DOM-basiertes Cross-Site Scripting im Web: Reise in ein unerforschtes Land*

Ben Stock
Friedrich-Alexander-Universität
91058 Erlangen
ben.stock@cs.fau.de

Sebastian Lekies Martin Johns
SAP AG
76131 Karlsruhe
{sebastian.lekies|martin.johns}@sap.com

Abstract: Cross-site Scripting (XSS) ist eine weit verbreitete Verwundbarkeitsklasse in Web-Anwendungen und kann sowohl von server-seitigem als auch von client-seitigem Code verursacht werden. Allerdings wird XSS primär als ein server-seitiges Problem wahrgenommen, motiviert durch das Offenlegen von zahlreichen entsprechenden XSS-Schwächen. In den letzten Jahren jedoch kann eine zunehmende Verlagerung von Anwendungslogik in den Browser beobachtet werden eine Entwicklung die im Rahmen des sogenannten Web 2.0 begonnen hat. Dies legt die Vermutung nahe, dass auch client-seitiges XSS an Bedeutung gewinnen könnte. In diesem Beitrag stellen wir eine umfassende Studie vor, in der wir, mittels eines voll-automatisierten Ansatzes, die führenden 5000 Webseiten des Alexa Indexes auf DOM-basiertes XSS untersucht haben. Im Rahmen dieser Studie, konnten wir 6.167 derartige Verwundbarkeiten identifizieren, die sich auf 480 der untersuchten Anwendungen verteilen.

1 Einleitung und Motivation

Der Begriff *Cross-Site Scripting (XSS)* beschreibt eine Klasse von Injektionsangriffen, bei der es dem Angreifer möglich ist, eigenen HTML- und JavaScript-Code in die Seiten der angegriffenen Anwendung einzuschleusen. In ihrem Ranking der kritischsten Sicherheitsprobleme von Web-Anwendungen platziert das Open Web Application Security Project (OWASP) XSS auf Platz drei [Ope13] und das Sicherheitsunternehmen Whitehat Security berichtete in der diesjährigen Ausgabe ihres Sicherheitsreports, dass 43% aller Schwachstellen in Web-Applikationen in die Kategorie Cross-Site Scripting einzuordnen waren [Whi13a]. In der allgemeinen Literatur wird XSS meist als server-seitiges Problem behandelt [Eck04]. Dementsprechend beschränken sich die etablierten Gegenmaßnahmen meist auf server-seitige Filterung der potenziell unvertrauenswürdigen Nutzereingaben und der Bereinigung des finalen HTML/JavaScript-Codes über Output Sanitization, bevor dieser an den Web-Browser geschickt wird.

Derweil führt die Unterklasse des DOM-basierten XSS [Kle05], in dem die XSS-Lücke durch rein client-seitigen JavaScript-Code verursacht wird, ein Schattendasein, bedingt durch eine deutlich kleinere Angriffsoberfläche: Sowohl die Menge der client-seitigen

*Technische Grundlage dieses Beitrags ist die englischsprachige Verffentlichung "25 Million Flows Later - Large-scale Detection of DOM-based XSS" [LSJ13]. Im Vergleich bietet dieser Beitrag eine erweiterte Analyse der gesammelten Daten (siehe Abschnitt 5), die in dieser Form im Originalartikel fehlt.

Quellen von Daten, die vom Angreifer kontrolliert werden können, wie auch die Menge an potenziell unsicheren Sprachkonstrukten, die zu XSS Lücken führen, sind ungleich kleiner als auf der Serverseite.

Bis dato gab daher noch keine Studie, welche das Vorkommen dieser Art von Schwachstellen systematisch untersucht hat. Ziel der Arbeit war es, diesen recht unerforschten Schwachstellentyp strukturiert zu erforschen und mögliche Muster in den Verwundbarkeiten aufzudecken. Um eine groß angelegte Untersuchung dieser Art zu ermöglichen, wurden im Rahmen der Arbeit mehrere Komponenten zur automatisierten Analyse von Webseiten entwickelt. Anschließend wurde diese Infrastruktur genutzt, um die Alexa Top 5000 Webseiten in einem *Shallow Crawl* zu untersuchen. Insgesamt konnten im Rahmen dieser Untersuchung auf 480 der Alexa Top 5000 Webseiten Schwachstellen dieser Art festgestellt werden. DOM-basiertes XSS ist also eine relevante Schwachstelle im Web von heute.

Dieses Papier gliedert sich wie folgt: In Abschnitt 2 werden die technischen Hintergründe zur Same-Origin Policy, Cross-Site Scripting sowie im speziellen DOM-basiertem Cross-Site Scripting vorgestellt. Abschnitt 3 stellt dann die Kern-Komponenten des von uns entwickelten Systems vor, mit dem die Untersuchungen durchgeführt wurden. Im Anschluss werden in Abschnitt 4 die Vorgehensweise und die Ergebnisse der durchgeführten empirischen Studie diskutiert. Abschnitt 5 analysiert dann die Ergebnisse der Studie und zeigt die gewonnenen Erkenntnisse auf. Zuletzt werden verwandte Arbeiten erläutert (Abschnitt 6) und abschließend der wissenschaftliche Beitrag der Arbeit zusammengefasst (Abschnitt 7).

