XAMPP Security

Hannes Kasparick

BAKKALAUREATSARBEIT

Nr. 239-003-011-A

eingereicht am

Fachhochschul-Bakkalaureatsstudiengang

Computer- und Mediensicherheit

in Hagenberg

im Dezember 2005

Diese Arbeit entstand im Rahmen des Gegenstands

Angriffsmethoden und deren Abwehr

im

Wintersemester 2005/06

Betreuer:

DI (FH) Daniel Fabian

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 20. Dezember 2005

Hannes Kasparick

Inhaltsverzeichnis

Erklärung iii

Vorwort vi

Zusammenfassung vii

Abstract viii

1 Die Umgebung des Apache Webservers 1

1.1 Chroot-Käfige 3

1.2 FreeBSD Jails 5

1.3 VServer 6

2 Absicherung der Kernkomponenten 7

2.1 Installation des Apache Webservers 8

2.1.1 Installation 8

2.1.2 Generelle Sicherheitskonfiguration 10

2.1.3 Denial of Service Gefahren 14

2.1.4 Das mod_security Modul 17

2.1.5 VirtualHosts Konfiguration 18

2.2 PHP 21

2.2.1 Sichere Konfiguration von PHP 22

2.2.1.1 Der Parameter open_basedir 24
2.2.1.2 Der Parameter disable_functions 25
2.2.1.3 Die safe_mode Parameter 26
2.2.1.4 Der Parameter register_globals 28
2.2.1.5 Die magic_quotes* Parameter 29
2.2.2 Hardened-PHP 29
2.2.3 suPHP 30
2.3 Perl 31
2.3.1 Sicherheitsrisiko Perl 32
2.3.2 suEXEC 34
2.4 MySQL 35
2.4.1 Sichere Konfiguration 36
2.5 Benutzerverwaltung 37

3 Analyse eines erfolgreichen Angriffs **39**
3.1 Ausgangssituation 39
3.2 Feststellen des Einbruchs 40
3.3 Analyse der Logfiles 41
3.4 Motivation des Angreifers 43
3.5 Gegenmaßnahmen 43

4 Resümee **44**

Literaturverzeichnis **46**

Vorwort

Die vorliegende Arbeit entstand im Zuge des Bakkalaureatsstudiums Computer- und Mediensicherheit (CMS) an der Fachhochschule Hagenberg im Wintersemester 2005 / 2006.

Grund für diese Arbeit war es, ein fundiertes Regelwerk für die sichere Administration eines Apache Webservers zu schaffen. Es gibt zwar schon das ein oder andere Buch oder Paper zu diesem Thema, aber leider werden oft essentielle Dinge vernachlässigt, oder aber sie sind schlicht veraltet. Auch diese Arbeit wird nur für einen begrenzten Zeitraum gültig sein, da die Entwicklung stetig voranschreitet. Die meisten Inhalte entsprechen dem Stand August 2005.

Mit dieser Arbeit möchte ich dem Apache Administrator ein Handbuch übergeben, dass alle essentiellen Gesichtspunkte einer sicheren Apache Installation abhandelt. Aufgrund des Umfangs der Materie, ist es nicht möglich jeden Aspekt bis ins kleinste Detail zu behandeln, daher wird oft auf externe Quellen und Erläuterungen verwiesen. Für das Verständnis dieser Arbeit sind grundlegende Unix- und Apachekenntnisse Voraussetzung.

Ich bedanke mich bei meinem Betreuer, DI(FH) Daniel Fabian, für die Unterstützung durch seine fachlichen Ratschläge und für die vielen Stunden, die er zum Lesen und Korrigieren meiner Arbeit aufgebracht hat. Des Weiteren möchte ich Jan Doberstein danken, der mir mit Erfahrungen aus dem Providerbereich zur Seite stand.

Zusammenfassung

Das WWW, wie es die meisten Menschen kennen, besteht aus HTML Seiten, die mehr oder weniger aufwändig durch zusätzliche Techniken dynamisch und interaktiv gestaltet werden. Die Infrastruktur dahinter besteht aus vielen Komponenten, von zentraler Bedeutung sind jedoch die Millionen von Webservern.

Der Apache Webserver ist dabei am weitesten verbreitet und tritt häufig in Kombination mit zusätzlichen Komponenten wie MySQL, PHP und Perl auf verschiedenen unixartigen Betriebssystemen auf. Die Absicherung eines solchen XAMPP[1]-Systems stellt eine große Herausforderung für Administratoren dar. Diese Arbeit soll alle kritischen Punkte der Konfiguration beleuchten und auf Schwachstellen hinweisen.

Sicherheit ist heutzutage auch ein Kostenfaktor. Sicherheit darf nur selten größere Zusatzkosten verursachen, daher werden verschiedene technische Ansätze verglichen, die einen unterschiedlich hohen Grad an Sicherheit und Kosten mit sich bringen. Im Verlauf der Arbeit wird auf verschiedene Open Source Lösungen eingegangen, welche nur in seltenen Fällen Lizenzkosten mit sich bringen. Diese ermöglichen einen sicheren und preiswerten Betrieb von Webservern.

[1] Unix / Linux, Apache, MySQL, PHP, Perl

Abstract

The World Wide Web known by most people is an extensive collection of HTML websites which are augmented by additional technologies that enable dynamic and interactive content. The infrastructure behind this consists of several components, web servers arguably being the most important layer.

Apache is the most common web server and often appears in combination with additional components such as MySQL, PHP and Perl on different Unix-like operating systems. The protection of such a XAMPP[2]-system is a big challenge for an administrator. This thesis aims to address all critical aspects of a configuration and relevant vulnerabilities.

Today security is also a major expense factor. However, the budget for security in many companies is usually fairly limited. For this reason, different technical approaches are compared which represent a different level of security and costs. In the course of this paper, a number of open source solutions are suggested that have very low license fees. This makes it possible to build and run fairly secure and inexpensive web servers.

Today security is also an expense factor. Only seldom security may cause additional costs, therefore different technical approaches are compared to each other which have a different level of security and costs. During this paper different open source solutions are suggested which only cause very few licence costs. This allows to build up an run fairly secure and inexpensive webservers.

[2]Unix / Linux, Apache, MySQL, PHP, Perl

Kapitel 1

Die Umgebung des Apache Webservers

Um einen Webserver sicher betreiben zu können, ist ein sicher konfiguriertes Betriebssystem als Grundlage notwendig. Hier stellt sich gleich zu Beginn die Frage, welches Betriebssystem überhaupt verwendet werden sollte. Im Unixbereich gibt es unzählige Variationen von Betriebssystemen. Die im Moment am stärksten wachsende Gruppe sind die verschiedenen Linux Distributionen. Hier tun sich allerdings sehr große Qualitätsunterschiede beim mittel- und langfristigen Support auf. Kostenpflichtige Enterprise Distributionen stehen nichtkommerziellen gemeinnützigen Gesellschaften gegenüber. Vor- und Nachteile müssen der Situation entsprechend gegeneinander abgewogen werden und werden hier nicht weiter besprochen.

Auf der anderen Seite gibt es die traditionellen Unixe sowie verschiedenste BSD-Derivate. Welche Plattform eingesetzt wird und welche Maßnahmen zur Absicherung ergriffen werden müssen, ist nicht Thema dieser Arbeit. Die Auswahl und die Absicherungsmöglichkeiten der unterschiedlichen Plattformen durch zusätzliche Komponenten sind dermaßen umfangreich, dass eine qualitativ hochwertige Abhandlung im Rahmen dieser Arbeit unmöglich ist.

Auch sorgfältig und sicher aufgesetzte Webserver können Opfer eines Angriffs werden. Dies könnte beispielsweise durch eine nicht sofort behobene, aber öffentlich bekannte, Sicherheitslücke passieren. Auch wenn Sicherheitsupdates oft schnell verfügbar sind, so kann es doch passieren, dass der Administrator gerade im Urlaub ist und die Vertretung keine Updates an kritischen Systemen vornehmen will. So bietet sich dem Angreifer unter Umständen ein ausreichend großes Zeitfenster, um den Webserver zu kompromittieren. Eine weitere Möglichkeit für den Angreifer wäre es, selbst eine Schwachstelle im Webserver zu finden und diese auszunutzen, ohne dem Hersteller die Chance zu geben den Fehler zu beheben.

Schafft es ein Angreifer, über einen Fehler im Webserver Zugang zum System zu erreichen, kann er sich mit den Rechten des Webservers frei im Dateisystem bewegen. Ein möglichst sicher installiertes Betriebssystem hindert ihn eventuell daran, größere Zerstörung anzurichten. Er besitzt allerdings die Rechte des Webservers, welche es ihm ermöglichen lesend auf große Datenbestände zuzugreifen. Um dies zu verhindern gibt es einige Techniken, den Apache Webserver in ein virtuelles Gefängnis zu sperren und ihm so einige der Zugriffsrechte zu nehmen. Diese virtuellen Gefängnisse werden oft als *Chroot* bezeichnet, auch wenn sie andere Verfahren als das klassische *chroot* einsetzen.

Die im Folgenden vorgestellten Konzepte, den Webserver einzuschränken, eignen sich leider oftmals nicht für den Masseneinsatz bei großen Providern. Sie eignen sich eher für einzelne Firmenwebserver mit wenigen Zugriffsberechtigten. Beim Massenhosting besteht immer das Problem, dass sich mehrere Kunden einen Webserver teilen müssen. Um die Daten der einzelnen Kunden effektiv zu schützen eignet sich das Einsperren des gesamten Webservers nicht. Techniken, die Kunden auf einem Webserver voneinander zu trennen versuchen, werden im Kapitel 2 beschrieben.

Ein weiterer Nachteil der nachfolgend vorgestellten Lösungen ist die zunehmende Komplexität, die sich durch die Module bzw. die Interaktion zwischen Webserver und anderen Programmen ergibt. Die vorgestellten Techniken trennen den Webserver möglichst weit vom darunter liegenden Betriebssystem weshalb alle essentiellen Teile zur Ausführung des Webservers in sein virtuelles Gefängnis mit installiert werden müssen. Dies führt teilweise dazu, dass Sicherheitsupdates vom Hersteller nicht mehr so einfach installiert werden können, da der Paketmanager die Struktur und Inhalte der erstellten Umgebung nicht kennt.

Kritiker behaupten, dass ein qualifizierter Angreifer aus virtuellen Umgebungen schneller ausbrechen kann, als der Administrator benötigt, um sie einzurichten. Diese Bedenken sind nicht ganz unberechtigt, gerade unter dem im Webserverbereich weit verbreiteten Linux Betriebssystemen ist es traditionell relativ einfach aus Chroot-Käfigen auszubrechen [5, Kapitel 8]. Diese Gefahr besteht prinzipiell aber auch bei jeder anderen der vorgestellten Techniken, da die virtuellen Umgebungen immer mit dem darunter liegenden Betriebssystem kommunizieren und interagieren müssen und so eine gewisse Angriffsfläche bieten. Trotz der angeführten Bedenken ist das Einsperren des Apache Webservers inklusive seiner Komponenten eine sehr wirkungsvolle Maßnahme zum Schutz des gesamten Servers.

1.1 Chroot-Käfige

Chroot-Käfige sind in unixartigen Betriebssystemen eine seit langem existierende Technik um Prozesse einzusperren und vom darunter liegenden Betriebssystem abzutrennen [24, Kapitel 16.5]. Dazu installiert man das potentiell gefährdete Programm zusammen mit den benötigten Bibliotheken in ein Verzeichnis und startet das Programm, mit Hilfe des *chroot*-Tools, in diesem. Für das Programm ist alles außerhalb des erstellten Verzeichnisses nicht zu sehen, sein root-Verzeichnis ist also ein anderes, als das des realen Betriebssystems. In [23, Kapitel 2.4] wird ausführlich beschrieben wie man Apache zusammen mit den benötigten Bibliotheken in ein Chroot installiert. Diese Art der Installation wird im weiteren Verlauf als „externes Chroot“ bezeichnet.

Einfacher haben es Benutzer des OpenBSD Betriebssystems. Hier wird Apache standardmäßig in einem Chroot vorinstalliert ausgeliefert. Der Administrator braucht sich also nicht mehr darum zu kümmern, welche Bibliotheken in welcher Version vom Apache Webserver benötigt werden. Auch die Installation von PHP in ein Chroot hinein gestaltet sich sehr einfach, da das vom Paketmanager installierte Paket für den Einsatz im Chroot vorbereitet ist. Wer also auf eine einfache und gut wartbare Chroot Installation Wert legt, sollte sich OpenBSD etwas näher ansehen, nicht zuletzt wegen seiner allgemein sehr guten Sicherheitspolitik[1]. Mittlerweile gibt es allerdings auch für einige Linux Distributionen, wie zum Beispiel Debian oder Gentoo vorgefertigte Apache-Chroot Pakete.

Bei der Installation von Skriptsprachen wie PHP und Perl sollte darauf geachtet werden, dass sie teilweise direkt auf zusätzliche Programme zugreifen. Ein Beispiel wäre das Sendmail Binary, welches von CGI Skripts zum Versenden von E-Mails verwendet wird. Das Gleiche gilt für die Verbindung zu Datenbanken, die außerhalb des Chroot laufen. Wenn versucht wird über den localhost auf eine Datenbank zuzugreifen, so würde das Skript über den Unix Socket versuchen eine Verbindung aufzubauen. Dies schlägt allerdings fehl, weshalb TCP/IP Sockets verwendet werden müssen. Eine andere Möglichkeit wäre es, die entsprechenden Unix Sockets in das Chroot zu legen. Letztere Möglichkeit bietet potentielle Ausbruchsmöglichkeiten und sollte daher nicht gewählt werden.

Ein großer Vorteil dieser Chroot-Käfige ist, dass sie nur minimale Funktionalität bieten. Angriffe die Systembefehle nutzen, können in einer Chroot-Umgebung aufgrund der fehlenden Binärprogramme nicht ausgeführt werden. Jegliche Funktionalität in einem Chroot muss vom Administrator bewusst hinzugefügt werden, was dem guten Grundsatz entspricht „alles was

[1] http://www.openbsd.org/security.html

nicht explizit erlaubt ist, ist verboten“.

Zusätzlich zur bisher vorgestellten Technik des externen Chroot gibt es mehrere Möglichkeiten für ein internes Chroot. Hierbei braucht sich der Administrator um weitaus weniger Dinge kümmern, da spezielle Patches bzw. Module für den Apache Webserver für die entsprechenden Vorkehrungen sorgen.

Eine Möglichkeit den Apachen selbst mit Chroot Funktionalität auszustatten ist der chroot(2) Patch, welcher unter [30] inklusive Dokumentation zur Verfügung steht. Er stellt einen zusätzlichen Parameter für die Konfigurationsdatei zur Verfügung, die *ChrootDir-Direktive*. Durch diesen Parameter kann für jede Directory-Direktive ein Chroot definiert werden. Dieser Patch hat jedoch einen großen Nachteil: er scheint nicht sehr gut gepflegt zu sein, da nur ein Patch für Apache Version 1.3.31 existiert. Aktuell ist aber Version 1.3.33. Eher noch zu verschmerzen ist die Tatsache, dass zwei File Deskriptoren mit Schreibzugriff auf Dateien außerhalb des Chroot geöffnet bleiben. Sie zeigen auf die Zugriffs- und Fehlerlogfiles.

