

# Absicherung eines (Web-)Applikationsservers auf Linux-Basis

Rene Schneider  
Hochschule der Medien  
Nobelstr. 10, 70569 Stuttgart  
e-mail: rs034@hdm-stuttgart.de

## Abstract

*In den letzten Jahren sind die Preise für dedizierte Server auf Linux-Basis, sogenannte Rootserver, kontinuierlich gefallen, während die Leistung der für wenig Geld erhältlichen Hardware auch das Hosten aufwendiger (Web-)Applikationen erlaubt. Dadurch bildete sich ein neuer Massenmarkt im Bereich Webhosting. Allerdings sind die Mieter solcher Server für die Administration in vollem Umfang selbst verantwortlich, was insbesondere auch Security-Maßnahmen betrifft. Dieses Paper widmet sich speziell dem Security-Aspekt beim Hosten von Applikationen auf Linux-basierten Servern und soll eine Übersicht über die gängigen Methoden und Techniken liefern.*

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Ausgangspunkt</b>	<b>3</b>
2.1	Erste Checks . . . . .	4
2.1.1	Portscan mittels nmap . . . . .	4
2.1.2	Abfrage offener Ports per netstat . . . . .	5
2.1.3	Was tun mit nicht benötigten Diensten? . . . . .	5
<b>3</b>	<b>Installation und Absicherung der Applikationsserver</b>	<b>6</b>
3.1	Sichere Konfiguration gängiger Applikationsserver . . . . .	6
3.1.1	Sichere Konfiguration von PHP . . . . .	6
3.1.2	Sichere Konfiguration eines Tomcat-AppServers . . . . .	8
3.2	Sicherheit durch Isolation des Applikationsservers . . . . .	10
3.2.1	Abschottung durch funktionalen User . . . . .	11

3.2.2	Abschottung durch chroot-Jail . . . . .	13
3.2.3	„Härten“ von Applikationsservern . . . . .	15
<b>4</b>	<b>Weitere Absicherungen</b>	<b>17</b>
4.1	Datenbankserver absichern . . . . .	17
4.2	FTP-Server absichern . . . . .	18
4.3	SSH-Server absichern . . . . .	19
4.4	Sinn und Unsinn von Firewalls . . . . .	19
4.4.1	Die Firewall als klassischer Paketfilter . . . . .	20
4.4.2	Die Firewall als Intrusion-Detection-System . . . . .	20
4.5	Absicherung gegen physikalischen Serverzugriff . . . . .	21
<b>5</b>	<b>Fazit</b>	<b>22</b>

# 1 Einleitung

Der klassische Weg, kleinere bis mittelgroße Webapplikationen günstig zu hosten, ist das Anmieten von sogenannten Shared-Hosting-Leistungen. Dabei erhält man als Kunde Webspace (in aller Regel mit Unterstützung für eine oder mehrere Plattformen zur serverseitigen Ausführung von Applikationen, z.B. PHP) sowie meist Platz auf einem Datenbankserver, ohne dass man sich dabei um die Administration des zugrundeliegenden Systems kümmern muss. Für das Einpflegen von Security-Patches, das Aktualisieren veralteter Software-Pakete und die Pflege der Server-Hardware ist der Hoster alleinig verantwortlich.

Dies hat gleichzeitig Vor- und Nachteile für den Anbieter einer Webapplikation zur Folge. Nachteilig ist, dass die Performance von Shared-Hosting-Paketen in der Regel davon abhängig ist, wie viele andere Kunden auf derselben physikalischen Maschine gehostet werden, und davon, welche Anwendungen diese Kunden betreiben. Auch bestehen nur sehr eng begrenzte Möglichkeiten, fehlende serverseitige Funktionen (PHP-Module, andere Webapplikationsserver, alternative Datenbank-Server) nachzurüsten: was der Hoster nicht ohnehin anbietet oder auf Nachfrage installiert, ist nicht zu erhalten.

Der große Vorteil liegt jedoch darin, dass man sich wirklich komplett auf das Anbieten und Administrieren der eigenen Anwendung konzentrieren kann, ohne sich um die darunterliegende Technik wie den Webapplikationsserver, das (zumeist Linux-basierte) Betriebssystem oder gar die Hardware kümmern zu müssen. Vor allem liegt die Verantwortung für sicherheitstechnische Probleme - Einbrüche in Server, Mißbrauch der Maschine zum Spam-Versand, für Hacking-Angriffe oder Verteilung von WareZ<sup>1</sup> - zunächst einmal beim Hoster. Dieser muss für Aktualisierung der Serversoftware sorgen sowie das zeitnahe Einpflegen von Security-Patches, wenn kritische Sicherheitslücken bekannt werden - der Kunde kann hierfür allein deswegen schon nicht in Verantwortung gezogen werden, weil er gar nicht die nötigen Rechte besitzt, um sich um diese Dinge zu kümmern. Einzig für die Aktualisierung der selbst installierten Webapplikation ist er verantwortlich.

Durch das Aufkommen günstiger dedizierter Server im Hosting-Bereich hat sich diese Situation seit einigen Jahren grundlegend geändert. Heute können sich auch Einzelpersonen mit geringen finanziellen Mitteln kom-

<sup>1</sup>Sammelbegriff für illegale, d.h. das Urheberrecht verletzende Kopien von Software oder Medien wie Filmen und Musik

plett eigene Serverhardware leisten, deren Leistung für viele Projekte gerade im Startup-Stadium ausreicht. Diese Angebotskategorie wird allgemein als „Rootserver“ bezeichnet, benannt nach der Tatsache, dass der Mieter eines solchen, in den meisten Fällen Linux-basierten Systems darauf Root-Rechte besitzt - als einzige Person sogar, der Provider selbst hat außer dem physischen Zugang zur Hardware keinerlei weitere Zugriffsmöglichkeiten. Dies impliziert, dass sich der Mieter - anders als bei den Shared-Hosting-Paketen - komplett selbst um die Software seines Systems kümmern muss, von den laufenden Anwendungen bis hinunter zum Betriebssystem.

Den unbestreitbaren Vorteilen dieser Server - günstiger Preis, alle Möglichkeiten dank Vollzugriff, gute Performance durch eigene, ungeteilte Hardware - steht also der Hauptnachteil der Eigenverantwortung gegenüber, insbesondere was Sicherheitslücken betrifft. Für jeglichen Mißbrauch eines Rootservers ist sein Mieter, der gleichzeitig auch sein Administrator ist, selbst verantwortlich. Diese Tatsache ist insbesondere aus dem Grund brisant, weil viele Mieter eines Rootservers anders als professionelle Administratoren nur vergleichsweise wenig Linux-Kenntnisse mitbringen; nicht wenige werden sogar durch einen solchen Server zum ersten Kontakt mit Sicherheitsmaßnahmen unter Linux gezwungen. Dementsprechend hoch ist die Gefahr, dass Rootserver bzw. das darauf laufende Betriebssystem und die Applikationsserver aufgrund mangelnder Kenntnisse schlecht abgesichert werden.

An dieser Stelle soll das vorliegende Dokument einhaken und eine Übersicht über die gängigen Maßnahmen der Absicherung von (Web-)Applikationsservern geben. Da ein möglichst sicheres Betriebssystem aber die Grundlage für einen sicheren Betrieb von Applikationsservern (und diese wiederum Grundlage für den sicheren Betrieb der Applikationen) ist, wird auch auf die grundsätzliche Absicherung von Linux-Betriebssystemen eingegangen.

## 2 Ausgangspunkt

Bereits bei der Wahl des Linux-Basissystems spielt die Security-Frage eine bedeutende Rolle, daher lohnt es, sich bereits an dieser Stelle einige Gedanken zu machen. In den meisten Fällen bieten Serverhoster eine Reihe von Linux-Distributionen zur Installation auf einem neu geordneten System an, oft sogar in mehreren Varianten pro Distribution. Welche Distribution man dabei wählt, ist zum Teil natürlich Geschmackssache und davon abhängig, ob mit einer Distribution bereits Erfahrungen gemacht worden sind - zum anderen Teil sollte man sich aber über einige ganz und gar objektive Punkte Gedanken machen:

**Paketauswahl:** Jede gängige Distribution bietet heutzutage ein Paketmanagement-System zur einfachen Installation und Aktualisierung von Anwendungen an - falls man nicht den zeitaufwendigen Weg gehen möchte, Applikationen selbst zu kompilieren, ist die Verwendung dieser Systeme in der Regel ein guter Weg. Da sich die Systeme von Distribution zu Distribution unterscheiden, unterscheidet sich auch die Paketauswahl teilweise deutlich. Falls man zum Betrieb seiner Applikation also einen bestimmten, weniger verbreiteten Applikationsserver oder spezifische Hilfsprogramme benötigt, sollte man vor der Distributionswahl die Paketlisten genauer in Augenschein nehmen und sowohl Verfügbarkeit als auch Aktualität der benötigten Pakete prüfen.

**Paketaktualität:** Speziell beim Bekanntwerden kritischer Sicherheitslücken ist es wichtig, dass die angebotenen Pakete möglichst zeitnah in aktualisierten Versionen zur Verfügung stehen. Bei den verbreiteten Distributionen (Debian, Ubuntu, SuSE, ...) ist dies in der Regel der Fall, bei weniger verbreiteten Distributionen ist ein genauerer Blick angebracht.

**Support-Zeitrahmen:** Von jeder Distribution gibt es in regelmäßigen oder unregelmäßigen Abständen neue Versionen. Da sich dadurch der Aufwand für die Pflege der Paket-Repositories aller jemals veröffentlichten Versionen mit der Zeit immer weiter erhöht, werden alte Versionen in der Regel nach einer gewissen Zeit nicht mehr weiter unterstützt. Dies bedeutet: keine Paket-Updates mehr, weder für Bugfixes noch beim Auftreten von Sicherheitslücken. Für einen verantwortungsvollen Server-Administrator ist spätestens bei Erreichen dieses Zeitpunkts ein Update auf eine noch unterstützte Distributionsversion Pflicht, daher sollte man sich vor dem

Aufsetzen eines neuen Servers bereits informieren, wie lange die Wunschdistribution in der ausgesuchten Version voraussichtlich noch unterstützt wird.

Ist eine passende Distribution gefunden, gilt es oftmals noch, zwischen unterschiedlichen Installationsvarianten zu wählen. Nicht selten werden sowohl eine Vollinstallation mit vielen oft genutzten Diensten (z.B. Apache, MySQL, PHP, eine Web-Admin-Oberfläche, ein MTA<sup>2</sup>, ein POP3-Server, ein IMAP-Server, ein FTP-Server und OpenSSH) als auch eine Minimalinstallation angeboten. Die Minimalinstallation enthält im Optimalfall nur das Basissystem sowie einen SSH-Server für die Fernadministration; ihr sollte stets den Vorzug vor dem Komplettpaket gegeben werden, auch wenn die Aussicht auf einen Server mit allen möglichen vorinstallierten Diensten und Tools auf den ersten Blick attraktiv sein mag.

Der Grund hierfür ist so einfach wie banal: Dienste, die nicht installiert sind, können keine Sicherheitslücken enthalten. In der Regel installiert man sich mit einer Komplettinstallation erheblich mehr an Software, als man für den Betrieb der gewünschten Endbenutzer-Anwendung auf dem Server wirklich braucht - oftmals kommen auf diesem Wege sogar Serverdienste auf den Rechner, von deren Existenz man nicht einmal etwas weiß. Durch die mittlerweile durch die Bank ausgereiften Paketverwaltungssysteme der Distributionen ist es jedoch ein Leichtes, fehlende Softwarepakete nachzuinstallieren, wenn erst einmal eine Basisinstallation vorhanden ist. Benötigt ein Paket zwingend das Vorhandensein anderer Pakete, lösen die Paketverwaltungen solche Abhängigkeiten selbstständig auf und installieren alles Nötige. Es gibt also - abgesehen von der geringfügig verkürzten Ersteinrichtungszeit - keinen wirklichen Grund, sich für ein umfangreiches Installationspaket zu entscheiden. Vom Standpunkt der Serversicherheit verbietet sich dieser Weg sogar, es sei denn man macht sich die Mühe, alle nicht benötigten Pakete nach der Installation zu identifizieren und zu entfernen. Da dies aber meist erheblich länger dauert als die Installation der benötigten Pakete und man leicht Pakete dabei „übersieht“ kann von dieser Herangehensweise nur abgeraten werden.

