

HANSER

Johann-Peter Hartmann, Björn Schotte

Enterprise PHP 5

Serviceorientierte und webbasierte Anwendungen für den
Unternehmenseinsatz

ISBN-10: 3-446-22563-3

ISBN-13: 978-3-446-22563-3

Leseprobe

Weitere Informationen oder Bestellungen unter
<http://www.hanser.de/978-3-446-22563-3>
sowie im Buchhandel.

4 Web 2.0 Security

4.1 Schöne neue (Hacker-)Welt

Netzcomputer, Slim Clients, Rich Internet Applications – diese Stichwörter werden mit den Online-Applikationen, die unter dem Stichwort „Web 2.0“ gehandelt werden, nach Jahren der Ankündigung und des Abwartens zu tatsächlichen Standards.

Bis dahin machten die Nutzer dem Webentwickler das Leben schwer. Er musste alte Browser-Versionen unterstützen, denn viele Konsumenten hatten einen nach Internetmaßstäben uralten Computer im Einsatz. Irgendwann zwischen 2004 und 2006 war es dann so weit. Draußen auf den Bildschirmen standen genug moderne Browser wie der Internet Explorer 6, Firefox oder Opera zur Verfügung. Oft war der Wechsel auf Windows XP ausschlaggebend, oft war der Nutzer aber auch aus Sicherheitsgründen dazu gezwungen, nicht nur das Betriebssystem, sondern auch den Browser auf einen aktuellen Stand zu bringen.

Zeitgleich verblichen die alten Vorgaben für eine gute Webapplikation. Es war in Ordnung, wenn JavaScript vorausgesetzt wurde. Wenn Netscape 3.0 nicht mehr unterstützt wurde, kam nicht mehr automatisch eine Beschwerde aus der Technikabteilung des Kunden. Viel zu viele große Sites setzten Cookies pauschal voraus, als dass man noch lange über deren Nutzung nachdenken müsste.

Dem Nutzer wurde diese Investition in Hardware und aktuelle Software in Komfort und Möglichkeiten heimgezahlt. Wo er sich früher durch viele Formulare klicken musste, passierte plötzlich alles auf einer einzigen Seite, in der Hälfte der Zeit. Seiten wie Google Mail, Yahoo Maps oder Flickr überzeugten durch einfache, intuitive Bedienung und kleine Einstiegsschwellen. Der große Featurekatalog wurde von Tools mit einfacher und intelligenter Bedienung geschlagen. Was Apple mit OS X im Betriebssystem vorgemacht hatte, sollte sich auch für die Online-Welt bewahrheiten: Usability und Spaß an der Bedienung schlägt Möglichkeiten und Flexibilität bei Weitem.

Es ist eine alte Weisheit, dass Security proportional zu Komplexität und Möglichkeiten ist. In ein Haus mit drei Türen und zwanzig Fenstern ist einfacher einzubrechen als in einen Bunker mit nur einer Tür. Eine Webapplikation, die aus nur einem Formular besteht, bietet weniger Angriffsfläche als eine reiche Internetapplikation, die nicht nur 20 unterschiedliche Masken, sondern auch Ajax-Schnittstellen und Webservices zur Verfügung stellt. Das heißt nicht, dass der Entwickler der Ajax-Anwendung per se zum Dilettantentum neigt – aber es wird ihm deutlich einfacher gemacht.

Auch die Hacker haben die neue, breite Angriffsfläche für sich entdeckt. Während ein XSS bislang im Wesentlichen für Demonstrationen taugte, war es auf einmal eine Waffe, mit der man nicht nur Hunderttausende von Nutzeraccounts im Handstreich erobern konnte, sondern damit auch die Macht in Händen hält, jungen Online-Unternehmen auf diese Weise die komplette Existenzgrundlage zu entziehen.

4.1.1 Alles beim Alten?

4.1.1.1 Klassische Sicherheit bei Webapplikationen

Sicherheit, das ist eine Sache der Administratoren, der Unix- und Netzwerkurus. Für uns Anwendungsentwickler spielt sie keine Rolle – schließlich handelt der Nutzer der Applikation in seinem eigenen Interesse, wenn er seinen PC durch eine Fehleingabe in der Anwendung nicht zum Absturz bringt.

In dieser inzwischen nicht mehr ganz vertrauenswürdigen Tradition entstanden jahrzehntelang Tausende von Anwendungen in Cobol, in Visual C++, in Delphi und C#. Sicherheit, das ist ein Problem anderer Leute.

