Grosser Beleg

Sichere Webbrowser-Nutzung in DROPS

Stefan Kalkowski

28. September 2005

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Christian Helmuth

Inhalt dieser Arbeit ist die Entwicklung einer sicheren Umgebung für den Zugriff auf Ressourcen des World Wide Web. Ausgangspunkt hierfür ist das Betriebssystem DROPS. Es wird gezeigt, dass es möglich ist, mit Hilfe eines L4 basierten HTTP Proxy, der als eine Art Application Firewall fungiert, potentiell unsichere Webclients unmodifiziert für den Zugriff auf sensitive Informationen weiter zu verwenden und dabei Vertraulichkeit der Daten zu garantieren.

Inhaltsverzeichnis

1 Einführung **1**

2 Stand der Technik **3**
2.1 L4 Microkernel . . . 3
2.2 Linux spielt im Sandkasten . . . 4
2.3 Nizza Architektur . . . 5
2.4 Aufsplitten von Anwendungen . . . 6
2.5 Vertrauenswürdiger Fenstermanager DOpE . . . 7
2.6 Flips . . . 8
2.7 MatrixSSL . . . 8
2.8 Webbrowser . . . 8

3 Entwurf **10**
3.1 Architektur Übersicht . . . 11
3.1.1 Restriktion auf Domains . . . 12
3.1.2 Erwägungen zur Performance . . . 12
3.2 L4Linux Proxy Stub . . . 13
3.3 L4 HTTP Proxy . . . 15
3.4 L4Linux Application Loader . . . 16
3.5 Vertrauenswürdigkeit der L4Linuxanwendungen . . . 16

4 Implementierung **18**
4.1 Generischer Multi Threads Server . . . 18
4.2 L4 HTTP Proxy . . . 19
4.3 L4Linux Proxy Stub . . . 20
4.4 L4Linux Application Loader . . . 20

5 Auswertung **22**
5.1 Sicherheit . . . 22
5.2 Performance . . . 23

6 Zusammenfassung und Ausblick **25**

Glossar **27**

Literaturverzeichnis **28**

Abbildungsverzeichnis

2.1 Beispielhafte Nizza Konfiguration 5
2.2 Umfang der betrachteten Webbrowser 9

3.1 Architektur Überblick 11
3.2 SSL Handshake 14

5.1 SLOC der herkömmlichem Linuxumgebung im Vergleich zum entwickelten sicheren Szenario 22
5.2 Mittlere Aufrufzeit in Millisekunden für HTTP und HTTPS Pakete im Vergleich 23

1 Einführung

Monolithische Betriebssysteme wie z.B. Linux können aufgrund ihrer enormen Komplexität nicht als vertrauenswürdige Plattform angesehen werden. Die Korrektheit lässt sich bei derartigen Größenordnungen nur unzureichend verifizieren. Programme, die sicherheitskritische Funktionen erfüllen, müssen jedoch den ihnen zugrundeliegenden Bibliotheken und Diensten, wie unter anderem dem Linuxkern, vertrauen, um den Schutz sensitiver Daten garantieren zu können. Beispielsweise ist das Verschlüsselungsprogramm GnuPG darauf angewiesen, dass die privaten Schlüssel und Zertifikate sicher, d.h. von anderen Instanzen unbesehen und unmodifiziert auf der Platte hinterlegt werden und sein Programmcode nicht korrumpiert wird.
Ein ähnliches Problem stellt sich bei WWW-Browsern, welche auf sensible Webanwendungen wie E-Mail-Interfaces und Onlinebanking zugreifen. Einerseits muss hier die Korrektheit der Verbindungsverschlüsselung sichergestellt werden, andererseits müssen die erhaltenen und zu versendenden unverschlüsselten Daten vertraulich behandelt werden.
Der Fiasco Mikrokernel zusammen mit L4-Env bildet aufgrund der Minimalität und Überschaubarkeit, im Gegensatz zu monolithischen Systemen die ideale Basis, der solche Anwendungen (wie z.B Verschlüsselungsprogramme oder sichere Speicherfunktionen) vertrauen können. Zusammen mit L4Linux können parallel dazu weitverbreitete und in Bezug auf die Datensicherheit weniger anspruchsvolle Anwendungen auf demselben Rechner, zur selben Zeit ablaufen und über wohldefinierte Schnittstellen miteinander kommunizieren.
Ziel dieser Arbeit ist es auf Basis von Fiasco und bestehenden L4 Anwendungen einschließlich L4Linux, eine sichere Nutzungsmöglichkeit für Webbrowser zu entwerfen und umzusetzen. Hohe Priorität besitzt dabei die Nutzerfreundlichkeit der Lösung. Unter Nutzerfreundlichkeit sei hierbei vor allem zu verstehen, dass Anwender und Anwenderin sich so wenig wie möglich auf eine neue Handhabung der Software umstellen muss, d.h. es dürfen keine komplizierten Befehlsfolgen abverlangt werden, nur damit auf eine verschlüsselt übertragene Webseite zugegriffen werden kann. Ebenso sollte die Vielzahl an Funktionen heutiger Webclients, wie z.B. Möglichkeiten multimedialer Repräsentationen, weiterhin zur Verfügung stehen. Optimal wäre eine Lösung, die es erlaubt, weiterhin den jeweiligen „Lieblings“-Browser zu verwenden. Der Hintergrund für diese Forderungen ist, dass die meisten AnwenderInnen im Zweifel den einfachen Weg wählen, ihren alten Webbrowser zu verwenden und eine ungewohnte Applikation ungenutzt lassen, zumindest solange ihnen die sicherheitstechnische Einsicht in die Notwendigkeit einer Umstellung fehlt.

Ziele der Belegarbeit:

1. Aufstellen möglicher Angriffsziele in Bezug auf WWW-Browser.
2. Herausfiltrieren von datensicherheitsrelevanten Browser- und/oder Betriebssystemteilen.
3. Möglichkeiten der Kapselung der sicherheitskritischen Teile und anschliessender Portierung auf L4 erarbeiten.
4. Wahrung des grösstmöglichen Nutzerkomforts, d.h. insbesondere, dass idealerweise der Browser weiterhin frei wählbar sein sollte, oder zumindest ein verbreiteter Browser adaptiert wird.
5. Aus Punkt 4 ergibt sich die Notwendigkeit, sowenig Modifikationen wie möglich an der Ausgangssoftware zu vollziehen.
6. Entwurf und Implementation einer Lösung.

Die vorliegende Arbeit beweist, dass es prinzipiell möglich ist, unmodifizierte und potentiell unsichere Webbrowser einzusetzen, und dennoch Vertraulichkeit und Integrität sensitiver Informationen, die durch diese übertragen und angezeigt werden, zu bewahren.
Die Arbeit gliedert sich in folgende Bereiche: Im ersten Teil werden bestehende Konzepte, die in der Entwicklung aufgegriffen wurden, ausführlich erläutert. Kapitel 3 gibt einen groben Einblick in den Entwurf, wie er vor Beginn der Implementation erarbeitet wurde. Darauffolgend werden Änderungen, sowie Tücken während der Umsetzungsphase erklärt. Eine Bewertung der Performance und Sicherheitseigenschaften der vorgestellten Anwendung erfolgt in Kapitel 5. Abschliessend werden mögliche Verbesserungen bzw. fehlende Teile diskutiert.

2 Stand der Technik

Die Komplexität von Rechnersystemen ist in den vergangenen Jahren ständig angewachsen. Das gilt sowohl für die einzelne Maschine und die auf ihr eingesetzten Software, wie für die Kommunikationsbeziehungen der Systeme untereinander, beispielsweise durch Integration verschiedener, vormals voneinander unabhängig genutzter Protokolle. Eine zunehmende Komplexität bedeutet jedoch auch eine steigende Fehlerrate. Ein gewisser Anteil von Fehlern, ein Pufferüberlauf in einem Programm oder möglicherweise ein Designfehler eines Protokolls, läßt sich wiederum als Schwachstelle von potentiellen Angreifern ausnutzen, um die Verfügbarkeit, Integrität oder Vertraulichkeit des Systems einzuschränken.

