2 Am Anfang war die URL

Das auffälligste Kennzeichen des Web ist ein einfacher Textstring, die sogenannte *URL* (*Uniform Resource Locator*). Jede wohlgeformte, vollqualifizierte URL dient als gültige Adresse und identifiziert eindeutig eine einzelne Ressource auf einem Server im Netzwerk (wodurch sie gleichzeitig eine Reihe damit verbundener Hilfsfunktionen umsetzt). Die URL-Syntax ist der Dreh- und Angelpunkt der Adressleiste, der wichtigsten Sicherheitsanzeige in der Benutzerschnittstelle eines Browsers.

Neben echten URLs für den Abruf von Inhalten gibt es auch mehrere Klassen von *Pseudo-URLs* mit einer ähnlichen Syntax. Sie geben bequemen Zugriff auf Browserfunktionen wie die integrierte Skript-Engine, mehrere Sondermodi zur Dokumentdarstellung usw. Es sollte nicht überraschen, dass solche Pseudo-URL-Aktionen die Sicherheit einer Website, die mit ihnen verlinkt ist, erheblich beeinträchtigen können.

Herauszufinden, wie der Browser eine bestimmte URL interpretiert und welche Nebenwirkungen dies hat, gehört zu den grundlegendsten und häufigsten Aufgaben, die Menschen und Webanwendungen durchführen, um die Sicherheit zu erhöhen. Allerdings kann dies eine schwierige Aufgabe sein. Die von Tim Berners-Lee erarbeitete allgemeine URL-Syntax ist hauptsächlich im RFC 3986 dokumentiert und spezifiziert [1]. Die praktischen Verwendungsmöglichkeiten im Web werden in den RFCs 1738 [2], 2616 [3] und einer Reihe weiterer, jedoch weniger bedeutender Standards umrissen. Diese Dokumente sind bemerkenswert ausführlich, was zu einem ziemlich komplizierten Interpretationsmodell führt. Sie sind aber nicht genau genug, um eine harmonische, kompatible Implementierung in jeglicher Clientsoftware sicherzustellen. Außerdem weichen einige Softwarehersteller aus jeweils eigenen Gründen von den Spezifikationen ab.

Sehen wir uns nun genauer an, wie eine einfache URL in der Praxis funktioniert.

2.1 Struktur einer URL

Abbildung 2–1 zeigt das Format einer *vollqualifizierten, absoluten URL*, also einer URL, die sämtliche erforderlichen Informationen für den Zugriff auf eine bestimmte Ressource enthält und in keiner Weise vom Ausgangspunkt der Navigation abhängt. Im Gegensatz dazu unterschlägt eine *relative URL* wie `../file.php?text=hello+world` einen Teil der Informationen und muss daher im Zusammenhang mit der Basis-URL für den aktuellen Browserkontext interpretiert werden.

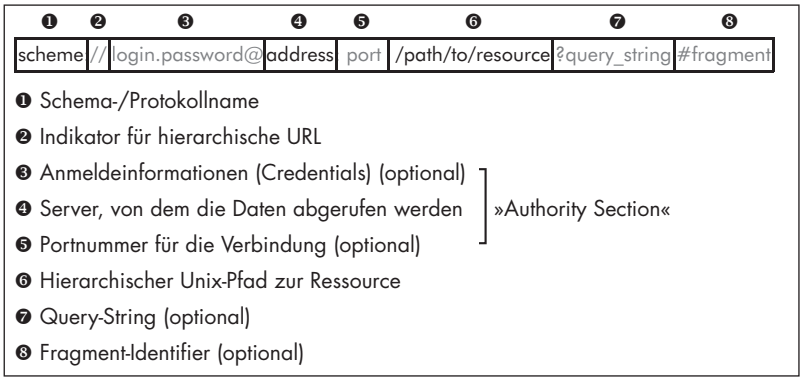


Abb. 2–1 *Struktur einer absoluten URL*

Die einzelnen Bestandteile einer absoluten URL scheinen auf den ersten Blick verständlich zu sein, weisen aber alle gewisse Fallstricke auf. Deshalb sehen wir sie uns im Folgenden genauer an.

2.1.1 Schema und Protokoll

Der *Schema- bzw. Protokollname* ist ein String, bei dem nicht zwischen Groß- und Kleinschreibung unterschieden wird und der mit einem Doppelpunkt abschließt. Er gibt das Protokoll an, das zum Abrufen der Ressource verwendet wird. Die offizielle Registrierung gültiger URL-Schemas unterliegt der *IANA* (*Internet Assigned Numbers Authority*), einer Organisation, die vor allem für die Verwaltung des IP-Adressraums bekannt ist [4]. Die derzeitige Liste aktueller Schemanamen der IANA enthält mehrere Dutzend Einträge, darunter *http:*, *https:* und *ftp:*, doch in der Praxis können die üblichen Browser und Drittanbieteranwendungen eine Vielzahl weiterer Schemas erkennen, die ohne formale Registrierung auskommen. (Von besonderem Interesse sind dabei mehrere Arten von Pseudo-URLs wie *data:* und *javascript:*, die wir weiter hinten in diesem Kapitel und noch an anderen Stellen in diesem Buch besprechen werden.)

Bevor Browser und Webanwendungen irgendeine weiter gehende Analyse durchführen können, müssen sie zunächst bestimmen, ob eine vollqualifizierte absolute URL oder eine relative vorliegt. Die Angabe eines gültigen Schemanamens am Anfang der Adresse ist laut RFC 1738 der Hauptunterschied: In einer konformen absoluten URL dürfen vor dem erforderlichen Doppelpunkt nur alphanumerische Zeichen sowie die Zeichen »+«, »-« und ».« auftreten. In der Praxis jedoch weichen die Browser von dieser Richtlinie ab. Alle ignorieren führende Zeilenumbrüche und Leerzeichen. Der Internet Explorer schenkt dem gesamten Bereich nicht druckbarer Zeichen mit den ASCII-Codes 0x01 bis 0x0F keine Beachtung, und Chrome kümmert sich auch nicht um 0x00, das `NUL`-Zeichen. Die meisten Implementierungen ignorieren auch Zeilenumbrüche und Tabulatoren mitten im Schemanamen, während Opera Zeichen mit High-Bit-Werten im String akzeptiert.

Aufgrund dieser Inkompatibilitäten müssen Anwendungen, bei denen zwischen absoluten und relativen URLs zu unterscheiden ist, sehr vorsichtig vorgehen und jegliche nicht konforme Syntax ablehnen. Wie wir aber bald sehen werden, reicht das nicht aus.

2.1.2 Kennzeichen hierarchischer URLs

Um die allgemeinen Syntaxregeln aus RFC 1738 zu erfüllen, muss jede absolute, hierarchische URL den String »//« unmittelbar vor dem sogenannten »Autoritäts«-Abschnitt (Authority Section) enthalten. Fehlt dieser String, sind Format und Funktion der folgenden Teile nach dieser Spezifikation nicht definiert und müssen daher als unklarer, schemaspezifischer Wert behandelt werden.

Hinweis

Ein Beispiel für eine nicht hierarchische URL ist das Protokoll *mailto:*, das zur Angabe von E-Mail-Adressen und eventuell einer Betreffzeile verwendet wird (*mailto:user@example.com?subject=Hello+world*). Solche URLs werden an den Standard-Mailclient weitergegeben, ohne dass versucht wird, sie weiter zu interpretieren.

Das Konzept einer allgemeinen, hierarchischen URL-Syntax ist im Grunde genommen sehr elegant. Es sollte Anwendungen ermöglichen, einige Informationen über die Adresse zu gewinnen, ohne genau wissen zu müssen, wie das jeweilige Schema funktioniert. Beispielsweise kann ein Browser allein durch die Anwendung der allgemeinen URL-Syntax herausfinden, dass *http://example.com/test1/* und *wacky-widget://example.com/test2/* auf denselben vertrauenswürdigen Host verweisen, ohne dass er irgendetwas über das Protokoll *wacky-widget:* wissen muss.

