.Net Sécurité

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

.Net Sécurité

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Rappels sur la sécurité applicative

Introduction

- **Importance de la sécurité** : protège l'**intégrité**, la **disponibilité** et la **confidentialité** des **données**.
- **Quelques Types d'attaques fréquentes** :
  - Injection SQL
  - Cross-Site Scripting (XSS)
  - Vol de sessions, etc.
- **Cycle de vie sécurisé du développement logiciel** : intégrer la sécurité à chaque étape (ex: DevSecOps).

Fonctionnement de la pile d'exécution

- **Gestion de la mémoire** : **Pile/Stack** pour les variables locales (valeurs), **Tas/Heap** pour les objets (références).
- **Débordements de pile (Stack Overflow)** : erreurs de récursion ou d'allocation mémoire excessive.
- **Protection des données en mémoire** : chiffrement ou nettoyage après utilisation (Garbage Collection).

L'analyse de code

- **Statique vs Dynamique** :
  - **Analyse statique** : inspecte le code sans l'exécuter.
  - **Analyse dynamique** : teste le comportement en temps réel.
- **Outils courants** :
  - Visual Studio Code Analysis, SonarQube/SonarLint, ReSharper.

Hijacking de ressources

- **Concept** : exploitation d'une ressource pour un usage non prévu (e.g., détournement d'API).
- **Risques** :
  - Vol de données
  - Utilisation abusive de services tiers.
- **Prévention** : authentification forte, contrôle d'accès strict (définition des **ACL**, Access Control Lists).

Les overflows

- **Buffer Overflow** : dépassement des limites de mémoire allouée à une variable.
- **Stack Overflow** : surcharge de la pile d'exécution.
- **Protection** :
  - Validation des entrées (ex: Model Validation).
  - Utilisation de types de données sécurisés (Types .NET).

Protections lors de l'exécution

- **ASLR (Address Space Layout Randomization)** : réorganisation aléatoire des segments mémoire pour éviter les exploits (emplacements pile/tas/librairies).
- **DEP (Data Execution Prevention)** : empêche l'exécution de code non exécutable (ex : injection par buffer overflow).
- **Systèmes de contrôle d'intégrité** : surveillent les modifications non autorisées de fichiers/programmes/config systèmes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Sécurité du Framework .NET

Namespaces de Sécurité

- Utilisation des **namespaces** spécialisés pour gérer la sécurité dans .NET:
  - `System.Security.Cryptography` : Chiffrement, hachage.
  - `System.Net.Security` : Sécurisation des communications réseau (TLS/SSL).
  - `System.Security.Principal` : Gestion des identités, rôles et permissions.

System.Security.Cryptography

Ce namespace contient les classes nécessaires pour implémenter des **algorithmes de cryptographie** comme le **chiffrement**, le **hachage**, la **signature numérique**, et plus encore. Il joue un rôle fondamental dans la **sécurisation des données sensibles**, le stockage de mots de passe et la protection de l'intégrité des données.

Exemples

Chiffrement symétrique avec AES :

```
using System.Security.Cryptography;

public byte[] EncryptData(string plainText, byte[] key, byte[] iv)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = key;
        aesAlg.IV = iv; // Initialisation Vector

        ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
        using (MemoryStream msEncrypt = new MemoryStream())
        {
            using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor,
                CryptoStreamMode.Write))
            {
                using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                {
                    swEncrypt.Write(plainText);
                }
                return msEncrypt.ToArray();
            }
        }
    }
}
```

Hachage des mots de passe avec **SHA256**.

```
using System.Security.Cryptography;
using System.Text;

public string HashPassword(string password)
{
    using (SHA256 sha256Hash = SHA256.Create())
    {
        byte[] bytes = sha256Hash.ComputeHash(
            Encoding.UTF8.GetBytes(password));
        return Convert.ToBase64String(bytes);
    }
}
```

System.Net.Security

Ce namespace fournit des classes qui permettent de **sécuriser les communications réseau**, par exemple avec des protocoles comme **SSL/TLS**. Il est utilisé pour garantir la **confidentialité** et l'**intégrité** des **données transmises via le réseau**.

Exemples : Sécurisation réseau

- Connexion sécurisée via **SSL/TLS**.
- Validation des certificats SSL.

```
using System.Net.Security;
using System.Net.Sockets;
using System.Security.Cryptography.X509Certificates;

public void SecureCommunication()
{
    TcpClient client = new TcpClient("example.com", 443);
    SslStream sslStream = new SslStream(client.GetStream(), false,
        new RemoteCertificateValidationCallback(ValidateServerCertificate), null);

    // Initie la connexion sécurisée avec TLS
    sslStream.AuthenticateAsClient("example.com");
}

public static bool ValidateServerCertificate(object sender, X509Certificate certificate, X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
    // Validation du certificat SSL du serveur
    return sslPolicyErrors == SslPolicyErrors.None;
}
```

System.Security.Principal

Ce namespace est utilisé pour la **gestion des identités**, des **rôles** et des **permissions**. Il fournit des classes comme **WindowsIdentity** et **WindowsPrincipal** pour gérer l'**authentification** et l'**autorisation** des utilisateurs dans des applications .NET.

Exemple : Gestion des Identités

- Cet exemple vérifie si l'utilisateur actuel est un administrateur sur la machine locale.

```
using System.Security.Principal;

public void CheckUserRole()
{
    WindowsIdentity identity = WindowsIdentity.GetCurrent();
    WindowsPrincipal principal = new WindowsPrincipal(identity);

    if (principal.IsInRole(WindowsBuiltInRole.Administrator))
        Console.WriteLine("L'utilisateur est administrateur.");
    else
        Console.WriteLine("L'utilisateur n'est pas administrateur.");
}
```

Le Sandboxing

Le sandboxing est une **technique de sécurité** qui permet d'**exécuter** des applications ou du code dans un **environnement isolé**, **limitant** ainsi **leur accès** aux ressources du système. Cela est particulièrement important pour les applications qui traitent des données non fiables ou qui proviennent de sources potentiellement dangereuses.

Le Sandboxing

- **Isolation d'exécution** : Limiter les accès d'un code à des ressources spécifiques (ex: machine virtuelle).
- **Contexte de sécurité** : empêche un code non fiable d'accéder aux ressources sensibles (privilèges limités fichiers/réseau/...).
- **Particularité en .NET** : `AppDomain` pour l'isolation d'applications.

```
AppDomain myDomain = AppDomain.CreateDomain("SandboxDomain");
try
{
    myDomain.ExecuteAssembly("MySandboxedApp.exe");
}
finally
{
    AppDomain.Unload(myDomain);
}
```

L'attribut APTCA

- **APTCA (Allow Partially Trusted Callers Attribute)** :
  - Permet aux assemblages partiellement approuvés d'appeler des méthodes de bibliothèques sécurisées.
  - **Risque** : expose des fonctionnalités sensibles si mal utilisé.

```
using System.Security;

[assembly: SecurityRules(SecurityRuleSet.Level2)] // appliqués à l'assembly complet
[assembly: AllowPartiallyTrustedCallers]

public class SecureLibrary
{
    public void SecureMethod()
    {
        // Code sécurisé qui peut être appelé par des assemblages partiellement approuvés (n'ayant pas tout les droits d'accès)
        Console.WriteLine("Méthode sécurisée appelée.");
    }
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Chiffrement avec C#

Concepts clés

- **Confidentialité** : Assure que les **informations** sont **accessibles uniquement aux entités autorisées** et protégées contre l'accès non autorisé.
- **Intégrité** : Garantit que les données n'**ont pas été altérées**.
- **Authenticité** : **Vérifie** l'**identité** de l'émetteur des données.
- **Non-Répudiation** : **Empêche** un expéditeur de **nier avoir envoyé un message**.

Hachage

- **Hachage** : Génération d'une empreinte unique et fixe à partir de données pour assurer leur **intégrité** (unidirectionnel).
- **Salage (Salting)** : Technique ajoutant des **données aléatoires** avant le hachage pour renforcer la sécurité.

Les Fonctions de Hash

- **Fonctions courantes** : SHA-256, SHA-512, MD5 (obsolète).
- **Propriétés** :
  - **One-way** : impossible de retrouver les données d'origine à partir du hash.
  - **Utilisation** : vérification de l'intégrité des données.

Exemple de Fonction de Hash (SHA-256)

