Sécurité de la machine virtuelle Java

JVM

Le terme « JVM » signifie Java Virtual Machine, il désigne la couche logicielle chargée de faire tourner les fichiers java compilés. Car oui, Java est un langage compilé, contrairement par exemple à Javascript (JS) qui lui est un langage interprété, çàd que le code source est directement lu par un programme tiers, à la volée. Dans le cas de Java, les fichiers sources sont compilés, non pas en binaire, mais dans un format propriétaire qui ne peut être interprété que par la JVM.

Introduction à la sécurité des applications Java

La sécurité des applications Java couvre l'authentification, l'autorisation, la confidentialité, l'intégrité et la disponibilité. Ce module présente les concepts clés et les bonnes pratiques pour développer et déployer des applications Java sécurisées.

- Principes fondamentaux : CIA (Confidentialité, Intégrité, Disponibilité)
- Authentification vs Autorisation
- Sécurité du cycle de vie (build, déploiement, exploitation)
- Gestion des dépendances et vulnérabilités

Maven, gestion des dépendances et logging sécurisé

Utiliser Maven permet de centraliser la gestion des dépendances et d'appliquer des contrôles de sécurité (scans CVE). Le logging structuré et l'audit sont essentiels pour la traçabilité des événements de sécurité.

Voir exercice: xml

Exercice : utiliser

```
dependency-check
```

pour analyser un projet Maven.

```
@echo off
REM TP Maven Dependency Check
echo Comment utiliser Maven pour vérifier les vulnérabilités dans les dépendances.
echo.

if exist pom.xml (
  echo Exécution de dependency-check...
  mvn org.owasp:dependency-check-maven:check
  echo.
  echo Rapport généré dans : target\dependency-check-report.html
) else (
  echo ERREUR : Fichier pom.xml non trouvé.
  echo Assurez-vous d'exécuter cet exercice dans un répertoire Maven.
)

pause
```

Chargement des classes et concept de 'bac à sable' (sandboxing)

Le **chargement des classes** en Java se fait au moment de l'exécution, via un processus appelé **class loading**. Ce processus est géré par le **ClassLoader**, qui est responsable de charger les fichiers '.class' contenant le bytecode des classes Java dans la machine virtuelle Java (JVM). La JVM utilise plusieurs types de **class loaders**, tels que :

- **Bootstrap ClassLoader** : Charge des classes de base du système Java (par exemple, les classes dans **java.lang.***).
- **Extension ClassLoader** : Charge des classes dans le répertoire des extensions ('jre/lib/ext').
- **System ClassLoader** : Charge des classes du classpath (les classes de votre application, y compris les bibliothèques externes)

Le mécanisme de chargement de classes permet à Java d'être très flexible, en chargeant les classes de manière dynamique lorsqu'elles sont nécessaires.

Le concept de **bac à sable** en Java fait référence à un environnement sécurisé où les programmes (souvent des applets ou des applications téléchargées) sont exécutés de manière à limiter leurs capacités et ainsi protéger le système hôte. Ce concept vise à empêcher une application d'accéder directement à des ressources sensibles ou d'effectuer des actions malveillantes. Le **bac à sable** restreint l'accès aux ressources système telles que :

- Les fichiers du système de fichiers
- Le réseau (accès au réseau externe)
- L'exécution de processus externes

Les applets Java et les applications exécutées avec une sécurité renforcée fonctionnent souvent dans ce mode. Le **SecurityManager** et d'autres mécanismes assurent que les actions du programme sont contrôlées et qu'il ne peut pas compromettre la sécurité du système.

Chargement des classes et concept de 'bac à sable' (sandboxing)

Le **SecurityManager** en Java est une classe qui contrôle l'accès aux ressources système. Lorsqu'une application tente d'effectuer une opération (par exemple, accéder à un fichier ou se connecter à un réseau), le **SecurityManager** vérifie si cette opération est autorisée en fonction des règles de sécurité définies dans la politique de sécurité.

Le **SecurityManager** fonctionne en interceptant les appels de méthodes sensibles, comme celles qui accèdent au système de fichiers ou au réseau, et en contrôlant si l'action peut être effectuée en fonction des permissions configurées. Si une action est interdite par les permissions, une **SecurityException** est levée.

L'**AccessController** est une classe complémentaire au **SecurityManager**. Elle fournit une API pour effectuer des contrôles de sécurité sur des actions spécifiques de manière plus fine, en vérifiant si un code a l'autorisation d'effectuer une certaine action.

Il est généralement utilisé pour appliquer une logique de sécurité plus précise dans un environnement plus contrôlé (par exemple, une application Java Enterprise). Il permet de vérifier les permissions sur des actions spécifiques, telles que l'accès à un fichier ou l'exécution d'une méthode.

Les **permissions** en Java sont des objets qui définissent ce qu'un code est autorisé à faire. Ces permissions sont définies dans des fichiers appelés **fichiers de politique** ('.policy'). Un fichier de politique contient des règles sous forme de déclarations qui attribuent des permissions spécifiques à des parties de code, souvent en fonction de leur provenance ou de leur identité.

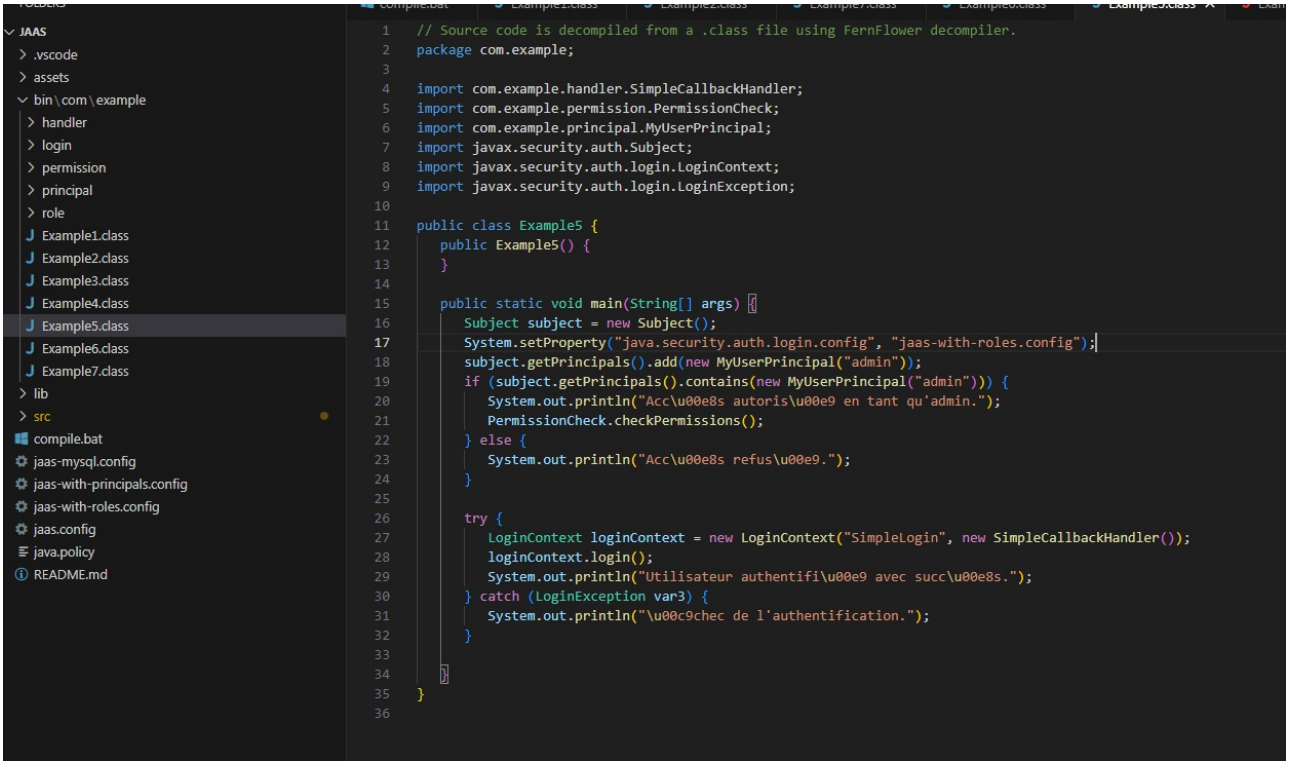
Les fichiers .policy permettent de spécifier quelles ressources et actions sont autorisées pour différents types de code. Par exemple, du code provenant d'une source de confiance peut avoir plus de permissions qu'un code téléchargé depuis Internet.