2 Technischer Hintergrund

Im Folgenden werden die Grundlagen zur Same-Origin Policy (SOP) und Cross-Site Scripting als Umgehung der SOP gelegt. Im Anschluss wird speziell auf DOM-basiertes Cross-Site Scripting eingegangen und die Kontext-Sensitivität von Cross-Site Scripting erläutert.

2.1 Die Same-Origin Policy und Cross-Site Scripting

Das grundlegendste, in Browser implementierte Prinzip zur Abgrenzung von Applikation voneinander ist die *Same-Origin Policy* (SOP) [Bar09]. Die SOP schützt Resourcen vor Fremdzugriff und erlaubt den Zugriff nur dann, wenn der *Origin* der beiden interagierenden Resourcen übereinstimmt. Der Origin ist dabei über die Kombination aus Protokoll, Domain und Port definiert.

Cross-Site Scripting beschreibt eine Reihe von Angriffen, die auf das Einschleusen von Script-Code abzielt, um die Same-Origin Policy zu umgehen. Gelingt es einem Angreifer, seinen eigenen JavaScript-Code durch eine Schwachstelle in eine Applikation einzuschleusen, wird dieser Code im Kontext der Applikation im Browser des Opfers ausgeführt. Konkret bedeutet dies, dass der Angreifer damit im Namen des Benutzers mit der Applikation interagieren kann. Das Schadpotenzial solcher Angriffe zeigt das aktuelle Beispiel der Seite `ubuntuforums.org`, von der 1,82 Millionen Benutzerdaten durch Ausnutzen einer Cross-Site Scripting-Schwachstelle gestohlen werden konnten [Whi13b].

Die in der Literatur erwähnten, altbekannten Arten von Cross-Site Scripting sind dabei

das persistente und reflektierte XSS. Bei ersterer Art hat ein Angreifer die Möglichkeit, seinen Schadcode beispielsweise in einer Datenbank zu persistieren, der dann bei jedem zukünftigen Besucher der Applikation ausgeführt wird. Beim reflektierten XSS hingegen entsteht die Verwundbarkeit dadurch, dass ein Teil der Anfrage des Clients in die Seite reflektiert wird. Ein Angreifer kann sein Opfer so in der verwundbaren Applikation auf eine präparierte URL locken, der der Schadcode angehängt ist. Sobald das Opfer diese Seite besucht, wird der Code des Angreifers ausgeführt.

2.2 DOM-basiertes Cross-Site Scripting

Neben den zwei bekannten Arten des Cross-Site Scriptings (XSS) wurde erstmal im Jahr 2005 von Amit Klein der Begriff des *DOM-basierten Cross-Site Scriptings* (DOMXSS) diskutiert. Klein fasste unter diesem Begriff jedwede XSS-Schwachstellen zusammen, welche auf Fehler im client-seitigen Code zurückzuführen sind. Dies grenzte DOMXSS wesentlich von den bereits erwähnten Arten ab, welche durch server-seitige Programmierfehler entstehen. Listing 1 zeigt eine solche Schwachstelle beispielhaft. Sowohl im Internet Explorer als auch in Googles Chrome werden Teile der URL, auf die per `location.href` zugegriffen wird, nicht automatisch enkodiert. Ein Angreifer kann sein Opfer auf eine Seite locken, welche etwa die Form

```
http://domain.de/seite#"></script>SchadCode
```

hat. Da mittels `href` die ganze URL an die entsprechende Stelle im Dokument geschrieben wird, kann der Angreifer hiermit aus dem bestehenden `script` ausbrechen und seinen eigenen Schadcode ausführen. Da dieser im Kontext der verwundbaren Seite ausgeführt wird, kann der Angreifer nun im Namen des Benutzers mit der Applikation interagieren.

Das unterliegende Problem von Cross-Site Scripting lässt sich zu einem Datenfluss-Problem abstrahieren. Vom Angreifer kontrollierbare Daten, welche aus verschiedenen *Quellen* stammen, fließen im Verlauf der Ausführung des Codes in sicherheitskritische *Senken*. Sofern die Daten nicht ausreichend validiert oder enkodiert werden, können solche *Flüsse* für einen XSS-Angriff genutzt werden. Die von uns betrachteten Quellen und Senken entsprachen dabei denen im DOMXSS Wiki aufgelisteten [DHB].