Complexity is the worst enemy of security. This has been true since the beginning of computers, and is likely to be true for the foreseeable future.[Sch05]

Die Komplexität eines Programms, als Maßstab mögen hier die **L**ines **o**f **C**ode dienen, ist somit eine wichtige Kenngrösse zur Einschätzung seiner Sicherheit, vorausgesetzt es wurden darüberhinaus alle möglichen qualitätssichernden Maßnahmen bei der Entwicklung der Anwendung ergriffen. Ebenso relevant ist die Komplexität der zugrundeliegenden Dienste, auf denen die Anwendung bzw. deren sicherheitsrelevanten Teile aufbauen. Die Dienste, deren korrektes Funktionieren für die Sicherheitseigenschaften des betrachteten Programms maßgeblich sind, werden im Folgenden mit der etwas überfrachteten Bezeichnung **T**rusted **C**omputing **B**ase gefasst. Die TCB ist somit anwendungsspezifisch zu definieren, wie in [SH05] beschrieben.

2.1 L4 Microkernel

Eingedenk der Größe derzeit verbreiteter, monolithischer Betriebssysteme, kommen diese nicht als TCB für Anwendungen in sicherheitskritischen Bereichen in Frage. Der Linux-Kern kommt beispielsweise auf eine Größe von 200,000 LOC bei minimaler Konfiguration. Dieser Code läuft komplett im priviligierten Modus ab, das bedeutet es besteht Zugriff auf die gesamte Hardware einschliesslich des gesamten Speichers. Somit kann beispielsweise jeder Gerätetreiber die Seitentabelle beliebig verändern, egal ob dies sinnvoll ist oder nicht. Beinhaltet der Kernel eine Schwachstelle, die sich von Programmen, die im unpriviligierten Modus laufen ausgenutzen lässt, oder wird bösartiger Code in Form eines Kernel-Moduls z.B. eines Gerätetreibers geladen, so ist potentiell das gesamte System korrumpiert.
Bereits seit nunmehr 15 Jahren herrscht weitgehend Einigkeit darüber, dass monolithische Betriebssysteme, unter anderem wegen ihres ungeregelten Zugriffs innerhalb einer komplexen Softwarearchitektur, ausgedient haben. Gefragt sind stattdessen Systeme,

die einen modularen Aufbau besitzen, deren Module aber voreinander isoliert werden können. Umfangreiche Betriebssystemkernel, die bis hin zu Netzwerkprotokollen und Gerätetreibern nahezu alles beinhalten, was für den Einsatz des Rechners erforderlich sein könnte, werden tendenziell durch Microkernel ersetzt, die nur sehr wenige Primitive in sich vereinen. Dazu gehören Mechanismen zur Thread- und Adressraumverwaltung, sowie zur **I**nter **P**rocess **C**ommunication. Aufgrund der Überschaubarkeit des Kernel lässt sich die Korrektheit ungleich einfacher verifizieren. Vormals im priviligierten Modus betriebene Dienste werden in den unpriviligierten ausgelagert und laufen voneinander isoliert, in getrennten Adressräumen ab. Ein solcher Entwurf erzwingt, im Gegensatz zu monolithischen Systemen, eine saubere Schnittstellendefinition der einzelnen Module. Ausserdem ermöglicht die Entkopplung der vormaligen Bestandteile des Kernel, dass das Prinzip der geringstmöglichen Privilegierung durchgesetzt werden kann. Das Prinzip besagt kurzgefasst, dass Komponenten ausschliesslich auf die Daten zugreifen können, für die sie den Zugriff auch benötigen.
Ein Thread kann mit anderen Threads mittels IPC, die der Kontrolle des Microkernel unterliegt, kommunizieren. Eine Kommunikationbeziehung zweier Threads kann nur im Einverständnis beider Seiten aufgebaut werden. Die gemeinsame Nutzung von Speicherbereichen wird ebenfalls über IPC etabliert.
Mit der Implementation der L4 Microkernel Familie wurde bewiesen, dass sich die Performance-Einbußen, die durch die Separation der Module enstehen, auf ein vertretbares Maß reduzieren lassen. Dank der Möglichkeit, Adressräume strikt voneinander zu isolieren und dem hohen Grad an Verifizierbarkeit durch niegrige Komplexität, stellen die L4 Microkernel derzeit den idealen Kern einer TCB dar.

2.2 Linux spielt im Sandkasten

Das Fehlen von grundlegenden Werkzeugen für den L4 Microkernel führte dazu, dass sich die Betriebssystemgruppe der TU-Dresden im Jahr 1996 dazu entschloss, das Linux Betriebssystem, aufgrund seiner großartigen Hardwareunterstützung und Unixkompatibilität, auf L4 zu portieren. Dazu wurde lediglich der architekturabhängige Teil von Linux modifiziert bzw. eine L4 Schnittstelle hinzugefügt. Der Microkernel selbst blieb unverändert. Das modifizierte nun L4Linux genannte Programm läuft fortan im unpriviligierten Modus, was im Kontext der Sicherheit einen schönen Seiteneffekt mit sich bringt. Dadurch, dass der priviligierte Zugriff auf die Hardware entzogen ist, können L4Linux gewisse Resourcen z.B. der Zugriff auf die Netzwerkkarte vorenthalten werden. Es befindet sich in einer Sandbox. Nun ist es wiederum möglich, vertrauliche Daten an das vertrauensunwürdige L4Linux weiterzureichen und ggf. von diesem bearbeiten zu lassen. Bedingung dafür ist, dass es diese Daten über keinen Kanal an unbefugte Instanzen weiterreichen kann.
Ein anderer schöner Effekt dieser Form von Virtualisierung ist, dass auch mehrere L4Linux Instanzen parallel eingesetzt werden können.

2.3 Nizza Architektur

Ein Beispiel für eine kleine und generische Plattform, die für Anwendungen mit hohen Sicherheitsanforderungen konzipiert wurde und auf L4 aufsetzt, ist Nizza. Die Architektur ist in [Här02] vorgestellt worden und soll im Folgenden kurz umrissen werden. Nizza besteht aus einer Sammlung von L4 Servern, welche zusammen mit dem Microkernel als TCB fungieren, sowie L4Linux als Basis für althergebrachte Unixanwendungen. Eine unvollständige, beispielhafte Konfiguration ist in Abbildung 2.1 dargestellt.

