

Inhaltsverzeichnis

| | | |
|----------|--------------------------------------|-----------|
| 1 | Einleitung | 1 |
| 1.1 | Aufgabenstellung | 1 |
| 1.2 | Vorgehensweise | 2 |
| 2 | Grundlagen | 3 |
| 2.1 | Webcrawler | 3 |
| 2.1.1 | Einsatzgebiete | 3 |
| 2.1.2 | Funktionsweise | 4 |
| 2.1.3 | Herausforderungen | 5 |
| 2.2 | Cross-Site Scripting | 6 |
| 2.2.1 | Cross-Site Scripting Typen | 7 |
| 2.2.2 | Beispiele | 8 |
| 2.2.3 | Taintfox | 8 |
| 2.3 | Firefox Plugins | 9 |
| 2.3.1 | Firefox Add-On SDK | 9 |
| 2.3.2 | WebExtension API | 10 |
| 2.3.3 | Plugin Aufbau | 10 |
| 3 | Entwurf | 12 |
| 3.1 | Anforderungsanalyse | 12 |
| 3.2 | Bestehende Umgebung | 15 |
| 3.3 | Webrawler Architekturen | 16 |

| | | |
|----------|------------------------------------|-----------|
| 3.4 | Neue Umgebung | 18 |
| 3.4.1 | Master-Server | 20 |
| 3.4.2 | Worker-Plugin | 20 |
| 3.4.3 | Analyse-Plugin | 21 |
| 3.4.4 | Validierungsservice | 22 |
| 4 | Implementierung | 23 |
| 4.1 | Master-Server | 23 |
| 4.1.1 | Klassendiagramm | 23 |
| 4.1.2 | Datenmodell | 25 |
| 4.1.3 | Crawl Strategien | 27 |
| 4.2 | Worker-Plugin | 29 |
| 4.2.1 | Aufbau des Plugins | 29 |
| 4.2.2 | Aufgetretene Probleme | 30 |
| 5 | Anwendung | 33 |
| 5.1 | Alexa Top 500 | 33 |
| 5.1.1 | Webcrawler Konfiguration | 34 |
| 5.1.2 | Daten zum Crawlvorgang | 34 |
| 5.1.3 | Daten zur Analyse | 35 |
| 5.2 | Webforum | 38 |
| 5.2.1 | Webcrawler Konfiguration | 38 |
| 5.2.2 | Daten zum Crawlvorgang | 39 |
| 5.2.3 | Daten zur Analyse | 39 |
| 6 | Zusammenfassung | 41 |
| 6.1 | Nächste Schritte | 42 |
| | Abbildungsverzeichnis | 43 |
| | Tabellenverzeichnis | 44 |

1 Einleitung

[myid] Im Zeitalter des Internets werden immer mehr Dienstleistungen im Internet angeboten. Diese gehen weit über statische Webseiten, wie man sie aus den Anfängen des Internets kennt, hinaus. Diese sogenannten Webapplikationen sind komplexe Anwendungen, wie zum Beispiel Onlineshops, soziale Netzwerke oder Onlinebanking-Portale. Aufgrund der Komplexität dieser Anwendungen kann es schnell zu ungewollten Sicherheitslücken kommen. Dies ist kritisch, da oft sehr sensible Daten im Spiel sind.

Cross-Site-Scripting ist ein Angriff auf eine solche Sicherheitslücke, der in vielen modernen Webapplikationen erfolgreich durchgeführt werden kann. Dabei wird versucht auf eine fremde Webseite Schadcode zu hinterlegen, welcher danach von anderen Benutzern unbeabsichtigt ausgeführt wird.

Bei SAP wird versucht, solche Schwachstellen automatisiert aufzudecken. Aus diesem Grund wurde ein modifizierter Firefox entwickelt, welcher es ermöglicht herauszufinden, welche Eingaben des Benutzers an kritische Stellen in der Webseite gelangen.

1.1 Aufgabenstellung

Oft ist es sinnvoll, nicht nur eine einzige Webseite auf Schwachstellen zu testen, sondern automatisiert eine Vielzahl von Webseiten zu besuchen. Um

dies zu ermöglichen, soll im Rahmen dieser Arbeit ein Webcrawler entwickelt werden, der eine Liste von Startseiten besucht und aus diesen Verweise auf weitere Webseiten extrahiert. Dafür muss eine Architektur gefunden werden, die sich in das aktuelle System einfügen lässt und gute Skalierungseigenschaften besitzt.

Zum Testen dieses Webcrawlers werden eine Reihe von Webseiten besucht und mit bestehenden Analysetools auf Cross-Site-Scripting Schwachstellen untersucht. Generell soll aber die Möglichkeit bestehen, auch andere Daten während des Crawlprozesses zu sammeln und analysieren.

1.2 Vorgehensweise

Diese Arbeit ist in sechs Kapitel aufgeteilt. In der Einleitung wurde die Ausgangslage und Aufgabenstellen bereits kurz beschrieben. Im folgenden Kapitel werden nun die grundlegenden Begriffe und Technologien, welche für das Verständnis dieser Arbeit notwendig sind, besprochen. Danach wird im dritten Kapitel der Entwurf des Webcrawlers vorgestellt. Dabei wird zum einen eine genaue Anforderungsanalyse betrieben und zum anderen die Interaktion der einzelnen Komponenten beschrieben. Genauere Details zur Implementierung dieser Komponenten werden dann im vierten Kapitel gegeben. Das fünfte Kapitel beschäftigt sich mit der praktischen Anwendung des erstellten Webcrawlers. Es werden zwei Crawldurchläufe mit unterschiedlichen Parametern durchgeführt und die Ergebnisse analysiert. Im letzten Kapitel werden die gewonnenen Erkenntnisse nochmals zusammengefasst und die nächsten Schritte erläutert.

2 Grundlagen

2.1 Webcrawler

Ein Webcrawler (oft auch Spider genannt) ist ein Computerprogramm, welches es ermöglicht, eine große Anzahl von Webseiten automatisiert zu besuchen.

2.1.1 Einsatzgebiete

Einer der bekanntesten Einsatzgebiete von Webcrawlern ist in Suchmaschinen. Diese bauen eine große Datenbank mit indizierten Webseiten auf und erlauben es einem Benutzer, diese Datenbanken mit einer Suchanfrage zu durchsuchen. Um die Datenbank zu füllen und aktuell zu halten, wird ein Webcrawler verwendet, wie zum Beispiel der Googlebot der Suchmaschine Google. [<https://support.google.com/webmasters/answer/182072?hl=de>] Preisvergleichsportale sammeln die Preise des gleichen Produkts in verschiedenen Onlineshops. Somit ist es einem Kunden möglich, einfach Preise zu vergleichen, ohne die einzelnen Onlineshops zu besuchen. Das Aggregieren der Preisdaten wird oft mithilfe eines Webcrawlers umgesetzt. Dieser besucht alle Webseiten der Onlineshops und extrahiert auf jeder die aktuellen Preise. Ein Crawler kann auch im Zuge eines Penetrationstests verwendet

werden. Dabei wird eine Webseite bzw. Webanwendung gezielt auf Schwachstellen untersucht. Der Webcrawler hilft dabei, alle Seiten der Webanwendung zu besuchen und auf Schwachstellen zu testen. Anwendungen, die dies umsetzen, sind zum Beispiel „Kali Grabber“ [<http://tools.kali.org/web-applications/grabber>], „Vega“ [<https://subgraph.com/vega/index.en.html>] und „WebSurgery“ [<http://www.surgeonix.com/blog/index.php/archives/category/tools>]. Webcrawler werden allerdings auch für kriminelle Zwecke eingesetzt. So ist es zum Beispiel möglich, E-Mail-Adressen aus Webseiten zu extrahieren. Nachdem genug Seiten gecrawlt wurden, entsteht eine große Datenbank mit E-Mail-Adressen, die zum Versenden von Spam verwendet werden kann.

2.1.2 Funktionsweise

Die Funktionsweise eines Webcrawlers lässt sich im Wesentlichen auf einen einfachen Algorithmus reduzieren. Im ersten Schritt wird eine Webseite identifiziert, welche heruntergeladen werden soll. Alle Webseiten, die noch besucht werden sollen, sind in einer Warteschlange gespeichert. Zu Beginn des Programms wird diese Schlange mit einer Reihe von Startseiten initialisiert, welche als Ausgangspunkt des Crawlprozesses dienen.

Nachdem der Schlange eine zu besuchende Webseite entnommen wurde, muss geprüft werden, ob das Crawlen dieser Webseite überhaupt erlaubt ist. Mit der Hilfe einer robots.txt Datei oder dem HTTP-Header kann festgestellt werden, ob der Webseitenbetreiber das Crawlen nicht erwünscht. Falls die Webseite besucht werden darf, wird sie komplett heruntergeladen und zu einer Liste mit besuchten Seiten hinzugefügt. Aus dem so erhaltenen HTML-Code werden nun alle Links auf andere Webseiten extrahiert. Diese neuen URLs werden der Warteschlange aus dem ersten Schritt hinzugefügt, sofern sie nicht in der Liste mit bereits besuchten Webseiten enthalten sind. Somit wird verhindert, dass eine Webseite mehrmals besucht wird.

Zum Schluss werden die heruntergeladenen Daten analysiert und daraus gewonnene Informationen abgespeichert. Eine Suchmaschine könnte zum Beispiel alle Wörter, die auf der Webseite zu finden sind, speichern. Danach beginnt der Prozess von vorne und die nächste zu besuchende Webseite wird bestimmt. [<http://www.ijcttjournal.org/Volume13/number-3/IJCTT-V13P128.pdf>] Bei speziellen Webcrawlern, wie zum Beispiel solche, die für Suchmaschinen eingesetzt werden, ist es außerdem nötig Webseiten nach einer gewissen Zeit erneut zu besuchen, da sich der Inhalt möglicherweise geändert hat. Die Einträge in der Liste mit besuchten Webseiten müssen also eine Lebensdauer zugewiesen bekommen. Läuft diese ab, so werden sie wieder in die Warteschlange des Webcrawlers eingefügt.

