DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin

CSRF

1. Qu’est-ce-que le CSRF ?

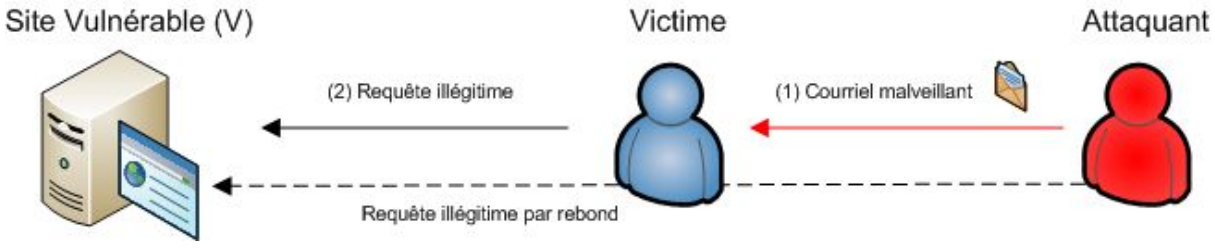
CSRF est souvent confondu avec XSS, XSS permet aux attaquants d'injecter des scripts côté client dans les pages Web consultées par les utilisateurs. Alors que CSRF est un type d'exploitation malveillante d'un site Web où des commandes non autorisées sont transmises par un utilisateur auquel le site fait confiance.

2. Comment réaliser une attaque CSRF

Le but d’une attaque CSRF est de forcer un utilisateur à exécuter une ou plusieurs requêtes non désirées sur un site donné par le biais d’une url interne au site. Par exemple si pour changer un mot de passe l’url est http://sitetest.fr/jechangedemdp?newmdp=toto on va chercher à dissimuler cette url pour qu’elle soit exécutée par l’utilisateur ciblé.
Ceci est possible par le biais d’une image dans un mail, par le biais d’un form auto-validé ou d’une image sur un site malveillant ou vulnérable, faire effectuer l’action que l’on veut à un utilisateur avec d’autres droits.

Exemple

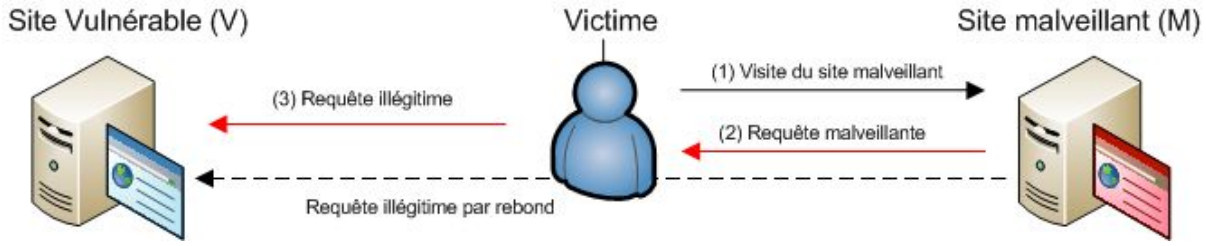
- Par mail, on intègre une image avec dans la balise src l’url du site vulnérable permettant de réaliser l’action désirée (cf ci-dessus) ici un changement de mail, quand la personne ouvrira ce mail l’action sera réalisé et son mot de passe sera changé.



- Sur un site malveillant on peut effectuer la même opération et cacher dans la balise src d’une image l’url pour changer l’email par exemple.

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin

- On peut aussi réaliser une attaque CSRF via un form auto-submit lorsque la cible visite notre site le formulaire malveillant avec l'instruction est validé automatique, ainsi l'action sera effectué par la cible.



3. Protection

3.1. Synchronizer token pattern (STP)

Pour se prémunir des attaques sur les token stockés côté client, le principe est de stocker le token au niveau du serveur plutôt que dans les cookies utilisateur. Le Synchronizer token pattern fonctionne comme suit :

1. L'utilisateur se connecte sur le site web
2. Le serveur web génère un token CSRF qu'il stocke côté serveur
3. Le serveur envoie l'ID de session à l'utilisateur, qui lui le stocke en Cookie
4. L'utilisateur réclame le token CSRF stocké par le serveur
5. Le serveur envoie le token CSRF à l'utilisateur en le masquant dans le formulaire

6. L'utilisateur fait une action qui appelle le formulaire
7. Le serveur vérifie que le token dans le formule est le même que celui qu'il stocke
8. Les deux coïncident : l'action est validée

6-bis. Un attaquant effectue une action qui appelle le même formulaire
7-bis. Le serveur vérifie le formulaire et observe que le token masqué dans le formulaire ne correspond pas avec celui qui appartient à l'utilisateur
8-bis. Les deux ne coïncident pas : l'action est refusée

3.2. Double submit pattern

Le double submit pattern est définie comme l'envoi d'une valeur aléatoire à la fois dans un cookie et comme paramètre de requête, le serveur vérifiant si la valeur du cookie et la valeur de la requête sont égales.

