INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption und Entwicklung von Methoden zur interaktiven Sicherheitsuntersuchung von Web 2.0 Anwendungen

Norbert Hoene

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Konzeption und Entwicklung von Methoden zur interaktiven Sicherheitsuntersuchung von Web 2.0 Anwendungen

Norbert Hoene

Aufgabensteller:	Prof. Dr. Rolf Hennicker
Betreuer:	Christian Kroiß Gefei Zhang Dr. Holger Dreger (Siemens) Dr. Jorge Cuellar (Siemens) Johann Wallinger (Siemens)
Abgabetermin:	18. März 2010

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. März 2010

.....

(Unterschrift des Kandidaten)

Abstract

Im Rahmen dieser Diplomarbeit steht die Konzeption und Erweiterung von Methoden zur interaktiven Sicherheitsuntersuchung mit Hilfe des Web Application Attack and Audit Frameworks (w3af).
Ziel ist es, eine client-seitige Schnittstelle für den Benutzer zu schaffen, über die er die zu untersuchenden Webanwendungen komfortabel analysieren kann. Dazu sollen zunächst ausgewählte Angriffsszenarien auf Webanwendungen modelliert werden, die den Ablauf eines solchen Angriffs näher spezifizieren. Danach wird eine neue Architektur entworfen, die im Kern aus einem Webproxy und einer WebGUI besteht. Hierbei soll sich die WebGUI in eine bestehende Webanwendung automatisch einbetten, welche dann dem komfortablen Analysieren der Webanwendung auf Schwachstellen dient. So soll diese WebGUI unter anderem eine Webanwendung auf ihre bestehenden Eingabeobjekte hin untersuchen und eine Möglichkeit der Manipulation der Daten bereitstellen. Hierbei sollen auch Techniken der neuen Webgeneration berücksichtigt werden. Der Webproxy dient dabei als Transformator der Kommunikation zwischen der WebGUI und die zu untersuchende Webanwendung. Er stellt die benötigten Daten zum client-seitigen Analysieren einer Webanwendung bereit. Außerdem soll im Webproxy eine Möglichkeit geschaffen werden, einzelne Methoden zum Angriff auf Webanwendungen zu integrieren, so dass diese Methoden über die WebGUI genutzt werden können. Hierbei wird zusätzlich ein Verfahren entwickelt, welches Schwachstellen in Webanwendungen über mehrere Webseiten erkennt. Im letzten Schritt soll die Architektur in einem Prototyp umgesetzt werden. Dazu werden zuvor ausgewählte Angriffsszenarien mit Unterstützung für das seitenübergreifende Erkennen von Schwachstellen in die neue Architektur integriert.

Inhaltsverzeichnis

1. Einleitung

Das Internet ist in der heutigen Zeit ein wichtiges Medium des Alltags geworden. Es unterstützt die Menschheit in vielen Aspekten des alltäglichen Lebens. So dient es als wichtiges Instrument zum Austauschen von Informationen auf der ganzen Welt. Aus diesem Grund sind Wissensdatenbanken wie Wikipedia oder auch soziale Netzwerke wie Facebook wichtige Plattformen für uns, um Informationen jeglicher Art mit unseren Mitmenschen auszutauschen. Aber auch im eCommerce-Bereich hat das Internet eine zentrale Position eingenommen. So werden heutzutage ein Großteil der Geschäfte über das Internet getätigt, so dass das Internet auch einen wichtigen Wirtschaftsfaktor für viele Menschen darstellt.

All diese Plattformen haben gemeinsam, dass diese als Webanwendungen im Internet abrufbar sind. Hierbei verwenden viele Plattformen Technologien der neuen Web-Generation, die es dem Benutzer erlauben, interaktiv mit der Webanwendung zu kommunizieren. Bei der Entwicklung derartiger Webanwendungen bleiben jedoch meistens sicherheitsrelevante Aspekte außen vor oder werden nicht korrekt umgesetzt. So zeigt eine aktuelle Studie des *Web Application Security Consortium* [Con07], dass rund 97 Prozent aller Webanwendungen Schwachstellen enthalten, bei denen sensible Daten gestohlen oder sogar manipuliert werden können. Um einem Missbrauch der sensiblen Daten vorzubeugen, ist Sicherheit im Internet heutzutage ein sehr wichtiges Thema. So lassen sich in diesem Bereich viele Publikationen und Anwendungen finden, die es sich zum Ziel gesetzt haben, einzelne Webapplikationen automatisch oder manuell auf Schwachstellen zu untersuchen. Durch die jedoch ansteigende Entwicklung von Technologien in Webanwendungen und damit der potentiellen Erhöhung der Schwachstellen, werden Untersuchungen zur Schwachstellen-Analyse in Webanwendungen immer wichtiger.

1.1. Motivation

Die Anforderungen von Applikationen zur Schwachstellen-Analyse steigen somit mit der Entwicklung des Internets stetig an. Zwar liefern automatische Analyse-Programme ein gutes Ergebnis bei klassischen Webanwendungen welche die aktuellsten Web-Technologien nicht einsetzen, sind jedoch bei neueren, interaktiven Web-Anwendungen zunehmend überfordert, was sich in der gefundenen Anzahl der Schwachstellen bei einer Webanwendung widerspiegelt. Auch können zur Zeit nicht alle Arten von bekannten Schwachstellen automatisch in Webanwendungen gefunden werden, da für manche Typen von Schwachstellen noch keine geeigneten Algorithmen zum Finden dieser Schwachstellen entwickelt wurden. Ein weiterer Aspekt ist, dass die auf dem Markt befindlichen Programme bei der Schwachstellen-Analyse nur den jeweiligen Response einer Anfrage untersuchen. Jedoch kann es aber auch vorkommen, dass sich Schwachstellen in Webanwendungen über mehrere Webseiten bewegen können, was mit den zur Zeit auf dem Markt befindlichen Methoden nur manuell erkannt werden kann. Diese Tatsachen wirken sich letztendlich negativ auf die Qualität des Ergebnisses des durchgeführten automatischen Analyse-Tests aus. Aus diesem Grund ist zusätzlich

das manuelle Untersuchen von Webanwendungen auf vorhandene Schwachstellen eine wichtige Aufgabe, um die Qualität der Sicherheit einer Webanwendung genauer zu beurteilen. Jedoch ist dieses Verfahren sehr zeitaufwendig, was sich auf die finanziellen Ausgaben einer Schwachstellen-Analyse auswirkt.
Das Ziel dieser Diplomarbeit stellt eine neue Variante zur Schwachstellen-Untersuchung bei Webanwendungen dar. Hierbei werden die Ansätze des automatischen und manuellen Testens vereint, so dass der Tester bei der manuellen Untersuchung der Webapplikation auf Schwachstellen in sämtlichen Phasen anhand automatischer Verfahren soweit wie möglich unterstützt wird. Auch werden hierbei Techniken der neuen Webgeneration berücksichtigt, so dass der Tester selbst die heutigen Webanwendungen der neuen Generation untersuchen kann. Als weitere Neuerung stellt diese Diplomarbeit ein Verfahren vor, das es erlaubt, selbst Schwachstellen über mehrere Webseiten hin automatisch zu erkennen. Hierbei wird auch aufgezeigt, wie Angriffsalgorithmen auf Webanwendungen dieses entwickelte Verfahren verwenden können.

1.2. Aufgabenstellung

Für diese Aufgabe soll das Open-Source-Projekt *Web Application Attack and Audit Framework* (w3af), welches bei der Siemens AG unter anderem zur Schwachstellen-Analyse in Webanwendungen eingesetzt wird, erweitert werden. Das in der Programmiersprache Python entwickelte Framework bietet durch seinen modularen Aufbau eine Möglichkeit zur Realisierung dieser neuen Architektur an. Die hierbei zu entwickelnden Konzepte werden dann in einem Prototypen zusammengefasst. Hierfür sollen zunächst Methoden entworfen und entwickelt werden, die es dem Tester ermöglichen, alle abgehenden Anfragen, einschließlich der neuen Web-Technologien, von einer Webanwendung zum Webserver noch im Webbrowser selber abzufangen und zu manipulieren. Dieses ermöglicht es dem Anwendungstester die gesamte Webanwendung auf Schwachstellen vom Webbrowser aus zu untersuchen.
Darauf aufbauend soll eine Methode entworfen werden, die es dem Tester einer Webanwendung ermöglicht, mit Hilfe von leicht konfigurierbaren Patterns zur Schwachstellen-Analyse, eine Suche auf Schwachstellen in einer Webanwendung zu ermöglichen. Hierbei ist zu beachten, dass spätere Entwickler zusätzliche Module (Plugins) zur Erkennung anderer Arten von Schwachstellen in den Prototypen implementieren können. Damit der Prototyp auch einsetzbar ist, sollen zusätzlich zwei Plugins zur Erkennung von SQL-Injection und Cross-site-scripting (XSS) implementiert werden. Im Anschluss soll anhand eines Anwendungsbeispiels die Qualität der entwickelten Architektur dargestellt werden.

1.3. Lösungsansatz

Um die oben beschriebenen Funktionen umzusetzen, wurden zunächst geeignete Konzepte zum client-seitigen Abfangen von Requests zum Webserver evaluiert und verglichen. Hierbei wurden auch die Technologien der neuen Web-Generation berücksichtigt. Das Ergebnis bei diesem Test ist die Verwendung von Javascript-Techniken zum Analysieren einer Webseite. Darauf aufbauend wurden verschiedene Methoden entwickelt, die mit den unterschiedlichen Techniken zum Erzeugen und Absenden eines Requests umgehen können.

Mit diesen Wissen wurde im nächsten Schritt ein Verfahren entwickelt, welches sogar Schwachstellen über mehrere Webseiten automatisch erkennt. Zusätzlich wurden zwei Plugins zur Erkennung von SQL-Injection und Cross-site-scripting (XSS) entworfen, die auf dieses Verfahren zur seitenübergreifende Suche von Schwachstellen zurückgreifen. Die Methode zur seitenübergreifenden Suche nach Schwachstellen basiert hierbei auf das Abspeichern zuvor definierter Requests, welche dann mit Hilfe der integrierten Plugins, auf bestimmte Merkmale (Patterns) untersucht werden.
Diese einzelnen Techniken wurden dann in einer grafischen Benutzeroberfläche zusammengefasst, die sich in die zu untersuchende Webanwendung automatisch einbettet, so dass der Tester die gesamte Webanwendung vom Webbrowser aus untersuchen kann. Auch wurde eine Möglichkeit in die eingebettete Benutzeroberfläche integriert, die es dem Tester erlaubt, einzelne zuvor definierte Abschnitte der Webanwendung automatisch mit Hilfe der seitenübergreifenden Suche auf Schwachstellen zu untersuchen, so dass im letzten Schritt der entwickelte Prototyp auf seine Praxistauglichkeit getestet wurde. Hierbei wurde anhand eines praxisnahen Beispiels aufgezeigt, dass die entwickelten Techniken, wie das seitenübergreifende Suchen nach Schwachstellen, korrekt funktionieren.

1.4. Aufbau der Arbeit

Die vorliegende Diplomarbeit ist in sechs Kapiteln aufgeteilt, wobei das Kapitel 1 als Einleitung die Motivation und das Ziel dieser Diplomarbeit erläutert. Im zweiten Kapitel dieser Arbeit werden die Begriffe erklärt, welche für das spätere Verständnis der darauf folgenden Abschnitte benötigt werden. Hier werden unter anderem sicherheitsrelevante Probleme zu den neuen Web-Technologien aufgezeigt. Außerdem wird das zu erweiternde *Web Application Attack and Audit Framework* (w3af) in seiner Funktionsweise dargestellt. Das Kapitel endet mit der Darstellung einer Klassifizierung der auf dem Markt befindlichen Ansätze, welches die Klassen auf ihre Vorteile und Nachteile untersucht. Das Ergebnis dieses Vergleichs stellt eine Rechtfertigung dieser Diplomarbeit dar.

Das Kapitel 3 stellt den Entwurf inklusive der getroffenen Entscheidungen dar. Es erläutert unter anderem die Funktionsweise des seitenübergreifenden Erkennens von Schwachstellen. Außerdem erklärt dieses Kapitel den Aufbau der neuen Architektur, welche als Prototyp umgesetzt wurde. Die hierbei gewonnenen Erkenntnisse werden im Kapitel 4, der Implementierung, in das w3af integriert. Hier wird auch aufgezeigt, wie spätere Entwickler zusätzliche Plugins zum Erkennen von Schwachstellen in den Prototypen integrieren können.

Im Kapitel 5 wird der implementierte Prototyp auf seine Praxistauglichkeit getestet. Dieses wird anhand eines praxisnahen Anwendungsbeispiels aufgezeigt. So zeigt dieses Beispiel wie automatisch die Schwachstellen unter Einbeziehung der seitenübergreifenden Suche mit Hilfe des Prototypen entdeckt werden können.

Der Hauptteil dieser Diplomarbeit endet mit dem sechsten Kapitel, welches eine Zusammenfassung der gewonnenen Erkenntnisse bei diesem Projekt darstellt. Außerdem bietet es Anregungen für zukünftige Arbeiten auf dem Gebiet der semi-automatischen Schwachstellen-Erkennung in Webanwendungen.

Der Anhang dieser Ausarbeitung beinhaltet eine kurze Installationsanleitung, wie der Prototyp korrekt installiert und konfiguriert wird.

2. Grundlagen und aktueller Stand der Technik

2.1. Grundlagen

Im folgenden Abschnitt werden wichtige Begriffe erläutert, die für das spätere Verständnis benötigt werden.

2.1.1. Techniken zum Senden eines Requests bei HTTP

Zur Darstellung einer Webseite bei HTTP oder HTTPS kommt im allgemeinen HTML zum Einsatz. Hierbei können Entwickler mit Hilfe von Javascript zusätzlich Programmlogik in ihre Webanwendungen einbetten. Damit nun auch Benutzer Informationen an eine Webanwendung übergeben können, gibt es in HTML und Javascript eine Reihe von Möglichkeiten die zum Übermitteln von Benutzereingaben dienen. Auch wenn die Technik bei den einzelnen Varianten sehr unterschiedlich ist, erzeugen sie letztendlich alle einen Request, welcher einer vorgegebenen Spezifikation folgt. Nachfolgend werden die zur Zeit verwendeten Techniker näher beschrieben.

Link
Ein Link auf einer Webseite stellt die einfachste Form der Übergabe von zusätzlichen Werten in einer Webanwendung dar. Alle benötigten Informationen werden bei diesem Vorgehen an den Link angehangen. Um einen Link in HTML zu erzeugen, wird das Element $\langle a\ href \rangle$ verwendet.

Link in HTML

```
<a href="http://www.mydomain.com/login.php?name=max&password=12345">
  Login
</a>
```

Submit
Ein Submit dient dem Absenden eines Webformulars in einer Webseite. Um benutzerspezifische Daten übertragen zu können, gibt der Anwender die erforderlichen Parameter in einem Formular ein. Im Anschluss betätigt er den Submit-Button, welcher aus den eingegebenen Daten einen Request erzeugt und an den Webserver sendet.

Webformular mit Submit-Button

```
<form action="http://mydomain.com/cgi-bin/output.php">
Vorname: <input name="Vorname" type="text" size="30" maxlength="30"><br>
Nachname: <input name="Zuname" type="text" size="30" maxlength="40"><br>
<input type="submit" value="Formular absenden">
</form>
```

Script-Tag
Das HTML-Tag ⟨*script src*⟩ dient zum Einbinden einer bestehenden Javascript-Datei in eine Webseite. Dieses *Includen* weiterer Javascript-Dateien wird üblicherweise im Header einer Webseite vorgenommen. Da mit Hilfe von Javascript neue Objekte hinzugefügt werden können, verwenden diese Variante oft Entwickler um weitere Funktionalitäten und Objekte in einer Webseite nachzuladen.

Dynamisches Einbinden einer Javascript-Datei

```
var head = document.getElementsByTagName("head")[0]
var newScript = document.createElement('script')
newScript.type = 'text/javascript'
newScript.src = 'http://www.mydomain.com/myNewScript.js?name=meier'
head.appendChild(newScript)
```

XMLHttpRequest
Das Objekt XMLHttpRequest steht im Zusammenhang mit Web 2.0. Mit Hilfe dieser Technik können Entwickler automatisch beim Eintreten bestimmter Ereignisse einen Request absenden. Zusätzlich bietet das XMLHttpRequest-Objekt die Möglichkeit der asynchronen Kommunikation an. Dadurch können jederzeit im Hintergrund der Webanwendung Informationen mit dem Webserver ausgetauscht werden.
Zum Absetzen eines XMLHttpRequest-Objekts kommen die Methoden *open*, *send* und *onreadystatechange* zum Einsatz. Die Methode *open* dient hierbei der Vorbereitung eines Request, welcher dann von der Methode *send* versendet wird. Zusätzliche Informationen für den Request werden je nach Request-Art (GET oder POST) über die open-Methode oder der send-Methode übergeben. Des Weiteren können Entwickler über die Methode *onreadystatechange* den aktuellen Status des Requests abfragen. Die genaue Spezifikation des XMLHttpRequest-Objekts ist unter [W3Ce] beschrieben.

Beispiel eines XMLHttpRequest

```
xmlHttp = new XMLHttpRequest()
xmlHttp.open('GET', 'example.php', true)
xmlHttp.onreadystatechange = function () {
        if (xmlHttp.readyState == 4) {
            alert(xmlHttp.responseText)
        }
    }
xmlHttp.send(null)
```

2.1.2. Schwachstellen in Webanwendungen

Obwohl die Entwickler von Webanwendungen in der heutigen Zeit alle Schwachstellen durch eine korrekte Implementierung vermeiden könnten, sind doch viele Webapplikationen als unsicher einzustufen. So belegt eine aktuelle Studie des *Web Application Security Consortiums* [Con07], dass rund 97 Prozent aller Webanwendungen Schwachstellen enthalten.

Grundsätzlich existieren Schwachstellen in Webanwendungen dadurch, wenn Entwickler einer Webanwendung die Eingaben vom Benutzer nicht ausreichend überprüfen (*Never trust user input*). Dieses hat zur Folge, dass der Benutzer auch Programmlogik in die Webanwendung übergeben kann, die im weiteren Verlauf im schlimmsten Fall sogar ausgeführt wird.

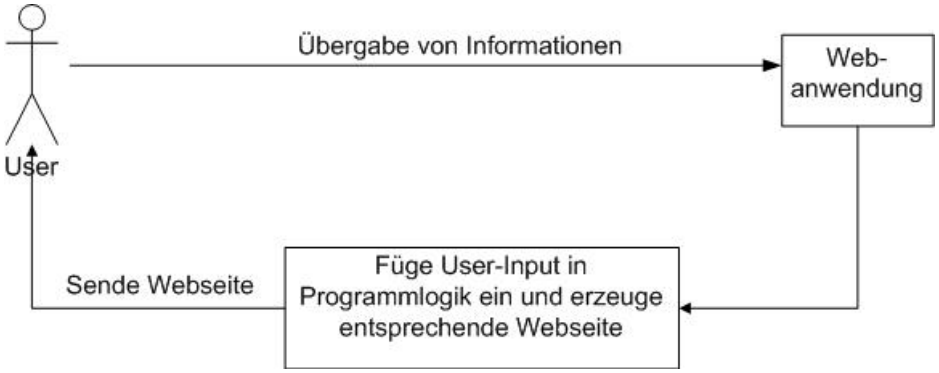


Abbildung 2.1.: Übergabe von Benutzereingaben an eine Webanwendung

Die Abbildung 2.1 stellt den Ablauf zwischen den Eingaben eines Benutzers (User) und der Verarbeitung einer Webapplikation dar. Hierbei sendet der User Daten an die Webanwendung, welche entsprechend in die Programmlogik der Webanwendung eingesetzt werden. Wenn hierbei die Applikation diese Daten ungeprüft in ihr Programmablauf übernimmt, kann unter Umständen an dieser Stelle eigener Programmcode eingeschleust werden. Nach dem Verarbeiten der Programmlogik sendet die Webanwendung im letzten Schritt eine entsprechende Antwort in Form einer Webseite zum Benutzer zurück.
Im Folgenden werden einzelne Schwachstellen, welche in Webanwendungen auftreten können, näher erläutert.

SQL-Injection

Eine SQL-Injection ist eine Variante einer Schwachstelle in Webanwendungen. Voraussetzung hierfür ist jedoch, dass die Webanwendung eine Verbindung zu einer Datenbank hat, in welcher zum Beispiel Benutzerinformationen abgespeichert sind. Werden hier keine Vorsorgemaßnahmen getroffen, ist ein Angriff auf die Datenbank möglich. Die Angriffsszenarien, die hierbei auftreten können sind vielseitig und können den gesamten Datenbankserver betreffen. So kann ein Angreifer zum Beispiel über präparierte Benutzereingaben sensible Informationen wie zum Beispiel Benutzerkonten inklusive der Passwörter ausspähen. Des Weiteren ist es möglich, Änderungen von Einträgen in Tabellen in der Datenbank persistent einzubringen. Ein weiteres Szenario wäre eine DoS-Attacke[1] gegen den Datenbankserver, so dass der Server nicht mehr ansprechbar ist.

All diese Möglichkeiten entstehen, wenn keine ausreichenden Maßnahmen im Vorhinein getroffen werden. Allgemein sollte eine Webanwendung sich über keinen Administrator-Account an die Datenbank anmelden. Dadurch können bei einer erfolgreichen SQL-Injection die Risiken erheblich abgemildert werden.

[1] Denial of Service (DoS) beschreibt einen Angriff auf einen Server, bei dem der Dienst nicht mehr genutzt werden kann.

Als weitere Maßnahme ist der richtige Einsatz von Filtern auf alle Benutzereingaben zu nennen. Da jedoch in der Realität viele Filter nicht implementiert sind, sollten darüber hinaus bei Aufrufen von SQL-Anweisungen *prepared statements* für Benutzereingaben verwendet werden.

Arten von SQL-Injection

- **normale SQL-Injection**
  Bei einem fehlerhaften SQL-Statement wird die Fehlermeldung vom SQL-Server direkt an dem Benutzer weitergeleitet ohne sie entsprechend zu verschleiern. Diese Information stellt jedoch für einen Angreifer eine wichtige Information dar, da der Angreifer sofort die Verwundbarkeit des Systems erkennen kann.

- **blinde SQL-Injection (blind SQL-Injection)**
  Hier wird die Fehlermeldung, welche bei einem fehlerhaften SQL-Statement ausgelöst wird, von der Webanwendung selber verschleiert, was dazu führt, dass es einem Angreifer erschwert wird, eine potentielle SQL-Schwachstelle in der Webanwendung zu erkennen.

Abbildung 2.2.: Verwundbare Webanwendung gegen SQL-Injection

Die Abbildung 2.2 zeigt, wie eine Schwachstelle bezüglich zu einer *normalen SQL-Injection* in einer Webananwendung aussehen kann. Nach der Eingabe eines Parameters (hier: *hallowelt'* ;beachte das Hochkomma), wird dieser Wert in eine SQL-Anweisung kopiert und an einem Datenbankserver gesendet. Da die SQL-Anweisung durch das Hochkomma eine falsche Syntax hat, führt der Aufruf zu einer Fehlermeldung, die dem Benutzer entsprechend angezeigt wird. Der Angreifer kann nun daraus die Schlussfolgerung ableiten, dass die Webanwendung prinzipiell verwundbar ist. Das allgemeine Vorgehen zum Finden einer Schwachstelle bezüglich zu SQL-Injection ist in Abbildung 2.3 in Form eines Sequenzdiagramms noch einmal dargestellt.

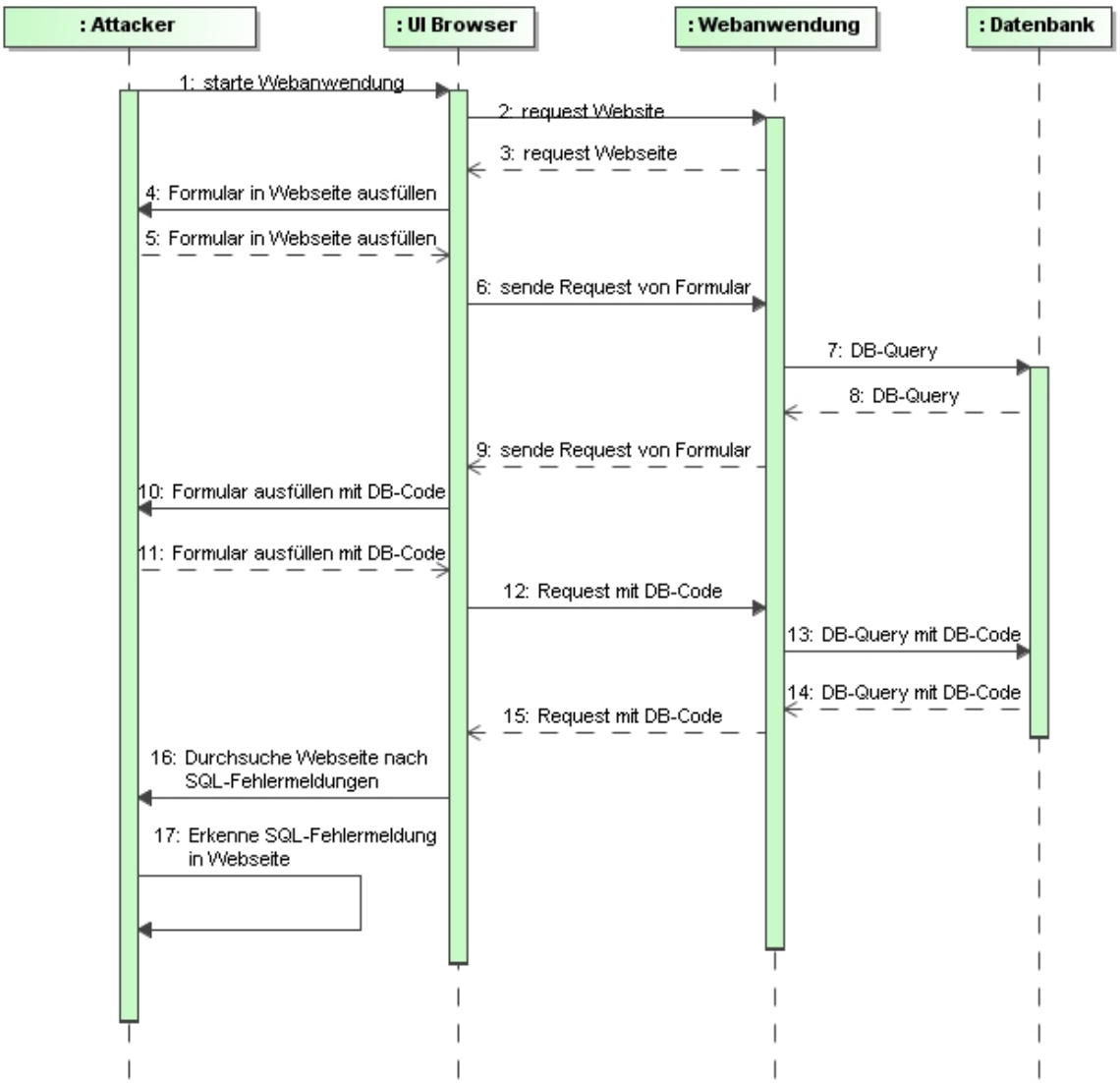


Abbildung 2.3.: Sequenzdiagramm bei einer SQL Injection

Cross-Site Scripting

Unter Cross-Site Scripting (XSS) versteht die Wissenschaft das Einschleusen von Javascript oder HTML in eine Webanwendung. Laut der Studie [Con07] aus dem Jahr 2007, liegt XSS mit 39% an erster Stelle aller gefundenen Schwachstellen in Webapplikationen.
Das Einschleusen funktioniert hierbei, indem ein Parameter, den die Webanwendung entgegennimmt, entsprechend mit Scriptcode manipuliert wird. Nach einem erfolgreichen Einschleusen von XSS-Code, führt die Webanwendung diesen Code automatisch aus. Damit jedoch dieser Angriff erfolgreich funktioniert, muss auf einer Webseite der Anwendung der eingegebene Parameter im unveränderten Eingabeformat wieder erscheinen. Die Gefahr die hieraus entsteht, ist nicht nur das simple Aufrufen von HTML-Kommandos, sondern, dass der Angreifer darüber hinaus einen kompletten Zugriff auf den DOM einer Webseite erhält. So kann er mit Hilfe von Javascript sämtliche, auch personenbezogene, Daten eines Benutzers ausspähen und auch manipulieren.

Abbildung 2.4.: Beispiel einer verwundbaren Webanwendungen gegen XSS

Die Abbildung 2.4 zeigt einen solchen Angriff auf eine Webanwendung. Nach der Eingabe eines Parameters, wird dieser in einer Ergebnisseite angezeigt. Der Angreifer hat jedoch zusätzlich zu einem Eingabewert ein Cross-site-scripting ($< script > alert(document.cookie) < /script >$) verwendet, welches alle Cookies (von einer Domain) des Benutzers ausliest.
Das allgemeine Vorgehen zum Finden und Ausnutzen einer Cross-Site-scripting-Schwachstelle ist in Abbildung 2.5 in Form eines Sequenzdiagramms noch einmal dargestellt.

