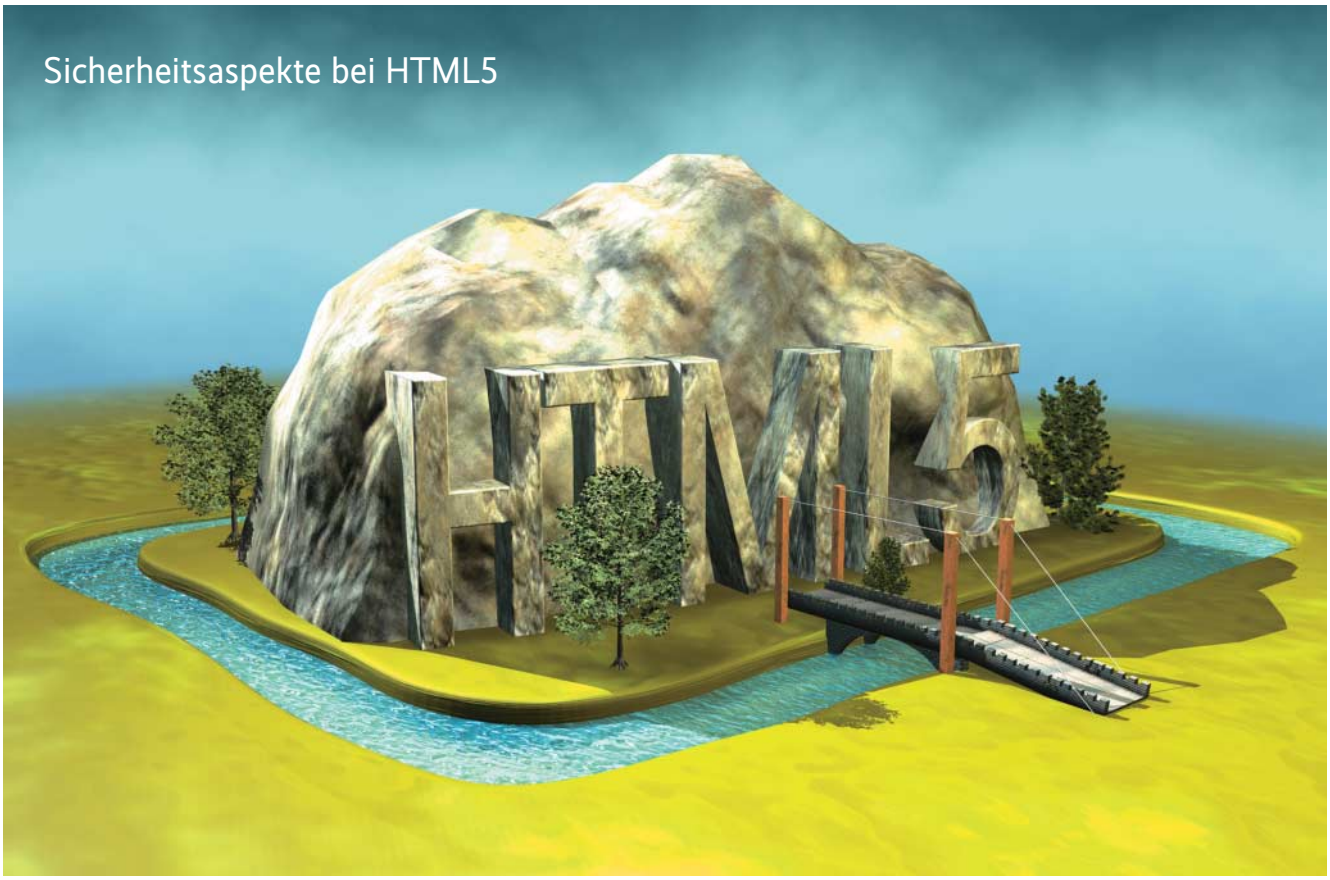


Felsenfest

Thomas Schreiber, Sven Schleier

Beim kommenden Webstandard HTML5 haben sich die Entwickler in Sachen Sicherheit viel Mühe gegeben. Doch mit den neuen Features wächst auch die Komplexität und damit die Angriffsfläche. Eine Übersicht über verbleibende sowie neue Risiken.

HTML5 ist der kommende Webstandard, den das W3C im vierten Quartal 2014 als Recommendation verabschieden will. Aber viele der Erweiterungen gegenüber HTML4 sind zumindest in den am meisten verbreiteten Browsern bereits implementiert. Sie können eingesetzt werden und viele Websites tun das auch schon. Welcher Browser in welcher Version über welche HTML5-Erweiterungen verfügt, kann man auf caniuse.com nachschlagen.

Auf die Sicherheit von Websites beziehungsweise -anwendungen haben die neuen Features erhebliche Auswirkungen. Konzepte wie die Content Security Policy (CSP) und das *sandbox*-Attribut in *iframes* verringern die Angreifbarkeit von Webanwendungen um einiges, und auch CORS (Cross-origin Resource Sharing) ist für bestimmte Anwendungsfälle sicherer als die bisher verwendeten Verfahren. Damit sind einige der positiven Auswirkungen von HTML5 genannt. Webentwickler sollten nicht zögern, diese Features in der eigenen Webanwendung einzusetzen.

Auf der anderen Seite stehen Techniken wie WebSockets oder Web Storage. Sie eröffnen ganz neue Möglichkeiten und damit neue Angriffsflächen im Browser. Und die signifikante Erhöhung der Komplexität – etwas, das immer mit einem Verlust an Sicherheit einhergeht – trägt dazu bei, dass die Browser-Welt mit HTML5 sicherheitstechnisch schwerer zu beherrschen sein wird als bisher.

Der Begriff HTML5 umfasst in diesem Artikel ganz umgangssprachlich auch diejenigen Konzepte, die streng genommen nicht oder nicht mehr der HTML5-Spezifikation angehören und in eigenen Standards vorangetrieben werden. Abbildung 1 gibt einen Überblick über die einzelnen Spezifikationen und ihre Verortung innerhalb der Standardisierungsgremien.

