9 Inhalte isolieren

Die meisten Sicherheitsvorkehrungen von Webbrowsern dienen dazu, Dokumente abhängig von ihrem jeweiligen Ursprung (Origin) voneinander zu isolieren. Das Prinzip ist einfach: Zwei Seiten aus unterschiedlichen Quellen sollen sich nicht – oder je nach Situation nur sehr bedingt – gegenseitig beeinflussen können. Die Umsetzung in der Praxis ist jedoch komplizierter, da es keine allgemeine Übereinkunft darüber gibt, wo ein einzelnes Dokument anfängt und wo es aufhört und wodurch sich ein einzelner Ursprung auszeichnet. Das Ergebnis ist ein manchmal unvorhersehbares Sammelsurium von einander widersprechenden Richtlinien (Policies). Diese Richtlinien passen zwar nicht wirklich gut zusammen, lassen sich aber auch nicht optimieren, ohne alle derzeitigen legitimen Verwendungen des Web erheblich zu beeinträchtigen.

Abgesehen von diesen Problemen herrscht auch wenig Klarheit darüber, welche Aktionen überhaupt einer Sicherheitsprüfung unterzogen werden sollen. Es ist offensichtlich, dass manche Formen der Interaktion (z.B. einem Link zu folgen) ohne besondere Einschränkungen zugelassen werden sollten, da sie für das Funktionieren der gesamten Umgebung unverzichtbar sind. Andere dagegen (z.B. die Bearbeitung von Inhalten auf einer Seite, die in einem eigenen Fenster geladen wurde) erfordern einige Sicherheitsprüfungen. Zwischen diesen Polen liegt jedoch eine umfangreiche Grauzone, und es scheint so, als würden die Entscheidungen in diesem Mittelfeld eher mit dem Würfel und nicht anhand eines durchdachten Plans getroffen werden. In diesem trüben Gewässer wimmelt es von Sicherheitsproblemen, wie etwa Cross-Site Request Forgery (siehe Kapitel 4).

Doch genug der Vorrede. Lassen wir selbst die Würfel entscheiden und fangen wir an mit ... JavaScript!

9.1 SOP für das DOM

Das Konzept der *Same-Origin Policy* (SOP, »Richtlinie des gemeinsamen Ursprungs«) wurde 1995 von Netscape parallel zu JavaScript und zum DOM eingeführt, und zwar nur ein Jahr nach den HTTP-Cookies. Hinter dieser Richtlinie

steht eine ganz einfache Regel: Wenn zwei JavaScript-Ausführungskontexte vorhanden sind, darf der eine nur dann auf das DOM des anderen zugreifen, wenn die Protokolle, DNS-Namen[1] und Portnummern der Hostdokumente exakt übereinstimmen. Jeglicher andere dokumentübergreifende DOM-Zugriff in JavaScript ist unzulässig.

Hinweis

Die SOP war im Wesentlichen eine notwendige Reaktion seitens Netscape auf ein aus heutiger Sicht mehr oder weniger unbedacht eingeführtes Feature, das Sicherheitsforschern und Online-Kriminellen heute noch ein Lachen auf die Wangen zaubert: Frames. Die Möglichkeit, verschiedene Dokumente unterschiedlichen Ursprungs in einem Fenster darstellen zu können, hat neben der Notwendigkeit der SOP zu vielen weiteren Problemen geführt. Der Begriff Cross-Site-Scripting rührt unter anderem aus der Kommunikation zweier Seiten her – eine Seite skriptet eine andere. Frames wie auch SOP wurden erstmals im Netscape Navigator 2.0 eingesetzt.

Das Tupel aus Protokoll, Host und Port in diesem Algorithmus wird allgemein als *Ursprung* (Origin) bezeichnet und bildet eine ziemlich stabile Grundlage für eine Sicherheitsrichtlinie; SOP ist in allen modernen Browsern mit einem hohen Maß an Einheitlichkeit und mit nur vereinzelten Bugs implementiert[2]. Tatsächlich tanzt nur der Internet Explorer aus der Reihe und ignoriert bei der Ursprungsprüfung die Portnummer. Diese Vorgehensweise ist weniger sicher, vor allem, da die Gefahr besteht, dass Nicht-HTTP-Dienste auf einem entfernten Host für HTTP/0.9-Webserver laufen (siehe Kapitel 3). Gewöhnlich ist aber kein Unterschied festzustellen.

1. Dieser und die meisten anderen Browsersicherheitsmechanismen beruhen auf DNS-Labels, nicht auf der Untersuchung der zugrunde liegenden IP-Adressen. Das führt zu einem kuriosen Verhalten: Wenn sich die IP-Adresse eines Hosts ändert, kann ein Angreifer über den Browser des Benutzers mit dem neuen Ziel kommunizieren und dabei Schaden anrichten und gleichzeitig den wahren Ursprung des Angriffs verborgen halten (was jedoch nicht sehr interessant ist). Er kann aber auch mit dem internen Netzwerk des Opfers herumspielen, das aufgrund einer Firewall normalerweise nicht zugänglich ist (was schon ein größeres Problem darstellt). Die absichtliche Änderung einer IP-Adresse für diesen Zweck wird als *DNS-Rebinding* bezeichnet. Browser versuchen, diese Gefahr in gewissem Maße zu verringern, indem sie die Ergebnisse von DNS-Lookups eine Zeitlang zwischenspeichern (*DNS-Pinning*), allerdings ist diese Abwehrmaßnahme unvollkommen.
2. Eine bedeutsame Quelle von SOP-Bugs ist die Verwendung mehrerer getrennter URL-Analyseroutinen im Browsercode. Wenn sich das Analyseverfahren im HTTP-Stack von dem unterscheidet, das zum Erkennen von JavaScript-Ursprungsangaben verwendet wird, können Probleme auftreten. Vor allem Safari hatte mit einer erheblichen Anzahl von SOP-Bypass-Fehlern zu kämpfen, die durch fehlerhafte URLs ausgelöst werden (unter anderem durch die in Kapitel 2 beschriebenen Eingaben).

Tabelle 9–1 zeigt das Ergebnis von SOP-Prüfungen in verschiedenen Situationen.

Ursprungs-dokument	Dokument, auf das zugriffen wird	Browser außer IE	Internet Explorer
*http://example.com/**a**/*	*http://example.com/**b**/*	Zugriff gestattet	Zugriff gestattet
http://example.com/	*http://**www**.example.com/*	Host stimmt nicht überein	Host stimmt nicht überein
***http**://example.com/*	***https**://example.com/*	Protokoll stimmt nicht überein	Protokoll stimmt nicht überein
*http://example.com**:81**/*	*http://example.com/*	Port stimmt nicht überein	Zugriff gestattet

Tab. 9–1 *Ergebnisse von SOP-Prüfungen*

Hinweis

Ursprünglich sollte die SOP nur den Zugriff auf das DOM regeln, also auf die Methoden und Eigenschaften im Zusammenhang mit den Inhalten des tatsächlich angezeigten Dokuments. Die Richtlinie wurde nach und nach erweitert, um andere sensible Bereiche des DOM zu schützen, deckt aber nicht alles ab. Beispielsweise können Skripte mit verschiedenem Ursprung immer noch `location.assign()` oder `location.replace(...)` für beliebige Fenster und Frames aufrufen. Ausmaß und Auswirkungen dieser Ausnahme sind Thema von Kapitel 11.

Die Einfachheit der SOP ist Segen und Fluch zugleich. Der Mechanismus ist recht gut verständlich und auch nicht schwierig korrekt zu implementieren. Seine Inflexibilität jedoch macht Webentwicklern das Leben schwer. In manchen Kontexten ist die Richtlinie zu breit gefasst, sodass es etwa unmöglich ist, die Homepages verschiedener Benutzer zu trennen (es sei denn, man gibt ihnen jeweils eine eigene Domäne). In anderen Fällen ist das Gegenteil der Fall: Die Richtlinie macht Websites, die völlig legitim zusammenwirken (z.B. *login.example.com* und *payments.example.com*), den nahtlosen Datenaustausch schwer.

Versuche, das erste Problem zu lösen – also den Begriff des Ursprungs stärker einzuengen –, sind gewöhnlich zum Scheitern verurteilt, was an der Wechselwirkung mit anderen Sicherheitseinrichtungen im Browser liegt. Häufiger werden Versuche unternommen, den Ursprungsbegriff zu erweitern oder domänenübergreifende Interaktionen zu fördern. Zwei weithin unterstützte Möglichkeiten dazu sind `document.domain` und `postMessage(...)`, die im Folgenden besprochen werden.

9.1.1 document.domain

Mit dieser JavaScript-Eigenschaft können sich zwei kooperierende Websites, die sich eine gemeinsame Top-Level-Domäne (wie *example.com* oder auch nur *.com*) teilen, darauf einigen, dass sie in zukünftigen Ursprungsprüfungen als äquivalent angesehen werden sollen. Beispielsweise können sowohl *login.example.com* als auch *payments.example.com* die folgende Zuweisung durchführen:

```
document.domain = "example.com"
```

Wird diese Eigenschaft festgelegt, überschreibt sie die übliche Logik des Hostnamenvergleichs für SOP-Prüfungen. Protokoll und Portnummer müssen jedoch immer noch übereinstimmen. Wenn das nicht der Fall ist, hat die Verwendung von `document.domain` nicht die gewünschte Wirkung.

Beide Parteien müssen sich ausdrücklich für diese Möglichkeit aussprechen. Nur weil die Website *login.example.com* den Wert von `document.domain` auf *example.com* gesetzt hat, hat sie noch lange keinen Zugriff auf Inhalte der Website unter *http://example.com/*. Diese Website muss ebenfalls die Zuweisung durchführen, auch wenn das nach allgemeinem Verständnis wie eine Nulloperation erscheint. Dieser Effekt funktioniert nur symmetrisch. Eine Seite, die `document.domain` festgelegt hat, ist nicht in der Lage, auf Seiten zuzugreifen, die das nicht getan haben. Dadurch gerät die Seite auch größtenteils (aber nicht ganz![3]) außerhalb der Reichweite normaler Dokumente, die zuvor als Seiten mit demselben Ursprung gegolten haben. Tabelle 9–2 zeigt die Auswirkungen verschiedener Einstellungen von `document.domain`.

Ursprungsdokument		Dokument, auf das zugegriffen wird		Ergebnis
URL	**document.domain**	**URL**	**document.domain**	
http://www.example.com/	*example.com*	*http://payments.example.com/*	*example.com*	Zugriff gestattet
http://www.example.com/	*example.com*	*https://payments.example.com/*	*example.com*	Protokoll stimmt nicht überein
http://payments.example.com/	*example.com*	*http://example.com/*	nicht festgelegt	Zugriff verweigert

Tab. 9–2 *Ergebnis der Prüfung von* `document.domain`

3. Im Internet Explorer ist es beispielsweise möglich, dass eine Seite zu jeglichen anderen Dokumenten wechselt, die nominell denselben Ursprung haben, aber »isoliert« wurden, nachdem `document.domain` auf `javascript:`-URLs gesetzt wurde. Dadurch kann jeglicher JavaScript-Code im Kontext solcher pseudoisolierter Domänen ausgeführt werden. Dazu kommt noch, dass die ursprüngliche Seite offensichtlich durch nichts daran gehindert wird, ihre eigene `document.domain`-Eigenschaft auf einen Wert zu setzen, der mit dem Ziel identisch ist, um diese Grenze aufzuheben. Mit anderen Worten: Für jeglichen Verwendungszweck, der auch nur im Entferntesten wichtig oder sicherheitsrelevant ist, dürfen Sie sich nicht auf die Möglichkeit verlassen, ein Dokument mithilfe von `document.domain` so zu isolieren, als hätte es einen anderen Ursprung.