En Java moderne, l'utilisation du **SecurityManager** est de plus en plus dépréciée au profit de mécanismes de sécurité plus robustes et flexibles, tels que les modules Java (JPMS) et les frameworks de sécurité externes. Cependant, comprendre son fonctionnement reste important pour maintenir et sécuriser les applications Java existantes.

```java
package com.example;

import java.io.*;

public class Main {
    public static void main(String[] args) {
        File file = new File("assets/sample.txt");

        try{
            if (file.exists()) {
                System.out.println("Fichier existe.");
                System.out.println("Peut lire : " + file.canRead());
                System.out.println("Peut écrire : " + file.canWrite());
                System.out.println("Peut exécuter : " + file.canExecute());
            } else {
                System.out.println("Le fichier n'existe pas.");
            }
        } catch (Exception e){
            e.printStackTrace();
        }

    }
}


grant {
// write permission granted
    permission java.io.FilePermission "assets/sample2.txt", "read, write";
    permission java.io.FilePermission "<<ALL FILES>>", "read, write, execute";
};




: samples
javac -d ./bin -cp ./src ./src/com/example/Main.java

: run
java -cp "bin" -Djava.security.manager ^ -Djava.security.policy=java.policy
com.example.Main
: java -cp "bin" -Djava.security.policy=java.policy com.example.Main
```

Mécanismes de protection de l'intégrité du bytecode, décompilation et obfuscation du code

Le **bytecode** est un code intermédiaire compilé à partir du code source Java. Il est contenu dans des fichiers '.class' et exécuté par la machine virtuelle Java (JVM). C'est un format binaire optimisé pour une exécution portable sur diverses plateformes. Le bytecode peut être facilement décompilé, exposant ainsi :

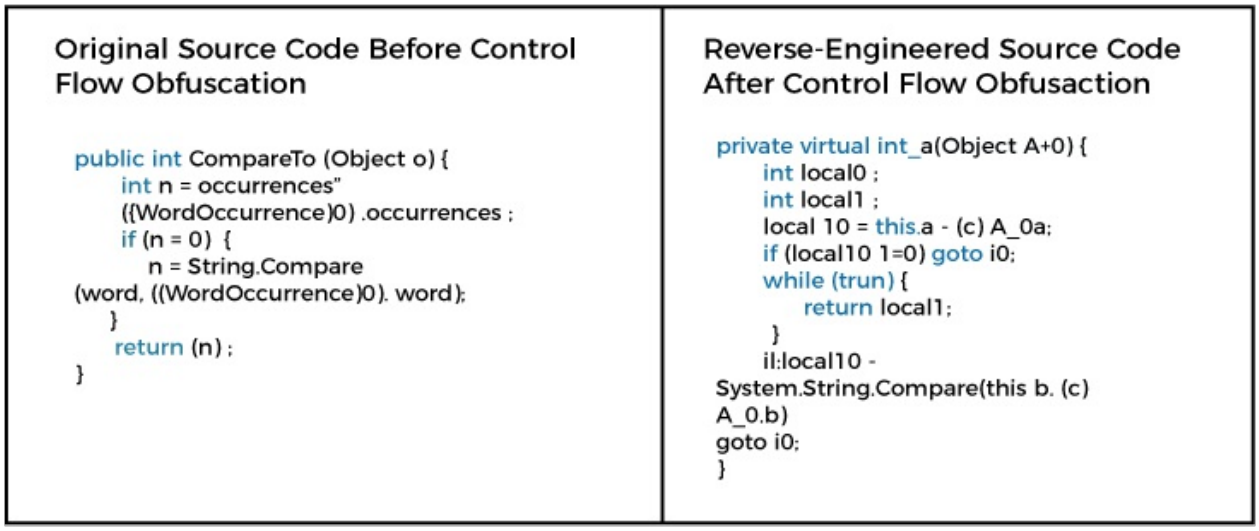


- **Propriété intellectuelle** : Les algorithmes et logiques métiers peuvent être volés.
- **Modifications malveillantes** : Un attaquant pourrait insérer des failles de sécurité ou des portes dérobées.
- **Atteinte à la confiance** : Les applications modifiées pourraient se comporter de manière imprévisible.
- **Signature numérique** : Le bytecode peut être signé numériquement pour garantir qu'il n'a pas été modifié et qu'il provient d'une source fiable.
- **Validation à l'exécution** : Des mécanismes peuvent être utilisés pour valider l'intégrité du bytecode durant l'exécution, détectant toute altération.
- **Chiffrement** : Certaines applications chiffrent le bytecode, ne le déchiffrant qu'au moment de son exécution.

Mécanismes de protection de l'intégrité du bytecode, décompilation et obfuscation du code

L'**obfuscation** est une technique visant à rendre le bytecode difficile à comprendre, tout en conservant son comportement fonctionnel. L'objectif est de compliquer la tâche des attaquants qui tentent de décompiler ou d'analyser le code. Voici quelques techniques d'obfuscation:

- **Renommage des éléments** : Les noms de classes, méthodes et variables sont remplacés par des chaînes de caractères peu significatives (ex. : 'a', 'b', 'c').
- **Suppression des informations inutiles** : Les commentaires, les noms de variables et les informations de débogage sont supprimés, ce qui rend l'analyse plus difficile.
- **Encodage des chaînes de caractères** : Les chaînes de caractères sensibles peuvent être encodées ou cryptées pour empêcher une lecture directe dans le bytecode
- **Transformation du code** : Ajout de code inutile ou modification des structures de contrôle pour rendre le bytecode moins compréhensible.



Mécanismes de protection de l'intégrité du bytecode, décompilation et obfuscation du code

Avantages de l'obfuscation:

- **Protection contre la décompilation** : L'obfuscation rend le code beaucoup plus difficile à comprendre, protégeant ainsi la propriété intellectuelle.
- **Rend l'analyse inverse plus complexe** : L'ajout de transformations complexes rend le reverse engineering plus difficile.
- **Chiffrement** : Certaines applications chiffrent le bytecode, ne le déchiffrant qu'au moment de son exécution.

Limites:

- **Performance** : Bien que l'impact soit souvent minime, l'obfuscation peut parfois altérer légèrement les performances du bytecode
- **Rend l'analyse inverse plus complexe** : L'ajout de transformations complexes rend le reverse engineering plus difficile.
- **Non infaillible** : Des attaquants expérimentés peuvent contourner certaines protections d'obfuscation et réussir à décompiler le code.

Outils d'obfuscation en Java:

- **ProGuard** : Un obfuscateur populaire pour Java, utilisé pour protéger les applications Android et Java.
- **Zelix KlassMaster** : Un outil offrant une protection avancée contre la décompilation.
- **Allatori** : Un obfuscateur commercial qui fournit des fonctionnalités avancées pour protéger les applications.

Conclusion:

- **Protéger l'intégrité du bytecode** est essentiel pour prévenir les attaques et les vols de propriété intellectuelle.
- **La décompilation** permet à un attaquant de récupérer le code source, ce qui expose des informations sensibles.
- **L'obfuscation** est une méthode efficace pour compliquer l'analyse du bytecode, mais elle ne rend pas le code complètement inviolable.
- Pour une sécurité renforcée, il est souvent nécessaire de combiner plusieurs techniques, telles que la signature numérique, l'obfuscation et la validation à l'exécution, afin de rendre le reverse engineering aussi difficile que possible.

Présentation des concepts liés à la sécurité

Identification et méthodes d'authentification

L'identification consiste à reconnaître un utilisateur ou un système à l'aide d'un identifiant unique. **L'authentification**, quant à elle, permet de vérifier l'identité déclarée à l'aide de différentes méthodes :

- **Mot de passe :** Méthode classique mais vulnérable aux attaques par force brute ou hameçonnage.
- **Authentification multi-facteurs (MFA) :** Combine plusieurs éléments (ex. mot de passe + code SMS) pour renforcer la sécurité.
- **Certificats numériques :** Utilisés notamment dans les connexions sécurisées (HTTPS).
- **Biométrie :** Empreintes digitales, reconnaissance faciale ou rétinienne.
- **Tokens physiques ou logiciels :** Clés USB sécurisées, applications d'authentification (Google Authenticator, Authy, etc.).

