IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/44SE/2024/4423924/
Vol. 2024 (24), Article ID 4423924

Security Architecture in Java – A pragmatic approach*

Jerzy KRAWIEC[1], Piotr GÓRNY[2], Maciej KIEDROWICZ[2] and Tomasz Kowalczyk[1]

Warsaw University of Technology, Warsaw, Poland

Correspondence should be addressed to: Jerzy KRAWIEC, jerzy.krawiec@pw.edu.pl

* Presented at the 44th IBIMA International Conference, 27-28 November 2024 Granada, Spain

Abstract

The Java security architecture model based on constructing secure code uses various security mechanisms, such as static typing, access modifiers, automatic memory management, and bytecode verification. However, such a model does not consider security principles that include interoperable and extensible vendor implementations and hashing processes. This paper proposes a new, pragmatic approach to Java security architecture that considers more security attributes. The new security architecture model extends the basic security model provided at the language function level. This pragmatic approach to Java security architecture includes Sandbox security principles, Java cryptographic architecture, interoperable and extensible implementations, and hashing processes.

Keywords: Java code, security architecture, sandbox, JCA, JAAS, JSSE

Introduction

Java security can be considered in two dimensions: ensuring the safe execution of Java applications and providing tools implemented in vulnerable applications (Java SE Platform Security Architecture, 2020).

The Java security architecture includes many APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols (Java Security Overview, 2020). It also can set system properties regarding the trace messages that will be displayed (Troubleshooting Security, 2020).

Java is at the forefront of programming security systems. Java's multi-layered security architecture includes the runtime environment, providing strong protection against various threats. Its design includes a strong security model for use in various applications (Obregon, 2023).

Java components enable a comprehensive approach to securing runtime environments. The security manager, access controller, bytecode verifier, and class loader work together to create a powerful barrier against multiple threats. These mechanisms ensure the integrity of the Java Virtual Machine (JVM) and the host system. This approach represents the first line of defence against executing potentially dangerous code. Java's strengths as a secure computing platform are its security functions and its integration.

As a versatile and efficient programming language, Java is used in various applications and environments, from web applications to embedded systems (Bruno, 2021). Java security covers the measures and practices used to protect Java applications and the Java Runtime Environment (JRE) from vulnerabilities, exploits, and unauthorized access. Java has a robust security model that covers various security areas such as bytecode verification, class loaders, Security Manager, cryptography, access control, authentication and authorization service (JAAS), and Java

Cite this Article as: Jerzy KRAWIEC, Piotr GÓRNY, Maciej KIEDROWICZ and Tomasz Kowalczyk, Vol. 2024 (24) “Security Architecture in Java – A pragmatic approach" Communications of International Proceedings, Vol. 2024 (24), Article ID 4423924, https://doi.org/10.5171/2024.4423924

Cryptography Architecture (Baeldung, 2024). In such a Java security model, Security Manager plays an important role. It is a critical component that controls access to system resources.

Java security is a critical aspect of building and maintaining applications. Understanding the security architecture, using Security Manager and policy files, and implementing cryptographic techniques are fundamental principles in securing Java applications.

Java's security architecture provides some standard cryptographic services. Java patterns create opportunities to extend security, primarily in authentication and access control (Chandrakant, 2019).

Java security is also created by following best practices, which include several stages. The security patches should be updated to eliminate security vulnerabilities. It uses user input validation and sanitization to avoid code injection. Authentication and authorization mechanisms should be introduced to control access to resources. Implementing the principle of least privilege involves granting only the minimum necessary encoding permissions. As part of secure coding, we should avoid typical vulnerabilities such as buffer overflows and SQL injection. The security tests involve assessing the code's vulnerability and performing penetration tests. The final step is to monitor and record security incidents to detect and respond appropriately.

Securing Java code is important because the standard JVM security features only sometimes provide specific security requirements. Leveraging the JVM's built-in security mechanisms provides a unified and proven approach to security, but may require greater flexibility for high-security requirements. However, in some situations, deviating from the classroom security rules, e.g., using an unsafe mechanism, is necessary. However, this mechanism is potentially very dangerous because it allows breaking basic linguistic rules (Evans, 2020). The situation is similar with methods and constructors that have some value restrictions on passing values to their parameters. In such a situation, the Builder pattern can be used to create objects with many optional parameters (Kiwy, 2021). Therefore, the appropriate approach depends on the application's security needs and trade-offs.

