7

Cross-Site Scripting (XSS)

Eines der berühmtesten Beispiele für eine Cross-Site-Scripting-Schwachstelle ist der von Samy Kamkar entwickelte Myspace-Samy-Wurm. Im Oktober 2005 nutzte Kamkar eine Schwachstelle auf Myspace aus, die es ihm erlaubte, JavaScript-Nutzdaten in sein Profil einzuschleusen. Besuchte ein eingeloggter Nutzer sein Myspace-Profil, wurde dieser Code ausgeführt, machte den Besucher zu Kamkars Freund auf Myspace und erweiterte das Profil des Besuchers um den Text »but most of all, samy is my hero«. Der Code kopierte sich dann in das Profil des Nutzers und infizierte andere Myspace-Nutzerseiten.

Zwar hatte Kamkar den Wurm nicht in bösartiger Absicht entwickelt, dennoch durchsuchten die Behörden daraufhin seine Wohnung. Kamkar wurde für die Verbreitung des Wurms verhaftet und einer schweren Straftat schuldig gesprochen.

Kamkars Wurm ist ein extremes Beispiel, doch dieser Exploit zeigt die umfassenden Auswirkungen, die eine XSS-Schwachstelle auf eine Website haben kann. Ähnlich wie bei anderen Schwachstellen, die ich bisher behandelt habe, kommt

es zu XSS-Schwachstellen, wenn Websites bestimmte Zeichen unkontrolliert ausgeben und der Browser bösartigen JavaScript-Code ausführt. Zu den Zeichen, die eine XSS-Schwachstelle ermöglichen, gehören doppelte Anführungszeichen (`"`), einfache Anführungszeichen (`'`) und spitze Klammern (`< >`).

Sichert eine Site Zeichen korrekt ab, werden sie aus HTML-Entitäten ausgegeben. Der Quellcode einer Webseite würde diese Zeichen dann wie folgt angeben:

- ein doppeltes Anführungszeichen (") als " oder &#34,
- ein einfaches Anführungszeichen (') als ' oder &#39,
- eine öffnende spitze Klammer (<) als < oder &#60 und
- Eine schließende spitze Klammer (>) als > oder &#62.

Diese speziellen Zeichen definieren, wenn man sie nicht absichert, die Struktur einer Webseite in HTML und JavaScript. Schützt sich eine Site beispielsweise nicht vor spitzen Klammern, können Sie über `<script></script>` ein Skript einschleusen:

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

Senden Sie das an eine Website, die es unkontrolliert ausgibt, weisen die `<script></script>`-Tags den Browser an, den zwischen den Tags liegenden JavaScript-Code auszuführen. Die Nutzdaten führen die `alert`-Funktion aus und öffnen einen Pop-up-Dialog, der die an `alert` übergebenen Informationen ausgibt. Die Referenz von `document` innerhalb der Klammern verweist auf das DOM, das den Domainnamen der Site zurückliefert. Wird das Skript beispielsweise auf *https://www.<example>.com/foo/bar/* ausgeführt, gibt der Pop-up-Dialog *www.<example>.com* aus.

Haben Sie eine XSS-Schwachstelle entdeckt, müssen Sie deren Auswirkungen untersuchen, da nicht alle XSS-Schwachstellen gleich sind. Die Auswirkungen eines Bugs zu bestätigen und diese Analyse in Ihren Report aufzunehmen, verbessert Ihren Report, hilft den Entscheidern bei der Validierung des Bugs und kann Ihr Bounty erhöhen.

Eine XSS-Schwachstelle, die das `httponly`-Flag für sensible Cookies nicht setzt, ist anders als eine XSS-Schwachstelle, die das macht. Nutzt eine Site das `httponly`-Flag nicht, kann Ihr XSS Cookie-Werte lesen. Enthalten diese Werte Sessions identifizierende Cookies, können Sie die Session des Opfers und seinen Account übernehmen. Sie können `document.cookie` in einem Dialog ausgeben, um zu bestätigen, dass Sie sensible Cookies lesen können. (Welche Cookies eine Site als sensibel erachtet, müssen Sie für jede Site ausprobieren.) Selbst wenn Sie nicht auf sensible Cookies zugreifen können, können Sie sich `document.domain` in einem

Dialog ausgeben lassen, um zu sehen, ob Sie auf sensible Nutzdaten aus dem DOM zugreifen und Aktionen im Namen des Opfers ausführen können.

Doch Ihr XSS muss für eine Site keine Sicherheitslücke sein, wenn Sie nicht die richtige Domain ansprechen. So kann Ihr JavaScript-Code für ein `document.domain` aus einem isolierten iFrame harmlos sein, weil Sie nicht auf Cookies zugreifen, über den Account des Nutzers keine Aktionen durchführen und nicht auf sensible Nutzerdaten aus dem DOM zugreifen können.

Der JavaScript-Code verursacht keinen Schaden, weil Browser die sogenannte *Same Origin Policy (SOP)*, übersetzt etwa die »Regel des gleichen Ursprungs«, als Sicherheitsmechanismus implementieren. Die SOP beschränkt, wie Dokumente (das D in DOM) mit Ressourcen anderen Ursprungs umgehen. SOP schützt unschuldige Websites vor bösartigen Sites, die versuchen, die Website durch den Nutzer zu missbrauchen. Wenn Sie beispielsweise *www.<malicious>.com* besuchen und diese versucht, den `GET`-Request *www.<example>.com/ profile* in Ihrem Browser abzusetzen, würde SOP *www.<malicious>.com* daran hindern, die Response von *www.<example>.com/profile* zu lesen. Die Site *www.<example>.com* könnte es Sites mit einem anderen Ursprung erlauben, mit ihr zu interagieren, doch üblicherweise sind solche Interaktionen auf Sites beschränkt, denen *www.<example>.com* vertraut.

Das Protokoll (zum Beispiel HTTP oder HTTPS), der Host (zum Beispiel *www.<example>.com*) und der Port bestimmten den Ursprung (Origin) einer Site, wobei der Internet Explorer die Ausnahme von der Regel ist. Er betrachtet den Port nicht als Teil des Ursprungs. Tabelle 7–1 zeigt Beispiele für den Ursprung und ob sie als Teil von *http://www.<example>.com/* betrachtet werden.