Weitere Möglichkeiten bieten sich über die Apache Module mod_security[2] und mod_chroot[3]. Beide Module sind für alle gängigen Plattformen in der aktuellen Version verfügbar und bauen auf dem chroot(2) Patch auf. Der entscheidende Vorteil bei den zuletzt genannten Modulen ist, dass der Apachequellcode nicht gepatcht werden muss, was unter anderem die Installation von Sicherheitsupdates erheblich erleichtert. Mod_chroot bietet ausschließlich Chroot-Funktionalität, während mod_security viele weitere nützliche Features mitbringt und damit potentiell fehleranfälliger ist.

Beide Module erstellen ein Chroot für den gesamten Webserver. Es ist leider nicht möglich Chroots für *<Directory>*, *<Files>*,*<Location>*, *<VirtualHost>* oder *.htaccess* einzusetzen. Zu beachten ist, dass der *DocumentRoot* Parameter entsprechend angepasst werden muss, da der Webserver ansonsten die Dateien nicht mehr finden kann. Nachdem die Module das Chroot eingerichtet haben, ist es nicht mehr möglich auf Bereiche außerhalb des definierten Verzeichnisses zuzugreifen. PHP quittiert den Zugriff auf externe Bereiche wie erwartet mit

> Warning: main(/etc/passwd): failed to open stream: No such file or directory in /index.php

Das Chrooten des Apache Webservers mit Hilfe von Modulen stellt eine komfortable und schnell zu realisierende Möglichkeit dar, um die Zugriffsrechte des Webservers auf einen kleinen Teil im Dateisystem zu beschränken. Dadurch, dass keine „richtigen“ Chroot Umgebungen konstruiert werden, ist es

[2] http://www.modsecurity.org
[3] http://core.segfault.pl/~hobbit/mod_chroot/

weiterhin problemlos möglich, den Server mit den gewohnten Updatetools aktuell zu halten. Außerdem braucht sich der Administrator nicht um jedes einzelne Chroot zu kümmern. Will der Administrator den Webserver noch strikter vom eigentlichen Betriebssystem trennen und einen höheren Grad an Sicherheit erreichen, so bieten sich unter FreeBSD die so genannten Jails an.

1.2 FreeBSD Jails

Jails (engl. Gefängnis) unter FreeBSD erschaffen eine virtuelle Umgebung, welche (fast) vollkommen isoliert vom Hostbetriebssystem existiert und damit deutlich flexibler sind als herkömliche Chroot-Umgebungen [1]. Veränderungen innerhalb einer Jail können das Hostsystem nicht verändern, es sei denn, dies wird vom Administrator explizit erlaubt. Innerhalb einer Jail kann der Administrator sogar einem Kunden volle Root-Rechte geben, da dies kein Risiko für seinen Server darstellt. Jails können sich auch untereinander nicht beeinflussen, sodass es möglich ist, viele virtuelle FreeBSD Systeme auf einem physischen Server laufen zu lassen. Eine Jail ist dabei nicht mit einem virtuellen Server, wie sie VMware oder VirtualPC bereitstellen zu vergleichen, da sie keinen kompletten PC emulieren und es einige weitere Einschränkungen gibt [11].

Jeder Jail kann nur eine IP-Adresse zugewiesen werden, das bedeutet, dass die oft verwendete Kommunikation über das Loopback Interface (127.0.0.1) nicht möglich ist. Einige Dienste, wie Samba oder NFS, funktionieren ebenfalls nicht problemlos. Für Apache Webserver sind Jails jedoch bestens geeignet.

Jails erhöhen die Sicherheit des Apachen an sich nicht, allerdings lassen sich durch Jails die Ausfallzeiten nach einem Einbruch erheblich verkürzen. Nach einem Einbruch ist es oft schwer festzustellen, was der Angreifer alles verändert hat, ob Backdoors installiert oder Konfigurationsdateien verändert wurden. Um den Betrieb des Webservers möglichst schnell wieder herzustellen wird einfach eine neue Jail gestartet und eine saubere Datensicherung hinein kopiert. Danach kann sich der Administrator in aller Ruhe mit dem Einbruch in die Jail beschäftigen und herausfinden, wie der Angreifer es geschafft hat in das System einzubrechen, um danach entsprechende Sicherheitsmaßnahmen zu treffen.

Wie jede Sicherheitsmaßnahme, die der Administrator zum Schutz seiner Daten und Server ergreift, sind auch Jails kein Allheilmittel, denn auf einschlä-

gigen Mailinglisten[4] und Sicherheitsseiten[5] im Internet finden sich hin und wieder Hinweise auf Sicherheitslücken im Zusammenhang mit Jails. Diese werden zwar sehr schnell gefixt, zeigen aber einmal mehr, dass es keine absolute Sicherheit gibt. Ein ähnliches Konzept wie die Jails unter FreeBSD gibt es natürlich auch für Linux.

1.3 VServer

Das Linux VServer Projekt[6] verfolgt ganz ähnliche Ziele und Wege wie Jails unter FreeBSD. Es erlaubt ein Linux innerhalb eines Linux laufen zu lassen, wobei jeder virtuelle Linux Server seine eigenen Pakete, Dienste, Benutzer und IPs hat. Sie laufen mit dem selben Kernel wie das Hostbetriebssystem und es gibt keinerlei Performanceeinbußen. Es ist sogar möglich in VServern andere Linux Distributionen zu installieren als das Hostsystem. Außerdem ist es leicht möglich Ressourcenbeschränkungen für die Kunden festzulegen.

Die Installation eines Apache Webserver auf einem VServer System bietet im Prinzip die gleichen Vorteile wie die Installation in eine Jail. Sollte der Webserver kompromittiert werden, so kann der Angreifer nur Schaden im VServer anrichten, nicht jedoch auf dem Hostbetriebssystem oder in anderen VServern. VServer werden heutzutage oft als „kleine“ Rootserver vermietet, stellen aber auch eine kostengünstige Möglichkeit zur Trennung von verschiedenen Apacheinstanzen und Kunden auf einem physischem Server dar. Des Weiteren kann man damit sehr leicht Testumgebungen realisieren.

Abschließend sei gesagt, dass alle vorgestellten Maßnahmen die Gesamtsicherheit des Webservers erhöhen. Welche Lösung angewendet wird, muss allerdings immer individuell entschieden werden.

[4] http://www.seclists.org
[5] http://www.securityfocus.com
[6] http://linux-vserver.org/

Kapitel 2

Absicherung der Kernkomponenten

Wie bereits erwähnt wurde, ist Apache der mit Abstand am weitesten verbreitete Webserver im Internet. Mit einem Marktanteil von fast 70%, weit vor dem IIS von Microsoft mit gut 20%, stellt der Apache Webserver bzw. die darauf gehosteten Webseiten ein beliebtes Angriffsziel für Hacker und Cracker dar. Diese verändern (defacen) Webseiten teilweise als Hobby und mit einem gewissen Sportsgeist, wie die Statistiken unter [12] zeigen. Dort werden aber keine Statistiken über die angegriffenen Webserver geführt, sondern nur über die angegriffenen Betriebssysteme auf denen die Webserver laufen.

Das Verändern von statischen HTML Seiten auf einem Webserver ohne serverseitige Skripttechnologien ist nur über Fehler im Apache selbst, oder anderen Diensten des Servers möglich. Bei ordnungsgemäß installierten und gewarteten Servern bieten sich für einen Angreifer kaum Möglichkeiten, Zugriff auf die Webseiten zu erlangen, um sie zu verändern. Es gibt allerdings nur noch sehr wenige Webserver, die nicht über serverseitige Skriptsprachen wie PHP oder Perl verfügen. Im Hintergrund wird zudem oftmals die sowohl kostenlos als auch kommerziell verfügbare Datenbanksoftware MySQL eingesetzt. Dieses Kapitel wird daher die Absicherung des Apache Webservers mit PHP, Perl und MySQL ausführlich erklären.

Vor der Installation des Webservers sollte auf jeden Fall für die nötige Hostsicherheit gesorgt sein. Alleine darüber lassen sich Bücher schreiben, dem interessierten Leser seien [24] und [3] als weiterführende Lektüre empfohlen. Nach der Auswahl eines möglichst sicheren Betriebssystems und einem entsprechendem Hardening können die gewünschten Komponenten des Webservers installiert werden. Diese werden entweder vom Hersteller direkt, oder

über vorgefertigte Pakete des Distributors bezogen. Beim Hersteller sind die originalen Quelltexte erhältlich, während dessen die Distributoren oft zusätzliche Patches für eine erweiterte Funktionalität oder zusätzliche Sicherheit integrieren.

Die Auswahl der Quellen sollte wohl überlegt sein. Zwar bieten Herstellerquellen die meiste Kontrolle und Flexibilität, allerdings gestaltet sich das Einspielen von Sicherheitsupdates oft schwierig. Diese Probleme steigen umso mehr, je mehr Komponenten installiert sind. Von daher ist eine umfassende Dokumentation unerlässlich. Installiert man die Webserversoftware und die dazugehörigen Komponenten aus den Quellen des Distributors, so garantiert dies im Normalfall ein problemloses Update bei Sicherheitsproblemen. Viele Distributoren stellen sowohl Binärpakete als auch Quelltexte zur Verfügung, sodass man, falls gewünscht, sehr detailliert Anpassungen vornehmen kann. In einigen Betriebssystemen, wie zum Beispiel OpenBSD, ist der Apache Webserver bereits vorinstalliert. Er ist mit zusätzlichen Sicherheitspatches ausgestattet und man sollte sich daher gründlich überlegen, ob man solche Systeme durch eigene Installationen ersetzen will. Grundsätzlich gilt, dass nicht jedes verfügbare Feature installiert und genutzt werden sollte, da dies die Komplexität und Störanfälligkeit des Gesamtsystems erhöht.

Damit der Webserveradministrator möglichst schnell von Sicherheitsrisiken erfährt, sollte er unbedingt die Security-Announce Mailinglisten des Betriebssystemherstellers und der beteiligten Webserverkomponenten abonnieren. Sie empfehlen sehr schnell erste Workarounds wenn Sicherheitsprobleme bekannt werden und benachrichtigen, wenn entsprechende Updates verfügbar sind.

2.1 Installation des Apache Webservers

Vor der Installation sollte festgelegt werden, was genau die Anforderungen an den Webserver sind. Danach können geeignete Installationsquellen ausgewählt werden, zusätzlich Sicherheitspatches integriert, sowie geeignete Module nachinstalliert werden. Dabei sollte nie das KISS-Prinzip (Keep it small & simple) aus den Augen verloren werden. Der Apache Webserver bietet eine Vielzahl an Konfigurationsmöglichkeiten an, die den eigenen Bedürfnissen angepasst werden sollten.

2.1.1 Installation

Am einfachsten und schnellsten gestaltet sich die Installation, wenn Binärpakete mit Hilfe des im Betriebssystem integrierten Paketmanagers installiert werden. Er löst alle Abhängigkeiten mit anderen Paketen auf und installiert

die nötigen Bibliotheken in den richtigen Versionen. Nach wenigen Minuten präsentiert sich ein lauffähiger Webserver. Eventuell müssen noch der Hostname und andere Kleinigkeiten angepasst werden, damit Fehlermeldungen beim Start behoben werden, aber prinzipiell ist der Webserver innerhalb kürzester Zeit einsatzbereit. Etwas umfangreicher ist die Installation aus den Quellen. Welche Module geladen werden, kann der Administrator entweder über Konfigurationsdateien oder symbolische Links festlegen. Die Vorgehensweise unterscheidet sich teilweise deutlich in den verschiedenen Betriebssystemen, ist aber letztendlich nicht schwer zu verstehen.

Hat man sich für die Installation aus den Quellen entschieden, zum Beispiel weil Performanceoptimierungen nötig sind, sollte nach dem Download zumindest die MD5-Prüfsumme mit der vom Apache Downloadserver verglichen werden. Wer sicher gehen will wirklich die echten Apache Quelltexte heruntergeladen zu haben, kann dies mit Hilfe der GnuPG Signaturen überprüfen, welche ebenfalls auf den Downloadservern bereit stehen. Nachdem die Quelltexte entpackt wurden, müssen eventuelle Sicherheitsupdates eingepflegt werden, da nicht immer sofort neue Komplettpakete erstellt werden, wenn es Sicherheitsprobleme gibt.

Die nächste Entscheidung, die wohl überlegt sein will, bestimmt, ob man Apache als statisches Binary kompiliert oder dynamisch nachladbare Module verwendet. Statische Installationen sind geringfügig schneller, was aber bei der heute vorhandenen Rechenleistung nahezu irrelevant ist. Ein statisch kompilierter Apache hat allerdings den klaren Vorteil, dass keine Backdoor Module nachgeladen werden können. Um bei einem statischen Apache Backdoors zu installieren, müsste der Angreifer den kompletten Webserver neu kompilieren. Diese Art der Installation kann auch für den Administrator von Nachteil sein, denn jedes Mal, wenn er etwas an einem Modul ändern oder ein Neues hinzufügen will, muss er den kompletten Apachen neu kompilieren.

Für den Apache in der Version 1.3 sind auf Packetstormsecurity[1] einige Backdoor Module verfügbar, beispielsweise mod_backdoor.c oder verschiedene Versionen von mod_rootme. Sie bieten unterschiedliche Funktionalitäten. So erlaubt mod_rootme über GET Requests eine Shell mit root-Rechten auszuführen, ohne dass irgendetwas geloggt wird.

Fällt die Entscheidung zu Gunsten der statischen Kompilierung, sollten nur die wichtigsten Module aktiviert werden. Eine gute Übersicht über die benötigten Module bietet [17]. Deaktiviert werden können die Module *charset-lite*, *include*, *env*, *setenvif*, *status*, *autoindex*, *asis*, *cgi*, *negotiation*, *imap*, *actions*, *userdir*, *alias* und *so*. Nach der Installation sollten nur noch die Module *core.c*, *mod_access.c*, *mod_auth.c*, *mod_log_config*.c, *prefork.c*, *http_core.c*, *mod_mime.c*, *mod_dir.c* vorhanden sein (Anzeige mit „httpd -l“).

[1] http://www.packetstormsecurity.org

Zur sicheren Installation gehört es auch, dass die Zugriffsrechte auf Dateisystemebene überprüft und gegebenenfalls geändert werden. Auf alle Dateien, die zur Apacheinstallation gehören, hat nur root schreibenden Zugriff. Allen anderen Benutzern muss dieser entzogen werden, um Manipulationen zu verhindern. Es gibt auch kaum einen Grund, warum andere Benutzer, außer root, die Logfiles lesen können sollten. Deshalb müssen auch hier unter Umständen die Zugriffsrechte geändert werden.

Nachdem der installierte Webserver nun lauffähig ist, muss er noch konfiguriert werden, da viele Standardeinstellungen nicht den Anforderungen an einen sicheren Webserver genügen. Die generelle Konfiguration soll an dieser Stelle nicht weiter erklärt werden, dazu gibt es eigene Bücher sowie ausreichend viele Informationen auf der Apache Webseite[2] und anderen Quellen im Internet.

2.1.2 Generelle Sicherheitskonfiguration

Zu einer sicheren Apachekonfiguration gehört, dass man nicht mehr Informationen preisgibt als nötig. Dies beinhaltet die Deaktivierung der Statuswebseiten, das Directory Listing und das Ausgeben von Informationen über den verwendeten Webserver. All diese Techniken gehören zwar der Kategorie „Security by Obscurity“ an, welche bekanntermaßen nicht sonderlich gut funktioniert, aber sie stellt einfach eine kleine zusätzliche Hürde für einen potentiellen Angreifer dar. Zum einen wird der so konfigurierte Server bei einem Netzwerkscan eher unauffällig sein und nicht das Interesse eines ungezielten Angriffs auf sich ziehen, zum anderen wird ein ernsthafter Angreifer erkennen, dass er einen fachkundigen Administrator als „Gegner“ hat und sich mit etwas Glück ein anderes Ziel suchen.

