JAVAWORLD

eveloping secure Java applications

Make Java security a top priority at every stage of application development, from class-level language features to API endpoint authorization

By Matthew Tyson

Java Developer, JavaWorld

FEB 11, 2020 10:45 AM PST

Security is one of the most complex, broad, and important aspects of software development. Software security is also frequently overlooked, or oversimplified to just a few minor adjustments at the end of the development cycle. We can see the results in the annual list of major data security breaches, which in 2019 amounted to over 3 billion exposed records. If it can happen to Capital One, it can happen to you.

The good news is that Java is a longstanding development platform with many built-in security features. The Java Security package has undergone intensive battle testing, and is frequently updated for new security vulnerabilities. The newer Java EE Security API, released in September 2017, addresses vulnerabilities in cloud and microservices architectures. The Java ecosystem also includes a wide range of tools for profiling and reporting security issues.

But even with a solid development platform, it is important to stay vigilant. Application development is a complex undertaking, and vulnerabilities can hide in the background noise. You should be thinking about security at every stage of application development, from class-level language features to API endpoint authorization.

The following ground rules offer a good foundation for building more secure Java applications.

Java security rule #1: Write clean, strong Java code

Vulnerabilities love to hide in complexity, so keep your code as simple as possible without sacrificing functionality. Using proven design principles like DRY (don't repeat yourself) will help you write code that is easier to review for problems.

Always expose as little information as possible in your code. Hiding implementation details supports code that is both maintainable and secure. These three tips will go a long way toward writing secure Java code:

- **Make good use of Java's access modifiers**. Knowing how to declare different access levels for classes, methods, and their attributes will go a long way to protecting your code. Everything that can be made private, should be private.
- **Avoid reflection and introspection**. There are some cases where such advanced techniques are merited, but for the most part you should avoid them. Using reflection eliminates strong typing, which can introduce weak points and instability to your code. Comparing class names as strings is error prone and can easily lead to namespace collision.
- **Always define the smallest possible API and interface surfaces**. Decouple components and make them interact across the smallest area possible. Even if one area of your application is infected by a breach, others will be safe.

Java security rule #2: Avoid serialization

This is another coding tip, but it's important enough to be a rule of its own. Serialization takes a remote input and transforms it into a fully endowed object. It dispenses with constructors and access modifiers, and allows for a stream of unknown data to become running code in the JVM. As a result, Java serialization is deeply and inherently insecure.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

The end of Java serialization

If you haven't heard, Oracle has long-term plans to remove serialization from Java. Mark Reinhold, chief architect of the Java platform group at Oracle, has said that he believes a third or more of all Java vulnerabilities involve serialization.

As much as possible, avoid serialization/deserialization in your Java code. Instead, consider using a serialization format like JSON or YAML. Never, ever expose an unprotected network endpoint that receives and acts upon a serialization stream. This is nothing but a welcome mat for mayhem.

Java security rule #3: Never expose unencrypted credentials or PII

It's hard to believe, but this avoidable mistake causes pain year after year.

When a user enters a password into the browser, it is sent as plaintext to your server. That should be the last time it sees the light of day. You *must* encrypt the password via a one-way cypher before persisting it to the database, and then do it again whenever comparing against that value.

The rules for passwords apply to all personally identifiable information (PII): credit cards, social security numbers, etc. Any personal information entrusted to your application should be treated with the highest level of care.

Unencrypted credentials or PII in a database is a gaping security hole, waiting for an attacker to discover. Likewise, never write raw credentials to a log, or otherwise transmit to file or network. Instead, create a salted hash for your passwords. Be sure to do your research and use a recommended hashing algorithm.

Jumping down to Rule #4: always use a library for encryption; do not roll your own.

Java security rule #4: Use known and tested libraries

Feast your eyes on this question-and-answer about rolling your own security algorithm. The tl;dr lesson is: use known, reliable libraries and frameworks whenever possible. This applies across the spectrum, from password hashing to REST API authorization.

Fortunately, Java and its ecosystem have your back here. For application security, Spring Security is the de facto standard. It offers a wide-range of options and the flexibility to fit with any app architecture, and it incorporate a range of security approaches.

Your first instinct in tackling security should be to do your research. Research best-practices, and then research what library will implement those practices for you. For instance, if you are looking at using JSON Web Tokens to manage authentication and authorization, look at the Java library that encapsulates JWT, and then learn how to integrate that into Spring Security.

Even using a reliable tool, it is fairly easy to bungle authorization and authentication. Be sure to move slowly and double check everything you do.

Java security rule #5: Be paranoid about external input

Whether it comes from a user typing into a form, a datastore, or a remote API, never trust external input.

SQL injection and cross-site scripting (XSS) are just the most commonly known attacks that can result from mishandling external input. A less known example--one of many--is the "billion laughs attack," whereby XML entity expansion can cause a Denial of Service attack.

Anytime you receive input, it should be sanity checked and sanitized. This is especially true of anything that might be presented to another tool or system for processing. For example, if something could wind up as an argument for a OS command-line: beware!

A special and well-known instance is SQL injection, which is covered in the next rule.

Java security rule #6: Always use prepared statements to handle SQL parameters

Anytime you build up an SQL statement, you risk interpolating a fragment of executable code.