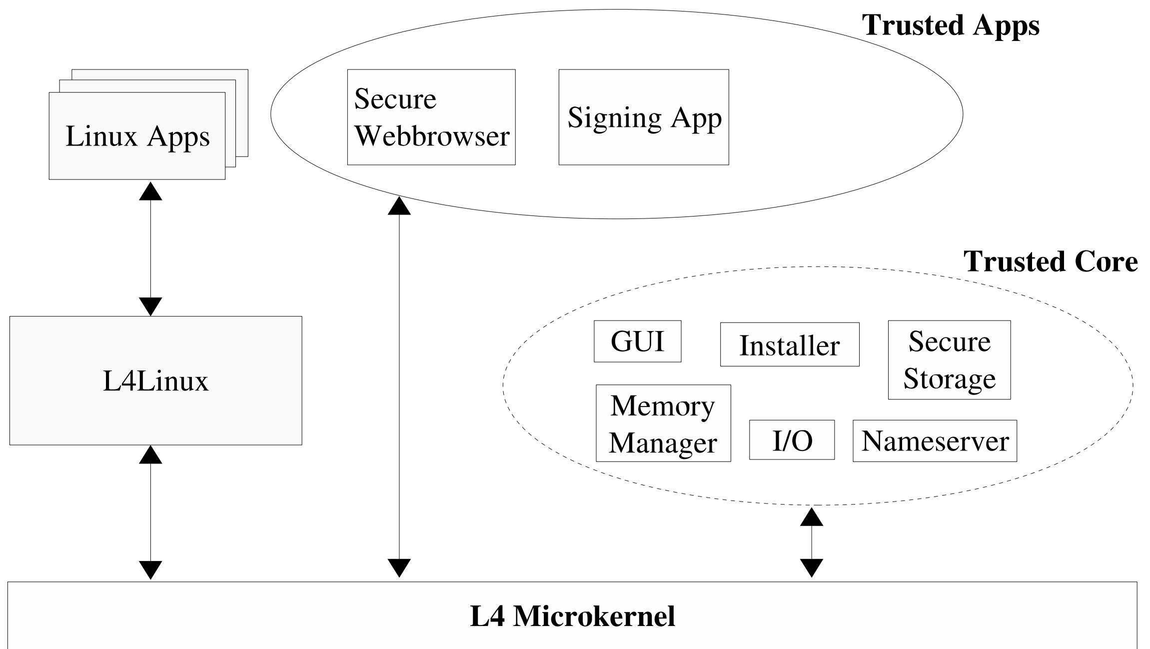

Abbildung 2.1: Beispielhafte Nizza Konfiguration

Die vertrauenswürdigen Basisdienste kapseln und kontrollieren sämtliche Ressourcen des Systems, wie Speicher, CPU und I/O. Weiterhin ist ein Nameserver notwendig, der bei Bedarf anhand symbolischer Namen die Identität der Basisdienste zurückgeben kann. Je nach Anforderungen der zu schützenden Anwendungen können optional weitere Server Teil der Basis sein, z.B. ein Windowmanager im Falle einer interaktiven Anwendung. Die in Abschnitt 2.1 beschriebenen Vorteile getrennter Addressräume und wohldefinierter Kommunikation zwischen den Threads, sowie die Bestrebung die vertrauenswürdige Basis so klein wie möglich zu gestalten, soll die erforderliche Sicherheit garantieren. Die Bestrebung, die TCB soweit wie möglich zu verkleinern, führte bei dieser Architektur zur Einführung sogenannter vertrauenswürdiger **Wrapper**. Die Idee dabei ist, vertrauensunwürdige Systemkomponenten durch besagte Wrapper zu kapseln und sie für gewisse Bearbeitungen vertrauenswürdiger Daten zu benutzen. Ein Beispiel soll den Grundgedanken illustrieren. Ein L4 Server, der Teil der vertrauenswürdigen Basis und für das persistente Speichern vertraulicher Daten zuständig ist, kann entsprechende Daten ver-

schlüsseln und die anschliessende persistente Speicherung einem Dateisystem überlassen, dass linuxbasiert und nicht Teil der TCB ist. Da die Daten verschlüsselt übergeben werden, wird die Vertraulichkeit und Integrität nicht eingeschränkt, aber gleichzeitig wird der Codeanteil der TCB drastisch reduziert.

2.4 Aufsplitten von Anwendungen

Der maßgebliche Vorteil, den die Portierung von Linux auf L4 mit sich brachte, wurde bereits genannt: die Verfügbarkeit einer ganzen Reihe von Unix und Linuxanwendungen. Weiterhin lassen sich durch den Einsatz von Wrappern Linuxkomponenten nutzen, wodurch die Größe der TCB klein gehalten werden kann. Eine ähnliche Vorgehensweise kann bei der Wiederverwendung von Anwendungprogrammen genutzt werden, die hohe Sicherheitsanforderungen erfüllen sollen.
Oft ist nur ein kleiner Teil einer solchen Anwendung mit der Bearbeitung des sensitiven Datenmaterials beschäftigt. Ist es möglich, ein vorhandenes Linuxprogramm in zwei Teile aufzusplitten, in einen informationssensitiven und einen Teil, in dem keine vertraulichen Informationen behandelt werden, so kann letzterer weiter auf L4Linux ablaufen und der sicherheitskritische Part wird auf Basis von L4 neu entwickelt bzw. auf L4 portiert. Die Filtration und Neuzusammensetzung der sicherheitskritischen Teile zu einer sicheren Kernanwendung beschreibt der Autor in [SH05] als *Application Kernelizing.*
Die grundsätzliche Vorgehensweise ist wie folgt:

1. **Analyse der Anwendung**
   Dieser Schritt beinhaltet das Erkennen der verschiedenen Komponenten der Anwendung, sowie die Zuordnung der Funktionalitäten zu den einzelnen Komponenten.

2. **Identifikation sicherheits-sensitiver Daten**
   Im zweiten Schritt wird der Fluss der zu schützenden Daten innerhalb der Anwendung analysiert.

3. **Identifikation sicherheits-relevanter Komponenten**
   Die Komponenten werden nun dahingehend unterschieden, ob sie ausschliesslich zu schützende Daten behandeln (Kat. A), lediglich mit Daten operieren, die nicht sicherheits-relevant sind (Kat. B), oder ob sie eine Menge beider Datentypen berühren (Kat. C).

4. **Entwurf der vertrauenswürdigen Komponente**
   Anschliessend wird eine neue, vertrauenswürdige Komponente aus den Komponenten der Kategorie A, sowie aus den sicherheits-relevanten Teilen der Kategorie C gebildet. Besitzt die Ausgangssoftware einen sauberen Entwurf, ist es mitunter möglich, dass die neu gewonnene Komponente diesselbe Schnittstelle anbietet, wie das Sammelsurium der vorher bestehenden Komponenten.

5. **Einsatz der neuen Komponente**
   Zuletzt werden die Aufrufe in der Ausgangssoftware an entsprechender Stelle durch Aufrufe der neu entwickelten vertrauenswürdigen Komponente ersetzt.

Im Prinzip kann die Trennung in sicherheits-kritische und -unkritische Komponenten auch auf Basis von herkömmlichen, monolithischen Systemen vollzogen werden und glücklicherweise forcieren auch viele Entwickler mittlerweile einen solchen sauberen Entwurf. Beispielsweise greifen viele Unixprogramme, wenn sie Daten signieren wollen, auf grundlegende Basisbibliotheken und Server wie GPG zurück. Die Kontrolle der Schlüssel, sowie der Signiervorgang selbst, ist dabei vollständig aus der Anwendung ausgelagert. Der qualitative Unterschied des Application Kernelizing besteht jedoch vor allem darin, dass die Komponenten, die vertrauensvolle Daten behandeln, vollständig in einer vertrauenswürdigen Umgebung ablaufen, was bei monolithischen Systemen, wie in Abschnitt 2.1 beschrieben, nicht der Fall ist.

2.5 Vertrauenswürdiger Fenstermanager DOpE

DOpE stellt einen echtzeitfähigen Windowmanager auf Basis von L4 dar. Als eine Art Multiplexer bildet DOpE die verschiedenen graphischen Ausgaben und Tastatur- und Mauseingaben auf entsprechende Fenster ab. Für jedes Fenster, welches eine Anwendung rendern will, bekommt es von DOpE einen Bereich des Framebuffers zugewiesen. Jedes Tastatur- oder Mouse-Event, das sich auf diesen Ausschnitt bezieht, wird an die jeweilige Anwendung durchgestellt, jeder andere Bereich des Framebuffers ist jedoch geschützt, d.h. der Fenstermanager besitzt exklusiven Zugriff auf den Adressbereich, welcher den Framebuffer abdeckt. Damit wird verhindert, dass Fenster fremder Anwendungen verändert oder deren Eingaben abgefangen werden können. Der von DOpE für jedes Fenster erzeugte Rahmen kennzeichnet die jeweils assoziierte Applikation eindeutig. Damit wird gewährleistet, dass NutzerInnen die Herkunft des Fensters eindeutig einer Anwendung zuordnen können. DOpE gewährleistet Vertraulichkeit in dem Sinne, dass keine Anwendung Eingabedaten oder dargestellte Pixel eines Bildschirmbereichs eines anderen Programms auslesen kann. Integrität wird dahingehend garantiert, dass der dargestellte Fensterinhalt ausschliesslich der den Rahmen betitelnden Anwendung entstammt. Mit weniger als 15,000 LOC ist DOpE klein genug, um als Teil der TCB für die zu erstellende Webbrowseranwendung zu fungieren.
Dank einem zusätzlichen Mechanismus dem **O**verlay **W**indow **M**anagement[FH04] ist es möglich herkömmliche Fenstermanager, wie z.B. X11 auf Basis von L4Linux, in den Desktop auf eine intuitiv bedienbare Art und Weise zu integrieren. Während bisher der zugrundeliegende Fenstermanager damit zu kämpfen hatte, dass das Gastbetriebssystem eine Blackbox darstellte, was das interne Arrangement seiner Fenster anging, so wird beim OWM durchgesetzt, dass zusätzliche Informationen über die Anordnung der Fenster vom Gastssystem an die Basis, in diesem Falle DOpE durchgereicht werden. Ganz praktisch bedeutet das, dass einzelne Fenster beispielsweise von verschiedenen X11 basierten Anwendungen neben nativen DOpE Anwendungen einzeln dargestellt werden können, während in vormaligen Ansätzen ein Fenstermanager jeweils immer nur ein Fenster zugeteilt bekommen hat, in dem alle von diesem verwalteten Fenster dargestellt wurden.

2.6 Flips

FLIPS stellt L4 Anwendungen die BSD Socket Schnittstelle zur Verfügung und ermöglicht es daher L4 Entwicklern, analog zur gewohnten Unix Umgebung, Netzwerkverbindungen zu etablieren. Dabei wird intern auf den Linux 2.4 Netzwerkprotokollstapel zurückgegriffen.