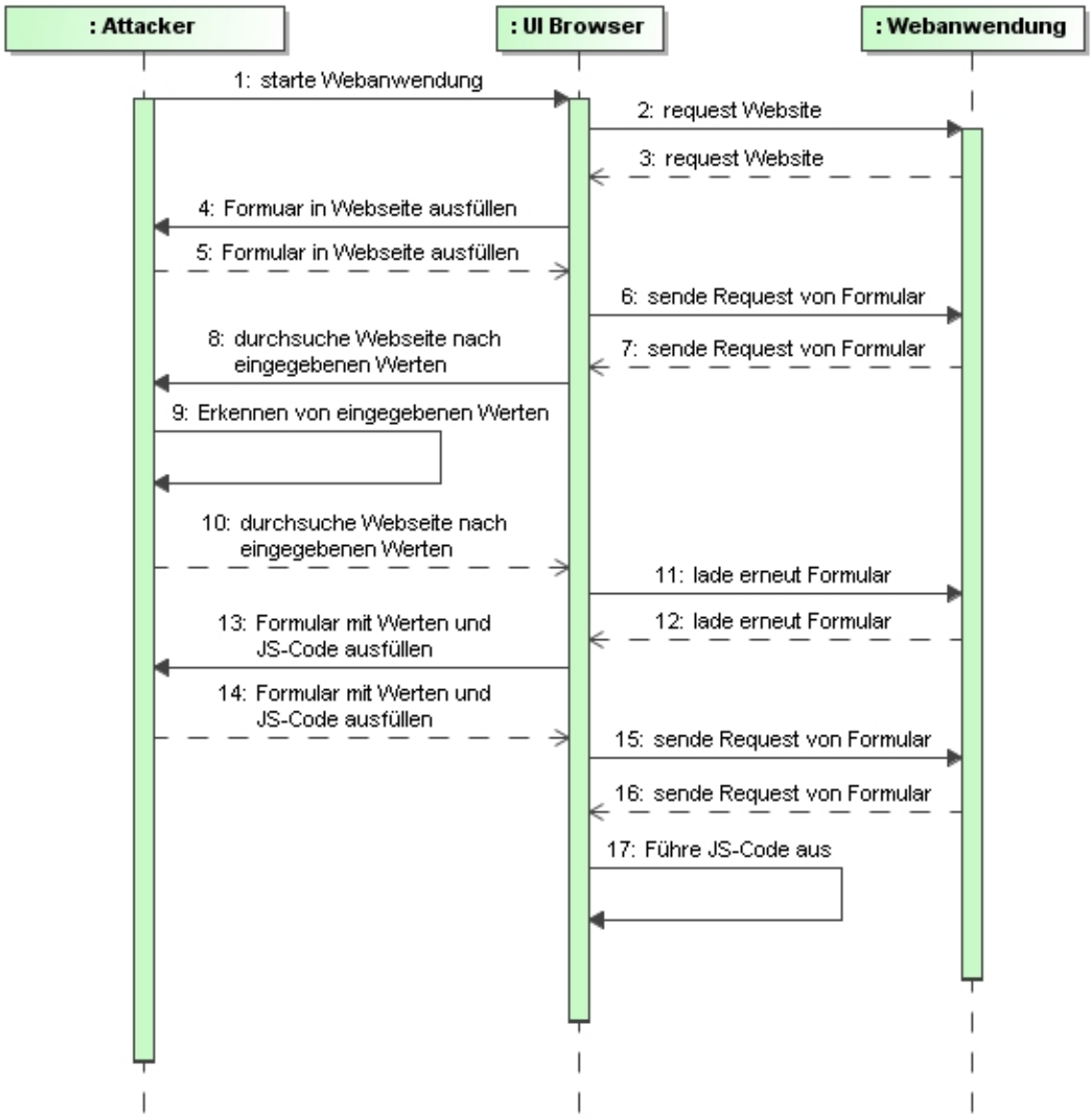


Abbildung 2.5.: Sequenzdiagramm bei Cross-site-scripting

Weitere Angriffsmöglichkeiten auf Webanwendungen

Zusätzlich zu den oben genannten Cross-site-scripting und SQL-Injection existieren noch viele weitere Techniken um einen Angriff auf Webanwendungen durchzuführen. So sind auch sehr häufig noch Sicherheitslücken zu finden, die im Bereich *Session hijacking* einzuordnen sind. Eine sehr gute Übersicht über die zur Zeit bekannten Schwachstellen in Webanwendungen stellt die OWASP Community unter [OWA] bereit.

2.1.3. Web 2.0

Unter dem Begriff *Web 2.0* wird eine neue Art der Benutzung von Ressourcen im Internet beschrieben. Die Wikipedia[2] sagt dazu, dass es sich bei Web 2.0 mehr um ein Schlagwort als um eine neue Art der Technik handelt. Dieses Schlagwort umfasst hierbei die beiden Begriffe **interaktiv** und **kollaborativ**. Interaktiv bedeutet in diesem Zusammenhang, dass der Benutzer nicht mehr die Requests selber an die Webappliaktion senden muss, sondern dass die Webanwendung eigenständig in der Lage ist automatisch ihren Zustand zu verändern und dieses dem Benutzer mitzuteilen. Kollaborativ bedeutet hingegen, dass eine Vielzahl von Personen an dieselben Dokumente gleichzeitig arbeiten können.
Entscheidend ist jedoch, dass es sich bei Web 2.0 nicht um eine neue Art der Kommunikation im Internet handelt, sondern lediglich als eine Weiterentwicklung zu sehen ist, die das Internet in seiner Zeit durchlaufen hat. So haben sich die bekannten Techniken wie zum Beispiel Javascript und HTML (Hypertext Markup Language) in ihrem Funktionsumfang natürlich mit der Zeit auch weiterentwickelt, so dass nun die Kombination dieser Techniken in der Lage ist, Webanwendungen dynamischer zu gestalten. Diese Kombination der einzelnen Techniken wird auch als *Ajax* (Asynchronous JavaScript and XML) bezeichnet.
Die folgenden beiden Abbildungen 2.6 und 2.7 stellen die Unterschiede in der Kommunikation zwischen einem Benutzer und einer Webanwendung noch einmal dar.

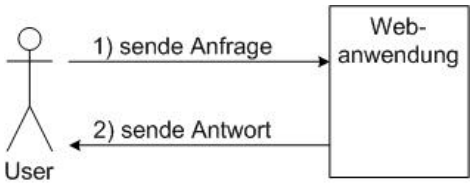


Abbildung 2.6.: Klassische Kommunikation zwischen Benutzer und Webanwendung

Die Abbildung 2.6 zeigt das klassische Vorgehen der Kommunikation im WWW. Um eine Zustandsänderung in einer Webapplikation herbeizuführen, muss der Benutzer selbst eine Anfrage in Form eines HTTP Requests zum Server senden (Schritt 1). Nachdem der Server die Anfrage vom Client entgegen genommen hat, sendet er daraufhin dem Benutzer die entsprechende Antwort zurück (Schritt 2). Die Webanwendung reagiert also auf die Anfragen, welche der Benutzer explizit versendet hat.

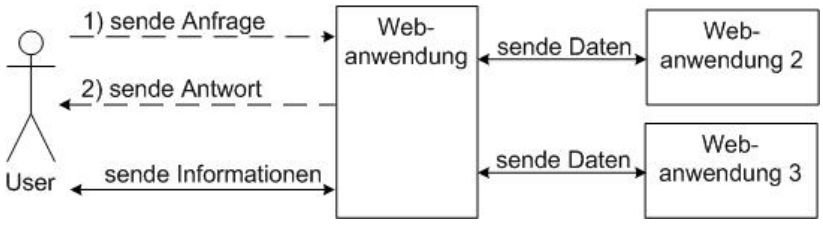


Abbildung 2.7.: Kommunikation mit Webanwendung bei Web 2.0

Im Web 2.0 hingegen muss für eine Zustandsänderung der Webseite ein Benutzer nicht mehr explizit selber eine Anfrage an die Webanwendung senden. Das automatische Senden

[2]siehe `http://de.wikipedia.org/wiki/Web_2.0`

von Anfragen vom Client an den Server kann hierbei auch von der Programmlogik selber ausgeführt werden. Dieses hat zum Ergebnis, dass dem Benutzer suggeriert wird, dass die Webanwendung ein *Eigenleben* entwickelt. In der Abbildung 2.7 tauscht der Benutzer, wie in der klassischen Kommunikation, mit dem Server Daten aus. Des Weiteren kommuniziert die Programmlogik der Webanwendung nicht nur automatisch mit dem Client und dem Server, sondern ebenfalls mit anderen Servern, die bestimmte Dienste bereitstellen können.

2.1.4. Document Object Model

Das Document Object Model oder kurz DOM ist eine Spezifikation zum Zugriff auf HTML-Dokumente welches vom W3C[3] entworfen wurde (siehe[W3Cc]). Es dient der einheitlichen Darstellung von HTML-Dokumenten bei HTTP und beschreibt den Aufbau von HTML-Dokumenten. Durch die Einführung immer neuer Anforderungen zur Präsentation von Inhalten im Internet, wie zum Beispiel durch Web 2.0, wurden mit der Zeit die Spezifikationen zum DOM überarbeitet, so dass zum aktuellen Zeitpunkt die Version DOM-Level 3 existiert.

DOM einer HTML-Seite

```
<html>
  <body>
    <h1>Ueberschrift</h1>
  </body>
</html>
```

Das obere Beispiel zeigt den Aufbau einer Webseite. Der DOM ergibt sich hierbei aus den einzelnen Elementen (Tags) der Webseite, welche jeweils als bestimmte Kommandos bei HTML angesehen werden können.

2.1.5. Vererbung in Javascript

Die Vererbung von Objekten ist heutzutage ein wichtiger Bestandteil von Programmiersprachen. So unterstützt auch Javascript Möglichkeiten zur Vererbung. Im Gegensatz zu vielen anderen objektorientierten Programmiersprachen die eine klassenbasierte Vererbung einsetzen, verwendet Javascript hingegen den *Prototyp-basierten* Ansatz. Er bietet den Vorteil, dass Objekte während der Laufzeit strukturell veränderbar sind. Das bewirkt, dass Programmierer von Webanwendungen die Struktur von Objekten zu eden beliebigen Zeitpunkt ändern können. Einen ausführlichen Vergleich der beiden Ansätze zwischen Klassenbasierter Vererbung und Prototyp-Basierter Vererbung, ist unter [Moza] beschrieben.

Prototyp-Basierte Vererbung bei Javascript

```
function myObject() {
  this.valA = 0;
}

myObject.prototype.valB = 1;
myObject.prototype.getValA = function(){
    return this.valA;
  }
```

[3] Consortium zur Standardisierung von WWW-Techniken. `http://www.w3.org/`

```
}

var o1 = new myObject();
o1.valA;     // gibt den Wert 0 aus
o1.valB      // gibt den Wert 1 aus
o1.getValA   // gibt den Wert 0 aus
```

Das obere Beispiel zeigt die Verwendung von Javascript Prototype. Im ersten Schritt wird eine Klasse *myObject* erzeugt, welche als Attribut die Variable *valA* besitzt (Zeile 1 bis 3). Nach dem Definieren der Klasse *myObject* wird diese Klasse mit Hilfe von Prototype entsprechend *von außen* erweitert. Hierbei wird die neue Variable *valB* und die neue Funktion *getValA* der Klasse *myObject* hinzugefügt (Zeile 5 bis 9). Nach dem anschließenden Erstellen des Objekts *o1* der Klasse *myObject* beinhaltet das Objekt *o1* die Variablen *valA, valB* und die Funktion *getValA*. Mit dieser Technik ist es also möglich, zur Laufzeit definierte Klassen und Objekte beliebig in ihrem Funktionsumfang abzuändern oder zu erweitern.

2.1.6. Sicherheitsrisiken bei Ajax Technologien

Diese neue Form der Kommunikation stellt für das Testen und Analysieren von Webapplikationen eine neue Herausforderung dar. So beschreibt der Artikel *Ajax Security Basics* auf [JSHJK06] mehrere neue Sicherheitsrisiken die im Zusammenhang mit Web 2.0 stehen.

Client-seitige Kontrolle der Sicherheitsmaßnahmen
Bei dieser Methode betten die Entwickler große Teile der Programmlogik client-seitig direkt in die Webanwendung ein. Hierbei kann das Einbetten von Programmcode auch sicherheitsrelevante Teile beinhalten. Der Vorteil bei dieser Möglichkeit ist, dass Anfragen zum Server verringert werden und nur noch *wichtige* Anfragen zur Webanwendung gesendet werden. Dieses führt jedoch unter Umständen dazu, dass Entwickler sensible Informationen für jeden Benutzer frei zur Verfügung stellen, da der client-seitige Quellcode von jedem Benutzer eingesehen werden kann.

Zunahme von Angriffsmöglichkeiten
Durch das Verwenden von zusätzlichen Diensten für Webanwendung wie zum Beispiel *Google earth* wird die Anwendung komplexer, weshalb auch die Anzahl von Sicherheitslücken in einer Webanwendung steigt. Hinzu kommt, dass durch die Verwendung zusätzlicher Dienste die Anzahl der offenen Ports auf einen Webserver steigen, welche prinzipiell als Angriffspunkte angesehen werden können.

Neue Möglichkeiten für Cross-Site-Scripting (XSS)
Des Weiteren ergeben sich im Bereich von Cross-Site-Scripting ganz neue Möglichkeiten. Mit Hilfe von asynchroner Übertragung von Requests, welche im Hintergrund ablaufen, können Angreifer Nachrichten austauschen, ohne dass der Benutzer einer Webanwendung die Datenübertragung mitbekommt.

2.1.7. Das Web Application Attack and Audit Framework

Das Web Application Attack and Audit Framework (w3af) ist ein Open Source Projekt und ist zum jetzigen Zeitpunkt in Version 1.0-rc2 unter [Riac] verfügbar. Es stellt die Kernkomponente dar, auf dem das in der Diplomarbeit entwickelte Projekt aufbaut. Da es in Python

entwickelt wurde, ist es plattformunabhängig und läuft somit unter Windows und Unix. Des Weiteren beinhaltet das w3af einen MITM-Proxy (siehe 2.1.8) der zum Manipulieren von HTTP-Requests und HTTP-Responses verwendet wird.
Die Kernaufgaben des Frameworks sind das Finden von Sicherheitslücken und teilweise auch das Ausnutzen eventueller gefundener Schwachstellen in Webanwendungen. Da das Erkennen von Schwachstellen durch eine Plugin-Architektur realisiert ist, können außerdem leicht neue Angriffstechniken auf Webanwendungen in das Framework integriert werden. Einen sehr guten Einstieg in das w3af liefert die Dokumentation unter [Riab]. Die Abbildung 2.8 zeigt die Oberfläche des w3af-Frameworks.

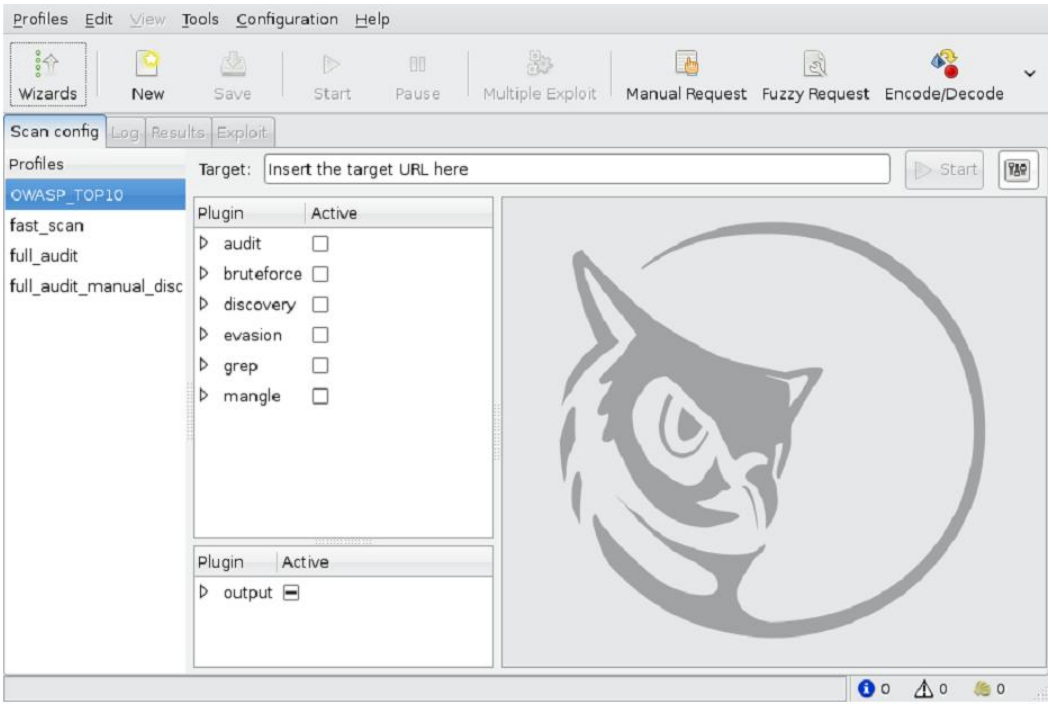


Abbildung 2.8.: GUI des w3af [Riac]

Plugin-Architektur bei w3af

Wie bereits erwähnt, können neue Angriffsszenarien durch das Hinzufügen von Plugins in das Framework integriert werden. Hierzu bietet das w3af grundsätzlich verschiedene Kategorien für die Ausführung der Plugins an, von denen die wichtigsten Kategorien *discovery, audit und attack* sind.

Discovery Plugins

Discovery Plugins haben die Aufgabe, neue URLs und Eingabe-Elemente in Webseiten zu finden. Dazu durchsuchen diese Discovery-Plugins eine Webseite und sammeln alle möglichen Links und HTML-Formulare. Die so gesammelten Daten, dienen später als Eingabeparameter für die Audit-Plugins.

Audit Plugins

Diese Plugins dienen dem Finden von Sicherheitslücken in Webanwendungen. Sie nehmen als

Eingabeparameter eine URL, welche von Discovery-Plugins gefunden wurde. Je nach Art des Angriffs, senden die Audit-Plugins spezielle Daten an die übergebene URL. Bei einem erfolgreichen Erkennen einer Schwachstelle, gibt das Audit-Plugin diese entsprechende URL als Ergebnis zurück und markiert diese als unsicher gegenüber einer bestimmten Schwachstelle.

Attack Plugins
Attack-Plugins dienen dem Ausnutzen der zuvor gefundenen Schwachstelle durch die Audit-Plugins. Ein Beispiel wäre die Herstellung einer Verbindung zu dem betroffenen Server. Aber auch andere Möglichkeiten wie DoS sind hier denkbar.

Andere Plugins
Des Weiteren gibt es im w3af noch fünf Kategorien. Zum Beispiel können durch *Output-Plugins* ein Benutzer verschiedene Ausgabeformate für eine Zusammenfassung der Testergebnisse bestimmen. Eine weitere Kategorie wären die *mangle-Plugins*. Mit Hilfe dieser Plugins kann der Benutzer einen stärkeren Einfluss auf die Eingabeparameter in HTML-Formularen nehmen. Die weiteren Kategorien mit ihrer jeweiligen Bedeutung sind in der Anleitung zur Verwendung vom w3af unter [Riab] beschrieben.

Die Beziehungen zwischen den einzelnen Kategorien der Plugins, ist in Abbildung 2.9 dargestellt. Diese zeigt auch den Ablauf der Plugin-Architektur im w3af. Im ersten Schritt werden alle Discovery-Plugins abgearbeitet. Auf die Discovery-Plugins haben auch die *Bruteforce-Plugins* einen Einfluss. Die so gesammelten Daten der Discovery-Plugins werden im Anschluss an die Audit-Plugins übergeben und im nächsten Schritt an die Attack-Plugins weitergeleitet. Hierbei können in jeder Phase Informationen an den Output-Plugins übergeben werden, welche den Benutzer über den aktuellen Status informiert.

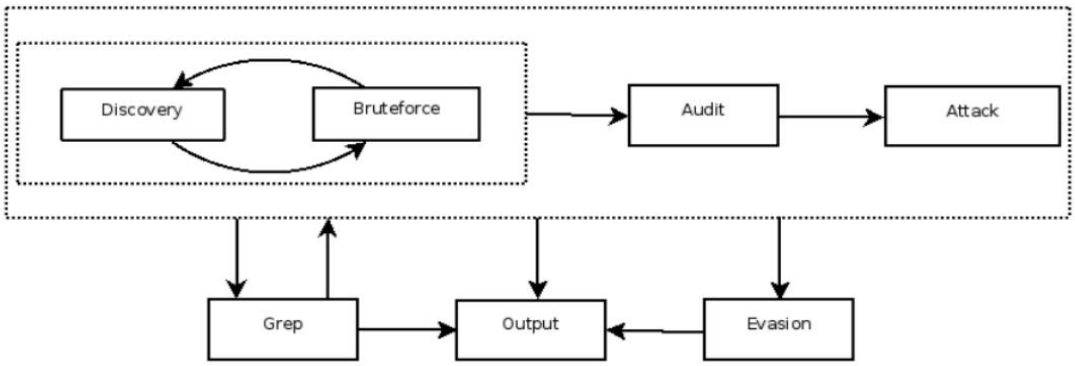


Abbildung 2.9.: Informationsfluss zwischen den Plugins [Riaa]

2.1.8. Man-in-the-middle-Proxy

Bei einem Man-in-the-middle-Proxy (MITM Proxy) handelt es sich um einen Proxy, der zwischen zwei Kommunikationspartnern entweder physikalisch oder logisch zwischengeschaltet ist. Das Ziel ist es hierbei, den gesamten Datenverkehr zwischen den beiden Kommunikationspartnern über den MITM-Proxy zu leiten. Der Besitzer dieses Proxys kann in diesem Fall den

gesamten Datenverkehr mithören und auch manipulieren. Um dieses auch für verschlüsselte Verbindungen zu gewährleisten, baut der MITM-Proxy zusätzlich eigene Verbindungen zu den Kommunikationspartnern auf, über welche die verschlüsselten Daten transportiert werden.

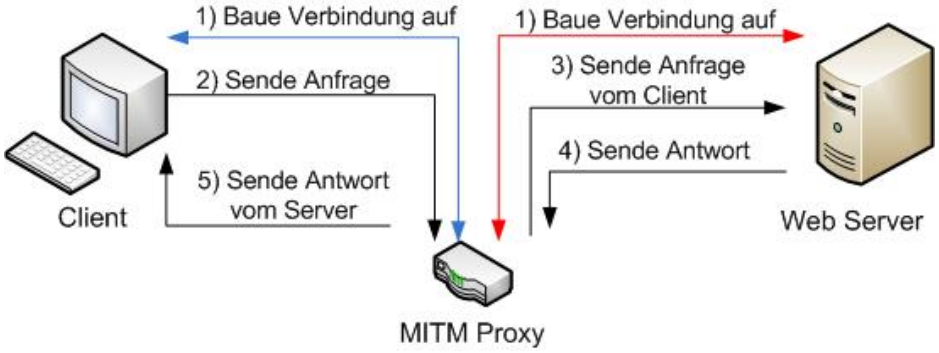


Abbildung 2.10.: Funktionsweise eines Man-in-the-middle-Proxys

Die Abbildung 2.10 zeigt, wie die Kommunikation zwischen einem Client und dem Webserver über ein MITM-Proxy verläuft. Der MITM-Proxy baut hierfür zunächst eine Verbindung zum Client und eine zweite Verbindung zum Webserver auf (Schritt 1). Auf diese Weise kann der MITM-Proxy ebenfalls auf verschlüsselte Daten zugreifen. In Schritt 2 sendet nun der Client Daten an den Webserver. Diese Übertragung wird jedoch vom MITM-Proxy abgefangen und kann an dieser Stelle eingesehen oder auch manipuliert werden. Im Anschluss wird dann die vom Proxy manipulierte Anfrage an den Webserver gesendet (Schritt 3), welcher die Anfrage entgegennimmt und eine entsprechende Antwort zum Proxy zurückliefert (Schritt 4). Auch diese Antwort kann nun der MITM-Proxy wieder einsehen und manipulieren. Im letzten Schritt sendet er die manipulierte Antwort an den Client zurück.

2.2. Aktueller Stand der Technik

Das automatisierte Testen von Webanwendungen auf Schwachstellen stellt für die heutige Wissenschaft eine große Herausforderung dar. So ist es auch nicht sonderlich, dass für diese Aufgabe eine Vielzahl von Programmen und Publikationen existieren. Einen guten Überblick bietet die Arbeit [Kil09], welche mehrere auf dem Markt befindliche Programme auf ihre Stärken und Schwächen hin vergleicht. Im Weiteren werden sämtliche Programme und Publikationen grob in zwei Klassen unterteilt.

- Automatisiertes Testen von Webanwendungen
- Manuelles Testen von Webanwendungen

Die Herausforderung des automatisierten Testens steht im Vordergrund vieler Publikationen und Programme. Hierbei reichen die Ansätze vom automatisierten Testen mit Hilfe einer Spezifikationsdatei (siehe [MBX] und [JL]) über modell-basiertes Testen (siehe [SLN04] und [XSP09]) bis hin zum Aufzeichnen von Browser-Sitzungen (vergleiche [SMSP] und [Gro]). Diese Ansätze sind entweder als Client oder als Browser-Erweiterungen implementiert. Die

Schwierigkeit hierbei besteht allerdings, dass Webseiten dynamische Inhalte und Javascript haben können. Dieses führt dazu, dass sich die Struktur des DOMs beliebig ändern kann, was das Testen von Webanwendungen bei dieser Variante extrem erschwert. Daraus ergibt sich, dass die client-seitigen Ansätze hierbei schlechte Ergebnisse im Bezug auf gefundene Schwachstellen liefern, da sie normalerweise die Logik von Webseiten nicht auswerten (besitzen keinen Javascript-Interpreter). Die Browser-Erweiterungen hingegen können durch den im Browser eingebauten Javascript-Interpreter die Logik der Webseiten verstehen, bieten jedoch dem Tester wenig Möglichkeiten eine Schwachstelle in eine Webanwendung genauer zu spezifizieren. Zudem sind diese Browser-Erweiterungen abhängig vom Webbrowser, so dass nicht alle Arten von Webbrowsern die Funktionalitäten der Erweiterungen nutzen können.

Vorteile	**Nachteile**
zuvor definierte Testszenarios liefern eine gute Qualität in Bezug auf Fehlererkennung	Benutzer kann bei einem laufenden Test nicht eingreifen
das verwendete Testszenario kann jederzeit automatisiert wiederholt werden	hoher Aufwand bei der Berücksichtigung von dynamischen Webseiten und Web 2.0 Anwendungen
Zusammenfassung am Ende des Tests	Testvorbereitungen müssen durchgeführt werden

Tabelle 2.1.: Vor- und Nachteile beim automatisierten Testen

Die zweite, oben genannte, Klasse beschreibt das manuelle Testen von Webanwendungen. In dieser Kategorie finden sich viele Programme, die als MITM-Proxy zum Einsatz kommen. Aber auch Webbrowser-Erweiterungen wie zum Beispiel TamperData[4] sind hier anzufinden, die ähnlich zu einem MITM-Proxy funktionieren.
Einen Vergleich zu den einzelnen Stärken der gängigen Programme zeigt [Kil09] auf. Die Funktionsweise solcher Programme basiert auf das Auslösen eines Alarms beim Absetzen eines Requests aus der Webanwendung. In der Regel bieten solche Programme zusätzlich die Möglichkeit der Manipulation des abgesetzten Requests an (siehe MITM-Proxy 2.1.8). Durch das Erkennen und Abfangen aller Requests, sind diese Programme von Haus aus in der der Lage mit Ajax-Techniken umzugehen. Leider weisen auch diese Programme Schwachstellen bei der Analyse von Webseiten auf.

Vorteile	**Nachteile**
Erkennen aller üblichen, von der Webanwendung abgesetzten, Requests	Benutzer muss Test manuell durchführen
Manipulation der Requests und Responses	Keine automatische Testauswertung der Webanwendung
	Testfälle können nicht gespeichert werden

Tabelle 2.2.: Vor- und Nachteile beim Testen mittels eines MITM-Proxys

[4] `http://tamperdata.mozdev.org`

Die Programme und Publikationen beider Klassen zeigen verschiedene Möglichkeiten zum Finden von Schwachstellen in Webanwendungen auf. Im Rahmen dieser Diplomarbeit soll ein neuer Ansatz zum Testen aufgezeigt werden. Er geht von der Annahme aus, dass ein vollständiges, automatisches Testen nicht möglich ist, dass aber Teile beim Testen einer Webanwendung automatisiert ausgeführt werden können. Diese Semi-Automatisierung hat zum Ziel, die Vorteile beider, oben genannten, Klassen zu vereinigen. Jedoch lassen sich auch hier zur Zeit nicht alle Nachteile der oben genannten Klassen vermeiden. So muss zum Beispiel der Tester zunächst die Webseiten definieren (Workflow), welche zur Durchführung einer semi-automatischen Analyse benötigt werden. Trotz dieser Einschränkung überwiegen jedoch die Vorteile, die diese Diplomarbeit rechtfertigen, so dass unter anderem die folgenden Vorteile zu nennen sind.

- Benutzer kann jederzeit in einen laufenden Test eingreifen
- Möglichkeit einer automatischen Testauswertung am Ende eines Analyse-Vorgangs
- Unterstützung des Testers bei der genaueren Analyse einer eventuell gefundenen Schwachstelle
- Unterstützung aktueller Web 2.0 Technologien

3. Entwurf

Um eine teilweise Automatisierung des Testens auf Sicherheitslücken zu erreichen, wird für diese Aufgabe eine neue Architektur in Form eines Prototypen entworfen. Dieser Prototyp baut grundsätzlich auf dem w3af Framework auf. Dazu wird eine Erweiterung für das w3af Framework entwickelt, die in der Lage ist, abgesetzte Requests zu manipulieren und über spezielle Ausdrücke (pattern machting) Gemeinsamkeiten zwischen mehreren Requests und einem speziell erhobenen Merkmal darzustellen. Hierfür wird zunächst der Aufbau der neuen Architektur in abstrakter Form dargestellt.