2.3 Kontext-Sensitivität von Cross-Site Scripting

Unter dem Begriff Cross-Site Scripting wird häufig das Einschleusen von Script-Elementen verstanden, obwohl dies nur eine Art eines XSS-Angriffs ist. Werden die eingeschleusten Daten beispielsweise innerhalb von JavaScript-Code in einem Aufruf an `eval` genutzt, muss der Angreifer kein `script`-Element einschleusen. Stattdessen muss er den an `eval`

Listing 1 Beispielhafte Verwundbarkeit

```
document.write('<script src="//werbung.de/werbung.js?referrer='
    + document.location.href +'"></script>');
```

übergebenen Code derart manipulieren, dass der gewünschte Schadcode ausgeführt wird. Wie dies verdeutlicht, sind Verwundbarkeiten bei allen Arten von Cross-Site Scripting kontext-abhängig. Im erstgenannten Fall muss der Angreifer seine eingeschleusten Daten derart wählen, dass valides HTML entsteht, wohingegen im zweiten Fall valides JavaScript geschrieben werden muss. Obwohl diese beiden Arten von Senken sehr unterschiedlich sind, lassen sich alle Angriffe wie folgt abstrahieren:

$$\textit{Angriff} := \textit{Ausbruch-Sequenz} \| \textit{Schadcode} \| \textit{Kommentar-Sequenz}$$

Hierbei steht das Zeichen $\|$ für die Konkatenation zweier Strings. Im Falle eines Angriffs ist der Schadcode beliebig vom Angreifer zu bestimmen. Im Gegensatz dazu ist die Ausbruch-Sequenz hochgradig kontext-abhängig. Hier muss geprüft werden, ob sich, beispielsweise im Falle von HTML, die eingeschleusten Daten innerhalb einer Attribut-Zuweisung befinden. In diesem Fall muss zuerst aus der Zuweisung und anschließend aus der Deklaration des HTML-Knotens an sich ausgebrochen werden. Die Kommentar-Sequenz hingegen kann anhand der Senke – hier wird zwischen HTML- und JavaScript-Kontexten unterschieden – sehr einfach bestimmt werden und bedarf keiner genaueren Analyse der Daten, die in die Senke geschrieben werden.

3 Entwickelte Komponenten

Die von uns im Rahmen dieser Arbeit entwickelte Infrastruktur zur automatischen Erkennung von DOM-basierten Schwachstellen besteht aus drei Komponenten: einem modifizierten Browser, welcher Taint-Tracking unterstützt, einem Programm zur automatischen Generierung von prototypischen Exploits sowie einem Crawling-Modul, welches als Browser-Erweiterung implementiert wurde. Im Folgenden stellen wir diese Komponenten vor.

3.1 Erweiterung von Chromium um Taint-Tracking

Um die Erkennung von Datenflüssen aus Quellen in sicherheitskritische Senken zu ermöglichen, entschieden wir uns für die Implementierung eines dynamischen Taint-Tracking-Systems in einen bereits bestehenden Browser. Unsere Wahl fiel dabei auf Chromium, dem Open-Source-Gegenstück von Googles bekanntem Browser Chrome [Goo] . Im Vergleich zu einer automatisierten Test-Suite wie HTMLUnit [HTM] implementiert Chromium bereits alle aktuellen HTML5-APIs und wird in Bezug auf weitere Neuerungen zeitnah aktualisiert.

Die wichtigsten Anforderungen an das von uns implementierte Taint-Tracking waren neben dem korrekten Markieren aller Daten aus relevanten Quellen auch die ordnungsgemäße Weitergabe der Informationen bei Transformation der Strings wie beispielsweise dem Verbinden zweier Strings oder der Extraktion einzelner Buchstaben. Dazu wurden die Implementierungen von Zeichenketten in der V8 JavaScript-Engine sowie die in der Webkit-Rendering-Engine so erweitert, dass diese zusätzlich Taint-Information speichern konnten. Die Änderungen erlaubten es uns, für jeden Buchstaben einer Zeichenkette die Quelle zu speichern. Im Kontext von DOM-basiertem XSS betrachten wir lediglich 14 verschiedene Quellen (darunter alle Teile der URL, Cookies, postMessages und Web Storages). Daher war für die Speicherung der Quellidentifizierer lediglich ein Byte pro Buch-

stabe von Nöten. Alle String-Operationen wurden entsprechend modifiziert, um eine Weitergabe des Taints sicher zu stellen. Nach den Änderungen der String-Implementierungen wurden nun alle Quellen im Sinne von DOM-basiertem XSS so modifiziert, dass diese Taint-Informationen entsprechend angehängt wurden. Analog dazu wurden ebenfalls alle Senken erweitert, um bei einem Fluss eines getainteten Strings eine Meldung an unsere Browser-Erweiterung zu senden. Die Funktion der Erweiterung wird im Laufe dieses Kapitels näher erläutert.

Wie bereits erläutert sind im Sinne von DOM-basiertem Cross-Site-Scripting lediglich 14 Quellen relevant, welche schon durch vier Bits abgebildet werden können. JavaScript bietet Progammieren die Möglichkeit, bereits fest eingebaute Funktionen zum Kodieren von Zeichenketten zu nutzen – *escape*, *encodeURI* und *encodeURIComponent*. Jede dieser Funktionen kann verhindern, dass eine potenzielle Schwachstelle ausnutzbar ist, wenn sie im jeweils richtigen Kontext genutzt wird. Um zusätzlich zu den Quellen eines jeden Buchstabens auch feststellen zu können, ob eine der vorgenannten Funktionen genutzt wurde, um den String zu maskieren, wurden die drei niederwertigsten der verbleibenden Bits genutzt, um die Maskierungen zu vermerken.