Ursprungsdokument		Dokument, auf das zugegriffen wird		Ergebnis
URL	document.domain	URL	document.domain	
http://www.example.com/	nicht festgelegt	*http://www.example.com/*	*example.com*	Zugriff verweigert

Tab. 9–2 *Ergebnis der Prüfung von* `document.domain` *(Forts.)*

Trotz einer gewissen Kompliziertheit, die auf ein intelligentes Verhalten schließen lässt, ist die Eigenschaft `document.domain` nicht besonders sicher. Die wichtigste Schwachstelle besteht darin, dass sie unwillkommene Gäste einlädt. Nachdem sowohl *login.example.com* als auch *payments.example.com* diese Eigenschaft gegenseitig auf *example.com* gesetzt haben, können nicht nur diese beiden Seiten miteinander kommunizieren – auch *funny-cat-videos.example.com* kann sich als Trittbrettfahrer hinzugesellen. Das Ausmaß des zulässigen Zugriffs von einer Seite zur anderen macht es unmöglich, eine vernünftige Garantie für die Integrität irgendeines der beteiligten JavaScript-Kontexte abzugeben. Mit anderen Worten, wenn Sie `document.domain` verwenden, machen Sie die Sicherheit Ihrer Seite vom schwächsten Glied der gesamten Domäne abhängig. Der Extremfall, den Wert auf **.com* zu setzen, stellt praktisch einen Online-Selbstmordversuch dar.

Hinweis

Die hier beschriebene sogenannte *Domain-Relaxation* kann in vielen Fällen zu unvorhergesehenen Problemen führen. Safari und andere WebKit-Browser erlauben beispielsweise, `document.domain` bis hinunter zu *.co.kr* oder gar *.kr* zu »relaxen«. Je nach Webseite kann dies natürlich zu XSS-Problemen führen – ohne dass auf der betroffenen Seite eine Injection-Lücke vorhanden ist. Firefox und andere Browser nutzen hingegen eine Blacklist – erlauben also nur bestimme Kombinationen als Zuweisung für den Wert von `document.domain`.

9.1.2 postMessage(...)

Die API `postMessage(...)` ist eine HTML5-Erweiterung, die eine weniger komfortable, aber bedeutend sicherere Kommunikation zwischen Websites unterschiedlichen Ursprungs erlaubt, ohne dabei automatisch die Integrität der Beteiligten aufzugeben. Zurzeit wird sie in allen aktuellen Browsern unterstützt, aufgrund ihrer Neuheit jedoch nicht im Internet Explorer 6 und 7.

Dieser Mechanismus gestattet es, eine Textnachricht beliebiger Länge an ein beliebiges Fenster zu senden, für das der Absender über einen gültigen JavaScript-Handle verfügt (siehe Kapitel 6). Die SOP weist zwar einige Lücken auf, die ähnliche Möglichkeiten auch durch andere Maßnahmen zulassen[4], doch diese ist, im Gegensatz zu den meist etwas kruden Hacks, tatsächlich sicher genug für den all-

täglichen Gebrauch. Damit kann der Absender angeben, welche Ursprünge die Nachricht überhaupt empfangen dürfen (falls sich die URL des Zielfensters geändert hat); außerdem erhält der Empfänger Informationen über die Identität des Senders, sodass die Integrität des Kanals auf einfache Weise sichergestellt werden kann. Veraltete Methoden, die sich auf Schlupflöcher in der SOP stützen, bieten solche Informationen zur Rückversicherung korrekter Ursprünge dagegen nicht an. Wenn eine bestimmte Aktion auch ohne umfangreiche Sicherheitsprüfungen erlaubt ist, kann sie gewöhnlich von böswilligen Dritten ausgelöst werden und nicht nur von den vorgesehenen Teilnehmern.

Um die korrekte Verwendung von `postMessage(...)` zu veranschaulichen, betrachten wir den Fall, dass ein Dokument der obersten Ebene in *payments.example.com* Anmeldeinformationen des Benutzers benötigt, um etwas in einem anderen Frame anzuzeigen. Um das zu erreichen, lädt es einen Frame, der auf *login.example.com* zeigt. Dieser Frame kann dann einfach den folgenden Befehl ausgeben:

```
parent.postMessage("user=bob", "https://payments.example.com");
```

Der Browser stellt die Nachricht nur dann zu, wenn die einbettende Website tatsächlich mit dem angegebenen, vertrauenswürdigen Ursprung übereinstimmt. Um die Antwort sicher verarbeiten zu können, muss das Dokument der obersten Ebene den folgenden Code verwenden:

```
// Registriert die Absicht, eingehende Nachrichten zu verarbeiten:
addEventListener("message", user_info, false);

// Verarbeitet die tatsaechlichen Daten bei ihrem Eintreffen:
function user_info(msg) {
   if (msg.origin == "https://login.example.com") {
      // Verwendet msg.data wie geplant
   }
}
```

`postMessage(...)` ist ein sehr zuverlässiger Mechanismus, der erhebliche Vorteile gegenüber `document.domain` und praktisch allen anderen früheren Guerilla-Verfahren aufweist. Daher sollte er so oft wie möglich verwendet werden. Allerdings ist auch hierbei ein Missbrauch möglich. Betrachten Sie die folgende Prüfung, die nach einem Teilstring im Domänennamen sucht:

```
if (msg.origin.indexOf(".example.com") != -1) { ... }
```

4. Mehr darüber erfahren Sie in Kapitel 11. Das bemerkenswerteste Beispiel sei hier jedoch trotzdem erwähnt, nämlich die Codierung von Daten in URL-Fragment-IDs. Das ist möglich, da die Navigation zu einer neuen URL in Frames in den meisten Fällen keinen Sicherheitseinschränkungen unterliegt und die Seite durch den Wechsel zu einer URL, bei der sich nur die Fragment-ID geändert hat, nicht neu geladen wird. JavaScript in Frames kann einfach `location.hash` abrufen und auf diese Weise eingehende Nachrichten erkennen.

Dieser Vergleich erkennt nicht nur Websites innerhalb von *example.com* als Übereinstimmung an, sondern akzeptiert auch fröhlich Nachrichten von *www.example.com.bunnyoutlet.com*. Höchstwahrscheinlich werden Sie mehr als einmal über Code wie diesen stolpern.

Hinweis

Änderungen an HTML5 haben die API `postMessage(...)` um leicht überkonstruierte »Ports« und »Kanäle« erweitert, die zur Förderung der Streamkommunikation zwischen Websites dienen. Die Browserunterstützung dafür ist zurzeit sehr beschränkt und die praktische Bedeutung ist unklar. Besondere Sicherheitsbedenken scheint es jedoch nicht zu geben.

9.1.3 Wechselwirkung mit sensiblen Browserdaten

Zum Abschluss unseres Überblicks der DOM-gestützten SOP müssen wir noch betonen, dass sie in keiner Weise abgestimmt ist mit Ambient-Credentials, SSL-Status, Netzwerkkontext und vielen anderen möglicherweise sicherheitsrelevanten Parametern, die der Browser nachverfolgt. Zwei Fenster oder Frames in einem Browser behalten den gemeinsamen Ursprung, selbst wenn sich der Benutzer von einem Konto abmeldet und mit einem anderen wieder anmeldet, wenn die Seite statt eines guten plötzlich ein schlechtes HTTPS-Zertifikat verwendet usw.

Diese mangelnde Abstimmung kann dazu führen, dass andere Sicherheitslücken ausgenutzt werden. Beispielsweise schützen verschiedene Websites ihre Anmeldeformulare nicht gegen Cross-Site Request Forgery, sodass jede Drittwebsite einfach einen Benutzernamen und ein Passwort übermitteln kann, um den Benutzer an einem vom Angreifer gesteuerten Konto anzumelden. Das mag auf den ersten Blick harmlos erscheinen. Aber wenn für den Inhalt, der vor und nach diesem Vorgang im Browser geladen ist, derselbe Ursprung angenommen wird, dann sind die Auswirkungen solcher »auf sich selbst zielender« XSS-Angriffe (bei denen der Besitzer eines Kontos also nur sich selbst als Ziel nehmen kann und die normalerweise ignoriert werden) plötzlich viel weitreichender. Im einfachsten Fall öffnet der Angreifer einen Frame, der auf eine sensible Seite der Zielwebsite zeigt (z.B. *http://www.fuzzybunnies.com/address_book.php*), und meldet das Opfer dann an einem von ihm gesteuerten Konto an, um Self-XSS in einem anderen Teil von *fuzzybunnies.com* durchzuführen. Trotz der geänderten HTTP-Anmeldeinformationen hat der im letzten Schritt injizierte Code uneingeschränkten Zugriff auf den zuvor geladenen Frame und ermöglicht damit Datendiebstahl.

9.2 SOP für XMLHttpRequest

Die `XMLHttpRequest`-API wurde in diesem Buch schon an verschiedenen Stellen erwähnt. Sie gibt JavaScript-Programmen die Möglichkeit, nahezu unbeschränkt HTTP-Requests an den Server zu stellen, von dem das Hostdokument stammt, sowie Header und Rumpf der Antwort zu lesen. Das wäre hier nicht so bedeutsam, wenn dieser Mechanismus nicht den vorhandenen HTTP-Stack des Browsers und seine Annehmlichkeiten wie Ambient-Authority-Anmeldeinformationen, Caching-Mechanismen, Keepalive-Sessions usw. nutzen würde.

Das folgende Beispiel zeigt eine einfache und selbsterklärende synchrone Verwendung von `XMLHttpRequest`:

```
var x = new XMLHttpRequest();
x.open("POST", "/some_script.cgi", false);
x.setRequestHeader("X-Random-Header", "Hi mom!");
x.send("...POST-Daten hier ...");
alert(x.responseText);
```

Asynchrone Requests sind sehr ähnlich, werden aber ohne Blockierung der JavaScript-Engine bzw. des Browsers ausgeführt. Stattdessen erfolgen sie im Hintergrund, wobei nach ihrem Abschluss ein Eventhandler aufgerufen wird.

Wie ursprünglich vorgesehen, wird die Möglichkeit, HTTP-Requests über diese API auszugeben und die zurückkommenden Daten zu lesen, von einer fast wörtlichen Kopie der SOP geregelt – mit zwei kleinen und anscheinend willkürlichen Änderungen. Erstens hat die Einstellung von `document.domain` keine Auswirkungen auf diesen Mechanismus, d.h., die für `XMLHttpRequest(...)` angegebene Zieladresse muss immer mit dem wahren Ursprung des Dokuments übereinstimmen. Zweitens wird die Portnummer in diesem Zusammenhang auch vom Internet Explorer vor Version 9 berücksichtigt, auch wenn diese Browser sie ansonsten ignorieren.