Um festzustellen welchen Webserver der Angreifer vor sich hat, kann er einfach eine ungültige URL aufrufen. Der Apache quittiert diese standardmäßig mit einer „URL Not Found“ Statusseite, bei der sich im unteren Teil die Identifikation des Servers findet:

> Apache/2.0.53 (Unix) mod_ssl/2.0.53 OpenSSL/0.9.7d PHP/4.3.10
> Server at www.fh-hagenberg.at Port 80

Anhand dieser Informationen kann der Angreifer nach bereits vorhandenen, vorgefertigten Exploits suchen. Der Parameter *„ServerSignature Off“* deaktiviert diese Art der Identifikation. Allerdings ist es immer noch problemlos

[2]http://www.apache.org

möglich, genaueres über die Apacheversion und die installierten Module heraus zu finden. Als einfaches Hilfsmittel lässt sich ein Telnet Client nutzen, welcher auf praktisch jedem verbreiteten Betriebssystem existiert:

```
server:~# telnet localhost 80
Trying 127.0.0.1...
Connected to server.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 04 Aug 2005 15:58:24 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-15 mod_chroot/0.4
X-Powered-By: PHP/4.3.10-15
Connection: close
Content-Type: text/html
```

Wie man sehen kann, werden weiterhin alle für einen Angreifer relevanten Informationen angezeigt. Die verschiedenen Optionen der *ServerTokens* Direktive sorgen hier für Abhilfe, empfehlenswert ist die Einstellung *ProductOnly*. Selbst NMAP (Open-Source Portscanner) findet die genaue Version des Apachen dann nicht mehr heraus:

```
PORT     STATE    SERVICE    VERSION
80/tcp   open      http      Apache httpd
```

Es gibt noch weitere Möglichkeiten die Identität zu verbergen. Eine Variante wäre die Produktinformationen in der *ap_release.h* (*httpd.h* in Apache 1.x) zu ändern, eine Andere das mod_security Modul einzusetzen, welches ebenfalls in der Lage ist, die Identifikation des Apache Webservers zu verändern. Eine Änderung in IIS bringt beispielsweise den Vorteil, dass Nessus in den Standardeinstellungen nicht nach Apache Sicherheitslücken suchen würde, was einen kleinen Schutz gegen Script Kiddies bietet. Von Hand lässt sich aber der Webserver immer noch über sein Verhalten bei Fehlern identifizieren. Ein IIS 6.0 reagiert anders als ein Apache Webserver, von daher müssen auch alle *ErrorDocument* Einstellungen so angepasst werden, dass der Apache nicht erkennbar ist.

Bei den vorgestellten Möglichkeiten sollte dem Administrator immer klar sein, dass es niemals unmöglich sein wird, die entsprechenden Informationen heraus zu finden. Auch sollte jedem klar sein, dass es wenig sinnvoll ist einen Apache Server auf Solaris als IIS ausgeben zu lassen. Ein Angreifer würde über TCP Fingerprinting das Betriebssystem trotzdem erkennen und daraus schließen, dass es sich unmöglich um einen IIS handeln kann, welcher ausschließlich auf Windows Betriebssystemen läuft.

Ein weiterer wichtiger Punkt sind die Rechte, mit denen der Apache ausgeführt wird. Um sich beim Start an den Port 80 zu binden, benötigt der Apache Root-Rechte, da es keine andere Möglichkeit gibt, Dienste auf privilegierten Ports (<1024) zu starten. Nach dem Start benötigt der Apache diese erweiterten Rechte nicht mehr und gibt sie ab. In vielen Linux Distributionen läuft er standardmäßig mit einschränkten Rechten, trotzdem sollten die *User* und *Group* Parameter überprüft und gegebenenfalls angepasst werden.

Damit der Administrator weiß, was auf seinem Server passiert, muss auch das Logging den Anforderungen entsprechend angepasst werden. Werden mehrere VirtualHosts betrieben, so ist ein separates Logging sinnvoll, um eine übersichtliche Logauswertung durch Drittanbietertools zu gewährleisten.

Standardmäßig deaktiviert ist meistens das so genannte „Directory Listing“, welches alle Dateien in einem Verzeichnis anzeigt, wenn keine Index Datei vorhanden ist. Wenn der Apache gemäß diesem Kapitel statisch kompiliert wurde, besteht diese Möglichkeit aufgrund eines fehlenden Moduls nicht. In einigen Fällen ist es aber aktiviert, was zu ernsthaften Problemen für die Kunden führen kann, sobald sie aus Versehen ihre Index Datei umbenennen oder der Administrator einen Fehler bei der *DirectoryIndex* Konfiguration macht. Dann liegen nämlich alle Dateien (kritisch sind vor allem Konfigurationsdateien von Anwendungen) frei zugänglich für jeden einsehbar auf dem Server bereit. Diese Konfigurationsdateien enthalten oft die Zugangsdaten für den Datenbankserver, wo vertrauliche Daten ausgelesen werden könnten. Deaktiviert werden kann dieses Verhalten mit dem *-Indexes* Parameter der *Options* Direktive.

Über die *Options* Direktive lassen sich noch weitere sicherheitsrelevante Einstellungen definieren. Eine davon ist die *FollowSymLinks* Option. Sie erlaubt es, dass der Apache symbolischen Links folgt. Systembenutzer könnten diese Option nutzen, um an Daten zu gelangen, zu denen sie eigentlich keine Zugriffsrechte besitzen. Die Option sollte daher durch *SymLinksIfOwnerMatch* ersetzt, oder komplett deaktiviert werden.

Um den Server vor versehentlichen Fehlkonfigurationen zu schützen, kann es sinnvoll sein, sichere Standardvorgaben für einige Verzeichnisse zu definieren. Der folgende Ausschnitt der Apache Konfiguration erlaubt ausschließlich den Zugriff auf das öffentliche Webverzeichnis, nicht jedoch auf die Daten im Root-Verzeichnis.

```
<Directory />
    Order Deny,Allow
    Deny from all
    AllowOverride None
</Directory>
<Directory /var/www/htdocs>
    Order Allow,Deny
    Allow from all
</Directory>
```

Manchmal kann es erwünscht sein, dass die Kunden einige Optionen des Webservers speziell für ihre Verzeichnisse anpassen können. Dies wird üblicherweise über *.htaccess* Dateien realisiert. Die Rechte für diese Dateien werden in der *AllowOverride* Direktive geregelt, welche sich für jedes Verzeichnis separat einstellen lässt. Diese *.htaccess* Abfrage mindert zwar die Performance des Webservers, aber sie ist sehr praktisch, um den Kunden selbstständig passwortgeschützte Verzeichnisse anlegen zu lassen. Problematisch wird die *AllowOverride* Direktive dann, wenn es dem Kunden erlaubt ist die *Options* Parameter bzw. sogar alle Einstellungen zu verändern.

In einigen Anleitungen, die sich im Internet finden, wird beispielsweise empfohlen *„AllowOverride All“* zu setzen, um die Passwortauthentisierung zu aktivieren. Dadurch ist es möglich, die globalen PHP Parameter der php.ini (mehr dazu im Verlauf dieses Kapitels) zu überschreiben was sämtliche PHP internen Schutzmechanismen aushebeln würde. Als Administrator sollte man daher die Optionen *AuthConfig*, *FileInfo*, *Indexes*, *Limit*, *Option*, *All* und *None* sorgfältig auswählen.

Um bei möglichen Fehlkonfigurationen zu verhindern, dass der Webserver *.htaccess* und einige andere, nicht für die Öffentlichkeit bestimmten, Dateien ausliefert, sollten folgende Optionen gesetzt sein. Einige Editoren speichern verschiedene Versionen einer Datei als *.bak automatisch ab, daher wird diese Endung ausgeblendet, um Dateiinhalte zu schützen.

```
<FilesMatch "(^\.ht|~$|\.bak$|\.BAK$)">
    Order Allow,Deny
    Deny from all
</FilesMatch>
```

Um den Zugang zu typischerweise nicht öffentlichen Verzeichnissen zu verwehren, bietet sich folgende Regel an, die den Zugang zum CVS-Verzeichnis verbietet.

```
<DirectoryMatch /CVS/>
    Order Allow,Deny
    Deny from all
</DirectoryMatch>
```

Sehr sicherheitskritisch sind auch alle Einstellungen, die das Ausführen von CGI-Skripts ermöglichen. Daher sollten solche Optionen standardmäßig entfernt werden, namentlich die *ExecCGI* Option und sämtliche *ScriptAlias*e. Weitere Informationen über CGI Einstellungen befinden sich in Abschnitt 2.3.

Falls nicht unbedingt benötigt, sollten auch die Optionen für Benutzerverzeichnisse deaktiviert werden („*UserDir disabled*“), da ein Angreifer über die Fehlermeldungen des Webservers herausfinden könnte, ob ein bestimmter Benutzer im System existiert oder nicht. Angreifer, die keine Ambitionen haben den Server zu übernehmen, benötigen solche Informationen nicht. Sie versuchen oft mittels Denial of Service Attacken den Betrieb des Webserver zu stören oder ihn komplett lahm zu legen.

2.1.3 Denial of Service Gefahren

In den letzten Jahren hat sich die Anzahl der Erpressungen von Firmen im Internet drastisch erhöht. Onlineshops, Wettbüros oder anderen Unternehmen, die vom Internet abhängig sind, wird damit gedroht ihren Webserver oder die gesamte Internetverbindung lahm zu legen (Beispiel: [20]). Diese Gefahren gehen von organisierter Kriminalität aus und sind von einem einzelnen Administrator kaum abzuwehren. Üblicherweise sind Webserver maximal mit 100MBit/s an das Internet angeschlossen. Die Botnetze der Angreifer bringen es aber problemlos auf mehrere GigaBit/s Bandbreite, wogegen man ohne Unterstützung des Internetproviders machtlos ist. Kleinere Angriffe gegen einen Apache Webserver lassen sich jedoch durch einigen Konfigurationsaufwand abwehren bzw. mindern.

Nicht immer ist viel Bandbreite nötig, um einen Apachen außer Gefecht zu setzen, auch Buffer Overflows, die den Webserver zum Absturz bringen, sind eine Form von Denial of Service. Daher sollte das Betriebssystem Maßnahmen gegen Buffer Overflows bereitstellen und Sicherheitsupdates für den Apachen zeitnah eingespielt werden. Auch für fehlerhaft programmierte Scripts, welche den Server lahmlegen, benötigt der Angreifer kaum Bandbreite. Die entsprechenden Konfigurationsmöglichkeiten werden in Kapitel 2.2 und 2.3 beschrieben.

Jedes verfügbare Apache Benchmark Tool lässt sich für einen Angriff missbrauchen. Solche Angriffe sind zwar leicht zu erkennen, führen den Angreifer aber

trotzdem zum Ziel. Bei den meisten Apache Installationen wird der Apache Benchmark „ab“ mitgeliefert, welcher für die nachfolgenden Brute-Force Angriffe verwendet wird.

```
$ ab -n 5000 -c 300  http://www.opfer.org/
```

Diese Form des Benchmarks würde 300 Anfragen parallel an ein beliebiges Ziel senden, 5000 insgesamt. Je weniger Bandbreite zum Server besteht, desto besser wird der Angriff funktionieren, wichtig für den Angreifer ist es, mehr parallele Anfragen zu generieren als der Apachen maximal erlaubt (*MaxClients*). Auf der Homepage[3] des Autors von [23] stehen drei Tools zum Download bereit, die bei der Bekämpfung von DoS Angriffen helfen können.

Verhindern kann der Administrator solche Angriffe durch entsprechende Firewallregeln, welche automatisch gesetzt werden. Über das mod_status Modul kann sich der Administrator über den aktuellen Zustand seines Servers informieren, bei der Aktivierung von mod_status müssen die Zugriffsrechte (*AllowFrom*) so gesetzt werden, dass nicht jeder die Statistiken unter „http://www.server.com/server-status“ auslesen kann. Um Denial of Service Gefahren möglichst lange stand halten zu können, gibt es einige Parameter, die der Rechenleistung und Bandbreite des Webservers angepasst werden müssen.

Über *MaxClients* wird definiert, wie viele gleichzeitige Anfragen erlaubt sind. Diese Zahl sollte möglichst hoch sein, aber nicht so hoch, dass der Arbeitsspeicher nicht mehr ausreicht, um alle Anfragen zu bearbeiten. Generelle Empfehlungen sind nur schwierig abzugeben, da der reale Speicherverbrauch vom Memory Management des Betriebssystems abhängig ist. Muss der Server Daten aus dem Speicher auslagern (Paging oder Swapping) bricht die Performance vollkommen zusammen, was durch richtige Konfiguration verhindert werden muss.

Der *KeepAlive* Parameter erlaubt das Wiederverwenden von TCP Verbindungen und sollte aktiviert werden [18]. Zusätzlich müssen *MaxKeepAliveRequests* und *KeepAliveTimout* gesetzt werden. Die Werte 100 bzw. 15 eignen sich als Startwerte und müssen angepasst werden, wenn sie nicht ausreichen sollten. Eine Verminderung des Timeouts von 15 auf 10 Sekunden kann sich sehr positiv auswirken. Zusätzlich gibt es noch einen generellen Timeout Parameter, welcher schlecht dokumentiert ist und daher nur bei Bedarf vermindert werden sollte. Die Apache Homepage[4] schreibt dazu wörtlich:

> It is not set any lower by default because there may still be odd

[3] http://www.apachesecurity.net
[4] http://httpd.apache.org/docs/2.0/mod/core.html.en#timeout

> places in the code where the timer is not reset when a packet is sent.

Weitere Parameter können in [3, Kapitel 10.2] nachgelesen werden.

Nicht nur die Anzahl von Verbindungen kann problematisch für den Server werden, auch die verwendete Bandbreite ist ein entscheidender Faktor für die Verfügbarkeit von Webservices. Eine sehr effektive Methode ist die Verwendung von Kompression beim Versand von Inhalten. Alle modernen Browser beherrschen Komprimierung, sodass dadurch viel Bandbreite gespart werden kann, wenn viele HTML Inhalte übertragen werden. Zuständig für Komprimierung sind die Module mod_gzip und mod_deflate (Apache 2), Komprimierung kann aber auch in PHP direkt implementiert werden.

Noch stärker auf das Transfervolumen wirken sich externe Verlinkungen auf Dateien aus. Dieses so genannte Hotlinking bindet fremde Dateien oder Bilder in Webseiten ein, wodurch der Verlinkte geschädigt wird, der Verlinkende jedoch sehr viel Bandbreite spart.

Eine Verlinkung von Bildern (1x1 Pixel Größe reicht vollkommen) auf stark frequentierten Seiten erzeugt viele gleichzeitig geöffnete Verbindungen, was zu einem Denial of Service Problem werden kann. Eine Möglichkeit, solche unerwünschten Zugriffe zu beseitigen, gelingt über das mod_rewrite Modul, welches den Zugriff mittels Refererauswertung verhindert. In [23, Kapitel 5.3] werden folgende Regeln vorgeschlagen, die die genannten Probleme beseitigen:

```
# leere Referer erlauben, für Besucher die eine Seite direkt aufrufen
RewriteCond %{HTTP_REFERER} !^$

# Benutzer von http://www\.apachesecurity\.net erlauben
RewriteCond %{HTTP_REFERER} !^http://www\.apachesecurity\.net [nocase]

# Hotlinking auf bestimmte Dateitypen verhindern - andernfalls
# könnte niemand mehr auf die Seite verlinken
RewriteRule (\.gif|\.jpg|.\png|\.swf)$ $0 [forbidden]
```

Um generell die Bandbreite pro Verbindung oder IP zu beschränken, können die Möglichkeiten des Betriebssystems oder spezieller Apache Module verwendet werden. Als Apache Modul kommt beispielsweise mod_bwshare[5] in Frage. Es gibt noch einige weitere Module, die allerdings schon länger nicht weiterentwickelt wurden bzw. nicht mit Apache2 zusammenarbeiten.