Auf dies bequeme Bett waren die Anwendungsentwickler leider nur bis Mitte der Neunzigerjahre gebettet, denn im Internet ist die Zurechnungsfähigkeit des Nutzers genauso wenig vorauszusetzen wie seine prinzipielle Gutartigkeit. Der Nutzer der Anwendung kann von jedem Punkt der Welt aus auf die Applikation zugreifen und dabei beliebig böse sein – und sich dabei anonym der Verfolgung entziehen.

Die ersten Webanwendungen, meist in Perl programmierte CGI-Lösungen, gingen noch von dem im Eigeninteresse handelnden netten Nutzer aus – und wurden, sobald sie einen nötigen Verbreitungsgrad erreicht hatten, mit dem Ausnutzen von Sicherheitslücken dafür belohnt.

Ende des letzten Jahrhunderts wurden Webanwendungen dann zu einer Standardform von Software, und niemand musste sich mehr vor einem irritierten Blick aus der Geschäftsleitung fürchten, weil er vorgeschlagen hatte, die Reisekostenabrechnung doch durch ein Webinterface bereitzustellen.

Dazu kam PHP – das „Basic der Neuzeit“, wie der Vater von PHP, Rasmus Lerdorf, richtig beobachtet hat. Bei fast jedem Provider verfügbar und schnell zu erlernen, entstand nicht nur ein immenser Berg an Applikationen für jeden erdenkbaren Zweck, sie wurden auch von sehr vielen Nutzern tatsächlich installiert.

Weil PHP so einfach zu benutzen ist, kamen Entwickler wie ihre Klientel aus allen drei Lagern: Profis, semiprofessionelle Anwender und Amateure installierten nicht nur Skripte bei ihrem Host, sondern erweiterten und programmierten auch selbst.

Eine Software ist vor allem dann sicher, wenn sie niemand benutzt. Diese Erfahrung musste Linux genauso machen wie Mac OS X: Kaum ist eine bestimmte Menge Nutzer vorhanden, folgen ihr auch Fehler, Sicherheitsprobleme und Lücken.

In die gleiche Falle stapfte auch PHP, nur dass hier sehr viel mehr Heimwerker aktiv an der Herstellung beteiligt waren. So folgte der Popularität der Sprache bei Entwicklern und Nutzern die der Hacker auf den Tritt, und schnell spiegelte sich der Anteil an neu geschaffenen Anwendungen auch auf den einschlägigen Security-Foren und -Listen wider.

Die Security-Experten begannen, das neue Feld der Web-Security zu beackern, und ernteten viele neue Dinge. Zu alten und bekannten Bugs wie Buffer Overflows und Privilege Escalations gesellten sich Cross Site Scripting, SQL-Injections, Code Inclusions und viele andere.

Die Könige der Exploits waren hier Code Inclusions oder Executions, denn sie gaben dem Nutzer die Möglichkeit, Programmcode unmittelbar auf dem Server auszuführen. SQL-Injections führten ein Schattendasein, wenn sie nicht gerade einen normalen Nutzer in der Datenbank zum Admin machten. Und Cross Site Scripting diente allenfalls dazu, einem Freund einen Link auf einer bekannten Site zu zeigen, der ein JavaScript-Alert-Popup erzeugte.

Natürlich haben wir Entwickler auf jede Herausforderung eine Antwort gefunden. Alles, was in die Datenbank geschrieben wird, wird vorher am besten mit einer von der Datenbank gestellten Funktion `escaped`. Das Gleiche gilt für Argumente, die der Kommandozeile etwa durch `exec()` in die Hand gegeben werden. XSS zeigte sich als endloses Rennen zwischen dem unbegrenzten Reichtum an komischen Dingen, die ein Browser noch zu interpretieren in der Lage ist, und immer ausgefeilteren Ein- und Ausgabefiltern.

Die Faustregel hinter allen diese Maßnahmen könnte von McCarthy stammen: Allen Dingen, die von außen kommen, ist erst einmal zu misstrauen.

Erst nach passender und detaillierter Filterung, Prüfung und Anpassung dürfen sie intern verwendet werden.