```
using System.Security.Cryptography;
using System.Text;

public string ComputeSha256Hash(string rawData)
{
    using (SHA256 sha256Hash = SHA256.Create())
    {
        byte[] bytes = sha256Hash.ComputeHash(Encoding.UTF8.GetBytes(rawData)); // hash en tableau d'octets
        StringBuilder builder = new StringBuilder();

        for (int i = 0; i < bytes.Length; i++)
        {
            builder.Append(bytes[i].ToString("x2")); // hexadécimal 2 digits (10 = 0a, 225 = fb, ...)
        }
        return builder.ToString(); // représentation hexadécimale du hash
    }
}

// Utilisation
string hash = ComputeSha256Hash("Hello, World!");
```

Principe de Salage

- Si de nombreux hash sont volés, il est possible de retrouver les données non chiffrées.
- **Sel** : **Valeur aléatoire** générée et **ajoutée** aux données **avant** le processus de **hachage**.
- Le sel est **stocké** avec le hash pour permettre la **vérification** des données ultérieurement.
- Protège contre les attaques par tables de hachage pré-calculées (rainbow tables) en **rendant chaque hash unique** même pour des données identiques.

Exemple avec Salage

```
public static string HashPassword(string password)
{
    // Générer un sel aléatoire
    byte[] salt = new byte[16];
    using (var rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(salt);
    }
    using (var sha256 = SHA256.Create())
    {
        // Combiner le mot de passe et le sel
        byte[] passwordBytes = Encoding.UTF8.GetBytes(password);
        byte[] saltedPassword = new byte[passwordBytes.Length
            + salt.Length];

        Buffer.BlockCopy(passwordBytes, 0, saltedPassword,
            0, passwordBytes.Length);
        Buffer.BlockCopy(salt, 0, saltedPassword,
            passwordBytes.Length, salt.Length);

        // Hacher le mot de passe salé
        byte[] hash = sha256.ComputeHash(saltedPassword);
        return Convert.ToBase64String(hash);
    }
}
```

```
// Utilisation
string password = "monMotDePasseSecurise";
byte[] salt = SalageExample.GenerateSalt();
string hashedPassword = SalageExample.HashPassword(password, salt);

Console.WriteLine("Sel (Base64) : " + Convert.ToBase64String(salt));
Console.WriteLine("Mot de passe haché (Base64) : " + hashedPassword);
```

Chiffrement

- **Chiffrement** : Transformation des données en texte illisible pour protéger leur **confidentialité** (bidirectionel).
  - **Symétrique** : même clé pour chiffrer/déchiffrer (ex. AES).
  - **Asymétrique** : clé publique pour chiffrer, clé privée pour déchiffrer (ex. RSA).
- **Rôle du chiffrement** : sécuriser les données sensibles en transit ou au repos/stockées.
- **Signature numérique** : Utilisation du chiffrement asymétrique pour vérifier l'**authenticité** et l'**intégrité** d'un message.

Algorithmes Symétriques (AES)

- **Une seule clé** pour **chiffrer** et **déchiffrer**.
- **AES (Advanced Encryption Standard)** :
  - Taille des clés : 128, 192, ou 256 bits.
  - **Rapide** et efficace pour de grandes quantités de données.
  - **Implémentation** dans C# via `System.Security.Cryptography`.

Génération de Clés Symétriques (AES)

```
using System.Security.Cryptography;

byte[] key = new byte[32]; // Clé de 256 bits pour AES
using (var rng = new RNGCryptoServiceProvider())
{
    rng.GetBytes(key); // Remplissage de la clé avec des valeurs aléatoires
}
Console.WriteLine(Convert.ToBase64String(key));
```

- **Taille de clé** : Peut être de 128, 192, ou 256 bits.

Exemple d'Implémentation AES

```
static byte[] Encrypt(string text, byte[] key, byte[] iv)
{
    using var aes = Aes.Create();
    aes.Key = key; aes.IV = iv;
    using var ms = new MemoryStream();
    using var cs = new CryptoStream(ms, aes.CreateEncryptor(),
        CryptoStreamMode.Write);
    using var sw = new StreamWriter(cs);
    sw.Write(text);
    return ms.ToArray();
}

static string Decrypt(byte[] cipherText, byte[] key, byte[] iv)
{
    using var aes = Aes.Create();
    aes.Key = key; aes.IV = iv;
    using var ms = new MemoryStream(cipherText);
    using var cs = new CryptoStream(ms, aes.CreateDecryptor(),
        CryptoStreamMode.Read);
    using var sr = new StreamReader(cs);
    return sr.ReadToEnd();
}
```

```
string plainText = "Hello, AES!";

byte[] key = new byte[32]; // Clé de 256 bits
byte[] iv = new byte[16];  // IV de 128 bits

using (var rng = new RNGCryptoServiceProvider())
{
    rng.GetBytes(key);
    rng.GetBytes(iv);
}

// Chiffrement
byte[] encrypted = Encrypt(plainText, key, iv);
Console.WriteLine("Chiffré : "
    + Convert.ToBase64String(encrypted));

// Déchiffrement
string decrypted = Decrypt(encrypted, key, iv);
Console.WriteLine("Déchiffré : " + decrypted);
```

Algorithmes Asymétriques (RSA)

- **RSA** : **clé publique** pour **chiffrer**, **clé privée** pour **déchiffrer**.
- **Utilisation** :
  - Transmission sécurisée de clés.
  - Authentification via certificats numériques.
- Exemple : Généralement utilisé pour le protocole **ssh**

Génération de Paires de Clés RSA

```
using System.Security.Cryptography;

using (RSA rsa = RSA.Create(2048)) // Génère une clé RSA de 2048 bits
{
    var privateKey = rsa.ExportRSAPrivateKey();
    var publicKey = rsa.ExportRSAPublicKey();
    Console.WriteLine("Clé publique : " + Convert.ToBase64String(publicKey));
}
```

- **RSA** : Algorithme de chiffrement asymétrique.
- **Utilisation** : Chiffrement/déchiffrement et signature numérique.

Exemple d'Implémentation RSA

```
static byte[] Encrypt(string text, byte[] publicKey)
{
    using var rsa = RSA.Create();
    rsa.ImportRSAPublicKey(publicKey, out _);
    return rsa.Encrypt(Encoding.UTF8.GetBytes(text),
        RSAEncryptionPadding.OaepSHA256);
}

static string Decrypt(byte[] cipherText, byte[] privateKey)
{
    using var rsa = RSA.Create();
    rsa.ImportRSAPrivateKey(privateKey, out _);
    return Encoding.UTF8.GetString(rsa.Decrypt(cipherText,
        RSAEncryptionPadding.OaepSHA256));
}
```

```
string plainText = "Hello, RSA!";

