

Das Konzept des Tainting in der Browser-Security

Rene Schneider
Hochschule der Medien
Nobelstr. 10, 70569 Stuttgart
e-mail: rs034@hdm-stuttgart.de

Abstract

Die Sicherheit des Web-Browsers erhält durch die wachsende Bedeutung von Webanwendungen immer mehr Gewicht. Insbesondere die Scriptsprache JavaScript als das wichtigste Mittel zur Erstellung moderner Webanwendungen ist immer häufiger Angriffen ausgesetzt bzw. wird zur Umsetzung von Angriffen genutzt. Dieses Paper fasst einige Ideen und Ansätze zusammen, wie das Security-Konzept des Data Tainting im Bereich der Browser- bzw. JavaScript-Sicherheit nutzbringend angewendet werden könnte.

1 Einleitung

Die Sicherheit des Web-Browsers, den wir schon seit einigen Jahren nicht mehr nur zum Betrachten statischer Seiten, sondern als Fenster in die Welt der web-basierten Anwendungen nutzen, hat heutzutage eine höhere Priorität denn je. Javascript, CSS und XML ermöglichten gemeinsam mit dem althergebrachten HTML eine noch nie dagewesene Interaktivität in der Bedienung von Webseiten, wie man sie bisher nur bei Desktop-Applikationen kannte. Dieser Prozess machte aus den zunächst einfach gestrickten Hypertext-Betrachtern, die früher als „Browser“ bezeichnet wurden, eine Art „zweites Betriebssystem“, auf dem eine eigene Gattung von Anwendungen, die sogenannten Webapplikationen, aufsetzt.

Die Browser machten daher dieselbe Entwicklung durch, wie sie die Betriebssysteme bereits lange hinter sich haben: Mit der Fähigkeit zur Ausführung von beliebigem Code und der Kommunikation mit Serverdiensten kamen unweigerlich auch viele Sicherheitsrisiken auf, die es zuvor nicht gab. Die Tatsache, dass immer mehr hochkritische, persönliche Bereiche betreffende Transaktionen komplett online getätigt werden (man denke an Online-Banking, Online-Auktionen, Online-Bestellungen oder das viel beschworene e-Government), erhöht zudem den potenziell zu erwartenden Schaden enorm. Es herrscht daher heute kein Zweifel daran, dass Browser-Sicherheitslücken zu den kritischsten Problemen sowohl auf Heimrechnern als auch auf Firmen-Workstations zählen.

Dieses Paper behandelt zunächst kurz den grundlegenden Sicherheitsmechanismus, auf dem die JavaScript-Sicherheit (und damit auch die Browser-Sicherheit im Allgemeinen) heute aufbaut. Anschließend wird das Verfahren des „Tainting“ allgemein vorgestellt sowie auf einen konkreten Anwendungsfall eingegangen, in welchem Tainting als Sicherheitskonzept erfolgreich angewendet wurde. Danach folgen schließlich Überlegungen, wie eine geschickte Anwendung des Tainting-Konzepts die heutige JavaScript-Security verbessern könnte, und an welchen Stellen auch Tainting nicht weiterhilft.

2 JavaScript-Security

Das zentrale JavaScript-Sicherheitsmodell ist die sogenannte „Same Origin Policy“. Dies bedeutet, dass eine JavaScript-Routine, die von einem Webserver A geladen wurde, nur Zugriff auf diejenigen Objekte bzw. Ressourcen hat, die von demselben Server stammen. Dabei bezieht sich der Handlungsbereich, in welchem ein Script agieren kann, auf den Server-Hostnamen, den Port und das Protokoll der HTML-Seite, in der es (direkt oder via `<script>`-Tag) eingebunden ist.

Ein Script kann - gesetzt den Fall, die Policy wird korrekt umgesetzt - folglich lediglich diejenigen geöffneten Dokumente im Browser per DOM erreichen, die vom selben Hostnamen über denselben Port sowie mittels desselben Protokolls geladen worden sind. Für die Erreichbarkeit lokal gespeicherter Daten wie Cookies gilt das Gleiche.

2.1 Die Lücken im System...

So gut die Same Origin Policy eine ganze Reihe von Angriffsszenarien vereitelt, so lässt sie dennoch eine ganze Reihe Lücken offen, die rein konzeptuell nicht mehr geschlossen werden können, ohne die Kompatibilität zu bestehenden Websites massiv zu beeinträchtigen und viele sinnvolle Anwendungen unmöglich zu machen.

