# Privilegientrennung als Programmiertechnik

Martin Richtarsky

26. Juli 2003

martin@codefactory.de

Studiengang Informatik, M97, Matrikel-Nr. 26356

Ergänzungslehrgebiet "IT-Sicherheit und Datenschutz"

Dipl.-Wirtsch.-Inf. Jana False, SS 2003

TU Ilmenau

## 1 Einführung

Dienste sind ein essentieller Bestandteil von Netzwerken. Im Internet zählen Email, Usenet, WWW, Chat und FTP (Dateitransfer) zu den klassischen Diensten. Darüber hinaus entstanden mit der zunehmenden Popularität des Netzes immer neue Anwendungen, Instant Messaging gehört dazu. Alle diese Dienste kommunizieren über ein Netzwerk unter Verwendung des Client-Server-Modells. So gibt es POP- und SMTP-Server zum Abrufen und Versenden von Mail, dedizierte Usenet- und Newsserver und eine Vielzahl von Web- und FTP-Servern. Sie bieten Dienste für jedermann an, teilweise mit Authentifizierung einer nutzerabhängigen Identität, oftmals auch für anonyme Benutzer. In beiden Fällen kann ein Client Aktionen auf einem Server auslösen (z.B. "Sende Listing des public-Verzeichnisses").

Solange es sich dabei um vom Server-Programmierer vorgesehene Aktionen handelt, ist dies kein Problem. Beeinhaltet das Programm allerdings Fehler, so lassen sich oft durch speziell präparierte Anfragen beliebige Aktionen ausführen. Der Dienst ist kompromittiert und steht dem Angreifer zur freien Verfügung.

Es gibt viele Möglichkeiten, mit dieser Problematik umzugehen. Dazu gehören:

- Sorgfältige Überprüfung des Programms. Dies ist mit viel Aufwand verbunden, ein Beweis für die Korrektheit kann mit vertretbarem Aufwand nur für kleine Teile erbracht werden.

- Verwendung von typsicheren Programmiersprachen. Hier kann der Compiler einen Großteil der Fehler abfangen, die zur Laufzeit zu Problemen führen könnten. Es bleibt allerdings ein Restrisiko, z.B. könnte der Compiler nicht korrekten Code erzeugen oder verwendete Betriebssystemroutinen Fehler enthalten.
- Restriktive Vergabe von Privilegien. Ein Prozess sollte nur Privilegien erhalten, die er zur Erbringung seiner Funktion benötigt. Dies stellt unter Unix-Derivaten ein Problem dar, da z.B. Netzwerkdienste, die Ports im Bereich von 1-1023 verwenden, als Superuser `root` laufen müssen. Dazu zählen praktisch alle wichtigen Internet-Dienste.

Die hier vorgestellte Methode der Privilegientrennung geht vom letzten Ansatz aus und versucht, dessen Einschränkung zu beseitigen. Zunächst wird in Abschnitt 2 kurz das Problem der Privilegieneskalation erläutert. Anschließend wird das Prinzip der Privilegientrennung als eine Lösungsmöglichkeit vorgestellt (Abschnitt 3).

Die Betrachtungen beziehen sich vor allem auf Unix-Systeme, Unterschiede zu anderen Architekturen liegen aber oftmals nur im Detail.

# 2 Das Problem der Privilegieneskalation

## 2.1 Begriffe

### 2.1.1 Privileg

Unter einem Privileg versteht man ein Attribut eines Objektes, das zum Ausführen einer Aktion berechtigt. Aktionen sind unter anderem das Löschen und Anlegen von Dateien oder das Warten auf Netzwerkverbindungen auf einem bestimmten Port.

### 2.1.2 Prozess

Ein Prozess ist eine abgeschlossene Einheit, die einem Nutzer (User ID, UID) und einer Gruppe (Group ID, GID) zugeordnet ist und ein Programm ausführt. In Abhängigkeit von den Privilegien von Nutzer und Gruppe darf ein Prozess bestimmte Aktionen ausführen. Somit ist ein Prozess ein Objekt im Sinne von Abschnitt 2.1.1. Ein Programm kann mehrere Prozesse als parallele Ausführungseinheiten verwenden.