3.2 Browser Extension

Nachdem unser Prototyp nun in der Lage war, Taint-Informationen anzuhängen, weiterzureichen und abschließend auch einen Flu in eine Senke zu melden, entwickelten wir eine Chrome Extension [Goo12], deren Aufgabe es war, die Taint-Informationen aufzubereiten and an unser zentrales Backend zu senden. Neben dieser Funktion wurde auch ein Crawler in der Erweiterung implementiert, um die Datengewinnung in groß angelegter, automatisierter Weise möglich zu machen. Die Funktionsweise des gesamten Systems inklusive erweitertem Browser zeigt Abbildung 1. Das Background-Script steuert dabei die einzelnen Tabs, in denen jeweils die zu analysierende Seite zusammen mit dem Content- und User-Script geladen wird. Das Content-Script hat gemeinsam mit dem User-Script die Aufgabe, alle Links einer gerade analysierten Seite einzusammeln und mit Hilfe des Background-Scripts an das Backend zu versenden. Ebenfalls ist es Aufgabe dieser Komponenten, die gemeldeten Flüsse an das Backend weiterzuleiten.

3.3 Automatische Generierung von Exploits

Mit den bisher vorgestellten Komponenten waren wir in der Lage, im großen Maße Daten über potenziell verwundbare Flüsse zu sammeln. Da allerdings neben den vom Browser zur Verfügung gestellten Kodierungsfunktionen häufig auch vom Webseiten-Entwickeler entworfene Filter-Funktionen genutzt werden, ist ein potenziell verwundbarer Fluss nicht gleichbedeutend mit einer Schwachstelle. Um dennoch zu verifizieren, ob es sich bei einem Fluss um eine Schwachstelle handelt, wurde im Rahmen dieser Arbeit daher ein Programm entwickelt, welches automatisch für einen gegebenen Fluss einen prototypischen Exploit generieren kann.

Wie bereits anfangs erläutert, ist die Generierung eines validen Exploits stark vom Kontext abhängig. Dementsprechend wurden im Rahmen der Arbeit eigene Komponenten für das Erzeugen von Ausbruch-Sequenzen für JavaScript- und HTML-Kontexte entwickelt. Im Falle von HTML bestehen diese Sequenzen typischerweise aus Zeichen, welche Attribut-

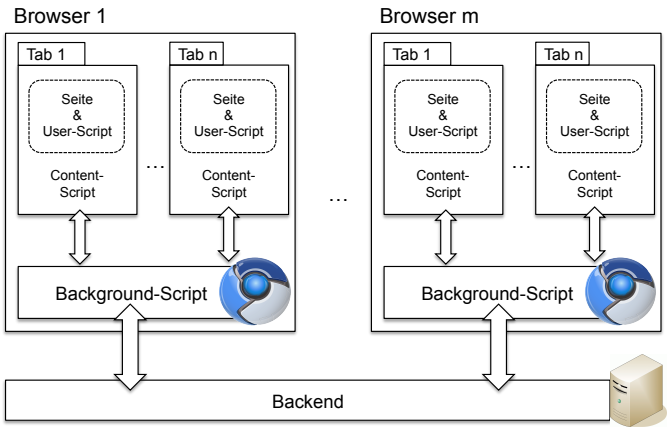

Abbildung 1: Übersicht der Crawler-Infrastruktur

Zuweisungen beenden, öffnende Deklarationen von HTML-Knoten schließen und gegebenfalls schließende HTML-Knoten anhängen. So interpretiert ein Browser beispielsweise jedwede Daten, die sich zwischen einem öffnenden und schließenden `iframe` befinden, nur dann, wenn keine Frames unterstützt werden. In diesen Fällen wurde vom Exploit-Generator eine entsprechende Ausbruch-Sequenz inklusive schließendem `iframe` generiert, um Ausführung des von uns gewünschten Codes zu garantieren.

Im Gegensatz zur Generierung von Ausbruch-Sequenzen für HTML-Kontexte, ist das Erzeugen von Ausbruch-Sequenzen für JavaScript komplizierter. Insbesondere ist hierbei zu beachten, dass im Falle eines Aufrufs an `eval` nur dann überhaupt Code ausgeführt wird, wenn der übergebene String syntaktisch korrektes JavaScript ist. Im ersten Schritt zur Generierung eines validen JavaScript-Exploits wird daher der abstrakte Syntaxbaum der Strings erzeugt. Innerhalb des so entstehenden Baumes wird dann die Position bestimmt, an denen getaintete Werte vorkommen. Ausgehend von dem gefundenden Ast wird der Baum nach oben durchlaufen und jeweils die Äste syntaktisch korrekt beendet. Listing 3 zeigt beispielhaft einen solchen Baum, generiert für den in Listing 2 gezeigten JavaScript-Code. Um einen validen Exploit zu erzeugen, muss zuerst das String-Literal mit `"` abgeschlossen werden. Anschließend wird die Deklaration der Variable `x` mittels `;` sowie der umgebende Block mit } beendet. Die entsprechende Ausbruch-Sequenz, auf die jetzt vom Angreifer gewählter Schadcode folgen kann, lautet also `";}`. In unserem Exploit-Generator wurden automatisch diese Ausbruchs-Sequenzen generiert, die im Anschluss