// Génération des clés RSA
using var rsa = RSA.Create();
byte[] publicKey = rsa.ExportRSAPublicKey();
byte[] privateKey = rsa.ExportRSAPrivateKey();

// Chiffrement
byte[] encrypted = Encrypt(plainText, publicKey);
Console.WriteLine("Chiffré : "
    + Convert.ToBase64String(encrypted));

// Déchiffrement
string decrypted = Decrypt(encrypted, privateKey);
Console.WriteLine("Déchiffré : " + decrypted);
```

Principe de la signature numérique

- **Objectif** : Assurer l'**authenticité**, l'**intégrité** et la **non-répudiation** d'un message ou document.
- **Applications** : Signatures de certificats, authentification de transactions, sécurisation des communications.

Fonctionnement de la signature numérique

1. L'émetteur **génère** un **hash** du message.
2. Ce hash est **chiffré** avec la **clé privée** de l'émetteur, créant la **signature numérique**.
3. La signature et le message sont **envoyés** au destinataire.
4. Le destinataire **déchiffre la signature** avec la **clé publique** de l'émetteur pour récupérer le hash.
5. Il **compare** ce hash avec celui qu'il génère à partir du message reçu pour vérifier l'intégrité et l'authenticité.

HMAC-SHA

- **Hash-based Message Authentication Code** : Combinaison d'une fonction de hash cryptographique (comme SHA-256) avec une **clé secrète** (symétrique).
- **Objectif** : Garantir à la fois l'**intégrité** et l'**authenticité** des données transmises.
- **Fonctionnement** :
  1. Le message est combiné avec une clé secrète (signature).
  2. Le résultat est passé dans la fonction de hash (ex: SHA-256).
  3. Un HMAC est produit, à la fois **unique** et sécurisé.

Exemple d'implémentation en C# (HMAC-SHA256)

```
using System.Security.Cryptography;
using System.Text;

public class HMACExample
{
    public static string ComputeHMACSHA256(string message, string key)
    {
        using (var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(key)))
        {
            byte[] hashValue = hmac.ComputeHash(Encoding.UTF8.GetBytes(message));
            return BitConverter.ToString(hashValue).Replace("-", "").ToLower();
        }
    }
}

// Utilisation
string message = "Message à protéger";
string secretKey = "maCléSecrète";
string hmacResult = HMACExample.ComputeHMACSHA256(message, secretKey);
Console.WriteLine("HMAC-SHA256 : " + hmacResult);
```

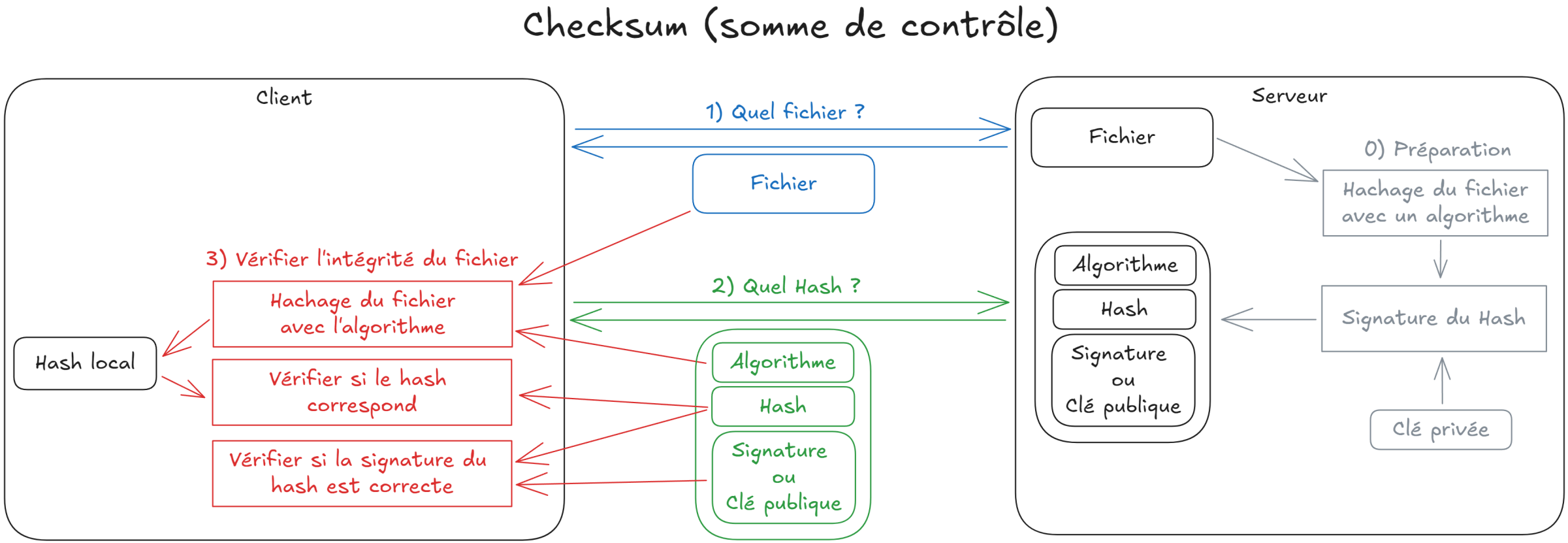
Checksum

- **Définition** : Un **checksum** est une valeur numérique calculée à partir d'un ensemble de données, utilisée pour vérifier l'**intégrité** des données.
- **Fonctionnement** :
  - Un algorithme additionne ou effectue des opérations sur les données pour produire un code unique.
  - Ce code est envoyé avec les données ou stocké pour vérification ultérieure.

Checksum

- **Utilisation** :
  - Détecte les erreurs lors de la transmission ou du stockage.
  - Utilisé dans des protocoles comme TCP/IP pour garantir l'intégrité des données.
- **Limitation** : Bien qu'il soit efficace pour détecter des erreurs, un checksum ne garantit pas une protection complète contre les modifications intentionnelles.
- Il existe une **version plus poussée** avec l'utilisation du **chiffrement asymétrique** et des **signatures** et resistant au **man-in-th-middle**.

Exemple Checksum avec Signature du Hash



[Exemple avec Debian]

Windows DPAPI (Data Protection API)

- **Protège** les données **localement**, utilise des **clés** liées à l'**utilisateur** et la **machine**.
- Facilite le chiffrement en lui laissant la gestion des clés (chiffrement sans couture).
- Utilisé pour les applications et services windows.

File.Encrypt :

- `File.Encrypt` est une méthode de la classe `System.IO.File` qui permet de **chiffrer un fichier sur le disque**.
- Cette méthode utilise la **DPAPI**, seul l'utilisateur qui a chiffré le fichier peut y accéder, car la **clé de déchiffrement est liée à l'utilisateur**.