Leider weist die Spezifikation jedoch einen bemerkenswerten Mangel auf: Der genannte RFC sagt nichts darüber aus, was bei URLs geschehen soll, bei denen das Schema bekanntermaßen nicht hierarchisch ist, bei denen aber trotzdem das Präfix »//« verwendet wird, und umgekehrt. Die Referenzimplementierung eines Parsers in RFC 1630 enthält sogar eine unbeabsichtigte Lücke, die dem letzteren Fall von URLs eine widersinnige Bedeutung gibt. In dem einige Jahre später veröffentlichten RFC 3986 haben die Autoren kleinlaut diese Lücke bestätigt und aus Kompatibilitätsgründen Implementierungen zugelassen, die versuchen, solche URLs zu interpretieren. Daher deuten viele Browser die folgenden Beispiele auf unerwartete Weise:

- ***http:example.com/***
  In Firefox, Chrome und Safari kann diese Adresse als *http://example.com* gedeutet werden, wenn ein vollqualifizierter URL-Basiskontext existiert, und als relativer Verweis zu einem Verzeichnis namens *example.com*, wenn eine gültige Basis-URL vorhanden ist.
- ***javascript://example.com/%0Aalert(1)***
  Dieser String wird in allen modernen Browsern als gültige, nicht hierarchische Pseudo-URL gedeutet. Der JavaScript-Code *alert(1)* wird ausgeführt und zeigt ein einfaches Dialogfeld.
- ***mailto://user@example.com***
  Der Internet Explorer akzeptiert diese URL als gültigen, nicht hierarchischen Verweis auf eine E-Mail-Adresse, wobei das »//« einfach übergangen wird. Andere Browser gehen nicht so vor.

2.1.3 Anmeldeinformationen

Der URL-Abschnitt mit den Anmeldeinformationen (Credentials) ist optional. An dieser Stelle kann man einen Benutzernamen und eventuell ein Passwort angeben, die beide zum Abruf der Daten vom Server erforderlich sind. Die abstrakte URL-Syntax schreibt nicht vor, mit welcher Methode diese Anmeldeinformationen übergeben werden. Dies hängt ganz vom Protokoll ab. Bei Protokollen, die keine Authentifizierung unterstützen, ist das Verhalten von URLs mit Anmeldeinformationen nicht definiert.

Werden keine Anmeldeinformationen übergeben, versucht der Browser die Ressource anonym abzurufen. Bei HTTP und mehreren anderen Protokollen bedeutet dies, dass keine Authentifizierungsdaten gesendet werden. Bei FTP jedoch erfolgt dadurch eine Anmeldung an einem Gastkonto namens *ftp* mit einem Bogus-Passwort.

Die meisten Browser akzeptieren an dieser Stelle fast alle Zeichen außer den allgemeinen URL-Abschnittstrennzeichen – mit zwei Ausnahmen: Safari lehnt aus unerfindlichen Gründen eine ganze Reihe von Zeichen ab, darunter »<«, »>«, »{« und »}«. Firefox akzeptiert außerdem keine Zeilenumbrüche.[1]

2.1.4 Die Serveradresse

Eine vollqualifizierte hierarchische URL muss im Abschnitt für die Serveradresse den Standort des Servers, der die gewünschte Ressource bereithält, enthalten, und zwar in einer von drei Varianten: entweder in Form seines DNS-Namens (ohne Berücksichtigung der Groß- und Kleinschreibung; z.B. *example.com*), als IPv4- (z.B. 127.0.0.1) oder als IPv6-Adresse in eckigen Klammern (z.B. [0:0:0:0:0:0:0:1]). Firefox akzeptiert auch IPv4-Adressen und Hostnamen in eckigen Klammern, doch andere Implementierungen weisen solche Eingaben sofort zurück.

Der RFC erlaubt zwar nur die kanonische Schreibweise für IP-Adressen, doch die C-Standardbibliotheken der meisten Anwendungen gehen viel entspannter vor und akzeptieren auch nicht kanonische Adressen, in denen die oktale, dezimale und hexadezimale Schreibweise vermischt sind oder in denen einige oder alle der Oktette zu einem einzigen Integerwert verkettet werden. Daher können die folgenden Varianten als gleichwertig angesehen werden:

- ***http://127.0.0.1/***
  Dies ist die kanonische Darstellung einer IPv4-Adresse.
- ***http://0x7f.1/***
  Bei dieser Darstellung derselben Adresse wird das erste Oktett durch eine Hexadezimalzahl wiedergegeben, während alle übrigen Oktette zu einem einzigen Dezimalwert zusammengefasst sind.
- ***http://017700000001/***
  Hier wird dieselbe Adresse durch einen Oktalwert (mit dem Präfix 0) dargestellt, bei dem alle Oktette zu einem einzigen 32-Bit-Integerwert aneinandergefügt sind.

Eine ähnlich liberale Vorgehensweise lässt sich bei DNS-Namen beobachten. Theoretisch dürfen DNS-Bezeichnungen nur aus einem stark begrenzten Zeichensatz zusammengesetzt werden (laut Definition in RFC 1035 insbesondere alphanumerische Zeichen, ».« und »-«), doch die meisten Browser bitten einfach

1. Das spielt für FTP wahrscheinlich keine Rolle, da die Anmeldeinformationen dabei ohne Codierung übertragen werden. In diesem Protokoll würde ein Zeilenumbruch vom Server als Anfang eines weiteren FTP-Befehls missverstanden. Andere Browser übertragen FTP-Anmeldeinformationen in nicht konformer Codierung mit Prozentzeichen oder entfernen später einfach alle problematischen Zeichen.

den Resolver des Betriebssystems, praktisch alle anderen Zeichen nachzuschlagen, und gewöhnlich macht das Betriebssystem dabei keine Schwierigkeiten. Welche Zeichen im Hostnamen akzeptiert und an den Resolver weitergegeben werden, hängt vom Client ab. Safari geht dabei besonders streng vor, während sich der Internet Explorer sehr entgegenkommend verhält. Die meisten Browser ignorieren in diesem Teil der URL jedoch mehrere Steuerzeichen aus den Bereichen 0x0A bis 0x0D und 0xA0 bis 0xAD.

Hinweis

Ein bemerkenswertes Verhalten der URL-Parser in allen wichtigen Browsern ist die Bereitschaft, mit der sie das Zeichen »。« (der ideografische Punkt, Unicode-Zeichen U+3002) in Hostnamen anstelle eines Punkts akzeptieren, aber nirgendwo sonst in der URL. Angeblich liegt das daran, dass es bei bestimmten chinesischen Tastaturbelegungen einfacher ist, dieses Symbol einzugeben als den erwarteten 7-Bit-ASCII-Wert.

2.1.5 Der Serverport

Der Abschnitt für den Serverport ist optional und gibt einen nicht standardmäßigen Netzwerkport an, über den eine Verbindung auf dem zuvor genannten Server hergestellt werden soll. Praktisch alle von Browsern und Drittanbieterprogrammen unterstützten Anwendungsprotokolle verwenden TCP oder UDP als Übertragungsmethode, und bei beiden wird der Datenverkehr unterschiedlicher Dienste auf dem Computer mithilfe von 16-Bit-Portnummern getrennt gehalten. Jedem Schema ist ein Standardport zugewiesen, den die Server für das entsprechende Protokoll gewöhnlich nutzen (z.B. 80 für HTTP, 21 für FTP). Diese Standardeinstellung kann aber in der URL überschrieben werden.

Hinweis

Ein interessanter, unbeabsichtigter Nebeneffekt dieses Merkmals besteht darin, dass man Browser dazu bringen kann, vom Angreifer gesendete Daten an beliebige Netzwerkdienste zu senden, die nicht das vom Browser erwartete Protokoll verwenden. Beispielsweise ist es möglich, einen Browser auf *mail.example.com:25* zu leiten, wobei 25 nicht der Port für HTTP ist, sondern der für SMTP (Simple Mail Transfer Protocol). Das hat zu einer Reihe von Sicherheitsproblemen geführt, für die dann auch verschiedene, meist mangelhafte Notlösungen erstellt wurden. Mehr darüber erfahren Sie im zweiten Teil dieses Buches.

2.1.6 Hierarchische Dateipfade

Der nächste Teil der URL ist der hierarchische Dateipfad, mit dem die vom Server abzurufende Ressource angegeben wird, z.B. */documents/2012/mein_tagebuch.txt*. Die Spezifikation kann ihre Herkunft aus der Verzeichnissemantik von Unix nicht verhehlen: Sie verlangt die Auflösung der Segmente »/../« und »/./« im Pfad und stellt eine verzeichnisgestützte Möglichkeit dar, um relative Verweise in nicht vollqualifizierten URLs zu bestimmen.

