App Isolation

Ruhr-Universität Bochum
Master-Seminar Netz- und Datensicherheit

Sommersemester 2012

Oliver Domke
oliver.domke@rub.de

19. März 2013

Zusammenfassung

Viele aktuelle Angriffe im Internet basieren auf einem von zwei Prinzipien. Entweder werden Benutzer dazu verleitet, Webseiten über manipulierte URLs aufzurufen, oder nicht vertrauenswürdige Webseiten erlangen Zugriff auf gespeicherte Zustandsinformationen sicherheitskritischer Seiten. Eine oftmals empfohlene, aber nicht besonders komfortable Lösung ist die Verwendung mehrerer Browser für verschiedene Webseiten. Der vorgestellte Mechanismus *App Isolation* will nicht nur die Verantwortung vom Anwender auf den Anwendungsentwickler übertragen, sondern auch ein gleichwertiges Sicherheitsniveau in nur einem einzelnen Browser erzielen. Die Implementierung wahrt die Webseitenkompatibilität und verringert die Leistung des Browsers nur in einem vernachlässigbaren Maß. *App Isolation* vereint dabei eine Zustandsisolierung zum Schutz von hinterlegten Zustandsinformationen (wie etwa Cookies), sowie eine Einschränkung der Eintrittspunkte zu sicherheitskritischen Webseiten, damit schützenswerte Resourcen nicht über manipulierte Links aufgerufen oder in fremde Seiten eingebunden werden können.

Inhaltsverzeichnis

1 Einleitung

Webseiten sind heutzutage oft viel mehr als miteinander verlinkte, statische Dokumente. Vor allem in der jüngeren Vergangenheit haben Internetauftritte stark an Komplexität und auch Funktionalität zugenommen. Die hohe Interaktivität zwischen Seiten und ihren Benutzern, der Austausch von Daten (auch zwischen unterschiedlichen Anwendungen) sowie die rege Verwendung von clientseitig ausgeführtem Code erfordern ebenso ausgefeilte Sicherheitsmechanismen in den verwendeten Browsern, um beispielsweise das Ausspähen persönlicher Daten über ungewünschte Informationsflüsse zu verhindern.[2]

Die häufig von Sicherheitsexperten vorgeschlagene Option, mehrere Browser zu verwenden, um sicherheitskritische Seiten und den Rest des Internets in unterschiedlichen Umgebungen aufzurufen, führt oft nicht zuletzt durch den geringeren Komfort zu wenig Akzeptanz bei den Usern.[3] Und auch die Klassiker unter den browserseitig verwendeten Sicherheitsmaßnahmen, wie die *Same-Origin Policy*, sind längst nicht mehr ausreichend, um Webseiten bzw. -anwendungen adäquat zu isolieren, ohne gleichzeitig ihre Funktionen einzuschränken.[4, 5] Vielmehr müssen Entwickler völlig neue Browserarchitekturen implementieren, um eine sichere Ausführungsumgebung für Web-Anwendungen, die Desktop-Programmen durchaus immer ähnlicher werden, anzubieten.[5] Einer dieser Ansätze, *App Isolation*, ist das Ergebnis einer gleichnamigen Arbeit von E. Chen, J. Bau, C. Reis, A. Barth und C. Jackson[1], welche in dieser Seminararbeit thematisiert wird.

Die Autoren identifizieren in ihrer Arbeit zwei Hauptmerkmale, die bei der Benutzung von mehreren Browsern zur Sicherheit beitragen, um diese letztlich auf einen einzelnen Browser zu übertragen. Zum einen ist dies *State Isolation* (Zustandsisolierung), da zwei unterschiedliche Browser in der Regel nicht auf die hinterlegten und gegebenenfalls sensiblen Daten des jeweils anderen zugreifen können. Das zweite Merkmal ist *Entry-Point Restriction* (Beschränkung der Eintrittspunkte). Diese Eigenschaft wird zum Beispiel dann erzielt, wenn ein Benutzer für sicherheitskritische Webseiten (Onlinebanking, Auktionen etc.) stets einen eigenen Browser verwendet und die Seiten grundsätzlich nur auf bestimmten Wegen, wie etwa hinterlegten Lesezeichen, aufruft. *Entry-Point Restriction* und *State Isolation* werden im Nachfolgenden noch genauer erläutert.

Im folgenden Kapitel werden zunächst die Grundlagen betrachtet. Es wird dargestellt, worin die Vorteile bei der Benutzung mehrerer Browser liegen und welche Möglichkeiten es gibt, diese in einem einzelnen Browser zu vereinen. In Kapitel 3 wird die konkrete Umsetzung dieser Methoden im Chromium-Browser durch E. Chen et al. vorgestellt. Anschließend werden in Kapitel 4 die von den Entwicklern ermittelten Evaluierungsergebnisse bezüglich Sicherheit, Anpassungsaufwand und Performance der Implementierung betrachtet. Ein Fazit folgt in Kapitel 5.

2 Hintergrund

In diesem Kapitel werden einige typische Angriffe auf Internetbrowser behandelt und es wird erläutert, warum die Benutzung verschiedener Browser für unterschiedliche Webseiten diese Attacken verhindert. Zudem werden Ideen und konkrete Ansätze aus thematisch verwandten Arbeiten vorgestellt, mit denen ein äquivalentes Sicherheitsniveau in einem

einzelnen Browser erreicht werden soll.

2.1 Angriffsszenarien

Die nachfolgende Aufzählung stellt einige typische browserbasierte Angriffe kurz vor. Diese Angriffe haben nicht nur aktuelle Relevanz, sondern eignen sich zudem als gute Beispiele, warum der Einsatz von mehreren Browsern oder einem Browser mit entsprechenden Isolationsmechanismen vorteilhaft sein kann; diese Gegenmaßnahmen werden in Kapitel 2.2 betrachtet. Die Liste orientiert sich an den Beispielen aus [1].

- **Reflected (Non-Persistent) Cross-Site Scripting:** Der Angreifer bewegt den User dazu, eine manipulierte URL aufzurufen. Die URL enthält meist Code, der wiederum von der aufgerufenen Webseite zurückgegeben und im Browser verarbeitet wird (z.B. im Rahmen einer Suchanfrage). Das kann dem Angreifer ermöglichen, Schadcode im Browser des Benutzers auszuführen. Dieser Angriff gehört inzwischen zu den am weitesten verbreiteten im Internet und ist in unzähligen Variationen zu beobachten.[6]

- **Cross-Site Request Forgery:** Hierbei versucht der Angreifer mit Hilfe einer HTTP-Anfrage in einer manipulierten URL eine Anfrage in der vom Opfer verwendeten Webanwendung abzusetzen. Ist das Opfer mit entsprechenden Rechten bei der Anwendung angemeldet, könnten durch das Aufrufen der Seite über den manipulierten Link auch administrative Vorgänge vom Angreifer durchgeführt werden.

- **Cross-Origin Resource Import & Content Inclusion:** Durch das Einbinden von Inhalten anderer Webseiten (wie etwa Skripte, Bilder oder Style Sheets) kann der Angreifer an sensible Informationen gelangen, wenn sein Opfer auf der Seite, deren Inhalte eingebunden werden, mit seinem Benutzerkonto angemeldet ist.[7]

- **Session Fixation:** Der Angreifer bindet eine ihm bekannte Session-ID (z.B. ein durch das Aufrufen der Zielseite erzeugtes Usertoken) in einen Link ein und bringt das Opfer dazu, diese URL aufzurufen und sich dort mit seinem Account anzumelden. Anschließend kann der Angreifer die ihm bekannte ID nutzen, um die Webseite mit den Rechten des Opfers zu verwenden.

- **Click-Jacking:** Elemente einer Webseite, wie beispielsweise Schaltflächen, werden mit einem transparenten Iframe überlagert. Klickt der Benutzer auf die manipulierte Position, klickt er in Wirklichkeit auf etwas nicht von ihm beabsichtigtes (ggf. sogar auf ein Element einer anderen Webseite), ohne es tatsächlich wahrnehmen zu können.

