Kapselung ausführbarer Binärdateien
slcaps: Implementierung von Capabilitys für Linux

Sebastian Lehmann* und Andreas Westfeld

Technische Universität Dresden
Institut für Systemarchitektur
01062 Dresden
sepp@prak.org, www.prak.org/slcap/
westfeld@inf.tu-dresden.de

Zusammenfassung In Internet-Dokumente eingebettete Programme ermöglichen größere Dynamik und Flexibilität als statische Dokumente. Ausführbare Binärdateien aus unsicherer Quelle haben jedoch unzulänglich geregelten Zugriff auf sensible Daten. Viren und Trojanische Pferde zeigen, dass aktive Inhalte sogar die Systemkonfiguration verändern können. Fremde Programme aus unsicherer Quelle dürfen demzufolge nur sehr eingeschränkte Zugriffsrechte bekommen. Wir beschreiben einen für Linux implementierten Mechanismus, mit dem fremde Programme gekapselt, d. h. mit differenzierten Zugriffsrechten auf Dateisystem und Netzwerk ausgeführt werden.

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

* Sebastian Lehmann studiert Informatik an der Technischen Universität Dresden

> and installing the client on any machine implies understanding and agreement with these terms."

Dieses Beispiel zeigt die Bedrohung durch Programme unbekannter Herkunft, welche aus dem Internet geladen, per E-Mail zugeschickt oder von Freunden auf selbstgebrannter CD weitergereicht werden. Wenn wir solche Programme auf Desktop-Systemen starten, können wir oft nur hoffen, dass sie wirklich nur das tun, was sie vorgeben, und nicht im Hintergrund auf dem Rechner Daten verändern oder löschen. Die beiden in Deutschland meist genutzten Betriebssysteme Windows und Linux haben keine Schutzmechanismen vor solchen trojanischen Pferden. In diesem Beitrag soll eine einfache Erweiterung des Linuxkerns vorgestellt werden, mit deren Hilfe jedes Programm in einer gesicherten Umgebung, einem „Sandkasten", gestartet werden kann. Alle Zugriffe auf die Daten und die Hardware des Rechners werden damit überwacht, protokolliert und gesteuert. Eine ähnliche Erweiterung für Windowsprogramme wird in [4] vorgestellt.

2 Bekannte Ansätze

Zur Festlegung der Grenze zwischen aktivem Inhalt und dem umliegenden System gibt es mehrere Ansätze. Wir wollen zunächst die verbreitetsten Varianten zur Abgrenzung vorstellen. Sie unterscheiden sich hauptsächlich darin, 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.

2.1 Interpreter

Dieser Ansatz implementiert die Sicherheit in einer Trennschicht zwischen der Hardware und dem Programm aus unsicherer Quelle. Ein Beispiel für diese Strategie ist eine Java Virtual Machine[7], welche Klassendateien mit Java-Bytecode interpretiert. Diese Methode hat den Vorteil, 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. Immer, wenn ein solches Konstrukt auftritt, kann der Interpreter Rechte nach bestimmten Kriterien vergeben, z. B. abhängig vom Ursprung des Programms[1],

[1] Viele Javainterpreter unterscheiden hier Programme, die aus dem Internet geladen wurden und Programme, die von der lokalen Festplatte stammen.

einer digitalen Signatur des Programms oder sogar dem Zustand von bestimmten Variablen[2][6].

Leider müssen diese Kontrollmechanismen für jeden Interpreter neu geschrieben werden. Neben Leistungseinbußen könnte sich das auch nachteilig auf die Qualität der verschiedenen Implementierungen auswirken und kostet schon allein bei der Konfiguration der verschiedenen Interpreter mehr Aufwand.