In den 1990er-Jahren erschien es ganz natürlich, dieses Modell des Dateisystems zu nutzen. Webserver fungierten damals ja nur als einfache Gateways zu einer Sammlung statischer Dateien und hier und da zu einem ausführbaren Skript. Inzwischen aber haben viele moderne Frameworks für Webanwendungen auch die letzten Bindungen an das Dateisystem gekappt, da sie jetzt direkt mit Datenbankobjekten oder registrierten Speicherorten im Programmcode umgehen. Die Zuordnung solcher Datenstrukturen zu wohlgeformten URL-Pfaden ist möglich, erfolgt manchmal aber gar nicht oder nicht mit der gebotenen Vorsicht. Dadurch werden der automatische Abruf von Inhalten, die Indizierung und auch Sicherheitstests schwieriger als zuvor.

2.1.7 Der Query-String

Der Query-String ist ein optionaler Abschnitt, in dem beliebige, nicht hierarchische Parameter an die zuvor bezeichnete Ressource übergeben werden können. Ein häufig anzutreffendes Beispiel ist die Übergabe vom Benutzer bereitgestellter Begriffe an ein serverseitiges Skript, das eine Suchfunktion ausführt:

```
http://example.com/search.php?query=Hello+world
```

Die meisten Webentwickler sind mit dem Layout von Query-Strings vertraut. Dieses bekannte Format wird von Browsern bei der Verarbeitung von HTML-Formularen verwendet und richtet sich nach folgender Syntax:

```
name1=wert1&name2=wert2...
```

Überraschenderweise wird diese Struktur in den URL-RFCs nicht vorgeschrieben. Stattdessen wird der Query-String als unklarer Datenbrei behandelt, den der endgültige Empfänger nach seinem Gutdünken interpretieren kann. Im Gegensatz zum Pfad gibt es keine besonderen Analyseregeln dafür.

Hinweise auf das üblicherweise verwendete Format finden sich in dem lediglich informierenden RFC 1630 [6], in RFC 2368 [7], in dem es um E-Mail geht, und in HTML-Spezifikationen über Formulare [8]. Keine dieser Angaben sind jedoch bindend. Wenn Webanwendungen daher eigene Formate für die Daten verwenden, die sie in diesem Teil der URL unterbringen möchten, mag das zwar unhöflich sein, ist aber nicht falsch.

2.1.8 Der Fragment-Identifier

Der Fragment-Identifier (oder die Fragment-ID) ist ein Wert, der eine ähnliche Rolle spielt wie der Query-String, aber optionale Anweisungen für die Clientanwendung gibt und nicht für den Server. (Der Wert soll nicht einmal an den Server gesendet werden.) Weder das Format noch die Funktion der Fragment-ID sind in den RFCs klar definiert, aber es wird angedeutet, dass damit »Unterressourcen« im abgerufenen Dokument angesprochen oder andere dokumentspezifische Darstellungsanweisungen gegeben werden können.

In der Praxis gibt es nur eine anerkannte Verwendungsmöglichkeit für Fragment-IDs im Browser, nämlich die Angabe eines HTML-Ankerelements für die Navigation innerhalb des Dokuments. Die Vorgehensweise ist ganz einfach: Wenn in der URL ein Ankername angegeben wird und ein entsprechendes HTML-Tag zu finden ist, wird das Dokument bei der Anzeige zu dieser Position gescrollt. Anderenfalls ist der Identifier wirkungslos. Da die Information in der URL codiert ist, lässt sich eine solche Detailansicht eines längeren Dokuments leicht an andere weitergeben oder als Lesezeichen speichern. Bei dieser Art der Verwendung ist die Bedeutung der Fragment-ID auf das Scrollen in einem vorhandenen Dokument eingeschränkt. Wenn aufgrund von Benutzeraktionen nur dieser Teil der URL aktualisiert wird, ist es also nicht nötig, neue Daten vom Server abzurufen.

Diese interessante Eigenschaft hat dazu geführt, dass vor nicht allzu langer Zeit eine weitere Verwendungsmöglichkeit für diesen Wert aus dem Hut gezaubert wurde: die Speicherung verschiedener Statusinformationen für clientseitige Skripte. Stellen Sie sich zum Beispiel eine Anwendung zur Darstellung von Landkarten vor, die die Koordinaten der zurzeit angezeigten Karte in der Fragment-ID platziert, sodass sie weiß, von welchem Standort sie ausgehen muss, wenn der Link als Lesezeichen gespeichert oder an andere Benutzer weitergegeben wird. Im Gegensatz zur Aktualisierung eines Query-Strings führt eine Änderung der Fragment-ID im laufenden Betrieb nicht zu einem langwierigen neuen Ladevorgang der Seite. Das macht diesen Datenspeicherungstrick zu einer wirklich praktischen Anwendung des Fragment- Identifiers.

2.1.9 Die URL im Ganzen

Die einzelnen zuvor erwähnten Abschnitte der URL sind durch reservierte Zeichen getrennt: Schrägstriche, Doppelpunkte, Fragezeichen usw. Damit das Ganze funktioniert, dürfen diese Trennzeichen an keiner anderen Stelle der URL auftreten. Stellen Sie sich unter dieser Voraussetzung nun einen Beispielalgorithmus vor, der absolute URLs in die zuvor erwähnten funktionalen Bestandteile zerlegt und dabei ungefähr so ähnlich vorgeht wie ein Browser. Nachfolgend ist ein halbwegs brauchbares Beispiel für einen solchen Algorithmus angegeben:

Schritt 1: Schemanamen extrahieren

Suchen Sie nach dem ersten Doppelpunkt. Der Teil links davon ist der Schemaname. Beenden Sie den Algorithmus, wenn der Schemaname nicht dem erwarteten Satz von Zeichen entspricht. In diesem Fall muss die URL möglicherweise als relativ behandelt werden.

Schritt 2: Den Identifier für hierarchische URLs verarbeiten

Auf den Schemanamen sollte der String »//« folgen. Wenn ja, wird er übersprungen, wenn nein, wird der Algorithmus abgebrochen.

Hinweis

Aus Gründen der Benutzerfreundlichkeit akzeptieren manche Implementierungen unter bestimmten Umständen URLs ohne Schrägstriche oder mit einem, drei oder mehr Schrägstrichen. In dieselbe Kerbe schlägt der Internet Explorer, der von Anfang an überall in der URL umgekehrte Schrägstriche (Backslashes, »\«) anstelle von normalen zugelassen hat. Vermutlich war das als Hilfestellung für unerfahrene Benutzer gedacht.[a] Alle Browser außer Firefox sind diesem Beispiel schließlich gefolgt und erkennen auch URLs wie *http:\\example.com*\.

a. Anders als Unix-basierte Betriebssysteme verwendet Microsoft Windows als Trennzeichen in Dateipfaden Backslashes anstelle von Schrägstrichen (z.B. *c:\windows\system32\calc.exe*). Vielleicht wollte man bei Microsoft den Benutzern nicht zumuten, im Web eine andere Art von Schrägstrich zu verwenden. Oder man hoffte, anderen möglichen Inkonsistenzen mit *file:*-URLs und ähnlichen Mechanismen, die direkt mit dem lokalen Dateisystem zusammenarbeiten, aus dem Weg zu gehen. Andere Besonderheiten des Windows-Dateisystems (z.B. die Nichtbeachtung der Groß- und Kleinschreibung) wurden jedoch nicht übernommen.

Schritt 3: Die Authority Section erfassen

Suchen Sie nach dem ersten »/«, »?« oder »#«, um die Authority Section aus der URL herauszuziehen. Wie bereits erwähnt, akzeptieren die meisten Browser auch »\« statt des normalen Schrägstrichs als Trennzeichen; daher ist dieses Zeichen ebenfalls zu berücksichtigen. Außer im Internet Explorer und Safari wird auch das Semikolon als Trennzeichen für die Authority Section erkannt. Die Gründe dafür sind unbekannt.

Schritt 3a: Die Anmeldeinformationen suchen (falls vorhanden)

Nachdem Sie die Authority Section extrahiert haben, suchen Sie in dem Teilstring nach dem Zeichen »@«. Wenn es vorhanden ist, stellt die vorausgehende Zeichenfolge die Anmeldeinformationen dar. Diese Anmeldeinformationen werden beim ersten Doppelpunkt (falls vorhanden) weiter in Anmeldename und Passwort zerlegt.

Schritt 3b: Die Zieladresse gewinnen

Beim Rest der Authority Section handelt es sich um die Zieladresse. Suchen Sie nach dem ersten Doppelpunkt, der den Hostnamen von der Portnummer trennt. IPv6-Adressen in eckigen Klammern bedürfen einer besonderen Behandlung.