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin
Lorsqu'un utilisateur s'authentifie sur un site, celui-ci doit générer un identifiant de session et placer un cookie dans le navigateur.

En même temps, il génère le token CSRF pour la session et le définit comme un cookie sur la machine de l'utilisateur, séparé de l'identifiant de session.

Le serveur n'a pas besoin de sauvegarder cette valeur de quelque façon que ce soit, c'est pourquoi ce modèle est parfois aussi appelé Stateless CSRF Defense.

Le site exige alors que chaque requête inclut cette valeur aléatoire comme une valeur de formulaire cachée (ou un autre paramètre de la requête).

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin

Cas pratiques d'attaques/défenses CSRF

1. Absence de validation

Le principe est de faire exécuter au navigateur de la victime une action, telle que poster un commentaire.
Si par exemple on peut poster un commentaire via un paramètre GET:

https://site.fr/post_message?message=COUCOU

L'ordinateur de la victime étant déjà connecté au site, c'est sa session qui va être utilisée pour publier le commentaire. L'attaquant peut faire exécuter cette requête à la victime, par exemple si celle-ci visite un site web créé par l'attaquant, ou lit un mail de celui-ci.

Si l'attaquant inclut dans son site l'image suivante:
<img src="https://site.fr/post_message?message=COUCOU">

Le navigateur de la victime va essayer de récupérer l'image, et va donc suivre le lien, ce qui va poster le message à sa place.

On pourrait se dire qu'il ne faut simplement pas que le navigateur cherche l'image, comme ce que les clients mails font par exemple. Il existe cependant différentes techniques autres que via le src d'une image, par exemple via javascript.

Un exemple en javascript pur, n'utilisant pas le src d'une image ni les balises <script> :

<img src="" onerror="a = new XMLHttpRequest(); a.open('GET', 'http://127.0.0.1:8000/csrf/save_comment?body=HACKED'); a.send();">

Toutes les attaques précédentes utilisent le protocole GET, on pourrait donc penser que le problème peut être évité avec la méthode POST:

<form method='POST' action='http://127.0.0.1:8000/csrf/save_comment_post' id="csrf-form"> <input type='hidden' name='body' value='HACKED FROM FORM'> <input type='submit' value='submit'> </form>
<script>document.getElementById("csrf-form").submit()</script>

Ici, on a un formulaire POST, qui va se valider tout seul, et ainsi envoyer la commande sous forme de requête POST.

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin

2. CSRF Statefull

Pour contrer cet envoi de message non voulu par le client, on peut utiliser un token secret, connu uniquement du serveur et du client, qui devra être fourni par le client à chaque requête.
Un attaquant externe ne peut donc pas deviner le token à l'avance, et forger la requête.
Quand le client va se connecter, le serveur va créer une variable, accessible uniquement par lui, et la garder en cache (redis par exemple):

user_JEAN_csrf = gfvhuiugvjhkuiojjuhbkuiohb

Cette variable va alors être transmise au client, par exemple via un champ ajouté dans le formulaire, créé par le serveur:

<form method="GET">
…
<input type="hidden" value="gfvhuiugvjhkuiojjuhbkuiohb" name="csrftoken">
...
</form>

Quand on va valider ce formulaire, on va avoir ce type de requête :

https://site.fr/post_message?message=COUCOU&csrftoken=gfvhuiugvjhkuiojjuhbkuiohb

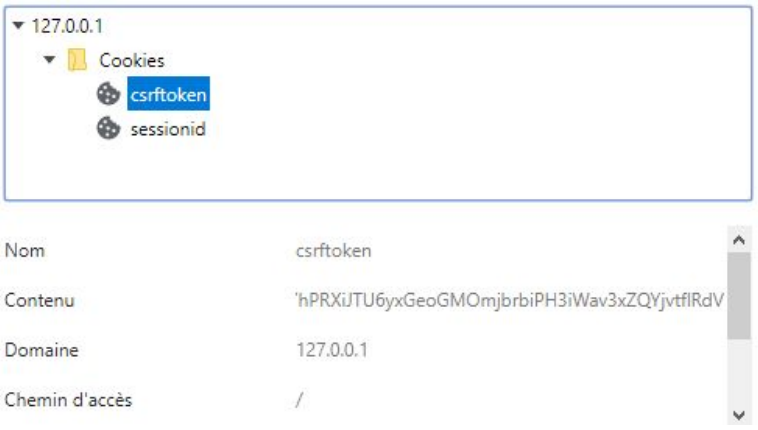
Le serveur peut ensuite comparer si le token fourni par le client est le même que celui qu'il a en interne. L'attaquant ne peut pas fournir le token, cela empêche donc les requêtes forgées.