Gleiche Herkunft schafft Vertrauen

Um einige der Sicherheitsmechanismen in HTML5 verstehen zu können, muss man wissen, wie die Same Origin Policy (SOP) funktioniert. Die SOP ist fundamental für die Browsersicherheit, denn sie sorgt dafür, dass eine bösartige Webseite, die gleichzeitig mit beispielsweise dem Onlinebanking im Browser geladen ist, nicht auf die Inhalte der Onlinebanking-Webseite zugreifen kann.

Erreichen kann man dies über eine im Kern sehr einfache Regel, die, noch etwas weiter vereinfacht, so lautet: Der Browser lässt den Zugriff einer Seite (besser gesagt, des JavaScript-Codes in der Seite) auf eine andere Seite nur dann zu, wenn beide Seiten vom selben Host geladen worden sind und sie damit denselben Ursprung (Origin) besitzen.

Auch die Vorstellung, dass die SOP eine schützende Sandbox realisiert, trifft die Sache einigermaßen. Dass bei näherem Hinsehen noch Protokoll und Port übereinstimmen müssen und dass man die SOP unter bestimmten Umständen auch auf die Domain ausweiten kann, sei hiermit erwähnt, ist aber für das weitere Verständnis nicht so wichtig. Viele coole Features scheitern daran, dass die SOP sie aus Sicherheitsgründen nicht zulässt. Was wiederum dazu geführt hat, dass so mancher Entwickler Schlupflöcher gesucht und gefunden hat, um sie trotzdem zu realisieren – zumeist auf Kosten der Sicherheit.

HTML5 ist unter anderem der im Großen und Ganzen gelungene Versuch, elegante Techniken zur Realisierung sicherheitstechnischer Anforderungen bereitzustellen. Im Folgenden ist immer, wenn es für das Verständnis leichter ist, vereinfachend von „Host“ die Rede, wo man streng genommen „Origin“ oder „Domain“ sagen müsste.

XSS, der SOP-Killer

Die mit Abstand am meisten verbreitete Sicherheitslücke in Webanwendungen ist nach wie vor das Cross-Site Scripting (XSS). Es tritt auf, wenn die Webanwendung bei der Ausgabe von Daten – zumeist solchen, die ein Benutzer zuvor über ein Formular eingegeben hat – nicht prüft, ob in ihnen HTML-Code enthalten ist. Dann kann ein Angreifer ganz gezielt Code eingeben, der beim Laden der Seite im Browser seine Wirkung entfaltet. Durch Verwendung des *<script>*-Tags (und auf verschiedene andere Arten) lässt sich damit auch JavaScript in die Seite einschleusen. Kann der Angreifer nun einen derart präparierten Aufruf im Browser einer anderen Person platzieren, kommt der Code dort zur Ausführung. Das Fatale daran: Das JavaScript kommt zwar gewissermaßen „von außen“, wird aber im Kontext der umgebenden Seite ausgeführt. Somit hat es freien Zugang zu all dem, was die SOP dem Skript verbieten würde, wäre es von einem fremden Host geladen worden. Um im obigen Bild zu bleiben: So erhält der Angreifer nun doch den Zugriff auf das Onlinebanking. Eine XSS-Schwachstelle hebt also den SOP-Schutz für die betroffene Seite komplett auf und wirkt sich meistens auch auf die Sicherheit der gesamten Website aus.

Kompromittierung mit schlimmen Folgen

Und genau hier hat eine grundlegende Schwäche von HTML5 ihre Wurzel: Die Sicherheit der mächtigen HTML5-Erweiterungen stützt sich an vielen Stellen auf die SOP. Jedes XSS reißt in dieses Konzept ein großes Loch hinein und eröffnet dem Angreifer enorme Möglichkeiten, etwa, seine Angriffe besser zu verstecken oder sie dauerhaft im Browser zu hinterlegen.

Eine XSS-Schwachstelle ist zwar sehr leicht zu beheben, wenn sie erst einmal aufgefallen ist, aber die Erfahrung der letzten Jahre zeigt, dass es enorm schwer ist, eine Website dauerhaft von XSS freizuhalten. Man muss also andere Ansätze finden. Warum kann beispielsweise eine Seite dem Browser nicht einfach sagen, er soll die Ausführung von JavaScript bei ihr komplett unterbinden? Dann könnte ein über ein XSS injiziertes JavaScript

iX-TRACT

- Bei HMTL5 haben die Autoren einen besonderen Schwerpunkt auf die Sicherheit gelegt und den Webentwicklern mit neuen Features wichtige Instrumente zur Verfügung gestellt – die sie ohne Detailwissen allerdings nur schwer nutzen können.
- Ein Schwachpunkt ist, dass sich viele Sicherheitsmechanismen auf die „Same Origin Policy“ stützen. Wird sie durch eine XSS-Schwachstelle kompromittiert, ist die Webanwendung wieder angreifbar.
- Webentwickler sollten sich Konzepte wie CSP oder CORS zunutze machen, die Applikationen aber, wo es geht, mit zusätzlichen Sicherheitsmechanismen schützen.

nichts ausrichten und zumindest diese Gefahr wäre gebannt. Mit HTML5 geht das tatsächlich. Die Seite muss nur einen Content Security Policy Header (CSP) wie den folgenden ausliefern:

```
X-Content-Security-Policy: script-src 'none'
```

Schon wird in ihr kein JavaScript ausgeführt. Selbst Seiten, die auf JavaScript angewiesen sind, lassen sich so schützen. Eine Strategie, die das ermöglicht, lautet: Verwende in der Seite kein Inline-JavaScript, sondern lade alle Skripte nach, am besten vom eigenen Host nach dem Muster *<script src="script1.js">*. Setzt man jetzt den Header