Listing 2 JavaScript-Auszug

```
var code =  'function test(){' +
      'var x = "' + location.href + '";' //innerhalb von test
    + '}'; //globaler Scope
eval(code);
```

Listing 3 Syntaxbaum zu Listing 2

```
FunctionDeclaration
  Identifier : test
  FunctionConstructor
    Identifier : test
    Block
      Declaration
        Identifier : x
        StringLiteral : "http://example.org"
```

im globalen Scope die Ausführung vom eingeschleusten JavaScript-Code ermöglichen.

4 Empirische Studie

Im Anschluss an die Implementierung der vorgestellten Komponenten führten wir eine empirische Studie durch, um die Anzahl der Verwundbarkeit, die sich auf DOM-basiertes Cross-Site Scripting zurückführen lassen, zu bestimmen. Dazu führten wir mit Hilfe unserer Crawler-Infrastruktur einen *Shallow Crawl* – das Besuchen der Startseite sowie aller auf der Startseite gefundenen Unterseiten – der Alexa Top 5000 Webseiten durch. Im Folgenden präsentieren wir die Vorgehensweise und Ergebnisse dieser Studie.

Für das Sammeln der potenziell verwundbaren Flüsse setzten wir insgesamt fünf PCs ein, welche jeweils mit vier Tabs die Alexa Top 5000 auf die vorgenannte Art besuchten. Im Zeitraum von fünf Tagen besuchten die Crawler dabei insgesamt 504.275 Webseiten. Im Durchschnitt beinhalteten diese Seite jeweils 8,64 Frames, wodurch sich die Anzahl der insgesamt untersuchten Dokumente auf insgesamt 4.538.031 belief. Auf diesen Seiten wurden insgesamt 24.474.306 Datenflüsse registriert und an das Backend gemeldet.

Aufgrund der überraschend hohen Anzahl an potenziell verwundbaren Flüsse und um die automatisierte Validierung in sinnvoller Zeit sicherzustellen, entschieden wir uns, nur für ausgewählte Flüsse prototypische Exploits zu generieren. Die Kriterien für die Auswahl waren dabei:

- Die Senke erlaubt direkte Ausführung von JavaScript. Dementsprechend wurden alle Flüsse in Quellen wie Cookies, WebStorage oder Attribute von DOM-Knoten ausgegrenzt.
- Die Quelle der Daten kann direkt vom Angreifer kontrolliert werden, muss also entweder Bestandteil der URL oder der Referrer sein.
- Lediglich solche Flüsse sollen analysiert werden, die nicht kodiert sind.

Auf diese Weise wurden insgesamt 313.794 Flüsse identifiziert, für die von unserem Generator prototypische Exploits erzeugt werden konnten. In den so entstandenen und zu besuchenden URLs mit angehängtem “Schadcode” befanden sich allerdings viele Duplikate, was darauf zurück zu führen ist, dass einzelne Dokumente oftmals den gleichen Script-Code an mehreren Stellen nutzen. Dementsprechend wird ein potenziell unsicherer Fluss mehrfach registriert. Nach Entfernung der Duplikate verblieben 181.238 eindeutige URLs,

die im Anschluss zur Verifizierung an unsere Crawling-Infrastruktur zurückgegeben wurden.

Um die Validierung der Schwachstelle automatisieren zu können, erweiterten wir die Chrome Extension um eine Funktion zur Exploit-Verifikation. Diese JavaScript-Funktion, welche über das Content-Script in jede besuchte Seite eingebunden wurde, erhielt als Parameter die eindeutige Identifikationsnummer eines Flusses und vermeldete diesen mit zusätzlichen Information wie der gerade geladenen URL an unser Backend. Somit konnten wir mit Hilfe des Exploit-Generators URLs mit prototypischen Exploits generieren, welche bei erfolgreichem Ausnutzen einer Schwachstelle `report(ID)` aufriefen. Die `ID` steht dabei für den numerischen Identifizierer eines Flusses in unserer Datenbank.