Dass `XMLHttpRequest` dem Benutzer eine beispiellose Kontrolle über die HTTP-Header im Request gibt, ist für die Sicherheit sogar von Vorteil. Beispielsweise lässt sich durch Einfügen eines eigenen HTTP-Headers wie `X-Coming-From:same-origin` auf einfache Weise betätigen, dass ein Request nicht aus einer Drittdomäne stammt, da keine andere Website in der Lage sein sollte, einen selbst definierten Header in eine vom Browser ausgegebene Anfrage einzufügen. Diese Garantie ist jedoch nicht sehr verlässlich, da es keine Spezifikation gibt, die besagt, dass sich die implizierten Einschränkungen für domänenübergreifende Header nicht ändern können[5]. Auf dem Gebiet der Websicherheit müssen Sie jedoch immer mit solchen Annahmen leben.

Es kann jedoch auch eine Last sein, die Kontrolle über die Struktur eines HTTP-Requests zu haben, denn die Ergänzung um bestimmte Arten von Headern kann die Bedeutung des Requests für die Zielserver oder die Proxys ändern, ohne dass sich der Browser dessen bewusst ist. Beispielsweise kann die Angabe

eines falschen Wertes für `Content-Length` einem Angreifer ermöglichen, einen zweiten Request in eine vom Browser unterhaltene Keep-alive-Session einzuschmuggeln, wie der folgende Code zeigt:

```
var x = new XMLHttpRequest();
x.open("POST", "http://www.example.com/", false);

// Überschreibt den vom Browser berechneten Content-Length-Header:

x.setRequestHeader("Content-Length", "7");
// Der Server nimmt an, dass die Nutzlast hinter den ersten sieben Zeichen
// endet und dass der Rest ein eigenständiger HTTP-Request ist.
x.send(
   "Erwischt!\n" +
   "GET /evil_response.html HTTP/1.1\n" +
   "Host: www.bunnyoutlet.com\n\n"
);
```

Wenn das geschieht, kann die Antwort auf die zweite, eingeschleuste Anfrage vom Browser später fehlinterpretiert werden, was möglicherweise zur Verunreinigung des Cache oder zur Einschleusung von Inhalten in eine andere Website führt. Dieses Problem wird noch verstärkt, wenn ein HTTP-Proxy verwendet wird und alle HTTP-Requests durch einen gemeinsamen Kanal übertragen werden.

Aufgrund dieser Gefahren, und nach häufigem Versuch und Irrtum, haben moderne Browser eine Reihe von HTTP-Headern und Request-Methoden auf Blacklists gesetzt. Die Vorgehensweise ist jedoch nicht einheitlich: Während `Referer`, `Content-Length` und `Host` allgemein untersagt sind, werden Header wie `User-Agent`, `Cookie`, `Origin` und `If-Modified-Since` in den einzelnen Browsern unterschiedlich gehandhabt. Ebenso wird die Methode `TRACE` überall blockiert, da sie unvorhersehbare Risiken für `httpOnly`-Cookies bietet. Demgegenüber ist `CONNECT` trotz des geringen Risikos der Manipulation von HTTP-Proxys in Firefox erlaubt.

Die Implementierung dieser Blacklists ist natürlich selbst eine unterhaltsame Übung. Sehen Sie sich – nur zu Ihrem Vergnügen – einmal die folgenden Fälle an, die in einigen Browsern noch vor vier Jahren funktionierten [1,2]:

```
XMLHttpRequest.setRequestHeader("X-Harmless", "1\nOwned: Gotcha");
```

5. Tatsächlich hatten früher viele Plug-ins Probleme auf diesem Gebiet. Vor allem Adobe Flash ließ bis 2008 willkürlich domänenübergreifende HTTP-Header zu. Danach wurde das Sicherheitsmodell generalüberholt. Bis 2011 litt dieses Plug-in außerdem unter einem langlebigen Implementierungsbug, durch den benutzerdefinierte Header an einen unbeteiligten Server gesendet wurden, nachdem es den HTTP-Umleitungscode 307 von einem Angreifer empfangen hatte. Beide Probleme sind jetzt behoben, aber in der Zeit zwischen der Entdeckung dieses Bugs bis zu seiner Korrektur traten Schwierigkeiten auf.

Oder:

```
XMLHttpRequest.setRequestHeader("Content-Length: 123 ", "");
```

Oder einfach:

```
XMLHttpRequest.open("GET\thttp://evil.com\tHTTP/1.0\n\n", "/", false);
```

Hinweis

CORS (Cross-Origin Resource Sharing) [3] ist ein Vorschlag für eine Erweiterung von `XMLHttpRequest`, mit der HTTP-Requests über Domänen hinweg gesendet und die zurückgegebenen Daten gelesen werden können, falls sie einen bestimmten Antwortheader enthalten. Dieser Mechanismus ändert die Semantik der in diesem Abschnitt beschriebenen API. Hiermit wird es möglich, über `XMLHttpRequest.open(...)` ohne weitere Überprüfungen bestimmte domänenübergreifende »08/15-Requests« zu senden, die sich nicht von der regulären Navigation unterscheiden sollen. Anspruchsvollere Anfragen erfordern zunächst einen »Preflight-Request« auf der Grundlage von `OPTIONS`. CORS ist in einigen Browsern schon verfügbar, stößt aber auf den Widerstand der Ingenieure von Microsoft, die im Internet Explorer 8 und 9 das Konkurrenzverfahren `XDomainRequest` verfolgen. Da der Ausgang dieses Konflikts noch offen ist, muss die ausführliche Erörterung von CORS bis Kapitel 16 warten; dort gibt es einen systematischen Überblick über kommende und experimentelle Mechanismen.

9.3 SOP für Web Storage

Web Storage ist eine einfache Datenbanklösung, die zuerst von Mozilla in Firefox 1.5 implementiert und schließlich in die Spezifikation von HTML5 aufgenommen wurde [4]. Sie ist in allen aktuellen Browsern verfügbar, nicht aber im Internet Explorer 6 und 7.

Nach mehreren dubiosen Zwischenstadien stützt sich das aktuelle Design auf die beiden einfachen JavaScript-Objekte `localStorage` und `sessionStorage`. Beide bieten eine identische, einfache API, um Name-Wert-Paare in einer vom Browser verwalteten Datenbank zu erstellen, abzurufen und zu löschen. Betrachten Sie dazu das folgende Beispiel:

```
localStorage.setItem("message", "Hi mom!");
alert(localStorage.getItem("message"));
localstorage.removeItem("message");
```

Das Objekt `localStorage` implementiert einen dauerhaften, ursprungsspezifischen Speicher, der über das Herunterfahren des Browsers hinaus Bestand hat. Dagegen ist `sessionStorage` an das aktuelle Browserfenster gebunden und bietet einen zeitweiligen Caching-Mechanismus, der am Ende der Browsersitzung zerstört wird. Die Spezifikation verlangt, dass sowohl `localStorage` als auch `sessionStorage` an einen SOP-ähnlichen Ursprung (ein Tupel aus Protokoll, Host und Port) gebun-

den werden sollen. Allerdings folgen die Implementierungen in manchen Browsern dieser Empfehlung nicht unbedingt in allen Punkten und öffnen damit mögliche Sicherheitslücken. Das gilt vor allem für den Internet Explorer 8, in dem bei der Bestimmung des Ursprungs das Protokoll nicht berücksichtigt wird, sodass HTTP- und HTTPS-Seiten in einen gemeinsamen Kontext gepackt werden. Dadurch ist es riskant, auf HTTP-Sites sensible Daten mithilfe dieser API zu speichern oder zu lesen. (Das Problem wurde im Internet Explorer 9 behoben, es scheint aber nicht vorgesehen zu sein, die Korrektur auch auf die ältere Version anzuwenden.)

In Firefox verhält sich `localStorage` korrekt, die Schnittstelle `sessionStorage` dagegen nicht. HTTP und HTTPS verwenden einen gemeinsamen Speicherkontext. Es erfolgen zwar Überprüfungen, damit HTTP-Inhalte keine Schlüssel lesen können, die von HTTPS-Skripten erstellt wurden, doch tut sich dabei eine ernst zu nehmende Lücke auf: Jeder Schlüssel, der erst über HTTP erstellt und dann über HTTPS aktualisiert wurde, bleibt für nicht verschlüsselte Seiten sichtbar. Dieser Bug, der zum ersten Mal 2009 gemeldet wurde [5], wird irgendwann korrigiert werden, aber der Zeitpunkt ist noch offen.

9.4 Sicherheitsrichtlinie für Cookies

Die Semantik von HTTP-Cookies haben wir bereits in Kapitel 3 besprochen. Wir haben dabei aber ein wichtiges Detail ausgespart, nämlich die Sicherheitsregeln, die umgesetzt werden müssen, um zu verhindern, dass unbeteiligte Seiten mit den Cookies herumspielen, die einer anderen Site gehören. Dies ist vor allem deshalb interessant, weil der dafür verwendete Ansatz der SOP vorausging und auf unerwartete Weise mit ihr in Wechselwirkung tritt.

Als Gültigkeitsbereich von Cookies ist eine Domäne vorgesehen, weshalb sie sich nicht einfach auf einen einzelnen Hostnamen beschränken lassen. Zwar kann der Parameter `domain` eines Cookies auf den aktuellen Hostnamen eingestellt werden (z.B. *foo.example.com*), doch verhindert das nicht, dass das Cookie an eventuell vorhandene Unterdomänen gesendet wird, etwa *bar.foo.example.com*. Dagegen kann ein qualifiziertes Fragment vom rechten Ende des Hostnamens wie *example.com* angegeben werden, um den Gültigkeitsbereich zu erweitern.

Ironischerweise lassen die ursprünglichen RFCs erkennen, dass die Ingenieure von Netscape als Gültigkeitsbereich von Cookies den Host vorgesehen hatten, dann aber ihrer eigenen Empfehlung nicht folgten. Die für diesen Zweck entworfene Syntax wurde von den Nachfolgern des Netscape Navigator nicht erkannt (und auch nicht von anderen Implementierungen). In einigen Browsern ist eine gewisse Beschränkung des Gültigkeitsbereichs von Cookies auf den Host möglich, indem der Parameter `domain` weggelassen wird, allerdings hat diese Methode im Internet Explorer keine Auswirkung.

Tabelle 9–3 zeigt die resultierenden Gültigkeitsbereiche von Cookies in einigen charakteristischen Fällen.

Parameter `domain` für ein auf *foo.example.com* festgelegtes Cookie	Gültigkeitsbereich des resultierenden Cookies	
	Browser außer IE	Internet Explorer
Wert nicht angegeben	*foo.example.com* (exakt)	**.foo.example.com*
bar.foo.example.com	Cookie wird nicht gesetzt: Die Domäne ist spezifischer als der Ursprung.	
foo.example.com	**.foo.example.com*	
baz.example.com	Cookie wird nicht gesetzt: Die Domäne stimmt nicht überein.	
example.com	**.example.com*	
ample.com	Cookie wird nicht gesetzt: Die Domäne stimmt nicht überein.	
.com	Cookie wird nicht gesetzt: Die Domäne ist zu weit gefasst, was ein Sicherheitsrisiko darstellt.	

