of the list as the number one vulnerability that year. When looking at a typical [SQL injection](https://snyk.io/learn/sql-injection/) in Java, the parameters of a sequel query are naively concatenated to the static part of the query. 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 prevent this Java security risk, 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.

2. Use OpenID Connect with 2FA

Identity management and access control is difficult and broken authentication is often the reason for data breaches. In fact, this is #2 in the [OWASP top 10 vulnerability list](https://snyk.io/learn/owasp-top-10-vulnerabilities/) and, therefore, a major Java security risk. 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](https://openid.net/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](https://oauth.net/2/) extension that provides user information. It adds an ID token in addition to an access token, as well as a /userinfoendpoint where you get additional information. It also adds an endpoint discovery feature and dynamic client registration.

Setting up OpenID Connect with libraries like [Spring Security](https://spring.io/projects/spring-security) is a straightforward and common task. Make sure that your application enforces 2FA (two-factor authentication) or MFA (multi-factor authentication) to add an extra layer of security in your system.

By adding the oauth2-client and Spring security dependencies to your [Spring Boot](https://spring.io/projects/spring-boot) application, you leverage third-party clients like Google, Github and [Okta](https://www.okta.com/) to handle OIDC. After creating your application you just need to connect it to the specific client of your choice, by specifying it in your application configuration; that could be your GitHub or Okta client-id and client-secret, as shown below.

*pom.xml*

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

*application.yaml*

spring:

security:

oauth2:

client:

registration:

github:

client-id: 796b0e5403be4729ca01

client-secret: f379318daa27502254a05e054361074180b840a9

okta:

client-id: 0oa1a4wascEpYu6yk358

client-secret: hqxj7a9lVe\_TudbS2boBW7AWwxTlZiHNrJxdc\_Sk

client-name: Okta

provider:

okta:

issuer-uri: https://dev-844689.okta.com/oauth2/default

3. 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 Java security vulnerabilities in the entire dependency tree of your application.

Snyk tests your application build artifacts, flagging those dependencies that have known vulnerabilities. It provides you with a list of Java security vulnerabilities that exist in the packages you’re using in your application as a dashboard.

Additionally, it suggests upgrade versions or provide patches to remediate your security issues, via a pull request against your source code repository. Snyk also protects your environment by ensuring that any future pull requests raised on your repository are automatically tested (via webhooks) to make sure they do not introduce new known vulnerabilities.

Snyk is available via a web UI as well as a CLI, so you integrate it with your CI environment and configure it to break your build when vulnerabilities exist with a severity beyond your set threshold.

Use[Snyk for free](https://snyk.io/signup) for open source projects or for private projects with a limited number of monthly tests.

Fix Java vulnerabilities with a pull request

Find vulnerabilities in your Java app in seconds. Fix with an automated pull request.

[FIX FOR FREE](https://app.snyk.io/login?cta=sign-up&loc=body&page=10-java-security-best-practices)

4. 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 a Java security vulnerabilty 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, autocompletion, 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 use project Lombok to generate your toString() method, try using @ToString.Exclude to prevent a field from being part of the toString() output.

Also, be very careful with exposing data to the outside world. For instance: If we have an endpoint in a system that shows all usernames, there is no need to show the internal unique identifier. This unique identifier may be used to connect other, more sensitive information to the user by using other endpoints. If you use Jackson to serialize and deserialize POJOs to JSON try using  @JsonIgnore and @JsonIgnoreProperties to prevent these properties from being serialized or deserialized.

If you need to send sensitive data to other services, encrypt it properly and ensure that your connection is secured with HTTPS, for instance.

5. 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” The basic solution to this Java security risk is to prevent untrusted data, as much as possible, and sanitize everything else before using the data. A good starting point is the OWASP Java encoding library that provides you with a lot of encoders.

<dependency>

<groupId>org.owasp.encoder</groupId>

<artifactId>encoder</artifactId>

<version>1.2.2</version>

</dependency>

String untrusted = "<script> alert(1); </script>";

System.out.println(Encode.forHtml(untrusted));

// output: <script> alert(1); </script>

Sanitizing user text input is an obvious one. But what about the data you retrieve from a database, even when it’s your own database? What if your database was breached and someone planted some malicious text in a database field or document?

Also, keep an eye on incoming files. The [Zip-slip](https://snyk.io/research/zip-slip-vulnerability) vulnerability in many libraries exists because the path of the zipped files was not sanitized. Zip-files containing files with paths ../../../../foo.xy could be extracted and potentially override arbitrary files. 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.

