Sécurité

Plan et objectifs

- La sécurité informatique
- Technique de reconnaissance
- Types d’attaques spécifiques au web et prévention
  - XSS - cross-site scripting
  - CSRF - cross-site request forgery
  - XXE - XML External Entity
  - Injection SQL
  - Déni de service

Sécurité informatique

La sécurité informatique s'assure que les composants matériels et logiciels d'une organisation sont **utilisés dans le cadre prévu** en implémentant des mesures de protection.

Elle s'intéresse aux éléments suivants:

- Confidentialité
  - Restreindre l'accès à certaines ressources ou certaines informations aux individus qui doivent y avoir accès.
  - Qui peut accéder à quoi?
- Intégrité
  - Vérifier l'exactitude et la fiabilité des données.
  - Est-ce que la source d'information est bien celle qu'elle prétend être? Est-ce que j'ai confiance que les données n'ont pas été altérées?
- Disponibilité
  - Assurer le fonctionnement correct du système.
  - Est-ce que les utilisateurs ont accès au système?

Quelques types d’attaques

Voici quelques types d’attaques perpétrées contres des systèmes informatiques:

- Accès non autorisée
- Interruption de service
- Interception d’information
- Altération de données
- Usurpation d’identité (agir en tant que quelqu’un d’autre)

Les vecteurs d’attaques possibles:

- Physique
- Réseau
- Logiciel

Ingénierie sociale

L'ingénierie sociale est une pratique de manipulation psychologique à des fins d'escroquerie.

Exemple:

Un individu a incorporé une société qui se faisait passer pour Quanta Computer, basée à Taiwan et qui fait affaire avec Facebook et Google.

Dans l'escroquerie, les conspirateurs ont créé de faux e-mails en utilisant de faux comptes de messagerie, qui semblaient avoir été envoyés par des employés de Quanta à Taïwan. Ils ont envoyé des e-mails de *phishing* avec de fausses factures aux employés de Facebook et de Google. Ces employés ont répondu en versant plus de 100 millions de dollars sur les comptes bancaires de la fausse entreprise.

https://www.cnbc.com/2019/03/27/phishing-email-scam-stole-100-million-from-facebook-and-google.html

Intentions des attaquants

Quels sont les motivations des attaquants:

- Curiosité
- Défi
- Gain financier
- Reconnaissance
- Motivations d’ordre politiques
- Espionnage industriel
- Chantage
- Revenge
- Perte financière chez la cible
- …

Types d’attaques contre les applications web

Nous allons étudier certains patrons d’attaques courant contre les applications web:

- XSS - cross-site scripting - injection de script
- CSRF - cross-site request forgery -
- XXE - XML External Entity
- Injection SQL
- Déni de service - DoS

Pour chacune, nous détaillerons le fonctionnement et les mesures de prévention possibles.

Prospection (se mettre dans la peau de l’attaquant)

Quelle information peut-on trouver sur les technologies et protocoles utilisés par une application web.

Utilisation des outils de développement pour faire de la rétro-ingénierie et déduire des failles potentielles.

- Sources de la page
- Message dans la console
- Information dans le localStorage, sessionStorage
- Appels APIs par le front-end
- En-têtes HTTP

XSS - cross-site scripting - injection de script

L'attaque XSS consiste à **injecter un code malicieux dans la page web**, ce qui permet à l'attaquant d'utiliser le contexte de la page pour effectuer des actions.

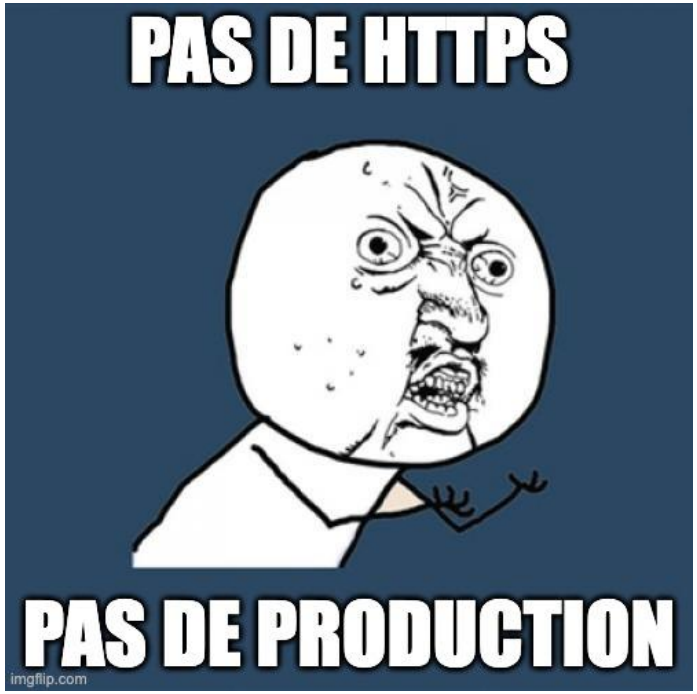
L'attaquant n'a pas besoin de modifier le code de l'application web. Il profite des champs d'entrées (inputs) et y insère des valeurs spécifiques qui injectent le script lorsqu'une page particulière est affichée. L'attaquant profite alors du fait que le script s'exécute dans le navigateur de l'utilisateur en utilisant le contexte de celui-ci (cookies, permissions, etc).