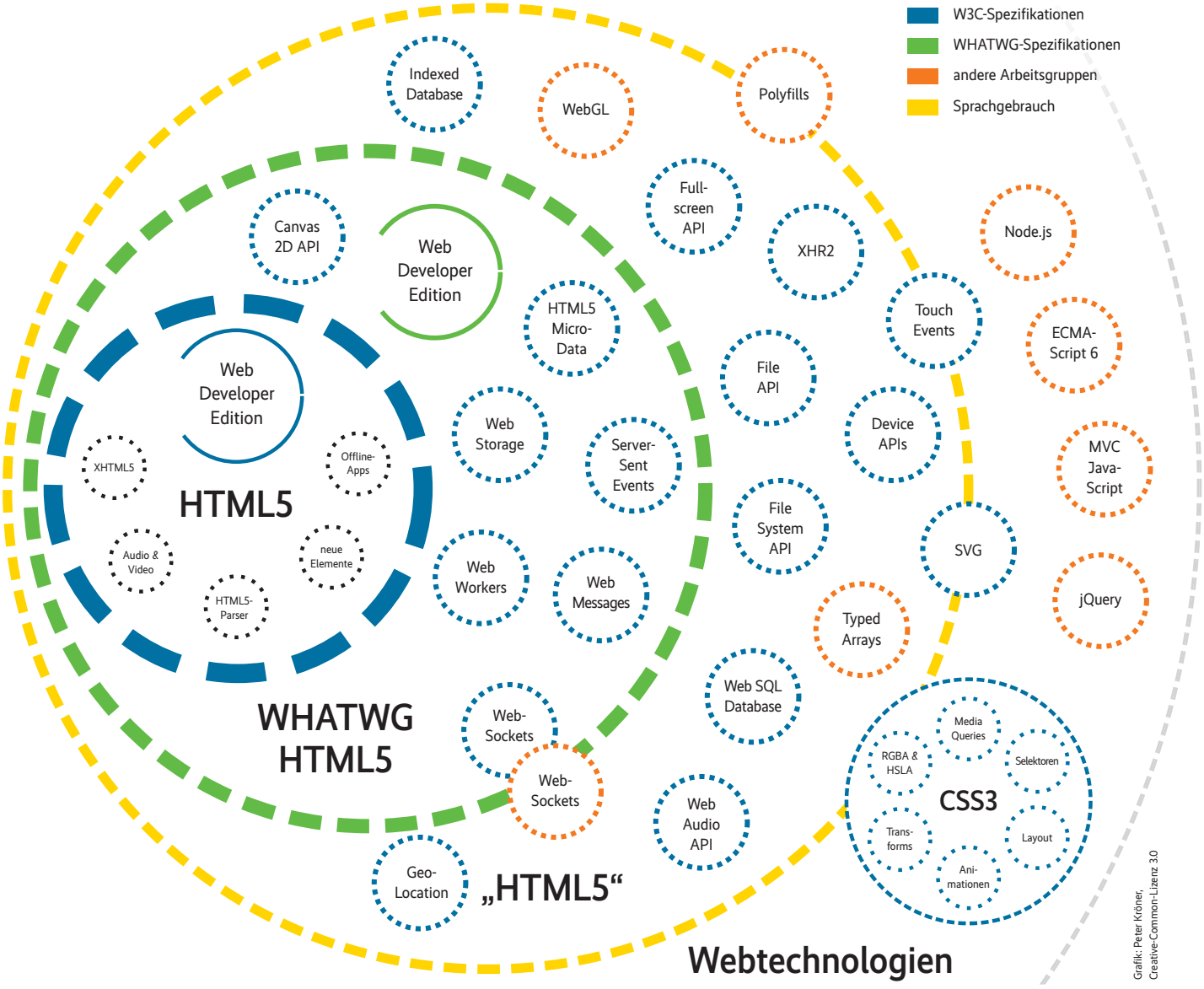
```
X-Content-Security-Policy: script-src "self"
```

stellt die CSP sicher, dass (über XSS injiziertes) Inline-JavaScript und solches, das von anderen (bösartigen) Hosts geladen wird, nicht zur Ausführung gelangt, wohl aber die vom eigenen Host geladenen (gutartigen) Skripte. Zusätzlich blockiert die CSP die Ausführung des *eval()*-Befehls und ähnlicher Befehle, sodass auch kein in Textform injizierter Code zur Ausführung kommen kann.

CSP kann aber noch viel mehr, etwa das Laden von Bildern und Frames sowie die Verbindungsaufnahme per WebSockets und XHR-Requests et cetera verhindern. Oder genauer gesagt kann sie einzelne Eigenschaften einer Seite individuell ein- und ausschalten. Insgesamt lässt sich durch geschicktes Einbinden der CSP in die Anwendungsarchitektur die Gefahr der Skriptausführung durch XSS weitgehend beherrschen, ohne deren Funktionen zu beeinträchtigen. Die Policy in vollem Umfang in bestehende Webanwendungen einzubinden ist jedoch nicht immer möglich, zum Beispiel wenn Webtracking oder Werbeeinblendungen nur mit Inline-JavaScript funktionieren. Überdies ist sie erst in Firefox, Chrome und Safari inklusive iOS vollständig implementiert, im Internet Explorer aber nur teilweise. Änderungen sind zudem nicht auszuschließen, da die CSP noch nicht offiziell verabschiedet ist.