Grundsätzlich muss man festhalten, dass die Vorbereitungen gegen Denial of Service Angriffe sehr schwierig sind. Man kann zwar Tests vornehmen, um

[5] http://www.topology.org/src/bwshare/

das Verhalten des Webservers zu optimieren, aber es ist nahezu unmöglich, das Verhalten des Servers bei einem echten Angriff vorauszusehen. Um den Betrieb während eines ernsten Denial of Service Angriffs aufrecht zu erhalten, braucht es weitaus mehr als nur eine richtige Konfiguration des Webservers.

Gegen virtuelle Demonstrationen, wie sie die Lufthansa erlebt hat [9], oder eine unerwartete Erwähnung und Verlinkung auf stark besuchten Webseiten (Slashdot, Heise) muss ein funktionierendes Zusammenspiel aus Infrastruktur, Bandbreite, Konfiguration der Dienste und hochqualifiziertes Personal gegeben sein. Schnelle Reaktionen auf diese Ereignisse sind entscheidend für die Verfügbarkeit der Dienste.

2.1.4 Das mod_security Modul

Mod_security ist ein Intrusion Detection und -Prevention Tool, welches den Apache Webserver gegen bekannte Angriffstechniken schützen kann. Es steht auf der offiziellen Homepage[6] frei zum Download, wird stetig weiterentwickelt und kommerzieller Support ist verfügbar. Eine gute Dokumentation ist ebenfalls auf der Homepage vorhanden und die Konfiguration der wichtigsten Parameter ist nicht sonderlich schwierig. Eine zu strenge Konfiguration hat zur Folge, dass Anwendungen nicht mehr funktionieren. Gerade einige weit verbreitete PHP-Foren sind sehr heikel und der Administrator sollte ausführlich testen, welche Schutzmechanismen er letztendlich aktiviert.

Es besteht die Möglichkeit der Inputanalyse, sodass Daten, die an den Webserver mittels POST oder GET gesendet werden, gefiltert werden können. Dadurch lassen sich je nach Konfiguration einige Arten von Cross-Site-ScriptingAngriffen[7] verhindern. Um Angriffe gegen den Webserver abzufangen, können verschiedene Maßnahmen der Inputvalidierung zum Einsatz kommen. URLs können decodiert werden, sodass hexadezimal übermittelte Zeichen in ASCII umgewandelt werden, bevor sie die Webanwendung erreichen. Referenzen auf sich selbst werden aufgelöst und doppelte Trennzeichen entfernt (aus // wird /). Bei solchen Maßnahmen sind immer Nebeneffekte zu beachten, das letztgenannte Beispiel konvertiert „http://“ in „http:/“, was im Normalfall nicht erwünscht sein wird. Sinnvoll wäre es, Inhalte, die aus NULL-Bytes bestehen, durch Leerzeichen zu ersetzen oder anderweitig zu codieren, da sie im Zusammenhang mit Datenbanken zu Problemen führen können.

Für den Apache 2 besteht zudem die Möglichkeit der Outputanalyse, wo ebenfalls Daten gefiltert werden können, um XSS-Angriffe zu verhindern. Um

[6] http://www.modsecurity.org

[7] Unter Cross-Site-Scripting (XSS) versteht man Techniken, die den Webserver dazu benutzen dem Besucher einer Webseite bösartigen HTML-Code zu präsentierten [13, S. 111]

Angriffe nachvollziehbar zu machen gibt es zusätzliche Log-Optionen, die umfangreicher sind, als die standardmäßig im Apache vorhandenen. Zudem ist es möglich, nach einem entdeckten Angriff eine Aktion ausführen zu lassen. Die Umleitung auf eine andere Seite oder ein kompletter Verbindungsabbruch wären denkbare Möglichkeiten. Die Option des Verbindungsabbruchs bietet wiederum ein erhebliches Gefahrenpotential, da durch DoS-Angriffe mit gefälschten Absender IP-Adressen auch legitime Verbindungen beendet werden könnten.

Alle Filterungen können auf verschiedensten Ebenen stattfinden. Es ist möglich globale Regeln zu erstellen, genauso wie es möglich ist, Regeln für eine einzelne Datei zu bestimmen. Hier sollte man sich dann aber doch überlegen, ob es nicht sinnvoller wäre die Anwendung sicherer zu programmieren, als an den Symptomen zu arbeiten. Abgesehen davon ist der Aufwand bei vielen Skripts, für jedes eine separate Filterregel zu erstellen, sehr hoch.

Es gibt sehr umfangreiche Möglichkeiten, einen Server mit mod_security abzusichern, die hier im Einzelnen nicht erörtert werden sollen. Dazu bieten [23, Kapitel 12.2] sowie www.modsecurity.org ausreichend viele Beispiele.

Die Funktionsvielfalt und Komplexität des Moduls hat natürlich auch ihre Nachteile. In den vergangenen Jahren hat es drei Sicherheitsprobleme mit dem mod_security Modul gegeben, welche wenig bis mittelmäßig gefährlich waren. Eine Alternative zu mod_security im Bereich der Input Validierung wäre das mod_parmguard[8], welches XML basierend arbeitet.

Im Webhostingbereich wird man viele der bisher und nachfolgend angeführten Möglichkeiten zur Absicherung des Webservers nicht global anwenden wollen, sondern für jeden Kunden separat. Insbesondere die für PHP geltenden Restriktionen machen nur auf diese Weise Sinn. Die Konfiguration erfolgt dabei über die so genannten VirtualHosts.

2.1.5 VirtualHosts Konfiguration

VirtualHosts werden für jede einzelne Domain, die der Webserver kennt, in den Konfigurationsdateien eingestellt. Dies beginnt beim Domainnamen und geht über die Verzeichniseinstellungen bis hin zu PHP- und Ressourcen-Parametern. Sicherheitsrelevant sind dabei vor allem die PHP Parameter (siehe auch Abschnitt 2.2), um die Kunden voneinander zu trennen. Die Einstellungen dafür werden über die *php_admin_value* Direktive getroffen, die es ermöglicht, für jeden Kunden ein separates Home-, Upload- und Sessionverzeichnis bereit zu stellen. Würde man das Homeverzeichnis global in der php.ini (*open_basedir*, näheres im nachfolgenden Abschnitt) einstellen, so

[8]http://www.trickytools.com/php/mod_parmguard.php

könnten die Benutzer untereinander immer noch die Dateien auslesen und so zu fremden Datenbankkennwörtern gelangen. Neben *php_admin_value* gibt es noch weitere PHP-Parameter: *php_value*, *php_flag* und *php_admin_flag*. Die Admin-Parameter werden in der VirtualHosts Konfiguration definiert und können nicht durch *php_value* oder *php_flag* überschrieben werden, welche in *.htaccess* Dateien konfiguriert werden.

Innerhalb eines VirtualHosts können die Optionen für bestimmte Verzeichnisse eingestellt werden und damit auch die bereits im vorangegangenen Abschnitt erwähnten *AllowOverride* Parameter. Entscheidend ist hier noch einmal, dass der Kunde auf gar keinen Fall die Möglichkeit bekommen darf, die *Options* Direktive über *.htaccess* zu verändern. Andernfalls könnte er die VirtualHosts Einstellungen überschreiben und mit PHP aus seinem Verzeichnis ausbrechen. Eine kurze Beispielkonfiguration soll dies veranschaulichen:

```
<VirtualHost 192.168.0.111:80>
       ServerName www.kunde1.de
       DocumentRoot /srv/www/kunde1

      php_admin_value open_basedir /srv/www/kunde1/
      php_admin_value upload_tmp_dir /srv/tmp/kunde1/
      php_admin_value session.save_path /srv/sessions/kunde1/

       <Directory /srv/www/kunde1>
                AllowOverride AuthConfig Indexes
        </Directory>
</VirtualHost>
```

Leider bieten diese Optionen bei mod_php nur eine trügerische Sicherheit, auch wenn sie in vielen Büchern, Anleitungen und auf der PHP Homepage[9] genau so beschrieben werden. Viele Versionen von PHP haben Fehler bei der Behandlung von *open_basedir*. Während der Tests wurde festgestellt, dass in obiger Konfiguration der Kunde1 auch Zugriff auf die Verzeichnisse von */srv/www/kunde10*, */srv/www/kunde11* usw. hat, also auf alle Verzeichnisse, die namentliche Übereinstimmungen haben. Der abschließende „/“ wird schlicht ignoriert.

Weitere Nachforschungen ergaben, dass dieses Problem unter [19] bereits seit Mai 2005 als Bug in PHP 5.0.4 bekannt ist. Interessant war es festzustellen, dass in Debian Sarge (PHP 4.3.10-15) und Gentoo Linux (PHP 4.4.0) diese Bugs im Juli ebenfalls noch existierten. Der (inoffizielle) Grund dafür ist auf der Debian-Security Mailingliste unter [31] zu finden. Hier wird zwar nur gesagt, dass *SafeMode* Bugs nicht behoben werden, weil es irrelevant sei, es ist aber zu vermuten, dass gleiches für *open_basedir* gilt.

[9] http://www.php.net

Eine Aussage auf der offiziellen PHP Homepage bestärkt diese Vermutung [21]

> For Local exploits we mostly hear about open_basedir or safe-mode problems on shared virtual hosts. These two features are there as a convenience to system administrators and should in no way be thought of as a complete security framework. With all the 3rd-party libraries you can hook into PHP and all the creative ways you can trick these libraries into accessing files, it is impossible to guarantee security with these directives. The Oracle and Curl extensions both have ways to go through the library and read a local file, for example. Short of modifying these 3rd-party libraries, which would be difficult for the closed-source Oracle library, there really isn't much PHP can do about this.

Nach Ansicht des PHP Entwicklers Jani Taskinen ist der Bug kein Bug mehr, da er bereits in der aktuellen Version 5 von PHP behoben wurde, Fehler in veralteten Versionen werden nicht behoben [25] , dafür sind die Distributoren zuständig. In Gentoo Linux wird der Bug ebenfalls nicht als sicherheitskritisch angesehen. 20 Minuten nach Absenden des Bugreports wird er mit

> Reassigning to php, we usually don't accept safe_mode bugs. See http://www.php.net/security-note.php for details. Thanks for reporting, though.

in das Bugtracking System der normalen PHP Fehler verschoben [27]. Bei Debian wird das Problem nach kurzer Diskussion genauso gehandhabt, es gibt kein Securityupdate für PHP, welches das Problem behebt [26].

Gut einen Monat später kam dann doch noch Bewegung in die Sache, nachdem ein Ubuntu Entwickler für besagten Bug eine CAN Nummer beantragt hat. Der von mir gefundene Bug ist nun unter der CAN Nummer CAN-2005-3054 zu finden. Außerdem wurde er von den beiden bekanntesten Securityseiten FrSirt [8] und Securityfocus [16] als Securitybug übernommen. Wie man in Abbildung 2.1 sehen kann, wird der Entdecker des Bugs unter „Credits“ erwähnt. Trotz all dem, gibt es für Debian Sarge noch immer kein Sicherheitsupdate.

Eine Möglichkeit dieses Problem zu umgehen, wäre der Einsatz von nicht vorhersehbaren bzw. nicht erratbaren Verzeichnisnamen, was so auch in manchen größeren Umgebungen gemacht wird. Ein Grund für den Einsatz von mod_php, trotz bekannter Sicherheitsrisiken ist, dass es eine sehr gute Performance im Vergleich zu anderen PHP Implementierungen liefert. Die Per-

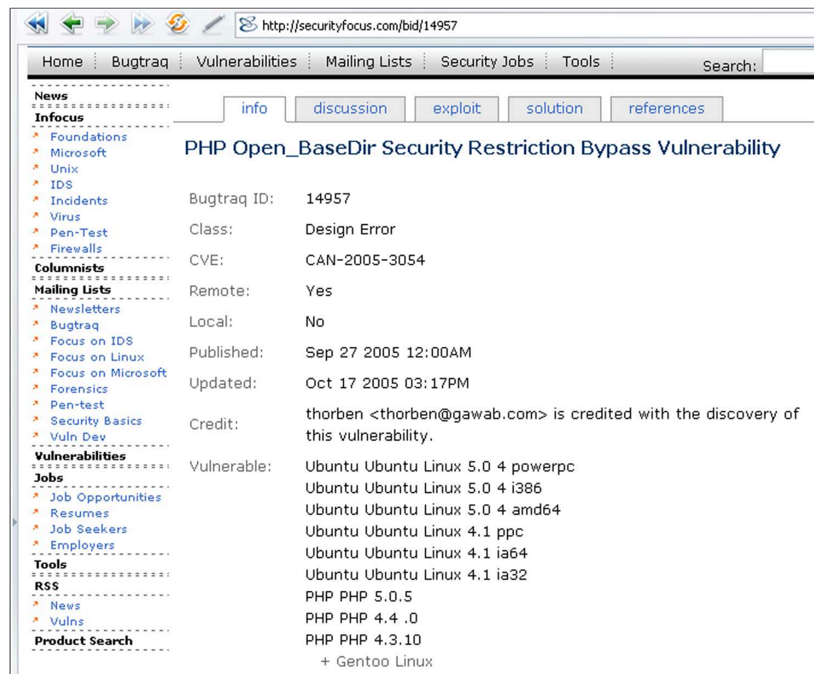

Abbildung 2.1: Advisory auf Securityfocus

formance wird oft höher bewertet als die Sicherheit. Wie PHP sicher eingerichtet wird, auch in Bezug auf die soeben erwähnte Problematik, beschreibt der folgende Abschnitt.

2.2 PHP

PHP (PHP Hypertext Preprozessor) ist eine weit verbreitete serverseitige Skriptsprache, welche von vielen Webhostern schon im Niedrigpreissegment angeboten wird. PHP ist relativ leicht zu erlernen und wird daher von Kunden oft verlangt. Die Absicherung von PHP ist jedoch nicht trivial und alle bekannten Standardinstallationen sind unsicher und ausschließlich auf größtmögliche Funktionalität ausgelegt. PHP ist eine sehr mächtige Sprache, da sie auf Dateien zugreifen, Befehle ausführen und Netzwerkverbindungen aufbauen kann.

Vor der Installation muss überlegt werden, ob PHP als Apachemodul oder in einer CGI-Version installiert wird. Die CGI-Version bietet den Vorteil, dass verschiedene CGI-Wrapper eingesetzt werden können, um chroot- oder setuid-Umgebungen einzurichten. Wird PHP als Modul installiert, so übernimmt es die Rechte des Apache (z. B. www-data), was Auswirkungen auf

die Sicherheit der gesamten Installation hat.

Erstens hat der Webserver mindestens Leserechte auf alle Kundenverzeichnisse und zweitens Schreibrechte für Verzeichnisse, in denen Uploads möglich sind. Dies kann erwartungsgemäß zu Sicherheitsproblemen führen und es ist von daher sehr zu empfehlen, entsprechende Dateisystemberechtigungen systemweit zu etablieren. Wie diese Dateisystemberechtigungen genau aussehen wird im Abschnitt „Sicherheitsrisiko Perl“ beschrieben. Der Vorteil der Modulinstallation liegt in seiner Geschwindigkeit und der einfachen Konfiguration. Wenn es bei dem zu installierenden Webserver nicht mehrere Kunden gibt, ist die Installation als Modul aus Performancegründen angebracht.

