Fiche pratique sur la prévention des attaques de type « cross site scripting »

Introduction

Cette fiche aide les développeurs à prévenir les vulnérabilités XSS.

Le terme « Cross-Site Scripting » (XSS) est impropre. À l'origine, ce terme provenait des premières versions de l'attaque qui visaient principalement à voler des données entre sites. Depuis, le terme s'est élargi pour inclure l'injection de pratiquement n'importe quel contenu. Les attaques XSS sont graves et peuvent conduire à l'usurpation d'identité, à l'observation du comportement des utilisateurs, au chargement de contenu externe, au vol de données sensibles, etc.

Cette fiche pratique contient des techniques permettant de prévenir ou de limiter l'impact du XSS. Étant donné qu'aucune technique ne permet à elle seule de résoudre le problème du XSS, il est nécessaire d'utiliser la bonne combinaison de techniques défensives pour prévenir le XSS.

Sécurité des frameworks

Heureusement, les applications développées à l'aide de frameworks web modernes comportent moins de bogues XSS, car ces frameworks orientent les développeurs vers de bonnes pratiques de sécurité et contribuent à atténuer les risques XSS grâce à l'utilisation de modèles, à l'échappement automatique, etc. Cependant, les développeurs doivent savoir que des problèmes peuvent survenir si les frameworks sont utilisés de manière non sécurisée, par exemple :

- *les échappements* utilisés par les frameworks pour manipuler directement le DOM
- React `dangerouslySetInnerHTML` sans assainir le HTML
- React ne peut pas gérer `javascript:` ou `data:` URL sans validation spécialisée
- Les fonctions `bypassSecurityTrustAs*` fonctions
- Lit's `unsafeHTML` fonction
- Polymer's `inner-h-t-m-l` attribut et `htmlLiteral` fonction
- Injection de modèle
- Plugins ou composants du framework obsolètes
- et plus encore

Lorsque vous utilisez un framework web moderne, vous devez savoir comment celui-ci empêche les attaques XSS et où se trouvent ses failles. Il arrivera parfois que vous deviez effectuer des opérations qui dépassent le cadre de la protection offerte par votre framework, ce qui signifie que le codage des sorties et la désinfection HTML

peuvent s'avérer essentiels. L'OWASP va produire des fiches pratiques spécifiques aux frameworks React, Vue et Angular.

Philosophie de défense contre les attaques XSS

Pour qu'une attaque XSS réussisse, un pirate doit être capable d'insérer et d'exécuter du contenu malveillant dans une page web. Ainsi, toutes les variables d'une application web doivent être protégées. S'assurer que **toutes les variables** sont validées puis échappées ou nettoyées est ce qu'on appelle **une résistance parfaite aux injections**. Toute variable qui ne passe pas par ce processus est une faiblesse potentielle. Les frameworks permettent de s'assurer facilement que les variables sont correctement validées et échappées ou nettoyées.

Cependant, aucun framework n'est parfait et des failles de sécurité existent encore dans les frameworks populaires tels que React et Angular. Le codage de sortie et le nettoyage HTML permettent de combler ces failles.

Codage de sortie

Lorsque vous devez afficher en toute sécurité les données exactement telles que l'utilisateur les a saisies, le codage de sortie est recommandé. Les variables ne doivent pas être interprétées comme du code plutôt que comme du texte. Cette section couvre chaque forme de codage de sortie, où l'utiliser et quand vous ne devez pas utiliser de variables dynamiques.

Tout d'abord, lorsque vous souhaitez afficher les données telles que l'utilisateur les a saisies, commencez par utiliser la protection par encodage de sortie par défaut de votre framework. La plupart des frameworks intègrent des fonctions d'encodage et d'échappement automatiques.

Si vous n'utilisez pas de framework ou si vous devez combler des lacunes dans le framework, vous devez utiliser une bibliothèque d'encodage de sortie. Chaque variable utilisée dans l'interface utilisateur doit passer par une fonction d'encodage de sortie. Une liste des bibliothèques d'encodage de sortie est incluse en annexe.