Sicherheit versus Vielfalt und Flexibilität

Die SOP ist restriktiv, lässt dafür aber kaum Fehlkonfigurationen zu, die die Sicherheit gefährden. Auf der Kehrseite können Webseitenentwickler oder -betrei-



Diverse Webtechnologie-Spezifikationen werden umgangssprachlich unter dem Begriff „HTML5“ subsumiert (Abb. 1).

ber nicht von den Vorzügen und der Flexibilität des Web 2.0 profitieren, wenn es darum geht, Dienste von Dritten zu integrieren. Ein Beispiel ist das Einbinden aktueller Tweets zu bestimmten Schlagworten. Da das aber sehr nützlich sein kann, haben sich in den letzten Jahren verschiedene Techniken zur Umgehung der SOP etabliert – darunter das verbreitete JSONP (JSON with padding) –, denen allen eines gemeinsam ist: Sie untergraben die Sicherheit des Browsers.

Mit Cross Origin Resource Sharing (CORS) stellt HTML5 eine sichere Technik zur Umsetzung derartiger Anwendungsfälle bereit, die alle aktuellen Browser inklusive der mobilen Varianten unterstützen. Von CORS profitiert in erster Linie das *XMLHttpRequest*-Objekt (XHR), mit dem man nun auch domainübergreifend arbeiten kann.

Die Steuerung per CORS erfolgt beim Ausliefern einer Seite durch den Server. Dieser übermittelt dem Browser mit den per XHR angeforderten Daten im Access-Control-Allow-Origin Header eine Liste von Hosts.

```
Access-Control-Allow-Origin: ⤷
                    http://www.example1.com ⤷
                  https://secure.example2.com
```

Der Browser erlaubt nun allen Seiten, die von http://www.example1.com, https://secure.example2.com oder dem eigenen Host geladen werden, den Zugriff auf die soeben geladene Seite oder, falls es sich zum Beispiel um eine JSON-Antwort handelt, auf die geladenen Daten. Alle anderen Seiten erhalten keinen Zugriff. Ohne CORS hätten gemäß SOP ausschließlich Seiten, die vom selben Host geladen werden, Zugriff.

Durch Angabe eines Sterns statt einer Hostliste, also

```
Access-Control-Allow-Origin: *
```

kann man den Zugriff für alle Seiten freigeben, egal von wo sie stammen. Diese Option birgt ein latentes Sicherheitsrisiko in sich, denn nicht alle Entwickler und Betreiber machen sich die Mühe, CORS richtig zu verstehen. Die Gefahr ist groß, dass sie, sei es aus Bequemlichkeit oder weil andere Webseiten es auch so machen, die in vielen Fällen gefährliche Konfiguration mit dem * übernehmen.

Auch interne Rechte begrenzen

Schauplatz Intranet. Auf dem Host *sso.intern* stellen die Zuständigen einen Single-Sign-on-Dienst zur Verfügung. Um eine hohe Flexibilität zu erreichen und weil alle anderen Anwendungen und Hosts im Intranet des Unternehmens als vertrauenswürdig gelten, stellen sie den *sso.intern* so ein, dass er alle Anfragen mit Access-Control-Allow-Origin: * beantwortet. Damit hat im generell als vertrauenswürdig geltenden Intranet jeder Host darauf Zugriff.

Was keiner bedacht hat: Auch alle aus dem Internet geladenen Seiten im Browser eines Intranet-Nutzers erhalten Zugriff auf diesen Host. Ein Angreifer, der die internen Verhältnisse kennt, kann nun einen Mitarbeiter des Unternehmens auf eine entsprechend präparierte Webseite locken, die aus dem Browser des Mitarbeiters heraus auf *sso.intern* zugreifen, aufgrund des Sterns die zurückgelieferten Daten auslesen sowie an den Angreiferserver übermitteln kann.

Man könnte meinen, das Setzen von Access-Control-Allow-Origin auf **.intern* würde das Problem lösen. Dem ist aber nicht so, denn CORS unterstützt diese Form von Wildcards nicht. Stattdessen muss die Behandlung auf Serverseite in der Anwendung erfolgen, indem diese den vom Browser mitgeschickten Origin Header prüft. Nur wenn er aus der *.intern*-Domain stammt, etwa von www.intern, liefert der Server die Response aus, zusammen mit dem Header

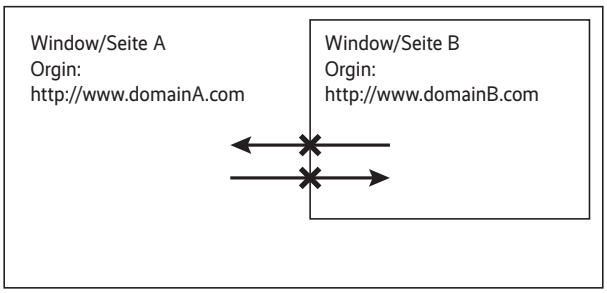
```
Access-Control-Allow-Origin: www.intern
```

der den gewünschten, dedizierten Zugriff gewährleistet.

Vorsicht bei Freigaben

Auch aus einem weiteren Grund sollte man generell die Freigabe der eigenen Origin für Dritte restriktiv handhaben: Ohne CORS kann die SOP nur unterlaufen werden, wenn die eigene Website eine XSS-Schwachstelle hat. Oder anders herum: Hat man dafür gesorgt, dass die eigene Website (streng genommen der Host oder die Domain) frei von XSS ist, kann man sich auf die SOP verlassen und ein missbräuchlicher Zugriff von einer bösartigen auf die eigenen Seiten ist nicht möglich. Kommt CORS zum Einsatz, kann der Angreifer sich nun auch auf allen anderen Hosts, denen der Zugriff per Cross-Domain-Liste erlaubt ist, nach XSS-Schwachstellen umsehen, um über sie den Zugriff auf die per CORS geschützte Ressource zu erhalten.

CORS in der bis hierher beschriebenen Form ist auf die sogenannten einfachen Requests *HEAD*, *GET* und *POST* in Verbindung mit bestimmten Content-



Dank *iframes* kann man zwar Inhalte Dritter sicher in eine Website einbetten, doch ohne weitere Schutzmaßnahmen bleibt die Webseite trotzdem angreifbar (Abb. 2).

