Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Eine der häufigsten Schwachstellen im Internet sind so genannte Cross-Site-Scripting Angriffe. Dabei wird versucht auf eine fremde Website Schadcode zu hinterlegen, welcher danach von anderen Benutzern unbeabsichtigt ausgeführt wird. Die Websecurity Abteilung von SAP versucht solche Schwachstellen automatisiert aufzudecken. Aus diesem Grund wurde ein modifizierter FireFox entwickelt, welcher es ermöglicht herauszufinden, welche Eingaben des Benutzers an kritischen Stellen in der Website gelangt.

1.2 Aufgabenstellung

Oft ist es Sinnvoll nicht nur eine einzige Seite auf Schwachstellen zu testen, sondern automatisiert eine Vielzahl von Seiten zu besuchen. Um dies zu ermöglichen soll im Rahmen dieser Arbeit ein Crawler entwickelt werden, der eine Liste von Startseiten besucht und aus diesen Verweise auf neue Seiten extrahiert. Dafür muss eine Architektur gefunden werden, die sich in das aktuelle System einfügen lässt und sich gut skalieren lässt.

2 Grundlagen

2.1 Crawler

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

2.2 Einsatzgebiete

Einer der bekanntesten Einsatzgebiete von Crawlern 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 diese Datenbank zu füllen und aktuell zu halten wird ein Crawler verwendet, wie zum Beispiel der Googlebot der Suchmaschine Google. [https://support.google.com/webmasters/answer/182072?hl=de]

2.3 Funktionsweise

Die Funktionsweise eines Webcrawlers lässt sich im wesentlichen auf einen einfachen Algorithmus reduzieren. Im ersten Schritt wird eine Seite identifiziert, welche heruntergeladen werden soll. Alle Seiten, 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 Seite entnommen wurde muss geprüft werden, ob das Crawlen dieser Seite 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 Seite

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 der Seite extrahiert. Diese neuen Links werden der Warteschlange aus dem ersten Schritt hinzugefügt, sofern sie nicht in der Liste mit besuchten Seiten enthalten sind. Somit wird verhindert, dass eine Seite mehrmals besucht wird.
Zum Schluss wird die heruntergeladene Seite analysiert und daraus gewonnene Informationen abgespeichert. Eine Suchmaschine könnte zum Beispiel alle Wörter die auf der Seite zu finden sind speichern. Danach beginnt der Prozess von die nächste zu besuchende Seite wird bestimmt. [http://www.ijcttjournal.org/Volume13/number-3/IJCTT-V13P128.pdf]

2.4 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 Crawlers im Arbeitsspeicher zu halten. Zum Beispiel die Liste mit bereits besuchten Seiten kann sehr schnell groß werden. Diese Daten auf ein sekundäres Speichermedium zu verlagern und dabei annehmbare Performance zu behalten ist keine leichte Aufgabe.

2.5 Cross Site Scripting

3 Architektur

3.1 Bestehende Umgebung

In diesem Abschnitt wird nun zuerst die bestehende Umgebung beschrieben. Die Abbildung XXX zeigt einen 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 validation service. 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 Seite gefunden wurden, und die entsprechenden Validierungsergebnisse werden dann in den Entwicklertools angezeigt.
Das Taintnotifier Plugin bietet außerdem die Möglichkeit einen export Server anzugeben. An diesen werden per Http alle Funde auf den besuchten Seiten gesendet. Dort werden sie in einer MongoDB gespeichert und können später über eine Weboberfläche abgerufen werden.
Dieser Aufbau erlaubt es einem Benutzer das Taintnotifier Plugin zu installieren und spezielle Seiten gezielt aufzurufen, um diese auf Schwachstellen zu testen. Wie in der Aufgabestellung bereits angesprochen, soll dieser Prozess nun automatisiert werden, damit viele Seiten in kurzer Zeit überprüft werden können.

3.2 Crawler Anforderungen

Die folgenden Anforderungen muss der Crawler erfüllen:

- 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 Rechner laufen.
- Der Crawlvorgang soll gestartet werden können, indem eine Liste von URLs angegeben wird, die als Startseiten verwendet werden.
- Nachdem der Crawler gestartet wurde soll dieser automatisch alle Startseiten besuchen und aus diesen weitere URLs extrahieren, die danach besucht werden.
- Auf jeder Seite, die vom Crawler besucht wird, soll ein Analyseplugin ausgeführt werden können. Erst wenn dieses mit der Bearbeitung der Seite fertig ist, darf sie geschlossen werden.
- Das Analyseplugin kann dem Crawler zu jeder besuchten Seite ein Datenpaket mit Analyseergebnissen übergeben. Diese Daten sollen dann zentral in einer Datenbank abgelegt werden.
- Es soll die Option auf allen Seiten keine externen Links zu verfolgen. Das heißt der Crawlvorgang wird auf die Domains der Startseiten beschränkt.
- Es soll eine maximale Suchtiefe angegeben werden können. Sind alle Seiten besucht, ist der Crawlvorgang beendet.