**Cross-Site Requests via **: Das ``-Tag zur Einbettung von Bildern in eine HTML-Seite kann hervor-

gend zum Absetzen von Cross-Site-GET-Requests genutzt werden („Cross-Site“ im Sinne von „außerhalb des eigenen Origin“). Durch Manipulation des DOM und Einfügen eines neuen Bild-Objekts an beliebiger Stelle einer Website wird der Browser dazu veranlasst, einen GET-Request zu starten. In der URL dieses Requests können beliebige (wenn auch mengenmäßig begrenzte) Daten mitgesendet werden, ohne dass der Browser eine Chance hätte, diese versendeten Daten zu stoppen - außer er zeigt das Bild nicht an. Zwar ist es einem Script nicht möglich, die Antwort auf diese Anfrage auszulesen (sie besteht ja in einem Bild, welches dem Nutzer auf dem Monitor dargestellt wird), aber für das Versenden von Daten über Servergrenzen hinweg ist diese Methode bestens geeignet.

Cross-Site Requests via <SCRIPT>: Das *<script>*-Tag ist noch weitaus mächtiger als das **-Tag, denn es erlaubt eine bidirektionale Kommunikation mit einem Server außerhalb des Same-Origin-Bereichs. Der „Trick“ besteht darin, eine Referenz auf ein neues JavaScript-File in das DOM einer Website einzufügen. Der Browser wird daraufhin das Script vom Server (welcher eine beliebige Adresse tragen kann, denn die Anfrage erfolgt jetzt vom Browser aus, nicht vom Script!) abrufen und verarbeiten (parsen). In der URL zum Script können zunächst - ähnlich wie im Fall des **-Tag - Daten an den Server gesendet werden. Der Server kann seine Antwort nun aber in Form von maschinenlesbarem JavaScript-Code zurückliefern, welcher beispielsweise Variablen mit einem Wert (den zu übermittelnden Daten!) belegt. Dies findet browserseitig im Kontext der Website und damit unserer ursprünglichen JavaScript-Routine, welche die Referenz auf das externe „Script“ ins DOM der Seite eingebracht hat, statt, was diesem Script die Möglichkeit des Zugriffs auf die übermittelten Daten im Rahmen der Same Origin Policy gibt. Damit ist die bidirektionale Kommunikationsmöglichkeit vollständig etabliert.

DNS-Rebinding: Da sich das Verständnis des „Origin“ eines Objektes in JavaScript primär auf den Hostnamen des Servers bezieht, besteht die Möglichkeit, durch Ändern des DNS-Eintrags für einen Hostnamen die Same Origin Policy auszuhebeln und Anfragen an einen anderen Server zu erlauben, die dennoch vollständig von der Policy abgedeckt sind. Allerdings erfordert ein Angriff basierend auf diesem Grundmuster eine nicht unerhebliche Menge an Vorbereitungen sowie eine Möglichkeit zur Manipulation der DNS-Auflösung direkt beim Client, so dass dieser Angriff nicht allein mit den Mitteln des Browsers umgesetzt werden kann.

2.2 ...und warum diese nicht gestopft werden können

Sicherheitslücken, die etwas erlauben, was eigentlich unterbunden werden sollte, werden normalerweise in irgendeiner Art und Weise gestopft - sei es mittels eines schnell aufgeklebten „Software-Pflasters“ oder einer grundlegenden Veränderung im Code. Die genannten Wege, die Same Origin Policy in Bezug auf die Kommunikation mit anderen Websites auszuhebeln, können jedoch aus teils konzeptionellen, teils praktischen Gründen nicht gestopft werden.

Ein vollständiges Verbot des Einbaus fremder Objekte (d.h. Objekte, die von einem anderen Server bezogen werden) beispielsweise, welches gleichzeitig beide Cross-Site-Requestmöglichkeiten unterbinden würde, wäre vollkommen unpraktikabel, weil dadurch das Konzept der im Web grundlegenden Sprache HTML radikal verändert würde. HTML sieht es explizit vor, dass Verweise auf Objekte, die auf einem anderen Server liegen, gesetzt werden können. Auf dieses „Feature“ verlassen sich Millionen von Websites; selbst solche, die keinerlei Daten von wirklich externen Servern laden, nutzen häufig zur Lastaufteilung die Möglichkeit, Bilder und anderen statischen Content von separaten Servern zu laden statt von demjenigen Host, der die eigentliche HTML-Seite dynamisch erzeugt.