2.2 Prozesse

Ein anderer Ansatz, bei dem der aktive Inhalt direkt als Prozess auf dem Prozessor gestartet wird, zieht die Grenze zwischen dem zu überwachenden und dem überwachten System direkt um den virtuellen Adressraum. Das hat Vorteile:

- 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 z. B. auf einem Linuxsystem ausgeführt wurden, können so überwacht werden.
- Es gibt z. B. nur 12 Systemcalls[3], welche direkten Zugriff auf Dateien erlauben. Diese benutzen alle eine von **drei Kernfunktionen** zur Rechteüberprüfung: open_namei( ), may_create( ) und may_delete( ). Der Aufwand für die Implementierung der zusätzlichen Rechteüberwachung ist also begrenzt, wodurch auch die Fehleranfälligkeit geringer ist.
- Unter Linux erfolgt der Zugriff auf die Hardware durch Zugriff auf Gerätedateien[4]. 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 vergleichbares Konstrukt, den Socket. Auch hier kann die Rechteüberprüfung an zentraler

[2] 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 Autorisierung eines Dateizugriffs genutzt wird.

[3] Version 2.4.0 des Linuxkerns

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

Stelle bei den Systemcalls connect( ), bind( ) und accept( ) erfolgen und nicht bei jedem read( ) und write( )-Aufruf.

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 können wir diesen Mechanismus nutzen und müssen auch hier nur an einer Stelle die Rechte überprüfen.

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 Überwachung der Systemaufrufe kann natürlich nicht den Inhalt von Dateien beurteilen, die ein Programm aus unsicherer Quelle – z. B. ein Compiler – schreiben darf. Wir müssen also sehr sorgfältig überlegen, bevor wir ein frisch erzeugtes Programm aufrufen, denn es könnte sich ja um ein Trojanisches Pferd handeln.

Verdeckte Kanäle können auch nicht durch diese Art der Überwachung ausgeschlossen werden. Wir müssen also davon ausgehen, dass zwei unsichere Programme auf dem gleichen System sind und miteinander auf obskure Weise kommunizieren.

Beim Einsatz eines Interpreters können wir die Granularität der Rechtevergabe beliebig klein wählen, z. B. kann der Zugriff auf eine Datei nur einer speziellen Funktion erlaubt sein, die nun die Einheit ist, deren Rechte überwacht werden.

2.3 Zugriffskontrolle

Eine einfache Möglichkeit der Zugriffskontrolle ist die Schutzmatrix. In ihr werden Subjekte (z. B. Nutzer) und Objekte (z. B. Dateien) in Beziehung zueinander gesetzt. Eine Schutzmatrix enthält z. B. für jedes Subjekt eine Spalte und für jedes Objekt eine Zeile. Die Felder der Matrix enthalten die erlaubten Operationen, die das betreffende Subjekt über dem Objekt ausführen darf (seine Rechte).

Dieses allgemeine Schema hat allerdings ein Problem: Die Matrix nimmt sehr viel Platz in Anspruch und ist größtenteils leer.

Tabelle 1. Beispiel für eine Schutzmatrix

	andi	betti	carl ...
vis.txt	rw	r	—
/bin/ed	x	x	x
carl.txt	—	—	rw

r – read (lesen)
w – write (schreiben)
x – execute (ausführen)

Zwei alternative Abspeicherungsformen lösen das Problem: Wir speichern entweder die Spalten oder die Zeilen als Liste und lassen die leeren Felder weg [9].

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 z. B. ein bestimmter Benutzer[5] lesend oder schreibend zugreifen darf. Das ist eine sehr grobe, aber einfach zu administrierende Rechteverwaltung, solange es genügt, nur auf Benutzerebene die Rechte zu vergeben. Soll die Granularität bis auf Programmebene verfeinert werden, steigt der Aufwand sehr schnell. Hier müsste nun für jedes Programm ein eigener Benutzer angelegt werden, in dessen Namen das Programm gestartet wird[6]. Eine Erweiterung dieser Rechtevergabe ist „LIDS“[7]. Mit dieser Erweiterung kann der Zugriff auf ganze Unterverzeichnisbäume mittels einer Regel festgelegt werden. Programmen können die Rechte damit auch ohne den Umweg über die Neuanlage von Nutzern zugewiesen werden. LIDS vereinfacht also die Administration von Zugriffsrechten oder macht sie überhaupt erst praktikabel.

