13 Mechanismen zur Inhaltserkennung

Bis jetzt haben wir einige gutgemeinte Browsermerkmale betrachtet, die sich im Laufe der Entwicklung der Technologie als kurzsichtig und geradezu gefährlich erwiesen haben. In der Geschichte des Web hat sich jedoch nichts als so fehlgeleitet herausgestellt wie das sogenannte *Content-Sniffing*.

Ursprünglich lag dem Content-Sniffing folgende simple Annahme zugrunde: Browseranbieter gingen davon aus, dass es in manchen Fällen angemessen – und sogar wünschenswert – sei, die normalerweise vom Server stammenden verbindlichen Metadaten eines geladenen Dokuments zu ignorieren, so etwa den Header `Content-Type`. Anstatt die erklärte Absicht des Entwicklers zu akzeptieren, versuchen viele existierende Browser stattdessen den Inhaltstyp zu erraten, indem sie proprietäre Heuristiken auf die vom Server zurückgegebenen Daten anwenden. Das Ziel dieses Vorgehens ist es, eventuelle Unstimmigkeiten zwischen Typ und Inhalt zu »korrigieren«. (Erinnern Sie sich an Kapitel 1, wo wir feststellten, dass Anbieter während der ersten Browserkriege die Fehlertoleranz und Kompatibilität in einen wenig durchdachten Wettbewerbsvorteil verwandelten.)

Es dauerte nicht lange, bis Funktionen zum Sniffen von Inhalten als wesentlicher und schädlicher Aspekt der gesamten Browsersicherheitslandschaft auftauchten. Die Webentwickler mussten zu ihrem Schrecken bald feststellen, dass sie bestimmte, dem Namen nach harmlose Dokumenttypen wie `text/plain` oder `text/csv` nicht gefahrlos im Auftrag ihrer Benutzer bereitstellen konnten. Jeder derartige Versuch barg unweigerlich das Risiko, dass der Inhalt als HTML fehlinterpretiert werden konnte.

Vielleicht zum Teil als Reaktion auf dieses Problem wurde die Praxis des unerwünschten Content-Sniffing 1999 in HTTP/1.1 explizit verboten:

> *»Wenn und **nur wenn** der Medientyp nicht durch ein Feld* `Content-Type` *angegeben ist, darf der Empfänger versuchen, den Typ durch Untersuchung des Inhalts und/oder der Namenserweiterung(en) der URI herauszubekommen, der zur Bezeichnung der Ressource verwendet wird.«*

Leider erschien diese ungewöhnlich deutliche Forderung ein wenig zu spät. Die meisten Browser verletzten diese Regel bereits in gewissem Maße. Und weil es keine einfache Methode gab, um die potenziellen Konsequenzen einer Anpassung der existierenden Implementierungen klar einzuschätzen, zogen es die Browserhersteller zunächst vor, einfach auf den möglicherweise problematischen Fix zu verzichten. Die eklatantesten Fehler wurden im Lauf des letzten Jahrzehnts vorsichtig behoben. Aber insbesondere zwei Firmen – Microsoft und Apple – leisteten erheblichen Widerstand. Sie beschlossen, dass die Interoperabilität mit schlampig und missverständlich programmierten Webanwendungen wichtiger sei als die offensichtlichen Sicherheitsprobleme. Um etwaige Kritiker ruhigzustellen, implementierten sie ein paar unvollständige Sicherheitsmechanismen, die das Risiko ein wenig mindern sollten.

Das Flickwerk aus Richtlinien zum Umgang mit Inhalten und die dazu formulierten Einschränkungen werfen seitdem einen großen Schatten auf die Onlinewelt. Dies macht es heute nahezu unmöglich, bestimmte Arten von Webdiensten zu schreiben, ohne Zuflucht zu ausgeklügelten und gelegentlich teuren Tricks zu nehmen. Um diese Einschränkungen zu verstehen, skizzieren wir zunächst einige Szenarios, in denen ein normalerweise passives Dokument wie beispielsweise eine harmlose Text-Datei vom Browser als aktives HTML oder Schlimmeres missdeutet werden kann.

13.1 Dokumenttypen erkennen

Die einfachste und am wenigsten umstrittene Art der Dokumenttypheuristik wird von allen modernen Browsern implementiert und betrifft den Umgang mit fehlenden `Content-Type`-Headern. Diese sehr selten auftretende Situation kann dadurch entstehen, dass der Entwickler den Header versehentlich weggelassen oder falsch geschrieben hat oder dass das Dokument über einen anderen Transportmechanismus geladen wird als HTTP, zum Beispiel über `ftp:` oder `file:`.

Für HTTP erlaubten die ursprünglichen RFCs dem Browser einst ausdrücklich, dass er die Nutzlast auf Hinweise zum Inhaltstyp untersuchen darf, wenn der Wert `Content-Type` nicht verfügbar ist. Bei anderen Protokollen wird üblicherweise derselbe Ansatz verfolgt, häufig als natürliche Folge der Gestaltung des zugrunde liegenden Codes.

Die Heuristik, um den Dokumenttyp zu bestimmen, umfasst normalerweise die Prüfung auf statische Signaturen für einige Dutzend bekannte Dateiformate (zum Beispiel Bilder und gängige Dateien, die von Plug-ins behandelt werden – auch als Magic Bytes bezeichnet). Außerdem wird die Antwort auf bekannte Teilzeichenfolgen untersucht, um signaturlose Formate wie HTML zu erkennen (in diesem Fall sucht der Browser nach vertrauten Tags – `<body>`, `<font>` usw.). Viele Browser berücksichtigen auch Hinweise, die aufgrund ihrer Manipulierbarkeit nicht sehr zuverlässig sind, etwa die angehängten Zeichenfolgen `.html` oder `.swf` im Pfadteil der URL.

Die Besonderheiten der Content-Sniffing-Verfahren unterscheiden sich von Browser zu Browser erheblich; sie sind weder gut dokumentiert noch standardisiert. Betrachten Sie zur Veranschaulichung die Behandlung von Adobe-Flash-Dateien (SWF), die ohne `Content-Type` geliefert werden: In Opera werden sie aufgrund einer Inhaltsprüfung ohne Weiteres korrekt erkannt; in Firefox und Safari ist das Suffix `.swf` in der URL explizit erforderlich; Internet Explorer und Chrome erkennen SWF überhaupt nicht automatisch.

Selbstverständlich ist das Dateiformat SWF kein Ausnahmefall. Wenn es beispielsweise um HTML-Dateien geht, identifizieren Chrome und Firefox das Dokument nur dann als HTML, wenn direkt am Dateianfang eines von mehreren vordefinierten HTML-Tags steht. Firefox »erkennt« HTML allein auf Grundlage der Extension `.html` in der URL – selbst wenn kein erkennbares Markup zu sehen ist. Der Internet Explorer wählt im Gegensatz dazu einfach standardmäßig HTML, wenn `Content-Type` fehlt, und Opera sucht in den ersten tausend Bytes des Dokuments nach bekannten HTML-Tags.

Hinter all diesem Wahnsinn steht die Annahme, dass das Fehlen von `Content-Type` einen bewussten Wunsch desjenigen ausdrückt, der die Seite veröffentlicht – diese Annahme trifft jedoch nicht immer zu und hat zu einer ganzen Anzahl von Sicherheitsfehlern geführt. Vor diesem Hintergrund erzwingen die meisten Webserver aktiv das Vorhandensein des Headers `Content-Type` und setzen einen Standardwert ein, wenn die serverseitigen Skripte, die die Benutzeranforderungen bearbeiten, nicht explizit einen erzeugen. Es gibt also keinen Grund zur Aufregung? Nun, leider endet die Geschichte des Content-Sniffing noch nicht an dieser Stelle.