New access control mechanisms must be backward compatible. It means that all checking methods in Security Manager must still be supported. Security Manager represents a central access control point, while Access Controller implements a specific access control algorithm with unique features. The Security Manager update provides backward compatibility and flexibility when implementing a multi-level security model ('The Security Manager, 2022). Therefore, implementing the Access Controller algorithm increases the application's security without requiring extensive code to be written. However, Security Manager should be used before invoking standard security checks, and in some cases, it can be supplemented with the Access Controller algorithm (Krawiec et al., 2024).

Even though the Java platform was designed with particular attention to security, it only avoided several flaws that are constantly discovered over time. One of the security flaws in Java is the lack of automatic updating, which creates confusion every time it is updated. Then, a message is generated about the application being blocked by Java Security (How to Launch The Application Blocked by Java Security, 2022).

In third-party libraries where native code (native methods) are used, there is a risk of security breaches because machine code may have access to the operating system (Parkinson, 2021); (Schildt, 2021).

This paper proposes a new, pragmatic approach to Java's security architecture that considers more security attributes. The new security architecture model extends the basic security model provided at the language function level. It allows one to write secure code and take advantage of hidden security features such as static data typing, access modifiers, automatic memory management, and bytecode verification. However, such a model does not include security policies that include interoperable and extensible vendor implementations and hashing processes.

Model of Security Architecture

Java's security architecture covers several areas. Some areas, such as access modifiers and class loaders, are part of the language itself, and others are available as services that include data encryption, secure communications, authentication, and authorization, among others.

Access modifiers (public, protected, private) determine the visibility of fields, methods, classes, and interfaces. They determine the method of access to data and ensure its security. In this way, access modifiers fulfil the principle of data encapsulation.

In contrast, class loaders are part of the Java Runtime Environment (JRE) that load class files into the Java Virtual Machine (JVM). Security is achieved by splitting class loaders into a system class loader and a user-defined class loader. The system loads classes from the local file system that should be trusted. In contrast, the user-defined class loader can load classes from various locations (e.g., the Internet) that may be potentially untrusted.

Ensuring application security requires avoiding critical programming errors. Even minor errors can constitute a severe vulnerability in the IT system, which poses significant threats to users. Therefore, the primary way to avoid these errors is to create stable source code and verify this process before implementing the application (Krawiec et al, 2022).

Ensuring Java security at the level of language functions allows one to write secure code and take advantage of many hidden security functions such as static data typing, access modifiers, automatic memory management, and byte code verification. Java is strongly typed and is a statically typed language, which limits the ability to detect type errors at runtime. Java uses various access modifiers such as public, protected, and private to control access to fields, methods, and classes. Java has a built-in garbage collector-based memory management feature, which frees developers from manually managing the process.

However, such a process is not entirely secure because malicious code, e.g., a subclass finalizer, may operate on a partially constructed object, and such an object should be removed during its creation. Additionally, the finalizer may store a reference to an object in the statistics field, which means that the object cannot be deleted. It means that a corrupted object is created from which arbitrary methods can be called. In such a situation, an exception raised from the constructor level should prevent the creation of such an object (Krawiec et al., 2023).

Java is a compiled language that converts code into platform-independent bytecode. In contrast, the runtime verifies every bytecode it loads for execution.

New approach to security architecture

Sandbox – foundation of the Java security architecture

The new security architecture model also considers such security attributes as secure communication techniques and tools, e.g., cryptography and interoperable and extensible vendor implementations.

The main component of the security model in Java is a sandbox that provides a controlled execution environment for untrusted code. It limits access to key application areas and information resources. Restricting access to untrusted code prevents any operation that may compromise system security.

The original security model provided by the Java platform is known as the sandbox model. We can run untrusted code from the open web to provide a minimal environment. A sandbox is an isolated environment in which a program or file can run without affecting the application in which it runs. It is a security mechanism that separates running programs, usually to minimize system failure or the spread of software vulnerabilities.