```
string filePath = "sensitiveData.txt";
// Écriture de données dans le fichier
File.WriteAllText(filePath, "Ceci est des données sensibles.");
// Chiffrement du fichier
File.Encrypt(filePath);
Console.WriteLine("Le fichier a été chiffré.");
// Pour déchiffrer le fichier, utiliser File.Decrypt
File.Decrypt(filePath);
Console.WriteLine("Le fichier a été déchiffré.");
```

Exploitation d'HTTP Basic

- **HTTP Basic** : transmission en clair d'un nom d'utilisateur et mot de passe encodé en Base64.
- **Dangers** :
  - Vulnérabilité aux attaques de type "**man-in-the-middle**".
  - **Recommandation** : **Utilisation obligatoire de HTTPS** pour protéger les données/identifiants.

Les Certificats

- Un **certificat numérique** est un fichier électronique qui lie une clé publique à l'identité de son propriétaire.
- Les certificats permettent de **vérifier l'identité** d'un serveur ou d'une entité lors des communications sécurisées.
- Ils sont utilisés dans les **protocoles de sécurité** tels que **TLS** pour sécuriser les échanges de données.

TLS et HTTPS

- **TLS (Transport Layer Security)** : Protocole de sécurité qui assure la confidentialité et l'intégrité des données échangées sur un réseau.
- **HTTPS** : Version sécurisée de HTTP qui utilise TLS pour chiffrer les échanges entre le navigateur et le serveur.
  - Les sites web utilisent des **certificats** pour prouver leur identité et activer le chiffrement.

Étapes d'une connexion HTTPS :

1. Le navigateur demande une connexion HTTPS.
2. Le serveur envoie son **certificat**.
3. Le navigateur vérifie que le certificat est **valide et approuvé**.
4. Si valide, une session sécurisée est établie via TLS.

Certificats X.509

X.509 : Standard utilisé pour les certificats numériques. Il définit le format des certificats et les champs requis. Il contient :

- **Nom du sujet** (identité de l'entité ou du serveur).
- **Clé publique** : Utilisée pour vérifier l'identité du serveur ou chiffrer les données.
- **Émetteur** : Autorité qui a émis le certificat.
- **Signature** : Preuve que l'autorité de certification a approuvé le certificat.
- **Date de validité** : Période durant laquelle le certificat est valide.

Certificats auto-signés

- Un **certificat auto-signé** est un certificat où l'émetteur est le même que le sujet.
- **Utilisation** :
  - Souvent utilisés dans des environnements de développement/test ou pour des applications internes.
  - Ne sont **pas approuvés** par les navigateurs, donc non adaptés pour un site web public en production.

Génération d'un Certificat X.509 (Auto-signé)

```
using System.Security.Cryptography.X509Certificates;

var rsa = RSA.Create(2048);
var request = new CertificateRequest("CN=MyCert", rsa,
    HashAlgorithmName.SHA256,
    RSASignaturePadding.Pkcs1);

var cert = request.CreateSelfSigned(DateTimeOffset.Now,
    DateTimeOffset.Now.AddYears(1));

Console.WriteLine(cert);

// Exportation du Certificat
byte[] certBytes = cert.Export(X509ContentType.Pfx, "password123!");
File.WriteAllBytes("myCert.pfx", certBytes);
```

Autorités de Certification (CA)

- **Autorités de Certification** (CA) : Entités de confiance qui vérifient et délivrent des certificats pour prouver l'identité des serveurs.
  - Exemple de CA : **Let's Encrypt**, **DigiCert**, **GlobalSign**.
- Un certificat émis par une CA approuvée est **fiable** par les navigateurs, contrairement aux certificats auto-signés.

Certificats approuvés (Let's Encrypt)

- Les **certificats approuvés** sont délivrés par une **Autorité de Certification (CA)**, comme **Let’s Encrypt** ou **DigiCert**.
- **Let's Encrypt** : Fournit des certificats **gratuits** pour sécuriser les sites web via HTTPS.
  - Utilise un processus automatisé appelé **ACME (Automatic Certificate Management Environment)** pour obtenir, installer et renouveler les certificats sans intervention manuelle.

Sécurisation des Données en Transit

- **Chiffrement Asymétrique** : Utilisation de **certificats** pour **sécuriser les communications**, assurant la **confidentialité** et l'**intégrité** des données.
- **Signature Numérique** : Garantit l'**authenticité** de l'expéditeur et vérifie l'**intégrité** du message.

Sécurisation des Données au Repos

- **Chiffrement Symétrique** : Déchiffrement **rapide** pour l'accès aux données **sensibles**.
- **Hachage** : Protège l'**intégrité** des données **sans possibilité de retrouver l'original**.

Quand Chiffrer et Quoi ?

- Évaluer le type de données sensibles et le besoin d'accès rapide.
- **Chiffrement Total** : Lent mais sécurisé.
- **Chiffrement Sélectif** : Efficace mais nécessite une évaluation des risques.

Exigences Légales et Normatives

- **Secteurs Réglementés** : Chiffrement obligatoire dans les secteurs de la santé et bancaire.
- **Conformité RGPD** : Protection des données sensibles contre les violations.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

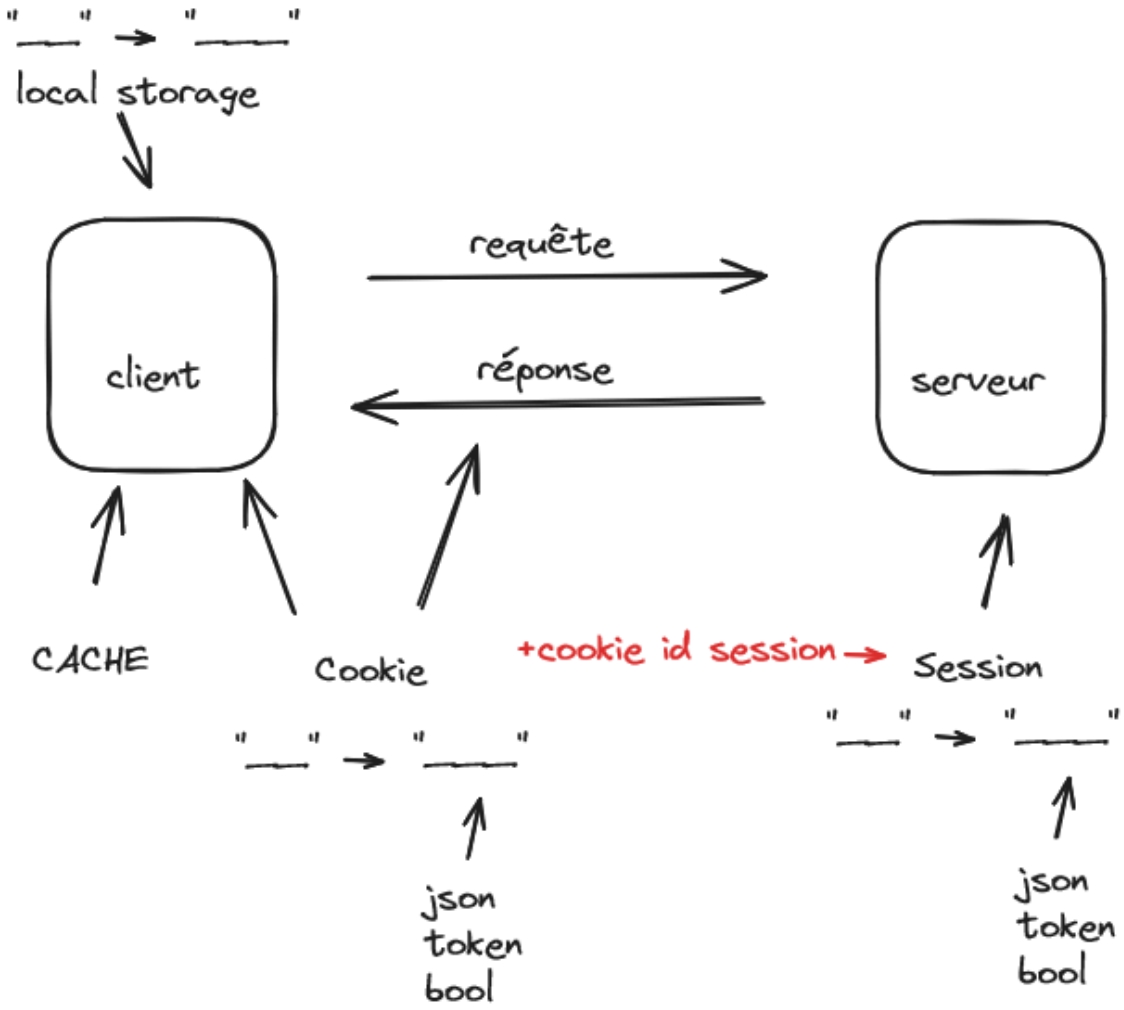
Authentification

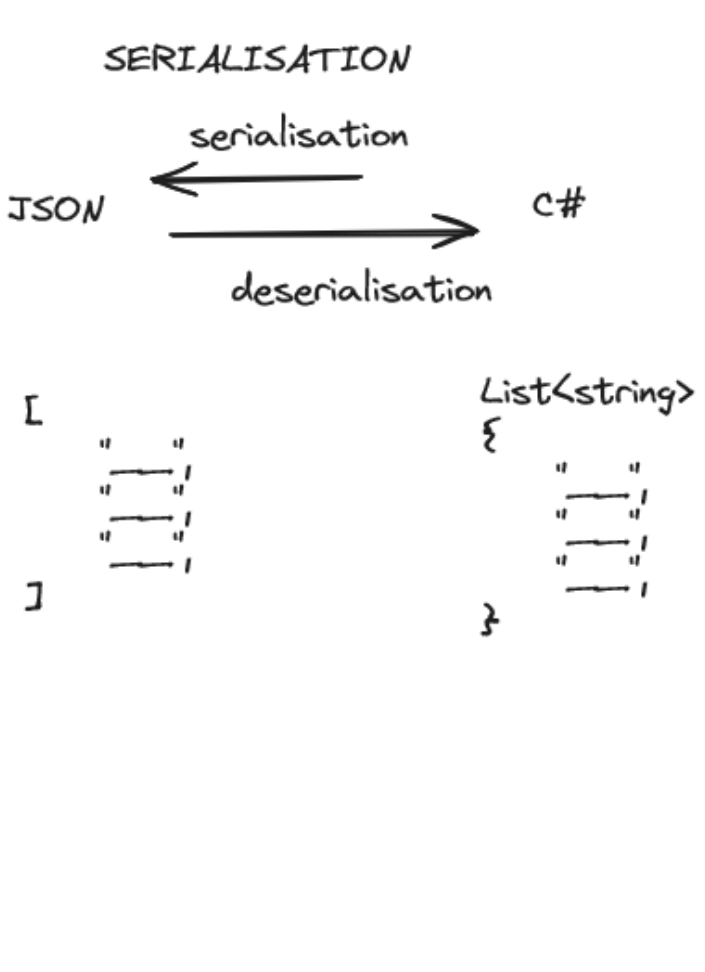
Les 3 phases de la connexion sécurisée:

1. **Identification** : On **identifie une entité** et on lui donne de quoi affirmer cette identité en retour (token/credentials/clé).
2. **Authentification** : On **vérifie l'authenticité d'un message**/une action envoyé (peut contenir token/credentials/clé à vérifier).
3. **Autorisation** : On **définit les accès à une ressources** en fonction des règles définis dans les **A.C.L.** (Access Control Lists). Ces accès sont donnés en fonction des information Authentiques.

- Les règles A.C.L. ne sont **pas forcément associées à une entité**.
  Le porteur d'un "token" n'est pas forcément identifié (anonyme).

Rappel sur les stockage du web





Authentification par Session en ASP.NET Core MVC

- **Principe** : Après une **connexion réussie**, le serveur stocke l'**identifiant de l'utilisateur** dans une **session**. Les requêtes suivantes utilisent cette session pour **vérifier l'identité de l'utilisateur**.
- **Session vs Cookie** : La **session** stocke les données **côté serveur**, tandis que les **cookies** permettent au **client** de stocker des informations minimales.

Configurer les Sessions dans ASP.NET Core

1. **Ajouter les services de session** dans `Program.cs` :

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(30);
});
```