13.1.1 Nicht wohlgeformte MIME-Typen

Der HTTP-RFC erlaubt das Sniffen von Inhalten nur, wenn Header-Daten zum Inhaltstyp fehlen; dem Browser wird klar untersagt, Vermutungen über die Absicht des Webmasters anzustellen, wenn ein Header in irgendeiner Form vorhanden ist. Dieser Rat wird jedoch in der Praxis nicht ernst genommen. Zudem beschlossen diverse Browserhersteller auch noch, eine Heuristik einzusetzen, falls der vom Server zurückgegebene MIME-Typ irgendwie ungültig schien.

Laut RFC soll der Header `Content-Type` aus zwei durch Schrägstrich getrennten alphanumerischen Token bestehen (`type/subtype`), auf die weitere, jeweils durch Semikolon abgetrennte Parameter folgen können. Diese Token dürfen alle 7-Bit-ASCII-Zeichen mit Ausnahme von Leerraum und einigen bestimmten »Separatoren« enthalten (eine generische Gruppe, die Zeichen wie @, ? und den Schrägstrich selbst umfasst). Die meisten Browser versuchen diese Syntax durchzusetzen, gehen dabei aber nicht konsistent vor. Das Fehlen eines Schrägstrichs gilt fast generell als Einladung zum Content-Sniffing, ebenso das Vorkommen von Leerraum oder bestimmter (aber nicht aller) Steuerzeichen im ersten Teil des

Bezeichners (des Tokens `type`). Nur bei Opera ist es so, dass die technisch unzulässige Verwendung von High-Bit-Zeichen oder Separatoren die Gültigkeit dieses Feldes beeinträchtigt.

Die Gründe für dieses Vorgehen sind schwer zu verstehen, aber der Einfluss auf die Sicherheit ist, um fair zu bleiben, trotzdem recht begrenzt. Soweit es die Entwickler von Webanwendungen betrifft, müssen sie Sorgfalt walten lassen: Sie müssen Schreibfehler im Wert `Content-Type` vermeiden und sicherstellen, dass die Benutzer nicht eigene Typen – beispielsweise per Header Injection oder META-Tag – angeben können. Insbesondere sollten Entwickler Blacklists zur Erkennung von potenziell schädlichen MIME-Types vermeiden – es ist quasi vorprogrammiert, dass man riskante Dateitypen übersieht. Diese Anforderungen mögen auf den ersten Blick überraschend sein, spielen jedoch eine substanziell wichtige Rolle.

13.1.2 Besondere Werte für den Inhaltstyp

Das erste deutliche Signal dafür, dass Content-Sniffing gefährlich wurde, war die Handhabung des anscheinend unauffälligen MIME-Typs `application/octet-stream`. Dieser besondere Wert wird in der HTTP-Spezifikation überhaupt nicht erwähnt, bekommt aber tief im Inneren von RFC 2046 eine spezielle (aber etwas unklare) Rolle:

> *»Die empfohlene Aktion für eine Implementierung, die eine Entität vom Typ* `application/octet-stream` *empfängt, besteht darin, die Daten in einer Datei abzulegen, wobei jedes* `Content-Transfer-Encoding` *rückgängig gemacht wird, oder sie vielleicht als Eingabe für einen vom Benutzer angegebenen Prozess zu verwenden.«[1]*

Die ursprüngliche Absicht dieses MIME-Typs geht allein aus der zitierten Passage möglicherweise nicht ganz klar hervor. Sie wird aber üblicherweise so gedeutet, dass der Webserver mit diesem MIME-Type mitteilen kann, dass die zurückgegebene Datei keine besondere Bedeutung für den Server hat und für den Client keine speziellen Prozesse anstoßen sollte – wie beispielsweise das Rendern von HTML oder Schlimmeres. Infolgedessen betrachten die meisten Webserver alle Arten undefinierter oder unbekannter Nicht-Webdateien – beispielsweise ausführbare Dateien oder Archive zum Herunterladen – standardmäßig als Typ `application/octet-stream`, wenn sie keinen besser passenden Inhaltstyp finden. Bei seltenen Administratorfehlern (zum Beispiel, wenn die unbedingt erforderlichen `AddType`-Direktiven in Apache-Konfigurationsdateien gelöscht wurden) können Webserver bei Dokumenten, die für browserinterne Nutzung gedacht sind, ebenfalls auf die Verwendung dieses MIME-Typs zurückfallen. Dieser Konfigurationsfehler lässt sich natürlich sehr leicht feststellen und beheben, aber Microsoft, Opera und Apple haben sich trotzdem dafür entschieden, ihn eigenhändig auszu-

gleichen. Die Browser dieser Anbieter betreiben eifrig Content-Sniffing, sobald sie den Typ `application/octet-stream` sehen.[1]

Dieses Design machte es Webanwendungen plötzlich schwerer, Binärdateien im Auftrag des Benutzers bereitzustellen. Jede Plattform, die Code beherbergt, muss zum Beispiel bei der Rückgabe von ausführbaren Dateien oder Quellarchiven als `application/octet-stream` Vorsicht walten lassen, weil das Risiko besteht, dass sie als HTML fehlinterpretiert und inline angezeigt werden können. Das stellt für jedes Softwarehosting- oder Webmailsystem und für viele andere Webanwendungen ein großes Problem dar. (Es ist etwas sicherer für sie, einen beliebigen anderen allgemein klingenden MIME-Typ zu verwenden, wie etwa `application/binary`, weil es dafür keinen Sonderfall im Browsercode gibt.)

Neben der besonderen Behandlung für `application/octet-stream` gibt es eine weitere, erheblich kritischere Ausnahme: `text/plain`. Diese Entscheidung, die nur den Internet Explorer und Safari betrifft, lässt sich bis zu RFC 2046 zurückverfolgen. In diesem Dokument bekommt `text/plain` eine doppelte Funktion zugeteilt: erstens zur Übertragung von Klartextdokumenten (solchen, die »*keine Formatierungsbefehle, Angaben für Schriftattribute, Verarbeitungsanweisungen, Interpretationsdirektiven oder Inhaltsauszeichnungen berücksichtigen oder zulassen*«) und zweitens als Rückfallwert für alle textbasierten Dokumente, die vom Absender nicht als etwas anderes erkannt werden.

Die Unterscheidung zwischen `application/octet-stream` und dem eventuellen Rückgriff auf `text/plain` war für E-Mails (einem Thema, mit dem sich dieser RFC ursprünglich befasste) ganz und gar sinnvoll. Sie erwies sich jedoch für das Web als erheblich weniger relevant. Trotzdem übernehmen einige Webserver `text/plain` als Rückfallwert für bestimmte Antworttypen (vor allem die Ausgabe von CGI-Skripten).

Die später im Internet Explorer und in Safari umgesetzte `text/plain`-Logik zur Erkennung von HTML ist in einem solchen Fall eine ziemlich schlechte Idee: Sie beraubt Webentwickler der Fähigkeit, diesen MIME-Typ gefahrlos zum Erstellen benutzerspezifischer Klartextdokumente zu nutzen, ohne ihnen dafür eine Alternative zu bieten. Daraus ergaben sich zahlreiche Schwachstellen von Webanwendungen. Aber bis heute scheinen die Entwickler des Internet Explorer dies nicht zu bedauern und haben das Standardverhalten ihres Codes nicht geändert.