Il existe de nombreuses méthodes d'encodage de sortie différentes, car les navigateurs analysent différemment le HTML, le JS, les URL et le CSS. L'utilisation d'une méthode d'encodage inappropriée peut introduire des faiblesses ou nuire au fonctionnement de votre application.

Codage de sortie pour les « contextes HTML »

Le « contexte HTML » fait référence à l'insertion d'une variable entre deux balises HTML de base, telles que `<div>` ou `<b>`. Par exemple :

```
<div> $varUnsafe </div>
```

Un pirate pourrait modifier les données affichées sous la for fe . Cela pourrait entraîner l'ajout d'une attaque à une page Web. Par exemple :

```
<div> <script>alert`1`</script> </div> // Exemple d'attaque
```

Afin d'ajouter une variable à un contexte HTML en toute sécurité à un modèle Web, utilisez le codage d'entité HTML pour cette variable.

Voici quelques exemples de valeurs encodées pour des caractères spécifiques :

Si vous utilisez JavaScript pour écrire en HTML, consultez **Sink** `.textContent` . Il s'agit d'un et le codage d'entité HTML sera effectué automatiquement.

```
&     &amp;
<     &lt;
>     &gt;
"     &quot;
'     &#x27;
```

Codage de sortie pour l'attribut « Contexts HTML Attribute »

Les « contextes d'attributs HTML » se produisent lorsqu'une variable est placée dans une valeur d'attribut HTML. Vous pouvez le faire pour modifier un lien hypertexte, masquer un élément, ajouter un texte alternatif à une image ou modifier des styles CSS en ligne. Vous devez appliquer le codage d'attribut HTML aux variables placées dans la plupart des attributs HTML. Une liste des attributs HTML sécurisés est fournie dans la section « **Sinks sécurisés** ».

```
<div attr="$varUnsafe">
<div attr=”*x” onblur=”alert(1)*”> // Exemple d'attaque
```

Il est essentiel d'utiliser des guillemets `_` **ou** `.` pour **encadrer vos variables.** Les guillemets rendent **comme**
difficile de modifier le contexte dans lequel une variable opère, ce qui contribue à prévenir les attaques XSS. L'utilisation de guillemets réduit également considérablement le jeu de caractères que vous devez encoder, ce qui rend votre application plus fiable et l'encodage plus facile à mettre en œuvre.

Si vous écrivez dans un attribut HTML avec JavaScript, consultez la section `.setAttribute` et `[attribut]` car elles encodent automatiquement les attributs HTML. Il s'agit de **« Safe Sinks »** tant que le nom de l'attribut est codé en dur et inoffensif, comme `id` ou `classe`
En général, les attributs qui acceptent JavaScript, tels que `onClick` , **ne** sont **PAS sûrs** à utiliser avec des des valeurs d'attribut non fiables.

Codage de sortie pour les « contextes JavaScript »

Les « contextes JavaScript » font référence à la situation où des variables sont placées dans du JavaScript en ligne, puis intégrées dans un document HTML. Cette situation se produit généralement dans les programmes qui utilisent beaucoup de JavaScript personnalisé intégré dans leurs pages web.

Cependant, le seul emplacement « sûr » pour placer des variables en JavaScript est à l'intérieur d'une « valeur de données entre guillemets ». Tous les autres contextes sont dangereux et vous ne devez pas y placer de données variables.

Exemples de « valeurs de données entre guillemets »

```
<script>alert('$varUnsafe')</script>
<script>x='$varUnsafe'</script>
<div onmouseover="'$varUnsafe'"</div>
```

Encodez tous les caractères à l'aide du `\xHH` format. Les bibliothèques d'encodage ont souvent un `EncodeForJavaScript` ou similaire pour prendre en charge cette fonction.

Veuillez consulter les exemples d'encodage JavaScript de l'OWASP Java Encoder pour obtenir des exemples d'utilisation correcte de JavaScript nécessitant un encodage minimal.