Im Anschluss wurden die Crawler erneut gestartet, um die generierten Exploit-Prototypen zu verifizieren. Auf 69,987 der 181.238 URLs, die von den Crawlern besucht wurden, wurde die `report`-Funktion erfolgreich aufgerufen und somit eine Schwachstelle nachgewiesen. Wie bereits erwähnt, wurden im Rahmen unseres initialen Crawls diverse Unterseiten der Alexa Top 5000 besucht. Viele der von uns besuchten Seiten setzten dabei Content-Management-Systeme ein, was den Verdacht nahe legt, dass Schwachstellen potenziell auf allen Unterseiten vorhandenen sind. Um ein genaueres Bild über die tatsächlich Anzahl der einzigartigen Verwundbarkeiten zu erhalten, wandten wir auf die URLs, die als verwundbar gemeldet wurden, ein Eindeutigkeitskriterium an. Dieses Kriterium bestand dabei aus dem Tupel

$$\{\text{Domain}, \text{Ausbruch-Sequenz}, \text{Code-Typ}, \text{Position im Code}\}\,.$$

Der *Code-Typ* unterscheidet dabei, ob der Script-Code *inline*, *eval* oder *extern* gespeichert war. Inline-Code meint dabei solchen JavaScript-Code, der direkt mittels `script`-Knoten innerhalb des Dokuments eingebettet ist, wohingegegen eval solchen Code meint, der innerhalb eines Aufrufs an `eval` ausgeführt wurde. Externer Code ist dabei Code, welcher per `script`-Element mit `src`-Attribut aus einer externen Datei nachgeladen wird. Die relevante Position im Code ist im Falle eines externen Scripts dabei sowohl die Zeile als auch die Position innerhalb der Zeile; für eval und inline Code lediglich die Position innerhalb der Zeile.

Nach Anwendung dieses Kriteriums wurde die Menge der einzigartigen Schwachstellen auf 8.163 auf insgesamt 701 Domains reduziert. Da jedoch im Rahmen des Crawls auch von solchen Seiten Frames nachgeladen wurden, die ausserhalb der Top 5000 lagen, analysierten wir die Daten erneut in Hinblick auf den Alexa-Rang. Dabei stellte sich heraus, dass sich die Anzahl der eindeutigen Verwundbarkeiten in den Alexa Top 5000 auf 6.167 (auf 480 Domains) belief. In Bezug auf die Anzahl der besuchten Domains bedeutet dies, dass sich auf 9,6% der 5000 meistbesuchten Webseiten mindestens eine DOMXSS-Schwachstelle befindet. Unter den verwundbaren Seiten befanden sich neben diversen Banken und einem sozialen Netzwerk auch verschiedene Seiten von Behörden sowie zwei Anti-Viren-Hersteller.

5 Erkenntnisse der Studie

In diesem Kapitel diskutieren wir die Erkenntnisse, die wir aus unserer durchgeführten Studie gewinnen konnten. Dabei geben wir eine detaillierte Analyse der Verwundbarkeiten

in Hinblick auf die genutzten Senken, die Herkunft des verwundbaren Script-Codes sowie einer allgemeinen Analyse der Anwendung von Kodierungs-Funktionen auf Daten der für DOMXSS relevanten Quellen.

5.1 Analyse der Senken

Tabelle 1 zeigt die Verteilung der Flüsse, die insgesamt für die Senken gemessen wurden im Vergleich zu Flüssen in diese Senken, welche verwundbar waren. Zudem bildet sie ab, wie die prozentuale Verteilung der Schwachstellen auf die drei von uns untersuchten Senken war. Auffällig ist dabei, dass offenbar in Fällen, in denen ein Datenfluss in `eval` endet, dieser zu einem größeren Anteil kodiert ist, wohingegen insbesondere im Falle von `innerHTML` in fast einem Viertel der Fälle gänzlich auf Kodierung verzichtet wird. Von allen Flüssen, die anhand unserer im Crawl gesammelten Daten als potenziell gefährlich erkannt wurden, konnten im Falle der HTML-Senken `innerHTML` und `document.write` bei jedem viertem Fluss eine Verwundbarkeit nachgewiesen werden. Im Gegensatz dazu konnten Flüsse, die in `eval` endeten, nur in 4% aller Fälle ausgenutzt werden. Dies legt die Vermutung nahe, dass Programmierer im Umgang mit `eval` mehr Gefahrenpotenzial sehen als mit HTML-Senken und dementsprechend häufiger die genutzten Daten kodieren bzw. validieren.

5.2 Herkunft des verwundbaren Codes

Der zweite Fokus bei der Auswertung der gewonnen Daten lag auf der Analyse der Verwundbarkeiten in Bezug auf ihre Herkunft. Häufig binden Webseiten Script-Inhalte – insbesondere im Falle von Werbung – von fremden Servern ein. Obwohl dieser Inhalt faktisch nicht den gleichen Origin im Sinne der Same-Origin Policy hat, wird der Script-Code dennoch im Kontext der Seite ausgeführt, welche den Inhalt eingebunden hat. Dies bedeutet, dass eine Verwundbarkeit in Drittanbieter-Code in dem Moment, in dem eine Seite diesen Code einbindet, zu einer Schwachstelle in dieser Seite führt. Unsere Untersuchung ergab, dass 13,09% aller Schwachstellen auf solchen, von Dritten bereitgestellten, Code zurückgingen (*cross-domain extern*). Der größte Anteil an Schwachstelle fiel mit 79,64% jedoch auf Scripte, die zwar aus einer separaten Datei nachgeladen wurden, deren Quelle allerdings die gleiche Domain wie die Seite selbst war (*same-domain extern*). Einen vergleichsweise kleinen Anteil machten mit 3,81% Inline-Scripte aus. Der Rest der Schwachstellen entstand durch Code, welcher mittels `eval` erzeugt und ausgeführt wurde. Aufgrund der Funktionsweise der JavaScript-Engine war es bei diesen verbleibenden Schwachstellen aber nicht möglich, die genaue Position (inline, cross-domain extern oder