URL	Gleicher Ursprung?	Grund
http://www.<example>.com/countries/Canada http://store.<example>.com/countries	Ja	-- Anderer Host
http://www.<example>.com/countries	Ja	--
https://www.<example>.com/countries	Nein	Anderes Protokoll
http://www.<example>.com:8080/countries	Nein	Anderer Port

Tab. 7–1 *Beispiele für SOP*

In einigen Fällen stimmt der URL nicht mit dem Ursprung überein. Zum Beispiel erben die `about:blank`- und `javascript:`-Schemata den Ursprung von dem sie öffnenden Dokument. Der `about:blank`-Kontext greift auf Informationen des Browsers zu oder interagiert mit ihm, während `javascript:` JavaScript ausführt. Der URL enthält keine Informationen zu dessen Ursprung, weshalb Browser diese beiden Kontexte unterschiedlich behandeln. Wenn Sie eine XSS-Schwachstelle

gefunden haben, ist `alert(document.domain)` für Ihren Machbarkeitsnachweis hilfreich: Es bestätigt den Ursprung, in dem das XSS ausgeführt wird, insbesondere wenn der im Browser ausgegebene URL sich vom Ursprung unterscheidet, gegen den das XSS ausgeführt wird. Genau das passiert, wenn eine Website einen `javascript:`-URL öffnet. Wenn *www.<example>.com* einen `javascript:alert(document.domain)`-URL öffnet, zeigt der Browser die Adresse `javascript:alert(document.domain)`. Doch der Alert-Dialog würde *www.<example>.com* enthalten, weil der Alert den Ursprung des vorherigen Dokuments erbt.

Ich habe ein Beispiel vorgestellt, das das HTML-Tag `<script>` für das XSS nutzt, doch Sie werden nicht immer HTML-Tags senden können, wenn Sie eine Möglichkeit zum Einschleusen entdecken. In diesen Fällen können Sie möglicherweise einfache oder doppelte Anführungszeichen nutzen, um die XSS-Payload einzuschleusen. Das XSS kann auch wesentlich davon abhängen, wo das Einschleusen erfolgt. Nehmen wir beispielsweise an, Sie können auf das `value`-Attribut des folgenden Codes zugreifen:

```
<input type="text" name="username" value="hacker" width=50px>
```

Indem Sie ein doppeltes Anführungszeichen in das `value`-Attribut einschleusen, können Sie das vorhandene Anführungszeichen schließen und eine bösartige XSS-Payload in das Tag einfügen. Dazu ändern Sie das `value`-Attribut in `hacker" onfocus=alert(document.cookie) autofocus "` ab, was zu folgendem Ergebnis führt:

```
<input type="text" name="username" value="hacker"
 onfocus=alert(document.cookie) autofocus "" width=50px>
```

Das `autofocus`-Attribut weist den Browser an, den Eingabefokus auf das Textfeld zu setzen, sobald die Seite geladen wird. Das JavaScript-Attribut `onfocus` weist den Browser an, den JavaScript-Code ausführen, sobald das Textfeld den Fokus hat. (Ohne `autofocus` wird `onfocus` ausgeführt, wenn jemand das Textfeld anklickt.) Doch diese beiden Attribute haben ihre Grenzen. Sie können den Autofokus nicht für verdeckte (hidden) Felder nutzen. Gibt es auf einer Seite mehrere Felder mit Autofokus, erhält je nach Browser das erste oder das letzte Element den Fokus. Wird die Payload ausgeführt, gibt sie `document.cookie` aus.

Nehmen wir nun an, dass Sie innerhalb eines `<script>`-Tags Zugriff auf eine Variable haben. Können Sie im folgenden Code ein einfaches Anführungszeichen in den Wert der `name`-Variablen einschmuggeln, schließen Sie die Variable, und Ihr eigener JavaScript-Code wird ausgeführt:

```
<script>
  var name = 'hacker';
</script>
```

Da Sie den Wert hacker kontrollieren, führt die Änderung der name-Variablen in `hacker';alert(document.cookie);'` zu folgendem Code:

```
<script>
  var name = 'hacker';alert(document.cookie);'';
</script>
```

Das Einfügen eines einfachen Anführungszeichens und eines Semikolons schließt die Variable `name` ab. Weil wir ein `<script>`-Tag nutzen, wird die von uns eingeschleuste JavaScript-Funktion `alert(document.cookie)` ausgeführt. Wir hängen ein zusätzliches `;'` an das Ende unseres Funktionsaufrufs an, um sicherzustellen, dass unser JavaScript-Code syntaktisch korrekt ist, da die Site ein `';` enthält, um die `name`-Variable abzuschließen. Ohne das `';` am Ende würde ein hängendes einfaches Anführungszeichen übrig bleiben, und die Syntax der Seite wäre nicht mehr korrekt.

Wie Sie nun wissen, kann XSS über verschiedene Methoden durchgeführt werden. Die Website *http://html5sec.org/*, die von den Penetrationstest-Experten von Cure53 gepflegt wird, ist eine großartige Quelle für XSS-Payloads.

7.1 Arten von XSS

Es gibt zwei wesentliche Arten von XSS: reflektiert und gespeichert. *Reflektiertes XSS* liegt vor, wenn ein einzelner HTTP-Request, der auf der Site nicht gespeichert wird, die XSS-Payload ausliefert und ausführt. Die Browser, einschließlich Chrome, Internet Explorer und Safari, versuchen, diese Art von Schwachstelle durch die Einführung von *XSS-Auditoren* zu unterbinden. (Im Juli 2018 hat Microsoft angekündigt, den XSS-Auditor im Edge-Browser einzustellen, da andere Sicherheitsmechanismen zur Verfügung stehen, um XSS zu verhindern.) XSS-Auditoren versuchen, Nutzer vor bösartigen Links zu schützen, die JavaScript-Code ausführen. Kommt es zu einem XSS-Versuch, sperrt der Browser die Seite und präsentiert dem Nutzer eine Meldung, dass die Seite zum Schutz des Nutzers blockiert wurde. Abbildung 7–1 zeigt ein Beispiel in Google Chrome.