Pour JSON, vérifiez que le type de contenu est correct. `Content-Type` est `application/json` et non `text/html` à

Encodage de sortie pour les « contextes CSS »

Les « contextes CSS » font référence aux variables placées dans le CSS en ligne, ce qui est courant lorsque les développeurs souhaitent que leurs utilisateurs personnalisent l'apparence de leurs pages web. Le CSS étant étonnamment puissant, il a été utilisé pour de nombreux types d'attaques. **Les variables ne doivent être placées que dans une valeur de propriété CSS. Les autres « contextes CSS » ne sont pas sûrs et vous ne devez pas y placer de données variables.**

```
<style> sélecteur { propriété : $varUnsafe; } </style>
<style> sélecteur { propriété : "$varUnsafe" ; } </style>
<span style="property : $varUnsafe">Oh non</span>
```

Si vous utilisez JavaScript pour modifier une propriété CSS, pensez à utiliser `st[illegible]` s'agit d'un **Safe Sink** qui encodera automatiquement les données CSS qu'il contient.

Lorsque vous insérez des variables dans des propriétés CSS, assurez-vous que les données sont correctement encodées et nettoyées afin d'éviter les attaques par injection. Évitez de placer des variables directement dans des sélecteurs ou d'autres contextes CSS.

Encodage de sortie pour les « contextes URL »

Les « contextes URL » font référence aux variables placées dans une URL. Le plus souvent, un développeur ajoute un paramètre ou un fragment d'URL à une base URL qui est ensuite affichée ou utilisée dans une opération. Utilisez le codage URL pour ces scénarios.

```
<a href="http://www.owasp.org?test=$varUnsafe">lien</a >
```

Encodez tous les caractères de la même manière que JS et CSS. `%HH` format d'encodage. Assurez-vous que tous les attributs sont entièrement entre guillemets.

Erreur courante

Il peut arriver que vous utilisiez une URL dans différents contextes. Le plus courant est l'ajouterait à un `href` ou `src` attribut d'une balise `<a>` . Dans ces cas, vous devriez Effectuer l'encodage URL, suivi de l'encodage des attributs HTML.

```
url = "https://site.com?data=" + urlencode(parameter)
<a href='attributeEncode(url)'>lien</a>
```

Si vous utilisez JavaScript pour construire une valeur de requête URL, pensez à utiliser `window.encodeURIComponent(x)` . Il s'agit d'un **Safe Sink** qui encodera automatiquement les données en URL.

Contextes dangereux

Le codage de sortie n'est pas parfait. Il n'empêchera pas toujours les XSS. Ces emplacements sont connus sous le nom de **contextes dangereux**. Les contextes dangereux comprennent :

```
<script>Directement dans un script</script>
<!-- À l'intérieur d'un commentaire HTML -->
<style>Directement dans CSS</style>
<div ToDefineAnAttribute=test />
<ToDefineATag href="/test" />
```

Autres points à prendre en compte :

- Fonctions de rappel
- Lorsque les URL sont gérées dans du code tel que ce CSS { background-url : « javascript:alert(xss) » ; }
- Tous les gestionnaires d'événements JavaScript (`onclick()` , `onerror()` , `onmouseover()`).
- Fonctions JS non sécurisées telles que `eval()` , `setInterva` `setTimeout()`

Ne placez pas de variables dans des contextes dangereux, car même avec un encodage de sortie, cela n'empêchera pas complètement une attaque XSS.

Assainissement HTML

Lorsque les utilisateurs doivent créer du code HTML, les développeurs peuvent leur permettre de modifier le style ou la structure du contenu dans un éditeur WYSIWYG. Dans ce cas, le codage de sortie empêchera les attaques XSS, mais il perturbera le fonctionnement prévu de l'application. Le style ne sera pas rendu. Dans ces cas, il convient d'utiliser la purification HTML.