2.7 MatrixSSL

MatrixSSL[1] ist eine SSL Bibliothek, die speziell für den Einsatz in eingebetteten Systemen entwickelt wurde und bietet aufgrund seiner minimalen Größe gute Voraussetzungen zur Bildung einer sicheren, überschaubaren Architektur. Die Bibliothek ist für verschiedene Plattformen verfügbar. Die übersetzte Linuxvariante für die x86 Architektur umfasst gerade einmal 38 Kilobytes und beinhaltet sowohl Client- als auch Serverfunktionalitäten. MatrixSSL war bereits zu Beginn dieser Arbeit auf L4 portiert worden und stand somit direkt zur Verfügung.

2.8 Webbrowser

Die Anzahl verschiedener Webbrowser ist beachtlich und sie alle in Hinblick auf ihren Nutzen im Kontext dieser Arbeit zu untersuchen erscheint aussichtslos, ich habe mich daher auf eine gewisse Auswahl konzentriert, wobei der Verbreitungsgrad, die Unterstützung diverser Features, die Verfügbarkeit des Quellcodes, sowie die Fähigkeit graphischer Darstellung eine Rolle spielte. Näher betrachtet wurden Mozilla Firefox, als wohl verbreitetster frei verfügbarer Webbrowser, sowie Konqueror Embedded und Dillo, die trotz ihrer geringen Größe vergleichsweise hohen Komfort bieten.
Die Aufgaben eines Webbrowsers sind alles andere als trivial, abgesehen davon, dass die Spezifikationen für HTTP und HTML zunehmend länger und komplizierter geworden sind und sich darüberhinaus viele Webdesigner und Designer von Webservern und Proxies nur bedingt an diesen orientieren, werden sie mittlerweile auch für vieles Andere eingesetzt wie z.B. multimediale Repräsentationen, Java Applets, Javascript, diverse weitere Protokolle und für das Dateimanagement. Die Folge hiervon ist eine stetige Zunahme ihres Umfangs.
Abbildung 2.2 zeigt eine Übersicht über die Größenordnung der von mir betrachteten Browser.[2]
Wie den Zahlen zu entnehmen ist, scheidet Firefox aufgrund seiner Komplexität als vertrauenswürdiger Browser aus. Auch die in Abschnitt 2.4 beschriebene Splitting Methode kommt für den Firefox kaum in Frage, da alleine die Rendering Engine und grundlegenden Netzwerkfunktionen, die für einen vertrauensvollen Kern der Applikation notwendig wären, auf ein umfangreiches Framework aufsetzen, welches Plattformunabhängigkeit gewährleisten soll.

[1] MatrixSSL ist eine eingetragene Handelsmarke von Peersec Networks, Inc.
[2] Die Codezeilen Angaben, welche in dieser Arbeit gemacht werden, wurden mittels SLOCcount von David A. Wheeler gemessen.

Browser	LOC
Mozilla Firefox	$> 1\ 000\ 000$
Konqueror Embedded	$\approx 150\ 000$
Dillo	$\approx 29\ 000$

Abbildung 2.2: Umfang der betrachteten Webbrowser

Der Konqueror Embedded oder Dillo muten vielversprechender an. Konqueror Embedded setzt auf QT Embedded auf und ist in C++ verfasst. Dillo ist in C geschrieben und benötigt die GTK+. Im Gegensatz zu Dillo beinhaltet die Minimalvariante des Konqueror bereits die Unterstützung von SSL und Javascript. Ansonsten ähnelt sich insgesamt die Komplexität der beiden Anwendungen und beide kämen als Ausgangspunkt zur Bildung einer vertrauensvollen Rendering Engine in Frage.

3 Entwurf

Grundlage des von mir entwickelten Entwurfs bildet die in Abschnitt 2.3 vorgestellte Nizza Architektur. Beim zugrundeliegenden Microkernel handelt es sich um Fiasco, eine Implementation der L4 Spezifikation Version 2. Ziel war es, linuxbasierte Webbrowser wiederzuverwenden und dabei die sicherheits-sensitiven Daten zu schützen. Um eine Vielfalt an Webclients zuzulassen, war ich dabei insbesondere bestrebt, wenig bis gar keine Modifikation am Webbrowser vorzunehmen.
Bei der Entwicklung des Entwurfs wurde analog zur beschriebenen Vorgehensweise in Abschnitt 2.4 zunächst analysiert, welche Teile eines Browsers sensitive Daten verarbeiten. Dazu ist es notwendig, entscheiden zu können, wann sensitive Daten vorliegen und wann nicht. Es wird im Folgenden davon ausgegangen, dass Daten, die über das Protokoll HTTPS übertragen werden, geschützt werden müssen, während bei *plain* HTTP damit zu rechnen ist, dass die Daten für potentielle Angreifer sichtbar übertragen werden und daher keines besonderen Schutzes bedürfen.[1]
Bei HTTPS wird HTTP durch SSL oder TLS getunnelt, d.h. es wird zunächst mit dem entsprechenden Webserver ein Handshake durchgeführt. Dabei werden diverse Schlüssel ausgetauscht, mit welchen in der Folge die HTTP-Pakete ent- und verschlüsselt werden. Der Vorgang des Handshakes und als Vorraussetzung für diesen die Verwaltung von Zertifikaten sowie die kryptographischen Funktionen zur Umkodierung der Daten müssen also definitiv Teil der vertrauenswürdigen Basis sein. Darüberhinaus müssen die noch unverschlüsselten Eingabedaten, beispielsweise eines Passwort-Eingabefeldes und die bereits entschlüsselten, vom Webserver empfangenen Daten, vertraulich behandelt werden. Das bedeutet, dass HTTP Parsing und Generierung und HTML Rendering ebenfalls in einer vertrauensvollen Umgebung verfügbar sein müssen.
Es sind nun zwei verschiedene Ansätze denkbar. Einerseits die Implementation einer sicheren L4 Kernanwendung, die diese Funktionen in sich vereint und darüberhinaus die Möglichkeit bietet, Plugins aus einer unsicheren Umgebung zu laden, um die gewünschten zahlreichen Zusatzfunktionen zu gewährleisten. Im Kontext von vertraulichen HTTPS Verbindungen müssen diese Plugins gesperrt werden. Damit wird der Komfort moderner Webbrowser gewahrt, allerdings ist die Realisierung mit erheblichem Aufwand verbunden, zumindest solange es nicht gelingt die Schnittstelle an bereits verfügbare Plugins anzupassen. Dies wiederum kann zu einer sehr komplexen Kernanwendung führen.
Die zweite Lösung ist ungleich einfacher und besteht darin, einen unmodifizierten Webbrowser freier Wahl in einer L4Linux Sandbox zu laden und über eine *Application Firewall* abzuschirmen. Letzterer Ansatz ist schnell und einfach zu realisieren, stellt die nutzerfreundlichste Lösung dar, da der Lieblingsbrowser unmodifiziert übernommen

[1] Mit anderen Worten: Webservern die vertrauensvolle Daten per unverschlüsseltem HTTP übertragen ist auf Clientseite nicht mehr zu helfen.

werden kann und bietet vergleichbaren Schutz, was in den folgenden Abschnitten näher erläutert werden soll.

3.1 Architektur Übersicht

Das zugrundeliegende System für den Entwurf besteht aus Fiasco mit L4Env, Flips und DOpE als vertrauensvoller Basis. Darauf läuft L4Linux als potentiell unsichere, herkömmliche Arbeitsumgebung mit Zugriff auf Massenspeichergeräte, Soundkarte, USB etc. Lediglich der Zugriff auf die Eingabegeräte und Grafikkarte, die DOpE vorbehalten sind und die Netzwerkkarte, die Flips nutzt, werden dieser Instanz vorenthalten. Abbildung 3.1 zeigt den Gesamtentwurf.