## 2.1 Erste Checks

Nachdem ein neuer Server mit einer möglichst minimalistischen Installation der gewünschten Linux-Distribution gemietet wurde, sollte man sich stets vergewissern, welche Serverdienste auf dem neuen System bereits laufen und Ports nach außen geöffnet halten. Hintergedanke hierbei: jeder offene Port ist ein potentielles „offenes Scheunentor“ ins System, denn jeglicher Schutz, den das Betriebssystem über seine Netzwerk-Konfiguration selbst gegen Angriffe von außen bietet, ist im Falle eines offenen Ports dahin - Rechner aus dem Internet **sollen** ja gerade auf offene Ports verbinden und darüber Daten austauschen können. Somit gilt hier dasselbe wie bei den installierten Programmen: Serverdienste, die nicht laufen (oder, noch besser, gar nicht erst installiert sind!) können keine Sicherheitsprobleme aufwerfen.

Um herauszufinden, auf welchen Ports Serverdienste lauschen, gibt es grundsätzlich zwei Herangehensweisen: entweder ein Portscan (von einem anderen Rechner über das Internet oder lokal auf dem System), oder eine Abfrage des eigenen Systems nach offenen Ports mittels *netstat*.

### 2.1.1 Portscan mittels nmap

Der Portscanner *nmap* ist wohl der bekannteste und beliebteste Portscanner unter Linux, daher ist er üblicherweise direkt aus den Paketquellen jeder Distribution erhältlich. Ist *nmap* installiert, kann ein Scan über alle auf der öffentlichen IP des Servers potentiell erreichbaren Ports durchgeführt werden.

```
1 root@ubuntu:~# nmap 192.168.0.216
2
3 Starting Nmap 4.62 ( http://nmap.org ) at 2009-02-10 17:29 CET
4 Interesting ports on ubuntu (192.168.0.216):
```

<sup>2</sup>MTA = Mail Transfer Agent; ein Serverprogramm zum Versenden, Empfangen und Weiterleiten von E-Mail. Bekannte MTAs sind z.B. Postfix oder Sendmail.

```

5 Not shown: 1714 closed ports
6 PORT      STATE SERVICE
7 22/tcp    open  ssh
8
9 Nmap done: 1 IP address (1 host up) scanned in 0.236 seconds

```

Wichtig ist, dass der Scan wirklich die öffentlich sichtbare IP des Servers berücksichtigt, nicht 127.0.0.1 oder „localhost“, da Serverdienste gezielt auf einem Interface Ports öffnen können (in diesem Beispiel wäre daher die - ohnehin aus einem privaten Netzbereich stammende - IP 192.168.0.216 durch die öffentliche IP des Servers zu ersetzen).

Der Scanner liefert anschließend eine Liste aller geöffneten Ports sowie die Namen der Services, die auf diesen Ports typischerweise laufen. Bei einer Minimalinstallation sollte *nmap* eine Ausgabe ähnlich der oben gezeigten liefern: nur ein Port ist offen, und dieser gehört zum SSH-Daemon.

### 2.1.2 Abfrage offener Ports per netstat

Das auf praktisch jedem Linux-System standardmäßig installierte Tool *netstat* kann bestehende Socketverbindungen<sup>3</sup> anzeigen. Dabei zeigt es sowohl Verbindungen über ein (IP-)Netzwerk als auch lokale, über Unix-Sockets aufgebaute Verbindungen an. Ein einfacher Aufruf des Programms ohne weitere Parameter liefert aber nur bestehende Verbindungen, keine offenen, d.h. sich im „LISTEN“-Status befindlichen Sockets. Um diese aufzulisten, muss der Parameter *-l* angehängt werden.

Auch dies liefert aber noch eine Reihe Unix-Sockets sowie UDP-Sockets, die uns typischerweise nicht interessieren. Angenehmer wird es, wenn man die Ausgabe von *netstat* noch per *grep* nach dem Schlüsselwort „tcp“ filtern lässt:

```

1 root@ubuntu:~# netstat -l | grep tcp
2 tcp      0      0  *:ssh      *:*        LISTEN
3 tcp6    0      0  [::]:ssh   [::]:*     LISTEN

```

Vor allem die dritte Spalte ist interessant, sie enthält die lokale Adresse und den Port des offenen Sockets, getrennt durch einen Doppelpunkt. Hier sehen wir, dass nur der SSH-Port auf allen Interfaces (erkennbar durch den Stern) geöffnet ist - in diesem Fall sogar für IPv4 und IPv6.

### 2.1.3 Was tun mit nicht benötigten Diensten?

Wenn auf einem neuen Server Dienste laufen, die nicht benötigt werden - etwa ein POP3-Daemon, obwohl der Server gar keine Mails zum Abruf bereithalten soll - dann sollten diese Dienste über das Paketmanagement der jeweiligen Distribution komplett deinstalliert werden. Das genaue Vorgehen ist hierbei vom Paketmanagementsystem abhängig, in aller Regel aber gut dokumentiert.

Ein erneuter Check mit *nmap* bzw. *netstat* nach einer Deinstallation eines Dienstes verschafft Gewissheit darüber, dass der zugehörige Port nun wirklich geschlossen ist.

<sup>3</sup>Ein Socket ist gewissermaßen ein Kommunikationsendpunkt; ein Netzwerk-Socket ist typischerweise charakterisiert durch eine lokale Adresse plus Port sowie eine entfernte Adresse plus Port.

## 3 Installation und Absicherung der Applikationsserver

Sind auf dem Server nur noch die wirklich benötigten Dienste aktiv, besteht der nächste Schritt in der Installation der jeweils benötigten Applikationsserver. Auch hier ist das Vorgehen je nach Paketverwaltungssystem sehr unterschiedlich, daher wird auf die reine Installation der Paketquellen hier nicht weiter eingegangen - entsprechende Informationen sind in der Dokumentation zur Distribution nachzulesen.

Was dann folgt, ist der eigentlich interessante Part: die möglichst gute Absicherung der installierten Applikationsserver. Diese Absicherung kann zunächst einmal in zwei Aspekte unterteilt werden.

**Absicherung durch Konfiguration:** Die meisten Applikationsserver bieten Konfigurationsoptionen an, über welche Aspekte ihres Verhaltens geregelt werden können. Einige Einstellungen wirken sich dabei mitunter stark auf die Gesamtsicherheit des Applikationsservers und damit auch auf die Sicherheit der darauf laufenden Applikationen aus - zumeist in negativer Weise. Wie weit und auf welche Weise hier abgesichert werden muss hängt direkt vom eingesetzten Applikationsserver ab.

**Absicherung durch Isolation:** Wenn ein Angreifer tatsächlich eine Verwundbarkeit in einer laufenden Applikation oder in einem Applikationsserver selbst finden konnte, besteht immer noch die Möglichkeit der Schadensbegrenzung. Entsprechende Vorbereitungen müssen jedoch getroffen worden sein; das Stichwort hierbei lautet „Isolation“: Isolation der Applikationsserver vom Betriebssystem und Isolation von Applikationsservern und Datenbanken untereinander.

### 3.1 Sichere Konfiguration gängiger Applikationsserver

Im Folgenden soll es um die Absicherung konkreter Applikationsserver durch sinnvolle, an höchstmöglicher Sicherheit orientierte Konfiguration gehen. Bei den näher vorgestellten Applikationsservern handelt es sich um PHP und Tomcat, zwei weit verbreitete serverseitige Umgebungen für die Ausführung von Webapplikationen, die häufig auf dedizierten Servern zum Einsatz kommen.

#### 3.1.1 Sichere Konfiguration von PHP

Die Kombination „Apache Webserver + PHP-Parser“ ist wohl die - gemessen an der reinen Zahl der Installationen - am weitesten verbreitete Applikationsserver-Lösung für Webapplikationen. Den hohen Verbreitungsgrad hat PHP in erster Linie einem relativ einfachen Einstieg mit flacher Lernkurve zu verdanken - und diese wiederum wurde unter anderem durch diverse „Komfortfunktionen“ erreicht, von denen einige von einem sicherheitstechnischen Standpunkt aus betrachtet aber leider alles andere als positiv zu bewerten sind.

In den zurückliegenden Jahren, in denen PHP seinen Ursprung als einfache Lösung für einfache und kleine Webanwendungen in immer größerem Maß hinter sich lassen und bereit für den Einsatz als Plattform für kritische, komplexe und hochfrequentierte Webanwendungen werden wollte, hat sowohl bei den Entwicklern des PHP-Interpreters selbst als auch bei vielen PHP-Entwicklern ein Umdenken eingesetzt, bei dem die Sicherheit weiter in den Vordergrund gerückt ist. Nicht zuletzt getrieben durch viel „mediale Schelte“ für die Sprache PHP und darin geschriebene Webanwendungen, die häufig durch eklatante Sicherheitslücken aufgefallen sind, sind einige der erwähnten Komfortfunktionen heutzutage problemlos deaktivierbar, weil zum Einen entsprechende Optionen in der PHP-Konfiguration vorgesehen und zum Anderen die meisten populären PHP-basierten Anwendungen so umgearbeitet worden sind, dass sie auch in einer auf Sicherheit bedacht konfigurierten PHP-Umgebung laufen.

Ein großes Hindernis können jedoch noch eigene Anwendungen darstellen, die nicht unter Berücksichtigung der im Folgenden vorgestellten Sicherheitsrichtlinien bei der PHP-Konfiguration erstellt worden sind. An die-

ser Stelle muss eine Entscheidung getroffen werden: entweder die Applikation wird entsprechend umgeschrieben, oder das erhöhte Sicherheitsrisiko bewusst in Kauf genommen.

Bei PHP werden die allermeisten Einstellungen, die direkt mit der Sicherheit zusammenhängen, in der Datei „php.ini“ global über alle in einer PHP-Instanz ausgeführten Webapplikationen festgelegt. Einerseits kann man als Administrator auf diese Weise zwar sehr einfach sicherheitskritische Einstellungen abändern, muss aber auf der anderen Seite dafür sorgen, dass auch wirklich alle installierten Applikationen mit den verschärften Einstellungen arbeiten können, denn für eine einzelne Applikation können viele der Einstellungen leider nicht gelockert werden.

Wo sich die „php.ini“ genau befindet, ist von der konkreten PHP-Installation abhängig, in der Regel ist sie unter Linux jedoch im Verzeichnis /etc/, dem klassischen Ablageplatz für Konfigurationsdateien, zu finden. Im Zweifelsfall hilft ein PHP-Script mit dem Befehl „phpinfo()“ weiter, welches darüberhinaus auch die Zustände aller globalen Konfigurationsparameter verrät und daher gut zur Kontrolle einer Konfigurationsänderung dienen kann.

Die interessanten Parameter in der Konfiguration sind die Folgenden:

**allow\_url\_fopen:** PHP kann die Nutzung von Funktionen zum Lesen lokaler Dateien für den Zugriff auf entfernte Dateien mittels Standardprotokollen wie HTTP oder FTP erlauben. Diese Funktion ist zunächst praktisch, kann doch der Entwickler Dateien auf anderen Servern wie lokale Dateien lesen. Die Gefahr liegt aber darin, dass ein Angreifer, der es schafft, einen derartigen Zugriff auf eine URL seiner Wahl umzuleiten, ebenfalls die Fähigkeit erlangt, Dateien von fremden (z.B. seinen eigenen) Servern zu lesen, ohne dass die Applikation dies bemerkt. Wenn diese Funktionalität nicht zwingend benötigt wird, sollte sie daher abgeschaltet werden.