Typen anwendbar. Es ermöglicht all das, was auch die eingangs angesprochenen Ersatzlösungen wie JSONP bieten, nur eleganter und besser steuerbar, und macht diese damit obsolet.

CORS geht aber noch weiter: Es erlaubt domainübergreifende Zugriffe auch für die anderen HTTP-Methoden, etwa *PUT* und *DELETE*, das Versenden von selbstdefinierten Request-Headern und den Zugriff auf beliebige Datentypen der Serverantworten wie JSON und XML. Aus Sicherheitsgründen war das alles für XHR ohne CORS nicht erlaubt.

Damit nun die Sicherheit nicht auf der Strecke bleibt, muss man umständlicher vorgehen. Mit einem sogenannten „Preflight Request" prüft der Browser vorab, ob der Webserver den geplanten domainübergreifenden Zugriff in der gewünschten Form erlaubt. Er führt den eigentlichen Request daraufhin nur aus, wenn dieser die zurückerhaltenen Restriktionen auch einhält. Das Ergebnis des Preflight Request muss der Browser sich merken. Die Zeitspanne gibt ebenfalls der Server dem Browser vor. Sie sollte, den Anforderungen entsprechend, möglichst kurz sein, damit später eingeführte Einschränkungen sich auch auswirken.

Whitelisting erschwert Denial of Service

Daraus abgeleitete Sicherheitsregeln: Der zuständige Webentwickler sollte ein Whitelisting implementieren, das nur das Bearbeiten von Anfragen erlaubter Origins zulässt. Dadurch können Anfragen, die rechenintensive Operationen ausführen, aber lediglich einen Denial of Service auf einem Webserver provozieren wollen, in ihrer Effektivität eingeschränkt werden. Da sich die Konfiguration auf Webseitenebene fein abstufen lässt, sollten nur diejenigen Seiten einer Webseite CORS-Anfragen unterstützen, für die es vorgesehen ist – und auch dann nur so restriktiv wie möglich. Zu guter Letzt sollte der Webserver Access-Control-Allow-Origin nicht als Standard im Response-Header bei jeder Seite ausliefern.

iframes sind ein sicheres Konzept, um Fremdinhalte in die eigene Seite einzubetten, denn sie unterliegen der SOP (Abb. 2). Das bedeutet, ein im *iframe* enthaltenes JavaScript hat keinen Zugriff auf die Inhalte der umgebenden Seite, wenn die Origins unterschiedlich sind. Doch angreifbar ist eine Website auch ohne Scripting, zum Beispiel dadurch, dass das *iframe* ein Login-Formular im Layout der ladenden Seite nachahmt und den Benutzer glauben macht, es gehöre zur übergeordneten Seite. Gibt der Benutzer hier seine Zugangsdaten ein, fallen sie in die Hände des Fremdanbieters. Um die Sicherheit von *iframes* weiter zu erhöhen, führt HTML5 das *sandbox*-Attribut für *iframes* ein. Ein damit versehenes *iframe* wie

```
<iframe sandbox
src="http://domainB.com/seite.html"></iframe>
```

darf nun keine Formulare mehr abschicken, der genannte Missbrauch ist somit unterbunden. Das *sandbox*-Attribut verhindert ebenfalls das Ausführen von Skripten, außerdem den Zugriff auf andere *iframes* mit derselben Origin (auf die per SOP der Zugriff erlaubt ist) und einiges mehr. Durch Angabe geeigneter Keywords lässt sich diese Beschränkung für die einzelnen Eigenschaften selektiv steuern. Ohne solche Keywords ist der Maximalschutz aktiviert.

Eine sichere Strategie, mit der man Fremdinhalte in die eigene Seite einbinden kann, sieht damit so aus: Die Inhalte werden über eine andere Domain als die der Hauptseite in einen *iframe* geladen. Das *iframe* hat das *sandbox*-Attribut gesetzt und dabei möglichst wenige Eigenschaften ausgenommen.

Aber Achtung: Nicht alle Browser, insbesondere nicht die älteren Versionen, kennen dieses Attribut. Das führt zwar zu keinem Fehler, aber auch nicht zu der beabsichtigten Schutzwirkung. Dieses Verfahren kann daher nur als zusätzliche Sicherheitsmaßnahme dienen.

Kommunizieren per Web Messaging

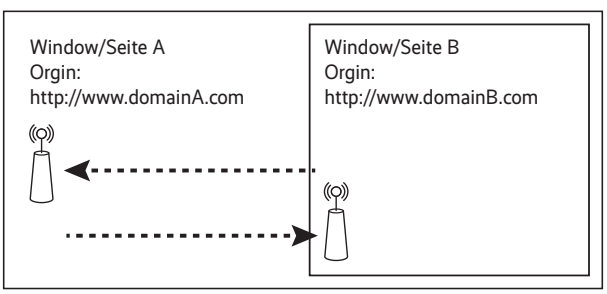
Ist das *sandbox*-Attribut angetreten, *iframes* noch stärker von der umgebenden Seite zu isolieren, so kommt Web Messaging (Abb. 3) zum Einsatz, wenn zwischen *iframe* und umgebender Seite ein sicherer Datenaustausch erfolgen soll.

Auch hier besteht der Anwendungsfall darin, dass die Herkunft (Origin) der miteinander kommunizierenden Seiten unterschiedlich ist. Bei gleicher Herkunft könnte dazu einfach die Schnittstelle DOM (Document Object Model) des jeweils anderen Frame dienen, was bei unterschiedlicher Herkunft jedoch leicht zu missbrauchen wäre und daher dank SOP nicht möglich ist. Die Erweiterung der bestehenden Navigationstechnik auf fremde Hosts wäre sicherheitstechnisch nicht lösbar gewesen und so ist man in HTML5 einen anderen Weg gegangen, den des Nachrichtenaustausches.