Capabilitys Capabilitylisten[8] beschreiben z. B. für jedes Programm die Betriebsmittel, auf die der Zugriff erlaubt ist. Die meisten Programme,

[5] und damit auch Prozesse, die von diesem Nutzer gestartet wurden
[6] Soll jede Kombination aus Programm und Nutzer möglich sein, kann es bis zu Anzahl der Nutzer mal Anzahl der Programme verschiedene „virtuelle“ Nutzer geben.
[7] Linux Intrusion Detection System, siehe auch [3]
[8] Der Begriff Capability wird im Linuxkern bereits für ein Bitfeld verwendet, welches den Zugriff eines Prozesses auf administrative Systemfunktionen wie z. B. das Ändern von Zugriffsrechten mit chmod regelt. Hier bezieht sich das Wort Capability allgemein auf Betriebsmittel und im Folgenden speziell auf den Zugriff auf Dateien.

speziell aktive Inhalte auf Webseiten, benötigen nur sehr wenige Betriebsmittel. 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.

Wunsch- und Vertrauenslisten Reuther und Härtig stellten eine Technik vor [2], bei der Programme vor ihrer Ausführung kundgeben, welche Betriebsmittel sie zur Erledigung der versprochenen Funktionalität benötigen. Anhand einer Vertrauensliste des Systems wird überprüft, ob dem Programm in Bezug auf die angeforderten Betriebsmittel Vertrauen entgegengebracht wird. Bei Programmstart entsteht aus der Wunsch- und Vertrauensliste eine Capabilityliste, die alle dem Programm zugänglichen Betriebsmittel enthält und deren Einhaltung durch die Umgebung des ausführenden Prozesses durchgesetzt wird.

3 Anwendungsmöglichkeiten

Dieser Abschnitt nennt einige Beispiele, wie der Einsatz von Capabilitys die Systemsicherheit verbessern kann. Dabei wird auf Mängel in existierenden Systemen eingegangen.

3.1 Programme unbekannter Herkunft

Ein Programm, welches unter Linux gestartet wird, hat alle Rechte, die der startende Nutzer hat[9]. 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 muss darauf vertrauen, dass der Programmierer keine

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

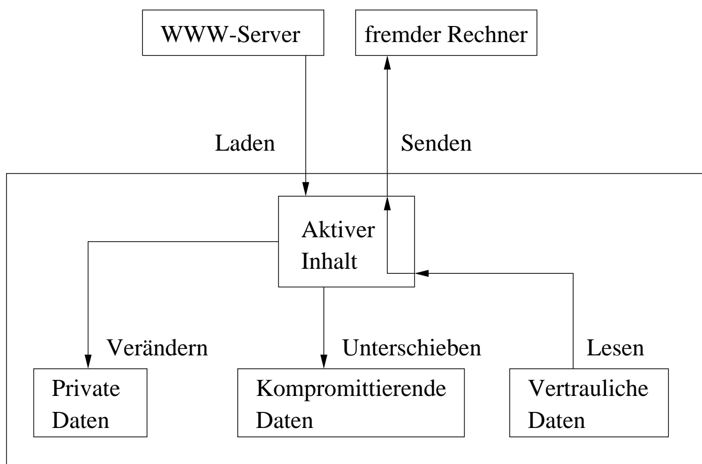

Abbildung 1. Gefahren durch aktive Inhalte

Schadensroutinen in das Programm eingebaut hat. Diese Befürchtung ist bei bekannten Herstellern, wie beispielsweise Corel, Netscape oder Real, nicht unbegründet: Deren Programme senden Daten über den Nutzer zum Hersteller. Beispielsweise sendet der Netscape-Browser durch das „Smart Browsing“ die URLs der vom Nutzer aufgerufenen Webseiten an Netscape. Bei Programmen unbekannter Herkunft hat der Anwender keine Kontrolle, welche Daten von dem Programm verarbeitet und verändert werden.