3. CSRF Stateless

L'exemple précédent fonctionne, mais pose un problème : on est obligé de garder côté serveur un token, pour chaque client, qui change à chaque session ou requête. On va alors implémenter le même système, mais en stateless: le serveur ne stocke pas d'informations.

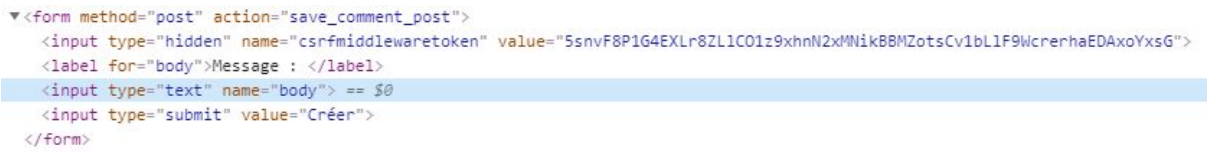
Le token original, auparavant sur le serveur, va maintenant être stocké chez le client, dans les cookies.
Les cookies étant accessibles uniquement par le site qui les a créés, un autre site attaquant ne peut pas les lire et les utiliser :

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin



On utilise ici la particularité des cookies à être envoyés à chaque requête, si celle-ci correspond au bon site.

En demandant au client de fournir le token dans un champs du formulaire, on peut vérifier si celui-ci est le même que celui présent dans les cookies, qu'un attaquant ne peut pas obtenir. On a donc une protection contre les attaques inter-site (Cross Site Request Forgery). Un attaquant ne peut plus juste faire faire la requête à un client visitant le site pirate :



Une autre possibilité est d'inclure le token dans les headers de la page, ce qui revient au même que le placement dans un champ du formulaire. Cette technique est couramment utilisée pour les appels ajax, ce qui simplifie la gestion du token.

On parle alors de "Double Submit Cookie", de par la présence double du token dans les cookies et le formulaire/ou le header.

L'attaque précédente ne fonctionne donc plus.

DAHMOUNI Yassine
VALERY Hugo
BERTAUX Corentin

4. Relation avec les failles XSS

Cette protection fonctionne bien, tant que les liens de l'attaquant se trouvent sur un domaine différent de celui du site officiel. Dans le cas d'une faille XSS (permettant l'exécution de code sur le site), cette technique ne fonctionne plus.
Par exemple, sur un site blog, où l'on poste des messages, où la sécurité a mal été configurée, un attaquant poste le message suivant:

<script>alert('ALED')</script>

Si, lors de l'affichage de son message sur l'ordinateur de la victime, le javascript est bien exécuté, et montre un popup, on parle de faille XSS.

Le script exécuté étant alors sur le site officiel, ce script a accès aux cookies de l'utilisateur.

5. Exploitation d'une faille XSS pour contourner la protection précédente

Un attaquant pouvant faire exécuter du javascript sur le site officiel à une victime peut contourner la protection, en récupérant le cookie, maintenant qu'il a les droits, et en le fournissant dans les détails du formulaire, comme réalisé dans le code suivant :

```
<form method='POST' action='http://127.0.0.1:8000/csrf/save_comment_post'
id="csrf-form">
<input type='hidden' name='body' value='HACKED FROM FORM'>
<input type='submit' value='submit'>
</form>
<script>
function getCookie(cname) {
var name = cname + "="; var decodedCookie = decodeURIComponent(document.cookie);
var ca = decodedCookie.split(';'); for(var i = 0; i <ca.length; i++) { var c = ca[i]; while
(c.charAt(0) == ' ') { c = c.substring(1); } if (c.indexOf(name) == 0) { return
c.substring(name.length, c.length); } } return "";
 }

var csrftoken = getCookie('csrftoken'); var newcsrf = document.createElement("input");
newcsrf.type = "text"; newcsrf.name = "csrfmiddlewaretoken"; newcsrf.value = csrftoken;
document.getElementById("csrf-form").appendChild(newcsrf);
document.getElementById("csrf-form").submit();
 </script>
```

Ici, on poste le même formulaire que précédemment, mais en ajoutant un champ avec le cookie dans le formulaire dynamiquement.