2. **Activer les sessions** dans le pipeline de requêtes :

```
app.UseSession();
```

Renommage du cookie identifiant session

- Il est intéressant de renommer le cookie d'id session :

```
// Ajouter le service de session avec des options personnalisées
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(30); // Durée d'inactivité avant l'expiration de la session
    options.Cookie.Name = "MonCookieDeSession"; // Changer le nom du cookie de session
    options.Cookie.HttpOnly = true; // Cookie accessible uniquement via HTTP (pas JavaScript)
    options.Cookie.IsEssential = true; // Essentiel pour permettre le cookie sans consentement dans des régulations RGPD
});
```

- Plus généralement on cherche à **changer les configuration par défaut** afin de **repousser** l'exploitation des failles zero-day
- Exemple : changer le nom du header Authorization (JWT)

Démo - Étape 1 : Page de Connexion

1. **Créer une vue pour la connexion (`Login.cshtml`) :**

```
<form method="post" action="/Account/Login">
    <input type="text" name="username" placeholder="Username" />
    <input type="password" name="password" placeholder="Password" />
    <button type="submit">Login</button>
</form>
```

2. **Ajouter l'action de connexion dans `AccountController.cs` :**

```
public IActionResult Login() => View();
```

Démo - Étape 2 : Vérification des Identifiants

- **Traitement de la soumission de formulaire** :

```
[HttpPost]
public IActionResult Login(string username, string password)
{
    if (username == "admin" && password == "password")
    {
        HttpContext.Session.SetString("username", username);
        return RedirectToAction("Index", "Home");
    }
    ViewBag.Error = "Invalid credentials";
    return View();
}
```

Démo - Étape 3 : Protéger les Pages avec Session

- **Protéger les actions des contrôleurs** :

```
public IActionResult SecurePage()
{
    if (HttpContext.Session.GetString("username") == null)
    {
        return RedirectToAction("Login", "Account");
    }
    return View();
}
```

- **Afficher la session dans une vue** :

```
<h1>Welcome, @HttpContext.Session.GetString("username")!</h1>
```

Démo - Étape 4 : Déconnexion

- **Ajouter l'action de déconnexion** :

```
public IActionResult Logout()
{
    HttpContext.Session.Clear();
    return RedirectToAction("Login", "Account");
}
```

- **Ajouter un bouton "Déconnexion"** dans les vues protégées :

```
<form method="post" action="/Account/Logout">
    <button type="submit">Logout</button>
</form>
```