Senke	Flüsse	unkodiert	ausnutzbar[1]	ausgenutzt[2]	eindeutig
eval	1.728.872	81.949	47.925	1.925 (4,02%)	319
document.write	2.587.206	291.837	170.793	44.049 (25,79%)	7.170
innerHTML	631.286	145.322	95.076	24.013 (25,26%)	674

[1] aus Quellen, die direkt vom Angreifer kontrolliert werden können
[2] Prozentangaben beziehen sich auf ausnutzbare Flüsse nach Spalte 4

Tabelle 1: Untersuchte Senken mit gesamten und exploitbaren Flüssen

same-domain extern) zu bestimmen.

5.3 Anwendung von Kodierungsfunktionen

Insgesamt zeichnete sich außerdem bei der Auswertung der Daten ab, dass es erhebliche Unterschiede in der Verarbeitung von Daten aus den unterschiedlichen Quellen gab. So wurden Daten aus der URL immerhin in fast 65% aller Fälle kodiert, bevor sie in einer der relevanten Senken endeten – beim Referrer, also der URL der Seite, über die der Benutzer auf die aktuelle Seite gekommen ist, wurden die Daten sogar in fast 84% aller Fälle kodiert. Im krassen Gegensatz dazu stehen Daten aus postMessages, welche nur in 1,57% aller Fälle kodiert waren. Die postMessage-API, welche im Zuge von HTML5 in Browser eingefügt wurde, erlaubt es Applikationen innerhalb eines Browsers mit Hilfe von Nachrichten zu kommunizieren. So lassen sich damit etwa zwischen einem geöffneten Popup und der öffnenden Seite Nachrichten auch über Origin-Grenzen hinweg austauschen. Die API erlaubt es der empfangenden Seite, den Absender der Nachricht zu überprüfen. Sofern diese Prüfung jedoch nicht korrekt oder gar nicht durchgeführt wird, kann ein Angreifer eine beliebige Nachricht an die empfangende Seite schicken. Obwohl wir in unserer Arbeit keine Verifikation von derartigen Schwachstellen durchgeführt haben, liegt der Verdacht – auch aufgrund von verwandten Arbeiten [SS13] – nahe, dass mehrere der von uns untersuchten Webseiten verwundbar sind.

Im Zuge unserer Untersuchung fanden wir zudem bei händischen Tests mit einer generierten Exploit-Payload eine Schwachstelle, in der zuerst lediglich ein Fluss in ein Cookie stattfand. Ein solcher Fluss ist noch nicht notwendigerweise verwundbar, allerdings wurde im konkreten Fall beim zweiten Laden der Seite dieser Cookie-Wert genutzt, um einen String zusammen zu setzen, welcher anschließend an `eval` übergeben wurde. Dementsprechend konnte beim zweiten Laden der Seite erfolgreich der Exploit ausgelöst werden. Dieses Beispiel verdeutlicht auch die Wichtigkeit der Benutzung von Kodierungsfunktionen, selbst wenn die zu kodierenden Daten nicht direkt in eine solche Senke, die sofortige Ausführung von Schadcode ermöglicht, fließen.

6 Verwandte Arbeiten

Das Konzept einer um Taint-Tracking erweiterten Browsing-Engine wurde erstmals bei DOMinator verfolgt [Di 12]. DOMinator bietet allerdings im Gegensatz zum von uns entwickelten Prototypen nicht die Möglichkeit, genaue Taint-Information zu jedem Zeichen eines Strings zu erhalten. Dementsprechend kann dieser auch nicht als Basis des von uns entwickelten, automatischen Exploit-Generators genutzt werden. Konzeptionell am nächsten zum vorgestellten Ansatz ist FLAX [SHPS10], welches ebenfalls byte-genaues Taint-Tracking nutzt, um unsichere Datenflüsse in JavaScript zu erkennen. Der große Unterschied besteht dabei allerdings darin, dass FLAX Teile des Programm-Codes in eine vereinfachte Variante von JavaScript überführt und anschließend auf den Teilresultaten nach potenziellen Verwundbarkeiten sucht. Im Gegensatz dazu erlaubt unser Ansatz eine Benutzung der gesamten JavaScript-Funktionalität. Criscione [Cri13] stellte im April 2013 ein von Google entwickeltes Systems vor, welches auf automatisierte Art und Weise nach Verwundbarkeiten sucht. Ähnlich zu dem von uns vorgestellten Ansatz werden hierbei instrumentarisierte Browser genutzt, wobei statt Taint-Tracking mit kontext-sensitiver