- **History Sniffing/History Hacks:** Unter diesen Begriff fallen verschiedene Möglichkeiten, vom Benutzer zuvor besuchte Webseiten in Erfahrung bringen zu können. So kann zum Beispiel mit Hilfe von JavaScript und CSS die Farbe von Links untersucht werden (besuchte Links sind üblicherweise anders gefärbt als noch nicht besuchte) oder anhand der Ladezeiten bestimmter Resourcen getestet werden, ob die zugehörigen Webseiten bereits im Cache des Browsers vorliegen.[8, 9]

- **Rendering Engine Hi-Jacking**: Hierbei nutzt der Angreifer Sicherheitslücken im Browser aus, um die Kontrolle über die Rendering Engine, einen essenziellen Bestandteil der Browser-Architektur zu übernehmen. Da die Rendering Engine für die Interpretation und Erzeugung der im Browser dargestellten Inhalte zuständig ist,

hätte ein Angreifer Zugriff auf sämtliche Zustandsinformationen des Benutzers.[10] Bei Browsern mit mehreren separierten Instanzen der Rendering Engine, wie bei Google Chrome oder aktuellen Versionen des Internet Explorers, wird die Gefahr eines solchen Angriffs verringert (aber nicht vollständig behoben).

Zusammenfassend sehen Chen et al. drei wesentliche Schwachpunkte, die bei den genannten Attacken ausgenutzt werden: Der Angreifer kann Anfragen absetzen, durch die er (a) eine Zielseite manipulieren kann oder (b) Zugang zu Sitzungsinformationen des Opfers bekommt. Zudem könnte er (c) Exploits ausnutzen, um mittels einer kompromittierten Rendering Engine Zugriff auf weitere Daten des Opfers zu haben, die im System hinterlegt sind. Die Autoren merken jedoch an, dass dies allesamt Schwachstellen der aktuell verwendeten Browserarchitekturen sind. Da die Verwedung von mehreren Browsern, wie im nachfolgenden Kapitel erläutert, Abhilfe schaffen kann, sollte ein neuartiger Architekturentwurf ein vergleichbares Sicherheitsniveau auch in einem einzelnen Browser umsetzen können.[1]

2.2 Gegenmaßnahmen

Es stellt sich nun die Frage, wie die in 2.1 genannten Angriffe verhindert werden können. Neben dem klassischen Ansatz, mehrere Browser zu verweden, gibt es inzwischen auch einige Konzepte, die versuchen, das so erzielte Sicherheitsniveau auf eine einzelne Anwendung zu übertragen. Beide Methoden werden nachfolgend betrachtet.

2.2.1 Verwendung mehrerer Browser

Werden mindestens zwei Browser für das Aufrufen unterschiedlicher Webseiten eingesetzt, können viele der genannten Angriffe stark eingeschränkt oder gar verhindert werden, sofern einige Regeln befolgt werden. Angenommen, ein Benutzer verwendet zwei Browser A und B. Mit Browser A ruft der User ausschließlich **sicherheitskritische** Webseiten auf, und dies auch nur über die selbstständige Eingabe der (vertrauenswürdigen) URL oder per Lesezeichen. Login-Informationen für Benutzerkonten werden nur in diesem Browser eingegeben. Browser B wird zum freien Surfen durch den Rest des Internets verwendet; hier (und nur hier) werden alle **nicht vertrauenswürdigen** Seiten aufgerufen. Für sicherheitskritische Webseiten wird Browser B nicht verwendet.

Intuitiv wird deutlich, dass durch dieses Vorgehen *State Isolation* bereits realisiert wird: Auf die Cookies, Verlaufsdaten, den Cache und weitere gespeicherte Zustandsinformationen von Browser A hat Browser B keinen Zugriff; realisiert wird dies auf Betriebssystemebene, da es sich bei den Browsern um unterschiedliche Anwendungen bzw. Prozesse handelt. Aber auch *Entry-Point Restriction* ist bereits erfüllt, da der Benutzer Webseiten in Browser A, wie bereits erwähnt, nur über vordefinierte URLs (Eintrittspunkte) aufruft und dann ausschließlich von der Webseite zur Verfügung gestellte Links anklickt. Einem Angreifer ist es somit nicht möglich, den Benutzer über manipulierte URLs auf eine sicherheitskritische Seite innerhalb von Browser A zu locken und ihn so zur Eingabe von persönlichen Daten zu verleiten oder Schadcode auszuführen.

Nun kann man die Durchführbarkeit der vorgestellten Angriffe analysieren. *Cross-Site Scripting* und *Session Fixation* sind nicht mehr ohne weiteres möglich: Da der Benutzer in

Browser *A* ein bestimmtes Vorgehen beim Öffnen von Webseiten (insbesondere keine fremden und möglicherweise manipulierten Links) verwendet und seine sensiblen Daten nur in diesem Browser angibt, schlagen die Angriffe fehl. Die übrigen Attacken werden durch die implizite *State Isolation* unterbunden. Der User ist gegenüber den Webseiten in Browser *B* nicht authentifiziert (nötig für erfolgreiches *Click-Jacking* und *Cross-Origin Resource Import*); Cookies und die Liste besuchter sicherheitskritischer Webseiten sind nicht im für Browser *B* zugänglichen Speicherbereich hinterlegt (verhindert *Cross-Site Request Forgery* und *History Sniffing*). Selbst das Erlangen der Kontrolle über die Rendering Engine von Browser *B* wäre nutzlos für den Angreifer, da sich die Rendering Engine von Browser *A* in einem anderen Prozess des Betriebssystems befindet.

Obwohl die Verwendung mehrerer Browser und die dadurch erreichte Trennung vertrauenswürdiger und weniger vertrauenwürdiger Webseiten häufig empfohlen wird, scheitert diese Vorgehensweise oft an der Akzeptanz der Benutzer. Die parallele Verwendung verschiedener Browserfenster und das Einhalten der eingangs erwähnten Benutzungsregeln ist umständlicher und verringert den Komfort des Users. Etwas zugänglichere Möglichkeiten, die dieses Konzept verfolgen, sind so genannte *Seitenspezifische Browser* (SSB) wie etwa Mozillas Prism[11] oder Fluid von T. Ditchendorf[12]. Diese Programme behandeln Webseiten und -anwendungen praktisch wie Desktop-Applikationen und legen vom Benutzer frei wählbare und anschließend über Symbole aufrufbare Browserinstanzen speziell nur für die gewünschten URLs an. Im Gegensatz zu vollwertigen Browsern verfügen SSBs gewöhnlich über weniger Funktionalität, da sie zum Beispiel keine weiteren Menü- oder Symbolleisten anbieten.[5, 13]

2.2.2 Sicherheit mit einem einzelnen Browser

Auch wenn der Einsatz von SSBs das Konzept der multiplen Browser recht effizient umsetzt, bedeutet es für den Anwender immer noch einen nicht zu vernachlässigenden Verwaltungsaufwand, wenn er mit einer Vielzahl von Webanwendungen interagieren möchte. Die Idee hinter *App Isolation* (und einigen verwandten Konzepten) ist nun, beim Einsatz eines einzelnen Browsers die Verwendung mehrerer Browser für den User transparent zu simulieren, schließlich stellt die Nutzung eines einzelnen Programms offensichtlich die komfortablere Lösung dar, denn so werden der zusätzliche Verwaltungsaufwand der SSBs oder der Wechsel zwischen verschiedenen Browsern obsolet.

Es existiert bereits eine Vielzahl von Ansätzen, die diese Probleme zu beheben versuchen; einige Implementierungen finden sich bereits in aktuellen Versionen populärer Browser. So verwenden zum Beispiel sowohl Microsofts **Internet Explorer** seit Version 7[14] als auch **Google Chrome** (siehe Kapitel 3.1) Sandbox-Technologien, die das lokale Dateisystem des Users vor einem kompromittierten Browser schützen sollen. Allerdings bieten diese Lösungen keine ausreichende *State Isolation*, da durch die Kontrolle über die Rendering Engine immer noch auf die Zustandsinformationen der verwendeten Web-Anwendungen zugegriffen werden kann.[15]

Die Subsysteme des **Opus Palladianum**, wie User Interface oder Netzwerkkommunikation, laufen allesamt in einer eigenen Sandbox. Die Kommunikation zwischen den einzelnen Komponenten wird von einem eigenen Browser Kernel verwaltet.[16] Zudem kann der OP-Browser mehrere Instanzen der Rendering Engine verwalten. Das Problem ist jedoch,

dass die der Rendering Engine zugrundeliegende Architektur nicht auf Sicherheit ausgelegt ist, so dass der OP-Browser anfällig für *Cross-Site Scripting* und *Rendering Engine Hi-Jacking* ist.[17, 7] Der **Gazelle**-Browser verwendet eine vergleichbare Architektur und trennt ebenfalls verschiedene Rendering Engines logisch voneinander. Durch eine strikte Kommunikationskontrolle sind Zustandsinformationen einer Webanwendung immer nur für die jeweils zugehörige Rendering Engine sichtbar. Dies verhindert zwar die Schwachstellen des OP-Browsers, geht aber auf Kosten der Kompatibilität zahlreicher Webseiten.[18, 7]