Die Safari-Entwickler erkannten dagegen das Risiko und versuchten, es zu reduzieren, ohne die Funktionalität zu beeinträchtigen – unterschätzten jedoch die Komplexität des Web. Die in ihrem Browser umgesetzte Lösung stützt sich auf ein zweites Indiz zusätzlich zum Vorhandensein einer plausibel aussehenden HTML-Auszeichnung im Dokumenttext. Das Vorkommen einer Erweiterung wie

1. Die Logik im Internet Explorer unterscheidet sich geringfügig von dem Szenario, in dem der Header `Content-Type` komplett fehlt. Anstatt grundsätzlich HTML anzunehmen, untersucht der Browser die ersten 256 Bytes auf beliebte HTML-Tags und andere vordefinierte Inhaltssignaturen. Vom Sicherheitsstandpunkt aus ist der Unterschied jedoch nicht besonders groß.

`.html` oder `.xml` am Ende des URL-Pfades wird von ihrer Implementierung als Hinweis interpretiert, dass Content-Sniffing gefahrlos durchgeführt werden kann. Der Websitebetreiber würde die Datei doch sonst nicht so nennen, oder?

Leider ist das von ihnen gewählte Indiz so gut wie wertlos. Fast alle Webframeworks unterstützen nämlich mindestens eine Methode zur Codierung von Parametern im Pfadsegment der URL anstatt im herkömmlich verwendeten Query-Segment. Ein Mechanismus dieser Art heißt zum Beispiel beim Apache-Server `PATH_INFO`, und zufälligerweise ist er standardmäßig aktiviert. Durch Ausnutzen einer solchen Parameterübergabe kann der Angreifer normalerweise Datenschrott an den Pfad anhängen und dadurch den Browser verwirren, ohne die Art und Weise zu beeinträchtigen, wie der Server auf die gestellte Anforderung als solche reagiert.

Die beiden folgenden URLs veranschaulichen dies. Sie haben für Websites, die in Apache oder IIS ausgeführt werden, wahrscheinlich dieselbe Wirkung:

http://www.fuzzybunnies.com/get_file.php?id=1234

und

http://www.fuzzybunnies.com/get_file.php/evil.html?id=1234

In einigen weniger gängigen Webframeworks mag auch der folgende Ansatz funktionieren:

http://www.fuzzybunnies.com/get_file.php;evil.html?id=1234

13.1.3 Nicht erkannte Inhaltstypen

Trotz der offensichtlichen Schwierigkeiten mit `text/plain` haben die Entwickler, die am Internet Explorer arbeiten, beschlossen, die Heuristik ihres Browsers sogar noch auszuweiten. Internet Explorer wendet sowohl Content-Sniffing als auch den Abgleich von Dateinamenserweiterungen (Extension Matching)[2] nicht nur auf eine Handvoll allgemeiner MIME-Typen, sondern auf jeden Dokumenttyp an, der vom Browser nicht sofort erkannt wird. Zu dieser umfangreichen Kategorie kann alles von JSON (`application/json`) bis hin zu Multimediaformaten wie Ogg Vorbis (`audio/ogg`) gehören.

Ein solches Vorgehen ist natürlich problematisch und verursacht ernsthafte Probleme, wenn benutzergesteuerte Dokumentformate bereitgestellt werden, die nicht auf der Liste der intern im Browser registrierten, universell unterstützten

2. Der pfadbasierte Abgleich von Erweiterungen ist aus den im vorhergehenden Kapitel erörterten Gründen natürlich wertlos, aber im Fall des Internet Explorer 6 wird es noch schlimmer. In diesem Browser kann die Erweiterung im Query-String der URL stehen. Nichts hindert den Angreifer daran, an die angeforderte URL einfach `?foo=bar.html` anzuhängen, was sicherstellt, dass diese Prüfung immer positiv ausfällt und der Browser in so gut wie allem Angebotenen HTML erkennt und rendert.

MIME-Typen stehen, oder wenn sie gar an eine vom Anwender installierte Software – wie beispielsweise den Adobe Reader – delegiert werden.

Das Content-Sniffing des Internet Explorer endet hier aber noch nicht: Der Browser untersucht die Nutzlast auch, wenn es um intern erkannte Dokumentformate geht, die sich aus irgendeinem Grund nicht sauber parsen lassen. In Versionen vor dem Internet Explorer 8 kann die Bereitstellung einer von einem Angreifer bereitgestellten, aber nicht weiter geprüften Datei, die behauptet, ein JPEG-Bild zu sein, dazu führen, dass diese vom Browser als HTML behandelt wird. Und es wird noch lustiger: Selbst ein kleiner Fehler, zum Beispiel die Bereitstellung einer gültigen GIF-Datei mit `Content-Type: image/jpeg` löst denselben Codepfad aus. Was soll's, vor ein paar Jahren erkannte der Internet Explorer sogar HTML in jeder gültigen, ordnungsgemäß bereitgestellten PNG-Datei. Die dafür verantwortliche Logik wurde dankenswerterweise inzwischen deaktiviert – aber die verbliebenen Macken sind immer noch ein Minenfeld.

Hinweis

Um das Risiko des Content-Sniffing bei gültigen Bildern voll und ganz einschätzen zu können, sollten Sie Folgendes wissen: Es ist nicht besonders schwer, Bilder zu erstellen, die korrekt ausgewertet werden, aber in den rohen Bilddaten vom Angreifer ausgewählte ASCII-Zeichenfolgen – zum Beispiel HTML-Auszeichnungen – transportieren. Tatsächlich ist es erschreckend einfach, Bilder zu konstruieren, die beim Validieren, Filtern, Skalieren und erneuten Komprimieren mit einem bekannten deterministischen Algorithmus im damit erzeugten Binärstream aus heiterem Himmel eine beinahe beliebige Zeichenfolge sichtbar werden lassen. Kurz: Das serverseitige Absichern des Bildes führt zur Entfaltung des eigentlichen Schadcodes.

Zu seiner Entlastung hat Microsoft ab Internet Explorer 8 beschlossen, die meisten Arten von unbegründetem Content-Sniffing bei bekannten MIME-Typen in der Kategorie `image/*` nicht mehr zuzulassen. Auch das Feststellen von HTML in Bildformaten, die der Browser nicht erkannt hat, etwa `image/jp2` (JPEG2000), wurde aufgegeben (aber nicht die XML-Erkennung).

Abgesehen von dieser kleinen Verbesserung sträubt sich Microsoft, bedeutsame Änderungen an seiner Logik zur Ausspähung von Inhalten vorzunehmen, und seine Entwickler haben öffentlich die Notwendigkeit verteidigt, die Kompatibilität mit schlampig programmierten Websites zu erhalten [2]. Wahrscheinlich will Microsoft sich nicht den Zorn großer institutioneller Kunden zuziehen, von denen sich viele auf alte, schlecht geschriebene Intranet-Apps stützen und die von den Macken der Internet-Explorer-basierten Monokultur auf Clientseite abhängig sind.

Wegen der Reaktion, die der Internet Explorer auf seine Behandlung von `text/plain` erfuhr, bieten neuere Versionen zumindest einen teilweisen Workaround an: einen optionalen HTTP-Header, `X-Content-Type-Options: nosniff`,

mit dem sich Websitebetreiber gegen die Anwendung eines Großteils der umstrittenen Inhaltsheuristik entscheiden können. Die Verwendung dieses Headers ist dringend zu empfehlen. Leider wurde seine Unterstützung nicht rückwärts auf die Versionen 6 und 7 des Browsers portiert, und von anderen Browsern wird er nur eingeschränkt unterstützt. Anders ausgedrückt: Als einzige Verteidigung gegen Ausspähung von Inhalten reicht er nicht aus.

Hinweis

Denkanstoß: Nach den von SHODAN und Chris John Riley 2011 erhobenen Daten [3] haben nur 0,6 Prozent der 10 000 beliebtesten Websites im Internet diesen Header auf Site-Ebene verwendet.