La plupart de ces méthodes sont complémentaires et peuvent être déployées en même temps afin de protéger un seul et même système. Notez bien qu'en terme de sécurité informatique (et en terme de sécurité en général), **la solidité d'un système repose sur la solidité de son maillon le plus faible**.

Ainsi, **cela ne sert à rien d'avoir un système techniquement impénétrable si vous laissez traîner vos mots de passes** sur la table du salon. Plus que jamais, **la formation et la sensibilisation des humains est primordiale** en matière de sécurité.

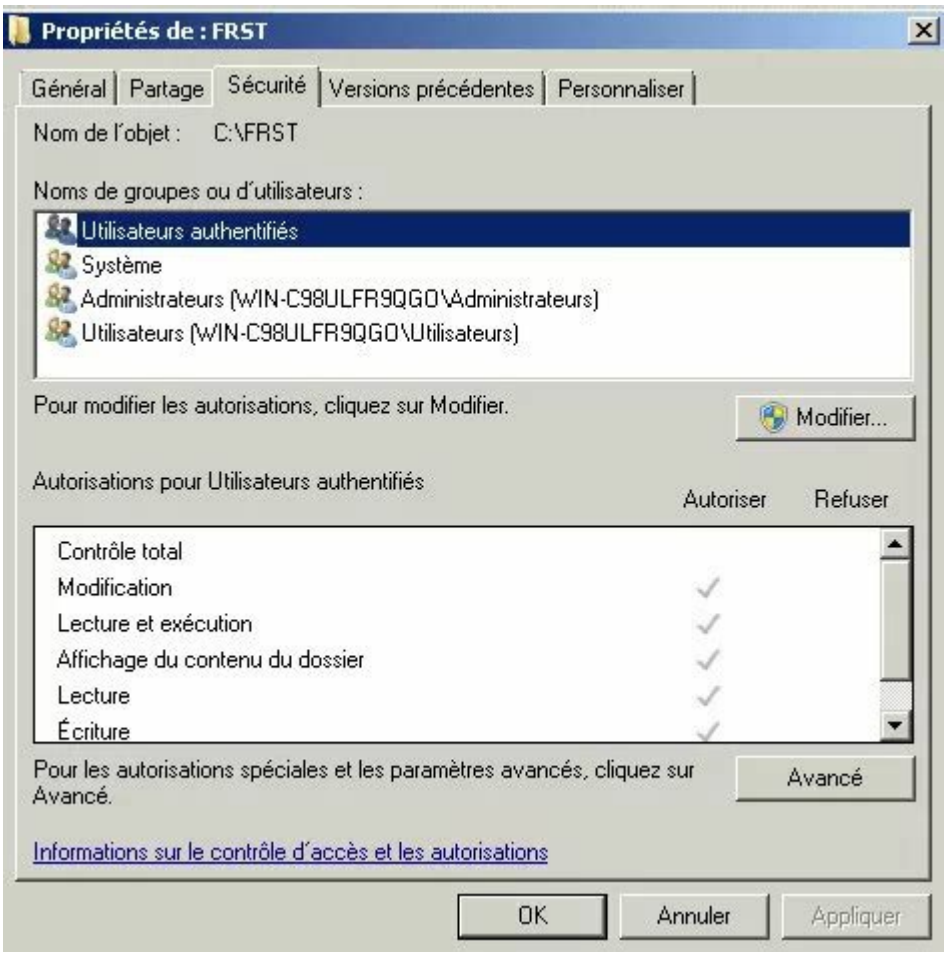
```java
package com.example;
import java.util.Scanner;

public class Example1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String secretPwd = "secret123";

        System.out.print("Entrez votre mot de passe: ");
        String userInput = scanner.nextLine();

        if (userInput.equals(secretPwd)) {
            System.out.println("Authentification réussie !");
        } else {
            System.out.println("Mot de passe incorrect.");
        }

        scanner.close();
    }
}
```

Autorisations et permissions

Une fois authentifié, un utilisateur doit être autorisé à accéder aux ressources. Cette gestion repose sur plusieurs modèles :

- **ACL (Access Control List)** : Liste d'autorisations associée à une ressource. Par exemple, un fichier peut n'être disponible qu'en **lecture**
- **RBAC (Role-Based Access Control)** : Permissions attribuées selon le rôle de l'utilisateur. Par exemple, les membres du groupe 'admin' peuvent être autorisés à consulter un fichier en **lecture** mais les autres ne le pourront pas.
- **ABAC (Attribute-Based Access Control)** : Permissions basées sur des attributs (ex. localisation, type d'appareil, heure de connexion). Par exemple, un utilisateur peut se voir autorisé à consulter une URL si son IP publique est localisée comme provenant du même pays.
- **Principes de moindre privilège** : Limiter les droits aux strictes nécessités. Par exemple, un utilisateur peut se voir octroyer un accès **en lecture à un fichier si et uniquement si il en a la nécéssité et pendant un temps donné**.

Comme pour les méthodes d'authentification, ces permissions & autorisations peuvent être cumulatives, cela dépend de la politique de l'entreprise, des besoins etc ...

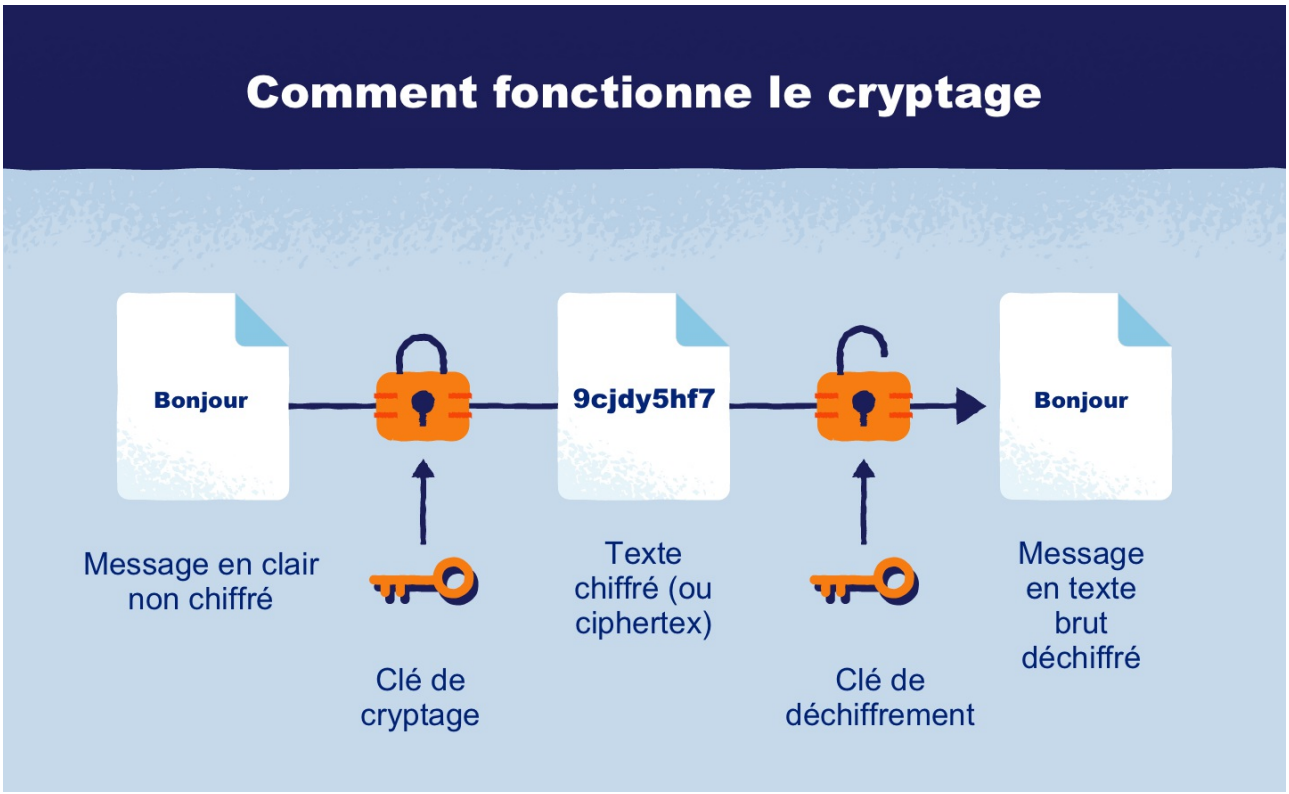


Confidentialité, non-répudiation et cryptage

Une fois authentifié, les utilisateurs se voient accorder des permissions & autorisations, mais encore faut-il s'assurer de la protection et de l'authenticité des informations échangées. Pour cela il existe quelques concepts:

- **Confidentialité** : Protection des données contre tout accès non autorisé (ex. chiffrement des e-mails, VPN).
- **Non-répudiation** : Garantit qu'un utilisateur ne peut nier avoir réalisé une action (ex. signature numérique, enregistrement de log dans une bdd, maintient d'un journal de connexions etc ..)
- **Cryptage**: Transformation des données pour les rendre inaccessibles sans clé de déchiffrement. **Symétrique** : Une seule clé pour chiffrer et déchiffrer (ex. AES). **Asymétrique** : Une clé publique pour chiffrer, une clé privée pour déchiffrer (ex. RSA)
- **Autorités de certification (CA)** : Organismes délivrant des certificats numériques garantissant l'authenticité d'un site ou d'un utilisateur.

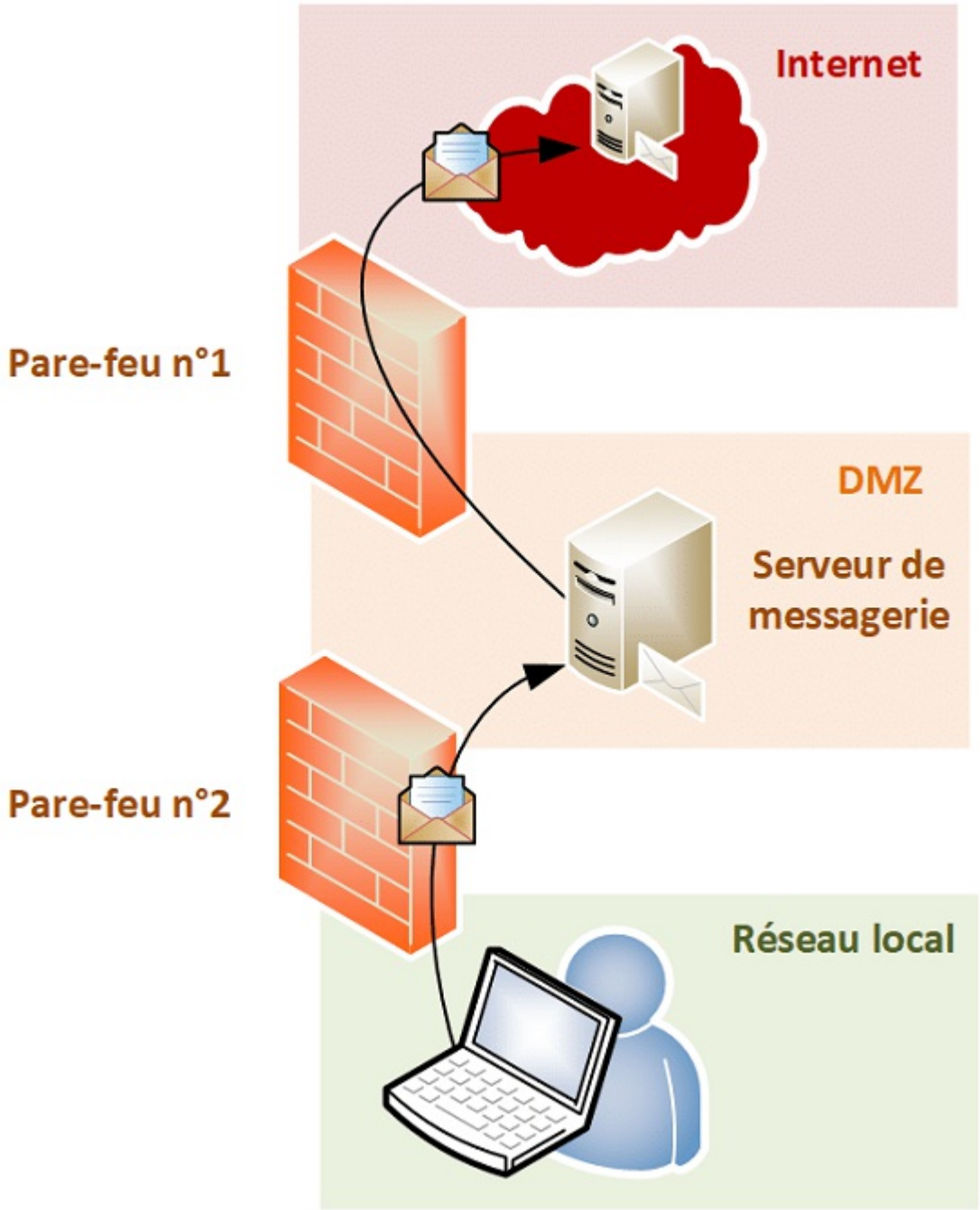
Comme pour le reste, rien n'empêche de cumuler ces différents concepts. Il est par exemple possible de ne donner accès à une ressource qu'à travers un VPN tout en maintenant un journal de connexion et des activités de l'utilisateur (c'est même intégré à la plupart des VPN).



Pare-feu, DMZ et rupture de protocole

Pour protéger les réseaux et limiter les attaques, plusieurs dispositifs sont utilisés :

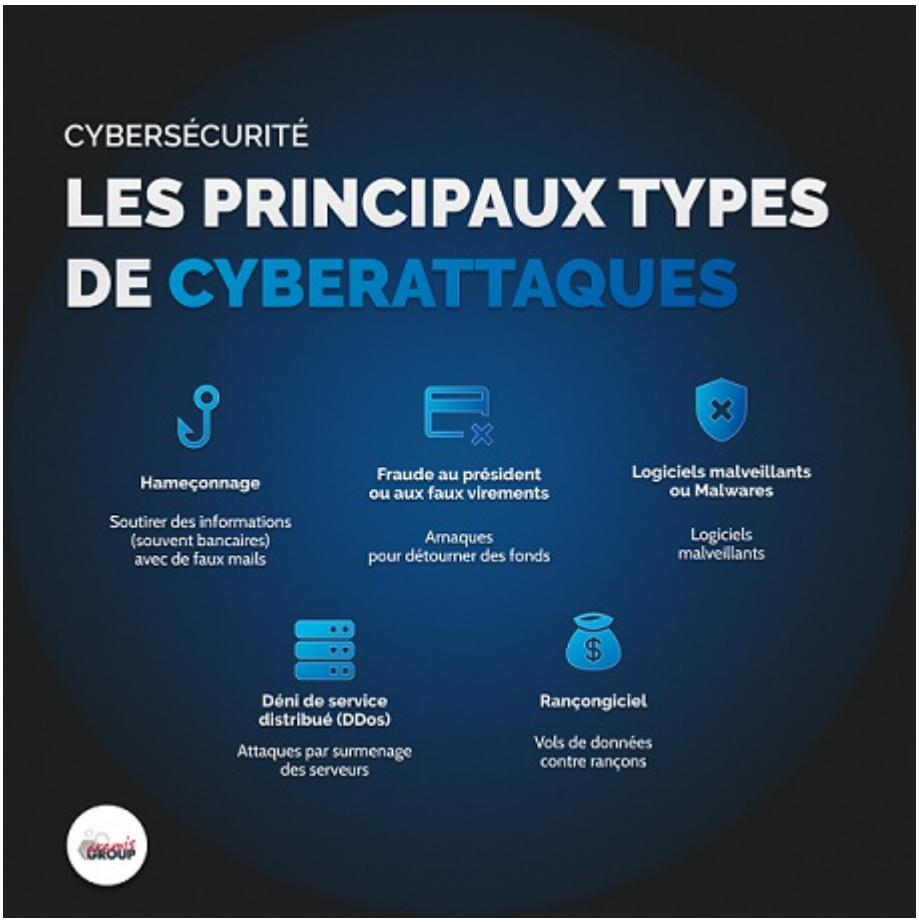
- **Pare-feu (firewall)**: Filtre le trafic réseau entrant et sortant selon des règles définies.
- **DMZ (Demilitarized Zone)**: Zone réseau isolée où sont placés les serveurs accessibles depuis l'extérieur pour limiter l'exposition du réseau interne.
- **Rupture de protocole** : Technique consistant à interdire un flux direct entre deux réseaux en utilisant un proxy ou une passerelle d'application.