OMash setzt auf eine sehr strikte *State Isolation*. Zustandsinformationen wie Cookies werden nur bei der Kommunikation mit Webseiten in der gleichen Browser-Sitzung und unter Beachtung der *Same-Origin Policy* verwendet, wobei jeder neue Aufruf einer Webseite eine neue Browser-Sitzung startet. Dadurch werden zwar einerseits Cross-Origin Attacken wie das genannte *Cross-Site Scripting*, *Cross-Site Request Forgery* oder *Click-Jacking* unterbunden (da auf keine bestehende Browser-Sitzung „zugegriffen" werden kann), andererseits ist es so aber auch nicht möglich, Zustände über mehrere Browser-Sitzungen hinweg zu verwalten.[19, 20] Weitere Browser mit eigenen, der Sicherheit zuträglichen, Architekturkonzepten sind unter anderem der **Tahoma**-Browser[21] und der **DarpaBrowser**.[17]

Die Entwickler von *App Isolation* verwenden als Grundlage für ihre Implementierung den OpenSource-Browser **Chromium**, da dieser bereits einige konzeptionelle Vorteile für die Umsetzung von *State Isolation* und *Entry-Point Restriction* bietet. Diese grundlegenden Vorteile und die konkrete Umsetzung werden im nächsten Kapitel thematisiert.

3 Umsetzung

Dieses Kapitel beschäftigt sich zunächst mit der Frage, welche Vorteile der Chromium-Browser bei der Umsetzung von *App Isolation* bietet. Da der Mechanismus als optionales Feature entwickelt wurde, werden in Kapitel 3.2 Wege erläutert, mit denen Betreiber von Webanwendungen *App Isolation* effizient nutzen können. Anschließend werden *Entry-Point Restriction* und *State Isolation*, ihre jeweilige Umsetzung und das Zusammenspiel der beiden Komponenten, behandelt.

3.1 Der Chromium-Browser

Chromium ist ein OpenSource-Webbrowser, auf dem auch Googles offizieller Browser **Chrome** basiert. Seine Architektur bietet eine Grundlage, auf der sich *App Isolation* verhältnismäßig einfach umsetzen lässt. Im Gegensatz zum klassischen, monolithischen Design, bei denen sämtliche Komponenten eines Webbrowsers in einen einzelnen Prozess geladen werden, trennt Chromium seine Bestandteile in verschiedene Sicherheitsbereiche auf, welche auf Betriebssystemebene in unterschiedlichen Prozessen gestartet werden.[22] Diesen Ansatz verfolgen zwar auch einige der in Kapitel 2.2.2 vorgestellten Browser, jedoch ist dies häufig mit erheblichen Einschränkungen in der Kompatibilität mit vielen Webseiten verbunden.

Chromium wurde mit der Anforderung entworfen, ein hohes Sicherheitsniveau zu erzielen und gleichzeitig eine hohe Kompatibilität zu Webseiten zu bieten. Dazu ist, ähnlich wie beim **Gazelle**-Browser, eine Trennung von Rendering Engine und Browser Kernel vorgesehen. Beide Komponenten werden in auf Betriebssystemebene getrennten Prozessen

mit unterschiedlichen Rechten gestartet und sind für unterschiedliche Aufgaben zuständig; Abbildung 1 zeigt dieses Vorgehen schematisch. Die Rendering Engine ist für die Erzeugung und Darstellung von Webseiten notwendig; sie interpretiert Inhalte von Webseiten und -anwendungen und führt sie aus. Bestandteile der Rendering Engine sind unter anderem der HTML-Parser, das Document Object Model, eine Ausführungsumgebung für JavaScript und zudem Mechanismen zur Umsetzung der Same-Origin Policy.[17]

Diese Elemente sind sicherheitskritisch, da sie direkt mit Inhalten des Webs arbeiten, die allgemein nicht vertrauenswürdig sind. Daher wird die Rendering Engine auch in eine Sandbox geladen, die nur über sehr wenig Privilegien im System des Users verfügt, wodurch die Interaktionsmöglichkeiten der Engine mit dem Betriebssystem stark eingeschränkt werden[1].[24] Selbst wenn es einem Angreifer gelingt, durch eine Sicherheitslücke die Kontrolle über die Rendering Engine zu erlangen, hat er keinen Zugriff auf das lokale Dateisystem des Benutzers; es ist ihm weder möglich, Informationen zu lesen (z.B. Daten mit Zustandsinformationen wie etwa Cookies), noch eigenen, möglicherweise bösartigen Code auf dem System zu installieren (Malware).[25] Für die Darstellung sicherheitsbezogener Informationen wie Zertifikatsfehler oder visuellen Rückmeldungen bei HTTPS-Verbindungen ist eine eigene Rendering Engine in einem separaten Prozess zuständig.

Jegliche Kommunikation der Rendering Engines mit der Außenwelt läuft über den Browser Kernel, der wiederum in einem eigenen Prozess gestartet wird. Der Browser Kernel ist mit weitaus mehr Rechten ausgestattet und verwaltet unter anderem den persistenten Speicher des Browsers (Cookies, Passwörter, Webseitenverlauf etc.), kann direkt mit dem Betriebssystem interagieren, Benutzereingaben entgegennehmen und mit dem Netzwerk kommunizieren. Die Rendering Engine wird vom Kernel als Black Box betrachtet, bei der Webinhalte wie HTML-Dateien als Eingabe dienen und vollständig erzeugte Webseiten zur Darstellung durch den Browser Kernel zurückgegeben werden; für diese Kommunikation stellt der Kernel eine eigene API zur Verfügung.[17]

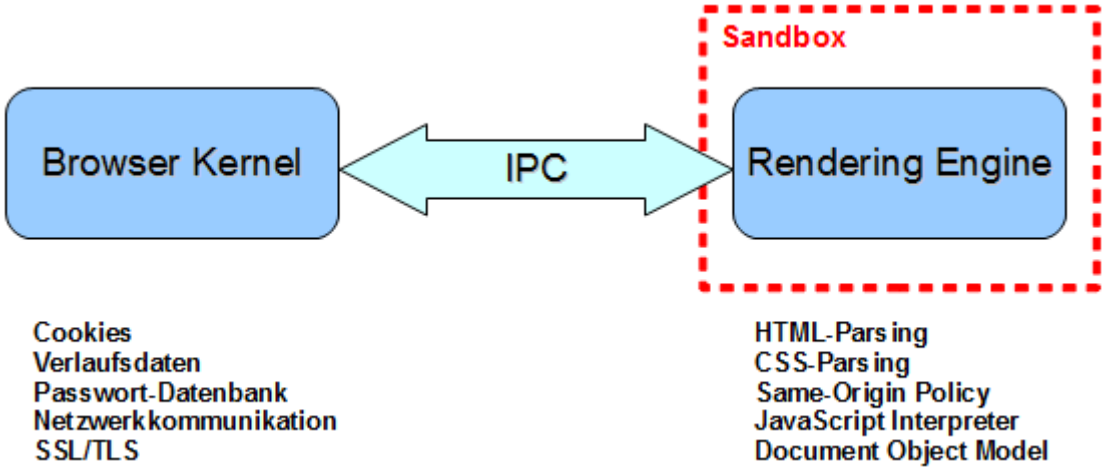

Abbildung 1: Trennung von Browser Kernel und Rendering Engine in Chromium sowie exemplarische Bestandteile der Prozesse (nach [17]).