**allow\_url\_include:** Noch gefährlicher als das Lesen entfernter Dateien ist die durch diesen Parameter kontrollierte Möglichkeit, Quellcode-Dateien von anderen Servern direkt in Scripts inkludieren zu können. Dieses Feature wurde in der Vergangenheit häufig ausgenutzt, indem eine Anwendung dazu überlistet wurde, eine dynamisch eingebundene PHP-Datei nicht vom lokalen Server, sondern vom Server des Angreifers zu laden. Auf diese Weise kann ein Angreifer praktisch beliebigen Code direkt in die Anwendung injizieren. Die Funktion wird eher selten für legitime Zwecke gebraucht, kann und sollte daher in den meisten Fällen deaktiviert werden.

**disable\_functions:** Über diesen Parameter können beliebig viele Funktionen (durch Komma getrennt) angegeben werden, die gesperrt werden sollen. Es kann hierbei keine generelle Empfehlung gegeben werden, was man guten Gewissens sperren kann und was nicht, da dies sehr stark davon abhängt, ob eine installierte Anwendung die Funktion zwingend erfordert. Ein guter Grundsatz ist es jedoch, Funktionen, die eine Ausführung beliebiger Shell-Kommandos auf dem System erlauben - wie z.B. *exec()* oder *shell\_exec()* - generell zu verbieten, wenn möglich. Vor dem Setzen dieser Konfigurationsoption ist jedoch ein Blick auf die Auswirkung der Option „safe\_mode“ anzuraten, denn kann diese gesetzt werden, so sind eine Reihe potentiell gefährlicher PHP-Funktionen automatisch gesperrt oder stark eingeschränkt.

**display\_errors:** Die vom PHP-Interpreter auf der generierten Website ausgegebenen Fehlermeldungen lassen sich durch diesen Parameter entweder erlauben oder unterdrücken. Während es zur Entwicklungszeit hilfreich ist, Fehlermeldungen sofort zu sehen, genügt es im Live-Betrieb einer Anwendung, wenn Fehlermeldungen in einer Logdatei auf dem Server landen. Da Fehlermeldungen oftmals eine hervorragende Informationsquelle für Angreifer sind und viel über den internen Aufbau einer im Quellcode unbekanntem Applikation verraten können sollten diese auf keinem Fall allen Nutzern einer Anwendung dargeboten werden - insbesondere, weil die Nutzer ohnehin selten etwas Nennenswertes aus den Fehlermeldungen lesen können außer, dass irgendetwas schief gelaufen ist.

**open\_basedir:** Hiermit kann der „Wirkungsbereich“ im Dateisystem, innerhalb welchem PHP-Scripts auf Dateien zugreifen dürfen, eingeschränkt werden. Dateien und Scripts außerhalb eines hier angegebenen Ver-

zeichnisses können weder geöffnet noch inkludiert werden. Es empfiehlt sich dringend, diese Möglichkeit zu nutzen, um PHP-Scripts auf das Wurzelverzeichnis der gesamten Webpräsenz einzuschränken.

**register\_globals:** Bei diesem Parameter handelt es sich um ein Relikt aus früheren Zeiten, als es in PHP noch gängig war, auf per HTTP GET oder POST übertragene Daten über automatisch erstellte globale Variablen zuzugreifen. Die große Gefahr hierbei besteht darin, dass ein Angreifer auf diese Weise im Grunde jede beliebige Variable mit einem beliebigen Wert vorbelegen kann, bevor das eigentliche Script ausgeführt wird - wenn in diesem anschließend eine solchermaßen vorbelegte Variable ohne zwingende Initialisierung mit einem festen Wert abgefragt wird, steht darin der vom Angreifer gewünschte Wert. Da PHP weder zu einer Initialisierung noch überhaupt einer Deklaration von Variablen vor ihrer Nutzung zwingt, werden auf diese Weise bereits kleinste Unachtsamkeiten zu großen Sicherheitslöchern.

Da der dringend empfohlene Weg zum Zugriff auf GET- und POST-Daten schon seit Längerem über Nutzung der superglobalen Spezialarrays `$_GET` bzw. `$_POST` verläuft, ist eine Aktivierung dieser Option nur noch für alte (oder sehr schlecht geschriebene) Scripts erforderlich. In diesen Fällen ist man gut damit beraten, nach Aktualisierungen bzw. Alternativen der installierten Scripts zu suchen, um das äußerst gefährliche `register_globals`-Verhalten abschalten zu können. Eigene Anwendungen, die noch globale Variablen erfordern, sollten schnellstmöglich entsprechend umgeschrieben werden, insbesondere auch, weil die `register_globals`-Option in PHP 6 endgültig abgeschafft wird.

**safe\_mode:** Der sogenannte Safe Mode ist ein weiteres Relikt, welches sehr viele Probleme beim Einsatz verursachen kann, dessen Nutzen für die Sicherheit aber im Gegensatz zu `register_globals` relativ gering ist - daher ist dieser Modus in PHP 5 noch vorhanden, wird aus PHP 6 aber wieder entfernt. Die erste Wirkung des Safe Mode ist eine Deaktivierung mehrerer PHP-Funktionen, in erster Linie solcher, die die Ausführung beliebiger Systembefehle ermöglichen. Die zweite Wirkung ist die zumeist problematischere: PHP-Scripts dürfen nur noch auf Dateien lesend oder schreibend zugreifen, deren Eigentümer (im Dateisystem) mit dem Eigentümer der Scriptdatei übereinstimmt. Dies sorgt besonders häufig im Shared-Hosting-Bereich, mitunter aber auch beim Betrieb eines einzelnen Servers für Probleme: in PHP selbst neu angelegte Dateien etwa, deren Besitzer automatisch der User ist, unter welchem der PHP-Interpreter läuft, können plötzlich nicht mehr in PHP gelesen werden, wenn die Scriptdatei einem anderen Benutzer gehört als dem, unter welchem die PHP-Instanz läuft. Der Sicherheitsgewinn ist - bei einer ansonsten sicheren Konfiguration - zu vernachlässigen, weil praktisch alle sicherheitstechnisch sinnvollen Beschränkungen des Safe Mode auch über Einzelparameter gezielt realisier- und steuerbar sind (`disable_functions` und `open_basedir` etwa). Daher empfiehlt es sich, diese Option zu deaktivieren und die nötige Sorgfalt bei der Konfiguration der anderen Optionen aufzubringen.

### 3.1.2 Sichere Konfiguration eines Tomcat-AppServers

Der Tomcat-Applikationsserver (im Java-Enterprise-Bereich auch „Servlet-Container“ genannt, weil er wie ein Container für Anwendungen, die in der Regel in Form von Servlets implementiert sind, fungiert) stellt die von Sun akzeptierte Referenzimplementierung für einen Java-Enterprise-Applikationsserver dar. Während es heute zahlreiche Servlet-Container auf dem Markt gibt, von kostenlos und mit offenem Quellcode verfügbaren Systemen bis hin zu teuren Closed-Source-Anwendungen für den Unternehmensbereich, hat sich dieser konkrete Applikationsserver unter anderem aus diesem Grund einen beträchtlichen Nutzerkreis erarbeiten können.

#### 3.1.2.1 Konfiguration des Containers

Die Konfiguration des Tomcat-Containers ist - insbesondere für Einsteiger in die Materie - erheblich komplexer als die Konfiguration beispielsweise einer PHP-Installation. Der Grund hierfür sind die hierarchisch angeordneten, unterschiedlichen Konfigurationsdateien für die einzelnen konzeptuellen Ebenen des Applikationsservers. Im Folgenden werden einige gängige Maßnahmen vorgestellt, um die Tomcat-Konfiguration auf

produktiv eingesetzten Servern auf Sicherheit zu optimieren.

**Deaktivierung aller unnötigen Konnektoren:** Der Tomcat-Container bietet seine Dienste der Außenwelt über sogenannte Konnektoren an. Dabei kann es sich im Wesentlichen um zwei Arten von Konnektoren handeln: Die erste nimmt direkt HTTP-Requests (verschlüsselt oder unverschlüsselt) entgegen, spricht das zugehörige Servlet an und liefert das Ergebnis via HTTP zurück, stellen also selbst eine Art Webserver dar, während Konnektoren der zweiten Kategorie der Kopplung eines vorgelagerten Webserver mit einem Tomcat-Container dienen und dazu Protokolle abseits von HTTP verwenden.

Je ein Konnektor beider Kategorien ist bei einer Tomcat-Standardinstallation typischerweise aktiviert, meist ein HTTP-Konnektor auf Port 8080 und ein AJP-Konnektor auf Port 8009. Nur in seltenen Fällen wird man jedoch wirklich beide Konnektoren gleichzeitig benötigen, so dass einer vermutlich überflüssig ist und deaktiviert werden kann. Ein unnötigerweise aus dem Internet erreichbarer, offener Konnektor-Port ist gleichbedeutend mit einem unnötigen Serverdienst: wenn darin eine Sicherheitslücke auftauchen sollte, ist man in jedem Fall auf der sicheren Seite, wenn der Port gar nicht offen ist, bzw. der Konnektor, der ohnehin nicht benötigt wird, deaktiviert.

Da es sich bei den Tomcat-Konfigurationsdateien um XML-Files handelt, erfolgt die Deaktivierung eines Konnektors einfach durch Auskommentieren:

```
1 <!--  
2 <Connector port="8080" protocol="HTTP/1.1"  
3     connectionTimeout="20000"  
4     redirectPort="8443" />  
5 -->
```

**Default-Webapplikationen konfigurieren bzw. deaktivieren:** Tomcat bringt eine Reihe von oftmals per Default aktivierten Webapplikationen mit, die Sicherheitsrisiken darstellen können. Im Einzelnen sind das eine Manager-Applikation, eine Admin-Applikation, eine WebDAV-Applikation und eine Beispielanwendung. Jede dient einem eigenen Zweck, so dass sie einzeln betrachtet und sicher konfiguriert bzw. deaktiviert werden müssen. Im Einzelfall kann es allerdings sein, dass eine der Applikationen bereits deaktiviert bzw. gar nicht installiert ist; dies hängt stark davon ab, wie das Tomcat-Paket im Paketmanagement einer Distribution zusammengestellt wurde.

*Die Manager-Applikation:* Diese minimalistisch gehaltene Anwendung dient dazu, neue Webapplikationen im Betrieb zu deployen, laufende Applikationen zu deaktivieren oder neu zu initialisieren oder alle existierenden Sessions einer Applikation aufzulisten. Erreichbar ist die Manager-Applikation typischerweise über die URL „/manager“. Zwar ist sie standardmäßig aktiviert, aber der Zugriff auf sie ist normalerweise nicht möglich, weil keiner der Tomcat-internen User die nötige Berechtigung dazu besitzt. Soll das so beibehalten werden, etwa weil die Management-Funktionen gar nicht genutzt werden, dann empfiehlt es sich, auch die Manager-Applikation aus der Host-Konfiguration zu streichen, indem die zugehörige Kontext-Konfigurationsdatei „manager.xml“ entfernt wird.

*Die Admin-Applikation:* Während die Manager-Applikation zur Verwaltung einzelner Webapplikationen dient, wird mit der Admin-Applikation der gesamte Tomcat-Server verwaltet. Über sie können z.B. Konnektoren, DataSources und Hosts erstellt, gelöscht und verwaltet werden, ohne diese Aufgaben per Hand in der „server.xml“-Konfigurationsdatei durchführen zu müssen. Essenziell ist also ein Zugriff auf die Admin-Applikation gleichbedeutend mit Zugang zur Kern-Konfigurationsdatei des Tomcat-Servers, daher sollte dieser Zugriff entsprechend abgesichert werden. Einerseits kann dies ebenfalls - wie bei der Manager-Applikation - durch einen Tomcat-User mit entsprechendem Privileg geschehen, andererseits könnte aber auch der Zugriff auf die Admin-Applikation auf bestimmte IPs oder IP-Ranges beschränkt werden. Auch eine Deaktivierung der kompletten Applikation ist denkbar und kann gerade auf Produktivservern praktikabel sein, denn Konfigurationsänderungen sind immer noch über die XML-Konfigurationsfiles möglich, wenn auch etwas weniger komfortabel.