13.1.4 Defensive Verwendung des Content-Disposition-Headers

Der in Teil I dieses Buches mehrfach erwähnte Header `Content-Disposition` kann in einigen Anwendungsfällen als Abwehrmaßnahme gegen Content-Sniffing angesehen werden. Seine Funktion wird in der Spezifikation von HTML/1.1 nicht zufriedenstellend erklärt; er wird lediglich in RFC 2183 [4] dokumentiert, und zwar nur in Bezug auf Mailanwendungen:

> *»Textteile können als ›Anhang‹ gekennzeichnet werden, um darauf hinzuweisen, dass sie nicht zum Haupttext der Mail gehören und nicht automatisch angezeigt werden sollen, sondern erst nach einer weiteren Benutzeraktion. Der MUA[3] kann dem Benutzer eines Bitmap-Terminals stattdessen eine Darstellung der Anhänge als Symbol anbieten bzw. auf Character-Terminals eine Liste der Anhänge, aus der der Benutzer einen Eintrag zur Darstellung oder Speicherung wählen kann.«*

Der HTTP-RFC gestattet die Verwendung von `Content-Disposition: attachment` im Webbereich, lässt sich jedoch nicht über die beabsichtigte Funktion aus.

Hinweis

Man beachte, dass HTML5 seinen eigenen Weg geht, Linkziele als Download anzubieten, anstatt den Browser zur Navigation und Darstellung des Dokuments zu veranlassen. Das `download`-Attribut für Links und ähnliche Elemente erfüllt ebendiesen Zweck. Es bleibt abzuwarten, ob die Verwendung von `Content-Disposition`-Headern aufgrund dieses einfacher zu wählenden Wegs zurückgeht – und somit für Verwundbarkeiten in älteren Browsern sorgt.

3. MUA steht für »Mail User Agent«, also eine Clientanwendung zum Abrufen, Anzeigen und Schreiben von Mails.

In der Praxis zeigen die meisten Browser, wenn sie während eines normalen Ladevorgangs auf diesen Header stoßen, ein Dialogfeld zum Herunterladen von Dateien an, das normalerweise drei Schaltflächen aufweist: ÖFFNEN, SPEICHERN und ABBRECHEN. Der Browser versucht nur dann, das Dokument weiter zu interpretieren, wenn man die Option ÖFFNEN wählt oder das Dokument auf Festplatte speichert und dann manuell öffnet. Bei der Option SPEICHERN wird ein optionaler `filename`-Parameter im Header auch als Vorschlag für den Namen der heruntergeladenen Datei verwendet. Fehlt dieses Feld, wird der Dateiname aus den bekanntermaßen unzuverlässigen Daten des URL-Pfades abgeleitet.

Da der Header die meisten Browser daran hindert, die zurückgegebenen Daten sofort zu interpretieren und anzuzeigen, eignet er sich besonders gut als gefahrlose Methode, um unbekannte Download-Dateien wie die bereits erwähnten Archive und ausführbaren Dateien zu hosten. Da er beim Laden von bestimmten Typen von Unterressourcen (zum Beispiel `<img>` oder `<script>`) ignoriert wird, kann er darüber hinaus eingesetzt werden, um benutzergesteuerte JSON-Antworten, Bilder usw. vor Content-Sniffing zu schützen. (Warum alle Implementierungen `Content-Disposition` für diese Arten der Navigation ignorieren, ist nicht besonders klar. Aber angesichts der Vorteile ist es das Beste, die Logik jetzt nicht infrage zu stellen.)

Als Beispiel für eine einigermaßen robuste Verwendung von `Content-Disposition` und anderen HTTP-Headern zum Verhindern von Content-Sniffing bei einer JSON-Antwort mag Folgendes dienen:

```
Content-Type: application/json; charset=utf-8
X-Content-Type-Options: nosniff
Content-Disposition: attachment; filename="json_response.txt"

{ "search_term": "<html><script>alert('Hi mom!')</script>", ... }
```

Nach Möglichkeit ist der defensive Einsatz von `Content-Disposition` dringend zu empfehlen, aber Sie müssen sich unbedingt darüber im Klaren sein, dass der Mechanismus weder für alle Mailprogramme funktioniert noch gut dokumentiert ist. In weniger verbreiteten Browsern wie Safari Mobile bewirkt der Header möglicherweise nichts; in gängigen wie Internet Explorer 6, Opera und Safari hat eine Reihe von `Content-Disposition`-Bugs den Header gelegentlich bei Angriffen wirkungslos gemacht. Zudem kursieren Tricks, mit denen via Iframes und JavaScript-Timeout-Funktionalität die `Content-Disposition`-Header außer Gefecht gesetzt werden können. Wie so oft sollten diese also nicht als »last line of defense« angesehen und verwendet werden.

Ein anderes Problem mit der Zuverlässigkeit von `Content-Disposition` liegt darin, dass der Benutzer nach wie vor geneigt sein kann, auf ÖFFNEN zu klicken. Von durchschnittlichen Usern ist nicht zu erwarten, dass sie bei der Anzeige von Flash-Applets oder HTML-Dokumenten nur deshalb vorsichtig sind, weil eine

Aufforderung zum Herunterladen dazwischenkommt. Bei den meisten Browsern versetzt die Auswahl von ÖFFNEN das Dokument in einen `file:`-Ursprung, d.h. in einen Zustand, der gleichbedeutend ist mit dem lokalen Öffnen der Datei. Oft lagert die Datei auch tatsächlich im TMP-Ordner des Betriebssystems oder Browsers und wird von dort geöffnet, was als solches schon problematisch sein kann (wobei die jüngsten Verbesserungen in Chrome sicher hilfreich sind). Opera verhält sich anders und zeigt das Dokument im Kontext der Herkunftsdomäne an. Der Internet Explorer trifft wohl die beste Wahl: HTML-Dokumente werden mithilfe eines sogenannten »Mark-of-the-Web«-Mechanismus (der in Kapitel 15 genauer vorgestellt wird) in einer besonderen Sandbox untergebracht. Aber selbst in diesem Browser profitieren Java- oder Flash-Applets nicht von diesem Merkmal.

13.1.5 Inhaltsdirektiven für Unterressourcen

Die meisten inhaltsbezogenen HTTP-Header, zum Beispiel `Content-Type`, `Content-Disposition` und `X-Content-Type-Options`, haben so gut wie keine Auswirkung auf das Laden typspezifischer Unterressourcen, etwa `<img>`, `<script>` oder `<embed>`. In diesen Fällen hat die einbettende Partei fast die vollständige Kontrolle darüber, wie der Browser die Antwort interpretiert.

`Content-Type` und `Content-Disposition` werden möglicherweise auch bei der Erledigung von Anforderungen, die von Plug-in-Code ausgelöst werden, nicht besonders beachtet. Denken Sie beispielsweise an Kapitel 9, wo gesagt wurde, dass alle Dokumente vom Typ `text/plain` oder `text/csv` von Adobe Flash als sicherheitsrelevante `crossdomain.xml`-Richtlinien interpretiert werden können, wenn im Stammverzeichnis auf dem Zielserver keine passende Metarichtlinie für die gesamte Site steht. Ob Sie es als Content-Sniffing oder lediglich als »Content-Type Blindness« (Blindheit gegenüber dem Inhaltstyp) bezeichnen wollen – das Problem ist auf jeden Fall sehr real.

Selbst wenn alle bisher erörterten HTTP-Header gewissenhaft eingesetzt werden, ist es wichtig, immer die Möglichkeit im Blick zu haben, dass eine beliebige andere Website den Browser dazu verleiten kann, die entsprechende Seite als einen problematischen Dokumenttyp zu interpretieren; dabei erfordern Applets, Appletbezogene Inhalte, PDF-Dokumente, Stylesheets und Skripte normalerweise besondere Aufmerksamkeit. Um das Fehlerrisiko zu minimieren, sollten Sie Struktur und Zeichensatz aller bereitgestellten Payloads sorgfältig einschränken oder zur Isolierung von Dokumenten, die sich nicht besonders gut einschränken lassen, »Sandbox«-Domänen benutzen.