[1] Auf Windowssystemen wird diese Privilegienrestriktion mit Hilfe der Integritätslevel (Vista, Win 7) bzw. Security Token (XP) realisiert.[23]

Plug-Ins werden von Chromium ebenfalls in separaten Prozessen, getrennt von Rendering Engine und Browser Kernel, ausgeführt. Standardmäßig werden Plug-Ins mit den vollen Nutzerrechten ausgestattet, um Kompatibilität mit den Webseiten zu gewährleisten. Um den Schaden durch die Nutzung von Exploits in Plug-Ins einzugrenzen, steht es Entwicklern jedoch frei, sie auf Wunsch ebenfalls in einer Sandbox auszuführen. Generell ist es aber Aufgabe des Entwicklers, für ausreichende Sicherheit des Plug-Ins zu sorgen.[5, 25, 26]

Üblicherweise legt Chromium pro Tab im Browser eine eigene Rendering Engine in einer separaten Sandbox an. Jedoch kann es vorkommen, dass zwei voneinander unabhängige Seiten die gleiche Rendering Engine teilen, wenn der Benutzer eine neue Seite in einem bereits verwendeten Tab öffnet - was im Widerspruch zu einer vollständige Isolation steht. Alternativ können aber auch andere Modelle zur Prozesstrennung verwendet werden. Neben dem klassischen monolithischen Ansatz, bei dem die einzige Rendering Engine zusammen mit sämtlichen Browserkomponenten in einen einzigen Prozess geladen wird, gibt es auch die Optionen, einen Prozess pro Tab-Gruppierung (*Browsing Instance*), Webseite oder Webseiten-Instanz zu erzeugen. Jede Variante bietet unterschiedliche Grade der Isolation, aber zum Teil auch individuelle Nachteile, wie beispielsweise einen deutlich größeren Speicherbedarf.[5, 27]

Die Architektur von Chromium grenzt die Auswirkungen von einigen Angriffen stark ein - insbesondere solchen, die auf das Dateisystem des Benutzers abzielen oder der Installation von Malware bzw. dem Auslesen von Dateien dienen. Durch das Sandboxing-Verfahren werden diese Arten von Angriffen selbst dann unterbunden, wenn der Angreifer die Kontrolle über die Rendering Engine erlangt hat. Jedoch schützt diese Architektur allein noch nicht vor Attacken wie *Cross-Site Scripting* oder *Cross-Site Request Forgery*, da sie keine adäquate Isolierung einzelner Zustandsinformationen innerhalb einer Browser-Sitzung bietet.[17] Die grundlegende Architektur stellt jedoch ein praktisches Grundgerüst bereit, das sich recht einfach für die Entwicklung von *App Isolation* nutzen lässt.

3.2 Identifikationsmöglichkeiten

Da *App Isolation* unter anderem die Zustandsinformationen von Webseiten isoliert, kann die Nutzung bei einigen ansonsten vertrauenswürdigen Webseiten zu unerwünschten Einschränkungen führen, insbesondere bei stark inhaltsorientierten Webseiten oder Social-Network-Features wie Facebook Connect[28]. Daher wurde *App Isolation* als optionaler Mechanismus entworfen, so dass die Wahl bei den Entwicklern von Webanwendungen liegt, ob und in welchem Umfang sie die Funktionen nutzen wollen. Prinzipiell existieren unterschiedliche Methoden, um zu isolierende Webanwendungen zu identifizieren.

Die auf den ersten Blick naheliegendste Möglichkeit, zu isolierende Webseiten zu erkennen, wäre die Identifikation über spezielle Header innerhalb einer HTTP-Antwort beim Aufruf der Seite. Ist ein speziell für diesen Zweck zu vereinbarendes Flag gesetzt, wird *App Isolation* automatisch vom Browser angewandt. Dieser Ansatz verfügt jedoch über keinerlei Verifikation: Betreiber von Unterseiten könnten die Konfiguration, die dann für die gesamte Domain verwendet wird, leicht manipulieren. Daher werden für *App Isolation* vorzugsweise Meta-Dateien verwendet.

Der 2010 von E. Hammer-Lahav vorgestellte *Web Host Metadata*-Mechanismus sieht eine

XML-Datei vor, die vom Webseitenbetreiber bzw. Besitzer der Domain in einem bestimmten Verzeichnis auf dem Server, auf das ausschließlich befugte Personen Zugriff haben, abgelegt wird.[29] Diese XML-Datei enthält die für die Isolierung benötigten Konfigurationen und wird beim initialen Aufruf der Webseite über eine gesicherte Verbindung (z.B. per HTTPS) übertragen. Wenn die Host-Meta-Daten Dritten nicht zugänglich sind, ist eine Manipulation wie bei den HTTP-Headern nicht mehr möglich, jedoch resultiert dieser Sicherheitsgewinn in einer etwas schlechteren Performance, da vor dem Öffnen der Webseite die Konfigurationsdaten separat angefordert werden müssen. Außerdem können zwar beliebige *Entry-Points* definiert werden, aber eine Isolierung ist (wie es auch bei der Nutzung der Header-Felder der Fall wäre) nur für die gesamte Domain möglich.

Eine genauere Abstufung bietet sich bei der Verwendung von anwendungsspezifischen Manifest-Dateien. Der Einsatz von Manifest-Dateien zur Anwendungsisolierung gestaltet sich im Chromium-Browser besonders einfach, da Anwendungen, die über den Chrome Web Store[30] angeboten werden, ohnehin stets mit einer Manifest-Datei verpackt sind.[31] Das Manifest einer Anwendung ist eine JSON-Datei, die Informationen über die Behandlung der Anwendung für den Browser bereithält, beispielsweise die zu verwendende Startseite, vorausgesetzte Browserversionen oder Zugriffsrechte.[32] So räumt der Browser einer Anwendung grundsätzlich nur die Rechte ein, die explizit im Manifest verankert sind[2].[34] Die Syntax der Datei erlaubt es sogar, nur bestimmte Teile oder Unterseiten einer Webseite zu isolieren; etwas, das bei der Nutzung von Meta-Daten nicht möglich ist.

Zudem bietet der Chrome Web Store Verifikationsmechanismen an, welche die Identität des Anwendungsentwicklers und die Integrität der Manifest-Datei bestätigen können.[1] Für **Anwendungs**entwickler bietet sich das Opt-In für die *App Isolation* über die Erweiterung der Manifest-Datei an, da diese einerseits ohnehin erstellt werden muss und andererseits über detailliertere Konfigurationsmöglichkeiten verfügt.[4] Der Weg über die Host-Metainformationen bietet aber eine nicht minder sichere Alternative für **Webseiten**entwickler.

3.3 Entry-Point Restriction

Eine der beiden Eigenschaften, die für die Umsetzung von *App Isolation* benötigt werden, ist *Entry-Point Restriction*. Dabei geht es darum, die Eintrittspunkte einer Seite oder Anwendung einzugrenzen und zu verhindern, dass ein Angreifer sein Opfer mit einer manipulierten URL auf eine bekannte, sicherheitskritische Seite lockt. Bei der Nutzung mehrerer Browser ist diese Eigenschaft dadurch gegeben, dass der Benutzer vertrauenswürdige Seiten zum Beispiel grundsätzlich über die Lesezeichen des Browsers aufruft. Mit *App Isolation* soll dem Nutzer diese Verantwortung abgenommen werden.

Entscheidet sich ein Entwickler zur Nutzung von *App Isolation*, kann er in der Konfiguration mittels Positivliste (*Whitelisting*) feste Eintrittspunkte, wie etwa seine Startseite (`index.html`) wählen. Resourcen dieser Webanwendung werden durch die *Entry-Point Restriction* fortan nur noch dann geladen, wenn es sich bei der aufgerufenen URL um einen definierten Eintrittspunkt handelt oder wenn sie von der Anwendung selbst angefragt werden (zum Beispiel durch das Einbinden von Bildern oder das Verlinken auf Unterseiten).

[2] Dieses Vorgehen ist vergleichbar mit der Rechtefreigabe von Smartphone-Apps, die über Googles Play Store[33] (ehemals Android Market) installiert werden.

Wenn die oben erwähnte Startseite beispielsweise der einzige Eintrittspunkt einer Webseite `www.trusted.com` ist, könnte diese problemlos auf eine Unterseite `login.html` verlinken. Eine fremde Homepage könnte zwar auf die URL `www.trusted.com/index.html` verlinken, jedoch nicht direkt auf `www.trusted.com/login.html`, da diese URL keinen gültigen Eintrittspunkt darstellt. Dieses Beispiel ist auch in Abbildung 2 zu sehen.