Les types d'attaques

Les systèmes informatiques sont exposés à divers types d'attaques :

- **Attaques par force brute** : Essai de toutes les combinaisons possibles pour casser un mot de passe.
- **Attaques par dictionnaire** : Essai des mots de passes les **plus probables**, tirés depuis un **dictionnaire** des mots de passes les plus utilisés ces derniès années.
- **Hameçonnage (phishing)** : Tromper un utilisateur pour obtenir ses informations sensibles.
- **Attaques par déni de service (DDoS)** : Saturation d'un serveur par un trafic massif pour le rendre indisponible.
- **Injection SQL** : Exploitation d'une faille pour manipuler une base de données.
- **Malwares** : Logiciels malveillants (virus, ransomwares, chevaux de Troie, etc.).
- **Man-in-the-middle (MITM)** : Interception et altération des communications entre deux parties.



Exemple d'attaque par dictionnaire

L'attaque par dictionnaire repose sur la prédictibilité d'un mot de passe. Les humains ont tendance à utiliser des mots de passe dont ils peuvent se souvenir (exemple, une date de naissance associée à un nom). Ce genre de mot de passe, en plus d'être trop court et généralement peu varié au niveau de la casse (majuscule, minuscule), a tendance à être trop vite récupéré (par social engineering) ou à être trop vite hacké par force brute à l'aide d'un dictionnaire.

En effet, il existe des mots de passe plus fréquents que d'autre et plusieurs personnes maintiennent des listes répertoriant les mots de passes les plus courants. Voici un petit exemple:

```
import java.util.Arrays;
import java.util.List;

public class DictionaryAttack {
    public static void main(String[] args) {
        String correctPassword = "secure123"; // Mot de passe réel
        List<String> dictionary = Arrays.asList("123456", "password", "secure123", "admin");

        for (String attempt : dictionary) {
            if (attempt.equals(correctPassword)) {
                System.out.println("Mot de passe trouvé : " + attempt);
                break;
            }
        }
    }
}
```

Il existe plusieurs solutions pour se prémunir d'une attaque par dictionnaire:

- L'utilisation d'un logiciel de type **KeyPass**, qui vous aide à générer des mots de passes compliqués et qui conserve ces mots de passes en les protégeant en local à l'aide d'un mot de passe. Le seul inconvénient de cette solution est qu'il suffira à un éventuel indiscret de deviner votre mot de passe pour avoir accès à tous les autres (**verrouillez votre ordinateur avant de quitter votre poste**)
- **Utiliser des mots de passes différents à chaque fois**, suffisamment **longs et imprévisibles** (caractères spéciaux, chiffres, lettres etc ...) et les **retenir** (oui c'est difficile).
- **Conserver ses mots de passes sur un support physique** (type clé USB chiffrée)
- **Toujours activer la double authentification.**

Conclusion

La sécurité informatique repose sur une combinaison de bonnes pratiques, de technologies adaptées et de sensibilisation des utilisateurs. Une approche proactive et une surveillance continue sont essentielles pour limiter les risques et protéger les systèmes contre les menaces en constante évolution.

Mise en place d'un système de logging sécurisé avec SLF4J

** SLF4J (Simple Logging Facade for Java) ** est une bibliothèque qui fournit une interface de logging abstraite pour les applications Java. Elle permet aux développeurs de choisir parmi plusieurs frameworks de logging sous-jacents (comme Logback, Log4j, etc.) sans modifier le code de l'application.

Pour mettre en place un système de logging sécurisé avec SLF4J, voici les étapes clés à suivre :

- **Intégration de SLF4J :** Ajoutez les dépendances SLF4J et le framework de logging choisi (par exemple, Logback) à votre projet Maven ou Gradle.
- **Configuration du framework de logging :** Configurez le framework de logging pour définir les niveaux de log, les formats de message et les destinations (fichiers, consoles, etc.). Assurez-vous que les fichiers de configuration sont sécurisés et accessibles uniquement aux utilisateurs autorisés.
- **Utilisation de SLF4J dans le code :** Utilisez l'API SLF4J pour enregistrer les messages de log dans votre application. Par exemple, utilisez des niveaux de log appropriés (DEBUG, INFO, WARN, ERROR) pour catégoriser les messages.
- **Sécurisation des messages de log :** Évitez d'enregistrer des informations sensibles (comme des mots de passe ou des données personnelles) dans les logs. Utilisez des techniques de masquage ou de chiffrement si nécessaire.
- **Gestion des accès aux logs :** Mettez en place des contrôles d'accès pour restreindre l'accès aux fichiers de log aux utilisateurs autorisés uniquement.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="
           http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>secure-logging-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>15</maven.compiler.source>
        <maven.compiler.target>15</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- SLF4J API -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>2.0.13</version>
        </dependency>

        <!-- Logback (implémentation) -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.5.6</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.3.0</version>
                <configuration>
                    <mainClass>com.example.SecureLoggingExample</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

```
mvn clean package
pause
mvn exec:java -Dexec.mainClass="com.example.SecureLoggingExample"


<configuration>

    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/app.log</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/app.%d{yyyy-MM-dd}.log</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>

        <encoder>
            <pattern>%d{HH:mm:ss} %-5level %logger - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="FILE"/>
    </root>

</configuration>


package com.example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SecureLoggingExample {

    private static final Logger logger =
LoggerFactory.getLogger(SecureLoggingExample.class);

    public static void main(String[] args) {

        String username = "admin";
        String action = "login";

        logger.info("Application started");
        logger.info("User {} action {}", username, action);
    }
}
```

Gestion de l'authentification

Panorama des mécanismes d'authentification

Les mécanismes modernes d'authentification couvrent OAuth2, OpenID Connect, JWT, MFA, et les serveurs d'identité (Keycloak, Auth0). Nous comparons leurs usages et choix selon les besoins (API, web, mobile).

- OAuth2 : délégation d'accès pour les APIs
- OpenID Connect : surcouche identité pour OAuth2
- JWT : tokens stateless pour authentification et claims
- MFA : renforcement par second facteur
- Serveurs d'identité : Keycloak, gestion centralisée des utilisateurs

OAuth2 et JWT - Principes et implémentation

OAuth2 est un standard d'autorisation permettant à une application cliente d'accéder à des ressources protégées sans exposer les identifiants de l'utilisateur. Il repose sur des flux bien définis (Authorization Code, Client Credentials, etc.) et introduit une séparation claire entre le client, le serveur d'autorisation et le serveur de ressources. OAuth2 ne définit pas le format du token, mais encadre strictement la manière dont il est délivré, utilisé et renouvelé.