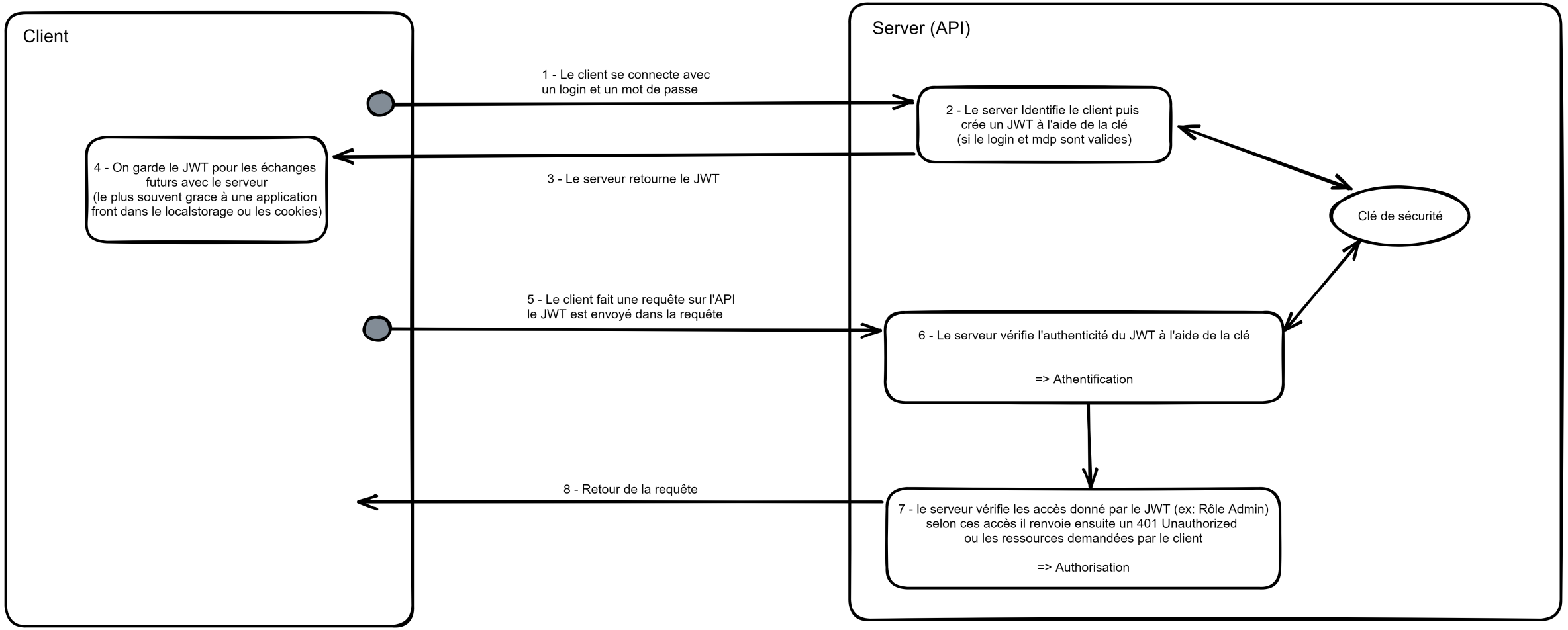
Récap Auth par Session

- **Sessions** : Permettent de stocker des **informations côté serveur** pour une durée limitée.
- **Authentification par session** : **Simple à implémenter** pour des applications où la gestion d'identité est nécessaire mais sans la complexité des tokens JWT.
- **Prochaine étape** : Ajout d'authentification plus robuste via Identity ou JWT.

Sécurité des Web Services REST (API REST)

- 3 alternatives populaires :
  - **JSON Web Tokens** (JWT)
  - OAuth 2.0
  - OpenID Connect

Fonctionnement JWT



C'est quoi un JWT ?

Un JWT est **toujours lisible par tout le monde** (**lecture seule**), cependant on ne peut l'**écrire** que si on possède la **clé de sécurité**.
Il est donc **Infalsifiable** sans la clé de sécurité.

1 Token = 3 parties :

- Headers : type de token et algorithme de chiffrement
- Payload (Body/Data) :
- Signature : ce qui permet de valider et déchiffrer le token

Exemple de JWT :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjI0MyIsIm5hbWUiOiJHdWlsbGF1bWUgTWFpcmVzc2UiLCJlbnRlcnByaXNlIjoiVXRvcGlvcyIsIm1lc3NhZ2Ugc2VjcmV0IjoiTGUgQyMgZG9taW5lIHRvdXQgbGVzIGF1dHJlcyBsYW5nYWdlcyBjJ2VzdCDDqXZpZGVudCJ9.b0ZbcGLQT-uxWEDvI-vlPpq1qq3IzfR0k5kUt0_bR-E

Déchiffrer ce token via jwt.io

Comment travailler avec le JWT

- Une fois le token généré, il sera nécessaire de le **garder côté front-end** car il sert à affirmer **qui** envoie les requêtes.
- Le token JWT est généralement stocké dans le **localStorage** ou dans un **cookie** sécurisé.

Où stocker le JWT

- **localStorage** : Accessible en JavaScript et persistant entre les sessions du navigateur.
  - **Avantages** : Facile d'accès pour les requêtes.
  - **Inconvénients** : Vulnérable aux attaques XSS.
- **Cookies** (avec l'attribut `HttpOnly` et `Secure`) :
  - **Avantages** : Sécurisé et peut être envoyé automatiquement avec chaque requête HTTP.
  - **Inconvénients** : Vulnérable aux attaques CSRF si mal configuré.

Utilisation dans les requêtes

- Le JWT est généralement envoyé dans les **en-têtes HTTP** de chaque requête en utilisant l'en-tête **Authorization** :

```
Authorization: Bearer <token>
```

- Dans **fetch** ou **HttpClient** (Javascript):

```
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Authorization': 'Bearer ' + localStorage.getItem('token')
  }
});
```

Authentication et Authorization dans ASP.NET Core

- En ASP.NET Core, ce fonctionnement est géré par des **middlewares** configurables et des **annotations** sur les actions des contrôleurs.
- Cela évite au développeur de tout refaire lui-même.

Démo - Étape 1 : Configurer l'Authentification

```
// récupération des AppSettings (clé+expiration)
// et ajout au conteneur de dépendances
builder.Services.Configure<AppSettings>(
    builder.Configuration.GetSection("AppSettings"));

// récupération de la clé de sécurité
var appSettingsSection = builder.Configuration.GetSection("AppSettings");
var appSettings = appSettingsSection.Get<AppSettings>();
var key = Encoding.ASCII.GetBytes(appSettings!.SecretKey!);

// ajout de l'authentication
builder.Services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(x =>
{
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        // utilisation de la clé secrète
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

```
app.UseHttpsRedirection();

// /!\ Ne pas oublier d'ajouter le middleware
app.UseAuthentication();

app.UseAuthorization();
```

Démo - Étape 2 : Créer des DTOs pour la Connexion

```
public class LoginRequestDTO
{
    [Required]
    public string? Email { get; set; }
    [Required]
    public string? Password { get; set; }
}
```

```
public class LoginResponseDTO
{
    public bool IsSuccessful { get; set; }
    public string? ErrorMessage { get; set; }
    public User? User { get; set; }
    public string? Token { get; set; }
}
```

```
public class RegisterRequestDTO
{
    [DataType(DataType.EmailAddress)]
    [Required(ErrorMessage = "Email is required")]
    // valider le regex d'un email
    [EmailAddress(ErrorMessage = "Email is invalid")]
    public string? Email { get; set; }
    [DataType(DataType.Password)]
    // validator custom style "AAaa00++"
    [PasswordValidator]
    public string? Password { get; set; }
    public bool IsAdmin { get; set; } = false;
    // champs nom prénom téléphone ...
}
```

```
public class RegisterResponseDTO
{
    public bool IsSuccessful { get; set; }
    public string? ErrorMessage { get; set; }
    public User? User { get; set; }
}
```

Démo - Étape 3 : Créer un AuthenticationController

```
[Route("api/[controller]")]
[ApiController]
public class AuthenticationController : ControllerBase
{
    private readonly IUserRepository userRepository;
    private readonly AppSettings appSettings;
    private readonly Encryptor encryptor;

    public AuthenticationController(IUserRepository userRepository,
                                    IOptions<AppSettings> appSettings)
    {
        this.userRepository = userRepository;
        this.appSettings = appSettings.Value;
        this.encryptor = new Encryptor(/*this.appSettings.SecretKey!*/);
    }
}
```

Démo - Étape 4 : Ajouter la route Register