Schritt 4: Den Pfad bestimmen (falls vorhanden)
Wenn auf die Authority Section sofort ein Schrägstrich folgt – wobei in manchen Implementierungen wie bereits erwähnt auch ein Backslash oder ein Semikolon akzeptiert werden –, suchen Sie nach dem ersten »?«, »#« oder dem Ende des Strings. Der Text dazwischen stellt den Pfad dar, der gemäß der Semantik von Unix-Pfaden normalisiert werden muss.

Schritt 5: Den Query-String extrahieren (falls vorhanden)
Wenn auf das letzte erfolgreich analysierte Teilstück ein Fragezeichen folgt, suchen Sie nach dem ersten »#« oder dem Ende des Strings. Der Text dazwischen ist der Query-String.

Schritt 6: Den Fragment-Identifier bestimmen (falls vorhanden)
Wenn auf das letzte erfolgreich analysierte Teilstück ein »#« folgt, ist der restliche Text bis zum Ende des Strings der Fragment-Identifier. In jedem Fall ist der Algorithmus damit beendet.

Dieser Algorithmus mag banal erscheinen, doch offenbart er einige Details, an die selbst erfahrene Programmierer normalerweise nicht denken. Außerdem veranschaulicht er, dass es für Otto Normalbenutzer recht schwierig sein kann, zu erkennen, wie eine bestimmte URL analysiert wird. Beginnen wir mit einem ziemlich einfachen Fall:

```
http://example.com&kauderwelsch=1234@167772161/
```

Das Ziel dieser URL – eine IP-Adresse in verketteter Form, die zu 10.0.0.1 decodiert wird – ist für Nichtfachleute nicht unmittelbar zu erkennen, und viele Benutzer würden glauben, dass diese Adresse sie zu *example.com* führt.[2] Aber das war ein einfaches Beispiel! Dringen wir nun ein wenig in die Tiefen der Syntax vor:

```
http://example.com\@coredump.cx/
```

Im Firefox führt diese URL zu *coredump.cx*, da *example.com*\ als gültiger Wert für die Anmeldeinformationen angesehen wird. In fast allen anderen Browsern wird »\« als Pfadtrennzeichen angesehen, weshalb der Benutzer stattdessen auf *example.com* landet.

Ein besonders entmutigendes Beispiel gibt es für den Internet Explorer. Betrachten Sie folgende URL:

```
http://example.com;.coredump.cx/
```

2. Der Trick mit dem @-Zeichen fand rasche Verbreitung für alle möglichen Arten von Onlinebetrug, mit denen unaufmerksame Benutzer hereingelegt wurden. Die Versuche, diese Gefahr zu unterbinden, reichten von schwerfälligen und sehr speziellen Vorgehensweisen (z.B. die Deaktivierung der URL-gestützten Authentifizierung im Internet Explorer oder ihre Beeinträchtigung durch Warnungen in Firefox) bis zu recht sinnvollen Maßnahmen (z.B. die Hervorhebung des Hostnamens in der Adressleiste verschiedener Browser).

Browser von Microsoft lassen ein Semikolon im Hostnamen zu und lösen diese Adresse bei entsprechender Konfiguration der Domäne *coredump.cx* erfolgreich auf. Die meisten anderen Browser korrigieren die URL automatisch zu *http://example.com/;.coredump.cx* und führen den Benutzer zu *example.com*. (Eine Ausnahme bildet Safari, wo diese Syntax einen Fehler hervorruft.) Wenn Sie das schon für unübersichtlich halten, dann denken Sie daran, dass wir gerade erst damit anfangen, uns mit der Arbeitsweise von Browsern zu beschäftigen!

Hinweis

Der Sicherheitsforscher Krzysztof Kotowicz publizierte Anfang 2011 eine interessante Testsuite, die Browsern ihre Geheimnisse im URL-Handling entlockt. Die einzelnen Testfälle demonstrieren eindrucksvoll, wie verwirrend allein die Unterscheidung zwischen relativen und absoluten URLs sein kann. Die Test-Suite findet sich unter folgender URL: *http://kotowicz.net/absolute/*

2.2 Reservierte Zeichen und URL-Codierung

Der im vorherigen Abschnitt vorgestellte Algorithmus zur URL-Analyse geht davon aus, dass bestimmte reservierte Syntaxtrennzeichen innerhalb der URL in keiner anderen Funktion auftreten (also nicht als Teil eines Benutzernamens, Anforderungspfades usw.). Es handelt sich dabei um folgende Zeichen:

```
: / ? # [ ] @
```

Der RFC nennt auch eine Reihe untergeordneter Trennzeichen, ohne ihnen einen bestimmten Zweck zuzuweisen, um innerhalb der Abschnitte der oberen Ebene schema- oder anwendungsspezifische Funktionen umsetzen zu können:

```
! $ & ' ( ) * + , ; =
```

Im Prinzip sind alle diese Zeichen verboten. Aber es gibt Fälle, in denen sie durchaus zur URL gehören können (z.B. für Suchbegriffe, die ein Benutzer eingegeben hat und die dann im Query-String an den Server übergeben werden). Anstatt sie völlig auszuschließen, bietet der Standard eine Methode, um alle gelegentlichen Vorkommen dieser Werte zu codieren.

Diese Methode, *URL-Codierung* (URL Encoding) oder *Codierung mit Prozentzeichen* (Percent Encoding) genannt, ersetzt die Zeichen durch ein Prozentzeichen und eine zweistellige Hexadezimalzahl, die dem zugehörigen ASCII-Wert entspricht. Beispielsweise wird »/« als *%2F* codiert (Großbuchstaben sind üblich, aber nicht zwingend erforderlich). Um Mehrdeutigkeiten zu vermeiden, muss ein Prozentzeichen, das ein Prozentzeichen sein soll, daher ebenfalls codiert werden, und zwar als *%25*. Alle Zwischenstationen, die bestehende URLs verarbeiten (Browser und Webanwendungen), dürfen nicht versuchen, reservierte Zeichen in

weitergeleiteten URLs zu codieren oder zu decodieren, da sich die Bedeutung der URL dadurch ändern würde.

Die Unverletzlichkeit der reservierten Zeichen in bestehenden URLs steht jedoch leider im Konflikt mit der Notwendigkeit, auf technisch unzulässige URLs zu reagieren, in denen diese Zeichen missbräuchlich verwendet werden. In der Praxis können Browser auf solche URLs stoßen, doch die Spezifikationen decken dieses Problem nicht ab, weshalb sich die Hersteller gezwungen sahen, zu improvisieren, was wieder zu Inkonsistenzen zwischen den Implementierungen führte. Betrachten Sie beispielsweise eine URL wie *http://a@b@c/*. Soll dies als *http://a@b%40c/* oder als *http://a%40b@c/* gedeutet werden? Internet Explorer und Safari halten die erste Variante für sinnvoller, andere entscheiden sich für die zweite.

Die restlichen nicht reservierten Zeichen sollen in der URL-Syntax keine bestimmte Bedeutung haben. Einige von ihnen (z.B. die nicht druckbaren ASCII-Steuerzeichen) widersprechen jedoch eindeutig der Vorstellung, dass URLs von Menschen lesbar und transportsicher sein sollen. Daher definiert der RFC eine Teilmenge mit dem merkwürdigen Namen *unreservierte Zeichen* (die aus alphanumerischen Zeichen sowie »-«, ».«, »_« und »~« besteht) und schreibt vor, dass formal nur die Elemente dieser Teilmenge und die reservierten Zeichen in ihrer jeweils vorgegebenen Rolle in URLs zulässig sind.

Hinweis

Sonderbarerweise dürfen nur die unreservierten Zeichen in nicht maskierter Form auftreten, aber es ist nicht vorgeschrieben. User Agents können sie nach eigenem Gutdünken mit Prozentzeichen codieren oder nicht, ohne dass sich dadurch die Bedeutung der URL ändern würde. Das führt jedoch zu einer neuen Möglichkeit, um die Benutzer zu verwirren: der Verwendung nicht kanonischer Darstellungen unreservierter Zeichen. Beispielsweise sind alle drei folgenden Varianten gleichwertig:

- *http://example.com/*
- *http://%65xample.%63om/*
- *http://%65%78%61%6d%70%6c%65%2e%63%6f%6d/*[a]

a. Ähnliche nicht kanonische Codierungen wurden häufig für verschiedene Arten von Social-Engineering-Angriffen verwendet, weshalb im Laufe der Jahre diverse Gegenmaßnahmen entwickelt wurden. Wie üblich sind einige dieser Gegenmaßnahmen störend (Firefox etwa lehnt einfach jegliche Prozentzeichencodierung in Hostnamen ab), andere dagegen sind ziemlich gut (etwa die erzwungene »Kanonisierung« der Adressleiste durch Decodierung jeglichen unnötig codierten Textes in der Darstellung).