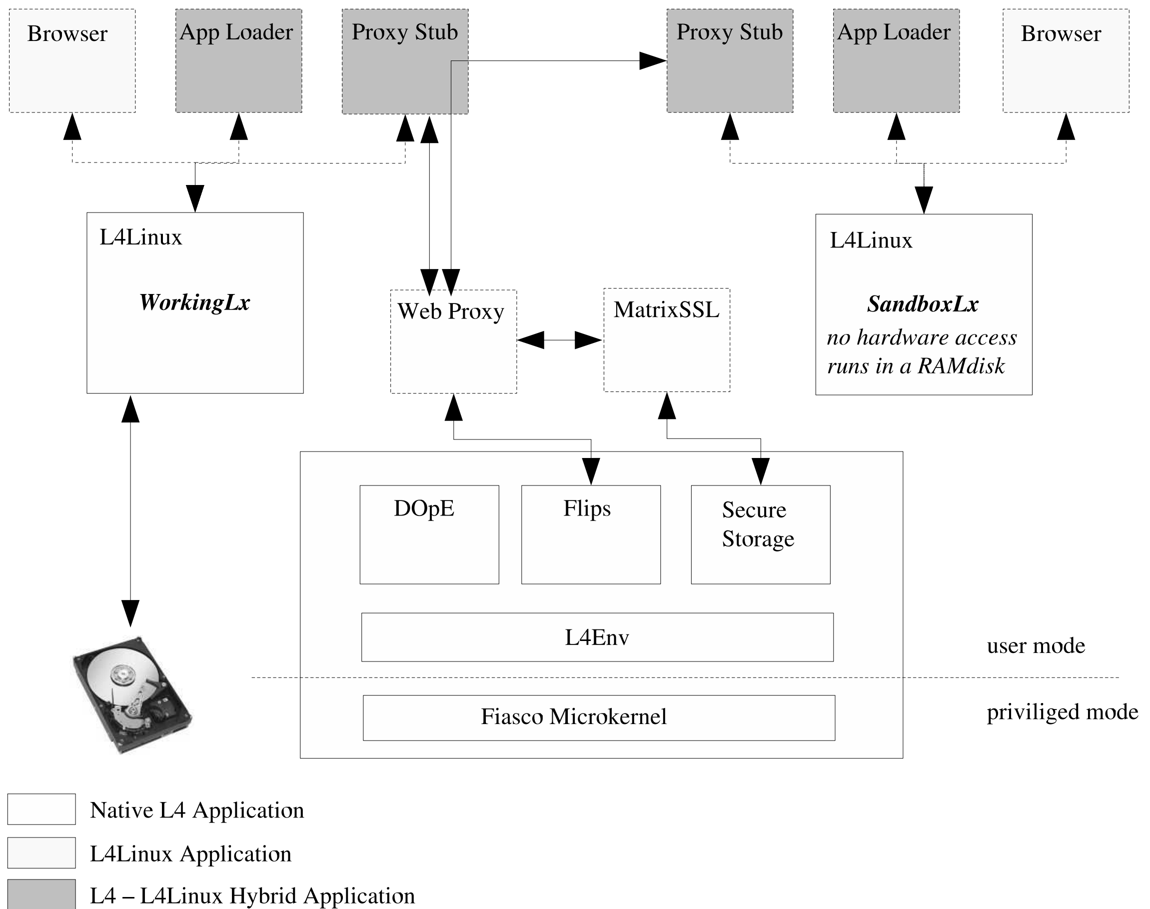

Abbildung 3.1: Architektur Überblick

Soll von einem Webclient der L4Linux Instanz, im Folgenden *WorkingLinux* genannt, eine HTTP oder HTTPS Verbindung hergestellt werden, so muss eine entsprechende Anfrage an die Application Firewall gestellt werden, die des Weiteren mit *L4 HTTP Proxy* bezeichnet wird. Der L4 HTTP Proxy prüft, ob es sich um eine HTTP oder

HTTPS Anfrage handelt. Im ersten Fall wird der Aufbau der Verbindung gestattet, andernfalls verweigert. Stattdessen wird eine zweite L4Linux Instanz gestartet, im Folgenden mit *SandboxLinux* bezeichnet, die vollkommen isoliert in einer RAM-Disk läuft. Da das SandboxLinux jeder Möglichkeit beraubt ist, einer anderen Instanz jenseits der TCB Informationen zukommen zu lassen und ihr nur eine SSL-Session zur anfänglich angeforderten Domain gestattet wird, kann die Kapselung der vertrauenswürdigen Daten als nahezu hundertprozentig angesehen werden, wenn verdeckte Kanäle über beispielsweise CPU-Nutzung und Speicherverbrauch aus der Betrachtung ausgeschlossen werden. Auf Basis von SandboxLinux wird ein Browser mit der soeben gewünschten HTTPS Anfrage gestartet, vorzugsweise der gleiche Webbrowser, der auch unter WorkingLinux verwendet wird.

3.1.1 Restriktion auf Domains

Soll aus dem isolierten L4Linux heraus eine neue HTTPS Anfrage an eine andere Domain gestellt werden, so muss wiederum ein neues SandboxLinux gestartet werden. Der Hintergrund für den restriktiven Zugriff des SandboxLinux auf lediglich eine Domain ist folgender: Wird eine verschlüsselte Seite angefordert, die von einem Angreifer kontrolliert wird, und gelingt es diesem Schwachstellen des Webclients im SandboxLinux auszunutzen, so ist der Webclient, oder womöglich die ganze Linuxinstanz korrumpiert. Bei einem anschliessenden Verbindungsaufbau zu einem anderen Server, oder auch nur zu einer anderen Domain[2] ist die Vertraulichkeit nicht länger gewährleistet. Selbiges gilt für eine Klartext-Anfrage, auch diese darf nicht weiter von Webclients des SandboxLinux bearbeitet werden. Die angeforderte URL muss stattdessen als Argument einer neu zu startenden Webbrowser Instanz im WorkingLinux übergeben werden.
Die Restriktionen können auch flexibel verwaltet werden. Dafür muss der L4 HTTP Proxy einen Dialog zur Verfügung stellen, in welchem der Nutzer oder die Nutzerin selbstständig bestimmen können, auf welche Domains von welcher Linuxinstanz zugegriffen werden kann. So liessen sich von einem SandboxLinux zu verschiedenen Domains, denen Vertrauen entgegengebracht wird, Verbindungen aufbauen.

3.1.2 Erwägungen zur Performance

Die zusätzlich erforderlichen Linuxinstanzen und RAMDisks belegen selbstverständlich mehr Arbeitsspeicher, als dies zuvor der Fall war. Da das SandboxLinux keine Hardware verwalten muss, kann der Kernel sehr klein gehalten werden. Problematischer ist da schon die Verwendung der RAMDisk, insbesondere wenn der zu verwendende Webbrowser groß, und auf einen X-Server angewiesen ist. Ein Linuxsystem mit X11 und ohne Webbrowser lässt sich auf einen Umfang von ca. 32 MB verkleinern.
Das Laden des SandboxLinux kann im Vorfeld, bevor ein entsprechender Zugriff notwendig wird, geschehen. Damit wird zwar zunächst unnötigerweise Speicher belegt, dafür

[2] Internet Service Provider bieten vielen Kunden die Möglichkeit ihre Internetpräsens auf deren Webservern zu gestalten, so dass häufig unterschiedliche Domains auf einen Server verweisen. Daher ist eine Restriktion auf Basis von IP Adressen an dieser Stelle unzureichend.

entfallen Ladezeiten der Linuxinstanz, bei einer HTTPS Anfrage. Positiv auf die Ladezeit des Webbrowser im SandboxLinux wirkt sich der Gebrauch der RAMDisk aus, da Plattenzugriffe und das Kopieren in den Arbeitsspeicher entfallen. Der zusätzliche Aufwand, der durch den Gebrauch des Proxies und Wechsel der Linuxinstanzen entsteht, kann durch den Gebrauch der RAMDisk ausgeglichen werden, wie im Kapitel 5 gezeigt werden wird.
Dank dem in Abschnitt 2.5 beschriebenen Overlay Window Management wirkt jeder Wechsel von einer Linuxinstanz zur anderen auf die BenutzerInnen so, als öffne sich lediglich ein neues Browserfenster. Um den Unterschied zwischen der völlig isolierten und damit sicheren Anwendung und der ungesicherten kenntlich zu machen, müssen die jeweiligen Fenster von DOpE verschiedentlich textuell und / oder farbig gekennzeichnet werden.