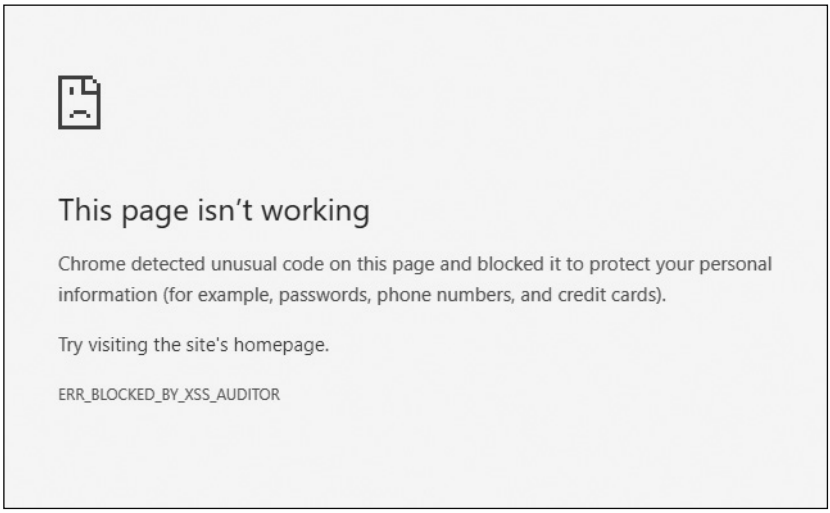

Abb. 7–1 *Eine vom XSS-Auditor in Google Chrome blockierte Seite*

Doch trotz aller Bemühungen der Browser-Entwickler können Angreifer XSS-Auditoren häufig umgehen, da JavaScript auf einer Site auf sehr komplexe Art und Weise ausgeführt werden kann. Da sich die Methoden zur Umgehung von XSS-Auditoren häufig ändern, würden sie den Rahmen dieses Buchs sprengen. Wenn Sie mehr erfahren wollen, sind FileDescriptors Blog-Post auf *https://blog.innerht.ml/the-misunderstood-x-xss-protection/* und Masato Kinugawas »filter bypass cheat sheet« auf *https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet/* zwei sehr gute Quellen.

Im Gegensatz dazu liegt *gespeichertes XSS* vor, wenn eine Site die bösartige Payload speichert und ungeprüft rendert. Sites können die übergebene Payload auch an unterschiedlichen Stellen rendern. Die Payload muss nicht unmittelbar nach der Übertragung ausgeführt werden, sondern erst wenn auf eine andere Seite zugegriffen wird. Wenn Sie beispielsweise auf einer Website unter Ihrem Namen ein Profil mit der XSS-Payload anlegen, muss der XSS-Code nicht ausgeführt werden, wenn Sie sich das Profil ansehen, sondern vielleicht erst, wenn jemand nach Ihrem Namen sucht oder eine Nachricht sendet.

Sie können XSS-Angriffe auch in eine der drei folgenden Unterkategorien einteilen: DOM-basiert, blind und selbstbezogen.

Bei *DOM-basierten XSS*-Angriffen wird der JavaScript-Code einer Website manipuliert, um bösartigen JavaScript-Code auszuführen. Das kann gespeichert oder reflektiert erfolgen. Nehmen wir zum Beispiel an, dass die Webseite *www.<example>.com/hi/* den folgenden HTML-Code nutzt, um den Seiteninhalt durch den Wert aus einem URL zu ersetzen, ohne diesen vorher auf bösartige Eingaben zu prüfen. Dann könnte es möglich sein, bösartigen XSS-Code auszuführen.

```
<html>
  <body>
    <h1>Hi <span id="name"></span></h1>
    <script>document.getElementById('name').innerHTML=location.hash.split('#')
      [1]</script>
  </body>
</html>
```

Bei dieser Beispiel-Webseite ruft das Skript-Tag die `getElementById`-Method des document-Objekts auf, um das HTML-Element mit der ID `'name'` zu ermitteln. Der Aufruf gibt eine Referenz auf das span-Element im `<h1>`-Tag zurück. Das Skript modifiziert dann den Text zwischen den Tags `<span id="name"></span>` mit der Methode `innerHTML`. Es setzt den Text zwischen `<span></span>` auf den Wert von `location.hash`, was jeglicher Text ist, der in einem URL hinter einem `#` steht (`location` ist ein weiteres Browser-API ähnlich dem DOM. Es ermöglicht den Zugriff auf Informationen über den aktuellen URL).

Daher führt *www.<example>.com/hi#Peter/* zu einer dynamischen Aktualisierung des HTML-Codes zu `<h1><span id="name">Peter</span></h1>`. Doch die Seite überprüft den `#`-Wert im URL nicht, bevor sie das `<span>`-Element aktualisiert. Wenn ein Nutzer also *www.<example>.com/h1#<img src=x onerror =alert(document .domain)>* eingibt, poppt ein Alert-Dialog auf und gibt *www .<example>.com* aus (vorausgesetzt es wurde kein Bild mit dem Namen x an den Browser zurückgeliefert). Der resultierende HTML-Code für die Seite sieht also wie folgt aus:

```
<html>
  <body>
    <h1>Hi <span id="name"><img src=x onerror=alert(document.domain)></span>
      </h1>
    <script>document.getElementById('name').innerHTML=location.hash.split('#')
      [1]</script>
  </body>
</html>
```

Diesmal wird nicht »Peter« zwischen den `<h1>`-Tags ausgegeben, sondern ein Alert-Dialog mit dem `document.domain`-Namen. Der Angreifer kann beliebigen JavaScript-Code ausführen, weil er ihn im `<img>`-Tag an `onerror` koppelt.

Blindes XSS ist ein gespeicherter XSS-Angriff, bei dem ein anderer Nutzer die XSS-Payload über einen Speicherort der Website rendert, auf die der Hacker selbst nicht zugreifen kann. Das ist beispielsweise der Fall, wenn es Ihnen gelingt, den XSS beim Anlegen eines persönlichen Profils über den Vor- oder Nachnamen einzuschleusen. Diese Werte können entschärft werden, wenn sich normale Nutzer Ihr Profil ansehen, doch wenn ein Administrator eine administrative Seite

besucht, die alle neuen Nutzer auflistet, werden diese Werte möglicherweise nicht bereinigt, und das XSS könnte ausgeführt werden. Das Tool XSSHunter (*https://xsshunter.com/*) von Matthew Bryant ist zur Erkennung blinden XSS ideal. Die von Bryant entwickelten Payloads führen JavaScript aus, das ein entferntes Skript ausführt. Bei der Ausführung gibt es das DOM, Browser-Informationen, Cookies und andere Informationen zurück an ihren XSSHunter-Account.

Selbstbezogene XSS-Schwachstellen (Self XSS) wirken sich nur auf den Nutzer aus, der die Payload übergibt. Da ein Hacker nur sich selbst angreifen kann, wird selbstgezogenem XSS nur eine geringe Schwere zugestanden, und von den meisten Bug-Bounty-Programmen gibt es kein Bounty. Ein solcher Fall kann beispielsweise eintreten, wenn das XSS per `POST`-Request übertragen wird. Da dieser Request durch CSRF geschützt ist, kann nur das Opfer die XSS-Payload übertragen. Selbstbezogenes XSS kann gespeichert sein, muss es aber nicht.