## 2.2 Privilegieneskalation durch Ausnutzen von Schwachstellen

Üblicherweise erlaubt ein Programm nur bestimmte, vor der Programmierung definierte Aktionen. Ein NFS-Daemon beispielsweise ermöglicht das Exportieren von Bereichen des Dateisystems eines Rechners. Andere Rechner im Netzwerk können diese Freigaben dann in ihr Dateisystem einbinden. Sobald dieser über das Netz erreichbare Dienst Fehler enthält, ist - abhängig von der Art des Fehlers - Ausführen von Code im Kontext des Serverprozesses möglich.

So wurde erst kürzlich im Linux-NFS-Daemon eine Schwachstelle gefunden [Niewiadomski03], die es dem Angreifer erlaubt, durch das Senden einer speziell präparierten Anfrage einen Buffer Overflow (siehe Abschnitt 2.3.2) auszulösen und dadurch Code mit den Privilegien des Serverprozesses auszuführen. Das betroffene Programm läuft üblicherweise als `root`, das Ausnutzen der Schwachstelle kann damit zu einer kompletten Übernahme des Servers führen.

Damit liegt eine Privilegieneskalation vor: Der Serverprozess stellt über die vorgesehenen Aktionen dem Client nur einen Bruchteil seiner Rechte zur Verfügung. Im Fall von NFS ist dies der Zugriff auf Dateien eines beliebigen Nutzers, die freigegeben wurden. Durch eine Kompromittierung lassen sich aber auch alle anderen dem Prozess zugewiesenen Privilegien ausnutzen.

## 2.3 Sicherheitskritische Fehler bei der Programmierung

### 2.3.1 Unzureichende Validierung von Eingabedaten

Dieser Fehler ist in sehr vielen Applikationen zu finden, unabhängig von der verwendeten Programmiersprache. Als Beispiel sei hier eine fiktive Webapplikation genannt, die per URL-GET-Parameter einen Dateiname erhält, die Datei aus einem lokalen Verzeichnis auf dem Server liest, einige Formatierungen vornimmt und ein HTML-Dokument an den Nutzer sendet. Werden keine Überprüfungen des Dateinamens vorgenommen, kann ein Angreifer beliebige Pfade konstruieren, die aus dem vorgesehenen Verzeichnis herausführen. Dazu können absolute Pfade, beginnend mit '/', oder die Referenz auf das übergeordnete Verzeichnis, '..', verwendet werden. Zwei Beispiele für mögliche Dateinamen:

- `../../../../etc/shadow`: Es wird mit einer hinreichenden Anzahl von '..' in das Wurzelverzeichnis gewechselt, dann wird die Datei mit den verschlüsselten Passwörtern eingelesen. Der Angreifer kann anschließend mit einem Brute-Force- oder Wörterbuch-Angriff offline diese Passwörter entschlüsseln. Diese Variante kann eingesetzt werden, wenn das Programm absolute Pfadnamen (mit `/` beginnend) zurückweist.
- `showpage.php`: Es wird der Name des Skripts als anzuzeigende Datei übergeben. Der Angreifer kann den Code dieses Skripts auf weitere Schwachstellen untersuchen (Fehler, Zugangsdaten für Datenbanken etc.).

### 2.3.2 Pufferüberläufe

Pufferüberläufe (im folgenden "Buffer Overflows") treten auf, wenn in einen Puffer mehr Daten kopiert werden, als Speicher dafür alloziert wurde. Die einfachste Form bezieht sich auf Funktionen, die einen Puffer auf dem Stack anlegen, dort hinein Daten schreiben und dann die Kontrolle an den Aufrufer zurück geben. Ein Beispiel in C:

```
void overflow(char* str)
{
    char buffer[1024];

    strcpy(buffer, str);
}

int main(int argc, char* argv[])
{
    overflow(sehr_langer_string);
}
```

Beim Aufruf von Funktionen in C werden bestimmte Daten auf dem Stack (FIFO-Struktur) gespeichert. Dazu gehören die Adressen der Parameter (`str` bei `overflow()`) und die Rücksprungadresse zum Aufrufer (`main()`). Auch lokale Datenstrukturen, im Beispiel `buffer`, werden auf dem Stack angelegt.