Von der Menge der sogenannten unreservierten Zeichen sind einige druckbare Zeichen ausgeschlossen, die eigentlich nicht reserviert sind. Streng genommen müssten die RFCs verlangen, sie ausnahmslos mit % zu codieren. Da sich die Browser jedoch nicht ausdrücklich um die Durchsetzung dieser Regel kümmern, wird sie nicht besonders ernst genommen. Vor allem lassen sämtliche Browser die

Zeichen »^«, »{«, »|« und »}« ohne Maskierung in URLs zu und senden sie so, wie sie sind, an den Server. Der Internet Explorer lässt darüber hinaus auch »<«, »>« und »`« gelten, und die Browser Internet Explorer, Firefox und Chrome akzeptieren »\«. Ein Anführungszeichen ist in Chrome und im Internet Explorer zulässig, und sowohl Opera als auch der Internet Explorer übergeben das nicht druckbare Zeichen 0x7F (DEL) unverändert.

Im Gegensatz zu den Anforderungen im RFC codieren die meisten Browser auch keine Fragment-Identifier. Das führt bei clientseitigen Skripten zu einem unerwarteten Problem, da sie mit diesem String arbeiten und erwarten, dass bestimmte unsichere Zeichen niemals darin vorkommen. Dieses Thema sehen wir uns in Kapitel 6 genauer an.

2.2.1 Umgang mit Text jenseits der ASCII-Welt

Viele Sprachen verwenden Zeichen außerhalb des grundlegenden 7-Bit-ASCII-Zeichensatzes und der standardmäßigen 8-Bit-Codepage CP437, die traditionell für alle PC-kompatiblen Systeme benutzt wird. Viele haben sogar ein ganz anderes Alphabet als das lateinische.

Um die Bedürfnisse der oft ignorierten, aber sehr zahlreichen Benutzer aus nicht englischsprachigen Ländern zu erfüllen, wurden schon lange vor dem Aufkommen des Web verschiedene 8-Bit-Codierungen mit einem alternativen Satz an High-Bit-Zeichen entwickelt: ISO 8859-1, CP850 und Windows 1252 für westeuropäische Sprachen, ISO 8859-2, CP852 und Windows 1250 für Ost- und Mitteleuropa sowie KOI8-R und Windows 1251 für Russisch. Da sich eine ganze Reihe von Alphabeten nicht in das Korsett von 256 Zeichen pressen ließen, kamen schließlich auch komplexe Codepages mit variabler Zeichenbreite wie Shift JIS für Katakana auf.

Die Inkompatibilität dieser verschiedenen Zeichenzuordnungen machte es schwierig, Dokumente zwischen Computern auszutauschen, die für verschiedene Codepages eingerichtet waren. Dieses zunehmende Problem führte Anfang der 1990er-Jahre zur Entwicklung von *Unicode*, einem universellen Zeichensatz, der nicht mehr vollständig in 8 Bit dargestellt werden kann und praktisch alle regionalen Schrift- und Piktogrammsysteme der Menschheit umfassen soll. Auf Unicode folgte UTF-8, eine relativ einfache Darstellung dieser Zeichen mit variabler Breite. Theoretisch war UTF-8 für alle Anwendungen sicher, die mit herkömmlichen 8-Bit-Formaten umgehen konnten. Leider erforderte UTF-8 zur Codierung der High-Bit-Zeichen jedoch mehr Bytes als die meisten Konkurrenzmodelle, was vielen Benutzern verschwenderisch und unnötig vorkam. Wegen dieser Kritik dauerte es mehr als ein Jahrzehnt, bis UTF-8 im Web Fuß fassen konnte, und dies geschah auch erst, nachdem alle relevanten Protokolle ihre Stellung gefestigt hatten.

Diese bedauerliche Verzögerung wirkte sich auf die Handhabung von URLs mit Benutzereingaben aus. Browser mussten schon sehr früh damit umgehen können, aber als sich die Entwickler den entsprechenden Standards zuwandten, fanden sie darin keine sinnvollen Richtlinien. Noch 2005 hatte RFC 3986 nicht mehr zu sagen als Folgendes:

> *Im lokalen und regionalen Zusammenhang und bei fortschreitender Technologie ist es für die Benutzer wahrscheinlich vorteilhaft, wenn sie eine breitere Palette von Zeichen einsetzen können. Diese Verwendung wird in dieser Spezifikation aber nicht definiert.*
>
> *Mit Prozentzeichen codierte Oktette [...] können innerhalb einer URI verwendet werden, um Zeichen außerhalb des US-ASCII-Zeichensatzes darzustellen, sofern diese Darstellung von dem Schema oder dem Protokoll, in dem auf die URI verwiesen wird, zulässig ist. Eine solche Definition sollte die Zeichencodierung festlegen, die zur Zuordnung der Zeichen zu den Oktetten verwendet wird, bevor sie mit Prozentzeichen in der URI codiert werden.*

Das blieb jedoch Wunschdenken, denn keiner der anderen Standards kümmerte sich um dieses Thema. Es war immer möglich, rohe High-Bit-Zeichen in eine URL aufzunehmen, aber ohne Kenntnis der Codierung, nach der sie interpretiert werden sollten, konnte der Server nicht erkennen, ob *%B1* nun »±«, »â« oder irgendeinen anderen Schnörkel aus dem nativen Skript des Benutzers bedeuten soll.

Leider haben die Browserhersteller nicht die Initiative ergriffen und keine einheitliche Lösung für dieses Problem geschaffen. Die meisten Browser übersetzen URL-Pfadsegmente in UTF-8 (oder ISO 8859-1, falls das ausreicht), erstellen den Query-String dann aber in der Codepage der Webseite, auf die verwiesen wird. Wenn URLs manuell eingegeben oder an bestimmte Sonder-APIs übergeben werden, können High-Bit-Zeichen auch auf ihre 7-Bit-US-ASCII-Entsprechungen heruntergestuft, durch Fragezeichen ersetzt oder aufgrund von Implementierungsmängeln komplett unkenntlich gemacht werden.

Die Möglichkeit, Query-Strings mit nicht lateinischen Zeichen übergeben zu können, erfüllte ein wichtiges Bedürfnis – ganz unabhängig davon, wie schlecht sie auch implementiert sein mochte. Bei der herkömmlichen Codierung durch Prozentzeichen blieb ein Teil der URL komplett außen vor: High-Bit-Eingaben waren bei der Angabe des Zielservers nicht zulässig, da der gängige DNS-Standard in Domänennamen zumindest prinzipiell nur durch Punkte getrennte alphanumerische Zeichen und Bindestriche zuließ. Zwar hielt sich niemand an diese Regel, aber jeder Namensserver machte andere Ausnahmen.

Als aufmerksamer Leser werden Sie sich wahrscheinlich fragen, warum diese Einschränkung überhaupt eine Rolle spielen sollte. Warum sollte es wichtig sein, lokalisierte Domänennamen auch in nicht lateinischen Schriftzeichen verwenden zu können? Diese Frage lässt sich heute nur noch schwer beantworten. Einfach gesagt, befürchteten einige Personen, dass das Fehlen solcher Codierungen Unter-

nehmen und Einzelpersonen rund um die Welt davon abhalten könnte, das Web ohne Einschränkung geschäftlich oder zur Unterhaltung zu nutzen. Ob nun zu Recht oder nicht, versuchten diese Leute für die allgemeine Akzeptanz des Web zu sorgen.

