Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur

Großer Beleg

Kapselung aktiver Inhalte

Sebastian Lehmann
Matr.-Nr. 2296852
sepp@prak.org

Betreuer: Dr.-Ing. Andreas Westfeld
Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig

19. Februar 2001

diese Seite durch die Aufgabenstellung ersetzen!

Inhaltsverzeichnis

1 Einleitung **4**

2 Ziele **4**
2.1 Schutzziele . . . 4
2.2 Angreifermodell . . . 5
2.3 Ausführende Instanz . . . 5
2.3.1 Interpreter . . . 5
2.3.2 Prozesse . . . 6
2.4 Zugriffskontrolle . . . 7
2.4.1 Zugriffskontrolllisten . . . 7
2.4.2 Capabilitys . . . 8

3 Analyse von Anwendungsmöglichkeiten der Schutzmethoden **8**
3.1 Programme unbekannter Herkunft . . . 8
3.2 Serverprozesse . . . 9

4 Implementierung **10**
4.1 Designentscheidungen . . . 10
4.2 Änderungen im Linuxkern . . . 11
4.2.1 Erweiterung von Kerndatenstrukturen . . . 11
4.2.2 exec() . . . 13
4.2.3 fork() . . . 13
4.2.4 open_namei(), may_create(), may_delete() . . . 13
4.2.5 slcapinit() . . . 15
4.2.6 slcapwait() . . . 15
4.2.7 slcapget() . . . 15
4.2.8 slcapset() . . . 16
4.2.9 slcapcache() . . . 17
4.2.10 slcapinfo() . . . 18
4.3 Hilfsprogramme . . . 18
4.3.1 slcaprun . . . 18
4.3.2 capcontrol . . . 19
4.4 Zusammenspiel . . . 19

5 Messungen und Bewertung **21**
5.1 Messanordnung . . . 21
5.2 Messungen . . . 21
5.2.1 Erstes open() . . . 21
5.2.2 Weiteres open() ohne Cachebenutzung . . . 22
5.2.3 Weiteres open() mit Cachebenutzung . . . 22
5.2.4 Praktischer Test mit rgrep . . . 23
5.3 Bewertung . . . 23

6 Zusammenfassung und Ausblick **23**

1 Einleitung

Auf der Webseite des Projekts „distributed.net“[1], welches durch verteiltes Rechnen kryptografische Schlüssel bricht, steht als Warnung:

> Important note
> This is the official listing of distributed.net clients. They have been tested to be functioning correctly. The binaries listed here are the ONLY ones you should be using. Trojan horses and other perverted versions have been known to have been circulated. Please do not make attempts to mirror or redistribute the client binaries, either via your own webftp server or other means. If you wish to provide a convenient method for your visitors to download clients, please provide a link to it on our FTP site or (preferably) to this page. distributed.net has set policies and terms regarding the use of these clients. Please read them. Downloading and installing the client on any machine implies understanding and agreement with these terms.

Dieses Beispiel zeigt die Bedrohung von Programmen unbekannter Herkunft, welche man aus dem Internet lädt, per E-Mail zugeschickt oder von Freunden auf selbstgebrannter CD bekommt. Diese Programme können im Hintergrund Daten auf dem Rechner verändern und löschen, sobald der Nutzer sie startet. Die beiden in Deutschland meist genutzten Betriebssysteme Windows und Linux haben keine Schutzmechanismen vor solchen trojanischen Pferden. In diesem Beleg soll eine einfache Erweiterung des Linuxkerns geschaffen werden, mit deren Hilfe jedes Programm in einer gesicherten Umgebung, einem „Sandkasten“, gestartet werden kann. Alle Zugriffe auf Daten und Hardware des Rechners können überwacht, protokolliert und gesteuert werden.
Zum Erreichen dieser Zielstellung wird im zweiten Kapitel analysiert, welche schützenswerten Objekte ein Rechensystem hat und welchen Gefahren diese ausgesetzt sind. Darauf folgt eine Vorstellung verschiedener bekannter Schutzmethoden. Im dritten Kapitel wird anhand von Szenarien der Nutzen solcher Schutzmethoden erläutert und im vierten Teil die Implementierung einer Schutzumgebung in Linux beschrieben. Es folgen Performancemessungen und Bewertungen der Arbeit im fünften Kapitel. Abgeschlossen wird die Arbeit mit einer Zusammenfassung und dem Ausblick auf noch offene Aufgaben.

2 Ziele

Zunächst wird analysiert, welche Gefahren aktive Inhalte für einen Rechner darstellen[1], und es werden Ansätze unterbreitet und diskutiert, wie diesen Gefahren entgegengewirkt werden kann.

2.1 Schutzziele

Geschützt werden sollen primär auf dem Rechner gespeicherte Daten. Dieses ist das wohl offensichtlichste Ziel. Niemand möchte, dass ein aus dem Internet geladenes Programm unberechtigt Dateien auf dem Rechner verändert oder löscht. Dieses bezieht sich sowohl auf private Daten als auch auf Programmcodes oder Konfigurationsdateien. Maßnahmen zur Sicherung vor unberechtigter Veränderung von Daten sind notwendig, um die Verfügbarkeit und Integrität[2] des Systems zu garantieren. Soll zusätzlich auch die Vertraulichkeit gesichert werden, ist mehr Aufwand zu investieren. Während es bei der Sicherung der Verfügbarkeit und Integrität keine Einschränkung beim

[1] Teile dieses Kapitels wurden [8] entnommen
[2] Verfügbarkeit und Integrität werden in [7] auch als totale Korrektheit bezeichnet

	Verfügbarkeit und Integrität	Vertraulichkeit
Dateizugriffe	Verändernd	Lesend und Verändernd
direkter Hardwarezugriff	Ja	Ja
Kommunikation zu anderen Rechnern	Nein	Ja
Starten von Prozessen	Ja	Ja

Tabelle 1: Notwendige Überwachungen zur Durchsetzung der Schutzziele

Lesen von Daten gibt, müssen nun auch lesende Zugriffe auf das Dateisystem überwacht werden. Das ausgeführte Programm soll nur auf von ihm zur Erfüllung seiner Aufgabe benötigte Dateien zugreifen können. Ein weiterer Punkt zur Sicherung der Vertraulichkeit der Daten des Systems ist die Überwachung der Kommunikation des Programms. Dieser Sachverhalt betrifft sowohl die Kommunikation zu anderen Prozessen auf dem gleichen Rechner, als auch Verbindungen zu anderen Rechnern, beispielsweise über das Internet. Auch muss der direkte Zugriff auf die Hardware überwacht werden, da dadurch ebenfalls Daten verändert werden können. Schlimmer noch, auch die Hardware des Rechners kann zerstört werden, z. B. durch exzessives Benutzen des Schrittmotors des Diskettenlaufwerkes oder das Löschen des BIOS. Neben diesen, das Dateisystem und die Kommunikation betreffenden Schutzzielen, kommen noch weitere hinzu, die das Zusammenwirken mit anderen Prozessen betreffen. Das System muss überwachen, ob und welche anderen Prozesse das Programm starten darf. Sonst könnte der Angreifer beispielsweise durch Starten des Programms /bin/ed Dateien editieren, da /bin/ed als Editor mehr Rechte eingeräumt sein können als dem Angreiferprogramm.

2.2 Angreifermodell

Im Folgenden werden die Möglichkeiten des Angreifers und Maßnahmen beschrieben, die diese Angriffe abwehren. Aus der Aufgabenstellung ist ersichtlich, dass der Anwender auf seinem Rechner aktive Inhalte, die aus unsicherer Quelle stammen, ausführen möchte. Das bedeutet aus der Sicht des Angreifers, dass er die Möglichkeit hat, beliebigen Programmcode auf dem angegriffenen System zu starten. Das unsichere Programm muss also innerhalb einer gesicherten Umgebung ausgeführt werden, einem sogenannten Sandkasten. Alle Zugriffe über die Grenzen dieses Kastens müssen überwacht werden.

2.3 Ausführende Instanz

Zur Festlegung der Grenze zwischen dem aktiven Inhalt und dem umliegenden System gibt es mehrere Ansätze. Hier sollen zwei verbreitete Varianten diskutiert werden, welche sich hauptsächlich dadurch unterscheiden, welche Instanz den aktiven Inhalt ausführt. Einerseits kann dieses ein Interpreterprogramm sein andererseits kann der aktive Inhalt als Prozess direkt auf dem Prozessor des Rechners gestartet werden.

2.3.1 Interpreter

Der erste Ansatz ist die Implementierung der Sicherheit innerhalb eines Interpreters, welcher den Code ausführt. Ein Beispiel für diese Strategie ist eine Java Virtual Machine[6], welche den Bytecode des Java Applets interpretiert. Vorteil dieser Methode ist, dass der Interpreter zu jeder Zeit die volle Kontrolle über alle Aktionen hat, die der aktive Inhalt auf dem zu schützenden System ausführt. Zugriffe auf Betriebsmittel des lokalen Systems oder die Kommunikation des Programms

mit anderen Rechnern oder Programmen müssen vom Programmierer durch Konstrukte der interpretierten Sprache ausgedrückt werden. Der Interpreter kann nun bei jedem Auftreten eines solchen Konstrukts dessen Rechtmäßigkeit überprüfen[5]. Als Grundlage dieser Entscheidung können nun verschiedenste Kriterien herangezogen werden, beispielsweise auch der Ursprung des Programms[3], eine digitale Signatur des Programms oder sogar der Zustand bestimmter Variablen[4]. Nachteilig wirkt sich an dieser Implementierung eines Sicherheitsmechanismus aus, dass die Kontrollmechanismen für jeden Interpreter neu geschrieben werden müssen. Dieses bedeutet, dass auch die Qualitiät der verschiedenen Implementierungen geringer ist, als wenn es eine einzige systemweite Kontrollinstanz gäbe, die ausreichend getestet ist. Des weiteren ist mehr Aufwand für die Konfiguration der verschiedenen Interpreter zu veranschlagen.

2.3.2 Prozesse

Der zweite Ansatz zieht die Grenze zwischen dem zu überwachenden und dem überwachten System an der Grenze des virtuellen Adressraums und zwar aus folgenden Gründen:

- Es gibt eine klare und einheitliche Schnittstelle zwischen dem aktiven Inhalt und dem Kern bzw. anderen Programmen. Der überwachte Prozess läuft im Usermode und hat dadurch keinerlei Zugriff auf die Hardware und auf Daten des Kerns oder anderer Programme. Die einzige Möglichkeit für das Programm, Zugriff auf Daten des Systems zu erlangen, ist der Aufruf von Systemcalls.

- Da die Überprüfung von Systemcalls im Kern erfolgt, bedarf es keiner Änderung von Programmen. Die gleichen Binärdateien, wie sie auf einem bisherigen Linuxsystem ausgeführt wurden, können nun überwacht werden.

- Die Anzahl der Systemcalls ist überschaubar, es gibt in der Version 2.4.0 des Linuxkerns zwar 221, wovon aber nur wenige[5] für den Zugriff auf das Dateisystem verantwortlich sind. Der Aufwand zur Implementierung und Test der zusätzlichen Rechteüberwachung ist also begrenzt, wodurch die Fehleranfälligkeit geringer ist.

- Im Dateisystem des Linuxkerns ist bereits eine Rechteverwaltung integriert, die man erweitern kann. Damit ein Programm Zugriff auf eine Datei erhält, führt es den open()-Systemcall aus. Dieser ist die zentrale Stelle der Rechteüberprüfung. open() gibt einen Dateideskriptor zurück, mittels dessen der Prozess nun alle weiteren Operationen auf der Datei ausführen kann. Bei allen folgenden write()- oder read()-Operationen müssen die Rechte nicht erneut überprüft werden. Durch Erweiterung der Rechteüberprüfung im open()-Systemcall kann man diesen Mechanismus nutzen, um auch hier nur an einer Stelle die Rechte überprüfen zu müssen.

- Unter Linux erfolgt der Zugriff auf die Hardware durch Zugriff auf Gerätedateien[6]. Mit der Zugriffskontrolle auf Dateien ist somit auch die Kontrolle des direkten Hardwarezugriffs möglich.

- Ebenso wie die Verbindung zum Dateisystem durch einen Dateideskriptor erfolgt, gibt es für Verbindungen zu anderen Computern oder zu anderen Prozessen auf dem gleichen System ein

[3] viele Javainterpreter unterscheiden hier Programme, die aus dem Internet geladen wurden und Programme, die von der lokalen Festplatte stammen

[4] Es ließe sich ein System bauen, bei dem durch eine Kommunikationsverbindung eine digitale Signatur geladen und in einer Stringvariablen abgelegt wird und diese dann zur Autentifikation eines Dateizugriffs genutzt wird.

[5] Es gibt 12 Systemcalls, welche direkten Zugriff auf Dateien erlauben. Diese benutzen jedoch alle eine von drei Kernfunktionen zur Rechteüberprüfung: open_namei(), may_create(), may_delete().

[6] wie z. B. /dev/audio für den Zugriff auf die Soundkarte

vergleichbares Konstrukt, den Socket. Auch hier kann die Rechteüberprüfung an zentraler Stelle bei den Systemcalls connect(), bind() und accept() erfolgen und nicht bei jedem read() und write()-Aufruf.

- Wenn der auszuführende aktive Inhalt kein Programm, sondern ein zu interpretierendes Skript ist, kann die Kombination aus Skript und Interpreter als Ausführungseinheit definiert werden. Die Ressourcen, die das Skript zur Erfüllung seiner Aufgabe benötigt, müssen noch um die Ressourcen ergänzt werden, die der Interpreter zum Ausführen des Skripts braucht.

Die eben beschriebene Lösung hat Grenzen:

Notwendiger Zugriff: Oft ist es notwendig, dass zur Erfüllung der Aufgabe ein eventuell auch verändernder Zugriff auf eine Datei erlaubt werden muss. Beispielsweise muss ein Compiler eine Programmdatei verändern können. Das beschriebene System kann nicht dafür sorgen, dass er dabei nicht ein trojanisches Pferd einschleust.

Keine Prüfung von Inhalten: Nach dem Öffnen einer Verbindung zu einem anderen Rechner können nicht mehr die übertragenen Inhalte überwacht werden. Es ist also nicht feststellbar, ob vertrauliche Daten über diesen Kanal an Unberechtigte gelangen.

Keine Sicherheit vor versteckten Kanälen: Es ist nicht auszuschließen, dass es noch nicht untersuchte Möglichkeiten gibt, Daten über die Grenzen des Systems zu transportieren und so die Schutzmechanismen auszuhebeln.

Keine feinere Unterteilung der verwalteten Einheit als Prozess: Während man beim Einsatz eines Interpreters die Granularität der Rechtevergabe beliebig klein wählen kann, z. B. kann der Zugriff auf eine Datei nur einer speziellen Funktion erlaubt sein, ist nun die Einheit, deren Rechte überwacht werden, ein Prozess. So integriert Netscape beispielsweise Browser und E-Mailprogramm in einem Prozess, und da dieser zur Erfüllung der Funktionalität als Mailprogramm verändernden Zugriff auf die Datei braucht, in welcher die Mails gespeichert sind, ist auch dem Browserteil der Zugriff gestattet.

2.4 Zugriffskontrolle

Bei der Implementierung der Zugriffskontrolle gibt es auch wieder mehrere Möglichkeiten. Die bekanntesten sind Zugriffskontrolllisten und Capabilitys[10].

2.4.1 Zugriffskontrolllisten

Zugriffskontrolllisten weisen einem Betriebsmittel eine Liste von Benutzern oder Programmen zu, welche darauf in einer bestimmten Art zugreifen können. Ein Beispiel hierfür sind die im virtuellen Dateisystem des Linuxkerns verwendeten Dateirechte. Zu jeder Datei kann angegeben werden, ob ein bestimmter Benutzer[7], eine bestimmte Benutzergruppe oder jeder lesend oder schreibend zugreifen darf. Dieses ist eine sehr grobe, aber einfach zu administrierende Rechteverwaltung, solange es genügt, nur auf Benutzerebene die Rechte zu vergeben. Soll die Granularität jedoch bis auf Programmebene gesenkt werden, steigt der Aufwand jedoch sehr schnell. Hier muss nun für jedes Programm ein eigener Benutzer angelegt werden, in dessen Namen das Programm gestartet wird[8].

[7] und damit auch Prozesse, die von diesem Nutzer gestartet wurden

[8] Soll jede Kombination aus Programm und Nutzer möglich sein, können es bis zu Anzahl der Nutzer mal Anzahl der Programme verschiedene „virtuelle“ Nutzer werden.

	Zugriffskontrolllisten	Capabilitys
Interpreter		Java Virtual Machine
Rechteverwaltung für Prozesse	bisherige Rechteverwaltung in Linux	zu entwickelnder Ansantz

Tabelle 2: Einordnung bekannter Ablaufumgebungen

Eine Erweiterung dieser Rechtevergabe ist „LIDS“[9]. Mit dieser Erweiterung kann der Zugriff auf ganze Unterverzeichnisbäume mittels einer Regel festgelegt werden. Außerdem können damit auch Programmen ohne den Umweg über die Neuanlage von Nutzern Rechte zugewiesen werden. LIDS vereinfachte also die Administration der Zugriffsrechte.

2.4.2 Capabilitys

Ein weiterer Ansatz der Implementierung der Zugriffskontrolle ist die Nutzung von Capabilitylisten[10]. Diese beschreiben für jedes Programm die Betriebsmittel, auf die der Zugriff erlaubt ist. Die meisten Programme, speziell aktive Inhalte auf Webseiten, benötigen nur sehr wenige Betriebsmittel zur Erbringung ihrer Aufgabe. Das heißt, dass die Capabilityliste des Programms klein und damit einfach zu administrieren ist. Für jedes neue Programm, welches auf dem Rechner gestartet werden soll, ist eine solche Liste vom Administrator oder Benutzer zu erstellen. Durch den Einsatz digitaler Signaturen kann man zuverlässig einem Programm einen Hersteller zuweisen. Dann können die Capabilitylisten auch Rechte an alle Programme eines Herstellers oder einer Zertifizierungsstelle vergeben, welches die Administration erleichtert.

3 Analyse von Anwendungsmöglichkeiten der Schutzmethoden

Dieser Abschnitt beschreibt einige Anwendungen des zu entwickelnden Schutzmechanismus. Dabei wird auf Mängel in existierenden Systemen eingegangen und gezeigt, wie der Einsatz von Capabilitys die Systemsicherheit verbessern kann.

3.1 Programme unbekannter Herkunft

Ein Programm, welches unter Linux gestartet wird, hat alle Rechte, die der startende Nutzer hat[11]. Damit kann es im Allgemeinen alle seine privaten Daten im Homeverzeichnis bearbeiten und Konfigurationsdateien lesen. Es darf außerdem Verbindungen zu anderen Rechnern herstellen. Böswillige Programme könnten also Konfigurationsdaten lesen und an jeden Rechner im Internet schicken, Nutzerdaten ändern oder löschen und aus dem Internet Daten empfangen und dem Nutzer unterschieben. Die Dateirechteverwaltung unter Linux erlaubt keinen Unterschied zwischen den Rechten des Nutzers und den Rechten, die ein vom Nutzer gestartetes Programm hat. Der Anwender kann nur darauf vertrauen, dass der Programmierer keine Schadensroutinen in das Programm eingebaut hat. Diese Hoffnung ist bei bekannten Herstellern, wie beispielsweise Corel, Netscape oder Real,

[9] Linux Intrusion Detection System, siehe auch [3]

[10] Der Begriff Capability wird im Linuxkern bereits für ein Bitfeld verwendet, welches den Zugriff eines Prozesses auf administrative Systemfunktionen, z. B. dem Setzen der Uhrzeit, regelt. Die in diesem Beleg verwendete Bedeutung des Wortes Capability bezieht sich allgemein auf Betriebsmittel und im Folgenden speziell auf den Zugriff auf Dateien

[11] Mittels des SetUID-Mechanismus ist es möglich, das Programm im Namen eines anderen Benutzers zu starten. Dann erhält das Programm alle Rechte dieses Nutzers.

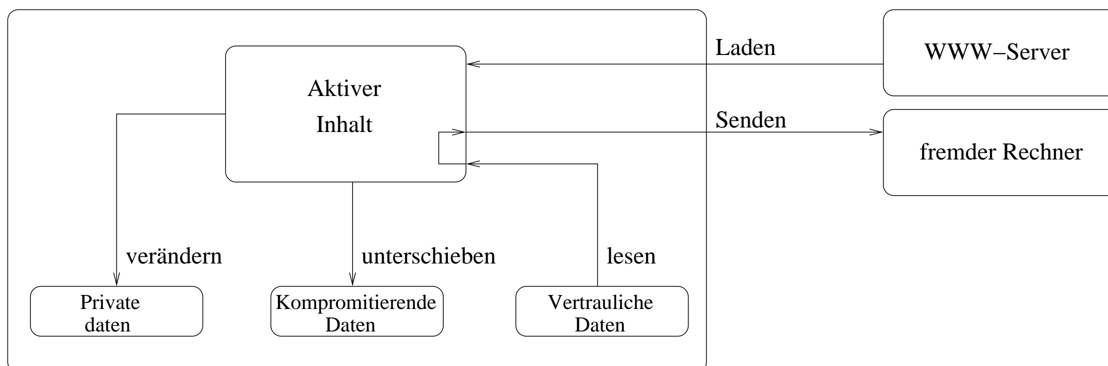

Abbildung 1: *Gefahren durch aktive Inhalte*

nicht unbegründet, wobei aber selbst bei solchen Programmen Daten über den Nutzer zum Hersteller gesendet werden können und werden. Beispielsweise sendet der Netscapebrowser durch das „Smart Browsing" die URLs der vom Nutzer aufgerufenen Webseiten an Netscape. Bei Programmen unbekannter Herkunft hat der Anwender jedoch keine Kontrolle, welche Daten von dem Programm verarbeitet und verändert werden. Die Kontrolle kann jedoch durch den Einsatz von Capabilitylisten wiedererlangt werden, da der Zugriff auf jede Datei überwacht werden kann. Am Anfang startet das Programm mit einer leeren oder einer vorbereiteten Capabilityliste, welche ungefährliche Zugriffe erlaubt, wie beispielsweise das Linken von Bibliotheken oder das Anlegen von Dateien im Verzeichnis /tmp. Alle nicht in der Liste erlaubten Zugriffe auf Daten werden abgefangen, und der Anwender muss seine Zustimmung zu diesen geben. Dadurch kann sich ein Anwender einen Überblick darüber verschaffen, welche Daten das unbekannte Programm benutzt.

3.2 Serverprozesse

Eine andere Anwendung von Capabilitylisten sind Serverprozesse. Aufgrund der Komplexität solcher Programme ist es nicht auszuschließen, dass die Implementierung fehlerhaft ist. Durch solche Fehler ist es möglich, den Prozess zu veranlassen, fremden Programmcode auszuführen. Wie im vorigen Kapitel beschrieben, hat dieser Code nun die gleichen Rechte wie der Nutzer, in dessen Namen der Serverprozess lief. Da viele Serverprozesse unter dem Root-Nutzer laufen, hat das fremde Programm unbeschränkten Zugriff auf alle Dateien und auch auf die Hardware. Dieses soll am Beispiel des Programms „Wu-FTPd" erläutert werden. Am 22.8.2000 wurde zum wiederholten Mal eine Sicherheitslücke in diesem FTP-Server entdeckt[9]. Unter bestimmten Bedingungen kann der Angreifer einen Pufferüberlauf provozieren, das heißt, dass Daten wie z. B. Rücksprungadressen von Funktionsaufrufen auf dem Stapel geändert werden können und so fremder Programmcode, der vorher als Nutzdaten an den Serverprozess geschickt wurde[12], ausgeführt wird. Der Wu-FTPd läuft mit den Rechten des Rootnutzers, wodurch der Angreifercode auch diese Rechte bekommt. Der Angreifer hat nun volle Kontrolle über den Rechner. Durch die Anwendung von Capabilitys kann man einen Pufferüberlauf und das Ausführen fremden Programmcodes nicht verhindern. Der neue Code läuft jedoch immer noch im Kontext des Serverprozesses und kann deshalb nur auf Daten zugreifen, die auch der Serverprozess zum Erfüllen seiner Aufgabe benötigte. Im Falle des oben beschriebenen FTP-Servers kann er also die Konfigurationsdateien des Servers und das FTP-Verzeichnis lesen und in das Uploadverzeichnis und das Logfile schreiben. Das bedeutet eine weit geringere

[12] z. B. kann beim FTP-Server der Login-Name Binärcode enthalten.

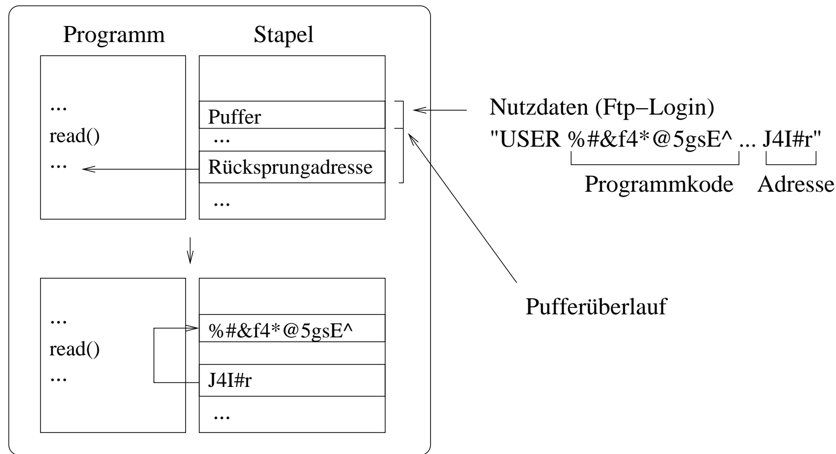

Abbildung 2: *Einbruch durch Pufferüberlauf*

Kontrolle über den Rechner als im Fall ohne Capabilitys. Der Angreifer ist nicht in der Lage, die Konfigurationsdateien des Servers oder anderer Programme zu verändern, da ein Zugriff darauf von der Capability-Kontrollinstanz abgelehnt wird. Das System kann auch zum Feststellen von Einbrüchen benutzt werden, da das Serverprogramm bei normaler Arbeit keine Systemdateien öffnen wird. Nur wenn ein Einbruch stattgefunden hat, werden abnormale Dateizugriffsversuche erfolgen. Dieses kann zur automatischen Alarmierung des Administrators und Abschalten des Serverprozesses benutzt werden. Diese Anwendung der in diesem Beleg entwickelten Capabilityimplementierung im Linuxkern kann mit sehr geringem Aufwand bei fast allen Serverprozessen eingesetzt werden und so entscheidend zur Absicherung von Servern im Internet beitragen.

4 Implementierung

Dieses Kapitel befasst sich mit der Implementierung eines Capability-Kontextes unter Linux. Ziel ist es, ein System zu schaffen, in dem Nutzerprozesse ohne nennenswerte Leistungseinbußen überwacht ausgeführt werden können.

4.1 Designentscheidungen

Grundlage der Implementierung war die zum Beginn des Beleges aktuelle Testversion des Linuxkerns 2.4.0-test10. Nachdem im Laufe der Implementierung die Finalrelease des Kerns veröffentlicht wurde, wurden die bisherigen Änderungen in die Version 2.4.0 übertragen und dann an dieser Version weiterentwickelt. Damit die Änderungen im Kern möglichst gering ausfallen, wurde entschieden, dass die Implementierung der Rechtepolitik[13] von einem eigenen Programm im Nutzeradressraum durchgeführt wird. Diese Entscheidung wirkt sich jedoch nachteilig auf die Performance des Systems aus, da bei jeder Überprüfung der Capabilitys ein Prozesswechsel erfolgen muss, und dieser unter Linux sehr viel länger dauert als ein normaler Systemcall. Um diese Verzögerung zu minimieren, wird ein Cache im Kern implementiert, in dem bereits im Voraus beantwortete Anfragen[14] abgelegt werden können, so dass der Prozesswechsel und der damit verbundene Zeitverlust überflüssig wird.

[13]Dieses beinhaltet das Verwalten der Zugriffsregeln und die Entscheidung, ob ein Zugriff rechtmäßig ist

[14]Die Information, welche Dateien das Programm öffnen wird, kann sich das Kontrollprogramm aus vorigen Programmläufen merken.

Die Zugriffskontrolle wird im Kern als Zusatz zur bisherigen Rechteverwaltung implementiert, das

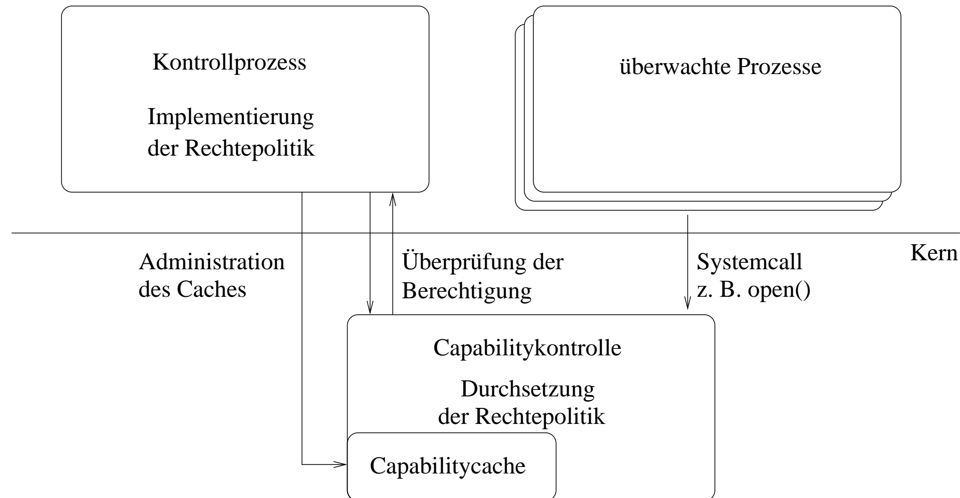

Abbildung 3: *Implementierung der Capabilitykontrolle*

heisst, dass ein Programm sowohl die bisher verwendeten Dateizugriffsrechte als auch die Zustimmung der Kontrollinstanz benötigt, um auf eine Datei zuzugreifen. Um Kollisionen mit bisher im Kern existierenden Strukturen und Namen vorzubeugen, wurde als Präfix für alle im Kern verwendeten Namen „slcap“[15] bzw. „SLCAP“ gewählt. Im Folgenden werden bei den Beschreibungen der Änderungen des Linuxkerns Quellcodezeilen zitiert. Dabei wurden die Fehlerbehandlngen nicht mit aufgeführt. Der vollständige Quellcode ist unter [4] zu finden.

4.2 Änderungen im Linuxkern

Der Zugriff des zu überwachenden Prozesses auf Funktionen des Kerns erfolgt durch Systemcalls. In diese muss die zusätzliche Funktionalität zum Überprüfen der Capalitys eingefügt werden. Die Implementierung dieser neuen Sytemcalls, die Änderungen vorhandener Kernfunktionen, sowie die Erweiterungen bisheriger Kerndatenstrukturen beschreiben die folgenden Abschnitte.

4.2.1 Erweiterung von Kerndatenstrukturen

In der Datei include/linux/sched.h wurde die Struktur struct task_struct um einen Eintrag slcap_id erweitert.

```
struct slcap_taskstruct {
 int flags;
 struct file *file;
};
#define SLCAP_PREPARED 1
#define SLCAP_ACTIVATED 2
#define SLCAP_FLAG_EXEC_HAPPEND 4

struct task_struct {
```

[15]„slcap“ bedeuted Sebastian-Lehmann-Capabilitys

```
 //...
 struct slcap_taskstruct slcap_id;
};
```

In diesem wird der Status des Prozesses bezüglich des Capability-Kontextes geführt. Wichtig sind zwei Flags, eines, welches anzeigt, ob der Capibilty-Kontext aktiviert ist (SLCAP_ACTIVATED) und eines, welches einen vorbereiteten Kontext anzeigt(SLCAP_PERPARED). Ein Prozess wird zuerst mit der Funktion slcapinit() in den Vorbereitungszustand versetzt. Nach dem nächsten Aufruf von exec() wird für diesen Prozess dann der Capability-Kontext aktiviert. Beide Flags werden durch ein fork() auch an die Kindprozesse weitergegeben und bleiben auch nach einem weiteren exec() erhalten. Das bedeutet, es gibt keine Möglichkeit, diese Flags wieder zurückzusetzen. Im

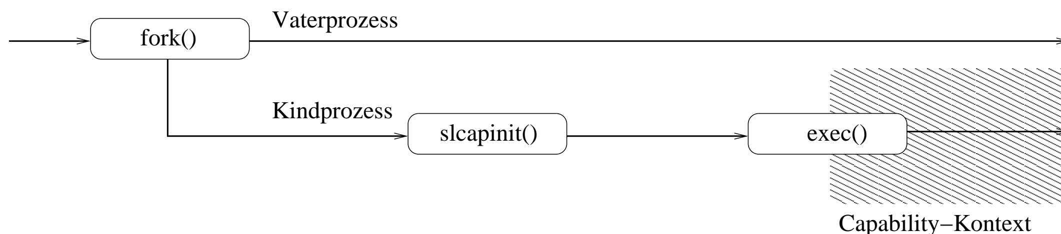

Abbildung 4: *Start eines Programms in einem Capability-Kontext*

Eintrag slcap_id der Struktur task_struct wird weiterhin der Dateideskriptor der ausgeführten Datei gespeichert. Dieser Eintrag ist redundant, wird aber zwecks einfacherem Zugriffs gespeichert. Des weiteren wurden neue Strukturen eingeführt:

Prozesswarteschlange: in die mit

```
wait_queue_head_t slcap_wait_queue;
atomic_t waitingcount=ATOMIC_INIT(0);
```

definierte Warteschlange „slcap_wait_queue“ werden Prozesse eingereiht, welche auf die Beantwortung einer Capabilityanfrage warten. Außerdem wird hier der Kontrollprozess eingereiht, wenn dieser die Funktion slcapwait() aufruft und zur Zeit keine Capabilitys zu überprüfen sind. Die Variable „waitingcount“ zählt die Anzahl der Prozesse, die in der Warteschlange warten.

Anfragenliste: Zu jedem wartenden Prozess existiert ein passender Eintrag in der Liste „slcap_waitlist“.

```
struct slcap_waitentry * slcap_waitlist=NULL;
```

Dieser Eintrag beschreibt die Anfrage, aufgrund derer der Prozess blockiert. Die Anfragen können von verschiedenem Typ sein, z. B. Zugriff auf eine Datei oder Öffnen einer Verbindung zu einem anderen Rechner. Zur Zeit ist nur der Eintrag für den Dateizugriff implementiert.

Capabilitycache: In dieser Liste kann der Kontrollprozess im Voraus beantwortete Anfragen ablegen. Eine genaue Beschreibung dieses Vorgangs erfolgt im Abschnitt slcapcache().

```
struct slcap_capentry * capentrycachelist=NULL;
```

Diese drei Strukturen sind mit einem gemeinsamen Lock gesichert. Jeder Prozess muss diesen mit slcapgetlock() erhalten, bevor er auf die Datenstrukturen zugreifen darf. Anschließend wird der Lock mit slcapreleaselock() wieder freigegeben.

4.2.2 exec()

Um Programme in einem Capability-Kontext starten zu können, muss die exec() Funktion so erweitert werden, dass sie den Kontext aktiviert, wenn dieser bei Aufruf vorbereitet war. Zusätzlich wird der Dateideskriptor der ausgeführetn Datei gespeichert.

```
if ((current->slcap_id.flags&SLCAP_PREPARED)!=0){
 current->slcap_id.flags|=(SLCAP_ACTIVATED|SLCAP_FLAG_EXEC_HAPPEND);
 current->slcap_id.file=file;
};
```

4.2.3 fork()

Analog zu exec() muss auch fork() angepasst werden, damit der Kontext beim Aufruf dieser Funktion mit an den neu erstellten Prozess übergeben wird.

```
p->slcap_id=current->slcap_id;
```

4.2.4 open_namei(), may_create(), may_delete()

Zur Überwachung der Zugriffe auf das Dateisystem müssen alle Systemcalls, welche Dateioperationen ausführen (z. B. open()), mit einer Erweiterung der Rechteüberprüfung versehen werden. Die bisherige Implementierung des Dateisystems im Linuxkern hat die Zugriffskontrolle aller Systemcalls in die Funktionen open_namei(),may_create() und may_delete() konzentriert, so dass nur diese erweitert werden müssen. Dazu wurde der bisherige Aufruf der Funktion permission() um einen weiteren Aufruf erweitert

```
static inline int may_delete(struct inode *dir,
                             struct dentry *victim, int isdir){
 int error;
 if (!victim->d_inode || victim->d_parent->d_inode != dir)
  return -ENOENT;

 error = permission(dir,MAY_WRITE | MAY_EXEC);
 if (error)
  return error;

 error = slcappermission(victim,MAY_WRITE);
 if (error)
  return error;

 ...
```

Diese Funktion slcappermission() stellt für Prozesse, welche innerhalb eines Capability-Kontexts laufen, Anfragen an den Kontrollprozess. Dieser Prozess verwaltet die Capabilitys und erlaubt den Zugriff auf die gewünschte Datei oder unterbindet ihn.

```
 1: int slcappermission(struct dentry *d,int mask){
 2: struct slcap_capentry * capentry;
 3: struct slcap_filewaitentry* waitentry;
 4: int i;
```

```
 5:
 6: if ((current->slcap_id.flags&SLCAP_ACTIVATED)==0) return 0;
 7:
 8:  slcapgetlock();
 9:  slcapgeneratefilename(d,SLCAPBUFFERLEN,slcapbuffer);
10:
11:  capentry=capentrycachelist;
12:  while (capentry!=0) {
13:   struct slcap_capentry * t=capentry;
14:   capentry=capentry->next;
15:   if (current->pid!=t->pid) continue;
16:   for (i=0;(slcapbuffer[i]!=0)&&(slcapbuffer[i]==t->filename[i]);i++);
17:   if ((slcapbuffer[i]!=t->filename[i])&&(t->filename[i]!='*')) continue;
18:   if (slcapcheckmask(mask,t->mask)!=0) continue;
19:   slcapreleaselock();
20:   return 0;
21:  };
22:
23:  waitentry=slcapadd(slcapbuffer,mask);
24:  slcapreleaselock();
25:
26:  wake_up(&slcap_wait_queue);
27:
28:  wait_event(slcap_wait_queue,((waitentry->flags&SLCAPENTRYVALID)!=0));
29:
30:  current->slcap_id.flags&=(~SLCAP_FLAG_EXEC_HAPPEND);
31:  i=slcapcheckmask(mask,waitentry->mask);
32:
33:  slcapgetlock();
34:  slcapremovewaitentry((struct slcap_waitentry *)waitentry);
35:  slcapreleaselock();
36:
37:  return i;
38: }
```

In Zeile 6 wird geprüft, ob ein Capabilitykontext aktiv ist. Ist das nicht der Fall, beendet sich die Funktion sofort. Für Prozesse ohne aktive Capabilitys entsteht so ein minimaler Mehraufwand durch einen Funktionsaufruf und einen Vergleich. Anschließend wird der Capabilitylock geholt und der bereinigte Dateiname generiert. Dieser enthält keine symbolischen Links und „..“-Verzeichnisse mehr[16]. Die Zeilen 11 bis 21 suchen im Cache nach einem Eintrag für die gerade untersuchte Datei. Dabei werden die aktuelle Prozess-ID, der Dateiname und die Rechte verglichen. Wird ein passender Eintrag gefunden, wird der Zugriff auf die Datei erlaubt. Wird kein Eintrag gefunden, wird in Zeile 23 bis 26 die Anfrage in die Capabilitywarteschlange eingereiht und der wartende Kontrollprozess aufgeweckt. Dann legt sich der Prozess schlafen, bis die Anfrage beantwortet ist. Abschließend werden noch mit slcapcheckmask() die geforderten mit den vom Kontrollprozess zurückgegebenen Rechte verglichen und der Eintrag aus der Warteschlange genommen.

[16]Dieses gilt auch für alle anderen Dateinamen, die im Folgenden besprochen werden.

4.2.5 slcapinit()

Diese Funktion wird vom Vaterprozess aufgerufen, bevor das zu überwachende Programm gestartet wird. Durch den Aufruf wird der Capability-Kontext vorbereitet. Der Prozess kann nun immer noch ohne Zugriffsbeschränkungen durch einen Kontext arbeiten. Erst der folgende Aufruf von exec() aktiviert den Capability-Kontext.

```
asmlinkage int sys_slcapinit(int flags)
{
 current->slcap_id.flags|=SLCAP_PREPARED;
 return 0;
};
```

4.2.6 slcapwait()

Der Kontrollprozess ruft slcapwait() auf, um solange zu warten, bis Capabilitys zu überprüfen sind. Dazu schläft der Prozess bis die Variable waitingcount, die die Anzahl der wartenden Anfragen enthält, einen Wert ungleich 0 hat. Zur Zeit ist noch kein Timeout oder eine Reaktion auf Signale eingebaut. Dieses sollte jedoch in der nächsten Ausbaustufe geschehen, da der Kontrollprozess zur Zeit nicht abgebrochen werden kann, wenn er in dieser Funktion wartet.

```
asmlinkage int sys_slcapwait(int flags)
{
 wait_event(slcap_wait_queue,(!atomic_sub_and_test(0,&waitingcount)));
 return 0;
};
```

4.2.7 slcapget()

Mit dieser Funktion kann der Kontrollprozess anstehende Capabilityüberprüfungen vom Kern entgegennehmen. Der Kern liefert die Prozess-ID, den angeforderten Dateinamen und die gewünschte Zugriffsart zurück. Die Funktion liefert immer den ersten Eintrag der Liste der wartenden Anfragen zurück. Erst das Beantworten dieser Anfrage lässt slcapget zur nächsten weiterspringen.

```
 1: asmlinkage int sys_slcapget(int bufferlen,struct slcap_request* buffer)
 2: {
 3: struct slcap_waitentry * t;
 4: struct slcap_request *r=(struct slcap_request *)buffer;
 5: int i;
 6:  slcapgetlock();
 7:  t=slcap_waitlist;
 8:  while ((t!=NULL) && ((t->flags&SLCAPENTRYVALID)!=0)) t=t->next;
 9:  if (t==NULL) {
10:   slcapreleaselock();
11:   return 0;
12:  };
13:
14:  switch (t->type) {
15:   case SLCAP_FILE:
16:    {
```

```
17:      struct slcap_filerequest *fr=(struct slcap_filerequest *)r;
18:      struct slcap_filewaitentry * ft=(struct slcap_filewaitentry *)t;
19:      for (i=0;ft->filename[i]!=0;i++);
20:      if (i+sizeof(struct slcap_filerequest)>bufferlen) {
21:       slcapreleaselock();
22:       return -1; //not enough buffer!
23:      };
24:      for (i=0;ft->filename[i]!=0;i++)
25:       fr->filename[i]=ft->filename[i];
26:      fr->filename[i]=0;
27:      fr->mask=ft->mask;
28:     };
29:     break;
30:    default:
31:     printk("SLCAP: Panic, unknown Type!!!!!\n");
32:   };
33:   r->pid=t->pid;
34:   r->flags=0;
35:   if (t->flags&SLCAP_FLAG_EXEC_HAPPEND)
36:    r->flags|=SLCAP_EXEC_HAPPEND;
37:   r->type=t->type;
38:
39:   slcapreleaselock();
40:   return t->pid;
41: };
```

In den Zeilen 6 bis 12 wird nach einem noch unbeantworteten Eintrag in der Anfragewarteliste gesucht. Wird kein solcher Eintrag gefunden, kehrt die Funktion mit dem Ergebnis 0 zurück. Wenn ein Eintrag gefunden wurde, werden in den Zeilen 14 bis 37 die Daten des Eintrages in den Puffer kopiert, der der Funktion vom Kontrollprozess bereitgestellt wurde. Dabei ist bereits eine Typunterscheidung vorbereitet. Zur Zeit existiert jedoch nur der Typ SLCAP_FILE, welcher eine Dateizugriffsanfrage anzeigt. In Zeile 40 kehrt die Funktion mit der ID des wartenden Prozesses zurück.

4.2.8 slcapset()

Der Kontrollprozess kann mit slcapset() dem Kern Capabilitys übemitteln. Dabei gibt er für die mit slcapget geholte Anfrage ein Flag zurück, welches anzeigt, ob der gewünschte Zugriff auf die Datei erlaubt ist oder nicht. Mit einem zusätzlichen Argument, welches z. Z. noch nicht ausgewertet wird, kann in späteren Implementierungen der Zugriff transparent auf eine andere Datei umgelenkt werden, wie dieses auch schon in [2] realisiert wurde.

```
 1: asmlinkage int sys_slcapset(int pid,int allow,struct slcap_request* buffer)
 2: {
 3: struct slcap_waitentry * t;
 4: int i;
 5:  slcapgetlock();
 6:  t=slcap_waitlist;
 7:  while ((t!=NULL)&&(pid!=t->pid)) t=t->next;
```

```
 8:  if (t==NULL) {
 9:   //request not fount in list!
10:   slcapreleaselock();
11:   return -1;
12:  };
13:
14:  switch (t->type) {
15:   case SLCAP_FILE:
16:    {
17:      if (!allow) ((struct slcap_filewaitentry *)t)->mask=0;
18:    };
19:    break;
20:   default:
21:    printk("SLCAP: Panic, unknown Type!!!!!\n");
22:  };
23:
24:  i=t->flags;
25:  t->flags|=SLCAPENTRYVALID;
26:
27:  slcapreleaselock();
28:  if ((i&SLCAPENTRYVALID)==0){
29:   atomic_dec(&waitingcount);
30:   wake_up(&slcap_wait_queue);
31:  };
32:
33:  return 0;
34: };
```

Zuerst wird der Eintrag in der Anfragewarteliste gesucht, welcher zu der übergebenen Prozess-ID passt. Anschließend wird, wie in der Funktion slcapget() typabhängig, das Ergebnis der Anfrage in den Wartelisteneintrag kopiert. In Zeile 25 wird der Eintrag als beantwortet gekennzeichnet. Abschließend wird ab Zeile 28 der auf die Anfrage wartende Prozess aufgeweckt.

4.2.9 slcapcache()

Zur Steigerung der Performance ist im Kern ein Capability-Cache eingerichtet. Dieser ermöglicht es dem Kontrollprozess Anfragen und Anworten, die in der Zukunft auftreten werden, bereits im Kern zwischenzuspeichern, indem er mehrfach slcapcache() mit den vollständigen Dateinamen und den Rechten zukünftiger Anfragen aufruft. Da nun die folgenden Anfragen aus dem Cache beantwortet werden können, wird jedesmal ein Prozesswechsel und damit sehr viel Zeit gespart. Im praktischen Einsatz merkt sich der Kontrollprozess die Dateien, auf welche ein Programm zugreift, und füllt beim ersten Blockieren den Cache mit, im Idealfall, allen zukünftigen Anfragen, wodurch das Programm nur einmal unterbrochen werden muss. Es ist nicht sichergestellt, dass ein Eintrag im Cache die gesamte Programmlaufzeit überdauert. Der Cache kann bei Speichermangel geleert werden. Anstatt des kompletten Dateinamens kann dieser auch mit einem Stern „$*$“ beendet werden. Das erweitert die Capability auf alle Dateinamen, die bis zu dieser Stelle mit dem Muster übereinstimmen. Man kann diese Notation nutzen, um ganze Unterverzeichnisbäume mit einem Eintrag abzudecken. Sollte der Kern für eine Anfrage keinen Eintrag im Capability-Cache finden oder die gefundenen Rechte

nicht zum Erlauben der Anfrage ausreichen, wird der Kontrollprozess mittels slcapget()/slcapset() gefragt.

```
asmlinkage int sys_slcapcache(int pid,int mask,char* buffer)
{
struct slcap_capentry * t;
int i;
 slcapgetlock();

 for (i=0;buffer[i]!=0;i++);
 t=(struct slcap_capentry *)kmalloc(
    sizeof(struct slcap_capentry)+i,GFP_KERNEL);
 if (t==NULL) return -1;
 for (i=0;buffer[i]!=0;i++) t->filename[i]=buffer[i];
 t->filename[i]=0;
 t->flags=SLCAPENTRYVALID;
 t->mask=mask;
 t->pid=pid;
 t->next=capentrycachelist;
 capentrycachelist=t;
 slcapreleaselock();
 return 0;
};
```

Die Funktion reserviert sich im Speicher einen Bereich und kopiert die übergebenen Daten hinein. Danach wird dieser Eintrag in die Cacheliste eingehängt.

4.2.10 slcapinfo()

Diese Funktion liefert Informationen über einen überwachten Prozess. Zur Zeit ist dieses nur der Pfadname der aktuell ausgeführten Binärdatei. In späteren Implementierung können hier weitere, für die Rechtevergabe wichtige Informationen übergeben werden, z. B. der Nutzer, der das Programm startete.

```
asmlinkage int sys_slcapinfo(int pid,int len,char* buffer){
 struct task_struct *t=find_task_by_pid(pid);
 if (t==NULL) return -1;
 return slcapgeneratefilename(t->slcap_id.file->f_dentry,len,buffer);
};
```

4.3 Hilfsprogramme

4.3.1 slcaprun

Dieses Programm erlaubt, andere Programme in einem Capability-Kontext zu starten. Der Aufruf erfolgt durch

slcaprun programm parameter ...

Hierbei wird zuerst mit slcapinit() der Capability-Kontext vorbereitet und danach mit exec() das Programm gestartet. Durch exec() wird auch der Capability-Kontext aktiviert.

4.3.2 capcontrol

Dieses Programm ist eine einfache Implementierung eines Kontrollprozess. Es verwaltet im Unterverzeichnis testlist des aktuellen Verzeichnisses Capabilitylisten für Programme. Der Dateiname ist hierbei das Ergebnis einer auf der Hashsumme MD5 basierenden Einwegfunktion über das zu überwachende Programm. In der Datei stehen zeilenweise Capabilitys für je eine Datei oder einen Verzeichnisbaum. In der Zeile stehen durch Leerzeichen oder Tabulatoren getrennt der absolute Dateiname, die „positiven“ und „negativen“ Rechte. Wenn der Dateiname mit einem Stern „*“ endet, bedeutet das, dass die Rechte für alle Dateien gelten, die bis zum Stern mit dem Muster übereinstimmen. Die nach dem Namen angegebenen Rechte sind Zahlen, die sich aus der Summe der zu vergebenden Rechte Execute 1, Write 2, Read 4 und Create 8 ergeben. Ein positives Recht heißt, dass das Kontrollprogramm dem überwachten Prozess die angegebenen Rechte einräumt. Ein negatives Recht bedeutet, dass das Kontrollprogramm ohne Benutzerinteraktion den Zugriff mit diesen Rechten nicht erlaubt. Ist für eine Datei weder ein positives noch ein negatives Recht angegeben, wird der Anwender gefragt und kann den Zugriff erlauben oder nicht. Zusätzlich kann er das positive Recht in die Capabilityliste eintragen lassen und so persistent machen. Anstatt eines negativen Rechts kann ein „p“ angegeben werden, welches bedeutet, dass das positive Recht in den Capability-Cache des Kerns geschrieben wird, sobald das Programm das erste Mal blockiert.

4.4 Zusammenspiel

Im Folgenden wird das Zusammenspiel der bisher beschriebenen Komponenten am Beispiel eines open()-Aufrufes beschrieben.
Das Programm caprun ruft die Funktion slcapinit() auf und bereitet den Capability-Kontext vor. Danach wird mit exec() das zu überwachende Programm gestartet und der Kontext aktiviert.

1. Bei jedem open()-Aufruf wird von der Kernfunktion sys_open() die Funktion open_namei() aufgerufen. Diese überprüft mit permission() ob der Nutzer das Recht hat, die Datei zu öffnen.

2. Zusätzlich wird nun noch mit slcappermission() eine weitere Überprüfung vorgenommen.

3. slcappermission() sucht zuerst im Cache nach einem Eintrag für die Datei.

4. Wenn die Funktion keinen Eintrag findet, wird eine Anfrage in die Anfragewarteschlange eingestellt, und der Prozess schläft, bis diese Anfrage beantwortet wurde. Der Kontrollprozess schläft mit slcapwait(), bis sich eine Anfrage in der Warteschlange befindet und wird nun aufgeweckt.

5. Er ruft slcapget() auf, um den ersten Eintrag aus der Warteschlange in einen Puffer zu kopieren. Danach kann der Kontrollprozess über die Rechtmäßigkeit des Dateizugriffs entscheiden. Dazu stehen ihm der Dateiname, die gewünschten Zugriffsrechte und die Prozessnummer des überwachten Programms zur Verfügung. Aus dem Proc-Dateisystem des Linuxkerns und mit der Funktion slcapinfo() kann sich der Kontrollprozess nun weitere Informationen über das Programm holen.

6. Die Entscheidung teilt er dann dem Kern mit der Funktion slcapset() mit,

7. welche dieser nun in die wartende Anfrage einträgt und den schlafenden Nutzerprozess aufweckt.

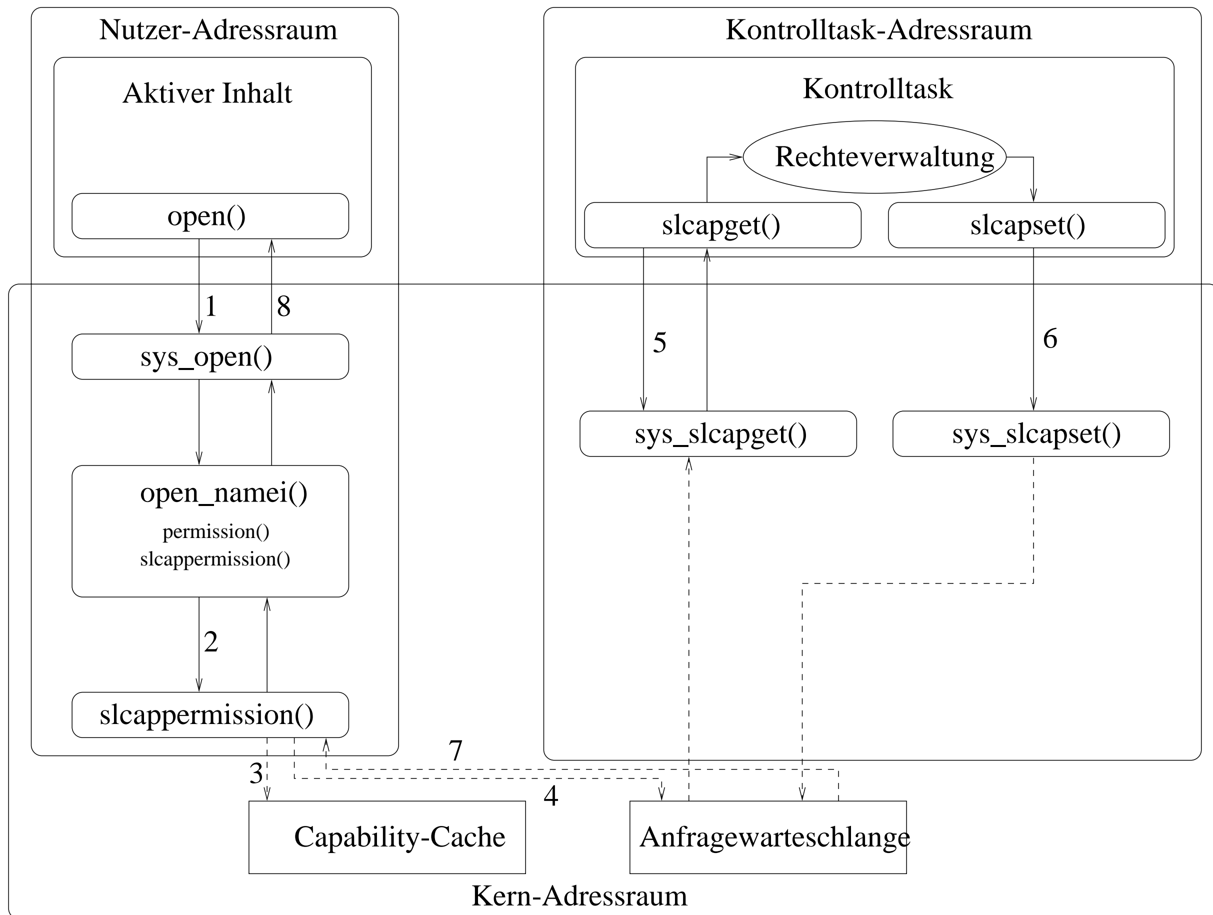

Abbildung 5: *Ablauf eines open()-Systemcalls*

Der Nutzerprozess entfernt den Eintrag aus der Warteschlange und gibt nun im positiven Fall einen Filedeskriptor als Ergebnis des open()-Aufrufes zurück oder bei nicht genügender Berechtigung einen Fehlercode.

5 Messungen und Bewertung

5.1 Messanordnung

Alle Messungen werden auf einem Rechner mit Mobile Pentium II 366 MHz und 64 MB Speicher durchgeführt. Gemessen wird mit dem Programm /usr/bin/time, welches die Laufzeit eines als Parameter angegebenen Programms ausgibt. Der Aufruf sieht folgendermaßen aus:

time slcaprun testprogramm

bzw.

time testprogramm

Als Testprogramm wird eine Schleife aufgerufen, die die untersuchte Funktion 100 000 mal aufruft. Jede Messung wird 10 mal wiederholt. Die Vergleichsmessungen werden auf dem gleichen Rechner und gleichen Linuxkern ohne Aktivierung der Capabilys gemacht. Um den Einfluss der Berechnung der Hashsumme zu untersuchen, wurden zwei Versionen des Testprogramms erstellt. Eines ist 5044 Byte groß, das andere wurde durch ein statisches Array auf 8393884 Byte vergrößert. Bei jeder Messung erfolgen also folgende 4 Durchläufe:

Durchlauf	Aktivierte Capalilitys	Größe desProgramms
1	nein	5044 Byte
2	nein	8393884 Byte
3	ja	5044 Byte
4	ja	8393884 Byte

Tabelle 3: Beschreibung der 4 Durchläufe pro Testfall

5.2 Messungen

5.2.1 Erstes open()

Hier wird die Dauer eines ersten open() Aufrufes gemessen. Das Testprogramm macht hier zu 100 000[17] mal die Datei /tmp/test auf und schließt sie sofort wieder. Der Kontrollprozess ist so modifiziert, dass er bei jeder Überprüfung der Zugriffsrechte die Hashsumme über das ausgeführte Programm neu berechnet, wie er es sonst nur das erste Mal nach einem exec() macht. Die gemessenen Zeiten sind sehr lang. Dieses ist verständlich, da hier eine Prozessumschaltung und das Berechnen der Hashsumme über die gesamte Datei erfolgen muss. Dieses Berechnen der Hashsumme erfolgt jedoch nur einmal im Programmlauf.

[17] Der Durchlauf des 8 Megabyte großen Programms mit aktivierten Capabilitys wurden nur 100 gemessen und dann die Zahl mit 1000 multipliziert.

Aktivierte Capalilitys	Größe des Programms	Aufrufdauer von 100 000 Systemcalls	Aufrufdauer eines Systemcalls
nein	5044 Byte	0,736 s	7,4 μs
nein	8393884 Byte	0,719 s	7,2 μs
ja	5044 Byte	122,83 s	1,2 ms
ja	8393884 Byte	86310 s	863,1 ms

Tabelle 4: Messergebnisse für erstes open()

5.2.2 Weiteres open() ohne Cachebenutzung

Bei diesem Test wird der normale Kontrollprozess bei 100 000 Durchläufen eingesetzt. Dieser berechnet nur beim ersten open() die Hashsumme und speichert sie für die folgenden 99 999 Aufrufe. Die Zeit für die Hashberechnung kann also vernachlässigt werden, wie man auch an der wesentlich geringeren Differenz der zwei Messungen mit aktivierten Capabilitys sieht.

Aktivierte Capalilitys	Größe des Programms	Aufrufdauer von 100 000 Systemcalls	Aufrufdauer eines Systemcalls
nein	5044 Byte	0,736 s	7,4 μs
nein	8393884 Byte	0,719 s	7,2 μs
ja	5044 Byte	16,713 s	167,1 μs
ja	8393884 Byte	17,629 s	176,3 μs

Tabelle 5: Messergebnisse für weiteres open() ohne Cachebenutzung

5.2.3 Weiteres open() mit Cachebenutzung

Bei diesem Durchlauf kommt der Cache zum Einsatz. Wieder wird 100 000 mal die Datei geöffnet und geschlossen. Diesmal wird der Kontrollprozess beim ersten open() die Hashsumme berechnen und das Leserecht auf die Datei /tmp/test in den Capabilitycache eintragen. Die nachfolgenden open()-Aufrufe können nun vom Kern ohne Prozessumschaltung beantwortet werden. Durch die höhere Geschwindigkeit macht sich hier auch wieder die Berechnung der Hashsumme bemerkbar.

Aktivierte Capalilitys	Größe des Programms	Aufrufdauer von 100 000 Systemcalls	Aufrufdauer eines Systemcalls
nein	5044 Byte	0,736 s	7,4 μs
nein	8393884 Byte	0,719 s	7,2 μs
ja	5044 Byte	0,875 s	8,8 μs
ja	8393884 Byte	1,633 s	16,3 μs

Tabelle 6: Messergebnisse für weiteres open() mit Cachebenutzung

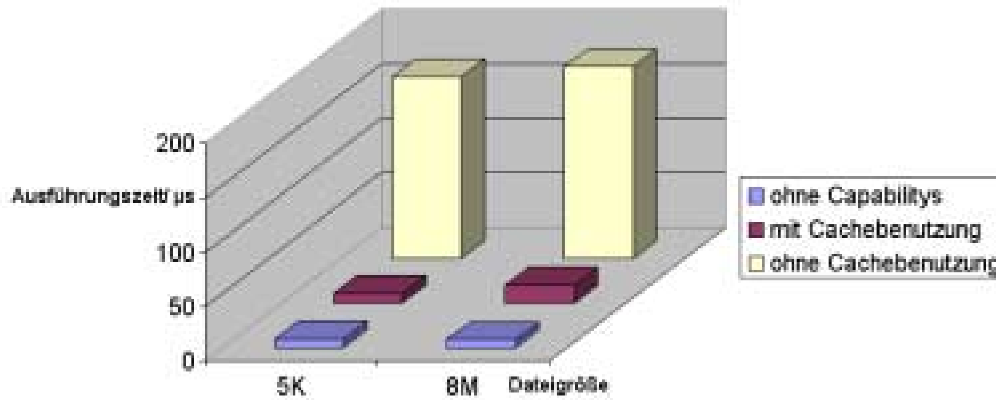

Abbildung 6: *Messergebnisse des open()-Aufrufes*

5.2.4 Praktischer Test mit rgrep

Um die Leistungsfähigkeit der Implementierung zu zeigen, wurde der folgende Programmaufruf gewählt:

time rgrep „slcap“ linux/

Dieser durchsucht den gesamten Linuxquellcode nach dem Auftreten der Zeichenkette „slcap“. Dabei werden 9420 Dateien mit 128 Megabyte Daten durchsucht.

Aktivierte Capalilitys	Dauer eines Programmdurchlaufes
nein	75,8 s
ja	76,1 s

Tabelle 7: Messergebnisse eines rgrep über den Linuxquellcode

5.3 Bewertung

Aus diesen Messergebnissen wird ersichtlich, dass die Nutzung von Capabilitys einen einmaligen Mehraufwand von einigen Millisekunden für den ersten Aufruf des Systemcalls open() und danach im häufigst auftretenden Fall, der Capabilityüberprüfung mit Cachetreffer, einen Verlust von 18 % der Aufrufzeit bei jedem weiteren open() kostet. Dieses ist im Vergleich zur gewonnenen Sicherheit eine minimale Verschlechterung der Ausführungszeit. Dieses zeigt auch der Test mit rgrep, welcher trotz intensiver Ein- und Ausgabe nur 0,4 % langsamer ist, wenn der Capabilitykontext aktiviert ist.

6 Zusammenfassung und Ausblick

Die in diesem Beleg entwickelte Erweiterung des Linuxkerns ist ein wirkungsvolles Werkzeug zur Absicherung eines Rechners. Programme können ohne nennenswerte Geschwindigkeitseinbußen in

einem Capability-Kontext ausgeführt werden, in dem alle Zugriffe auf das Dateisystem und die Hardware überwacht und einzeln erlaubt werden können. Das System läuft so stabil, dass in den 2 Monaten des Tests und der Dokumentation kein Absturz aufgetreten ist, obwohl der Autor auf einem System mit aktivierten Capabilitys arbeitete. Dieses liegt nicht zuletzt daran, dass sich die Änderungen im Quellcode des Linuxkerns auf ein Minimum beschränkten, insgesamt wurden nur 350 Zeilen Code geändert. Die eigentliche Entscheidung über die Rechtmäßigkeit des Zugriffs wurde in ein Programm im Nutzeradressraum ausgelagert, wodurch sich die Möglichkeit ergibt, diese sehr einfach zu ändern, beispielsweise für Erweiterungen oder Änderungen der Rechtepolitik oder zur Fehlersuche.
Der Autor hofft, dass die Capabilitys schnell in den Entwicklerkern integriert werden und so Verbreitung finden. Dadurch kann die Erweiterung in großem Rahmen getestet, weiterentwickelt und zur Anwendungsreife gebracht werden. Zur Förderung dieser Entwicklung wurde eine Webseite eingrichtet, welche unter http://prak.org/slcap/ neben der aktuellen Version des Kernelpatchs und dieses Papiers auch eine Mailingliste zur Diskusion enthält.
Schwerpunkte für die weitere Arbeit sind dabei:

- Erweiterung der Capabilitys auf die Überwachung von Sockets. Der aktuelle Patch enthält bereits die Definition einer Schnittstelle zur Übertragung der Anfragen vom Kern zum Kontrollprozess. Diese muss nun implementiert und getestet werden.

- Änderung der Kernschnittstelle zum Betrieb von mehreren Kontrollprozessen. Jedem Programm kann dadurch ein eigener Kontrollprozess zugeordnet oder Kontrollprozesse selbst von einem übergeordneten Prozess kontrolliert werden.

- Eine Zusammenführung der in diesem Beleg entwickelten Capabilitys mit den bereits im Kern integrierten Capabilitys, welche den Zugriff auf bestimmte Kernfunktionen, wie das Setzen der Uhrzeit steuern. Dadurch wird eine einheitliche Schnittstelle geschaffen, die Rechte eines Prozesses auf dem System einzugrenzen.

- Der Kontrollprozess, welcher zur Messung und zum Test verwendet wurde, muss erweitert werden. Zum Einen ist eine benutzbare Bedienoberfläche zu schaffen und zum Anderen die Funktionalität zu erweitern, z. B. die Implementierung von den in [2] beschriebenen signierten Wish- und Trustlisten.

Literatur

[1] distributed.net. Client download. http://distributed.net/download/clients.html.

[2] Hermann Härtig and Lars Reuther. Encapsulating mobile objects. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, pages 355–362, Baltimore, Md., Mai 1997.

[3] Philippe Biondi Huagang Xie and Steve Bremer. Linux intrusion detection system. http://www.lids.org.

[4] Sebastian Lehmann. Capabilitys for linux. http://prak.org/slcap/, Februar 2001.

[5] G. McGraw and E. Felten. Java security: Hostile applets, 1997.

[6] Sun Microsystems. Java technology. http://www.sun.com/java/.

[7] Andreas Pfitzmann. Sicherheit in Rechnernetzen. Script, Oktober 1999.

[8] K. Pommerening. Datenschutz und Datensicherheit, 1991.

[9] SecurityFocus.com. Multiple vendor wu-ftpd buffer overflow vulnerability, August 2000. http://www.securityfocus.com/frames?content=/vdb/bottom.html%3Fvid%3D1387.

[10] Andrew S. Tannenbaum. Verteilte Betriebssysteme, 1995.