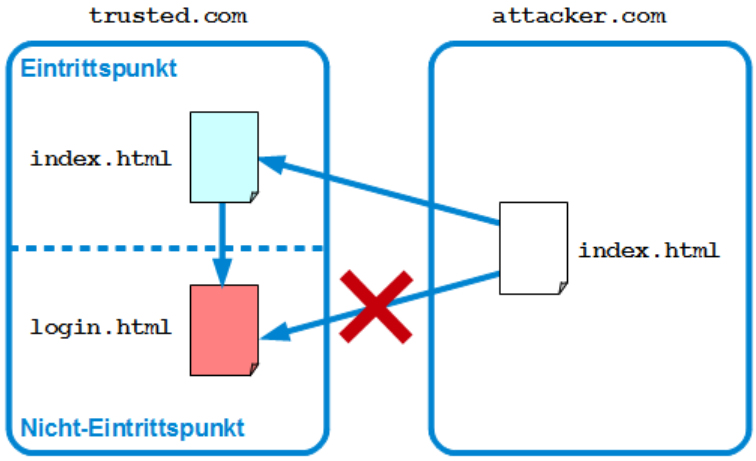

Abbildung 2: Entry-Point Restriction (nach [1])

Um Anwendungsentwicklern die Festlegung von Eintrittspunkten zu vereinfachen, lassen sich nicht nur statische URLs definieren, sondern auch URL-Muster, die beispielsweise mit dem Wildcard-Symbol (*) zusammengesetzt sind. Die eigentliche Schwierigkeit für Entwickler besteht jedoch darin, adäquate Eintrittspunkte zu finden, die einerseits möglichst wenig Angriffsfläche für Attacken bieten, andererseits aber auch den ehrlichen Benutzer nicht einschränken. E. Chen et al. empfehlen in [1], die Menge an Eintrittspunkten einer Seite so klein wie möglich zu halten und Wildcard-Symbole nur in Ausnahmen einzusetzen. Zudem sollten Seiten, die als Eintrittspunkte verwendet werden, keine kritischen Informationen verwenden oder Zustandsinformationen verändern können.

Die *Proof-of-Concept*-Implementierung im Chromium-Browser ist lediglich eine minimale Erweiterung einer Funktion der WebKit Rendering Engine[3], die mit Hilfe der erhaltenen Konfigurationsinformationen (vgl. Kapitel 3.2) prüft, ob es sich um einen gültigen Eintrittspunkt handelt und gegebenenfalls die HTTP-Anfrage entsprechend unterbindet. Die Konfiguration kann auch im Browser-Cache hinterlegt werden, um bei späteren Seitenaufrufen die Performance zu erhöhen. Die genaue Art der Speicherung hängt dabei vom verwendeten Übermittlungsweg der Informationen ab. Host-Metadaten können analog zu Cookies oder dem Browserverlauf in einem persistenten Speicher gesichert werden, der automatisch oder manuell durch den User gelöscht werden bzw. auch temporär deaktiviert werden kann (durch das Surfen im „Privaten Modus“). Manifest-Dateien werden zusammen mit der Anwendung im Anwendungsspeicher abgelegt und sind an diese gebunden.

Entry-Point Restriction sorgt dafür, dass Webanwendungen und Resourcen daraus nicht mehr über unerwünschte, da möglicherweise durch Angreifer manipulierte URLs aufgeru-

[3]WebKit ist eine von Apple entwickelte, freie HTML-Rendering-Engine, die u.a. von Chromium und Google Chrome verwendet wird.[35]

fen oder von fremden Seiten eingebunden werden können. Dadurch werden insbesondere Angriffe wie *Reflected Cross-Site Scripting*, *Session Fixation*, *Cross-Origin Resource Import* und *Cross-Site Request Forgery* unterbunden. Allerdings werden dadurch unter Umständen auch ansonsten erlaubte Interaktionen vertrauenswürdiger Webseiten untereinander eingeschränkt, was jedoch im Zusammenspiel mit *State Isolation* umgangen werden kann.

3.4 State Isolation

Die Zustandsisolierung hat den Zweck, Angriffe zu verhindern, bei denen versucht wird, die Zustandsinformationen eines Benutzers zu erlangen. Dies können zum Beispiel Cookies von Webseiten, bei denen dieser aktuell angemeldet ist, oder auch Login-Daten sein. Die Umsetzung von *State Isolation* fällt im Chromium-Browser verhältnismäßig leicht, da dieser bereits eine Trennung von Browser Kernel, Rendering Engine und installierten Webanwendungen auf Betriebssystemebene umsetzt (vgl. Kapitel 3.1). Es muss jedoch darauf geachtet werden, das eigentliche Seitenverhalten durch die Isolierung möglichst nicht zu verändern.

Die Implementierung von *App Isolation* stellt zunächst sicher, dass alle in der Konfiguration einer Webanwendung spezifizierten URLs grundsätzlich in denselben Prozess geladen und andere außen vor gelassen werden (dadurch wird ein so genannter Anwendungs-Container innerhalb des Skript-Kontextes erzeugt). Nicht vertrauenswürdige oder zu anderen Apps gehörende URLs werden stets in andere Rendering-Prozesse geladen. Durch die Sandboxing-Technologie des Chromium-Browsers wird so sichergestellt, dass nicht zur isolierten Anwendung gehörende Prozesse auch nicht auf die schützenswerten Zustandsinformationen der App (Sitzungs-Cookies, Authentisierungsinformationen etc.) zugreifen können. Der Browser Kernel erlaubt nur dem Prozess der isolierten Anwendung Zugang zu diesen Daten.[2]

Allgemein verfügt Chromium bereits über die Technologie, diese Anforderungen umzusetzen; mit *App Isolation* werden aber zusätzlich einige Eventualitäten ausgeschlossen. Um Inkompatibilitäten zu vermeiden, lädt Chromium bisweilen nicht direkt zusammengehörende Seiten zusammen in einen Prozess; dies kann alternativ auch dann passieren, wenn bereits die Maximalzahl an erlaubten Prozessen vom Browser genutzt wird. Die implementierte Ergänzung der *State Isolation* verhindert nicht nur dieses unerwünschte Verhalten, sondern sorgt zusätzlich dafür, dass der isolierende Prozess stets verlassen wird, wenn der Benutzer zu einer URL navigiert, die nicht mehr Bestandteil der zu schützenden App ist, wodurch ungewollte Informationsflüsse vermieden werden.

Weitere schützenswerte Informationen können sich auch im persistenten Speicher des Browsers befinden; hier werden in der Regel Cookies oder Cache- und Verlaufsdaten hinterlegt. Auch hier ist der Browser Kernel für die Partitionierung und Verwaltung des Speichers zuständig; jeder Prozess kann nur „seinen“ Speicherbereich sehen und verwenden.[36] Die Implementierung von *App Isolation* realisiert dies durch das Anlegen von neuen URL-Kontexten in den Benutzerprofilinformationen für jede zu isolierende Anwendung. Die mit dieser App verbundenen Daten werden dadurch in einem eigenen Verzeichnis gespeichert, auf das nur der zugehörige Rendering-Prozess Zugriff bekommt.

Aufgrund der beschriebenen strikten Isolierung können nicht-vertrauenswürdige Anwen-

dungen und Webseiten durch das Ausnutzen von Exploits keine sicherheitskritischen Daten mehr auslesen. Durch das Abgrenzen des Sitzungs- und persistenten Speichers einer Anwendung sind ihre Informationen insbesondere vor Angriffen wie *Click-Jacking*, *Cross-Origin Resource Import*, *Cross-Site Request Forgery*, *History Sniffing* und *Rendering Engine Hi-Jacking* sicher. Im Zusammenspiel mit der *Entry-Point Restriction* sorgt *App Isolation* somit für Schutz vor sämtlichen in Kapitel 2.1 vorgestellten Angriffen.

Auch die am Ende von Kapitel 3.3 genannte Einschränkung legitimer Interaktionen durch die Verwendung von *Entry-Point Restriction* ist in Verbindung mit der *State Isolation* obsolet: Webseiten können problemlos Resourcen einer isolierten Webseite einbinden, auch wenn diese zuvor keinen gültigen Eintrittspunkt dargestellt haben; die sicherheitskritischen Daten sind bereits durch die Abgrenzung der Zustandsinformationen vor unerwünschten Zugriffen geschützt.