Wenn man sich nun veranschaulicht, dass der Stack von unten nach oben wächst, und somit `buffer` im Speicher vor den Parametern der Funktion und vor der Rücksprungadresse liegt, wird klar, wie man die Kontrolle des Prozesses übernehmen kann. Dazu ist es lediglich nötig, der `overflow()`-Funktion einen String zu übergeben, der länger als 1024 Zeichen ist und somit die Rücksprungadresse mit einer definierten, eigenen Adresse überschreibt. Der Ablauf im Einzelnen:

- Aus `main()` heraus wird `overflow()` mit einem String länger als 1024 Zeichen aufgerufen.
  - Die Argumente der Funktion, hier ein `char*`, werden auf den Stack geschrieben.
  - Die Rücksprungadresse wird auf den Stack geschrieben.
  - Die Funktion `overflow()` wird ausgeführt.
    - `buffer` wird mit einer Länge von 1024 Zeichen auf dem Stack angelegt.
    - Der übergebene String wird in den Puffer kopiert. Dabei tritt nach dem 1024. Zeichen ein Überlauf auf, nachfolgende Speicherbereiche werden überschrieben.

– Am Ende der Funktion wird die (überschriebene) Rücksprungadresse vom Stack geholt und die Programmausführung dort fortgesetzt.

Damit lässt sich an eine beliebige Stelle im Adressraum des Prozesses springen, die Codeausführung wird dann dort fortgesetzt. Da nur in den seltensten Fällen der Code, der ausgeführt werden soll, bereits vorhanden ist, wird er bei den meisten Buffer Overflow-Exploits zusammen mit der modifizierten Rücksprungadresse und Daten zum Auffüllen des Puffers übertragen. Dieser Code muss vorher für die zu attackierende Architektur kompiliert worden sein und ist meist ein kleines Programm zum Ausführen einer Shell, die mit den Privilegien des Prozesses ausgestattet ist und dem Angreifer leicht weitere Schritte ermöglicht.

Buffer Overflows sind die wichtigste Klasse von sicherheitsbezogenen Fehlern in Programmen [Festa99]. Dies ist darauf zurückzuführen, dass ein Großteil der eingesetzten Dienste in C/C++ geschrieben sind. In diesen Programmiersprachen gibt es - im Gegensatz zu Java oder C# - kein Bounds Checking, womit das Überschreiben von Puffergrenzen erkannt werden könnte. Weiterhin werden Buffer Overflows durch ausführbare Stack-Seiten begünstigt. Wie bereits erwähnt, wird der auszuführende Code meist vom Angreifer in den Puffer geladen, der sich auf dem Stack befindet. Dies ist möglich, weil viele Architekturen das Ausführen von Code auf dem Stack (der eigentlich nur als "Datenhalde" dienen sollte) nicht verhindern können (z.B. Intel x86). Es existieren jedoch Patches für einige Betriebssysteme, die dies beheben. Der wichtigste ist der Linux-Patch des Openwall-Projekts [SolarDesigner03].

# 3 Das Prinzip der Privilegientrennung

## 3.1 Verhinderung der Privilegieneskalation

Zur Vermeidung sind zwei Strategien denkbar:

- Den Prozess von vornherein mit so wenig wie möglich Privilegien ausstatten. Dieses grundlegende Prinzip wird bereits von einem Großteil der Dienste befolgt, ist aber aufgrund von gewissen Einschränkungen auf Unix-Systemen nicht konsequent durchsetzbar. So muss ein Dienst, der auf einem Port zwischen 1-1023 Anfragen entgegen nimmt, als `root` gestartet werden, um diesen Port binden zu können.
- Aufteilen des Programms in zwei Prozesse und Privilegientrennung.

Der zweite Ansatz soll im folgenden näher erläutert werden.