13.1.6 Downloads und andere Nicht-HTTP-Inhalte

Das Verhalten von HTTP-Headern wie `Content-Type`, `Content-Disposition` und `X-Content-Type-Options` kann schlecht nachvollziehbar und voller Ausnahmen sein, aber sie bilden zumindest ein einigermaßen konsistentes Ganzes. Trotzdem kann man leicht vergessen, dass die in diesen Headern enthaltenen Metadaten in vielen echten Fällen einfach nicht verfügbar sind – und dann ist alles möglich. Der Umgang mit Dokumenten, die über FTP abgerufen oder auf der Festplatte gespeichert und mit dem Protokoll `file:` geöffnet wurden, ist in hohem Maße browser- und protokollspezifisch und überrascht häufig selbst die erfahrensten Sicherheitsfachleute.

Beim Öffnen lokaler Dateien nutzen Browser am liebsten die Informationen, die über die Dateiendung bereitgestellt werden, und wenn es sich dabei um einen der hart codierten Werte handelt, die der Browser kennt, etwa `.txt` oder `.html`, übernehmen die meisten Browser diese Hinweise. Eine Ausnahme bildet Chrome; er versucht selbst, bestimmte »passive« Dokumenttypen zu erkennen, zum Beispiel JPEG, sogar innerhalb von `.txt`-Dokumenten. (HTML ist jedoch streng verboten.)

Geht es um andere Erweiterungen, die für externe Programme registriert sind, ist das Verhalten etwas weniger vorhersehbar. Der Internet Explorer ruft üblicherweise die externe Anwendung auf. Die meisten anderen Browser setzen aber auf Content-Sniffing und verhalten sich, als würden sie das Dokument über HTTP laden und der Header `Content-Type`-Header nicht gesetzt wäre – ist er ja lokal auch nicht. Außerdem greifen alle Browser auf Content-Sniffing zurück, wenn die Erweiterung nicht bekannt ist (etwa `.foo`).

Das starke Vertrauen auf Dateinamensendungen und Content-Sniffing für `file:`-Dokumente bildet einen interessanten Kontrast zum normalen Umgang mit Ressourcen aus dem Internet. Im Web ist `Content-Type` im Großen und Ganzen die maßgebliche Beschreibung von Dokumenttypen. Dateierweiterungsdaten werden weitgehend ignoriert, und es ist vollkommen in Ordnung, eine funktionierende JPEG-Datei an einem Ort wie *http://fuzzybunnies.com/gotcha.txt* zu hosten. Was geschieht jedoch, wenn dieses Dokument auf die Festplatte heruntergeladen wird? Nun, dann ändert sich unerwartet die tatsächliche Bedeutung der Ressource. Erfolgt der Zugriff darauf über das Protokoll `file:`, besteht der Browser darauf, sie aufgrund der Dateinamenserweiterung als Textdatei wiederzugeben.

Das gezeigte Beispiel ist recht harmlos, aber andere Vektoren zum Einschmuggeln von Inhalten – etwa wenn ein Bild zur ausführbaren Datei wird – können mehr Ärger verursachen. Zu diesem Zweck versuchen der Internet Explorer und Opera bei einer Reihe bekannter `Content-Type`-Werte, die Erweiterung passend zum MIME-Typ zu ändern. Andere Browser bieten dieses Maß an Schutz nicht und lassen sich möglicherweise von der Situation, in der sie sich befinden, gründlich verwirren. Abbildung 13–1 hält Firefox in einem solchen peinlichen Augenblick fest.

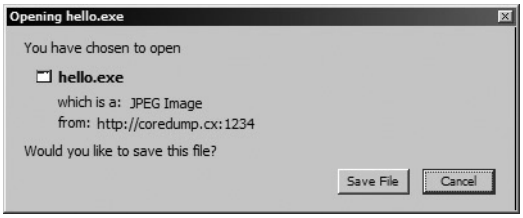

Abb. 13–1 *Diese Eingabeaufforderung zeigt Firefox beim Versuch an, ein Dokument mit dem Header* `Content-Type: image/jpeg` *zu speichern, das mit einem Header* `Content-Disposition: attachment` *bereitgestellt wird. Den Dateinamen »hello.exe« hat der Browser aus dem an sich bedeutungslosen Suffix* `PATH_INFO` *abgeleitet, das der Angreifer an die URL angehängt hat. Die Eingabeaufforderung behauptet zu Unrecht, die* `.exe`*-Datei sei ein »JPEG-Bild«. Wird sie auf der Platte gespeichert, ist sie eine ausführbare Datei.*

Dieses Problem unterstreicht die Wichtigkeit, einen expliziten, harmlosen Wert für `filename` zurückzugeben, sobald ein `Content-Disposition`-Anhang verwendet wird. Damit kann man ein Opfer effektiv davor schützen, ein Dokumentformat herunterzuladen, das der Eigentümer der Site niemals bereitstellen wollte.

Angesichts der komplexen Logik, die für `file:`-URLs verwendet wird, mag die Einfachheit des Umgangs mit `ftp:` überraschend sein. Beim Dokumentzugriff über FTP kümmern sich die meisten Browser nämlich nicht besonders um Dateierweiterungen, sondern schwelgen in ungezügeltem Content-Sniffing. (Eine Ausnahme bildet Opera; dort haben die Dateinamenserweiterungen immer noch Vorrang.) Aus Entwicklersicht mag diese Behandlung von Dokumenten via FTP logisch erscheinen, denn das FTP-Protokoll kann man als ungefähr gleichwertig mit HTTP/0.9 einstufen. Dennoch verletzt dieses Vorgehen das Prinzip der geringsten Überraschung (Principle of Least Astonishment): Die Serverbetreiber gehen sicherlich nicht davon aus, dass sie durch die Genehmigung, `.txt`-Dateien auf eine FTP-Site hochzuladen, automatisch auch dem Hosting aktiver HTML-Inhalte in ihrer Domäne zustimmen.

13.2 Zeichensätze erkennen

Das Erkennen des Dokumenttyps gehört zu den wichtigeren Aspekten des komplexen Roulettespiels bei der Verarbeitung von Webcontent, aber es ist sicher nicht der einzige. Für alle Typen textbasierter Dateien, die der Browser darstellt, muss eine weitere Festlegung erfolgen: Die richtige Zeichensatztransformation muss ermittelt und auf den Eingabestream angewendet werden. Die vom Browser normalerweise angenommene Codierung der Ausgabe ist UTF-8 oder UTF-16. Die Eingabe obliegt dagegen dem Autor der Seite.

Im einfachsten Fall wird die passende Codierungsmethode vom Server im Parameter `charset` des `Content-Type`-Headers angegeben. Bei HTML-Dokumenten können diese Informationen in gewissem Maße auch durch die Direktive

<meta> erfolgen. (Der Browser versucht, diese Direktive spekulativ zu extrahieren und zu interpretieren, bevor er das Dokument tatsächlich zerlegt.)