Die Nachrichtenübermittlung über Web Messaging erfolgt ausschließlich im Browser, die Webserver der geladenen Seiten sind dabei nicht mehr involviert. Das Versenden erfolgt durch die Methode *postMessage()*. Sie wird im vorliegenden Beispiel in der Seite A aufgerufen und übermittelt die Message „Hallo" an das Ziel-*iframe* (window_B):

```
window_B.postMessage("Hallo", ⤸
                     "http://www.domainB.com");
```

Im zweiten Parameter muss man zusätzlich die Origin des Ziels (nicht der gesamte URL) angeben, also „http://www.domainB.com". Damit unterbindet man die Übermittlung von Daten, falls window_B nicht das erwartete *iframe* referenziert – denn es könnte es sich ja beispielsweise um ein Passwort handeln, das



Das Web Messaging soll im Browser einen sicheren Datenaustausch zwischen Webseiten ermöglichen, die nicht aus derselben Quelle stammen (Abb. 3).

nicht an den Falschen gelangen soll. Allerdings kann diese Zusatzprüfung auch wieder umgangen werden, indem an der Stelle der String „*“ übergeben wird, was aus Sicherheitsgründen zu vermeiden ist.

Auf Empfängerseite muss man einen Event-Listener definieren, der die Nachrichten annimmt und die Verarbeitung anstößt. In B müsste sich also ein Codefragment wie dieses befinden:

```
window_B.addEventListener("message", 7
                        receiveMessage, false);
function receiveMessage(event){  if 7
       (event.origin !==  http://domainA.com )
          return;
   console.log(event.data);
}
```

Zeile 1 definiert den Event-Listener. Geht eine Nachricht ein, ruft er die Funktion *receiveMessage()* auf und erhält in *event.data* die Nachricht „Hallo“. *event.origin* übergibt die Information, woher der Absender (A) stammt. Die hier gezeigte Prüfung der Origin und der Abbruch der Verarbeitung, falls sie nicht der erwarteten entspricht, ist immer auszuführen, wenn es sich nicht um eine öffentliche Schnittstelle handelt. Anderenfalls besteht kein Schutz vor einem Missbrauch durch Dritte. Zusätzlich wird (in diesem Beispiel nicht verwendet) *event.source* übergeben, in dem die Referenz auf das Window A enthalten ist. Sie ist erforderlich, um Nachrichten zurückzusenden, und kann zusätzlich der Prüfung des Absenders dienen.

Kreuz und quer miteinander reden

Neben der Kommunikation einer Webseite mit einer eingebetteten Seite existiert noch das „Channel Messaging“. Damit können zum Beispiel zwei in derselben Seite eingebettete *iframes* miteinander kommunizieren, die keine direkte Vertrauensbeziehung untereinander haben, sondern nur über die sie einbettende Seite. Webseiten, die kein Messaging anbieten wollen, schützen sich vor Missbrauch dieses Features ganz einfach dadurch, dass sie keinen Event-Listener für Message-Events anbieten.

Da die Datenübertragung bei Web Messaging ohne Serverbeteiligung erfolgt, kann dieser die auf diesem Wege eingegangenen Daten auch nicht validieren. Zur Vermeidung von Cross-Site Scripting (XSS) muss die Validierung daher im empfangenden Client stattfinden. Es handelt sich hier um eine Verschiebung der Sicherheitsprüfung vom Server auf den Client. Die alte Web-Application-Security-Regel „Datenvalidierung hat auf dem Server zu erfolgen“ ist ab sofort zu erweitern auf „in bestimmten Fällen muss der Client ebenfalls validieren“. Generell verzichten sollte man darauf, empfangene Messages mit *eval()* auszuwerten oder sie in den DOM einzubauen.

WebSockets: Praktisch, aber gefährlich

WebSockets ermöglichen eine Browser-Server-Kommunikation jenseits des klassischen Schemas, bei dem der Browser dem Server zunächst eine Anfrage senden muss, damit dieser seine Daten übermitteln kann. Ist die WebSocket-Verbindung einmal hergestellt, bleibt sie bestehen, und beide Seiten können unabhängig voneinander und ohne großen Overhead Daten an die Gegenseite übermitteln. Damit lassen sich Verfahren wie Server-Push auf effiziente Weise abwickeln. Zur Umsetzung dieser Technik mussten die HTML5-Architekten von HTTP abrücken. Das Protokoll ist nur noch für die Anforderung der WebSocket-Verbindung im Spiel, dann erfolgt der Wechsel zum WebSocket-Protokoll, das dieselbe TCP/IP-Verbindung nutzt. WebSockets können, ebenso wie HTTP, unverschlüsselt oder verschlüsselt aufgebaut werden. Der Aufruf einer WebSocket-URI erfolgt nach folgendem Schema:

```
ws://www.example.com/chat
wss://www.example.com/abhörsichererchat
```

WebSockets statten die Browser mit einer Kommunikationsmöglichkeit einer ganz neuen Art aus, deren Sicherheitsimplikationen noch nicht weitreichend untersucht sind. Sorge bereiten sollte jedem Sicherheitsverantwortlichen die Tatsache, dass mit der Bereitstellung eines WebSocket-Servers im eigenen Unternehmen über die offenen und zumeist unbewachten HTTP- und HTTPS-Ports nun von draußen nach drinnen bidirektionale, persistente und in Echtzeit arbeitende Kanäle gelegt werden können.

Tatsache ist, dass viele Firewalls, IDS und IPS das WebSocket-Protokoll noch nicht umfassend analysieren und daher Angriffe darüber auch noch nicht entdecken können. Das auf der Black Hat 2012 veröffentlichte freie Werkzeug Waldo demonstriert verschiedene Angriffsvarianten und kann dem Studium der Sicherheit von WebSockets dienen.