Wenn man sich aus Sicherheitsgründen für die Installation einer CGI-Version entschieden hat, so hat man eine recht umfangreiche Auswahl aus mehr oder weniger performanten und featurereichen Möglichkeiten. Einige davon werden im Laufe dieses Kapitels vorgestellt, doch zuerst soll die generelle Sicherheitskonfiguration von PHP beschrieben werden.

2.2.1 Sichere Konfiguration von PHP

Über die sichere Konfiguration von PHP gibt es bereits einige Bücher (z. B. [23], [3]) und sehr viele Anleitungen im Internet (z. B. www.php.net, [16]), doch oftmals werden wichtige Punkte vergessen. In den nachfolgenden Betrachtungen wird von einer Installation als Modul ausgegangen.

In der php.ini-Datei werden die globalen PHP Einstellungen getätigt. Sie sind vom Sicherheitsstandpunkt aus meistens nicht ausreichend detailliert und sollten durch Einstellungen in den VirtualHosts ergänzt werden. Im Folgenden werden die wichtigsten Parameter und deren Einstellung und Wirkung erklärt.

Eine der ersten Optionen, die deaktiviert werden sollte ist, *allow_url_fopen*. Diese Einstellung ermöglicht es externen Code, von beliebigen anderen Webservern, in eine Anwendung einzubinden, wodurch wiederum XSS-Angriffe möglich werden können. Aber nicht nur XSS-Angriffe sind darüber möglich, auch einige Abkömmlinge des Santy Wurms haben sich über das Einbinden externen Codes verbreitet, um anschließend DDoS[10] Angriffe zu starten. Im Phrack Magazin [2] wird ein Angriff über das dynamische Nachladen von Modulen beschrieben, um diese Möglichkeit zu unterbinden wird der *enable_dl* Parameter auf „Off“ gesetzt.

Wie bereits bei der Apache Konfiguration erwähnt, sollte man nicht mehr Informationen preisgeben als nötig und daher *expose_php* ebenfalls auf „Off“

[10]Distributed Denial of Service

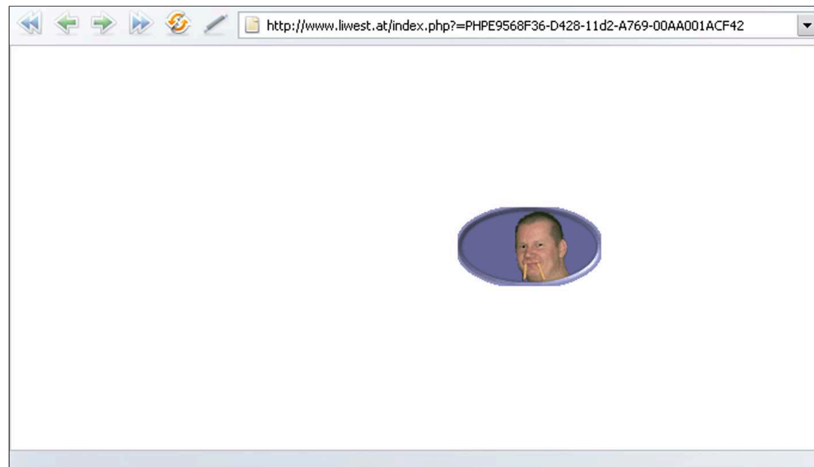

Abbildung 2.2: Achten Sie auf die URL!

setzen. Die Deaktivierung von *expose_php* führt allerdings zu einem Featureverlust (siehe Abb. 2.2). Bei deaktiviertem *expose_php* würde der Aufruf nicht funktionieren, was aber für die meisten Administratoren verschmerzbar sein dürfte.

Zur Beschränkung der Informationsweitergabe gehört es auch, dass keine PHP Fehlermeldungen vom Server ausgegeben werden. Diese erlauben oft Rückschlüsse auf die Datenbankstrukturen und die Funktionsweise einer Anwendung. Um die Fehlerausgaben zu verhindern wird „*display_errors = Off*“ und „*display_startup_errors = Off*“ gesetzt.

Ohne irgendeine Ausgabe von Fehlermeldungen ist es für den Kunden / Benutzer fast unmöglich, Fehler in einer Anwendung zu finden, weshalb die Fehlermeldungen in einem Logfile gespeichert werden. Dazu wird der Loglevel mit „*error_reporting = E_ALL*“ erhöht und das Logging aktiviert („*log_error = On*“). Danach wird noch der Speicherort des Logfiles mit Hilfe des *error_log* Parameters festgelegt. Sinnvoll ist es, jedem Benutzer sein eigenes Logfile (im Webroot) zur Verfügung zu stellen, daher sollte dieser Parameter in der VirtualHosts Konfiguration definiert werden.

Um nicht Opfer eines DoS Angriffs zu werden, bietet die PHP Konfiguration einige Ressourcen Limits an, die man setzen sollte. Diese sind in der Sektion „Resource Limits“ zu finden und selbsterklärend. Die Werte müssen entsprechend der Serverhardware und Bandbreite angepasst werden. Etwas genauer sollte der *max_input_time* Parameter betrachtet werden, welcher bei Dateiuploads zu Problemen führen kann. Er ermöglicht standardmäßig eine Ausführungszeit von 60 Sekunden, das bedeutet, dass jemand mit einer Uploadgeschwindigkeit von 5KB/s gerade einmal 300KB hochladen kann.

Um Dateiuploads zu steuern gibt es einige weitere Einstellungen. „*File_ uploads = Off*“ verbietet ihn komplett und über die *upload_ max_ filesize* kann die maximale Größe von Uploads definiert werden. Die *post_ max_ size* muss natürlich ein wenig größer sein, um widersprüchliche Einstellungen zu vermeiden. Das *upload_ tmp_ dir* wird, wie bereits beschrieben, für jeden VirtualHost separat festgelegt, genau wie die Speicherung der Sessions (*session.save_ path*). Die Session Sicherheit lässt sich über weitere Parameter noch erhöhen, beispielsweise über den *session.referer_ check* und die *session.gc_ maxlifetime.*

2.2.1.1 Der Parameter open_ basedir

Nach der PHP Installation obliegt PHP keinerlei Einschränkungen, auf welche Daten im Dateisystem es zugreifen darf. Es kann auf alle Dateien zugreifen, die für den Apache Webserver erreichbar sind. Unter unixartigen Betriebssystemen gibt es nur selten ACLs (Access Control Lists), wie sie die Windows NT Linie seit der Einführung von NTFS bietet. ACLs würden es auf einfache Weise ermöglichen, einem Benutzer (die diesem Fall dem Webserver) den Zugriff auf Bereiche des Dateisystems zu verbieten. Um PHP in ein oder mehrere Verzeichnisse einzusperren, wird der *open_ basedir* Parameter in der php.ini bzw. den VirtualHosts Einstellungen entsprechend angepasst.

Bei der Konfiguration des *open_ basedir* Parameters muss beachtet werden, dass er einen Prefix darstellt, „/srv/incl“ ermöglicht also auch den Zugriff auch „/srv/include“. Daher muss der Parameter mit einem angehängten „/“ geschlossen werden, um PHP in ein bestimmtes Verzeichnis zu sperren. Wie bereits im Kapitel 2.1.5 beschrieben, ist die Maßnahme mit dem „/“ fehlerbehaftet [19].

Was die Beschreibung auf der offiziellen PHP Homepage[11] verschweigt ist, dass Befehle die durch PHP beispielsweise mittels *shell_ exec* ausgeführt werden, nicht von diesen Restriktionen betroffen sind. Ein einfaches „shell_exec(“ls -la”)“ gibt weiterhin den Inhalt des Rootverzeichnisses preis. Eingeschränkt werden nur die in PHP gegebenen Dateisystem-Funktionen, zum Beispiel *opendir()* oder *fopen()*, nicht aber externe Programme, die von PHP aufgerufen werden. Im Oktober 2004 wurde zudem auf der BugTraq Mailingliste [7] eine Möglichkeit gefunden, die *open_ basedir* Einstellungen über externe Funktionen zu umgehen.

Open_ basedir ist nur eine Selbstbeschränkung von PHP, vergleichbar mit Chroot-Modulen für Apache aus Kapitel 1 und längst nicht so effektiv wie echte Chroot-Umgebungen mit Betriebssystemunterstützung. Außerdem ist

[11] http://www.php.net/manual/

die Option zwecklos, solange das Ausführen von Shellbefehlen möglich ist. Um Shellbefehle kurzfristig zu deaktivieren, reicht es, den *Safe_mode* zu aktivieren. Auf die Einstellungen von *open_basedir* sollte sich ein verantwortungsbewusster Administrator niemals verlassen. Er muss den Server immer mit Dateisystem-Berechtigungen so absichern, als gäbe es keine PHP-Sicherheitseinstellungen. Wie man die Ausführung kritischer Befehle im Einzelnen verhindert, zeigen die folgenden Abschnitte.

2.2.1.2 Der Parameter disable_functions

Disable_functions erlaubt es dem Administrator, einzelne PHP-Befehle komplett zu verbieten. Als Erweiterung kann man auch einzelne Klassen mit Hilfe von *disable_classes* verbieten. Um zu wissen, welche Funktionen man verbieten kann, muss der Administrator relativ genaue Kenntnisse von PHP haben. Wird zu viel verboten, werden die Kunden unzufrieden sein, denn in der Regel zeigen sie kein Verständnis für zu restriktive Sicherheitsmaßnahmen. Eine genaue Beschreibung der nachfolgend genannten Funktionen ist in der PHP Referenz[12] zu finden.

Als Erstes sollten alle Funktionen deaktiviert werden, die es erlauben, Befehle auf der Shell auszuführen. Dazu gehören *system*, *exec*, *shell_exec*, *passthru*, *popen* und *proc_open*. In der PHP Funktionsreferenz im Bereich „Program Execution Functions“ werden zusätzlich noch *proc_close*, *proc_get_status*, *proc_nice* und *proc_terminate* angeführt, welche zusätzlich verboten werden könnten. Da aber aufgrund der bereits konfigurierten Restriktionen keine Prozesse gestartet werden können, ist es überflüssig Befehle zu verbieten, die diese Prozesse beeinflussen können.

Da PHP zusammen mit Apache installiert wird, ist es sinnvoll, einige apachespezifische Funktionen zu verbieten. Dazu zählen *apache_child_terminate, apache_get_modules, apache_get_version, apache_getenv, apache_note, apache_setenv* und *virtual*. Gleiches gilt für die Syslog-Funktionen *syslog* und *openlog*.

Ob man die Funktionen *show_source* bzw. *highlight_file* und *phpinfo* deaktiviert, darüber lässt sich streiten. Die Funktionen zum Anzeigen des Quelltexts sollten durch andere Restriktionen nur auf die eigenen PHP Dateien Wirkung haben, von daher bräuchten sie nicht verboten werden. Falls es einem Angreifer allerdings gelingen sollte, diese Funktion mangels korrekter Inputvalidierung auf fremdem Webspace zum Laufen zu bringen, stellt sie eine erhebliche Gefahr dar. Der Angreifer kann so Zugangsdaten, welche oft in PHP Dateien gespeichert werden, auslesen.

[12] http://at2.php.net/manual/en/funcref.php

Die *phpinfo*-Funktion erlaubt es, sehr viele Details über die PHP Installation zu erfahren. Auf der anderen Seite ist sie Benutzern bei der Fehlersuche hilfreich, wenn Skripts auf dem Webserver nicht funktionieren. Ein möglicher Kompromiss wäre eine statische HTML Datei, in der der Administrator kritische Stellen entfernt und die er allen Benutzern zur Verfügung stellt.

Manchmal kann es erforderlich sein, dass die Ausführung von Systembefehlen erlaubt werden muss. Mit Hilfe der *safe_mode* Optionen kann eine Auswahl bestimmter Programme, die ausgeführt werden dürfen, definiert werden.

2.2.1.3 Die safe_mode Parameter

Die unterschiedlichen *safe_mode* Parameter versuchen die Sicherheitsprobleme bei Servern mit mehreren Benutzern zu lösen. Wie bereits erwähnt wurde, ist der *safe_mode* vom Design und der Architektur her unzureichend. Es ist prinzipiell falsch, Benutzerrechte auf PHP-Ebene vergeben zu wollen, dafür sollten Webserver oder Betriebssystem zuständig sein. Da diese die entsprechenden Möglichkeiten jedoch nicht bieten, wird *safe_mode* von vielen Administratoren eingesetzt. Sucht man nach Schwachstellen in der *safe_mode* Implementierung, so wird man auf den einschlägigen Securityseiten schnell fündig. Ein Verlass auf *safe_mode* ist also wie bei *open_basedir* keinesfalls empfehlenswert.

Der *safe_mode* Parameter überprüft, ob die UID des ausgeführten Skripts mit der UID der zu lesenden Datei übereinstimmt. Ergänzt wird der *safe_mode* durch den *safe_mode_gid* Parameter, welcher die Gruppenzugehörigkeit überwacht. Am einfachsten lässt sich das an einem kurzen Beispiel erklären. Es gibt ein PHP Skript und eine Datei, die es auslesen soll, mit folgenden Rechten:

```
-rw-r--r--   1  testuser testuser   43 Aug  9 14:59 readfile.php
-rw-r--r--   1  root     testuser   11 Aug  9 14:58 testfile
```

readfile.php:

```
<?php
    readfile('/var/www/testfile');
?>
```

php.ini Optionen:

```
safe_mode = On
safe_mode_gid = Off
```

Ergebnis:

> Warning: readfile(): SAFE MODE Restriction in effect. The script whose uid is 1002 is not allowed to access /var/www/testfile owned by uid 0 in /var/www/readfile.php on line 2.

Das Skript bricht ab, weil die UIDs von „readfile.php“ und „testfile“ nicht übereinstimmen. Verzichtet man auf den GID Vergleich, also setzt die Option „*safe_mode_gid = On*“, so erhält man als Ergebnis den Inhalt von „testfile“:

> Geheimer Text aus testfile.

Hier wird nur überprüft, ob die GIDs übereinstimmen, was der Fall ist, da sowohl „readfile.php“ als auch „testfile“ der Gruppe „testuser“ gehören.

Für einen Provider wird „*safe_mode_gid = On*“ oft die einzige Möglichkeit sein, seine Kunden zufrieden zu stellen, da viele PHP Content Management Systeme Dateien erstellen. Die einfachste Lösung wäre es, den Kunden in die Apache Gruppe zu geben und alle Dateien der Apache Gruppe zuzuordnen. Damit geht zwar jeder Schutz über die Zugriffsrechte verloren, aber zumindest wird das Ausführen von externen Programmen durch den *safe_mode* unterbunden. Soll das Einbinden von Dateien mittels *include*-Befehl in bestimmten Pfaden erleichtert werden, bietet sich der *safe_mode_include_dir* Parameter an, der UID und GID Überprüfungen deaktiviert.

Der *safe_mode* beschränkt auch das Verändern von Umgebungsvariablen. Welche Variablen der Benutzer ändern darf und welche nicht, hängt von *safe_mode_allowed_env_vars* und *safe_mode_protected_env_vars* ab. Die Standardeinstellungen sehen *PHP_ ** für die erlaubten Variablen und *LD_LIBRARY_PATH* für die nicht erlaubten Umgebungsvariablen vor, was sinnvoll ist.