*Die WebDAV-Applikation:* WebDAV ist ein Protokoll zur Dateiübertragung und -Verwaltung. Über die WebDAV-

Applikation können z.B. die Dateien einer Webpräsenz editiert und verwaltet werden. Hierzu benötigt der Nutzer lediglich einen WebDAV-fähigen Client, der z.B. so ausgelegt sein kann, dass per WebDAV zur Verfügung gestellte Verzeichnisse transparent ins Dateisystem des Client-Rechners eingehängt werden. Benötigt man diese Funktionalität gar nicht, ist es auch hier ratsam, die Applikation komplett zu deaktivieren oder zu deinstallieren, um jegliche damit verbundenen Sicherheitsrisiken bereits im Keim zu ersticken.

*Die Beispiel-Applikation:* Tomcat bringt standardmäßig einige Beispiel-Servlets und -JavaServerPages mit, die zeigen sollen, wie der Tomcat-Container praktisch eingesetzt werden kann. Diese Anwendungen sind - insbesondere auf einem Produktivsystem - oft überflüssig und sollten daher deaktiviert werden.

### 3.1.2.2 Konfiguration der JVM

Zur Erhöhung der Sicherheit bei der Ausführung von Applikationen bietet die Java Virtual Machine (JVM) ein umfassendes Hilfsmittel: den Security Manager. Dabei handelt es sich um einen Mechanismus, über welchen ein Administrator einer Anwendung gezielt bestimmte Aktionen wie das Lesen einer Datei oder das Öffnen eines Sockets erlauben oder verbieten kann.

Den Security Manager einzuschalten ist denkbar einfach: es genügt, den Kommandozeilenparameter „-security“ beim Aufruf des Tomcat-Startscripts zu spezifizieren. Das größere Problem ist die Erstellung und Pflege des sogenannten Policy-Files. Dieses Regelwerk bestimmt letztlich, welche Befugnisse eine laufende Applikation erhält. Während Tomcat selbst bereits ein Default-Policy-File mitbringt, welches dem Applikationsserver und seinen Libraries eine Reihe von zur Ausführung von Applikationen unabdingbaren Rechten sowie allen Webapplikationen einen Satz von grundlegenden Rechten einräumt, ist es in aller Regel unabdingbar, einer Applikation gezielt weitere, zu ihrer Ausführung nötige Rechte einzuräumen. Hierzu muss das Policy-File entsprechend erweitert werden. Eine umfassende Erklärung zur Konfiguration des Java Security Managers sprengt jedoch aufgrund der enormen Komplexität des Themas den Rahmen dieses Papers; bei Interesse sei daher auf weiterführende Literatur wie [1] und [2] verwiesen.

Erwähnt werden soll jedoch, dass die Nutzung des Security Managers eine zusätzliche Sicherheitsschicht darstellen kann, deren Aktivierung aber leider oft mit viel Ärger und Arbeit bei der Erstellung und Wartung der Policy-Files für eine bestimmte Webapplikation und einem gewissen Performance-Verlust einher geht. Ob die zusätzliche Sicherheit diesen Preis wert ist, muss in jedem Einzelfall gezielt evaluiert werden - eine grundsätzlich gültige Aussage ist hier nicht zu treffen, denn Tomcat selbst ist auch ohne Security Manager bereits eine vergleichsweise sichere Plattform. Für hochkritische Anwendungen kann der Security Manager aber in der Tat eine unabdingbare weitere Abwehrmaßnahme gegen potentielle Angreifer darstellen.

## 3.2 Sicherheit durch Isolation des Applikationsservers

Zwar kann man durch die gebotene Sorgfalt bei der Anwendungsentwicklung und sichere Konfiguration des Applikationsservers effektiv für einen sicheren Applikationsbetrieb sorgen. Verlassen sollte man sich auf diese beiden „Schutzwälle“ jedoch nicht; die Praxis zeigt immer wieder, dass Sicherheitslücken auch in vermeintlich hochqualitativer und unter dem Sicherheitsaspekt entwickelter Software nicht auszuschließen sind. Das gilt in gleichem Maß für die Webapplikationsserver, welche die Plattform für die eigentlichen Applikationen darstellen.

Es ist also vorteilhaft, weitere Sicherheitsschichten einzuziehen, um im Falle eines Falles - also dem erfolgreichen Angriff auf eine verwundbare Applikation bzw. einen verwundbaren Applikationsserver - den Schaden so weit wie möglich zu begrenzen.

Wenn ein Angreifer eine Applikation erfolgreich attackiert hat, ist stets davon auszugehen, dass der Angreifer sich von diesem Punkt an mit den Rechten des Applikationsprozesses frei auf dem System bewegen kann.

Die naheliegendste Maßnahme zur Einschränkung dieser Bewegungsfreiheit liegt also darin, den Application Server unter einem funktionalen Nutzer-Account mit reduzierten Rechten zu betreiben statt als allmächtiger „root“. Diese Rechte sollten im Idealfall gerade ausreichend sein, dass der Application Server seiner Aufgabe nachkommen kann - dieses Prinzip ist auch unter dem Namen „POLA“ oder „Principle of Least Authority“ bekannt.

Vor einer solchen Einschränkung steht zunächst die Frage, welche Rechte eine konkrete Webapplikation überhaupt benötigt. Grundsätzlich werden die meisten Webapplikationen lesenden und in vielen Fällen auch schreibenden Zugriff auf ihr Applikationsverzeichnis erfordern. Darüberhinaus steht meist eine Verbindung zu einer Datenbank auf der Wunschliste, entweder über TCP oder ein Unix-Socket-File. Je nach Anwendung können aber weitere Bedürfnisse hinzukommen, etwa die Möglichkeit, ein Programm oder Script auf dem System auszuführen.

Steht fest, welche Rechte erforderlich sind, können verschiedene Methoden evaluiert werden, den Applikationsserver vom restlichen System abzuschotten.

### **3.2.1 Abschottung durch funktionalen User**

Eine übliche Maßnahme, um die Rechte eines Applikationsservers auf dem System zu beschneiden und ihn somit zu isolieren, ist die Ausführung des Applikationsserverprozesses unter einem speziellen, nur für diesen Zweck angelegten System-User mit geringen Rechten. Die Vorteile liegen auf der Hand: Während ein Angreifer durch Übernehmen einer Anwendung mit Root-Rechten selbst Root-Rechte und damit praktisch unbegrenzten Zugriff auf das gesamte System erlangt, ist ein Angreifer mit normalen User-Rechten zunächst eingeschränkt, es sei denn er findet eine Verwundbarkeit im System selbst, um sich Root-Rechte zu beschaffen. Bei unter geringeren Rechten laufenden Applikationsservern handelt es sich also wie bei den meisten Sicherheitsmaßnahmen auch nicht um ein Allheilmittel, sondern um eine zusätzliche Sicherheitsschicht.

#### **3.2.1.1 Ausführung der Kombination Apache + PHP unter funktionalem User**

PHP kann nicht „standalone“ als Webapplikationsserver genutzt werden, sondern wird in der Regel als Modul in einen Webserver integriert. Sehr beliebt ist hierbei die Kombination „Apache + PHP“, da die Zusammenarbeit dieser beiden Programme im Praxiseinsatz tausendfach erprobt wurde, einfach einzurichten ist und mit dem Apache-Webserver ein sehr funktionsreicher Open-Source-Webserver mit guter Performance und Stabilität zur Verfügung steht.

Da PHP als Modul im Apache-Webserver läuft, läuft es im selben Prozess und übernimmt seine Rechte. Der Fokus muss also darauf liegen, den Apache-Webserver nicht als Root, sondern unter einem normalen User-Account zu starten. Hier stört jedoch eine Linux-Eigenheit: ein Socket für eingehende Verbindungen auf den Netzwerk-Ports 1-1024 kann unter Linux nur von einem Prozess mit Root-Rechten geöffnet werden. Da ein Webserver üblicherweise auf Port 80 betrieben wird, ist es meist zwingend, den Apache-Prozess unter „root“ zu starten.

Glücklicherweise gibt es jedoch seit längerer Zeit einen vom Apache-Webserver direkt unterstützten Weg, seine Prozesse unter einem normalen User auszuführen und trotzdem auf Port 80 Anfragen zu bedienen. Der Trick besteht darin, dass der Webserver in jedem Fall als Root gestartet wird, woraufhin er sich an Port 80 binden kann. Anschließend spaltet dieser Master-Prozess die Kind-Prozesse ab, welche die eigentliche Anfragebearbeitung übernehmen - und diese Prozesse werden nun nicht mehr unter „root“, sondern unter dem gewünschten normalen User ausgeführt.

Die Einrichtung dieser Funktion ist denkbar einfach: im Apache-Konfigurationsfile gibt es die beiden Direktiven „User“ und „Group“, über welche der gewünschte Username sowie die Gruppenzugehörigkeit festgelegt

werden kann. Der dort eingetragene User muss selbstverständlich zuvor im System erstellt werden.

### 3.2.1.2 Ausführung eines Tomcat-Containers unter funktionalem User

Bei einem Tomcat-Applikationsserver ist das Vorgehen ähnlich wie bei der Kombination Apache + PHP: Da nur privilegierte Prozesse die Ports 1-1024 nutzen dürfen, wird der Tomcat-Server mit Root-Rechten gestartet, initialisiert sich und schaltet anschließend auf einen nicht-privilegierten User um.

Dies erledigt normalerweise das kleine Tool „jsvc“ aus dem Apache-Commons-Daemon-Projekt. Es wird von vielen Paketmanagern gleich bei der Installation des Tomcat-Pakets mitinstalliert und als normale Startmethode für den Applikationsserver genutzt, so dass es in den meisten Fällen genügt, den Kommandozeilenparameter „-user“ von jsvc im Tomcat-Startscript einzutragen, um einen Wechsel auf einen zuvor angelegten, unprivilegierten User nach dem Start anzuordnen. Manche Tomcat-Installationspakete sind sogar schon so konfiguriert, dass ein spezieller Tomcat-User angelegt und als User für den Tomcat-Prozess eingetragen wird.

Ist dies nicht der Fall, oder wurde Tomcat aus den Originaldateien der Apache Foundation installiert, so muss jsvc vor der Nutzung erst einmal kompiliert werden. Der Sourcecode ist jedoch im normalen Tomcat-Binary-Paket bereits enthalten, entsprechende Anweisungen zur Kompilation können in der Tomcat-Dokumentation<sup>4</sup> gefunden werden.

Im Falle des Tomcat bleibt noch anzumerken, dass auch dieser - trotz seiner Fähigkeit, direkt HTTP-Requests zu beantworten - sehr oft mit einem Apache-Webserver als Frontend-Server betrieben wird, so dass der Apache-Webserver die HTTP-Requests bearbeitet und bei Bedarf an den Application Server über ein in der Regel HTTP-fremdes Protokoll (etwa AJP) weiterreicht. Kommt eine solche Konfiguration zum Einsatz, ist es natürlich unerlässlich, auch den Apache-Frontend-Server unter einem nichtprivilegierten User-Account auszuführen.

### 3.2.1.3 Mögliche Problemstellungen bei Ausführung ohne Root-Privilegien

Ein häufig auftretendes Problem betrifft Dateizugriffsrechte. Während es, so lange der Web/Application-Server mit Root-Rechten ausgeführt wird, egal ist, wie die Besitzrechte der Dateien im Verzeichnis der Webapplikation gesetzt sind, spielt dies bei Ausführung des Servers mit normalen User-Rechten plötzlich eine Rolle: der normale User ist nicht berechtigt, jede beliebige Datei zu lesen (und zu schreiben), sondern kann lediglich seine eigenen Dateien, die Dateien von Usern in seiner Gruppe mit entsprechend gesetzten Gruppenrechten und für alle User lesbare Dateien lesen. Die Dateien der Webapplikation müssen folglich entsprechend gesetzte Lese- und wenn nötig auch Schreibrechte aufweisen.