Une application est vulnérable aux attaques de type XSS lorsqu'elle permet aux utilisateurs d'entrer de l'information qui sera ensuite ajoutée à la page sans validation. Par exemple, si une application permet d'entrer des balises HTML et qu'elle honore ensuite ses balises dans l'affichage, elle est probablement vulnérable aux attaques XSS.

Prémisse - HTTPS

L’utilisation du protocole HTTPS est une prémisse pour le reste de la discussion.

Sans l’utilisation de ce protocole sécurisé, plusieurs mesures proposées sont inutiles, car les valeurs des formulaires seront transmises en clair et le navigateur n’aura aucun moyen de valider que le destinataire de la requête est bien celui qu’il prétend être.



XSS Stocké

Lorsqu’un application web stocke des données qui sont ensuite afficher sur une page sans valider que les éléments à insérer sont sécuritaires, il est possible que l’attaquant profite de cette vulnérabilité pour construire une valeur qu’il sauvegardera dans l’application (e.g. un commentaire dans un forum). Cette valeur sera stockée dans la base de données. Lorsqu'une page affichera cette valeur, le script malicieux sera exécuté.

XSS Réflexif

Quand l’application web affiche le contenu d’une entrée (input) à l’utilisateur, on peut utiliser ce comportement pour modifier le comportement de la page en y injectant un script malicieux.

Comme ce type d’attaque n’affecte que l’utilisateur du site. Donc l’attaquant forgera souvent le URL ou la valeur qu’il voudra que la cible entre dans l’application, ce qui exécutera l’attaque sans que l’utilisateur concerné ne s’en rende compte.

Ce type d’attaque demande souvent d’utiliser des techniques d’ingénierie sociale.

Protection contre XSS

De manière générale, on veut restreindre les origines des scripts qui peuvent s'exécuter dans l'application.

Une façon simple de le faire est d'utiliser la **Politique de Sécurité de Contenu** (Content Security Policy**, CSP**).

En utilisant soit une en-tête (`Content-Security-Policy`) ou une balise `meta` `<meta http-equiv="Content-Security-Policy" …>`

Il est possible de spécifier au navigateur quelles sont les origines des scripts (et feuilles de styles CSS) permises. Le navigateur s'assurera que les éléments provenant d'une autre origine ne seront pas utilisés.

Attention mieux vaut bien tester, car la **CSP** peut être très restrictive et empêcher votre application de fonctionner.

Exemple:

```
Content-Security-Policy: default-src 'self'
                         *.source.example.net;
```

Par défaut, bloque les scripts ne provenant pas du domaine du site consulté, mais permet les scripts des sous-domaines `*.source.exemple.net`.

Protection contre XSS

Le CSP ne protège pas nécessairement contre les attaques provenant d'entrées malveillantes insérées dans le DOM (si vous devez permettre `unsafe-inline` par exemple).

Pour prévenir ce type d'attaque, il faut empêcher au code malicieux de s'insérer dans le DOM. Ceci ne s'applique pas seulement aux balises `<script>`, mais à toutes les balises permettant l'exécution de code comme `<img onload="...">`.

Documentation Angular: *Pour bloquer systématiquement les vulnérabilités XSS, Angular traite toutes les valeurs comme non fiables par défaut. Lorsqu'une valeur est insérée dans le DOM à partir d'une liaison de modèle ou d'une interpolation, Angular assainit et échappe les valeurs non fiables.*

Pour celà il ne faut pas utiliser les fonctions bas niveau du DOM comme `innerHTML` dans le code Angular!

Cas réels

*Fin 2015 et début 2016, **eBay** présentait une grave vulnérabilité XSS. Le site web utilisait un **paramètre « url »** qui redirigeait les utilisateurs vers différentes pages de la plateforme, mais la valeur du paramètre n'a pas été validée. **Cela permettait aux attaquants d'injecter du code malveillant dans une page**.*

*En 2018, **British Airways** a été attaquée par Magecart, un groupe de pirates de haut niveau célèbre pour ses attaques d'écrémage de cartes de crédit. Le groupe a exploité une **vulnérabilité XSS dans une bibliothèque JavaScript appelée Feedify**, qui était utilisée sur le site Web de British Airway.*

https://brightsec.com/blog/xss-attack/

CSRF - cross-site request forgery

Ce type d'attaque concerne un utilisateur qui active un **lien malicieux** qui exécute une opération avec ses privilèges.