2.1.3 Herausforderungen

Dieser Algorithmus erscheint zunächst recht simpel, allerdings ist das World Wide Web sehr groß, wodurch sich einige Probleme ergeben. Um eine akzeptable Performance zu erreichen, muss der Crawlvorgang hoch parallel ablaufen. In modernen Anwendungen ist es sogar üblich, diesen Vorgang auf viele verschiedene Computer zu verteilen. In einem solchen Fall spricht man dann von „distributed web crawling“.

Durch die enormen Datenmengen wird es auch schwierig, die Datenstrukturen des Webcrawlers im Arbeitsspeicher zu halten. Zum Beispiel die Liste mit bereits besuchten Webseiten kann sehr schnell groß werden. Diese Daten auf ein sekundäres Speichermedium zu verlagern und dabei annehmbare Performance zu behalten ist keine leichte Aufgabe.

Es gibt so viele Webseiten im Internet, dass selbst große Firmen wie Google mit ihren Webcrawlern diese nicht alle abdecken können. Deshalb ist es nötig, gezielt Seiten zu besuchen, welche einen hohen Wert haben. Wie genau sich dieser Wert einer Seite berechnet, kommt auf den speziellen Anwendungsfall des Webcrawlers an. Für Suchmaschinen sind zum Beispiel

Seiten interessant, die von vielen anderen Stellen verlinkt wurden und hohe Aufrufzahlen haben. Seiten mit unwichtigem oder sogar schädlichem Inhalt müssen hingegen vermieden werden.

Ein hochperformanter Crawler kann schnell versehentlich einen Webserver lahmlegen. Indem in kurzer Zeit zu viele Anfragen an diesen geschickt werden, kann er unter der Last zusammenbrechen. Dies kommt einem Denial of Service (DOS) Angriff gleich. Um das zu verhindern, muss der Algorithmus so angepasst werden, dass nicht zu viele Anfragen an einen einzigen Webserver gestellt werden.

2.2 Cross-Site Scripting

Das „Open Web Application Security Project“ (OWASP) publiziert eine Liste mit den Top 10 Sicherheitsrisiken in Webanwendungen. Sowohl in der Liste aus dem Jahr 2013, als auch im Release Candidate für das Jahr 2017, steht Cross-Site-Scripting auf dem dritten Platz. Diese Art von Sicherheitslücke ist also schon lange bekannt, aber noch immer stark verbreitet und somit relevant.

Beim Cross-Site Scripting, kurz XSS, versucht ein Angreifer eigenen Code in eine fremde Webseite einzuschleusen. Häufig werden vom Nutzer manipulierbare Eingaben, wie z.B. die URL, Query-Parameter, Cookies oder auch Eingabefelder, in der Webseite eingefügt. Bei der Programmierung müssen diese Eingaben auf HTML und JavaScript Code überprüft werden. Wird diese Überprüfung versäumt und die Eingabe ohne Änderung auf der Webseite wiedergegeben, so ist es möglich, fremden Code einzuschleusen.

Das Ziel des Angreifers ist es, die Webseite so zu manipulieren, dass im Browser eines Opfers der Code des Angreifers ausgeführt wird. So kann der Angreifer das Opfer beispielsweise unbemerkt auf eine Phishing-Webseite weiterleiten, die zwar genauso aussieht wie die Originalseite, jedoch die

Login-Daten des Benutzers abfängt. Auch denkbar ist das Stehlen der Cookies des Opfers. Dadurch kann es dem Angreifer gelingen, eine Session des Opfers zu übernehmen, ohne sich anmelden zu müssen. Es gibt noch viele weitere Angriffe, die mithilfe einer XSS Lücke durchgeführt werden können. Aus diesem Grund ist diese Art der Sicherheitslücke sehr problematisch.

2.2.1 Cross-Site Scripting Typen

Cross-Site Scripting Lücken lassen sich in drei Typen untergliedern: stored XSS, reflected XSS und DOM Based XSS.

Stored XSS

Bei stored XSS Angriffen wird eine Benutzereingabe an den Server gesendet, welche dort in einer Datenbank persistiert wird. Zu einem späteren Zeitpunkt besucht das Opfer eine Webseite, die diese gespeicherten Daten wieder lädt und sie ungeprüft in die Webseite einfügt.

Dies könnte zum Beispiel in einem Forum passieren. Der Angreifer macht einen Foreneintrag, der schädlichen JavaScript Code enthält und zum Server gesendet wird. Ruft ein anderer Benutzer diesen Forenbeitrag auf, so wird der Schadcode automatisch in die Seite eingefügt und ausgeführt.

Reflected XSS

Wird eine Benutzereingabe unmittelbar, ohne Überprüfung, in die Webseite eingefügt, zum Beispiel als Fehlermeldung oder Suchergebnis, so spricht man von reflected XSS. Diese Benutzereingabe wird im Gegensatz zum stored XSS nirgendwo permanent gespeichert, allerdings wird sie möglicherweise trotzdem an den Server gesendet.

DOM Based XSS

Hier findet der gesamte Datenfluss von der Benutzereingabe bis zum DOM der Webseite im Browser statt. Der Schadcode des Angreifers bleibt also beim Klienten und erreicht nie den Server. Oft werden zum Beispiel Teile der URL direkt über JavaScript Funktionen wie `document.write` in die entsprechende Webseite eingefügt. Wird hierbei die Eingabe nicht richtig geprüft, so ist DOM Based XSS möglich.

2.2.2 Beispiele

2.2.3 Taintfox

Als Anwendungsfall des Webcrawlers dient in dieser Arbeit das automatische Finden von DOM basierten Cross-Site-Scripting Schwachstellen. Dazu wird der Taintfox verwendet, welcher diese Schwachstellen erkennen kann. Der Taintfox ist eine modifizierte Version des Mozilla Firefox. Um Cross-Site-Scripting zu erkennen, werden alle Benutzereingaben markiert (tainting). Diese Eingaben kommen dabei aus unterschiedlichen Quellen, wie zum Beispiel der URL (`document.location`) oder den Cookies (`document.cookies`). Der Taintfox verfolgt den Verlauf dieser markierten Werte und merkt sich dabei, welche Funktionen auf diese angewendet werden. Landet der Wert zum Schluss in einer sicherheitskritischen Funktion wie `document.write`, so liegt ein potenziell kritischer Datenfluss (Taintflow) vor. Die Funktion, in welcher der markierte Wert landet, nennt man auch Senke. Erkennt der Taintfox einen solchen Datenfluss, so wird das Taintfox spezifische Event `taintReport` ausgelöst. Auf dieses kann dann reagiert werden, um den Taintflow weiter zu analysieren.

2.3 Firefox Plugins

In diesem Abschnitt wird der generelle Aufbau von Firefox Plugins beschrieben. Für diese Arbeit ist der Einsatz von solchen Plugins nötig, um Webseiten automatisiert im Taintfox zu öffnen. Da der Taintfox nur ein modifizierter Firefox ist, können für beide Browser die gleichen Plugins verwendet werden. In der Firefox Dokumentation ist statt Plugin auch oft die Rede von Add-On. Diese beiden Begriffe werden im weiteren Verlauf der Arbeit synonym verwendet.

Für die Entwicklung von Firefox Plugins gibt es zwei Optionen: das Add-On SDK und die Webextension API. Beide Ansätze werden im Folgenden kurz angesprochen.

2.3.1 Firefox Add-On SDK

Das Firefox Add-On SDK ist lange Zeit die einzige Möglichkeit gewesen, um Plugins für den Firefox zu erstellen. Es bietet eine High Level Javascript API, mit welcher es möglich ist Tabs zu öffnen, das DOM zu manipulieren, Stylesheets zu ändern und vieles mehr. Eine Low-Level-API bietet zusätzlich tiefen Zugriff auf Firefox interne Schnittstellen, um zum Beispiel alle HTTP-Abfragen abzufangen. Mit Hilfe von XPCOM, einem Komponentenmodell von Mozilla, ist es möglich vom JavaScript Code aus, auf viele interne C++ basierte Komponenten zuzugreifen. Dadurch ist es Plugins möglich, den Firefox stark anzupassen. Dies ist ein Vorteil, der den Firefox lange Zeit von andern Browsern abgehoben hat.

Diese hohe Individualisierbarkeit hat allerdings auch Nachteile. So ist es zum Beispiel auch Schadware möglich, sensible APIs des Firefox zu verwenden. Auch das Einführen von neuen Technologien in den Firefox ist

schwierig, da die Abhängigkeiten zwischen den Komponenten unübersichtlich sind. Aus diesen Gründen hat sich Mozilla dazu entschieden, in zukünftigen Versionen das Addon-SDK nicht mehr vollständig zu unterstützen. [<https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/>]

2.3.2 WebExtension API

Die WebExtension API ist der Nachfolger des Add-On SDKs. Auch mit ihr lassen sich Plugins erstellen, die typische Browserfunktionalitäten ansteuern. Da die WebExtension API zum größten Teil kompatibel mit der Extension API des Chrome ist, sind die damit erzeugten Plugins browserunabhängig und laufen somit auch im Chrome, Opera und Microsoft Edge. Zum Zeitpunkt, zu dem diese Arbeit geschrieben wurde, ist die WebExtension API allerdings noch nicht ganz vollständig und tiefe Eingriffe in den Firefox sind nicht mehr möglich. Trotzdem empfiehlt Mozilla den Umstieg auf WebExtensions, da dies der zukünftig unterstützte Weg ist, Plugins für den Firefox zu erstellen.