La désinfection HTML supprimera le code HTML dangereux d'une variable et renverra une chaîne HTML sûre. L'OWASP recommande DOMPurify pour la désinfection HTML.

```
let clean = DOMPurify.sanitize(dirty);
```

Il y a d'autres éléments à prendre en compte :

- Si vous purifiez le contenu puis le modifiez par la suite, vous risquez de réduire à néant vos efforts en matière de sécurité.
- Si vous nettoyez le contenu puis l'envoyez à une bibliothèque pour utilisation, vérifiez qu'il ne modifie pas cette chaîne de caractères d'une manière ou d'une autre. Sinon, là encore, vos efforts en matière de sécurité seront vains.
- Vous devez régulièrement mettre à jour DOMPurify ou les autres bibliothèques de nettoyage HTML que vous utilisez. Les navigateurs modifient régulièrement leurs fonctionnalités et des contournements sont régulièrement découverts.

Sinks sécurisés

Les professionnels de la sécurité parlent souvent en termes de sources et de puits. Si vous polluez une rivière, elle s'écoulera quelque part en aval. Il en va de même pour la sécurité informatique. Les puits XSS sont des endroits où des variables sont placées dans votre page web.

Heureusement, de nombreux puits où les variables peuvent être placées sont sûrs. En effet, ces puits traitent la variable comme du texte et ne l'exécutent jamais. Essayez de refactoriser votre code pour supprimer les références à des puits non sécurisés tels que innerHTML, et utilisez plutôt textContent ou value.

```
elem.textContent = dangerVariable ;
elem.insertAdjacentText(dangerVariable) ; elem.className
= dangerVariable ; elem.setAttribute(safeName,
dangerVariable) ; formfield.value = dangerVariable ;
document.createTextNode(dangerVariable) ;
document.createElement(dangerVariable) ; elem.innerHTML =
DOMPurify.sanitize(dangerVar) ;
```

Les attributs HTML sécurisés comprennent : `align` , `alink` , `alt` , `bgcolor` , `border` , `cellpaddin cellspacing` , `class` , `color` , `cols` , `colspan` , `coords` , `dir` , `face` , `height` , `hspace` , `ismap` , `lang` , `marginheight` , `marginwidth` , `multiple` , `nohref` , `noresize` , `noshade` , `nowrap` , `ref` , `rel` , `rev` , `rows` , `rowspan` , `scrolling` , `shape` , `span` , `summary` , `tabindex` , `title` , `usemap` , `valign` , `value` , `vlink` , `vspace` , `width` .

Pour les attributs non mentionnés ci-dessus, assurez-vous que si du code JavaScript est fourni comme valeur, il ne peut pas être exécuté.

Autres contrôles

Les protections de sécurité du framework, le codage de sortie et la désinfection HTML offrent la meilleure protection pour votre application. L'OWASP les recommande dans toutes les circonstances.

Envisagez d'adopter les contrôles suivants en plus de ceux mentionnés ci-dessus.

- Attributs des cookies - Ils modifient la manière dont JavaScript et les navigateurs peuvent interagir avec les cookies. Les attributs des cookies tentent de limiter l'impact d'une attaque XSS, mais n'empêchent pas l'exécution de contenus malveillants et ne s'attaquent pas à la cause profonde de la vulnérabilité.
- Politique de sécurité du contenu - Une liste blanche qui empêche le chargement de contenu. Il est facile de commettre des erreurs lors de la mise en œuvre, c'est pourquoi elle ne doit pas constituer votre principal mécanisme de défense. Utilisez une CSP comme couche de défense supplémentaire et consultez l'aide-mémoire ici.
- Pare-feu d'applications Web - Ils recherchent les chaînes d'attaque connues et les bloquent. Les WAF ne sont pas fiables et de nouvelles techniques de contournement sont régulièrement découvertes. Les WAF ne s'attaquent pas non plus à la cause profonde d'une vulnérabilité XSS. De plus, les WAF ne détectent pas une catégorie de vulnérabilités XSS qui opèrent exclusivement côté client. Les WAF ne sont pas recommandés pour prévenir les XSS, en particulier les XSS basés sur le DOM.

Résumé des règles de prévention XSS

Ces extraits de code HTML montrent comment afficher des données non fiables en toute sécurité dans différents contextes.