Während man Leserechte für sämtliche Dateien einräumen sollte, kann der Umgang mit Schreibrechten sparsamer gestaltet werden. Schreibrechte erhält der Webapplikationsserver optimalerweise nur auf Dateien und Verzeichnisse, die er wirklich schreiben muss. Daher ist es auch nicht empfehlenswert, die Dateien generell demselben Nutzer zuzuordnen, unter welchem der Serverprozess betrieben wird - auf diese Dateien hat der Besitzer entweder ohnehin Schreibzugriff oder kann ihn sich per Änderung der Dateizugriffsrechte auf einfachste Weise besorgen. Ein gangbarer Weg wäre es daher z.B., die Gruppenrechte zu nutzen: man verwalte die Dateien der Webapplikation unter einem separaten User-Account, der jedoch mit dem Server-Account einer Gruppe angehört, so dass diesem über die Gruppenrechte gezielt Leserechte für alle Dateien und Schreibrechte, wo erforderlich, eingeräumt werden können.

Zugriffsrechte können auch bei der Verbindung zu anderen Diensten, etwa einem Datenbankserver, Probleme bereiten. Befindet sich dieser Dienst auf demselben Rechner wie der Application Server, können in manchen

<sup>4</sup><http://tomcat.apache.org/tomcat-5.5-doc/setup.html>

Fällen Unix-Sockets bzw. Named Pipes genutzt werden, um die Verbindung ressourcensparender und schneller als über das Netzwerk-Interface herzustellen. Da Named Pipes unter Linux in Form normaler „Dateien“ ins Dateisystem eingehängt werden, kommt auch beim Zugriff auf sie die Rechteverwaltung zum Einsatz und kann unter Umständen den Zugriff verweigern. Dieses Problem lässt sich jedoch durch eine entsprechende Rechtevergabe leicht lösen.

Ein weiteres, eventuell wichtiges Detail: Wird der Applikationsserver-Prozess unter einem normalen User ausgeführt, gehören auch alle von ihm neu erstellten Dateien und Verzeichnisse automatisch diesem User. Dieser Umstand kann Probleme bereiten, falls auf solche Dateien von anderer Stelle, etwa einem anderen Prozess auf dem Server oder manuell per FTP, SCP o.ä. zugegriffen werden muss. Auch hier hilft im Falle eines Falles ein bewusstes Bearbeiten der Zugriffsrechte weiter.

Allgemein ist jedoch zu sagen, dass die Vorteile der Ausführung des Serverprozesses unter einem normalen User-Account die Nachteile weit überwiegen, zumal die zu erwartenden Problemlösungen wie beschrieben durch bewussten Umgang mit Zugriffsrechten in den Griff zu bekommen sind. Aus diesem Grund ist eine solche Installation mittlerweile sogar bei einigen Paketmanagement-Systemen der Standard: bei der Serverinstallation wird nicht selten bereits ein passender normaler User-Account erzeugt, unter welchem der Serverprozess später ausgeführt wird.

### **3.2.1.4 Wovor funktionale User-Accounts schützen - und wovor nicht**

Die Verwendung normaler User-Accounts ist bei weitem kein Allheilmittel, weshalb es wichtig ist, klar zu erkennen, wovor eine solche Serverinstallation schützt und wovor nicht.

Zunächst das Positive: Wenn man von der Annahme ausgeht, ein Angreifer habe eine Sicherheitslücke in einer Webapplikation gefunden, die ihm höchstmögliche Freiheit auf dem kompromittierten Server bzw. im kompromittierten Serverprozess bringt - z.B. weil er es geschafft hat, eine Kommando-Shell zu öffnen - dann ist er im Falle eines normalen Users zunächst in einer deutlich schlechteren Ausgangsposition wie bei einem Server mit Root-Rechten. Er kann sich nur als normaler Nutzer auf dem System bewegen, kann keine kritischen Systemeinstellungen ändern, keine anderen laufenden Prozesse direkt schädigen. Für vollen Zugriff müsste er nun erst eine weitere Schwachstelle an anderer Stelle (etwa im Kernel oder in einem mit Root-Rechten laufenden Programm) finden und ausnutzen.

Für manche Dinge benötigt ein Angreifer jedoch gar keine Root-Rechte: Spam-Versand ist in der Regel auch ohne Root-Rechte möglich, der Zugriff auf `/etc/passwd` ist möglich (auf `/etc/shadow` glücklicherweise nicht!) und allgemein stehen ihm sämtliche Dateien offen, die für alle Nutzer lesbar auf dem System abgelegt wurden. Er kann auch Programme ausführen - zwar nicht mit Root-Rechten, doch aber mit normalen Nutzerrechten, was aber meist genügt, um etwa durch Erzeugen extrem vieler Prozesse den Server lahmzulegen. Auch kann er zumeist an irgendeiner Stelle im System schreiben, womit auch die Umfunktionierung der Maschine zum FTP-Server zur Verteilung urheberrechtlich geschützter Inhalte im Rahmen des Möglichen liegt. Als Basis für Angriffe auf weitere Rechner genügt ein Server mit Zugriff unter normalen User-Rechten in der Regel ebenfalls.

### **3.2.2 Abschottung durch chroot-Jail**

Da die Ausführung eines Serverprozesses mit User-Rechten nicht vor einem Zugriff auf die vielen für jedermann lesbaren Dateien eines Linux-Systems schützt, ist es oft sinnvoll, diese Sicherheitsmaßnahme durch den Einsatz der sogenannten chroot-Jails zu verstärken. Dabei handelt es sich im wahrsten Sinnes des Wortes um „Gefängnisse“: Das Wurzelverzeichnis eines Prozesses wird gewissermaßen einige Verzeichnisebenen nach unten verlegt, wodurch der Prozess nur noch Verzeichnisse unterhalb des neuen Wurzelverzeichnisses erreichen kann - er ist darin sozusagen „eingesperrt“.

### 3.2.2.1 Ausführung von Programmen im chroot-Jail

Das Vorgehen, um Programme - egal welcher Art - innerhalb eines chroot-Jails „einzusperren“, ist grundsätzlich immer das Gleiche:

1. Kopieren der Programm-Executables sowie -Libraries ins Jail-Verzeichnis
2. Herausfinden, welche Bibliotheken das Programm im Betrieb dynamisch linkt
3. Hineinkopieren aller nötigen Bibliotheken ins Jail
4. Starten des Programms im Jail

Der erste Punkt ist in der Regel schnell erledigt; dass das Programm selbst im chroot-Verzeichnis liegen muss, um später innerhalb der eingeschränkten Umgebung ausgeführt zu werden, ist naheliegend. Allerdings nutzen so gut wie alle Programme unter Linux, selbst die einfachsten Systembefehle, dynamisch gelinkte Bibliotheken. Wird ein solches Programm nun innerhalb eines chroot-Jails ausgeführt, erwartet es die Bibliotheken selbstverständlich am gewohnten Platz - nur liegt dieser nun außerhalb des Jails und ist damit nicht erreichbar. Folglich müssen sämtliche genutzten Bibliotheken zusätzlich zum auszuführenden Programm selbst in das chroot-Jail kopiert werden.

Auch rufen manche Programme wiederum andere Programme auf; das können auch Systembefehle sein, etwa „ls“. Deren Executables müssen in diesem Fall auch ins chroot-Jail kopiert werden - gemeinsam mit allen von diesen wiederum genutzten Bibliotheken.

Welche Bibliotheken ein Programm im Betrieb benötigt lässt sich unter Linux mit dem Kommando „ldd“ herausfinden. Angewendet auf ein beliebiges Executable liefert es eine Liste aller benötigten Bibliotheken sowie den Platz, an welchem diese erwartet werden. Am Beispiel von Apache soll dies kurz demonstriert werden:

```
1 root@firehead:/usr# ldd /usr/sbin/apache2
2     libpcre.so.3 => /usr/lib/libpcre.so.3 (0xb7f9f000)
3     libaprutil-1.so.0 => /usr/lib/libaprutil-1.so.0 (0xb7f85000)
4     libapr-1.so.0 => /usr/lib/libapr-1.so.0 (0xb7f60000)
5     libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7f48000)
6     libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7df9000)
7     libldap_r-2.4.so.2 => /usr/lib/libldap_r-2.4.so.2 (0xb7db9000)
8     liblber-2.4.so.2 => /usr/lib/liblber-2.4.so.2 (0xb7dac000)
9     libdb-4.6.so => /usr/lib/libdb-4.6.so (0xb7c84000)
10    libpq.so.5 => /usr/lib/libpq.so.5 (0xb7c65000)
11    libsqlite3.so.0 => /usr/lib/libsqlite3.so.0 (0xb7c00000)
12    libexpat.so.1 => /usr/lib/libexpat.so.1 (0xb7bdf000)
13    libuuid.so.1 => /lib/libuuid.so.1 (0xb7bdb000)
14    librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7bd2000)
15    libcrypt.so.1 => /lib/tls/i686/cmov/libcrypt.so.1 (0xb7b9f000)
16    libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7b9b000)
17    libresolv.so.2 => /lib/tls/i686/cmov/libresolv.so.2 (0xb7b88000)
18    libsasl2.so.2 => /usr/lib/libsasl2.so.2 (0xb7b71000)
19    libgnutls.so.13 => /usr/lib/libgnutls.so.13 (0xb7afb000)
20    libssl.so.0.9.8 => /usr/lib/i686/cmov/libssl.so.0.9.8 (0xb7ab9000)
21    libcrypto.so.0.9.8 => /usr/lib/i686/cmov/libcrypto.so.0.9.8 (0xb7976000)
22    libkrb5.so.3 => /usr/lib/libkrb5.so.3 (0xb78e9000)
23    libcom_err.so.2 => /lib/libcom_err.so.2 (0xb78e6000)
24    libgssapi_krb5.so.2 => /usr/lib/libgssapi_krb5.so.2 (0xb78bd000)
25    libtasn1.so.3 => /usr/lib/libtasn1.so.3 (0xb78ad000)
26    libz.so.1 => /usr/lib/libz.so.1 (0xb7897000)
27    libgcrypt.so.11 => /lib/libgcrypt.so.11 (0xb784a000)
28    libk5crypto.so.3 => /usr/lib/libk5crypto.so.3 (0xb7827000)
29    libkrb5support.so.0 => /usr/lib/libkrb5support.so.0 (0xb781f000)
```

```
30 libkeyutils.so.1 => /lib/libkeyutils.so.1 (0xb781c000)
31 libpgp-error.so.0 => /lib/libpgp-error.so.0 (0xb7817000)
```

Sämtliche aufgelisteten Bibliotheken müssen anschließend in das Jail hineinkopiert werden. Auch ist es oft erforderlich, dass bestimmte Verzeichnisse existieren - beispielsweise, weil ein Programm dort temporäre Dateien anlegen möchte. Dies ist jedoch von Programm zu Programm unterschiedlich, und leider existiert kein generell funktionierender Weg, solche benötigten Verzeichnisse zu ermitteln. Hier hilft „Trial and error“ am effektivsten weiter.

Das Starten des Programms im chroot-Jail ist dann sehr einfach, insbesondere im Falle eines Dienstes, der ohnehin in der Regel per init-Script gestartet wird. Grundsätzlich wechselt man zunächst per „chroot“-Kommando ins Jail hinein und führt dann das gewünschte Programm aus. Den Wechsel ins Jail kann man nun beispielsweise auch in ein init-Script vor die eigentliche Ausführung des Programms eintragen.