Bei der Implementierung von WebSockets sind folgende Sicherheitsregeln zu beachten:

– Die Webanwendung am Serverende des WebSocket muss genauso wie eine „herkömmliche“ Webanwendung frei von Schwachstellen sein und eine Validierung aller eingehenden Daten durchführen, denn diese sind mit geeigneten Werkzeugen beliebig manipulierbar.
– Zum Erschweren von DoS-Angriffen auf den Server sollte dieser den Origin Header, der beim Aufbau der Verbindung mitgeschickt wird, auswerten und die Verbindung nur für erlaubte Clients aufbauen. Die Annahme, dass dieser ja auch gefälscht werden kann, ist in diesem Zusammenhang nicht gültig.
– *ws://*-Verbindungen erfolgen unverschlüsselt und können somit ausgelesen oder von einem „Man in the Middle“ manipuliert werden. Das verschlüsselte *wss://* ist daher vorzuziehen.
– Nachrichten, die der Browser über WebSockets empfängt, sollten sicher interpretiert, das heißt nicht direkt in das DOM eingebaut oder mit *eval()* ausgeführt werden. Wenn es sich um JSON handelt, ist *JSON.parse()* ein sicheres Verfahren zur Auswertung der Antwort.
– WebSockets besitzen selbst keine eigenständigen Möglichkeiten zur Authentifizierung. Daher muss man auf die Mechanismen der Protokolle auf Applikationsebene zurückgreifen.

Arbeiter im Hintergrund

Web Workers verleihen dem Browser eine wichtige Fähigkeit, damit dieser als Anwendungsplattform oder gar Betriebssystem fungieren kann: die Möglichkeit, verschiedene Tätigkeiten parallel per Multi-Threading auszuführen. Das war bisher selbst mit den asynchronen Aufrufen des XMLHttpRequest nicht machbar. Anwendungsfälle für Web Workers sind zum Beispiel das Abrufen von Daten im Voraus für die spätere Verwendung, im Hintergrund ablaufende Rechtschreibprüfungen oder die Analyse und Bearbeitung von Mediendaten. Web Workers sind an das aufrufende Window-Objekt gebunden und werden vom Browser spätestens dann beendet, wenn der Benutzer das betreffende Fenster oder den Tab schließt.

Für eine sichere Umsetzung des Konzepts haben die Entwickler den Web Workers eine Reihe von Einschränkungen auferlegt. Sie fallen etwa, was nicht weiter verwundert, unter die SOP und können nicht mit Seiten anderer Origins kommunizieren. Und sie laufen in einer eigenen Sandbox ab, was in diesem Fall unter anderem bedeutet, dass sie keinen direkten Zugriff auf die DOM-API sowie das *document*-, *window*- und *parent*-Objekt der Webseite besitzen.

Auf der anderen Seite werden die sicherheitsrelevanten Abhängigkeiten sehr schnell unübersichtlich, wenn man in Web Workers die Möglichkeiten nutzt, untergeordnete Web Workers zu starten, WebSocket-Verbindungen aufzubauen, CORS-Requests auszuführen und Web Messaging oder das Skriptimport-Feature zu nutzen. So können auch DOM-Veränderungen per Web Messaging indirekt an die Webseite übergeben werden, die diese Änderungen einbaut.

Der Verantwortliche muss sich in solchen Szenarien genau mit den Sicherheitsimplikationen auseinandersetzen und die Sicherheitsregeln zum Umgang mit den jeweiligen Technologien beherzigen. Startet eine Benutzereingabe Web Workers oder wird diese an Web Workers zur Verarbeitung weitergereicht, ist darauf zu achten, dass kein Schadcode übergeben oder der Browser durch Starten vieler Web Workers überlastet werden kann.

Web Storage: Speicher im Browser

Web Storage ermöglicht Webanwendungen, die nicht auf die ständige Verbindung zum Server angewiesen sind, und er beschleunigt deren interaktive Bedienung. Die Bezeichnung wird meistens als Oberbegriff für Local Storage, Session Storage und Web SQL Database verwendet; neuerdings ist die IndexedDB hinzugekommen.

Schlussendlich dienen alle Web-Storage-Techniken dem Ziel, in Webanwendungen anfallende Daten auf dem Client (also im Browser) abzulegen und darauf strukturiert und komfortabel per JavaScript zuzugreifen. Hatten Entwickler zu diesem Zweck bisher nur die stark limitierten Cookies zur Verfügung, können sie jetzt gleich auf eine ganze Palette von Möglichkeiten zurückgreifen:
– Local Storage erlaubt, per JavaScript einfache Name-Value-Paare abzulegen, auf die man über den Namen wieder zugreifen kann. Diese Technologie ist in allen relevanten Browsern verfügbar und ermöglicht Entwicklern, Daten so abzulegen, dass sie auch nach dem Neustart des Browsers noch da sind. Erst durch explizites Löschen verschwinden sie wieder.
– Session Storage ist dem Local Storage sehr ähnlich und unterscheidet sich nur darin, dass die gespeicherten Daten beim Schließen des Browsers automatisch gelöscht werden.
– Web-SQL-Database ist eine clientseitige SQLite-Datenbank, die es ermöglicht, auch komplexe Datenstrukturen zu verwalten. Das W3C hat diesen Draft jedoch als „deprecated“ klassifiziert, was bedeutet, dass der Standard nicht mehr vorangetrieben wird und Browserhersteller dazu angehalten sind, diese Technologie durch
– IndexedDB zu ersetzen. Dieser allgemein gehaltene Datenbank-Entwurf ist allerdings noch sehr jung und wird bislang nur rudimentär von den gängigen Browsern unterstützt.

Der Zugriff auf den Speicher ist in allen Fällen durch die SOP geschützt – und damit durch eine XSS-Schwachstelle vollständig kompromittierbar. Ein Angreifer kann leicht vorhandene Daten auslesen oder löschen, er kann neue oder falsche Daten hinzufügen, ohne dass der Server und die darauf liegende Anwendung davon Kenntnis erlangen.

