Attaques CSRF

Cross-Site Request Forgery

Un peu d’histoire

- L’attaque commence à être documentée en 2001
  - Peter Watkins introduit le terme *Cross-Site Request Forgery*
- Représente environ 4 fois moins d’attaques que XSS à travers le temps
  - mais leur nombre reste constant (voire augmente sensiblement)
- À partir de 2020, les navigateurs activent par défaut certaines protections contre CSRF
- Mais certaines **mauvaises pratiques de développement** restent courantes
  - ce qui permet à l’attaque de perdurer

HTTP - Rappel important

- HTTP est un protocole **sans état**
- Pour conserver un état, on utilise notamment des **sessions**
- Les sessions sont identifiées par un **identifiant de session**
  - l'identifiant de session est transmis au serveur par le biais d'un **cookie**
  - le cookie est **stocké côté client**
  - et envoyé par le navigateur à chaque requête HTTP au serveur d'origine

Rappels sur les cookies

- Sont en général envoyés par le serveur au client
  - `Set-Cookie: lang=fr; path:/`
- Sont ensuite stockés localement par le client
- Et renvoyés par le client au serveur à chaque requête
  - `Cookie: lang=fr`

Rappels sur les cookies de session

- Session = ensemble d'infos client stockées sur le serveur
  - identifiée par un **identifiant de session** (*Session ID*)
- *Session ID* stocké sur le client (en général : cookie)
- Client se connecte ⇒ serveur crée session et renvoie le *SessionID* au client
  - `Set-Cookie: SessionID=abcde12345`
- Le client stocke le *SessionID* et l'envoie avec chaque requête
  - `Cookie: SessionID=abcde12345`
- Le serveur peut alors identifier ce client et retrouver les infos associées
  - `abcde12345 ⇒ User=pgahide, Role=Admin, LastActivity=2023-05-05 12:01:58`

Anatomie d’une attaque CSRF

- CSRF exploite le fait que **le cookie de session est envoyé à chaque requête**
- *Client* envoie une requête vers *SiteAttaquant*
- *SiteAttaquant* répond avec une redirection vers le *SiteVictime*
- La redirection est en réalité une **requête forgée** pour faire une action sur *SiteVictime* au nom de *Client*
- *Client* transmet donc le requête de l’attaquant à *SiteVictime*, **avec éventuellement son cookie de session**
- *SiteVictime* reçoit et traite la requête, sans soupçonner que celle-ci n’est pas une action intentionnelle du client
- *Client* n’est pas non plus au courant de l’action effectuée sur *SiteVictime*

Acronyme CSRF

- ***Cross-Site*** = à travers le site de l'attaquant
- ***Request Forgery*** = la requête forgée par l'attaquant qui va être *forcée* sur le navigateur victime

CSRF - Exemple

Requête légitime

```
POST /store/addToCart HTTP/2
Host: sitevictime.fr
Cookie: SessionId=abcde12345
...
Content-Length: 12

ProductId=41
```

attaquant.html

```
<body>
  <form method="post"
        action="http://sitevictime.fr/store/addToCart">
    <input type="hidden" name="productId" value="42">
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
```

Résultat : requête forgée

```
POST /store/addToCart HTTP/2
Host: sitevictime.fr
Cookie: SessionId=abcde12345
Referer: http://siteattaquant.com/attaquant.html
...
Content-Length: 12

ProductId=42
```

CSRF - Considérations

- CSRF *cible toujours un ou des site(s) spécifique(s)
  - il faut que la requête forgée soit à destination du site en question
- *Parfois*, CSRF vise également un *client victime spécifique*
  - si l'attaquant a absolument besoin de faire effectuer une action par ce client sur le site cible
- CSRF ne fonctionne que si le client est **déjà connecté sur le site victime**
  - pas connecté ⇒ pas de cookie de session ⇒ pas d'action intéressante possible
- Souvent, ni le client ni le serveur n'ont connaissance de l'attaque

2 exemples avec requêtes GET

Redirection web

```
<head>
  ...
  <meta http-equiv="refresh"
        content="0; URL=http://sitevictime.fr/store/addToCart.php/ProductId=42">
</head>
```