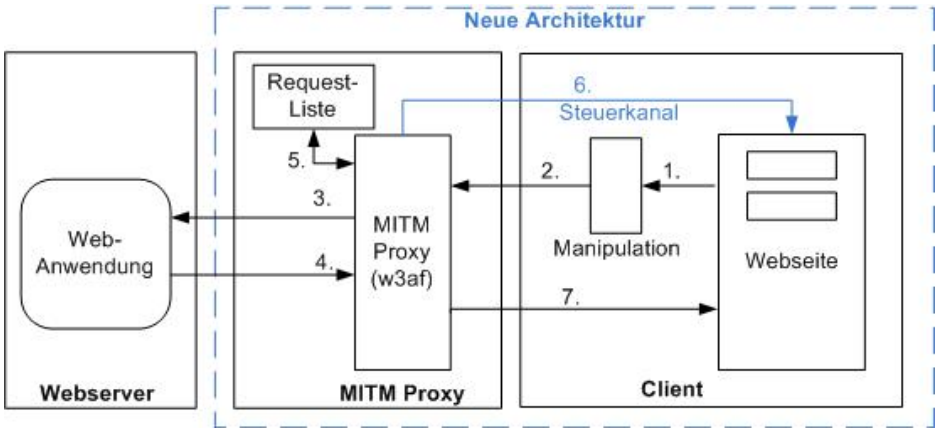


Abbildung 3.1.: Abstrakte Darstellung der neuen Architektur

Die Abbildung 3.1 zeigt eine abstrakte Darstellung der neuen Architektur. Der Benutzer sendet zunächst einen Request von der Webseite über den MITM-Proxy an die Webanwendung. Bevor der Request jedoch den MITM-Proxy erreicht, kann dieser zusätzlich noch einmal eingesehen und manipuliert werden (Schritt 1). Dadurch kann der Tester etwaige Änderungen des Requests (zum Beispiel durch client-seitige Programmlogik) einsehen und auch manipulieren. In Schritt 2 und 3 wird nun der Request über den MITM-Proxy zum Webserver weitergeleitet. Die daraus resultierende Antwort wird in Schritt 4 an den MITM-Proxy zurückgesendet. Dieser speichert zunächst den Request mit einem speziellen Merkmal (Token) in eine zuvor definierte Liste (Schritt 5). Ein derartiges Merkmal kann zum Beispiel der Response des aktuellen Requests sein. Im nächsten Schritt werden eventuelle Konstrukte (Gemeinsamkeiten) zwischen den einzelnen in der Request-Liste befindlichen Einträge gesucht. Hierbei werden spezielle Metriken verwendet, die mit Hilfe von *pattern matching*-Verfahren die Gemeinsamkeiten zwischen den einzelnen Requests und ihren jeweiligen Merkmalen darstellen. Bei einem Auftreten einer Gemeinsamkeit zwischen den Requests und ihren Merkmalen wird im nächsten Schritt über den Steuerkanal ein Alarm an den Browser des Benutzers (Schritt 6) gesendet, welcher den Benutzer über eine gefundene Schwachstelle informiert. Im letzten Schritt 7 wird der Response zum Client weitergeleitet, der diesen dann in seinem Browser entsprechend darstellt.

Im weiteren Verlauf werden zu der, in Abbildung 3.1, entwickelnden Architektur zunächst die Probleme analysiert, welche für die Umsetzung der beschriebenen Architektur entscheidend sind. Des Weiteren wird jeweils eine entsprechende Lösung aufgezeigt, die dann, am Ende des Kapitels in Form eines Prototyps zusammengefasst sind.

3.1. Analyse des DOMs

Damit Benutzer mit einer Webanwendung kommunizieren können, besitzen Webanwendungen verschiedene Objekte, über die der Benutzer Daten an die Applikation übergeben kann. Solche Objekte können zum Beispiel Links, Eingabefelder oder auch Buttons sein. Die Schwierigkeit, die bei der Analyse des DOMs auftritt, ist das Finden und Extrahieren aller in Frage kommenden Objekte. Dieses stellt insofern ein Problem dar, da durch den Einsatz von Javascript viele Webanwendungen ein dynamisches Verhalten entwickeln und sich so die Anzahl der Objekte zur Laufzeit ändern kann. Ein weiteres Problem ist außerdem, dass einzelne Webseiten nicht konform zum standardmäßigen Aufbau von HTML-Dokumenten sind. So müssen die Programme zum Analysieren einer Webseite ebenfalls fehlertolerante Extraktionsalgorithmen besitzen um nicht konforme Webseiten parsen zu können.

3.1.1. Möglichkeiten zur Analyse des DOMs in Webseiten

Um eine Webseite auf bestimmte Objekte zu analysieren, wird hierfür der Quelltext der Webseite durchsucht. Für das Extrahieren der Objekte einer Webseite existieren auf dem Markt für die Programmiersprache Python viele sogenannte *HTML-Parser*. Hierbei untersucht der Artikel [Bic] die in Python gängigen *HTML-Parser* auf ihre Performance. Jedoch lassen sich keine Informationen finden, die die Qualität der einzelnen HTML-Parser bezüglich der gefundenen Objekte vergleicht. Darüber hinaus werden weitere Möglichkeiten zum Parsen einer Webseite betrachtet. So existieren Techniken, die es erlauben, Webseiten mit Hilfe von Javascript zu analysieren. Aber auch browserabhängige Ansätze ermöglichen das Extrahieren von Objekten in einer Webseite. Im weiteren Verlauf werden einzelne Testfälle generiert, welche die Qualität der Ergebnisse für ausgewählte Parser untersucht. Für die Durchführung der Tests werden hierbei die folgenden Parser verwendet.

Python Modul "Beautiful Soup"
Das Python-Modul *Beautiful Soup* stellt Mechanismen zum Parsen von HTML- und XML-Dokumenten zur Verfügung. Beautiful Soup zeichnet sich damit aus, dass es auch mit fehlerhaften, nicht konformen HTML-Dokumenten zurechtkommt. Für die Untersuchung wurde die Version 3.07a verwendet. Eine genaue Beschreibung des Moduls kann auf der Entwicklerseite [Com] eingesehen werden.

Javascript Parser
Eine weitere Möglichkeit ist den DOM einer Webseite über Javascript-Befehle zu analysieren. Hierzu muss jedoch der Javascript-Code für die Analyse in die Webseite eingebracht werden. Auch muss darauf geachtet werden, dass zunächst die Webseite vollständig geladen sein muss, da sonst nicht alle Objekte ermittelt werden können.

Externe Schnittstelle beim Internet Explorer
Der Microsoft Internet Explorer bietet die Möglichkeit über eine externe Schnittstelle den Browser komplett fernzusteuern. Die genaue Technik dazu nennt sich COM [1]. Da im Internet Explorer ein Javascript-Interpreter verwendet wird, kann auf diesen über die externe Schnittstelle zugegriffen werden. So kann von außerhalb des Internet Explorers eine Webseite auf ihre Objekte hin untersucht werden. Dieses Vorgehen ist jedoch vom Internet Explorer abhängig.

3.1.2. Auswertung der einzelnen Möglichkeiten zur Analyse des DOMs

Zum Vergleichen der Qualität der einzelnen Parser ist es hilfreich, Testfälle zu generieren, welche die jeweiligen Parser zu bewältigen haben. Für diese Aufgabe werden fünf Testszenarien entwickelt, die in ihrer Komplexität von Testfall zu Testfall ansteigen.

- Test 01 beschreibt eine Webseite die nur HTML-Code enthält. Die Webseite beinhaltet alle gängigen Eingabe-Elemente, sowie Verweise (Links) auf andere Webseiten, jedoch kein Javascript. Dieser Test stellt den einfachsten Fall einer Analyse durch ein Parser dar und sollte von jeden Parser vollständig gelöst werden.
- Test 02 enthält einfache Javascript-Anweisungen wie z.B. den Befehl "document.writeln". Der Test soll zeigen, ob die einzelnen Parser grundsätzlich Javascript-Konstrukte verarbeiten können.
- Test 03 beschreibt einzelne Javascript-Funktionen und logische Konstrukte wie zum Beispiel eine if-then-else-Anweisung. Dadurch sollen die Parser logische Abläufe in Webseiten erkennen.
- Test 04 simuliert eine Kodierung des Quelltextes. Dieser Test ist notwendig, da einige Webseiten ihren HTML Code *verschleiern*. Das Dekodieren des Quelltextes wird mit Hilfe des Javascript-Befehls "unescape"vorgenommen.
- Test 05 bindet zusätzlich externe Javascript-Dateien in die Webseite ein. In HTML wird dieses üblicherweise mit dem HTML-Element $< script\ language = "JavaScript" src = "testdatei.js" >< /script >$ erreicht.

Die Ergebnisse der einzelnen Tests sind in der Tabelle 3.1 zusammengefasst. Der Python-Parser Beautiful Soup liefert insgesamt die schlechtesten Ergebnisse bei den einzelnen Tests. Sobald eine Webseite logische Konstrukte in Form von Javascript enthält, stolpert der Parser Beautiful Soup. Auch mit der kodierten Webseite bei Test 04 stieg Beautiful Soup komplett aus. Dieses ist nicht ganz überraschend, da Beautiful Soup keinen Javascript-Interpreter beinhaltet, der den Javascript-Code im Vorhinein ausführt.
Die beiden anderen Möglichkeiten, zum Parsen der Objekte einer Webseite, zeigen durch die Verwendung eines Javascript-Interpreters sehr gute Ergebnisse. Sie fanden alle Objekte und interpretierten selbst die kodierte Webseite bei Test 04 korrekt. Da der Internet Explorer eine Benutzung der neuen Architektur bezüglich der Verwendung der Plattform sehr einschränkt, wird für die DOM-Analyse im weiteren Verlauf die Auswertung mittels eines Javascript-Interpreters durchgeführt.

[1] COM (Component Object Model) bietet die Möglichkeit bestimmte Funktionen von externen Programmen zu benutzen. `http://de.wikipedia.org/wiki/Component_Object_Model`

Schwierigkeitsgrad	Erkennungsrate		
	Beautiful Soup	**Schnittstelle IE**	**Javascript Parser**
Kein Javascript (Test 01)	Alle Objekte erkannt	Alle Objekte erkannt	Alle Objekte erkannt
Verwendung der Javascript-Funktion document.writeln (Test 02)	Alle Input-Elemente erkannt. Javascript-Objekte nicht erkannt	Alle Objekte erkannt	Alle Objekte erkannt
Funktionen und logische Checks in Javascript (Test 03)	Keine Objekte erkannt	Alle Objekte erkannt	Alle Objekte erkannt
Kodierte Webseite mit der Javascript-Funktion 'unescape' (Test 04)	Keine Objekte erkannt	Alle Objekte erkannt	Alle Objekte erkannt
Objekte in externen Javascript-Dateien (Test 05)	Keine Objekte erkannt	Alle Objekte erkannt	Alle Objekte erkannt

Tabelle 3.1.: Vergleich der Möglichkeiten zum Parsen von HTML Dokumenten

3.2. Abfangen von Requests

Viele Webanwendungen lagern heutzutage Teile der Programmlogik auf die Client-Seite aus. So ist es heutzutage üblich, Eingaben vom Benutzer im Voraus auf ungültige Zeichen zu untersuchen, entfernen und erst dann an die Webanwendung weiterzuleiten. Dieses *Filtern von Zeichen* passiert im Hintergrund der Webanwendung und ist in der Regel für den Benutzer nicht sichtbar. Um jedoch trotzdem zu erkennen, wie eine Anfrage nach eventuellen Änderungen der client-seitigen Programmlogik aussieht, kommt normalerweise ein MITM-Proxy zum Einsatz. Dieser ist in der Lage, alle Anfragen zum Webserver auf Netzwerkebene anzuzeigen und auch zu manipulieren.
Jedoch weist auch der Einsatz eines MITM-Proxys Schwächen beim Abfangen von Requests auf. Grundsätzlich kann ein MITM-Proxy alle Requests abfangen. Da durch den Einsatz von Ajax-Techniken (zum Beispiel mittels eines XMLHttpRequests) die Webanwendung auf der Client-Seite eigenständig Requests absetzen kann, treffen beim MITM-Proxy nicht nur die Requests vom Benutzer, sondern auch die automatisch abgesetzten Requests ein. Bei vielen eingehenden Requests kann hierbei die Zuordnung zu den Requests von den Benutzern und den automatisch generierten Requests verloren gehen.

Aus dem oben genannten Grund wird ein weiterer Ansatz zum Abfangen von Requests aufgezeigt. Diese Methode arbeitet komplett auf der Client-Seite, weshalb der Einsatz eines MITM-Proxys nicht mehr notwendig ist. Durch die Verlagerung des Abfangens von Requests auf die Anwendungsebene im Browser, existieren viele neue Möglichkeiten für spätere Erweiterungen. So können zum Beispiel durch den Einbau eines Javascript-Debuggers die Interaktionen zwischen dem Benutzer und der Webanwendung besser untersucht werden.
Ein weiterer Vorteil dieser Methode ist die Zuordnung zwischen den einzelnen Requests. Da alle Requests (Benutzer-Requests und automatische Requests) in der Webanwendung direkt

abgefangen werden, kann der Benutzer alle ausgelösten Requests sofort einsehen, was die Kausalität der Requests stark verbessert.
Die Tabelle 3.2 stellt in einer Zusammenfassung beide Möglichkeiten, zum Abfangen eines Requests, noch einmal gegenüber. Für die spätere Implementierung der neuen Architektur wird das browser-seitige Abfangen von Requests verwendet. Obwohl die Komplexität gegenüber eines MITM-Proxys bei dieser Variante höher ist, überwiegen die Möglichkeiten der Erweiterbarkeit und einer besseren Zuordnung der einzelnen Requests.

	MITM-Proxy	**Browser-Seitig**
Lage im Netzwerk	Proxy	Client
Abstraktionsniveau	Netzwerkebene	Anwendungsebene
Komplexität	Hauptfunktionalität eines MITM-Proxys	hoch, da für jede Routine ein eigener Mechanismus geschrieben werden muss
Kausalität von Requests in Web 2.0	schlecht bei vielen Anfragen	sehr gut
Erweiterbarkeit	gering	sehr gut

Tabelle 3.2.: Vergleich der Möglichkeiten zum Abfangen von Requests

3.2.1. Browser-seitges Abfangen von Requests

Wie im Kapitel 2.1.1 (Techniken zum Senden eines Requests bei HTTP) gezeigt, existieren bei HTML und Javascript mehrere Möglichkeiten zum Absenden eines Requests. Im Weiteren wird anhand der Javascript-Funktion *Submit()* gezeigt, dass prinzipiell jede Methode, die zum Senden eines Requests aus einer Webanwendung dient, so abgeändert werden kann, dass der Benutzer den ausgehenden Request einsehen und auch manipulieren kann. Dieser Nachweis basiert auf der Verwendung von *Javascript prototype*, welche in Kapitel 2.1.5 beschrieben wurde.

Überschreiben der Submit-Methode in Javascript

```
var newSubmit=function(){
  alert("Submit discovered");
  this.oldSubmit();
}

HTMLFormElement.prototype.oldSubmit=HTMLFormElement.prototype.submit;
HTMLFormElement.prototype.submit=newSubmit;
```

Das obige Beispiel zeigt, wie die originale Submit-Methode in Javascript überschrieben werden kann. Dazu wird zunächst eine neue Funktion *newSubmit* erzeugt, welche die neue Funktionalität beinhaltet. Hierbei ist es auch möglich, die originale Submit-Methode (hier: oldSubmit) zusätzlich kontrolliert aufzurufen. Um nun die Programmlogik der originalen Submit-Funktion zu erhalten, wird im nächsten Schritt mit Hilfe von *Javascript prototype* zunächst die originale Submit-Funktion in eine neue Funktion (hier: oldSubmit) gespeichert. Im letzten Schritt kann nun die am Anfang erstellte Funktion *newSubmit* mit der originalen Submit-Funktion überschrieben werden.

3.3. Erkennen von Schwachstellen

Das voll automatisierte Testen von Webanwendungen stellt für viele Programme eine große Hürde dar. So ist es auch nicht verwunderlich, dass die gefundenen Schwachstellen zwischen den einzelnen Programmen zum Teil erheblich variieren. Jedoch verwenden viele Programme ein *Standardvorgehen* zum Finden von bestimmten Sicherheitslücken in Webanwendungen. Bei dieser Methode wird jeweils nur der entsprechende Response eines Requests auf bestimmte Merkmale, die für eine spezielle Schwachstelle typisch sind, untersucht. Solche Merkmale können zum Beispiel Fehlermeldungen von einem SQL-Server sein (siehe Abbildung 2.2). Beim Auftreten dieser Merkmale wird im weiteren Verlauf dieser Request als unsicher gekennzeichnet. Oft ist es jedoch so, dass eine Schwachstelle bei einem Request nicht direkt im entsprechenden Response zum Vorschein kommt. Solche Schwachstellen, die sich über mehrere Webseiten einer Webanwendung bewegen können (Workflow), sind beim Standardverfahren nicht zu finden. Nachfolgend sind die beiden eben beschriebenen Fälle noch einmal aufgelistet.

- Die Schwachstelle eines Requests befindet sich in seinem dazugehörigen Response.
- Die Schwachstelle eines Requests befindet sich in einem anderen Response eines Requests.

Wie bereits erläutert, existieren für den ersten Fall viele Lösungen, die im Response des dazugehörigen Requests die Schwachstellen erkennen. Im weiteren Verlauf wird eine Möglichkeit beschrieben, wie der zweite Fall, in dem sich die Schwachstelle nicht im entsprechenden Response des Requests befindet, gelöst werden kann. Die Abbildung 3.2 zeigt ein Beispiel, wie eine Schwachstelle über mehrere Webseiten aussehen kann. Dieses Vorgehen ähnelt einer Registrierung im Internet für einen beliebigen Dienst.

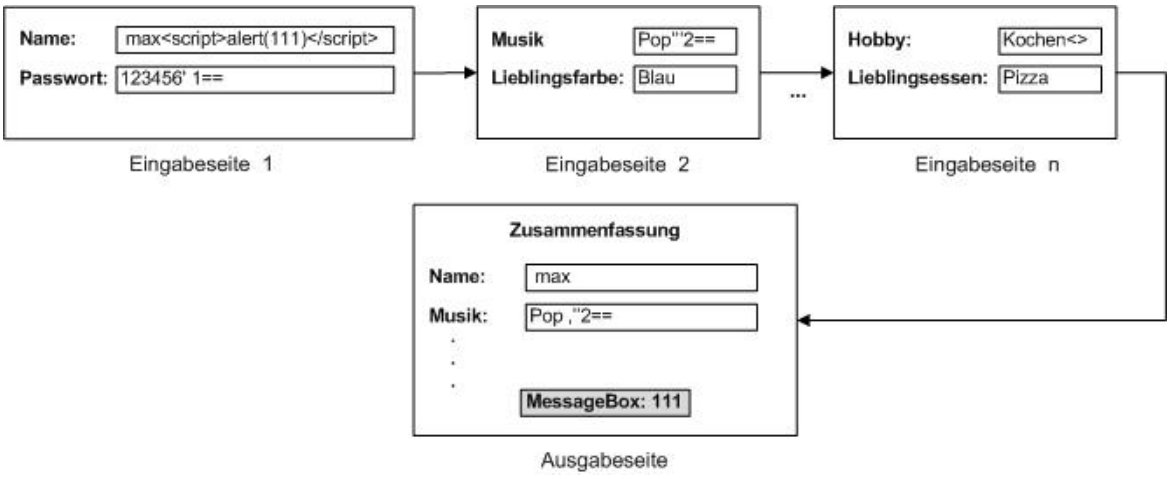


Abbildung 3.2.: Seitenübergreifende Schwachstelle einer Webanwendung

Zunächst wird der Benutzer auf der *Eingabeseite 1* aufgefordert ein Name und ein Passwort einzugeben. Bei den hierbei eingegebenen Werten fügt jedoch der Angreifer zusätzlichen Javascript-Code hinzu. So erweitert der Angreifer den eingegebenen Namen *max* um die Zeichenkette $< script > alert(111) < /script >$ welches einen typischer Test auf eine XSS-Schwachstelle darstellt. Des Weiteren fügt der Angreifer bei seinem eingegebenen Passwort *123456* zusätzlich die Zeichenkette $'1 ==$ ein, was unter anderem den Parameter *Passwort* der Webanwendung auf eine Verwundbarkeit gegenüber SQL-Injection testet. Zusätzlich muss der Angreifer, nach dem Absenden dieses ersten Formulars, weitere Informationen eingeben, die er zum Teil ebenfalls mit Programmcode versieht (siehe Parameter *Musik* und *Hobby*).
Im letzten Teil dieser Webseite (Ausgabeseite) bekommt der Angreifer eine Zusammenfassung seiner eingegebenen Werte angezeigt. Hierbei ist auch zu erkennen, dass auf der *Ausgabeseite* zusätzlich eine Message-Box mit dem Wert *111* ausgegeben wurde, welcher in der *Eingabeseite 1* beim Parameter *Name* zusätzlich als Javascript-Code mit übergeben wurde. Anhand dieser angezeigten Message-Box kann nun der Angreifer erkennen, dass die Webanwendung prinzipiell verwundbar ist, so dass ein Angreifer beliebigen Programm-Code ausführen kann.

Das Erkennen dieser seitenübergreifenden Schwachstelle kann hierbei nicht durch das *Standardvorgehen* (Schwachstelle ist im gleichen Response des Requests), wie es am Anfang des Abschnitts 3.3 beschrieben wurde, entdeckt werden, so dass ein Verfahren zum Finden von seitenübergreifenden Schwachstellen in Webanwendungen gerechtfertigt ist.

3.3.1. Erkennen von seitenübergreifenden Schwachstellen

Zum Erkennen solcher seitenübergreifenden Schwachstellen in Webanwendungen wie diese in Abbildung 3.2 werden die von der Webanwendung abgehenden Requests mit einem für eine Schwachstelle typischen Merkmal (Token) in einer Liste verwaltet. Ein solches Merkmal kann zum Beispiel der Response einer Webseite sein, der zum Erkennen von XSS-Schwachstellen benötigt wird. Aber auch die von einem Request erzeugten SQL-Anweisungen die als Eingabe für einen Datenbank-Server dienen, stellt ein typisches Merkmal dar, welches zum Beispiel zur Ermittlung von SQL-Schwachstellen dient. Das entsprechende Verfahren, zum Finden von SQL-Schwachstellen ist im Abschnitt 3.3.2 dargestellt.
Diese, in einer Liste (RequestToken-Liste) verwalteten Requests mit ihrem jeweiligen Merkmal, werden im nächsten Schritt untereinander auf ihre Gemeinsamkeiten verglichen. Hierbei werden alle abgespeicherten Merkmale (Tokens) aus der RequestToken-Liste mit jeden Request aus dieser Liste durchsucht. Dabei wird untersucht, ob der Wert eines Parameters eines Requests in einem beliebigen Merkmal (Token) enthalten ist. Auf diese Weise können beide Fälle, die in Abschnitt 3.3 beschrieben wurden, erkannt werden. Die folgende Tabelle 3.3 zeigt anschaulich eine verwaltete RequestToken-Liste mit Requests und ihren jeweiligen abgespeicherten Merkmalen, die zum Beispiel in der Abbildung 3.2 vorkommen. Zur Vereinfachung wurde hierbei jedoch nur das Merkmal *Response* eines Requests betrachtet.

ID	Request	Token (Response)
1	reg1.php	Name:... Passwort:...
2	reg2.php?id=1&name= `max<script>alert(111)</script>` &passwort=`123456$' 1==`	Musik:... Lieblingsfarbe:...
3	regN.php?id=2&farbe=Blau &musik=`Pop$'''2==`	Hobby:... Lieblingsessen:...
4	Summary.php?id=N&essen=Pizza &hobby=`Kochen<>`	Zusammenfassung Datum: 10.03.1999 Name: `max<script>alert(111)</script>` Musik: `Pop$'''2==` ...

Tabelle 3.3.: RequestToken-Liste mit abgespeicherten Requests und ihren Merkmalen

Die RequestToken-Liste mit den jeweiligen Responses als Merkmal in der Tabelle 3.3 besteht insgesamt aus 4 Einträgen. Diese Einträge werden nun miteinander auf Gemeinsamkeiten verglichen. Hierbei wird jeweils überprüft, ob ein Response einen Wert eines beliebigen Parameters von einem abgespeicherten Request enthält. Bei dem Durchsuchen der Responses auf die jeweiligen Werte der einzelnen Parameter treten dabei folgende Gemeinsamkeiten auf.

- Die Zeichenkette `max<script>alert(111)</script>` des Parameters *name* vom Request mit der ID 2 existiert in dem Response mit der ID 4
- Die Zeichenkette `Pop$'''2==` des Parameters *musik* von dem Request mit der ID 3 existiert in dem Response mit der ID 4
- Die Zahl (oder auch Zeichenkette) `1` des Parameters *id* vom Request mit der ID 2 kommt im Response mit der ID 4 vor (siehe Datum: 10.03.1999)

Die in diesem Beispiel gefundenen Gemeinsamkeiten sind bis auf den letzten Treffer (existierende 1 im Datum 10.03.1999) alle korrekt gefundene Schwachstellen in der Webanwendung. Der letzte Treffer hingegen liefert als *false positve* eine falsche Schwachstelle. Dieses zeigt, dass die Länge des Wertes eines Parameters einen signifikanten Einfluss auf das Erkennen einer korrekten, existierenden Schwachstelle hat. Somit hängt also die Erkennungsrate zum Herausfiltern der logischen Gemeinsamkeiten erheblich von der eingesetzten Metrik zum Finden von logischen Zusammenhängen ab.
Aus diesem Grund wird zusätzlich, neben dem reinen Durchsuchen des Merkmals der übergebene Wert eines Parameters im Request auf entsprechende Muster untersucht. Bei dieser Methode wird der Parameter-Wert zusätzlich auf Zeichen überprüft, welche *Anwender bei einer normalen Verwendung der Applikation* nicht eingeben. So wird der übergebene Wert eines Parameters im Request auf die folgenden Zeichen, welche in Tabelle 3.4 dargestellt sind, durchsucht.

Zeichen	Beschreibung	Verwendung
’	einzelnes Hochkomma	Abgrenzung von Werten bei SQL-Anweisungen oder Javascript
"	doppeltes Hochkomma	Abgrenzung von Werten bei SQL-Anweisungen oder Javascript
<	spitze Klamma auf	Leitet in HTML ein Tag ein
>	spitze Klamma zu	Beendet in HTML ein Tag
;	Semikolon	Leitet ein Kommentar in SQL-Anweisungen ein
␣	Leerzeichen	Abgrenzung zwischen einzelnen Kommandos bei SQL und javascript

Tabelle 3.4.: Liste der Sonderzeichen in den Werten der Parameter.

Anhand der hierbei verwendeten Metrik können so die false positives bei einem Treffer im jeweiligen Merkmal verringert werden. So kann bei der späteren Entwicklung des Prototypen das seitenübergreifende Suchen nach Schwachstellen unter Verwendung der beschriebenen Metrik, verwendet werden.

3.3.2. Finden von SQL-Injection-Schwachstellen

Die derzeitigen automatischen Verfahren zum Suchen einer Schwachstelle liefern eine gute Erkennungsrate bei einer normalen SQL-Injection. Hierbei wird eine entsprechende Fehlermeldung vom SQL-Server über die Webanwendung zum Client weitergeleitet, die dann durch automatische Verfahren entsprechend gedeutet wird. Zu beachten ist jedoch, dass auch hier keine seitenübergreifende Analyse auf SQL-Injection eingesetzt wird, wie es im Abschnitt 3.3 beschrieben wurde. Komplizierter wird das automatische Testen bei blinden SQL-Injections, da die Webanwendungen keine Fehlermeldungen vom SQL-Server mehr an den Client weiterleiten. So beschreiben die Artikel [Anl02] und [CA] Möglichkeiten zum Finden von blinden SQL-Injections mittels *Time delays*. Die Qualität des Erkennens von blinden SQL-Injections ist jedoch nicht immer gegeben. Da Webanwendungen einzelne Eingabeparameter unterschiedlich verarbeiten können (zum Beispiel durch Filtern von bestimmten Zeichen), und erst im Anschluss an den SQL-Server weiterreichen, kann sich der Aufbau einer SQL-Injection von Parameter zu Parameter ändern. Dieses bewirkt, dass eine Webanwendung nicht grundsätzlich gegen SQL-Injection sicher sein muss, bloß weil die eingegebenen blinden SQL-Injections kein Anzeichen auf eine Schwachstelle liefern. Das folgende Beispiel zeigt einen derartigen Fall.

Beispiel zum Nicht-Finden einer SQL-Injection.

In einer Webanwendung existiere der Request R mit dem Parameter *id*. Durch Verwendung eines Filters F werden die einzelnen Parameter des Requests R überprüft und gefiltert, so dass die Webanwendung gegenüber SQL-Injection geschützt ist. Des Weiteren werden alle eventuell auftretenden Fehlermeldungen vom SQL-Server nicht an den Benutzer der Webanwendung weitergeleitet, so dass eventuelle Angreifer eine Fehlermeldung bei der Ausführung eines SQL-Kommandos nicht erkennen können. Leider ist jedoch der integrierte Filter F schlecht umgesetzt, so dass er nur die Leerzeichen der einzelnen Parameter entfernt.

R := http://myservice.com/readData.php?id=1
F = {' '}

Des Weiteren existiert die Kombinationsmenge K, welche von einem Angreifer verwendet wird, um SQL-Injections in der Webanwendung zu finden. Hierbei verknüpft der Angreifer den Request R mit jedem Element aus der Kombinationsmenge K und versucht dadurch etwaige Schwachstellen bezüglich zu SQL-Injection zu finden. Da jedoch der Filter F alle Leerzeichen in einem Request R entfernt, erzeugen die Requests R_1 und R_2 nur eine Fehlermeldung, die der Angreifer jedoch nicht erkennen kann und somit denkt, dass die Webanwendung gegen SQL-Injection nicht verwundbar ist.

K = {' OR 1=1', 'OR 1=1'}
R_1 = R + K_1 = http://myservice.com/readData.php?id=1 OR 1=1
R_2 = R + K_2 = http://myservice.com/readData.php?id=1OR 1=1

k_{new} ='%20OR%201=1
R_3 = R + k_{new} = = http://myservice.com/readData.php?id=1%20OR%201=1

Wenn jedoch der Angreifer nun die Menge K um den Wert k_{new} erweitert, kann er die Verwundbarkeit gegenüber SQL-Injection erkennen, da mit Hilfe des Wertes k_{new} der Filter F *umgangen* wird. Somit wurde nachgewiesen, dass das Erkennen von SQL-Schwachstellen erheblich von der Kombinationsmenge K abhängt.