Unglücklicherweise machen die gefährlichen Eigenschaften gewisser Zeichencodierungen sowie die Aktionen des Browsers beim Fehlen oder Nichterkennen des Parameters charset das Leben wieder einmal erheblich interessanter, als die erwähnte einfache Regel es täte. Um zu verstehen, was falsch laufen kann, müssen wir zunächst drei besondere Klassen von Zeichensätzen betrachten, die die Semantik von HTML- oder XML-Dokumenten ändern können:

- **Zeichensätze, die eine unvorschriftsmäßige Darstellung standardmäßiger 7-Bit-ASCII-Codes zulassen:**
  Solche Zeichenfolgen können eingesetzt werden, um HTML-Syntaxelemente, etwa spitze Klammern oder Anführungszeichen, so zu codieren, dass sie eine einfache serverseitige Prüfung überstehen. Die bekanntermaßen problematische Codierung UTF-7 erlaubt zum Beispiel, das Zeichen »<« als die fünfstellige Folge »+ADw-« zu codieren, eine Zeichenfolge, die die meisten Filter auf Serverseite ohne Bedenken so übernehmen. Ebenso verbietet die Spezifikation UTF-8 zwar formal die Darstellung von »<« durch unnötig umfangreiche Sequenzen von 2 bis 5 Bytes (von 0xC0 0xBC bis zu 0xFC 0x80 0x80 0x80 0x80 0xBC), lässt sie aber technisch zu.[4]
- **Variable-Length-Codierungen, bei denen ein oder mehrere Bytes mit einem besonderen Präfix eine spezielle Bedeutung haben:**
  Solche Logik kann dazu führen, dass zulässige HTML-Syntaxelemente als Bestandteil eines unbeabsichtigten Multi-Byte-Literals »verbraucht« werden. Bei der Shift-JIS-Codierung kann der Präfixcode 0xE0 zum Beispiel dazu führen, dass eine folgende spitze Klammer oder ein Anführungszeichen in Internet Explorer, Firefox und Opera (aber nicht in Chrome) verschwindet, was die Bedeutung der zeileninternen Auszeichnung möglicherweise erheblich ändert. Hierbei handelt es sich eigentlich um ein »Feature« des Zeichensatzes. Die Browser, die Zeichen auf diese Weise verschlucken, verhalten sich weitestgehend korrekt.

  Auch das entgegengesetzte Problem kann auftreten: Der Server kann überzeugt sein, dass er ein mehrere Bytes umfassendes Literal ausgibt, aber dieses kann vom Browser zurückgewiesen und als mehrere Einzelzeichen interpretiert werden. In EUC-KR wird das Präfix 0x8E nur dann berücksichtigt, wenn das folgende Zeichen einen ASCII-Code von 0x41 oder höher aufweist. Liegt er niedriger, tritt die erwartete Wirkung nicht ein, was aber möglicherweise nicht alle serverseitigen Implementierungen bemerken.

4. Heute ist das Problem bei den meisten Browsern weniger schwerwiegend. Ihre Parser umfassen jetzt zusätzliche Prüfungen, um überlange UTF-8-Codierungen prinzipiell abzulehnen. Dasselbe lässt sich allerdings nicht von allen möglichen UTF-8-Bibliotheken auf Serverseite sagen.

- **Codierungen, die mit 8-Bit-ASCII völlig inkompatibel sind:** Diese Fälle führen lediglich zu einer anderen Darstellung der Dokumentstruktur zwischen Client und Server. UTF-16 und UTF-32 sind gängige Beispiele.

Unter dem Strich ergibt sich Folgendes: Hat der Server nicht die vollständige Kontrolle über den Zeichensatz, den er generiert, und ist er nicht sicher, dass der Client keine unerwartete Transformation der Daten vornimmt, dann kann es zu ernsten Komplikationen kommen. Nehmen Sie beispielsweise eine Webanwendung, die im folgenden HTML-Code die spitzen Klammern aus der hervorgehobenen benutzergesteuerten Zeichenfolge entfernt:

```
You are currently viewing:
<span class="blog_title">
+ADw-script+AD4-alert("Hi mom!")+ADw-/script+AD4-
</span>
```

Interpretiert der Empfänger dieses Dokument als UTF-7, sieht das zerlegte Markup wie folgt aus:

```
You are currently viewing:
<span class="blog_title">
<script>alert("Hi mom!")</script>
</span>
```

Ein ähnliches Problem, das dieses Mal mit dem erwähnten »Verschlucken« von Zeichen in der Shift-JIS-Codierung zu tun hat, findet sich im folgenden Code. Ein mehrere Bytes langes Präfix darf ein schließendes Anführungszeichen verschwinden lassen. Infolgedessen wird das zugehörige HTML-Tag nicht erwartungsgemäß geschlossen, was dem Angreifer die Möglichkeit gibt, einen zusätzlichen `onerror`-Handler in das Markup einzufügen:

```
<img src="http://fuzzybunnies.com/[0xE0]">
... dies gehört noch zum Markup ...
... das weiß der Server aber nicht ...
" onerror="alert('Dies wird ausgeführt!')"
<div>
...der Seiteninhalt geht hier weiter...
</div>
```

Es ist einfach unverzichtbar, die automatische Zeichensatzerkennung für sämtliche textbasierten Dokumente zu verhindern, die benutzergesteuerte Daten irgendeiner Art enthalten. Die meisten Browser befassen sich dann mit Zeichensatzerkennung, wenn sie weder im `Content-Type`-Header noch im `<meta>`-Tag den Parameter `charset` finden. Es gibt einige deutliche Unterschiede zwischen den Implementierungen (beispielsweise ist nur der Internet Explorer wild darauf, UTF-7 zu erkennen), aber Sie sollten niemals davon ausgehen, dass das Ergebnis der Zeichensatzerkennung zuverlässig ist.

Hinweis

In älteren Firefox-Versionen gab es für Angreifer viele Möglichkeiten, krude Zeichensätze zum Verschleiern von XSS-Vektoren zu verwenden. Unter anderem konnte man auf zweifelhafte Zeichensätze via `x-imap-modified-utf7` zurückgreifen und Markup wie dieses einschleusen:

```
<meta charset="x-imap4-modified-utf7">&<script&S1&TS&1>
alert&A7&(1)&R&UA;&&<&A9&11/script&X&>
```

Eine automatische Zeichensatzerkennung wird auch dann versucht, wenn der Zeichensatz nicht erkannt wird oder falsch geschrieben ist; dieses Problem tritt zusammen mit dem Umstand auf, dass die Benennung von Zeichensätzen mehrdeutig sein kann und dass Webbrowser bei der Tolerierung von gängigen Namensvarianten inkonsistent sind. Ein Beispiel dafür: Der Internet Explorer akzeptiert sowohl »ISO-8859-2« als auch »ISO8859-2« (ohne Bindestrich hinter ISO) als gültige Zeichensatzbezeichner im Header `Content-Type`; »UTF8« dagegen erkennt er nicht als Alias für »UTF-8«. Die falsche Wahl kann erhebliche Kopfschmerzen verursachen.

Hinweis

Nebenbei: Der Header `X-Content-Type-Options` hat keinen Einfluss auf die Logik der Zeichensatzerkennung.

13.2.1 BOM: Kennzeichnung der Bytereihenfolge

Wir sind noch nicht ganz fertig mit der Zeichensatzerkennung! Der Internet Explorer zeichnet sich als einziger Browser durch eine weitere dramatisch fehlgeleitete Praxis bei der Behandlung von Inhalten aus. Dabei geht es um das Kennzeichen für die Bytereihenfolge (Byte Order Mark, BOM), eine Bytesequenz, die am Anfang einer Datei stehen kann, um deren Codierung anzugeben. Der Internet Explorer hat hier die Tendenz, dem BOM Vorrang gegenüber den explizit bereitgestellten `charset`-Daten zu gewähren. Wird am Beginn der eingelesenen Datei eine solche Zeichenkette erkannt, ignoriert der Browser den deklarierten Zeichensatz. Übrigens: Laut der aktuellen Version der HTML5-Spezifikation könnte dies zudem bald für alle Browser der Fall sein.