```
[HttpPost("register")] // créer un utilisateur
public async Task<ActionResult<RegisterResponseDTO>> Register([FromBody] RegisterRequestDTO request)
{
    // Seul les administrateurs peuvent créer des administrateurs...
    if (request.IsAdmin && User.FindFirstValue(ClaimTypes.Role) != Constants.RoleAdmin)
        return Unauthorized(new RegisterResponseDTO { IsSuccessful = false, ErrorMessage = "You can't create an administrator as a user." });

    // Vérifier si l'email de l'utilisateur existe déjà
    if (userRepository.GetByEmail(request.Email!) is not null)
        return BadRequest(new RegisterResponseDTO() { IsSuccessful = false, ErrorMessage = "Email already exist !" });

    // Créer l'utilisateur
    User user = new User()
    {
        Email = request.Email,
        Password = encryptor.EncryptPassword(request.Password!),
        IsAdmin = request.IsAdmin
    };

    // On l'ajoute
    User? userAjoute = userRepository.Add(user);

    if(userAjoute == null)
        return BadRequest(new RegisterResponseDTO { IsSuccessful = false, ErrorMessage = "Problème creation User" });

    return Ok(new RegisterResponseDTO() { IsSuccessful = true, User = user });
}
```

Démo - Étape 5 : Ajouter la route Login

```
[HttpPost("login")] // créer un JWT
public async Task<ActionResult<LoginResponseDTO>> Login([FromBody] LoginRequestDTO request)
{
    var user = userRepository.GetByEmail(request.Email!);

    if (user == null)
        return BadRequest(new LoginResponseDTO() { IsSuccessful = false,
            ErrorMessage = "Invalid Authentication !" });

    // Vérification du mot de passe
    var (verified, needsUpgrade) = encryptor.Check(user.Password!, request.Password!);

    if (!verified)
        return BadRequest(new LoginResponseDTO { IsSuccessful = false,
            ErrorMessage = "Invalid Authentication !" });

    // on devra potentiellement mettre à jour le mot de passe chiffré en bdd
    if (needsUpgrade)
    {
        user.Password = encryptor.EncryptPassword(request.Password!);
        userRepository.Edit(user);
    }

    // ternaire pour définir le role en utilisant les constantes
    string role = user.IsAdmin ? Constants.RoleAdmin : Constants.RoleUser;

    #region JWT
    #endregion

    return Ok(new LoginResponseDTO
    {
        IsSuccessful = true,
        Token = token,
        User = user
    });
}
```

```
#region JWT
// Claims = affirmation infalsifiables
// que l'on retrouvera dans le payload du jwt
var claims = new List<Claim>
{
    new Claim (ClaimTypes.Role, role),
    new Claim ("email", user.Email!),
    new Claim ("aimeLesChats", "true"),
    new Claim (ClaimTypes.NameIdentifier, user.Id.ToString()),
};

// récupérer la clé et préparer le chiffrement du token
var signingCredentials = new SigningCredentials(
    new SymmetricSecurityKey(
        Encoding.ASCII.GetBytes(appSettings.SecretKey!)),
        SecurityAlgorithms.HmacSha256);

//création de l'objet JWT
var jwt = new JwtSecurityToken(
    claims: claims,
    expires: DateTime.Now.AddDays(
        (double)appSettings.TokenExpirationDays!),
    signingCredentials: signingCredentials
    );

// conversion en chaine de caractère (token réel)
string token = new JwtSecurityTokenHandler().WriteToken(jwt);
#endregion
```

Démo - Étape 6 : Ajouter la route ValidateToken

```
// vérifier si un token est toujours valide
[HttpGet("validate")]
[Authorize] // vérifie que l'on est connecté
            // (token dans le header "Authorization: Bearer <token>")
public IActionResult ValidateToken()
{
    // Si cette action est atteinte, le token est valide
    return Ok(new { Message = "Token is valid." });
}
```

Démo - Étape 7 : Ajouter les Authorize

- `[Authorize]` : **Restreint l'accès** aux **actions** ou aux **contrôleurs** aux utilisateurs authentifiés uniquement.

```
[Authorize] // ou  avec Role [Authorize(Roles = "admin")]
public IActionResult SecurePage() => Ok();
```

- `[AllowAnonymous]` : Permet l'accès à une **action sans authentification**, même si `[Authorize]` est appliqué au contrôleur.

```
[AllowAnonymous]
public IActionResult Login() => Ok();
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Sécurité applicative et OWASP

Anatomie d'une faille applicative

- **Étapes clés** :
  - **Exploitation d'une vulnérabilité** (e.g., injection SQL).
  - **Escalade des privilèges** : accès à des données ou ressources non autorisées.
  - **Extraction ou modification des données**.
- **Contre-mesures** :
  - Validation des entrées.
  - Contrôles d'accès appropriés.

Principe de Zero Trust

- **Zero Trust** : Modèle de sécurité où **aucune entité n'est implicitement digne de confiance**, que ce soit à l'intérieur ou à l'extérieur du réseau.
- **Principe Clé** : "Ne jamais faire confiance, toujours vérifier";
  Chaque tentative d'accès doit être **authentifiée**, **autorisée**, et **validée**, peu importe sa provenance.
- **Avantage** : Réduit les risques de compromission, même si un attaquant réussit à accéder à une partie du système.
- Ce modèle est particulièrement adapté aux **environnements cloud** et aux **architectures distribuées** (ex: Architecture Microservices).

Introduction à la sécurité des applications web

- La **sécurité des applications web** est cruciale pour protéger les applications accessibles via Internet contre diverses menaces. Avec la **numérisation** croissante, ces applications deviennent des cibles pour les **cybercriminels**.
- **30 % des cyberattaques en 2023** ciblaient les applications web.
- **Veracode (2022)** a révélé que **60 % des applications** avaient des vulnérabilités critiques.
- Les **attaques par injection SQL** touchent environ **50 % des applications web** mal protégées.

Incidents notables de sécurité

1. **Yahoo (2013)** :
   - Une vulnérabilité a permis l'accès aux comptes de **3 milliards d'utilisateurs**. Cela souligne l'importance de la **gestion des sessions** et de l'**authentification**.
2. **Equifax (2017)** :
   - Une faille dans **Apache Struts** a exposé les données de **147 millions de personnes**. La mise à jour des serveurs était insuffisante.

Impact des failles de sécurité

Les applications web sont souvent le premier point d'entrée pour les entreprises. Les conséquences d'une faille incluent :

- **Données sensibles** : Des informations comme les **données bancaires** peuvent être compromises.
- **Réputation** : Les attaques peuvent nuire à la confiance des utilisateurs.
- **Compliance (conformité)** : Des réglementations comme le **RGPD** imposent des normes strictes.

Conséquences des cyberattaques

- **Perte de données** : Les informations peuvent être volées et revendues sur le **dark web**.
- **Pertes financières** : Les entreprises peuvent subir des pénalités financières à travers des sanctions légales ou en raison des coûts de récupération et de réparation..
- **Interruption de service** : Les attaques DDoS peuvent rendre une application inaccessible.

Les acteurs de la menace

- **Hackers malveillants** : Motivés par des gains financiers, ils ciblent les failles des applications pour
  voler des données ou extorquer de l'argent.
- **Hacktivistes** : Ils mènent des attaques pour des raisons idéologiques ou politiques.
- **Insiders** : Des employés ou des partenaires qui ont accès aux systèmes internes peuvent exploiter les
  vulnérabilités à des fins malveillantes.

Solutions pour améliorer la sécurité

- **Automatisation des tests de sécurité** : Les entreprises utilisent de plus en plus des outils automatisés comme **SAST** (Static Application Security Testing) et **DAST** (Dynamic Application Security Testing) pour identifier les failles avant le déploiement des applications.
- **DevSecOps** : Intégration de la sécurité dès le début du développement et tout au long de la chaine.
- **Intelligence artificielle et machine learning**: Utilisées pour détecter et réagir plus rapidement aux menaces.