2.3.3 Plugin Aufbau

Der Aufbau eines Firefox Plugins, welches mit dem Firefox Add-On SDK erstellt wurde, ist grundlegend gleich zu dem Aufbau einer WebExtension. Der Code ist in zwei getrennte Bereiche geteilt. Der Grundcode des Plugins hat Zugriff auf alle Firefox APIs und kann diese nutzen, um zum Beispiel neue Tabs zu öffnen, Benachrichtigungen anzuzeigen oder das Kontextmenü zu erweitern. Dieser Code kann zwar eine neue Seite aufrufen, allerdings kann nicht direkt auf den Inhalt dieser Seite zugegriffen werden. Genauer gesagt gibt es keinen Zugriff auf das DOM der geladenen Seite.

An dieser Stelle kommen sogenannte „Content Scripts“ zum Einsatz. Diese haben Zugriff auf den Webinhalt und verhalten sich wie ein Script, welches auf der entsprechenden Webseite eingebettet ist. Der Grundcode kann diese Content Scripts in beliebige Webseiten injizieren, um so gewisse Aufgaben auf dieser Seite zu erledigen. Die Kommunikation zwischen Grundcode und Content Script findet dabei über ein `port` Objekt statt, welches das asynchrone Versenden von Nachrichten zwischen den beiden erlaubt.

3 Entwurf

3.1 Anforderungsanalyse

Im Gespräch mit der Web-Security Abteilung von SAP wurden eine Reihe von Anforderungen an den Webcrawler erarbeitet. Die folgende Liste beschreibt diese jeweils in einem Abschnitt und gibt jeder Anforderung eine eindeutige Kennung (A + Zahl), sodass auf sie an späteren Stellen dieser Arbeit referenziert werden können. Sie sind dabei in keiner besonderen Reihenfolge gelistet, sondern alle wichtig, um den Webcrawler effektiv einsetzen zu können.

- A1** Der Crawlvorgang soll gestartet werden können, indem eine Liste von URLs angegeben wird, die als Startseiten verwendet werden. Die Liste von URLs soll als CSV-Datei abgelegt sein. Durch eine Kommandozeilenoption kann der Pfad zu dieser Datei beim Start mit übergeben werden. Alternativ kann auch nur eine einzige URL direkt in der Kommandozeile angegeben werden
- A2** Nachdem der Crawler gestartet wurde, soll dieser automatisch alle Startseiten besuchen und aus diesen weitere URLs extrahieren, die danach besucht werden.

- A3** Das Besuchen einer Webseite soll im Taintfox erfolgen. Das heißt, sie wird in einem Tab normal geöffnet, wodurch unter anderem auch alle eingebetteten JavaScript Dateien ausgeführt werden.
- A4** Auf jeder Webseite, die vom Webcrawler besucht wird, soll ein Analyse-Plugin ausgeführt werden können, welches ebenfalls im Taintfox installiert ist. Erst wenn dieses mit der Bearbeitung der Webseite fertig ist, darf sie vom Webcrawler geschlossen werden.
- A5** Das Analyse-Plugin kann dem Webcrawler zu jeder besuchten Webseite ein Datenpaket mit Analyseergebnissen übergeben. Diese Daten sollen dann zur späteren Weiterverarbeitung zentral in einer Datenbank abgelegt werden.
- A6** Es soll eine Option geben, die es erlaubt auf allen Webseiten keine externen Links zu verfolgen. Dies ist sinnvoll, da so die ursprünglichen Startdomains nicht verlassen werden. Somit sind die Crawlergebnisse von mehreren Crawlvorgängen auf dem gleichen Datensatz besser vergleichbar. Auf einer fiktiven Webseite „www.webseite.co.uk“ sollen also Beispielsweise folgende Links ignoriert werden:
„www.anders.co.uk/home“, „www.webseite.de/123“, „www.webseite.uk“.
Diese Links hingegen sollten normal weiterverfolgt werden:
„www.webseite.co.uk/home“, „www.subdomain.webseite.co.uk/123“.
Genauso soll es auch möglich sein, ausschließlich externe Links zu verfolgen. Dies kann zum Beispiel sinnvoll sein, wenn versucht wird, so viele verschiedene Domains wie möglich zu erreichen.
- A7** Es soll eine maximale Suchtiefe angegeben werden können. Sind alle Webseiten besucht, ist der Crawlvorgang beendet. Die Startseiten besitzen die Tiefe 0. Die Webseiten, welche von den Startseiten aus erreicht wurden, haben dann entsprechend die Tiefe 1. Setzt man als maximale Tiefe 0, so werden nur alle Startseiten besucht und keine

Links extrahiert. Es findet dann also kein wirkliches Crawling mehr statt, sondern ein automatisiertes Besuchen aller Webseiten in einer vorgegebenen List.

- A8** Der Crawler soll horizontal skalierbar sein. Das heißt, es soll möglich sein mehrere Taintfox Instanzen zu starten, die alle an dem Crawlprozess teilnehmen. Dies soll auch möglich sein, wenn diese Instanzen auf unterschiedlichen Rechnern laufen. Um den Crawlprozess zu beschleunigen, kann also nicht nur die Hardware eines Computers verbessert werden (vertikale Skalierung), sondern es können auch mehr Rechner hinzugefügt werden.
- A9** Es soll möglich sein den Crawlprozess zu stoppen und an einem späteren Zeitpunkt fortzusetzen. Wird der Crawler gestoppt, so gibt es einige Webseiten, die schon an eine Taintfox Instanz geschickt wurden und dort gerade geladen werden, aber noch keine Ergebnisse zurück geliefert haben. Diese Seiten sollen beim Fortsetzen des Crawlvorgangs wieder als unbesucht eingestuft werden, sodass sie nochmals an eine Taintfox Instanz verteilt werden.
- A10** Das Laden von sehr vielen Webseiten hintereinander kann die Netzwerkverbindung oft stark belasten. Darum kann es passieren, dass einige von ihnen vom Taintfox nicht erreicht werden können und es einen entsprechenden Timeout-Fehler gibt, obwohl die Webseite eigentlich online ist. Damit diese Webseiten nicht durch einen Timeout übersprungen werden, soll der Webcrawler sich alle Webseiten, auf denen ein Fehler aufgetreten ist, merken. Später können diese dann erneut besucht werden.

3.2 Bestehende Umgebung

In diesem Abschnitt wird nun zuerst die bestehende Umgebung beschrieben. Damit sind die einzelnen Komponenten und Tools, die bereits existieren, und deren Zusammenarbeit gemeint. Die Abbildung 3.1 zeigt ein Diagramm mit den einzelnen Komponenten. Wie zu sehen ist, existiert bereits ein Plugin für den Taintfox mit dem Namen „Taintnotifier“. Dieses Plugin reagiert auf das `taintreport` Event des Taintfox. Um zu überprüfen, ob es sich bei dem gefundenen Taintflow um eine echte Schwachstelle handelt, die ausgenutzt werden kann, gibt es einen extra Validierungsservice. Das Plugin sendet diesem die potenzielle Schwachstelle per HTTP. Der Service verwendet PhantomJS, um verschiedene Angriffe auszuprobieren, ohne einen kompletten Browser nutzen zu müssen. Sobald die Validierung abgeschlossen ist, werden die Ergebnisse an das Plugin zurückgeliefert. Alle Taintflows, die auf einer Webseite gefunden wurden, werden zusammen mit den entsprechenden Validierungsergebnissen in den Entwicklertools angezeigt.

Das Taintnotifier Plugin bietet außerdem die Möglichkeit, einen export Server anzugeben. An diesen werden per Http alle Funde gesendet, wo sie dann in einer MongoDB gespeichert werden. Später können diese Informationen dann über eine Weboberfläche abgerufen werden.

Dieser Aufbau erlaubt es einem Benutzer das Taintnotifier Plugin zu installieren und spezielle Webseiten gezielt aufzurufen, um diese auf Schwachstellen zu testen. Wie in der Aufgabestellung bereits angesprochen, soll dieser Prozess nun automatisiert werden, damit viele Webseiten in kurzer Zeit überprüft werden können.

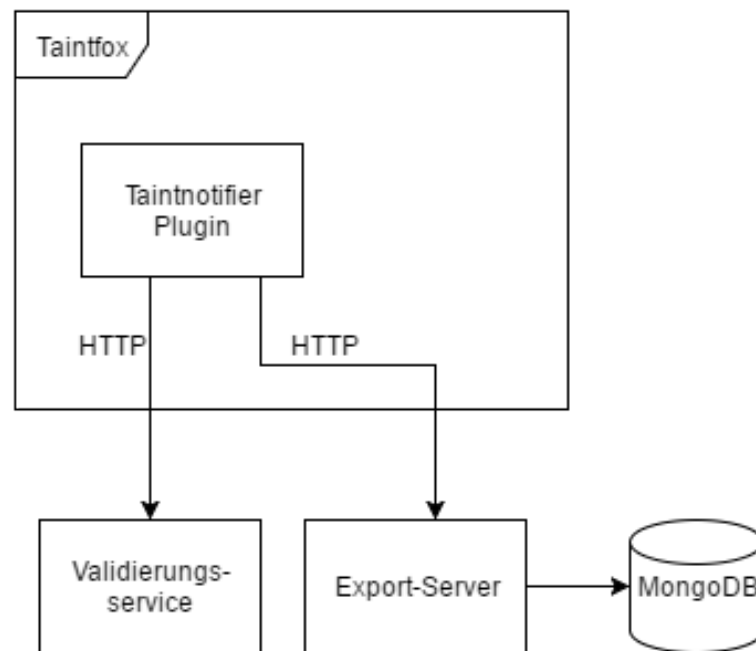


Abbildung 3.1: Diagramm mit Komponenten der bestehenden Umgebung

3.3 Webrawler Architekturen

Anforderung A8 verlangt, dass der Crawlvorgang beschleunigt werden kann, indem mehrere Taintfox Instanzen gestartet werden, die alle am Crawlprozess teilnehmen. Jeder dieser Instanzen wird als Arbeiter bezeichnet.