Tab. 9–3 *Gültigkeitsbereiche für Cookies*

Der einzige andere Parameter, der tatsächlich verwendet werden kann, um den Gültigkeitsbereich von Cookies einzuschränken, ist das Pfadpräfix. Jedes Cookie kann mit einem `path`-Wert versehen werden. Dadurch wird der Browser angewiesen, das Cookie nur bei Requests an übereinstimmende Verzeichnisse zurückzusenden. Ein Cookie mit einem Gültigkeitsbereich, der mit `domain` auf *example.com* und mit `path` auf */some/path* festgelegt ist, wird beispielsweise in Requests wie dem folgenden eingebunden:

```
http://foo.example.com/some/path/subdirectory/hello_world.txt
```

Dieser Mechanismus kann jedoch irreführend sein. Bei SOP-Prüfungen werden URL-Pfade nicht berücksichtigt, weshalb sie keine sinnvolle Sicherheitsgrenze bilden. Unabhängig von der Funktionsweise von Cookies kann JavaScript-Code willkürlich zwischen beliebigen URLs auf einem einzelnen Host wechseln und dabei schädliche Nutzlasten einschleusen, um die Funktionen zu missbrauchen, die durch pfadgebundene Cookies geschützt werden. (Mehrere Whitepapers und Bücher zum Thema Sicherheit empfehlen bis heute, die Festlegung von Gültigkeitsbereichen durch Pfade als Sicherheitsmaßnahme zu verwenden. Ist den meisten Fällen liegt man mit diesem Ratschlag jedoch total daneben.)

Abgesehen von den echten Features für Gültigkeitsbereiche (die zusammen mit dem Namen des Cookies ein Tupel bilden, das jedes Cookie eindeutig bezeichnet) können Webserver auch Cookies mit den zwei besonderen, voneinander unabhängigen Flags `httpOnly` und `secure` ausgeben. Das erste, also `httpOnly`, verhindert den Zugriff auf das Cookie über die API `document.cookie`. Damit soll es schwieriger gemacht werden, die Anmeldeinformationen eines Benutzers nach dem Einschleusen eines schädlichen Skripts auf der Seite einfach zu kopieren. Das

zweite, `secure`, sorgt dafür, dass das Cookie nicht mehr bei Requests über unverschlüsselte Protokolle übertragen wird. Damit ist es möglich, HTTPS-Dienste zu erstellen, die aktiven Angriffen widerstehen können.[6]

Der Nachteil dieser Mechanismen besteht darin, dass sie Daten nur gegen Lesen und nicht gegen Überschreiben schützen. So ist es für JavaScript-Code, der über HTTP übertragen wird, immer noch möglich, den Cookie-Jar für eine Domäne einfach zu überschwemmen und ein neues Cookie ohne `secure`-Flag festzulegen.[7] Da der vom Browser gesendete Header `Cookie` keine Metadaten über den Ursprung oder den Gültigkeitsbereich eines Cookies angibt, lässt sich dieser Trick nur schwer erkennen. Dieses Verhaltens hat ernste Konsequenzen: Die übliche »zustandslose« Vorgehensweise zur Verhinderung von XSRF-Angriffen, bei der ein geheimes Token gleichzeitig in einem clientseitigen Cookie und in einem verborgenen Formularfeld gespeichert wird und die beiden Vorkommen anschließend verglichen werden, ist für HTTPS-Websites nicht besonders sicher. Können Sie herausfinden, warum das so ist?

Hinweis

Apropos destruktive Konflikte: Bis 2010 kamen `httpOnly`-Cookies auch `XMLHttpRequest` ins Gehege. Die Autoren dieser API hatten einfach nicht darüber nachgedacht, ob die Funktion `XMLHttpRequest.getResponseHeader(...)` in der Lage sein soll, `Set-Cookie`-Werte mit dem Flag `httpOnly` zu inspizieren. Was dabei herauskam, war vorhersehbar.

9.4.1 Der Einfluss von Cookies auf die SOP

Die SOP hat einige unschöne Auswirkungen auf die an sich sehr granular konfigurierbare Sicherheit von Cookies (vor allem auf den Mechanismus zur Festlegung des Gültigkeitsbereichs über den Pfad). Der umgekehrte Fall – Cookies stehlen, ohne die SOP zu verletzen – ist allerdings häufiger anzutreffen und als wesentlich problematischer einzuschätzen. Die Schwierigkeit liegt darin, dass HTTP-Cookies häufig als Anmeldeinformationen fungieren. Wer Zugriff auf die Cookies erhält, hat also ähnliche Macht, wie er sie durch einen SOP-Bypass erlangen würde. Mit

6. Es spielt keine Rolle, dass *https://webmail.expample.com/* nur über HTTPS angeboten wird. Wenn bei diesem Request ein Cookie verwendet wird, das nicht an verschlüsselte Protokolle gebunden ist, kann ein Angreifer einfach warten, bis das Opfer zu *http://www.fuzzybunnies.com/* wechselt. Dann kann er heimlich einen Frame zu *http://webmail.example.com/* auf der Seite injizieren und den resultierenden TCP-Handshake abfangen. Der Browser sendet dann alle Cookies für *webmail.example.com* über einen unverschlüsselten Kanal. Und damit hat der Angreifer auch schon gewonnen.
7. Es ist zwar möglich, dies zu verhindern, indem man die Container für `httpOnly`- und normale Cookies trennt, doch in jedem Fall können mehrere identisch benannte Cookies mit unterschiedlichem Gültigkeitsbereich nebeneinander existieren, die dann aber bei jedem übereinstimmenden Request zusammen gesendet werden. Sie werden dabei nicht von irgendwelchen sinnvollen Metadaten begleitet, und ihre Reihenfolge ist nicht definiert, sondern browserspezifisch.

dem richtigen Satz von Cookies versehen, kann ein Angreifer ganz einfach seinen eigenen Browser verwenden, um im Namen des Opfers mit der Zielwebsite zu kommunizieren. Die SOP wird umgangen und alles ist möglich.

Aufgrund dieses Verhaltens können jegliche Diskrepanzen zwischen diesen beiden Sicherheitsmechanismen zu Schwierigkeiten für den strengeren Mechanismus führen. Beispielsweise ist es aufgrund der eher wahllosen Regeln für den Gültigkeitsbereich von HTTP-Cookies nicht möglich, den sensiblen Inhalt auf *webmail.example.com* vollständig von dem weniger vertrauenswürdigen HTML-Inhalt auf *blog.example.com* zu trennen. Selbst wenn die Besitzer der Webmailanwendung den Gültigkeitsbereich ihrer Cookies eng fassen (was gewöhnlich jedoch den Anmeldevorgang erschwert), kann jeder Angreifer, der einen Ansatzpunkt zur Script-Injection auf der Blogwebsite findet, einfach den domänenweiten Cookie-Speicher überschwemmen, die aktuellen Anmeldeinformationen verwerfen und seine eigenen Cookies mit dem Bereich **.example.com* festlegen. Diese injizierten Cookies werden bei allen folgenden Requests an *webmail.example.com* gesendet und sind von den echten größtenteils nicht zu unterscheiden.

Dieser Trick mag harmlos aussehen. Aber denken Sie daran, dass das Opfer durch einen solchen Vorgang an einem falschen Konto angemeldet werden kann. Und damit können bestimmte Aktionen (wie das Verschicken einer E-Mail) unbemerkt in diesem Konto aufgezeichnet und an den Angreifer weitergegeben werden, bevor der Fehler erkannt wird. Wenn Ihnen Webmail als Beispiel zu weit hergeholt erscheint, stellen Sie sich das Gleiche bei Amazon oder Netflix vor: Bevor Sie irgendeine Seite erreichen, die Ihnen ungewöhnlich erscheint, kann der Angreifer Ihre Suche nach Produkten mitverfolgen. (Darüber hinaus sind viele Websites einfach nicht für den Umgang mit schädlichen Nutzlasten in injizierten Cookies gerüstet und daher anfällig für XSS und ähnliche Angriffe.)

Die Eigenheiten von HTTP-Cookies erschweren es auch, verschlüsselten Datenverkehr gegen Angriffe auf Netzwerkebene zu schützen. Ein `secure`-Cookie, das von *https://webmail.example.com/* festgelegt wurde, kann immer noch aus dem Weg geräumt und durch ein fingiertes von einer gefälschten Seite unter *http://webmail.example.com/* ersetzt werden, selbst wenn es auf dem Zielhost keinen Webdienst gibt, der auf Port 80 lauscht.

9.4.2 Probleme mit Domäneneinschränkungen

Die dumme Idee, Cookies mit domänenweitem Gültigkeitsbereich zuzulassen, stellt auch Browserhersteller vor Probleme und ist eine Ursache für dauernden Ärger. Die wichtigste Frage lautet: Wie lässt es sich zuverlässig verhindern, dass *example.com* ein Cookie für **.com* festlegt und dieses Cookie unerwartet an jedes andere Ziel im Internet gesendet wird?

Auf den ersten Blick bieten sich mehrere einfache Lösungen an, die sich aber als untauglich erweisen, wenn man auch Länderdomänen berücksichtigen muss:

So muss beispielsweise auch *example.com.pl* daran gehindert werden, ein Cookie mit dem Gültigkeitsbereich **.com.pl* festzulegen. Angesichts dessen wurde in der ursprünglichen Cookie-Spezifikation von Netscape folgende Empfehlung ausgesprochen:

> *Nur Hosts innerhalb der angegebenen Domäne können ein Cookie für eine Domäne festlegen, und Domänen müssen mindestens zwei (2) oder drei (3) Punkte enthalten, um Domänen der Form ›.com‹, ›.edu‹ und ›va.us‹ zu verhindern.*
>
> *Jede Domäne, die in eine der sieben unten aufgeführten besonderen Top-Level-Domänen fällt, benötigt nur zwei Punkte. Alle anderen Domänen brauchen mindestens drei. Die sieben besonderen Top-Level-Domänen sind ›COM‹, ›EDU‹, ›NET‹, ›ORG‹, ›GOV‹, ›MIL‹ und ›INT‹.*

Die Drei-Punkte-Regel ist jedoch nur für landesweite Registrierungsstellen sinnvoll, die die Top-Level-Hierarchie widerspiegeln (*example.co.uk*) und nicht für den genauso häufigen Fall von Ländern, in denen eine direkte Registrierung erlaubt ist (*example.de*). Es gibt sogar Orte, an denen beide Vorgehensweisen zulässig sind. Beispielsweise sind sowohl *example.jp* als auch *example.co.jp* gültig.

Aufgrund ihrer praxisfernen Natur wird diese Empfehlung von den meisten Browsern ignoriert, die stattdessen ein Sammelsurium bedingter Ausdrücke verwenden, was jedoch noch mehr Probleme bereitet. (In einem Fall war es mehr als ein Jahrzehnt lang möglich, tatsächlich Cookies für **.com.pl* festzulegen.) Umfassende Korrekturen der Handhabung von Top-Level-Domänen sind in den letzten vier Jahren in allen modernen Browsern vorgenommen worden, allerdings wurden sie bis heute nicht auf die älteren Versionen Internet Explorer 6 und 7 übertragen, was wohl auch niemals geschehen wird.