Darüber hinaus basiert eine große Zahl der moderneren Webanwendungen (insbesondere diejenigen aus der Kategorie der sogenannten „Mashups“: Seiten, welche kaum eigenen Content bieten, sondern in erster Linie Daten von anderen Seiten beziehen und aggregieren bzw. deren Darstellung verändern) auf den ausgefeilten Umgehungsmöglichkeiten, welche bidirektionale Kommunikation mit anderen Servern erlauben. Auch die Funktion dieser Webanwendungen würde empfindlich gestört. Das Grundproblem ist schlicht und einfach, dass ein gewisser Grad von Cross-Site-Requests (auch durch JavaScript initiiert) erlaubt und erwünscht ist, während die Browser keinerlei Konzept zur kontrollierten Erlaubnis solcher Requests besitzen und das bestehende Verbot dann auch noch lückenhaft umsetzen.

Neben diesen grundlegenden Problemen ist die JavaScript-Security heutzutage auch permanent durch Implementierungsfehler in den Routinen, die Zugriffsrechte validieren sollen, bedroht. Da das Ausmaß dieses Prüfcodes stetig zugenommen hat, während sich am grundlegenden Sicherheitskonzept nichts änderte, konnten sich an dieser Stelle in allen Browser-Implementierungen Fehler einschleichen, welche sehr oft die Grundlage für Browser-Exploits darstellen. Hier besteht das Grundproblem im Zwang zum permanenten Ergänzen des „Sicherheitscodes“ mit weiteren Bedingungsprüfungen zum Schließen von auftretenden Sicherheitslücken.

3 Das Konzept des Tainting

Tainting bezeichnet ein Verfahren, bei dem Ressourcen beliebiger Art (etwa eine Variable, eine Funktion, ein Objekt, ...) während der Laufzeit eines Programms mit zusätzlichen Tags versehen werden, die für das eigentlich ausgeführte Programm in aller Regel nicht erreichbar sind. Im einfachsten Fall (daher hat das Konzept seinen Namen) geht es um ein 1-Bit-Flag, welches entweder auf „tainted“ oder „untainted“ steht. Im Zustand „untainted“ wird die Ressource wie gewohnt behandelt, im Zustand „tainted“ (auf Deutsch etwa „verschmutzt“) gelten hingegen gewisse Restriktionen, die je nach Anwendungsfall sehr unterschiedlich aussehen können.

Wichtig ist, dass beim Kontakt von Ressourcen miteinander (etwa durch die Ausführung einer „tainted“ Funktion mit „untainted“ Parametern) in aller Regel die „unverschmutzten“ Ressourcen automatisch „verschmutzt“ werden, sofern die Möglichkeit besteht, dass sie bei dem Kontakt verändert worden sind. Jedoch können Ressourcen häufig auch „gereinigt“ werden (d.h. sie wechseln ihren Zustand von „tainted“ zu „untainted“), etwa indem sie spezielle, vom Laufzeitsystem zur Verfügung gestellte Prüfroutinen durchlaufen, die sicherstellen, dass von der Ressource keine Gefahr ausgeht. Ob eine derartige „Reinigung“ allerdings möglich ist, hängt von der konkreten Implementierung des Konzepts ab.

Wichtig ist, zu verstehen, dass Tainting nur in solchen Szenarien sinnvoll angewendet werden kann, in welchen eine Anwendung für eine zweite Anwendung die Laufzeit-Umgebung bereitstellt. Tainting in einer Applikation, die nicht für andere Anwendungen als Ausführungsumgebung dient, macht nur selten Sinn. Allerdings ist im Falle des Browsers genau die erstere Situation gegeben: der Browser dient als Laufzeitumgebung für in JavaScript geschriebene Webanwendungen. Es lohnt sich folglich, den Einsatz des Tainting-Konzepts im JavaScript-Interpreter des Browsers zu erwägen.

3.1 Tainting an einem Beispiel

Doch bevor es an die Browser-Sicherheit geht, möchte ich ein Beispiel für den erfolgreichen und vor allem kreativen Einsatz von Tainting in einer Anwendung vorstellen. Dieses Beispiel zeigt recht gut, dass das Tainting-Konzept sehr vielseitig einsetzbar ist und als Kernidee von erstaunlich eleganten Lösungen dienen kann.

Bei der „Anwendung“ handelt es sich um das MMORPG¹ „World of Warcraft“ (abgekürzt WoW). Blizzard Entertainment, der Hersteller des Spiels, hat für das User In-