Onlinequellen

[a] Übersicht über unterstützte Webbrowser-Technologien, u. a. HTML5
caniuse.com

[b] Zusammenfassung von Ressourcen zu HTML5-Security
html5security.org

[c] Best Practices zum Einsatz von HTML5-Technologien der OWASP
https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

[d] Bugs in HTML5 und ihre Verbreitung in unterschiedlichen Browsern
html5sec.org

[e] Beispiel für den Einsatz von Channel Messaging
http://dev.opera.com/articles/view/window-postmessage-messagechannel/#channel

[f] Proof-of-Concept Waldo zum Test von Schwachstellen von WebSockets
https://community.qualys.com/blogs/securitylabs/2012/08/03/the-tiny-mighty-waldo

[g] Der Einsatz des Sandbox-Attributs von *iframes*
msdn.microsoft.com/en-us/hh563496

Unter „Alle Links“ finden sich außer den hier genannten zahlreiche weitere Links zu den einzelnen Aspekten der HTML5-Sicherheit.

Das führt in der Praxis zu einem bedeutenden Sicherheitsverlust gegenüber dem „konkurrierenden“ Modell für serverunabhängige Webanwendungen in Form von Apps, namentlich iOS- und Android-Apps. Wie bei diesen lassen sich mit Web Storage Anwendungen realisieren, die (möglicherweise vertrauliche) Daten für die Offline-Nutzung oder zur Komforterhöhung auf dem Client speichern. Allerdings sind echte, sogenannte „native“ Apps im Allgemeinen nicht anfällig für XSS.

Keine Gefahr im abgeschotteten Browser

Auch Apps, die nur aus einer Kapselung um den Browser bestehen und in den Browsertechniken HTML und JavaScript realisiert sind, sind nicht durch XSS angeifbar, da jede App ihren eigenen, abgeschotteten Browserkontext besitzt. Die Daten sind in einer solchen App also vor XSS gut geschützt. Dagegen ermöglicht XSS im herkömmlichen Browser dem Angreifer weitreichenden Zugriff auf alle unter der Origin gespeicherten Web-Storage-Daten.

Erschwerend kommt bei allen Web-Storage-Verfahren hinzu, dass der Browser anfallende Daten unverschlüsselt auf dem Rechner des Benutzers ablegt. Erlangt ein Angreifer physisch oder per Trojaner Zugriff auf einen Rechner, kann er ohne Weiteres den Web Storage des Browsers auslesen. Bei physischem Zugriff braucht er nur die betreffende Speicherdatei zu kopieren. Auch hier ist das Schutzniveau mobiler Plattformen mit ihren Verschlüsselungs-APIs und der Abgeschlossenheit des Systems ungleich höher.

Als Schutzmaßnahme gegen unbefugten Zugriff sollte man in Web-Storage-Objekten keine vertraulichen Daten ablegen, diese gehören weiterhin auf den Server. Auch das verschlüsselte Ablegen ist keine Lösung, da ein Angreifer die clientseitig in JavaScript vorliegenden Verschlüsselungsverfahren manipulieren kann. Die aus dem Web Storage genutzten Daten können vom Benutzer selbst oder einem Angreifer verändert worden sein und gelten somit nicht als vertrauenswürdig. Die Ähnlichkeit einer HTML5-Browser-Anwendung mit einer mobilen App darf daher nicht dazu verleiten, dass der Entwickler die Speicherung lokaler Daten ebenso umsetzt.

Ein Nachteil von Anwendungen auf der Basis von Web Storage ist, dass der Browser nicht mehrbenutzerfähig ist. Weder Local Storage noch die beiden Datenbank-Konzepte bieten hier Lösungen; dies muss die Anwendung regeln. Wann immer keine persistente Speicherung der Daten vonnöten ist, sollte der Entwickler Session Storage statt Local Storage verwenden, damit die Daten nach dem Schließen des Browsers gelöscht sind.

Der Einsatz der IndexedDB unter HTML5 erweitert außerdem die Angriffsmöglichkeiten der SQL-Injection, der Manipulation der Datenbankabfrage durch von außen eingeschleuste Zeichenketten – statt nur auf dem Webserver kann ein Angreifer nun auch clientseitig Code einschleusen. Entsprechend sind die Eingabevalidierungen zusätzlich auf dem Client durchzuführen.

Fazit

Mit HTML5 halten viele neue Techniken Einzug in den Browser. Auch wenn die Erfinder sich sichtlich darum bemüht haben, Sicherheit bereits „by design“ umzusetzen und dazu teilweise elegante Lösungen gefunden haben, an einem kamen sie nicht vorbei: der Notwendigkeit, alles Bisherige unversehrt zu lassen – einschließlich der sicherheitstechnischen Unzulänglichkeiten der bestehenden Browsertechniken. Eine Erweiterung der Browserfähigkeiten, durch die bestehende Anwendungen plötzlich nicht mehr zuverlässig funktionieren würden, wäre undenkbar.

So erhöht HTML5 zwangsläufig die Komplexität, den ärgsten Gegenspieler aller Sicherheitsbemühungen. Zwar gibt es den Entwicklern etliche Möglichkeiten, die Sicherheitsbandagen enger anzulegen, verlangt ihnen aber gleichzeitig viel Wissen zur Umsetzung ab. Der Spielraum für Fehler engt sich auch von einer anderen Seite her beträchtlich ein, will man beim Einsatz von HTML5 kein sicherheitstechnisches Desaster erleben: Die serverseitige Webanwendung darf keine Sicherheitslücken enthalten, insbesondere keine XSS-Schwachstellen. (ur)

Thomas Schreiber

ist Gründer und Geschäftsführer der SecureNet GmbH und als Berater und Seminarleiter im Bereich Web Application Security tätig.

Sven Schleier

ebenfalls SecureNet, ist IT-Sicherheitsberater im Bereich Web Application Security.

Alle Links: www.ix.de/ix1301088