Wenn Sie ein selbstbezogenes XSS entdeckten, suchen Sie nach Möglichkeiten, es mit anderen Schwachstellen zu kombinieren, die sich auf andere Nutzer auswirken, wie etwa *Login/Logout CSRF*. Bei einem solchen Angriff wird das Opfer aus seinem Account ausgeloggt und in den Account des Angreifers eingeloggt, um den bösartigen JavaScript-Code auszuführen. Üblicherweise muss ein Login/Logout CSRF in der Lage sein, das Opfer über bösartigen JavaScript-Code wieder einzuloggen. Wir gehen auf Bugs, die Login/Logout CSRF ermöglichen, nicht weiter ein, doch ein hervorragendes Beispiel hat Jack Whitton auf einer Uber-Site gefunden. Sie können auf *https://whitton.io/articles/uber-turning-self-xss-into-good-xss/* mehr darüber nachlesen.

Die Auswirkung eines XSS-Angriffs hängt von einer Vielzahl von Faktoren ab: ob er gespeichert oder reflektiert ist, ob auf Cookies zugegriffen werden kann, ob die Payload ausgeführt wird und so weiter. Trotz des möglichen Schadens, den XSS anrichten kann, ist die Behebung von XSS-Schwachstellen oft recht einfach, da die Softwareentwickler (wie bei der HTML-Injection) nur die Benutzereingaben bereinigen müssen, bevor sie sie ausgeben.

7.2 Shopify-Großhandel

Schwierigkeitsgrad: Niedrig
URL: *wholesale.shopify.com/*
Quelle: *https://hackerone.com/reports/106293/*
Meldezeitpunkt: 21. Dezember 2015
Gezahltes Bounty: $ 500

XSS-Payloads müssen nicht kompliziert sein, man muss sie aber an einem Ort einschleusen, an dem sie auch ausgeführt werden. Dabei ist auch zu beachten,

ob sie in HTML- oder JavaScript-Tags enthalten sind. Im Dezember 2015 war Shopifys Großhandels-Website eine einfache Webseite mit einem eigenständigen Suchfeld. Die XSS-Schwachstelle auf dieser Seite war simpel, aber leicht zu übersehen: Der im Suchfeld eingegebene Text wurde ungeschützt in vorhandenen JavaScript-Tags ausgegeben.

Der Bug wurde übersehen, weil die XSS-Payload keinen ungeschützten HTML-Code ausnutzte. Nutzt XSS das HTML-Rendering aus, sehen Angreifer die Auswirkung der Payload, da HTML das Erscheinungsbild einer Site definiert. Im Gegensatz dazu kann JavaScript-Code das Erscheinungsbild einer Site ändern oder eine andere Aktion ausführen, doch er *definiert* nicht das Erscheinungsbild der Site.

In diesem Fall führte `"><script>alert('XSS')</script>` die XSS-Payload `alert('XSS')` nicht aus, weil Shopify die HTML-Tags <> codierte. Diese Zeichen wurden entschärft, indem sie als `<` und `>` ausgegeben wurden. Ein Hacker entdeckte aber, dass die Eingabe auf der Webseite zwischen `<script></script>`-Tags ungeschützt ausgegeben wurde. Sehr wahrscheinlich ist er zu dieser Erkenntnis gelangt, indem er sich den Quellcode der Seite angesehen hat, die den HTML- und JavaScript-Code enthält. Sie können sich den Quellcode jeder beliebigen Webseite ansehen, indem Sie *view-source:URL* in der Adressleiste des Browsers eingeben. Abbildung 7–2 zeigt beispielhaft einen Teil des Seiten-Quellcodes für *https://nostarch.com/*.

Nachdem er erkannt hatte, dass die Eingabe ungeschützt ausgegeben wurde, gab der Hacker `test';alert('XSS');'` in Shopifys Suchfeld ein, was beim Rendern der Seite einen JavaScript alert-Dialog mit dem Text `'XSS'` ausgab. Im Report wird es zwar nicht klar, doch sehr wahrscheinlich hat Shopify den gesuchten Begriff in einer JavaScript-Anweisung wie `var search_term = '<INJECTION>'` gerendert. Der erste Teil der Injection, `test';` schloss dieses Tag und fügte `alert('XSS');` als separate Anweisung ein. Das abschließende `'` stellte sicher, dass die JavaScript-Syntax korrekt war. Das Ergebnis war dann vermutlich `var search_term = 'test';alert('xss'); '';`.

Abb. 7–2 *Seiten-Quellcode für* https://nostarch.com/

7.2.1 Was wir mitnehmen

XSS-Schwachstellen müssen nicht kompliziert sein. Die Shopify-Schwachstelle war nicht schwierig: nur ein einfaches Texteingabefeld, das die Eingabe des Nutzers nicht entschärfte. Wenn Sie XSS-Lücken suchen, sehen Sie sich den Seiten-Quellcode an und prüfen Sie, ob Ihre Payload in HTML- oder JavaScript-Tags gerendert wird.

7.3 Shopifys Währungsformatierung

Schwierigkeitsgrad: Niedrig
URL: <*YOURSITE*>*.myshopify.com/admin/settings/general/*
Quelle: *https://hackerone.com/reports/104359/*
Meldezeitpunkt: 9. Dezember 2015
Gezahltes Bounty: $ 1.000

XSS-Payloads werden nicht immer sofort ausgeführt. Hacker sollten daher prüfen, ob die Payload an allen Orten richtig entschärft wird, an denen sie gerendert werden könnte. In diesem Beispiel erlaubten Shopifys Shop-Einstellungen den Nutzern das Ändern des Währungsformats. Im Dezember 2015 wurden die Werte dieser Eingabefelder nicht korrekt entschärft, wenn Social-Media-Seiten eingerichtet wurden. Ein bösartiger Nutzer konnte einen Shop einrichten und eine XSS-Payload in die Währungseinstellungen einschleusen (siehe Abbildung Figure 7–3). Die Payload wurde in den Social-Media-Vertriebskanälen des Shops gerendert. Der bösartige Nutzer konnte den Shop so konfigurieren, dass die Payload ausgeführt wurde, wenn ein anderer Shop-Administrator den Vertriebskanal besuchte.

Shopify nutzte die Template-Engine Liquid zum dynamischen Rendering des Inhalts von Shop-Seiten. Zum Beispiel ist `${{ }}` die Liquid-Syntax. Die zu rendernde Variable wird innerhalb der inneren geschweiften Klammern angegeben. In Abbildung 7–3 ist `${{amount}}` ein legitimer Wert, an den aber der Wert `"><img src=x onerror=alert(document.domain)>`, die XSS-Payload, angehängt wurde. Das `">` schließt das HTML-Tag ab, in das die Payload eingeschmuggelt wird. Wird das HTML-Tag geschlossen, rendert der Browser das Image-Tag und sucht nach einem Image namens `x`, wie es das `src`-Attribut verlangt. Weil ein Image mit diesem Namen auf der Shopify-Website wahrscheinlich nicht existiert, erkennt der Browser einen Fehler und ruft den JavaScript-Eventhandler `onerror` auf. Der Eventhandler führt den im Handler definierten JavaScript-Code aus, in diesem Fall also die Funktion `alert(document.domain)`.