- Une redirection HTML utilise la balise `<meta>` avec les attributs `http-equiv` et `content`
- Cette attaque fonctionne notamment parce qu'une requête GET change l'état du serveur (mauvaise pratique)

Exemple avec un élément `img`

```
<body>
  <img src="http://sitevictime.fr/store/addToCart.php/ProductId=42">
</body>
```

- L’image n’existe pas ici, mais le navigateur va quand même essayer de la charger en suivant l’URL
- De même, on a une requête GET qui change l’état du serveur

Exemple avec POST

```
POST /store/updateCart HTTP/2
Host: sitevictime.fr
Cookie: SessionId=abcde12345
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

ProductId=42&Amount=0
```

Attaque CSRF sur cette requête

```
<body>
  <form method="post"
        action="http://sitevictime.fr/store/updateCart">
    <input type="hidden" name="ProductId" value="42">
    <input type="hidden" name="Amount" value="0">
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
```

Jusqu’ici, aucune protection n’a été mise en place par le site victime.

Contre-mesure

- Pour forger la requête, l’attaquant doit savoir quelle forme elle a
- Donc si la requête est en partie « imprévisible », la requête forgée est impossible

Token CSRF

- L’application inclut, dans le formulaire envoyé au client, un champ caché contenant une valeur aléatoire (*token*)
  - le client va donc renvoyer ce *token* avec la requête
  - le serveur vérifie que la valeur reçue est correcte et rejette la requête si ce n’est pas le cas
  - l’attaquant n’a aucun moyen de deviner le *token*, qui a été généré aléatoirement et à la volée, *pour cette requête spécifique*

Token CSRF - Exemple de requête

- Cette requête HTTP résulte d'un formulaire envoyé par le serveur et contenant un token CSRF

```
POST /store/addToCart HTTP/2
Host: sitevictime.fr
Cookie: SessionId=abcde12345
...
Content-Length: 29

ProductId=42&CSRFToken=xyz789
```

- La requête forgée POST classique (cf plus haut) ne fonctionne plus avec la protection du *token* CSRF
  - elle ne contient pas le *token* ⇒ elle serait refusée
  - il en résulterait typiquement une erreur de type 400 (*Bad Request*)

Token CSRF - HTML résultant

- Voici un exemple de code HTML résultant de l'utilisation de *token* CSRF :

```
<body>
  <form method="post"
        action="store/addToCart.php">
    <input type="hidden" name="productId" value="42">
    <input type="hidden" name="CSRFToken" value="xyz789">
  </form>
</body>
```

- L'attaque classique n'envoie pas le *token*, et ne fonctionne donc pas

Token CSRF - Mauvaises implémentations

- Si le *token* envoyé est **toujours le même**, cela signifie que le site cible utilise une sorte de constante (au moins pour un intervalle de temps donné)
  - dans ce cas, une fois le *token* connu, l'attaque peut être rejouée tant que le *token* est valide
- Si le *token* est **prévisible** (ex : incrémenté de 1 à chaque requête), l'attaque est toujours possible
  - il suffit de « deviner » le prochain *token*
- Si le *token* est **trop court** (quelques octets), il peut être deviné par force brute

Token CSRF et attaque côté client

- Si l'attaquant peut lancer une attaque *depuis le client*, il peut récupérer automatiquement le *token* CSRF ainsi que le cookie de session
- Ex : l'attaquant peut utiliser une attaque XSS sur la page depuis laquelle il veut effectuer une action
- Il peut alors utiliser JS pour forger la requête qui contiendra alors automatiquement le *token* CSRF et le cookie de session

```
<script>window.onload = () => document.forms[1].submit();</script>
```

Contre-mesure - Vérification du *referrer*

- Le *referrer* est l'URL de la page qui a initié la requête
- Le serveur peut vérifier que le *referrer* est bien le site cible
  - si ce n'est pas le cas, la requête est rejetée

Vérification du *referrer* - Exemple de requête

- Cette requête HTTP résulte d'une attaque CSRF

```
POST /store/addToCart HTTP/2
Host: sitevictime.fr
Cookie: SessionId=abcde12345
Referer: http://siteattaquant.com/
...
Content-Length: 12

ProductId=42
```