Ist der *safe_mode* aktiviert, so sind nur noch Programme, die sich im durch *safe_mode_exec_dir* definierten Verzeichnis befinden, ausführbar. Standardmäßig ist kein Verzeichnis eingetragen, es ist also nicht mehr möglich, externe Programme auszuführen. Über *safe_mode_exec_dir* kann der Administrator effektiv steuern, was ausgeführt werden darf. Alles, was nicht explizit erlaubt wurde (durch Eingriffe des Administrators), funktioniert nicht. Betroffen von diesen Restriktionen sind *exec*, *system*, *passthru* und *popen*. *Shell_exec* und der Backtick Operator sind durch den safe_mode deaktiviert. Welche weiteren Funktionen von *safe_mode* Einschränkungen betroffen sind, kann unter [22] nachgeschlagen werden.

2.2.1.4 Der Parameter register_globals

Der *register_globals* Parameter ist seit PHP 4.2.0 standardmäßig deaktiviert. Es gibt aber trotzdem Gründe ihn genauer zu beleuchten. Erstens, weil er unter Umständen starke Auswirkungen auf die PHP Funktionalität hat, zweitens weil er bei der Sicherheitsüberprüfung fremder Webserver hilfreich sein kann und drittens, weil es immer noch Programmierer gibt, die mit *register_globals = On* programmieren und sie vom Administrator verlangen.

Register_globals erlaubt bei unsauberer Programmierung, dass ein Angreifer auf einfachste Weise Werte von Variablen setzen kann, indem er die Werte z. B. per GET-Request an eine PHP Anwendung überreicht. Wichtig ist die Bemerkung „bei unsauberer Programmierung“. Bei sicher programmierten Skripts ist es irrelevant ob *register_globals* aktiviert oder deaktiviert ist. Abgesehen davon, werden unsicher programmierte Skripts durch das Deaktivieren von *register_globals* nur bedingt sicherer. Das Einzige was sich ändert ist, dass der Programmierer sich einmal Gedanken machen muss, wo die Werte seiner Variablen herkommen. Dazu ein kurzes Beispiel, welches einen Benutzer als Administrator erkennt, wenn er das Passwort „geheim“ kennt:

```
<?php
if($password == "geheim"){
    $authorized = "true";
}

if ($authorized == "true"){
    echo "hallo admin"
}else{
  echo "hallo user"
};

?>
```

Ein Angreifer braucht das Passwort nicht zu kennen, er setzt die *$authorized* Variable über eine manipulierte URL direkt:

http://www.example.de/admin.php?authorized=true

Ist *register_globals* deaktiviert, so ist es für den Angreifer nicht möglich Variablen von externer Seite im Programm zu setzen. Andererseits hätte es auch gereicht, wenn der Programmierer *$authorized* auf „false“ gesetzt hätte, wenn das Passwort falsch ist. Im Prinzip ist *register_globals*, genau wie *magic_quotes*, nur eine Maßnahme um die Benutzer vor sich selbst zu schützen.

2.2.1.5 Die magic_quotes* Parameter

Die *magic_quotes** Parameter sorgen dafür, dass einfache Anführungszeichen und andere Sonderzeichen (double quote, backslash, NULL), die durch Datenbanken interpretiert werden, mit einem Backslash maskiert und damit von der Datenbank ignoriert werden. Einfache SQL Injection Angriffe, wie sie in ziemlich jedem SQL-Injection Paper (z. B. [10]) beschrieben werden, können dadurch verhindert werden. Der am häufigsten verwendete Parameter ist *magic_quotes_gpc*. Er filtert GET-, POST- und COOKIE-Daten. Zusätzlich können auch zur Laufzeit generierte Daten gefiltert werden. Dies passiert über *magic_quotes_runtime* Einstellungen. Diese Parameter klingen nach einem Patentrezept gegen SQL Injections, sind es aber nicht.

Einem unerfahrenen Programmierer werden *magic_quotes** eventuell die Kompromittierung seiner Datenbank ersparen, da viele Eingaben eines Angreifers unschädlich gemacht werden. Kontraproduktiv wirkt sich dieses Feature allerdings auf Anwendungen aus, die bereits eine Inputvalidierung mit den gegebenen PHP Funktionen haben. Dies führt dann zu doppelter Maskierung und damit zur Verfälschung der Daten. Ein Programmierer muss dann dafür sorgen, dass alle übertragenen Eingaben zuerst von *magic_quotes* gesäubert (z. B. mit „strip_slashes“) werden, bevor er sie weiterverwendet.

Auch wenn *magic_quotes* einen gewissen Nutzen haben, so sollten sie im Interesse der Mehrheit der Benutzer deaktiviert bleiben, so wie es die Standardeinstellung in PHP vorschlägt. Der Programmierer muss sich dann zwar selbst um die Behandlung von Sonderzeichen kümmern, hat so aber auch die Möglichkeit speziell kodierte (z. B. hexadezimal, oktal) Anfragen zu sanktionieren.

Zu Beginn des Kapitels wurde festgestellt, dass das mod_php Modul keine besonders hohe Sicherheit bietet, deshalb werden nun einige Ansätze vorgestellt, die versuchen, PHP besser abzusichern.

2.2.2 Hardened-PHP

Das Hardened-PHP Projekt[13] besteht aus einer Reihe von Patches, die Sicherheitsfeatures in PHP ergänzen, um bekannte Probleme in fehleranfälligen PHP-Skripts zu beheben, soll aber auch gegen potentiell unbekannte Schwachstellen in PHP helfen. Das Projekt wird von Stefan Esser geleitet, der sich als Securityexperte unter Anderem in den Bereichen Websecurity und Buffer Overflows einen Namen gemacht hat.

[13] http://hardened-php.sourceforge.net/

Hardened-PHP bietet erweiterte Loggingmechanismen und neue Filterfunktionen, die in PHP nicht existieren. Speziell fehlgeschlagene SQL Anfragen können durch die neuen Mechanismen geloggt werden, um SQL-Injection Angriffe zu erkennen. Außerdem gibt es eine Reihe von Variablen, die weder über GET und POST noch über Cookies gesetzt werden können um die Sicherheit zu erhöhen. Zudem ist es möglich, Beschränkungen für Cookie, GET und POST Variablen einzuführen. Beispielsweise kann die Anzahl der Variablen und auch die Länge von Arrays definiert werden. Interessante Features bietet die Behandlung von hochgeladenen Dateien. So kann ihre Anzahl limitiert werden oder sie können an ein externes Skript zur Überprüfung weitergegeben werden. Das ist vorteilhaft, wenn die Dateien durch Drittanbieterprogramme auf Viren überprüfen werden sollen.

Hardened-PHP bietet darüber hinaus - nach eigenen Angaben - eine bessere Widerstandsfähigkeit gegen Buffer Overflows und Formatstring Angriffe. Eine sehr sinnvolle Erweiterung ist auch, dass Dateien, die über PHP auf den Webserver hochgeladen wurden oder die sich extern befinden, nicht mittels „include“ oder „require“ ausgeführt werden können. Falls man gezwungen ist, die *allow_url_fopen* Option zu aktivieren, sollte man diesen Patch in Erwägung ziehen.

Der Patch an sich scheint bisher keine weiteren Schwachstellen in PHP geöffnet zu haben, jedenfalls wurden bis zum jetzigen Zeitpunkt (09.08.2005) auf den bekannten Securitymailinglisten keine Sicherheitslücken gemeldet. Das entscheidende Manko an mod_php kann diese Patchsammlung natürlich nicht beheben, denn dazu müsste jeder Benutzer seine Skripts unter einer eigenen UID / GID ausführen. Eine Möglichkeit das zu erreichen bietet suPHP.

2.2.3 suPHP

SuPHP ist ein Tool, das es ermöglicht PHP Skripts mit den Berechtigungen des Besitzers auszuführen. Es besteht aus einem Apachemodul (mod_suphp) und einem Setuid-Root-Binary (suphp), welches vom Apachemodul aufgerufen wird, um die UID des PHP Prozesses zu ändern. SuPHP wird dabei als CGI ausgeführt. Die Installation von suPHP verläuft unter Debian Sarge problemlos. Nach der Installation ändert man den Benutzer der PHP Skripts wie gewünscht und die Skripts funktionieren wie mit mod_php. Nicht ausgeführt werden können PHP Skripts die root oder dem Apachen gehören. SuPHP wurde ursprünglich für Linux geschrieben, mittlerweile läuft es aber auch auf FreeBSD und möglicherweise auch auf anderen unixartigen Betriebssystemen.

Wenn man sich mit *phpinfo()* die Server API ansieht, ist zu erkennen, dass

sie sich von „Apache" auf „CGI / FastCGI" verändert hat. Dieser Umstand macht sich ebenfalls bemerkbar, wenn der Administrator Änderungen an der PHP Konfiguration vornimmt, denn er braucht den Apachen nicht mehr neu zu starten, damit die Änderungen wirksam werden. Die Ursache liegt darin, dass bei jedem PHP Aufruf das suPHP Binary gestartet wird. Trotz der Tatsache, dass PHP nun als CGI läuft, ist es nicht nötig, das suPHP Binary in das cgi-bin Verzeichnis eines jeden Benutzers abzulegen, oder in jedem Skript den Interpreter anzugeben.

Der sicherheitstechnisch relevante Vorteil des CGI Interfaces liegt klar auf der Hand. Der Administrator erstellt für jeden Kunden einen Systembenutzer, passt die Besitzrechte entsprechend an und setzt die Zugriffsrechte für die Dateien auf 640. Aufgrund der nun fehlenden Leserechte ist es für einen Angreifer, der eine andere UID hat, unmöglich über PHP auf die Dateien anderer Kunden zuzugreifen. Einige Provider überprüfen die Zugriffsrechte per Cronjob, was sehr zu Lasten der Performance geht und daher gut überlegt sein sollte. Beim Anlegen der Kunden bei einem großen Provider sollten mögliche Nebeneffekte beachtet werden, die bei großen UIDs auftreten können. Theoretisch scheint es in Linux möglich zu sein 2^{31} Benutzer anzulegen, was auch für große Provider ausreichen sollte.

Wie unter [29] und [28] nachzulesen ist, ist auch diese Software nicht frei von Fehlern. Die Beschreibung lässt jedoch vermuten, dass die Ausnutzung dieser Schwachstelle nicht einfach ist. SuPHP bietet damit eine deutlich höhere Sicherheit als mod_php.

Was für einen großen Provider allerdings problematisch sein dürfte, ist die Performance. Auf einem Testsystem (Pentium IV 1,8GHz, 512MB Ram, Debian Sarge, Apache2) im 100MBit/s LAN war die Performance von mod_php beim Apache Benchmark ca. sechs mal besser (4.498s vs. 26.668s). Der Benchmark wurde mit folgenden Parametern gestartet:

```
ab -n 1000 -c 2000 http://testrechner/phpinfo.php
```

PHP ist bei weitem nicht die einzige Skriptsprache, die auf Apache Webservern verbreitet ist. Historisch gesehen ist Perl ein Vorgänger von PHP beim Erstellen von dynamischen Webseiten, der immer mehr an Bedeutung verliert, aber vom Sicherheitsstandpunkt aus noch schwieriger zu kontrollieren ist als PHP.

2.3 Perl

Perl ist eine immer noch weit verbreitete Programmiersprache, um serverseitig Skripts auszuführen und damit dynamische Webseiten zu generieren.

Perlskripts werden häufig über die CGI Schnittstelle des Apache Webservers ausgeführt. Im einfachsten Fall werden die Skripts im cgi-bin Verzeichnis abgelegt und dort ausgeführt. Für jeden Benutzer oder Kunden werden üblicherweise eigene cgi-bin Verzeichnisse definiert. Der Administrator kann aber auch in beliebigen Verzeichnissen die Ausführung von CGI Skripts erlauben.

In jedem CGI Skript wird zu Beginn der Interpreter festgelegt. Für Perlskripts wäre das beispielsweise „#!/usr/bin/perl“. Wenn ein Perlskript im cgi-bin Verzeichnis liegt, ist die Dateiendung irrelevant, der Webserver sucht im Skript nach dem Interpreter und führt es aus. Würde man Pythonskripts ausführen wollen, so bräuchte der Interpreter nur auf „#!/usr/bin/python“ geändert werden. Sollen Skripts außerhalb des cgi-bin Verzeichnisses ausgeführt werden, müssen die Dateiendungen dem Apachen bekannt gemacht werden. Dies geschieht über die *AddHandler* Option. Vergisst der Administrator diese zu setzen, bietet der Webserver das Skript zum Download an, anstatt es auszuführen.

Ein Vorteil von Perl gegenüber anderen Skriptsprachen ist die Geschwindigkeit. Ein einfaches „Hello World!“, welches über den Apache Benchmark aufgerufen wurde, benötigt in Python etwa viereinhalb mal so lange wie in Perl. PHP ist als Modul, also nicht über die CGI Schnittstelle, allerdings noch fast viereinhalb mal schneller als Perl. Dieser Vergleich macht deutlich, dass die Skriptingtechnologie und die Art wie man sie einsetzt, einen erheblichen Stellenwert für die Performance hat.

Perl ist für einen Programmieranfänger weniger leicht zu erlernen als PHP und deutlich seltener in Hostingangeboten enthalten. Dies hat neben der geringeren Nachfrage oft auch Sicherheitsgründe, denn Perl ist deutlich schwerer abzusichern als PHP.

2.3.1 Sicherheitsrisiko Perl

Für Perl gibt es keine zentrale Konfigurationsdatei wie für PHP. Es gibt daher auch keine Parameter, um Perl einschränken zu können. Standardmäßig wird der Perl Interpreter durch den Apache Webserver aufgerufen, er läuft also mit seinen Rechten. Die einzige Möglichkeit zu verhindern, dass ein vom Angreifer geschriebenes Skript Schäden anrichtet, sind Dateisystemberechtigungen, die das Betriebssystem überwacht, und Limitierungen, die der Apache vorgibt.

Um den Betrieb des Webservers negativ zu beeinflussen, braucht es aber nicht einmal einen Angreifer. Ein kleiner Programmierfehler, der zu einer Endlosschleife führt, kann die CPU des Servers zu 100% auslasten. Selbst wenn der Programmierer seinen Fehler bemerkt, kann er nichts mehr tun,

um sein Skript zu beenden. Normalerweise wird er keinen Shellzugang haben, um das Skript mit *kill* zu beenden und selbst wenn er den hätte, so könnte er das Skript nicht beenden, da es mit Rechten des Webservers ausgeführt wird. Nur root kann fremde Prozesse beenden.

Um Problemen beim Ressourcenverbrauch zu begegnen, gibt es in Apache eine Reihe von Parametern, die diesen einschränken. Die wichtigsten Parameter sind in diesem Fall *RLimitCPU* und *RLimitMEM*. Damit kann man die CPU-Zeit auf die Sekunde und den Speicherverbrauch bytegenau regulieren. Ist die Zeit abgelaufen, wird das fehlerhafte Skript einfach beendet und ein „Internal Server Error“ ausgegeben. Die *RLimits* sind sehr flexibel, sie lassen sich für den gesamten Server, einzelne VirtualHosts, Verzeichnisse oder per *.htaccess* steuern. Sie unterstützen Soft- und Hardlimits.