Tabelle 13–1 zeigt einige übliche Kennzeichnungen. Von diesen ist der ausdruckbare UTF-7-BOM besonders hinterhältig.

Name der Codierung	BOM-Sequenz
UTF-7	"+/v" gefolgt von "8", "9", "+", oder "/"
UTF-8	0xEF 0xBB 0xBF
UTF-16 Little Endian	0xFF 0xFE
UTF-16 Big Endian	0xFE 0xFF
UTF-32 Little Endian	0xFF 0xFE 0x00 0x00
UTF-32 Big Endian	0x00 0x00 0xFE 0xFF
GB -18030	0x84 0x31 0x95 0x33

Tab. 13–1 *Übliche Markierungen für die Bytereihenfolge (BOMs)*

Hinweis

Die Entwickler von Microsoft geben zu, dass es bei dieser Konstruktion ein Problem gibt, und erklären derzeit, dass die Logik abhängig vom Ergebnis von Kompatibilitätstests möglicherweise überarbeitet wird. Sollte das Problem behoben sein, wenn dieses Buch in den Handel kommt, Hut ab! Bis dahin könnte es ungünstig sein, dem Angreifer die Kontrolle über die ersten Bytes einer HTTP-Antwort zu überlassen, die nicht anderweitig durch `Content-Disposition` geschützt ist – und es gibt keine andere Möglichkeit, diese Macke zu umgehen, als die Antwort aufzufüllen.

13.2.2 Vererbung und Überschreiben von Zeichensätzen

Wenn wir die potenziellen Auswirkungen der Strategien zur Zeichensatzbehandlung in aktuellen Webbrowsern bewerten wollen, dann sollten wir zwei weitere, wenig bekannte Mechanismen berücksichtigen. Beide Features können es einem Angreifer ermöglichen, unerwünschte Zeichencodierungen auf einer anderen Seite zu erzwingen, ohne dass er die Zeichensatzerkennung missbrauchen müsste.

Der erste heißt *Zeichensatzvererbung* (Character Set Inheritance) und wird von allen Browsern außer dem Internet Explorer unterstützt. Bei diesem Verfahren kann jede für den obersten Frame definierte Codierung automatisch auf alle Dokumente in Frames übertragen werden, die nicht über einen eigenen `charset`-Wert verfügen. Zunächst wurde diese Art von Vererbung auf alle Frameszenarios ausgeweitet, selbst über Websites, die überhaupt nichts damit zu tun haben. Als Stefan Esser, Abhishek Arya und einige andere Forscher jedoch eine Reihe plausibler Angriffe vorführten, die dieses Merkmal nutzten, um ein UTF-7-Parsing auf arglosen Zielen zu erzwingen, beschlossen die Entwickler von Firefox und WebKit, das Verhalten auf Frames desselben Ursprung zu beschränken. (Opera lässt immer noch domänenübergreifende Vererbung zu. Obwohl es UTF-7 nicht unterstützt, sind andere problematische Codierungen, etwa Shift JIS, Freiwild.) Gareth Heyes publizierte wenig später eine weitere Reihe von Angriffen, die die Verwendung vom BOM in JSON und CSS beinhalteten und teils heute noch funktionieren[5].

Der andere erwähnenswerte Mechanismus ist die Möglichkeit, den aktuell verwendeten Zeichensatz manuell zu überschreiben. Diese Funktion ist in den meisten Browsern über das Menü ANSICHT|ZEICHENKODIERUNG oder ähnlich erreichbar. Wird der Zeichensatz mithilfe dieses Menüs geändert, dann wird die Seite mit sämtlichen untergeordneten Frames (einschließlich domänenübergreifender!) mithilfe der gewählten Codierung neu analysiert, ohne Rücksicht auf etwaige zuvor für den betreffenden Inhalt vorgefundene `charset`-Direktiven.

Da sich Benutzer leicht dazu verleiten lassen, für eine angreifergesteuerte Seite einen anderen Zeichensatz zu wählen (einfach, um sie korrekt zu sehen), sollte Ihnen dieser Mechanismus Unbehagen verursachen. Durchschnittliche User sind kaum in der Lage, zu erkennen, dass ihre Auswahl auch für verborgene `<iframe>`-Tags gilt und dass eine solch ungefährlich erscheinende Aktion XSS-Angriffe auf beliebige andere Webseiten ermöglichen kann. Bleiben wir lieber realistisch: Die meisten wissen nicht, was ein `<iframe>` ist – und sollten es auch nicht wissen müssen.

13.2.3 Markup-gesteuerte Zeichensätze für Unterressourcen

Wir nähern uns dem Ende unserer epischen Reise durch die Eigenarten der Inhaltsbehandlung, sind aber noch nicht ganz fertig. Aufmerksame Leser erinnern sich vielleicht, dass ich in Abschnitt 4.5.4 »Typspezifisches Einbinden von Inhalten« erwähnt habe, dass die einbettende Seite für bestimmte Arten von Unterressourcen (insbesondere Stylesheets und Skripte) einen eigenen `charset`-Wert festlegen kann. Damit kann sie bestimmte Umformungen, wie zum Beispiel die folgende, auf das abgerufene Dokument anwenden:

```
<script src="http://fuzzybunnies.com/get_js_data.php" charset="EUC-JP">
```

Dieser Parameter wird von allen Browsern außer Opera akzeptiert. Wo er unterstützt wird, hat er normalerweise keinen Vorrang vor `charset` in `Content-Type`, es sei denn, der Wert für `charset` fehlt oder ist für den Browser nicht verständlich. Aber zu jeder Regel gibt es eine Ausnahme, und allzu häufig heißt diese Internet Explorer 6. Bei diesem immer noch beliebten Browser überschreibt die vom Markup festgelegte Codierung die HTTP-Daten.

Spielt dieses Verhalten in der Praxis eine Rolle? Kehren wir kurz zu Kapitel 6 zurück, um die Konsequenzen vollständig zu verstehen. Dort hatten wir uns ausführlich damit beschäftigt, wie man servergenerierten, benutzerspezifischen, JSON-ähnlichen Code davor bewahren kann, dass er in potenziell bösartigen Domänen eingebunden wird. Ein Beispiel für eine Anwendung, die einen solchen Schutz benötigt, ist ein durchsuchbares Adressbuch in einer Webmailanwendung:

---

5. http://www.thespanner.co.uk/2009/02/24/inline-utf-7-e4x-javascript-hijacking/

Dabei wird der Suchbegriff in der URL bereitgestellt und eine JavaScript-Serialisierung der passenden Kontakte an den Browser zurückgegeben, wo sie jedoch gegen die Einbindung in nicht dazugehörige Sites abgeschirmt werden muss.

Nehmen wir nun an, der Entwickler habe einen einfachen Trick benutzt, um zu verhindern, dass Seiten Dritter diese Daten mithilfe von `<script src=...>` laden. Es bedarf lediglich des Präfixes »//«, um die gesamte Antwort in einen Kommentar umzuwandeln. Wer mit der API `XMLHttpRequest` die Daten vom gleichen Ursprung holt, kann die Antwort natürlich einfach untersuchen, das Präfix entfernen und die Daten an `eval(...)` übergeben – aber Requests, die von anderen Domänen ausgehen und versuchen, die Syntax `<script src=...>` zu missbrauchen, haben kein Glück.