3.3 Crawler Typen

In diesem Kapitel wird eine Architektur für den Crawler entworfen. Dabei wird darauf geachtet, die Anforderungen aus Kapitel XXX zu erfüllen und gleichzeitig in die Bestehende Umgebung zu passen.

Anforderung XXX verlangt, dass der Crawlvorgang beschleunigt werden kann, indem mehrere Taintfox Instanzen gestarten werden, die alle am Crawlprozess teilnehmen. Dabei muss allerdings gewährleistet werden, dass keine Seite doppelt besucht wird. Die einzelnen Instanzen müssen sich also in irgendeiner Form darüber verständigen, wer für welche URLs zuständig ist. 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 Seiten von verschiedenen Arbeiter besucht wird. Dieser Ansatz ist nur sinnvoll, wenn davon ausgegangen werden kann, dass diese Überlappung insignifikant ist.

Dynamic assignment Bei diesem Aufbau gibt es zusätzlich zu den Arbeitern einen zentralen Koordinator. Dieser hält fest welche Seiten schon besucht wurden und speichert die URLs der noch unbesuchten Seiten. 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 Arbeitspakte verteilen kann.

Static assignment Hier wird kein zentraler Server benötigt, da es eine feste Regel gibt, die jede URL einem Arbeiter zuweist. Es kann zum Beispiel eine Hash-Funktion verwendet werden, um festzustellen von welchem Arbeiter eine URL verarbeitet werden soll. Da es passieren kann, dass ein Arbeiter eine URL findet, die einem anderen Arbeiter zugewiesen ist, muss es eine Möglichkeit 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 einzelne Arbeiter

dafür untereinander kommunizieren müssten.
Bei der dynamischen Zuweisung kann der Koordinator zwar zum Flaschenhals werden, allerdings ist dies unwahrscheinlich, da die Analyse einer Seite oft mehrere Sekunden dauert. 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 Arbeiter ist außerdem Hilfreich, um die Anforderung XXX aus Kapitel XXX umzusetzen. Dabei ging es darum die Analyseergebnisse von jeder Seite 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 Architektur Überblick

Abbildung XXX zeigt ein Diagramm mit allen Komponenten des Crawlers und wie diese miteinander interagieren. Wie in Abschnitt XXX erarbeitet wurde, gibt es einen zentralen Koordinator, welcher den Crawlprozess steuert. Im Diagramm heißt dieser Koordinator Master-Server. Er speichert alle bereits besuchten URLs, verwaltet die Warteschlange mit neuen URLs, verteilt Arbeitspakete an die Arbeiter und speichert die Analyseergebnisse jeder besuchten Seite. Realisiert wird der Master-Server durch eine NodeJS Anwendung. Im Diagramm ist außerdem das Worker-Plugin zu sehen. Dies ist der Arbeiter, welcher die Arbeitspakete entgegen nimmt und die einzelnen Seiten besucht. Er ist als Plugin innerhalb des Taintfox implementiert. Für die Kommunikation zwischen Arbeiter und Master kamen 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.

Zusätzlich zum Worker-Plugin muss im Taintfox ein Analyseplugin installiert werden. Dieses Plugin untersucht jede Seite, die vom Worker-Plugin geöffnet wird. Sobald die Analyse abgeschlossen ist, muss dem Worker-Plugin mitgeteilt werden, dass diese Seite nun geschlossen werden kann, sodass die nächste Seite aufgerufen wird. Dabei können Analyseergebnisse zu dieser Seite mitgegeben werden. Diese werden dann vom Worker-Plugin weitergeleitet zum Master.
Im Diagramm ist außerdem angedeutet, dass es mehrere Taintfox Instanzen geben kann, die jeweils ein Analyseplugin und Arbeiterplugin installiert haben. Wird eine neue Instanz gestartet, so meldet sich diese bei dem Master-Server an und wartet auf das erste Arbeitspaket.