Constructing the sandbox starts with a Security Policy containing a list of security domains. The *PolicyFile* class is the default implementation of the Security Policy. Settings can be changed via the *policy.provider* properties. Editing Security Policy files is provided by the *policytool* utility. An example Security Policy file might look like this:

```
keystore "${user.home}${/}.keystore";
grant codeBase "http://pjwstk.edu.pl/" {
  permission java.io.FilePermission "/tmp", "read";
  permission java.lang.RuntimePermission "queuePrintJob";};
grant signedBy "George", codeBase "file:${java.home}/lib/ext/." {
  permission java.security.AllPermission;};
grant signedBy "George", codeBase "file:${java.class.path}/." {
  permission java.net.SocketPermission "*:1024-",
    "accept, connect, listen, resolve";};
grant {
  permission java.util.Permission "java.version", "read";};
```

The sandbox implementation includes a Security Manager, Access Controller, and Class Loader. Security Manager provides mechanisms used in the API to check operations for violations of established security policy. When verifying the rights to perform a given operation, the Security Manager uses the functionality provided by the Access Controller. The Class Loader, on the other hand, ensures loading data streams (e.g., files) and converting them into class definitions in the JRE. The Class Loader encapsulates information about the loaded classes and cooperates with the JRE to determine the namespace. A separate instance of Class Loader is created for each codebase. This mechanism makes it impossible to replace classes from other codebases. To check whether the loaded code of a class has access to it, the Class Loader calls the Security Manager.

Figure 1 shows the logical diagram of the sandbox model.

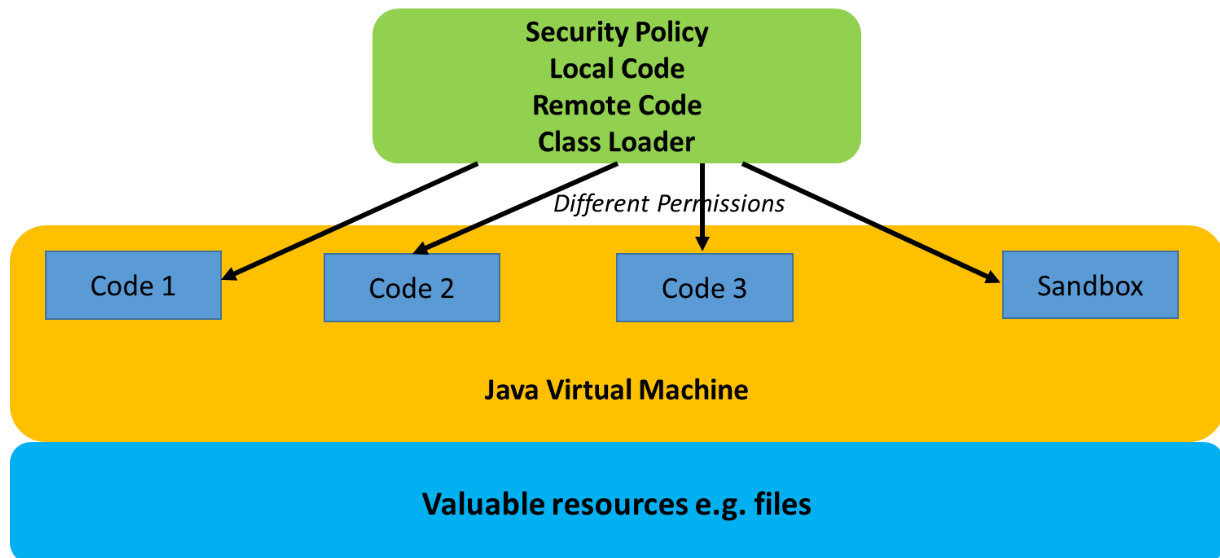

Figure 1. Sandbox model in Java.

A sandbox typically provides a tightly controlled set of resources on which other programs can run, such as limited disk space and memory. Sandbox constraints limit the system resources that an applet can request or access.

Bytecode compilers and verifiers ensure that only valid Java bytecodes are executed. The bytecode verifier and the Java virtual machine ensure the language is safe at runtime. Class Loader defines a local namespace that can be used to ensure that an untrusted applet does not interfere with other programs.

Access to key system resources takes place via the JVM and is checked in advance by the *SecurityManager* class, which limits the operation of untrusted code fragments to an absolute minimum (*SandBoxing*). Sandboxing is often used to test unverified programs that may contain viruses or other malicious code, preventing the software from harming the host.

Starting with the release of JDK 8u51, the following changes have been made to the native Sandbox environment. The native Sandbox environment is only available on Windows operating systems. They can be turned on and off. By default, the native sandbox environment is disabled.