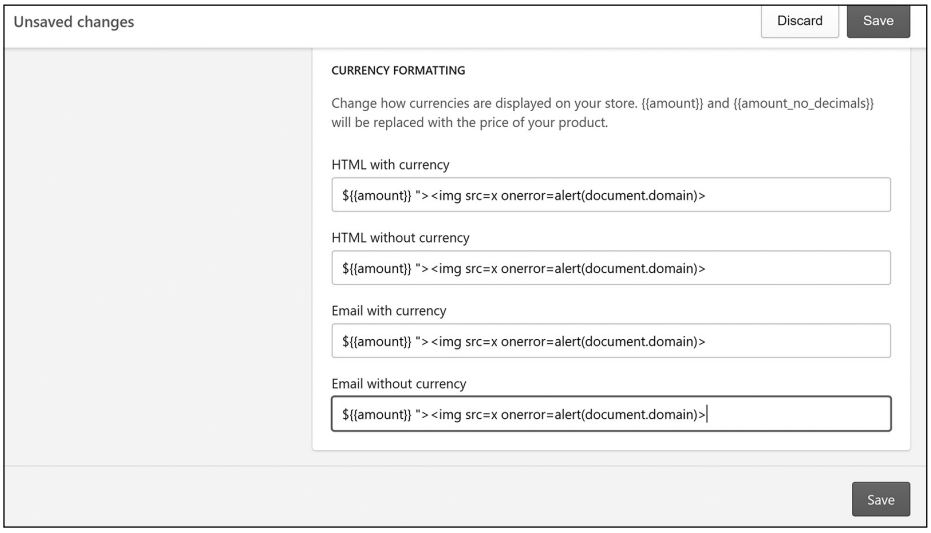

Abb. 7–3 *Shopifys Währungseinstellung zum Zeitpunkt des Reports*

Zwar wird der JavaScript-Code nicht ausgeführt, wenn der Nutzer die Währungsseite besucht, doch sie taucht in den Social-Media-Vertriebskanälen der Shopify-Shops auf. Klickt ein anderer Shop-Administrator den verwundbaren Vertriebskanal-Tab an, wird der bösartige XSS-Code ungeschützt gerendert und der JavaScript-Code ausgeführt.

7.3.1 Was wir mitnehmen

XSS-Payloads werden nicht immer sofort ausgeführt, nachdem sie übertragen wurden. Da eine Payload an mehreren Stellen einer Site genutzt werden kann, sollten Sie alle Orte untersuchen. In diesem Fall hat das Senden der bösartigen Payload auf der Währungsseite den XSS-Code nicht sofort ausgeführt. Der Angreifer musste ein anderes Feature der Website konfigurieren, um den XSS-Code zur Ausführung zu bringen.

7.4 Gespeichertes XSS bei Yahoo! Mail

Schwierigkeitsgrade: Mittel
URL: Yahoo! Mail
Quelle: *https://klikki.fi/adv/yahoo.html*
Meldezeitpunkt: 26. Dezember 2015
Gezahltes Bounty: $ 10.000

Die Bereinigung der vom Nutzer übergebenen Eingaben durch die Modifikation des übergebenen Texts kann zu Problemen führen, wenn man es nicht richtig macht. In diesem Beispiel erlaubte es der Editor von Yahoo! Mail den Nutzern, Bilder in HTML mittels `<img>`-Tag in E-Mails einzufügen. Der Editor bereinigte die Daten, indem er jegliche JavaScript-Attribute wie `onload`, `onerror` und so weiter entfernte, um XSS-Schwachstellen zu verhindern. Leider wurden dabei die Schwachstellen vergessen, zu denen es kommt, wenn ein Nutzer bewusst fehlerhafte `<img>`-Tags verwendet.

Die meisten HTML-Tags akzeptieren Attribute, die zusätzliche Informationen für den Tag enthalten. Zum Beispiel benötigt das `<img>`-Tag ein `src`-Attribut, das die Adresse des auszugebenden Bilds enthält. Das Tag erlaubt auch die Attribute `width` und `height`, die die Größe des Bilds definieren.

Einige HTML-Attribute sind boolesche Attribute: Sind sie in einem HTML-Tag enthalten, sind sie definiert, anderenfalls nicht.

Bei dieser Schwachstelle fand Jouko Pynnonen heraus, dass, wenn er boolesche Attribute an HTML-Tags mit einem Wert anhängte, Yahoo! Mail zwar die Werte entfernte, das Gleichheitszeichen des Attributs aber erhalten blieb. Hier eines von Pynnonens Beispielen:

```
<INPUT TYPE="checkbox" CHECKED="hello" NAME="check box">
```

Hier kann das HTML-input-Tag ein `CHECKED`-Attribut enthalten, das festlegt, ob die Checkbox aktiv ist. Basierend auf Yahoos Tag-Parsing wurde die Zeile zu:

```
<INPUT TYPE="checkbox" CHECKED= NAME="check box">
```

Das sieht harmlos aus, doch HTML erlaubt null oder mehr Whitespace-Zeichen um einen nicht in Anführungszeichen stehenden Attribut-Wert. Browser interpretieren `CHECKED` dann so, als hätte es den Wert `NAME="check`, und glauben, dass das Input-Tag ein drittes Argument namens `box` ohne einen Wert enthält.

Um das auszunutzen, hat Pynnonen das folgende `<img>`-Tag gesendet:

```
<img ismap='xxx' itemtype='yyy style=width:100%;height:100%;position:fixed;
  left:0px;top:0px; onmouseover=alert(/XSS/)//'>
```

Die Yahoo!-Mail-Filter machten daraus Folgendes:

```
<img ismap= itemtype='yyy' style=width:100%;height:100%;position:fixed;left:
  0px;top:0px; onmouseover=alert(/XSS/)//>
```

Das boolesche `ismap`-Attribut des `<img>`-Tags gibt an, ob ein Bild klickbare Bereiche enthält. Im obigen Beispiel hat Yahoo! das `'xxx'` entfernt, und das einfache Anführungszeichen vom Ende des Strings wurde an das Ende von `yyy` verschoben.