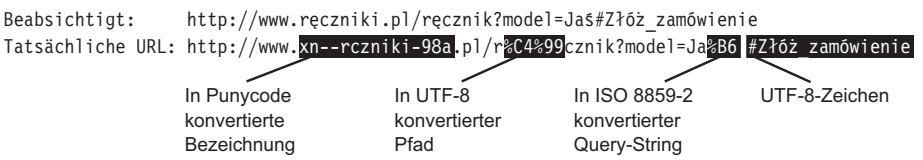
Dies führte zur Bildung der IDNA (Internationalized Domain Names in Applications). Als Erstes erschien RFC 3490 [9] mit einem ziemlich gekünstelten Schema zur Codierung beliebiger Unicode-Strings mithilfe alphanumerischer Zeichen und Bindestriche. Danach wurde in RFC 3492 [10] eine Möglichkeit beschrieben, um diese Codierung mit einem als *Punycode* bezeichneten Format auf DNS-Bezeichnungen anzuwenden. Punycode sah ungefähr wie folgt aus:

```
xn--[US-ASCII-Teil]-[codierte Unicode-Daten]
```

Ein konformer Browser, der eine technisch unzulässige URL mit Nicht-US-ASCII-Zeichen im Hostnamen erhielt, sollte diesen Namen in Punycode übersetzen, bevor er das DNS-Lookup durchführte. Bei Punycode in einer vorhandenen URL sollte der Browser eine decodierte, für Menschen lesbare Version dieses Strings in der Adressleiste anzeigen:

Hinweis

Die Kombination all dieser inkompatiblen Codierungsverfahren kann zu einer lustigen Mischung führen. Betrachten Sie die folgende Beispiel-URL eines fiktiven polnischen Handtuch-Shops:



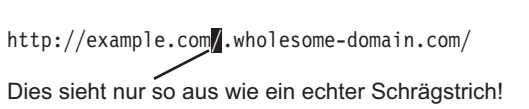
Von allen URL-Codierungsverfahren erwies sich IDNA als das problematischste. Das liegt vor allem daran, dass die in der Adressleiste des Browsers angezeigte URL zu den wichtigsten Sicherheitsindikatoren im Web gehört, denn darüber können die Benutzer auf einen Blick die Websites, denen sie vertrauen und mit denen sie in geschäftlichen Beziehungen stehen, vom Rest des Internets unterscheiden. Wenn der Hostname im Browser aus 38 bekannten und gut unterscheidbaren Zeichen besteht, können nur sehr sorglose Opfer dazu verleitet werden, zu glauben, ihre Lieblingsdomäne *example.com* sei mit der Website *examp1e.com* eines Hochstaplers identisch. Durch IDNA jedoch wird die Menge dieser 38 Zeichen unbedacht auf die von Unicode unterstützten ca. 100.000 Glyphen erweitert, von denen viele genau identisch aussehen und nur aufgrund funktionaler Unterschiede auseinanderzuhalten sind.

Wie gravierend ist das? Betrachten wir als Beispiel das kyrillische Alphabet. Es enthält eine Reihe von Homoglyphen, die zwar genauso aussehen wie lateinische Buchstaben, aber ganz andere Unicode-Werte aufweisen und daher auch zu völlig anderen Punycode-DNS-Namen aufgelöst werden:

Lateinisch	**a**	**c**	**e**	**i**	**j**	**o**	**p**	**s**	**x**	**y**
	U+0061	U+0063	U+0065	U+0069	U+006A	U+006F	U+0070	U+0073	U+0078	U+0079
Kyrillisch	**a**	**c**	**e**	**i**	**j**	**o**	**p**	**s**	**x**	**y**
	U+0430	U+0441	U+0435	U+0456	U+0458	U+004E	U+0440	U+0455	U+0445	U+0443

Als IDNA vorgeschlagen und in Browsern implementiert wurde, hatte niemand die Folgen dieses Problems ernsthaft bedacht. Die Browserhersteller gingen offensichtlich davon aus, dass die DNS-Registrierungsstellen die Verwendung identisch aussehender Namen verhindern würden, während die Registrierungsstellen meinten, es sei Sache der Browserhersteller, für eine eindeutige grafische Darstellung in der Adressleiste zu sorgen.

2002 wurde die Bedeutung dieses Problems endlich von allen beteiligten Parteien erkannt. In diesem Jahr veröffentlichten Evgeniy Gabrilovich und Alex Gontmakher den Artikel »Der Homographen-Angriff«, in dem sie diese Schwachstelle ausführlich behandelten [11]. Sie stellten fest, dass alle Sicherheitsmaßnahmen von Registrierungsstellen, wenn sie denn umgesetzt würden, immer noch einen entscheidenden Mangel aufwiesen. Nichts konnte einen Angreifer davon abhalten, eine wohlgeformte Top-Level-Domäne zu erwerben und dann auf seinem eigenen Namensserver einen Subdomäneneintrag einzurichten, der bei Anwendung der IDNA-Transformation zu einem optisch mit *example.com/* identischen String decodiert wird (wobei das letzte Zeichen nicht der echte ASCII-Schrägstrich ist, sondern ein funktionsloser Doppelgänger). Das Ergebnis sieht dann wie folgt aus:



Eine Registrierungsstelle kann nichts tun, um einen solchen Angriff zu verhindern. Damit liegt die Verantwortung also wieder bei den Browserherstellern. Aber welche konkreten Möglichkeiten haben sie?

Wie sich herausstellte, sind es nicht viele. Wir erkennen jetzt, dass der unzureichend durchdachte IDNA-Standard nicht kurz und schmerzlos repariert werden kann. Browserhersteller haben auf diese Gefahr auf verschiedene Weise reagiert: Sie greifen auf unverständlichen Punycode zurück, wenn die Gebietscodierung des Benutzers nicht mit dem Skript in einer DNS-Bezeichnung

übereinstimmt (was zu Problemen führt, wenn jemand fremdsprachige Websites aufsucht oder einen importierten oder falsch konfigurierten Computer verwendet), sie lassen IDNA nur in bestimmten länderspezifischen Top-Level-Domänen zu (was die Verwendung von internationalisierten Domänennamen in *.com* und anderen wertvollen TLDs verhindert), oder sie schließen bestimmte »gefährliche« Zeichen aus, die Schrägstrichen, Punkten, Leerraum usw. ähneln (was aber angesichts der Vielzahl von Schriftarten in aller Welt ein aussichtsloses Unterfangen ist).

Diese Maßnahmen sind drastisch genug, um die Ausbreitung internationalisierter Domänennamen stark zu behindern. Das wird womöglich so weit gehen, dass das Fortbestehen dieses Standards mehr Sicherheitsprobleme mit sich bringt als echten Nutzen für Benutzer aus Sprachräumen mit nicht lateinischer Schrift.

2.3 Übliche URL-Schemas und ihre Funktion

Damit wollen wir die bizarre Welt der URL-Analyse hinter uns lassen und uns wieder den Grundlagen zuwenden. Weiter vorn in diesem Kapitel haben wir schon angedeutet, dass manche Schemas unerwartete Sicherheitsprobleme aufwerfen. Deshalb muss jede Webanwendung URLs, die vom Benutzer bereitgestellt werden, mit großer Vorsicht behandeln. Um dieses Problem deutlicher zu machen, wollen wir uns alle URL-Schemas ansehen, die zurzeit in einer typischen Browserumgebung unterstützt werden. Sie lassen sich in vier grundlegende Gruppen einteilen.

2.3.1 Vom Browser unterstützte Protokolle für den Dokumentabruf

Diese Schemas, die intern vom Browser verarbeitet werden, bieten eine Möglichkeit, um beliebige Inhalte über ein bestimmtes Transportprotokoll abzurufen und dann mithilfe einer üblichen browserinternen Darstellungslogik anzuzeigen. Das ist die grundlegendste und offensichtlichste Funktion einer URL.

Die Liste der üblichen unterstützten Schemas in dieser Kategorie ist erstaunlich kurz: Sie umfasst *http:* (RFC 2616), den wichtigsten Übertragungsmodus im Web, der auch das Hauptthema des nächsten Kapitels ist, *https:*, eine verschlüsselte Version von HTTP (RFC 2818 [12]), und *ftp:*, ein älteres Dateiübertragungsprotokoll (RFC 959 [13]). Außerdem unterstützen alle Browser auch *file:* (früher auch *local:*), eine systemspezifische Methode für den Zugriff auf das lokale Dateisystem und auf NFS- und SMB-Freigaben. (Letzteres Schema ist über Seiten im Internet jedoch gewöhnlich nicht direkt zugänglich.)

Es gibt noch zwei weitere, eher obskure Fälle, die eine kurze Erwähnung verdienen, nämlich die noch in Firefox vorhandene integrierte Unterstützung für das Schema *gopher:*, einen der erfolglosen Vorläufer des Web (RFC 1436 [14]), und *shttp:*, einen alternativen, aber fehlgeschlagenen Ansatz für HTTPS (RFC 2660

[15]), der noch im Internet Explorer erkannt wird (aber heute einfach zu HTTP umgesetzt wird).