JWT (JSON Web Token) est un format de jeton auto-contenu, signé (et éventuellement chiffré), utilisé pour transporter des informations de manière sécurisée. Un JWT contient des claims (identité, rôles, expiration, audience) et permet une authentification stateless. Il est fréquemment utilisé comme token d'accès dans OAuth2, mais peut aussi être employé indépendamment, avec des bonnes pratiques essentielles telles que la rotation des clés, la gestion de l'expiration et la révocation.

```
: samples
javac src/main/java/com/example/*.java -d bin

: run
java -cp bin com.example.JwtClientExample


package com.example;

public class JwtClientExample {

    public static void main(String[] args) throws Exception {

        JwtServer server = new JwtServer();

        System.out.println("Client ask for jwt token...");
        String token = server.generateToken("alice");

        System.out.println("received token :");
        System.out.println(token);

        System.out.println("Access to protected ressource...");

        if (server.validateToken(token)) {
            System.out.println("Acces granted (valid token)");
        } else {
            System.out.println("Acces refused (invalid token)");
        }
    }
}
```

```java
package com.example;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

// RAPPEL du principe de JWT (JSON Web Token)
// Un JWT est composé de 3 parties encodées en Base64URL et séparées par des points
:
// 1. Header : contient le type de token et l'algorithme de signature
// 2. Payload : contient les claims (informations sur l'utilisateur)
// 3. Signature : permet de vérifier l'intégrité du token

// HEADER.PAYLOAD.SIGNATURE


public class JwtServer {

    // secret key
    private static final String SECRET = "mySuperSecretKey";

    // JWT generation
    public String generateToken(String username) throws Exception {

        String headerJson = "{\"alg\":\"HS256\",\"typ\":\"JWT\"}";
        String payloadJson = "{\"sub\":\"" + username + "\"}";

        String header = base64UrlEncode(headerJson);
        String payload = base64UrlEncode(payloadJson);

        String signature = sign(header + "." + payload);

        return header + "." + payload + "." + signature;
    }

    // JWT validation
    public boolean validateToken(String token) throws Exception {

        String[] parts = token.split("\\.");
        if (parts.length != 3) {
            return false;
        }

        String data = parts[0] + "." + parts[1];
        String expectedSignature = sign(data);

        return expectedSignature.equals(parts[2]);
    }
```

```
    // --- utilitaries ---

    private String sign(String data) throws Exception {
        Mac mac = Mac.getInstance("HmacSHA256");
        SecretKeySpec key = new
SecretKeySpec(SECRET.getBytes(StandardCharsets.UTF_8), "HmacSHA256");
        mac.init(key);

        byte[] raw = mac.doFinal(data.getBytes(StandardCharsets.UTF_8));
        return Base64.getUrlEncoder().withoutPadding().encodeToString(raw);
    }

    private String base64UrlEncode(String value) {
        return Base64.getUrlEncoder()
                .withoutPadding()
                .encodeToString(value.getBytes(StandardCharsets.UTF_8));
    }
}
```

MFA et Keycloak - intégration pratique

MFA (Multi-Factor Authentication) ajoute une couche de sécurité supplémentaire en exigeant plusieurs formes de vérification avant d'accorder l'accès. Keycloak, un serveur d'identité open-source, prend en charge divers mécanismes MFA, y compris TOTP (Time-based One-Time Password).

L'un des serveurs d'identié les plus connus est Keycloak, un projet open-source développé par Red Hat. Keycloak offre une gestion complète des utilisateurs, des rôles et des permissions, ainsi que des fonctionnalités avancées telles que l'authentification multi-facteurs (MFA), l'intégration avec des annuaires externes (LDAP, Active Directory), et la prise en charge de protocoles d'authentification modernes comme OAuth2 et OpenID Connect.

```
: samples
javac src/main/java/com/example/*.java -d bin

: run
java -cp bin com.example.MfaClient
```

```java
package com.example;

import java.util.Scanner;

/**
 * Client MFA (simulation utilisateur)
 */
public class MfaClient {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        MfaServer server = new MfaServer();

        System.out.print("Username : ");
        String username = scanner.nextLine();

        System.out.print("Password : ");
        String password = scanner.nextLine();

        // Facteur 1
        if (!server.authenticate(username, password)) {
            System.out.println("Auth failed");
            return;
        }

        System.out.println("Valid password");

        // Facteur 2
        server.generateOtp(username);

        System.out.print("Enter the OTP Code : ");
        String otp = scanner.nextLine();

        if (server.verifyOtp(username, otp)) {
            System.out.println("Access granted (MFA validated)");
        } else {
            System.out.println("Invalid OTP code");
        }

        scanner.close();
    }
}


package com.example;

import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map;
```

```java
/**
 * Serveur MFA simplifié
 */
public class MfaServer {

    private static final SecureRandom random = new SecureRandom();

    // utilisateurs "en base"
    private final Map<String, String> users = new HashMap<>();

    // stockage temporaire des OTP
    private final Map<String, String> otpStore = new HashMap<>();

    public MfaServer() {
        // username -> password
        users.put("admin", "admin123");
    }

    /**
     * Étape 1 : authentification classique
     */
    public boolean authenticate(String username, String password) {
        return users.containsKey(username)
                && users.get(username).equals(password);
    }

    /**
     * Étape 2 : génération du code OTP
     */
    public String generateOtp(String username) {
        String otp = random.nextInt(1000000) + "";
        otpStore.put(username, otp);

        // simulation d'envoi (SMS / email)
        System.out.println("[SERVEUR] OTP sent : " + otp);

        return otp;
    }

    /**
     * Étape 3 : vérification du code OTP
     */
    public boolean verifyOtp(String username, String otp) {
        String expected = otpStore.get(username);
        return expected != null && expected.equals(otp);
    }
}
```

Conclusion - Choisir la bonne stratégie d'authentification

Récapitulatif des critères de choix : niveau de sécurité requis, compatibilité API/web, scalabilité, gestion des sessions, exigences de conformité.

- Pour APIs : OAuth2 + JWT (stateless)
- Pour applications web : OpenID Connect + sessions sécurisées ou JWT selon contexte
- Toujours activer MFA pour accès sensibles
- Utiliser un serveur d'identité pour centraliser la gestion

Sécurisation des données

Exemple d'attaque par injection SQL

**Un site web dispose d'un formulaire de connexion classique où l'utilisateur entre son nom d'utilisateur et son mot de passe. En arrière-plan, le site exécute une requête SQL comme celle-ci **

```
SELECT * FROM utilisateurs WHERE username = 'admin' AND password =
'password123';
```

**Un attaquant peut tenter une injection SQL en entrant ' OR '1'='1 dans le champ du mot de passe. Cela modifie la requête de la manière suivante **

La condition '1'='1' étant toujours vraie, l'attaquant obtient un accès non autorisé au compte administrateur. **Comment s'en protéger ?**

- **Utiliser des requêtes préparées (paramétrées)** pour éviter l'interprétation directe des entrées utilisateur.
- **Limiter les permissions en base de données** pour empêcher les modifications malveillantes.
- **Valider et filtrer les entrées utilisateur** avant leur utilisation dans une requête SQL.

Dans le TP suivant, nous allons modifier notre application pour voir la différence entre une requête SQL classique et une requête paramétrée(voir le TP dbAuth sur le dépôt github).

Introduction SSL/TLS et X.509

Le protocole SSL (Secure Sockets Layer) et son successeur TLS (Transport Layer Security) sont des protocoles cryptographiques conçus pour fournir des communications sécurisées sur un réseau informatique. Ils assurent la confidentialité, l'intégrité et l'authenticité des données échangées entre les clients et les serveurs. Cette introduction couvre les concepts fondamentaux de SSL/TLS, y compris la négociation de la connexion sécurisée, l'utilisation des certificats numériques X.509, et les meilleures pratiques pour configurer SSL/TLS dans les applications Java.

Il existe plusieurs versions de SSL/TLS, avec TLS 1.2 et TLS 1.3 étant les versions les plus récentes et recommandées en raison de leurs améliorations en matière de sécurité et de performance. SSL/TLS utilise des mécanismes de chiffrement symétrique et asymétrique pour protéger les données, ainsi que des certificats numériques pour authentifier les parties impliquées dans la communication.