¹Massively Multiplayer Online Role Playing Game - ein Online-Spiel, bei dem mehrere tausend Spieler in einer virtuellen Welt gemeinsam spielen

terface des Spiels eine virtuelle Maschine auf der Basis der Scriptsprache Lua² in den Client integriert, so dass das komplette Interface in Form von Scriptdateien und XML-Layoutinformationen (für die Darstellung von UI-Elementen auf dem Bildschirm) vorliegt. Diese Scripts werden beim Start des Spiels geladen und in einen Bytecode übersetzt, der schließlich ausgeführt wird. Dabei kann jedoch weit mehr als nur die von Blizzard selbst zur Verfügung gestellten Scripts, die das Standard-UI bilden, geladen werden: es besteht die Möglichkeit, sogenannte Addons - also zusätzliche, von Spielern erstellte UI-Scripts - in das Spiel zu laden, um auf diese Weise das Standard-Interface zu erweitern oder abzuändern. Zur Interaktion mit dem eigentlichen Spielclient greifen diese Scripts auf ein global verfügbares API³ zurück, welches den Informationsaustausch sowie die UI-getriggerte Interaktion erlaubt. Der umgekehrte Weg - d.h. der Client veranlasst das UI zu einer Aktion - ist über ein Event-System realisiert, welches gleichzeitig die eigentliche Basis des UI-Runtime-Systems bildet: es gibt wie bei den meisten Plug-In-Architekturen keine „main“-Routine in WoW-Addons oder dem Standard-UI, stattdessen bestehen diese aus vielen Event-Handlern, die vom Runtime-System aufgerufen werden.

Das Problem bestand nun darin, dass die Spieler sich sehr mächtige Addons geschrieben haben, die viele Entscheidungen für den Spieler automatisch treffen und in die Tat umsetzen konnten - der Spieler wurde dadurch zum „Äffchen, das auf das Knöpfchen drückt“⁴, während das Addon für den Spieler die Entscheidungen traf. Das konnte beispielsweise folgendermaßen aussehen: Ein mächtiger Gegner im Spiel besitzt häufig die Fähigkeit, seine Kontrahenden zu „verfluchen“ - sie mit Schwächungszaubern zu belegen, die schädliche Auswirkungen haben, so lange diese auf dem Spieler lasten. Einige Spieler wiederum besitzen Fähigkeiten, mit denen sie solche Flüche entfernen können. Die Herausforderung soll darin bestehen, schnell zu erkennen, welcher Spieler einer größeren Gruppe einen solchen Fluch auf sich trägt und diesen zu entfernen sowie im Falle von mehreren Spielern zu entscheiden, in welcher Reihenfolge die Flüche entfernt werden sollen. Es dauerte allerdings nicht lange, bis ein findiger Spieler ein Addon geschrieben hatte, welches automatisch alle erreichbaren

²<http://www.lua.org>

³Eine interessante Parallele zu den Betriebssystemen: globale, mächtige APIs verhindern die feingranulare Zuteilung von Rechten. Es wäre daher sicher auch ein sehr interessanter Ansatz, das UI-System von WoW in ein Capability-basiertes System umzukonzipieren, ähnlich wie dies bei Betriebssystemen in der Diskussion ist.

⁴Dies ist durchaus wörtlich zu nehmen - um eine Aktion im Spielclient auszulösen, war das „echte“ Drücken einer Taste oder das Anklicken eines Buttons erforderlich, sonst führte der Client die Aktion nicht aus. Vollständig automatisch konnte daher auch mit Addons nie gespielt werden, aber so manches Addon degradierte den Spieler tatsächlich zum reinen Erzeuger des nötigen Tastatur-Inputs, während es die Entscheidung übernahm, welche Aktion daraufhin ausgeführt werden sollte.

Spieler in der Umgebung nach Flüchen absuchen konnte. Fand es einen solchen auf einem Spieler, zauberte es automatisch den passenden Reinigungszauber auf den entsprechenden Spielcharakter. Waren mehrere Spieler von einem Fluch betroffen, konnte man im Voraus eine Prioritätenliste für das Addon erstellen, nach welcher es die Entscheidung traf, in welcher Reihenfolge die betroffenen Spieler entflucht werden sollten. Das alles geschah in Sekundenbruchteilen und ermöglichte es, ohne großen Aufwand ein nahezu perfektes Spielverhalten in Bezug auf das Bannen von Flüchen zu erlangen.

Das eigentliche Problem, vor dem Blizzard hier auf „höchster Ebene“ stand, betraf aber nicht, wie man zunächst vermuten könnte, eine eventuelle Ungleichheit von Spielern mit und ohne dem „magischen“ Entfluch-Addon - dieses kam in erster Linie im Spiel gegen computergesteuerte Gegner zum Einsatz. Das Problem bestand darin, dass Blizzard nun, da dieses Addon existierte und den Spielern zur Verfügung stand, dazu gezwungen war, die Spielkomponente „Flüche“ bei Kämpfen gegen Bossgegner in exzessiver Weise einzusetzen, um die durch das Addon allesamt „perfekt“ spielenden Spieler auszugleichen. Dies drängte die Spieler noch viel stärker in die Rolle des reinen „Tastatur-Input-Erzeugers“ - sie drückten einfach nur noch regelmäßig einen Knopf, der es ihnen erlaubte, beliebige Aktionen auszuführen. Und das Ganze betraf nicht nur diese eine Spielkomponente - viele andere Aspekte des Spiels sind auf ähnliche Weise automatisiert worden, womit es dort zu denselben Balancing-Schwierigkeiten gab.

