Web et Sécurité

La sécurité sur le Web

Authentification

Informations privées

Attaques classiques

Ingénierie sociale

HTTP Authentification: étape 1

Le serveur rejette les accès non-authentifiés:

```
HTTP/1.0 401 Unauthorized
...
WWW-Authenticate: {authentication-description}
...
```

En réponse, le navigateur demande à l'utilisateur de s'authentifier...

...puis envoie la requête une deuxième fois

HTTP Authentification: étape 2

Une fois authentifié:

Le client (r)envoie les *credentials* dans chaque requête

Elles prennent la forme suivante:

```
GET {blabla} HTTP/1.0
...
Authorization: {authentication-credentials}
...
```

Navigateur envoie ces *credentials* à toutes les pages du même serveur

HTTP Basic Access Authentification

Authentication simple avec envoi du mot de passe en clair

Le serveur rejette les accès non-authentifiés avec:

```
HTTP/1.0 401 Unauthorized status
...
WWW-Authenticate: Basic realm="bugit.org"
```

Le client envoie les *credentials*:

```
GET {blabla} HTTP/1.0
...
Authorization: Basic RW1hY3M6Um9ja3Mh
```

La “clé” est simplement un encodage en *base64* de `{user}:{password}`.

HTTP Digest Authentification: étape 1

Basic Access envoie le mot de passe en clair!

Digest essaie d'éviter ce problème avec un peu de cryptographie

Le serveur rejette les accès non-authentifiés avec:

```
HTTP/1.0 401 Unauthorized status
...
WWW-Authenticate: Digest realm="bugit.org",
                         nonce="dcd98b7102dd"
```

- `nonce`: unique et sert à éviter des *replay attack*
- `domain`: où utiliser ces *credentials*
- `stale`: *nonce* invalide mais *credentials* valides

HTTP Digest Authentification: étape 2

Le client envoie les *credentials* comme suit:

```
GET {uri} HTTP/1.0
...
Authorization: Digest realm="bugit.org",
                      nonce="dcd98b7102dd",
                      username="Emacs",
                      uri="{uri}",
                      response="{md5-digest}"
```

La clé est dans le `{md5-digest}`

HTTP Digest Authentification: md5-digest

Le {md5-digest} est défini comme suit:

```
{HA1}         = MD5 ({username}:{realm}:{password})
{HA2}         = MD5 ({method}:{uri})
{md5-digest} = MD5 ({HA1}:{nonce}:{HA2})
```

E.g.

```
% echo -n 'Emacs:bugit.org:Rocks!' | md5sum
8f165ccd2f2acf215c1677b88bb25368  -
% echo -n 'GET:/index.html' | md5sum
5f751b15eae8c79635edae8bf3b92354  -
% echo -n '8f1..368:dcd98b7102dd:5f7..354' | md5sum
2b67f30deca35c8c58ec0d565a3a68f9  -
%
```

HTTP Digest Authentification: avantages

Mot de passe pas envoyé en clair

Le serveur n'a pas besoin de connaître le mot de passe

Le *nonce* protège des *replay attacks*:

- Chaque client reçoit un *nonce* différent
- Le serveur refuse un *nonce* qui ne vient pas du bon client

Exemple de *nonce*: `MD5({client-IP}:{day}:{secret})`

- Serveur recalcule *nonce* et compare
- Désaccord mauvais client ou vieux *nonce*

HTTP Digest Authentification: attaques

Pas besoin du mot de passe: `HA1` suffit

Replay attack: renvoyer un `Authorization:` déjà utilisé

- Seulement si *nonce* est valide (i.e. même IP et récent)
- Accède seulement au même *URI*

Attaque *Man In The Middle* (MITM):

- L'intermédiaire capture la demande de *digest authentication*
- La remplace par *basic authentication*
- Reçoit le mot de passe en clair!

Ni les réponses ni les requêtes ne sont encryptées!

HTTP Authentification: logout

Le protocole n'offre pas d'option pour se dé-authentifier!

Par contre le serveur peut causer une ré-authentification:

- Renvoie un `401 Unauthorized`
- Suffit pour permettre de changer d'utilisateur

Compliqué à mettre en œuvre

Chaque navigateur devrait offrir cette possibilité

Authentification HTML

N'utilise pas l'authentification de HTTP

Formulaire HTML envoie l'utilisateur et mot de passe

Le serveur répond en plaçant un *cookie* contient les *credentials*

- *credentials* sont renvoyés par le client avec les autres cookies

Chaque site web invente sa propre technique

- Certains *credentials* sont plus sécures que d'autres

Technique la plus populaire

Fuite d'information

Impératifs commerciaux opposés à la vie privée

Peu de motivation et d'efforts pour régler les trous béants

[Comparer aux efforts de style DRM et tivoization, par exemple]