Type de données : chaîne de caractères Contexte : corps HTML Code : défense : encodage d'entité HTML (règle n° 1) `<span>DONNÉES NON FIABLES </span>` Exemple

Type de données : Chaîne de caractères Contexte : Attributs HTM `<input type="text" name="fname" value="DONNÉES NON FIABLES` Exemple de défense : encodage agressif des entités HTML (règle n° 2), n'è placer que les données non fiables dans une liste d'attributs sûrs (énumérés ci-dessous), valider strictement les attributs non sûrs tels que background, ID et name.

Type de données : Chaîne de caractères Contexte : Parar `<a href="/site/search?value=NON FIABLE DATA ">clickme</a>` Exemple de défense : encodage d'URL (règle n° 5).

Type de données : Chaîne Contexte : URL non fiable dans un attribut SRC ou HREF Cod `<a href="URL NON FIABLE ">clickme</a> <iframe src="URL NON FIABLE " />` Exemple de défense : Canonicalisation des entrées, validation des URL, vérification des URL sécurisées, autorisation des URL http et HTTPS uniquement (éviter le protocole JavaScript pour ouvrir une nouvelle fenêtre), encodeur d'attributs.

Type de données : chaîne de caractères Contexte : vi `HTML <div style="width: DONNÉES NON FIABLES ;">Sélection</div>` Exemple de défense : validation structurelle stricte (règle n° 4), encodage hexadécimal CSS, bonne conception des fonctionnalités CSS.

Type de données : Chaîne de caractères Contexte : Variable Ja `<script>var currentValue='DONNÉES NON DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>` Exemple de défense :

Assurez-vous que les variables JavaScript sont entre guillemets, encodage hexadécimal JavaScript, encodage Unicode JavaScript, évitez l'encodage ksla `\'` ou `\\` `\"` ou

Type de données : HTML Contexte : Corps HTML Coc `<div>HTML NON FIABLE</div>` Exemple de défense : Validation HTML (JSoup, AntiSamy, HTML Sanitizer...).

Type de données : Chaîne de caractères Contexte : `<script>document.write("ENTRÉE NON FIABLE : " + document.location.hash );<script/>` Exemple de défense : Cheat sheet pour la prévention des XSS basés sur le DOM

Feuille |

Résumé des règles de codage de sortie

Le but du codage de sortie (en ce qui concerne le Cross Site Scripting) est de convertir les entrées non fiables en une forme sûre où elles sont affichées sous forme **de données** à l'utilisateur sans être exécutées comme **du code** dans le navigateur. Les tableaux suivants fournissent une liste des méthodes de codage de sortie essentielles pour empêcher le Cross Site Scripting.

Type de codage : Entité HTML Mécanisme de codage : Convertir `&` à `&` , Convertir `<` en `<` , Convertir `>` en `>` , Convertir `"` en « , Convertir `'` en `&#x27;`

Type d'encodage : Encodage des attributs HTML Mécanisme d'encodage : Encode tous les caractères avec l'format d'entité HTML `&#xHH ;` , y compris les espaces, où **HH** représente la valeur hexadécimale de le caractère en Unicode. Par exemple, `A` devient `&#x41;` . Tous les caractères alphanumériques (lettres A à Z, a à z et chiffres 0 à 9) restent non codés.

Type d'encodage : encodage URL Mécanisme d'encodage : utilisez l'encodage en pourcentage standard, tel que spécifié dans la spécification W3C, pour encoder les valeurs des paramètres. Soyez prudent et n'encodez que les valeurs des paramètres, et non l'URL entière ou les fragments de chemin d'une URL.

Type d'encodage : encodage JavaScript Mécanisme d'encodage : encodez tous les caractères à l'aide de l'encodage Unicode.
format d'encodage Unicode , où **XXXX** représente le code hexadécimal Unicode . Par exemple, `A` devient `\u0041` . Tous les caractères alphanumériques (lettres A à Z, a à z et chiffres de 0 à 9) ne sont pas codés.