2.3.2 Protokolle für Drittanbieteranwendungen und Plug-ins

URLs mit diesen Schemas werden einfach an externe, spezialisierte Anwendungen weitergegeben, die Funktionen wie Medienwiedergabe, die Anzeige von Dokumenten oder IP-Telefonie bereitstellen. An diesem Punkt endet (meistens) die Beteiligung des Browsers.

Heutzutage gibt es Dutzende von externen Protokollhandlern, sodass ein eigenes dickes Buches nötig wäre, um sie alle zu beschreiben. Zu den gebräuchlichsten Vertretern dieser Art gehören das Schema *acrobat:*, das, wie der Name schon erkennen lässt, an Adobe Acrobat Reader weitergeleitet wird, *callto:* und *sip:*, die von vielen Arten von Instant-Messenger- und Telefonie-Software verwendet werden, die von Apple iTunes genutzten Schemas *daap:*, *itpcs:* und *itms*, die Protokolle *mailto:*, *news:* und *nntp:* für E-Mail- und Usenet-Clients, die Protokolle *mmst:*, *mmsu:*, *msbd:* und *rtsp:* für Streaming-Media-Player und viele weitere. Auch manche Browser tauchen in dieser Liste auf. Das bereits erwähnte Schema *firefoxurl:* startet Firefox innerhalb eines anderen Browsers, während *cf:* aus dem Internet Explorer heraus Zugriff auf Chrome gibt.

Wenn solche Schemas in URLs auftreten, haben sie meistens keine Auswirkungen auf die Sicherheit der Webanwendungen, die sie durchreichen. (Das ist allerdings nicht garantiert, vor allem nicht im Fall von Plug-in-Inhalten.) Beachten Sie jedoch, dass Protokollhandler von Drittanbietern häufig fehlerverseucht sind und manchmal missbraucht werden, um die Sicherheit des Betriebssystems zu knacken. Daher gehört es für jede auch nur halbwegs vertrauenswürdige Website zum guten Ton, die Möglichkeiten der Navigation über solche obskuren Protokolle einzuschränken.

2.3.3 Nicht kapselnde Pseudoprotokolle

Eine Reihe von Protokollen sind für den bequemen Zugriff auf die Skript-Engine des Browsers und andere interne Funktionen reserviert, ohne dass sie tatsächlich irgendwelche Inhalte über das Netzwerk abrufen. Manchmal richten sie nicht einmal einen isolierten Dokumentkontext ein, um das Ergebnis anzuzeigen. Viele dieser Pseudoprotokolle sind hochgradig browserspezifisch und entweder nicht vom Internet aus zugänglich oder nicht in der Lage, irgendwelchen Schaden anzurichten. Allerdings gibt es mehrere bedeutende Ausnahmen von dieser Regel.

Die wahrscheinlich bekannteste dieser Ausnahmen ist das Schema *javascript:* (das in früheren Jahren auch unter Namen wie *livescript:* oder in Netscape-Brow-

sern unter *mocha:* verfügbar war). Dieses Schema gibt Zugriff auf die JavaScript-Programmier-Engine im Kontext der zurzeit betrachteten Website.

Hinweis

In Firefox war es lange Zeit möglich, die gefährlichen JavaScript-URLs einfach mit dem Präfix `feed:` oder gar `pcast:` auszustatten. Auf diesem Wege konnte so mancher XSS-Filter umgangen werden. Gefunden und berichtet wurde die Lücke vom Sicherheitsforscher Soroush Dalili.

Im Internet Explorer bietet *vbscript:* ähnliche Möglichkeiten über die proprietäre Visual-Basic-Schnittstelle.

Ein weiterer wichtiger Fall ist das Protokoll *data:* (RFC 2397 [16]), mit dem es möglich ist, kurze Inline-Dokumente ohne zusätzliche Netzwerkanforderungen zu erstellen, wobei der Betriebskontext zum großen Teil von der Seite geerbt wird, auf der sich der Verweis befindet. Das folgende Beispiel zeigt eine *data:*-URL:

```
data:text/plain,Why,%20hello%20there!
```

Solche extern zugänglichen Pseudo-URLs sind von großer Bedeutung für die Sicherheit der Website. Werden sie aufgerufen, können ihre Nutzdaten im Kontext der Ursprungsdomäne ausgeführt werden, wobei möglicherweise Daten gestohlen oder das Erscheinungsbild der Seite für den betroffenen Benutzer geändert wird. Diese besonderen Fähigkeiten von Browserskriptsprachen werden wir in Kapitel 6 besprechen, aber wie Sie schon ahnen können, sind sie beachtlich. (Die Regeln für URL-Kontextvererbung sind Thema von Kapitel 10.)

2.3.4 Kapselnde Pseudoprotokolle

Diese besondere Klasse von Pseudoprotokollen kann als Präfix für andere URLs verwendet werden, um einen speziellen Decodierungs- oder Darstellungsmodus für die abgerufene Ressource zu erzwingen. Das vielleicht bekannteste Beispiel ist das von Firefox und Chrome unterstützte Schema *view-source:*, das zur Anzeige des Quellcodes der HTML-Seite in einem ansprechenden Format dient. Dieses Schema wird wie folgt verwendet:

```
view-source:http://www.example.com/
```

Protokolle mit einer ähnlichen Funktion sind *jar:*, mit dem in Firefox im laufenden Betrieb Inhalte aus ZIP-Dateien gewonnen werden können, *wyciwyg:* und *view-cache:*, die in Firefox bzw. Chrome Zugriff auf gecachte Seiten geben, das obskure Schema *feed:*, das in Safari für den Zugriff auf Newsfeeds gedacht ist [17], sowie eine Heerschar schlecht dokumentierter Protokolle im Zusammen-

hang mit dem Hilfesystem und anderen Komponenten von Windows (*hcp:*, *its:*, *mhtml:*, *mk:*, *ms-help:*, *ms-its:* und *ms-itss:*).

Eine gemeinsame Eigenschaft vieler dieser kapselnden Protokolle besteht darin, dass sie es einem Angreifer ermöglichen, schädliche Dinge hinter der tatsächlichen URL zu verstecken, die letzten Endes im Browser von sorglosen Filtern interpretiert wird: *view-source:javascript:* (oder sogar *view-source:view-source: javascript:*) gefolgt von schädlichem Code ist eine einfache Möglichkeit, um so etwas zu erreichen. Hier und da mögen zwar einige Sicherheitseinschränkungen in Kraft sein, die solche Tricks verhindern sollen, allerdings kann man sich auf sie nicht verlassen. Ein weiteres erhebliches Problem, das vor allem beim Schema *mhtml:* von Microsoft immer wieder auftritt, ergibt sich daraus, dass bei der Benutzung dieses Protokolls einige der vom Server auf HTTP-Ebene vorgesehenen Inhaltsregeln ignoriert werden, was weitreichende Folgen haben kann [18].

2.3.5 Ein letztes Wort zur Schemaerkennung

Die Vielzahl an Pseudoprotokollen ist der Hauptgrund dafür, dass Webanwendungen vom Benutzer bereitgestellte URLs sorgfältig überwachen müssen. Die unsicheren und browserspezifischen URL-Analysemuster zusammen mit der potenziell wachsenden Liste der unterstützten Schemas haben zur Folge, dass es nicht ausreicht, bekanntermaßen gefährliche Schemas auf eine Blacklist zu setzen. Beispielsweise kann eine Überprüfung auf *javascript:* umgangen werden, indem man das Schlüsselwort durch einen Tabulator oder einen Zeilenumbruch aufteilt, durch *vbscript:* ersetzt oder ihm ein anderes, kapselndes Schema voranstellt.

2.4 Auflösung relativer URLs

Relative URLs wurden schon an mehreren Stellen in diesem Kapitel erwähnt und bedürfen noch einer ausführlicheren Erläuterung. Dass sie auf fast jeder Webseite im Internet anzutreffen sind, liegt daran, dass eine erhebliche Menge an URLs auf Ressourcen auf demselben Server, vielleicht sogar im selben Verzeichnis verweist. Es wäre lästig und unnötig, bei jedem dieser Verweise eine vollqualifizierte URL im Dokument zu verlangen, weshalb stattdessen kurze, relative URLs verwendet werden (wie *../andere_datei.txt*). Die fehlenden Angaben werden aus der URL des Mutterdokuments abgeleitet.