With native Sandbox enabled, applets and Web Start applications designed to run in the Sandbox environment will run in a restricted environment. This access is managed by the operating system. Applications with all permissions are not affected.

Native Sandbox will be disabled for applications included in the Exception Site List (ESL) and when Deployment Rule Set (DRS) is used. Sandbox-targeted applets with an "applet" HTML tag that includes JAR files with all permissions derived from the Class-Path attribute of the manifest run in the native Sandbox environment. In this case, when such an applet tries to access JAR files with all permissions, a special warning dialog box will be displayed, informing the user about the possible incorrect operation of the applet.

When the native Sandbox environment is enabled, the custom preloader may be disabled. Typically, this module is disabled when applets or Web Start applications intended for the Sandbox environment are initialized. In this situation, the default bootstrap module is used. When initializing the application, the JVM starts with native Sandbox enabled and a custom preloader will be used.

For applications with all permissions, the custom bootstrap module is disabled until the user allows the application to run from the "Security" dialog, which means that the custom bootstrap module will gain privileged access to the application. Therefore, the proposed Java security architecture model should be as shown in Figure 2.

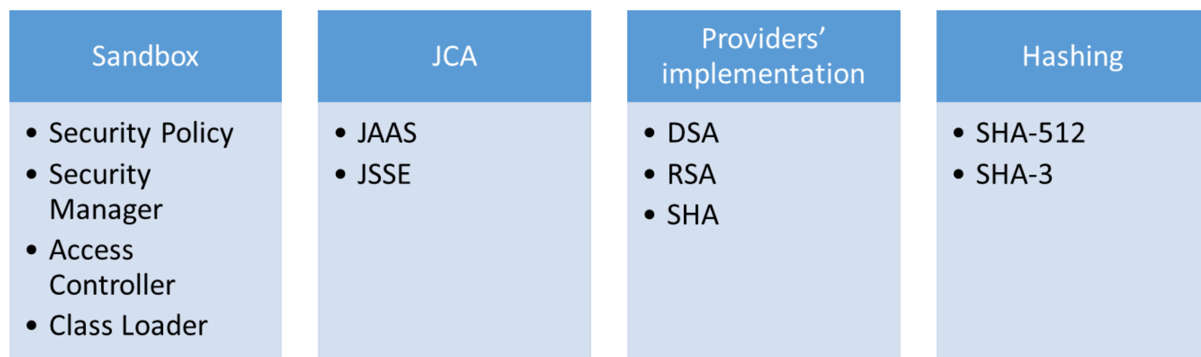

Figure 2. The new model of Java security architecture.

The presented Java security architecture model includes key components that ensure the secure design of Java applications.

Java Cryptographic Architecture

The Java Cryptographic Architecture (JCA) includes a platform for accessing and implementing cryptographic functions such as: digital signatures, message digests, ciphers (symmetric and asymmetric), message authentication codes and key generators and key factories. Java uses provider-based implementations of cryptographic functions. Contains commonly used cryptographic algorithms such as DSA, RSA and AES. These algorithms can be used to protect data at various stages of processing. A very common use case in applications is storing user passwords. One solution is to encrypt passwords using a hash function, e.g. SHA-512. Then the code looks like this:

```
MessageDigest beta = MessageDigest.getInstance("SHA-512");
bajt[] hashedPassword = beta.digest("hasło".getBytes());
```

In this case, we use the *MessageDigest* cryptographic service and the *getInstance()* method to request this service from any available security providers.

Public key infrastructure (PKI) is the configuration that enables the secure exchange of information over a network using public key encryption. This configuration is based on trust built between the parties involved in communication. This trust is based on digital certificates issued by a neutral and trusted authority known as a Certificate Authority (CA).

The Java platform has APIs to facilitate creating, storing, and verifying *KeyStore* and *CertStore* digital certificates. Java provides the *KeyStore* class for persistent storage of cryptographic keys and trusted certificates. In this case, *KeyStore* can represent both key-store and trust-store files. These files have similar content but differ in how they are used. *CertStore* represents a public repository of potentially untrusted certificates and revocation lists.

Java has a built-in trust-store called "cacerts," which contains certificates from well-known certificate authorities. Java has some valuable tools to facilitate trusted communication such as "keytool" (a built-in tool called for creating and managing a key-store and a trust-store) and "jarsigner," which can be used to sign and verify JAR files.