Die Kontrolle kann durch den Einsatz von Capabilitylisten wiedererlangt werden. Am Anfang startet das Programm wahlweise mit einer leeren oder einer vorbereiteten Capabilityliste, welche ungefährliche Zugriffe erlaubt (beispielsweise das Linken dynamischer Bibliotheken oder das Anlegen von Dateien im Verzeichnis /tmp). Alle nicht in der Liste erlaubten Zugriffe auf Daten werden abgefangen. Der Anwender wird informiert und kann seine Zustimmung zu diesen geben. Gleichzeitig erhält er so einen Überblick über die Daten, die das unbekannte Programm benutzt.

3.2 Serverprozesse

Serverprozesse sind oft sehr komplexe Programme. Es ist nicht auszuschließen, dass deren Implementierung fehlerhaft ist. Durch einen bei Hackern beliebten Fehler, der zwar vermeidbar und bekannt ist, aber immer wieder in neuer Software zu finden ist, kann fremder Programmcode ausgeführt werden.

Das Programm „WU-FTPD“ (FTP-Daemon, an der Washington University entwickelt) soll hier als Beispiel dienen. Im Abstand weniger Monate (zuletzt im August 2000) werden Sicherheitslücken in diesem FTP-Server entdeckt[8]. Unter bestimmten Bedingungen kann der Angreifer einen Pufferüberlauf provozieren. Durch den Überlauf können Daten außerhalb des vorgesehenen Puffers wie z. B. Rücksprungadressen von Funktionsaufrufen auf dem Stapel geändert werden.

Zum Beispiel kann beim FTP-Server der Login-Name Binärcode enthalten. ausgeführt wird. Werden statt des Login-Namens nun die Bytes eines Programms samt einer geeigneten Rücksprungadresse gesendet, dann wird der Funktionsaufruf im FTP-Server nicht durch Rückkehr zur aufrufenden Stelle beendet, sondern durch Aufruf des eingeschleusten Programms (siehe Abbildung 2). Wie in Abschnitt 3.1 beschrieben, hat dieser Code

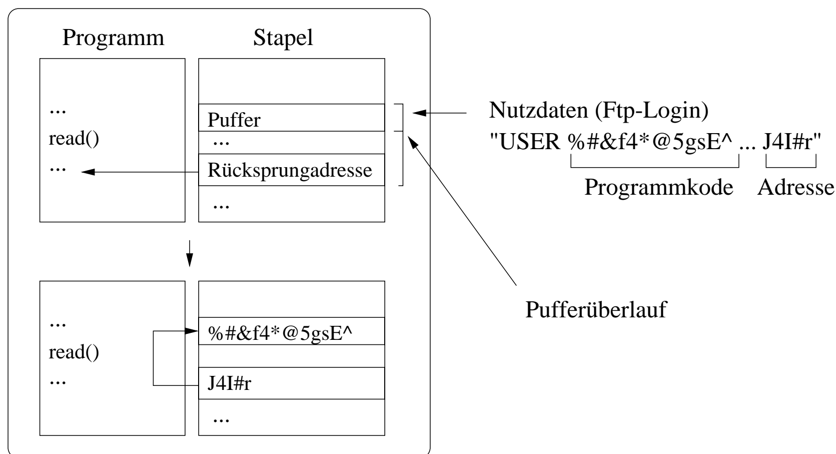

Abbildung 2. Einbruch durch Pufferüberlauf

nun die gleichen Rechte wie der Nutzer, in dessen Namen der Serverprozess läuft. Da viele Serverprozesse unter dem Root-Nutzer laufen, hat das fremde Programm unbeschränkten Zugriff auf alle Dateien und auch auf die Hardware. Ein Angreifer hat nun volle Kontrolle über den Rechner.

Durch die Anwendung von Capabilitys können wir zwar nicht den Überlauf eines Puffers verhindern, aber das eingeschleuste Programm hat nicht mehr die volle Kontrolle über den Rechner. Es kann nur noch im Kontext des Serverprozesses zugreifen, also nur auf Daten, die auch der Server-

prozess braucht. Im Beispiel des oben beschriebenen FTP-Servers kann das Angreiferprogramm also die Konfigurationsdateien des Servers und das FTP-Verzeichnis lesen und in das Uploadverzeichnis und das Logfile schreiben. Das bedeutet eine weit geringere 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.