Knowing this, it's a good practice to *always* use the java.sql.PreparedStatement class to create SQL. Similar facilities exist for NoSQL stores like MongoDB. If you are using an ORM layer, the implementation will use `PreparedStatements` for you under the hood.

Java security rule #7: Don't reveal implementation via error messages

Error messages in production can be a fertile source of information for attackers. Stack traces, especially, can reveal information about the technology you are using and how you're using it. Avoid revealing stack traces to end users.

Failed-login alerts also fall into this category. It is generally accepted that an error message should be given as "Login failed" versus "Did not find that user" or "Incorrect password." Offer as little help to potentially nefarious users as possible.

Ideally, error messages should not reveal the underlying technology stack for your application. Keep that information as opaque as possible.

Java security rule #8: Keep security releases up to date

As of 2019, Oracle has implemented a new licensing scheme and release schedule for Java. Unfortunately for developers, the new release cadence does not make things easier. Nonetheless, you are responsible for frequently checking for security updates and applying them to your JRE and JDK.

Make sure you know what critical patches are available by regularly checking the Oracle homepage for security alerts. Every quarter, Oracle delivers an automated patch update for the current LTS (long-term-support) release of Java. The trouble is, that patch is only available if you are paying for a Java support license.

If your organization is paying for such a license, follow the auto-update route. If not, you're probably using OpenJDK, and you'll have to do the patching yourself. In this case, you could apply the binary patch, or you could simply replace your existing OpenJDK install with the latest version. Alternatively, you could use a commercially supported OpenJDK like Azul's Zulu Enterprise.

Do you need every security patch?

If you watch the security alerts closely, you may find you don't need a given set of updates. For example, the January 2020 release *appears* to be a critical Java update; however, a close reading shows that the update only patches holes in Java applet security, and does not affect Java servers.

Java security rule #9: Look for dependency vulnerabilities

There are many tools available to automatically scan your codebase and dependencies for vulnerabilities. All you have to do is use them.

OWASP, the Open Web Application Security Project, is an organization dedicated to improving code security. OWASP's list of trusted, high-quality automated code scanning tools includes several Java-oriented tools.

Check your codebase regularly, but also keep an eye on third-party dependencies. Attackers target both open- and closed-source libraries. Watch for updates to your dependencies, and update your system as new security fixes are released.

Java security rule #10: Monitor and log user activity

Even a simple brute-force attack can be successful if you aren't actively monitoring your application. Use monitoring and logging tools to keep an eye on app health.

If you'd like to be convinced why monitoring is important, just sit and watch TCP packets on your applications listening port. You'll see all kinds of activity, well beyond simple user interactions. Some of that activity will be bots and evil-doers scanning for vulnerabilities.

You should be logging and monitoring for failed login attempts and deploying counter-measures to prevent remote clients from attacking with impunity.

Monitoring can alert you to unexplained spikes, and logging can help unravel what went wrong following an attack. The Java ecosystem includes a wealth of commercial and open source solutions for logging and monitoring.

Java security rule #11: Watch out for Denial of Service (DoS) attacks

Anytime you are processing potentially expensive resources or undertaking potentially expensive operations, you should guard against runaway resource usage.

Oracle maintains a list of potential vectors for this type of problem in its Secure Coding Guidelines for Java SE document, under the "Denial Of Service" heading.

Basically, anytime you go to perform an expensive operation, like unzipping a compressed file, you should monitor for exploding resource usage. Don't trust file manifests. Trust only the actual on-disk or in-memory consumption, monitor it, and guard against bring-the-server-to-its-knees excesses.

Similarly, in some processing it is important to watch for unexpected forever-loops. If a loop is suspect, add a guard that ensures the loop is making progress and short-circuit it if it appears to have gone zombie.

Java security rule #12: Consider using the Java security manager

Java has a security manager that can be used to restrict the resources a running process has access to. It can isolate the program with respect to disk, memory, network, and JVM access. Narrowing down these requirements for your app reduces the footprint of possible harm from an attack. Such isolation can also be inconvenient, which is why `SecurityManager` isn't enabled by default.

You'll have to decide for yourself whether working around `SecurityManager`'s strong opinions is worth the extra layer of protection for your applications. See the Oracle docs to learn more about the syntax and capabilities of the Java security manager.

Java security rule #13: Consider using an external cloud authentication service

Some applications simply must own their user data; for the rest, a cloud service provider could make sense.

Search around and you'll find a range of cloud authentication providers. The benefit of such a service is that the provider is responsible for securing sensitive user data, not you. On the other hand, adding an authentication service increases the complexity of your enterprise architecture. Some solutions, like FireBase Authentication, include SDKs for integrating across the stack.

Conclusion

I've presented 13 rules for developing more secure Java applications. These rules are tried-and-true, but the greatest rule of all is this: be suspicious. Always approach software development with wariness and a security-minded outlook. Look for vulnerabilities in your code, take advantage of the Java security APIs and packages, and use third-party tools to monitor and log your code for security issues.

Here are three good high-level resources for staying abreast of the ever-changing Java security landscape:

- OWASP Top 10
- CWE Top 25
- Oracle's Secure Code Guidelines