Es handelt sich also um „Distributed Web Crawling“, da viele einzelne Prozesse verwendet werden, um das Crawlen zu beschleunigen. Cho and Garcia-Molina haben in diesem Umfeld 3 Typen von verteilten Crawlern erforscht:

Independent Bei diesem Typ von Crawlern sind die einzelnen Arbeiter vollkommen unabhängig voneinander und es findet keinerlei Kommunikation statt. Jeder Arbeiter erhält zu Beginn eine Liste von URLs, die er besucht, ohne sich mit anderen Arbeitern zu verständigen. Dabei kann es dazu kommen, dass es Überlappungen gibt und die gleiche Webseite von verschiedenen Arbeitern besucht wird. Dieser Ansatz ist nur sinnvoll, wenn davon ausgegangen werden kann, dass die Anzahl diese Überlappungen nicht signifikant ist.

Dynamic assignment Bei diesem Aufbau gibt es zusätzlich zu den Arbeitern einen zentralen Koordinator. Dieser hält fest, welche Webseiten schon besucht wurden und speichert die URLs der noch unbesuchten Webseiten. Die Arbeiter erhalten dann nur noch Arbeitspakete mit einer Liste von URLs, die untersucht werden sollen. Hat ein Arbeiter dies getan, schick er alle gefundenen URLs zurück. Die Arbeiter nehmen hier also eine sehr passive Rolle ein und bearbeiten nur die Anfragen des Koordinators. Es kann zusätzlich eine dynamische Lastverteilung implementiert werden. Ist ein Arbeiter beispielsweise besonders schnell, so können ihm größere Arbeitspakete zugewiesen werden. Falls nötig, können sogar zur Laufzeit weitere Arbeiter hinzugefügt oder abgeschaltet werden.

Problematisch ist bei diesem Ansatz nur, dass alle Anfragen über den Koordinator gehen und dieser dadurch schnell zum Flaschenhals der Anwendung wird. Das System kann dann nicht mehr über die Anzahl der Arbeiter horizontal skaliert werden, da der Koordinator nicht schnell genug neue Arbeitspakete verteilen kann.

Static assignment Hier wird kein zentraler Koordinator benötigt, da es eine feste Regel gibt, die zu jeder URL einen Arbeiter zuweist. Es kann zum Beispiel eine Hashfunktion auf die URL angewendet werden, um dann jedem Arbeiter einen festen Bereich im Werteraum des Hash

zuzuweisen. Dadurch ist es dann möglich festzustellen, von welchem Arbeiter eine bestimmte URL verarbeitet werden soll. Da es passieren kann, dass ein Arbeiter eine URL findet, die einem anderen Arbeiter zugewiesen ist, muss es eine Möglichkeit geben, die URLs zwischen den Arbeitern auszutauschen.

Im Rahmen dieses Projektes wurde sich für einen verteilten Crawler mit dynamischer Zuweisung (dynamic assignment) entschieden. Komplett unabhängige Arbeiter kommen nicht in Frage, da die Überlappung zu hoch ist. Eine statische Zuweisung ist schwer umzusetzen, da die einzelnen Arbeiter dafür untereinander kommunizieren müssten.

Bei der dynamischen Zuweisung kann der Koordinator zwar zum Flaschenhals werden, allerdings ist dies unwahrscheinlich, da die Analyse einer Webseite oft mehrere Sekunden in Anspruch nimmt. Dadurch brauchen die Arbeiter vergleichsweise lange, um ein Arbeitspaket zu bearbeiten und der Koordinator wird nicht überlastet.

Die Aufteilung in einen Koordinator-Server und vielen Arbeitern ist außerdem Hilfreich, um die Anforderung A5 umzusetzen. Dabei geht es darum, die Analyseergebnisse von jeder Webseite an einer zentralen Stelle zu speichern. Die einzelnen Arbeiter können dafür einfach die Daten an den Koordinator schicken, welcher sie dann in einer Datenbank ablegt.

3.4 Neue Umgebung

Aufgrund der Anforderungen des Webcrawlers muss die bestehende Umgebung geändert werden. Abbildung 3.2 zeigt ein Diagramm mit allen wichtigen Komponenten und wie diese miteinander interagieren. In den folgenden Abschnitten werden nun die einzelnen Komponenten genauer beschrieben.

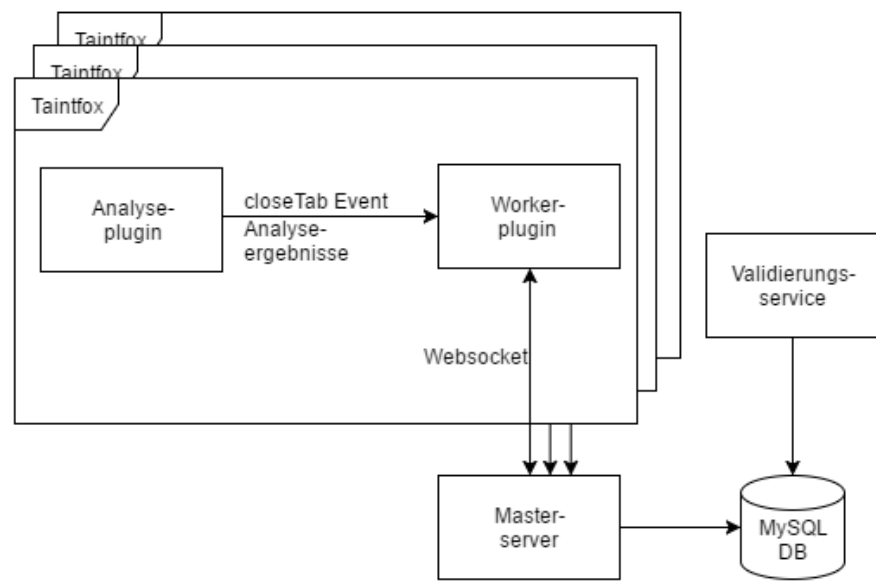


Abbildung 3.2: Diagramm mit Komponenten der neuen Umgebung

3.4.1 Master-Server

In Abschnitt 3.3 wurde sich für ein Crawler mit dynamischer Zuweisung entschieden. Bei dieser Crawlerarchitektur gibt es einen zentralen Koordinator. Diese Rolle übernimmt in der neuen Umgebung der Master-Server. Er steuert den Crawlprozess, indem er alle bereits besuchten URLs speichert, die Warteschlange mit neuen URLs verwaltet, Arbeitspakete an die Arbeiter verteilt und die Analyseergebnisse jeder besuchten Webseite speichert. Realisiert wird der Master-Server durch eine NodeJS Anwendung. Der Master-Server ist an eine Datenbank angebunden, die verwendet wird, um Daten wie die bereits besuchten Seiten, unbesuchte Seiten und Analyseergebnisse permanent zu speichern. Dies ist nötig, da der Webcrawler, wie in Anforderung A9 beschrieben, nach einem Neustart an der Stelle weiter machen soll, an der er aufgehört hat.

3.4.2 Worker-Plugin

Innerhalb des Taintfox kann man im Diagramm 3.2 eine Komponente mit dem Namen „Worker-Plugin“ sehen. Dabei handelt es sich um ein Firefox Plugin, welches als Arbeiter für den Webcrawler fungiert. Es nimmt die Arbeitspakete entgegen und weist den Taintfox an, die einzelnen Webseiten zu besuchen. Im Diagramm ist angedeutet, dass es mehrere Taintfox Instanzen mit entsprechenden Worker-Plugins geben kann. Jeder dieser Arbeiter meldet sich bei seinem Start bei dem Master-Server an.

Für die Kommunikation zwischen Arbeiter und Master kommen zwei Protokolle in Frage: HTTP und Websocket. Bei HTTP handelt es sich um ein Request/Response Protokoll. Der Klient stellt also eine Anfrage an den Server und bekommt von diesem eine Antwort. Jeder Arbeiter müsste also ein Polling durchführen, um neue Arbeitspakete zu erhalten. Das bedeutet, sobald ein Arbeiter sein Arbeitspaket bearbeitet hat, muss er in gewissen

Zeitabständen den Master nach einem neuen Fragen.

Polling ist generell keine besonders schöne Lösung. Abhilfe schafft das WebSocket Protokoll. Hier ist es dem Server möglich direkt Nachrichten an einen Klienten zu senden, ohne das dieser eine explizite Anfrage gestellt hat. Diese Art des Datenaustausches nennt man „pushing“. Für dieses Projekt wurde sich für Websockets entschieden, damit eine bidirektionale Verbindung zwischen Master und Arbeiter möglich ist.

3.4.3 Analyse-Plugin

Der Master-Server und das Worker-Plugin stellen die Hauptkomponenten des Crawlers da. Diese können unabhängig betrieben werden, um eine Reihe von Webseiten zu besuchen. Wie in Anforderung A4 beschrieben, soll es aber auch die Möglichkeit geben, ein Analyse-Plugin zu verwenden, welches die besuchten Webseiten auf bestimmte Schwachstellen untersucht. Die konkrete Implementierung des Analyse-Plugins hängt dabei vom Anwendungsfall ab. Für diese Arbeit wird ein Analyse-Plugin erstellt, welches die taintreport Events des Taintfox sammelt, um diese später auf Cross-Site-Scripting Schwachstellen zu prüfen.