3.2 L4Linux Proxy Stub

Da der jeweilige Webbrowser unmodifiziert übernommen werden soll und von sich aus nicht in der Lage ist, Anfragen per IPC an den L4 HTTP Proxy zu stellen, bedarf es eines Hybridprogramms - eines Linuxprogramms, in welchem gewisse L4 Bibliotheken eingeblendet werden, so dass dieses stellvertretend IPC Calls versenden kann. Dieses Programm kann auf zwei verschiedene Arten integriert werden, um an die angefragten Daten des Webbrowsers zu gelangen. Entweder wird die komplette Socketschnittstelle des Linuxkerns adaptiert, so dass Anfragen an die Standard HTTP und HTTPS Ports 80 und 443 an den Proxy umgeleitet werden, oder es wird ein eigener Proxyserver auf der L4Linux Seite implementiert, der als Ansprechpartner zur Verfügung steht.
Die erste Variante besitzt allerdings den Nachteil, dass wenn sich nicht nur auf die Standardports konzentriert wird, sämtlicher Datenverkehr daraufhin analysiert werden muss, ob es sich um HTTP handelt oder nicht. Im Gegensatz dazu kann ein auf ein spezifisches Anwendungsprotokoll orientierter Proxy, Anfragen ungleich einfacher behandeln. Es wurde darum letztere Variante umgesetzt. Dieser Proxy Stub ist also ein eigenständiger Webproxy, der auf einem bestimmten Port lauscht, Anfragen vom Webbrowser entgegen nimmt und an den L4 Server weiterreicht, sowie die erhaltene Antwort über den Socket zurück an den Browser schreibt. Die erhaltenen Anfragen und Antworten müssen zu einem gewissen Teil analysiert werden, um herauszufinden wie lange die Verbindung zum Webbrowser aufrechterhalten werden muss und damit entschieden werden kann, ob eine SSL gesicherte oder Klartext-Anfrage an den L4 HTTP Proxy gestellt werden soll.
Hierbei ergibt sich ein fundamentales Problem: Die Etablierung von SSL Verbindungen ist der jeweiligen Anwendung vorbehalten, d.h. es gibt keinen einheitlichen systemweiten Mechanismus dafür. Das bedeutet wiederum, dass potentiell jeder Webrowser eine eigene SSL Bibliothek verwendet. Wird eine HTTPS URI aufgerufen, so sendet der Browser, bei Verwendung eines Proxys, in der Regel eine CONNECT Anfrage an diesen, um zu signalisieren, dass eine Verbindung zu dem übermittelten Server aufgebaut werden soll. Anschliessend wird die eigentliche Anfrage verschlüsselt, und für den Proxy unverständlich zum Server getunnelt. Da jedoch die Anfrage gefiltert, und an den isolierten

Webbrowser umgeleitet werden soll, muss die Verschlüsselung unterbunden werden. Das bedeutet, dass Modifikationen am Browser vorgenommen werden müssten, was aber ja gerade verhindert werden sollte. Mit einem Trick, der im Folgenden erläutert wird, lässt sich dies auch weiterhin umgehen.
Bei dem bereits erwähnten Handshake einer SSL Session wird, nach dem der Client signalisiert hat, dass der Wunsch nach einem Verbindungsaufbau besteht, vom Server der öffentliche Schlüssel eines asymetrischen Kryptographieprotokolls gesendet, mit welchem der Client die von ihm generierten Session-Schlüssel für die eigentliche Umkodierung der Pakete verschlüsseln und an den Server senden soll. Um die Authentizität des Servers gegenüber dem Client zu gewährleisten, unterschreibt der Server oder besser eine Zertifizierungsstelle sein Schlüsselpaket. Der Client kennt entweder das Serverzertifikat, oder fragt bei der Zertifizierungsstelle nach, ob die Unterschrift gültig ist. Abbildung 3.2 stellt den Handshake in vereinfachter Form dar.

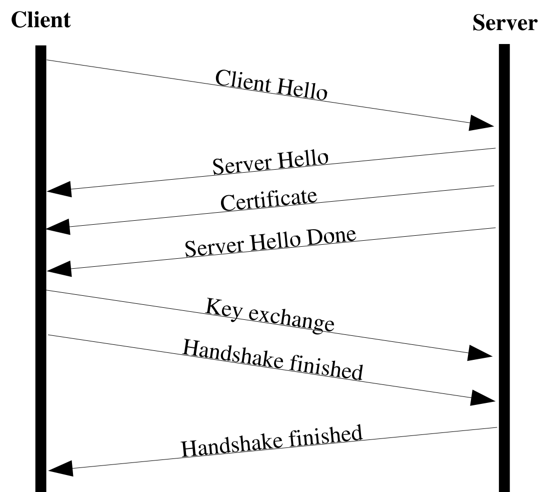

Abbildung 3.2: SSL Handshake

Wird ein Zertifikat, welches dem L4Linux Proxy Stub gehört und gültig zur Validierung beliebiger Domänen ist, der Zertifikatsverwaltung des Browsers hinzugefügt, so kann der Proxy bei jedem Versuch des Browsers ein Handshake mit einem Webserver zu vollziehen, sich als der gewünschte Server ausgeben, die Anfrage entgegennehmen und dem Browser eine Negativantwort übermitteln. Diese quasi gewünschte *man in the middle attack* ist allerdings mit einem hohen Performanceverlust verbunden, da für jede SSL Anfrage aus der ungesicherten Umgebung heraus, ein Handshake und ein Ver- und Entschlüsselungsvorgang zusätzlich getätigt werden müssen. Ohne Modifikation des Webbrowsers lässt sich dieser Verlust allerdings nicht umgehen. Durch den bereits erwähnten Performancegewinn, dank der Verwendung einer RAMDisk durch das SandboxLinux, wird der Verlust etwas ausgeglichen. Auch hier sei auf die Auswertung im Kapitel 5 verwiesen.

3.3 L4 HTTP Proxy

Der L4Linux Proxy Stub soll also in der Lage sein, sowohl Klartext als auch verschlüsselte Anfragen des Webbrowsers zu verstehen und dem HTTP Proxy zu übermitteln. Dazu sollte letzterer zwei Schnittstellen zur Verfügung stellen, eine für HTTP und eine für HTTPS Anfragen. Bei HTTPS Anfragen müssen über den eigentlichen HTTP Request hinaus, der Host und Port, zu dem die Verbindung hergestellt werden soll, angegeben werden. Diese Informationen werden vom Proxy Stub in der im vorigen Abschnitt beschriebenen Weise, aus einem CONNECT Request filtriert.
Darüberhinaus muss für jede Anwendung, die berechtigt ist Anforderungen zu stellen, eine Struktur verwaltet werden, anhand derer entschieden werden kann, welche Anforderungen legitim sind und welche nicht. Konkret muss die Struktur die Entscheidung ermöglichen, an welche Domains Anfragen gesendet werden können und ob dies für HTTP, HTTPS oder beide Anfragetypen zulässig ist. Für die erwähnte WorkingLinux Instanz würde beispielsweise ein Eintrag generiert werden, der beinhaltet, dass die Anwendung HTTP Anfragen an beliebige Domains versenden darf.
Der interne Ablauf ist wie folgt. Zunächst prüft der Proxy, ob eine Anwendung berechtigt ist, die übermittelte Anfrage zu stellen, wenn nicht, wird eine berechtigte Instanz geschaffen und mit der Anfrage beauftragt. Konkret wird ein Webbrowser geladen, mit der geforderten URL als Argument, entweder auf Basis einer vorhandenen Linuxinstanz, die berechtigt ist, die geforderte Domain zu kontaktieren, oder einer zu diesem Zweck neu geladenen und mit den nötigen Rechten versehenen Linuxinstanz. Als Antwort einer unberechtigten Anfrage wird ein HTTP Response mit eigens definiertem Fehlercode und HTML Seite generiert. Die HTML Seite informiert die AnwenderInnen darüber, dass die gewünschte Seite in einer entsprechend anderen, entweder sicheren oder ungesicherten Umgebung, geladen wird.
Ist die gestellte Anfrage legitim, wird nach folgendem Schema verfahren:

1. Parsen des HTTP Header
2. Analyse der Informationen zum Webserver, der Verbindungsdauer und spezifischer Proxy Header
3. Sofern noch keine Verbindung besteht: Aufbau der Verbindung zum Webserver
4. Sofern es sich um eine HTTPS Anfrage handelt und keine Verbindung bestand: SSL Handshake
5. Senden der Anfrage
6. Empfangen der Antwort
7. Sofern die Verbindung nicht aufrechterhalten werden soll, schliessen der Verbindung
8. Im Falle eines Fehlers: Generierung der Antwort
9. Übermittlung der Antwort an den Client

3.4 L4Linux Application Loader

Wie bereits erwähnt, muss der HTTP Proxy in der Lage sein, neue L4Linux Instanzen zu laden und auf deren Basis einen Webbrowser mit entsprechender URI als Argument zu starten. Für das Laden der L4Linux Instanzen kann ohne weiteres auf eine bestehende L4 Anwendung, den sogenannten Loader zurückgegriffen werden. Problematischer ist es L4Linux Anwendungen, wie den Webbrowser, zu laden. Dies kann nicht direkt vom HTTP Proxy vollzogen werden.
Es bedarf einer eigenen L4Linux Anwendung, dem Application Loader.[3] Diese muss eine Schnittstelle anbieten, die es ermöglicht, den Namen der zu startenden Anwendung, sowie gewünschte Argumente zu übergeben. Außerdem muss entschieden werden können, ob das Programm als Vorder- oder Hintergrundprozess gestartet werden soll. Die Unterscheidungsmöglichkeit nach Vorder- oder Hintergrundprozess ist dahingehend notwendig, da beispielsweise textbasierte Webbrowser durchaus als Vordergrundprozess auf dem Terminal des Loaders gestartet werden müssen. Dagegen können X11 basierte Webclients als Hintergrundprozesse gestartet werden. Es muss lediglich das Display als Umgebungsvariable gesetzt werden.
Der L4Linux Application Loader kann vom Proxy über den Nameserver ermittelt werden. Soll eine neue Anwendung im Vordergrund gestartet werden, muss er den bestehenden Vordergrundprozess solange anhalten, bis die neu gestartete Anwendung terminiert.