Da das regelmäßige Drücken eines einzigen Knöpfchens nicht ganz den Vorstellungen von Spielspaß bei Blizzard entsprach, entschied man dort, dass der Automatisierung Einhalt geboten werden musste. Gleichzeitig wollte man aber die vielseitige Addon-Schnittstelle erhalten, die tausenden legitimen und den Spielspaß tatsächlich steigernden Addons als Schnittstelle zum Spielclient diente. Es galt folglich, nur an den Stellen Restriktionen anzuwenden, die entweder das Auslösen einer Aktion oder das Auswählen eines Gegners/Teammitglieds (als Vorbereitungshandlung zum Ausführen einer Aktion) zur Folge haben.

Die Schwierigkeit bestand darin, dass es eine Unmenge von Programmpfaden gab, über die eine Aktion ausgelöst werden könnte. Dies musste nicht zwangsläufig durch das Aufrufen der API-Funktion, welche die Aktion direkt zur Folge hat, geschehen - auch das Aufrufen einer Funktion des Standard-UI, die (womöglich durch den Aufruf weiterer Funktionen) irgendwann bei der kritischen API-Funktion endet, würde den Zweck erfüllen und musste daher verhindert werden.

Gelöst wurde das Problem schließlich durch den Einsatz des Tainting-Konzepts. Das Tainting-Bit erhielten dabei sowohl Variablen als auch Funktionen. Dabei sind sämtliche Funktionen des Original-Interface automatisch „un-

tainted“, auch die durch sie angelegten Variablen genießen diesen Status. Demgegenüber sind Funktionen und Variablen von Addons stets „tainted“ (eine Ausnahme bilden lediglich spezielle signierte Addons von Blizzard, die logisch zum Standard-Interface zählen, technisch aber als Addon, welches erst bei Bedarf geladen wird, ausgeführt sind).

Die Besonderheit des WoW-Tainting-Konzepts besteht darin, dass auch der Ausführungspfad ein Tainting-Bit besitzt. Der Ausführungspfad wird immer dann „beschmutzt“, wenn er Kontakt mit einer Funktion oder Variable hat, deren Tainting-Bit gesetzt ist. Klappert der Funktionsaufruf-Stack später wieder nach oben, wird das Tainting-Bit des Ausführungspfads an der Stelle, an der es gesetzt wurde, wieder zurückgesetzt - das Prinzip entspricht also in etwa dem Stack-Walkback im Java-2-Security-Modell. Wird nun an einer Stelle eine kritische API-Funktion aufgerufen, so prüft diese Funktion zuallererst den Zustand des Tainting-Bit des Ausführungspfads. Ist es gesetzt, bedeutet dies, dass der Ausführungspfad - begonnen mit beispielsweise einem Tastendruck des Users - irgendwo Kontakt mit einem durch ein Addon beeinflussten Wert hatte, woraufhin die Durchführung der gewünschten Aktion verweigert wird.

Um es einem Addon trotzdem zu erlauben, in begrenztem Rahmen Aktionen auszuführen - beispielsweise, um Buttons auf dem Bildschirm zu erstellen, über die Spieler ausgewählt werden können - gibt es eine Reihe von speziellen Proxy-UI-Komponenten. Diese können von einem Addon-Entwickler über ein Set von Parametern mit Instruktionen versehen werden, welche bestimmen, was beim Anklicken der Komponente passieren soll. Die Komponente nutzt dabei privilegierte Funktionen, um eingegebene Parameter zu „säubern“, damit der Ausführungspfad nicht „getainted“ wird. Allerdings besteht durch den Umweg über die Proxy-Komponente die Möglichkeit, gezielt nur diejenigen Aktionen zu erlauben, die keine Automatisierung ermöglichen. Genutzt wird dies beim erwähnten Beispiel eines Buttons, mit welchem ein Spieler ausgewählt werden kann, beispielsweise dazu, eine einmal getätigte Verknüpfung eines Buttons mit einem Spieler beim Beginn eines Kampfes im Spiel „einzufrieren“ und eine Änderung erst wieder zu ermöglichen, wenn der Kampf beendet wurde.

4 Tainting in der Browser-Sicherheit

Im Zusammenhang mit Browsern sind eine ganze Reihe möglicher Wege denkbar, wie das Konzept des Tainting angewendet werden könnte. Einige Ideen sind im Folgenden erläutert.