Hinweis

Um alles noch schlimmer zu machen, hat die IANA (Internet Assigned Numbers Authority) in den letzten Jahren eine gehörige Menge weiterer Top-Level-Domänen hinzugefügt (z.B. *.int* und *.biz*) und berät gerade über einen Vorschlag, willkürliche, generische Registrierungen von Top-Level-Domänen zuzulassen. Wenn es dazu kommen sollte, muss der Cookie-Mechanismus wahrscheinlich von Grund auf neu erstellt werden.

9.4.3 Die ungewöhnliche Gefahr von »localhost«

Eine unmittelbar einleuchtende Konsequenz von domänenweiten Gültigkeitsbereichen für Cookies besteht darin, dass es ziemlich unsicher ist, Hostnamen innerhalb einer sensiblen Domäne an irgendeine nicht vertrauenswürdige (oder auch nur gefährdete) Partei zu delegieren. Das kann die Vertraulichkeit und Integrität aller in Cookies gespeicherten Anmeldeinformationen beeinträchtigen und

damit auch die aller anderen Informationen, die in der Zielanwendung verarbeitet werden.

So weit ist alles klar, doch 2008 entdeckte Tavis Ormandy ein weit weniger eingängiges und doch bedrohlicheres Risiko [6]: Da sich HTTP-Cookies nicht um den Port kümmern, besteht eine zusätzliche Gefahr in dem häufig anzutreffenden und bequemen Vorgehen von Administratoren, einer Domäne einen `localhost`-Eintrag hinzuzufügen, der auf 127.0.0.1[8] zeigt. Als Ormandy seine Ergebnisse veröffentlichte, belegte er die weite Verbreitung dieser Praxis – wobei diese Behauptung ohnehin nicht angezweifelt wurde – mit der folgenden Ausgabe eines Resolver-Programms:

```
localhost.microsoft.com has address 127.0.0.1
localhost.ebay.com has address 127.0.0.1
localhost.yahoo.com has address 127.0.0.1
localhost.fbi.gov has address 127.0.0.1
localhost.citibank.com has address 127.0.0.1
localhost.cisco.com has address 127.0.0.1
```

Aber warum stellt dies eine Sicherheitslücke dar? Ganz einfach: Dadurch werden die HTTP-Dienste auf dem Computer des Benutzers in dieselbe Domäne gestellt wie der Rest der Website, und, was noch wichtiger ist, auch alle Dienste, die nur wie HTTP *aussehen*. Diese Dienste aber sind gewöhnlich nicht dem Internet ausgesetzt, sodass keine Notwendigkeit dafür gesehen wird, sie sorgfältig zu gestalten oder auf dem neuesten Stand zu halten. Ormandys Fallbeispiel war ein von CUPS (Common UNIX Printing System) bereitgestelltes Druckerverwaltungssystem, das einen von einem Angreifer gelieferten JavaScript-Code im Kontext von *example.com* ausführte, wenn es wie folgt aufgerufen wurde:

```
http://localhost.example.com:631/jobs/?[...]
   &job_printer_uri=javascript:alert("Hi mom!")
```

Diese Schwachstelle in CUPS kann repariert werden, aber es gibt wahrscheinlich viele andere fragwürdige lokale Dienste in allen Betriebssystemen, von Festplatten-Verwaltungsprogrammen bis zu Dashboards für den Status der Virenabwehr. Wenn Sie Eintrittspunkte öffnen, die auf 127.0.0.1 oder irgendein anderes Ziel weisen, das nicht Ihrer Kontrolle unterliegt, machen Sie die Cookie-Sicherheit in Ihrer Domäne von der Sicherheit irgendeiner Drittanbietersoftware abhängig. Und das sollten Sie tunlichst vermeiden.

8. Diese IP-Adresse ist für Loopback-Schnittstellen reserviert. Jeder Versuch, Verbindung damit aufzunehmen, führt zurück zu den Diensten auf dem Ursprungshost.

9.4.4 Cookies und »legitimes« DNS-Hijacking

Die Gefahren domänenweiter Gültigkeitsbereiche für Cookies hören nicht mit `localhost` auf. Ein weiteres unvorhergesehenes Problem hängt mit der weitverbreiteten und auch oft kritisierten Vorgehensweise mancher Internetprovider und anderer DNS-Dienstanbieter zusammen, Domänen-Lookups für nicht existierende (wahrscheinlich falsch eingegebene) Hosts abzufangen. Dabei gibt ein Namensserver, der weiter oben in der Hierarchie steht, nicht die in den Standards verlangte Antwort `NXDOMAIN` zurück (die anschließend eine Fehlermeldung im Browser oder einer anderen Netzwerkanwendung auslösen würde). Stattdessen fälscht der Provider einen Eintrag, um vorzugaukeln, dass der Name zu seiner eigenen Website aufgelöst wird. Diese Website wiederum untersucht den vom Browser angegebenen `Host`-Header und zeigt dem Benutzer unverlangte kontextbezogene Werbung an, die vage mit den eigentlich gesuchten Inhalten zu tun hat. Als übliche Rechtfertigung dafür wird eine höhere Benutzerfreundlichkeit beim Surfen vorgebracht, doch in Wirklichkeit geht es darum, Geld zu verdienen.

Internetprovider wie Cablevision, Charter, Earthlink, Time Warner, Verizon und viele andere handeln danach. Ihr Vorgehen ist jedoch nicht nur moralisch fragwürdig, sondern bietet auch ein erhebliches Sicherheitsrisiko. Wenn die Werbewebsite für Script-Injection anfällig ist, kann ein Angreifer diese Schwachstellen im Kontext jeder anderen Domäne ausnutzen, indem er einfach durch eine Adresse wie *nonexistent.example.com* darauf zugreift. In Verbindung mit dem Verhalten von HTTP-Cookies untergräbt diese Praxis die Sicherheit jeglicher angesprochener Dienste im Internet.

Wie nicht anders zu erwarten, lassen sich Script-Injection-Schwachstellen in solchen hastig zusammengeschusterten Werbefallen ohne große Anstrengung finden. Beispielsweise entdeckte und veröffentlichte Dan Kaminsky 2008 eine XSS-Schwachstelle auf den Seiten von Earthlink [7].

Ja, ich weiß, ich habe mich jetzt schon lange genug über Cookies ausgelassen. Es ist an der Zeit, das Thema zu wechseln.

9.5 Sicherheitsregeln für Plug-ins

Browser geben Plug-in-Entwicklern keine einheitliche und erweiterbare API zur Durchsetzung von Sicherheitsrichtlinien an die Hand. Stattdessen entscheidet jedes Plug-in, welche Regeln auf den ausgeführten Inhalt angewendet werden sollen und wie das zu geschehen hat. Folglich weichen die Sicherheitsmodelle von Plug-ins in verschiedenen Aspekten voneinander ab, auch wenn sie alle in gewissem Maße von der SOP inspiriert sind.

Diese Abweichungen können jedoch gefährlich sein. In Kapitel 6 haben wir gesehen, dass sich Plug-ins gern auf die Untersuchung des JavaScript-Objekts `location` verlassen, um den Ursprung ihrer Wirtsseite zu bestimmen. Diese unsin-

nige Vorgehensweise hat Browserentwickler dazu gezwungen, die Fähigkeiten von JavaScript-Programmen einzuschränken, sich an Teilen ihrer Laufzeitumgebung zu schaffen zu machen. Eine damit verwandte Quelle ständiger Inkompatibilitäten ist die Interpretation von URLs. Beispielsweise entdeckte ein Forscher Mitte 2010, dass Adobe Flash mit der folgenden URL Probleme hatte [8]:

```
http://example.com:80@bunnyoutlet.com/
```

Das Plug-in entschied, den Ursprung jeglichen Codes, der über diese URL eingeht, auf *example.com* zu setzen, während der Browser die Daten natürlich von *bunnyoutlet.com* abruft und dem verwirrten Plug-in zur Ausführung übergibt.

Dieser Fehler ist mittlerweile behoben, doch können wir davon ausgehen, dass es jetzt und in Zukunft noch andere Schwachstellen dieser Art gibt. Die in den Kapiteln 2 und 3 erörterten Eigentümlichkeiten bei der URL-Analyse nachzuvollziehen kann daher verlorene Liebesmüh sein und sollte am besten gar nicht erst versucht werden.

Es wäre allerdings unschön, dieses Kapitel mit einem so düsteren Hinweis enden zu lassen. Schieben wir die systematischen Probleme also beiseite und sehen wir uns an, wie einige der am weitesten verbreiteten Plug-ins bei der Durchsetzung von Sicherheitsrichtlinien vorgehen.

9.5.1 Adobe Flash

Das Sicherheitsmodell von Flash wurde 2008 notwendigerweise generalüberholt [9] und ist seitdem recht stabil. Allen geladenen Flash-Applets wird jetzt ein SOP-ähnlicher Ursprung zugewiesen, der von der Ursprungs-URL abgeleitet ist[9]. Außerdem werden ihnen nominelle Berechtigungen auf der Grundlage des Ursprungs zugewiesen, die im Großen und Ganzen mit denen von JavaScript vergleichbar sind. Insbesondere kann jedes Applet über Cookies authentifizierte Inhalte und einige eingeschränkte Datentypen von der Ursprungssite laden und `XMLHttpRequest`-ähnliche HTTP-Aufrufe mit demselben Ursprung durch die API `URLRequest` vornehmen. Der Satz der zugelassenen Methoden und Request-Header für die letztgenannte API ist ziemlich vernünftig zusammengestellt und zurzeit stärker eingeschränkt als die meisten der browsereigenen Blacklists für `XMLHttpRequest` selbst [10].

Über diese sinnvolle Grundausstattung hinaus gibt es noch drei flexible, aber leicht zu missbrauchende Mechanismen, mit denen dieses Verhalten in gewissem Maße verändert werden kann, wie im Folgenden erläutert wird.

---

9. In manchen Kontexten kann Flash den Zugriff von HTTPS-Ursprüngen auf HTTP implizit erlauben, aber nicht umgekehrt. Das ist gewöhnlich harmlos, weshalb wir uns im Rest dieses Abschnitts auch nicht weiter darum kümmern werden.

9.5.1.1 Sicherheitsfunktionen auf Markup-Ebene

Die einbettende Seite kann drei spezielle Parameter angeben, die in den Tags `<embed>` und `<object>` zur Verfügung stehen, und mit ihnen steuern, wie ein Applet mit seiner Wirtsseite und dem Browser umgeht:

- **`AllowScriptAccess`**
  Diese Einstellung regelt die Fähigkeit eines Applets, die JavaScript-Brücke `ExternalInterface.call(...)` zu verwenden (siehe Kapitel 8), um JavaScript-Anweisungen im Kontext der einbettenden Site auszuführen. Mögliche Werte sind `always`, `never` und `sameorigin`. Die letztgenannte Einstellung gewährt nur dann Zugriff auf die Seite, wenn sie denselben Ursprung hat wie das Applet selbst. (Vor den Änderungen im Jahr 2008 lautete der Standardwert für das Plug-in `always`. Der jetzige Standard `sameorigin` ist viel sicherer.)