3.5 Vertrauenswürdigkeit der L4Linuxanwendungen

Die Loaderanwendung kann in seiner Eigenschaft als Linuxprogramm, ebenso wie der L4Linux Proxy Stub, nicht als Teil der vertrauenswürdigen Basis angesehen werden. Da beiden Applikationen keine sicherheits-sensitiven Daten anvertraut werden, zumindest sofern sie nicht in der abgeschlossenen Umgebung des SandboxLinux laufen, stellt dies jedoch kein Problem dar. Lediglich die korrekte Authentifikation der Applikationen, die auf SanboxLinux laufen, muss sichergestellt werden können. Derzeit existiert hierfür zwar noch keine Implementation, dennoch lässt sich eine Lösung skizzieren. Um die SandboxLinux Instanz sowie deren Anwendungen korrekt zu identifizieren, bedarf es zunächst eines Speichers der die Integrität von Kernel und RAMDisk gewährleisten kann. Es wurde bereits in Abschnitt 2.3 kurz darauf eingegangen, wie mittels eines Wrappers die Daten verschlüsselt beispielsweise dem unsicheren WorkingLinux überantwortet werden können. Ein L4 Server, der für das sichere Nachladen von Anwendungen zuständig ist - Installer genannt - lässt sich die verschlüsselte RAMDisk und den Kernel für das SandboxLinux vom Wrapper des WorkingLinux übergeben. Anschliessend werden die Daten von ihm, der die Schlüssel verwaltet, entschlüsselt und das SandboxLinux wird geladen. Beim Booten des SandboxLinux wird der in der RAMDisk befindliche L4Linux Application Loader gestartet. Dieser verfügt über einen Identitätsschlüssel, der zur Authentifikation beim Nameserver dient. Da die Daten dank der kryptographi-

[3] Zwar wurde im Rahmen einer anderen Belegarbeit bereits eine ähnliche Anwendung implementiert, diese ist jedoch stark vereinfacht und wenig flexibel, so dass an ihrer statt eine neue Anwendung entworfen werden musste.

schen Maßnahmen vertraulich hinterlegt waren und ihre Integrität überprüft ist, bleibt gewährleistet, dass sich keine andere Instanz die Identität aneignen kann. Ebenso wird beim L4Linux Proxy Stub verfahren.

4 Implementierung

Die von mir umgesetzte experimentelle Implementation leitet sich direkt von dem in Kapitel 3 vorgestellten Entwurf ab. Auf Abweichungen oder Vereinfachungen wird in der Folge detailliert eingegangen. Einige Einschränkungen ergaben sich insbesondere, weil ich den Proxy ohne die Möglichkeit der Verwaltung persistenter Verbindungen implementiert habe. Das Fehlen von HTTP Basisfunktionalitäten führte dazu, dass viel Energie in die Erstellung der benötigten Parsing- und Genese-Methoden einfloss. Rückblickend wäre die anfängliche Portierung einer entsprechenden Bibliothek, wie z.B. der libwww des W3C, möglicherweise sinnvoller gewesen. HTTP 1.1 ist insbesondere bei der Realisierung einer Proxyanwendung kein triviales Protokoll. Wird darüberhinaus auf die Abweichungen der Webserver und Browser vom Standard eingegangen, wächst der Aufwand entsprechend. Anfänglich erschien mir eine Portierung bestehender Bibliotheken, aufgrund ihres enormen Umfangs als schlechter Weg. Die verbreitete libcurl und die erwähnte libwww enthalten jeweils ca. 50 000 LOC. Das Problem besteht darin, dass kleine Bibliotheken z.B. für eingebettete Systeme meist nur die Client- oder die Serverseite beinhalten, ein Proxy benötigt jedoch beide Interfaces. Die Größe der beiden zuvor erwähnten Bibliotheken rührt allerdings vor allem daher, dass sie eine Vielzahl von Protokollen über HTTP hinaus beinhalten. Insofern wäre für die Zukunft die Entkopplung einer der beiden Bibliotheken denkbar.

4.1 Generischer Multi Threads Server

Moderne Webbrowser versuchen lange Verzögerungszeiten dadurch zu umgehen, dass sie mehrere Verbindungen hin zum Webserver gleichzeitig aufbauen, um parallel Bilder, Seiten etc. herunterzuladen. Somit behindern lange Antwortzeiten einzelner Ressourcen nicht den Gesamtablauf. Dementsprechend sollte ein Webproxy ebenso in der Lage sein, mehrere Prozesse gleichzeitig laufen zu lassen, um parallel Anfragen entgegenzunehmen und weiterzustellen. Normalerweise wäre dies keiner Erwähnung wert, gäbe es nicht ein gewisses Problem in Hinblick auf L4 Server mit multiplen Threads.
Ein L4 Server stellt Anderen Dienste zur Verfügung, in dem in der Regel ein beim Nameserver angemeldeter Thread auf eingehende IPC wartet. Die Clientanwendung führt, nachdem sie sich beim Nameserver die Thread ID des gewünschten Dienstes geholt hat, einen Call zu diesem Thread aus. Ein IPC Call bedeutet bei Fiasco jedoch, dass die Nachricht an den Thread zugestellt wird und auf eine Antwort von genau diesem Thread gewartet wird. Das heisst also insbesondere, dass kein anderer Thread die Antwort zurücksenden kann. Für einen L4 Server mit einem auf IPC wartenden Thread und mehreren anderen, die die eigentliche Arbeit verrichten, bedeutet das, dass ein zusätzliches, internes Kommunikationsprotokoll zwischen *Lauschendem* und *Arbeitenden* bestehen muss.

Als Teil dieser Arbeit wurde eine Bibliothek implementiert, die als generische Lösung dieses Problems gedacht ist. Sie kapselt zwei Warteschlangen von eingehenden Anfragen und ausgehenden Antworten an die Clients. Es gibt die Möglichkeit für den Lauscherthread eine neue Anfrage hinzuzufügen, sowie die älteste, noch nicht gesendete Antwort zu erhalten. Außerdem können Arbeiterthreads hinzugefügt und entfernt werden. Hierfür muss eine Funktion übergeben werden, die bestimmt, was mit einer einzelnen Anfrage geschehen soll. Die Schleife, in welcher verfügbare Anfragen mit Hilfe dieser Funktion abgearbeitet werden sowie die Realisierung des Informationsflusses hin zum Lauscher-Thread, sind Teil der Bibliothek und für die Entwicklerin oder den Entwickler transparent. Bei der Anwendung der Bibliothek muss lediglich darauf geachtet werden, dass der Lauscher-Thread zusätzlich zur gewünschten Schnittstelle die IPC-Calls der Arbeiter-Threads behandelt.
Eine derzeit in der Entwickelung befindliche Version der L4 API namens *L4 Sec* wird die soeben beschriebene Problemstellung beheben und die vorgestellte Bibliothek überflüssig machen, da in dieser Spezifikation Kommunikationsendpunkte definiert sind, die es erlauben, dass mehrere Threads gleichzeitig an ihnen lauschen. Anfragen werden dann nicht mehr an einzelne Threads, sondern an solche Kommunikationsknoten gesandt.