- Si l'application serveur vérifie le *referrer*, la requête sera rejetée : ce n'est pas le site cible

Vérification du *referrer* - Peu efficace

- Problèmes :
  - le *referrer* n’est pas toujours envoyé par le navigateur
  - le *referrer* peut être modifié par un proxy
  - on peut manipuler la requête en JS ou HTML pour ne pas envoyer le *referrer* (slide suivant)

```
<head>
  <meta name="referrer" content="no-referrer">
</head>
<body>
  <form method="post"
        action="http://sitevictime.fr/store/addToCart">
    <input type="hidden" name="productId" value="42">
  </form>
  <script>
    document.forms[0].submit();
  </script>
```

Contre-mesure - Flag `SameSite`

- Si on vérifie l'origine réelle de la requête, l'attaque CSRF devient beaucoup plus compliquée
- Le *flag* `SameSite` d'un cookie permet de spécifier si le cookie doit être envoyé ou non par le client avec une requête venant d'un autre site
- Ainsi le cookie de session ne sera envoyé que si la requête vient réellement du client

Niveaux du flag `SameSite`

- Le *flag* `SameSite` peut avoir 3 valeurs, pour 3 niveaux de protection :
  - `SameSite=None` : le cookie sera envoyé dans tous les cas
  - `SameSite=Lax` : si la requête vient d'un autre site, le cookie ne sera envoyé que s'il s'agit d'une requête GET
  - `SameSite=Strict` : le cookie ne sera envoyé que si la requête vient du même site
- `SameSite=Lax` est en passe de devenir la valeur par défaut sur les navigateurs populaires

Distinction "Site" / "Origin"

https://app.monsite.fr:8087

- *Site* : **protocole** + **domaine**
- *Origin* : on ajoute **sous-domaines** + **port**
- la vérification `SameSite` se fait sur le *site*, et pas sur l'*origin*

Conséquence

- Si le cookie est défini sur `app.monsite.fr`, il sera **également envoyé** depuis `autresousdomaine.monsite.fr`
- `app.monsite.fr` est alors de nouveau vulnérable **si un sous-domaine du même site héberge une application vulnérable**

Exemple

- Application principale hébergée sur `app.monsite.fr`
- Ancienne version de l'application (*legacy*) hébergée sur `legacy.monsite.fr`
  - ancienne version vulnérable à XSS
- Attaquant utilise XSS sur `legacy.monsite.fr` pour forger une requête vers `app.monsite.fr`
  - **comme c'est le même *site*** du point de vue de la politique `SameSite`, le cookie de session est envoyé
- Même le niveau de protection `SameSite=Strict` est inefficace ici

```
<script>
  fetch('http://sitevictime.fr/store/updateCart.php',
        { method: 'POST',
          headers: {'Content-Type': 'application/x-www-form-urlencoded'},
          credentials: 'include',
          body: 'ProductId=42&Amount=0'
        });
</script>
```

Contre-mesure - Réauthentification

- L’application peut rendre nécessaire une *réauthentification* pour certaines actions sensibles
- Exemples :
  - pour modifier le mot de passe, l’application demande le mot de passe actuel
  - une double authentification (2FA) pourra même être nécessaire
- Problème : cela nuit à l’UX…

CSRF - Conclusion

- CSRF vise à ce que le **navigateur victime envoie une requête forgée à un site victime spécifique**
  - la requête peut être forgée depuis un site attaquant distant (*Cross-Site*)
  - ou depuis le client victime (par XSS)
- L'attaque requiert un **certain niveau de prédiction** de la requête HTTP à effectuer
- La combinaison ***token* CSRF + *flag* `SameSite`** (`Lax` `ou` `Strict`) permet de se prémunir de la majorité des attaques
- D'autres vulnérabilités (**notamment XSS**) permettent de contourner ces protections pour lancer le même type d'attaque ***token* CSRF devient inutile (automatiquement envoyé avec une requête provenant de la même page)
  - *flag* `SameSite` inefficace sur une app hébergée sur un sous-domaine vulnérable du même site