Das Analyse-Plugin wird zusätzlich zum Worker-Plugin im Taintfox installiert. Sobald die Analyse abgeschlossen ist, muss dem Worker-Plugin mitgeteilt werden, dass diese Webseite nun geschlossen werden kann, sodass die nächste Seite aufgerufen wird. Anforderung A5 beschreibt die Möglichkeit, dass dabei Analyseergebnisse zu dieser Webseite mitgegeben werden. Diese werden dann vom Worker-Plugin weitergeleitet zum Master. Die Kommunikation zwischen den beiden Plugins findet dabei über ein internes Eventsystem im Taintfox statt.

3.4.4 Validierungsservice

In der vorherigen Umgebung wurde jedes Analyseergebnis auch gleich vom Validierungsservice überprüft, um festzustellen, ob es sich um eine tatsächliche Sicherheitslücke handelt. Da dieser Prozess aber auch unabhängig vom Crawling stattfinden kann und es nur unnötig verlangsamt, wurde er ausgelagert. Während des Crawlprozesses werden die potentiellen Lücken, die das Analyseplugin findet, an den Master-Server weiter gereicht, welcher sie in der Datenbank speichert. Dort können sie später von dem Validierungsservice abgerufen werden, ohne den Crawlprozess zu stören. Auch der Validierungsservice ist, wie das Analyse-Plugin, abhängig von dem konkreten Anwendungsfall. Da in dieser Arbeit Cross-Site-Scripting Schwachstellen aufgedeckt werden sollen, wird der bestehende Validierungsservice mit dem PhantomJs Backend verwendet. Es sind allerdings auch Anwendungsfälle denkbar, bei denen keine Validierung nötig ist, aber die Analyseergebnisse stattdessen anders weiterverarbeitet werden.

4 Implementierung

Nachdem im vorhergehenden Kapitel die einzelnen Komponenten der Webcrawler Architektur erarbeitet wurde, wird in diesem Kapitel nun die Implementierung des Master-Servers und Worker-Plugins genauer betrachtet. Diese beiden Komponenten stellen den Kern des Webcrawlers dar und sind unabhängig von dem speziellen Anwendungsfall, für den der Crawler eingesetzt werden soll.

4.1 Master-Server

4.1.1 Klassendiagramm

Die Klasse Crawler stellt die zentrale Klasse des Webcrawlers dar. Sie bietet die Methode *start*, mit der ein neuer Crawlprozess gestartet werden kann. Dabei muss ein Array mit entsprechenden Startwebseiten angegeben werden.

Für jeden Arbeiter, der sich bei dem Master-Server registriert hat, existiert eine Instanz der Klasse Worker. Diese kümmert sich um die Verwaltung der Websocketverbindung und speichert, wie viele Arbeitspakete momentan bearbeitet werden. Dadurch kann abgefragt werden, ob dieser Arbeiter noch ein weiteres Arbeitspaket verarbeiten kann.

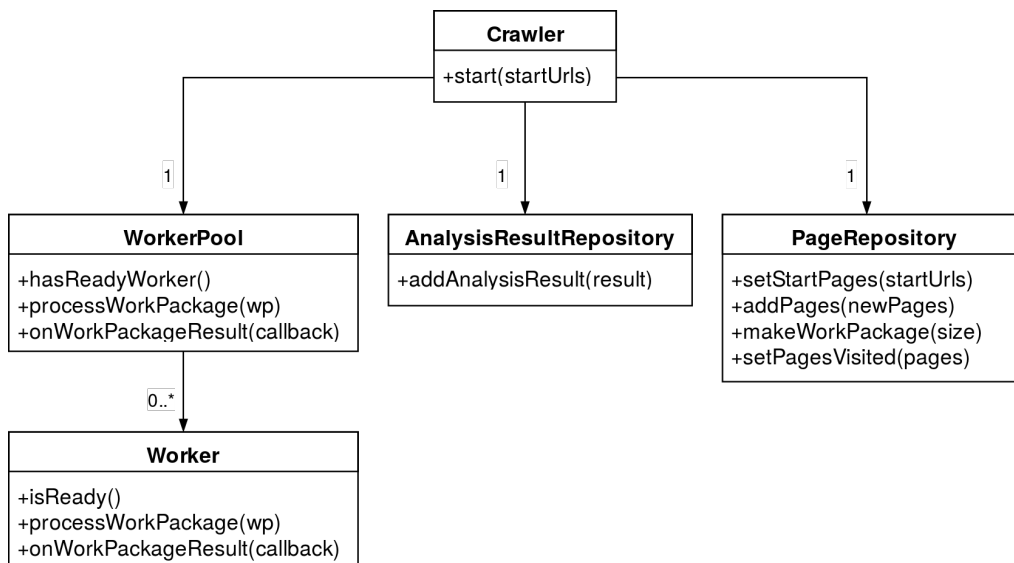


Abbildung 4.1: Klassendiagramm des Master-Servers

Alle Worker werden von der Klasse `WorkerPool` verwaltet, welche Auskunft darüber gibt, ob es momentan einen Worker gibt, der noch Arbeit benötigt und es durch die Methode `processWorkPackage` erlaubt ein Arbeitspaket zu verarbeiten. Dabei wird vom `WorkerPool` entschieden, welcher Arbeiter am besten dafür geeignet ist. An dieser Stelle kann also ein Loadbalancing eingebaut werden, um die Arbeiter gleichmäßig auszulasten.

Die Klasse `PageRepository` hat die Aufgabe alle gefundenen Webseiten zu verwalten. Mit Hilfe der „`addPages`“ Methode können neu gefundene Webseiten hinzugefügt werden. Diese sind dann zunächst als unbesucht markiert. Wird versucht eine Webseite hinzuzufügen, die bereits enthalten ist, wird diese ignoriert. Die Methode `makeWorkPackage` erlaubt es ein Arbeitspaket einer beliebigen Größe zu erstellen. Dabei werden entsprechend viele Webseiten, die noch als unbesucht markiert sind, zurückgeliefert. In dieser Funktion kann also definiert werden, welche Webseiten als nächstes bear-

beitet werden sollen. Es gibt verschiedene Strategien, wie dies bestimmt werden kann. Diese Strategien werden in Abschnitt 4.1.3 genauer beschrieben. Durch das Aufrufen der Methode *setPagesVisited* kann eine Liste von Webseiten als besucht markiert werden.

Mit welcher unterliegenden Technik die Klasse *PageRepository* implementiert wurde, ist dabei von außen nicht zu erkennen. Für dieses Projekt wurde sich für eine Implementierung mit einer MySQL Datenbank entschieden, da die gefundenen Webseiten so persistent gespeichert werden und der Crawlprozess somit auch nach einem Neustart des Master-Servers fortgesetzt werden kann. Es ist aber durchaus vorstellbar, zu einem späteren Zeitpunkt noch andere Implementierungen bereitzustellen. So zum Beispiel eine In-Memory-Datenbank, wodurch die Abfragen an das *PageRepository* beschleunigt werden können.

Die Klasse *AnalysisResultRepository* speichert die Analyseergebnisse jeder besuchten Webseite. Das Interface der Klasse ist sehr simpel und bietet nur die Methode *addAnalysisResult* mit der ein neues Analyseergebnis gespeichert werden kann. Wie auch bei dem *PageRepository* wird zum Speichern auf eine MySQL Datenbank zurückgegriffen. Es könnte stattdessen aber zum Beispiel auch eine MongoDB verwendet werden.

4.1.2 Datenmodell

Der Crawler verwendet eine relationale Datenbank, um seine Daten zu persistieren. Dabei werden 2 Tabellen verwendet. In der Tabelle „page“ werden alle gefundenen Webseiten gespeichert. Dazu gehören sowohl besuchte als auch unbesuchte Webseiten.

In der Tabelle „analysis_result“ werden die Analyseergebnisse der besuchten Seiten gespeichert. Da diese Ergebnisse von einem beliebigen Analyse-Plugin kommen, werden die eigentlichen Daten als JSON gespeichert. Da-

| Spaltenname | Beschreibung |
|----------------|---|
| id | Eine eindeutige Identifikationsnummer. |
| url | Komplette URL der Seite samt Query Parameter. Diese Spalte ist als unique markiert, sodass nicht 2 mal die gleiche Webseite eingefügt werden kann. |
| foundAt | Der Zeitpunkt zu dem die Seite gefunden wurde. |
| domain | Die Domain der Seite. Auf der Webseite https://www.dhbw-karlsruhe.de/allgemein/ ist dies zum Beispiel „dhbw-karlsruhe“ |
| topLevelDomain | Die Top-Level-Domain der Webseite. Zum Beispiel de, com, co.uk, net usw. |
| crawlDepth | Gibt an wie viele Links von der Startseite aus verfolgt wurden, um auf diese Seite zu gelangen. |
| visitStatus | Kann vier verschiedene Werte annehmen. Ist der Wert „visited“, so wurde diese Seite bereits besucht und muss nicht nochmal gecrawlt werden. Der Status „unvisited“ gibt an, dass die Webseite noch nicht besucht wurde und in ein Arbeitspaket gepackt werden kann. Den Zustand „pending“ erhält eine Seite, die zur Verarbeitung an einen Arbeiter geschickt wurde, bis eine Antwort erhalten wurde. Wird der Crawlprozess unterbrochen und später fortgesetzt, so werden beim Neustart des Master-Servers alle Webseiten mit dem Zustand „pending“ zurück auf „unvisited“ gesetzt. Der letzte mögliche Status ist „timedOut“. Dies bedeutet die Webseite konnte nicht erreicht werden (vgl. Anforderung A10). |

Tabelle 4.1: Datenmodell der Tabelle page

| Spaltenname | Beschreibung |
|-------------|--|
| id | Eine eindeutige Identifikationsnummer. |
| payload | Ein JSON Objekt, in welchem die spezifischen Analyseergebnisse gespeichert sind |
| page | Ein Fremdschlüssel für die Tabelle „pages“. Referenziert die Webseite, auf dem die Analyse durchgeführt wurde. |

Tabelle 4.2: Datenmodell der Tabelle analysis_result

durch ist nicht fest vorgegeben, welche Daten gespeichert werden können. Ein Eintrag in diese Tabelle wird nur gemacht, wenn auf einer Webseite tatsächlich Analyseergebnisse anfallen.