Da es für Perl keine „*open_ basedir*“ Direktive gibt, muss mit Dateisystemberechtigungen gearbeitet werden. Um zu verhindern, dass jemand auf die Verzeichnisse anderer Kunden zugreift, verbietet man dem Webserver, dass er bestimmte Verzeichnisse anzeigen (listen) kann. Die Kundenverzeichnisse müssen dann nicht vorhersehbare Namen haben, sodass ein Angreifer nicht weiß, wo sich die für ihn interessanten Verzeichnisse befinden. Er benötigt für Angriffe den genauen Pfad, um Daten lesen zu können. Als konkretes Beispiel könnten die Ordnerberechtigungen folgendermaßen aussehen:

Alle Daten liegen in „/srv“, die Unterverzeichnisse mit den Kundendaten sind zufällig.

Rechte des /srv Verzeichnisses:

```
drwx--x--x    5 root www-data  120 Aug 12 09:10 srv
```

Rechte der Kundenverzeichnisse:

```
drwxr-x---   2 kunde1  www-data  112 Aug  12 09:17 masdjfdjiofgjdfinfg
drwxr-x---   2 kunde2  www-data   48 Aug  12 09:10 khljkotjhziogrnnbfn
drwxr-x---   2 kunde3  www-data   48 Aug  12 09:10 riptrettziruorutpzi
```

Diese Konfiguration erlaubt es jedem Benutzer, sein eigenes Verzeichnis zu listen, aber es verbietet den Zugriff auf die darüber liegende Verzeichnisstruktur. Wenn es dem Angreifer gelingt, die Verzeichnisnamen heraus zu finden, ist dieser Schutz natürlich wirkungslos. Deshalb müssen auch viele andere Verzeichnisberechtigungen geändert werden, da ein Angreifer sonst im einfachsten Fall die Konfigurationsdateien des Apache lesen könnte, in denen alle Pfade sichtbar sind. Welche Verzeichnisse dies im Einzelnen sind, muss

anhand des verwendeten Betriebssystems und der Ordnerstruktur durch den Administrator entschieden werden.

Zusätzlich zum Schutz der Verzeichnisse könnte man den Zugriff auf essentielle Programme verbieten. Im Normalfall wird ein Webserver keine Rechte zur Ausführung von Programmen im „/bin“ oder „/sbin“ Verzeichnis benötigten. Den Zugriff hierauf kann man also verbieten.

Die Sicherungsmaßnahmen für Perl gelten auch für PHP, sie sollten sogar unbedingt angewendet werden, da die PHP-eigenen Sicherheitsparameter aufgrund von Bugs nicht immer zuverlässig funktionieren. Eine mit suPHP vergleichbare Möglichkeit zur Absicherung von Perl bietet suExec.

2.3.2 suEXEC

SuEXEC funktioniert prinzipiell ähnlich wie suPHP. Es gibt einen CGI-Wrapper, der die Zugehörigkeit der UID und GID überprüft. Wenn suEXEC korrekt mit dem Apachen geladen wurde, findet man im Apachelog einen Eintrag wie:

[notice] suEXEC mechanism enabled (wrapper: /usr/lib/apache2/suexec2)

Falls suEXEC nicht von Hand kompiliert wurde, kann man die wichtigsten Parameter mit „suexec2 -V“ feststellen. Die Ausgabe wird etwa wie folgt aussehen:

```
-D AP_DOC_ROOT="/var/www"
-D AP_GID_MIN=100
-D AP_HTTPD_USER="www-data"
-D AP_LOG_EXEC="/var/log/apache2/suexec.log"
-D AP_SAFE_PATH="/usr/local/bin:/usr/bin:/bin"
-D AP_UID_MIN=100
-D AP_USERDIR_SUFFIX="public_html"
```

Das DOC_ROOT gibt an, in welchem Pfad sich die CGI Skripts befinden müssen, in diesem Fall unterhalb von /var/www. Versucht man CGIs in anderen Pfaden, beispielsweise /srv/www auszuführen, würde man eine „command not in docroot“ Fehlermeldung bekommen. Die GID_MIN und UID_MIN Parameter geben an, welche UID / GID die Skripts mindestens haben müssen. Dies verhindert, dass Skripts mit Rechten von Systembenutzern ausgeführt werden. Diese haben in der Regel UIDs / GIDs kleiner als 100. Weitere Informationen sind unter [6] online verfügbar. Es wird empfohlen, dass nur die Gruppe, der der Apache Webserver angehört, suEXEC ausführen darf. Dementsprechend werden die Berechtigungen gesetzt:

```
chown root /usr/lib/apache2/suexec2
chgrp www-data /usr/lib/apache2/suexec2
chmod 4750 /usr/lib/apache2/suexec2
```

Um suExec für Benutzer zu aktivieren, trägt man in der VirtualHosts Konfiguration den Benutzer und die Gruppe ein, unter der die Skripts später laufen sollen:

```
<VirtualHost *>
...
SuexecUserGroup kunde1 kunde1

<Directory /var/www/kunde1>
        Options +ExecCGI
</Directory>
...
</VirtualHost>
```

Dann müssen nur noch die Ordner- und Dateiberechtigungen auf den Benutzer und die Gruppe angepasst werden. Wichtig ist, dass keine Schreibberechtigungen für weitere Benutzer vergeben werden, da suEXEC sonst mit „directory is writable by others“ abbricht. Bei „Internal Server Error“ Meldungen lohnt sich ein Blick in suexec.log. Die Fehlermeldungen dort sind präzise und aussagekräftig. Hilfreich bei der Fehlersuche ist auch die Fehlertabelle in [23, Kapitel 6.3]. Die Skripts werden mit den Rechten des in der VirtualHosts Konfiguration eingetragenen Benutzers ausgeführt, die Performance verschlechtert sich dabei um gut 50% im Vergleich zu normalem Perl CGI.

Nachdem nun die größten Sicherheitsrisiken durch serverseitige Skriptsprachen behoben wurden, fehlt noch ein wichtiger Teil im Programmpaket. Für die Speicherung, Abfrage und Weiterverarbeitung von Daten sind Datenbanken sehr beliebt. Eine der bekanntesten und am weitesten verbreiteten im Webserverbereich ist MySQL.

2.4 MySQL

MySQL ist eine Open Source Datenbank, die sowohl kostenlos, als auch in einer kommerziellen Variante verfügbar ist. Beim Thema Datensicherheit und Performance von MySQL scheiden sich die Geister, trotzdem wird MySQL oft im Bereich der kleinen und mittleren Anwendungen eingesetzt. Der Umgang mit dem Thema Sicherheit ist hier weitaus leichter als beim Apache Webserver und den serverseitigen Programmiersprachen. Trotzdem gibt

es einiges zu beachten. Wie vor jeder Installation einer Software, ist sicher zu stellen, dass die Installationsdateien aus einer vertrauenswürdigen Quelle stammen.

2.4.1 Sichere Konfiguration

Nach der Installation ist MySQL normalerweise ausschließlich an das lokale Interface gebunden, reagiert also nicht auf Netzwerkzugriffe von außen. Ist der MySQL Server wider Erwarten von außen erreichbar, muss überlegt werden, ob dies nötig ist. Wenn ja, sollten Firewallregeln den Zugriff einschränken. Falls der Netzwerkzugriff nötig ist, sollten die Verbindungen bei MySQL Versionen vor 4.1.1 zusätzlich verschlüsselt werden, da der Verschlüsselungsalgorithmus für Passwörter nicht sehr sicher ist. Eine Möglichkeit zur Absicherung des Authentisierungsvorgangs bieten SSH-Tunnel. Alle anderen Daten, die von MySQL über ein unsicheres Netzwerk transportiert werden, können ab der Version 4.0 mittels integrierter OpenSSL Unterstützung verschlüsselt werden.

Der MySQL Server sollte grundsätzlich nicht mit Root-Rechten laufen, da jeder Benutzer mit *FILE* Rechten dann beliebige Dateien als root erstellen kann. Positiv anzumerken ist, dass es bei einer Standardinstallation nicht möglich ist, MySQL als root zu betreiben, da hierfür die „ user=root“ Option beim Kompilieren gesetzt sein muss. Gleiches gilt für die Verwendung von symbolischen Links. Sie sollten schon bei der Installation deaktiviert werden, um SymLink-Angriffe auszuschließen.

Wie auch bei anderen Teilen der Webserverinstallation müssen die Zugriffsrechte auf Dateisystemebene möglichst restriktiv gesetzt werden. Für das Datenbankverzeichnis braucht nur der Benutzer, unter dem der MySQL Server läuft, Lese- und Schreibzugriff. Für alle anderen Benutzer sollte er gesperrt sein.

Was leider nicht in MySQL implementiert ist, sind Quotas. Man kann also nicht für einzelne Benutzer festlegen, wie groß eine Datenbank werden darf. Eine einfache, aber effektive Möglichkeit mit diesem Problem umzugehen, sind tägliche Überprüfungen der Datenbankgröße und eine Benachrichtigung des betreffenden Benutzers, wenn das Limit überschritten wurde. Bei mehreren Verwarnungen könnte dann der Zugang gesperrt werden. Bei Dateisystemen, die Quotas unterstützen, ist es möglich die Datenbankdateien den Benutzern zuzuordnen und darüber die Quotas zu regeln. Dies könnte über symbolische Links geschehen, die prinzipiell aber vermieden werden sollten. Ein Paket, das diese Aufgaben automatisch erledigt ist im Rahmen des „Linux for Schools Project“ [15] entstanden.

Die meisten sicherheitsrelevanten Einstellungen lassen sich nicht global bestimmen, sie werden individuell für jeden Benutzer in der Benutzertabelle eingerichtet.

2.5 Benutzerverwaltung

Die Benutzerverwaltung über die MySQL Konsole ist zwar problemlos möglich, allerdings wenig übersichtlich. Um Fehler zu vermeiden wird empfohlen, ein grafisches Interface (z. B. phpMyAdmin[14]) zu verwenden. Wie in jedem System mit Benutzern, sollte jeder Benutzer ein eigenes, sicheres Passwort besitzen. Dies gilt natürlich umso mehr für Benutzer mit Root-Privilegien. Für jeden Benutzer lassen sich eine große Anzahl von unterschiedlichen Rechten definieren, einige sollten wohl überlegt vergeben werden.

Nicht-administrative Benutzer sollten auf keinen Fall das *PROCESS* oder *SUPER* Recht besitzen, da sie dadurch auf die Ausgaben von „mysqladmin processlist“ zugreifen können, wodurch unter Umständen Passwörter im Klartext lesbar sind. Das Gleiche gilt für das bereits erwähnte *FILE* Recht. Ein Benutzer kann damit Dateien mit Rechten des MySQL Servers erstellen, was nicht unbedingt erwünscht ist. Einem Angreifer ist es so möglich, Datenbanken anderer Kunden zu überschreiben, oder einen Denial of Service herbei zu führen, indem er die Festplatte mit sinnlosen Daten anfüllt. Dies gilt es unbedingt zu verhindern.

Wie auch in anderen Datenbanksystemen gibt es die Möglichkeit, Zugriffsrechte für Datenbanken und Tabellen zu vergeben. Jedem Benutzer können individuell Rechte vergeben werden, welche Befehle er wo ausführen darf. Normale Benutzer benötigen außer dem „USAGE“ Attribut keine globalen Rechte. Um Denial of Service Angriffe zu vermeiden, kann für jeden Benutzer festgelegt werden, wie viele Abfragen, Updates oder Verbindungen er pro Stunde machen darf.

Des Weiteren sollten alle Benutzer, die nach der Installation kein Passwort besitzen, gelöscht oder mit einem Passwort versehen werden. Es ist ebenfalls möglich zu definieren, von wo sich Benutzer zur Datenbank verbinden dürfen. Dies ist insbesondere für den Zugriff über ein Netzwerk von Bedeutung. Nach der Installation von MySQL gibt es eine Datenbank mit Namen „test“. Diese kann gelöscht werden, da sie nicht benötigt wird.

Ein weiterer wichtiger Punkt ist, dass nicht-administrative Benutzer keinen Zugriff auf die user-Tabelle der Datenbank „mysql“ haben dürfen. In der user-Tabelle stehen die verschlüsselten Passwörter, welche ein Angreifer eventuell nutzen kann, um die Klartextpasswörter herauszufinden.

[14] http://www.phpmyadmin.net

Nachdem nun beschrieben wurde wie ein Apache Webserver gegen Unbefugte abgesichert wird, analysiert das nachfolgende Kapitel einen erfolgreichen Angriff.

Kapitel 3

Analyse eines erfolgreichen Angriffs

Dieses letzte Kapitel beschreibt einen im April 2005 erfolgten Angriff auf einen Webserver und verdeutlicht damit am praktischen Beispiel, warum es wichtig ist einen Webserver abzusichern.

Mit steigender Verbreitung von Breitbandinternetanschlüssen in den Haushalten installieren immer mehr Privatnutzer Mail- und Webserver, die rund um die Uhr verfügbar sind. Die wenigsten Privatnutzer machen sich jedoch Gedanken über die Sicherheit ihrer installierten Server. Teilweise herrscht der Glaube, dass ihre Linux-Server sicher seien, weil sie auf Linux statt Windows laufen. Andere glauben, dass das automatische Installieren von Sicherheitsupdates ihren Windows Server zur uneinnehmbaren Festung macht. In mittelständischen Unternehmen mit einem „Hobbyadministrator“ findet man teilweise sogar ungepatchte Windows-Server, die ohne Firewall oder Paketfilter direkt an das Internet angeschlossen sind.

3.1 Ausgangssituation

Im Folgenden wird der Angriff auf einen privaten Apache Webserver analysiert, der über einen Breitbandanschluss mit statischer IP-Adresse an das Internet angebunden war. Als Betriebssystem wurde Gentoo Linux auf einem handelsüblichen x86 PC verwendet:

> Linux young 2.6.11-gentoo-r4 #1 Wed Mar 30 16:02:13 CEST 2005 i686 AMD Athlon(tm) XP 2800+ AuthenticAMD GNU/Linux

Als Webserver kam Apache in der Version 2.0.52 mit PHP 4.3.10 zum Einsatz, zusätzlich waren Mail- und DNS-Server installiert. Auf dem Server wurden gut ein Dutzend private Webseiten gehostet, also keine Angebote die hochverfügbar sein mussten. Abgesichert wurde der Server durch den im Linuxkernel integrierten Paketfilter *netfilter / iptables*[1]. Weder ein netzwerk- noch ein hostbasierendes Intrusion Detection System waren installiert, ebensowenig wurden Maßnahmen zur Absicherung des Apachen getroffen, wie sie in den vorangegangenen Kapiteln beschrieben sind.

Zur Auswertung der Zugriffe auf die unterschiedlichen Domains wurde AWStats in der Version 6.3-r2 verwendet, welches sich komfortabel über den im Betriebssystem enthaltenen Paketmanager installieren ließ. Am 18.01.2005 berichtete Heise unter [4] über einen Fehler in AWStats, der die Ausführung von Befehlen auf dem Webserver erlaubt. Möglich war dies durch die ungenügende Filterung des vom Benutzer beinflussbaren Parameters „configdir“. Der Artikel beruft sich auf ein iDEFENSE Advisory vom 17.01.2005 [14] in dem berichtet wird, dass der Hersteller bereits am 21.10.2004 über die Schwäche benachrichtigt wurde.

Dies verdeutlicht, dass der Administrator ausreichend Zeit hatte, Sicherheitsupdates zu installieren. Sicherheitslücken in AWStats sind keine Seltenheit, alleine dieses Jahr wurde bereits mehrfach über schwere Sicherheitslücken berichtet.

3.2 Feststellen des Einbruchs

Am 02.04.2005 wurde festgestellt, dass alle Startseiten, der auf dem Server gehosteten Domains defaced[2] wurden. Die Indexseiten wurden durch eine Nachricht der Gruppe „OutLaw“ ersetzt (siehe Abbildung 3.1).