In order to establish a secure connection using the SSL protocol, we must present the certificate and verify it. So, we need to present a valid certificate to the other party of communication. For this, we need to load the key-store file where we need to have our public keys:

```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
char[] keyStorePassword = "changeit".toCharArray();
try(InputStream keyStoreData = new FileInputStream("keystore.jks")){
   keyStore.load(keyStoreData, keyStorePassword);}
```

We also need to verify the certificate presented by the other party in the communication. For this, we need to load a trusted store where we must have previously trusted certificates of other parties:

```
KeyStore trust-store = KeyStore.getInstance(KeyStore.getDefaultType());
// We load the trust-store from filesystem as before
-Djavax.net.ssl.trust-store=trust-store.jks
-Djavax.net.ssl.keyStore=keystore.jks
```

We typically pass system parameters to Java at runtime.

One of JCA's strengths is its vendor-driven architecture. It enables a flexible and extensible approach to cryptography. We can consider providers as packages implementing cryptographic services, algorithms, and keys. Java ships with default providers but can plug in external providers.

An example of such a Delta supplier attachment could be:

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
public class Delta {
   public static void main(String[] jekr) {
      Security.addProvider(new BouncyCastleProvider());}}
```

Thus, JCA provides a flexible and secure platform for integrating cryptographic functions into Java applications. Developers can create robust security features critical to protecting sensitive data and ensuring the integrity and authenticity of information.

Java Authentication and Authorization Service

Java Authentication and Authorization Service (JAAS) assumes that a secure system must know the identity of users accessing the system and should be able to control the resources to which the user has access. JAAS has two processes: authentication and authorization. These processes are crucial in any secure system. JAAS extends Java's security capabilities with a flexible and scalable authentication (who you are) and authorization (what you can do) framework.

The authentication phase consists of confirming or denying the declared identity of the entity. A subject means an entity such as a user, process, or device trying to access the system. JAAS uses an architecture that can integrate any authentication mechanism without changing the basic structure of the application. *LoginModules* provide authentication configured in the *login.config* file. The files can challenge an entity's credentials, such as passwords, biometrics, smart cards, and more. *LoginContext*, on the other hand, is the central class in the authentication process that checks the configuration for the use of *LoginModule* modules. Initializing the *LoginContext* object and using it to authenticate the user might look like this:

```
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
public class JAASAuthenticationExample {
   public static void main(String[] jekr) {
      try {
// Create a LoginContext with a callback handler
         LoginContext gamma = new LoginContext("Sample", new MyCallbackHandler());
  // Attempt to authenticate the user
         gamma.login();
 // If authentication succeeds, proceed with the application logic
      } catch (LoginException le) {
```

```
System.err.println("Authentication failed: " + le.getMessage());}}
```

The *MyCallbackHandler* class implements *CallbackHandler* and handles prompts for user credentials.
The second stage of JAAS is authorization, which uses system-recognized identities such as usernames or group names. The purpose of this stage is to determine the activities that a given entity can perform. If a policy is configured for a given application, security principals should be associated with a set of permissions. The Policy class defines the permissions that are granted to security principals in various contexts. If an entity tries to perform a secured action while a given application is running, the security policy checks whether the entity has been granted such permissions.

The authorization code in JAAS, assuming that all files (*sample_jaas.config*, *samplezn.policy*, *SampleAzn.java*, *SampleAction.java*, *SampleLoginModule.java* and *SamplePrincipal.java*) are saved in the *Sample* subdirectory), may be as follows:

```
Sample {
  sample.module.SampleLoginModule required debug=true; };
grant codebase "file:./SampleLM.jar" {
   permission javax.security.auth.AuthPermission "modifyPrincipals";};
grant codebase "file:./SampleAzn.jar" {
  permission javax.security.auth.AuthPermission "createLoginContext.Sample";
  permission javax.security.auth.AuthPermission "doAsPrivileged";};
grant codebase "file:./SampleAction.jar",
          Principal sample.principal.SamplePrincipal "testUser" {
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "user.home", "read";
  permission java.io.FilePermission "a.txt", "read";};
```

Based on the *sampleazn.policy* file, user actions can be authorized:

```
javac sample/SampleAction.java sample/SampleAzn.java sample/module/SampleLoginModule.java
sample/principal/SamplePrincipal.java
jar -cvf SampleAzn.jar sample/SampleAzn.class  sample/MyCallbackHandler.class
jar -cvf SampleAction.jar sample/SampleAction.class
jar -cvf SampleLM.jar sample/module/SampleLoginModule.class sample/principal/SamplePrincipal.class
java -classpath SampleAzn.jar;SampleAction.jar;SampleLM.jar
 -Djava.security.manager
 -Djava.security.policy==sampleazn.policy
 -Djava.security.auth.login.config==sample_jaas.config sample.SampleAzn
```

Therefore, JAAS allows the creation of secure systems that manage user identities and enforce strict access control policies. JAAS facilitates the integration of various authentication and authorization mechanisms and creates a secure and friendly user environment.

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) is a set of packages that support and implement the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. These protocols are commonly used to secure data transmitted over networks. JSSE enables secure network communications using standard protocols and encryption techniques.

JSSE provides the structure and implementation of SSL and TLS protocol versions and includes features for data encryption, server authentication, message integrity, and optional client authentication. This service is commonly used in web browsers, servers, and web applications requiring secure communication, e.g., HTTPS protocols.
Considering the SSL context, it is essential to mention that the *SSLContext* class acts as a factory for *SSLSocketFactory* and *SSLServerSocketFactory*. The *SSLContext* instance encapsulates all security settings and creates *SSLSocket* and *SSLServerSocket* instances. The *SSLSocket* and *SSLServerSocket* classes are subclasses of *Socket* and *ServerSocket*, respectively, and provide secure communication through sockets. For this purpose, they use SSL or TLS protocols to encrypt data sent over the network. A standard SSL/TLS handshake involves several steps, including exchanging cryptographic parameters, server authentication, optionally client authentication, and establishing session keys for data encryption. JSSE can be used as a simple way to implement an SSL socket on the client side. The record may take the form:

```
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.SSLSocket;
```

```
public class SimpleSSLSocketClient {
   public static void main(String[] jekr) {
      try {
// Use SSLSocketFactory
         SSLSocketFactory kappa = (SSLSocketFactory) SSLSocketFactory.getDefault();
// Create SSLSocket
         SSLSocket omega = (SSLSocket) factory.createSocket("hostname", 443);
// Start handshake
         omega.startHandshake();
// Secure communications using omega and close the socket
         omega.close();
      } catch (Exception e) {
         e.printStackTrace();}}}
```

For a server-side SSL socket, *SSLServerSocketFactory* can create an *SSLServerSocket*.

Providers' implementations

A specific provider implementation can implement selected security services. Typical services that a vendor can extend Java's security architecture include cryptographic algorithms (DSA, RSA, or SHA) and the generation, conversion, and management of cryptographic keys, e.g., algorithm-specific keys. Figure 3 shows an example application configuration using additional security services.

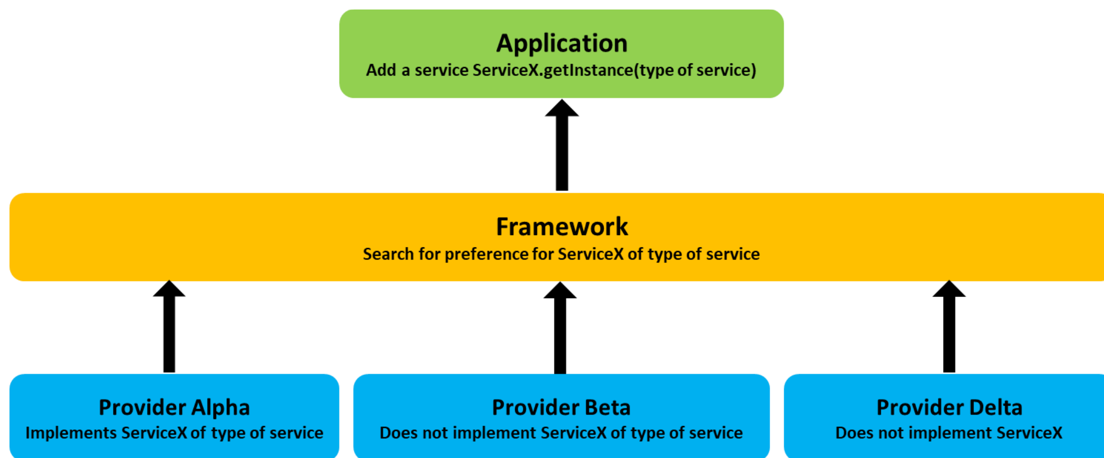

Figure 3. The example of different application configuration for security by 3 providers.

It is an essential list of security features available in Java and is good enough to provide an appropriate level of security for the application. Based on this set of security functions, the framework searches for a specific service from providers depending on the preferences set for them. In this architecture, it is always possible to implement appropriate security functions from different vendors. A specific provider implementation may implement some or all of the security services.

Hashing

Hashing is important from the security point of view of the password programming language. Hashing is the process of generating a string of characters (hash) from a given message using a mathematical function known as a cryptographic hash function. Such a function should be deterministic, irreversible, have high entropy, and be collision-resistant (Millington, 2024). When met simultaneously, these features increase the difficulty of reconstructing the password based on the hash.

Some hash functions previously used, such as MD5, SHA-1, and SHA-2, are no longer recommended because new vulnerabilities are found in the algorithms when computers become more powerful.

As computers become more powerful and new vulnerabilities are found, researchers are developing new versions of SHA. Although there are currently more secure versions of SHA, such as SHA-3, SHA-512 is the strongest algorithm implemented in Java.

Implementing the SHA-512 hashing algorithm in Java uses the so-called salt, a random sequence generated for each new hash. Introducing randomness increases hash entropy and protects the database from pre-compiled lists of hashes called rainbow tables. So, the new hash function has the following form::