Ansatz zum Finden jeglicher SQL-Injections

Aus diesem Grund wird ein Ansatz entworfen, der es ermöglicht, selbst blinde SQL-Injections leichter zu erkennen. Für dieses Vorgehen wird das von der Webanwendung verwendete Datenbanksystem in einem Logging-Modus betrieben. Da diese Funktionalität heutzutage alle gängigen Datenbanksysteme wie zum Beispiel *MySQL, Oracle* und *MS SQL Server* plattformunabhängig unterstützen, können so alle aus der Webanwendung abgesetzten SQL-Statements aufgezeichnet werden. Hierbei verwenden die verschiedenen Datenbanksysteme unterschiedliche Konzepte zum Mitprotokollieren der SQL-Statements. So erlauben einige Datenbanksysteme wie *Oracle* (siehe [Ora]) nur das Mitprotokollieren in einer Datei, wohin gegen andere Datenbanksysteme wie *MySQL* (siehe [MyS]) zusätzlich eine interne Tabelle für das Logging verwenden.
Um nun die abgesetzten SQL-Statements der Webanwendung zu ermitteln muss jedoch bei dieser Methode jedes einzelne Datenbanksystem gesondert in den Prototypen integriert werden, da jedes Datenbanksystem anders angesprochen wird. Für die Erstellung des Prototypen wird aus diesem Grund nur das MySQL-Datenbanksystem betrachtet. Andere Systeme können jedoch auf ähnlicher Weise integriert werden. Eine sehr gute Möglichkeit, um eine einheitliche Methode für die verschiedenen Datenbanksysteme zu integrieren, beschreibt das Projekt *Sania* unter [KKH+07]. Es löst das Problem der unterschiedlichen Datenbanksysteme über einen SQL-Proxy, der alle SQL-Statements entgegennimmt und sie im zweiten Schritt an das Datenbanksystem weiterleitet.

Vorgehensweise zum Finden von SQL-Injections

Die Vorgehensweise bei MySQL-Datenbanksystemen ist zunächst das Aktivieren des Logging-Modus, so dass alle SQL-Statements aufgezeichnet werden. Hierbei protokolliert der MySQL-Server alle abgesetzten SQL-Statements in eine interne Datenbanktabelle, was das Auslesen

der einzelnen SQL-Statements sehr vereinfacht. Über eine anschließende aktive Verbindung zur Datenbank können dann die von der Webanwendung abgesetzten SQL-Statements ermittelt und so auf Schwachstellen hin untersucht werden. Hierfür werden zunächst bei einem HTTP-Request die einzelnen Parameter mit einem zusätzlichen Muster (Pattern) verknüpft, welches später zur Erkennung einer Schwachstelle dient. Solche Patterns sind zum Beispiel ein einzelnes Hochkomma (’), ein doppeltes Hochkomma (’’), ein Leerzeichen (␣) oder ein Kommentar (−−) und dienen unter anderem in SQL zur Abgrenzung einzelner Eingabeparameter. Nachdem der abgesetzte Request mit den manipulierten Parametern neue SQL-Statements ausgelöst hat, werden diese auf ihre übergebenen Parameter inklusive ihrer übergebenen Patterns untersucht. Bei einem Auftreten eines aus dem Request übergebenen Eingabeparameters inklusive seines Patterns in einem SQL-Statement kann dann auf eine Schwachstelle bezüglich einer SQL-Injection geschlossen werden. Das folgende Beispiel zeigt vereinfacht, wie das beschriebene Vorgehen in der Praxis funktioniert.

Beispiel zum Finden einer SQL-Injection.

Der Request R beinhaltet die Parameter *user* und *passwd* mit den jeweiligen Werten *max* und $1234'56$. Das aus diesem Request resultierende SQL-Statement Q übernimmt die beiden übergebenen Werte der Parameter ungefiltert. Im nächsten Schritt wird versucht das SQL-Statement Q auszuführen (führt jedoch aufgrund des Zeichens $'$ im Wert $1234'56$ zu einem Fehler). Bei der anschließenden Auswertung, ob eine Schwachstelle in diesem SQL-Statement vorliegt, wird nun der im Request übergebene Wert des Parameters *passwd* in dem abgesetzten SQL-Statement gefunden, was aufgrund des Zeichens $'$ im Parameter *passwd* auf eine Sicherheitslücke hinweist.

```
R:= http://myservice.com/login.php?user=max&passwd=1234’56

Q:= SELECT * FROM users WHERE username = ’max’
       AND password = ’1234’56’;
```

Den genauen Ablauf zum Ermitteln einer SQL-Injection unter der Verwendung der seitenübergreifenden Schwachstellen-Suche mit Datenbank-Logging zeigt das Aktivitätsdiagramm in Abbildung 3.3. Hierbei ist zu beachten, dass die Variable *reqTok* ein *RequestToken-Objekt* darstellt, welches später im Abschnitt 4.6.2 beschrieben wird. Des Weiteren stellt die Variable *reqTokList* die in Abschnitt 3.3.1 beschriebene RequestToken-Liste dar.

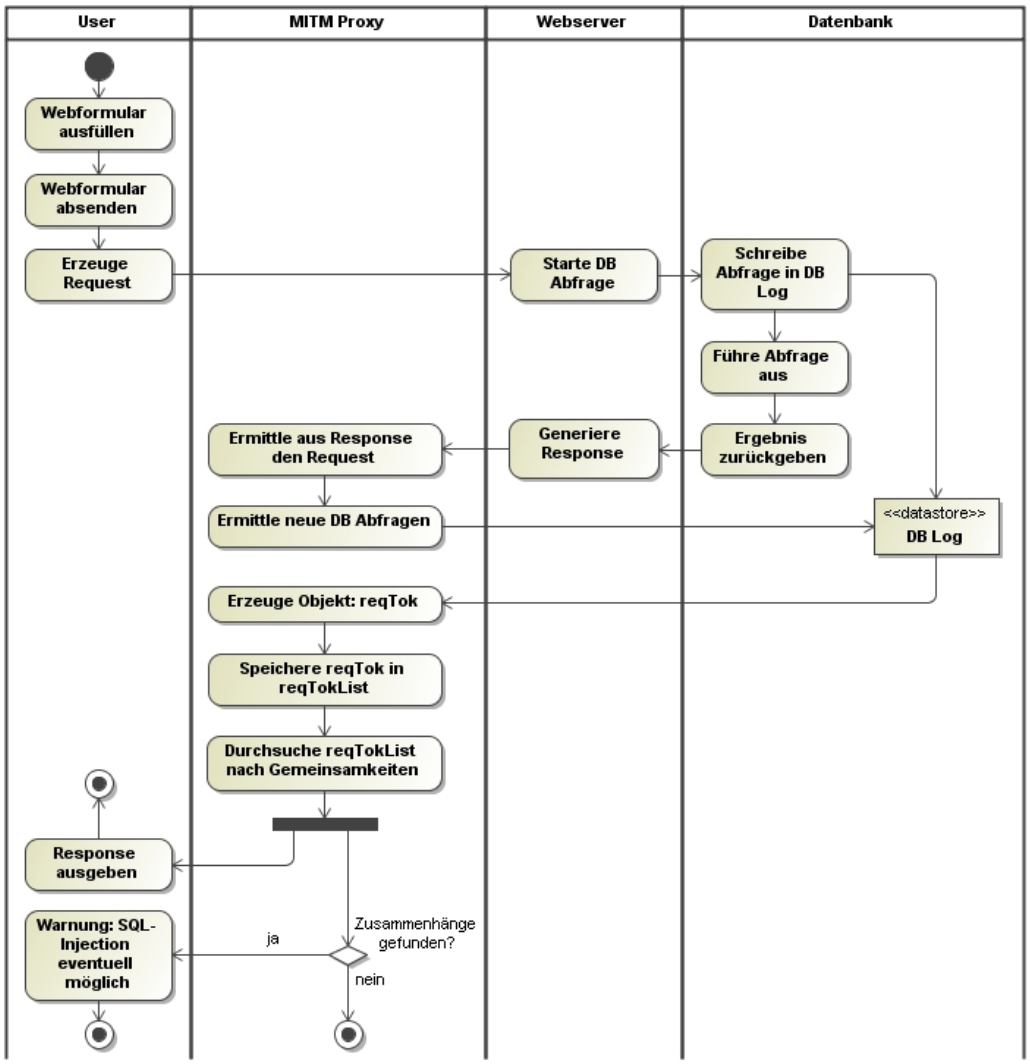


Abbildung 3.3.: Finden einer SQL-Injection - Aktivitätsdiagramm

3.3.3. Finden von XSS-Schwachstellen

Das Standardverfahren zum automatischen Testen auf eine XSS-Schwachstelle basiert auf der Quellcodeanalyse einer Webseite. Hierbei wird mit einfachen Bordmitteln, wie das Suchen einer Zeichenkette überprüft, ob eine übergebene XSS-Zeichenkette beim Request im Response der Webseite wieder auftaucht. Bei einem Auftreten dieser XSS-Zeichenkette wird daraus auf eine Schwachstelle bezüglich von XSS geschlossen. Diese Methode ist in ihrer Funktionsweise korrekt. Jedoch kann es vorkommen, dass eine, im Request übergebene, XSS-Zeichenkette im Response der Webseite auftaucht, aber in einem Kontext der Webseite steht, in der diese Zeichenkette nicht ausnutzbar ist. Das folgende Beispiel zeigt einen solchen Kontext, in dem eine Schwachstelle bezüglich zu Cross-site-scripting nicht verwundbar ist.

Beispiel einer nicht ausnutzbaren XSS-Schwachstelle.

<body>

```
 <font color="red" style="font-style:italic;<script>alert(1)</script>">
  nicht ausnutzbar
 </font><br>
 <font color="red" style="font-style:italic;"<script>alert(2)</script>"">
  ausnutzbar
 </font>
</body>
```

Daher wird ein neuer Ansatz aufgezeigt, der über das übliche Verfahren der Quellcodeanalyse hinausgeht. Hierbei wird zusätzlich auf der Client-Seite überprüft, ob die im Request übergebene XSS-Zeichenkette in der Webanwendung ausnutzbar ist. Für dieses Verfahren wird zunächst an den einzelnen Parametern des abzusetzenden HTTP-Requests zusätzlich eine Zeichenkette (Pattern) angehängt, welche typisch für XSS ist. Ein solcher Ausdruck kann zum Beispiel die Zeichenkette $< script > w3afAttack1() < /script >$ sein, der bei korrekter Ausführung die Javascript-Funktion $w3afAttack1()$ startet, die den Benutzer informiert, dass die XSS-Schwachstelle auch ausgenutzt werden kann.
Nachdem der Request mit den Patterns an den Webserver gesendet wurde und eine entsprechende Antwort (Response) zurückgesendet hat, wird zunächst der Response auf die vom Request übergebenen Parameter einschließlich der Patterns untersucht. Bei einem Entdecken der übergebenen Zeichenkette im Response wird dieses dem Benutzer mitgeteilt, dass der übergebene Parameter vom Request eventuell eine XSS-Lücke aufweist. Darüber hinaus wird zusätzlich auf der Client-Seite versucht, das übergebene Pattern auszuführen, welches dann den Benutzer informiert, dass die XSS-Lücke auch ausnutzbar ist. Das folgende Beispiel demonstriert das Finden einer XSS-Schwachstelle in einem Request.

Beispiel zum Finden einer XSS-Schwachstelle.

```
R:= http://myservice.com/login.php?passwd=123456&
           user=max<script>w3afAttack1()</script>

RES:= <html>...Passwort für 'max<script>w3afAttack1()</script>'
      ist falsch...</html>

function w3afAttack1() {
    // informiere Benutzer über Verwundbarkeit
}
```

Der Request R beinhaltet die Parameter *passwd* und *user*, wobei der Parameter *user* zusätzlich ein Pattern zum Finden einer XSS-Lücke enthält. Der aus dem Request R resultierende Response RES beinhaltet, ohne irgendwelche gefilterten Zeichen des Requests, unter anderem den übergebenen Wert *max* des Parameters *user* einschließlich des Patterns `<script>w3afAttack1()</script>`, welcher beim Durchsuchen des Response gefunden wurde. Über diesen Treffer wird der Benutzer benachrichtigt. Zusätzlich wird nun versucht die Javascript-Funktion $w3afAttack1()$ des übergebenen Patterns auf der Browser-Seite auszuführen, welche dann bei einem Erfolg dem Benutzer ebenfalls mitteilt, dass diese Funktion korrekt interpretiert wurde und somit auch in einem verwundbaren Kontext der Webseite steht.

Das Aktivitätsdiagramm 3.4 zeigt noch einmal die Funktionsweise zum Finden einer XSS-Schwachstelle mit zusätzlicher client-seitiger Analyse und der seitenübergreifenden Suche wie

sie in Abschnitt 3.3.1 beschrieben wurde. Wie bereits erwähnt, dient die Variable *reqTok* zur Darstellung eines *RequestToken-Objekts*, welches im späteren Abschnitt 4.6.2 beschrieben wird. Außerdem dient die Variable *reqTokList* der Darstellung der RequestToken-Liste aus Abschnitt 3.3.1.

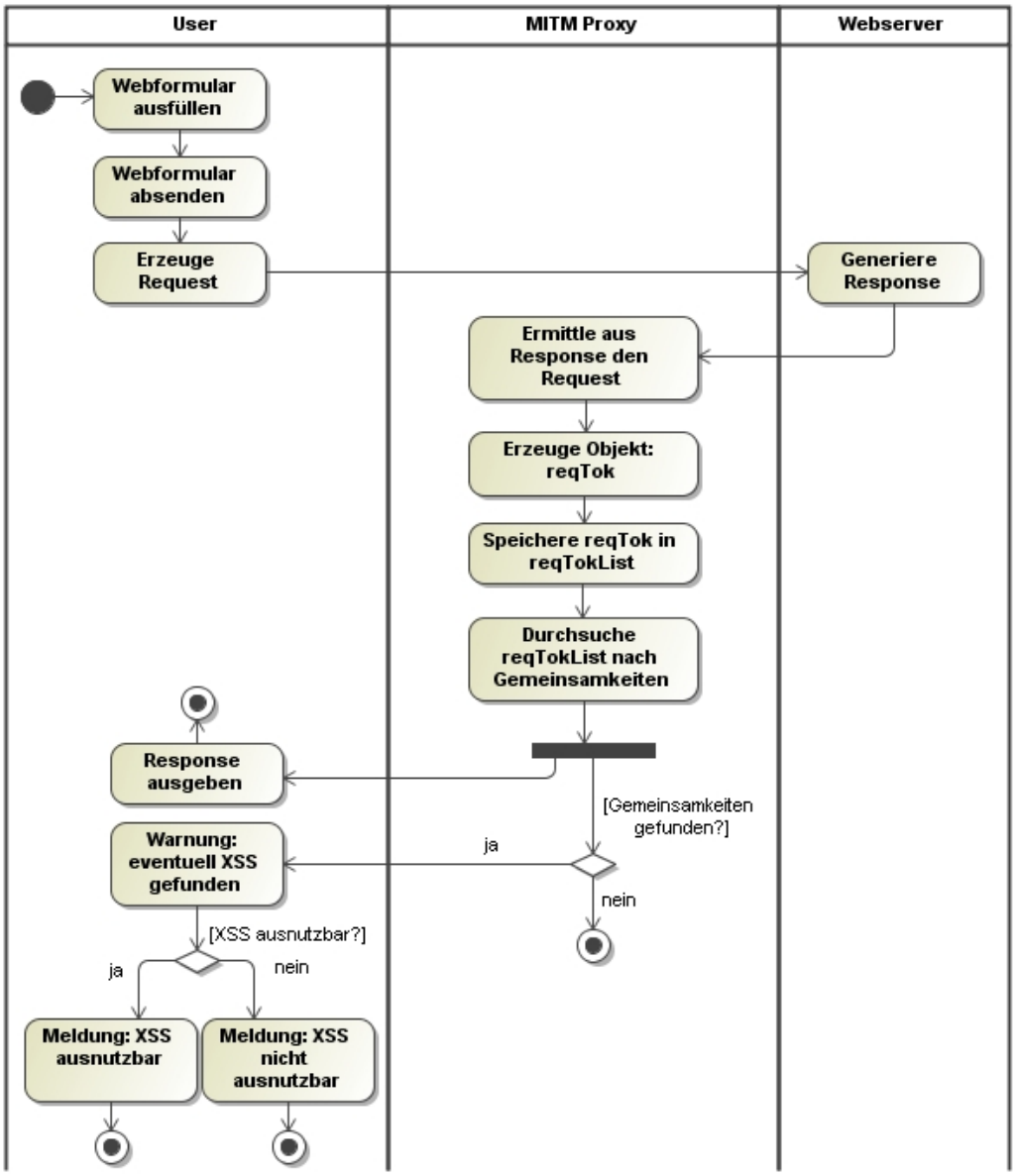


Abbildung 3.4.: Finden einer XSS-Schwachstelle - Aktivitätsdiagramm

3.4. Integration der Schwachstellenanalyse in die Architektur

Das Einbinden der in den Abschnitten 3.3.2 und 3.3.3 beschriebenen Techniken zum Finden von SQL-Injections und XSS wird durch einen eigenen entwickelten Plugin-Manager vorgenommen. Mit Hilfe dieses Vorgehens können so zu einem späteren Zeitpunkt zusätzliche Methoden zur Erkennung von Schwachstellen in Form von Plugins in den Prototypen integriert werden. Hierbei stellt der Plugin-Manager automatisch die Funktionsweise, des in Abschnitt 3.3.1 beschriebenen Verfahrens zur seitenübergreifenden Suche zur Verfügung, so dass alle Plugins auf diese Methode zugreifen können.

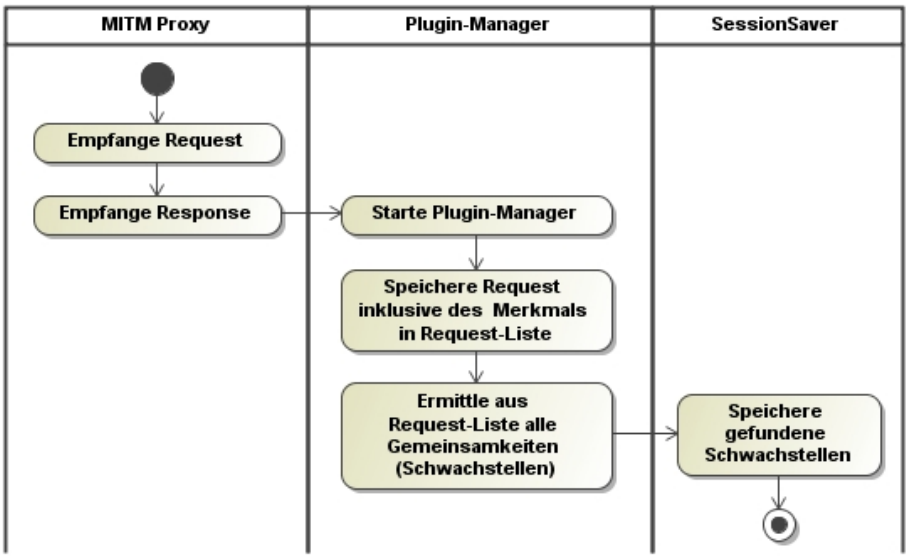


Abbildung 3.5.: Vereinfachte Funktionsweise des Plugin-Managers

Die Abbildung 3.5 zeigt die vereinfachte Funktionsweise des Plugin-Managers im zu erstellenden Prototypen. Nachdem der MITM-Proxy den Response eines Requests empfangen hat, sendet er diese Informationen zum Plugin-Manager, welcher selber ein Teil des MITM-Proxys ist. Dort wird zusätzlich zum Request und Response ein Merkmal (Token) erhoben, welches für das Finden einer bestimmten Sicherheitslücke im weiteren Verlauf benötigt wird. Diese nun gesammelten Informationen werden in eine Request-Liste verwaltet, die im weiteren Verlauf auf Gemeinsamkeiten mit Hilfe von *pattern matching*-Verfahren zwischen den einzelnen Einträgen in der Liste durchsucht wird. Die dabei gefundenen Schwachstellen in Form von gefundenen Gemeinsamkeiten werden im letzten Schritt für die spätere Analyse im *SessionSaver* gespeichert. So kann jede Komponente der entwickelten Architektur auf die gefundenen Schwachstellen zugreifen.

3.5. Benutzeroberfläche zur Analyse der Webanwendung

Eine grundlegende Vorgehensweise zur Untersuchung einer Webanwendung auf Schwachstellen ist es, die Webanwendung aus dem Webbrowser heraus zu untersuchen. Für diese

Aufgabe wird zusätzlich in die originale Webanwendung auf der Client-Seite eine grafische Benutzeroberfläche in Form einer WebGUI integriert. So kann der Benutzer später alle Tests und Analysen direkt aus der Webanwendung vornehmen. Die folgende Abbildung 3.6 zeigt eine abstrakte Darstellung dieser WebGUI.

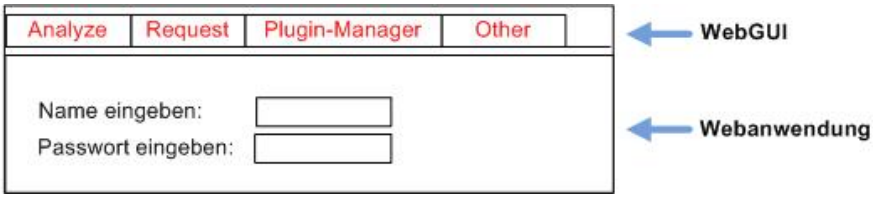


Abbildung 3.6.: Vereinfachte Darstellung der WebGUI

Das Bild zeigt eine vereinfachte Darstellung einer Webseite im Browser. Hier ist zu erkennen, dass im oberen Teil des Bildes die WebGUI sich in die originale Webanwendung als Menü einbettet. Dieses Menü besteht hierbei aus den Funktionsgruppen *Analyze*, *Request*, *Plugin-Manger* und *Other*, welche die in den vorangegangen Abschnitten beschriebenen Techniken für den Benutzer darstellt. Hierbei beschreibt der Menüpunkt *Analyze* die Konzepte aus Kapitel 3.1, welches die originale Webseite auf ihre existierenden DOM-Objekte analysiert. Unter der Funktionsgruppe *Request* kann der Benutzer die zuvor abgespeicherten Requests, welche für die seitenübergreifende Suche dienen, verwalten. Über die Funktionsgruppe *Plugin-Manager* können die gefundenen Schwachstellen der Webanwendung detailliert eingesehen und verwaltet werden. Der letzt Menüpunkt *Other* stellt dem Benutzer die Möglichkeit zum client-seitigen automatisierten Testen zur Verfügung. Außerdem können über diesen Menüpunkt alle notwendigen Konfigurationen aus dem Browser direkt vorgenommen werden.
Im unteren Teil des Bildes befindet sich die originale Webanwendung, so dass der Benutzer mit Hilfe der eingebetteten WebGUI jederzeit den Prozess zum Erkennen von Schwachstellen beeinflussen kann. Dadurch ist er unter anderem in der Lage nur die für ihn relevanten Teile der Webanwendung zu analysieren.

3.6. Desgin der neuen Architektur

Die in den vorangegangenen Abschnitten gezeigten Techniken und Vorgehensweisen können nun in die in Abbildung 3.1 beschriebene Architektur integriert werden. Der vollständige Aufbau und der Zusammenhang der einzelnen Komponenten ist in Abbildung 3.7 dargestellt.

Zum Darstellen der in Abschnitt 3.5 beschriebenen WebGUI, muss zusätzlich Programmcode in die originale Webanwendung eingeschleust werden. Diese Aufgabe wird mit Hilfe des MITM-Proxys vom w3af-Framework vorgenommen. Zusätzlich kommuniziert dieser MITM-Proxy über einen Steuerkanal mit den einzelnen Komponenten der Architektur. Über diesen können Informationen innerhalb der Architektur ausgetauscht werden.
Ein zentraler Bestandteil ist der *SessionSaver*. Er ist logisch gesehen ein Teil des MITM-Proxys und speichert alle notwendigen Informationen wie Requests mit den zugehörigen Parametern für ein späteres semi-automatisches Testen. Darüber hinaus können sämtliche Komponenten lesend auf den SessionSaver zugreifen. Das Ändern des SessionSavers geschieht jedoch grundsätzlich über den MITM-Proxy.

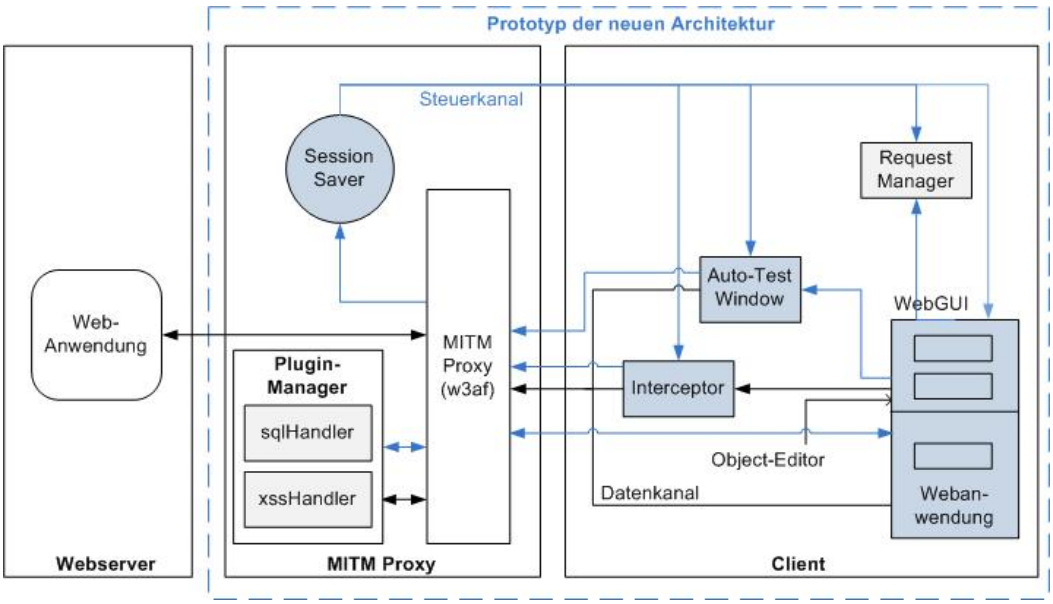


Abbildung 3.7.: Entwurf der neuen Architektur

Der Startpunkt zum Testen der Webanwendung ist eine erweiterte Browseroberfläche (siehe Abschnitt 3.5), die aus der WebGUI und der originalen Webanwendung besteht. Die WebGUI zeigt hierbei in Form eines *Object-Editors* alle vorhandenen DOM-Objekte an, die für das Analysieren der Webanwendung von Bedeutung sind. So werden beispielsweise alle Links, Eingabefelder, Buttons und *versteckte Objekte* (in HTML meist hidden-fields) angezeigt. So kann der Benutzer alle Werte der einzelnen Objekte komfortabel editieren und im nächsten Schritt den hierbei erzeugten Request mit den Werten an den Webserver senden. Da jedoch Webanwendungen die Werte einzelner Parameter durch Programmlogik verändern können (zum Beispiel durch Filterfunktionen in Javascript), durchläuft jeder abgesendete Request aus der WebGUI den *Interceptor*. Dort können etwaige Änderungen der Parameter durch Filterfunktionen eingesehen und manipuliert werden. Zusätzlich dazu erlaubt der Interceptor das Speichern des endgültigen Requests für eine spätere seitenübergreifende Suche nach Schwachstellen. Dieses ermöglicht ihm ein späteres automatisiertes Testen einzelner Teile (Workflows) der Webanwendung. Auch kann er an dieser Stelle die Antwort für den jeweiligen Request speichern (Request verwerfen, manipulieren oder unverändert senden), so dass dieser beim nächsten Aufrufen automatisch ausgeführt wird. Die aufgenommenen Requests können im Anschluss mit Hilfe des *Request-Managers* verwaltet werden.
Darüber hinaus verfügt die neue Architektur über ein *Auto-Test-Window*. Dieses stellt die Oberfläche zum semi-automatischen Testen bereit. Durch den Zugriff auf die im SessionSaver gespeicherten Requests, kann der Benutzer Teile (Workflows) der Webanwendung explizit auf bestimmte Schwachstellen untersuchen. Zur Zeit sind im Prototyp die Erkennung von SQL-Injection und Cross-Site-Scripting vorgesehen, welche als Module im *Plugin-Manager* verankert sind. Der eigens hierfür entwickelte Plugin-Manager wertet hierbei die einzelnen Requests mit ihren Merkmal aus. Das daraus resultierende Ergebnis wird dann mit Hilfe des MITM-Proxys im SessionSaver gespeichert.

4. Implementierung

Nach der Erstellung des System-Entwurfs in Kapitel 3.6 wird im weiteren Verlauf die eigentliche Realisierung der Architektur in Form eines Prototypen beschrieben. Hierfür werden zunächst die zu lösenden Aufgaben der neuen Architektur in zwei Kategorien unterteilt. Die erste Kategorie beschäftigt sich mit der Implementierung eines *MITM-Proxys* der logisch betrachtet auf der Proxy-Seite arbeitet. Die zweite Kategorie befasst sich mit der Implementierung einer client-seitigen *WebGUI*, über die der Benutzer letztendlich alle Befehle zur Steuerung des Proxys absetzen kann. Die Tabelle 4.1 zeigt alle zu bewältigen Aufgaben, die in der neuen Architektur umgesetzt werden müssen.

MITM-Proxy	**WebGUI**
Integration einer WebGUI zur benutzerfreundlichen Analyse einer Webanwendung	Integration eines Mechanismus zur Analyse des DOMs einer Webseite
Integration eines SessionSavers zur Konfigurationsverwaltung	Integration eines Interceptors zum Abfangen von Requests
Integration eines Mechanismus zum Finden von Schwachstellen in Form eines Plugin-Managers	Integration einer Methode zur automatischen Analyse von Requests

Tabelle 4.1.: Aufteilung der Aufgaben zur Realisierung des Prototypen

Im weiteren Verlauf dieses Kapitels werden die Lösungen zu den in der Tabelle 4.1 beschriebenen Aufgaben aufgezeigt. Hierbei wird der gesamte zu entwickelnde MITM-Proxy in der Programmiersprache *Python* entwickelt, da bereits das gesamte *w3af*-Framework in dieser Sprache implementiert wurde. Die WebGUI hingegen wird in den Sprachen HTML und Javascript entwickelt, da diese sich innerhalb einer Webapplikation integriert.