Exploit-Generierung Blackbox-Fuzzing eingesetzt wird. In Bezug auf die groß angelegte Analyse von Cross-Site Scripting-Verwundbarkeiten legten 2009 Yue und Wang [YW09] erste Grundlagen, in dem sie unsichere Programmierpraktiken von JavaScript untersuchten. Der Fokus bei dieser Arbeit lag jedoch nur auf der statitischen Auswertung von potenziell verwundbarem Code und nicht auf Verifikation der Schwachstellen. Richards et al. [RHBV11] untersuchten 2011 die unsichere Benutzung von `eval` und zeigten Wege auf, die gleiche Funktionalität auf sicherem Wege zu implementieren. Son und Shmatikov [SS13] untersuchten kürzlich, wie viele der Alexa Top 10.000 Webseiten bereits die HTML5-API postMessage nutzen. Dabei stellten sie fest, dass diese neue API bereits auf 22,5% der Top 10.000 Webseiten genutzt wurde, wobei insgesamt 84 Webseiten aufgrund fehlender Prüfung des Nachrichten-Absenders anfällig für Cross-Site Scripting-Angriffe waren.

7 Zusammenfassung

In diesem Beitrag stellten wir die Ergebnisse einer umfassend angelegten praktischen Studie vor, die sich mit dem Vorkommen von DOM-basiertem Cross-site Scripting befasst und die erste groß angelegte Untersuchung dieser Art von Schwachstellen darstellt.

Die technische Kernkomponente unserer Infrastruktur ist ein erweiterter Chromium-Browser, welcher mittels einer Chrome Extension instrumentarisiert wurde, um vollautomatisch Daten zu potentiell unsicheren Flüssen zu sammeln. Diese Daten wurden anschließend als Eingabe für einen Exploit-Generator genutzt, welcher prototypische Testfälle für die möglicherweise verwundbaren Flüsse generierte. Mit diesem System konnten wir auf 480 der Alexa Top 5000 Domains DOMXSS-Schwachstelle identifizieren.

Wie wir in Abschnitt 5 aufzeigen, erlaubt die Studie erste Einblicke in Muster von Programmierfehlern, die zur Entstehung der Schwachstellen beitragen. Das Hauptergebnis der Studie ist die Erkenntnis, dass DOM-basiertes XSS ein signifikantes Problem ist, das in vielen produktiven Webseiten auftritt und augenscheinlich von den aktuell verwendeten Methoden der sicheren Web-Programmierung nicht hinreichend verhindert wird. Dem zu Folge ist es notwendig, dass in der Zukunft die Ursachen dieser Verwundbarkeitsklasse verstärkt untersucht werden, um wirkungsvolle Gegenmanahmen entwickeln zu können.

Danksagungen

Wir danken Felix Freiling für hilfreiche Kommentare zu einer vorherigen Version dieses Textes. Zudem danken wir den anonymen Reviewern für die hilfreichen Kommentare.

Literatur

[Bar09] Adam Barth. The Web Origin Concept, November 2009.

[Cri13] Claudio Criscione. Drinking the Ocean - Finding XSS at Google Scale. Talk at the Google Test Automation Conference, (GTAC'13), `http://goo.gl/8qqHA`, April 2013.

[DHB] Stefano Di Paola, Mario Heiderich und Frederik Braun. DOM XSS Test Cases Wiki Cheatsheet Project. [online] `https://code.google.com/p/domxsswiki/wiki/Introduction`, abgerufen am 9.12.2013.

[Di 12] Stefano Di Paola. DominatorPro: Securing Next Generation of Web Applications. [software], https://dominator.mindedsecurity.com/, 2012.

[Eck04] Claudia Eckert. *IT-Sicherheit*. Oldenbourg, Muenchen [u.a.], 3., überarb. und erw. aufl. Auflage, 2004.

[Goo] Google. Google Chrome. [online], https://www.google.com/intl/de/chrome/browser/, abgerufen am 9.12.2013.

[Goo12] Google Developers. Chrome Extensions - Developer's Guide. [online], http://developer.chrome.com/extensions/devguide.html, abgerufen am 9.12.2013, 2012.

[HTM] HTMLUnit Development Team. HtmlUnit Webseite.

[Kle05] Amit Klein. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

[LSJ13] Sebastian Lekies, Ben Stock und Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, Seiten 1193–1204. ACM, 2013.

[Ope13] Open Web Application Security Project. OWASP Top 10 - 2013, Oktober 2013.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg und Jan Vitek. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In Mira Mezini, Hrsg., *ECOOP*, Jgg. 6813 of *Lecture Notes in Computer Science*, Seiten 52–78. Springer, 2011.

[SHPS10] Prateek Saxena, Steve Hanna, Pongsin Poosankam und Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*. The Internet Society, 2010.

[SS13] Sooel Son und Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[Whi13a] Whitehat Security. Website security statistics report, Mai 2013.

[Whi13b] Zack Whittaker. Ubuntu forums hacked; 1.82M logins, email addresses stolen. Online-Artikel, Juli 2013.

[YW09] Chuan Yue und Haining Wang. Characterizing insecure javascript practices on the web. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek und Wolfgang Nejdl, Hrsg., *WWW*, Seiten 961–970. ACM, 2009.