Dans l'exemple suivant, nous allons nous connecter à une base MySQL sécurisée via SSL/TLS en utilisant un certificat auto-signé. Nous créerons un keystore Java pour stocker le certificat et configurerons la connexion JDBC pour utiliser SSL/TLS.

```
: samples
 javac -d ./bin -cp ./src ./src/com/example/Main.java


: run
java -cp ".;lib/mysql-connector.jar;bin" com.example.Main
```

```
@REM export du certificat privé et création du keystore
keytool -genkeypair ^
  -alias mysql-server ^
  -keyalg RSA ^
  -keysize 2048 ^
  -validity 365 ^
  -keystore server-keystore.jks ^
  -storepass changeit ^
  -keypass changeit ^
  -dname "CN=localhost, OU=DEV, O=Example, L=Paris, C=FR"

@REM export du certificat public
keytool -exportcert ^
  -alias mysql-server ^
  -keystore server-keystore.jks ^
  -storepass changeit ^
  -file server-cert.crt

@REM   création du truststore et import du certificat public
keytool -importcert ^
  -alias mysql-server ^
  -file server-cert.crt ^
  -keystore client-truststore.jks ^
  -storepass changeit ^
  -noprompt
```

```java
package com.example;
import java.util.Scanner;
import com.example.login.DatabaseLogin;
import com.example.user.MyUser;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Entrez votre nom d'utilisateur: ");
        String username = scanner.nextLine();

        System.out.print("Entrez votre mot de passe: ");
        String userpwd = scanner.nextLine();

        DatabaseLogin dbLogin = new DatabaseLogin();
        MyUser user = dbLogin.login(username, userpwd);

        if( user.getIsConnected() ) {
            System.out.println("Authentification reussie pour l'utilisateur : " + user.getName());
            System.out.println("Role attribue : " + String.join(", ", user.getRole()));
        } else {
            System.out.println("Echec de l'authentification.");
        }

        scanner.close();
    }
}
```

```java
package com.example.user;

public class MyUser {
    private String name;
    private String role;
    private boolean isConnected = false;

    public MyUser(String name, String role) {
        this.name = name;
        this.role = role;
        this.isConnected = false;
    }

    public void setIsConnected(boolean isConnected) {
        this.isConnected = isConnected;
    }

    public void setRole(String role) {
        this.role = role;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public String getRole() {
        return role;
    }

    public boolean getIsConnected() {
        return isConnected;
    }
}


package com.example.login;

import java.security.MessageDigest;
import java.sql.*;
import com.example.user.MyUser;

public class DatabaseLogin {

    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/jaas_example";
    // + "?useSSL=true"
    // + "&requireSSL=true"
```

```
    // + "&verifyServerCertificate=true";

    private static final String DB_USER = "root"; // Change selon ton utilisateur DB
    private static final String DB_PASSWORD = "root"; // Change selon ton mot de passe
DB

    public MyUser login(String username, String password) {

        // System.setProperty("javax.net.ssl.trustStore", "client-truststore.jks");
        // System.setProperty("javax.net.ssl.trustStorePassword", "changeit");

        // créer un nouvel utilisateur par défaut
        MyUser user = new MyUser("", "");
        user.setIsConnected(false);

        try {
            // Enregistrer le driver JDBC pour MySQL
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Connexion à la base de données MySQL
            Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER,
DB_PASSWORD);

            // Vérifier si les informations d'identification sont correctes
            String sql = "SELECT password, username FROM users WHERE username=?
AND password=?";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, password);
                // stmt.setString(2, hashPassword(password));
                ResultSet rs = stmt.executeQuery();
                if (rs.next()) {
                    user.setName(rs.getString("username"));
                    user.setIsConnected(true);
                } else {
                    System.out.println("Invalid credentials");
                }
            }

            // Récupérer les rôles de l'utilisateur
            // ce code est naturellement protégé des injections SQL grâce aux requêtes
            // préparées.
            sql = "SELECT role_name FROM roles WHERE username=?";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                ResultSet rs = stmt.executeQuery();
                while (rs.next()) {
                    String role = rs.getString("role_name");
                    user.setRole(role);
                }
            }
```

```
            // libère la connexion à la base de données
            connection.close();

        } catch (ClassNotFoundException e) {
            System.out.println("Driver MySQL non trouvé !");
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Error during login process: " + e.getMessage());
        }
        return user;
    }

    public boolean addUser(String username, String password, String role) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER,
DB_PASSWORD);

            String hashedPassword = hashPassword(password);

            // Ajouter dans la table users
            String sql = "INSERT INTO users (username, password) VALUES (?, ?)";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, hashedPassword);
                stmt.executeUpdate();
            }

            // Ajouter le rôle
            sql = "INSERT INTO roles (username, role_name) VALUES (?, ?)";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, role);
                stmt.executeUpdate();
            }

            connection.close();
            return true;

        } catch (ClassNotFoundException e) {
            System.out.println("Driver MySQL non trouvé !");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Erreur SQL lors de l'ajout de l'utilisateur : " + e.getMessage());
        }

        return false;
    }

    // ------------------------------
```

```java
    // Méthode de hash SHA-256
    // ------------------------------
    private String hashPassword(String password) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hashedBytes = digest.digest(password.getBytes("UTF-8"));
            // conversion bytes -> hex
            StringBuilder sb = new StringBuilder();
            for (byte b : hashedBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (Exception e) {
            throw new RuntimeException("Erreur de hash du mot de passe", e);
        }
    }
}
```

Chiffrement des données dans une base de données

Les données sensibles stockées dans une base de données doivent être protégées contre les accès non autorisés. Le chiffrement des données au repos est une méthode efficace pour garantir la confidentialité des informations stockées. Voici un exemple simple de chiffrement et de déchiffrement des données avant de les insérer dans une base de données à l'aide de Java et de la bibliothèque JCE (Java Cryptography Extension).

Normalement, et dans le but d'éviter des attaques par annuaire inversén on préfixe ou on suffixe les données sensibles par une chaîne de caractères suffisament longue et compliquée, que l'on nomme **SALT**. Cette chaîne doit rester la même pour l'utilisateur.

Comme on peut le voir dans cet exemple, nous utilisons l'algorithme SHA-256 pour créer une version hachée du mdp. Cette façon de faire est aussi bien utilisée en lecture qu'en écriture.

```
package com.example.login;

import java.security.MessageDigest;
import java.sql.*;
import com.example.user.MyUser;

public class DatabaseLogin {

    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/jaas_example";
    // + "?useSSL=true"
    // + "&requireSSL=true"
    // + "&verifyServerCertificate=true";

    private static final String DB_USER = "root"; // Change selon ton utilisateur DB
    private static final String DB_PASSWORD = "root"; // Change selon ton mot de passe
DB

    public MyUser login(String username, String password) {

        // System.setProperty("javax.net.ssl.trustStore", "client-truststore.jks");
        // System.setProperty("javax.net.ssl.trustStorePassword", "changeit");

        // créer un nouvel utilisateur par défaut
        MyUser user = new MyUser("", "");
        user.setIsConnected(false);

        try {
            // Enregistrer le driver JDBC pour MySQL
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Connexion à la base de données MySQL
            Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER,
```

```
DB_PASSWORD);

            // Vérifier si les informations d'identification sont correctes
            String sql = "SELECT password, username FROM users WHERE username=?
AND password=?";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, password);
                // stmt.setString(2, hashPassword(password));
                ResultSet rs = stmt.executeQuery();
                if (rs.next()) {
                    user.setName(rs.getString("username"));
                    user.setIsConnected(true);
                } else {
                    System.out.println("Invalid credentials");
                }
            }

            // Récupérer les rôles de l'utilisateur
            // ce code est naturellement protégé des injections SQL grâce aux requêtes
            // préparées.
            sql = "SELECT role_name FROM roles WHERE username=?";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                ResultSet rs = stmt.executeQuery();
                while (rs.next()) {
                    String role = rs.getString("role_name");
                    user.setRole(role);
                }
            }

            // libère la connexion à la base de données
            connection.close();

        } catch (ClassNotFoundException e) {
            System.out.println("Driver MySQL non trouvé !");
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Error during login process: " + e.getMessage());
        }
        return user;
    }

    public boolean addUser(String username, String password, String role) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER,
DB_PASSWORD);

            String hashedPassword = hashPassword(password);
```

```
            // Ajouter dans la table users
            String sql = "INSERT INTO users (username, password) VALUES (?, ?)";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, hashedPassword);
                stmt.executeUpdate();
            }

            // Ajouter le rôle
            sql = "INSERT INTO roles (username, role_name) VALUES (?, ?)";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setString(1, username);
                stmt.setString(2, role);
                stmt.executeUpdate();
            }

            connection.close();
            return true;

        } catch (ClassNotFoundException e) {
            System.out.println("Driver MySQL non trouvé !");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Erreur SQL lors de l'ajout de l'utilisateur : " + e.getMessage());
        }

        return false;
    }

    // ----------------------------
    // Méthode de hash SHA-256
    // ----------------------------
    private String hashPassword(String password) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hashedBytes = digest.digest(password.getBytes("UTF-8"));
            // conversion bytes -> hex
            StringBuilder sb = new StringBuilder();
            for (byte b : hashedBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (Exception e) {
            throw new RuntimeException("Erreur de hash du mot de passe", e);
        }
    }
}
```

Infrastructures sécurisées modernes

Les différents certificats HTTPS

Les certificats HTTPS reposent sur le protocole TLS et permettent d'assurer l'authenticité d'un serveur ainsi que le chiffrement des échanges. On distingue principalement trois types de certificats selon leur niveau de validation.