```
salt <- generate-salt;
hash <- salt + ':' + sha512(salt + password);
```

Introducing the salt involves using the *SecureRandom* class from *java.security* package:

```
SecureRandom alpha = new SecureRandom();
byte[] salt = new byte[16];
alpha.nextBytes(salt);
// We will use the MessageDigest class to configure the SHA-512 hash function with our salt
MessageDigest beta = MessageDigest.getInstance("SHA-512");
// We can now use the digest() method to generate our hashed password
beta.update(salt);
byte[] hashedPassword = beta.digest(passwordToHash.getBytes(StandardCharsets.UTF_8));
```

The SHA-512 function, combined with salt, is a good option, but there are stronger and slower algorithms that better suit our requirements in this regard, such as PBKDF2, BCrypt, and SCrypt.
Each of these algorithms is slow and has an important feature - configurable strength. As computers become more powerful, we can slow down the algorithm by changing the input data.
Since salt is the basic principle of password hashing, we also use it in the PBKDF2 algorithm:

```
SecureRandom alpha = new SecureRandom();
byte[] salt = new byte[16];
alpha.nextBytes(salt);
/* We will create a PBEKeySpec and a SecretKeyFactory which we'll instantiate using the
PBKDF2WithHmacSHA1 algorithm*/
KeySpec delta = new PBEKeySpec(password.toCharArray(), salt, 65536, 128);
SecretKeyFactory omega = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512");
// We use SecretKeyFactory to generate the hash
byte[] hash = factory.generateSecret(spec).getEncoded();
```

The third parameter (65536) is a strength parameter. This parameter indicates this algorithm's time (number of iterations), which increases the time needed to generate the hash.

Java natively supports PBKDF2 hash algorithms but does not support BCrypt and SCrypt. Support for BCrypt and SCrypt algorithms has yet to ship with Java, although some Java libraries, such as Spring Security, support them. The Spring Security library provides support for all recommended algorithms via the *PasswordEncoder* interface: *Pbkdf2PasswordEncoder* (for PBKDF2), *BCryptPasswordEncoder* (for BCrypt), and *SCryptPasswordEncoder* (for SCrypt).

All password encoders for PBKDF2, BCrypt, and SCrypt support configuring the desired password hash strength. We can use these encoders directly even without a Spring Security-based application. If we protect our site with Spring Security, we can configure the desired password encoder through its DSL or dependency injection. These encryption algorithms generate the salt internally; the algorithm stores the salt in the output hash for later use in password validation.

Conclusions

Digital evolution and increasingly sophisticated threats drive constant changes in security architecture, and Java's security features play an essential role. Taking full advantage of these security features enables all architecture components to protect data and IT systems.

The new security architecture model is a pragmatic and comprehensive platform to protect IT systems and their users against external and internal threats. We can create secure applications that meet high-security requirements by understanding the sandbox model and using JAAS, JCA, and JSSE systems. JSSE is a good framework for securing communications in Java applications using the well-known SSL and TLS protocols. This platform facilitates the process of implementing secure sockets and is a key tool for ensuring the confidentiality and integrity of data in network communications.