Manchmal ist das Backend einer Site eine Blackbox, und Sie wissen (wie in diesem Fall) nicht, wie der Code verarbeitet wird. Wir wissen nicht, warum das `'xxx'` entfernt wurde und warum das einfache Anführungszeichen an das Ende von `yyy` verschoben wurde. Yahoos Parsing-Engine oder die Art und Weise, wie der Browser mit den von Yahoo! gelieferten Daten umgeht, können für diese Änderungen verantwortlich sein. Trotzdem können Sie dieses kuriose Verhalten nutzen, um Schwachstellen aufzuspüren.

Die Verarbeitung des Codes führte dazu, dass ein `<img>`-Tag mit einer Höhe und Breite von 100 Prozent gerendert wurde, sodass das Bild das gesamte Browser-Fenster einnahm. Bewegte der Nutzer die Maus über die Webseite, wurde die XSS-Payload aufgrund des eingeschleusten `onmouseover=alert(/XSS/)`-Teils der Injection ausgeführt.

7.4.1 Was wir mitnehmen

Entschärfen Sites die Nutzereingaben, indem sie diese modifizieren, statt Werte zu codieren oder zu entfernen, sollten Sie die serverseitige Logik der Site tiefer untersuchen. Denken Sie darüber nach, welche Annahmen die Entwickler getroffen und wie sie ihre Lösung codiert haben. Prüfen Sie zum Beispiel, ob Entwickler darüber nachgedacht haben, was passiert, wenn zwei `src`-Attribute gesendet oder wenn Leerzeichen durch Slashes ersetzt werden. In diesem Fall hat sich der Hacker angesehen, was passiert, wenn boolesche Attribute mit Werten übertragen werden.

7.5 Google-Bildersuche

Schwierigkeitsgrad: Mittel
URL: *images.google.com/*
Quelle: *https://mahmoudsec.blogspot.com/2015/09/how-i-found-xss-vulnerability-in-google.html*
Meldezeitpunkt: 12. September 2015
Gezahltes Bounty: Nicht veröffentlicht

Je nachdem, wo Ihre Eingabe gerendert wird, müssen Sie nicht immer spezielle Zeichen verwenden, um XSS-Schwachstellen auszunutzen. Im September 2015 nutzte Mahmoud Jamal Google-Bilder, um ein Bild für sein HackerOne-Profil

zu finden. Beim Stöbern bemerkte er den Image-URL *http://www.google.com/imgres?imgurl=https://lh3.googleuser.com/...* von Google.

Als er die Referenz auf `imgurl` im URL bemerkte, erkannte Jamal, dass er den Wert des Parameters kontrollieren konnte. Er würde auf der Site wahrscheinlich als Link gerendert werden. Er bewegte sich über das Vorschaubild seines Profils und konnte so bestätigen, dass ein `href`-Attribut im `<a>`-Tag des gleichen URLs enthalten war. Er versuchte daher, den `imgurl`-Parameter in `javascript:alert(1)` zu ändern, und bemerkte, dass das `href`-Attribut den gleichen Wert enthielt.

Diese `javascript:alert(1)`-Payload ist nützlich, wenn Sonderzeichen herausgefiltert werden, denn er enthält keine Sonderzeichen, die die Website codieren müsste. Beim Klick auf einen Link auf `javascript:alert(1)` öffnet sich ein neues Browser-Fenster, und die `alert`-Funktion wird ausgeführt. Da der JavaScript-Code im Kontext der den Link enthaltenden, ursprünglichen Webseite ausgeführt wird, kann der Code auf das DOM dieser Seite zugreifen. Mit anderen Worten würde ein Link auf `javascript:alert(1)` die `alert`-Funktion gegen Google ausführen. Dieses Ergebnis zeigte, dass ein bösartiger Angreifer potenziell auf Informationen der Webseite zugreifen konnte. Wird durch das Anklicken eines Links auf das JavaScript-Protokoll nicht der Kontext der ursprünglichen Site geerbt, die den Link rendert, ist das XSS harmlos: Angreifer können nicht auf das DOM der anfälligen Site zugreifen.

Entzückt klickte Jamal die Stelle an, wo er seinen bösartigen Link vermutete, doch es wurde kein JavaScript-Code ausgeführt. Google bereinigte die URL-Adresse über `onmousedown`-JavaScript-Attribut des Anker-Tags, wenn eine Maustaste gedrückt wurde.

Um das zu umgehen, versuchte Jamal, sich mit der Tabulator-Taste durch die Seite zu bewegen. Als er den View-Image-Button erreichte, drückte er `enter`. Der JavaScript-Code wurde ausgeführt, weil er den Link besuchen konnte, ohne die Maustaste drücken zu müssen.

7.5.1 Was wir mitnehmen

Halten Sie immer nach URL-Parametern Ausschau, die auf der Seite wiedergegeben werden, da Sie die Kontrolle über diese Werte haben. Finden Sie URL-Parameter, die auf einer Seite gerendert werden, dann beachten Sie auch den Kontext. URL-Parameter können eine Möglichkeit sein, Filter zu umgehen, die spezielle Zeichen aussortieren. In diesem Beispiel musste Jamal keine speziellen Zeichen senden, weil der Wert als `href`-Attribut in einem Anker-Tag gerendert wurde.

Darüber hinaus sollten Sie auch bei Google und anderen großen Sites nach Schwachstellen Ausschau halten. Nur weil ein Unternehmen sehr groß ist, heißt

das noch lange nicht, dass dort alle Schwachstellen entdeckt wurden. Natürlich ist das nicht immer der Fall.

7.6 Google Tag Manager: Gespeichertes XSS

Schwierigkeitsgrad: Mittel
URL: *tagmanager.google.com/*
Quelle: *https://blog.it-securityguard.com/bugbounty-the-5000-google-xss/*
Meldezeitpunkt: 31. Oktober 2014
Gezahltes Bounty: $ 5.000

Eine bei Websites übliche Praxis besteht darin, Nutzereingaben während des Renderings zu filtern und nicht, wenn sie bei der Übertragung gespeichert werden. Das liegt oft daran, dass neue Möglichkeiten, Daten an eine Site zu senden (etwa ein Datei-Upload), schnell eingeführt sind, dabei aber das Filtern der Eingaben vergessen wird. In manchen Fällen folgen Unternehmen dieser Praxis allerdings nicht: Patrik Fehrenbach von HackerOne entdeckte einen solchen Lapsus im Oktober 2014, als er Google auf XSS-Schwachstellen untersuchte.