4 Evaluation

Die durch das Zusammenspiel von *State Isolation* und *Entry-Point Restriction* erreichte *App Isolation* wurde im Anschluss an ihre Implementierung in [1] ausführlich getestet. Die Autoren konzentrierten sich dabei auf die Evaluierung dreier wesentlicher Leistungsmerkmale. Anhand der Erfüllung der gesetzten Sicherheitsziele unter der Annahme bestimmter Angreifermodelle wurde die erreichte Sicherheit der Anwendungsisolierung getestet. Danach wurde der durchschnittliche Anpassungsaufwand ermittelt, den Anwendungsentwickler zu erwarten haben, wenn sie den Mechanismus verwenden möchten. Zuletzt wurde die Performance des Browsers bei der Verwendung von *App Isolation* getestet.

4.1 Sicherheit

Um die Sicherheit bewerten zu können, müssen zunächst die zu erreichenden Ziele sowie die Fähigkeiten eines Angreifers definiert werden. Zwar sind die Isolierung einer Webanwendung und das Verhindern der in Kapitel 2.1 vorgestellten Angriffe spezielle Zielsetzungen, jedoch sollte man verstehen, was das allgemein betrachtet eigentlich bedeutet. E. Chen et al. spezifizieren die Sicherheitsziele, die sie mit *App Isolation* erreichen wollten, wie folgt:

Quellen, die nicht Bestandteil der isolierten Anwendung sind, dürfen Zustandsinformationen dieser Anwendung weder lesen noch verändern. Zudem dürfen sie keinen Zugriff auf Resourcen der isolierten Anwendung bekommen, welche keinen gültigen Eintrittspunkt zu dieser Anwendung darstellen.[1]

Als Angreifermodell wird ein Webangreifer mit etwas erweiterten Fähigkeiten verwendet. Konkret bedeutet dies:

Der Angreifer ist im Besitz mindestens eines Webservers und DNS-Namens. Er kann sich gültige SSL/TLS-Zertifikate für die von ihm betriebenen Webseiten und -anwendungen ausstellen lassen und verwenden. Besucht das Opfer eine Webseite des Angreifers, hat dieser (im Rahmen der Sicherheitsrichtlinien des Browsers) Zugriff auf die API des Browsers. Zusätzlich kann der Angreifer diejenige Rendering Engine, in der seine Skripte ausgeführt werden, kompromittieren.[1, 36]

Zur Erinnerung: Der Angreifer kann nach einer erfolgreichen Attacke lediglich auf die Inhalte der kompromittierten Rendering Engine zugreifen, nicht aber auf Kontexte und deren Zustandsinformationen, die in andere Rendering Engines geladen werden - oder gar auf den Browser Kernel, der für die sichere Verwaltung der Inhalte zuständig ist.

Die Entwickler analysierten die Sicherheit von *App Isolation* mit Hilfe von *Alloy*[37]. Dabei handelt es sich um eine Spezifizierungssprache auf Basis relationaler Logik, die es in diesem Fall erlaubt, die zu testende Implementierung anhand der modellierten Sicherheitsziele und Angreiferfähigkeiten automatisiert zu verifizieren (und möglicherweise auftretende Fehlfunktionen gezielt zu erkennen).[36] Dieses Vorgehen deckte bereits eine Verletzung der Sicherheitsziele bezüglich des Ladens von Resourcen aus Nicht-Eintrittspunkten und somit einen Verstoß gegen die Regeln der *Entry-Point Restriction* auf.

Bei der so entdeckten Sicherheitslücke in der Implementierung spielten HTTP-Redirects eine besondere Rolle. In einem exemplarischen Szenario, das auch in Abbildung 3 zu sehen ist, existieren eine vertrauenswürdige Seite `trusted.com`, die *Entry-Point Restriction* verwendet, und eine vom Angreifer kontrollierte und ohne diese Schutzmaßnahme ausgestattete Seite `attacker.com`. Die Seite `trusted.com` kann folglich eine HTTP-Anfrage (Schritt 1) nach einer beliebigen Resource von `attacker.com` absetzen. Verbirgt sich hinter der angefragten Resource jedoch eine Weiterleitung (2) zu einer Resource auf `trusted.com`, welche **nicht** als gültiger Eintrittspunkt definiert ist, wird diese dennoch geladen, da die ursprüngliche Anfrage aus dem selben, vertrauenswürdigen Kontext stammt. Dieses Verhalten ist unerwünscht, da die Seite `attacker.com` durch den Redirect eine Resource anfragt, auf die sie keine Zugriff haben sollte. In der verbesserten Version der *Entry-Point Restriction* implementierten die Entwickler einen Mechanismus, der sämtliche Redirects bei einer Anfrage protokolliert und Anfragen unterdrückt, die zu einem beliebigen Zeitpunkt den vertrauenswürdigen Kontext per Redirect „verlassen“ haben.

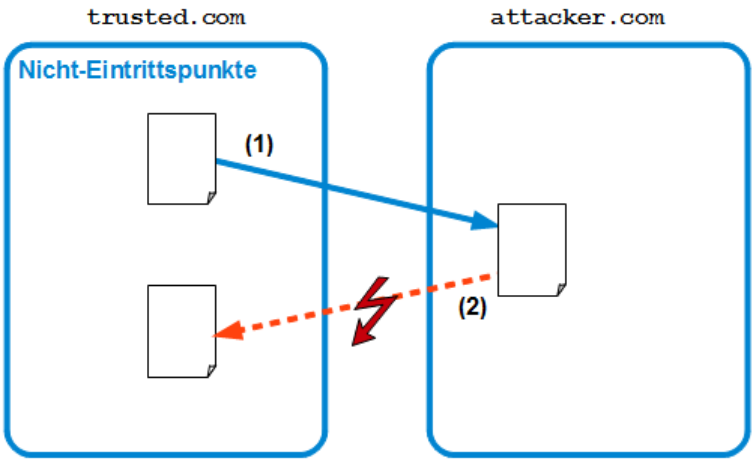

Abbildung 3: Sicherheitslücke durch fehlerhafte Behandlung von Redirects (nach [1])

Mit Hilfe von *Alloy* wurde ebenso ein weiterer Fehler in der Behandlung von Skript-Kontexten im Zusammenspiel mit der *Same-Origin Policy* entdeckt; dieses Beispiel ist in Abbildung 4 dargestellt. Angenommen, ein Benutzer ruft `trusted.com` als Anwendung auf und tauscht mit der Seite schützenswerte Daten aus (Schritt 1). Zusätzlich ruft der

Benutzer `attacker.com` auf; diese Seite lädt gemäß der Browser-Spezifikation in einer eigenen Rendering Engine, welche der Angreifer wiederum seiner Modellierung entsprechend kompromittieren kann (2). In der kompromittierten Engine legt der Angreifer einen neuen Skriptkontext von `trusted.com` an, mit dem er eine Anfrage in einem neuen Browserfenster zu einer Nicht-Eintrittspunkt-URL in der vermeintlich gesicherten Anwendung absetzen kann. Das neue Browserfenster öffnet sich in der Rendering Engine der geschützten Anwendung (3), da es zu `trusted.com` gehört - dadurch kann jedoch auch die angefragte Resource von `trusted.com` geladen werden, da die Prüfung des Eintrittspunkts nun nicht mehr fehlschlägt (4). Zur Ausbesserung wurde durch eine Erweiterung der *Same-Origin Policy* in Chromium dafür gesorgt, dass der Zugriff auf die Nicht-Eintrittspunkt-URL unterbunden wird, da die Anfrage darauf nicht aus dem ursprünglichen, in der Rendering Engine der Anwendung erstellten Skriptkontext stammt.