L'attaquant, qui connaît ou a déduit l'API de l'application web, fournit un lien à la victime.

Lorsque la victime clique sur le lien, elle déclenche une requête (`GET`) du navigateur web qui transmet les informations de session de l'utilisateur au serveur. **L'opération s'exécute alors sans son consentement, mais avec ses privilèges, car le navigateur va faire suivre les cookies.**

L'attaquant peut aussi créer un faux formulaire web sur un site frauduleux qu'il contrôle et qui déclenche un `POST` avec les informations entrées par l'utilisateur.

Exemple de CSRF

L'attaquant à découvert qu'un `GET` sur `transfer` avec un paramètre `amount` et `to_user` s'exécute dans le contexte du compte de banque de l'utilisateur et transfère l'argent.

1. L'attaquant construit un URL et envoie un email avec du contenu HTML:
2. `<a href="http://www.mega-bank.com/transfer?`**`to_user=hacker&amount=100000`**`">Check ça!</a>`
3. L'attaquant envoie le lien via email aux victimes.
4. La victime clique sur le lien et exécute l'opération.

Exemple de CSRF

Simple pour les `GET`, mais normalement un `GET` ne modifie pas l'état. Comment faire si l'API utilise un `POST`?

L'attaquant peut utiliser un formulaire web et y ajouter des champs invisibles pour dissimuler l'attaque.

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="to_user" value="hacker">
  <input type="hidden" name="amount" value="10000">
  <input type="submit" value="Submit">
</form>
```

Pas de cookies

Si votre application web n’utilise pas de cookies pour authentifier la session de ses utilisateurs, les attaques de type CSRF ne seront *pas** possibles, car le contexte de la session ne sera pas envoyé avec la requête malicieuse.

Attribut `SameSite`

L'attribut `SameSite` des cookies permet de réduire les problématiques liés aux CSRF.

En effet, en assignant la valeur `Lax` ou `Strict` pour cet attribut aux cookies de session, permet de limiter leur utilisation.

`Strict`: Le cookie n'est envoyé que si la source de la requête est la même que la destination. Limite la redirection vers un site ou l'utilisateur est déjà connecté par exemple.

`Lax` (défaut pour navigateurs récents): Le cookies est envoyé seulement si la requête engendre un navigation complète (`GET`).

Partage des ressources entre origines multiples - CORS

La politique de même origine (Same Origin Policy) empêche les scripts de charger du contenu à partir d'un serveur d'une origine différente via `XMLHttpRequest` ou `fetch`. L'origine étant définie par le protocole, le domaine et le port.

Ceci permet d'éviter qu'un site malveillant accède aux ressources d'une autre origine (e.g. `api.bank.com`) en envoyant les cookies d'une victime (pour `api.bank.com`).

Le navigateur effectue la requête, mais ne considère pas la réponse sauf si le serveur permet les origines multiples (`Access-Control-Allow-Origin`) et que l'origine de la requête est acceptée par le serveur. Dans le cas mentionné plus haut, le navigateur va simplement refuser la réponse à la requête.

Par contre, il est possible que le site `bank.com` (propriétaire de `api.bank.com`) ait besoin de faire des requêtes à `api.bank.com`. Donc les serveurs d'`api.bank.com` accepterons les requêtes ayant comme origine `bank.com`.

Note sur les CORS et les Cookies

Par défaut, l'API `XMLHttpRequest` ou `fetch` ne va inclure les cookies seulement si la requête est destinée à la même origine.

Si on veut envoyer les cookies vers une autre origine, on doit utiliser l'option `credentials` avec la valeur `include` (`same-origin` est la valeur par défaut).

Pour que la réponse à la requête soit considérée par le navigateur, le serveur devra envoyer l'en-tête `Access-Control-Allow-Credentials: true`.

Protection contre les CSRFs - Jetons

Pour protéger contre les CSRFs, il est possible d'utiliser un jeton de synchronisation. Le serveur génère un jeton (aléatoire, par session ou par requête) qui est retourné au client dans la page HTML ou dans la réponse JSON.

Ce jeton doit être envoyé au serveur leur de l'opération `POST` (formulaire) soit comme paramètre, soit dans une en-tête HTTP.

Par exemple pour un site dynamique: le HTML généré contient le jeton:

```
<input type="hidden" name="csrfmiddlewaretoken"
value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt" />
```

L'attaquant ne pourra pas avoir le bon jeton dans son faux formulaire et son `POST` sera rejeté par le serveur.

Protection contre les CSRFs - Jetons

Pour une application monopage, le jeton peut être transmis dans une réponse d'API du serveur et stocké en mémoire.

Lors de l'appel à l'API, le code javascript de l'application ajoute le jeton dans une en-tête HTTP personnalisée.

Le serveur valide ensuite l'en-tête avant d'effectuer l'opération pour s'assurer que la requête est légitime.

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern

Protection contre les CSRFs - Option sans état

Pour éviter de maintenir un état dans l'application pour valider les jetons de synchronisation, il est possible d'utiliser la stratégie du double submit cookie.

Dans ce cas, le serveur crée un jeton aléatoire et l'ajoute comme cookie (e.g. `XSRF-TOKEN`).

Lorsqu'une requête légitime est envoyée, le code de l'application ajoute la valeur du cookie dans un en-tête particulier (e.g. `X-XSRF-TOKEN`).

Le serveur valide donc que la valeur du cookie et de l'en-tête son identique.

À noter qu'il n'est pas possible pour un script de lire les valeurs des cookies provenant d'une autre origine.

Cas Réels

*En 2008, des chercheurs de Princeton ont découvert une **vulnérabilité CSRF sur YouTube, qui permettait aux attaquants d'effectuer presque toutes les actions au nom de n'importe quel utilisateur**, y compris l'ajout de vidéos aux favoris, la modification des listes d'amis/famille, l'envoi de messages aux contacts d'un utilisateur et le signalement inapproprié. contenu. La vulnérabilité a été immédiatement corrigée.*

*En 2008, **ING Direct**, le site Web bancaire d'un groupe bancaire multinational appartenant aux Pays-Bas, avait **une vulnérabilité CSRF qui permettait aux attaquants de transférer de l'argent à partir des comptes des utilisateurs**, même si les utilisateurs étaient authentifiés avec SSL. Le site Web n'avait aucune protection contre les attaques CSRF, et le processus de transfert de fonds était facile à voir et à reproduire pour les attaquants.*

XXE - XML Entité Externe

Exploite une fonctionnalité du XML qui permet de charger des entités externes au documents XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Pour prévenir ce type d'attaque, il faut désactiver l'option qui permet au parser de charger des entités externes.

Injection SQL

L’injection SQL consiste à construire une valeur particulière pour un entrée (input) de formulaire, sachant que cette valeur sera utilisée pour bâtir une requête SQL dans le serveur.

Exemple: Si le code ressemble à

```
sqlQuery = 'SELECT * FROM users WHERE USER = ' + user_id
```

et qu’il est possible pour l’attaquant de construire `user_id`.

Que se passe-t-il si `user_id` correspond à

- '1=1' -> `SELECT * FROM users where USER = true`
- '123abc; DROP TABLE users;' -> `SELECT * FROM users WHERE USER = 123abd; DROP TABLE users;`
- '123abc; UPDATE users SET credits = 10000 WHERE user = 123abd;' ->`$$$`

Injection SQL - Prévention

Pour prévenir les injections SQL, il faut s'assurer de ne pas utiliser de valeur dans les requêtes SQL sans les assainir.

Normalement les ORMs font le travail pour vous.

En java, il est aussi possible d'utiliser des **`PreparedStatements`**:

```
PreparedStatement pstmt =
    con.prepareStatement("UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```

Le `PreparedStatement` assainit les valeurs avant de les insérer dans la requête.

Attaque par déni de service (DoS)

L’objectif de ce type d’attaque est simplement d’empêcher un site de fonctionner correctement en l’inondant de requêtes ou en lui envoyant des requêtes qui demandent beaucoup de ressources.

Si plusieurs attaquants sont impliqués, on parle alors d’attaque distribuée (DDoS).

Prévenir les DoS

- Évaluer la validité des requêtes et refuser celles qui semblent malicieuses ou automatique
- Limiter la charge
  - par utilisateur
  - pour un type de requête
  - pour un API
  - par IP (blacklist)
- Stratégie du trou noir
  - On déploie des serveurs dédiés qui agissent comme l'application mais n'effectue aucunes opérations
  - On redirige les requêtes suspectes vers ces serveurs

OWASP Top 10

L'Open Web Application Security Project® (OWASP) est une fondation à but non lucratif qui travaille à améliorer la sécurité des logiciels.

Bonne source d’information sur les tendances en termes de sécurité. Ils publient le Top 10 des failles de sécurité les plus courantes.

Sources

OWASP Cheatsheets : https://cheatsheetseries.owasp.org/index.html

Web Application Security: Exploitation and Countermeasures for Modern Web Applications – Andrew Hoffman