### 3.2.2.2 Die Grenzen und Probleme eines chroot-Jails

Die Wirksamkeit dieser Methode beruht darauf, dass nur der Systemuser „root“ den „chroot“-Befehl, der das Wurzelverzeichnis neu setzt, ausführen darf, weshalb ein unter normalen Rechten laufendes Programm diese Einschränkung seines Aktionsraumes nicht rückgängig machen kann. Grundsätzlich können daher chroot-Jails durchaus als Sicherheitshürde eingesetzt werden, wenngleich sie unter Linux nicht offiziell als Sicherheitsfeature gelten. Ein Angreifer jedoch, der einen Weg findet, die Hürde des normalen User-Accounts zu nehmen - sich also Root-Rechte verschafft hat - kann automatisch auch aus einem chroot-Jail ausbrechen. Hier gibt es allerdings wiederum Methoden, dies zu erschweren bzw. gänzlich unmöglich zu machen: sogenannte „gehärtete Kernel“<sup>5</sup> implementieren z.B. Änderungen am chroot-Syscall, die es generell unmöglich machen sollen, mit seiner Hilfe aus einem chroot-Jail auszubrechen - auch mit Root-Rechten.

Auch kann ein chroot-Jail nicht davor schützen, dass ein Angreifer innerhalb des „Käfigs“ sein Unwesen treibt und Programme herunterlädt und/oder startet. Für manche Nutzungsmöglichkeiten eines gehackten Servers ist es daher auch hier gar nicht erst erforderlich, die Sicherheitshürde zu überwinden.

Daher und aufgrund des doch je nach „eingesperrtem“ Programm recht hohen Installations- und Wartungsaufwands (man darf keinesfalls vergessen, dass das Paketmanagement einer Distribution weder von den Kopien der Bibliotheken noch von der Kopie des Programms selbst etwas weiß - bei Updates werden diese Versionen also nicht angerührt und müssen manuell aktualisiert werden) muss der Einsatz von chroot-Jails als „zusätzliche Hürde“ in jedem Einzelfall gut überdacht werden.

### 3.2.3 „Härten“ von Applikationsservern

Neben den erwähnten Techniken zur Abschottung, die grundsätzlich auf jeden Applikationsserver angewendet werden können, gibt es für manche Server noch sogenannte Hardening-Patches, also Zusatzmodule oder Sourcecode-Veränderungen zum „Abhärten“ der Standard-Releases gegen Angriffe. Diese Zusätze werden meist nicht von den Original-Entwicklern der Serversoftware, sondern von Drittherstellern gepflegt, haben in manchen Fällen auch negative Auswirkungen auf die Performance der Software, erhöhen aber gemeinhin die Sicherheit gegen einige Angriffsformen signifikant - wenngleich man in jedem Einzelfall genauestens prüfen sollte, ob die Anwendung eines solchen Patches Sinn macht und der Sicherheitsgewinn den Aufwand wert ist.

Da es für den Tomcat-Applikationsserver keine bekannten Hardening-Patches gibt, wird im Folgenden der sehr bekannte und erprobte PHP-Hardening-Patch „Suhosin“ näher vorgestellt.

<sup>5</sup>Ein veränderter Kernel, welcher durch zahlreiche spezielle Sicherheitspatches widerstandsfähiger gegen Angriffe ist; siehe z.B. <http://www.grsecurity.net>

### 3.2.3.1 „Abhärten“ von PHP mit Suhosin

Suhosin besteht aus zwei Komponenten, die sowohl getrennt als auch gemeinsam eingesetzt werden können.

- Ein **PHP-Core-Patch** verändert Code des PHP-Kerns selbst und implementiert Schutzmechanismen gegen Buffer Overflows im PHP-Memory-Manager sowie einige weitere Low-Level-Verwundbarkeiten
- Eine **PHP-Extension**, welche den größeren Teil der Schutzmechanismen gegen unsauber programmierten PHP-Code bzw. dadurch geöffnete Sicherheitslücken implementiert

Um den Code-Patch anwenden zu können, muss PHP aus dem Sourcecode kompiliert werden (oder ein Paket im Paketmanagement vorliegen, in welchem dieser Patch bereits angewendet wurde!), während die PHP-Extension als normale Erweiterung in einer Standard-PHP-Binary geladen werden kann. Daher ist es von großem Vorteil, dass beide Komponenten unabhängig voneinander genutzt werden können - und dass der Großteil der Schutzmechanismen bereits in der leichter zu installierenden Extension untergebracht sind.

Um einen Überblick über alle von Suhosin implementierten Verbesserungen zu bekommen, sei die Suhosin-Feature-Liste<sup>6</sup> empfohlen. Im Folgenden werden einige Punkte aus der Liste etwas näher beleuchtet, um einen Eindruck zu geben, was unter „Hardening“ im Fall von Suhosin zu verstehen ist.

**Schutz gegen Remote-Inclusion-Attacken:** Die Remote Inclusion gehört wohl zu den am häufigsten angewendeten Angriffen auf PHP-Websites. Suhosin sperrt sämtliche Versuche, PHP-Code von entfernten Webservern zu inkludieren. Darüberhinaus verhindert Suhosin auch ein Inkludieren von durch eine Webapplikation hochgeladenen Dateien.

**Transparente Cookie-Verschlüsselung:** Suhosin verschlüsselt Cookies transparent, d.h. für die PHP-Anwendung (und den User) unsichtbar. Dabei können unterschiedliche Komponenten in beliebiger Kombination als Schlüssel zum Einsatz kommen, um z.B. Cookies automatisch zu invalidieren, wenn sie von einem anderen Client-Browser kommen. Ein „geheimer“ Teil, den nur der Serverbetreiber kennt, verhindert, dass Clients die Cookies lesen oder bearbeiten können.

**Transparente Session-Verschlüsselung:** Auch serverseitig gespeicherte Sessiondaten kann Suhosin verschlüsseln, so dass diese möglicherweise sensiblen Daten nicht ungeschützt auf der Festplatte des Servers liegen. Dadurch, dass der Schlüssel aus frei definierbaren Teilen sowie auch aus dem User-Agent-String des anfragenden Browsers und beliebig vielen Bytes der Client-IP bestehen kann, ist auf diesem Weg auch ein deutliches Erschweren von Session-Hijacking-Angriffen möglich.

**Schutz von phpinfo-Seiten vor Indexierung:** Ein beliebter Weg, potentielle neue Angriffsziele zu finden, führt über die Nutzung gängiger Suchmaschinen: indem man beispielsweise gezielt nach festen Begriffkombinationen auf phpinfo-Seiten sucht, stößt man sehr schnell auf zahlreiche Server, auf denen diese zur Vorbereitung eines Angriffs äußerst informativen Diagnoseseiten offen zugänglich sind. Von dort aus kann z.B. gezielt nach PHP-Versionen mit bekannten Lücken gefahndet werden. Suhosin fügt der phpinfo-Seite ein Meta-Tag hinzu, welches den Suchmaschinen-Robots verbietet, die Info-Seite in ihren Index aufzunehmen, so dass versehentlich offen erreichbare Info-Seiten wenigstens nicht auch noch per Suchmaschine auffindbar sind.

**Schutz gegen Newline-Angriffe auf mail():** PHP stellt eine sehr rudimentäre Funktion zum Versenden von Mail zur Verfügung, die z.B. in vielen Formularen auf Webseiten zum Einsatz kommt, um etwa dem Sitebetreiber eine Nachricht zukommen zu lassen. Das Problem hierbei besteht darin, dass wichtige Parameter der Mail, wie z.B. die Absenderadresse, direkt in den Mail-Header geschrieben und dieser anschließend der mail()-Funktion übergeben werden muss. Wird hier einfach String-Konkatenation genutzt, kann ein Angreifer zusätzliche Headerzeilen durch Einfügen von newline-Zeichen in Formularfelder anhängen und etwa Bcc-Adressen hinzufügen, woraufhin die Mail plötzlich an weitere Empfänger geleitet wird. Da der Mail-Body

<sup>6</sup>[http://www.hardened-php.net/suhosin/a\\_feature\\_list.html](http://www.hardened-php.net/suhosin/a_feature_list.html)

vom Header nur durch zwei Newline-Zeichen getrennt wird, kann er auch diesen auf dieselbe Weise frei festlegen. So hat ein Angreifer schnell ein Gateway zum Versand beliebiger Mails an beliebige Empfänger in der Hand. Hier greift Suhosin ein und filtert hintereinanderfolgende Newline-Zeichen aus dem Mail-Header aus, so dass der Text der Mail nicht mehr unsinnigerweise über den Header-Parameter von mail() übergeben werden kann.

Wie man sehen kann, fügen sich die Suhosin-Veränderungen in der Regel vollständig transparent in die PHP-Umgebung ein, d.h. installierte Anwendungen müssen nicht verändert werden und profitieren dennoch vom verbesserten Schutz. Für den Fall, dass eine aufwendigere Anwendung aufgrund einer Suhosin-Installation plötzlich den Dienst verweigert, bietet Suhosin einen „Simulations-Modus“ an, in welchem es Aufrufe von geschützten bzw. veränderten Funktionen loggt, aber keinerlei Filter- oder Schutzfunktionen ausführt. Dieser Modus soll dabei helfen, die problematischen Stellen zu finden und die Anwendung gegebenenfalls abzuändern, denn meist sind Funktionalitäten, die von Suhosin plötzlich unterbunden werden, ohnehin als sicherheitstechnisch bedenklich bis kritisch einzustufen.

## 4 Weitere Absicherungen

Abgesehen vom Applikationsserverprozess selbst gibt es oft noch eine ganze Reihe weiterer, möglicherweise kritischer Stellen und Anwendungen auf einem Server - insbesondere bei für kleinere Projekte gemieteten Servern, auf denen häufig auch gleich noch Datenbankserver u.ä. betrieben wird. Einige typische Fallstricke sollen im Folgenden beleuchtet werden.

### 4.1 Datenbankserver absichern

Egal, welcher Datenbankserver - MySQL, PostgreSQL, etc. - letztendlich eingesetzt wird, die Maßnahmen zu seiner Absicherung ähneln sich weitestgehend. Daher kann das hier gesagte prinzipiell auf jeden Datenbankservertyp angewendet werden.

Zunächst sollte der Datenbankserver - wenn irgendwie möglich - nur lokal, d.h. von anderen Prozessen auf dem Rechner selbst, erreichbar sein. Dies kann etwa durch ausschließliches Binden des Server-TCP-Ports auf die Loopback-IP-Adresse „127.0.0.1“ erfolgen. Falls der Server dies unterstützt, bietet sich auch ein kompletter Verzicht auf die Netzwerkschnittstelle und eine Nutzung von Named Pipes zur Verbindung des Applikationsservers mit dem Datenbankserver an, was neben der höheren Sicherheit auch den angenehmen Nebeneffekt von besserer Performance gegenüber dem Weg über das Netzwerk-Interface hat.

Der Sinn und Zweck dieser Einschränkung: Eventuell auftretende Sicherheitslücken im Datenbankserver, etwa in der Authentifizierungsroutine, werden auf diese Weise für Angreifer praktisch wertlos, da sie ohne vollen Zugriff auf den Server selbst gar keine Verbindung zum Datenbankserver aufnehmen können. Die Hürde für einen Angriff auf die Datenbank wird auf diese Weise also ein gutes Stück nach oben verlegt.

Weiterhin ist auch das Konzept der Isolation im Bereich der Datenbanken wichtig: laufen auf einem Applikationsserver mehrere voneinander unabhängige Applikationen (etwa eine Hauptanwendung sowie ein zugehöriges Forum und ein Bugtracker), dann sollten deren Datenbanken so weit wie möglich voneinander getrennt werden. Nutzen die Anwendungen unterschiedliche Datenbankserver, wäre dies automatisch gegeben; im Falle desselben Datenbankservers sollten mindestens separate User-Accounts für jede Anwendung eingerichtet und deren Zugriff auf die zu einer Applikation gehörenden Datenbanken beschränkt werden. Auf diese Weise ist weitgehend ausgeschlossen, dass eine verwundbare Anwendung zum Zugriff auf die Daten einer anderen, eigentlich sicheren Anwendung genutzt wird.