Type de codage : codage hexadécimal CSS Mécanisme de codage : le codage CSS prend en charge à `\XX` et `\XXXXXX` formats. Pour garantir un encodage correct, envisagez les options suivantes : (a) Ajoutez un espace
après l'encodage CSS (qui sera ignoré par l'analyseur CSS), ou (b) utilisez les six caractères complets en ajoutant des zéros à la valeur. Par exemple, `A` devient `\41` (format court) ou `\000041` (format complet). Caractères alphanumériques (lettres A à Z, a à z et chiffres 0 à 9) restent non codés.

Anti-modèles courants : approches inefficaces à éviter

Il est difficile de se défendre contre les attaques XSS. C'est pourquoi certains ont cherché des raccourcis pour les prévenir.

Nous allons examiner deux anti-modèles courants qui apparaissent fréquemment dans d'anciens articles, mais qui sont encore souvent cités comme solutions dans les articles modernes sur la défense contre les attaques XSS sur les forums de programmeurs

tels que Stack Overflow et d'autres lieux de rencontre pour développeurs.

Dépendance exclusive aux en-têtes Content-Security-Policy (CSP)

Tout d'abord, soyons clairs, nous sommes de fervents partisans de la CSP lorsqu'elle est utilisée correctement. Dans le contexte de la défense contre les XSS, la CSP fonctionne mieux lorsqu'elle est :

- Utilisé comme technique de défense en profondeur.
- est personnalisé pour chaque application individuelle plutôt que déployé comme une solution d'entreprise unique pour tous.

Ce à quoi nous nous opposons, c'est une politique CSP globale pour l'ensemble de l'entreprise. Les problèmes liés à cette approche sont les suivants :

Problème n° 1 - Hypothèse selon laquelle toutes les versions de navigateur prennent en charge la CSP de la même manière

On part généralement du principe implicite que tous les navigateurs des clients prennent en charge toutes les constructions CSP utilisées par votre politique CSP globale. De plus, cette hypothèse est souvent formulée sans tester explicitement l' `User-Agent` en-tête de requête pour vérifier s'il s'agit bien d'un type de navigateur et refuser l'utilisation du site s'il ne correspond pas. Pourquoi ? Parce que la plupart des entreprises ne veulent pas refuser des clients qui utilisent un navigateur obsolète ne prenant pas en charge certaines constructions CSP de niveau 2 ou 3 sur lesquelles elles s'appuient pour la prévention des XSS. (Statistiquement, presque tous les navigateurs prennent en charge les directives CSP de niveau 1. À moins que vous ne craigniez que votre grand-père sorte son vieil ordinateur portable Windows 98 et utilise une version archaïque d'Internet Explorer pour accéder à votre site, vous pouvez probablement partir du principe que le niveau 1 est pris en charge.

Problème n° 2 - Problèmes liés à la prise en charge des applications héritées

Les en-têtes de réponse CSP obligatoires à l'échelle de l'entreprise vont inévitablement perturber certaines applications Web, en particulier les applications héritées. Cela conduit l'entreprise à rejeter les directives AppSec et aboutit inévitablement à la délivrance par AppSec de dérogations et/ou d'exceptions de sécurité jusqu'à ce que le code de l'application puisse être corrigé. Mais ces exceptions de sécurité créent des failles dans votre armure XSS, et même si ces failles sont temporaires, elles peuvent tout de même avoir un impact sur votre entreprise, au moins en termes de réputation.

Dépendance vis-à-vis des intercepteurs HTTP

L'autre anti-modèle courant que nous avons observé est la tentative de traiter la validation et/ou le codage de sortie dans une sorte d'intercepteur tel que Spring Interceptor, qui implémente généralement implémente `org.springframework.web.servlet.HandlerInterceptor` ou en tant que servlet JavaEE filtre qui implémente `javax.servlet.Filter` . Bien que cela puisse fonctionner pour des applications très spécifiques (par exemple, si vous validez que toutes les requêtes d'entrée qui sont rendues ne contiennent que des données alphanumériques), cela va à l'encontre du principe fondamental de la défense XSS, qui consiste à effectuer le codage de sortie aussi près que possible de l'endroit où les données sont rendues. En général, la requête HTTP est