Da relative URLs in sämtlichen Szenarios verwendet werden dürfen, in denen auch absolute URLs auftreten können, muss der Browser über eine Möglichkeit verfügen, diese beiden Arten auseinanderzuhalten. Auch für Webanwendungen ist es von Vorteil, wenn sie diese Unterscheidung treffen können, da die meisten Arten von URL-Filtern nur absolute URLs kritisch überprüfen müssen und lokale Verweise unverändert durchgehen lassen können.

Die Spezifikation lässt diese Aufgabe einfach erscheinen: Wenn der URL-String nicht mit einem gültigen Schemanamen gefolgt von einem Doppelpunkt und nach Möglichkeit der Zeichenfolge »//« beginnt, soll er als relativer Verweis angesehen werden. Steht kein Kontext für die Analyse einer relativen URL bereit, soll sie zurückgewiesen werden. Alles andere ist dann ein sicherer relativer Link. Einleuchtend, oder?

Wie zu erwarten war, ist es jedoch nicht so einfach, wie es scheint. Wie in den vorherigen Abschnitten schon erwähnt, gibt es zwischen den Implementierungen oft Unterschiede hinsichtlich des Satzes an Zeichen, die in gültigen Schemanamen zugelassen werden, sowie hinsichtlich der gültigen Muster, die anstelle von »//« akzeptiert werden. Noch wichtiger aber ist wahrscheinlich die verbreitete Fehleinschätzung, ein relativer Link könne nur auf Ressourcen auf demselben Server zeigen. Es gibt auch eine Reihe anderer, weniger bekannter Varianten von relativen URLs.

Um diese Möglichkeiten zu verdeutlichen, sehen wir uns kurz die bekannten Klassen relativer URLs an:

- **Mit Schema, aber ohne Authority** (*http:foo.txt*)
  Diese berüchtigte Sicherheitslücke wird in RFC 3986 angedeutet und geht wahrscheinlich auf ein Versehen in einer früheren Spezifikation zurück. Darin wurden diese URLs zwar als (ungültige) absolute Verweise beschrieben, allerdings wurde auch ein wirrer Analysealgorithmus bereitgestellt, der diese Verweise bereitwillig fehldeutete.

  Bei dieser Interpretation können solche URLs ein Protokoll und einen Pfad, einen Query-String oder eine Fragment-ID festlegen, während die Authority Section von der verweisenden Stelle kopiert wird. Diese Syntax wurde von mehreren Browsern übernommen, allerdings nicht konsistent. In einigen Fällen wird die Angabe *http:foo.txt* als relativer Verweis gedeutet, *https:example.com* dagegen als absoluter!

- **Ohne Schema, aber mit Authority** (*//example.com*)
  Das ist ein weiterer berüchtigter, aber zumindest gut dokumentierter Mangel. Während */example.com* ein Verweis auf eine lokale Ressource auf dem vorliegenden Server ist, zwingt der Standard die Browser dazu, *//example.com* auf ganz andere Weise zu behandeln, nämlich als Verweis auf eine andere Authority über das aktuelle Protokoll. In diesem Fall wird das Schema von der verweisenden Stelle übernommen, alle anderen URL-Angaben dagegen aus der relativen URL.

- **Ohne Schema und Authority, aber mit Pfad** (*../notes.txt*)
  Dies ist die übliche Variante eines relativen Links. Die Protokoll- und Authority-Informationen werden aus der verweisenden URL kopiert. Beginnt die relative URL nicht mit einem Schrägstrich, wird auch der Pfad bis zum äußersten rechten Schrägstrich kopiert. Lautet die Basis-URL beispielsweise *http://*

www.example.com/files/ ist der Pfad derselbe. Bei *http://www.example.com/files/index.html* dagegen wird der Dateiname abgeschnitten, der neue Pfad angehängt und der verkettete Wert der üblichen Pfadnormalisierung unterzogen. Query-String und Fragment-ID werden nur von der relativen URL bezogen.

- **Ohne Schema, Authority und Pfad, aber mit Query-String** (*?search=bunnies*)
  In diesem Fall werden Protokoll, Authority- und Pfadinformation komplett aus der verweisenden URL kopiert. Query-String und Fragment-ID werden nur von der relativen URL bezogen.
- **Ausschließlich mit Fragment-ID** (*#bunnies*)
  In diesem Fall werden alle Informationen außer der Fragment-ID komplett aus der verweisenden URL kopiert und nur die Fragment-ID ersetzt. Wie zuvor erwähnt, wird die Seite dabei normalerweise nicht neu geladen.

Da beim Umgang mit diesen Arten von relativen Verweisen die Gefahr von Missverständnissen zwischen den URL-Filtern von Anwendungen und dem Browser besteht, ist es gute Programmierpraxis, niemals vom Benutzer bereitgestellte relative URLs wörtlich auszugeben. Wenn möglich sollten sie ausdrücklich zu absoluten Verweisen umgeschrieben werden, worauf dann alle Sicherheitsprüfungen an den resultierenden vollqualifizierten Adressen erfolgen.

Spickzettel für Webentwickler

Wenn Sie neue URLs aus Benutzereingaben zusammenstellen

☑ **Wenn Sie im Pfad, im Query-String oder in der Fragment-ID vom Benutzer bereitgestellte Daten verwenden**
Wenn ein Abschnittstrennzeichen ohne korrekte Maskierung durchschlüpfen kann, bekommt die URL möglicherweise eine andere Bedeutung, als Sie vorgesehen hatten (beispielsweise die Verknüpfung einer der HTML-Schaltflächen, die der Benutzer sieht, zu einer falschen serverseitigen Aktion). Seien Sie lieber übervorsichtig: Wenn Sie Werte aus Feldern einfügen, die dem Einfluss eines Angreifers unterliegen können, maskieren Sie einfach alles außer alphanumerischen Zeichen durch Prozentzeichen.

☑ **Wenn Sie vom Benutzer angegebene Schemanamen oder Authority Sections verwenden**
Das ist eine der größten Gefahrenquellen für Code-Injection und Phishing! Wenden Sie jeweils die im Folgenden angeführten Regeln zur Validierung der Eingaben an.

Wenn Sie URL-Eingabefilter entwerfen

☑ **Relative URLs**
Lassen Sie sie nicht zu oder schreiben Sie sie ausdrücklich in absolute Verweise um. Alles andere ist höchstwahrscheinlich unsicher.

☑ **Schemaname**
Lassen Sie nur bekannte Präfixe wie *http://*, *https://* oder *ftp://* zu. Verwenden Sie keine Blacklist, denn dies ist extrem unsicher. Und noch etwas: Dank HTML5 kann man XSS nicht nur mit dem Schema `javascript:` provozieren; auch die neue Entität `&colon;` funktioniert seit einiger Zeit in HTML-Attributen. Viele Filter ließen und lassen sich mit dem String `javascript&colon;alert(1)` austricksen – selbst wenn ein String wie `javascript:alert(1)` korrekt blockiert oder entfernt wird.

☑ **Authority Section**
Hostnamen sollten nur alphanumerische Zeichen, »-« und ».« enthalten, und darauf dürfen nur »/«, »?«, »#« und das Ende des Strings folgen. Wenn Sie irgendetwas anderes zulassen, kann das böse Folgen haben. Falls Sie den Hostnamen untersuchen müssen, stellen Sie sicher, dass Sie den Teilstring rechts davon korrekt bestimmen.

In seltenen Fällen kann es notwendig sein, Vorsorge zu treffen für IDNA, die IPv6-Schreibweise in eckigen Klammern, Portnummern oder HTTP-Anmeldeinformationen in der URL. In einem solchen Fall müssen Sie die URL komplett analysieren, alle Abschnitte validieren, jegliche anomalen Werte

ablehnen und die URL anschließend wieder als eindeutige, kanonische und sauber maskierte Darstellung serialisieren.

Wenn Sie in URLs empfangene Parameter decodieren

☑ Gehen Sie nicht davon aus, dass irgendein Zeichen maskiert wird, nur weil das in den Standards so verlangt wird oder weil Ihr Browser das tut. Bevor Sie irgendwelche aus URLs bezogene Werte zurückgeben oder in Datenbankabfragen, neuen URLs usw. einfügen, untersuchen Sie sie sorgfältig auf gefährliche Zeichen.