4.1. Implementierung des MITM-Proxys

Das Integrieren eines neuen Proxys ist die wichtigste Aufgabe bei der Entwicklung des Prototypen, da mit dessen Hilfe alle Analysen zur Schwachstellensuche einer Webanwendung realisiert werden können.
Grundsätzlich existiert bereits im *w3af* ein MITM-Proxy mit dem Namen *localproxy*, der jedoch im Funktionsumfang zum Bezug auf die in Tabelle 4.1 genannten Aufgaben sehr beschränkt ist. Aus diesem Grund wurde zusätzlich ein weiterer MITM-Proxy implementiert, der jedoch von dem bereits existierenden Proxy abgeleitet ist. So kann der neue MITM-Proxy bereits auf vorhandene Funktionalitäten zurückgreifen und darüber hinaus weitere Möglichkeiten, wie die Integration des Plugin-Managers zur Schwachstellenanalyse anbieten, die für die neue Architektur benötigt werden.

4.1.1. Aufbau des MITM-Proxys

Das im w3af integrierte Konzept sieht vor, dass zu jedem MITM-Proxy ein dazugehöriger *Proxy-Handler* vorhanden sein muss, der für die Steuerung und den Ablauf der eingehenden Requests und deren Response verantwortlich ist. So existiert zu dem bereits implementierten Proxy *localproxy* der dazugehörige Proxy-Handler *w3afLocalProxyHandler*, der Grundfunktionalitäten, wie das Abfangen und Weiterleiten von Requests zur Verfügung stellt.
Wie bereits erwähnt, wurde ein neuer Proxy mit dem Namen *localanalyzeproxy* mit dem dazugehörogen Proxy-Handler *w3afLocalAnalyzeProxyHandler* implementiert, der zusätzlich zu den bereits im existierenden Proxy integrierten Techniken, die Funktionen aus der Tabelle 4.1 zur Verfügung stellt. Daher ist der zu entwickelnde Proxy einschließlich seines Proxy-Handlers von der Oberklasse *localproxy* und *w3afLocalProxyHandler* abgeleitet. Die Abbildung 4.1 zeigt eine Übersicht zur Einordnung des neuen Proxys.

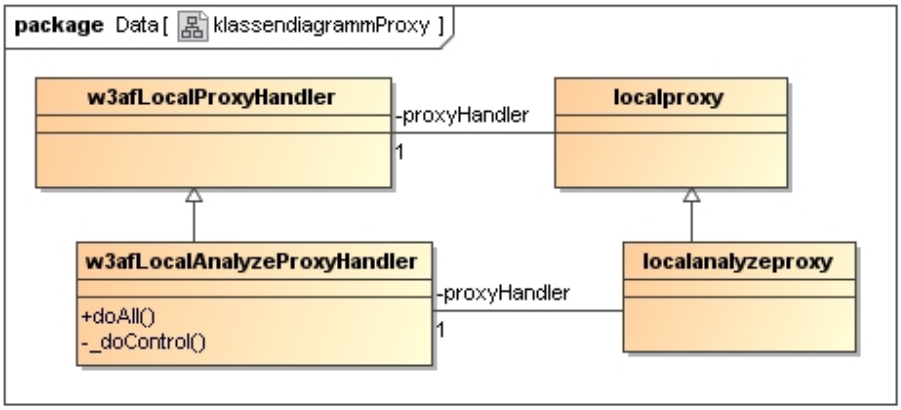


Abbildung 4.1.: Einordung des neues Proxys in die Struktur des w3af-Frameworks.

Methode: doAll
Die Methode *doAll* fängt alle abgesetzten Requests vom Browser zum Webserver ab. Sie kann als Einstiegspunkt für das Lösen der in der Tabelle 4.1 genannten Aufgaben angesehen werden. Des Weiteren bietet diese Funktion die Möglichkeit, einzelne eingehende Requests auf bestimmte Muster hin zu analysieren, welche dann im weiteren Verlauf entsprechende Funktionalitäten im MITM-Proxy auslösen.

Methode: _doControl
Die Methode *_doControl* ist eine interne Funktion, die von der Methode *doAll* aufgerufen wird. Sie wird immer aufgerufen, wenn der im Browser abgesetzte Request ein Steuerungs-Request ist, welcher zur Kontrolle und Steuerung des MITM-Proxys dient. Solche Requests können zum Beispiel das Löschen oder Setzen von bestimmten Informationen im SessionSaver auslösen.

4.1.2. Integration in das w3af-Framework

Das w3af-Framework beinhaltet für alle Benutzer-Interaktionen neben einer textbasierten Oberfläche zusätzlich auch eine grafische Oberfläche, die im Funktionsumfang jedoch mächtiger ist als die textbasierte Darstellung. So kann zum Beispiel der im w3af bereits integrierte MITM-Proxy zusätzlich über die grafische Benutzeroberfläche gesondert verwendet werden. Aus diesem Grund wurde der zu entwickelnde Prototyp in Form eines neuen MITM-Proxys ebenfalls in die grafische Benutzeroberfläche integriert. Hierbei wurden die Anpassungen direkt in der entsprechenden Klasse *ProxiedRequests* der Datei *proxywin.py* vorgenommen, da die Erstellung einer Unterklasse von *ProxiedRequests* keinen nennenswerten Vorteil erbringt. Der folgenden Code-Ausschnitt zeigt, wie der neue MITM-Proxy in die Klasse *ProxiedRequests* eingebunden wird.

Einbinden des neuen Proxys in das w3af-Framework.

```
from core.controllers.daemons import localanalyzeproxy
...
class ProxiedRequests(entries.RememberingWindow):
  def __init__(self, w3af):
    ...
    # Create a button to control the proxy
    ('DomAn', gtk.STOCK_JUMP_TO, _('_DOM Analyze'),
      None, _('Analyze DOM of webpage'),
      self._toggle_dom, False),
    ...

  def _toggle_dom(self, widget):
    '''Toggle the dom flag.'''
    self.proxy.setW3afInstance(self.w3af)
    domactive = widget.get_active()
    self.proxy.setDomAnalyze(domactive)

  ...
  def _startProxy(self):
    # original: self.proxy = localproxy.localproxy(ip, int(port))
    self.proxy = localanalyzeproxy.localanalyzeproxy(ip, int(port))
```

Beim Einbinden des neuen Proxys müssen mehrere kleine Anpassungen vorgenommen werden. Im ersten Schritt wird hierbei die neue Klasse des MITM-Proxys in das vorhandene Python-Script eingebunden (Zeile 1). Um hierbei Inkonsistenzen zu vermeiden, ist jedoch zu beachten, dass sich der neue Proxy an die Struktur des originalen Proxys des w3af-Frameworks hält. Sind diese Voraussetzungen gegeben, kann im zweiten Schritt (Zeile 4 bis 9) der Konstruktor der Klasse *ProxiedRequests* dahin gehend erweitert werden, dass der neue Proxy über einen Schaltknopf an- und ausschaltbar ist. Über diesen Button können so die zusätzlichen Funktionalitäten des Prototypen verwendet werden. Die Funktionsweise dieses Schaltknopfs ist in der Methode *_toggle_dom* hinterlegt (Zeile 12 bis 16). Im letzten Schritt wird die Methode *_startProxy* entsprechend abgeändert, so dass der neue Proxy *localanalyzeproxy* für den weiteren Verlauf verwendet wird (Zeile 19 bis 21).

Die Abbildung 4.2 zeigt die grafische Oberfläche mit der neuen Funktionalität des Prototypen. Des Weiteren ist hier der erstellte Schaltknopf (*DOM Analyze*), der zum Starten und Beenden der zusätzlichen Funktionen im Prototypen dient, zu erkennen.

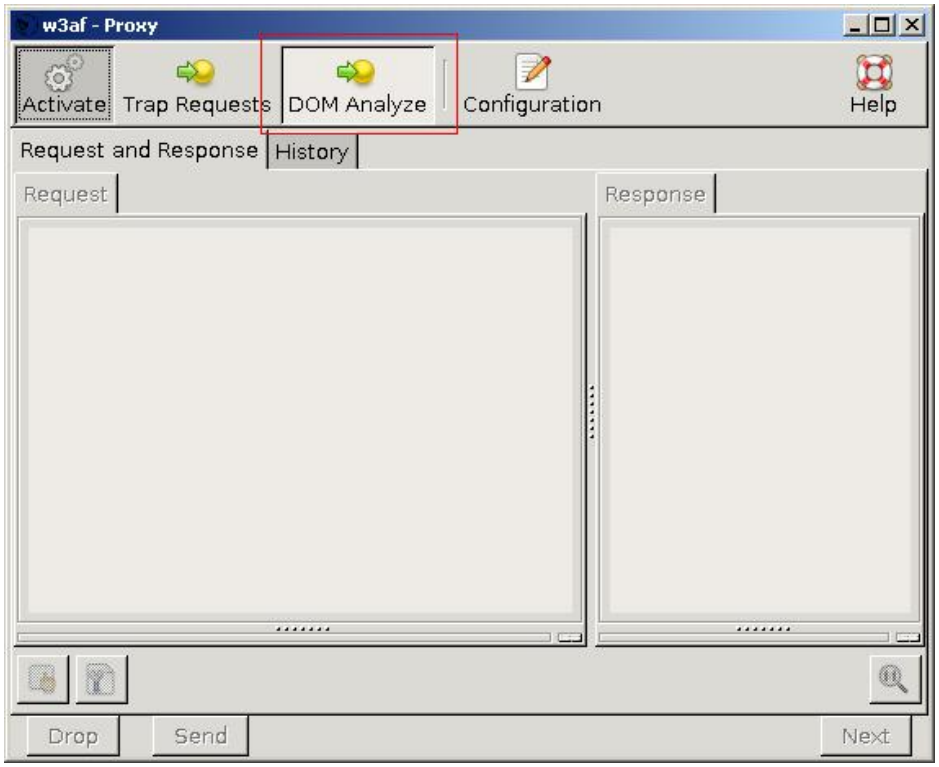


Abbildung 4.2.: Darstellung des neuen Proxys.

4.1.3. Funktionsweise des MITM-Proxys

Wie bereits im Abschnitt 4.1.1 erwähnt, implementiert die Methode *doAll* unter anderem die Funktionalität zur Steuerung des MITM-Proxys über spezielle Requests. Diese internen Requests werden vom Webbrowser abgesetzt und ermöglichen zum einen die Ansteuerung einzelner Funktionen (Control-URLs), die im MITM-Proxy implementiert sind und zum anderen die Darstellung einer WebGUI im Webbrowser des Benutzers (GUI-URLs). Hierbei ist zu beachten, dass diese internen Requests nicht an den Webserver weitergeleitet werden, da dieser die einzelnen Requests nicht interpretieren kann. Die folgende Tabelle 4.2 zeigt eine Darstellung, der im Prototypen verwendeten Request-Typen.

	interner Request		**externer Request**
	Control-URL	**GUI-URL**	
Aufgabe	Steuerung der einzelnen Funktionen des MITM-Proxys	Laden einzelner Teile zur Darstellung der WebGUI im Browser	normaler Request der Webanwendung
Muster	analyzeApp/control/	analyzeApp/gui/	
Beispiel	http://.../analyzeApp/control/?clearvul	http://.../analyzeApp/gui/w3af_layout.html	http://.../index.html

Tabelle 4.2.: Übersicht der verwendeten URLs im Prototypen.

Mit Hilfe des oben beschriebenen Vorgehens ist es möglich, neben der Analyse der Webanwendung ebenfalls den MITM-Proxy über eine WebGUI zu steuern. Die gesamte Funktionsweise des MITM-Proxys ist in Abbildung 4.3 noch einmal vereinfacht dargestellt.

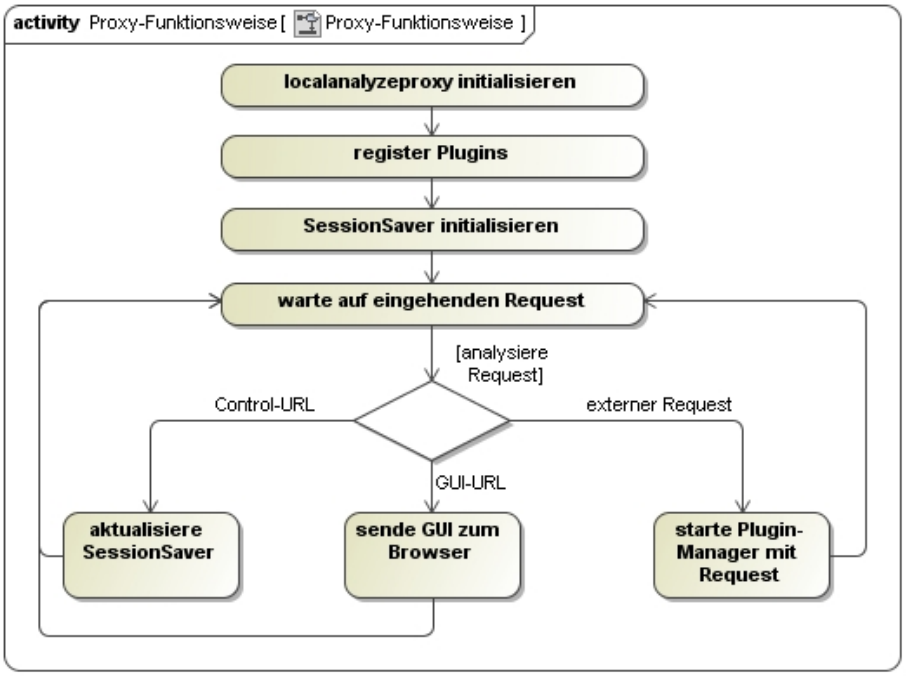


Abbildung 4.3.: Funktionsweise des neuen Proxys

Nach dem Starten des w3af-Frameworks und der Proxy-Oberfläche wird der neue Proxy *localanalyzeproxy* zusammen mit seinem Proxy-Handler *w3afLocalAnalyzeProxyHandler* initialisiert. Dieser registriert zunächst die existierenden Plugins beim Plugin-Manager (siehe Abschnitt 4.6). Nach dem anschließenden Initialisieren des SessionSavers (siehe Abschnitt 4.3) ist der Proxy bereit um eingehende Request entgegenzunehmen.
Nach dem Empfangen eines neuen Requests aus vom Webbrowser, wird dieser zunächst auf die in Tabelle 4.2 beschriebenen Muster untersucht. Hierbei wird bei einer Control-URL die entsprechende Funktionalität ausgeführt, welche im weiteren Verlauf den Zustand des SessionSavers verändert. Bei einer entsprechenden GUI-URL wird hierbei die angefragte Webseite der WebGUI entsprechend an den Webbrowser gesendet, wohingegen bei einem eingehenden externen Request dieser Request an den Plugin-Manager weitergeleitet wird, der im Weiteren den Request mit seinem entsprechenden Response auf Schwachstellen untersucht.
Nach der Verarbeitung des eingegangenen Requests, wechselt der Proxy in den Zustand zurück, in dem er weitere Requests entgegen nehmen kann. Bei der anschließenden Verarbeitung weiterer Requests, folgt er dem gleichen, zuvor beschriebenen Vorgehen.

4.2. Integration der WebGUI

Wie die Abbildung 3.7 im Kapitel 3.6 aufzeigt, wird für die Bedienung des Prototypen eine WebGUI implementiert, mit dessen Hilfe der Benutzer die Analyse der Schwachstellen bei

einer Webanwendung vornehmen kann. Aber auch die Steuerung des MITM-Proxys ist über dieses Web-Frontend möglich.

4.2.1. Einschleusen von Programmcode in die Webanwendung

Da jedoch die WebGUI komplett in der originalen Webseite einer Anwendung integriert werden soll, muss für das Lösen der Aufgabe zunächst eigener Programmcode in den Hauptteil der originalen Webseite eingeschleust werden. Nach dem Einschleusen des eigenen Codes kann dieser dann in den nächsten Schritten die weiteren Teile der WebGUI vom MITM-Proxy nachladen. Die folgende Abbildung 4.4 zeigt hierbei den Prozess, wie zusätzlich eigener Programmcode in die Webseite eingeschleust werden kann.

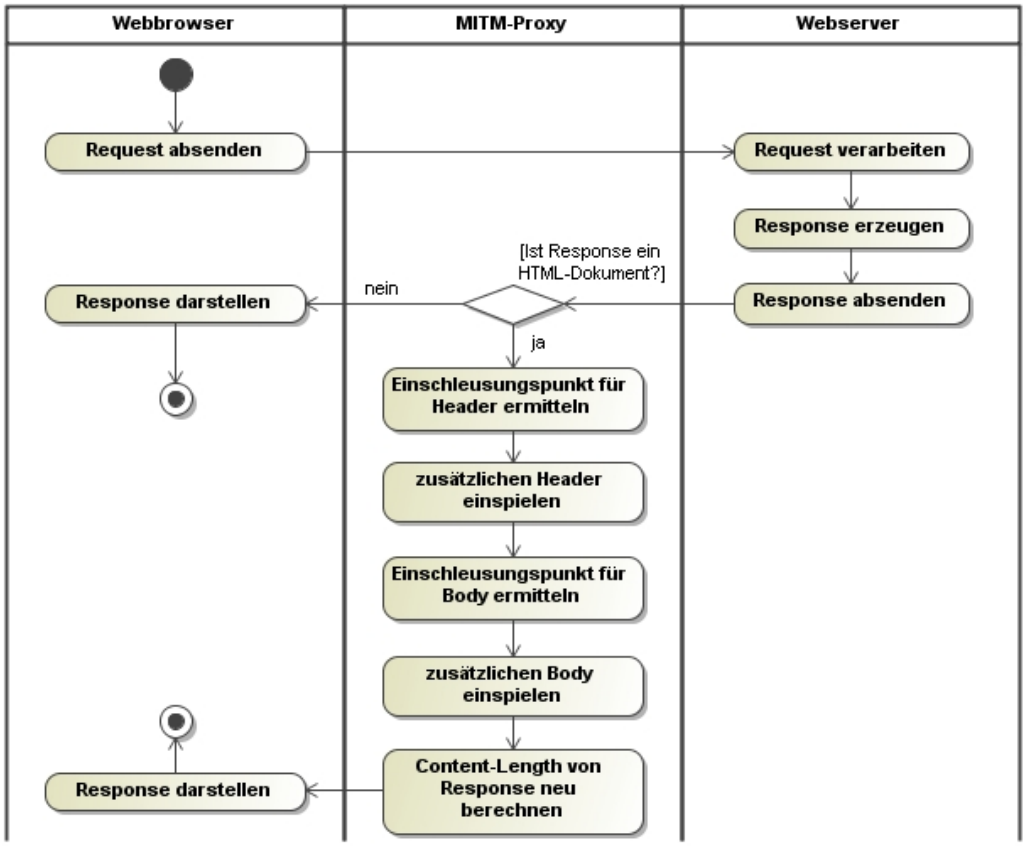


Abbildung 4.4.: Einschleusen von Code in die Webseite

Für diese Aufgabe werden die Antworten von einem *externen Request* (siehe Tabelle 4.2) entsprechend beim MITM-Proxy manipuliert. Hierbei ist jedoch zu beachten, dass das Einschleusen des eigenen Codes für die WebGUI nur in Textdokumente wie HTML erfolgt, da binäre Dateien wie Bilder im Prototypen nicht auf Schwachstellen untersucht werden sollen. Aus diesem Grund muss zunächst das Dateiformat über den Parameter *Content-type* im HTTP-Header abgefragt werden. Dieser liefert bei einem HTML-Dokument den Wert *text/html*. Eine detaillierte Auflistung der wichtigsten Datentypen für den Parameter Content-Type ist unter [W3Cb] zu finden.

Im nächsten Schritt kann nun der eigene Code in den Quelltext des Responses eingeschleust werden. Dieser Code spiegelt sich in zwei Teile wieder - einen zusätzlicher Header und eine Erweiterung des Bodys. Der zusätzliche Header-Code weist den Webbrowser an, die Dateien für die WebGUI nachzuladen, wohingegen die Erweiterung im Body den Inhalt der nachgeladenen Dateien entsprechend darstellt. Hierzu wurden die jeweiligen Teile in zwei separate Dateien (*w3af_mainHeader.html* und *w3af_mainBody.html*) ausgelagert, um ein späteres Ändern der WebGUI zu ermöglichen. Im letzten Schritt muss der Parameter *Content-length* im HTTP-Header des Responses neu berechnet werden. Dieses ist notwendig, da durch die Änderungen am Quelltext der Webseite die Bytegröße verändert wurde.
Das folgende Beispiel zeigt ein HTML-Dokument mit den entsprechenden Codestellen, die für das Einschleusen des eigenen Codes benötigt werden.

Beispiel einer HTML-Datei mit allen notwendigen Einschleusungsstellen

```
<html>
  <head>
    ...
  </head> <!-- Injizierungspunkt für Header gefunden
              - integriere Inhalt der Datei w3af_mainHeader.html -->
  <body>  <!-- Injizierungspunkt für Body gefunden
              - integriere Inhalt der Datei w3af_mainBody.html -->
    ...
  </body>
</html>
```

Da jedoch eine HTML-Datei nicht zwingend sich an die standardisierte Form halten muss, kann es unter Umständen vorkommen, dass nicht alle entsprechenden Stellen, zum Einschleusen des Codes, im Response ermittelt werden können. So kann es zum Beispiel vorkommen, dass ein HTML-Dokument kein Header hat und dadurch ein Teil des einzuschleusenden Codes nicht in die Seite eingebracht werden kann. Dieses kann unter Umständen dazu führen, dass die WebGUI im Browser nicht korrekt dargestellt werden kann. Ein solcher Fall wird bei der Implementierung des Prototypen zur Zeit nicht berücksichtigt, da das Lösen dieses Problems eine genauere Analyse des Aufbaus der originalen Webseite voraussetzt und darüber hinaus ein solches HTML-Dokument nicht konform ist.

4.2.2. Darstellung der WebGUI

Nachdem im ersten Schritt der Grundbaustein in Form des eigenen Codes in die originale Webseite eingeschleust wurde, kann nun im zweiten Schritt die WebGUI selbstständig die restlichen Teile zur Darstellung des Web-Frontends nachladen. Hierfür sendet die WebGUI für jede einzelne nachzuladende Datei eine entsprechende GUI-URL an den MITM-Proxy, welcher den jeweiligen Teil für die WebGUI bereitstellt. Das folgende Beispiel zeigt einen Ausschnitt der nachzuladenden Dateien.

Auszug aus der Datei w3af_mainHeader.html zum Nachladen weiterer Dateien.

```
<script language="JavaScript" type="text/javascript"
  src="/analyzeApp/gui/w3af_layout.html"></script>

<script language="JavaScript" type="text/javascript"
  src="/analyzeApp/gui/w3af_controlFrame.js"></script>
```

Die daraus resultierende WebGUI ist in der Abbildung 4.5 im gesamten Umfang dargestellt. Bei dieser Abbildung wurde die Webseite der Suchmaschine Google mit zusätzlichen injizierten Code aufgerufen, welcher vom MITM-Proxy geladen wurde.
Der Aufbau der WebGUI ist hierbei in einzelne Funktionsgruppen unterteilt. Die einzelnen Funktionsgruppen im Menü werden später in diesem Kapitel in ihrem Aufbau und ihrer Funktionsweise genauer beschrieben. Über den Menüpunkt *Analyze* werden alle sichtbaren und unsichtbaren Form-Elemente der aktuellen Webseite ausgegeben. Der Menüpunkt *Request* (siehe 4.5) ermöglicht dem Benutzer eine Verwaltung der abgespeicherten Requests, welche für die automatische Schwachstellenanalyse benötigt werden. Der Menüpunkt *Plugin-Manager* erlaubt es dem Benutzer sämtliche Funktionen, die der Plugin-Manager (siehe 4.6) bereitstellt, zu verwenden. Unter dem Menüpunkt *Other* kann der Benutzer sämtliche Konfigurationseinstellungen (siehe 4.3) vornehmen, sowie eine automatische Analyse der Webseite starten (siehe 4.7).

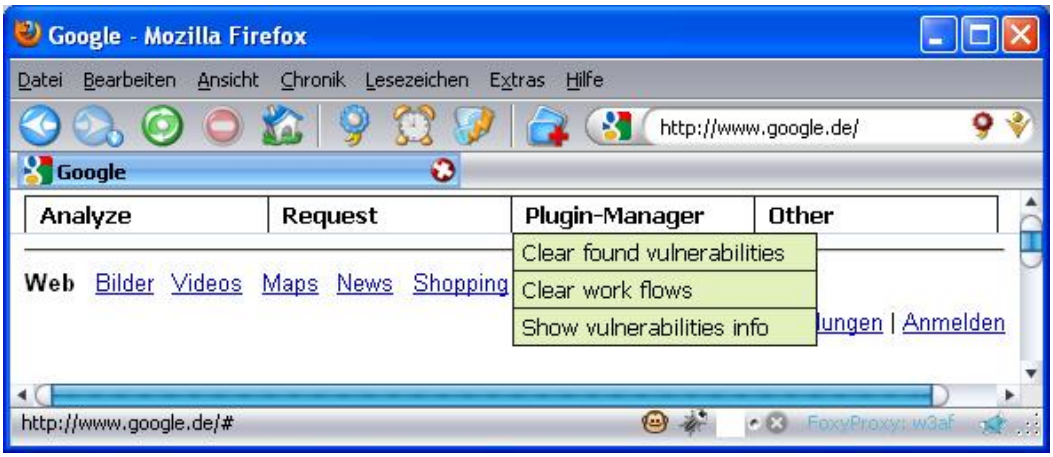


Abbildung 4.5.: Darstellung der WebGUI.

4.3. Konfigurationsverwaltung

Das Speichern und Auslesen der Einstellungen ist eine wichtige Funktionalität für den zu entwickelnden Prototypen. Anhand dieser Möglichkeit ist der Benutzer zum Beispiel in der Lage, Workflows von Requests aufzunehmen und automatisch abzuspielen, die er zu einem späteren Zeitpunkt auf Schwachstellen untersuchen will. Aber auch allgemeine Einstellungen wie das Aktivieren bestimmter Plugins für die Schwachstellenanalyse können so vorgenommen werden. Da jedoch HTTP zustandslos (stateless) ist, gehen alle durch den Benutzer vorgenommene Änderungen an der Konfiguration auf der Client-Seite bei einem Neuladen einer Webseite verloren. Aus diesem Grund ist es notwendig, dass die Einstellungen auf der Seite des MITM-Proxys vorgenommen werden, da Änderungen an dieser Stelle dauerhaft gespeichert sind.
Für diese Aufgabe wurde ein SessionSaver entwickelt, bei dem die Konstanz der Konfiguration erhalten bleibt. Dieser SessionSaver ist, wie in Abbildung 3.7 dargestellt, logisch gesehen ein Teil des MITM-Proxys und dient neben dem Speichern der Einstellungen auch dem Speichern der von der Plugin-Architektur gefundenen Schwachstellen. Im Folgenden

wird der Aufbau des SessionSavers näher beschrieben. Danach wird gezeigt, wie Änderungen permanent im SessionSaver vorgenommen werden und wie die WebGUI auf die aktuellen Einstellungen des SessionSavers zugreifen kann.

4.3.1. Aufbau des SessionSavers

Der SessionSaver ist in seiner Grundfunktion eine Javascript-Datei. Auf diese Weise kann die WebGUI später auf einzelne Einstellungen problemlos über Javascript-Funktionalitäten auf den SessionSaver zugreifen. Um ein späteres Auslesen der einzelnen Konfigurationseinstellungen zu gewährleisten, sind alle Informationen im SessionSaver in ein für Javascript lesbares Format (*JSON*[1]) abgespeichert. Nachfolgend werden alle Parameter mit ihrer entsprechenden Funktionalität aufgelistet.

Parmeter: automateTest

Dieser Parameter beinhaltet alle Informationen, die für ein automatisches Testen einzelner Workflows von Webanwendungen benötigt werden.

Typ: Javascript Objekt

Struktur der einzelnen Elemente:

automate	Gibt an, ob ein automatisches Testen durchgeführt werden soll oder nicht.
process	Über diesen Parameter kann der aktuelle Status zum automatischen Testens definiert werden. (Gültige Werte sind: prepare=Initialisere Testvorbereitungen; play, update, pause, next, stop)
startURL	Definiert den Anfang des Workflows zum automatischen Testen.
endURL	Definiert das Ende des Workflows zum automatischen Testen.
manipulateParams	Gibt an, welche Parameter vom jeweiligen Request mit den Patterns (cheats) manipuliert werden sollen.
otherParams	Beinhaltet alle anderen Parameter, die nicht manipuliert werden sollen.
cheats	Beinhaltet alle übergebenen Patterns, die zur Manipulation der einzelnen Parameter verwendet werden.
cid	Bestimmt den aktuellen Eintrag aus der internen Pattern-Liste, welche im Workflow verwendet wird.

[1] JavaScript Object Notation ist ein kompaktes Format, das komplexe Objekte in einer für Javascript lesbaren Form abspeichert.

Parmeter: foundVulnerabilities

Der Parameter foundVulnerabilities wird vom Plugin-Manager gefüllt und fasst alle Ergebnisse zu den gefundenen Schwachstellen in einer Webanwendung zusammen. Darüber hinaus zeigt er alle gefundenen Merkmale zwischen den beiden Requests bei denen eine Gemeinsamkeit gefunden wurde.

Typ: Javascript Objektliste

Struktur der einzelnen Einträge:

type	Gibt an, welches Plugin aus dem Plugin-Manager die Schwachstelle gefunden hat.
shortMessage	Kurze Zusammenfassung der gefundenen Schwachstelle.
message	Detaillierte Zusammenfassung der gefundenen Schwachstelle.
reqList	Workflow mit den verwundbaren Parametern, bei dem die Schwachstelle gefunden wurde.

Parameter: settings

Dieser Parameter wird für die Speicherung aller notwendigen Einstellungen benötigt. Neben den Einstellungen für die einzelnen Plugin-Konfigurationen können ebenfalls die Arten der Requests angegeben werden, die bei der Analyse der Webanwendung automatisch erkannt (intercepted) werden sollen. Das Ändern aller Einstellungen erfolgt über die WebGUI.