Die Capabilityimplementierung im Linuxkern kann mit sehr geringem Aufwand bei fast allen Serverprozessen eingesetzt werden und so entscheidend zur Absicherung von Servern im Internet beitragen.

3.3 Sammeln von Beweisen

Mit dem System können auch Einbrüche festgestellt werden, da das Serverprogramm bei normaler Arbeit keine Systemdateien öffnen wird. Wenn ein Einbruch stattgefunden hat, werden abnormale Dateizugriffsversuche erfolgen. Dadurch kann der Administrator alarmiert oder der Server abgeschaltet werden.

Das System könnte so erweitert werden, dass verbotene Schreibzugriffe in einer Kopie simuliert werden. So wird der Einbruch nicht nur verhindert, sondern auch nachweisbar, wie er stattgefunden hätte. Die veränderte Kopie könnte gleichzeitig als Beweismittel gegen den Angreifer dienen.

4 Implementierung

Die Implementierung des Capability-Kontexts unter Linux hat zum Ziel, die Nutzerprozesse ohne nennenswerte Leistungseinbußen zu überwachen.

Damit die Änderungen im Kern möglichst gering ausfallen, wird die Rechtepolitik[10] von einem eigenen Programm im Nutzeradressraum durchgeführt. Diese Entscheidung wirkt sich jedoch nachteilig auf die Leistungsfähigkeit des Systems aus, da bei jeder Überprüfung der Capabilitys ein Prozesswechsel erfolgen muss. Dieser dauert unter Linux sehr viel länger als ein normaler Systemcall. Um die Verzögerung zu minimieren, wird ein Cache im Kern implementiert, in dem bereits beantwortete Anfragen[11] abgelegt werden können, so dass der Prozesswechsel und der damit verbundene Zeitverlust überflüssig werden. Die Zugriffskontrolle wird im Kern als Zusatz zur bisherigen Rechteverwaltung implementiert, das heißt,

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

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

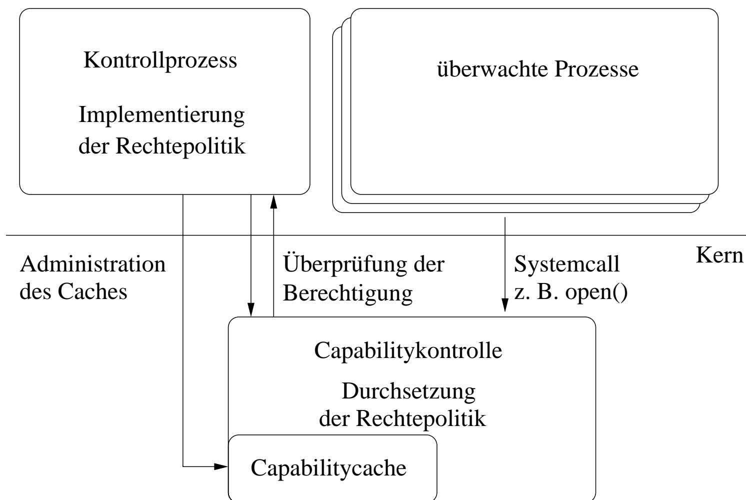

Abbildung 3. Implementierung der Capability-Kontrolle

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“[12] bzw. „SLCAP“ gewählt. Im Folgenden werden bei den Beschreibungen der Änderungen des Linuxkerns Quellcodezeilen zitiert. Dabei wurden die Fehlerbehandlungen nicht mit aufgeführt. Der vollständige Quellcode ist unter [5] zu finden.

4.1 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 Capabilitys eingefügt werden. Die Implementierung dieser neuen Systemcalls, die Änderungen vorhandener Kernfunktionen, sowie die Erweiterungen bisheriger Kerndatenstrukturen beschreiben die folgenden Abschnitte.