Der Google Tag Manager ist ein SEO-Tool, das es Vermarktern erlaubt, Website-Tags einzufügen und zu aktualisieren. Das Tool verfügt dazu über eine Reihe von Webformularen, mit denen die Nutzer interagieren. Fehrenbach begann mit der Untersuchung verfügbarer Formularfelder und fügte XSS-Payloads wie `#"><img src=/ onerror=alert(3)>` ein. Würde die Payload aus dem Formularfeld akzeptiert werden, würde die Payload das vorhandene HTML-Tag schließen und dann versuchen, ein nicht vorhandenes Bild zu laden. Da das Bild nicht vorhanden ist, würde die Website die `onerror`-Funktion `alert(3)` ausführen.

Doch Fehrenbachs Payload funktionierte nicht. Google filterte seine Eingaben sauber heraus. Fehrenbach bemerkte eine alternative Möglichkeit, seine Payload zu senden. Zusätzlich zu den Formularfeldern bot Google die Möglichkeit, eine JSON-Datei mit mehreren Tags hochzuladen. Fehrenbach lud also die folgende JSON-Datei an Googles Dienst hoch:

```
"data": {
  "name": "#"><img src=/ onerror=alert(3)>",
  "type": "AUTO_EVENT_VAR",
  "autoEventVarMacro": {
    "varType": "HISTORY_NEW_URL_FRAGMENT"
  }
}
```

Wie Sie sehen, entspricht der Wert des `name`-Attributs der XSS-Payload, mit der es Fehrenbach vorher probiert hatte. Google folgte der bewährten Praxis diesmal nicht und filterte die Eingabe aus dem Web bei der Übertragung, nicht bei der Ausgabe. Folglich vergaß Google das Filtern der Eingabe aus der hochgeladenen Datei, und Fehrenbachs Payload wurde ausgeführt.

7.6.1 Was wir mitnehmen

Zwei Details sind in Fehrenbachs Report erwähnenswert. Erstens fand Fehrenbach eine alternative Methode, um seine XSS-Payload zu übertragen. Auch Sie sollten nach alternativen Möglichkeiten Ausschau halten. Probieren Sie alle Methoden aus, die ein Angriffsziel zur Eingabe bereitstellt, da die Verarbeitung bei jeder Methode anders ausfallen kann. Zweitens filterte Google die Eingabe bei der Übertragung und nicht bei der Ausgabe. Google hätte diese Schwachstelle vermeiden können, wenn es den bewährten Praktiken gefolgt wäre. Selbst wenn Sie wissen, dass Website-Entwickler die üblichen Gegenmaßnahmen treffen, um sich vor bestimmten Angriffen zu schützen, sollten Sie nach Schwachstellen suchen. Entwickler können Fehler machen.

7.7 XSS bei United Airlines

Schwierigkeitsgrad: Hoch
URL: *checkin.united.com/*
Quelle: *http://strukt93.blogspot.jp/2016/07/united-to-xss-united.html*
Meldezeitpunkt: Juli 2016
Gezahltes Bounty: Nicht veröffentlicht

Im Juli 2016 suchte Mustafa Hasan nach günstigen Flügen und hielt gleichzeitig nach Bugs auf United-Airlines-Sites Ausschau. Er fand heraus, dass der Besuch der Subdomain *checkin.united.com* ihn auf einen URL umleitete, der einen `SID`-Parameter enthielt. Er bemerkte, dass jeder an den Parameter übergebene Wert auf der HTML-Seite gerendert wurde, und probierte daraufhin `"><svg onload=confirm(1)>` aus. Ein unsauberes Rendering würde das vorhandene HTML-Tag schließen und Hasans `<svg>`-Tag einschleusen, was wiederum aufgrund des `onload`-Events ein JavaScript-Pop-up öffnen würde.

Doch als er seinen HTTP-Request sendete, passierte nichts, obwohl seine Payload ungefiltert gerendert wurde. Statt aufzugeben, öffnete Hasan die JavaScript-Dateien der Site, vermutlich mit den Entwickler-Tools des Browsers.

Er fand den folgenden Code, der JavaScript-Attribute überschrieb, die zu XSS führen konnten, etwa die Attribute `alert`, `confirm`, `prompt`, und `write`:

```
[function () {
/*
XSS prevention via JavaScript
*/
var XSSObject = new Object();
XSSObject.lockdown = function(obj,name) {
    if (!String.prototype.startsWith) {
        try {
            if (Object.defineProperty) {
                Object.defineProperty(obj, name, {
                    configurable: false
                });
            }
        } catch (e)  { };
    }
}
XSSObject.proxy = function (obj, name, report_function_name, ❶exec_original)
{
    var proxy = obj[name];
    obj[name] = function () {
        if (exec_original) {
            return proxy.apply(this, arguments);
        }
    };
    XSSObject.lockdown(obj, name);
};
❷ XSSObject.proxy(window, 'alert', 'window.alert', false);
XSSObject.proxy(window, 'confirm', 'window.confirm', false);
XSSObject.proxy(window, 'prompt', 'window.prompt', false);
XSSObject.proxy(window, 'unescape', 'unescape', false);
XSSObject.proxy(document, 'write', 'document.write', false);
XSSObject.proxy(String, 'fromCharCode', 'String.fromCharCode', true);
}]();
```

Selbst wenn Sie JavaScript nicht kennen, werden Sie durch die Nutzung bestimmter Wörter wohl ahnen, was passiert. Zum Beispiel impliziert der Parameter-Name `exec_original` ❶ in der Definition von `XSSObject proxy` eine Beziehung, die etwas ausführt. Unmittelbar unter dem Parameter ist eine Liste aller uns interessierender Funktionen und der übergebene Wert `false` (mit Ausnahme der letzten Instanz) ❷. Wir können davon ausgehen, dass sich die Site zu schützen versucht, indem sie die Ausführung der an `XSSObject proxy` übergebenen JavaScript-Attribute unterbindet.

Da JavaScript das Überschreiben existierender Funktionen erlaubt, versuchte Hasan zuerst, die Funktion `document.write` wiederherzustellen, indem er den folgenden Wert an `SID` übergab:

```
javascript:document.write=HTMLDocument.prototype.write;document.write('STRUKT');
```

Dieser Wert setzt die `write`-Funktion des Dokuments mithilfe des `write`-Funktionsprototyps auf seine ursprüngliche Funktionalität zurück. Mit dem Aufruf von `HTMLDocument` setzte Hasan die aktuell gesetzte `write`-Funktion auf die ursprüngliche Implementierung von `HTMLDocument` zurück. Dann rief er `document.write('STRUKT')` auf, um seinen Namen im Klartext in die Seite einzufügen.

