Royaume du Maroc

Ministère de l'Éducation Nationale, de la Formation Professionnelle,

De l'Enseignement Supérieur et de la Recherche Scientifique

ENSIAS

جامعة محمد الخامس بالرباط
Université Mohammed V de Rabat

Sécurité des applications Web avec JWT

Projet fédérateur

Département : Réseaux de communication

Filière : Sécurité des Systèmes d'Information

Réalisé par :
ADNI Fahd
KHALLOUF Faycal

Professeur encadrant :
Mme AJHOUN Rachida

Examinatrice :
Mme EL BAKKALI Hanane

Année universitaire 2021/2022

Remerciements

Au terme de ce travail, nous tenons à exprimer notre profonde gratitude à notre cher encadrant Mme.**AJHOUN Rachida** pour son suivi et pour son énorme soutien, qu'il n'a cessé de nous prodiguer tout au long de la période du projet.

Nous adressons aussi notre vif remerciement à Mme.**EL BAKKALI Hanane** notre examinatrice pour avoir bien voulu examiner et juger ce travail.

Nous ne laisserons pas cette occasion passer, sans remercier tous les enseignants et le personnel de **l'Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes**.

Enfin, notre remerciement à tous ceux qui ont contribué de prés ou de loin au bon déroulement de ce projet.

Résumé

Un processus d'authentification consiste à prouver l'identité d'un utilisateur lorsqu'il entre dans un système. L'authentification par jeton est un type d'authentification qui est sans état. Cela signifie que lorsque le processus d'authentification est exécuté, il n'y a absolument aucune information sur l'utilisateur car l'utilisation de jetons dans chaque requête est faite du client au serveur. Le jeton Web JSON (Java Script Object Notation) est une technique d'authentification qui fournit un moyen ouvert et sûr de représenter les réclamations entre deux parties, signé de manière cryptographique, et conçu pour ne pas être falsifié. Cependant, il faut prouver que cette technique est sûre et non vulnérable. L'objectif de cette étude est de réaliser des tests de pénétration de la sécurité du concept et du stockage de JSON Web Token (JWT) en utilisant différentes techniques. Des scénarios d'exécution de ces techniques ont été préparés dans le cadre de l'expérience. L'architecture du système et les outils à utiliser sont préparés avant la réalisation de l'expérience. Les résultats expérimentaux de cette étude montrent que les JSON Web Token ne sont pas aussi sécurisés qu'il n'y paraît. Les techniques qui ont été essayées dans la recherche ont réussi à utiliser les jetons JWT stockés dans les cookies pour envoyer de fausses demandes et. De plus, ces techniques ont réussi à voler les jetons JWT stockés dans le localStorage du navigateur. Finalement, le compte de la victime a été utilisé, et la ressource a été prise en charge.

Abstract

An authentication process is an act of proving the identity of a user when entering a system. Token-based authentication is a type of authentication that is stateless. This means that when the authentication process is carried out, there is absolutely no information about the user because the use of tokens in every request is made from the client to the server. Java Script Object Notation (JSON) Web Token is an authentication technique that provides an open and secure way to represent claims between two parties, cryptographically signed, which is designed not to be forged. However, this needs to be proven safe and not vulnerable. The purpose of this study is to conduct penetration testing of the security of JSON Web Token (JWT) concept and storage using different techniques. Scenarios for performing these techniques were prepared in the experiment. The system architecture and tools to be used are prepared before the experiment is carried out. The experimental results in this study show that JSON Web Token are not as secured as they might seem. The techniques that was tried in the research has succeeded in utilizing JWT tokens stored in cookies to send faked requests and. Also, these techniques has succeeded in stealing JWT tokens stored in localStorage of the browser. Eventually, the victim's account was used, and the resource was taken over.

Table des matières

Table des figures

Liste des abréviations

— API : Application Programming Interface
— URL : Uniform Resource Locator
— HTTP : HyperText Transfert Protocol
— HTTPS : L'HyperText Transfert Protocol Secure
— HMAC : Hash-based Message Authentification Code
— JSON : JavaScript Object Notation
— JWA : JSON Web Algorithms
— JWT : JSON Web Token
— JWS : JSON Web Signature
— XSS : Cross Site Scripting
— CSRF : Cross Site Request Forgery

Introduction générale

L'authentification est une problématique récurrente du développement d'applications web et mobile. Dans un monde où les API sont consommées par toutes sortes de clients (navigateurs, applications natives mobile, ...), il est important de pouvoir offrir une solution commune performante et sécurisée.

Historiquement, pour remplir ce rôle, les cookies sont utilisés et retournés par le serveur au client suite à une authentification réussie. Ces mêmes cookies accompagnent chaque requête du client afin de l'identifier. Les sessions sont maintenues côté serveur (stockées en mémoire, sur le file-system ou encore en base de données) avec par conséquent une problématique de charge à gérer.

D'autres solutions existent telles que HTTP Basic Authentication (les identifiants de l'utilisateur doivent cependant transiter dans chaque requête). Avec les applications mobiles natives, la gestion des cookies est relativement fastidieuse et les problématiques de Cross-origin resource sharing (CORS) récurrentes. Ajoutez à cela qu'une API REST se doit d'être stateless (pas de session côté serveur).

Les web tokens sont une alternative intéressante. Le principe est simple, après s'être authentifié, le serveur génère un hash de plus de 100 caractères qui servira de signature pendant une certaine durée. Ce token va ensuite transiter dans chaque requête entre le client et le serveur. Pas de cookie ni de session serveur. De plus, cette solution est applicable pour tout type de plateforme et peut permettre à une API d'être utilisée par des applications natives, des applications web...

JWT ou JSON Web Token est un standard ouvert désormais industriel (RFC 7519) qui tend à répondre à cette problématique. De plus en plus utilisé il dispose d'une communauté active et de librairies pour plus d'une vingtaine de langages. Pour comprendre son fonctionnement, prenons l'exemple d'une application mobile sur laquelle l'utilisateur doit s'authentifier afin d'accéder à des ressources exposées par un ensemble de web-services (API). La première étape consiste donc pour l'utilisateur à envoyer son couple login/mot de passe via un service d'authentification. Il n'est pas envisageable de stocker ce couple en clair sur le device de l'utilisateur et d'envoyer celui-ci à chaque échange suivant pour des raisons de sécurité. Nous avons donc besoin de stocker un artéfact qui servira à authentifier l'utilisateur auprès de l'API.

CHAPITRE 1

Introduction

1.1 Introduction

JSON Web Token, ou JWT ("jot") pour faire court, est une norme permettant de transmettre des demandes en toute sécurité dans des environnements à espace restreint. Il a trouvé sa place dans tous les principaux frameworks web. La simplicité, la compacité et la convivialité sont les principales caractéristiques de son architecture. Bien que des systèmes beaucoup plus complexes soient encore utilisés, les JWT ont un large éventail d'applications. Dans ce petit manuel, nous couvrirons les aspects les plus importants de l'architecture des JWT, y compris leur représentation binaire et les algorithmes utilisés pour les construire, tout en examinant comment ils sont couramment utilisés dans l'industrie.

1.2 Dénominations :

— API : L'Application Programming Interface, une interface de programmation applicative décrivant les méthodes et paramètres acceptés par une application
— Claim : information sur un utilisateur détenue par une entité
— HTTP : L'HyperText Transfert Protocol, un protocole de communication client serveur développé pour Internet
— HTTPS : L'HyperText Transfert Protocol Secure, le protocole HTTP sécurisé au travers de tunnels TLS
— HTML : L'HyperText Markup Language, un langage de balisage utilisé par les navigateurs pour représenter les pages d'un site Web
— HMAC : Hash-based Message Authentification Code, un code d'authentification reposant sur une empreinte calculée à l'aide d'une fonction de hachage et d'une clé
— JSON : JavaScript Object Notation, un format de données textuel permettant de représenter une information structurée.
— JWT : JSON Web Token, standard défini par la RFC 7519[7] permettant d'échanger de l'information sur une entité au format JSON.
— JWS : JSON Web Signature, standard défini par la RFC 7515[6] permettant de protéger un JWT en intégrité par un HMAC ou une signature.
— OAuth2.0 : cadre, défini par la RFC 6749[4], décrivant des mécanismes pour obtenir l'autorisation d'accéder à des ressources protégées
— TLS : Transport Layer Security, un protocole de sécurisation des communications réseaux
— URL : L'Uniform Resource Locator, l'adresse Web d'une ressource (HTML, API, etc.)
— XSS : Cross Site Scripting, une attaque consistant à injecter du code malveillant afin qu'il soit exécuté par le navigateur d'un utilisateur
— CSRF : Cross Site Request Forgery, une attaque permettant de faire exécuter à un utilisateur une action à son insu lorsqu'il est connecté à une application

1.3 Qu'est-ce qu'un JSON Web Token ?

Un jeton Web JSON,standard défini par la RFC 7519[7] ressemble à ceci :
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
Bien que cela ressemble à du jargon, il s'agit en fait d'une représentation très compacte et imprimable d'une série de revendications, accompagnée d'une signature pour en vérifier l'authenticité.

Les revendications sont des définitions ou des affirmations faites à propos d'une certaine partie ou d'un certain objet. Certaines de ces revendications et leur signification sont définies dans le cadre de la spécification JWT. D'autres sont définies par l'utilisateur. La magie des JWT réside dans le fait qu'ils normalisent certaines revendications qui sont utiles dans le contexte

```
{
  "alg": "HS256",
  "typ": "JWT"
}

{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

de certaines opérations communes. Par exemple, l'une de ces opérations communes consiste à établir l'identité d'une certaine partie. Ainsi, l'une des revendications standard que l'on trouve dans les JWT est la revendication sub (de "subject").

Un autre aspect essentiel des JWT est la possibilité de les signer, à l'aide des signatures Web JSON (JWS), et/ou de les crypter, à l'aide du cryptage Web JSON (JWE). Avec JWS et JWE, les JWT constituent une solution puissante et sûre à de nombreux problèmes différents.

1.4 Quel problème résout-il ?

Bien que l'objectif principal des JWT soit de transférer des demandes entre deux parties, l'aspect le plus important est sans doute l'effort de normalisation sous la forme d'un modèle simple et éventuellement validé. l'aspect le plus important est l'effort de normalisation sous la forme d'un format de conteneur simple, éventuellement validé et/ou crypté, d'un format de conteneur simple. Des solutions ad hoc à ce même problème ont été mises en œuvre à la fois privées et publiques dans le passé. Des normes plus anciennes[8] permettant d'établir des affirmations sur certaines parties sont également disponibles. Ce que JWT apporte, c'est un format de conteneur simple, utile et standard. format.

Bien que la définition donnée soit un peu abstraite jusqu'à présent, il n'est pas difficile d'imaginer comment ils peuvent être utilisés : les systèmes de connexion (bien que d'autres utilisations soient possibles).

Certaines de ces applications comprennent :

- Authentication
- Authorization
- Federated identity
- Client-side sessions (“stateless” sessions)
- Client-side secrets

1.5 L'histoire de JWT :

Le groupe JSON Object Signing and Encryption (JOSE) a été créé en 2011. L'objectif du groupe était de "normaliser le mécanisme de protection de l'intégrité (signature et MAC) et de chiffrement ainsi que le format des clés et des identifiants d'algorithme afin de soutenir l'interopérabilité des services de sécurité pour les protocoles qui utilisent JSON". En 2013, une série d'ébauches, dont un livre de cuisine avec différents exemples d'utilisation des idées produites par le groupe, étaient disponibles. Ces ébauches deviendront plus tard les RFCs JWT, JWS, JWE, JWK et JWA. Depuis l'année 2016, ces RFC sont dans le processus de suivi des normes et aucun errata n'y a été trouvé. Le groupe est actuellement inactif. Les principaux auteurs de ces spécifications sont Mike Jones, Nat Sakimura, John Bradley et Joe Hildebrand.

1.6 Pourquoi JWT ?

L'authentification par jeton pour les API Web est le processus d'authentification des utilisateurs ou des processus pour les applications Cloud. L'application de l'utilisateur envoie une demande au service d'authentification, qui confirme l'identité de l'utilisateur et émet un jeton. L'utilisateur peut alors accéder à l'application.

1.6.1 Authentification avec état :

L'authentification avec état est couramment utilisée dans de nombreuses applications, en particulier pour les applications qui ne nécessitent pas trop d'évolutivité.

Une session avec état est créée du côté du backend, et l'Id de référence de la session correspondante est envoyée au client. Chaque fois que le client fait une demande au serveur, le serveur localise la mémoire de session en utilisant l'Id de référence du client et trouve les informations d'authentification.

Dans ce modèle (figure 1.1), on peut facilement imaginer que si la mémoire de la session est supprimée du côté du backend, l'Id de référence de la session, que le client détient, n'a plus aucun sens.

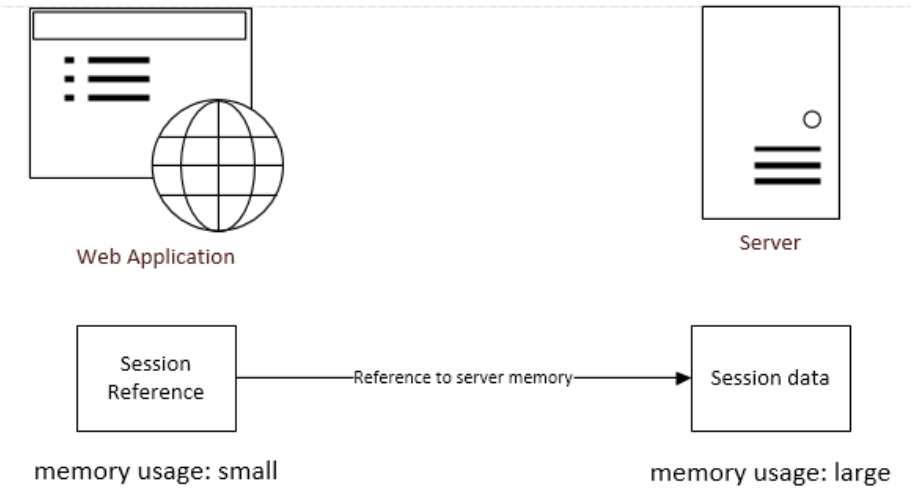

FIGURE 1.1 – Modèle d'authentification avec état

1.6.2 Authentification sans état :

L'authentification sans état est utilisée pour résoudre les inconvénients de l'authentification avec état. Elles sont très différentes et sont utilisées dans des scénarios différents.

L'authentification sans état stocke les données de session de l'utilisateur du côté du client (navigateur).Les données sont signées par la clé du fournisseur d'identité (IdP) pour garantir l'intégrité et l'autorité des données de session.

Dans ce modèle(figure 1.2), la session de l'utilisateur étant stockée côté client, le serveur ne peut que vérifier sa validité en contrôlant la correspondance entre la charge utile et la signature.

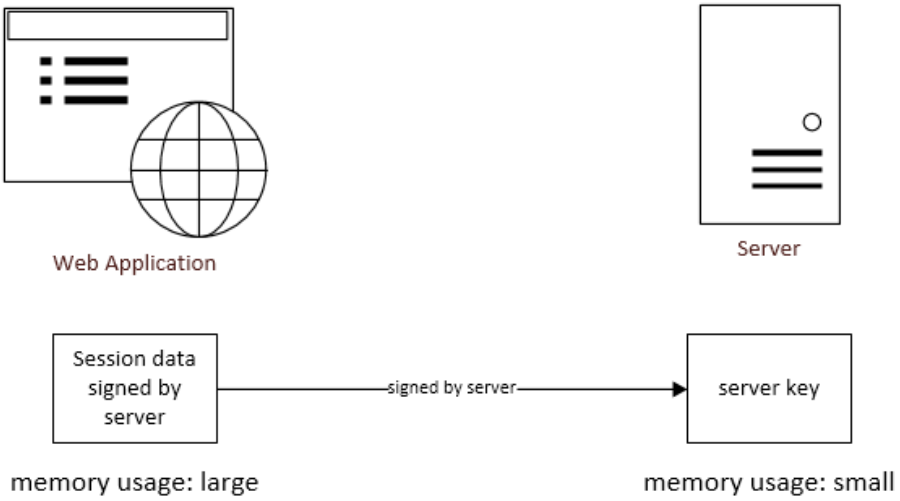

FIGURE 1.2 – Modèle d'authentification sans état

1.6.3 JWT et session Id token :

Avant l'apparition du JWT, la plupart des solutions se reposer sur l'utilisation des jetons générés et stockés pour chaque utilisateur, ce qui nécessitait l'accès au base de données pour la récupération des informations d'utilisateur à chaque requête en résultant en une authentification avec état.

Avantages de JWT :

- Faible usage du serveur : Le grand nombre de données de session n'est pas stocké sur le serveur. Nous pouvons stocker plus de propriétés de l'utilisateur dans les données de session côté client afin de réduire le nombre d'accès à la base de données sans se soucier de l'encombrement de la mémoire sur le serveur.
- Facilité de mise à l'échelle : Comme les données de session sont stockées côté client, le serveur dorsal vers lequel la demande est acheminée n'a pas d'importance. Tant que tous les serveurs dorsaux partagent la même clé privée, tous les serveurs ont la même capacité à vérifier la validité de la session.
- Facilité de l'intégration avec des applications tierces : Dans les protocoles d'authentification unique (SSO), les applications tierces et le fournisseur d'identité (IdP) doivent pouvoir communiquer entre eux via les agents utilisateurs (navigateur). Pendant le processus de liaison de compte entre les applications tierces et l'IdP, l'IdP envoie un message signé au navigateur, et le navigateur redirige ce message vers les applications tierces. En utilisant un secret partagé préconfiguré, les applications tierces peuvent déterminer elles-mêmes si la liaison de compte (single sign-on) est valide.

CHAPITRE 2

Applications réels

2.1 Introduction :

Avant de plonger dans la structure et la construction d'un JWT, nous allons examiner plusieurs applications pratiques. Ce chapitre vous donnera une idée de la complexité (ou de la simplicité) des solutions basées sur JWT utilisées dans l'industrie aujourd'hui. Tout le code est disponible dans des dépôts publics1 pour votre commodité. Sachez que les démonstrations suivantes ne sont pas destinées à être utilisées en production. Les cas de test, la journalisation et les meilleures pratiques de sécurité sont tous essentiels pour un code prêt à être utilisé en production. Ces exemples sont uniquement destinés à des fins éducatives et restent donc simples et directs. l'essentiel.

2.2 Sessions côté-client/sans-état :

Les sessions dites "sans état" ne sont en fait rien d'autre que des données côté client. L'aspect essentiel de cette application réside dans l'utilisation de la signature et éventuellement du cryptage pour authentifier et protéger le contenu de la session. le contenu de la session. Les données côté client sont susceptibles d'être altérées. En tant que telles, elles doivent être traitées avec avec le plus grand soin par le backend.

Les JWT, en vertu de JWS et JWE, peuvent fournir différents types de signatures et de cryptage. Les signatures sont utiles pour valider les données contre la falsification. Le cryptage est utile pour protéger les données contre la lecture par des tiers.

La plupart du temps, les sessions doivent simplement être signées. En d'autres termes, il n'y a pas de problème de sécurité ou de confidentialité lorsque les données stockées dans ces sessions sont lues par des tiers. lorsque les données qui y sont stockées sont lues par des tiers. Un exemple courant d'une revendication qui peut généralement être lue en toute sécurité par des tiers est la sous-créance ("sujet"). La revendication objet identifie généralement identifie l'une des parties à l'autre (pensez aux identifiants d'utilisateur ou aux courriels). Il n'est pas nécessaire que cette revendication soit unique. En d'autres termes, des revendications supplémentaires peuvent être nécessaires pour identifier de manière unique un utilisateur. Cette décision est laissée à l'appréciation des utilisateurs.

Une réclamation qui pourrait ne pas être laissée en suspens de manière appropriée pourrait être une réclamation "articles" représentant le panier d'un utilisateur. Ce panier peut être rempli d'articles que l'utilisateur est sur le point d'acheter et qui sont donc associés à sa session. Un tiers (un script côté client) pourrait être en mesure de récolter ces articles s'ils sont stockés dans un JWT non crypté, ce qui pourrait poser des problèmes de confidentialité.

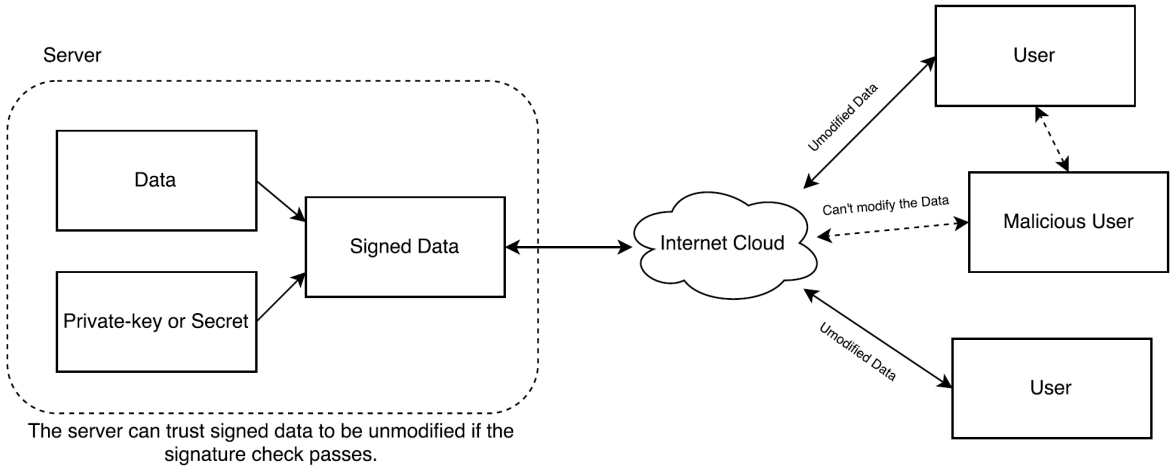

FIGURE 2.1 – Données signées du côté client

2.3 Considérations de sécurité :

2.3.1 Décapage de la signature :

Une méthode courante pour attaquer un JWT signé consiste simplement à supprimer la signature. Les JWT signés sont construits à partir de trois parties différentes : l'en-tête, la charge utile et la signature. Ces trois parties sont codées séparément. Il est donc possible de supprimer la signature, puis de modifier l'en-tête pour affirmer que le JWT n'est pas signé. Une utilisation imprudente de certaines bibliothèques de validation JWT peut faire en sorte que des jetons non signés soient considérés comme des jetons valides, ce qui peut permettre à un attaquant de modifier la charge utile à sa guise. Ce problème est facilement résolu en s'assurant que l'application qui effectue la validation ne considère pas les JWT non signés comme valides.

FIGURE 2.2 – Décapage de la signature

2.3.2 Falsification de requête intersite (CSRF) :

Les attaques par falsification de requêtes intersites tentent d'exécuter des requêtes contre des sites où l'utilisateur est connecté en trompant le navigateur de l'utilisateur pour qu'il envoie une requête provenant d'un autre site.Pour ce faire, un site (ou un élément) spécialement conçu doit contenir l'URL de la cible.Un exemple courant est une balise <img> intégrée dans une page malveillante dont la source pointe vers la cible de l'attaque.par exemple :

```
<!-- This is embedded in another domain's site -->
<img src="http://target.site.com/add-user?user=name&grant=admin">
```

La balise <img> ci-dessus enverra une demande à target.site.com chaque fois que la page qui la contient sera chargée. Si l'utilisateur s'est précédemment connecté à target.site.com et que le site a utilisé un cookie pour maintenir la session active, ce cookie sera également envoyé.Si le site cible ne met en œuvre aucune technique d'atténuation CSRF, la demande sera traitée comme une demande valide au nom de l'utilisateur. Les JWT, comme toute autre donnée côté client, peuvent être stockés sous forme de cookies.

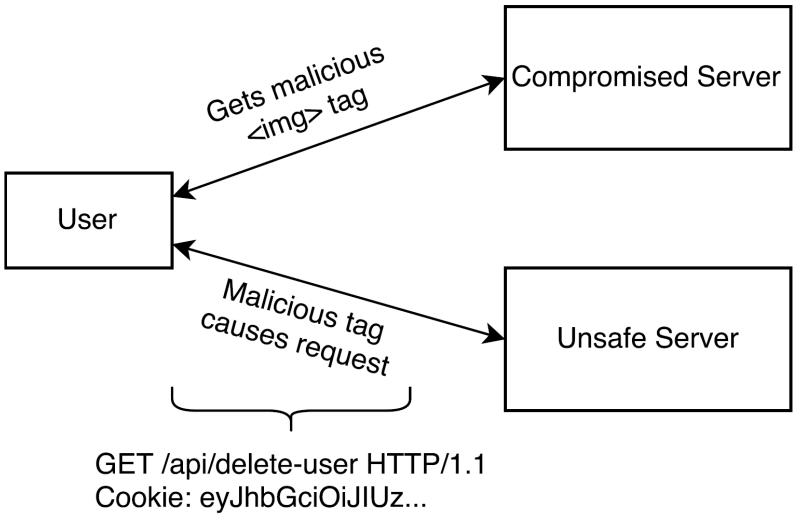

FIGURE 2.3 – Falsification de requête intersite(CSRF)

Les JWT à courte durée de vie peuvent être utiles dans ce cas. Les techniques courantes d'atténuation des attaques CSRF comprennent des en-têtes spéciaux qui ne sont ajoutés aux demandes que lorsqu'elles sont exécutées depuis la bonne origine, des cookies par session et des jetons par demande. Si les JWT (et les données de session) ne sont pas stockés sous forme de cookies, les attaques CSRF ne sont pas possibles. Les attaques de type "cross-site scripting" restent toutefois possibles.

2.3.3 Cross-Site Scripting (XSS) :

Les attaques par cross-site scripting (XSS) tentent d'injecter du JavaScript dans des sites de confiance. Le JavaScript injecté peut alors voler les jetons d'accès des cookies et du stockage local. Si un jeton d'accès est divulgué avant son expiration, un utilisateur malveillant peut l'utiliser pour accéder à des ressources protégées. Les attaques XSS courantes sont généralement dues à une mauvaise validation des données transmises au backend (de manière similaire aux attaques par injection SQL).

Un exemple d'attaque XSS pourrait être lié à la section des commentaires d'un site public. Chaque fois qu'un utilisateur ajoute un commentaire, celui-ci est stocké par le backend et affiché aux utilisateurs qui chargent la section des commentaires. Si le backend ne nettoie pas les commentaires, un utilisateur malveillant pourrait écrire un commentaire de telle manière qu'il pourrait être interprété par le navigateur comme une balise <script>. Ainsi, un utilisateur malveillant pourrait insérer un code JavaScript arbitraire et l'exécuter dans le navigateur de chaque utilisateur, volant ainsi les informations d'identification stockées dans les cookies et dans le stockage local.

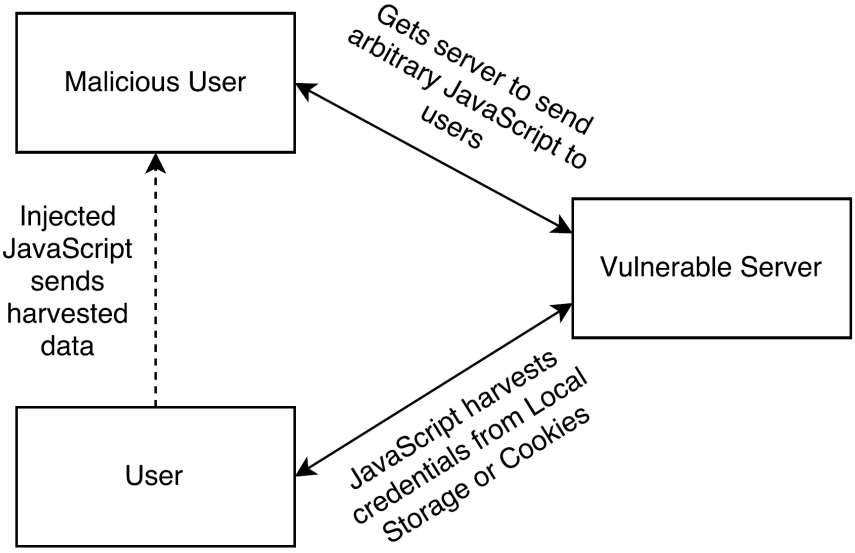

FIGURE 2.4 – Cross Site Scripting persistant

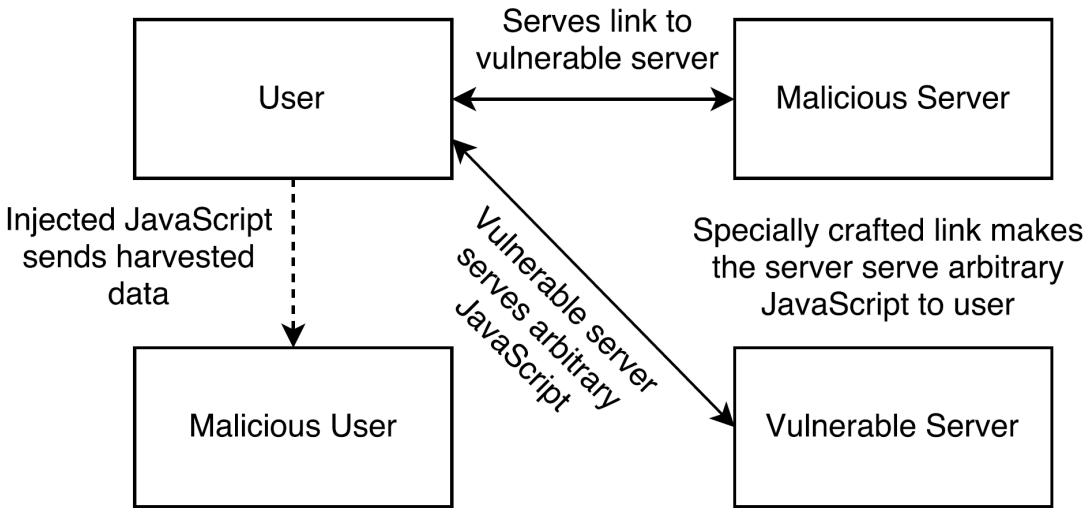

FIGURE 2.5 – Cross Site Scripting réfléchi

Les techniques d'atténuation reposent sur une validation adéquate de toutes les données transmises au backend. En particulier, toutes les données reçues des clients doivent toujours être nettoyées. Si des cookies sont utilisés, il est possible de les protéger contre l'accès par JavaScript en activant l'indicateur HttpOnly. L'indicateur HttpOnly, bien qu'utile, ne protège pas le cookie contre les attaques CSRF.

2.4 Les sessions côté-client sont-elles utiles ?

Toute approche présente des avantages et des inconvénients, et les sessions côté client ne font pas exception. Certaines applications applications peuvent nécessiter de grosses sessions. L'envoi de cet état dans les deux sens pour chaque requête (ou groupe de requêtes) peut facilement annuler les avantages de la réduction des conversations dans le backend. de requêtes) peut facilement contrecarrer les avantages de la réduction des conversations dans le backend. Un certain équilibre entre les données côté client et les consultations de bases de données dans le backend est nécessaire. Cela dépend du modèle de données de votre application. Certaines

applications ne se prêtent pas bien aux sessions côté client. D'autres peuvent dépendre entièrement des données côté client. Le dernier mot sur cette question vous appartient ! Exécutez des benchmarks, étudiez les avantages de conserver certains états côté client. Les JWT sont-ils trop volumineux ? Est-ce que Cela a-t-il un impact sur la bande passante ? Cette bande passante supplémentaire annule-t-elle la réduction de la latence dans le le backend ? Les petites demandes peuvent-elles être regroupées en une seule demande plus importante ? Ces requêtes nécessitent-elles toujours d'importantes consultations de la base de données ? La réponse à ces questions vous aidera à choisir la bonne approche.

2.5 Identité fédérée :

Les systèmes d'identité fédérés permettent à différentes parties, éventuellement non liées, de partager des services d'authentification et d'autorisation avec d'autres parties. En d'autres termes, l'identité d'un utilisateur est centralisée. Il existe plusieurs solutions pour la gestion des identités fédérées : SAML[8] et OpenID Connect[3] sont deux des solutions les plus courantes. Certaines entreprises fournissent des produits spécialisés qui centralisent l'authentification et l'autorisation. Celles-ci peuvent mettre en œuvre l'une des normes mentionnées ci-dessus ou utiliser quelque chose de complètement différent. Certaines de ces entreprises utilisent les JWTs à cette fin.

L'utilisation des JWT pour l'authentification et l'autorisation centralisées varie d'une entreprise à l'autre, mais le flux essentiel du processus d'autorisation est le suivant :

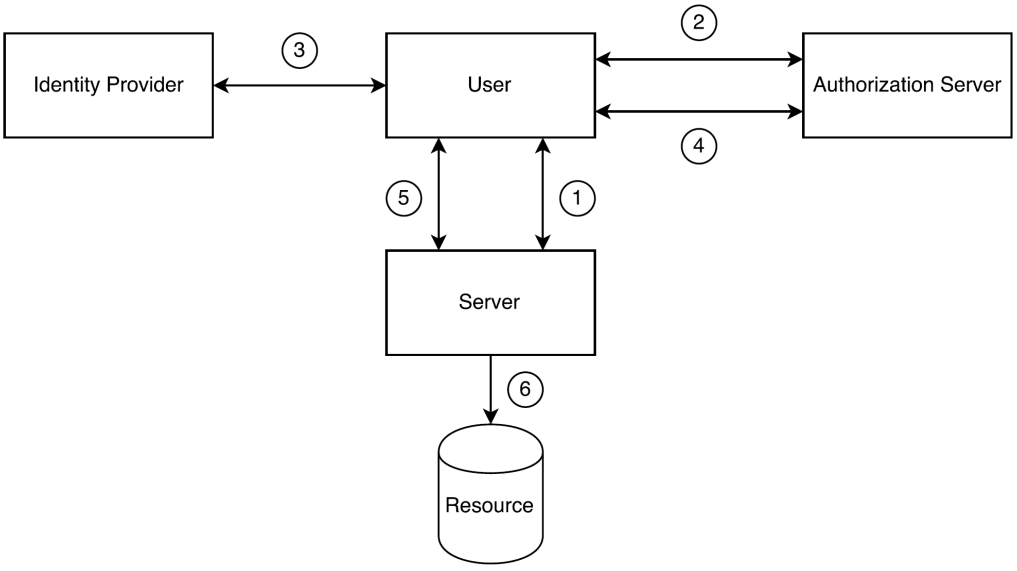

FIGURE 2.6 – Flux commun d'identités fédérées

1- L'utilisateur tente d'accéder à une ressource contrôlée par un serveur.
2- L'utilisateur ne dispose pas des informations d'identification appropriées pour accéder à la ressource, le serveur le redirige donc vers le serveur d'autorisation. Le serveur d'autorisation est configuré pour permettre aux utilisateurs de se connecter en utilisant les informations d'identification gérées par un fournisseur d'identité.
3- L'utilisateur est redirigé par le serveur d'autorisation vers l'écran de connexion du fournisseur de l'identité.
4- L'utilisateur se connecte avec succès et est redirigé vers le serveur d'autorisation. Le serveur d'autorisation utilise les informations d'identification fournies par le fournisseur d'identité pour accéder aux informations d'identification requises par le serveur de ressources.

5- L'utilisateur est redirigé vers le serveur de ressources par le serveur d'autorisation. La demande a maintenant les informations d'identification correctes requises pour accéder à la ressource.

6- L'utilisateur obtient l'accès à la ressource avec succès.

Toutes les données transmises d'un serveur à l'autre passent par l'utilisateur en étant intégrées dans les demandes de redirection (généralement dans l'URL). La sécurité du transport (TLS) et des données est donc essentielle.

Les informations d'identification renvoyées par le serveur d'autorisation à l'utilisateur peuvent être codées sous la forme d'un JWT. Si le serveur d'autorisation permet les connexions par l'intermédiaire d'un fournisseur d'identité (comme c'est le cas dans cet exemple), on peut dire que le serveur d'autorisation fournit une interface unifiée et des données unifiées (le JWT) à l'utilisateur.

Dans notre exemple, plus loin dans cette section, nous utiliserons Auth0 comme serveur d'autorisation et traiterons les connexions via Twitter, Facebook et une base de données d'utilisateurs ordinaire.

2.6 Jetons d'accès et de rafraîchissement :

Les jetons d'accès et de rafraîchissement sont deux types de jetons que vous verrez souvent lorsque vous analyserez différentes solutions d'identité fédérée. Nous allons brièvement expliquer ce qu'ils sont et comment ils aident dans le contexte de l'authentification et de l'autorisation.

Ces deux concepts sont généralement mis en œuvre dans le contexte de la spécification OAuth2. La spécification OAuth2[4] définit une série d'étapes nécessaires pour fournir un accès aux ressources en séparant l'accès de la propriété (en d'autres termes, elle permet à plusieurs parties ayant des niveaux d'accès différents d'accéder à la même ressource). Plusieurs parties de ces étapes sont définies par la mise en œuvre. En d'autres termes, des implémentations concurrentes d'OAuth2 peuvent ne pas être interopérables. Par exemple, le format binaire réel des jetons n'est pas spécifié. Leur objectif et leur fonctionnalité le sont :

Les jetons d'accès sont des jetons qui permettent à ceux qui les possèdent d'accéder à des ressources protégées. Ces jetons ont généralement une durée de vie courte et peuvent être assortis d'une date d'expiration. Ils peuvent également porter ou être associés à des informations supplémentaires (par exemple, un jeton d'accès peut porter l'adresse IP à partir de laquelle les demandes sont autorisées). Ces données supplémentaires sont définies par l'implémentation.

Les jetons de rafraîchissement permettent aux clients de demander de nouveaux jetons d'accès. Par exemple, après l'expiration d'un jeton d'accès, un client peut demander un nouveau jeton d'accès au serveur d'autorisation. Pour que cette demande soit satisfaite, un jeton de rafraîchissement est nécessaire. Contrairement aux jetons d'accès, les jetons de rafraîchissement ont généralement une longue durée de vie.

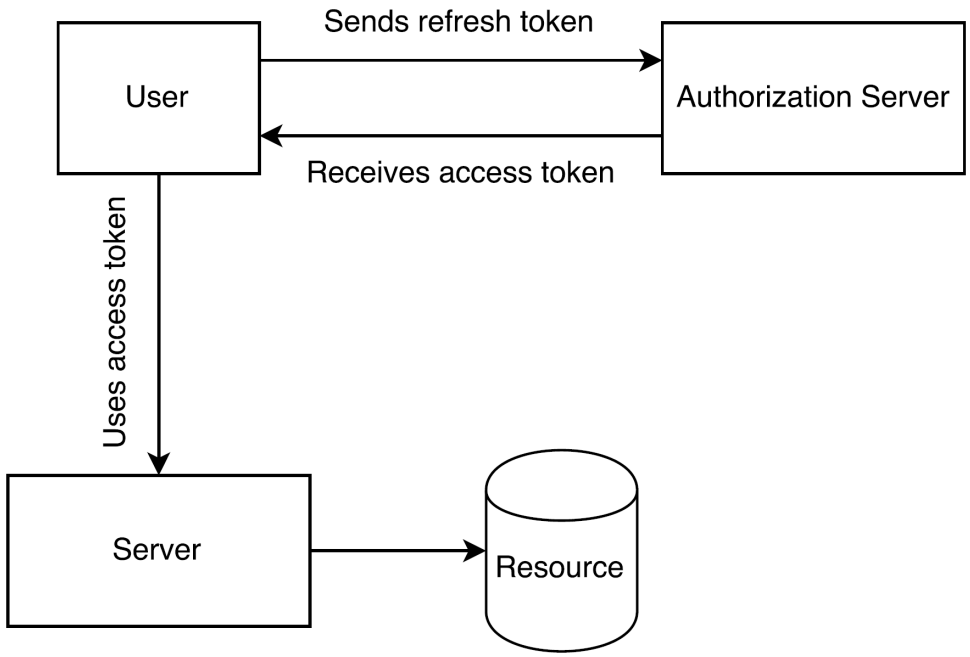

FIGURE 2.7 – Rafraîchissement et jetons d'accès

L'aspect essentiel de la séparation entre les jetons d'accès et de rafraîchissement réside dans la possibilité de rendre les jetons d'accès faciles à valider. Un jeton d'accès qui porte une signature (comme un JWT signé) peut être validé par le serveur de ressources de lui-même. Il n'est pas nécessaire de contacter le serveur d'autorisation à cette fin.

Les jetons de rafraîchissement, en revanche, nécessitent un accès au serveur d'autorisation. En gardant la validation séparée des requêtes au serveur d'autorisation, une meilleure latence et des modèles d'accès moins complexes sont possibles. Une sécurité appropriée en cas de fuite des jetons est obtenue en rendant les jetons d'accès aussi éphémères que possible et en y intégrant des contrôles supplémentaires (tels que des contrôles du client).

Les jetons d'actualisation, du fait de leur longue durée de vie, doivent être protégés contre les fuites. En cas de fuite, il peut être nécessaire de mettre le serveur sur liste noire (les jetons d'accès à courte durée de vie obligent à utiliser des jetons de rafraîchissement, ce qui permet de protéger la ressource une fois qu'elle a été mise sur liste noire et que tous les jetons d'accès ont expiré).

2.7 JWTs et OAuth2 :

Bien qu'OAuth2[4] ne fasse aucune mention du format de ses jetons, les JWT correspondent bien à ses exigences. Les JWT signés sont de bons jetons d'accès, car ils peuvent coder toutes les données nécessaires pour différencier les niveaux d'accès à une ressource, peuvent porter une date d'expiration et sont signés pour éviter les requêtes de validation auprès du serveur d'autorisation. Plusieurs fournisseurs d'identité fédérés émettent des jetons d'accès au format JWT.

Les JWT peuvent également être utilisés pour les jetons de rafraîchissement. Il y a cependant moins de raisons de les utiliser à cette fin. Comme les jetons de rafraîchissement nécessitent un accès au serveur d'autorisation, la plupart du temps, un simple UUID suffira, car il n'est pas nécessaire que le jeton porte une charge utile (il peut toutefois être signé).

2.8 JWTs et OpenID Connect :

OpenID Connect[3] est un effort de normalisation visant à rassembler les cas d'utilisation typiques d'OAuth2 dans une spécification commune et bien définie. Étant donné que de nombreux détails concernant OAuth2 sont laissés au choix des implémenteurs, OpenID Connect tente de fournir des définitions appropriées pour les parties manquantes. Plus précisément, OpenID Connect définit une API et un format de données pour exécuter les flux d'autorisation OAuth2. De plus, il fournit une couche d'authentification construite au-dessus de ce flux. Le format de données choisi pour certaines de ses parties est JSON Web Token. En particulier, le jeton d'identification est un type spécial de jeton qui porte des informations sur l'utilisateur authentifié.

2.9 Flux et JWTs d'OpenID Connect :

OpenID Connect définit plusieurs flux qui renvoient des données de différentes manières. Certaines de ces données peuvent être au format JWT.

- **Flux d'autorisation :** le client demande un code d'autorisation au point final d'autorisation (/authorize). Ce code peut être utilisé sur le point d'accès aux jetons (/token) pour demander un jeton d'identification (au format JWT), un jeton d'accès ou un jeton de rafraîchissement.
- **Flux implicite :** le client demande les jetons directement au point de terminaison d'autorisation (/authorize). Les jetons sont spécifiés dans la demande. Si un jeton d'identification est demandé, il est renvoyé au format JWT.
- **Flux hybride :** le client demande à la fois un code d'autorisation et certains jetons au point de terminaison d'autorisation (/authorize). Si un jeton d'identification est demandé, il est renvoyé au format JWT. Si un jeton d'identification n'est pas demandé à cette étape, il peut être demandé ultérieurement directement au point de terminaison des jetons (/token).

CHAPITRE 3

Les signatures Web JSON

3.1 introduction :

Les signatures Web JSON, standard défini par la RFC 7515[6] sont probablement la caractéristique la plus utile des JWT. En combinant un format de données simple avec une série bien définie d'algorithmes de signature, les JWTs deviennent rapidement le format idéal pour partager des données en toute sécurité entre clients et intermédiaires.

L'objectif d'une signature est de permettre à une ou plusieurs parties d'établir l'authenticité du JWT. Dans ce contexte, l'authenticité signifie que les données contenues dans le JWT n'ont pas été altérées. En d'autres termes, toute partie capable d'effectuer une vérification de la signature peut se fier au contenu fourni par le JWT. Il est important de souligner qu'une signature n'empêche pas les autres parties de lire le contenu du JWT.

Le processus de vérification de la signature d'un JWT est appelé validation ou validation d'un jeton. Un jeton est considéré comme valide lorsque toutes les restrictions spécifiées dans son en-tête et sa charge utile sont satisfaites. Il s'agit d'un aspect très important des JWT : les implémentations sont tenues de vérifier un JWT jusqu'au point spécifié par son en-tête et sa charge utile (et, en outre, tout ce que l'utilisateur exige). Ainsi, un JWT peut être considéré comme valide même s'il n'a pas de signature (si l'en-tête a la revendication alg définie sur none). En outre, même si un JWT possède une signature valide, il peut être considéré comme invalide pour d'autres raisons (par exemple, il peut avoir expiré, selon la revendication exp). Une attaque courante contre les JWT signés consiste à supprimer la signature puis à modifier l'en-tête pour en faire un JWT non sécurisé. Il incombe à l'utilisateur de s'assurer que les JWT sont validés conformément à ses propres exigences.

3.2 Structure d'un JWT signé :

Un JWT signé est composé de trois éléments : l'en-tête, la charge utile et la signature.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Le processus de décodage des deux premiers éléments (l'en-tête et la charge utile) est identique au cas des JWT non sécurisés.

```
{
  "alg": "HS256",
  "typ": "JWT"
}

{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Les JWTs signés comportent toutefois un élément supplémentaire : la signature. Cet élément apparaît après le dernier point (.) dans la forme de sérialisation compacte.

Plusieurs types d'algorithmes de signature sont disponibles selon la spécification JWS, de sorte que la façon dont ces octets sont interprétés varie. La spécification JWS exige qu'un seul algorithme soit pris en charge par toutes les implémentations conformes :

- HMAC utilisant SHA-256, appelé HS256 dans la spécification JWA.

La spécification définit également une série d'algorithmes recommandés :

- RSASSA PKCS1 v1.5 utilisant SHA-256, appelé RS256 dans la spécification JWA.
- ECDSA utilisant P-256 et SHA-256, appelé ES256 dans la spécification JWA.

Les autres algorithmes pris en charge par la spécification, en option, sont les suivants :

- HS384, HS512 : variantes SHA-384 et SHA-512 de l'algorithme HS256.
- RS384, RS512 : variantes SHA-384 et SHA-512 de l'algorithme RS256.

- ES384, ES512 : Variations SHA-384 et SHA-512 de l'algorithme ES256.
- PS256, PS384, PS512 : RSASSA-PSS + MGF1 avec variantes SHA256/384/512.

3.3 Aperçu de l'algorithme pour la sérialisation compacte :

Afin de discuter de ces algorithmes en général, définissons d'abord quelques fonctions dans un environnement JavaScript 2015 :

- **base64** : une fonction qui reçoit un tableau d'octets et renvoie un nouveau tableau d'octets en utilisant l'algorithme Base64-URL.
- **utf8** : une fonction qui reçoit du texte dans n'importe quel encodage et retourne un tableau d'octets avec un encodage UTF-8.
- **JSON.stringify :** une fonction qui prend un objet JavaScript et le sérialise sous forme de chaîne (JSON).
- **sha256** : une fonction qui prend un tableau d'octets et retourne un nouveau tableau d'octets en utilisant l'algorithme SHA-256.
- **hmac** : une fonction qui prend une fonction SHA, un tableau d'octets et un secret et retourne un nouveau tableau d'octets en utilisant l'algorithme HMAC.
- **rsassa** : une fonction qui prend une fonction SHA, un tableau d'octets et la clé privée et retourne un nouveau tableau d'octets en utilisant l'algorithme RSASSA.

Pour les algorithmes de signature basés sur HMAC :

```
const encodedHeader = base64(utf8(JSON.stringify(header)));
const encodedPayload = base64(utf8(JSON.stringify(payload)));
const signature = base64(hmac(‘${encodedHeader}.${encodedPayload}‘,
secret, sha256));
const jwt = ‘${encodedHeader}.${encodedPayload}.${signature}‘;
For public-key signing algorithms:
const encodedHeader = base64(utf8(JSON.stringify(header)));
const encodedPayload = base64(utf8(JSON.stringify(payload)));
const signature = base64(rsassa(‘${encodedHeader}.${encodedPayload}‘,
privateKey, sha256));
const jwt = ‘${encodedHeader}.${encodedPayload}.${signature}‘;
```

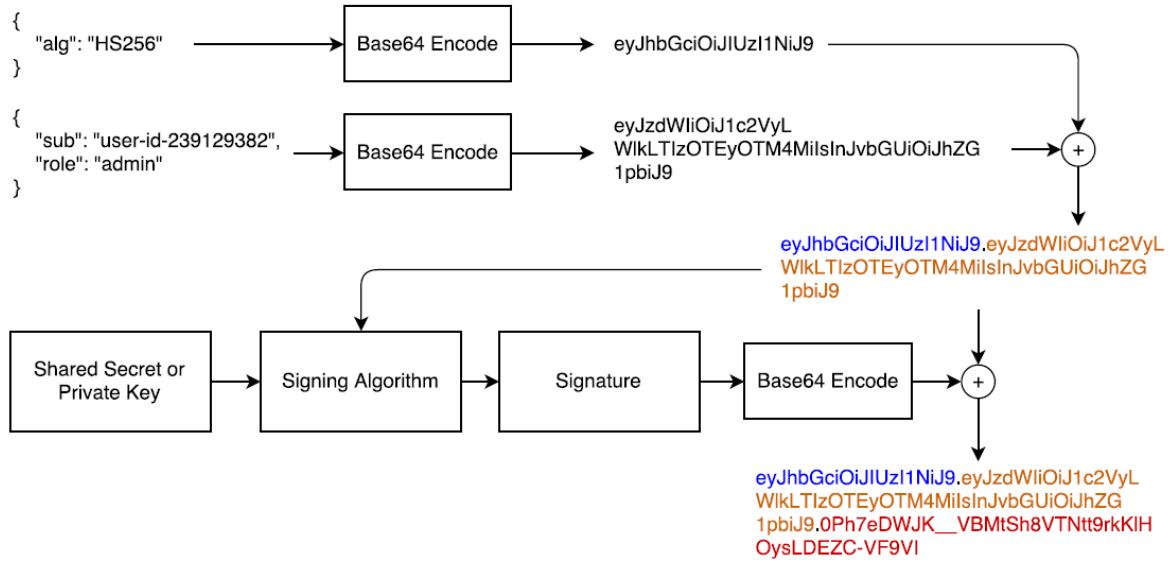

FIGURE 3.1 – Sérialisation compacte de JWS

3.3.1 Aspects pratiques des algorithmes de signature :

Tous les algorithmes de signature accomplissent la même chose : ils fournissent un moyen d'établir l'authenticité des données contenues dans le JWT. des données contenues dans le JWT. La façon dont ils le font varie.

Le code d'authentification de message de hachage (HMAC) est un algorithme qui combine une certaine charge utile avec un secret en utilisant une fonction de hachage cryptographique. Le résultat est un code qui ne peut être utilisé pour vérifier un message que si les parties génératrices et vérificatrices connaissent toutes deux le secret. En d'autres termes, les HMAC permettent de vérifier des messages au moyen de secrets partagés.

La fonction de hachage cryptographique utilisée dans HS256, l'algorithme de signature le plus courant pour les JWT, est SHA-256. SHA-256. Les fonctions de hachage cryptographiques prennent un message de longueur arbitraire et produisent une sortie de longueur fixe. Le même message produira toujours le même résultat. La partie cryptographique d'une fonction de hachage s'assure qu'il est mathématiquement infaisable de récupérer le message original à partir de la sortie de la fonction. Ainsi, les fonctions de hachage cryptographiques sont des fonctions à sens unique qui peuvent être utilisées pour identifier des messages sans avoir à les partager. Une légère variation du message (un seul octet, par exemple) produira un résultat entièrement différent.

RSASSA est une variante de l'algorithme RSA adaptée aux signatures. RSA est un algorithme à clé publique. Les algorithmes à clé publique génèrent des clés partagées : une clé publique et une clé privée. Dans cette variante spécifique de l'algorithme, la clé privée peut être utilisée à la fois pour créer un message signé et pour vérifier son authenticité. La clé publique, en revanche, ne peut être utilisée que pour vérifier l'authenticité d'un message. Ainsi, ce schéma permet la distribution sécurisée d'un message de type "un à plusieurs". Les parties réceptrices peuvent vérifier l'authenticité d'un message en conservant une copie de la clé publique qui lui est associée, mais elles ne peuvent pas créer de nouveaux messages avec cette clé. Cela permet des scénarios d'utilisation différents des schémas de signature à secret partagé tels que HMAC. Avec HMAC + SHA-256, toute partie qui peut vérifier un message peut également créer de nouveaux messages. Par exemple, si un utilisateur légitime devient malveillant, il peut modifier des messages sans que les autres parties s'en aperçoivent. Avec un système à clé publique, un utilisateur malveillant n'aurait que la clé publique en sa possession et ne pourrait donc pas créer de nouveaux messages signés avec celle-ci.

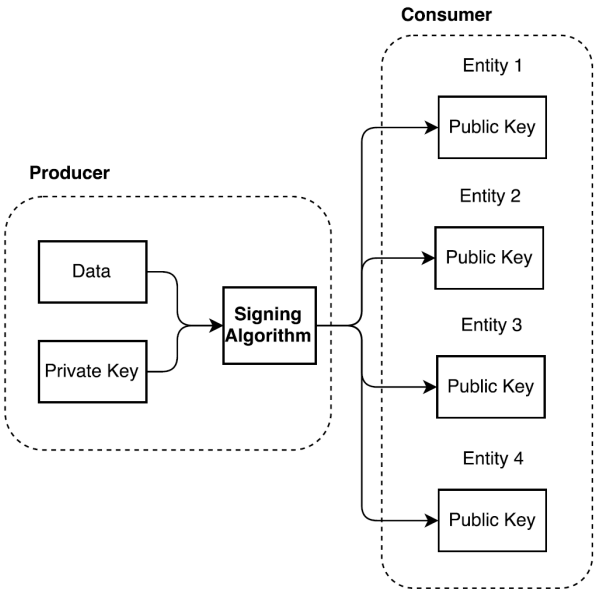

FIGURE 3.2 – Signature de type "un à plusieurs

La cryptographie à clé publique permet d'autres scénarios d'utilisation. Par exemple, en utilisant une variation du même algorithme RSA, il est possible de chiffrer des messages en utilisant la clé publique. Ces messages ne peuvent être décryptés qu'à l'aide de la clé privée. Cela permet de construire un canal de communication sécurisé de plusieurs à un. Cette variante est utilisée pour les JWTs chiffrés.

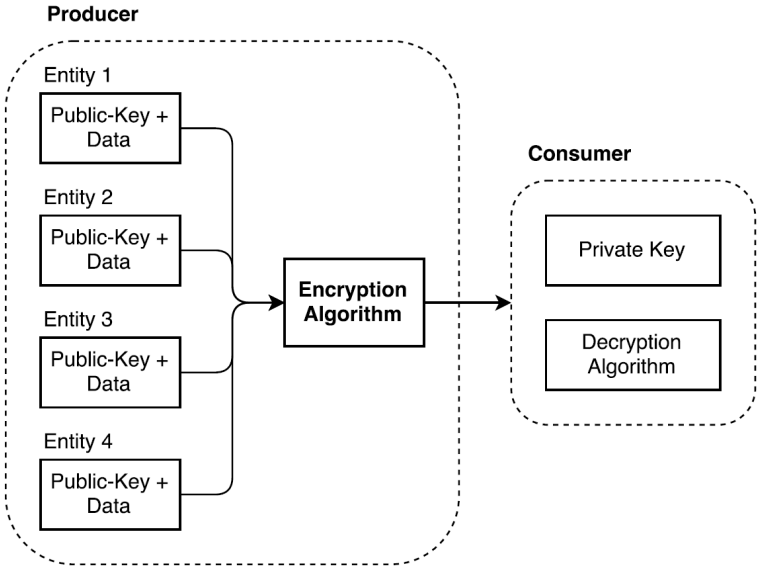

FIGURE 3.3 – Cryptage bi-univoque

L'algorithme de signature numérique à courbes elliptiques (ECDSA) est une alternative au RSA. Cet algorithme génère également une paire de clés publique et privée, mais les mathématiques qui le sous-tendent sont différentes. Cette différence permet d'exiger moins de matériel que RSA pour des garanties de sécurité similaires.

3.4 Revendications d'en-tête JWS

JWS permet des cas d'utilisation particuliers qui obligent l'en-tête à contenir davantage de revendications. Par exemple, pour les algorithmes de signature à clé publique, il est possible d'intégrer l'URL de la clé publique dans une déclaration. Ce qui suit est la liste des revendications d'en-tête enregistrées disponibles pour les jetons JWS. Toutes ces revendications s'ajoutent à celles disponibles pour les JWT non sécurisés et sont facultatives en fonction de la manière dont le JWT signé est censé être utilisé.

— **jku** : URL de l'ensemble de clés Web JSON (JWK). Une URI pointant vers un ensemble de clés publiques codées en JSON utilisées pour signer ce JWT. La sécurité du transport (telle que TLS pour HTTP) doit être utilisée pour récupérer les clés. Le format des clés est un ensemble JWK.
— **jwk** : Clé Web JSON. La clé utilisée pour signer ce JWT au format JSON Web Key.
— **kid** : Key ID. Une chaîne définie par l'utilisateur représentant une seule clé utilisée pour signer ce JWT. Cette déclaration est utilisée pour signaler les changements de signature de clé aux destinataires (lorsque plusieurs clés sont utilisées).
— **x5u** : URL X.509. Un URI pointant vers un ensemble de certificats publics X.509 (une norme de format de certificat) encodés sous forme PEM. Le premier certificat de l'ensemble doit être celui utilisé pour signer ce JWT. Les certificats suivants signent chacun le certificat précédent, complétant ainsi la chaîne de certificats. La norme X.509 est définie dans la RFC 52807. La sécurité du transport est nécessaire pour transférer les certificats.
— **x5c** : Chaîne de certificats X.509. Un tableau JSON des certificats X.509 utilisés pour

signer cette JWS. Chaque certificat doit être la valeur encodée en Base64 de sa représentation DER PKIX. Le premier certificat du tableau doit être celui utilisé pour signer ce JWT, suivi du reste des certificats de la chaîne de certificats.
— **x5t** : Empreinte SHA-1 du certificat X.509. L'empreinte SHA-1 du certificat X.509 codé en DER utilisé pour signer ce JWT.
— **x5t#S256** : Identique à x5t, mais utilise SHA-256 au lieu de SHA-1.
— **typ** : Identique à la valeur type pour les JWT non chiffrés, avec les valeurs supplémentaires "JOSE" et "JOSE+JSON" utilisées pour indiquer respectivement la sérialisation compacte et la sérialisation JSON. Cette valeur n'est utilisée que dans les cas où des objets similaires porteurs d'en-tête JOSE sont mélangés à ce JWT dans un seul conteneur.
— **crit** : de critique. Un tableau de chaînes contenant les noms des réclamations présentes dans ce même en-tête, utilisées comme des extensions définies par l'implémentation qui doivent être traitées par les analyseurs de ce JWT. Il doit soit contenir les noms des revendications, soit ne pas être présent (le tableau vide n'est pas une valeur valide).

3.5 Sérialisation JWS JSON

La spécification JWS définit un autre type de format de sérialisation qui n'est pas compact. Cette représentation permet des signatures multiples dans le même JWT signé. Elle est connue sous le nom de JWS JSON Serialization. Sous la forme JWS JSON Serialization, les JWT signés sont représentés sous forme de texte imprimable au format JSON (c'est-à-dire ce que vous obtiendriez en appelant JSON.stringify dans un navigateur). Un objet JSON supérieur qui porte les paires clé-valeur suivantes est requis :
— **payload** :une chaîne codée en base64 de l'objet de données utiles JWT réel
— **signatures** :un tableau d'objets JSON contenant les signatures. Ces objets sont définis ci-dessous.

À son tour, chaque objet JSON contenu dans le tableau des signatures doit contenir les paires clé-valeur suivantes :
— **protégé** : une chaîne codée en Base64 de l'en-tête JWS. Les déclarations contenues dans cet en-tête sont protégées par la signature. Cet en-tête n'est requis que s'il n'y a pas d'en-têtes non protégés. Si des en-têtes non protégés sont présents, cet en-tête peut être présent ou non.
— **en-tête** : un objet JSON contenant des déclarations d'en-tête. Cet en-tête n'est pas protégé par la signature. Si aucun en-tête protégé n'est présent, cet élément est obligatoire. Si un en-tête protégé est présent, cet élément est facultatif.
— **signature** :Une chaîne encodée en Base64 de la signature JWS.

Contrairement à la forme de sérialisation compacte (où seul un en-tête protégé est présent), la sérialisation JSON admet deux types d'en-têtes : protégés et non protégés. L'en-tête protégé est validé par la signature. L'en-tête non protégé n'est pas validé par celle-ci. C'est à l'implémentation ou à l'utilisateur de choisir les déclarations à mettre dans l'un ou l'autre. Au moins un de ces en-têtes doit être présent. Les deux peuvent également être présents en même temps.

Lorsque des en-têtes protégés et non protégés sont présents, l'en-tête JOSE réel est construit à partir de l'union des éléments des deux en-têtes. Aucun en-tête en double ne peut être présent.

```
{
"payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
            h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
"signatures": [
    {
        "protected": "eyJhbGciOiJSUzI1NiJ9",
        "header": { "kid": "2010-12-29" },
        "signature":
                    "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AA
                    uHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
                    RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
                    _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6
                    IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlU
                    PQGe77Rw"
    },
    {
        "protected": "eyJhbGciOiJFUzI1NiJ9",
        "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
        "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                     w5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
        }
    ]
}
```

Cet exemple encode deux signatures pour la même charge utile : une signature RS256 et une signature ES256.

3.5.1 Sérialisation JWS JSON aplatie

La sérialisation JWS JSON définit une forme simplifiée pour les JWT ne comportant qu'une seule signature. Cette forme est connue sous le nom de sérialisation JWS JSON aplatie. La sérialisation aplatie supprime le tableau des signatures et place les éléments d'une seule signature au même niveau que l'élément de charge utile.

Par exemple, en supprimant l'une des signatures de l'exemple précédent, l'objet de sérialisation JSON aplati serait le suivant :

```
{
    "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQog
                Imh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
    "protected": "eyJhbGciOiJFUzI1NiJ9",
    "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
    "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFC
                 gfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
}
```

3.6 Signature et validation des jetons

L'utilisation de JWT signés est suffisamment simple en pratique pour que vous puissiez appliquer les concepts expliqués jusqu'à présent pour les utiliser efficacement. En outre, il existe de bonnes bibliothèques que vous pouvez utiliser pour les mettre en œuvre de manière pratique. Nous allons passer en revue les algorithmes nécessaires et recommandés en utilisant la plus populaire de ces bibliothèques pour JavaScript. Des exemples d'autres langages et bibliothèques populaires peuvent être trouvés dans le code d'accompagnement.

```
import jwt from 'jsonwebtoken'; //var jwt = require('jsonwebtoken');
const payload = {
    sub: "1234567890",
    name: "faycal fahd",
    admin: true
};
```

3.6.1 HS256 : HMAC + SHA-256

Les signatures HMAC nécessitent un secret partagé. N'importe quelle chaîne de caractères fera l'affaire :

```
const secret = 'my-secret';
const signed = jwt.sign(payload, secret, {
    algorithm: 'HS256',
    expiresIn: '5s'
});
```

La vérification du jeton est tout aussi simple :

```
const decoded = jwt.verify(signed, secret, {
        algorithms: ['HS256'],
});
```

La bibliothèque **jsonwebtoken** vérifie la validité du jeton en fonction de la signature et de la date d'expiration. Dans ce cas, si le jeton devait être vérifié après 5 secondes de sa création, il serait considéré comme invalide et une exception serait levée.

3.6.2 RS256 : RSASSA + SHA256

La signature et la vérification des jetons signés RS256 sont tout aussi simples. La seule différence réside dans l'utilisation d'une paire de clés privée/publique plutôt que d'un secret partagé. Il existe de nombreuses façons de créer des clés RSA. OpenSSL est l'une des bibliothèques les plus populaires pour la création et la gestion des clés :

```
# Generate a private key
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
# Derive the public key from the private key
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Les deux fichiers PEM sont de simples fichiers texte. Leur contenu peut être copié et collé dans les fichiers sources JavaScript et transmis à la bibliothèque jsonwebtoken.

```
// We can get this from private_key.pem above.
const privateRsaKey = `<OUR-PRIVATE-RSA-KEY>`;
const signed = jwt.sign(payload, privateRsaKey, {
        algorithm: 'RS256',
        expiresIn: '5s'
});
// We can get this from public_key.pem above.
const publicRsaKey = `<OUR-PUBLIC-RSA-KEY>`;
const decoded = jwt.verify(signed, publicRsaKey, {
        algorithms: ['RS256'],
});
```

CHAPITRE 4

Implémentation et Analyse

4.1 Introduction

Ce dernier chapitre présente la partie de l'implémentation et l'analyse des différents composants décrits au niveau du chapitre précédent.Dans un premier temps,on présente l'environnement matériel et logiciel.Ensuite,on décrit le travail réalisé en détaillant quelques captures d'écrans des tests réalisés.

4.2 Environnement de travail

4.2.1 Environnement logiciel

4.2.1.1 Choix des technologies de développement

Python Flask : **Flask** est un micro framework open-source de développement web en Python. Il est classé comme microframework car il est très léger. Flask a pour objectif de garder un noyau simple mais extensible. Il n'intègre pas de système d'authentification, pas de couche d'abstraction de base de données, ni d'outil de validation de formulaires. Cependant, de nombreuses extensions permettent d'ajouter facilement des fonctionnalités.

Interfaces de programmation utilisés

Ci-dessous, une liste de composants contenus dans notre implémentation :

- Sqlite Connector : API de connexion à des bases de données
- Flask : Python framework pour le développement web permettant de créer des services Web avec une architecture REST.

ReactJs : **ReactJs** est une bibliothèque JavaScript libre développée par Facebook depuis 2013. Le but principal de cette bibliothèque est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page (ou portion) HTML à chaque changement d'état.

4.2.1.2 Les Outils de développement

⇒ **VS Code**

VS Code est un environnement de développement intégré qui supporte une large variété des langages de programmation et d'outils de collaboration. L'éditeur intégré propose des fonctions de contrôles syntaxiques et sémantiques, d'avertissements et de conseil, de reprise de codes (« refactoring » : renommage, changement des méthodes, gestion des classes,...), de sauvegarde et de reprise.

⇒ **SQLite**

Sqlite est une base de données relationnelles libre. Elle est très employée sur les bases de bonnées locales, souvent en association avec Python (langage) et fonctionne indifféremment sur tous les systèmes d'exploitation. Il a le principe d'une base de données relationnelle et d'enregistrer les informations dans des tables, qui représentent des regroupements de données par sujets. Les tables sont reliées entre elles par des relations.

Le langage SQL (Structured Query Language) est un langage reconnu par Sqlite et les autres bases de données et permettant de modifier le contenu d'une base de données.

4.3 Étapes de réalisation

4.3.1 L'architecture de l'application Web

Au sein des applications Web on retrouve généralement trois couches :

- La première couche est la présentation qui contient les contrôleurs en cas d'utilisation de Framework MVC, elle est chargée d'assurer le dialogue entre l'application et les services extérieurs.
- La deuxième couche est la couche métier qui contient les services et s'occupe des traitements.
- La dernière couche est la DAO qui s'occupe de l'accès aux bases de données.

La couche présentation (Vue) qui s'occupe de la saisie, le contrôle et l'affichage des résultats. Généralement la couche présentation respecte le pattern MVC qui fonctionne comme suit :

1. La vue permet de saisir les données, envoie ces données au contrôleur
2. Le contrôleur récupère les données saisies. Après la validation de ces données, il fait appel à la couche métier pour exécuter des traitements.
3. Le contrôleur stocke le résultat du modèle.
4. Le contrôleur fait appel à la vue pour afficher les résultats.
5. La vue récupère les résultats à partir du modèle et les affiche

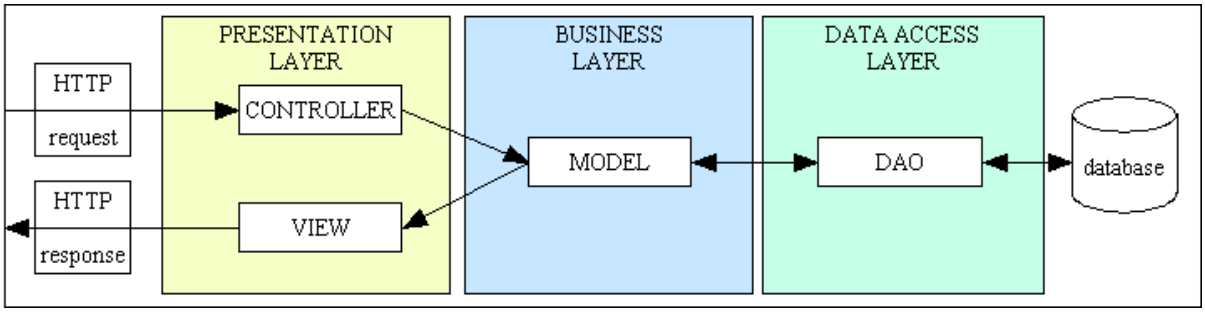

FIGURE 4.1 – Architecture d'une application Web

On a réalisé une application web dont le code est vulnérable en se basant sur l'architecture décris ci-dessus et aussi en implémentant JWT.

4.4 Interfaces d'application et analyse des cas

La figure 4.2 présente l'accès à la page du login.

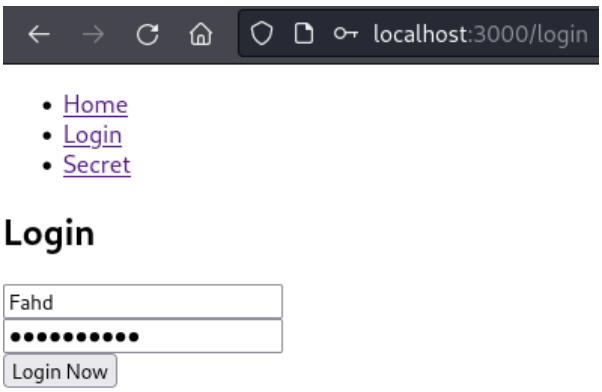

FIGURE 4.2 – Page de login

Après le login, on reçoit le token JWT du serveur, la figure 4.3 présente la page log de la console.

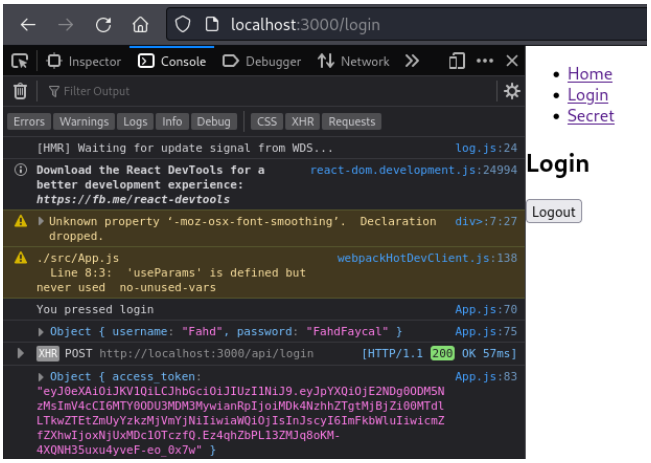

FIGURE 4.3 – Page log de la console

Dans un premier temps, on stocke le token JWT dans le stockage local (localStorage) du navigateur comme indiqué dans la figure 4.4.

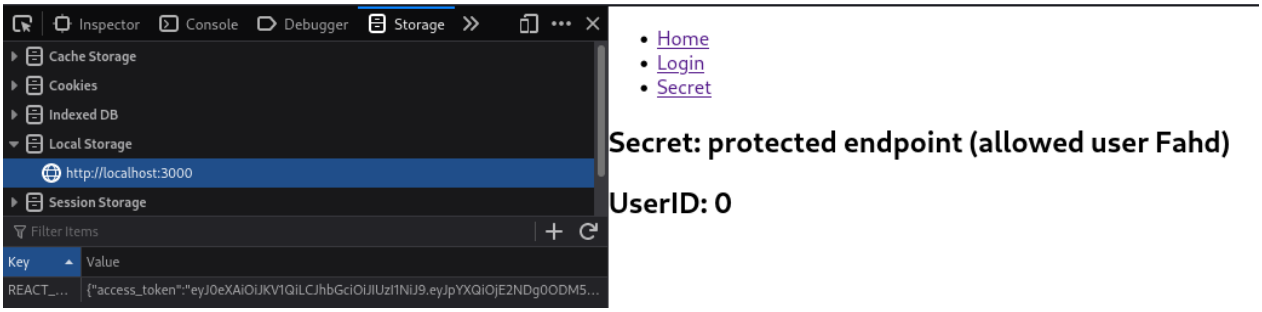

FIGURE 4.4 – JWT stocké dans localStorage

La figure 4.5 présente la page du endpoint sécurisé avec l'id d'utilisateur (URL/secret/id).

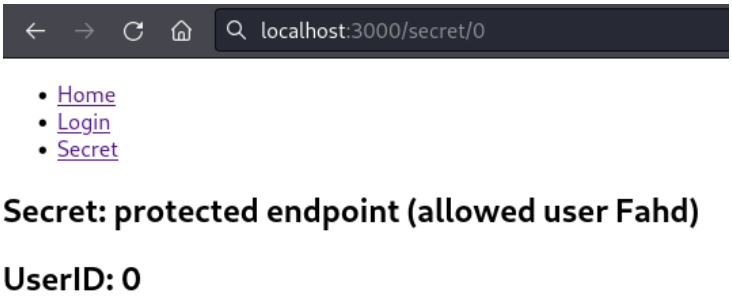

FIGURE 4.5 – Page du endpoint sécurisé

4.4.1 Vulnérabilité Web et JWT

On constate que l'endpoint (secret/id) est susceptible à une attaque XSS, la figure 4.6 démontre une attaque xss contre ce endpoint.

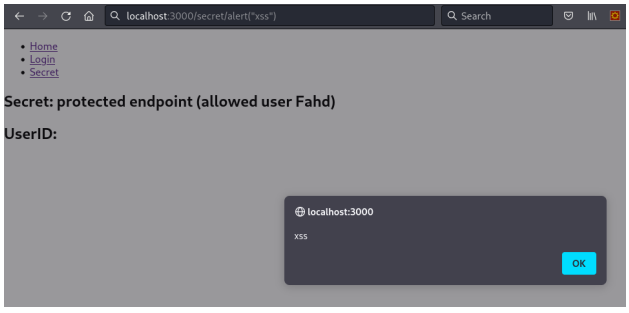

FIGURE 4.6 – Démonstration d'une attaque XSS

Sachant que le token JWT est stocké dans localStorage, on peut accéder à ce dernier en utilisant Javascript, et vu que notre endpoint est affecté par une vulnérabilité XSS, on peut facilement voler le token JWT, la figure 4.7 présente un preuve du concept(POC).

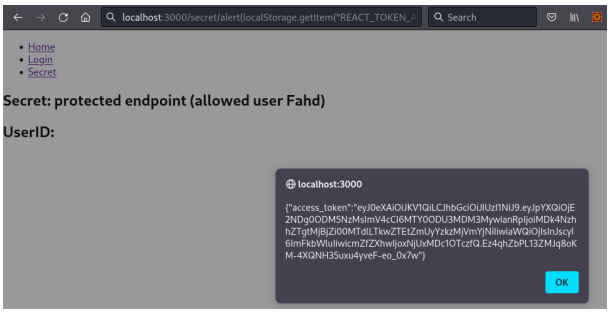

FIGURE 4.7 – Accès au token JWT à partir d'une attaque XSS

Pour mitiger cette attaque contre le token JWT, on adopte une solution universelle même avec la présence d'une vulnérabilité XSS, l'attaquent ne peut pas accéder au token JWT. la solution qu'on a adopté s'implique à stocker le token dans les cookies(figure 4.8) au lieu du localStorage aussi en rendant les cookies accessible juste par les requêtes HTTP justement par définir le paramètre HttpOnly, ceci rend le token inaccessible en utilisant Javascript(figure 4.9).

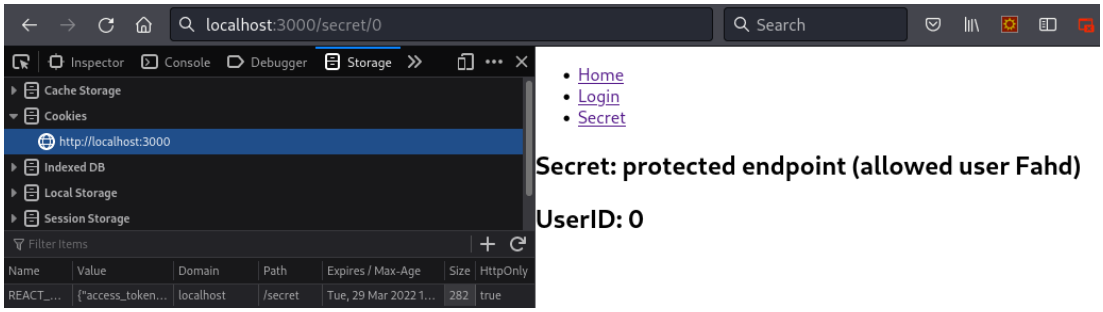

FIGURE 4.8 – Token JWT stocké dans les cookies

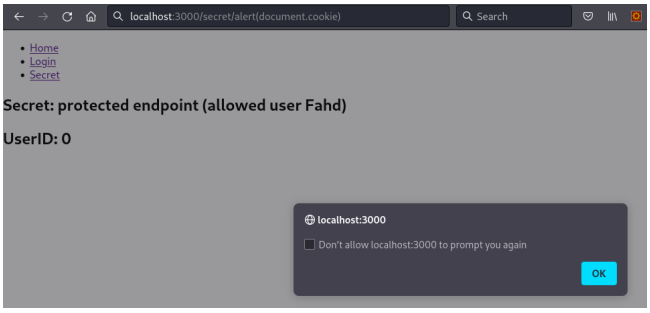

FIGURE 4.9 – Cookies inaccessible à partir d'une attaque XSS

La mitigation de l'attaque d'accès au token a créé une vulnérabilité CSRF(Cross-Site Request Forgery) qui est une attaque qui force les utilisateurs authentifiés à soumettre une requête à une application Web contre laquelle ils sont actuellement authentifiés, la figure 4.10 présente une requête authentifié vers l'endpoint (secret/test) par l'utilisateur Fahd à partir d'une autre application Web.

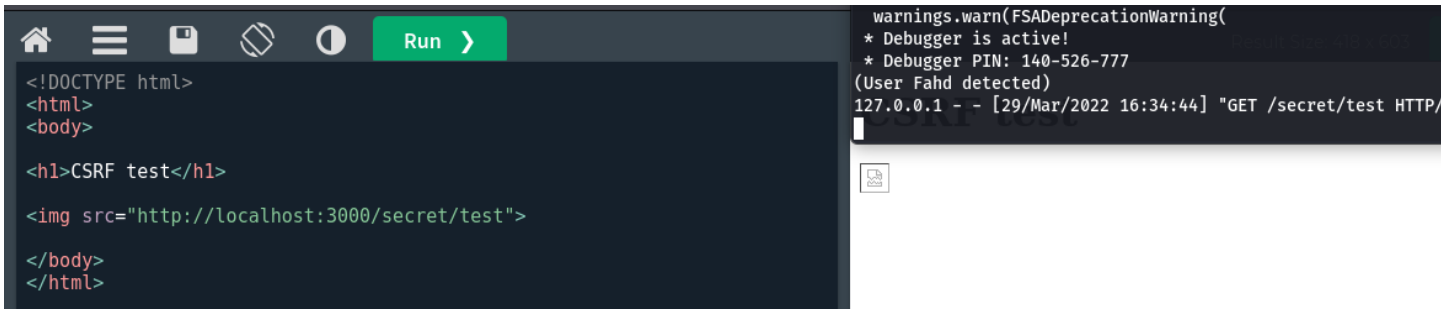

FIGURE 4.10 – Attaque CSRF

Entre le choix du cookies et localStorage, c'est simple de mitiger les attaques CSRF que les attaques XSS, parmis les solutions proposées, c'est l'introduction d'un token CSRF unique à chaque requête, sans la présence de ce dernier, le serveur simplement ignore la requête.

4.4.2 JWT et révocation de l'accès

On ajoute un utilisateur (User1) à la base de données pour tester la révocation d'accès, on définie le paramètre (is_active=1) (figure 4.11).

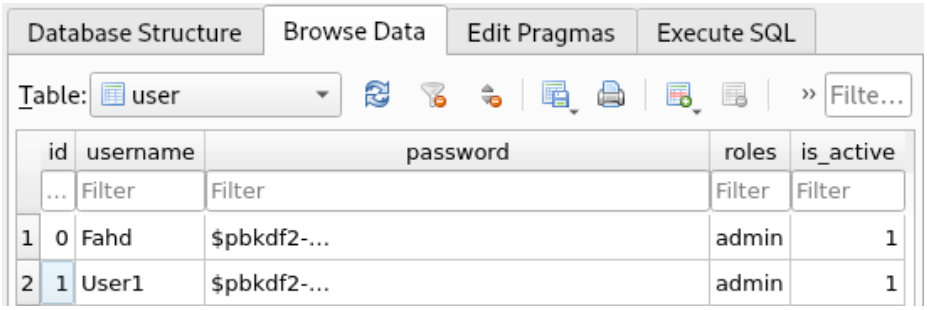

FIGURE 4.11 – L'utilisateur User1 ajouté à la base de données

On s'authentifie en tant que User1 (figure 4.12)

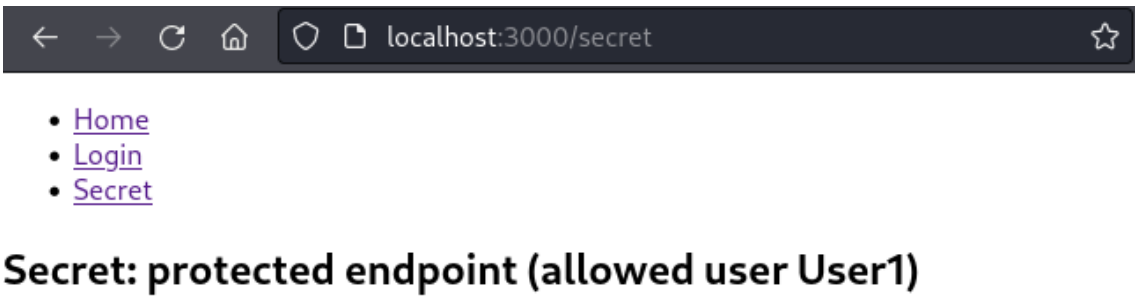

FIGURE 4.12 – Authentification en tant que User 1

On veut maintenant révoquer l'accès du User1, on change le paramètre (is_active=0) (figure 4.14).

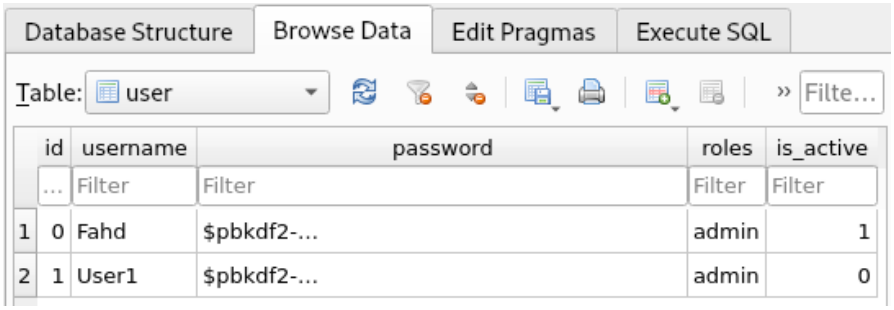

FIGURE 4.13 – Changement du paramètre (is_defined) pour User1

Maintenant l'utilisateur User1 doit pas avoir accès à l'endpoint (secret), mais on constate que User1 maintient l'accès (figure **??**), cela est causé par le faite que les tokens JWT sont sans état, cela permet le serveur de ne peut constater la base de données à chaque requête mais justement vérifier si le token est signé par le serveur pour faire confiance des données inclus dans le token, et puisque le serveur ne constate pas la base de données après la réception du token pour savoir si l'utilisateur peut avoir accès ou non, mais il se base seulement sur les deux revendications (émis au moment(**iat**) et expiration(**exp**)) pour décider si le token est encore valable.

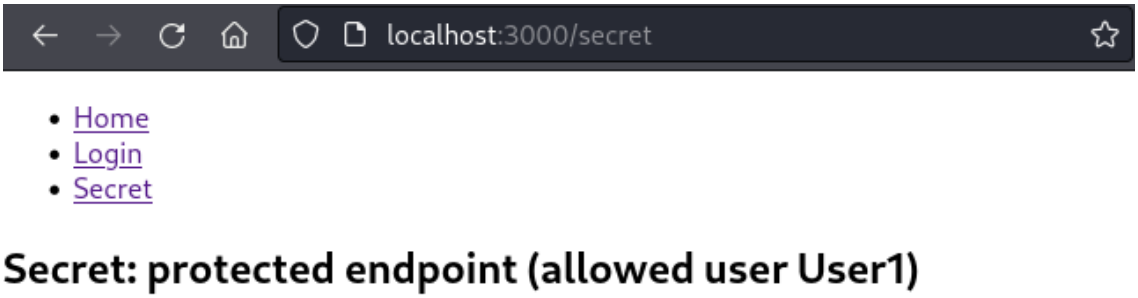

FIGURE 4.14 – L'utilisateur User1 reste encore authentifier

Parmi les solutions disponibles pour le moment pour la révocation d'accès, on trouve la minimisation de la durée d'expiration mais ça va rendre l'expérience d'utilisateur déplaisant puisque il doit à chaque fois se réauthentifier, et aussi le stockage des tokens révoqués dans une table mais cela va contredirai le concept que les tokens JWT sont sans état.

4.4.3 JSON Web Encryption

Le cryptage laisse beaucoup de place aux erreurs de mise en œuvre potentielles, en particulier lorsqu'il s'agit d'un cryptage asymétrique.

Les algorithmes de cryptage autorisés par JWE sont décrits dans la RFC7518[5] qui se compose de deux sections :

- Le cryptage à clé publique, qui offre des options telles que :
  - RSA avec PKCS #1v1.5 rembourrage(padding)
  - RSA avec OAEP rembourrage(padding)
  - ECDH
- Le cryptage à clé privé(symétrique), qui offre des options telles que :
  - AES-CBC + HMAC
  - AES-GCM

Examinons en détail certains des choix de chiffrement à clé publique. Le chiffrement à clé privé(symétrique) est acceptable (en supposant qu'on dispose d'une implémentation GCM solide et d'un support matériel adéquat).

RSA avec PKCS #1v1.5 rembourrage(padding) RSA avec le padding PKCS #1v1.5 est vulnérable à un type d'attaque par texte chiffré choisi, appelé oracle de padding, démontré par Bleichenbacher Daniel[1].

RSA avec OAEP padding) RSA avec un padding OAEP est probablement sécurisé. L'OAEP a une preuve de sécurité bidon mais est bien plus sûr que le PKCS1v1.5. Cependant, il existe de sérieux doutes quant à la sécurité à long terme du RSA lui-même.
La plupart des cryptographes recommandent de s'éloigner de RSA.

ECDH JWT n'autorise que le protocole Diffie-Hellman à courbes elliptiques (ECDH) sur l'une des courbes du NIST (courbes de Weierstrass, qui introduisent le risque d'attaques par courbes invalides permettant aux attaquants de voler les clés secrètes).
Si on tente à éviter les attaques par courbe invalide en utilisant l'une des courbes elliptiques pour la sécurité, on n'est plus conforme aux normes JWT[5].

4.5 Conclusion

Dans cette partie, on a trouvé que les tokens JWT sont imparfaits vu qu'ils possèdent des limitations tel que la révocation d'accés et le cryptage pour la signature. En fin, la norme JWS est complètement cassée et la conformité totale à la RFC rend vos applications vulnérables..

Conclusion

Les jetons Web JSON sont un outil qui fait appel à la cryptographie. Comme tous les outils qui le font, et surtout ceux qui sont utilisés pour traiter des informations sensibles, ils doivent être utilisés avec précaution. Leur apparente simplicité peut dérouter certains développeurs et leur faire croire que l'utilisation des JWTs se résume à choisir le bon algorithme de secret partagé ou de clé publique. Malheureusement, comme nous l'avons vu ci-dessus, ce n'est pas le cas. Il est de la plus haute importance de suivre les meilleures pratiques pour chaque outil de votre boîte à outils, et les JWT ne font pas exception. Il faut notamment choisir des bibliothèques de haute qualité qui ont fait leurs preuves, valider les déclarations relatives aux charges utiles et aux en-têtes, choisir les bons algorithmes, s'assurer que des clés solides sont générées et prêter attention aux subtilités de chaque API.

JWT est un standard qui tend à s'imposer pour l'authentification. Simple de mise en œuvre lors du développement du client, il l'est tout autant côté serveur grâce aux nombreuses librairies à disposition. Son format compact et le choix du format JSON lui permettent d'être utilisé dans tous les contextes. S'il sert ici à l'authentification, il peut également être utilisé pour échanger des informations de manière sécurisée entre 2 parties qui pourront s'assurer de la validité et de la provenance de ces dernières en échangeant la clé de signature.

Bibliographie

[1] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 1–12, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[2] Sebastián E. Peyrott. The jwt handbook, 2018.

[3] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect : In-depth formal security analysis and security guidelines. *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, 2017.

[4] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.

[5] Michael Jones. JSON Web Algorithms (JWA). RFC 7518, May 2015.

[6] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015.

[7] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[8] Organization for the Advancement of Structured Information Standards. Security assertion markup language (saml) v2.0, 2005.