- **`AllowNetworking`**
  Diese ungeschickt benannte Einstellung schränkt die Berechtigungen eines Applets zum Öffnen von Browserfenstern, zur Navigation darin und zum Senden von HTTP-Requests an den Ursprungsserver ein. Beim Standardwert `all` kann das Applet mit dem Browser zusammenarbeiten, bei `internal` kann es nur eine interne Kommunikation über das Flash-Plug-in durchführen, die keine anderen Abläufe unterbrechen. Wird der Parameter auf `none` gesetzt, werden damit die meisten Netzwerk-APIs ausgeschaltet[10]. (Vor den letzten Sicherheitsverbesserungen bot `allowNetworking=all` mehrere Möglichkeiten, um `allowScriptAccess=none` zu umgehen, z.B. durch den Aufruf von `getURL(...)` für eine `javascript:`-URL. Inzwischen aber sind alle Skript-URLs in dieser Situation ausgeschlossen.)

- **`AllowFullScreen`**
  Dieser Parameter legt fest, ob ein Applet den Vollbildmodus verwenden darf. Die möglichen Werte sind `true` und `false`, wobei `false` der Standardwert ist. Wie in Kapitel 8 erwähnt, ist es nicht ungefährlich, Flash-Applets diese Möglichkeit einzuräumen, da das Risiko einer Fälschung der Benutzeroberfläche besteht. Dies sollte daher nur dann geschehen, wenn es wirklich notwendig ist.

10. Mit dieser Einstellung lässt sich aber nicht verhindern, dass sensible Daten, die einem schädlichen Applet ausgesetzt sind, an Dritte weitergeleitet werden. Es gibt viele Seitenkanäle, die jedes Flash-Applet nutzen kann, um Informationen an eine mitwirkende Partei durchsickern zu lassen, ohne direkte Netzwerkanforderungen stellen zu müssen. Im einfachsten und allgemeinsten Fall können CPU-Lasten so manipuliert werden, dass sie einzelne Informationen an ein gleichzeitig geladenes Applet senden, das ständig die Reaktivität seiner Laufzeitumgebung abfragt.

9.5.1.2 Security.allowDomain(...)

Die Methode `Security.allowDomain(...)` [11] erlaubt Variablen und Funktionen in Flash-Applets Zugriff auf jeglichen JavaScript-Code und auf Applets mit einem anderen Ursprung. Doch Vorsicht: Sobald dieser Zugriff gewährt ist, gibt es keine zuverlässige Möglichkeit, um die Integrität des ursprünglichen Flash-Ausführungskontextes zu bewahren. Die Entscheidung, solche Berechtigungen zu vergeben, darf nicht auf die leichte Schulter genommen werden. Aufrufe von `allowDomain("*")` vorzunehmen, sollten unter Strafandrohung gestellt werden.

Es gibt auch eine Methode mit dem kuriosen Namen `allowInsecureDomain(...)`, was jedoch nicht bedeutet, dass `allowDomain(...)` besonders sicher sei. Die »unsichere« Variante ist in Wirklichkeit für die Rückwärtskompatibilität mit einer Uraltsemantik aus den Jahren vor 2003 gedacht, bei der noch nicht zwischen HTTP und HTTPS unterschieden wurde.

9.5.1.3 Security-Policy-Dateien

Mit `loadPolicyFile(...)` kann ein Flash-Applet seine Laufzeitumgebung anweisen, eine Sicherheitsrichtliniendatei (Security Policy File) von einer fast beliebigen URL abzurufen. Dieses XML-Dokument, das gewöhnlich *crossdomain.xml* heißt, definiert die domänenübergreifenden Zugriffsrechte basierend auf seinem Inhalt und der URL, von der es geladen wird [12]. Die Syntax einer solchen Policy-Datei ist größtenteils selbsterklärend und sieht wie folgt aus:

```
<cross-domain-policy>
   <allow-access-from domain="foo.example.com"/>
   <allow-http-request-headers-from domain="*.example.com"
      headers="X-Some-Header" />
</cross-domain-policy>
```

Die Policy kann Aktionen zulassen, die durch den HTTP-Stack des Browsers ausgeführt werden, z.B. das Laden von Ressourcen aus verschiedenen Ursprüngen oder die Ausgabe willkürlicher `URLRequest`-Aufrufe mit Headern von einer Whitelist. Flash-Entwickler versuchen, eine gewisse Trennung der Pfade durchzusetzen: Eine Richtlinie, die aus einem bestimmten Unterverzeichnis geladen wird, kann im Prinzip nur Zugriff auf Dateien in diesem Pfad erlauben. Angesichts der Wechselwirkungen mit SOP und den unterschiedlichen Semantiken der Pfadzuordnung in moderner Browsern und Web-Application-Frameworks ist es in der Praxis jedoch unklug, sich auf diese Grenze zu verlassen.

Hinweis

Es ist auch möglich, »Raw TCP-Sockets« über `XMLSocket` zu nutzen, was durch eine XML-Richtlinie geregelt wird. Nach der Generalüberholung von Flash im Jahre 2008 erfordert `XMLSocket` jedoch eine eigene Richtliniendatei, die an TCP-Port 843 des Zielservers bereitgestellt werden muss. Das ist ziemlich sicher, da keine anderen üblichen Dienste auf diesem Port laufen und auf vielen Betriebssystemen nur Benutzer mit erweiterten Rechten Dienste auf Ports unter 1024 starten können. Aufgrund von Wechselwirkungen mit einzelnen Firewall-Mechanismen (z.B. FTP-Protokollhelfern) kann dies jedoch zu Störungen auf Netzwerkebene führen [13]. Aber dieses Thema würde den Rahmen dieses Buches bei Weitem sprengen.

Wie nicht anders zu erwarten, sind schlecht konfigurierte *crossdomain.xml*-Richtlinien ein beträchtliches Sicherheitsrisiko. Besonders schlecht ist es, `allow-access-from`-Regeln anzugeben, die auf eine nicht völlig vertrauenswürdige Domäne verweisen. Die Angabe von `"*"` für diesen Parameter ist ungefähr genauso, als würden Sie `document.domain = "com"` ausführen. Das kommt schon einer gewissen Todessehnsucht gleich.

9.5.1.4 Das Risiko gefälschter Policy-Dateien

Abgesehen von eventuellen Konfigurationsfehlern ist mit dem richtliniengestützten Sicherheitsmodell von Adobe noch ein anderes Sicherheitsrisiko verbunden, nämlich die Möglichkeit, dass vom Benutzer kontrollierte Dokumente als domänenübergreifende Richtlinien interpretiert werden, was nicht im Sinne des Websitebesitzers ist.

Vor 2008 verwendete Flash einen Richtlinien-Parser, der für sein nachlässiges Verhalten beim Verarbeiten von `loadPolicyFile(...)`-Dateien berüchtigt war. Dabei übersprang er einfach führende unverständliche Zeichenfolgen, bis er zum öffnenden `<cross-domain-policy>`-Tag kam. Beim Herunterladen der Ressource ignorierte er überdies den vom Server zurückgegebenen MIME-Typ. Infolgedessen konnte schon die Bereitstellung eines gültigen, vom Benutzer gelieferten JPEG-Bildes ein ernsthaftes Sicherheitsrisiko darstellen. Das Plug-in übersprang auch alle HTTP-Redirects, sodass selbst einfache (und ansonsten harmlose) Dinge gefährlich wurden, wie etwa eine HTTP-Umleitung zu einem Ort, der nicht Ihrer Kontrolle unterliegt.

Hinweis

Es bleibt anzumerken, dass im Firefox-Browser aufgrund eines Designproblems die Umleitung auf eine `data:`-URL dafür sorgt, dass diese anschließend im Kontext der umleitenden Domain ausgeführt wird. Somit ist XSS möglich, wo es eigentlich keines geben sollte. Statistisch sind eine große Anzahl der offenen Redirects auch mit `data:`-URLs möglich – und sorgen so in Firefox für eine stark vergrößerte Angriffsoberfläche. Viele Browser und Webserver verhalten sich sonderbar bei HTTP-Redirects auf `javascript:`-URLs – `data:`-URLs funktionieren aber nur zu gut.

Mit der dringend erforderlichen Änderung des Verhaltens von `loadPolicyFile` wurden viele der groben Fehler korrigiert, aber die Standardwerte sind immer noch nicht ideal. Einerseits funktionieren Umleitungen jetzt so, wie man es erwarten darf, wobei es sich bei der Datei um ein wohlgeformtes XML-Dokument handeln muss. Andererseits erscheint die Auswahl der zulässigen MIME-Typen, zu denen `text/*`, `application/xml` und `application/xhtml+xml` gehören, doch etwas weit gefasst. `text/plain` und `text/csv` könnten dadurch als Richtliniendatei missverstanden werden, was nicht der Fall sein sollte.

Um das Problem abzuschwächen, haben sich die Ingenieure von Adobe zum Glück dazu entschieden, *meta-policies*, also *Metarichtlinien* einzuführen. Sie sind an einem vordefinierten Ort der obersten Ebene (*/crossdomain.xml*) abgelegt, der von Angreifern nicht überschrieben werden kann. Eine Metarichtlinie kann seitenweit gültige Einschränkungen für alle restlichen Richtlinien angeben, die von URLs aus dem Einflussbereich von Angreifern geladen werden. Die wichtigste dieser Einschränkungen ist `<site-control permitted-cross-domain-policies="...">`. Wenn dieser Parameter auf `master-only` gesetzt wird, ignoriert das Plug-in jegliche Unterrichtlinien. Ein weniger radikaler Wert, `by-conent-type`, erlaubt es zwar, zusätzliche Richtlinien zu laden, verlangt aber, dass sie einen eindeutigen `Content-Type`-Header mit dem Wert `text/x-cross-domain-policy` aufweisen müssen.

Es ist äußerst ratsam, eine Metarichtlinie zu verwenden, die eine dieser beiden Direktiven enthält.

9.5.2 Microsoft Silverlight

Für Silverlight hat man sich äußerst großzügig bei Flash bedient. Die Unterschiede zwischen den Sicherheitsmodellen dieser beiden Plug-ins liegen größtenteils nur in der Nomenklatur. Die Plattform von Microsoft verfolgt einen SOP-Ansatz, ersetzt dabei aber `allowScriptAccess` durch `enableHtmlAccess` und `crossdomain.xml` durch `clientaccesspolicy.xml` mit einer leicht abweichenden Syntax. Sie stellt die API `System.Net.Sockets` statt `XMLSocket` bereit und verwendet `HttpWebRequest` anstelle von `URLRequest`. Eigentlich geschieht hier nichts anderes, als die Blumen im Wohnzimmer neu zu arrangieren und die Vorhänge auszutauschen.

Die verblüffende Ähnlichkeit setzt sich mit der Liste der blockierten Request-Header für die API `HttpWebRequest` fort, die sogar die `X-Flash-Version` aus der Spezifikation von Adobe einschließt [14]. Diese Übereinstimmung ist jedoch kein Problem, sondern besser als ein komplett neues Sicherheitsmodell. Außerdem hat Microsoft eine Reihe willkommener Verbesserungen vorgenommen. Unter anderem wurde `allowDomain` zugunsten von `RegisterScriptableObject` aufgegeben. Dadurch können nur ausdrücklich angegebene Callbacks gegenüber Drittdomänen offengelegt werden.