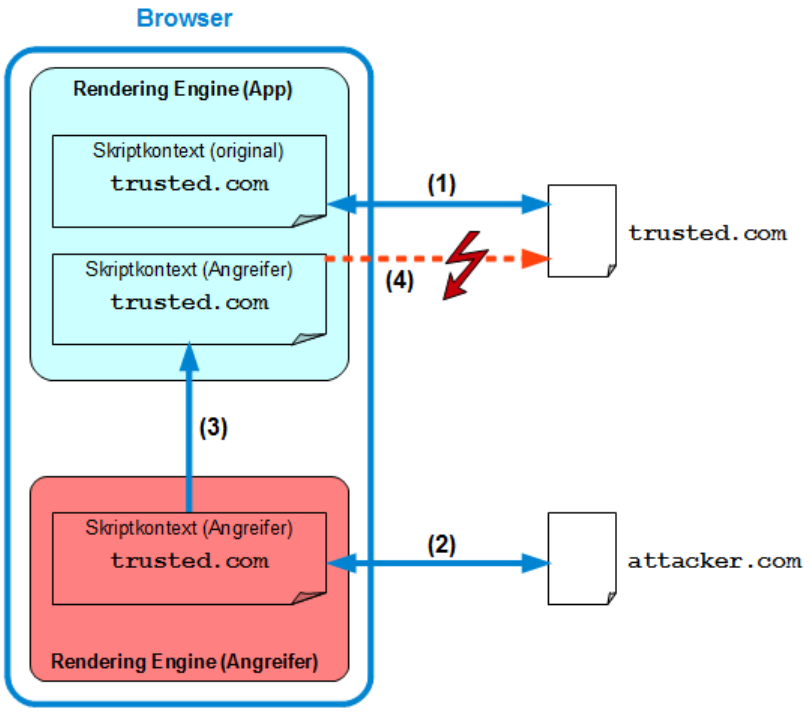

Abbildung 4: Sicherheitslücke durch fehlerhafte Behandlung von Skriptkontexten (nach [1])

Nach der Behebung der zwei genannten Probleme konnte *App Isolation* von *Alloy* in Bezug auf die gesetzten Sicherheitsziele erfolgreich verifiziert werden. Mit den oben genannten Modellen für die Verifikation durch *Alloy* lässt sich vor allem an dem zweiten Beispiel auch die Aussage bestätigen, dass *Entry-Point Restriction* und *State Isolation* nur im Zusammenspiel das geforderte Sicherheitsniveau erreichen. Verzichtet man auf die *State Isolation*, werden alle Skriptkontexte in eine Rendering Engine geladen. Kann ein Angreifer diese kompromittieren, hat er sofort Zugriff auf sämtliche gespeicherten Zustandsinformationen. Ohne *Entry-Point Restriction* läge hingegen eine Situation vor, die mit dem beschriebenen Fehlerfall vergleichbar ist - die Eintrittspunkt-Überprüfung der letzten Anfrage wird schließlich ohnehin durch den Angriff umgangen.

4.2 Anpassungsaufwand

Nicht jede Webseite eignet sich für die Verwendung von *App Isolation*. Im wesentlichen ist es die Anzahl möglicher *Entry-Points*, die ausschlaggebend für den Anpassungs- und Konfigurationsaufwand ist. Im Rahmen einer Evaluierung unter der Verwendung von Mozilla Test Pilot[4] wurden die Webseiten von neun populären (amerikanischen) Unternehmen, stellvertretend für inhaltlich verschieden orientierte Webseiten, auf die Menge möglicher Eintrittspunkte untersucht, indem *Entry-Point Restriction* in den teilnehmenden Browsern simuliert wurde. Zur ermittelten Gesamtzahl trugen dabei sowohl Links bei, über welche die Seiten aufgerufen wurden, als auch woanders eingebettete Resourcen der untersuchten Seiten.

Basierend auf den Messergebnissen lassen sich die Webseiten grundlegend in zwei verschiedene Gruppen unterteilen:

- Stark inhaltsabhängige bzw. „Social Media"-Seiten, die darauf basieren, Inhalte zu teilen, werden *App Isolation* nur schwerlich umsetzen können, da die Anzahl an *Entry-Points*, die konfiguriert werden müssten, schlicht zu groß ist - was zumeist am Teilen der Inhalte auf fremden Webseiten und der dadurch entstehenden Dynamik liegt. Im Test betraf dies insbesondere das „soziale" Internetradio Last.fm (mit mehr als 5.000 *Entry-Points*), die Webseite der New York Times (über 12.000), aber auch das Social Network Facebook (über 1 Million *Entry-Points*).

- Seiten, die spezielle (teilweise sicherheitskritische) Anwendungen und Services anbieten, aber nicht auf dem Teilen von Inhalten mit anderen Mitgliedern basieren, sind prädestiniert für die Verwendung von *App Isolation*. Sie haben meist nur eine überschaubare Menge an *Entry-Points*, was eine verhältnismäßig simple Konfiguation ermöglicht. Im Test waren dies die Online-Banking Webseiten von Wells Fargo und Capital One (56 und 33 *Entry-Points*), der Audio-on-Demand Service Grooveshark (90) sowie die Anwendung Flixster aus dem Chrome Web Store (13).

Diese Unterteilung ist natürlich keine feststehende Regel; letztendlich ist es immer vom Einzelfall abhängig, wie groß der Konfigurationsaufwand der *Entry-Point Restriction* ist. Nicht so einfach in die beiden oben genannten Gruppen einteilen lassen sich beispielsweise die Seiten der Bank of America (477 sind für diese Art von Webseite relativ viele *Entry-Points*) oder Googles E-Mail-Dienst Gmail (mit knapp 240.000 *Entry-Points*). Diese überraschend hohen Zahlen können verschiedene Gründe haben, zum Beispiel eine große Anzahl an Login-Seiten und Sub-Domänen oder auch Unterseiten zur Lastverteilung. Der Anpassungsaufwand für eine Webseite lässt sich also nicht pauschal vorhersagen, sondern ist letztlich immer abhängig von der zugrundeligenden Architektur der Webseite.[1]

4.3 Performance

Bei der Bewertung der Performance des Browsers unter Benutzung von *App Isolation* sind insbesondere zwei Werte interessant: Die Ladezeiten von Webseiten und -anwendungen (da stets die Gültigkeit einer URL bezogen auf die *Entry-Point Restriction* geprüft werden muss) und der benötigte Speicherplatz (zur Isolierung der einzelnen Anwendungen).

[4] Test Pilot ist eine Firefox-Erweiterung, die es ermöglicht, verschiedene Experimente und Untersuchungen mit Anwendern des Browsers durchzuführen. Die Ergebnisse dieser meist statistischen Untersuchungen dienen beispielsweise sowohl der Forschung als auch der Entwicklung neuer Software.[38]

Die Ladezeiten von Webseiten mit *App Isolation* hängt in erster Linie von der Anzahl der konfigurierten *Entry-Points* ab, da bei jeder Anfrage nach einer Resource einer Seite geprüft werden muss, ob es sich dabei um einen erlaubten Eintrittspunkt handelt oder nicht. Diese Prüfung wurde mit Hilfe einer Hashwert-Tabelle realisiert, wodurch der gesamte Vorgang beschleunigt wird. Durch das Laden von Webseiten jeweils mit und ohne aktivierter *Entry-Point Restriction* konnte in [1] ermittelt werden, dass sich die Ladezeiten bei 10.000 definierten - also im Kontext des Anpassungsaufwands sehr vielen - Eintrittspunkten im Schnitt um weniger als 0,1 ms erhöhen; die zusätzlich erzeugte Latenz ist also vernachlässigbar gering. Zusätzlich beachtet werden muss jedoch noch das Laden der Konfigurationsdateien. Liegen diese nicht im Cache vor oder wird die entsprechende Anwendung erstmalig eingerichtet, müssen diese separat angefragt bzw. installiert werden. Da dieser Vorgang jedoch nur einmalig einen zusätzlichen Paketumlauf (*Round Trip*) erfordert, kann der erhaltene Overhead ebenfalls vernachlässigt werden.

Im Gegensatz zur *Entry-Point Restriction* lassen sich die Auswirkungen der *State Isolation* auf die Performance am besten am Speicherplatzverbrauch ermitteln. Getestet wurde dabei der mittlere Speicherbedarf von insgesamt zwölf unterschiedlichen, parallel aufgerufenen Webseiten, bei denen sich die testenden Personen (sofern die Möglichkeit vorhanden war) auch mit einem Benutzerkonto angemeldet haben. Das Aufrufen ohne *State Isolation* in einem einzelnen Browser benötigt dabei knapp 20 MB. Bei aktivierter *State Isolation* wird rund viermal soviel Speicherplatz verbraucht; die Verwendung separater Browserinstanzen für jede Webseite benötigt sogar etwa sechsmal soviel. Der erhöhte Speicherbedarf resultiert hier aus den multiplen Benutzerprofilen, die Chromium bei der Isolierung der Webseiten anlegt (vgl. Kapitel 3.1). Durch ein optionales Caching lässt sich der benötigte Speicherplatz bei aktivierter *App Isolation* jedoch auch auf knapp 10 MB reduzieren.