## 3.2 Privilegientrennung

Das Programm wird in zwei unabhängige Prozesse geteilt. Der Elternprozess ist privilegiert und erzeugt einen unprivilegierten Kindprozess. Die Prozesse kommunizieren über Interprocess Communication (IPC) miteinander. Die Idee ist nun, einen Großteil des Programms im Kindprozess ablaufen zu lassen und nur für Aktionen, die Privilegien benötigen, den Elternprozess zu kontaktieren.

### 3.2.1 Kindprozess

Der Kindprozess unterliegt folgenden Einschränkungen:

- Zuordnung zu einem unbenutzten User/einer unbenutzten Gruppe im System (UID/GID). Dieser User hat keinerlei Rechte (kein Home-Verzeichnis, keine Login-Shell usw.).
- Setzen des Wurzelverzeichnisses des Prozesses per `chroot()` auf ein unbenutztes Verzeichnis, in das der Prozess nicht schreiben kann. Damit kann er garantiert auf keine wichtigen Dateien oder Verzeichnisse zugreifen, auch nicht auf normalerweise world-writable Verzeichnisse wie `/tmp` , wodurch auch bestimmte Denial-of-Service-Attacken verhindert werden (Erzeugen von großen Files, um das Dateisystem zu füllen).

Eine Kompromittierung dieses Prozesses würde dem Angreifer kaum verbesserte Privilegien verschaffen. Eine verbleibende Schwachstelle wäre der mögliche Netzwerkzugriff und damit ein Spoofing von Verbindungen. Verbesserte Sicherheit lässt sich aber durch das Einschränken der Systemrufe, die der Prozess nutzen darf, erreichen [Provos03].

### 3.2.2 Elternprozess

Dieser Prozess implementiert eine minimale, überschaubare Schnittstelle. Es werden Anfragen vom Kindprozess entgegen genommen, eine Gültigkeitsprüfung durchgeführt und gegebenenfalls die gewünschte Aktion ausgeführt. Anschließend wird ein Ergebnis oder eine Ressource (z.B. ein Netzwerksocket) an den Kindprozess zurückgegeben. Durch den geringen Funktionsumfang ist dieser Prozess sehr einfach zu implementieren. Der sicherheitskritische Teil des Programms ist sehr gut isoliert, die Codegröße ist begrenzt und damit sehr gut geeignet für ein gezieltes Auditing.

### 3.2.3 Schnittstelle Elternprozess-Kindprozess

Vor der Implementierung eines Programms müssen die Teile identifiziert werden, die bestimmte Privilegien benötigen. Diese Teile werden in einer Schnittstelle

gekapselt. Der Kindprozess bedient sich dieser, um privilegierte Aktionen vom Elternprozess durchführen zu lassen; die Anfragen werden durch Mechanismen der Interprocess Communication umgesetzt, die Prozesse kommunizieren dabei über Pipes oder Sockets.

# 4 Realisierung der Privilegientrennung am Beispiel von OpenSSH

Das SSH-Protokoll ermöglicht das sichere Einloggen auf andere Rechner über eine nicht vertrauenswürdige Verbindung. OpenSSH ist eine populäre, freie Implementierung dieses Protokolls. Seit März 2002 existiert eine komplett privilegienseparierte Version von OpenSSH, die in vielen Distributionen bereits standardmäßig verwendet wird. Hier soll auf einige interessante Punkte aus dem Paper dazu [Provos02] eingegangen werden.

## 4.1 Struktur

Der Standard-Port von SSH ist 22, deshalb muss OpenSSH als `root` gestartet werden. Bei einer eingehenden Verbindung eines Clients auf diesem Port startet der Server einen ersten Kindprozess, der ebenfalls `root`-Privilegien hat, um Aktionen wie das Erstellen von Pseudo-Terminals durchführen zu können. In älteren Versionen von OpenSSH lief der Großteil des Programms in diesem Prozess ab. Fehler führten sofort zu `root`-Rechten [ISS02, Zalewski01, Pol02]. Die Privilegientrennung ändert dies dahingehend, dass dieser Kindprozess als Monitor läuft und wiederum einen unprivilegierten Kindprozess erzeugt ("Worker"), der einen Großteil der Arbeit, inklusive Netzwerkkommunikation, erledigt und nach dem in Abschnitt 3.2 geschilderten Prinzip nur für einige Aktionen auf den Elternprozess zurückgreift.