4.1 Firefox und das Isolations-Debakel

Die Benutzeroberfläche des Firefox-Browsers ist nicht in C++ geschrieben, wie der Kern des Browsers, sondern

in JavaScript. Der Bereich, in welchem der UI-Code residiert, nennt sich „Chrome“. Das große Problem dabei ist, dass dieser Bereich nicht „hart“ von der JavaScript-Laufzeitumgebung, in welcher Website-Skripte laufen, abgetrennt ist - im Gegenteil, es handelt sich offenbar um dieselbe Runtime-Umgebung und denselben JavaScript-Interpreter. Auch einzelne Websites werden in keiner Weise hart voneinander isoliert, sie laufen im selben Interpreter und teilen sich einen großen Speicherbereich. Daher ist es bereits mehrfach vorgekommen, dass einfache, unprivilegierte Websites...

- ...JavaScript-Code in den Chrome-Bereich injizieren konnten, um diesen mit den erweiterten Rechten des UI auszuführen
- ...auf Daten von anderen gerade geöffneten Websites zugreifen konnten
- ...JavaScript-Code im Browser an Stellen platzieren konnten, an denen dieser das Verlassen der Website „überlebt“

Bis auf den ersten Punkt betreffen die Problemstellungen nach derzeitigen Erkenntnissen allerdings nicht nur den Firefox-Browser, sondern auch auf alle anderen verbreiteten Browser (so wird etwa in 4.1.3 auf ein Beispiel für einen IE-Exploit, der eine derartige Lücke ausnutzt, verwiesen).

Nun wäre es natürlich denkbar, das Übel an der Wurzel zu packen und eigene, vollständig voneinander isolierte Interpreter sowie Speicherbereiche für UI und Websites (im Optimalfall sogar getrennt für jede Website) einzuführen. Ist ein solcher tiefgehender Eingriff jedoch - aus welchen Gründen auch immer - nicht praktikabel, bieten sich Techniken aus dem Tainting-Bereich an, um dennoch ein höheres Sicherheits-Niveau zu erreichen.

4.1.1 Isolation von UI- und Website-Skripts

Um das User Interface eines Browsers in JavaScript erstellen zu können, ist es erforderlich gewesen, erweiterte JavaScript-Funktionen zu implementieren, die weit über den Funktionsumfang, der einem Website-Skript zur Verfügung steht, hinausgehen. Diese privilegierten Funktionen sollen selbstverständlich nicht für Skripts, die von einer Website stammen, erreichbar sein - es stellt sich also die Frage, wie eine Isolation an dieser Stelle stattfinden kann, selbst wenn die JavaScript-Interpreter-Instanz sowie der Script-Speicherbereich für beide Arten von Skripts dieselben sind.

Tainting kann diese Isolation ermöglichen. Dazu müssten alle JavaScript-Objekte und alle Funktionen, welche von einer Website stammen, bei ihrer Erzeugung mit einem Flag als „tainted“ markiert werden. Alle JavaScript-basierten Teile des UI sind hingegen „untainted“. Privilegierte Funktionen müssten nun bei jedem Aufruf prüfen, ob

der Ausführungspfad, über welchen sie aufgerufen worden sind, „tainted“ ist (d.h. mit einem Objekt in Berührung kam, dessen Taint-Bit gesetzt war). Ist dies der Fall, ist davon auszugehen, dass ein Script einer Website Einfluss auf den Ausführungspfad genommen hat, der zum Aufrufen der privilegierten Funktion geführt hat, und die Ausführung ist aus Sicherheitsgründen zu verweigern.

Durch das Tainting-Grundprinzip der Verschmutzung von sauberen Objekten beim Kontakt mit bereits verschmutzten Objekten werden - unter der Annahme, dass das Taint-System selbst fehlerfrei implementiert wurde - zuverlässig auch noch unentdeckte „Seitenkanäle“, über welche etwa Schadcode ins UI-System an sämtlichen Sicherheitschecks vorbei eingespeist werden könnte, abgesichert. Diese Eigenschaft ist einer der Kernvorteile des Tainting-Konzepts im Vergleich zu simplen Sicherheitsprüfungen.

4.1.2 Isolation von Webseiten untereinander

Durch den Umstand, dass sich die Skripts aller gleichzeitig geöffneten Websites technisch im selben Speicherraum befinden und vom selben JavaScript-Interpreter verarbeitet werden, besteht trotz aller Vorsichtsmaßnahmen stets die Gefahr, dass Scriptcode einer Website über Lücken in der Isolation auf Daten einer anderen Website zugreifen kann. Als zusätzliche, wenn korrekt implementiert starke Schutzmaßnahme böte sich auch hier ein dem Tainting ähnliches Konzept an.