- Les certificats **DV (Domain Validation)** vérifient uniquement que le demandeur contrôle le nom de domaine ; ils sont rapides à obtenir et largement suffisants pour la majorité des sites web, API et microservices (ex. On peut en générer des gratuits avec **let's encrypt** par exemple).
- **Les certificats OV (Organization Validation)** ajoutent une vérification de l'identité légale de l'organisation, ce qui permet d'identifier clairement l'entité derrière le service, souvent requis dans des contextes B2B.
- **Enfin, les certificats EV (Extended Validation)** impliquent une validation juridique approfondie de l'organisation ; bien qu'ils aient historiquement offert une meilleure visibilité dans les navigateurs, leur valeur ajoutée est aujourd'hui limitée SQL.

En complément, on distingue les certificats serveur, utilisés pour sécuriser les connexions HTTPS classiques, et les certificats client, utilisés dans des architectures plus avancées comme le mTLS (mutual TLS), où client et serveur s'authentifient mutuellement, notamment dans les environnements Zero Trust.

Principe de Zero trust models.

Le modèle Zero Trust repose sur le principe fondamental selon lequel **aucun acteur n'est considéré comme fiable par défaut**, qu'il soit à l'intérieur ou à l'extérieur du réseau. Contrairement aux modèles de sécurité traditionnels basés sur un périmètre réseau, le Zero Trust impose une **vérification systématique de l'identité**, du contexte et des autorisations à chaque requête. L'accès aux ressources est accordé selon le principe du moindre privilège, pour une durée limitée, et peut être réévalué dynamiquement. Ce modèle s'appuie généralement sur des mécanismes tels que l'authentification forte, les jetons courts (JWT), le chiffrement généralisé, le mTLS entre services et une journalisation fine des accès, ce qui le rend particulièrement adapté aux architectures cloud, microservices et environnements distribués.

Sécurité Java au sein des conteneurs

La sécurité d'une application Java exécutée dans un conteneur repose autant sur la **configuration de la JVM** que sur celle de **l'image** et de **l'environnement d'exécution**.

Les bonnes pratiques consistent à:

- Utiliser des images minimales
- Pas d'exécution en tant que root
- À s'assurer que la JVM est correctement configurée pour respecter les limites mémoire et CPU du conteneur.
- Les infos sensibles (mots de passe, clés, certificats) ne doivent jamais être intégrés à l'image mais fournis dynamiquement via des mécanismes sécurisés (variables d'environnement etc..).
- L'exposition réseau doit être réduite au strict nécessaire, les dépendances maintenues à jour, et les logs de sécurité centralisés afin de permettre la détection et l'analyse des comportements anormaux.

Les SIEMS

Un **SIEM** (Security Information and Event Management) est **une solution de sécurité** dont le rôle est de centraliser, corréler et **analyser les événements de sécurité** provenant de multiples sources d'un système d'information (applications, systèmes, réseaux, identités).

En agrégeant les logs et en appliquant des règles de détection, **le SIEM permet d'identifier des comportements suspects**, de générer des alertes et de faciliter les investigations ainsi que les audits de conformité.

Un SIEM ne bloque pas directement les attaques. Voici une liste de 3 SIEMS assez connus:

- **Elastic SIEM (Elastic Security)** Basé sur la stack ELK (Elasticsearch, Logstash, Kibana). Très utilisé, flexible, open source dans ses briques de base, et largement adopté dans les environnements cloud et DevSecOps.
- **Splunk Enterprise Security** Solution SIEM commerciale très puissante souvent utilisée dans les grandes entreprises et les SOC.
- **IBM QRadar** SIEM commercial orienté environnements enterprise

Le protocole CORS (Cross-Origin Resource Sharing)

Le protocole **CORS** permet de contrôler les requêtes cross-origin dans les navigateurs.CORS définit un mécanisme par lequel le serveur indique explicitement quels domaines, méthodes et headers sont autorisés à accéder à ses ressources. Les principaux headers utilisés sont :

- Access-Control-Allow-Origin : liste les domaines autorisés à accéder à la ressource.
- Access-Control-Allow-Methods : méthodes HTTP autorisées (GET, POST, OPTIONS, etc.).
- Access-Control-Allow-Headers : headers autorisés dans la requête.
- Access-Control-Allow-Credentials : indique si les cookies ou autres identifiants peuvent être envoyés.

Le protocole permet aussi l'utilisation d'un **preflight** avec la méthode **OPTIONS** pour vérifier quelles sont les méthodes disponibles définies en amont. Il est important de noter que **CORS est une protection côté navigateur, pas côté serveur**. Le serveur doit toujours valider l'authentification et les autorisations, même si CORS est activé.

Utiliser * pour Access-Control-Allow-Origin est pratique mais dangereux si les cookies ou tokens sont utilisés, c'est pour ça que le standard refuse que l'on combine allow-origin à * avec Access-Control-Allow-Credentials: true (les requêtes sont bloquées)

Techniquement, autoriser n'importe quelle source à interroger le serveur AVEC allow-credentials à true, cela revient à laisser la possibilité à un simple lien envoyé par mail d'éxécuter une action sur le serveur cible **mais avec le compte du client qui l'éxécute, c'est le principe de l'attaque CSRF**.

Les différents types d'attaque

Validation des entrées utilisateur (Never Trust User Input)

La validation des entrées utilisateur est un principe fondamental de la sécurité applicative, résumé par la maxime « Never Trust User Input ». **Toute donnée provenant d'un utilisateur ou d'une source externe doit être considérée comme potentiellement malveillante**.

De manière générale, il faut donc partir du principe que tout input provenant d'un éventuel client doit être analysé, vérifié et éventuellement 'assainit' (sanitizing).

Cela contribue à éviter un certain nombre d'attaques notamment, les **injections SQL**, les injections de scripts (**failles XSS**) etc ...

La faille CSRF (Cross-Site Request Forgery)

La faille CSRF se produit lorsqu'un site malveillant **parvient à forcer un utilisateur authentifié à exécuter une action à son insu sur une application web de confiance**. Typiquement, si un utilisateur est connecté à un site bancaire, **une page tierce peut déclencher des requêtes HTTP (comme un virement) sans que l'utilisateur le sache, exploitant sa session active**

La protection passe par plusieurs mécanismes : **jetons anti-CSRF (CSRF token)** uniques par session et par formulaire, vérification de l'en-tête Origin ou Referer, et utilisation de cookies SameSite pour limiter l'envoi automatique de cookies sur des domaines tiers. En résumé, CSRF exploite la confiance implicite du serveur envers le navigateur, et la mitigation consiste à rompre cette confiance par une vérification explicite de la légitimité de la requête.

Architectures sécurisées by design

Une architecture sécurisée by design consiste à intégrer la sécurité dès la conception d'une application ou d'un système, plutôt que de la traiter comme un ajout après coup. Elle repose sur plusieurs principes clés :

- Principe du moindre privilège : chaque composant, service ou utilisateur n'a accès qu'aux ressources strictement nécessaires.
- Segmentation et isolation : les différents modules ou services sont cloisonnés pour limiter la portée d'une éventuelle compromission.
- Sécurité dès la couche réseau : utilisation de HTTPS/TLS, pare-feu, contrôle des flux entrants et sortants.
- Protection des entrées et sorties : validation stricte des données utilisateurs (Never Trust User Input), encodage à la sortie, et contrôle des headers (CORS, CSP).
- ournalisation et supervision : collecte centralisée des logs de sécurité pour détection d'anomalies et intégration à un SIEM.
- Défense en profondeur : combinaison de plusieurs mécanismes de sécurité (authentification forte, tokens, validation côté serveur) pour que la compromission d'un seul niveau ne mette pas l'ensemble en danger.

En pratique, une architecture sécurisée by design intègre la sécurité à chaque étape du cycle de vie : conception, développement, déploiement et exploitation. Cela permet de réduire les vulnérabilités critiques comme XSS, CSRF, injection SQL, et de faciliter la conformité réglementaire et la résilience des systèmes face aux attaques.