In dieser Form kann die Anforderung `/contact_search.php?q=smith` folgende Antwort liefern:

```
// var result = { "q": "smith", "r": [ "j.smith@example.com" ] };
```

Solange der Suchbegriff ordnungsgemäß mit Escape-Sequenzen versehen oder gefiltert wird, scheint dieses Schema sicher zu sein. Wird uns jedoch klar, dass der Angreifer die Interpretation der Antwort als UTF-7 erzwingen kann, dann ändert sich das Bild dramatisch. Ein anscheinend harmloser Suchbegriff, der, soweit es den Server betrifft, keine unzulässigen Zeichen enthält, kann sich dennoch unerwartet wie folgt decodieren lassen:

```
// var result = { "q": "smith[CR][LF]
var gotcha = { "", "r": [ "j.smith@example.com" ] };
```

Wird diese Antwort mithilfe von `<script src=... charset=utf-7>` in den Browser des Opfers geladen, bekommt der Angreifer Zugriff auf einen Teil des Benutzeradressbuches.

Hierbei handelt es sich nicht einfach um ein ausgedachtes Beispiel. Der »//«-Ansatz wird im Web recht häufig verwendet, und Masato Kinugawa, ein bekannter Sicherheitsforscher, stellte fest, dass mehrere beliebte Webanwendungen von diesem Fehler betroffen sind. Eine ausgefeiltere Variante dieses Angriffs kann sich auch gegen andere Präfixe richten, die die Ausführung verhindern sollen, zum Beispiel `while(1);`. Letzten Endes ist das Problem des domänenübergreifenden Überschreibens von `charset` in `<script>`-Tags einer der Gründe, weshalb wir in Kapitel 6 dringend die Verwendung eines stabilen, den Parser stoppenden Präfixes empfohlen haben, um zu verhindern, dass der Interpreter überhaupt vom Angreifer gesteuerte Bits zu sehen bekommt. Ach ja – und wenn Sie die Unterstützung von E4X einbeziehen, wird das Ganze noch interessanter [5]. Aber dabei wollen wir es jetzt belassen.

13.2.4 Nicht-HTTP-Dateien erkennen

Sehen wir uns das letzte fehlende Detail an, um das Kapitel abzurunden: das Erkennen der Zeichensatzcodierung für Dokumente, die über andere Protokolle als HTTP transportiert werden. Wie zu erwarten ist, werden Dokumente, die auf der Festplatte gespeichert und anschließend mithilfe des `file:`-Protokolls geöffnet oder mit anderen Mitteln geladen werden, bei denen die üblichen `Content-Type`-Metadaten fehlen, normalerweise der Logik zur Zeichensatzerkennung unterworfen.

Anders als bei der Heuristik zur Dokumenttyperkennung gibt es jedoch keinen wesentlichen Unterschied zwischen den möglichen Übermittlungsmethoden. In allen Fällen ist das Sniffing-Verhalten in etwa dasselbe.

Es gibt keine saubere, portierbare Methode, dieses Problem für sämtliche textbasierten Dokumente zu lösen, aber speziell für HTML lässt sich der Einfluss der Zeichensatzerkennung durch Einbetten einer `<meta>`-Direktive in den Dokumenttext abmildern:

```
<meta http-equiv="Content-Type" content="text/html;charset=...">
```

Sie sollten jedoch nicht zugunsten dieses Indikators auf `Content-Type` verzichten. Anders als `<meta>` funktioniert der Header für Nicht-HTML-Inhalte, und er lässt sich leichter für die gesamte Site erzwingen und überwachen. Im Gegensatz dazu profitieren Dokumente, die wahrscheinlich auf Festplatte gespeichert werden und vom Angreifer gesteuerte Häppchen enthalten, von einem redundanten `<meta>`-Tag. (Sorgen Sie aber dafür, dass dieser Wert tatsächlich mit `Content-Type` übereinstimmt.)

Spickzettel für Webentwickler

Gute Sicherheitspraktiken für alle Websites

- ☑ Weisen Sie den Webserver an, den Header `X-Content-Options: nosniff` an alle HTTP-Antworten anzuhängen.
- ☑ Informieren Sie sich anhand des Spickzettels in Kapitel 9 über die Einrichtung einer geeigneten `/crossdomain.xml`-Metarichtlinie.
- ☑ Konfigurieren Sie den Server so, dass er Standardwerte für `charset` und `Content-Type` an alle Antworten anhängt, die sonst keine hätten.
- ☑ Wenn Sie keine pfadbasierte Parameterübergabe einsetzen (etwa `PATH_INFO`), sollten Sie erwägen, dieses Merkmal zu deaktivieren.

Wenn Sie Dokumente mit teilweise angreifergesteuerten Inhalten generieren

- ☑ Geben Sie immer einen expliziten, gültigen und wohlbekannten Wert für `Content-Type` zurück. Verwenden Sie nicht `text/plain` oder `application/octet-stream`.
- ☑ Geben Sie für alle textbasierten Dokumente im `Content-Type`-Header einen expliziten, gültigen und wohlbekannten Wert an; UTF-8 ist allen anderen Codierungen mit variabler Länge vorzuziehen. Gehen Sie nicht davon aus, dass `application/xml+svg`, `text/csv` und andere Nicht-HTML-Dokumente keinen festgelegten Zeichensatz brauchen. Ziehen Sie für HTML eine redundante `<meta>`-Direktive in Betracht, wenn absehbar ist, dass die Datei vom Benutzer heruntergeladen werden kann. Achten Sie auf Schreibfehler – UTF8 ist kein gültiges Alias für UTF-8.
- ☑ Verwenden Sie für Antworten, die nicht direkt angezeigt werden müssen, – auch für JSON-Daten – `Content-Disposition: attachment` sowie einen geeigneten, expliziten Wert für `filename`.
- ☑ Sorgen Sie dafür, dass der Benutzer die ersten Bytes der Datei nicht beeinflussen kann, und beschränken Sie die vom Nutzer kontrollierten Daten so weit wie möglich. Liefern Sie Daten mit Nullbytes, Steuerzeichen oder High-Bit-Werten nur dann aus, wenn es absolut notwendig ist.
- ☑ Achten Sie bei serverseitigen Umwandlungen der Codierung darauf, dass die Konverter alle unerwarteten oder ungültigen Eingaben zurückweisen (zum Beispiel überlange oder ungültige UTF-8-Zeichen).

Wenn Sie vom Benutzer generierte Dateien hosten

Ziehen Sie wenn möglich die Verwendung einer Sandbox-Domäne in Betracht. Wenn Sie vorhaben, nicht eingeschränkte oder unbekannte Dateiformate bereitzustellen, ist eine Sandbox-Domäne unverzichtbar. Das absolute Minimum sind folgende Maßnahmen:

- ☑ Benutzen Sie `Content-Disposition: attachment` sowie einen geeigneten, expliziten Wert für `filename`, der mit dem Parameter `Content-Type` übereinstimmt.
- ☑ Prüfen Sie die Gültigkeit der Eingabedaten sorgfältig und verwenden Sie immer den passenden, allgemein anerkannten MIME-Typ. Das Bereitstellen von JPEG als `image/gif` kann zu Problemen führen. Nehmen Sie Abstand davon, MIME-Typen bereitzustellen, die von beliebten Browsern wahrscheinlich nicht unterstützt werden.
- ☑ Verzichten Sie auf `Content-Type: application/octet-stream` und benutzen Sie stattdessen `application/binary`, insbesondere bei unbekannten Dokumenttypen. Geben Sie nicht `Content-Type:text/plain` zurück. Lassen Sie keine vom Benutzer festgelegten `Content-Type`-Header zu.
- ☑ Sichere Uploads sind sehr schwer umzusetzen. Sorgen Sie am besten für Speicherplatz auf einer anderen Domäne, durchsuchen Sie den Dateiinhalt nach verdächtigen XSS- und RCE-Signaturen, stellen Sie sicher, dass MIME-Typ und Dateiendung auch wirklich zusammenpassen, und beachten Sie, dass auch der Dateiname als Angriffsvektor herhalten kann: `<img src=x onerror=alert(1)>.gif` ist für viele Betriebssysteme ein valider Dateiname.