Bei Betrachtung des Platzbedarfs im Arbeitsspeicher werden die Vorteile von *App Isolation* gegenüber der Verwendung mehrerer Browser besonders deutlich. Während es mit jeweils knapp 730 MB benötigtem RAM kaum einen Unterschied macht, ob *App Isolation* in einer einzelnen Browserinstanz aktiviert ist oder nicht, verbraucht Chromium zum Öffnen der zwölf Webseiten in separaten Browserfenstern mehr als 1,8 GB RAM und bietet somit bei einem vergleichbaren Sicherheitsniveau eine deutlich schlechtere Performance.[1] Die genauen Messergebnisse sind in der nachfolgenden Tabelle zu finden:

	Speicherbedarf	RAM-Bedarf
Ohne *App Isolation*	19 MB	729 MB
Mit *App Isolation*	86 MB	730 MB
Mehrere Browser	117 MB	1,83 GB

Tabelle 1: Speicher- und RAM-Bedarf von Chromium[1]

5 Fazit

Mit *App Isolation* wurde ein Mechanismus entwickelt, der es ermöglicht, mehrere Browserinstanzen in einem einzelnen Browser zu simulieren und gleichzeitig dasselbe Sicherheitsniveau wie bei der Benutzung mehrerer Browser zu erreichen. Ein Großteil heutzutage relevanter internetbasierter Angriffe kann durch die implementierte Zustandsisolierung und

Zugriffsbeschränkungen verhindert werden. Der Benutzer erfährt dabei weder signifikante Verschlechterungen bezogen auf die Performance des Browsers, noch erhebliche Einschränkungen beim Anwendungskomfort.

Jedoch muss auch erwähnt werden, dass sich *App Isolation* durch die obligatorische Konfiguration der Eintrittspunkte nicht für jede Webseite und -anwendung eignet. Der Einrichtungsaufwand für Anwendungsentwickler richtet sich hauptsächlich nach dem Umfang an Interaktionsmöglichkeiten der Applikation mit fremden Seiten und der zugrundeliegenden Architektur. Wenn in solchen Fällen eine Beschränkung der Features nicht in Frage kommt, resultiert das Einrichten der *Entry-Point Restriction* gelegentlich in einem nicht mehr vertretbaren Aufwand. Bei adäquater Umsetzung durch die Entwickler kann *App Isolation* aber dazu beitragen, eine sichere Umgebung für Seiten und Anwendungen zur Verfügung zu stellen, ohne diese Verantwortung dem Benutzer zu überlassen.

Literatur

[1] E. Chen, J. Bau, C. Reis, A. Barth, C. Jackson: „App Isolation: Get the Security of Multiple Browsers with Just One“, *CCS 11, Chicago, Illinois, USA*, Oktober 2011.

[2] C. Reis, S. Gribble, H. Levy: „Architectural Principles for Safe Web Programs“, *Sixth Workshop on Hot Topics in Networks 2007, Atlanta, Georgia, USA*, November 2007.

[3] E. Iverson: „Two Web Browsers can be More Secure than One“, Blue Ridge Networks, Oktober 2009.

[4] C. Jackson, A. Barth: „Beware of Finer-Grained Origins“, *Web 2.0 Security and Privacy 2008, Oakland, Kalifornien, USA*, Mai 2008.

[5] C. Reis, S. Gribble: „Isolating Web Programs in Modern Browser Architectures“, *EuroSys 09, Nürnberg, Deutschland*, April 2009.

[6] D. Bates, A. Barth, C. Jackson: „Regular Expressions Considered Harmful in Client-Side XSS Filters“, *WWW 2010, Raleigh, North Carolina, USA*, April 2010.

[7] L. Huang, Z. Weinberg, C. Evans, C. Jackson: „Protecting Browsers from Cross-Origin CSS Attacks“, *CCS 10, Chicago, Illinois, USA*, Oktober 2010.

[8] D. Jang, R. Jhala, S. Lerner, H. Shacham: „An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications“, *CCS 10, Chicago, Illinois, USA*, Oktober 2010.

[9] C. Jackson, A. Bortz, D. Boneh, J. Mitchell: „Protecting Browser State from Web Privacy Attacks“, *WWW 2006, Edinburgh, Schottland, Großbritannien*, Mai 2006.

[10] T. Garsiel, P. Irish: „How browsers work: Behind the scenes of modern web browsers“, online: `http://taligarsiel.com/Projects/howbrowserswork1.htm`, August 2011.

[11] Mozilla Prism (`http://prism.mozilla.com/`).

[12] T. Ditchendorf, Fluid (`http://fluidapp.com/`).

[13] Wikipedia: „Site-specific browser“, online: http://en.wikipedia.org/wiki/Site-specific_browser (aufgerufen am 29. Mai 2012).

[14] M. Silbey, P. Brundrett: „Understanding and Working in Protected Mode Internet Explorer“, *Microsoft Corporation*, online: http://msdn.microsoft.com/en-us/library/bb250462%28v=vs.85%29.aspx), Januar 2006.

[15] Accuvant Labs: „Browser Security Comparison - A Quantitative Approach“, 2011.

[16] C. Grier, S. Tang, S. King: „Secure web browsing with the OP web browser“, *Proceedings of the 2008 IEEE Symposium on Security and Privacy, Oakland, Kalifornien, USA*, Mai 2008.

[17] A. Barth, C. Jackson, C. Reis: „The Security Architecture of the Chromium Browser“, *Stanford Technical Report*, September 2008.

[18] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, H. Venter: „The Multi-Principal OS Construction of the Gazelle Web Browser“, *Proceedings of the 18th conference on USENIX security symposium, Montreal, Kanada*, August 2009.

[19] R. Cook: „The Next Big Browser Exploit“, *CSO Magazine*, Februar 2008.

[20] S. Crites, F. Hsu, H. Chen: „OMash: Enabling Secure Web Mashups via Object Abstractions“, *CCS 08, Alexandria, Virginia, USA*, Oktober 2008.

[21] R. Cox, J. Gorm Hansen, S. Gribble, H. Levy: „A Safety-Oriented Platform for Web Applications“, *Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland, Kalifornien, USA*, Mai 2006.

[22] The Chromium Projects: „Multi-process Architecture “, online: http://dev.chromium.org/developers/design-documents/multi-process-architecture (aufgerufen am 29. Mai 2012).

[23] W. Venema: „Isolation Mechanisms for Commodity Applications and Platforms “, *IBM Research Report*, Januar 2009.

[24] The Chromium Projects: „Sandbox “, online: http://dev.chromium.org/developers/design-documents/sandbox (aufgerufen am 30. Mai 2012).

[25] C. Reis, A. Barth, C. Pizano: „Browser Security: Lessons from Google Chrome“, *ACM Web Security*, 2009.

[26] The Chromium Projects: „Plugin Architecture “, online: http://dev.chromium.org/developers/design-documents/plugin-architecture (aufgerufen am 30. Mai 2012).

[27] The Chromium Projects: „Process Models “, online: http://dev.chromium.org/developers/design-documents/process-models (aufgerufen am 29. Mai 2012).

[28] D. Morin: „Announcing Facebook Connect“, online: http://developers.facebook.com/blog/post/108/, Mai 2008.

[29] E. Hammer-Lahav: „Web Host Metadata“, *Internet Engineering Task Force RFC 6415*, online: http://tools.ietf.org/html/rfc6415, Oktober 2011.

[30] Google Chrome Web Store (`https://chrome.google.com/webstore/`).

[31] Google Code Labs: „Packaged Apps“, online: `http://code.google.com/chrome/extensions/apps.html` (aufgerufen am 29. Mai 2012).

[32] Google Code Labs: „Formats: Manifest Files “, online: `http://code.google.com/chrome/extensions/manifest.html` (aufgerufen am 29. Mai 2012).

[33] Google Play (`https://play.google.com/store`).

[34] A. Barth, A. Porter Felt, P. Saxena, A. Boodman: „Protecting Browsers from Extension Vulnerabilities“, *Technical Report, University of California at Berkeley*, Dezember 2009.

[35] Wikipedia: „WebKit“, online: `http://en.wikipedia.org/wiki/WebKit` (aufgerufen am 29. Mai 2012).

[36] D. Akhawe, A. Barth, P. Lamy, J. Mitchelly and D. Song: „Towards a Formal Foundation of Web Security“, *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium, Edinburgh, Schottland, Großbritannien*, Juli 2010.

[37] D. Jackson, Alloy (`http://alloy.mit.edu/alloy/`).

[38] Mozilla Test Pilot (`https://testpilot.mozillalabs.com/`).