examinée pour les paramètres de requête et POST, mais d'autres éléments, tels que les en-têtes de requête HTTP qui peuvent être rendus, comme les données de cookies, ne sont pas examinés. L'approche courante que nous avons observée est la suivante
quelqu'un appellera soit `ESAPI.validator().getValidSafeHTML()` ou `ESAPI.encoder.canonicalize()` et, en fonction des résultats, redirigera vers une page d'erreur ou appeler quelque chose comme `ESAPI.encoder().encodeForHTML()` . Outre le fait que cette approche passe souvent à côté des entrées corrompues telles que les en-têtes de requête ou les « informations de chemin supplémentaires » dans une URI, elle ignore complètement le fait que le codage de sortie est totalement hors contexte. Par exemple, comment un filtre de servlet peut-il savoir qu'un paramètre de requête d'entrée va être rendu dans un contexte HTML (c'est-à-dire entre des balises HTML) plutôt que dans un contexte JavaScript
tel que dans une balise `<script>` ou utilisé avec un attribut de gestionnaire d'événements JavaScript ? Il . Et comme les encodages JavaScript et HTML ne sont pas interchangeables, vous restez exposé aux attaques XSS.

À moins que votre filtre ou votre intercepteur ait une connaissance approfondie de votre application et, plus précisément, de la manière dont celle-ci utilise chaque paramètre pour une requête donnée, il ne peut pas fonctionner correctement dans tous les cas limites possibles. Et nous affirmons qu'il ne sera jamais en mesure de le faire en utilisant cette approche, car fournir le contexte supplémentaire requis est une conception beaucoup trop complexe et introduire accidentellement une autre vulnérabilité (dont l'impact pourrait être bien pire que celui du XSS) est presque inévitable si vous essayez de le faire.

Cette approche naïve présente généralement au moins l'un des quatre problèmes

suivants. **Problème n° 1 : l'encodage pour un contexte spécifique n'est pas satisfaisant**

pour tous les chemins d'URI

L'un des problèmes est le codage incorrect qui peut encore permettre l'exploitation de XSS dans certains chemins d'URI de votre application. Un exemple pourrait être un paramètre de formulaire « lastname » provenant d'un POST qui s'affiche normalement entre des balises HTML, de sorte que le codage HTML est suffisant, mais il peut y avoir un ou deux cas limites où lastname est en fait rendu comme faisant partie d'un bloc JavaScript où le codage HTML n'est pas suffisant et où il est donc vulnérable aux attaques XSS.

Problème 2 - L'approche par intercepteur peut entraîner un rendu incorrect en raison d'un encodage incorrect ou double

Un deuxième problème lié à cette approche est que l'application peut entraîner un encodage incorrect ou double. Par exemple, supposons que dans l'exemple précédent, un développeur ait effectué un encodage de sortie correct pour le rendu JavaScript du nom de famille. Mais s'il a déjà été encodé en sortie HTML, lors du rendu, un nom de famille légitime tel que « O'Hara » pourrait s'afficher sous la forme « O\'Hara ».

Bien que ce deuxième cas ne soit pas strictement un problème de sécurité, s'il se produit assez souvent, il peut entraîner une réticence de l'entreprise à utiliser le filtre et celle-ci peut donc décider de désactiver le filtre ou de spécifier des exceptions pour certaines pages ou certains paramètres filtrés, ce qui affaiblira la défense XSS qu'il fournissait.

Problème n° 3 - Les intercepteurs ne sont pas efficaces contre les attaques XSS basées sur le DOM

Le troisième problème est que cette méthode n'est pas efficace contre les attaques XSS basées sur le DOM. Pour cela, il faudrait disposer d'un intercepteur ou d'un filtre capable d'analyser tout le contenu JavaScript faisant partie d'une réponse HTTP, d'identifier les sorties corrompues et de déterminer si elles sont vulnérables aux attaques XSS basées sur le DOM. Cela n'est tout simplement pas pratique.