Der Rest der Seiten schien unverändert, weshalb ein Backup eingespielt wurde, um die Spuren des Angriffs zu beseitigen. Anhand der hinterlassenen Nachricht war es relativ einfach festzustellen, woher der Angriff gekommen war. Auf Zone-H[3], einem Verzeichnis aktueller Defacements, war der Angriff gelistet. Auf den ersten Blick war es nicht möglich zu erkennen, wie der Einbruch abgelaufen war. In Frage kamen sowohl ein Bug im Apache oder in anderen Diensten als auch in den installierten Webapplikationen. Die Analyse der Logdateien brachte die entscheidenden Erkenntnisse.

[1]http://www.netfilter.org
[2]durch Fremdeinwirkung verändert
[3]http://www.zone-h.org

Abbildung 3.1: Defacement der Gruppe Outlaw

3.3 Analyse der Logfiles

Zur Analyse des Angriffs wurden sowohl die *access.log* als auch die *error.log* Dateien der unterschiedlichen VirtualHosts verwendet. Die Daten der Logfiles reichten zurück bis in den Juni 2004, boten also beste Bedingungen für eine ausführliche Analyse. Dabei musste jedoch die Möglichkeit in Betracht gezogen werden, dass der Angreifer unter Umständen auch Teile der Logfiles löschen konnte, um so seine Spuren zu verwischen.

Die ersten interessanten Einträge im error.log beginnen am 01.02.2005, also zwei Wochen nach der Veröffentlichung der AWStats Sicherheitslücke.

```
[error] sh: /awstats.xxxxxxx.conf: No such file or directory
[error] --21:25:19--  http://www.xflows.org/twiki/backdoor/r0nin
[error] [client 80.182.117.83]            => `r0nin'
[error] [client 80.182.117.83] Resolving www.xflows.org...
[error] [client 80.182.117.83] 217.67.140.156
[error] Connecting to www.xflows.org[217.67.140.156]:80...
[error] [client 80.182.117.83] connected.
[error] [client 80.182.117.83] HTTP request sent, awaiting response
[error] [client 80.182.117.83] 200 OK
[error] [client 80.182.117.83] Length: 19,242 [text/plain]
```

Das Log zeigt, wie jemand mit einer spanischen IP-Adresse eine Datei „r0nin“

herunterlädt, eine Backdoor. Im error.log waren jedoch nicht alle ausgeführten Befehle zu finden, das access.log war auskunftsfreudiger. Hier war zu erkennen, dass eine Reihe von Befehlen ausgeführt wurde, um sich ein Bild vom Server zu machen. Die bisherigen Erkenntnisse zeigten, dass die ersten erfolgreichen Angriffe unbemerkt blieben. Im Verlauf des Februar und März sind weitere Downloads zu erkennen, unter Anderem der eines eggdrop (IRC Bot).

Daneben gab es auch ein paar erfolglose Angriffe, so probierte am 20. Februar jemand die Kernel Version des Servers festzustellen, benutzte aber die falschen Parameter und ließ nach kurzer Zeit vom Ziel ab:

```
[error] [client 201.8.126.174] sh: ?configdir=: command not found
[error] sh: /awstats.xxxxxxx.conf: No such file or directory
[error] File does not exist: /srv/www/vhosts/awstats/awstats
[error] [client 201.8.126.174] uname: invalid option -- e
[error] [client 201.8.126.174] Try `uname --help' for more info
```

Am 08.03.2005 gab es ein interessantes Login auf einen brasilianischen FTP-Server, welches ebenfalls geloggt wurde:

> [Tue Mar 08 20:12:16 2005] [error] [client 200.165.225.64] –20:12:16–
> ftp://henriqueee:*password*@ftp.hpg.com.br/hacked.txt

Der komplette Inhalt des FTP Accounts wurde zur Untersuchung heruntergeladen, es befanden sich ca. 15MB Daten darauf, darunter einige Backdoors, ein UDP Flooder, Tools zum Verstecken von Prozessen, ein paar Konfigurationsdateien und diverse IRC Tools (BNCs und Bots).

Der entscheidende Angriff, der zur Verunstaltung der Webseiten führte, fand am 1. April statt. Er begann um 01:12:39 und endete um 07:50:50 MEZ. In den Logdateien dieses Zeitraums konnte man genau erkennen, wie der Angreifer sich zuerst ein Bild von der Lage machte und dann systematisch alle index.php und index.html Dateien durch eigene ersetzt. Gegen 17 Uhr des selben Tages kontrollierte er noch einmal, ob die Seiten noch so vorhanden waren, wie von ihm gewünscht.

Danach erfolgten noch ein paar sporadische Zugriffe über die *configdir* Variable, aber es passierte nichts weiter Interessantes. Die IP-Adresse, von der der Angriff kam, befindet sich im Netz eines brasilianischen Providers, es ist aber nicht gesagt, dass der Angreifer wirklich aus diesem Netz kam. Es ist genauso möglich, dass er nur einen brasilianischen Proxy verwendete, um unerkannt zu bleiben.

3.4 Motivation des Angreifers

Die Motivation des Angreifers, der das Defacing begangen hat, war vermutlich die sportliche Herausforderung. Alle Seiten die defaced wurden, sind in einer Rangliste in der Kategorie „mass defacement“ wieder zu finden. Es ging ihm also darum, möglichst viele Webseiten in kürzester Zeit zu defacen, um in der Rangliste möglichst weit oben zu stehen. Die Vorgehensweise dabei war denkbar einfach: Der Angreifer suchte mit Hilfe einer beliebigen Internetsuchmaschine nach einer verwundbaren Version von AWStats (oder einer anderen Webanwendung) und hinterließ eigene Seiten auf dem Webserver.

Vor strafrechtlichen Folgen brauchte sich der Angreifer nicht zu fürchten, da er, wenn er klug war, alle seine Angriffe über anonyme Proxies ausführte. Theoretisch hätte er sogar mehrere HTTP-Proxies über so genannte Anonymisierer hintereinander schalten können, um so die Wahrscheinlichkeit einer Entdeckung weiter zu vermindern.

Über die Motivation der anderen Angreifer lässt sich anhand der Apache Logfiles wenig sagen, es ist aber anzunehmen, dass der Webserver für Angriffe auf andere Ziele missbraucht werden sollte. Dazu scheint es aber nie gekommen zu sein, da man dies anhand eines hohen Datenübertragungsvolumen bei der monatlichen Abrechnung hätte sehen können.

3.5 Gegenmaßnahmen

Aufgrund der Tatsache, dass die ersten Angriffe lange unbemerkt blieben und kein hostbasiertes Intrusion Detection System installiert war, ist eine vollkommene Neuinstallation des Servers die sicherste Variante. Nach einer Neuinstallation sollten möglichst viele Maßnahmen dieses Dokuments übernommen werden und für eine geeignete Hostsicherheit gesorgt werden.

Dieser Angriff hätte durch verschiedene Maßnahmen, welche in den vorangegangenen Kapiteln beschrieben wurden, verhindert werden können. Wäre *suExec* zum Einsatz gekommen, wären Veränderungen der verschiedenen Webseiten aufgrund der Zugriffsrechte nicht möglich gewesen. Ein Chroot hätte aufgrund der fehlenden Programme verhindert, dass ein Angreifer die benötigten Befehle überhaupt ausführen kann. Geeignete Filter für *mod_security* hätten den Angriff schon frühzeitig erkennen und verhindern können. Korrekt gesetzte Zugriffsrechte für die Verzeichnisse hätten den Angriff ebenfalls scheitern lassen.

Kapitel 4

Resümee

Die populäre Apache / MySQL / PHP / Perl Installation auf einem unixartigen Betriebssystem ist aus sicherheitstechnischer Sicht sehr vielschichtig und erfordert eine genaue Planung der Anforderungen. Ohne ein gut abgesichertes Betriebssystem braucht man mit der Härtung des Webservers gar nicht erst zu beginnen.

Nach der Auswahl eines „sicheren“ Betriebssystems müssen das Anforderungsprofil an den Webserver festgestellt und entsprechende Sicherheitsmaßnahmen auswählt und umgesetzt werden. Hier sollte immer auch auf die langfristige Wartbarkeit geachtet werden. Ein gut funktionierendes (Sicherheits) Updatemanagement sollte unbedingt gewährleistet sein. Gerade im professionellen Einsatz bei Providern ergeben sich schnell Performanceprobleme. Hier muss zwischen Sicherheit und Performance verantwortungsbewusst abgewogen werden.

Soll ein Webserver langfristig, sicher und zuverlässig betrieben werden, wird qualifiziertes Personal benötigt, das in der Lage ist Bedrohungssituationen zu erkennen und gegebenenfalls schnell und effektiv zu reagieren.

Es wäre grob fahrlässig anzunehmen, dass quelloffene Systeme, wie sie hier gezeigt wurden, von Haus aus sicher sind und nach der Installation keine weitere Wartung benötigen. Auch in der OpenSource Entwicklergemeinde gehen die Meinungen über sicherheitsrelevante Fehler sehr weit auseinander, von daher kann der Administrator nur einen Teil zur Gesamtsicherheit eines Webservers beitragen.

Im Idealfall kann der Webserveradministrator entscheiden, welche Webapplikationen auf seinem Server betrieben werden. Meistens können die Administratoren jedoch nicht selbst bestimmen welche Webapplikationen installiert werden. Daher sind die beschriebenen Absicherungsmaßnahmen die einzige

Möglichkeit Kunden voneinander zu trennen und vor Angreifern aus dem Internet zu schützen.

Wie im letzten Kapitel gezeigt wurde, können Webapplikationen große Löcher in die Systemsicherheit reißen. Viele Maßnahmen zur Absicherung wären nicht nötig, wenn Webanwendungen sicher wären.

Literaturverzeichnis

[1] *The Jail Subsystem*. http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/jail.html, note=Kopie auf CD-ROM (jail.pdf), 2001.

[2] Andi: *Attacking Apache with builtin Modules in Multihomed Environments*. http://www.phrack.org/phrack/62/p62-0x0a_Attacking_Apache_Modules.txt, 2004. Kopie auf CD-ROM (phrack.pdf).

[3] Bauer, M. D.: *Linux Server Security 2nd*. O'Reilly, 2005.

[4] dab@ct.heise.de: *Statistik-Tool AWStats erlaubt Ausführen von Kommandos auf Servern*. http://www.heise.de/security/result.xhtml?url=/security/news/meldung/55246, 2005. Kopie auf CD-ROM (lsfp.pdf).

[5] Foster, J., V. Osipov, N. Bhalla und N. Heinen: *Buffer Overflow Attacks*. todo, 2222.

[6] Foundation, A. S.: *suEXEC Support*. http://httpd.apache.org/docs/2.0/suexec.html. Kopie auf CD-ROM (suexec.pdf).

[7] FraMe: *PHP4 cURL functions bypass open_basedir*. http://www.securityfocus.com/archive/1/379657/2004-10-26/2004-11-01/0, 2004. Kopie auf CD-ROM (bugtraq1.pdf).

[8] FrSirt: *PHP openbasedir Directive Security Bypass Vulnerability.pdf*. http://www.frsirt.com/english/advisories/2005/1862. Kopie auf CD-ROM (FrSIRT AdvisoriesPHP openbasedir Directive Security Bypass Vulnerability.pdf).

[9] fr@tp.heise.de: *War die Online-Demo gegen Lufthansa ein Flop?*. http://www.heise.de/newsticker/meldung/18620, 2001. Kopie auf CD-ROM (heise1.pdf).

[10] Fuentes, S.: *Hakin9 - Aufspüren und Ausnutzen von Bugs in PHP Code*. Hakin9, 3(10):20–26, 2005.

[11] GRUNER, A.: *Jails unter FreeBSD*, 2004. Quelle auf CD-ROM (jail2.pdf).

[12] H.ORG ZONE: *AZone-H.org - IT Security Information Network*. http://www.zone-h.org/, 2005. Kopie auf CD-ROM (zone1.pdf).

[13] HUSEBY, S. H.: *Sicherheitsrisiko Webanwendung*. todo, 2004.

[14] IDEFENSE, I.: *AWStats Remote Command Execution Vulnerability*. http://www.idefense.com/application/poi/display?id=185, 2005. Kopie auf CD-ROM (idefense.pdf).

[15] JONES, P.: *Linux for Schools Project*. http://www.lfsp.org/\#mysqlquota, 2005. Kopie auf CD-ROM (lsfp.pdf).

[16] MAJ, A.: *Securing PHP: Step-by-Step*. http://www.securityfocus.com/infocus/1706, 2003. Kopie auf CD-ROM (securityfocus3.pdf).

[17] MAJ, A.: *Securing Apache 2: Step-by-Step*. http://www.securityfocus.com/infocus/1786, 2004. Kopie auf CD-ROM (securityfocus2.pdf).

[18] MOGUL, J. C.: *The Case for Persistent-Connection HTTP*. ftp://gatekeeper.research.compaq.com/pub/DEC/WRL/research-reports/WRL-TR-95.4.pdf, 1995. Kopie auf CD-ROM (WRL-TR-95.4.pdf).

[19] O.SCHAD@WEB.DE: *Bug #32937 PHP don't respect trailing slashes in open_basedir*. http://bugs.php.net/bug.php?id=32937, 2005. Kopie auf CD-ROM (php1.pdf).

[20] PAB@CT.HEISE.DE: *DDoS-Erpressung gegen Online-Wettbüros*. http://www.heise.de/security/news/meldung/print/48613, 2004. Kopie auf CD-ROM (DDoS-Erpressung gegen Online-Wettbueros.pdf).

[21] PHP.NET: *A Note on Security in PHP*. http://www.php.net/security-note.php. Kopie auf CD-ROM (php3.pdf).

[22] PHP.NET: *Functions restricted/disabled by safe mode*. http://www.php.net/manual/en/features.safe-mode.functions.php, 2005. Kopie auf CD-ROM (debmail.pdf).

[23] RISTIC, I.: *Apache Security*. O'Reilly, 2005.

[24] SIMON GARFINKEL, ALAN SCHWARTZ, G. S.: *Practical Unix and Internet Sercurity 3rd*. O'Reilly, 2003.

[25] TASKINEN, J.: *http://bugs.php.net/bug.php?id=32937 in other versions, too*. E-Mail. Kopie auf CD-ROM (taskinen.pdf).

[26] THORBEN: *libapache2-mod-php4 - open_basedir bug - security*. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=323585, 2005. Kopie auf CD-ROM (debianbug.pdf).

[27] THORBEN: *open_basedir bypass vulnerability in PHP 4.4.0*. http://bugs.gentoo.org/show_bug.cgi?id=102943, 2005. Kopie auf CD-ROM (gentoo.pdf).

[28] VAN ACKER, S.: *SUPHP Design Flaw Local Privilege Escalation Weakness*. http://www.securityfocus.com/bid/11020, 2004. Kopie auf CD-ROM (securityfocus3.pdf).

[29] VAN ACKER, S.: *A word of caution on the use of suphp*. http://seclists.org/lists/bugtraq/2004/Aug/att-0320/suphp-advisory_txt, 2004. Kopie auf CD-ROM (seclist2.pdf).

[30] VET, A. DE: *Apache chroot(2) patch*. http://www.devet.org/apache/chroot/, Jun 2004. Kopie auf CD-ROM (devent.pdf).

[31] WEIMER, F.: *Document the bug fix policy regarding PHP Safe Mode*. http://lists.debian.org/debian-security/2005/07/msg00160.html, 2005. Kopie auf CD-ROM (debmail.pdf).