4.1.3 Crawl Strategien

Die PageRepository Klasse des Master-Servers erlaubt das Erstellen von neuen Arbeitspaketen. Es gibt dabei verschiedenen Strategien, wie entschieden wird, welche Webseiten als nächstes besucht werden sollen. Einige dieser Strategien und deren Implementierung auf der page-Tabelle werden im Folgenden vorgestellt.

Least Recent First

Bei dieser Strategie werden immer die Webseiten als nächstes besucht, die schon am längsten als unbesucht in der Tabelle markiert sind. Stellt man sich alle gefunden Seiten als Baum vor, so ähnelt diese Strategie einer Breitensuche. Es werden daher zuerst alle Seiten einer Crawltiefe bearbeitet, bevor die nächst tiefere Ebene abgearbeitet wird. Diese Strategie ist oft der Standard bei einfachen Webcrawlern, da sie einfach zu implementieren ist

und garantiert, dass zuerst eine Crawltiefe abgearbeitet wird, bevor tiefere Links in Betracht gezogen werden.

Das Listing 4.1 zeigt die Implementierung mit einer einfachen SQL-Abfrage. Es werden dabei alle Webseiten, die noch den Status unbesucht haben, aufsteigend anhand der Spalte foundAt sortiert. Mit dem Schlüsselwort LIMIT wird das Ergebnis dann auf die 10 ältesten URLs begrenzt. Somit erhält man ein Arbeitspaket mit 10 Elementen.

```
1 SELECT id , url FROM page
2 WHERE visitStatus='unvisited'
3 ORDER BY foundAt ASC
LIMIT 10
```

Listing 4.1: SQL-Abfrage für least recent first Strategie

Most Recent First

Diese Strategie stellt das Gegenstück zur vorherigen dar. Es werden die neusten Webseiten zuerst besucht. Dadurch gelangt man sehr schnell an die maximale Crawltiefe. Dieses Verfahren ist mit einer Tiefensuche vergleichbar. Vorteilhaft kann das sein, wenn man versucht schnell auf sehr tiefe Webseiten zu gelangen, ohne dabei unbedingt alle Webseiten einer bestimmten Crawltiefe zu besuchen. Das entsprechende SQL-Statement ähnelt dem aus Listing 4.1, allerdings wird absteigend sortiert.

Least Visited Domain

Hierbei handelt es sich um eine spezielle Strategie für den Anwendungsfall dieser Webcrawler. Es wird nach der Domäne gesucht, zu der es noch unbesuchte Webseiten in der Tabelle gibt, aber möglichst wenige bereits

besuchte Webseiten. Somit werden schnell viele Webseiten auf verschiedenen Domänen besucht. Dies findet Anwendung, wenn so schnell wie möglich viele Sicherheitslücken gefunden werden sollen, denn dabei ist es sinnvoll viele möglichst unterschiedliche Seiten zu besuchen.

Die SQL-Abfrage für diese Strategie, zusehen in Listing 4.2, ist etwas komplizierter und kann bei größeren Tabellen inperformant sein.

```
...
```

Listing 4.2: SQL-Abfrage für least visited domain Strategie

4.2 Worker-Plugin

Jeder Arbeiter des Crawlers ist ein Taintfox Instanz mit einem entsprechenden Worker-Plugin. Das Plugin verbindet sich per Websocket zum Master-Server und erhält von diesem Arbeitspakete mit Webseiten. Diese werden dann nacheinander im Browser geöffnet und alle vorhandenen Links werden extrahiert. Außerdem kann ein externes Analyse-Plugin Daten an das Worker-Plugin weiter geben. Sind alle Webseiten eines Arbeitspaketes abgearbeitet worden, so werden die neu gefundenen Links und gegebenenfalls vorhandene Analyseergebnisse an den Master-Server zurückgesendet.

4.2.1 Aufbau des Plugins

[<https://mdn.mozillademos.org/files/7873/content-scripting-overview.png>] Wie in Kapitel 2.3.3 bereits erklärt, besitzt ein Plugin einen Grundcode, in dem die Hauptlogik steckt, und mehrere Content Scripts, welche in beliebige Webseiten injiziert werden können, um zum Beispiel mit dem DOM zu interagieren. Für das Worker-Plugin werden 2 Content Scripts benötigt.

Das Content Script `extractLinks.js` wird in jede geöffnete Webseite injiziert und extrahiert alle darin befindlichen Links. Mithilfe eines Events werden diese zurück an den Grundcode geliefert, wo sie weiter verarbeitet werden. Der Grundcode selber hat keinen Zugriff auf eine API zur Websocketkommunikation. Aus diesem Grund wird ein Content Script verwendet, das die Standard Javascript Websocket API benutzt. Über entsprechende Events werden alle ankommenden und ausgehenden Nachrichten zwischen Grundcode und Content Script ausgetauscht. Da es keine spezielle Webseite gibt, in welche dieses Content Script eingebettet werden soll, wird ein so genannter Page Worker verwendet. Dies ist eine für den Benutzer unsichtbare Seite, welche für das Script genutzt werden kann.

4.2.2 Aufgetretene Probleme

In diesem Abschnitt werden kurz einige Probleme und deren Lösungen beleuchtet, die während der Entwicklung des Worker-Plugins aufgetreten sind. Diese waren dabei nicht in den Anforderungen aus Kapitel 3.1 vorhergesehen.

Bilder und andere Ressourcen

Nachdem eine Webseite im Taintfox geladen wurde, werden vom Worker-Plugin alle Links extrahiert. Die meisten dieser Links führen zu anderen Webseiten, die ebenfalls besucht werden sollen. Es gibt allerdings auch Links auf Bilder und andere Ressourcen, welche nicht geöffnet werden brauchen, da hier keine Analysedaten gewonnen werden können. Um das unnötige Laden von diesen Ressourcen zu vermeiden, wird die Eigenschaft „contentType“ des Tab-Objektes inspiziert. Diese beinhaltet den Internet Media Type

(MIME-Type) der zu ladenden Ressource, welcher die Art der Ressource genauer beschreibt. Interessant für das Crawling sind MIME-Types wie „text/html“ und „application/xhtml+xml“ welche auf HTML Seiten hinweisen. Wird hingegen zum Beispiel der MIME-Type „image/png“, „audio/mpeg“ oder „application/pdf“ vorgefunden, so muss diese Ressource nicht weiter verfolgt werden, da sie für das Analyse-Plugin unwichtig ist.

Downloads

Einige Webseiten starten über ein Skript automatisch Downloads. Dies führt dazu, dass der Taintfox ein separates Fenster mit Informationen zu diesem Download anzeigt, wie in Abbildung 4.2 zu sehen ist. Für den Crawler sind diese Downloads nicht relevant und werden ignoriert. Mit der Zeit werden dadurch immer mehr solcher Downloadfenster geöffnet.

Um dieses Problem zu umgehen, kann der Firefox so eingestellt werden, dass alle Dateien automatisch heruntergeladen werden, ohne den Benutzer in einem Popup danach zu fragen. Da die Dateien nicht tatsächlich heruntergeladen werden sollen, meldet das Worker-Plugin einen Listener an, welcher benachrichtigt wird, sobald ein neuer Download gestartet wurde. Dieser wird dann im nächsten Schritt sofort abgebrochen.

Zu viele Anfragen auf eine Seite

Ein Webserver kann so konfiguriert werden, dass er bei zu vielen Anfragen von der selben IP-Adresse den Fehler 429 „Too many requests“ zurück liefert. Um dieses Problem zu umgehen, sollten die Arbeitspakete möglichst unterschiedliche URLs enthalten, die jeweils zu einer anderen Domäne gehören.

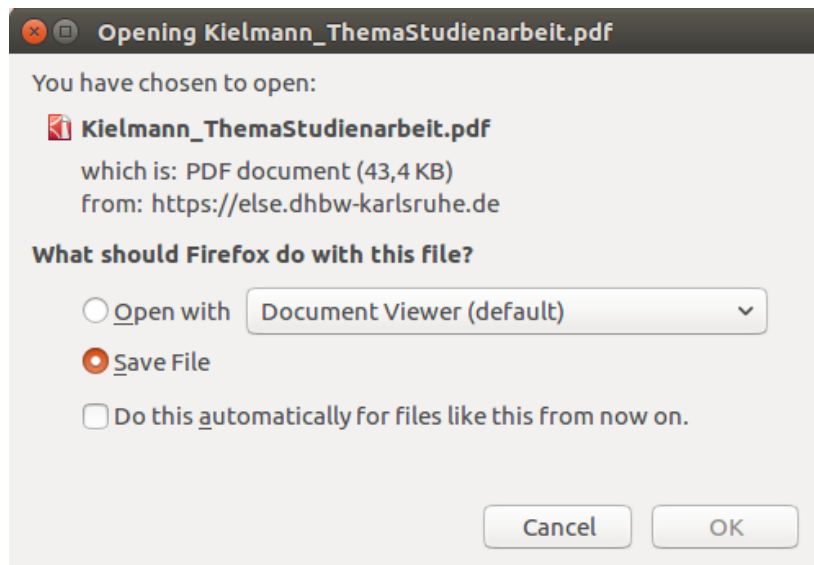


Abbildung 4.2: Störendes Popup bei Downloads

Taintfox Abstürze

Der Firefox und somit auch der Taintfox ist nicht dafür ausgelegt, eine große Menge von Webseiten hintereinander, für eine lange Zeitperiode, zu laden. Aus diesem Grund kann es hin und wieder zu einem Absturz kommen. Um ein manuelles Neustarten zu vermeiden, sollte der Taintfox mithilfe eines Skriptes so gestartet werden, dass bei einem Absturz der Neustart automatisch erfolgt. Während der Testdurchläufe hat sich ergeben, dass ein automatisierter Neustart des Taintfox alle 3-4 Stunden am besten funktioniert.