Problème n° 4 : les intercepteurs ne sont pas efficaces lorsque les données des réponses proviennent de l'extérieur de votre application.

Le dernier problème lié aux intercepteurs est qu'ils ignorent généralement les données contenues dans les réponses de votre application qui proviennent d'autres sources internes, telles qu'un service Web interne basé sur REST ou même une base de données interne. Le problème est que, à moins que votre application ne valide strictement ces données *au moment où elles sont récupérées* (ce qui est généralement le seul moment où votre application dispose d'un contexte suffisant pour effectuer une validation stricte des données à l'aide d'une approche par liste blanche), ces données doivent toujours être considérées comme corrompues. Mais si vous essayez d'effectuer un encodage de sortie ou une validation stricte des données pour toutes les données corrompues du côté de la réponse HTTP d'un intercepteur (tel qu'un filtre Java servlet), à ce stade, l'intercepteur de votre application n'aura aucune idée de la présence de données corrompues provenant des services web REST ou d'autres bases de données que vous avez utilisés. L'approche généralement utilisée sur les intercepteurs côté réponse qui tentent d'assurer une défense XSS consiste à ne considérer que les « paramètres d'entrée » correspondants comme corrompus et à effectuer un encodage de sortie ou un nettoyage HTML sur ceux-ci, tout le reste étant considéré comme sûr. Mais parfois, ce n'est pas le cas. Bien que l'on suppose souvent que tous les services Web internes et toutes les bases de données internes peuvent être « fiables » et utilisés tels quels, il s'agit d'une très mauvaise hypothèse, à moins que vous ne l'ayez incluse dans une modélisation approfondie des menaces pour votre application.

Par exemple, supposons que vous travaillez sur une application destinée à présenter à un client le détail de sa facture mensuelle. Supposons également que votre application interroge une base de données interne externe (qui ne fait pas partie de votre application spécifique) ou un service web REST que votre application utilise pour obtenir le nom complet, l'adresse, etc. de l'utilisateur. Mais ces données proviennent d'une autre application que vous considérez comme « fiable », mais qui présente en réalité une vulnérabilité XSS persistante non signalée dans les différents champs liés à l'adresse du client. Supposons en outre que le service clientèle de votre entreprise puisse examiner la facture détaillée d'un client afin de l'aider lorsqu'il a des questions à ce sujet. Un client malveillant décide alors de placer une bombe XSS dans le champ d'adresse, puis appelle le service client pour obtenir de l'aide concernant sa facture. Si un tel scénario se produisait, un intercepteur tentant d'empêcher le XSS passerait complètement à côté et le résultat serait bien pire que l'affichage d'une boîte d'alerte indiquant « 1 » ou « XSS » ou « pwn'd ».

Résumé

Une dernière remarque : si le déploiement d'intercepteurs/filtres comme défense contre les attaques XSS était une approche utile, ne pensez-vous pas qu'elle serait intégrée à tous les pare-feu d'applications Web (WAF) commerciaux et recommandée par l'OWASP dans cette fiche pratique ?

Articles connexes

Fiche pratique sur les attaques XSS :

L'article suivant décrit comment les pirates peuvent exploiter différents types de vulnérabilités XSS (et cet article a été créé pour vous aider à les éviter) :

- OWASP : fiche pratique sur le contournement des filtres XSS.

Description des vulnérabilités XSS :

- Article OWASP sur les vulnérabilités XSS.

Discussion sur les types de vulnérabilités XSS :

- Types de cross-site scripting.

Comment examiner le code à la recherche de vulnérabilités de type Cross-Site Scripting :

- Article du guide de révision du code OWASP sur la révision du code pour détecter les vulnérabilités de type Cross-Site Scripting.

Comment tester les vulnérabilités liées au cross-site scripting :

- Article du guide de test OWASP sur le test des vulnérabilités de type Cross-Site Scripting.
- Règles minimales expérimentales d'encodage XSS Fournit des exemples et des directives pour des stratégies minimales expérimentales d'encodage visant à prévenir les attaques de type Cross-Site Scripting (XSS).