4 Implementierung

4.1 Master Server

4.1.1 Klassendiagramm

Die Klasse Crawler stellt die zentrale Klasse des Crawlers da. 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 Arbeitspakte momentan bearbeitet werden. Dadurch kann abgefragt werden, ob dieser Arbeiter noch ein weiteres Arbeitspaket verarbeiten kann.
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 dann ein Arbeitspaket einer beliebigen Größe zu erstellen. Dabei werden entsprechend viele Seiten, die noch als unbesucht markiert sind, zurückgeliefert. In dieser Funktion kann also definiert werden, welche Webseiten als nächstes bearbeitet werden sollen. Es gibt verschiedene Strategien, wie dies bestimmt werden kann. Diese Strategien werden in Abschnitt XXX 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 diese Projekt wurde sich für eine Implementierung mit einer MySQL Datenbank entschieden, da die gefundenen Seiten 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 bereit zu stellen. 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 simple 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 "pages"werden alle gefundenen Seiten gespeichert. Dazu gehören sowohl besuchte als auch unbesuchte Seiten. id
Eindeutige Identifikationsnummer
url
Komplette URL der Seite samt Query Parameter
foundAt
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 drei verschiedene Werte annehmen. Ist der Wert "visited", so wurde diese Seite bereits besucht und muss nicht nochmal gecrawlt werden. Der Status ünvisited"gibt an, dass die Seite noch nicht besucht wurde und in ein Arbeitspakt gepackt werden kann. Der letzte Zustand ist "pending". Eine Seite, die zur Verarbeitung an einen Arbeiter geschickt wurde, erhält diesen Zustand bis eine Antwort erhalten wurde. Wird der Crawlprozess unterbrochen und später fortgesetzt, so wird beim Neustart des Master-Servers alle Seiten mit dem Zustand "pendingßurück auf ünvisited"gesetzt.

In der Tabelle „analysis_result“ werden die Analyseergebnisse der besuchten Seiten gespeichert. Da diese Ergebnisse von einem beliebigen Analyseplugin kommen, werden die eigentlichen Daten als JSON gespeichert. Dadurch ist nicht fest vorgegeben, welche Daten gespeichert werden können. Ein Eintrag in diese Tabelle wird nur gemacht, wenn auf einer Webseite tatsächlich Analyseergebnisse anfallen. id
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

4.1.3 Crawl Strategien

4.2 Worker Plugin

Jeder Arbeiter des Crawlers ist ein Taintfox Instanz mit einem entsprechendem 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 Analyseplugin 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 zurück gesendet.

Da der Taintfox nur eine modifizierte Version des Firefox ist, kann das Add-On SDK des Firefox verwendet werden, um das Plugin zu erstellen. Alternativ kann auch die WebExtension API verwendet werden, um Plugins zu erstellen. Diese sind zum größten Teil Browserunabhänig und laufen somit auch in Browsern wie Chrome, Opera und Microsoft Edge. Da das zu entwickelnde Plugin allerdings nur im Tainfox Sinn ergibt, bringt diese Unabhängigkeit in diesem Fall keine Vorteile. Darum wird in diesem Projekt das Add-On SDK verwendet, welches tiefere Integration mit dem Taintfox bietet. In der Zukunft muss allerdings der Umstieg auf WebExtensions in Betracht gezogen werden, da Mozilla, der Entwickler hinter dem Firefox, angekündigt hat das Add-On SDK in zukünftigen Versionen nicht mehr vollständig zu unterstützen.

4.2.1 Aufbau von Firefox Plugins

Der Code eines Firefox Plugin 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. Um genauer zu sein gibt es keinen Zugriff auf das DOM der geladenen Seite.
An dieser Stelle kommen sogenannte "Content Scriptsßum Einsatz. Diese haben Zugriff auf den Webinhalt und verhalten sich wie ein Javascript, welches auf der entsprechenden Seite eingebettet ist. Der Grundcode kann diese Content Scripts in beliebe Webseiten injizieren, um so gewisse Aufgaben auf dieser Seite zu erledigen. Die Kommunikation zwischen dem Grundcode und dem Content Script findet dabei über ein port Objekt statt, welches das asynchrone Versenden von Nachrichten zwischen den beiden erlaubt. Abbildung XXX illustriert wie die Methoden emit und on des port Objekts verwendet werden, um zu kommunizieren. [https://mdn.mozillademos.org/files/7873/content-scripting-overview.png]

4.2.2 Sequenzdiagramm

4.2.3 Firefox Plugin Aufbau

5 Anwendung

6 Fazit

Listings

Abbildungsverzeichnis

Tabellenverzeichnis