5 Anwendung

In diesem Kapitel wird der zuvor entwickelte Webcrawler eingesetzt, um eine große Anzahl von Webseiten auf DOM basierte Cross-Site Scripting Schwachstellen zu untersuchen. Dies dient einerseits dazu, die Fähigkeiten und Grenzen des Webcrawlers zu testen, andererseits sollen die gewonnenen Daten analysiert werden, um festzustellen, wie häufig DOM basiertes XSS auftritt. Zusätzlich wird erfasst, welche Quellen und Senken dabei am häufigsten sind. Im Rahmen dieser Arbeit wurden zwei Crawlvorgänge mit verschiedenen Parameter durchgeführt.

5.1 Alexa Top 500

Das kalifornische Unternehmen Alexa Internet veröffentlicht regelmäßig eine Liste mit den meist besuchten Webseiten im Internet. Dabei werden Daten wie die Anzahl der täglichen Benutzer und die durchschnittliche Zeit, die jeder Benutzer auf der Webseite verbringt, verwendet. Damit stellt die Alexa Top Websites Liste einen guten Datensatz für die Suche von Sicherheitslücken dar, denn es handelt sich um tatsächlich relevante Webseiten und die Ergebnisse sind durch den genau definierten Datensatz vergleichbar.

5.1.1 Webcrawler Konfiguration

Die ersten 500 Webseiten auf der Alexa Liste werden für den folgenden Crawldurchlauf als Start-URLs verwendet. Der Webcrawler wird so konfiguriert, dass nur interne Links auf den Webseiten verfolgt werden, um somit auf der gleichen Domäne zu bleiben. Dadurch wird sichergestellt, dass das Ergebnis nicht verfälscht wird und ausschließlich die Alexa Top 500 Webseiten besucht werden.

Es wird eine maximale Crawltiefe von zwei eingestellt. Da die Anzahl der zu besuchenden Webseiten exponentiell mit der Suchtiefe wächst, muss bei 500 Startseiten bereits hier mit einer Größenordnung im sechsstelligen Bereich gerechnet werden. Eine solche Anzahl von Webseiten ist für ein kleines Hardwaresetup schon sehr viel.

Der Webcrawler läuft auf einer virtuellen Maschine (Windows 10) mit 5 Gigabyte Arbeitsspeicher. Angebunden ist diese an eine 50 Mbit Internetleitung. Es werden 2 Taintfox Instanzen, mit jeweils 3 Tabs, als Arbeiter gestartet.

5.1.2 Daten zum Crawlvorgang

Es wurden insgesamt 3.346.905 Webseiten vom Crawler entdeckt, von denen 77.025 besucht wurden. Tabelle 5.1 zeigt die Anzahl der gefundenen und besuchten Seiten pro Crawltiefe. Auf der Crawltiefe 0 wurden alle 500 Startseiten besucht. Auch auf Ebene 1 wurden die meisten gefundenen Webseiten auch besucht. Die ca. 500 unbesuchten Webseiten sind Links auf fehlerhafte oder nicht mehr existierende Seiten. In Ebene 2 sind vergleichsweise nur sehr wenige Webseiten besucht worden. Dies liegt daran, dass der Crawlvorgang, aufgrund von Zeitmangel, frühzeitig beendet werden musste.

Für das Crawlen der 77.025 Webseiten wurden insgesamt ca. 3 Tage bzw.

| Crawltiefe | besuchte Seiten | gefundene Seiten |
|------------|-----------------|------------------|
| 0 | 500 | 500 |
| 1 | 72.232 | 72.734 |
| 2 | 4293 | 3.273.671 |
| gesamt | 77.025 | 3.346.905 |

Tabelle 5.1: Statistik der besuchten und gefundenen Seiten

72 Stunden benötigt. Bei dem gegebenen Setup, liegt die Performance des Webcrawlers also ungefähr bei 1000 Webseiten pro Stunde.

5.1.3 Daten zur Analyse

Während des Crawlvorgangs wurden für 69.997 von insgesamt 77.025 Webseiten ein Analyseergebnis erstellt. Das bedeutet auf 9,1% der Webseiten wurde überhaupt kein Datenfluss (Taintflow) gefunden. Jedes Analyseergebnis beinhaltet mehrere Taintflows. Insgesamt wurden 283.911 entdeckt. Pro Webseite, auf der es überhaupt einen Taintflow gibt, wurden im Durchschnitt also 4 solcher Flows gefunden.

Die Abbildung 5.1 zeigt die Verteilung der Quellen und Senken. Zuerst fällt ins Auge, dass *document.cookie* sowohl bei den Quellen, als auch bei den Senken den ersten Platz belegt. Gerade bei diesen Datenflüssen ist allerdings ein Angriff eher schwer. Die restlichen Quellen beziehen sich alle auf die URL oder einen Teil davon. Bei den Senken stehen *document.write* und *innerHTML* auf zweiter und dritter Stelle. Beide erlauben es, dynamisch HTML zu einer Seite hinzuzufügen. Unter „Rest“ sind alle anderen Senken enthalten, wie zum Beispiel eval.

Der Exploit-Generator, welcher für die verschiedenen Flows automatisch Angriffe generiert, kann dies nur für einen kleinen Teil der Quellen und Sen-

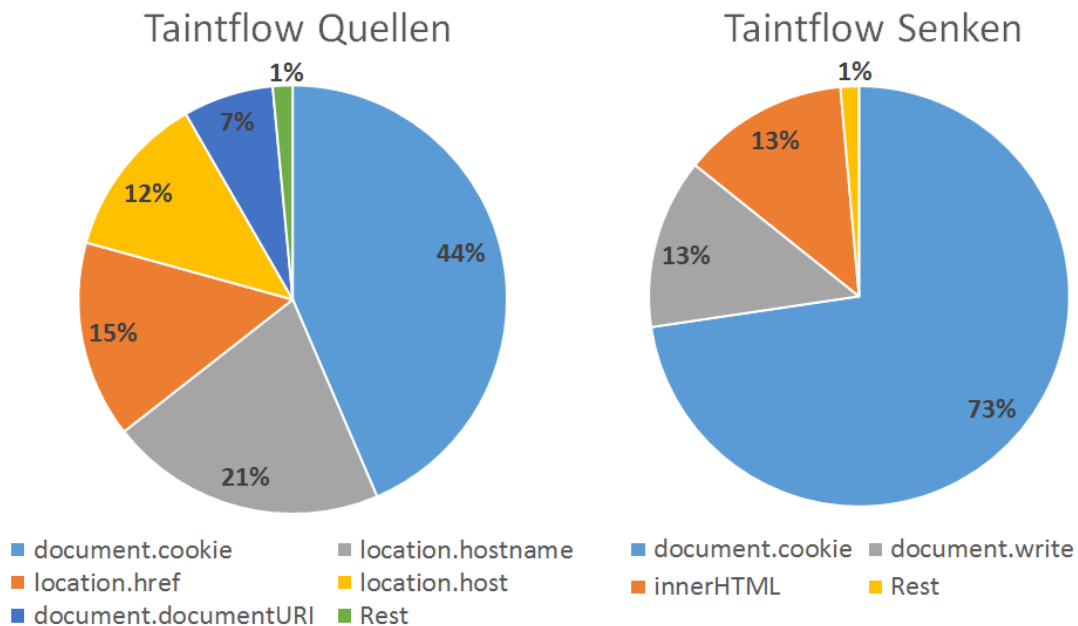


Abbildung 5.1: Verteilung der Quellen und Senken von allen Taintflows

ken machen. Aus diesem Grund werden alle Taintflows, welche eine Quelle oder Senke besitzen, für die kein Exploit generiert werden kann, herausgefiltert. Dabei entsteht eine Liste mit allen potentiell angreifbaren Taintflows, welche in diesem konkreten Fall 52.933 Elemente hat. Für 19% aller gefundenen Taintflows kann also automatisch ein Exploit generiert werden.

Die Diagramme in Abbildung 5.2 zeigen ebenfalls die Verteilung von Quellen und Senken, allerdings diesmal nur von den potentiell angreifbaren Taintflows. Als Quellen kommen hier nur noch *location.href* und *document.documentURI* in Frage. Beide dieser Eigenschaften verweisen auf die komplette URL der Webseite. Ein schädlicher Code könnte also zum Beispiel an das Ende der URL angehängt werden.

Die beiden Top Senken sind *document.write* und *innerHTML*. Die meis-

ten Cross-Site Scripting Möglichkeiten entstehen also beim dynamischen Einfügen von HTML. Das dynamische Ausführen von JavaScript über eval scheint hingegen sehr viel seltener einen kritischen Datenfluss zu erzeugen (nur 1%).