6. Limites

L'attaque précédente a cependant des limites, et exige certains prérequis. Il faut surtout une faille XSS sur le serveur permettant l'exécution de code sur l'ordinateur de la victime.
Il faut aussi que le site sur lequel on réalise l'attaque soit dans le même domaine ou sous-domaine que le serveur officiel. Par exemple, si le site shop.yop.fr est compromis, les cookies du site bank.yop.fr seront accessibles.

Note: L'attaque précédente fonctionne si l'on réalise directement l'attaque, avec un formulaire s'auto validant en HTML, mais la requête est bloquée si elle est exécutée dans une iframe, pour des raisons de sécurité.

7. Mesures défensives

7.1. Restriction d'accès aux cookies

Il est possible de limiter l'accès aux cookies via l'utilisation d'attributs pour ces cookies, via l'instruction SameSite :

- None pour autoriser tous les sites
- Lax pour autoriser tous les sites du même domaine (même protocole (http/s), même domaine (yop.fr), même port).
- Strict pour n'autoriser que les sites correspondants aux critères LAX + même sous domaine.

Ceci peut-être une première mesure pour contrer le CSRF, mais reste inutile si le site d'attaque est le même (faille XSS).

7.2. Analyse du site d'origine

Une autre technique pour filtrer les fausses requêtes serait d'examiner les détails contenus dans les headers "Origin" et "Referrer", décrivant le site d'origine de la requête. Si celui-ci n'est pas le site officiel, la requête ne peut pas être légitime.

Plusieurs problèmes se présentent à cette approche. Certaines failles existent, par exemple en Flash, permettant de changer ces champs. De plus, pour des raisons de vie privée, certains sites utilisent l'option "no-referrer", indiquant au navigateur de ne pas montrer le site d'origine.

7.3. Réactions aux requêtes de changement de header

Une particularité de l'ajax (XmlHttpRequest), est que lors d'un changement de header personnalisé, le navigateur va envoyer une requête OPTIONS au serveur, avec la liste des headers ajoutés. Le serveur peut ainsi prendre une mesure appropriée, comme répondre qu'il ne supporte pas le header, ce qui bloquerait l'envoi de la requête suivante par le navigateur. Il peut aussi simplement décider de ne plus accepter les requêtes suivantes de cet utilisateur pendant un certain temps. Le même phénomène se produit lors du changement des headers via Flash, mais via une requête GET.

7.4. Changement du principe : chiffrement

Depuis le début, le token est comparé, pour voir si la version détenue par le serveur (ou les cookies) est la même que celle envoyée avec les données. Une autre méthode consisterait à envoyer un ensemble chiffré au client lors de la demande du formulaire, token devant ensuite être retourné au serveur.

Soit les différents éléments suivants :

- session_id = id de la session utilisateur, présente dans les cookies et utilisée par le mécanisme de sessions, indépendamment du CSRF.
- timestamp = heure actuelle en secondes
- password = un mot de passe connu seul du serveur, enregistré par exemple dans le fichier de configuration de celui-ci, et aléatoire.
- clef privée = une clef privée connue uniquement par le serveur, pour réaliser un chiffrement symétrique.

Le serveur, à l'envoi du formulaire au client, va envoyer un token composé comme suit :

csrf_token = chiffrement(session_id + timestamp + password)

Le chiffrement se fait en utilisant la clef privée du serveur.
Ce token est ensuite envoyé au client via un champ caché dans le formulaire, et uniquement dans celui-ci (pas dans les cookies). On n'a donc plus un double submit pattern.

Lorsque le client renvoie au serveur le formulaire, le token est déchiffré par le serveur, puis il compare le session_id et le password pour s'assurer que ce sont les mêmes que ceux

connus. Il est possible de rajouter une période d'expiration sur le timestamp, afin de contrer les attaques "en replay".

Cette technique, stateless pour le serveur, présente l'avantage de ne pas utiliser les cookies, mais reste sensible à une faille XSS, si jamais l'attaquant peut faire récupérer le formulaire à l'utilisateur, le remplir avec le token récupéré, et le valider automatiquement.

7.5. Une vraie défense

Une vraie option de défense serait la validation supplémentaire de chaque action par une réauthentification, un captcha, ou un autre moyen, via une redirection,ce qui est par exemple utilisé sur le sites utilisant des transactions bancaires.

8. Notes

Dans tous les cas, une défense CSRF peut être déjouée par un Man In The Middle, si par exemple le site n'utilise pas HTTPS.

Lien Github du projet : https://github.com/c-bertal/csrf