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

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

In diesem wird der Status des Prozesses bezüglich des Capability-Kontexts geführt. Wichtig sind zwei Flags, eines, welches anzeigt, ob der Capability-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 Eintrag slcap_id der Struktur task_struct wird

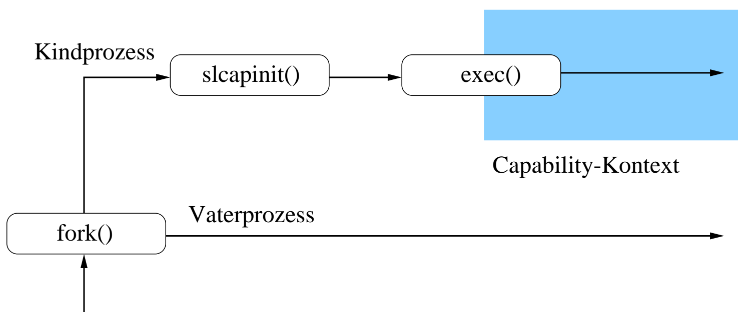

Abbildung 4. Start eines Programms in einem Capability-Kontext

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 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. 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( ).

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.

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 ausgeführten 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;
};
```

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

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); i++)
          if (slcapbuffer[i]==t->filename[i]) break;
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 Capability-Kontext 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[13]. 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

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

den vom Kontrollprozess zurückgegebenen Rechte verglichen und der Eintrag aus der Warteschlange genommen.

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.

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.

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:       // Kopieren der Anfrage von t nach r
18:       ...
```

```
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 von 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.

slcapset( ) Der Kontrollprozess kann mit slcapset( ) dem Kern Capabilitys übermitteln. 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.

slcapcache( ) Zur Steigerung der Performance ist im Kern ein Capability-Cache eingerichtet. Dieser ermöglicht es dem Kontrollprozess Anfragen und Antworten, 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.

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);
};
```

4.2 Hilfsprogramme

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.

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.

4.3 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.

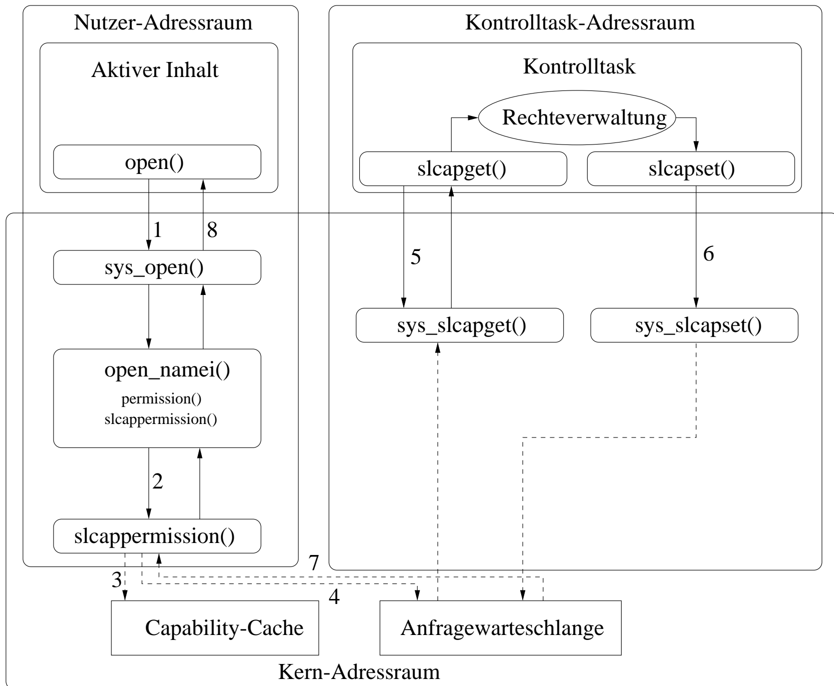

Abbildung 5. Ablauf eines open( )-Systemcalls

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

5 Messungen und Bewertung

Alle Messungen beziehen sich auf eine Plattform mit Mobile Pentium II 366 MHz und 64 MB Speicher.

Um den zeitlichen Einfluss der Berechnung der Hashsumme zu untersuchen, wurden zwei Testprogramme mit unterschiedlicher Größe verwendet (5 KB und 8 MB).