Les grandes familles de vulnérabilités

- **Injection** :
  - Commande OS, SQL, LDAP, etc.
- **Attaques XSS (Cross-Site Scripting)** :
  - **Reflected** ou **Stored XSS**.
- **Attaques CSRF (Cross-Site Request Forgery)** :
  - Exploite la confiance entre un utilisateur authentifié et le serveur.

Les CVE et CWE

- **CVE (Common Vulnerabilities and Exposures)** :
  - Base de données publique des vulnérabilités identifiées.
  - Utilisée pour identifier les failles dans les logiciels.
- **CWE (Common Weakness Enumeration)** :
  - Classification des **faiblesses connues** dans le code.
  - Exemple : **CWE-89** pour l'injection SQL.

Le scoring CVSS

- **CVSS (Common Vulnerability Scoring System)** :
  - Méthode de calcul du **niveau de gravité** des vulnérabilités.
  - Score de 0 (faible) à 10 (critique).
  - Prend en compte les **vecteurs d'attaque**, la **complexité**, et l'**impact**.
- Selon la politique d'entreprise on peut considérer qu'**une application doit être totalement retirée de production** et traitée si ce **score** est **trop élevé**.

Le cas des librairies

- **Risques** :
  - Bibliothèques contenant des failles de sécurité.
  - Dépendance à des versions obsolètes et vulnérables.
- **Recommandations** :
  - Utiliser des **versions à jour**.
  - Effectuer des audits réguliers des **dépendances** (avec des applications style OWASP Check Dependencies).

Le cas des API

- **Risques liés aux API** :
  - Exposition des données sensibles.
  - Mauvaise gestion des authentifications et autorisations.
- **Contre-mesures** :
  - **Authentification robuste** (OAuth 2.0, JWT).
  - **Filtrer et valider** toutes les entrées et sorties.

Se protéger de la décompilation

- **Risques** : le code C# peut être facilement décompilé.
- **Solutions** :
  - Utiliser des **outils d'obfuscation** (ex : Dotfuscator).
  - Chiffrer les sections critiques du code.

L'obfuscation du code

- **Principe** : rendre le code difficile à comprendre pour protéger la propriété intellectuelle.
- **Outils** : Dotfuscator, ConfuserEx.
- **Attention** : ne remplace pas une stratégie de sécurité complète.

Tests et validations

- **Outils** :
  - **Postman** : tester les API avec différentes requêtes.
  - **Wireshark** : capturer et analyser le trafic réseau.
  - **Proxy** : interception et modification des requêtes HTTP/HTTPS.

DAST et SAST

- **DAST (Dynamic Application Security Testing)** :
  - Teste une application en cours d'exécution.
  - Simule des attaques pour identifier les failles.
  - Exemple : OWASP Zap
- **SAST (Static Application Security Testing)** :
  - Analyse du code source sans l'exécuter.
  - Détecte les failles potentielles dès le développement.
  - Exemple : SonarQube

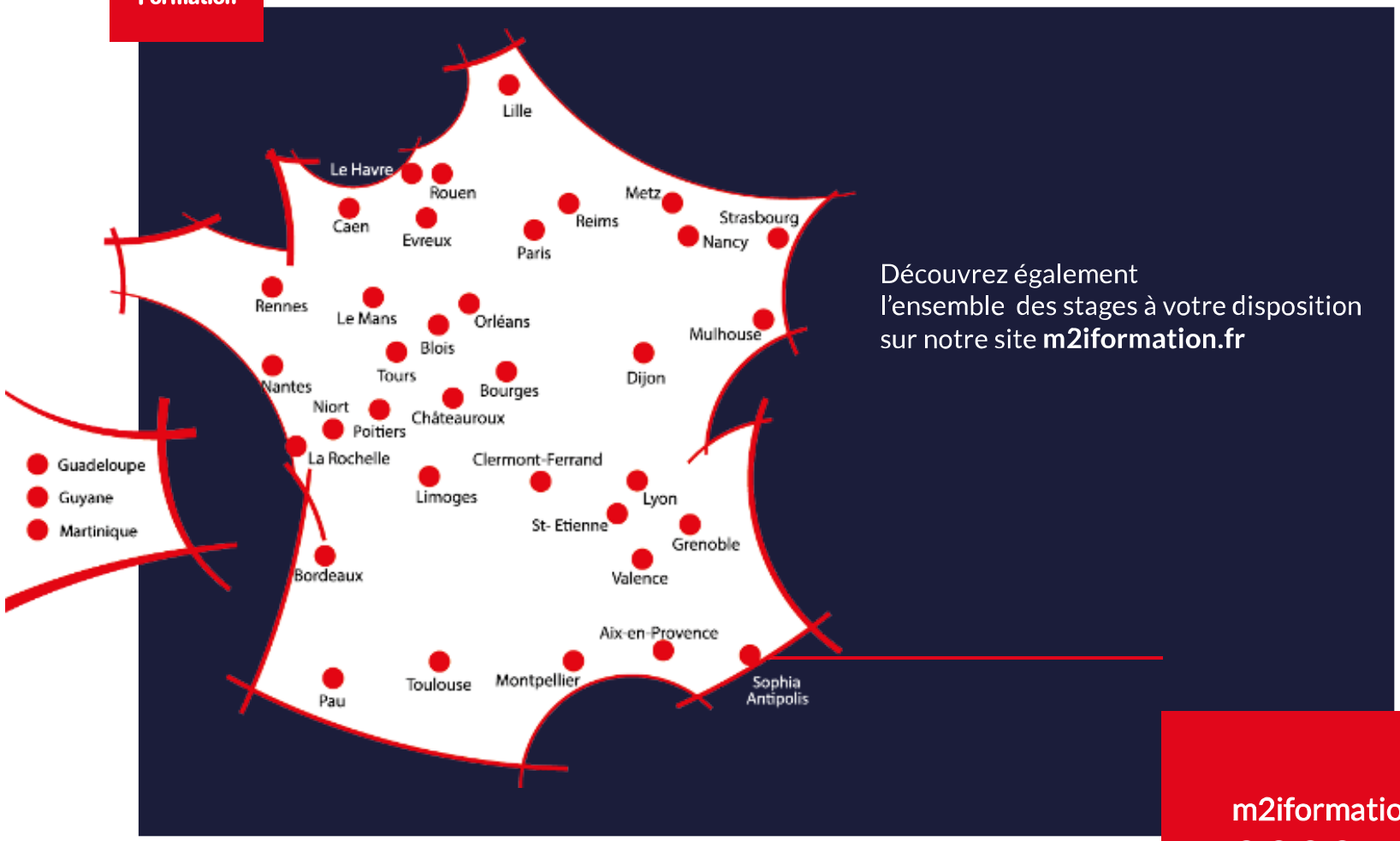
Filtrer les échanges

- **WAF (Web Application Firewall)** : Filtre et bloque les requêtes malveillantes avant qu'elles n'atteignent l'application.
  - Exemples : AWS WAF, Azure WAF, Cloudflare WAF
- **IPS** (Intrusion Prevention System) : bloque activement les attaques.
  - Exemples : Cisco Firepower, Snort
- **IDS** (Intrusion Detection System) : alerte en cas de tentative d'intrusion.
  - Exemples : Suricata, IBM QRadar, Snort

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Suite et fin dans le support dédié à OWASP ...



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr