Sécurité : identifier des vulnérabilités et les corriger

Sommaire

1. Introduction

1.1 Considérations générales

La sécurité en informatique, aussi connue sous le terme de cybersécurité, consiste dans la mise en œuvre de moyens pour protéger un site internet ou une application par la **détection**, la **prévention** et la **réponse** aux cybermenaces.

La sécurité web est un système de mesures de protection et de protocoles qui permettent la protection des applications web d'être piratées ou utilisées sans autorisation. Il existe plusieurs standards de sécurité qui doivent être suivis afin de se protéger contre les **failles de sécurités**.

Les failles de sécurité sont des faiblesses qui peuvent être exploitées par des personnes malveillantes afin de réaliser des opérations **non autorisées**. Pour exploiter une faille de sécurité, un attaquant doit avoir au moins un outil ou une technique lui permettant de se connecter à la vulnérabilité du système.

Les étapes essentielles dans la protection d'applications web contre les attaques incluent :

- l'application de méthodes de chiffrement les plus performantes ;
- la mise en place d'une authentification correcte ;
- la correction continue des vulnérabilités découvertes ;
- la protection contre le vol de données par la mise en place de pratiques de développement sécurisées.

La réalité est que la cybersécurité est un cycle continu du jeu du chat et de la souris. Un attaquant compétent peut l'être suffisamment pour trouver des failles même dans les environnements de sécurité les plus robustes, notamment grâce à l'évolution des outils et du matériel.

La sécurité intervient dans toute la chaine de diffusion de l'application. S'il est nécessaire de sécuriser ses données et de ne jamais faire confiance à ce que le back-end reçoit de l'extérieur, le front-end joue aussi un rôle important dans la sécurisation des échanges avec le back-end. La responsabilité de sécuriser les données de l'utilisateur est partagée entre le back-end et le front-end.

1.2 OWASP

L'**OWASP** - Open Web Application Security Project - est une communauté en ligne travaillant sur la réalisation d'articles, de méthodologies, de documentation, d'outils et de technologies dans le domaine de la sécurité des applications web. Ces ressources sont toutes gratuites.

L'OWASP a publié en 2013 un top 10 des failles de sécurité - **Top Ten** - basé sur des données recueillies auprès de plus de 40 organisations. Le but est d'éveiller à la sécurité en identifiant les risques de sécurité applicatifs Web les plus critiques auxquelles font face les organisations. Ce top 10 est devenu une référence dans l'industrie. Il a été mis à jour en 2017 et plus récemment en 2021.

L'OWASP, c'est aussi :

- WebGoat : une application délibérément mal sécurisée créée par l'OWASP. Elle peut être téléchargée et sert alors de plateforme de formation permettant d'apprendre à exploiter les vulnérabilités les plus courantes sur les applicaitons web dans l'intention d'apprendre comment écrire du code plus sécurisé ;
- Testing guide : le guide permet à un utilisateur d'implémenter des tests d'évaluation du niveau de sécurité d'une application Web ;
- Development guide : un guide de développement contenant des exemples couvrant de nombreuses typologies de failles, notamment celles que nous allons aborder dans ce cours ;
- Des librairies et outils de sécurité ;
- Et de nombreux autres guides.

2. Top 10 de l'OWASP en JavaScript

1. **Broken Access Control - Contrôle d'accès rompu**
2. **Cryptographic Failures - Défaillances cryptographiques**
3. **Injection**
4. **Insecure Design - Conception non sécurisée**
5. **Security Misconfiguration - Mauvaise configuration de la sécurité**
6. **Vulnerable and outdated Components - Composants vulnérables et obsolètes**
7. **Identification and Authentication Failures - Défauts d'identification et d'authentification**
8. **Software and Data integrity Failures - Défauts d'intégrité des logiciels et des données**
9. **Security Logging and monitoring Failures - Défaillances de la journalisation et de la surveillance de la sécurité**
10. **Server-Side Request Forgery (SSRF) - Falsification de requête côté serveur**

2.1 Broken Access Control - Contrôle d'accès rompu

Description

Le contrôle d'accès applique une politique telle que les utilisateurs ne peuvent pas agir en dehors de leurs autorisations prévues. Les défaillances entraînent généralement la divulgation non autorisée d'informations, la modification ou la destruction de toutes les données ou l'exécution d'une fonction commerciale en dehors des limites de l'utilisateur.

Les vulnérabilités courantes du contrôle d'accès incluent :

- Violation du principe du moindre privilège ou refus par défaut, où l'accès ne devrait être accordé qu'à des capacités, des rôles ou des utilisateurs particuliers, mais qui est accessible à tous ;
- Contourner les contrôles d'accès en modifiant l'URL (falsification des paramètres ou navigation forcée), l'état de l'application interne ou la page HTML, ou en utilisant un outil d'attaque modifiant les requêtes API ;
- Autoriser l'affichage ou la modification du compte de quelqu'un d'autre, en fournissant son identifiant unique (références d'objet directes non sécurisées) ;
- Accès à l'API avec des contrôles d'accès manquants pour POST, PUT et DELETE ;
- Élévation de privilèges. Agir en tant qu'utilisateur sans être connecté ou agir en tant qu'administrateur lorsqu'il est connecté en tant qu'utilisateur ;
- Manipulation des métadonnées, telle que la relecture ou la falsification d'un jeton de contrôle d'accès JSON Web Token (JWT), ou un cookie ou un champ masqué manipulé pour élever les privilèges ou abuser de l'invalidation JWT ;
- Une mauvaise configuration CORS permet l'accès à l'API à partir d'origines non autorisées/non approuvées ;
- Forcez la navigation vers des pages authentifiées en tant qu'utilisateur non authentifié ou vers des pages d'administration en tant qu'utilisateur standard.

Comment éviter ces attaques ?

Le contrôle d'accès n'est efficace que dans le code de confiance côté serveur ou l'API sans serveur (*server-less API*), où l'attaquant ne peut pas modifier la vérification du contrôle d'accès ou les métadonnées.

- Sauf pour les ressources publiques, refuser par défaut ;
- Implémentez des mécanismes de contrôle d'accès une seule fois et réutilisez-les dans l'ensemble de l'application, notamment en minimisant l'utilisation du partage de ressources

cross-origin (CORS). Ces mécanismes sont souvent utilisés via des *middlewares*, voir l'exemple ci-dessous ;

- Les modèles de contrôles d'accès doivent imposer la propriété des enregistrements plutôt que d'accepter que l'utilisateur puisse créer, lire, mettre à jour ou supprimer tout enregistrement ;
- Les modèles de domaine doivent faire respecter les limites commerciales des applications uniques ;
- Désactivez la liste des répertoires du serveur Web et assurez-vous que les métadonnées des fichiers (par exemple, .git) et les fichiers de sauvegarde ne sont pas présents dans les racines Web ;
- Enregistrez les échecs de contrôle d'accès, alertez les administrateurs le cas échéant (par exemple, les échecs répétés) ;
- Limitez le débit de l'API et de l'accès au contrôleur pour minimiser les dommages causés par les outils d'attaque automatisés ;
- Les identifiants de session avec état doivent être invalidés sur le serveur après la déconnexion. Les jetons JWT sans état devraient plutôt être de courte durée afin que la fenêtre d'opportunité pour un attaquant soit minimisée. Pour les JWT de plus longue durée, il est fortement recommandé de suivre les normes OAuth pour révoquer l'accès.

Les développeurs et le personnel d'assurance qualité devraient inclure une unité de contrôle d'accès fonctionnel et des tests d'intégration.

Les routes et autorisations côté client ne devraient être implémentées que pour une expérience utilisateur. Les contrôles d'authentification et d'autorisation implémentés côté client peuvent toujours être contournés.

Tous les contrôles d'autorisation, d'authentification et de logique commerciale doivent être réalisés côté serveur.

Important : ne jamais croire le client !

Exemples

L'application utilise des données non vérifiées dans ses appels en base de données pour accéder à des informations de compte :

```JavaScript
pstmt.setString(1, request.getParameter("compte"));
ResultSet results = pstmt.executeQuery( );
```

Un attaquant peut simplement modifier le paramètre `compte` du navigateur pour envoyer n'importe quel numéro de compte qu'il souhaite. Si ce n'est pas vérifié correctement, l'attaquant pourra accéder au compte de n'importe quel utilisateur.

```
https://example.com/app/accountInfo?compte=comptedebidule
```

Un attaquant peut aussi forcer le navigateur à cibler des adresses URL. Des droits d'administrateur sont requis pour accéder à la page d'administration.

```
https://example.com/app/getappInfo
https://example.com/app/admin_getappInfo
```

Si un utilisateur non authentifié peut accéder à l'une ou l'autre de ces pages, il s'agit d'une faille. Si un utilisateur non administrateur peut accéder à la page d'administration, il s'agit d'une faille.

Pour éviter cela, réalisez des contrôles d'autorisation sur vos points d'API, via notamment des *middlewares* (ci-dessous, `auth.requiresRole('admin')`).

```JavaScript
app.get('/api/users/:id', auth.requiresRole('admin'), admin.getUserById);
app.get('/api/users/edit/:id', auth.requiresRole('admin'));
app.put('/api/users/:id', auth.requiresRole('admin'), admin.updateUser);
app.delete('/api/users/:id', auth.requiresRole('admin'), admin.deleteUser);
app.post('/api/admin/logs', auth.requiresRole('admin'), admin.checkTmpFolder);
```

Pour aller plus loin :

node-casbin - **https://github.com/casbin/node-casbin**

passport - **https://github.com/jaredhanson/passport**

auth0 - **https://github.com/auth0/passport-auth0**

2.2 Cryptographic Failures - Défaillances cryptographiques

Description

La cryptographie est le procédé de lecture ou d'écriture de messages secrets ou de codes. Elle est utilisée pour rendre un message inintelligible à autre que le destinataire.

La première chose est de déterminer les besoins de protection des données en transit et au repos. Par exemple, les mots de passe, les numéros de carte de crédit, les dossiers médicaux, les informations personnelles et les secrets commerciaux nécessitent une protection supplémentaire, principalement si ces données relèvent des lois sur la confidentialité, par exemple, le règlement général sur la protection des données (RGPD) de l'UE, ou des réglementations, par exemple, la

protection des données financières comme la norme de sécurité des données PCI (PCI DSS pour *PCI Data Security Standard*). Pour toutes ces données, posez-vous ces questions :

- Des données sont-elles transmises en clair ? Cela concerne les protocoles tels que HTTP, SMTP, FTP utilisant également des mises à jour TLS comme STARTTLS. Le trafic Internet externe est dangereux. Vérifiez tout le trafic interne, par exemple entre les répartiteurs de charge (*load balancers*), les serveurs Web ou les systèmes principaux ;
- Des algorithmes ou protocoles cryptographiques anciens ou faibles sont-ils utilisés par défaut ou dans un code plus ancien ?
- Des clés de chiffrement par défaut sont-elles utilisées, des clés de chiffrement faibles sont-elles générées ou réutilisées, ou manque-t-il une gestion ou une rotation appropriée des clés ? Les clés de chiffrement sont-elles archivées dans les dépôts de code source (sur github ou autre) ?
- Le chiffrement est-il forcé ? Par exemple, des directives de sécurité ou des en-têtes HTTP (navigateur) sont-ils manquants ?
- Le certificat de serveur reçu et la chaîne de confiance sont-ils correctement validés ?
- Les vecteurs d'initialisation (*initialization vector ou IV*, bloc de bits combiné avec le premier bloc de données lors d'une opération de chiffrement) sont-ils ignorés, réutilisés ou générés insuffisamment sécurisés pour le mode de fonctionnement cryptographique ? Un mode de fonctionnement non sécurisé tel que l'ECB est-il utilisé ? Le chiffrement est-il utilisé lorsque le chiffrement authentifié est plus approprié ?
- Les mots de passe sont-ils utilisés comme clés cryptographiques en l'absence d'une fonction de dérivation de la clé de base du mot de passe ?
- Le caractère aléatoire est-il utilisé à des fins cryptographiques qui n'ont pas été conçues pour répondre aux exigences cryptographiques (par exemple utilisation de uuid pour de la cryptographie) ? Même si la fonction correcte est choisie, doit-elle être *salée* par le développeur, et si ce n'est pas le cas, le développeur a-t-il écrasé la fonctionnalité intégrée de salage avec un sel qui manque d'entropie/imprévisibilité suffisante ?
- Des fonctions de hachage obsolètes telles que MD5 ou SHA1 sont-elles utilisées, ou des fonctions de hachage non cryptographiques sont-elles utilisées lorsque des fonctions de hachage cryptographiques sont nécessaires ?
- Des méthodes de remplissage cryptographique obsolètes telles que PKCS numéro 1 v1.5 sont-elles utilisées ?
- Les messages d'erreur cryptographiques ou les informations de canal auxiliaire sont-ils exploitables, par exemple sous la forme d'attaques de type *padding oracle* (envoi de données manipulées par l'attaquant et observation des résultats fournis pour trouver la clé de chiffrement) ?

Comment éviter ces attaques ?

Procédez comme suit, au minimum, et consultez les références :

- Classer les données traitées, stockées ou transmises par une application. Identifiez les données sensibles selon les lois sur la confidentialité, les exigences réglementaires ou les besoins de l'entreprise ;
- Ne stockez pas de données sensibles inutilement. Supprimez-les dès que possible ou utilisez la tokenisation conforme à la norme PCI DSS ou même la troncature. Les données non conservées ne peuvent pas être volées ;
- Assurez-vous de chiffrer toutes les données sensibles au repos (dans une base de données, un fichier etc.) ;
- S'assurer que des algorithmes, des protocoles et des clés standard à jour et solides sont en place ; utiliser une bonne gestion des clés.

- Chiffrez toutes les données en transit avec des protocoles sécurisés tels que TLS avec des chiffrements de confidentialité persistante (FS pour *forward secrecy*), la hiérarchisation des chiffrements par le serveur et des paramètres sécurisés. Appliquez le chiffrement à l'aide de directives telles que HTTP Strict Transport Security (HSTS, cela force le navigateur à utiliser HTTPS) ;
- Désactivez la mise en cache pour les réponses contenant des données sensibles ;
- Appliquer les contrôles de sécurité requis selon la classification des données ;
- N'utilisez pas de protocoles hérités tels que FTP et SMTP pour le transport de données sensibles ;
- Stockez les mots de passe à l'aide de fonctions de hachage adaptatives et fortement salées avec un facteur de travail (facteur de retard), comme Argon2, scrypt, bcrypt ou PBKDF2 ;
- Les vecteurs d'initialisation doivent être choisis en fonction du mode de fonctionnement. Pour de nombreux modes, cela signifie utiliser un CSPRNG (*cryptographically secure pseudo random number generator*, en français générateur de nombres pseudo-aléatoires cryptographiquement sécurisé). Pour les modes qui nécessitent un *nonce* (*number used once*, un nombre à usage unique), le vecteur d'initialisation (IV) n'a pas besoin d'un CSPRNG. Dans tous les cas, le IV ne doit jamais être utilisé deux fois pour une clé fixe ;
- Utilisez toujours un chiffrement authentifié au lieu d'un simple chiffrement ;
- Les clés doivent être générées cryptographiquement de manière aléatoire et stockées en mémoire sous forme de tableaux d'octets. Si un mot de passe est utilisé, alors il doit être converti en clé via une fonction appropriée de dérivation de clé basée sur de mot de passe ;
- Assurez-vous que l'aléatoire cryptographique est utilisé le cas échéant et qu'il n'a pas été ensemencé de manière prévisible ou avec une faible entropie. La plupart des API modernes n'exigent pas que le développeur amorce le CSPRNG pour obtenir la sécurité ;
- Évitez les fonctions cryptographiques obsolètes et les schémas de remplissage, tels que MD5, SHA1, PKCS numéro 1 v1.5 ;
- Vérifier indépendamment l'efficacité de la configuration et des paramètres.

Exemple

Voici un exemple sur les risques liés à la divulgation de données sensibles.

Il est facile d'écrire ce code pour retourner une erreur

```JavaScript
try {
    // Fait quelques chose
} catch (e) {
    return e;
}
```

Sur une route mal protégée, voici le message que l'on peut récupérer en entrant volontairement des données incorrectes

```
{"user": {"id": "DB2 SQL-ERROR: -104 ILLEGAL SYMBOL SOME SYMBOLS THAT MIGHT BE LEGAL ARE
```

N'incluez pas de messages d'erreurs verbeux qui pourraient mener à une divulgation des informations de votre système.

Bien que votre application client affiche des messages d'erreurs génériques, le JSON de réponse peut toujours contenir le message d'erreur entier.

Les utilisateurs malveillants peuvent utiliser un proxy pour lire la sortie de la trace de pile en JSON.

Attention : les informations système détaillées peuvent ne pas sembler significative à première vue. Cependant, elles peuvent informer les attaquants sur les systèmes internes et l'infrastructure, et les aider à lancer de futures attaques.

Ce code montre un message d'erreur SQL précédemment passé en JSON, avec le code JavaScript utilisé pour le masquer.

```JavaScript
const executeur = new Promise((resolve, reject) => {
    connection.query('SELECT * FROM utilisateurs WHERE utilisateurID=?', [obj.utilisateur
        if (erreur) {
            logger.log(`SQL ERROR: ${erreur}`);
            // Plutôt que de renvoyer l'erreur produite
            // On produit un message personnalisé sans
            // informations détaillées sur la nature
            // de l'erreur
            reject(`La requête n'a pas pu être exécutée.`);
        }
        resolve(resultats);
    });
});
```

2.3 Injection

Description

L'injection est un groupe de méthodes d'exploitation de failles de sécurité lors de l'interaction avec une base de données. Elle permet d'injecter dans la requête en cours un morceau de requête non prévu par le système et pouvant en compromettre la sécurité.

L'application est vulnérable à l'injection quand :

- Les données fournies par l'utilisateur ne sont pas validées, filtrées ou purgée par l'application ;
- Les requêtes non paramétrées sans échappement contextuel sont utilisées directement dans l'interpréteur ;
- Des données hostiles sont utilisées dans les paramètres de recherche (query params) pour extraire des données sensibles ;
- Des données hostiles sont directement utilisées ou concaténées. La requête ou la commande contient la structure et les données malveillantes dans les requêtes dynamiques, les commandes ou les procédures stockées.

Comment éviter ces attaques ?

Pour prévenir ces attaques il faut séparer les données des commandes et requêtes :

- En utilisant des API sécurisées, qui évitent l'utilisation de l'interpréteur, fournissent une interface paramétrée, ou qui migrent vers des outils de mappage d'objets relationnel ;

- Utiliser des procédures stockées. Les données entrées par l'utilisateur sont alors transmises comme paramètres, qui évitent l'injection s'ils sont correctement utilisés par la procédure ;
- Vérifier côté serveur les données reçues. Les expressions rationnelles permettent de valider que le format est bien celui attendu, ou le langage fournit des fonctions de transformation ;
- Utiliser les mots-clefs du langage comme LIMIT ou d'autres contrôles permettent d'éviter une fuite massive des données en cas de succès de l'injection.

Exemples

Injection SQL

```JavaScript
var mysql = require('mysql');

const obj = JSON.parse(req.body);

connexion.query(
    "SELECT * FROM utilisateurs WHERE uid='" + obj.utilisateur + "'",
    function (erreur, resultats) {
        console.log(resultats);
    }
);
```

L'attaquant peut ici entrer une injection SQL en entrant du code qui sera interprété par le langage SQL, par exemple `' or 1 --`. L'apostrophe aura pour effet de terminer la zone de frappe de l'utilisateur, `or 1` demande au script si 1 est vrai (rappelez-vous, 1 est *truthy*), et `--` indique le début d'une zone de commentaire.

Les requêtes paramétrées permettent d'éviter cela.

```JavaScript
var mysql = require('mysql');

const obj = JSON.parse(req.body);

connexion.query(
    'SELECT * FROM utilisateurs WHERE uid=?', [obj.utilisateur],
    function (erreur, resultats) {
        console.log(resultats);
    }
);
```

Injection MongoDB

L'utilisateur insère un sélecteur de requête MongoDB tels que :

```JavaScript
$ne, $lt, $gt, $eq, $regex, etc.
```

La saisie de l'utilisateur est directement insérée dans la méthode de collection.

```JavaScript
find, findOne, findOneAndUpdate, etc.
```

Prenons le code non sécurisé ci-dessous :

```JavaScript
db.collection('collection').findOne({
    nom: req.query.nom,
    motDePasse: req.query.motDePasse,
    actif: true,
});
```

Que se passe-t-il si l'utilisateur injecte cet URL : **https://url.to/login**?nom=admin&motDePasse[$ne]= ?

Les variables seront remplacées comme suit dans la requête :

```JavaScript
db.collection('collection').findOne({
    nom: "admin",
    motDePasse: {
        $ne: "",
    },
    actif: true,
});
```

$ne sélectionne les documents où la valeur du champ n'est pas égale à la valeur spécifiée. L'utilisateur admin possédant un mot de passe, la condition sera toujours vérifiée.

La solution est de **toujours** valider et purger les entrées utilisateur !

Assurez-vous que l'entrée utilisateur soit un *String* au sein de la méthode de collection.

```JavaScript
db.collection('collection').findOne({
    nom: String(req.query.nom),
    motDePasse: String(req.query.motDePasse),
    actif: true,
});
```

2.4 Insecure Design - Conception non sécurisée

Description

La conception non sécurisée est une vaste catégorie représentant différentes faiblesses, exprimées comme "conception de contrôle manquante ou inefficace". La conception non sécurisée n'est pas à l'origine de toutes les autres catégories de risques du Top 10. Il existe une différence entre une conception non sécurisée et une mise en œuvre non sécurisée. Nous faisons la différence entre les défauts de conception et les défauts de mise en œuvre pour une raison, ils ont

des causes profondes et des solutions différentes. Une conception sûre peut néanmoins présenter des défauts de mise en œuvre conduisant à des vulnérabilités qui peuvent être exploitées. Une conception non sécurisée ne peut être corrigée par une mise en œuvre parfaite puisque, par définition, les contrôles de sécurité nécessaires n'ont jamais été créés pour se défendre contre des attaques spécifiques. L'un des facteurs qui contribuent à une conception non sécurisée est l'absence de profilage des risques commerciaux inhérents au logiciel ou au système en cours de développement, et donc l'incapacité à déterminer le niveau de sécurité requis.

Gestion des exigences et des ressources

Recueillez et négociez avec l'entreprise les exigences commerciales d'une application, y compris les exigences de protection concernant la confidentialité, l'intégrité, la disponibilité et l'authenticité de tous les actifs de données et la logique commerciale attendue. Tenez compte du degré d'exposition de votre application et de la nécessité éventuelle d'une séparation des composants (en plus du contrôle d'accès). Compilez les exigences techniques, y compris les exigences de sécurité fonctionnelles et non fonctionnelles. Planifiez et négociez le budget couvrant l'ensemble de la conception, de la construction, des tests et de l'exploitation, y compris les activités de sécurité.

Conception sécurisée

La conception sécurisée est une culture et une méthodologie qui évaluent constamment les menaces et garantissent que le code est conçu et testé de manière robuste afin de prévenir les méthodes d'attaque connues. La modélisation des menaces doit être intégrée aux sessions de perfectionnement (ou à des activités similaires) ; recherchez les changements dans les flux de données et le contrôle d'accès ou d'autres contrôles de sécurité. Lors de l'élaboration d'une description fonctionnelle (*user story*), déterminez le flux correct et les états d'échec, assurez-vous qu'ils sont bien compris et acceptés par les parties responsables et touchées. Analysez les hypothèses et les conditions pour les flux attendus et les états d'échec, assurez-vous qu'elles sont toujours exactes et souhaitables. Déterminez comment valider les hypothèses et appliquer les conditions nécessaires aux comportements appropriés. Assurez-vous que les résultats sont documentés dans la description fonctionnelle. Tirez les leçons de vos erreurs et proposez des incitations positives pour promouvoir les améliorations. La conception sécurisée n'est ni un complément ni un outil que l'on peut ajouter au logiciel.

Cycle de vie du développement sécurisé

Un logiciel sécurisé nécessite un cycle de développement sécurisé, une certaine forme de modèle de conception sécurisé, une méthodologie de route pavée, une bibliothèque de composants sécurisés, des outils et une modélisation des menaces. Faites appel à vos spécialistes de la sécurité dès le début d'un projet logiciel, tout au long du projet et de la maintenance de votre logiciel. Envisagez d'utiliser le modèle de maturité d'assurance logicielle (SAMM pour *Software Assurance Maturity Model*) de l'OWASP pour vous aider à structurer vos efforts de développement de logiciels sécurisés.

Comment éviter ces attaques ?

- Établir et utiliser un cycle de vie de développement sécurisé avec des professionnels de l'AppSec pour aider à évaluer et concevoir des contrôles de sécurité et de confidentialité ;
- Créez et utilisez une bibliothèque de modèles de conception sécurisés ou de composants prêts à l'emploi ;
- Utiliser la modélisation des menaces pour l'authentification critique, le contrôle d'accès, la logique métier et les flux de clés ;
- Intégrer le langage et les contrôles de sécurité dans les descriptions fonctionnelles ;
- Intégrer des contrôles de plausibilité à chaque niveau de votre application (du front-end au back-end) ;
- Rédiger des tests unitaires et d'intégration pour valider que tous les flux critiques résistent au modèle de menace. Compilez les cas d'utilisation et de mauvaise utilisation pour chaque niveau de votre application ;

- Séparez les couches de niveau sur les couches système et réseau en fonction de l'exposition et des besoins de protection ;
- Séparez les composants de manière robuste par conception dans tous les niveaux ;
- Limitez la consommation de ressources par utilisateur ou service.

Exemple

Une chaîne de cinémas accorde des remises pour les réservations de groupe et ne demande qu'un maximum de quinze participants avant d'exiger un acompte. Les attaquants pourraient modéliser ce flux et tester s'ils peuvent réserver six cents places et tous les cinémas à la fois en quelques requêtes, ce qui entraînerait une perte de revenus massive.

2.5 Security Misconfiguration - Mauvaise configuration de la sécurité

Description

Des failles peuvent apparaître avec une mauvaise configuration de votre environnement. L'application peut être vulnérable si :

- Elle manque d'un durcissement de sécurité approprié sur n'importe quelle partie de la pile d'applications ou de permissions mal configurées sur les services cloud ;
- Des fonctionnalités inutiles sont activées ou installées (par exemple, des ports, services, pages, comptes ou privilèges inutiles) ;
- Les comptes par défaut et leurs mots de passe sont toujours activés et inchangés ;
- Le traitement des erreurs révèle des traces de pile ou d'autres messages d'erreur trop informatifs pour les utilisateurs (voir **l'exemple du chapitre 2** ;
- Pour les systèmes mis à niveau, les dernières fonctions de sécurité sont désactivées ou ne sont pas configurées de manière sécurisée ;
- Les paramètres de sécurité des serveurs d'application, des cadres d'application (par exemple, Struts, Spring, ASP.NET), des bibliothèques, des bases de données, etc. ne sont pas définis sur des valeurs sécurisées ;
- Le serveur n'envoie pas d'en-têtes ou de directives de sécurité, ou ceux-ci ne sont pas définis sur des valeurs sûres ;
- Le logiciel est obsolète ou vulnérable (voir **le chapitre 6**).

En l'absence d'un processus de configuration de la sécurité des applications concerté et reproductible, les systèmes sont exposés à un risque plus élevé.

Comment éviter ces attaques ?

Des processus d'installation sécurisés doivent être mis en œuvre, notamment :

- Un processus de durcissement répétable permet de déployer rapidement et facilement un autre environnement verrouillé de manière appropriée. Les environnements de développement, d'assurance qualité et de production doivent tous être configurés de manière identique, avec des informations d'identification différentes utilisées dans chaque environnement. Ce processus doit être automatisé pour minimiser l'effort nécessaire à la mise en place d'un nouvel environnement sécurisé ;
- Une plateforme minimale sans fonctionnalités, composants, documentation et échantillons inutiles. Supprimez ou n'installez pas les fonctionnalités et les frameworks inutilisés ;
- Une tâche pour examiner et mettre à jour les configurations appropriées à toutes les notes de sécurité, mises à jour et correctifs dans le cadre du processus de gestion des correctifs (voir **le chapitre 6**). Passez en revue les autorisations de stockage dans le cloud (par exemple, les autorisations de seau S3) ;

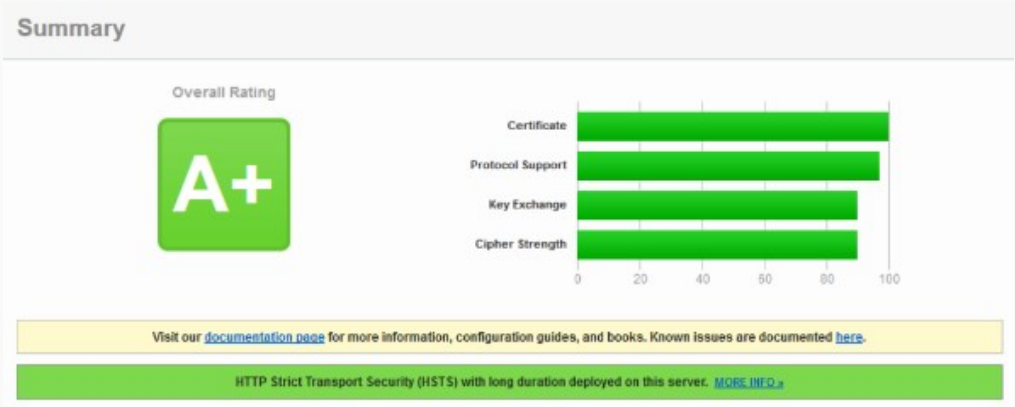
- Une architecture d'application segmentée fournit une séparation efficace et sécurisée entre les composants, avec la segmentation, la conteneurisation ou les groupes de sécurité du cloud (ACL) ;
- L'envoi de directives de sécurité aux clients, par exemple, les en-têtes de sécurité ;
- Un processus automatisé pour vérifier l'efficacité des configurations et des paramètres dans tous les environnements.

Exemple

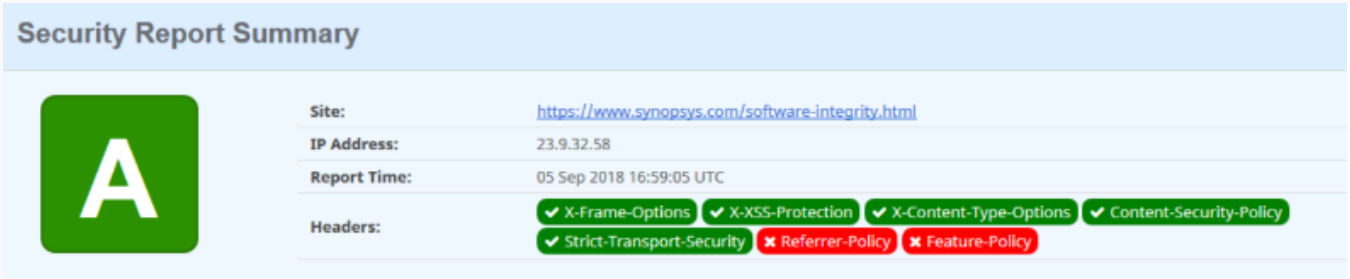
Le serveur d'applications est livré avec des exemples d'applications qui ne sont pas retirées du serveur de production. Ces exemples d'applications présentent des failles de sécurité connues que les attaquants utilisent pour compromettre le serveur. Supposons que l'une de ces applications soit la console d'administration et que les comptes par défaut n'aient pas été modifiés. Dans ce cas, l'attaquant se connecte avec les mots de passe par défaut et prend le contrôle.

Il existe des outils en ligne pour tester la sécurité de votre configuration :

Outils pour tester votre configuration SSL/TLS - **https://www.ssllabs.com/ssltest/**



Outil pour vérifier la sécurité de vos en-têtes HTTP - **https://securityheaders.com/**



2.6 Vulnerable and outdated Components - Composants vulnérables et obsolètes

Description

Vous êtes probablement vulnérable :

- Si vous ne connaissez pas les versions de tous les composants que vous utilisez (côté client et côté serveur). Cela inclut les composants que vous utilisez directement ainsi que les dépendances imbriquées ;
- Si le logiciel est vulnérable, non pris en charge ou obsolète. Cela inclut le système d'exploitation, le serveur web/application, le système de gestion de base de données (SGBD), les applications, les API et tous leurs composants, les environnements d'exécution et les bibliothèques ;
- Si vous ne recherchez pas régulièrement les vulnérabilités et ne vous abonnez pas aux bulletins de sécurité relatifs aux composants que vous utilisez ;

- Si vous ne corrigez pas ou ne mettez pas à niveau la plate-forme sous-jacente, les cadres et les dépendances en fonction des risques et en temps voulu. Cela se produit souvent dans les environnements où l'application de correctifs est une tâche mensuelle ou trimestrielle dans le cadre du contrôle des changements, ce qui expose les organisations à des jours ou des mois d'exposition inutile aux vulnérabilités corrigées ;

- Si les développeurs de logiciels ne testent pas la compatibilité des bibliothèques mises à jour, mises à niveau ou corrigées ;

- Si vous ne sécurisez pas les configurations des composants (voir **le chapitre 5**).

Comment éviter ces attaques ?

Un processus de gestion des correctifs doit être mis en place pour :

- Supprimer les dépendances **inutilisées**, les fonctionnalités, les composants, les fichiers et la documentation **superflus** ;

- **Inventorier en permanence** les versions des composants côté client et côté serveur (par exemple, les frameworks, les bibliothèques) et leurs dépendances à l'aide d'outils tels que versions, OWASP Dependency Check, retire.js, etc. **Surveiller en permanence** les sources telles que Common Vulnerability and Exposures (CVE) et National Vulnerability Database (NVD) pour détecter les vulnérabilités dans les composants. Utilisez des outils d'analyse de la composition des logiciels pour automatiser le processus. Souscrivez à des alertes e-mail pour les vulnérabilités de sécurité liées aux composants que vous utilisez.

- N'obtenez les composants qu'auprès de sources officielles par des liens sécurisés. Préférez les paquets signés pour réduire le risque d'inclure un composant modifié ou malveillant (voir **le chapitre 8**).

- Surveillez les bibliothèques et les composants qui ne sont pas maintenus ou qui ne créent pas de correctifs de sécurité pour les anciennes versions. Si l'application de correctifs n'est pas possible, envisagez de déployer un correctif virtuel pour surveiller, détecter ou protéger contre le problème découvert.

Chaque organisation doit mettre en place un plan permanent de surveillance, de triage et d'application des mises à jour ou des modifications de configuration pendant toute la durée de vie de l'application ou du portefeuille.

Concernant les composants tiers :

La réutilisation des composants rend le développement plus facile. Mais ces composants ne sont pas exempt de failles de sécurité.

- Le site NPM contient des composant tiers ;

- Une notation élevée sur GitHub ne garanti pas que le composant est sécurisé ;

Vérifiez toujours les publications de sécurité connues :

- GitHub signale automatiquement les publications de sécurité ;

- En fonction du projet utilisez ces outils :

Exemple	Commande
npm	npm audit --fix
yarn	yarn audit --fix
bower	auditjs --bower bower.json
JavaScript côté client	retire --js /chemin/
Node.js Open-Source	snyk test

Exemple

Les composants s'exécutent généralement avec les mêmes privilèges que l'application elle-même, de sorte que des failles dans n'importe quel composant peuvent avoir des conséquences

graves. Ces failles peuvent être accidentelles (par exemple, une erreur de codage) ou intentionnelles (par exemple, une porte dérobée dans un composant). Quelques exemples de vulnérabilités de composants exploitables découvertes sont :

- CVE-2017-5638, une vulnérabilité d'exécution de code à distance de Struts 2 qui permet l'exécution de code arbitraire sur le serveur, a été mise en cause dans des brèches importantes ;
- Bien que l'internet des objets (IoT) est difficile ou impossible à patcher, l'importance de le faire peut être critique (par exemple, les appareils biomédicaux).

Il existe des outils automatisés qui aident les attaquants à trouver des systèmes non patchés ou mal configurés. Par exemple, le moteur de recherche Shodan IoT peut vous aider à trouver des appareils qui souffrent encore de la vulnérabilité Heartbleed patchée en avril 2014.

2.7 Identification and Authentication Failures - Défauts d'identification et d'authentification

Description

La vérification de l'identité de l'utilisateur, de son authentification et la gestion de sa session sont des éléments critiques pour se protéger des attaques liées à l'authentification. Un attaquant peut exploiter une faille pour se faire passer pour un utilisateur légitime. Des failles peuvent exister si l'application :

- Permet d'automatiser des attaques telles que le bourrage d'identifiant (*credential stuffing*), où l'attaquant a une liste d'identifiants et de mots de passe valides ;
- Permet des attaques par *brute force* ou d'autres mécanismes automatisés ;
- Permet l'utilisation de mots de passe faibles ou très bien connus comme `Password1` ou `admin/admin` ;
- Utilise des processus de récupération d'informations d'identification faible ou inefficaces et de mot de passe oublié, tels que les réponses basées sur des questions (*Quel était le nom de votre premier chat ?*), qui ne peuvent pas être sécurisées ;
- Utilise un stockage du mot de passe en clair ou faiblement haché (*hash*). Voir **Cryptographic Failures** ;
- Possède une authentification multifacteurs manquante ou inefficace ;
- Expose l'identifiant de session dans l'URL ;
- Réutilise l'identifiant de session après une connexion réussie ;
- N'invalide pas correctement les identifiants de session. Les sessions utilisateur ou les jetons d'authentification (principalement les jetons d'authentification uniques (SSO pour *Single Sign-On*)) ne sont pas correctement invalidés lors de la déconnexion ou lors d'une période d'inactivité.

Comment éviter ces attaques ?

- Dans la mesure du possible, mettez en œuvre une authentification multifacteur pour empêcher le bourrage d'identifiant, le *brute force* et les attaques de réutilisation d'informations d'identification volées ;
- N'expédiez pas ou ne déployez pas avec des informations d'identification par défaut, en particulier pour les administrateurs ;
- Mettez en place des contrôles de mots de passe faibles, tels que le test des mots de passe nouveaux ou modifiés par rapport à la liste des 10 000 pires mots de passe ;
- Alignez les politiques de longueur, de complexité et de rotation des mots de passe avec les directives 800-63b du *National Institute of Standards and Technology (NIST)* dans la section 5.1.1 ;

- Assurez-vous que l'enregistrement, la récupération des informations d'identification et les routes d'API sont renforcées contre les attaques par énumération en utilisant les mêmes messages pour tous les résultats ;
- Limitez ou retardez de plus en plus les tentatives de connexion infructueuses, mais veillez à ne pas créer de scénario de déni de service. Enregistrez tous les échecs et alertez les administrateurs lorsque le bourrage d'identifiant, le *brute force* ou d'autres attaques sont détectés ;
- Utilisez un gestionnaire de session intégré, sécurisé et côté serveur qui génère un nouvel ID de session aléatoire avec une entropie élevée après la connexion. L'identifiant de session ne doit pas figurer dans l'URL, être stocké en toute sécurité et invalidé après la déconnexion, l'inactivité et les délais d'expiration absolus.

Exemples

Que se passe-t-il si nous créons notre propre *middleware* d'authentification ?

JavaScript

```
const SESSIONS = {};

const doitEtreAuthentifie = (req, res, next) => {
    if (req.cookies) {
        const token = req.cookies.token;

        if (token && SESSIONS[token]) {
            // On l'autorise
            next();
        }
    }
    res.send(`Vous n'êtes pas autorisé !`);
};
```

Certains résultats peuvent être surprenants :

Valeur	Retour
SESSIONS['chaineNonValide']	False
SESSIONS['']	False
SESSIONS['constructor']	True
SESSIONS['hasOwnProperty']	True

Que se passe-t-il quand on crée un objet en JavaScript ?

JavaScript

```
const SESSIONS = {};

__proto__:
    constructor: function Object(),
    hasOwnProperty: function hasOwnProperty(),
    isPrototypeOf: function isPrototypeOf(),
    [...]

SESSIONS['constructor'] === SESSIONS.constructor // Retourne true
```

Comment vérifier correctement ?

```JavaScript
// ES6
SESSIONS.has('__proto__') // Retourne false
SESSIONS.has('chaineValide') // Retourne true

// Common JS
SESSIONS.hasOwnProperty['__proto__'] // Retourne false
SESSIONS.hasOwnProperty['chaineValide'] // Retourne true
```

2.8 Software and Data integrity Failures - Défauts d'intégrité des logiciels et des données

Description

Les défaillances de l'intégrité des logiciels et des données sont liées au code et à l'infrastructure qui ne sont pas protégés contre les violations de l'intégrité. C'est le cas, par exemple, lorsqu'une application s'appuie sur des plugins, des bibliothèques ou des modules provenant de sources, de dépôts et de réseaux de diffusion de contenu (CDN) non fiables. Un pipeline CI/CD non sécurisé peut introduire un risque d'accès non autorisé, de code malveillant ou de compromission du système. Enfin, de nombreuses applications comportent désormais une fonction de mise à jour automatique, qui permet de télécharger des mises à jour sans vérification suffisante de l'intégrité et de les appliquer à l'application précédemment approuvée. Les attaquants pourraient potentiellement télécharger leurs propres mises à jour pour les distribuer et les exécuter sur toutes les installations. Un autre exemple est celui des objets ou des données qui sont codés ou sérialisés dans une structure qu'un attaquant peut voir et modifier et qui sont vulnérables à une désérialisation non sécurisée.

Comment éviter ces attaques ?

- Utilisez des signatures numériques ou des mécanismes similaires pour vérifier que le logiciel ou les données proviennent de la source attendue et n'ont pas été modifiés ;
- Assurez-vous que les bibliothèques et les dépendances, telles que npm ou Maven, utilisent des dépôts de confiance. Si vous présentez un profil de risque plus élevé, envisagez d'héberger un dépôt interne connu et vérifié ;
- Assurez-vous qu'un outil de sécurité de la chaîne logistique logicielle, tel que OWASP Dependency Check ou OWASP CycloneDX, est utilisé pour vérifier que les composants ne contiennent pas de vulnérabilités connues.
- Assurez-vous qu'il existe un processus de révision des modifications de code et de configuration afin de minimiser les risques d'introduction de code ou de configuration malveillants dans votre pipeline logiciel ;
- Assurez-vous que votre pipeline CI/CD dispose d'une ségrégation, d'une configuration et d'un contrôle d'accès appropriés pour garantir l'intégrité du code passant par les processus de construction et de déploiement ;
- Veillez à ce que les données sérialisées non signées ou non cryptées ne soient pas envoyées à des clients non fiables sans une forme de contrôle d'intégrité ou de signature numérique pour détecter la falsification ou la répétition des données sérialisées.

Exemples

Mise à jour sans signature : De nombreux routeurs domestiques, décodeurs, firmwares de périphériques et autres ne vérifient pas les mises à jour via un firmware signé. Les microprogrammes non signés constituent une cible de plus en plus importante pour les attaquants

et leur nombre ne devrait cesser d'augmenter. Il s'agit d'une préoccupation majeure car, bien souvent, il n'existe aucun mécanisme permettant de remédier à la situation, si ce n'est d'intégrer la correction dans une version ultérieure et d'attendre que les versions précédentes soient dépassées.

Mise à jour malveillante de SolarWinds : Les États-nations sont connus pour s'attaquer aux mécanismes de mise à jour. L'une des dernières attaques notables est celle de SolarWinds Orion. La société qui développe le logiciel disposait de processus sécurisés de construction et d'intégrité des mises à jour. Ces processus ont néanmoins pu être contournés et, pendant plusieurs mois, la société a distribué une mise à jour malveillante très ciblée à plus de 18 000 organisations, dont une centaine ont été touchées. Il s'agit de l'une des violations de cette nature les plus étendues et les plus importantes de l'histoire.

2.9 Security Logging and monitoring Failures - Défaillances de la journalisation et de la surveillance de la sécurité

Cette catégorie a pour but d'aider à la détection et à la réponse aux brèches actives. Sans journalisation et surveillance, les brèches ne peuvent être détectées. Une journalisation, une détection, une surveillance et une réponse active insuffisantes peuvent survenir à tout moment :

- Les événements vérifiables, tels que les connexions, les échecs de connexion et les transactions de grande valeur, ne sont pas enregistrés ;
- Les avertissements et les erreurs ne génèrent aucun message de journalisation, ou des messages inadéquats ou peu clairs ;
- Les journaux d'applications et d'API ne sont pas surveillés pour détecter toute activité suspecte ;
- Les journaux sont uniquement stockés localement ;
- Des seuils d'alerte appropriés et des processus d'escalade de réponse ne sont pas en place ou efficaces ;
- Les tests de pénétration et les analyses effectuées par des outils de test dynamique de la sécurité des applications (DAST pour *dynamic application security testing*) (tels que OWASP ZAP) ne déclenchent pas d'alertes ;
- L'application ne peut pas détecter, faire remonter ou alerter des attaques actives en temps réel ou quasi réel.

Vous êtes vulnérable aux fuites d'informations en rendant les événements de journalisation et d'alerte visibles pour un utilisateur ou un attaquant (voir **le chapitre 1**).

Comment éviter ces attaques ?

Les développeurs doivent mettre en œuvre tout ou partie des contrôles suivants, en fonction du risque de l'application :

- S'assurer que tous les échecs de connexion, de contrôle d'accès et de validation des entrées côté serveur peuvent être consignés avec un contexte utilisateur suffisant pour identifier les comptes suspects ou malveillants et conservés suffisamment longtemps pour permettre une analyse technique différée ;
- Assurez-vous que les journaux sont générés dans un format que les solutions de gestion des journaux peuvent facilement utiliser ;
- Veillez à ce que les données des journaux soient correctement codées pour éviter les injections ou les attaques sur les systèmes de journalisation ou de surveillance ;
- Veillez à ce que les transactions de grande valeur disposent d'une piste d'audit avec des contrôles d'intégrité afin d'éviter toute altération ou suppression, comme des tables de base de données en appendice seulement ou similaires ;
- Les équipes DevSecOps doivent mettre en place une surveillance et des alertes efficaces afin de détecter les activités suspectes et d'y répondre rapidement ;

- Établissez ou adoptez un plan de réponse aux incidents et de récupération, tel que le National Institute of Standards and Technology (NIST) 800-61r2 ou une version ultérieure.

Il existe des cadres de protection des applications commerciales et open-source, comme le ModSecurity Core Rule Set de l'OWASP, et des logiciels de corrélation de logs open-source, comme Elasticsearch, Logstash, Kibana (ELK), qui proposent des tableaux de bord et des alertes personnalisés.

Exemples

L'injection des journaux est une méthode d'attaque régulière qui consiste à créer des informations de log ou à attaque les administrateurs lisant les journaux.

Les bibliothèques de journalisation JavaScript log4js et console.log sont vulnérables aux attaques d'injection des journaux.

Exemple avec ces données à injecter :

```
http://exemple.com/?texte=coucou%0DARecu:%20donneescompromises
```

Avec ce code JavaScript :

```JavaScript
app.get('/', function (req, res) {
    console.log("Recu : "+req.query.texte)
})
```

Voici ce qui s'affichera dans les journaux :

```Bash
Recu : coucou
Recu : donneescompromises
```

Pour éviter cela, utilisez la bibliothèque **Winston**, qui échappe les données pour éviter l'injection des journaux.

```
{
    "level":"info",
    "message":"Recu : coucou\r\nRecu : donneescompromises",
    "timestamp":"2022-20-01T01:09:57.024Z"
}
```

Winston fournit aussi plusieurs contrôles de sécurité intégrés, comme les filtres de contenu :

```JavaScript
var logger = new Winston.logger({
    filters: [
        function(level, msg, meta) {
```

```
            return maskCardNumbers(msg);
        }
    ]
});
winston.log('info', "Panier 12345678912345 traité.");
```

Ce qui donnera :

```Bash
info: Panier 123456****2345 traité.
```

2.10 Server-Side Request Forgery (SSRF) - Falsification de requête côté serveur

Description

La faille SSRF se produit lorsqu'une application web récupère une ressource distante sans valider l'URL fournie par l'utilisateur. Elle permet à un attaquant de contraindre l'application à envoyer une requête élaborée à une destination inattendue, même si elle est protégée par un pare-feu, un VPN ou un autre type de liste de contrôle d'accès au réseau (ACL).

Comme les applications web modernes offrent aux utilisateurs finaux des fonctions pratiques, la récupération d'une URL devient un scénario courant. Par conséquent, l'incidence du SSRF augmente. De même, la gravité de ce phénomène augmente en raison des services cloud et de la complexité des architectures.

Comment éviter ces attaques ?

Les développeurs peuvent prévenir le SSRF en mettant en œuvre tout ou partie des contrôles de défense en profondeur suivants :

De la couche réseau

- Segmentez la fonctionnalité d'accès aux ressources à distance dans des réseaux distincts afin de réduire l'impact du SSRF ;
- Appliquez des politiques de pare-feu ou des règles de contrôle d'accès au réseau "refus par défaut" pour bloquer tout le trafic intranet sauf celui qui est essentiel.

  Conseils :

  - Établissez une propriété et un cycle de vie pour les règles de pare-feu en fonction des applications ;
  - Consignez tous les flux réseau acceptés et bloqués sur les pare-feu (voir **le chapitre 9**).

Depuis la couche Application :

- Nettoyez et validez toutes les données d'entrée fournies par le client ;
- Appliquez le schéma URL, le port et la destination avec une liste d'autorisations ;
- N'envoyez pas de réponses brutes aux clients ;
- Désactivez les redirections HTTP ;
- Veillez à la cohérence de l'URL pour éviter les attaques telles que le rebinding DNS et les conditions de course "time of check, time of use" (TOCTOU).

N'atténuez pas le SSRF par l'utilisation d'une liste de refus ou d'une expression régulière. Les attaquants disposent de listes de charges utiles, d'outils et de compétences pour contourner les

listes de refus.

Mesures supplémentaires à prendre en compte :

- Ne déployez pas d'autres services pertinents pour la sécurité sur les systèmes frontaux (par exemple, OpenID). Contrôlez le trafic local sur ces systèmes (par exemple, localhost) ;
- Pour les systèmes frontaux avec des groupes d'utilisateurs dédiés et gérables, utilisez le chiffrement du réseau (par ex. VPN) sur les systèmes indépendants pour tenir compte des besoins de protection très élevés.

Exemples

Analyse des ports des serveurs internes - Si l'architecture du réseau n'est pas segmentée, les attaquants peuvent cartographier les réseaux internes et déterminer si les ports sont ouverts ou fermés sur les serveurs internes à partir des résultats de connexion ou du temps écoulé pour connecter ou rejeter les connexions de charges utiles SSRF.

Accès au stockage des métadonnées des services cloud - La plupart des fournisseurs de services cloud disposent d'un stockage de métadonnées tel que `http://169.254.169.254/`. Un attaquant peut lire les métadonnées pour obtenir des informations sensibles.

3. Conclusion

Les sites web sont par nature des éléments très exposés du système d'information. Leur sécurisation revêt une grande importance, et ce à plusieurs titres. La protection contre ces menaces passe à la fois par des mesures préventives et par des mécanismes permettant de détecter les tentatives d'attaques.

En plus de ce top 10, applicable à tout développement web peu importe le langage utilisé, il existe des vulnérabilités particulièrement applicables au langage JavaScript. En voici quelques unes :

- **AJAX**. En front-end ou en back-end, plusieurs vulnérabilités peuvent être exploitées, notamment par l'utilisation de l'interpréteur JavaScript. **AJAX Security**
- **Les scripts tiers**. Inclure des scripts tiers comme les outils analytics ou de marketing, qui communique des informations sur un serveur, peut représenter un risque si ce serveur est compromis, avec l'injection de code malveillant. **Third Party Javascript Management**
- **Cross-site scripting (XSS)**. Un type de faille de sécurité complexe à sécuriser car les possibilités sont très larges. Elle permet d'injecter du contenu dans une page. Entrez ce simple code dans un champ de formulaire : `<script>alert('bonjour')</script>`. Si une boîte de dialogue apparaît, alors le site est sensible aux attaques de type XSS. **XSS Filter Evasion**
- **Cross-Site Request Forgery (CSRF ou XSRF)**. L'attaquant transmet à un utilisateur authentifié une requête HTTP falsifié qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés. **Cross-Site Request Forgery**

En plus de la référence de l'OWASP, vous pouvez consulter les ressources en français de l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) sur **les bonnes pratiques** et plus particulièrement **les recommandations pour la sécurisation d'un site web**.