Wenn ein Programm erstmalig nach dem Start open( ) aufruft, so wird sein Hashwert gebildet. Dadurch ist dieser Aufruf um Größenordnungen langsamer und praktisch nur noch von der Dateigröße des Programms abhängig. Ohne Verwendung eines Caches im Kern muss für jeden weiteren Aufruf in den Nutzeradressraum gewechselt werden. Ein solcher Wechsel kostet das 20-fache eines normalen Systemaufrufs. Wird jedoch ein Cache

Tabelle 2. Leistungsmessung

open( )	Programmgröße	
	5 KB	8 MB
ohne Capabilitys	7 µs	7 µs
mit Capabilitys		
1. Aufruf	1200 µs	860 **ms**
weitere Aufrufe	170 µs	170 µs
mit Cache	8 µs	8 µs

im Kern verwendet, dann verlängert sich die Bearbeitungszeit für einen Systemaufruf je nach Programmgröße ungefähr ein- bis zweimal.

Abschließend sollen diese Messungen noch an einem praktischen Beispiel durchgeführt werden. Mit folgendem Aufruf durchsucht rgrep den gesamten Linux-Quellcode nach dem Auftreten der Zeichenkette „slcap".

```
time rgrep "slcap" linux/
```

Dabei werden 9420 Dateien mit 128 Megabyte Daten durchsucht. Die Messung wurde zweifach ausgeführt. Beim ersten Durchlauf wurden alle Dateien von Festplatte geladen. Entsprechend groß ist die Zeit, die für die Ein-/Ausgabeaktivität der Festplatte benötigt wird. Beim zweiten Durchlauf waren alle Dateien bereits zwischengespeichert und es erfolgte kein einziger Festplattenzugriff.

Tabelle 3. Messergebnisse eines rgrep über den Linux-Quellcode

Aktivierte	Dauer eines Programmdurchlaufes mit Daten	
Capabilitys	von Festplatte	aus dem Cache
nein	75,8 s	1,422 s
ja	76,1 s	1,465 s

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. Das ist im Vergleich zur gewonnenen Sicherheit eine minimale Verschlechterung der Ausführungszeit. Der Test mit rgrep ist trotz intensiver Ein- und Ausgabe nur 0,4–3,0 % langsamer, wenn der Capability-Kontext aktiviert ist.

6 Schlussfolgerung und Ausblick

Slcaps ist als Erweiterung des Linuxkerns 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 stabil; es hat in den Monaten des Tests und der Dokumentation aktivierten Capabilitys keinen Absturz erlebt. Das liegt nicht zuletzt daran, dass sich die Änderungen im Quellcode des Linuxkerns auf ein Minimum beschränken, insgesamt wurden nur 350 Zeilen Code geändert. Die eigentliche Entscheidung über eine Zugriffserlaubnis wurde in ein Programm im Nutzeradressraum ausgelagert. Somit lassen sich beispielsweise Erweiterungen oder Änderungen in der Rechtepolitik besonders einfach und schnell durchführen.

Schwerpunkte für die weitere Arbeit sind die

- 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 entwickelten Capabilitys mit den bereits im Kern integrierten Capabilitys, welche den Zugriff auf bestimmte Kernfunktionen (wie z. B. 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 Wunsch- und Vertrauenslisten.

Unter http://prak.org/slcap/ sind die aktuellen Quelltexte frei verfügbar.

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., 1997.
3. Philippe Biondi Huagang Xie and Steve Bremer. Linux intrusion detection system. http://www.lids.org.
4. Christian Kaiser. Sandkastenspiele. Windows-Anwendungen überwacht ausführen. Seiten 232 ff. in c't Magazin für Computertechnik 10/2001.
5. Sebastian Lehmann. Capabilitys for linux. 2001.
   http://prak.org/slcap/
6. G. McGraw and E. Felten. Java security: Hostile applets, 1997.
7. Sun Microsystems. Java technology. http://www.sun.com/java/
8. SecurityFocus.com. Multiple vendor wu-ftpd buffer overflow vulnerability, 2000.
   http://www.securityfocus.com/
   frames?content=/vdb/bottom.html%3Fvid%3D1387.
9. Andrew S. Tannenbaum. Verteilte Betriebssysteme, 1995.