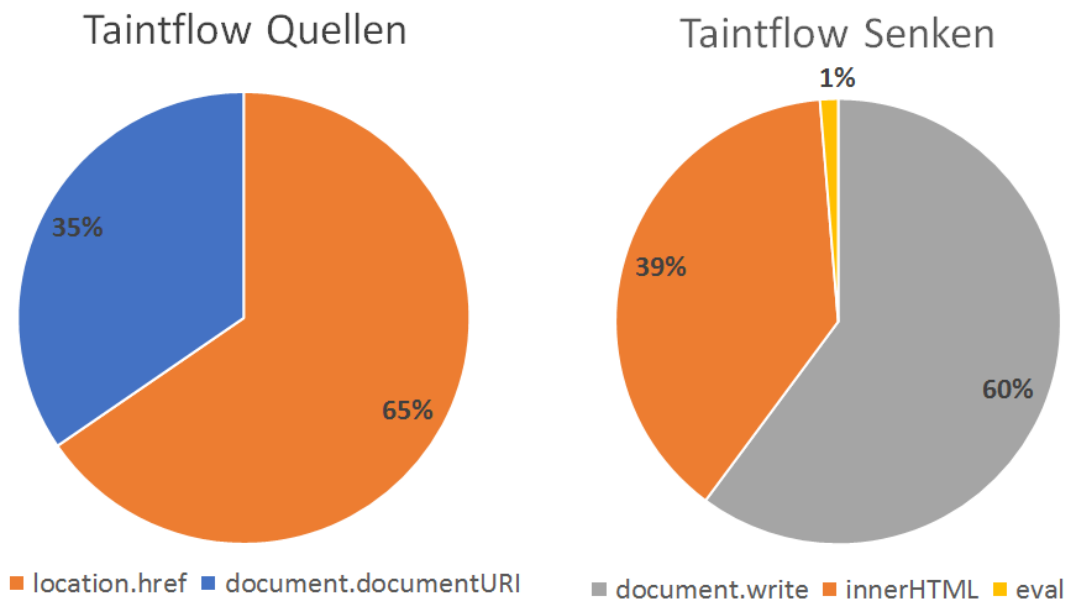


Abbildung 5.2: Verteilung der Quellen und Senken von potentiell angreifbaren Taintflows

Mithilfe des Validierungsservice, wurden alle 52.933 potenziellen XSS Schwachstellen überprüft. Das Ergebnis sind 180 erfolgreiche Exploits. Darin sind allerdings auch teilweise die gleichen Verwundbarkeiten auf verschiedenen Webseiten enthalten. Tritt auf unterschiedlichen Webseiten mit derselben Domäne, im selben Skript, in derselben Zeile ein Exploit auf, so sollte dies nur als eine Verwundbarkeit gezählt werden.

Nach einer entsprechenden Filterung liegen dann noch 31 ausführbare Cross-

Site-Scripting Angriffe, auf 15 verschiedenen Domänen vor. Es sind somit 3% der Alexa Top 500 Seiten angreifbar. Dies zeigt, dass XSS immer noch ein großes Problem darstellt, zumal hier nur auf einfache DOM basierte Angriffe getestet wurde. Es ist davon auszugehen, dass bei komplexeren XSS Angriffen noch mehr Seiten betroffen sind.

5.2 Webforum

In Abschnitt 5.1 wurde ein sehr beschränkter Crawldurchlauf der Alexa Top 500 Webseiten durchgeführt. Dies ist sinnvoll, da so ein fester Datensatz verwendet wurde. In einem zweiten Crawldurchlauf sollen nun viele möglichst verschiedene Seiten besucht werden. Die Hoffnung ist, dadurch so viele Sicherheitslücken wie möglich zu finden.

5.2.1 Webcrawler Konfiguration

Als Startpunkt des Crawlprozesses wird das Webforum reddit.com gewählt, da auf diesem viele externe Links vorhanden sind. Werden auf einer Webseite keine Schwachstellen gefunden, so ist es unwahrscheinlicher auf anderen Webseiten der selben Domäne fündig zu werden. Mit diesem Gedanken im Hinterkopf wird der Crawler so eingestellt, dass ausschließlich externe Links weiterverfolgt werden.

Die maximale Crawltiefe wird auf 7 gestellt, da nur eine einzige Startseite angegeben wurde und es in der Regel nicht so viele externe Links auf einer Seite gibt.

5.2.2 Daten zum Crawlvorgang

Am Ende des Crawlvorgangs wurden insgesamt 1.253.853 Webseiten entdeckt und 75.621 besucht. Das sind etwa gleich viele besuchte Webseiten, wie im vorherigen Versuch. Auf eine Tabelle mit den besuchten Seiten pro Suchtiefe wird hier verzichtet. Die Geschwindigkeit, mit der die Webseiten besucht wurden, ist gleich geblieben. Der Crawlvorgang wurde nach ca. drei Tagen gestoppt.

5.2.3 Daten zur Analyse

Von den 75.621 besuchten Webseiten gab es 43.677, die einen Taintflow enthalten. In diesem Versuch gibt es mit 42,2% deutlich mehr Webseiten ganz ohne Taintflows, verglichen zum Alexa Top 500 Crawl. Das lässt sich möglicherweise dadurch erklären, dass mehr Homepages besucht wurden, die oft nur statischen Inhalt besitzen. Insgesamt wurden 102.291 Taintflows gefunden.

Die Verteilung der Quellen und Senken ist sehr ähnlich wie im vorherigen Versuch. Daher wird diese hier nicht noch einmal genau aufgezeigt. Insgesamt wurden 10.424 potenziell angreifbare Taintflows gefunden, was 10% der gesamt gefundenen Flows entspricht.

Nach der Validierung bleiben 115 erfolgreiche Exploits übrig. Da sehr viele verschiedene Webseiten besucht worden sind, fällt die Anzahl der duplizierten Exploits auch deutlich geringer aus. Nach Filterung sind noch 95 Exploits übrig. Das sind dreimal so viele, wie bei den Alexa Top 500, obwohl es nur ein Fünftel der potenziellen Taintflows gibt.

Unter den gefundenen Webseiten befinden sich auch einige bekannte, wie zum Beispiel accuweather.com, unicef.fr, abc.es und www.telefonica.com.

Auffällig sind auch Skripte, die gleich auf mehreren unterschiedlichen Seiten für Schwachstellen sorgen. So gibt es Beispielsweise ein Skript eines Werbeanbieters, welches für acht Exploits auf unterschiedlichen Webseiten verantwortlich ist. Dieses Skript macht wahrscheinlich noch hunderte andere Webseiten unsicher.

6 Zusammenfassung

Die am Anfang aufgestellten Anforderungen konnten erfolgreich in einen flexiblen Webcrawler umgesetzt werden. Dieser besteht aus einem Master-Server, der den Crawlprozess verwaltet und entsprechende Daten persistiert und einem Worker-Plugin, welches Taintfox installiert wird, um dort Arbeitspakete vom Master-Server zu verarbeiten. Diese Aufteilung in einen Koordinator und viele Arbeiter ermöglicht es, die Performance des Crawlers durch das Hinzufügen von Arbeitern zu erhöhen.

Um den Webcrawler für verschiedene Anwendungsfälle universell einsetzbar zu machen, wurde die Analyseaufgabe in ein separates Analyse-Plugin verlagert, welches über Events mit dem Worker-Plugin kommuniziert. Durch diese lose Kopplung ist der Austausch des Analyseverfahren einfach zu bewerkstelligen.

Die Funktionalität des Webcrawlers wurde in zwei Experimenten getestet. Dabei ging es um das Finden von DOM basierten Cross-Site Scripting Schwachstellen. Hierbei wurde gezeigt, dass der Crawler auf mittlerer Hardware zufriedenstellende Ergebnisse liefert. Es wurden insgesamt 126 erfolgreiche Exploits gefunden. Diese Zahl zeigt, dass Cross-Site Scripting noch immer ein großes Problem darstellt. Mithilfe des neu entwickelten Webcrawlers wird es in Zukunft einfacher werden, diese Probleme zu erkennen und reduzieren.

6.1 Nächste Schritte

Die gestellten Anforderungen werden vom Entwickelten Webcrawler erfüllt. Allerdings ist die Entwicklung noch nicht Abgeschlossen und es gibt einige Bereiche, die noch verbessert und erweitert werden können.

Während der Entwicklung des Webcrawlers wurden nur zwei relativ kleine Crawldurchläufe gemacht. Bei diesen wurden jeweils unter 100.000 Webseiten besucht. Es sollten weitere Experimente gemacht werden, bei denen mehr Hardware und Zeit zur Verfügung steht, um die Grenzen des Webcrawlers zu testen. Gerade bei sehr vielen gefundenen Webseiten könnte die Verwaltung in einer einzigen relationalen Tabelle zu Performance-Problem führen. Die PageRepository Klasse des Master-Servers müsste in diesem Fall optimiert oder mit einer anderen Technologie implementiert werden.

Auf gewissen Webseiten wird der Klient beim Aufruf auf eine andere Webseite weitergeleitet. Der Crawler bekommt diese Umleitung nicht mit und geht davon aus, auf der ursprünglichen Webseite zu sein. In Zukunft sollte sich dieses Problem genauer angeschaut werden. Es muss definiert werden, was genau bei einer Umleitung passieren soll und wie dies dokumentiert wird.

Schlussendlich sollte ein Umstieg auf die Webextension API in Betracht gezogen werden. Zum Zeitpunkt der Implementierung des Webcrawlers haben diese noch nicht alle benötigten Funktionalitäten unterstützt, inzwischen ist die API aber vergleichbar mit der des Firefox Add-On SDKs.

Abbildungsverzeichnis

| | | |
|-----|--|----|
| 3.1 | Diagramm mit Komponenten der bestehenden Umgebung . . | 16 |
| 3.2 | Diagramm mit Komponenten der neuen Umgebung | 19 |
| 4.1 | Klassendiagramm des Master-Servers | 24 |
| 4.2 | Störendes Popup bei Downloads | 32 |
| 5.1 | Verteilung der Quellen und Senken von allen Taintflows . . . | 36 |
| 5.2 | Verteilung der Quellen und Senken von potentiell angreifbaren Taintflows | 37 |

Tabellenverzeichnis

| | | |
|-----|---|----|
| 4.1 | Datenmodell der Tabelle page | 26 |
| 4.2 | Datenmodell der Tabelle analysis_result | 27 |
| 5.1 | Statistik der besuchten und gefunden Seiten | 35 |