### 4.1.1 Monitor

Dieser Prozess ist als ein endlicher Automat implementiert. Für jeden Zustand sind die Aktionen definiert, die der Worker anfordern kann. Sobald eine ungültige Anfrage gestellt wird, terminieren beide Prozesse. Zu den implementierten Aktionen zählen:

- Schlüsselaustausch. Hierfür ist Zugiff auf die Datei */etc/moduli* nötig (der unprivilegierte Worker hat keinen Zugriff auf das Dateisystem).
- Überprüfen von Login und Passwort (Zugriff auf */etc/passwd*, */etc/shadow* oder PAM nötig).
- Public Key-Authentifizierung.

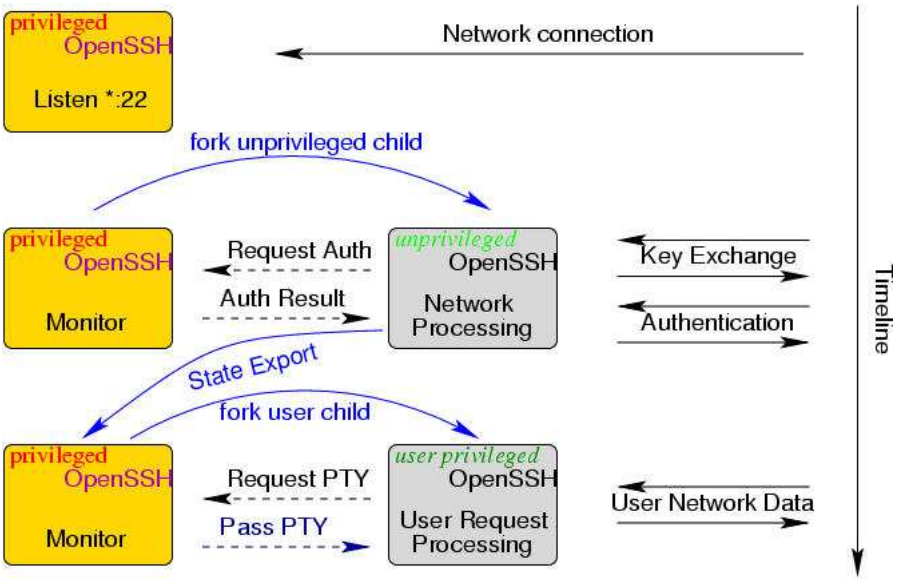


Abbildung 1: Privilegientrennung in OpenSSH [Provos02]

## 4.2 Ablauf

Der Ablauf einer OpenSSH-Sitzung ist in Abbildung 3.2 dargestellt.

Ein Client verbindet sich zum OpenSSH-Server, der daraufhin einen Monitor-Prozess per `fork()` erstellt. Dieser erstellt wiederum einen Worker-Prozess, der nun für die weitere Bearbeitung inklusive Netzwerkkommunikation zuständig ist.

Zunächst wird der Schlüsselaustausch durchgeführt. SSHv2 unterstützt dazu zwei Verfahren, es gibt für beide eine Funktion in der Monitor-Worker-Schnittstelle. Dies sind die einzigen Aktionen, die der Worker im Initialzustand anfragen darf. Nach erfolgreichem Schlüsseltausch findet im Monitor ein Zustandsübergang statt. Nun ist als einzige Aktion die Überprüfung erlaubt, ob der zu authentifizierende Nutzer auf dem System existiert. Ist dies erfolgt, kann die eigentliche Authentifizierung stattfinden. Wenn auch diese erfolgreich ist, kann der Nutzer Zugang zum System erhalten. Dazu muss dem Worker die UID/GID des Nutzers zugeordnet werden, um den Zugriff auf dessen Dateien und Verzeichnisse zu ermöglichen. Dies ist allerdings nur einem `root`-Prozess möglich, deshalb muss OpenSSH einen Umweg gehen. Zunächst wird der aktuelle Zustand des Workers in den Monitor exportiert. Dazu gehören die ausgewählten Verschlüsselungsalgorithmen mit ihren Schlüsseln; Statistiken sowie das Wörterbuch für die Kompression. Dann wird der Worker beendet, der Monitor erzeugt einen neuen Worker, der zunächst mit `root`-Privilegien ausgestattet ist und sich deshalb auf die UID/GID des authentifizierten Nutzers binden kann. Dann wird

der Zustand zurückexportiert und das Protokoll kann fortgesetzt werden.