## 4.2 FTP-Server absichern

Häufig läuft auf einem Applikationsserver auch ein FTP-Server, um den Fernzugriff auf Dateien der Applikation zu ermöglichen. Das ist grundsätzlich - sofern auf sichere Konfiguration des FTP-Serverprozesses Wert gelegt wurde - kein Problem, jedoch sollte man vor Einsatz eines FTP-Servers für Datenübertragungszwecke evaluieren, ob die Nutzung des SCP<sup>7</sup>-Protokolls nicht eine bessere Alternative darstellen könnte. SCP hat gegenüber FTP mehrere Vorteile:

- Die Daten und Login-Credentials werden stets verschlüsselt übertragen, während dies bei FTP standardmäßig nicht der Fall ist (erst FTP over SSL verschlüsselt Steuer- und Nutzdaten)
- SCP nutzt als Übertragungskanal eine SSH-Session, die Authentifizierung erfolgt daher auf dieselbe Weise wie beim Remote Login über SSH, und es wird kein weiteres, potentiell Sicherheitslücken enthaltendes, Authentifizierungsverfahren benötigt
- SCP ist auf Linux-Maschinen mit installiertem SSH-Server in der Regel ohne weitere Installation von Software verfügbar

Doch wo Licht ist, gibt es auch Schatten: der Hauptnachteil von SCP ist seine im Vergleich zu SSH geringere Geschwindigkeit und höhere Prozessorauslastung, beides direkte Folgen der verschlüsselten Datenübertragung. SCP eignet sich daher je nach Hardwareausstattung von Server und Client evtl. nicht, um sehr große Datenmengen mit hoher Geschwindigkeit zu übertragen.

Soll ganz bewusst ein FTP-Server eingerichtet werden, so muss auch hierbei auf eine sichere Konfiguration Wert gelegt werden. Dies beginnt damit, dass anonymen Zugriff - ein in der FTP-Welt häufig genutztes Feature, welches einen Login in einen speziellen „anonymous“-Account ohne Angabe eines bestimmten Passworts ermöglicht - unbedingt abgeschaltet werden sollte, sofern ein solcher Zugang nicht zwingend erforderlich ist - auf typischen Applikationsservern gibt es jedoch selten einen Grund dafür.

Bestimmte lokale Nutzer sollen sich aber per FTP einloggen können, um die ihnen zugehörigen Dateien - etwa die Dateien der Webapplikation - zu verwalten. Es empfiehlt sich also, ganz gezielt diesen Nutzern einen FTP-Login zu erlauben; die meisten FTP-Server können an dieser Stelle so konfiguriert werden, dass nur bestimmte lokale Nutzer sich mit ihrem normalen Nutzer-Passwort einloggen können. „Root“ sollte allerdings auf gar keinen Fall zu diesen Nutzern gehören!

Desweiteren besteht bei den meisten FTP-Serverdiensten die Möglichkeit, User per chroot-Syscall in ihrem Home-Verzeichnis „einzusperren“ - das funktioniert ganz ähnlich wie die in Kapitel 3.2.2 vorgestellte Methode, ganze Serverprozesse in chroot-Jails einzusperren. Während die Nutzung von chroot-Jails bei Applikationsservern jedoch nicht uneingeschränkt empfehlenswert ist, gilt für chroot-Jails bei FTP-Servern: unbedingt nutzen! Dieses Feature verhindert effektiv, dass ein FTP-User sich frei im Dateisystem bewegen und auf Dateien zugreifen kann, die für alle Nutzer lesbar sind. Damit dies allerdings praktikabel ist, muss das Home-Verzeichnis des Users korrekt gesetzt sein - nur, was sich unterhalb dieses Verzeichnisses befindet, wird per FTP erreichbar sind.

Zu guter Letzt empfiehlt es sich, Logging für FTP-Server-Logins sowie Dateitransfers einzuschalten und die Logs an einer für keinen FTP-User zugänglichen Stelle aufzubewahren. Im Falle eines Falles, d.h. sollte sich wirklich ein fremder Nutzer per FTP Zugang zum Server verschaffen, kann mit den Logfiles evtl. wenigstens der Einstiegsweg des Angreifers nachvollzogen werden.

---

<sup>7</sup>Secure Copy Protocol - ein Protokoll zur verschlüsselten Übertragung von Dateien

### 4.3 SSH-Server absichern

Auf so gut wie jedem Linux-Server läuft ein SSH-Daemon zur Fernadministration über eine sichere, verschlüsselte Verbindung. Zwar gibt es noch alternative Möglichkeiten, die Fernadministration durchzuführen, etwa per Telnet, jedoch sollte dem SSH-Protokoll in aller Regel der Vorzug gewährt werden: das Protokoll selbst und die eingesetzten Verschlüsselungs- und Schlüsselaustauschverfahren gelten allgemein als sicher (wenngleich auch hier dasselbe gilt wie für jede andere Server-Software auch: immer auf aktuellem Patchstand und über neu gefundene Sicherheitslücken informiert bleiben! Auch mit SSH bzw. bestimmten SSH-Implementierungen gab es in der Vergangenheit schon Sicherheitsprobleme).

Die Konfiguration des SSH-Daemons selbst offeriert nur wenige Ansatzpunkte, um Absicherung zu betreiben. Der Wichtigste betrifft die Frage, welche dem System bekannten User sich per SSH einloggen dürfen: in der Regel ist ein SSH-Login nur für sehr wenige User erforderlich, möglicherweise sogar nur für den Root-User (der wiederum per „su“-Befehl nach dem Login jeden beliebigen User impersonifizieren kann). Funktionale User, etwa für den Web- oder FTP-Server angelegt, benötigen in der Regel keine Login-Möglichkeit. Daher empfiehlt es sich, die SSH-Daemon-Konfiguration so anzupassen, dass nur die gezielt dafür „freigeschalteten“ User per SSH einen Login durchführen können. Wie das genau geht, ist vom eingesetzten SSH-Daemon abhängig; der populäre OpenSSH-Daemon beispielsweise akzeptiert zu diesem Zweck in seiner Konfigurationsdatei eine Direktive „AllowUsers“, gefolgt von den Usernamen mit Login-Erlaubnis.

Eines näheren Blickes ist aber auch die Art der Authentifizierung beim Server würdig. Die gängigen SSH-Daemons bieten hier sowohl die klassische passwortgestützte Authentifizierung als auch eine Authentifizierung über Keyfiles an. Authentifizierung via Passwort ist jedermann vertraut, die Authentifizierung via Keyfile ist unbekannter, aber - wenn richtig genutzt - noch sicherer. Das Konzept beruht darauf, dass als zur Authentifizierung erforderliches „Geheimnis“ nicht ein geheimes Passwort, sondern ein binäres Datenwort mit einer Länge gleich der verwendeten Schlüssellänge, gespeichert meist in einer Datei (dem sogenannten Keyfile), genutzt wird. Ein zu diesem geheimen Keyfile mit dem Private Key passender Public Key wird auf dem Server hinterlegt. Durch die Anwendung asymmetrischer Kryptographie kann der SSH-Client beim Login dem Server beweisen, dass er im Besitz des Private Key zu einem auf dem Server hinterlegten Public Key ist - ohne, dass der Private Key selbst dazu übertragen werden muss.

Die Stärke des Keyfile-Authentifizierungsverfahrens ergibt sich aus der nahezu perfekten Zufallsverteilung der möglichen Schlüssel über den gesamten Schlüsselraum und der maximalen Ausnutzung des Schlüsselraums: anders als bei einem Passwort, bei welchem bestimmte Kombinationen wahrscheinlicher sind als andere und welches selten lang genug ist, um den gesamten zur Verfügung stehenden Schlüsselraum wirklich auszunutzen, sind bei einem sorgfältig - d.h. mithilfe eines guten Zufallsgenerators - erzeugten Keyfile Schwachstellen dieser Art von vorneherein ausgeschlossen. Allerdings kann der Speicherort des Keyfiles eine große Schwachstelle darstellen: da man sich den Inhalt eines Keyfiles normalerweise nicht merken kann, muss es irgendwo gespeichert (und darüberhinaus gegen Datenverlust gesichert) werden. Der Sicherheit des gewählten Speicherorts vor unerwünschtem Zugriff kommt also entscheidende Bedeutung zu. Optional kann jedoch ein Keyfile auch noch einmal mit einem Passwort abgesichert werden, um eine besonders starke Zwei-Faktor-Authentifizierung zu realisieren: sowohl Kenntnis des Passwort als auch Besitz des Keyfiles sind dann zum erfolgreichen Login erforderlich.

### 4.4 Sinn und Unsinn von Firewalls

Firewalls gehören wohl zu den am häufigsten missverstandenen Sicherheitsmaßnahmen in der Netzwerk- und Servertechnik. Die Missverständnisse beginnen dabei schon beim Begriff „Firewall“ selbst - denn was ist eine Firewall? Klassischerweise ist eine Firewall ein Paketfilter, also ein Filterungssystem, welches Pakete nach bestimmten Regeln entweder passieren lässt oder ausfiltert. In jüngster Zeit ist der Begriff „Firewall“ jedoch sehr weit gedehnt worden und bezeichnet mittlerweile vom simplen Paketfilter bis hin zum Intrusion-Detection-System eine sehr breite Softwarepalette. Je nachdem, welche Sorte „Firewall“ gemeint ist, unterscheiden sich

die Argumente für und wider einem Einsatz auf einem Linux-Server.

#### 4.4.1 Die Firewall als klassischer Paketfilter

Dieser Anwendungsbereich einer Firewall entspricht ganz ihrem ursprünglichen Sinn und Zweck: Pakete anhand vorher definierter Regeln als „erwünscht“ oder „nicht erwünscht“ zu klassifizieren und anschließend zu verwerfen oder weiterzuleiten. Unter Linux gibt es hierfür zwei Softwarepakete: ipchains (Kernel 2.2.x) und iptables (ab Kernel 2.4.x), die sich in der Funktionsweise etwas unterscheiden, aber ähnliches bezwecken.

Eingesetzt werden kann ein Paketfilter zum Einen natürlich als separates, zwischen Netzwerke geschaltetes Element zur Flusskontrolle. Auf diese Weise kann detailliert bestimmt werden, welche Art von Paketen für welche Dienste zwischen den beiden Netzwerken ausgetauscht werden dürfen. Im Falle von Angriffen kann über die Firewall eine Sperrung bestimmter IPs oder ganzer IP-Ranges vorgenommen werden, bevor die Pakete der Angreifer überhaupt den Zielsever erreichen.

In der Regel ist es bei günstigen dedizierten Servern aber nicht möglich, eigene Subnetze im Rechenzentrum zu bilden und diese über eine Firewall ans Netz des Hosters und damit ans Internet anzuschließen - stattdessen hängen die Server direkt am Netz. In diesen Fällen bleibt nur die Option, einen Paketfilter auf dem Server selbst einzusetzen. Auch dies kann man tun, wenn man einen Grund dafür hat. Möglich wäre etwa der Wunsch nach Sperrung bestimmter Ports außer für einzelne, vorher festgelegte IPs oder IP-Ranges (z.B. Sperrung des absichtlich „von außen“ erreichbaren MySQL-Ports für alle außer einer einzigen IP, über welche man Administrationaufgaben erledigen möchte).

Allerdings gilt auch bei einem Paketfilter auf dem Server: jegliche zusätzlich aktive Software ist ein Sicherheitsrisiko - auch der Paketfilter ist da keine Ausnahme! Paketfilter sind ebenfalls nicht gefeit vor Sicherheitslücken, die es Angreifern ermöglichen könnten, in das System einzudringen. So kann sich die vermeintliche „Sicherheitsmaßnahme“ schnell zu einem Sicherheitsrisiko entwickeln. Daher gilt auch und insbesondere bei Host-basierten Paketfiltern: wenn man keinen guten Grund hat, sie einzusetzen, sollte man auf sie verzichten. Einen Paketfilter einzusetzen nur weil „eine Firewall ja unbedingt nötig ist und den Server sicherer macht“ ist Unsinn und kann im Einzelfall die Sicherheit schwächen, statt sie zu erhöhen.