In diesem Fall müssten die Daten bzw. Skripts allerdings mit mehr als nur einem 1-Bit-Taint-Flag versehen werden. Erforderlich wäre eine Kennung, die jedes von einer Website stammende, mittels JavaScript erreichbare Objekt fest an die Website koppelt. Anschließend kann der JavaScript-Interpreter z.B. bei jeder Datenverarbeitung in einer Funktion prüfen, ob die von der Funktion verarbeiteten Daten dieselbe Kennung besitzen wie die Funktion selbst. Ist dies der Fall, ist davon auszugehen, dass der Code von derselben Website stammt wie die zu verarbeitenden Daten. Schlägt diese Überprüfung fehl, hat eine Website Daten auf irgendeinem Weg erhalten, auf die sie keinen Zugriff haben dürfte.

Das Tainting-Konzept der Propagation von Taint käme in diesem Beispiel auf etwas andere Weise zum Einsatz: Wenn ein Script einer Website ein neues Datum „erzeugt“ (beispielsweise durch eine Berechnung oder durch einen asynchronen HTTP-Request), erhielte dieses neue Datum automatisch die Kennung des erzeugenden Skripts und wäre daraufhin an die erzeugende Website gekoppelt. Ein „Diebstahl“ durch Skripts einer anderen Website wäre aufgrund der permanenten Checks nicht mehr möglich.

Diese Idee, Labels zur Isolation von Websites zu nutzen, birgt evtl. eine Gefahr für die JavaScript-Performance, deren Auswirkungen schwer abzuschätzen sind und über praktische Tests ermittelt werden müssten. Insbesondere ist

schwer abzuschätzen, wie weit der Grundansatz im Hinblick auf Performance optimiert werden kann. Der Sicherheit jedoch wäre eine Implementierung dieses Verfahrens definitiv zuträglich.

4.1.3 Zuverlässiges Entfernen von JavaScript-Code beim Verlassen einer Website

Dadurch, dass der JavaScript-Interpreter samt dem zugehörigen Script-Speicherbereich global im Browser nur einmal existiert und nach dem Verlassen einer Website nicht zerstört wird, konnte es bereits mehrfach dazu kommen, dass JavaScript-Code von einer Website das Verlassen derselbigen „überlebt“. Dieses Problem ist in gewisser Weise verwandt mit den zuvor beschriebenen Gefahren einer unzureichenden Isolation zwischen Website und UI sowie zwischen mehreren Websites: Script kann beispielsweise durch eine Injektion in den Bereich der UI-Scripts oder in den Scriptbereich einer anderen Website das Schließen der Site, von welcher der Code ursprünglich stammte, überleben.

Ein sehr schöner Fall einer solchen Sicherheitslücke wurde unlängst im IE7 gefunden⁵. Dabei gelang es einer Website, ein eigenes Script in den Bereich einer anderen geöffneten Site zu injizieren, so dass dieses Script anschließend trotz Verlassen der ersten Site auf der zweiten Site weiterhin aktiv war und jegliche Tastatureingaben mitlesen konnte⁶.

Eine an Tainting angelehnte Verbesserungsmöglichkeit bestünde auch hier im Versehen jeder Funktion und jedes Objekts einer Website mit einem Tag. Das Tag kann nicht nur während der „Laufzeit“ einer Website wie in 4.1.2 beschrieben zur Prüfung herangezogen werden, sondern auch nach dem Verlassen einer Site (d.h. dem Schließen des letzten Tabs/Fensters) genutzt werden, um in einer Art Garbage-Collection-Prozess den kompletten JavaScript-Speicherraum von den der betreffenden Site zugeordneten Objekten und Funktionen zu säubern.

4.2 Abwehr von Cross-Site-Scripting-Angriffen

Tainting eignet sich auch als Mittel zur Verhinderung von Cross-Site-Scripting-Angriffen, welche gern zum Ausspionieren sensibler Informationen wie Login/Passwort-Kombinationen oder Session-Cookies genutzt werden. Ein sehr gutes Paper zu diesem Thema hat eine Forschergruppe an der TU Wien zusammen mit der University of California verfasst [1]. Die Idee dazu beruht auf der automatischen

⁵„Geister bedrohen Internet-Explorer-Anwender“: <http://www.heise.de/security/Geister-bedrohen-Internet-Explorer-Anwender-/news/meldung/110083>

⁶Proof-of-Concept: <http://sirdarckcat.blogspot.com/2008/05/ghosts-for-ie8-and-ie75730.html>