Der Monitor wird noch für einige wenige Operationen benötigt, unter anderem das Erzeugen von Pseudo-Terminals und das Erneuern der Schlüssel.

### 4.3 Implementierungsaufwand, Ergebnis

Die ursprüngliche OpenSSH-Implementierung bestand aus 44000 Zeilen Quellcode, wovon 950 modifiziert werden mussten. Weitere 3000 Zeilen wurden für die Implementierung des Monitors und der Schnittstelle hinzugefügt. Als Resultat läuft ein Großteil des Codes ohne Privilegien.

Diese Implementierung ist damit immun gegen drei bekannte OpenSSH-Schwachstellen:

- Ein Integer-Overflow einer Paketverarbeitungsroutine ermöglichte `root`-Zugriff vor der Authentifizierung [Zalewski01]. Um diese Schwachstelle auszunutzen, braucht der Angreifer somit kein gültiges Login auf dem System. Mit Privilegientrennung hat der Worker-Prozess zu diesem Zeitpunkt keine ausnutzbaren Rechte (mit der in Abschnitt 3.2.1 genannten Ausnahme der Netzwerkkommunikation).
- Ein off-by-one-Fehler erlaubte nach der Authentifizierung das Erlangen von `root`-Rechten [Pol02]. Bei Privilegientrennung erlangt der Angreifer nur die Rechte des bereits authentifizierten Nutzers.
- Ein Fehler in der von OpenSSH verwendeten zlib-Bibliothek führte zu `root`-Rechten vor der Authentifizierung. Mit Privilegientrennung erlangt der Angreifer wiederum nur die Rechte des unprivilegierten Worker-Prozesses.

Eine durch Privilegientrennung gehärtete Implementierung ist natürlich auch immun gegen noch unbekannte Angriffe, die den unprivilegiert laufenden Code betreffen. Da Fehler nie ganz ausgeschlossen werden können, ist ein sicheres Design, das die Folgen eines Angriffs mildert und begrenzt, essentiell.

## 5 Weitere Implementierungen

### 5.1 vsftpd (Very Secure FTP Daemon)

Dieser FTP-Server von Chris Evans verwendet ebenfalls Privilegientrennung als zentrales Konzept [Evans]. Einige Besonderheiten:

- Das FTP-Protokoll verlangt, dass Datenverbindungen standardmäßig von Port 22 ausgehen. Da dies ein privilegierter Port ist, existiert im Monitor und in der Schnittstelle eine Funktion, die dies für den Worker realisiert. Dafür nutzt vsftpd die Möglichkeit moderner Unix-Systeme, Socket-Deskriptoren über eine bestehende Socket-Verbindung auszutauschen.
- Der Monitor verwendet Linux Capabilities, die es einem `root`-Prozess erlauben, bestimmte Privilegien feingranular abzugeben. Abhängig von der Konfiguration des Servers kann dies dazu führen, dass überhaupt keine Privilegien benötigt werden. Dann beendet sich der Monitor, der Worker kann alle Aufgaben allein erledigen.

### 5.2 Postfix und qmail

Diese beiden Message Transfer Agents (MTAs) implementieren das Konzept der Privilegienseparierung ebenfalls mit mehreren Prozessen, die über IPC kommunizieren. Dabei steht die Eltern-Kind-Beziehung nicht so sehr im Vordergrund wie bei OpenSSH, es handelt sich vielmehr um kooperierende, gleichberechtigte Prozesse.

## 6 Privman

NAI Labs stellt die Opensource-Bibliothek Privman für C zur Verfügung, mit der leicht privilegienseparierte Programme erstellt werden können [Privman].