4.2 XSS und CSRF oder neue Probleme mit alten Bekannten

4.2.1 Cross Site Scripting (XSS)

Eigentlich hätte es CSS heißen müssen, doch diese Abkürzung war aber schon durch die Cascading Stylesheets belegt, und so griff man auf die amerikanische Abkürzung X für

Cross zurück. XSS erblickte das Licht der Welt, kurz nachdem 1995 mit dem Netscape Navigator 2 der erste Browser herauskam, der Skripte auf dem Client ausführen konnte. Die ursprünglichen Attacken waren so aufgebaut, dass der Nutzer auf Server A eine Seite besuchte, die ohne sein Wissen auf Server B Skripte auf Server B einschmuggelte, die der Browser dort mit den Rechten des Nutzers ausführte. So kam es zu der Formulierung „Cross Site“, die in den wenigsten Fällen die Attacken, die heute unter dem Begriff geführt werden, beschreibt. Ein besserer Name wäre wohl „JavaScript Injections“ gewesen.

Kern aller XSS-Angriffe ist die Möglichkeit, auf einer Webseite eigenen JavaScript-Code einzuschleusen, der dann vom Browser ausgeführt wird. Eine andere Seite muss dabei aber in keiner Weise beteiligt sein; es reicht völlig aus, dass der Nutzer einen gefälschten Link anklickt oder auf eine Seite trifft, die bereits vorher mit einem fremden Skript infiziert wurde.

Es wird zwischen drei Varianten beim Cross Site Scripting unterschieden:

4.2.1.1 Dom-basierte XSS

DOM-basierte Attacken brauchen eigentlich gar keinen Server, sie können auch auf einer einfachen HTML-Seite stattfinden. Voraussetzung ist, dass auf der Seite JavaScript ausgeführt wird, das auf Parameter zugreift, die vom Nutzer geändert werden können. Als Beispiel eine kleine HTML-Seite, deren Titel und Inhalt bequem über den Anchor-Part der URL gesetzt werden kann, um dem Server Arbeit zu ersparen.

Listing 4.1 Einfaches Skript, das den Kapitelnamen per JavaScript aus der URL ermittelt

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
  <title>[Wird durch Kapitelnamen ersetzt.]</title>
</head>
<body>
  <script type="text/javascript" language="JavaScript1.2">

      // Kapitelnamen aus der URL ermitteln
      var start = document.URL.indexOf('#')+1;
      var ende = document.URL.length;
      var name = document.URL.substring(start, ende);
      var kapitel = decodeURIComponent(name);
      // Kapitelnamen als Dokumententitel setzen
      document.title = kapitel;
      // Kapitelnamen als Überschrift setzen
      document.write('<h1>Kapitel '+kapitel+'</h1>');

  </script>
</body>
</html>
```

Ruft man dieses Skript über eine URL wie `http://localhost/dom_xss.html#Dom-based%-20XSS` auf, dann wird automatisch der Dokumententitel auf „Dom-based XSS“ gesetzt und eine dementsprechende Überschrift angefügt – und das ganz ohne Serverinteraktion. Allerdings wird hier der Name des Kapitels per `document.write` in das Dokument ge-

geschrieben. Dabei wird es als ganz normales HTML behandelt und kann damit auch Skripte enthalten. Auf diese Weise enthält diese Seite ein Cross-Site-Scripting-Problem, obwohl es sich nur um statisches HTML handelt.

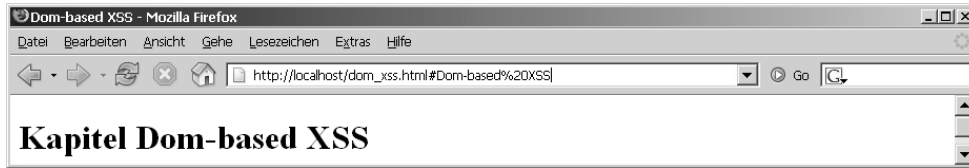


Abbildung 4.1 Dynamisches HTML ohne PHP

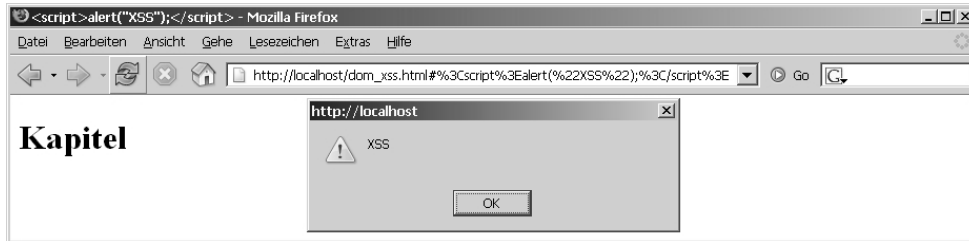


Abbildung 4. Dynamisches HTML ohne PHP, aber mit XSS

Die häufigste Form von Typ 0 XSS basiert auf der escaped Verwendung von `location.search`. `location.search` enthält den Query-String der aktuellen URL, beginnend mit dem Fragezeichen. Dieser Parameter wird gerne verwendet, um auf ihm basierend bestimmte Unterframes in einer Frameseite auszuwählen. Das Frameset wird in diesem Fall komplett über die Methode `document.write()` neu aufgebaut, und falls der Query-String ungefiltert verwendet wird, ist ein Setzen des Onload-Events des aktuellen Frames oder Framesets möglich. Typ 0 XSS ist aus zwei Gründen besonders gefährlich. Zunächst erwartet ein Entwickler auf statischen Seiten keine Sicherheitsprobleme, daneben können auch lokale HTML-Dateien einen XSS enthalten – und das mit weitreichenden Konsequenzen.

4.2.1.2 Nichtpersistente, reflektierte XSS-Attacken

Hier handelt es sich um die am weitesten verbreitete Variante von XSS. Über eine manipulierte Nutzereingabe, etwa eine geänderte URL, wird ein Skript auf dem Server dazu gebracht, auf dynamisch erzeugten Seiten JavaScript-Code zu zeigen.

Ein typisches Beispiel sind PHP-Skripte, die direkt in der Ausgabe externe Eingaben verwenden:

Listing 4.2 Das Beispiel von oben, dieses Mal dynamisch vom Server befüllt

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
  <title><?php echo $_GET['kapitel']; ?></title>
</head>
<body>
  <h1>Kapitel <?php echo $_GET['kapitel']; ?></h1>
</body>
</html>