Typ: Javascript Objekt

Struktur der einzelnen Elemente:

detection	Objekt mit allen verfügbaren Plugins und ihren jeweiligen Konfigurationen. Zusätzlich wird hier angegeben, ob das Plugin aktiviert oder deaktiviert ist.
discover	Objekt, das angibt, was für Request-Arten (z.B. XMLHTTPRequest) intercepted werden sollen.

Parmeter: requestData

Der Parameter requestData dient zum Abspeichern von Requests und den dazugehörigen Informationen. Mit dessen Hilfe kann zu einem späteren Zeitpunkt ein automatisiertes Testen auf Schwachstellen durchgeführt werden. Darüber hinaus bietet der Parameter ebenfalls die Möglichkeit, bestimmte Aktionen für einen externen Request automatisch auszuführen, wenn dieser Request von der WebGUI abgesetzt wird.

Typ: Javascript Objektliste

Struktur der einzelnen Einträge:

id	Bezeichnet einen eindeutigen Identifier für jeden gespeicherten Request.
type	Gibt an, wie der Request von der Webanwendung abgesetzt wurde (zum Beispiel: XMLHttpRequest, Submit).
method	Gibt an, ob es sich um einen GET- oder POST-Request handelt.
fromURL	Enthält die URL, von welcher der zu speichernde Request abgesetzt wurde.
toURL	Enthält die Ziel-URL vom abgesetzten Request.
paramValue	Speichert alle übergebenen Parameter des Requests.
choiceEnabled	Bestimmt, ob der Request beim nächsten Aufruf gesondert behandelt werden soll.
choice	Bestimmt, wie der Request beim nächsten Aufruf behandelt werden soll (zum Beispiel: Verwerfen, Manipulieren, sofort absenden). Wird jedoch nur verwendet, wenn *choiceEnabled* aktiviert ist.
recordEnabled	Bestimmt, ob der zu speichernde Request für ein späteres automatisches Testen verwendet wird.

4.3.2. Speichern im SessionSaver

Um dauerhaft Informationen für das spätere Testen speichern zu können, sendet die WebGUI über einer Steuerkanal einen speziellen Request mit einen internen Befehl an den MITM-Proxy. Dieser Request dient nur der Übermittlung von Befehlen an den MITM-Proxy und wird nicht an den Webserver der Webanwendung weitergeleitet. Nach dem der Proxy diesen Request empfangen hat, wird über den mitgelieferten Befehl auf eine bestimmte Funktionalität beim MITM-Proxy zugegriffen. Ein derartiger Request mit einem Befehl ist im unteren Beispiel abgebildet.

Beispiel einer Control-URL zum Löschen aller aufgenommenen Requests im SessionSaver

```
http://myserver.com/analyzeApp/control/?delrec
```

Anhand dieses Verfahrens können so über die WebGUI alle Einstellungen im SessionSaver bequem vorgenommen werden. Da jedoch standardmäßig jeder abgehende Request vom Browser eine neue Webseite anfordert, wird nach der Verarbeitung einer solchen Control-URL eine Antwort an den Webbrowser gesendet, die den Statuscode 204 zurückliefert. Dieser

Code informiert den Browser, dass die entsprechende Antwortseite keinen Inhalt enthält, so dass der Browser die aktuelle Webseite beibehält. Eine ausführliche Liste mit allen HTTP-Returncodes findet sich unter [W3Cd]. Zusammenfassend stellt die Abbildung 4.6 den Ablauf für das Speichern in Form eines Aktivitätsdiagramms noch einmal dar.

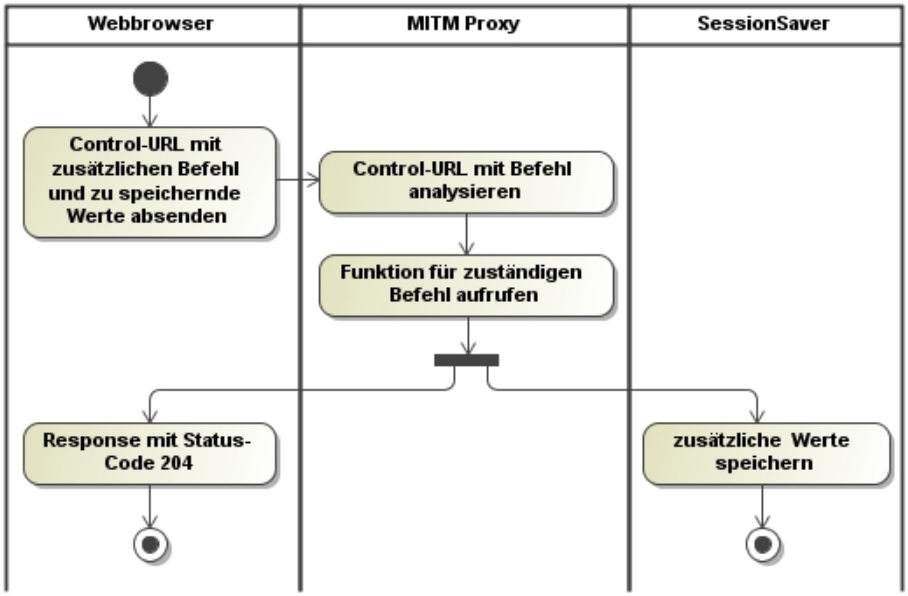


Abbildung 4.6.: Ablauf beim Speichern von Einstellungen.

4.3.3. Auslesen des SessionSavers

Das Auslesen des SessionSavers stellt im Gegensatz zur Aktualisierung der Einstellungen einen geringeren Aufwand dar. Für diese Aufgabe muss die WebGUI lediglich eine spezielle Anfrage an den MITM-Proxy über den Steuerkanal senden, der den SessionSaver als Javascript-Datei an die WebGUI sendet. Das folgende Beispiel zeigt hierfür die notwendige Anfrage von der WebGUI an den MITM-Proxy.

Request zum Auslesen des SessionSavers aus derWebGUI

```
http://myserver.com/analyzeApp/gui/w3af_state.js
```

4.4. Umsetzen der Analyse des DOMs

Die automatische Analyse einer Webseite auf ihre möglichen Eingabefelder ist ein wichtiger Punkt, mit dem sich ein Benutzer einen Überblick über eine Webseite verschaffen kann. Des Weiteren können die hierbei gewonnenen Erkenntnisse später beim automatischen Testen verwendet werden, welches im Kapitel 4.7 beschrieben wird. Hierbei wurden die Erkenntnisse aus dem Kapitel 3.1 übernommen, die eine client-seitige Analyse der Webseite mittels *Javascript* bevorzugen. Der folgende Codeausschnitt zeigt anhand eines vereinfachten Beispiels, wie unter anderem einzelne Teile eines *Formulars* mit seinen dazugehörigen Eingabefeldern, aus einer Webseite extrahiert werden können.

Automatisches Auslesen von HTML-Formularen.

```
function w3af_printForm(formObject, ...){
  // get attributes for current form
  var faction=_checkString(formObject.getAttribute("action"))
  var fmethod=_checkString(formObject.getAttribute("method"))
  var fname=formObject.getAttribute("name");

  // read input tags
  for(var i=0;i<formObject.getElementsByTagName("input").
    length;i++){
    inputNode=formObject.getElementsByTagName("input")[i]
    var iname=_checkString(inputNode.getAttribute("name"))
    var itype=_checkString(inputNode.getAttribute("type"))
    var ivalue=_checkString(inputNode.value)

  // print input tag
  ...
  }
}
```

Zur Analyse der Webseite wird hierbei die Funktion *w3af_printForm* aufgerufen, die unter anderem den Parameter *formObject* entgegennimmt, welches ein Form-Element einer Webseite repräsentiert. Mittels diesem Form-Element können nun, im ersten Schritt, die einzelnen Attribute wie *action* oder *method* eines Formulars extrahiert werden (Zeile 2 bis 5). Die Beschreibungen zu den einzelnen Attributen eines Form-Tags ist unter [W3Ca] zu finden. Nach dem Ermitteln der einzelnen Form-Attribute werden im nächsten Schritt alle dazugehörigen *Input-Elemente* mit ihren einzelnen Attributen extrahiert (Zeile 7 bis 13). Diese Elemente spiegeln die eigentlichen Eingabemöglichkeiten eines HTML-Formulars wieder. Im letzten Schritt werden alle gesammelten Informationen des Form-Elements mit den dazugehörigen Eingabefeldern in die originale Webseite als neues, weiteres Formular integriert (Zeile 15). Hierbei werden alle Eingabefelder, inklusive der *Hidden-Elemente* so dargestellt, dass der Benutzer die einzelnen Werte der Eingabefelder beliebig ändern kann. Das Ergebnis einer solchen Analyse zeigt die Abbildung 4.7.

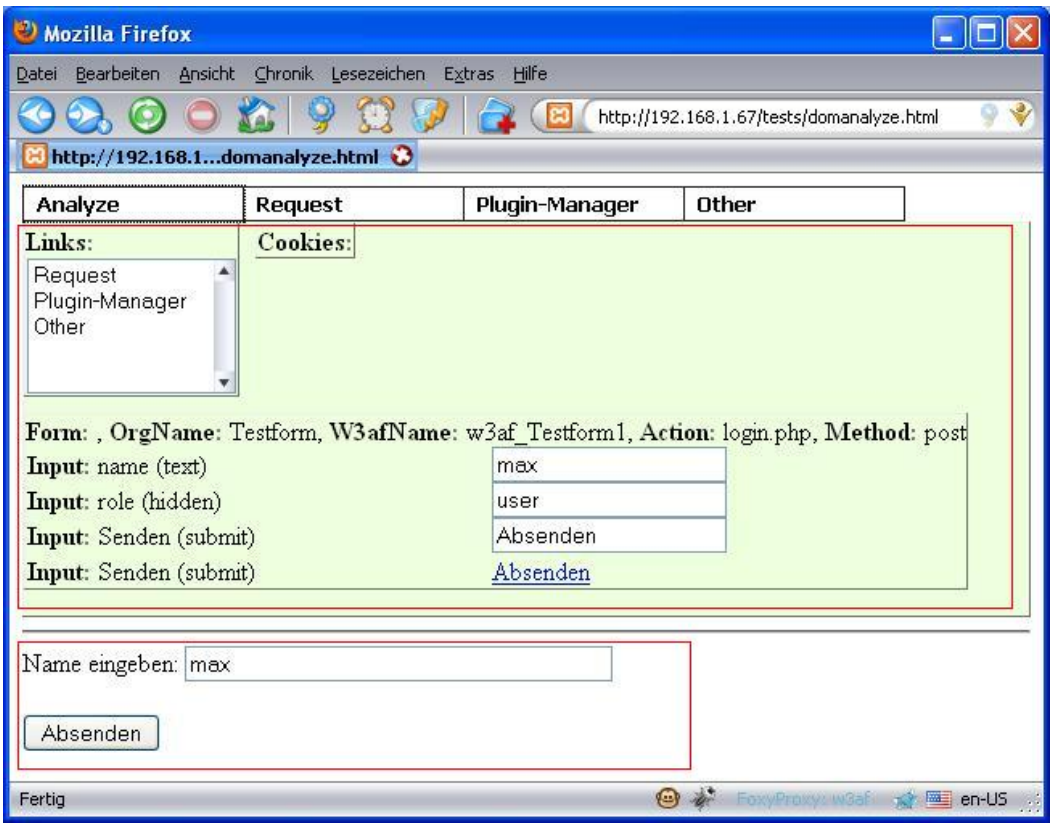


Abbildung 4.7.: Darstellung des AutomateTest-Window.

Im unteren Teil des Bildes ist die originale Webseite zu erkennen, welche aus einem HTML-Formular besteht. Nach dem *Drücken* des Menüknopfes *Analyze* im oberen Teil der Webseite, wird die Webseite auf spezielle Informationen wie HTML-Formulare, eingebettete Links oder Cookies mittels Javascript untersucht. Hierbei ist zu erkennen, dass zusätzlich zu den beiden im unteren HTML-Formular befindlichen Eingabefeldern (eine Textbox und ein Button) ein *Hidden-Feld* erkannt wurde.

4.5. Integration des Interceptors

Wie bereits im Kapitel 3.2 gezeigt, wird ein Interceptor in die neue Architektur integriert, der es ermöglicht, alle abgehenden Requests einer Webanwendung nachträglich zu manipulieren. Der Interceptor besteht hierbei aus einem *Request-Handler* und einem *Interception-Window*. Die Aufgabe des Request-Handlers ist das Abfangen des von der Webanwendung abgesetzten Requests, wohingegen das Interception-Window den abgefangenen Request, für eine Manipulation durch den Benutzer, entsprechend darstellt. Die Abbildung 4.8 verdeutlicht noch einmal die Funktionsweise des Interceptors.
Im ersten Schritt wird ein von der WebGUI *externer Request* (siehe Tabelle 4.2) *req_org* erzeugt und an den Webserver gesendet. Beim Absenden dieses Requests wird dieser jedoch zunächst mit Hilfe des *Request-Handlers* abgefangen. Dieser liest aus dem Request alle relevanten Informationen wie zum Beispiel zusätzliche übergebene Parameter aus. Danach werden diese Informationen in die Variable *args_org* gespeichert und an das *Interception-*

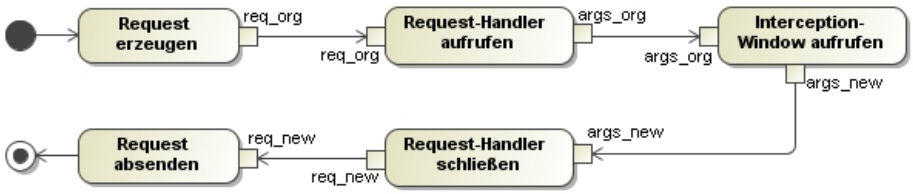


Abbildung 4.8.: Allgemeine Funktionsweise des Interceptors

Window gesendet. Hier kann nun der Benutzer etwaige Änderungen an den einzelnen Daten des Requests vornehmen. Im Anschluss werden die abgeänderten Informationen *args_new* an den Request-Handler zurückgegeben, welcher im letzten Schritt den neu erstellten Request *req_new* zum Webserver weiterleitet.

4.5.1. Aufbau eines Request-Handlers

Wie bereits erwähnt, dient der Request-Handler dem Abfangen eines abgehenden Requests. Da jedoch der Request direkt auf der Client-Seite abgefangen wird, muss für jede mögliche Request-Funktion (zum Beispiel Submit, XMLHttpRequest, scriptSrc), ein eigener Handler entwickelt werden. Aus diesem Grund wird eine einheitliche Struktur entwickelt, die jeder Handler einzuhalten hat. So kann später das *Interception-Window*, welches die entsprechenden Request-Daten vom Handler entgegen nimmt, ohne zusätzlichen Aufwand die Daten vom Request verarbeiten. Das folgende Beispiel zeigt den Aufbau eines Request-Handlers.

Handler zum Manipulieren von Requests bei der Javascript Submit-Methode.

```
var newSubmit=function(){
  var keepChoice=false
  var args=new Array()

  args[0]="REQUEST"
  args[1]=this.method    // GET or POST request
  args[2]="Submit"       // type of handler
  args[3]=this.action    // target url of this request
  args[4]=w3af_getArrayOfFormElements(this) // array of parameters
                                            // and values
  args[5]=document.location.href  // start url from request

  // check, if we saved our choice for this request
  for(var i=0;i<requestData.length;i++){
    ...
  }

  if(keepChoice == false){
    retVal = window.showModalDialog("analyzeApp/gui/w3af_intercept.html",
        args, w3af_paramsAjaxWindow)
    ...

    if(args[0] == "SEND"){
      this.oldSubmit()
```

```
25      }
        else if(args[0] == "EDIT"){
          // copy new request values to original request
          ...
          this.oldSubmit()
30      }
        else{
          // drop request
        }
      }
35  }

...
HTMLFormElement.prototype.oldSubmit=HTMLFormElement.prototype.submit;
HTMLFormElement.prototype.submit=newSubmit;
```

Der im obigen Teil abgebildete Javascript-Code zeigt anhand des Beispiels für die Javascript *Submit-Funktion*, wie ein Handler zum Abfangen eines Requests aufgebaut ist. Zum Transportieren der Daten vom Request-Handler zum Interception-Window dient hierbei das Array *args*, welches zunächst mit den entsprechenden Werten des aktuellen Requests gefüllt wird (Zeile 3 bis 11). Im nächsten Schritt wird überprüft, ob der Benutzer bereits diesen Request mit einer automatischen Antwort („Request unverändert senden“„Request manipulieren“, „Request verwerfen“) versehen hat (Zeile 13 bis 16). Dieses ermöglicht dem Benutzer nur diejenigen Requests zu bearbeiten, die für die Analyse der Webanwendung entscheidend sind. Wenn für diesen Request keine Antwort hinterlegt wurde, oder der Benutzer für diesen Request die Manipulation ausgewählt hat, wird die eigentliche Arbeit des Handlers ausgeführt. Hierzu wird mit Hilfe der Funktion *window.showModalDialog* (siehe [Mozb]) das Manipulations-Fenster geöffnet, welches als zusätzlichen Parameter das, am Anfang erzeugte, Array *args* übernimmt (Zeile 19 bis 20). Die dort vorgenommenen Änderungen an dem Request, sind in dem Rückgabewert *retVal* von der Funktion *window.showModalDialog* gespeichert.
Diese werden im nächsten Schritt in die Variable *args* kopiert, welche dann entsprechend interpretiert werden (Zeile 23 bis 33). Hierbei spiegelt der erste Wert des Arrays *args* die vorgenommene Auswahl des Benutzers wieder, wie mit dem Request zu verfahren ist. Bei dem Wert *SEND* wird dieser Request unverändert an den Webserver gesendet, wohin gegen bei dem Wert *EDIT* der Request mit den im Interception-Window eingegebenen Daten zuvor noch manipuliert wird. Beim Auftreten eines anderen Wertes (zum Beispiel *DROP*) wird dieser Request vom Request-Handler entsprechend verworfen.
Der letzte Teil (Zeile 38 bis 39) des oberen Code-Auszugs dient zur Sicherung der originalen Javascript Submit-Funktion und dem Neu-Definieren dieser Funktion, so dass diese Funktion vom Request-Handler entsprechend abgefangen werden kann.

4.5.2. Darstellung des Requests

Bei der Darstellung des Requests kann der Benutzer letztendlich alle Änderungen für einen Request vornehmen. Die Abbildung 4.9 zeigt die entsprechende Darstellung des Interception-Windows.

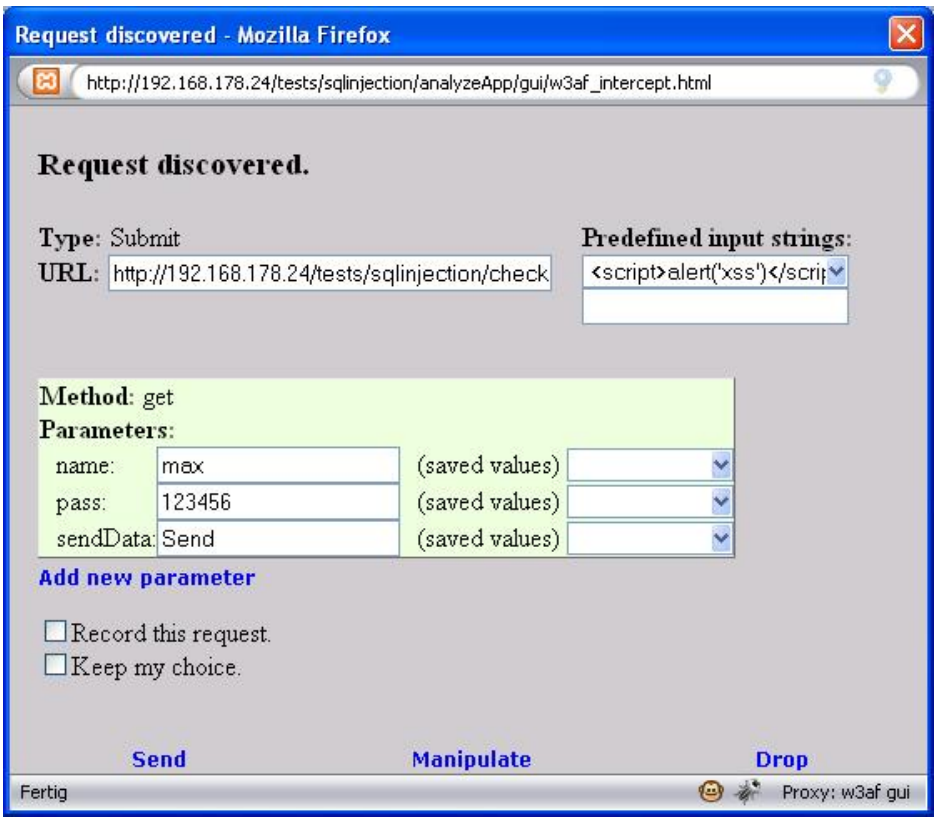


Abbildung 4.9.: Darstellung des Interceptor-Windows

Das Interception-Window zeigt alle Informationen über den abgehenden Request an. Im oberen Teil des Fensters wird neben dem Typ des Requests (zum Beispiel Submit oder XMLHttpRequest) zusätzlich die URL angezeigt, welche auch abgeändert werden kann. Des Weiteren zeigt das Interception-Window bereits vordefinierte Zeichenketten an, die für typische Analysen von Schwachstellen in Webanwendungen verwendet werden. Diese kann der Benutzer in die jeweiligen Parameterfelder des Requests verwenden. Der Kern des Fensters ist die Darstellung der einzelnen Parameternamen mit ihren entsprechenden Werten. Diese kann der Benutzer nach Belieben verändern. Außerdem können zusätzlich neue Parameter für den Request definiert werden. Das Optionsfeld „Record this request.“ermöglicht den Request für ein späteres automatisiertes Testen zu speichern. Auf diese Weise können einzelne Workflows einer Webanwendung dargestellt werden. Die Option „Keep my choice.“speichert die Antwort (Send, Manipulate, Drop) des Requests. So kann der Benutzer definieren, welche Requests er für die Zukunft bearbeiten möchte.
Am unteren Teil des Interception-Windows bestimmt der Benutzer, wie der Request letztendlich verwendet wird (unverändert senden, verändert senden, verwerfen).

4.6. Umsetzen des Plugin-Managers

Wie bereits im Kapitel 3.4 aufgezeigt, wurde im zu entwickelnden Prototypen ein Plugin-Manager integriert, der es erlaubt, Schwachstellen in Webanwendungen über mehrere Webseiten zu finden. Hierzu wurden die einzelnen Techniken zum Finden bestimmter Schwach-

stellen in einzelne Plugins integriert, die dann über den Plugin-Manager korrekt in den Prototypen integriert werden.

4.6.1. Aufbau des Plugin-Managers

Der Plugin-Manager besteht aus einem Paket *AttackHandler*, welches alle benötigten Klassen und Objekte zusammenfässt. Wie die Abbildung 4.10 zeigt, beinhaltet dieses Paket die Oberklasse *AttackHandler* und die einzelnen Plugins *SqlHandler* und *XssHandler*, die von der Oberklasse AttackHandler abgeleitet sind. Des Weiteren beinhaltet der Plugin-Manager zusätzlich die Klassen für *RequestToken-Objekte* und *Vulnerability-Objekte*, die für das Speichern von einzelnen Informationen benötigt werden.

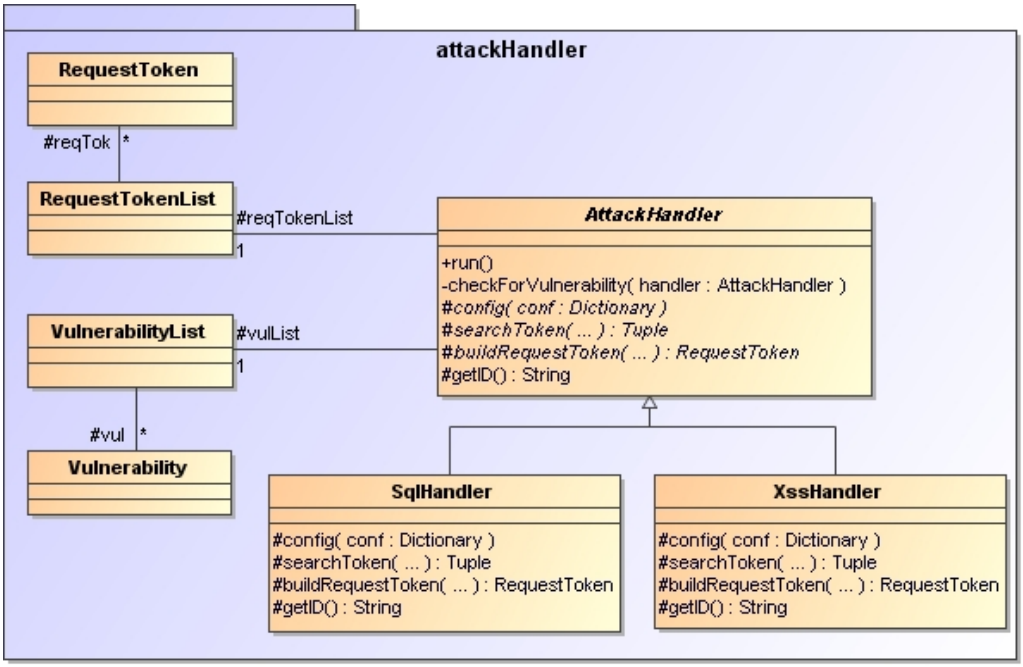


Abbildung 4.10.: Klassendiagramm der Plugin-Architektur

Oberklasse: AttackHandler
Die Oberklasse AttackHandler ist die zentrale Klasse des Plugin-Managers. Neben der Verwaltung der RequestToken-Liste und der Vulnerability-Liste beinhaltet sie unter anderem die Methoden *run* und *checkForVulnerability*, die für einen korrekten Ablauf der einzelnen Plugins zuständig sind. Des Weiteren definiert diese Klasse die abstrakten Methoden *buildRequestToken, searchToken, getID* und *config*, welche von jedem zu implementierenden Plugin überschrieben werden.

Unterklassen: SqlHandler und XssHandler
Die Klassen *SqlHandler* und *XssHandler* sind Plugins zum Finden von Sicherheitslücken in Webanwendungen. Um eine einheitliche Struktur aller Plugins zu gewährleisten, implementieren diese die abstrakte Klasse *AttackHandler*. Von dieser Klasse überschreiben die Plugins die Methoden *buildRequestToken, searchToken, getID* und *config*.

Klassen: RequestToken und RequestTokenList
Die Klassen RequestToken und RequestTokenList werden für das Abspeichern der Requests mit ihren jeweiligen Merkmalen (Token) verwendet. Da jedes Plugin ein anderes Merkmal zum Finden einer Schwachstelle verwendet, kann so gewährleistet werden, dass alle, durch die Plugins erzeugten, RequestToken-Objekte den gleichen Aufbau haben, was ein späteres Vergleichen der einzelnen Objekte ermöglicht.

Klassen: Vulnerability und VulnerabilityList
Die Klassen Vulnerability und VulnerabilityList dienen dem Abspeichern von gefundenen Zusammenhängen (Gemeinsamkeiten) zwischen den einzelnen RequestToken-Objekten. So speichern sie unter anderem Informationen, wo ein Zusammenhang zwischen den einzelnen RequestToken-Objekten besteht. Diese Klassen liefern also das Ergebnis über die gefundenen Gemeinsamkeiten.

Methode: run
Die Methode *run* ist in der Klasse AttackHandler implementiert und wird direkt vom MITM-Proxy aufgerufen. Sie dient der Erzeugung eines neuen RequestToken-Objekts mittels der Methode *buildRequestToken*, welches dann in die RequestToken-Liste gespeichert wird. Zusätzlich dazu ruft sie die Methode *checkForVulnerability* auf, welche nach Gemeinsamkeiten in der RequestToken-Liste sucht.

Methode: checkForVulnerability
Die Methode *checkForVulnerability* ist eine interne Funktion der Klasse AttackHandler. Ihre Aufgabe besteht in dem Durchsuchen der globalen RequestToken-Liste nach Gemeinsamkeiten. Dazu ruft sie von jedem integrierten Plugin die Methode *searchToken* auf, welche alle Gemeinsamkeiten zwischen zwei RequestToken-Objekten ermittelt. Die gefundenen Gemeinsamkeiten werden im Anschluss in die globale Vulnerability-Liste gespeichert.

Methode: config
Die abstrakte Methode *config* der Klasse AttackHandler wird von jeder Unterklasse (Plugin) überschrieben. Mit ihrer Hilfe können alle Einstellungen von den jeweiligen Plugins, die zum Ermitteln einer Schwachstelle benötigt werden, vorgenommen werden. Dabei werden alle Einstellungen extern über die WebGUI getätigt. Auf diese Weise hat der Benutzer eine zentrale Stelle, bei der er die Verwaltung der Plugins steuern und zusätzlich die Webanwendung testen kann.

Methode: getID
Die abstrakte Methode *getID* dient der eindeutigen Identifizierung eines Plugins im PluginManager. Sie ist in der Klasse AttackHandler definiert und wird von jeder Unterklasse (Plugin) überschrieben. Als Ergebnis liefert diese Methode eine Zeichenkette zurück. Diese Zeichenkette (Name) wird später bei der Identifizierung der gefundenen Schwachstellen verwendet. So kann der Benutzer erkennen, welches Plugin die gefundene Schwachstelle entdeckt hat.

Methode: buildRequestToken
Die Methode *buildRequestToken* ist in der Klasse AttackHandler als abstrakt gekennzeichnet. Diese Methode wird von jedem zu implementierenden Plugin verwendet, um ein RequestToken-Objekt zu erzeugen, welches dann durch die *run-Methode* in der globalen RequestToken-Liste verwaltet werden kann. Um ein RequestToken-Objekt zu erzeugen wird hierzu ein spezielles Merkmal verwendet(zum Beispiel der Response einer Webseite) , welches für eine bestimmte Kategorie einer Schwachstelle typisch ist.