### 6.1 Verwendung

Privman implementiert die Privilegientrennung wie in Abschnitt 3.2 dargestellt. Ein Programm ruft am Anfang die Funktion `priv_init()` auf, hier wird der unprivilegierte Kindprozess erzeugt, in dem das eigentliche Programm abläuft. Die Verbindung zum privilegierten Elternprozess erfolgt über eine Pipe. Möchte man eine privilegierte Funktion verwenden, fügt man einfach das Präfix `priv_` hinzu. Der Aufruf wird dann von der Bibliothek an den Elternprozess geleitet und die entsprechende Aktion durchgeführt.

Beispiel: Binden eines Sockets auf Port 80. Dafür sind `root`-Privilegien nötig.

```
...
addr.sin_port = htons(80);
...
bind(socket, &addr, sizeof(struct sockaddr));
```

ändern in:

```
priv_init();
...
addr.sin_port = htons(80);
...
priv_bind(socket, &addr, sizeof(struct sockaddr));
```

Die Rechte des privilegierten Prozesses lassen sich in einer Konfigurationsdatei einstellen. Die Konvertierung eines existierenden Dienstes in eine privilegiengetrennte Version gestaltet sich damit sehr einfach.

### 6.2 Einsatz

Mit Hilfe der Bibliothek wurden bereits WU-FTPD, BSD ftpd und THTTPD erfolgreich gepatcht.

## 7 Diskussion

Privilegientrennung ist eine einfache Programmiertechnik, die das Erstellen sehr sicherer Systeme ermöglicht. Das Prinzip, einem Prozess nur die wirklich notwendigen Privilegien zu geben, wird konsequent verfolgt und führt zur Aufteilung von Programmen in privilegierte und unprivilegierte Teile. Es werden wohldefinierte Schnittstellen zwischen den Prozessen benötigt, deshalb muss in der Entwurfsphase besonders sorgfältig gearbeitet werden.

Bei der Modifikation existierender Programme hängt das Ergebnis weitgehend von der Struktur und Dokumentation des Codes ab.

Geschwindigkeitseinbußen sind praktisch nicht zu beobachten. Bei der Realisierung neuer Dienste sollte die Privilegientrennung unbedingt eingesetzt werden. Zusammen mit weiteren Mechanismen, wie z.B. ACLs [grsecurity], bietet sie sehr gute Möglichkeiten der feingranularen Rechtevergabe.

Privilegientrennung bleibt wirkungslos, wenn Fehler in Betriebssystemroutinen ausgenutzt werden oder die Implementierung des Monitors selbst Schwachstellen enthält.

## Literatur


[Evans] Evans, C.: *vsftpd Design.* http://vsftpd.beasts.org/DESIGN

[Festa99] Festa, P.: *Study says "buffer overflow" is most common security bug.* CNET News.com, November 1999, http://news.com.com/2100-1001-233483.html?legacy=cnet

[grsecurity] grsecurity, http://www.grsecurity.net

[ISS02] ISS X-Force: *OpenSSH Remote Challenge Vulnerability.* Juni 2002, http://www.securityfocus.com/archive/1/278818

[Privman] NAI Labs: *Privman.* http://opensource.nailabs.com/privman/

[Niewiadomski03] Niewiadomski, J.: *Linux nfs-util xlog() off-by-one Bug.* Bugtraq, Juli 2003, http://marc.theaimsgroup.com-/?l=bugtraq&m=105820223707191&w=2

[Pol02] Pol, J.: *OpenSSH Channel Code off-by-one Vulnerability.* März 2002, http://www.securityfocus.com/bid/4241

[Provos02] Provos, N.: *Preventing Privilege Escalation.* CITI Techreport 02-2, August 2002.

[Provos03] Provos, N.: *Systrace - Interactive Policy Generation for System Calls.* http://www.citi.umich.edu/u/provos/systrace/

[SolarDesigner03] Solar Designer: *Non-executable User Stack Linux Patch.* Openwall Project, 2003, http://openwall.com/linux/README

[Zalewski01] Zalewski, M.: *Remote Vulnerability in SSH Daemon CRC32 compensation attack detector.* Februar 2001, http://razor.bindview.com/publish-/advisories/adv_ssh1crc.html