6. 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 list and a Java security vulnerability we need to prevent. 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>

A naive implementation of the DefaultHandler and the Java SAX parser, like that seen below, parses this XML file and reveal the content of the passwd file. The Java SAX parser case is used as the main example here but other parsers, like DocumentBuilder and DOM4J, have similar default behavior.

SAXParserFactory factory = SAXParserFactory.newInstance();

SAXParser saxParser = factory.newSAXParser();

DefaultHandler handler = new DefaultHandler() {

public void startElement(String uri, String localName,String qName,Attributes attributes) throws SAXException {

System.out.println(qName);

}

public void characters(char ch[], int start, int length) throws SAXException {

System.out.println(new String(ch, start, length));

}

};

Changing the default settings to disallow external entities and doctypes for [xerces1](https://xerces.apache.org/xerces-j/) or [xerces2](https://xerces.apache.org/xerces-j/), 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/external-general-entities", false);

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

...

For more hands-on information about preventing malicious XXE injection, please take a look at the [OWASP XXE Cheatsheet](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html)

7. 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 biggest problem is with the deserializing part. Typically it 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.

Java serialization is also called “the gift that keeps on giving” because of the many problems it has produced over the years. Oracle is planning to eventually remove Java serialization as part of [Project Amber](https://openjdk.java.net/projects/amber/). However, this may take a while, and it’s unlikely to be fixed in previous versions. Therefore, it is wise to avoid Java serialization as much as possible. If you need to implement serializable on your domain entities, it is best to implement its own readObject(), as seen below. This prevents deserialization.

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

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

}

If you need to Deserialize an inputstream yourself, you should use an ObjectsInputStream with restrictions. A nice example of this is the ValidatingObjectInputStream from Apache Commons IO. This ObjectInputStream checks whether the object that is deserialized, is allowed or not.

FileInputStream fileInput = new FileInputStream(fileName);

ValidatingObjectInputStream in = new ValidatingObjectInputStream(fileInput);

in.accept(Foo.class);

Foo foo\_ = (Foo) in.readObject();

Object deserialization problems are not restricted to Java serialization. Deserialization from JSON to Java Object can contain similar problems. An example of such a deserialization issue with the Jackson library is in the blog post “[Jackson Deserialization Vulnerability](https://snyk.io/blog/jackson-deserialization-vulnerability/)”

8. 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.

If, for instance, 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. To encrypt and decrypt, there is no reason to deep-dive into into low level Java crypto. We recommend that you use a library that does the heavy lifting for you. For example, [Google Tink](https://github.com/google/tink).

<dependency>

<groupId>com.google.crypto.tink</groupId>

<artifactId>tink</artifactId>

<version>1.3.0-rc1</version>

</dependency>

Below, there’s a short example of 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.

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);

}

For 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.

Spring security provides excellent support for a wide variety of algorithms. Try using the SCryptPasswordEncoder and BCryptPasswordEncoder that Spring Security tool 5 provides for the purpose of password hashing

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. Use vetted security libraries for these tasks and keep your libraries up to date.

9. 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 don’t 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.

I think the scariest one is the Attach API. With this API you connect to other running JVM and control them. For instance, it is quite easy to change the byte the bytecode of a running JVM, if you have access to the machine. This [blog post](https://blog.frankel.ch/jvm-security/4/) by Nicolas Frankel gives an example of how this is done.

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

Note the double equals sign —this replaces the default policy. Using a single equals sign,  expands the default policy with your custom policy.

For more information on permissions in the JDK and how to write policy files check out [the official Java documentation](https://docs.oracle.com/en/java/javase/11/security/permissions-jdk1.html#GUID-1E8E213A-D7F2-49F1-A2F0-EFB3397A8C95)

10. 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. The important part is that you log a lot because insufficient logging is still a huge problem, according to the OWASP Top 10. 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. If you use log4j of logback, it is fairly easy to connect this to a centralized Elastic Stack, for instance. With tools like Kibana, all logs from all servers or systems can be made accessible and searchable for investigation.

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.

Thinks like admin password reset, internal server getting  accessed by an external IP or a URL parameter like ‘UNION’,  are just a few indicators that something is not ok. When you get the proper alerts on issues like these, and trace back what really happened, there’s a high chance you manage  to prevent any larger damage and fix the leak in time.