Chaque visite de page web avise une foule d'acteurs

Situation franchement déprimante

Source de fuites directes

Recherche DNS, en clair

Envois/réception de paquets dont la destination est publique

Le nom de la page, bien sûr

Votre adresse IP dévoile votre lieu géographique

Le *header* dévoile votre langue, navigateur, SE et versions

Le *header* vous identifie de manière unique (ou presque)

Le `Referrer:` indique comment vous avez trouvé cette page

Les cookies dévoilent encore plus

Ces infos sont disponibles: au serveur et aux passants (moins HTTPS)

Source de fuites indirectes

Chaque page nécessite des dizaines/centaines de transferts

Souvent une dizaine de serveur webs différents

Les fuites directes sont donc multipliées

Beaucoup de ces requêtes sont *dédiées* à collecter ces infos

C'est à la base même de la gratuité des services!

Firefox a un cookie `.google.com` avant même la première page!

Probablement pas trop grave. Après tout, ces serveurs ont aussi:

- vos fichiers, vos courriels, votre liste de contacts, l'histoire de votre vie, tous vos rendez-vous passés et futurs, l'historique de vos achats, votre position GPS, ...

Phishing

Obtenir des informations secrètes en se passant pour quelqu'un d'autre

Typiquement: reproduire fidèlement un site web

Peut être assez sophistiqué (tout le monde est stupide!)

Peut profiter de fautes de frappe: e.g. `www.dejardins.com`

Ou utiliser un *URL redirector*:

- `www.desjardins.com/redirect?dst=http://elsewhere/`

Ou de la complexité de Unicode: e.g. le "e" cyrillique

Reproduction du site web peut se faire par MITM

Réutilisation de courriel légitime, en changeant un URL ou attachement

Cross-Site Request Forgery (CSRF)

Profiter de la confiance d'un serveur en l'origine d'une requête

Imposer une requête dangereuse envoyée depuis la victime

- Suffit d'avoir le contrôle sur le `src` d'une balise `img`
- `http://192.168.1.1/admin?op=enableRemote`
- `http://localhost:631/printers/HP?op=print-test-pag`

Bien sûr, des `GET` ne devraient pas être dangereux

- Beaucoup de serveurs répondent aux `GET` comme aux `POST`
- Javascript permet de faire des `POST`

Le serveur peut vérifier le `Referrer:`

Isolation entre pages

Visiter `cutecats.com` en même temps que `mybank.com`

`XMLHttpRequest` permet des requêtes HTTP depuis Javascript

La page de `cutecats.com` peut donc contacter `mybank.com`

- Venant de votre browser
- avec vos *cookies* et vos *credentials*?
- Comme si c'était vous!?!?

Same-origin policy

Isoler différents documents utilisés dans le même navigateur

Visiter `cutecats.com` en même temps que `mybank.com`

Limiter la puissance de `XMLHttpRequest`

- Idée de base: interdire `XMLHttpRequest` à un autre site
- Requêtes toujours possibles par éléments `img`, `script`, `style`

Différences:

- `XMLHttpRequest` permet d'autres méthodes que `GET`
- Seul `XMLHttpRequest` a accès direct au résultat

Cross-Origin Resource Sharing (CORS)

`Referrer:` n'est pas fiable

Interdire `XMLHttpRequest` à un autre site est trop restrictif

Permettre au serveur HTTP de contrôler les requêtes acceptées

- Autoriser à priori tous les `XMLHttpRequest`
- Par défaut pas de *credentials* sauf pour *same-origin*
- Ajouter aux requêtes un header `Origin:`
- Ajouter aux réponses des `Access-Control-Allow-*:`
- `XMLHttpRequest` peut bloquer les réponses
- Requêtes "dangereuses" précédées par une requête `OPTIONS`

Cross-Site Scripting (XSS)

Permet à l'attaquant de servir du code depuis un serveur légitime

Contourne la sécurité basée sur le *same-origin policy*

E.g. classique:

- Attaquant poste un message dans un forum de discussion
- Le message inclut inclut `<script>...</script>`
- Le serveur omet d'encoder ces balises
- Le script est exécuté par chaque visiteur du forum
- Le script a les même droits que le code légitime du site web
- Il lit les cookies d'authentification et les envoie ailleurs

Session hijacking

Connections HTTP envoient les cookies en clair

Authentification HTML met *credentials* dans les cookies $\Rightarrow$ vulnérables!

E.g. un réseau wifi ouvert:

- Les autres machines du réseau voient tous les paquets
- L'attaquant peut donc capturer vos *credentials*
- Lorsqu'ils les renvoie, ils viennent même de la même IP!

Authentication HTML normalement sur HTTPS

- Usage partiel fréquent de HTTP, pour réduire la charge
- Cookies devraient être marqués `secure`!