#### 4.4.2 Die Firewall als Intrusion-Detection-System

Während Firewalls klassischerweise nur als Paketfilter verstanden werden, wird dieser Begriff in den letzten Jahren zunehmend in breiterem Zusammenhang verwendet: einerseits für „Personal Firewalls“ auf Desktop-Systemen, die einzelnen Applikationen Netzzugang erlauben oder verweigern, E-Mails auf Virenbefall scannen und viele Dinge mehr tun, die mit dem Aufgabengebiet einer Firewall nicht mehr viel zu tun haben, und andererseits für Intrusion-Detection-Systeme auf Servern oder in Netzwerken.

Der Zweck dieser Systeme ist es, Angriffsversuche noch während der Angriff läuft zu entdecken, eine Alarmmeldung an den zuständigen Administrator abzusetzen und in manchen Fällen auch gleich Gegenmaßnahmen wie etwa eine Sperrung der fraglichen IP einzuleiten. Auf dem Markt gibt es eine Vielzahl kommerzieller IDS-Lösungen, die jedoch auf Unternehmensnetzwerke abzielen und sich daher preislich weit außerhalb des für die Absicherung eines einzelnen dedizierten Servers sinnvollen Rahmens bewegen. Für diesen Zweck haben sich stattdessen einige Open-Source-Produkte etabliert, deren Funktionsweise sich sehr ähnelt: durch das Scannen diverser Logfiles werden sich in rascher Folge wiederholende fehlgeschlagene Loginversuche erkannt, die zugehörige IP-Adresse isoliert und weitere Pakete von dieser per iptables oder ipchains für eine gewisse Zeit ausgefiltert. Zu nennen wäre hierbei das Tool „fail2ban“<sup>8</sup> oder „BlockHosts“<sup>9</sup>.

---

<sup>8</sup><http://www.fail2ban.org>

<sup>9</sup><http://www.aczoom.com/cms/blockhosts/overview>

Jedoch stellt sich bei genauerem Hinsehen die Frage nach dem Nutzen solcher automatischer Sperren: Geht man davon aus, dass das Passwort zu einem Serverdienst (etwa SSH) sicher genug gewählt worden ist, dass es sich nicht in adäquater Zeit per Brute-Force überwinden lässt - oder hat man gar ein sicher verwahrtes Keyfile eingesetzt - schützt eine Sperre nach  $x$  Login-Versuchen in  $y$  Minuten höchstens den Angreifer vor weiteren sinnlosen Versuchen. Startet ein Angreifer eine massive DDoS-Attacke auf einen Dienst, hilft in der Regel auch ein automatisiertes Sperren aller beteiligten IPs nicht viel, da die Pakete dann zwar vom Paketfilter aussortiert werden, bevor sie den Dienst erreichen, aber dennoch am System ankommen und Traffic sowie Rechenzeit beanspruchen. Im Falle einer Sicherheitslücke in einem Serverdienst verpufft eine automatische Sperre bei zu vielen Fehlversuchen in aller Regel auch wirkungslos: bei der Ausnutzung von Sicherheitslücken sind nur sehr selten Fehlversuche erforderlich, stattdessen wird die Authentifizierung gezielt umgangen.

Der Nutzen reduziert sich also praktisch darauf, dass die Logfiles bei einem Brute-Force-Angriff kleiner bleiben, weil der Angreifer nach einer gewissen Anzahl von Versuchen abgeblockt wird. Ob dieser sicherheitstechnisch weitgehend bedeutungslose Nutzen die Kosten des Einsatzes einer Host-IDS-Lösung überwiegen, darf bezweifelt werden, denn auch hier gilt: ein zusätzliches System schafft zusätzliche Angriffsfläche. Im Falle der Logfile-basierten automatischen Sperrmechanismen ist es beispielsweise in der Vergangenheit bereits dazu gekommen, dass diese durch geschickte Wahl des Usernamens beim Login-Versuch zur Sperre einer vom Angreifer beliebig wählbaren IP zu überlisten waren (siehe [3]) - kennt ein Angreifer die IP des Server-Administrators, könnte er ihn durch Nutzung solcher Sicherheitslücken gezielt von seinem eigenen Server aussperren.

## 4.5 Absicherung gegen physikalischen Serverzugriff

Alle bisher diskutierten Sicherungsmaßnahmen haben eines gemeinsam: sie helfen gegen Angreifer, die „von außen“, d.h. über das Internet, auf ihnen nicht zugängliche Daten zugreifen wollen, sind aber praktisch wirkungslos gegen einen Angreifer, der physischen Zugang zur Server-Hardware hat. Natürlich ist der Angriff über das Netz sehr viel wahrscheinlicher wie ein Angriff direkt auf die Server-Hardware - immerhin stehen gemietete Server in der Regel in großen, gesicherten Rechenzentren, zu denen nur Mitarbeiter des Hosters Zugang haben, während die Netzwerkschnittstelle „offen für jedermann“ ist. Unmöglich ist jedoch auch ein physischer Angriff nicht. Auch ist in der Regel wenig darüber bekannt, was ein Hostler mit noch funktionsfähigen, aber ausgemusterten Systemen macht: möglicherweise landen die Server-Festplatten auf diesem Weg in ungelöscht Zustand bei einem Käufer, der daraufhin vollen Zugang zu sämtlichen gespeicherten Daten des Servers hat, in welchem die Platte zuvor ihren Dienst verrichtete.

Je nach Kritikalität der gespeicherten Daten kann es also wünschenswert sein, diese gegen physischen Zugang zum Server gesondert abzusichern. Theoretisch kann der hierbei zu betreibende Aufwand ins Unermessliche steigen, denn der Zugang zur Server-Hardware eröffnet einem imaginären, mit unendlichen Mitteln ausgestatteten Angreifer zahlreiche neue Angriffswege bis hin zum „Schockfrosten“ der RAM-Bausteine und Auslesen des Inhalts. Glücklicherweise gibt es aber, wenn man sich mit einem gewissen Maß an Sicherheit zufrieden gibt, eine sehr einfache und erprobte Antwort auf die Frage nach Sicherheitsmaßnahmen gegen physischen Zugang: transparente Datenverschlüsselung.

Die Idee: kritische Daten werden, statt direkt in unverschlüsselter Form auf die Platte gespeichert zu werden, auf einer verschlüsselten Partition abgelegt. Beim Schreiben von Daten in diese Partition werden sie für die Anwendung transparent ver- und beim Lesen wieder entschlüsselt, so dass dieses Verfahren grundsätzlich mit jeder beliebigen Serversoftware funktioniert. Auch liefern heute gängige Prozessoren üblicherweise so viel Rechenleistung, dass transparente Verschlüsselung außer in performancekritischen Szenarien keine merkliche Schmälerung der Serverleistung verursacht und bedenkenlos eingesetzt werden kann.

Es gibt jedoch einen Knackpunkt: zum Mounten der verschlüsselten Partition muss ein Geheimnis irgendeiner Art - ein Keyfile oder ein Passwort - angegeben werden. Dieses Geheimnis auf dem Server in unverschlüsselter Form zu speichern würde die gesamte Verschlüsselung ad absurdum führen, da jedermann mit physischem

Zugang zur Rechnerhardware den „geheimen“ Schlüssel zur Entschlüsselung nutzen könnte. Das Geheimnis muss also bewusst vom Serverbetreiber erst beim Mounten der verschlüsselten Partition preisgegeben werden und darf anschließend nicht in persistenter Form auf dem Server gespeichert sein. Dies bedeutet praktisch: bei jedem Server-Neustart muss der Serverbetreiber die verschlüsselte Partition selbst unter Nutzung des geheimen Schlüssels mounten. Dadurch besteht keine direkte Gefahr mehr: das Geheimnis ist nicht auf dem Server dauerhaft abgelegt, nur im RAM gespeichert, wodurch Prozesse bis zum Shutdown ungehindert die verschlüsselten Daten lesen und schreiben können. Sollte irgendein Dritter versuchen wollen, etwa durch Ausbau der Festplatten oder Booten eines alternativen Betriebssystems von einem Wechseldatenträger an die verschlüsselten Daten zu gelangen, wird er spätestens in dem Moment, in dem er den Server abschaltet oder neu startet, den Zugang zu den verschlüsselten Informationen verschließen. Wichtig ist natürlich, dass ein Angreifer im Zweifelsfall auch zu einem Neustart bzw. Hardwareausbau gezwungen wäre: ein lokaler Login darf ihm nicht möglich sein.

Auch das Problem der ausgemusterten Festplatten ist auf diese Weise gelöst: ohne Mounten der verschlüsselten Partition unter Nutzung des geheimen Schlüssels kommt niemand an die zwar vorhandenen, aber unzugänglichen Daten auf der Festplatte heran. Nichtsdestotrotz ist natürlich zusätzliche Sicherheit immer besser: ein gründliches Löschen der Festplatten vor Beendigung eines Server-Mietvertrags schadet nie.

Wie weiter oben bereits angedeutet hat ein solch einfaches Schutzverfahren natürlich Grenzen: wenn ein Angreifer den RAM-Inhalt irgendwie im laufenden Betrieb auslesen kann, käme er an den erforderlichen Schlüssel. Auch schützt Verschlüsselung der Daten auf der Festplatte nicht gegen Angriffe über die Netzwerkschnittstelle. Vielen gängigen Angriffsmethoden auf physisch zugängliche Hardware kann aber ein Riegel vorgeschoben werden.

Die praktische Umsetzung einer verschlüsselten Partition unter Linux ist auf mehreren Wegen möglich: bekannt und vielfach im Einsatz sind vor allem die Multiplattform-Software „TrueCrypt“<sup>10</sup> sowie das Linux Unified Key Setup - kurz LUKS - in Kombination mit dm-crypt. Eine Beschreibung der genauen Vorgehensweise führt an dieser Stelle jedoch zu weit und sprengt den Rahmen dieses Papers. Nähere Informationen hierzu sind jedoch leicht im Internet zu finden; auch das Lesen der TrueCrypt-Online-Dokumentation unter <http://www.truecrypt.org> sowie der Man-Page zum zentralen LUKS-Utility „cryptsetup“ hilft dabei weiter.

## 5 Fazit

Mit größerer Macht kommt größere Verantwortung - so etwa lässt sich die Situation beschreiben, in die sich ein Mieter eines dedizierten Servers begibt. Er kann auf „seinem“ Rechner zwar schalten und walten, wie er will, muss sich aber auch Gedanken um die Sicherheit seines Systems machen, denn im Zweifelsfall ist stets er der Verantwortliche und muss damit rechnen, für Angriffe, die von seinem System ausgehen, haften zu müssen.

Während es bei selbst geschriebenen Anwendungen unentbehrlich ist, diese unter sicherheitstechnischen Gesichtspunkten zu entwickeln, hat auch die Konfiguration des als Ausführungsumgebung dienenden Applikationsservers sowie die Gesamtkonfiguration des Server-Systems einen großen Einfluss auf die Sicherheit der laufenden Anwendung: eine hochsichere Applikation ist wertlos, wenn ihre Sicherheitsmaßnahmen durch Ausnutzung von Schwachstellen in unsicheren Basissystemen oder anderen, nicht ausreichend voneinander isolierten Anwendungen unterwandert werden können.

Gängige Maßnahmen und Methoden zur Vermeidung dieser Situation wurden in diesem Paper vorgestellt. Es ist aber letztlich an jedem Serverbetreiber selbst gelegen, die für ihn sinnvollen Maßnahmen zu implementieren und evtl. um weitere zu ergänzen, um ein sicherheitstechnisch ausgewogenes Gesamtkonzept zu realisieren.

---

<sup>10</sup><http://www.truecrypt.org>

## Literatur

- [1] Matthew Moodie, Kunal Mittal, *Pro Apache Tomcat 6*, Apress, 2007, Seite 228ff
- [2] Jason Brittain, Ian F. Darwin, *Tomcat - The Definitive Guide*, O'Reilly, 2007, Seite 205ff
- [3] Daniel B. Cid, *Attacking Log analysis tools*, <http://www.ossec.net/en/attacking-loganalysis.html>