Markierung (also dem „Tainting“) von sensitiven Informationen im Browser (beispielsweise Formulareingaben, Cookies, URLs,...), kombiniert mit einer Propagation des Taint-Flags und einer Erweiterung sämtlicher Wege, über welche Daten potenziell an den Server eines Angreifers gesendet werden können: Als „sensitiv“ markierte Daten können ohne Weiteres nur noch an den Server gesendet werden, von welchem die Website ursprünglich stammte. Versucht ein Script, sensitive Daten (worunter dank Taint Propagation auch sämtliche Daten fallen, die Kontakt mit sensitiven Daten hatten) an einen anderen Server zu senden, wird dieser Versuch erkannt und eine Warnmeldung an den Benutzer ausgegeben, über welche er die Kommunikation mit dem fremden Server autorisieren oder unterbinden kann.

4.3 Noch mehr Cross-Site-Datenaustausch...

Eine andere Anwendung könnte im gezielten Erlauben von Datenaustausch zwischen Scripts unterschiedlicher Seiten im Browser liegen. Die Same Origin Policy unterbindet (so sie fehlerfrei implementiert wurde) solchen Datenaustausch vollständig, jedoch sind gerade in Verbindung mit Web-2.0-Ansätzen einige Funktionalitäten denkbar, die auf Basis einer sicheren Austauschmöglichkeit von Daten im Browser realisiert werden könnten. So könnte beispielsweise eine Online-Banking-Applikation ohne Copy&Paste oder „Abtippen“ die Kontodaten einer parallel in einem anderen Browser-Tab geöffneten Webshop-Bestellung in eine Überweisung übernehmen. Voraussetzung wäre, dass einer Webapplikation ein Mittel zur Verfügung steht, mit welchem es bestimmte Daten als „öffentlich zugänglich“ markieren kann - und ein solches Mittel könnte Tainting darstellen (in diesem Fall umgekehrt angewendet, d.h. zur Markierung von frei zugänglichen statt privaten Daten). In einer erweiterten Version könnte über einfaches binäres Tainting hinaus auch ein Mechanismus zur Zugriffsbeschränkung implementiert werden, der es erlaubt, einem Datum eine Liste von Websites mitzugeben, welche auf das Datum zugreifen dürfen.

Allerdings dient eine solche Anwendung von Tainting nicht direkt der Sicherheit (sie erlaubt ja etwas in gewissen Grenzen, was zuvor vollständig unterbunden wurde), sie sei daher hier nur der Vollständigkeit halber erwähnt.

5 Fazit

Tainting ist nach wie vor einer der interessantesten Mechanismen im IT-Security-Bereich, der gerade im Zusammenhang mit modernen, in virtuellen Maschinen oder Interpretern ausgeführten Programmen heute mehr denn je eingesetzt werden könnte. Die besondere Eigenschaft der Weitergabe von „Taint“ beim Zusammentreffen zweier beliebiger

Programmobjekte verleiht ihm eine hohe Resistenz gegenüber typischer „Verschleierungs-Taktiken“, mit denen sensible Informationen gern in scheinbar harmlosen Daten verpackt und auf diese Weise an Sicherheitsmechanismen wie einfachen String-Vergleichen, Regular Expressions o.ä. vorbeigeschleust werden.

Auch im Browser-Bereich gibt es, wie auf den vorhergehenden Seiten gezeigt wurde, Anwendungsmöglichkeiten für das Tainting-Prinzip. Speziell beim Firefox-Browser könnte Tainting die Isolation von Scripts der privilegierten Benutzeroberfläche und „gemeinen“ Website-Scripts verbessern, die bereits mehrfach für Sicherheitslücken verantwortlich war (wenngleich eine solche Lösung niemals so effektiv wie komplett separierte JavaScript-Laufzeitumgebungen und -Interpreter sein kann, aber wenn diese Lösung aus welchen Gründen auch immer nicht in Frage kommen sollte lohnt es sich, Alternativen in Betracht zu ziehen). Allgemein würde sich Tainting aber auch als effektives Hilfsmittel gegen heimlichen Datenaustausch mittels Cross-Site-Scripting-Angriffen anbieten - eine solche Funktionalität würde jeden modernen Browser deutlich sicherer gegen das versteckte Mitlesen und Absenden von sensiblen Informationen machen - eine Angriffsart, die gerade in Zeiten komplexer, an vielen Stellen mit ungeprüftem User-Generated Content arbeitenden Webanwendungen immens an Popularität und damit Gefahrenpotenzial gewinnt.

Literatur

- [1] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, Giovanni Vigna, *Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis*, Secure Systems Lab, Technical University Vienna, University of California, Santa Barbara, http://iseclab.org/papers/xss_prevention.pdf