Methode: searchToken
Die Methode *searchToken* ist in der Klasse AttackHandler als abstrakt deklariert. Sie wird von der Methode *checkForVulnerability* aufgerufen und dient zum Ermitteln von Gemeinsamkeiten zwischen zwei RequestsToken-Objekten.

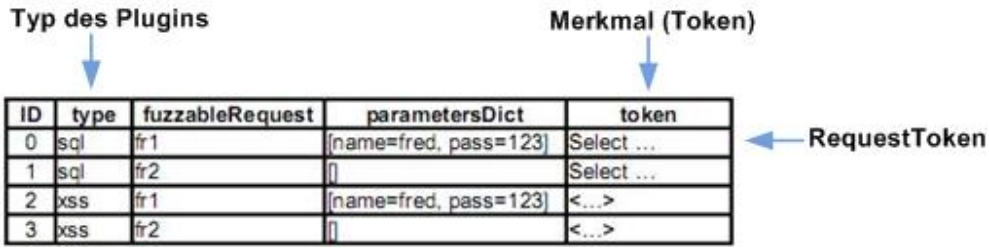
4.6.2. Aufbau eines RequestToken-Objekts

Wie bereits beschrieben, werden zum späteren Finden von Zusammenhängen die einzelnen Requests als Objekte der Klasse *RequestToken* abgespeichert. Diese RequestToken-Objekte beinhalten alle Informationen eines Requests und speichern zusätzlich ein bestimmtes Merkmal (Token). Ein solches Merkmal ist zum Beispiel der Response einer Webseite oder die einzelnen SQL-Statements bei Datenbankabfragen. So können zu einem späteren Zeitpunkt die einzelnen RequestToken-Objekte komfortabel miteinander verglichen werden. Im Folgenden werden die einzelnen Parameter eines RequestToken-Objekts beschrieben.

id	Bezeichnet ein Identifier, der jedes RequestToken-Objekt eindeutig kennzeichnet.
type	Der Parameter *type* stellt den Typ eines RequestToken-Objekts dar. Hierbei spezifiziert jedes im Plugin-Manager verwendete Plugin einen eigenen Typ.
fuzzableRequest	Objekt, welches alle Informationen, wie Parameter oder die URL eines Requests beinhaltet. Dieses, von den Entwicklern des w3af-Frameworks definierte Objekt, speichert alle Informationen in ein URL-kodiertes Format.
parametersDict	Beinhaltet eine Liste mit allen übergebenen Parametern und den entsprechenden Werten des Requests. Hierbei ist zu beachten, dass alle URL-kodierten Parameter und Werte, wie sie im Parameter *fuzzableRequest* hinterlegt sind, in dekodierter Form abgespeichert werden.
token	Bezeichnet ein Merkmal welches zum Finden der Gemeinsamkeiten dient.

Tabelle 4.3.: Beschreibung der einzelnen Parameter eines RequestToken-Objekts

Um die oben beschriebenen RequestToken-Objekte später verwalten zu können, wurde eine Klasse *RequestTokenList* entwickelt, die alle einzelnen Objekte in einer Liste zusammenfasst.



ID	type	fuzzableRequest	parametersDict	token
0	sql	fr1	[name=fred, pass=123]	Select ...
1	sql	fr2	[]	Select ...
2	xss	fr1	[name=fred, pass=123]	<...>
3	xss	fr2	[]	<...>

Abbildung 4.11.: Beispiel einer RequestToken-Liste

Die Abbildung 4.11 zeigt ein Beispiel einer RequestToken-Liste. Sie beinhaltet insgesamt vier Einträge, die jeweils ein RequestToken-Objekt darstellen, von denen jeweils zwei Einträge vom XSS-Plugin und SQLi-Plugin sind.

4.6.3. Aufbau eines Vulnerability-Objekts

Des Weiteren wurde die Klasse *Vulnerability* eingeführt, die zum Speichern von Gemeinsamkeiten (Schwachstellen) zwischen zwei RequestToken-Objekten benötigt wird. Mit ihrer Hilfe können so zusätzliche Informationen über eine gefundene Gemeinsamkeit abgespeichert werden, die den Zusammenhang näher beschreiben. Im Folgenden werden die einzelnen Parameter eines Vulnerability-Objekts beschrieben.

type	Der Parameter type bestimmt, durch welches Plugin die Schwachstelle entdeckt wurde.
shortMessage	Dieser Parameter stellt eine kurze Zusammenfassung der gefundenen Sicherheitslücke durch ein Plugin dar. Zusätzlich zu dieser Information wird hier der Workflow von den Requests angezeigt, bei denen die Gemeinsamkeiten (Schwachstellen) gefunden wurden.
infoList	Dieser Parameter enthält eine ausführliche Beschreibung der gefundenen Schwachstelle. Er zeigt auf, in welchen Teilen eines Merkmals die Gemeinsamkeiten zwischen zwei Request entdeckt wurden.
reqList	Der Parameter reqList gibt alle Request eines Workflows an, bei denen Gemeinsamkeiten untereinander gefunden wurden.

Tabelle 4.4.: Beschreibung der einzelnen Parameter eines Vulnerability-Objekts

Zur globalen Verwaltung aller gefundenen Gemeinsamkeiten werden die einzelnen Vulnerability-Objekte in eine Liste der Klasse *VulnerabilityList* gespeichert. Die Abbildung 4.12 zeigt eine solche Vulnerability-Liste mit zwei Einträgen. Diese beinhaltet jeweils eine gefundene Gemeinsamkeit zu einer XSS-Schwachstelle und einer SQL-Injection-Schwachstelle.

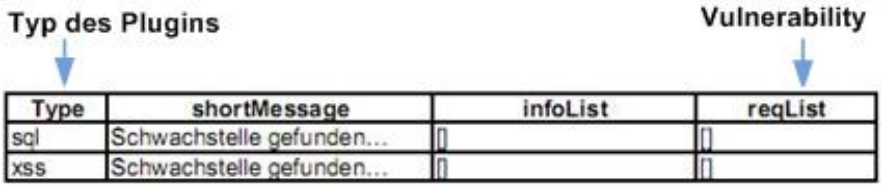


Abbildung 4.12.: Beispiel einer Vulnerability-Liste

4.6.4. Funktionsweise des Plugin-Managers

In Kapitel 3.4 wurde bereits kurz die Funktionsweise des Plugin-Managers vereinfacht beschrieben. Im weiteren soll nun aus technischer Sichtweise der Ablauf des Plugin-Managers betrachtet werden.

Die Abbildung 4.13 zeigt, wie der Ablauf des Plugin-Managers funktioniert. Beim erstmaligen Starten des Plugin-Managers werden zunächst die RequestToken-Liste (reqTokenList) und die Vulnerability-Liste (vulList) initialisiert. Im Anschluss der Initialisierung der Listen erfolgt die Konfiguration der einzelnen Plugins, welche über die WebGUI vorgenommen werden kann. So kann der Benutzer bestimmte Einstellungen, wie das Definieren eines Datenbankservers, für einzelne Plugins vornehmen.
Nach dem Beenden der einmaligen Initialisierung des Plugin-Managers, können nun Requests und ihre entsprechenden Antworten vom Webserver durch den MITM-Proxy abgefangen werden. Dieser übergibt die abgefangenen Daten (Request und Response) an die *run*-Methode des Plugin-Managers, die im weiteren Verlauf den gesamten Prozess steuert. Dabei ruft sie im ersten Schritt die Methode *buildRequestToken* auf, die jedes Plugin besitzt. Diese Methode dient der Erzeugung eines RequestToken-Objekts (reqTok) mit einem eindeutigen Merkmal. Im nächsten Schritt wird das eben erzeugte Objekt (reqTok) der am Anfang initialisierten RequestToken-Liste reqTokenList hinzugefügt. Dabei ist zu beachten, dass eine Aktualisierung des entsprechenden Listeneintrags erfolgt, wenn das Objekt reqTok bereits in der Liste reqTokenList abgespeichert wurde. Auf diese Weise werden die Einträge in der reqTokenList stets aktuell gehalten.
Im nächsten Schritt wird die Funktion *checkForVulnerability* aufgerufen, die zum Finden von Zusammenhängen (Gemeinsamkeiten) dient. Hierbei wird überprüft, ob ein Parameter eines in der reqTokenList gespeicherten Requests in einem Merkmal eines anderen abgespeicherten Requests enthalten ist. Für diese Aufgabe ruft die *checkForVulnerability*-Methode die entsprechende *searchToken*-Methode eines Plugins auf. Diese überprüft jeweils zwei RequestToken-Objekte und liefert ein entsprechendes Ergebnis über einen Zusammenhang zwischen den beiden RequestToken-Objekten zurück. Hierbei ist auch zu beachten, dass die zwei RequestToken-Objekte auch dieselben sein können. Wird hierbei ein Zusammenhang zwischen zwei RequestToken-Objekten gefunden, so wird im nächsten Schritt ein Vulnerability-Objekt erzeugt, welches die Gemeinsamkeiten der beiden RequestToken-Objekten beinhaltet, und speichert dieses in die am Anfang initialisierte List vulList. Nachdem alle RequestToken-Objekte aus der Liste reqToken-List miteinander verglichen wurden, werden im letzten Schritt die gefundenen Gemeinsamkeiten im SessionSaver abgespeichert.

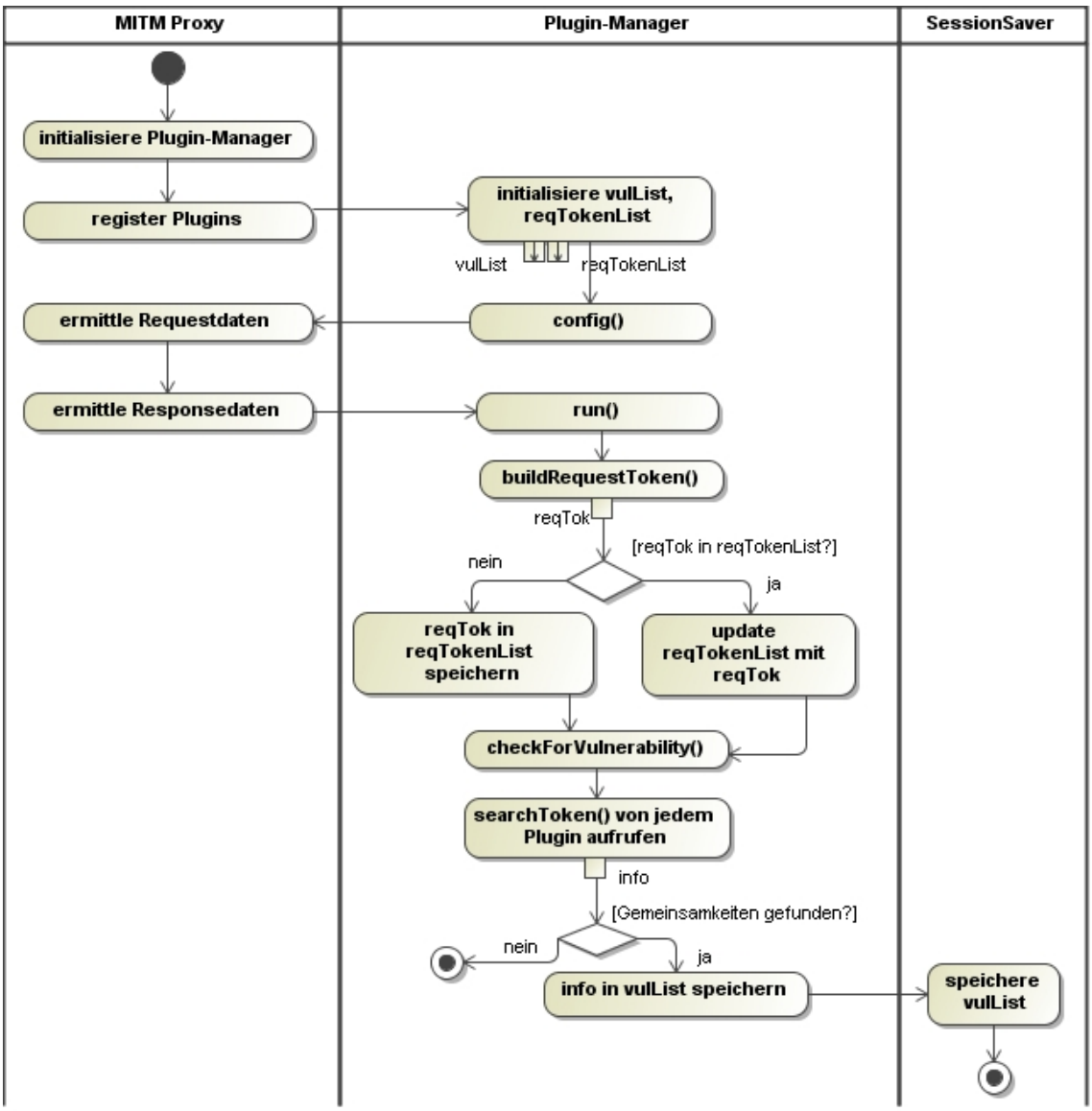


Abbildung 4.13.: Ablauf der Ausführung der Plugins

4.6.5. Verwalten der RequestToken-Liste

Die Verwaltung der RequestToken-Liste wird von der *run*-Methode der Klasse *AttackHandler* vorgenommen. Diese Methode wird direkt vom MITM-Proxy aufgerufen und stellt somit den Einstiegspunkt des Plugin-Managers dar. Neben der Verwaltung der RequestToken-Liste steuert diese Funktion die *checkForVulnerability*-Methode, welche die RequestToken-Liste auf ihre Gemeinsamkeiten hin untersucht (siehe Abschnitt 4.6.6). Im Folgenden wird die Funktionsweise der *run*-Methode näher erläutert.

Auszug aus der run-Methode

```
def run(self):
  del vulList[:]
  for handler in self.handlerList:
    if self.enabledHandlers[handler.getID()] == True:
      if self.httpResponse._content_type.find("text/html") != -1
        and self.httpResponse._code == 200:
        reqTok=handler.buildRequestToken(handler.getID(),
          self.fuzzableRequest, self.httpResponse)

        if not reqTokenList.find(reqTok):
          reqTokenList.append(reqTok)
        else:
          reqTokInList=reqTokenList.getRequestToken(reqTok)
          reqTokInList.updateRequestToken(reqTok)

    self.checkForVulnerability(handler)
```

Der obere Codeausschnitt der *run*-Methode überprüft zunächst, ob ein registriertes Plugin am Plugin-Manager durch den Benutzer aktiviert wurde (Zeile 4). Wenn hierbei des Plugin als aktiviert gekennzeichnet wurde, erfolgt im nächsten Schritt die Überprüfung, ob der Webserver den aktuellen Request erfolgreich beantwortet hat (Rückgabewert: 200 (OK)) und ob die Antwort eine entsprechende HTML-Datei darstellt (Zeile 5 bis 6). Sollte auch dieser Test erfolgreich sein, wird im nächsten Schritt die Methode *buildRequestToken* vom jeweiligen Plugin aufgerufen, welches als Rückgabewert ein entsprechendes RequestToken-Objekt (reqTok) liefert (Zeile 7 bis 8).
Nach der Erzeugung des RequestToken-Objekts wird im weiteren Verlauf ermittelt, ob das Objekt bereits in der globalen RequestToken-Liste (reqTokInList) gespeichert wurde. Hierzu wird die globale RequestToken-Liste durchsucht und überprüft, ob die URL des aktuellen Requests und der Typ des Plugins bereits gespeichert wurden. Bei einem Nichtvorhandensein in der Liste wird dann das zuvor erzeugte RequestToken-Objekt entsprechend zur Liste hinzugefügt, wohingegen bei einem Existieren des Objekts in der Liste, die aktuellen Werte des RequestToken-Objekts mit dem korrekten Eintrag aus der Liste aktualisiert werden (Zeile 10 bis 14). Im Anschluss, wird im letzten Schritt die Funktion *checkForVulnerability* aufgerufen, welche die globale RequestToken-Liste auf Gemeinsamkeiten untersucht (Zeile 16).

4.6.6. Verwalten der gefundenen Gemeinsamkeiten

Zur Ermittlung und Verwaltung der gefundenen Zusammenhänge zwischen den einzelnen RequestToken-Objekten dient die Funktion *checkForVulnerability* in der Klasse *AttackHandler*, die direkt von der zuvor beschriebenen *run*-Methode aufgerufen wird. Für diese Aufgabe greift die Funktion *checkForVulnerability* auf die in den Plugins definierte *searchToken*-Methode zu. Der folgende Codeausschnitt zeigt die Funktionsweise der *checkForVulnerability*-Methode.

Auszug aus der Methode checkForVulnerability

```
def checkForVulnerability(self, handler):
  for reqTok1 in reqTokenList:
    for reqTok2 in reqTokenList:
      if reqTok1.getType() == reqTok2.getType() and
5       reqTok1.getType() == handler.getID():
        retTuple=handler.searchToken(reqTok1, reqTok2)
        if retTuple[0] == True:
          # create srList and shortMessage
          ...
10        vul=rt.Vulnerability(handler.getID(), shortMessage,
            retTuple[1], srList)
          if not vulList.find(vul):
            vulList.append(vul)
```

Das obere Codefragment zeigt, wie die Verwaltung der einzelnen Vulnerability-Objekte vorgenommen wird. Zunächst wird die Methode *checkForVulnerability* mit dem entsprechenden Plugin-Handler als Parameter aufgerufen. Im nächsten Schritt wir dann die globale RequestToken-Liste doppelt durchiteriert. So kann jedes RequestToken-Objekt mit sich selbst und mit jedem anderen verglichen werden. Beim Durchgehen dieser Liste wird jeweils überprüft, ob die beiden aktuell betrachteten RequestToken-Objekte vom gleichen Typ sind und ob der Typ mit dem aktuellen Plugin-Handler übereinstimmt (Zeile 2 bis 5). Bei einem erfolgreichen Testen auf diese Kriterien, wird nun die *searchToken*-Methode des am Anfang übergebenen Plugin-Handlers aufgerufen, welche als Eingabeparameter die beiden aktuell betrachteten RequestToken-Objekte entgegennimmt. Hierbei ist auch zu beachten, dass die beiden übergebenen RequestToken-Objekte dieselben sein können. Im nächsten Schritt ermittelt die *searchToken*-Methode alle Gemeinsamkeiten zwischen den beiden übergebenen RequestToken-Objekten. Als Ergebnis liefert diese Methode ein Tupel zurück, welches die Informationen über eine eventuell gefundene Gemeinsamkeit beinhaltet (Zeile 6 bis 7).
Bei einem Erkennen einer Gemeinsamkeit wird im nächsten Schritt ein entsprechendes Vulnerability-Objekt erzeugt. Hierzu werden zunächst zusätzlich Informationen wie eine kurze Beschreibung generiert, welche im Anschluss als Eingabeparameter für ein neues Vulnerability-Objekt dienen. Diese Informationen in Form eines Vulnerability-Objekts werden im letzten Schritt in die globale Vulnerability-Liste hinzugefügt (Zeile 10 bis 13).

4.6.7. Erzeugung und Integration eines Plugins am Beispiel einer SQL-Injection

Im Weiteren wird aufgezeigt, wie ein Plugin erstellt werden kann und wie dieses korrekt in den Plugin-Manager integriert wird. Hierzu wird dieses am Beispiel von SQL-Injection aufgezeigt. Andere Plugins können jedoch auf dieselbe Weise in dem Plugin-Manager eingebracht werden. Für das Umsetzen der in Kapitel 3.3.2 beschriebenen Techniken, wurde das Plugin *SqlHandler* entwickelt, welches zum Finden von SQL-Injections dient. Dieses Plugin besitzt die Funktionalität, SQL-Statements von einem entfernten Datenbankserver auszulesen und auf Schwachstellen hin auszuwerten. Voraussetzung jedoch für diese Aufgaben sind, wie bereits erwähnt, neben einer aktiven Verbindung zur Datenbank, ebenfalls ein Betreiben des Datenbankservers im Loggingmodus. Da jedoch die einzelnen Datenbankmanagementsysteme (DBMS) Unterschiede beim Logging aufweisen, wurden bei der Umsetzung des Plugins nur der MySQL-Server berücksichtigt. Weitere DBMS können jedoch bei Bedarf zusätzlich

in das Plugin integriert werden. Für das Mitloggen aller eingehenden SQL-Statements bieten MySQL-Server seit der Version 5.1.6 neben dem Logging in eine Datei, die Möglichkeit an, alle mitprotokollierten Informationen in eine spezielle Datenbanktabelle mit dem Namen *mysql.general_log* zu schreiben (siehe dazu [MyS]). Diese Variante, das Mitloggen in eine Datenbanktabelle, bietet eine einfache Möglichkeit die abgesetzten SQL-Statements mit Hilfe von SQL-Befehlen auszulesen, weshalb diese Variante bei der Umsetzung dieses Plugins verwendet wurde.

Konfiguration eines Plugins

Wie bereits beschrieben, wird für eine korrekte Arbeitsweise des SQL-Plugins eine aktive Verbindung zur Datenbank benötigt. Zum Aufbau einer solchen Verbindung müssen jedoch dem Plugin einige Parameter wie zum Beispiel der Datenbank-Host sowie die Zugangsdaten übergeben werden.
Für diese Aufgabe existiert, wie bereits im Abschnitt 4.6.1 erwähnt, die Methode *config*, über die alle Einstellungen für die Funktionsweise des Plugins vorgenommen werden können. Das folgende Beispiel zeigt, wie die Methode *config* verwendet wird, um die über die WebGUI vorgenommenen Einstellungen zu übergeben.

Konfiguration des Plugin zum Finden von Schwachstellen für SQL-Injection.

```
def config(self, conf):
  if self.db == None:
    self.db=MySQLdb.connect(host=conf["dbhost"],
        user=conf["dbuser"],
        passwd=conf["dbpassword"],
        db=conf["dbname"])
        ...
```

Die Methode *config* wird direkt von der *run*-Methode der Oberklasse *AttackHandler* mit den Konfigurationsparameter (conf) aufgerufen. Dieser Parameter enthält alle Einstellungen, die über die WebGUI vorgenommen wurden. Der Kern der *config*-Methode besteht aus der Erzeugung eines neuen MySQL-Datenbank-Objekts, welches alle benötigten Parameter wie den Datenbank-Host und die Zugangsdaten verwendet und mit diesen Informationen eine Verbindung zur Datenbank herstellt.

Erzeugen eines RequestToken-Objekts

Zum Finden von Schwachstellen in den einzelnen SQL-Statements ist es entscheidend, dass die einzelnen zu überprüfenden Requests mit ihren Merkmalen als RequestToken-Objekte in die RequestToken-Liste aufgenommen werden. Hierbei wurden für jedes RequestToken-Objekt die abgesetzten SQL-Statements, die der Datenbankserver empfangen hat, als Merkmal hinzugefügt. Der folgende Codeausschnitt der Methode *buildRequestToken* zeigt, wie die jeweiligen SQL-Statements zu einem Request ermittelt werden.

Erzeugen eines RequestToken-Objekts mit den SQL-Statements als Merkmal.

```
def buildRequestToken(self, type, fuzzReq, httpRes):
  SQLStatementList = []

  # create SQLStatement object list
  self.dbCursor.execute("""select event_time, user_host,
    command_type, argument from mysql.general_log""")
  for row in self.dbCursor.fetchall():
     st = SQLStatement(row[0], row[1], row[2], row[3])
     SQLStatementList.append(st)

  self._clearLogTable()
  myReq = rt.RequestToken(type, fuzzReq, SQLStatementList)
  myReq.loadParaemters()
  return myReq
```

Da bei einem Request die Webanwendung mehrere SQL-Statements absetzen kann, werden alle SQL-Statements in eine SQL-Statement-Liste gespeichert. Hierzu werden zunächst alle existierenden SQL-Statements aus der Tabelle *mysql.general_log* ausgelesen und im Anschluss in die *SQLStatementList* gespeichert (Zeile 2 bis 9). Nachdem dieser Vorgang beendet wurde, müssen alle Einträge aus der Logging-Tabelle gelöscht werden. Dadurch kann für einen späteren Request ermittelt werden, ob dieser ebenfalls Datenbankabfragen ausgelöst hat (Zeile 11). Im letzten Schritt wird das entsprechende RequestToken-Objekt erzeugt und an die aufrufende Funktion zurückgegeben (Zeile 12 bis 14).

Durchsuchen der RequestToken-Objekte

Für das Durchsuchen von Zusammenhängen zwischen den einzelnen RequestToken-Objekten wird, wie schon im Abschnitt 4.6.6 erwähnt, die Methode *searchToken* verwendet. Der folgende Codeauszug zeigt die Funktionsweise, wie Gemeinsamkeiten zwischen den einzelnen RequestToken-Objekten bei SQL-Injections gefunden werden.

Durchsuchen von zwei RequestToken-Objekten nach Gemeinsamkeiten.

```
def searchToken (self, reqTok1, reqTok2):
    infoList =rt.InformationList()
    vulnerableParameterList=[]

    paramList1 = reqTok1.getParametersDict()
    SQLStatementList = reqTok2.getToken()

    ret = False
    for param, value in paramList1.items():
        for statement in SQLStatementList:
            argument = statement.getArgument()
            if findByRegEx(value, argument):
                info=rt.Information(statement.toString())
                infoList.append(info)
                vulnerableParameterList.append(param+"="+value)
                ret = True
    tuple=(ret,infoList, vulnerableParameterList)
    return tuple
```

Zum Vergleichen von zwei RequestToken-Objekten wird die Funktion *searchToken* mit den beiden RequestToken-Objekten als Parameter aufgerufen. Diese ermittelt zunächst aus dem ersten RequestToken-Objekt die Werte der Request-Parameter und aus dem zweiten Request-Token-Objekt das abgespeicherte Merkmal eines Requests, welches bei diesem Plugin aus den gesammelten SQL-Statements besteht (Zeile 2 bis 6). Im Anschluss werden nun die einzelnen Request-Parameter in den SQL-Statements gesucht und bei einem Auftreten eines Treffers die entsprechenden Informationen wie den gefundenen Parameter im SQL-Statement und das SQL-Statement in eine Liste (*vulnerableParameterList*) gespeichert (Zeile 8 bis 16).
Nach einem Durchsuchen aller Request-Parameter in den einzelnen SQL-Statements werden im letzten Schritt die gesammelten Informationen als Tupel an die übergeordnete Funktion zurückgesendet (Zeile 17 bis 18).

Registrieren des Plugins im Plugin-Manager

Um das Plugin im Plugin-Manager korrekt in den Prototypen zu integrieren, muss im letzten Schritt das Plugin beim Plugin-Manager registriert werden. Dieser Vorgang wird direkt im Prototypen in der Datei *localanalyzeproxy.py* vorgenommen.

Registrieren des Plugins für SQL-Injection.

```
attackHandle = AttackHandler.AttackHandler()
sqlAttackHandler = SqlHandler.SqlHandler()
attackHandle.registerHandler(sqlAttackHandler)
```

Der obige Codeauszug demonstriert den Ablauf zum Registrieren des Plugins für SQL-Injection. Hierbei wird zunächst der Plugin-Manager initialisiert (*AttackHandler*). Im Anschluss wird ein neues Objekt (*sqlAttackHandler*) des zu registrierenden Plugins erstellt, welches im nächsten Schritt mittels der Funktion *registerHandler* von der Oberklasse *AttackHandler* in den Plugin-Manager integriert wird.

4.6.8. Entwicklung und Integration des XSS-Plugins

Die im Abschnitt 4.6.7 beschriebene Vorgehensweise hat anhand der Erzeugung eines Plugins für SQL-Injection aufgezeigt, wie ein Plugin entwickelt und in den Plugin-Manager integriert werden kann. Das Erstellen des Plugins zum Finden von XSS ist ähnlich zu diesem beschriebenen Vorgehen zum Erkennen von SQL-Injection. Jedoch wurden für die Entwicklung des XSS-Plugins die Ansätze aus Kapitel 3.3.3 berücksichtigt. Das bei diesem Plugin betrachtete Merkmal, welches für das Erstellen eines RequestTokens benötigt wird, ist der Response eines Requests. Wie im Kapitel 3.3.3 ebenfalls dargestellt wurde, können jedoch XSS-Schwachstellen in einem Kontext stehen, der im Browser des Anwenders nicht ausführbar ist. Um solche *false positives* zu vermeiden, wird zusätzlich zur Analyse im MITM-Proxy eine Auswertung des übergebenen XSS-Codes auf der Client-Seite vorgenommen. So kann überprüft werden, ob das XSS-Statement in einen ausführbaren Kontext des Responses steht, was zu einer Verringerung der false positives führt.

4.7. Automatisches Testen von Requests

Ziel dieser Diplomarbeit ist unter anderem das semi-automatische Testen von Webanwendungen auf diverse Sicherheitslücken. Für diese Aufgabe wurde eine Funktion in den Prototypen implementiert, die es ermöglicht, zu einem im Voraus gespeicherten Request, die einzelnen Parameter des Requests automatisch mit vordefinierten Zeichenketten (Patterns) zu manipulieren und den daraus resultierenden Request im Anschluss zu versenden. Zur automatischen Auswertung eines abgesendeten Requests auf Schwachstellen wird außerdem auf den Plugin-Manager, der in Abschnitt 4.6 erläutert wurde, zurückgegriffen.

Abbildung 4.14.: Allgemeines Vorgehen zum automatischen Testen

Die Abbildung 4.14 zeigt den allgemeinen Ablauf beim automatischen Testen auf Schwachstellen. Zunächst werden über den in Abschnitt 4.5 beschriebenen *Interceptor* einzelne Requests oder komplette Workflows abgespeichert. Nach diesem Vorgang öffnet der Benutzer das *AutomateTest-Window*, tätigt sämtliche Einstellungen für die Testdurchführung und startet im Anschluss das automatische Testen. Nach Ablauf des gesamten Analysevorgangs bekommt der Benutzer eine Zusammenfassung dargestellt, die unter anderem alle gefundenen Schwachstellen anzeigt.

4.7.1. Requests automatisches ausführen

Eine wichtige Funktionalität bei dem in Abbildung 4.14 beschriebenen Vorgang ist das automatisierte Ausführen von einzelnen Request oder gesamten Workflows. Für diese Aufgabe wurde eine Funktion mit dem Namen *w3af_makeSubmit* implementiert, die in der Datei *w3af_record.js* abgespeichert ist. Zusätzlich zum Ausführen der einzelnen Requests, ermöglicht diese Funktion das Manipulieren einzelner Parameter bei einem Request. Im folgenden Codeauszug ist die Funktionalität dieser Methode dargestellt.

Gespeicherte Requests automatisch ausführen.

```
function w3af_makeSubmit(automateRequest, overwrite_cid) {
  for ( var i = 0; i < requestData.length; i++) {
    if (document.location.href == requestData[i].fromURL) {
      // get saved data and create new form element
      var myForm = document.createElement("form")
      ...

        // add parameters and values to form element
        for ( var param in requestData[i].paramValue) {
          value = requestData[i].paramValue[param];

          otherParamValue = automateProcess['otherParams'][param];
          manipulateParamValue =
              automateProcess['manipulateParams'][param];

          if (otherParamValue) {
            value = otherParamValue;
          }

          if (manipulateParamValue == "manipulate") {
            value = value + automateProcess['cheats'][cid]
          }

          var myInput = document.createElement("input")
          ...
        }
      }
      ...
      document.body.appendChild(myForm);

      // send website to server
      myForm.oldSubmit();
      ...
    }
  }
}
```

Die oben definiere Funktion *w3af_makeSubmit* wird nach dem Laden einer Webseite automatisch mit den Eingabeparametern *automateRequest* und *overwrite_cid* aufgerufen. Der Parameter *automateRequest* definiert hierbei, ob der auszuführende Request automatisch durchgeführt werden soll, wohingegen der Parameter *overwrite_cid* das aktuelle Pattern darstellt, mit dem bestimmte Parameter des Requests manipuliert werden sollen.
Für das automatische Ausführen des Requests wird zunächst ermittelt, ob die aktuell im Browser angezeigte URL in der Menge der zuvor abgespeicherten Requests enthalten ist. Bei einer positiven Ermittlung werden im nächsten Schritt die einzelnen Parameter des zu automatisierenden Requests ermittelt und in einem HTML-Formular abgespeichert (Zeile 5 bis 27). Hierbei werden zu den manipulierenden Parametern des Requests die zuvor bei der Testdeklaration definierten Patterns angehangen. Nach der Erstellung des HTML-Formulars wird dieses in den DOM der Webseite eingefügt und mit Hilfe der Javascript-Funktion *submit()* automatisch abgeschickt (Zeile 31 bis 32).

4.7.2. Darstellung des Testfensters zur automatischen Analyse

Wie bereits erwähnt, wurde eine Oberfläche implementiert, die es dem Benutzer ermöglicht, alle, für den Test benötigten Informationen zu definieren und den automatischen Testdurchlauf zu starten. Des Weiteren stellt das *AutomateTest-Window* eine Zusammenfassung des durchgeführten Tests dar und zeigt zusätzlich die durch die einzelnen Plugins gefundenen Schwachstellen an.

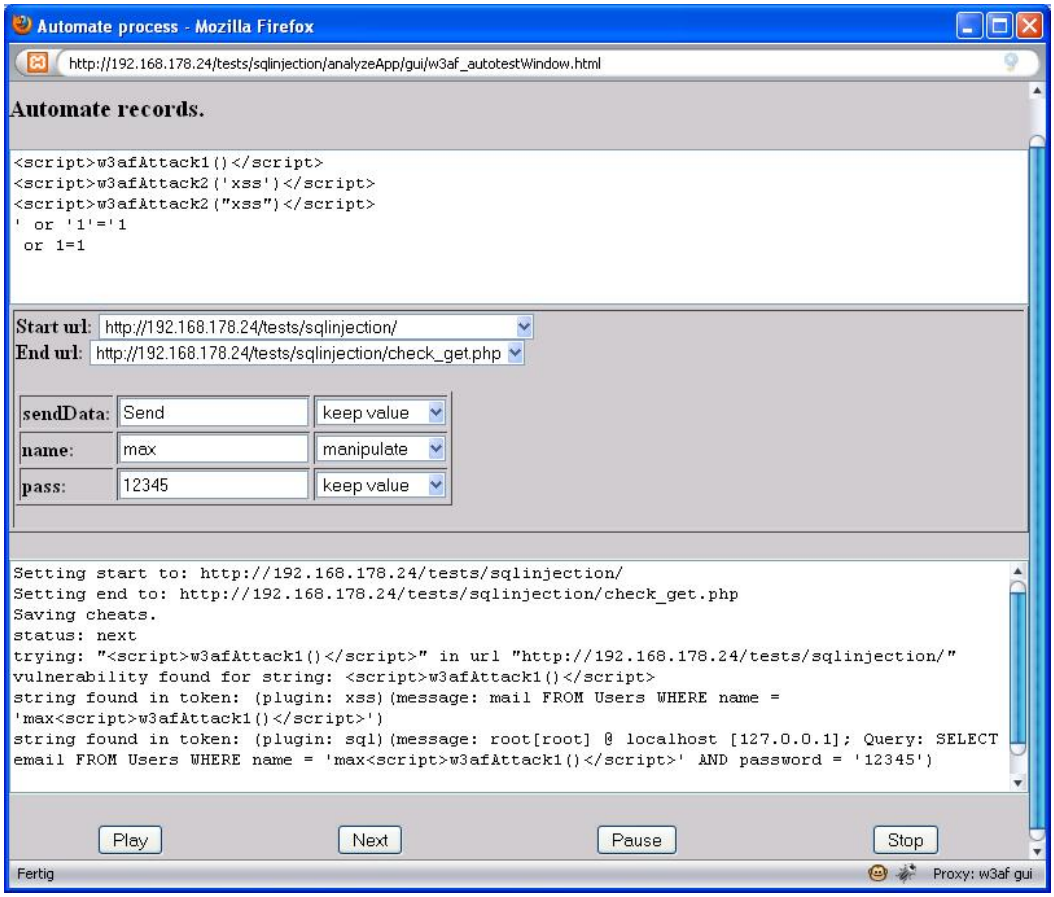


Abbildung 4.15.: Darstellung des AutomateTest-Window.

Die Abbildung 4.15 zeigt das Aussehen und die Möglichkeiten der einzelnen Einstellungen für das automatisierte Testen. Im oberen Bereich des Fensters kann der Benutzer die Menge der Patterns bestimmen, die später an den einzelnen Parametern eines Requests angehangen werden. Hierbei werden bereits dem Tester vordefinierte Patterns angezeigt, die er beliebig abändern oder erweitern kann. Diese vordefinierten Patterns sind in der Javascript-Datei (*w3af_cheatsheet.js*) gespeichert.
Im mittleren Teil des *AutomateTest-Windows* definiert der Benutzer den Anfang und das Ende des zu untersuchenden Workflows. Zusätzlich kann er für den Anfang des Workflows die Werte der einzelnen Parameter definieren. Hierbei ist es auch möglich, zu bestimmen, welche Parameter mit den oberen definierten Patterns manipuliert werden sollen. Im nächsten Teil des Fensters wird eine Zusammenfassung des Tests dargestellt. Dort kann der Benutzer erkennen, wie der aktuelle Fortschritt des Testdurchlaufs ist und ob Schwachstellen bereits

gefunden wurde. Der unterste Teil des Fensters gibt dem Benutzer die Möglichkeit, den Testdurchlauf zu steuern. Neben dem gesamten Abspielen des Tests hat er die Möglichkeit den Test Schritt für Schritt auszuführen, sowie kurz zu unterbrechen oder komplett zu beenden.

5. Anwendungsbeispiel

Im weiteren Teil dieser Arbeit wird anhand eines Anwendungsbeispiels gezeigt, wie die in Kapitel 4 implementierte Architektur in der Praxis eingesetzt werden kann. Das hierbei verwendete Anwendungsbeispiel stellt einen Registrierungsvorgang einer Webapplikation dar, wie man diesen oft im Internet vorfindet.

5.1. Aufbau und Funktionsweise der Testanwendung

Um das seitenübergreifende Finden von Schwachstellen darzustellen, besteht der Registrierungsvorgang aus drei Webseiten, die der Benutzer durchlaufen muss. Dabei sind jedoch in der Webanwendung Schwachstellen bezüglich XSS und auch SQL-Injection enthalten. Die folgende Abbildung 5.1 zeigt die einzelnen Webseiten mit der Reihenfolge, in der sie durchlaufen werden.

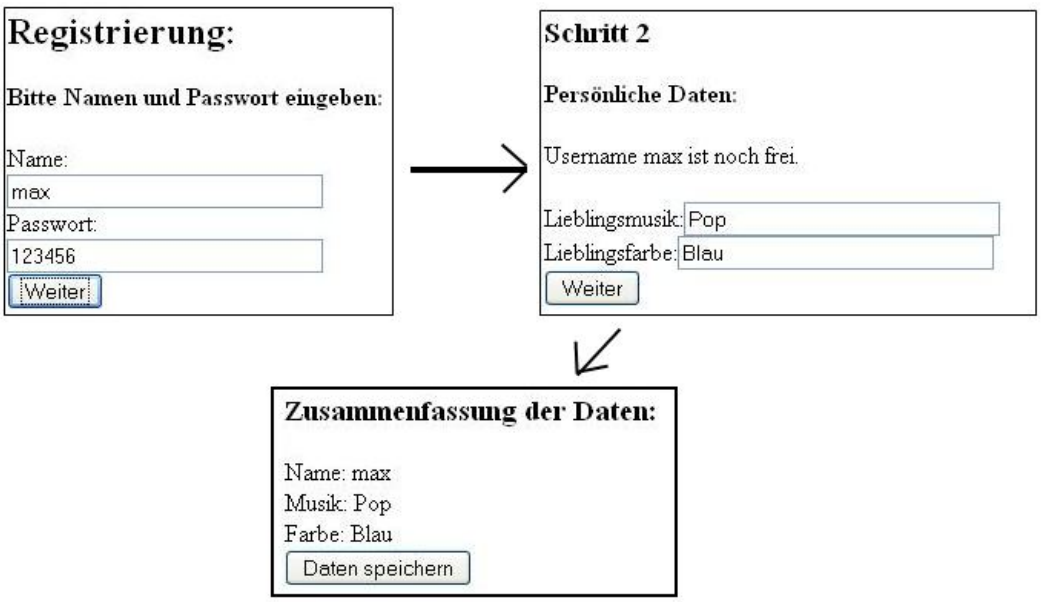


Abbildung 5.1.: Darstellung eines Anwendungsbeispiels.

Auf der ersten Webseite des Registrierungsvorgangs wird der Benutzer aufgefordert, einen Namen und ein neues Passwort einzugeben. Beim Betätigen der Schaltfläche *Weiter* wird im nächsten Schritt überprüft, ob der eingegebene Benutzername bereits im System vorhanden ist. Bei einem Nicht-Existieren dieses Namens in der Datenbank wird der Benutzer aufgefordert, weitere persönliche Daten wie *Lieblingsmusik* oder seine *Lieblingsfarbe* einzugeben. Leider ist die Anfrage an die Datenbank zur Überprüfung des Namens unsicher, so dass hier eine Schwachstelle bezüglich zu SQL-Injection besteht. Nach dem Absenden der zweiten

Webseite wird im letzten Teil der Webanwendung eine Zusammenfassung der eingegebenen Daten gezeigt. Die ausgegebenen Daten werden jedoch nicht auf spezielle Zeichen untersucht, so dass hier ebenfalls eine Schwachstelle in Form von XSS vorhanden ist. Die Tabelle 5.1 zeigt noch einmal zusammengefasst die vorhandenen Schwachstellen dieser Webawendung.

Webseite	**XSS**	**SQLi**
seite1.php	nein	nein
seite2.php	nein	ja
seite3.php	ja	nein

Tabelle 5.1.: Zusammenfassung der existierenden Schwachstellen.

Beim Eingeben und Absenden der einzelnen Teile der Webanwendung ist zu beachten, dass der Benutzer die abgesetzten Requests zunächst mit Hilfe des Interceptors aufzeichnet. Dieses ermöglicht ihm die logische Darstellung des existierenden Workflows für den Registrierungsvorgang in der Applikation, welcher zur späteren automatischen Analyse der Webanwendung benötigt wird. Nach dem Abspeichern des Workflows kann dann das automatische Suchen nach Schwachstellen für die Webanwendung vorgenommen werden. Hierzu wird zunächst das Automatisierungs-Fenster für die automatischen Analyse geöffnet, in der dann die einzelnen Einstellungen vom Tester vorgenommen werden. Die Abbildung 5.2 zeigt die vorgenommenen Einstellungen zum automatischen Testen des vorher definierten Workflows.

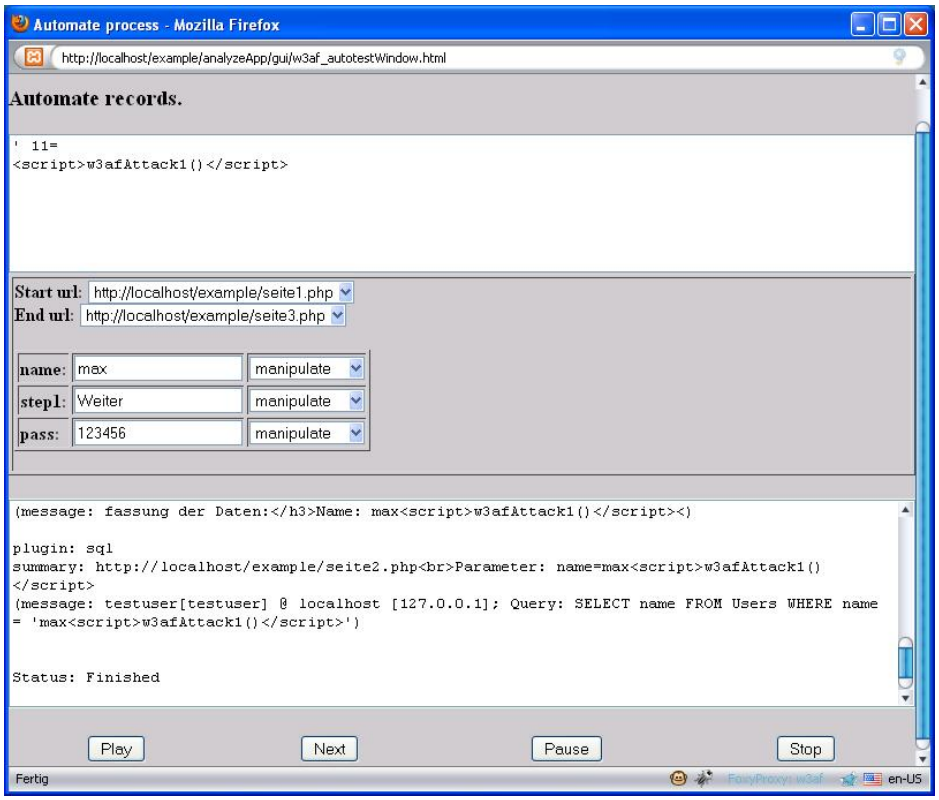


Abbildung 5.2.: Ausgabe der automatischen Schwachstellenanalyse.

Bei den vorgenommenen Einstellungen wurden zunächst die *Patterns*, die an den einzelnen Eingabeparametern angehangen werden sollen, definiert. Hierbei kann der Benutzer die einzelnen Patterns beliebig erweitern oder abändern. Nachdem die Patterns definiert wurden, kann der Benutzer den Anfang und das Ende des Workflows definieren. Hierbei ist zu beachten, dass alle Requests im Workflow automatisch ausgeführt werden. Im Anschluss definiert der Benutzer die Werte der einzelnen Parameter des Start-Requests vom Workflow. Hierbei kann er auch angeben, welche Parameter er mit Hilfe der im oberen Teil definierten Patterns auf XSS oder SQL-Injection testen möchte.

5.2. Interpretation des Ergebnisses der automatischen Analyse

Das Ergebnis der automatischen Analyse ist in Abbildung 5.2 im unteren Teil zu erkennen. Nachfolgend wird das Ergebnis der gefundenen Schwachstellen noch einmal aufgelistet und genauer interpretiert.

Auszug der automatischen Analyse

```
...
trying pattern: "' 11=" in url "http://localhost/example/seite1.php"
plugin: xss
summary: Start: http://localhost/example/seite2.php<br>
End: http://localhost/example/seite3.php<br>Parameter: name=max' 11
(message: fassung der Daten:</h3>Name: max' 11=<br>Musik: Pop<br>Farbe: B)

plugin: sql
summary: http://localhost/example/seite2.php<br>Parameter: name=max' 11
(message: testuser[testuser] @ localhost [127.0.0.1];
Query: SELECT name FROM Users WHERE name = 'max' 11=')

trying pattern: "<script>w3afAttack1()</script>" in
url "http://localhost/example/seite1.php"
vulnerability found for string: <script>w3afAttack1()</script>
plugin: xss
summary: Start: http://localhost/example/seite2.php<br>
End: http://localhost/example/seite3.php<br>
Parameter: name=max<script>w3afAttack1()</script>
(message: fassung der Daten:</h3>Name: max<script>w3afAttack1()
</script><)

plugin: sql
summary: http://localhost/example/seite2.php<br>
Parameter: name=max<script>w3afAttack1()</script>
(message: testuser[testuser] @ localhost [127.0.0.1];
Query: SELECT name FROM Users WHERE name = 'max<script>w3afAttack1()
</script>')

Status: Finished
```

Für das Pattern $'11 =$ wurde eine XSS-Schwachstelle in der Antwort des Requests *seite*3.*php* gefunden. Dabei konnte der Parameter *name* des Requests *seite*2.*php* als unsicher eingestuft werden (siehe Zeilen 1 bis 6). Des Weiteren wurde derselbe Parameter *name* des Requests *seite*2.*php* als unsicher in dieser Antwortseite identifiziert, der zu einer SQL-Injection führt (siehe Zeilen 8 bis 11).
Für das zweite untersuchte Pattern $< script > w3afAttack1() < /script >$ wurden dieselben Schwachstellen gefunden, die bereits beschrieben wurden (siehe Zeilen 13 bis 26). Zusätzlich konnte ebenfalls ermittelt werden, dass die gefundene XSS-Schwachstelle in der Antwortseite des Requests *seite*3.*php* auch ausnutzbar ist.
Zusammenfassend kann gesagt werden, dass alle Schwachstellen der Webanwendung, die in Tabelle 5.1 dargestellt sind, mit Hilfe der automatischen Analyse des Prototypen gefunden worden sind.

6. Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein Prototyp zum Testen von Schwachstellen in Webanwendungen entwickelt. Diese neue Architektur verbindet dabei die aktuellen Techniken und Vorteile wie das client-seitige aber auch das proxy-seitige Testen einer Webanwendung, welche in Kapitel 2.2 dargestellt wurden. Die Vorteile die sich aus dieser Kombination ergeben sind sehr umfangreich. So können zum Einen mit Hilfe von client-seitigen Techniken wie die Verwendung von *Javascript.prototype* alle Requests einer Webanwendung im Browser des Anwenders abgefangen und manipuliert werden. Der sich daraus ergebende Vorteil ermöglicht es dem Tester ein besseres Verständnis über den Ablauf und die Funktionsweise der einzelnen Teile der Webanwendung zu bekommen, welches beim semi-automatischen Testen sehr hilfreich ist.

Zum Anderen können mit Hilfe von proxy-seitigen Techniken die Probleme, welche bei reinen client-seitigen Analyseprogrammen auftreten, besser kontrolliert werden. So wurde eine neue Technik aufgezeigt, die es dem Tester ermöglicht, Schwachstellen in Webanwendungen über mehrere Webseiten zu finden. Durch einen hierbei zusätzlich entwickelten Plugin-Manager können weitere Plugins zur Schwachstellen-Analyse auf die seitenübergreifende Suche zurückgreifen.

Dass diese neuen entwickelten Ansätze durchaus ihre Berechtigung bei der Schwachstellenanalyse in Webanwendungen haben, zeigt das praxisnahe Anwendungsbeispiel im Kapitel 5, bei dem sich die Schwachstellen über mehrere Webseiten bewegen. Jedoch müssen zum Finden weiterer Schwachstellen zusätzliche Plugins in den proxy-seitigen entwickelten Plugin-Manager aus Kapitel 4.6 integriert werden. Ein Vorgehen zum Einbinden weiterer Plugins, die automatisch das seitenübergreifende Suchen von Schwachstellen verwenden, zeigt der Abschnitt 4.6.7.

Eine weitere Möglichkeit ist die Integration weiterer client-seitiger Methoden zum Abfangen eines Requests durch den entwickelten *Interceptor*. Durch den modularen Aufbau des Interceptors können auch zukünftige Web 2.0 Mechanismen problemlos in die neue Architektur integriert werden. Eine Vorgehensweise zeigt der Abschnitt 4.5.

Des Weiteren ist die Kernfunktionalität des semi-automatischen Testens im Prototypen zu erweitern. Da zur Zeit bei einem abgespeicherten Workflow nur der erste Request auf Schwachstellen untersucht wird, stellt das Ausweiten des Analysevorgangs zum Finden von Schwachstellen auf alle Teile des abgespeicherten Workflows eine sinnvolle Erweiterung dar.

A. Benutzerhandbuch

Das folgende Dokument gibt ein Überblick über die Installation der Proxy-Erweiterung des w3af-Frameworks an.

Vorteile der Proxy-Erweiterung

- client-seitige Analyse einer Webanwendung
- Unterstützung beim automatischen Testen von Patterns in Webanwendungen
- automatisches Testen auf Schwachstellen von einzelnen Workflows in Webanwendungen
- Anhand des Plugin-Managers können neue Analysen für Schwachstellen integriert werden.

A.1. Installation

Grundsätzlich ist das w3af-Framework plattform-unabhängig. Die Entwicklung des Prototypen wurde jedoch unter Windows XP realisiert, weshalb die neue Proxy-Erweiterung nicht unter Unix-Systeme getestet wurde. Des Weiteren ist es notwendig, dass das w3af-Framework selbst installiert und lauffähig ist. Eine Anleitung über die Installation und Verwendung des w3af-Frameworks kann unter der folgenden Adresse heruntergeladen werden.

`http://w3af.sourceforge.net/`

Die Proxy-Erweiterung selber kann unter der folgenden Adresse heruntergeladen werden:

`http://www.cip.ifi.lmu.de/~hoene/`

A.1.1. Installations-Anforderungen

Die folgenden Programme bzw. Pakete müssen für eine korrekte Benutzung der Proxy-Erweiterung installiert sein. Bei der Installation unter Windows sind diese bereits in der Installationsdatei integriert und werden bei der Ausführung der Installationsroutine automatisch mit installiert.

- Python SetupTools
- Python Simplejson
- Python MySQL

A.2. Starten der Proxy-Erweiterung

Das w3af-Framework stellt grundsätzlich zwei Arten der Benutzerinteraktion zur Verfügung (konsolen-basierte Verwendung und GUI-basierte Verwendung). Zur Verwendung der Proxy-Erweiterung muss die grafische Benutzeroberfläche des w3af gestartet werden. Im Anschluss wird über den Menüpunkt *Tools* der integrierte MITM-Proxy aufgerufen. Um nun die Proxy-Erweiterung zu verwenden, muss die Schaltfläche *DOM Analyze* aktiviert werden. Die folgende Abbildung zeigt den gestarteten MITM-Proxy, welcher die Proxy-Erweiterung verwendet.

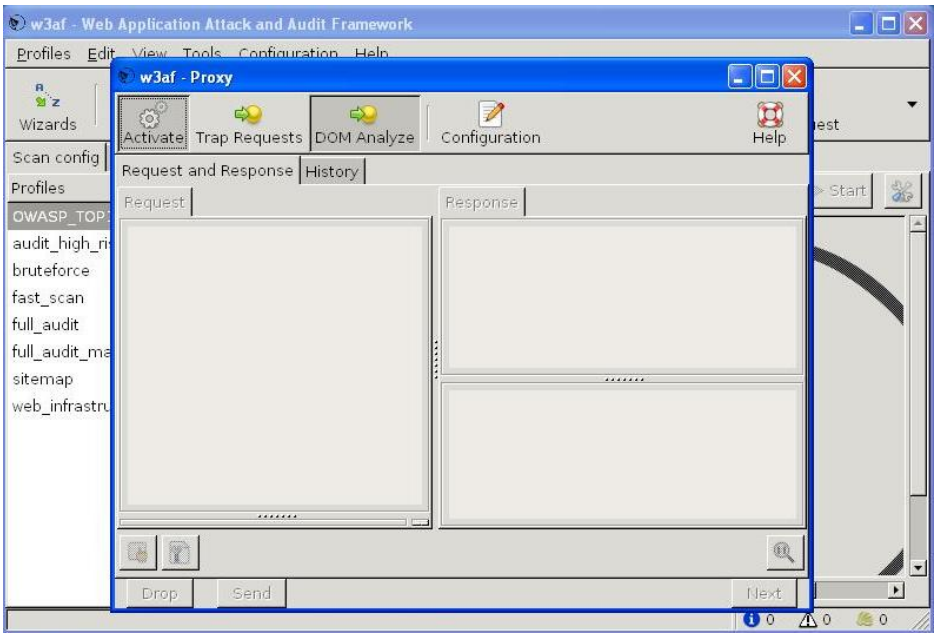


Abbildung A.1.: Starten der Proxy-Erweiterung.

Nach dem Starten der Proxy-Erweiterung muss nun der Webbrowser entsprechend konfiguriert werden, damit dieser alle Verbindungen über den MITM-Proxy leitet. Hierfür muss bei dem entsprechenden Browser ein *Proxy* eingestellt werden. Der hierbei standardmäßige verwendete *Hostname* lautet *localhost* mit dem *Standard-Port 8080*.

Abbildungsverzeichnis

Literaturverzeichnis

[Anl02] Anley, C.: *Advanced SQL injection in SQL Server applications.* White paper, Next Generation Security Software Ltd, 2002.

[Bic] Bicking, Ian: *Python HTML Parser Performance.* http://blog.ianbicking.org/2008/03/30/python-html-parser-performance/.

[CA] Chema Alonso, Daniel Kachakil, Rodolfo Bordón Antonio Guzmán y Marta Beltrán: *Time-Based Blind SQL Injection using Heavy Queries.*

[Com] Community, Beautiful Soup: *Beautiful Soup.* http://www.crummy.com/software/BeautifulSoup/.

[Con07] Consortium, Web Application Security: *Web Application Security Statistics*, 2007. http://projects.webappsec.org/Web-Application-Security-Statistics.

[Gro] Group, Selenium: *Selenium web application testing system.* http://seleniumhq.org/.

[Hun] Hunlock: *Howto Dynamically Insert Javascript And CSS.* http://www.hunlock.com/blogs/Howto_Dynamically_Insert_Javascript_And_CSS.

[JL] Jia, X. und H. Liu: *Rigorous and automatic testing of Web applications.* Citeseer.

[JSHJK06] Jaswinder S. Hayre, CISSP und CISSP Jayasankar Kelath: *Ajax Security Basics.* Security focus, 2006. http://www.securityfocus.com/infocus/1868.

[Kil09] Kilic, Fatih: *Evaluierung vorhandener Abfangproxyserver und Spezifikation eines neuartigen Proxykonzeptes für Sicherheitsuntersuchungen von Web 2.0 Applikationen.* Diplomarbeit, FH Ingolstadt, 2009.

[KKH+07] Kosuga, Y., K. Kono, M. Hanaoka, M. Hishiyama und Y. Takahama: *Sania: Syntactic and semantic analysis for automated testing against SQL injection.* In: *23rd Annual Computer Security Applications Conference. IEEE Computer Society*, Seiten 107–117, 2007.

[MBX] Martin, E., S. Basu und T. Xie: *Automated Testing and Response Analysis of Web Services.*

[Moza] Mozilla: *Class-Based vs. Prototype-Based Languages.* https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Class-Based_vs._Prototype-Based_Languages.

[Mozb] MOZILLA: *window.showModalDialog*. `https://developer.mozilla.org/En/DOM/Window.showModalDialog`.

[MyS] MYSQL: *The General Query Log*. `http://dev.mysql.com/doc/refman/5.1/en/query-log.html`.

[Ora] ORACLE: *Administering the Query Log*. `http://download.oracle.com/docs/cd/E12096_01/books/admintool/admintool_AdministerQuery14.html`.

[OWA] OWASP: *Category:Attack*. `http://www.owasp.org/index.php/Category:Attack`.

[Riaa] RIANCHO, ANDRES: *w3af A framework to own the Web*. cybsec.com.

[Riab] RIANCHO, ANDRES: *w3af User Guide*. Document Version 1.6.7.

[Riac] RIANCHO, ANDRES: *Web Application Attack and Audit Framework*. `http://w3af.sourceforge.net`.

[Rit07] RITCHIE, P.: *The security risks of AJAX/web 2.0 applications*. Network Security, Seite 4, 2007.

[SLN04] SINGH, S., J. LYONS und D.M. NICOL: *Fast model-based penetration testing*. In: *Proceedings of the 36th conference on Winter simulation*, Seite 317. Winter Simulation Conference, 2004.

[SMSP] SAMPATH, S., V. MIHAYLOV, A. SOUTER und L. POLLOCK: *A scalable approach to user-session based testing of web applications through concept analysis*. Citeseer.

[W3Ca] W3C: *17 Forms*. `http://www.w3.org/TR/html401/interact/forms.html#h-17.13.1`.

[W3Cb] W3C: *Basic HTML data types*. `http://www.w3.org/TR/html4/types.html`.

[W3Cc] W3C: *Document Object Model (DOM)*. `http://www.w3.org/DOM/`.

[W3Cd] W3C: *Status Code Definitions*. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

[W3Ce] W3C: *XMLHttpRequest*. `http://www.w3.org/TR/XMLHttpRequest/`.

[Wik] WIKIPEDIA: *Web 2.0*. Version vom 22.10.2009, `http://de.wikipedia.org/wiki/Web_2.0`.

[XSP09] XIONG, P., B. STEPIEN und L. PEYTON: *Model-Based Penetration Test Framework for Web Applications Using TTCN-3*. In: *E-Technologies: Innovation in an Open World: 4th International Conference, MCETECH 2009, Ottawa, Canada, May 4-6, 2009, Proceedings*, Seite 141. Springer, 2009.