9.5.3 Java

Das von Sun entwickelte (und jetzt offiziell zu Oracle gehörende) Java-Plug-in ist ein sehr spezieller Fall. Es wird kaum noch benutzt, weshalb seine Sicherheitsarchitektur seit etwa zehn Jahren nicht mehr genau untersucht wurde. Aufgrund seiner weiten Verbreitung kann es jedoch nicht einfach übergangen werden.

Je genauer man hinsieht, umso offensichtlicher wird es leider, dass die Vorstellungen, die bei Java Pate gestanden haben, mit dem modernen Web nicht mehr vereinbar sind. Beispielsweise erlaubt die Klasse `java.net.HttpURLConnection` [15], HTTP-Requests, die Anmeldeinformationen enthalten, an die Ursprungswebsite eines Applets zu senden. Dabei ist unter der »Ursprungswebsite« jedoch *jede* Website zu verstehen, die an einer bestimmten, von einer Überprüfung mit `java.net.URL.equals(...)` abgesegneten IP-Adresse bereitgestellt wird! Im Grunde genommen hebt dieses Modell die Trennung zwischen einzelnen virtuellen HTTP/1.1-Hosts auf, die von der SOP, von HTTP-Cookies und praktisch allen anderen heute üblichen Browsersicherheitsmechanismen durchgesetzt wird.

In die gleiche Kerbe schlagen die Klassen `java.net.URLConnection` [16], mit der ein Applet willkürliche Request-Header (darunter auch `Host`) senden kann, und `Socket` [17], die uneingeschränkte TCP-Verbindungen zu beliebigen Ports auf dem Ursprungsserver erlaubt. All diese Verhaltensweisen rufen im Browser und in allen modernen Plug-ins Stirnrunzeln hervor.

Hinweis

Java 7 macht die Angelegenheit noch prekärer. In Java 6 galt die Java-Applet-SOP weitestgehend für Requests auf Basis von IP-Adressen – in Java 7 nun aber für Protokolle, Hostnamen und Ports, ähnlich wie im Browser. Das Problem ist, wenn der Domain-SOP-Check fehlschlägt, wird im Anschluss *per Default* der IP-Check durchgeführt. Ist dieser erfolgreich, so erlaubt die Runtime dem Applet eine domänenübergreifende Kommunikation. Das sorgt dafür, dass Seiten unterschiedlicher Domänen auf gleicher IP-Adresse fast unbeschränkt voneinander lesen können. Java macht's möglich.

Ein Zugriff vom Applet zur einbettenden Seite ohne Berücksichtigung des Ursprungs ist durch den `JSObject`-Mechanismus möglich. Geregelt werden soll dies von der einbettenden Partei durch die Angabe des Attributs `mayscript` in den Tags `<applet>`, `<embed>` oder `<object>` [18]. In der Dokumentation wird dies als eine Sicherheitsvorkehrung herausgestellt:

> *Aus Sicherheitsgründen ist die Unterstützung von* `JSObject` *im Java-Plug-in standardmäßig nicht aktiviert. Um die Unterstützung von* `JSObject` *im Java-Plug-in zu aktivieren, muss das neue Attribut* `MAYSCRIPT` *im Tag* `EMBED/OBJECT` *vorhanden sein. Zumindest gilt dies für die meisten modernen*

Browser – einige ignorieren das Fehlen des Attributs und erlauben Scripting auch ohne explizite Befugnis.

Leider wird in der Dokumentation nicht erwähnt, dass der nahe damit verwandte Mechanismus `DOMService` [19] diese Einstellung ignoriert und Applets größtenteils unbeschränkten Zugriff auf die einbettende Seite gewährt. Zwar wird `DOMService` in Firefox und Opera nicht unterstützt, ist aber in anderen Browsern verfügbar. Dadurch wird jeder Versuch, Java-Inhalte von Dritten zu laden, gleichbedeutend damit, Vollzugriff auf die einbettende Website zu gewähren.

Na toll!

Hinweis

Eine interessante Feststellung am Rande: In neuen Versionen von Java wird versucht, die in Flash verfügbare Unterstützung für `crossdomain.xml` zu kopieren.

9.6 Umgang mit unklaren oder unerwarteten Ursprungsangaben

Mit diesem Abschnitt beenden wir unseren Überblick über die grundlegenden Mechanismen für Sicherheitsrichtlinien und die Isolierung von Inhalten. Wenn es eine Lehre gibt, die wir daraus ziehen können, dann diese: Die meisten dieser Mechanismen stützen sich auf die Verfügbarkeit eines wohlgeformten, kanonischen Hostnamens, von dem sie den Kontext alle nachfolgenden Operationen abrufen. Was aber, wenn diese Information nicht vorhanden ist oder nicht in der erwarteten Form bereitgestellt wird?

Das ist der Punkt, an dem die Sache wirklich interessant wird. Sehen wir uns einige der nicht ganz so seltenen Ausnahmefälle an, und sei es auch nur so zum Spaß.

9.6.1 IP-Adressen

Da beim Entwurf von HTTP-Cookies und der SOP versäumt wurde, IP-Adressen zu berücksichtigen, haben fast alle Browser früher zugelassen, dass Dokumente, die z.B. von *http://1.2.3.4/* geladen wurden, Cookies für die »Domäne« **.3.4* festlegten. Das hat ähnliche Auswirkungen wie eine entsprechende Einstellung von `document.domain`. Einige dieser Verhaltensweisen sind in älteren Versionen des Internet Explorer immer noch anzutreffen.

Dieses Verhalten hat höchstwahrscheinlich keinen Einfluss auf etablierte Webanwendungen, da sie nicht für den Zugriff über eine IP-Adresse gedacht sind und in diesem Fall meistens gar nicht richtig funktionieren. Es gibt jedoch eine Handvoll Systeme, hauptsächlich im technischen Bereich, die über ihre IP-Adresse zugänglich sind und für die vielleicht nicht einmal DNS-Einträge eingerichtet sind.

In diesem Fall kann es ein Problem sein, wenn *http://1.2.3.4/* in der Lage ist, Cookies für *http://123.234.3.4/* einzuschleusen. Auch die Administrationsschnittstellen von Routern für Heimnetzwerke, die über die IP-Adresse aufgerufen werden, sind in diesem Zusammenhang von Interesse.

9.6.2 Hostnamen mit zusätzlichen Punkten

Algorithmen zum Festlegen von Cookies verlassen sich im Grunde genommen immer noch darauf, die Anzahl der Punkte in der URL zu zählen, um zu bestimmen, ob ein bestimmter `domain`-Parameter akzeptiert wird. Um den Aufruf durchzuführen, wird die Anzahl gewöhnlich mit einer Liste aus mehreren Hundert Einträgen der von den Herstellern gepflegten Public Suffix List (*http://publicsuffix.org/*) verglichen.

Leider ist es häufig möglich, zusätzliche Punkte in einen Hostnamen einzufügen und sie dann trotzdem korrekt auflösen zu lassen. Eine nicht kanonische Darstellung von Hostnamen mit zusätzlichen Punkten wird gewöhnlich von Auflösungsmechanismen des Betriebssystems akzeptiert, was dann den Browser in Verwirrung stürzt. Zwar wird der Browser nicht automatisch davon ausgehen, dass *www.example.com.pl.* (mit dem zusätzlichen Punkt am Ende) mit der echten Domäne *www.example.com.pl* identisch ist, doch dieser feine und scheinbar harmlose Unterschied in der URL entgeht leicht der Aufmerksamkeit auch umsichtiger Benutzer.

Der Umgang mit einer solchen URL mit nachfolgendem Punkt kann gefährlich sein, da andere Dokumente der Domäne **.com.pl.* relativ leicht domänenübergreifende Cookies einschleusen können.

Dieses Problem beim Punktezählen wurde zuerst um 1998 erkannt [20]. Etwa ein Jahrzehnt später haben viele Browserhersteller einfache Vorkehrungen zur Reduzierung des Risikos eingeführt und dem betroffenen Code damit noch einen weiteren Sonderfall hinzugefügt. Opera ist zurzeit jedoch immer noch für diesen Trick anfällig.

9.6.3 Nicht vollständig qualifizierte Hostnamen

Viele DNS-Auflösungsmechanismen sind so eingerichtet sind, dass sie – oft ohne Wissen des Benutzers – lokale Suffixe an alle gefundenen Hostnamen anhängen. Internetprovider oder Unternehmen billigen diese Einstellung häufig durch die automatische Netzwerkkonfiguration (Dynamic Host Configuration Protocol, DHCP).

Ist ein Benutzer mit einer solchen Einstellung im Web unterwegs, ist die Auflösung von DNS-Labels nicht mehr eindeutig. Enthält der DNS-Suchpfad beispielsweise *coredump.cx*, dann kann *www.example.com* zu der echten Website *www.example.com* aufgelöst werden, aber auch zu *www.example.com.core-*

dump.cx, falls ein solcher Eintrag existiert. Das Ergebnis wird teilweise von den Konfigurationseinstellungen geregelt und kann in gewissem Maße von einem Angreifer gesteuert werden.

Für den Browser erscheinen beide Orte identisch, was bemerkenswerte Nebenwirkungen haben kann. Betrachten Sie folgenden widersinnigen Fall: Sollte *http://com*, das tatsächlich zu *http://com.coredump.cx/* aufgelöst wird, in der Lage sein, durch Weglassen des Parameters `domain` Cookies für **.com* festzulegen?

9.6.4 Lokale Dateien

Da lokale Ressourcen, die über das Protokoll `file:` geladen werden, keinen ausdrücklichen Hostnamen aufweisen, kann der Browser keine normalen Ursprungsangaben ermitteln. Lange Zeit haben die Hersteller in diesem Fall einfach auf die SOP verzichtet. Alle auf der Festplatte gespeicherten HTML-Dokumente erhielten dadurch über `XMLHttpRequest` oder das DOM automatisch Zugriff auf alle anderen lokalen Dateien. Noch unerklärlicher ist die Tatsache, dass sie auf dieselbe Weise sogar Zugriff auf beliebige Inhalte aus dem Internet erhalten.

Diese Entscheidung erwies sich als katastrophal. Niemand ging davon aus, dass das bloße Herunterladen eines HTML-Dokuments alle lokalen Dateien des Benutzers und seine Online-Anmeldeinformationen gefährdete, vor allem, da der Webzugriff auf dasselbe Dokument völlig sicher war.

Bei vielen Browsern wurde versucht, diese Lücke in den letzten Jahren zu schließen, wobei sehr unterschiedliche Erfolge erzielt wurden:

- **Chrome (und andere WebKit-Browser)**
  Chrome verbietet jeglichen domänenübergreifenden DOM- und `XMLHttpRequest`-Zugriff von `file:`-URIs und ignoriert Aufrufe von `dokument.cookie` sowie `<meta http-equiv="Set-Cookie" ...>`-Direktiven in dieser Einstellung. Der Zugriff auf den gemeinsamen Container `localStorage` aller `file:`-Dokumente ist erlaubt, aber das mag sich bald ändern.
- **Firefox**
  Der Browser von Mozilla lässt nur den Zugriff auf Dateien aus dem Verzeichnis des ursprünglichen Dokuments sowie aus benachbarten Unterverzeichnissen zu. Diese Richtlinie ist ganz gut, birgt aber immer noch Risiken für Dokumente, die an diesem Ort gespeichert oder zuvor dort heruntergeladen wurden. Ein Zugriff auf Cookies über `document.cookie` oder `<meta http-equiv="Set-Cookie" ...>` ist möglich, und alle `file:`-Cookies sind für jeden anderen lokalen JavaScript-Code sichtbar[11]. Das Gleiche gilt für den Zugriff auf Speichermechanismen.