Doch als Hasan versuchte, die Schwachstelle auszunutzen, blieb er wieder hängen und bat Rodolfo Assis um Hilfe. Bei ihrer gemeinsamen Arbeit entdeckten sie, dass Uniteds XSS-Filter eine `write` sehr ähnliche Funktion nicht überschrieb: die `writeln`-Funktion. Der Unterschied zwischen diesen beiden Funktionen besteht darin, dass `writeln` ein Newline an den Text anhängt, `write` hingegen nicht.

Assis glaubte die `writeln`-Funktion nutzen zu können, um Inhalte in das HTML-Dokument einzufügen. Das würde es ihm erlauben, einen Teil von Uniteds XSS-Filter zu umgehen. Er versuchte es mit der folgenden Payload:

```
";}{document.writeln(decodeURI(location.hash))-"#<img src=1 onerror=alert(1)>
```

Doch dieser JavaScript-Code wurde ebenfalls nicht ausgeführt, weil der XSS-Filter immer noch geladen wurde und die `alert`-Funktion überschrieb. Assis musste eine andere Methode nutzen. Bevor wir die finale Payload vorstellen und zeigen, wie Assis den `alert`-Override umgehen konnte, sehen wir uns die ursprüngliche Payload genauer an.

Der erste Teil, `";}`, schließt den vorhandenen JavaScript-Code ab, in den er eingeschleust wird. Das `{` öffnet dann die JavaScript-Payload, und `document.writeln` ruft die `writeln`-Funktion des JavaScript-Document-Objekts auf, um Inhalte in das DOM zu schreiben. Die an `writeln` übergebene Funktion `wdecodeURI` decodiert im URL codierte Entitäten (aus `%22` wird beispielsweise `"`). Der `location.hash`-Code in `decodeURI` gibt alle Parameter zurück, die im URL auf das (später definierte) `#` folgen. Nach diesem ersten Setup ersetzt `-"` das Anführungszeichen am Anfang der Payload, um die korrekte JavaScript-Syntax sicherzustellen.

Der letzte Teil, `#<img src=1 onerror=alert(1)>`, fügt einen Parameter hinzu, der nie an den Server gesendet wird. Er bildet einen definierten, optionalen Teil eines URLs, ein sogenanntes *Fragment* und verweist auf einen Teil des Dokuments. Doch in diesem Fall nutzte Assis das Fragment, um den Hash (`#`) nutzen

zu können, der den Anfang des Fragments definiert. Die Referenz auf `location.hash` gibt den gesamten auf das `#` folgenden Inhalt zurück. Der zurückgegebene Inhalt ist allerdings URL-codiert, weshalb `<img src=1 onerror=alert(1)>` als `%3Cimg%20src%3D1%20onerror%3Dalert%281%29%3E%20` zurückgegeben wird. Darum wird `decodeURI` verwendet, um den Inhalt wieder in den HTML-Code `<img src=1 onerror=alert(1)>` umzuwandeln. Das ist wichtig, weil der decodierte Werte an die `writeln`-Funktion übergeben wird, die das HTML-Tag `<img>` in das DOM schreibt. Das HTML-Tag führt den XSS-Code aus, wenn die Site das Image `1` nicht findet, das im `src`-Attribut des Tags referenziert wird. Ist die Payload erfolgreich, erscheint ein Alert-Dialog mit der Zahl `1`. Doch es passierte nichts.

Assis und Hasan erkannten, dass sie ein neues HTML-Dokument im Kontext der United-Site benötigen: Eine Seite, bei der JavaScript-XSS-Filter nicht geladen sind, die aber dennoch Zugriff auf die Webseiten-Informationen, Cookies und so weiter der United-Webseite hat. Sie verwendeten daher ein iFrame mit der folgenden Payload:

```
";}{document.writeln(decodeURI(location.hash))-"#<iframe
src=javascript:alert(document.domain)><iframe>
```

Diese Payload verhält sich wie der ursprüngliche URL mit dem `<img>`-Tag. Doch diesmal wird ein `<iframe>` in das DOM geschrieben und das `src`-Attribut so geändert, dass es das JavaScript-Schema für `alert(document.domain)` nutzt. Diese Payload ähnelt der XSS-Schwachstelle, die wir in »Google-Bildersuche« auf Seite 77 vorgestellt haben, da das JavaScript-Schema den Kontext von seinem Parent-DOM erbt. Der XSS-Code kann nun auf das United-DOM zugreifen, weshalb `document.domain` den Wert *www.united.com* ausgibt. Die Schwachstelle wurde bestätigt, als die Site den Alert-Dialog öffnete.

Ein iFrame kann ein src-Attribut angeben, um entferntes HTML einzubinden. Dementsprechend konnte Assis die Quelle als JavaScript-Code festlegen, wodurch die `alert`-Funktion sofort aufgerufen wurde und die Domäne ausgab.

7.7.1 Was wir mitnehmen

Beachten Sie drei wichtige Details dieser Schwachstelle. Erstens was Hasan hartnäckig. Statt aufzugeben, als seine Payload nicht lief, tauchte er in den JavaScript-Code ein, um herauszufinden, warum. Zweitens ist die Verwendung von JavaScript-Attribut-Blacklisting ein Hinweis für Hacker, dass XSS-Bugs existieren könnten, da die Entwickler dabei Fehler machen können. Drittens ist JavaScript-Wissen unerlässlich, um komplexere Schwachstellen erfolgreich aufspüren zu können.

7.8 Zusammenfassung

XSS-Schwachstellen sind eine reale Bedrohung für Site-Entwickler und auf Sites noch weit verbreitet (und oft deutlich sichtbar). Durch das Senden einer bösartigen Payload wie `<img src=x onerror=alert(document.domain)>` können Sie prüfen, ob ein Eingabefeld anfällig ist. Doch das ist nicht die einzige Möglichkeit, XSS-Schwachstellen aufzuspüren. Filtert eine Site Eingaben durch Modifikation (der Entfernung von Zeichen, Attributen und so weiter),dann sollten Sie die Filterfunktionalität sorgfältig untersuchen. Achten Sie auf Gelegenheiten, bei denen Sites Eingaben während des Sendens filtern und nicht während der Ausgabe. Untersuchen Sie auch alle Eingabemethoden. Achten Sie auch darauf, wie von Ihnen kontrollierte URL-Parameter auf der Seite erscheinen. Sie könnten einen XSS-Exploit finden, der die Codierung unterwandert, zum Beispiel, indem man `javascript:alert(document.domain)` in den `href`-Wert eines Anker-Tags einfügt.

Es ist auch wichtig, auf alle Stellen zu achten, an denen eine Site Ihre Eingaben rendert, und zu wissen, ob es sich dabei um HTML oder JavaScript handelt. Denken Sie daran, dass XSS-Payloads nicht sofort ausgeführt werden müssen.