The high-level security architecture in Java relies primarily on implementing cryptographic services. Moreover, using standard patterns allows for extensible and pluggable security measures in various Java security areas. These areas should be carefully examined to improve Java's security architecture. The architecture must be constantly updated in the face of emerging security vulnerabilities.

References

- Baeldung, (2024), Guide to the Cipher Class, https://www.baeldung.com/java-cipher-class, [Last updated: 2024-01-16].
- Bruno E., J. (2021). ‘Binary bit manipulation and CAN bus hardware interfaces in Java’. *Java Magazine*, December 2, 2021.
- Chandrakant K. (2019). ‘The Basics of Java Security’. https://www.baeldung.com/java-security-overview, [Last updated: 2019-09-06].
- Evans B. (2020). ‘The Unsafe Class: Unsafe at Any Speed’. *Java Magazine*, May 4, 2020.
- Kiwy F. (2021). ‘Exploring Joshua Bloch’s Builder design pattern in Java’. *Java Magazine*, May 28, 2021.
- Krawiec J., Górny P., Kiedrowicz M., Gepner P., Wybraniak-Kujawa M. (2022). ‘Security mechanisms for applications developed in Java’. Proceedings of the 39th International Business Information Management Association (IBIMA), 30-31 May 2022, Granada, Spain. Theory and Practice in Modern Computing: Vision 2025 in the Era of Pandemic / Soliman Khalid S. (red.), Proceedings of the International Business Information Management Association (IBIMA), 2022*, IBIMA Publishing*, ISBN 978-0-9998551-9-5.
- Krawiec J., Górny P., Kiedrowicz M. (2023). ‘Concurrent Programs, Finalizers and Cleaners in Java-Security Problems’. Proceedings of the 41st International Business Information Management Association Conference (IBIMA) 26-27 June 2023, Granada, Spain. IBIMA Conference on Artificial intelligence and Machine Learning, Springer Nature Switzerland 2023, Proceedings of the International Business Information Management Association (IBIMA), 2023, IBIMA Publishing, ISBN 978-0-9998551-9-5.
- Krawiec J., Górny P., Kiedrowicz M., Gepner P., Moroz L. (2024). ‘Access control mechanisms in Java applications’. Proceedings of the 43th International Business Information Management Association Conference (IBIMA) 26-27 June 2024, Madrid, Spain. IBIMA Publishing, ISBN 978-0-9998551-9-5.
- Millington S. (2024), ‘Hashing a Password in Java’, https://www.baeldung.com/java-password-hashing, [Last updated: 2024-01-08].
- Obregon A. (2023), ‘Understanding Java Security Architecture’, https://medium.com/@AlexanderObregon/understanding-javas-security-architecture-c5fa0925d318, [Last updated: 2023-11-04]
- Parkinson P. (2021). ‘Fast, flexible data access in Java using the Helidon microservices platform’. *Java Magazine*, February 12, 2021.
- Schildt H. (2021). ‘Java: The Complete Reference, Twelth Edition’. *McGraw-Hill Education*, 2021.
- ‘How to Launch The Application Blocked by Java Security’.
- https://www.istartips.com/application-blocked-by-java-security.html, [Last updated: 2022-04-03].
- ‘Java SE Platform Security Architecture’. *Security Developer’s Guide*,
- https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html,
- [Last updated: 2020-08- 04].
- ‘Java Security Overview’. *Security Developer’s Guide*,
- https://docs.oracle.com/en/java/javase/11/security/java-security-overview1.html, [Last updated: 2020-08-04].
- ‘The Security Manager’. *The Java™ Tutorials*, ORACLE Java Documentation,
- https://docs.oracle.com/javase/tutorial/essential/environment/security.html, [Last updated: 2022-03-04].
- ‘Troubleshooting Security’. *Security Developer’s Guide*,
- https://docs.oracle.com/en/java/javase/14/security/troubleshooting-security.html, [Last updated: 2020-08-04].