```

In diesem Beispiel lässt sich der Kapitelname über den Get-Parameter „kapitel“ setzen und wird dynamisch in die Seite eingefügt. Wie Sie sehen, wird die Eingabe aus der URL direkt verwendet und nicht vorher gefiltert. Das hat zur Folge, dass hier wieder vollständiger HTML-Code eingeschmuggelt werden kann und damit auch JavaScript-Befehle.

Opfer dieser Variante wurde praktisch jede größere Internetseite, von Yahoo, Google, MySpace bis hin zum Verlag dieses Buches selbst.

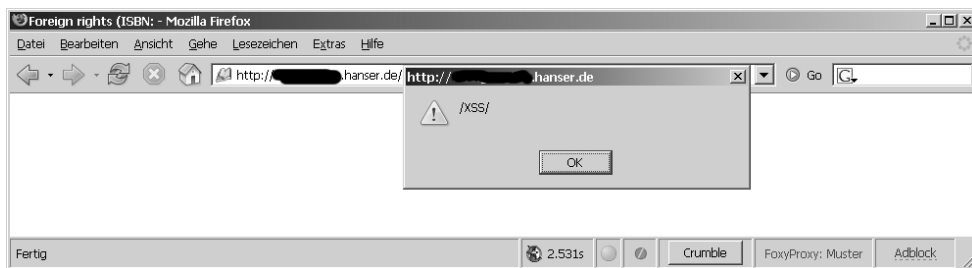


Abbildung 4.2 XSS gibt es überall – auch beim Verlag dieses Buches.

Diese Art XSS-Angriffe werden meist über GET-Parameter durchgeführt, dies ist aber nicht Vorbedingung. XSS können genauso gut über POST-Formulare durchgeführt werden, nur ist hier meist eine separate Seite notwendig, die das Abschicken des Formulars gewährleistet.

4.2.1.3 Persistente XSS-Angriffe

Der dritte Typ von XSS wird auf dem Server gespeichert und steht so für viele Abrufe zur Verfügung. Klassisches Beispiel für diese Art Angriff sind Foren, in deren Postings JavaScript eingeschmuggelt werden kann, in letzter Zeit sind diese Angriffe aber auch vermehrt im Bereich Community-Portale und Blogs aufgetreten. Eine besondere Rolle spielen sie bei web-basierten Würmern und Viren, weil persistente XSS hier die Rolle des Verbreitungsvektors einnehmen.

Persistente XSS-Angriffe entstehen genau dann, wenn eine Seite Eingaben erlaubt, die von der Nutzerseite kommen, und diese auch wieder ausgibt. Das sind typischerweise Texteingaben, es kann sich aber auch um andere, vom Browser mitgelieferte Daten wie etwa den User-Agent oder den Referrer handeln. Die Texteingaben können auch aus der