- **Internet Explorer 7 und höher**
  Ein uneingeschränkter Zugriff auf lokale und Internetinhalte aus `file:`-URIs ist zulässig, erfordert aber, dass der Benutzer auf eine unspezifische Meldung klickt, um zunächst die Ausführung von JavaScript zu erlauben. Welche Folgen das hat, wird jedoch nicht deutlich erklärt (das Hilfesystem ergeht sich in dunklen Andeutungen: »Internet Explorer schränkt diesen Inhalt ein, da diese Programme gelegentlich Fehlfunktionen aufweisen oder Ihnen unerwünschte Inhalte bereitstellen können.«). Viele Benutzer lassen sich trotzdem dazu verleiten, auf OK zu klicken.

  Die Cookie-Semantik des Internet Explorer ähnelt der von Firefox. Web Storage wird in diesem Ursprung jedoch nicht unterstützt.

- **Opera und Internet Explorer 6**
  Beide Browser lassen den uneingeschränkten DOM- und `XMLHttpRequest`-Zugriff ohne weitere Prüfung zu. Auch sind nicht getrennte `file:`-Cookies erlaubt.

Hinweis

Was `file:` angeht, folgen alle Plug-ins ihren eigenen Regeln: Flash verwendet das Sandboxmodell `local-with-filesystem` [21], was unabhängig von der vom Browser durchgesetzten Richtlinie einen größtenteils uneingeschränkten Zugriff auf das lokale Dateisystem gewährt. Die Ausführung von Java-Applets oder Applets des Windows Presentation Framework auf dem lokalen Dateisystem entspricht in einigen Fällen der Verwendung nicht vertrauenswürdiger Binärdateien.

9.6.5 Pseudo-URLs

Das Verhalten von Pseudo-URLs wie `about:`, `data:` und `javascript:` stellte anfangs eine erhebliche Sicherheitslücke in der SOP dar. Bei allen diesen URLs wurde ein gemeinsamer Ursprung für alle darüber geladenen Ressourcen angenommen, so dass diese einen unbeschränkten domänenübergreifenden Zugriff aufeinander erhielten. Das jetzige Verhalten ist jedoch ganz anders und wird uns im nächsten Kapitel beschäftigen. Kurz gesagt, ist der jetzige Zustand das Ergebnis mehrerer hastiger Verbesserungen und bildet eine Mischung aus browserspezifischen Sonderfällen und Regeln zur Ursprungsvererbung.

---

11. Da es keine Trennung von `file:`-Cookies gibt, ist es gefährlich, sie für legitime Zwecke heranzuziehen. Einige lokal installierte HTML-Anwendungen ignorieren diese Empfehlung jedoch, weshalb ihre Cookies sehr leicht von jedem heruntergeladenen und möglicherweise schädlichen HTML-Dokument manipuliert werden können, das sich der Benutzer ansieht.

9.6.6 Browsererweiterungen und Benutzerschnittstelle

Verschiedene Browser erlauben die Ausführung von JavaScript-gestützten Benutzeroberflächenelementen und von Benutzern installierten Browsererweiterungen mit erweiterten Rechten. Das kann die Umgehung einzelner SOP-Prüfungen oder den Aufruf normalerweise unerreichbarer APIs zum Schreiben von Dateien, zum Ändern von Konfigurationseinstellungen usw. umfassen.

JavaScript mit erweiterten Rechten ist ein herausragendes Merkmal von Firefox, wo es zusammen mit XUL verwendet wird, um große Teile der Browseroberfläche aufzubauen. Auch Chrome stützt sich in geringerem, aber immer noch bemerkenswertem Maße auf JavaScript mit erweiterten Rechten.

Die SOP hat keine besonderen Vorkehrungen zur Unterstützung von Kontexten mit erweiterten Rechten. Der Mechanismus, durch den die zusätzlichen Rechte gewährt werden, kann darin bestehen, das Dokument über ein besonderes und normalerweise nicht zugängliches URL-Schema wie `chrome:` oder `res:` zu laden und in anderen Teilen des Browsercodes Sonderfälle dafür hinzuzufügen. Als eine andere Möglichkeit kann einfach ein binäres Flag für einen JavaScript-Kontext unabhängig von seinem Ursprung umgeschaltet und später untersucht werden. In jedem Fall lässt sich das Verhalten von Standard-APIs wie `localStorage`, `document.domain` und `document.cookie` nicht vorhersagen, weshalb man sich nicht darauf verlassen sollte: Manche Browser versuchen die Isolierung der Kontexte verschiedener Erweiterungen aufrechtzuerhalten, die meisten aber nicht.

Hinweis

Wenn Sie Browsererweiterungen schreiben, müssen Sie jegliche Interaktion mit Kontexten ohne erweiterte Rechte mit äußerster Vorsicht vornehmen. Die Untersuchung nicht vertrauenswürdiger Kontexte kann schwierig sein, und die Verwendung von Mechanismen wie `eval(...)` und `innerHTML` kann Möglichkeiten zur Übertragung von Rechten eröffnen.

9.7 Andere Verwendungen für Origin-Angaben

Das ist alles, was es zunächst zur Inhaltsisolierung auf Browserebene zu sagen gibt. Das Prinzip des Ursprungs und der host- und domänengestützten Sicherheitsmechanismen ist nicht auf bestimmte Aufgaben beschränkt und taucht bei Browsern noch an vielen anderen Stellen auf. Beispielsweise stützen sich auch andere Datenschutz- und Sicherheitsvorkehrungen in gewissem Sinne auf Ursprungsangaben, etwa Caching für Cookies, Pop-up-Blocker, Geodaten-Sharing, Passwortverwaltung, Kamera- und Mikrofonzugriff (in Flash) und viele andere Features. Diese Einrichtungen hängen zumindest teilweise mit den in diesem Kapitel beschriebenen Sicherheitsmaßnahmen zusammen. Wir schauen uns dieses Thema in Kürze genauer an.

Spickzettel für Webentwickler

Gute Sicherheitsrichtlinien für alle Websites

Um Ihre Benutzer zu schützen, sollten Sie auf der obersten Ebene die Datei *crossdomain.xml* platzieren, bei der der Parameter `permitted-cross-domain-policies` auf `master-only` oder `by-content-type` gesetzt ist, auch wenn Sie Flash nirgendwo auf Ihrer Website verwenden. Dadurch verhindern Sie, dass Inhalte von Angreifern als sekundäre `crossdomain.xml`-Datei interpretiert werden, was die Wirkung der SOP in Browsern mit aktiviertem Flash untergraben würde.

Wenn Sie sich zur Authentifizierung auf HTTP-Cookies verlassen

- ☑ Verwenden Sie das Flag `httpOnly`: und entwerfen Sie die Anwendung so, dass JavaScript keinen Grund hat, direkt auf Authentifizierungscookies zuzugreifen. Sensible Cookies sollten einen so engen Gültigkeitsbereich aufweisen wie möglich. Am besten ist es, `domain` überhaupt nicht anzugeben.
- ☑ Bei einer reinen HTTPS-Anwendung sollten die Cookies als `secure` gekennzeichnet sein. Außerdem müssen Sie sich darauf vorbereiten, verantwortungsvoll mit einer Cookie Injection umzugehen. (HTTP-Kontexte können `secure`-Cookies zwar nicht lesen, aber überschreiben.) Eine kryptografische Signierung von Cookies kann gegen ungehinderte Änderungen helfen. Sie hilft aber nicht, wenn die Cookies eines Opfers durch andere, legitim erworbene Anmeldeinformationen ersetzt werden.

Wenn Sie in JavaScript die domänenübergreifende Kommunikation einrichten

- ☑ Verwenden Sie in diesem Fall nicht `document.domain`, sondern nutzen Sie nach Möglichkeit `postMessage(...)` und geben Sie den Ursprung des Ziels korrekt an. Überprüfen Sie den Ursprung des Absenders, wenn Sie die Daten am anderen Ende empfangen. Hüten Sie sich vor naiven Teilstringvergleichen für Domänennamen: `msg.origin.indexOf(".example.com")` ist äußerst unsicher.
- ☑ Bedenken Sie, dass verschiedene Tricks zum Umgehen der SOP, die `postMessage` vorausgingen, nicht gegen Manipulationen geschützt sind (beispielsweise die Verwendung von `window.name`). Daher sollten sie für den Austausch sensibler Daten nicht eingesetzt werden.

Wenn Sie aktive Inhalte von Drittanbietern für Plug-ins einbetten

Schlagen Sie zuerst im Spickzettel von Kapitel 8 nach. Dort erhalten Sie allgemeine Ratschläge.

- ☑ **Flash:**
  Geben Sie `allowScriptAccess=always` nur dann an, wenn Sie dem Besitzer der Ursprungsdomäne und der Sicherheit ihrer Website absolut vertrauen. Verwenden Sie diese Einstellung nicht, wenn Sie HTTP-Applets auf HTTPS-Seiten einbetten. Außerdem sollten Sie `allowFullScreen` und `allowNetworking` sinnvoll einschränken.
- ☑ **Silverlight:**
  Geben Sie `enableHTMLAccess=true` nur dann an, wenn Sie der Ursprungsdomäne vertrauen (wie oben).
- ☑ **Java:**
  Java-Applets aus nicht vertrauenswürdigen Quellen lassen sich nicht auf sichere Weise einbetten. Der Verzicht auf `mayscript` bietet keinen absoluten Schutz gegen den Zugriff auf die einbettende Seite. Versuchen Sie das also erst gar nicht.

Wenn Sie eigene Inhalte für Plug-ins bereitstellen

- ☑ Beachten Sie, dass viele von Browser-Plug-ins bereitgestellte Mechanismen zur domänenübergreifenden Kommunikation unvorhergesehene Auswirkungen haben. Vermeiden Sie insbesondere, in `crossdomain.xml`, `clientaccesspolicy.xml` und `allowDomain(...)` Regeln zu verwenden, die auf nicht absolut vertrauenswürdige Domänen verweisen.

Wenn Sie Browsererweiterungen schreiben

- ☑ Nutzen Sie weder `innerHTML`, `document.write(...)` und `eval(...)` noch irgendwelche anderen fehleranfälligen Codemuster, die zu einer Code-Injection auf Drittseiten oder in einem JavaScript-Kontext mit erweiterten Rechten führen können.
- ☑ Treffen Sie keine sicherheitskritischen Entscheidungen auf Basis von DOM-Eigenschaften, die nicht vertrauenswürdigen Kontexten entstammen. Deren Verhalten kann irreführend oder vom Angreifer gezielt verändert worden sein.