4.2 L4 HTTP Proxy

Der L4 HTTP Proxy wurde auf Basis der im vorangegangenen Abschnitt vorgestellten Bibliothek implementiert. Persistente Verbindungen wurden nicht implementiert, das heisst es wird nach jedem Request/Response-Paar die Verbindung zum Webserver geschlossen. Weiterhin existiert in der experimentellen Lösung nur ein festes SandboxLinux, an die alle HTTPS Anfragen geleitet werden und keine Verwaltung von Domains. Die Domainverwaltung wurde vorerst außen vor gelassen, weil es bisher keine Möglichkeit der Auflösung von Domainnamen in L4 gibt und daher der Proxy lediglich IP-Adressen aufrufen kann. Das Fehlen der Domainverwaltung und damit die feste Zuordnung aller HTTPS Anfragen an das eine SandboxLinux schränkt das vorangestellte Sicherheitskonzept ein, weil eine SSL Session die Vertraulichkeit späterer Verbindungen korrumpieren könnte. Da weiter davon ausgegangen werden muss, dass L4Linux und der verwendete Browser Schwachstellen besitzen, könnte eine bösartig konstruierte Webseite, die über HTTPS abrufbar ist, dazu gebraucht werden, den Webbrowser zu infiltrieren und vertrauliche Informationen folgender Verbindungen per HTTPS an den Angreifer zu übermitteln. Diese Schwachstelle muss in einer nicht experimentellen Version zuvor behoben werden, was jedoch auch keine Hürde darstellt.
Konkret werden beim Starten des L4 HTTP Proxy jeweils ein Proxy Stub in der unsicheren L4Linux Arbeitsumgebung und einer in einer L4Linux Instanz ohne Hardwarezugriff gestartet. Hierfür wird der L4Linux Application Loader bemüht, der beim Booten der jeweiligen Linux Instanz geladen wurde und über den Nameserver erreichbar ist. Die beiden Loader geben nach erfolgreichem Starten der Stubs deren Thread ID zurück. Die Thread IDs der gestarteten Stubs werden gespeichert und zur späteren Prüfung der Herkunft einer Anfrage verwendet.

Für die Realisierung der SSL Funktionalitäten wurde auf MatrixSSL zurückgegriffen. Hierbei stellte sich das Problem, dass aufgrund des derzeitigen Fehlens einer Dateisystemstruktur in L4 Zertifikate in die Anwendung einkompiliert werden müssen und kein dynamisches Laden von Zertifikaten zu diesem Zeitpunkt möglich ist.

4.3 L4Linux Proxy Stub

Der Proxy Stub ist analog zum Entwurf umgesetzt worden, wenngleich auch hier keine persistenten Verbindungen implementiert wurden. Dies wäre angesichts des Fehlens dieser in der Hauptanwendung auch nutzlos. Es wurde für das Parsing der HTTP Anfragen auf dieselbe, kleine, für diesen Zweck entwickelte HTTP Bibliothek zurückgegriffen wie beim L4 HTTP Proxy. Ebenso wurde MatrixSSL benutzt, um die SSL Verbindung hin zum Webbrowser umsetzen zu können.
Insgesamt war die Implementierung mit Hilfe der beiden Bibliotheken eine einfache und überschaubare Übung, es trat jedoch folgende Schwierigkeit auf. Der Stub muss sich beim HTTP Proxy authentifizieren. Dies geschieht am einfachsten und sichersten mit Hilfe seiner Thread ID, die bei einem IPC Call dem L4-Server sowieso übermittelt wird, damit dieser antworten kann. Der L4 HTTP Proxy kennt die Thread ID der Stubs, weil er sie mittels des Application Loader selbst gestartet hat, siehe Abschnitt 4.2. Allerdings arbeitet der L4Linux Proxy Stub aus Performancegründen mit mehreren Threads. Dies ist im Prinzip kein Problem, gibt es doch noch den gemeinsamen Kontext der Task bzw. des Adressraums, in welchem die Threads einer Anwendung stehen. So können zwei Thread IDs darauf geprüft werden, ob die assoziierten Threads der gleichen Task angehören. L4Linux Applikationen die per `fork()` System Call einen neuen Prozess generieren, erzeugen jedoch eine vollkommen neue Task. Die Authentifikation der Stubs müsste derart scheitern.
Das Problem liess sich mit dem System Call `clone()` umgehen, der es ermöglicht, Threads im selben Adressraum zu erzeugen.

4.4 L4Linux Application Loader

Der Application Loader läuft auf Basis von L4Linux, verhält sich aber gleichzeitig wie ein L4 Server. Der Haupt-Thread des Loaders wartet auf IPC-Calls von anderen L4 Anwendungen, mittels derer der Name, der zu startenden Anwendung, übermittelt wird. Ausserdem kann mit Hilfe eines zusätzlichen Parameters entschieden werden, ob die neue Anwendung als Hintergrundprozess gestartet werden soll. Die Möglichkeit Programme als Daemon zu laden, wird unter anderem beim Starten des L4Linux Proxy Stub durch den HTTP Proxy genutzt. Die L4 Thread ID des neu gestarteten Programms wird jeweils dem aufrufenden Client zurückgegeben. Die Rückgabe der Thread ID ist für den HTTP Proxy notwendig, damit er die Proxy Stubs korrekt identifizieren kann. Mit der Verfügbarkeit eines sicheren Authentifikationsmechanismus, so wie er in Abschnitt 3.5 beschrieben wurde, entfällt die Notwendigkeit des Rückgabewerts.
Erhält der Haupt-Thread des Loaders eine neue Anfrage, so erzeugt er zunächst einen neuen Kindprozess mittels `fork()`. Anschliessend muss der Kindprozess seine L4 Thread

ID mittels einer Pipe an den Elternprozess schreiben, da dieser sonst nicht in der Lage ist besagte ID anhand der Linux Process-ID zu ermitteln. Die L4 Thread ID kann nun per IPC an den Client zurückgesandt werden.
Soll das angeforderte Programm im Vordergrund laufen, so wird der Prozess, der aktuell Zugriff auf das Terminal besitzt gestoppt und an den Anfang einer Warteschlange eingefügt. Terminiert das im Vordergrund befindliche Programm, so wird der erste in der Warteschlange befindliche Prozess fortgesetzt und diesem das Terminal zugewiesen.

5 Auswertung

Das folgende Kapitel enthält Bewertungen der Sicherheitseigenschaften und der Performance der entwickelten Proxy Lösung.

5.1 Sicherheit

Der L4 HTTP Proxy ermöglicht Vertraulichkeit und Integrität sensitiver Informationen. Verfügbarkeit kann, angesichts der unvorhersagbaren Verfügbarkeit von Ressourcen im Internet, nicht gewährleistet werden. Darüberhinaus ist es in der derzeitigen Implementation möglich, aus dem unsicheren WorkingLinux heraus eine DoS Attacke durchzuführen, in dem der Proxy mit Anfragen überschüttet wird. Dieser Mangel könnte durch eine Auftrennung von verschiedenen Threads für unterschiedliche Clientklassen umgangen werden. Darüberhinaus können dem Proxy, dank der Kontrolle von CPU und Speicher durch die TCB, seine Ressourcen zugesichert werden.
Die Korrektheit der Implementation und die damit verbundene Sicherheit sollte durch eine deutliche Reduktion der Komplexität der Anwendungsteile erzielt werden, die Zugriff auf die sensitiven Informationen erhalten müssen. Ein Korrektheitsnachweis bleibt an dieser Stelle aus, ist aufgrund der wesentlich kleineren Codebasis aber prinzipiell vorstellbar.
Stattdessen soll Abbildung 5.1 den wesentlich geringeren Codeumfang des HTTP-Proxy sowie seiner TCB im Gegensatz zur herkömmlichen Linuxumgebung darstellen.
Der Proxy Stub, sowie der Application Loader sind als L4Linux Programme generell als vertrauensunwürdig einzustufen und fallen daher aus der Betrachtung heraus.
Wie bereits in Abschnitt 4.2 angesprochen, ist die Vertraulichkeit in der aktuellen Implementation nicht hundertprozentig gewährleistet, da aus der gekapselten L4Linux Sandbox heraus, SSL Verbindungen zu beliebigen Domains aufgebaut werden können.

unsichere Architektur	LOC	sichere Architektur	LOC
Mozilla Firefox	$>$ 1 000 000	HTTP Proxy	2 000
X Server	1 250 000	DOpE	15 000
minimal Linux	200 000	Fiasco	14 000
		L4 Env Basis	$\approx$ 55 000
		FLIPS	4 500
		MatrixSSL	9 200
total	**$>$ 2 450 000**	**total**	**$\approx$ 100 000**

Abbildung 5.1: SLOC der herkömmlichem Linuxumgebung im Vergleich zum entwickelten sicheren Szenario

	L4Linux direkt	L4Linux mit Proxy	Proxy + Instanzwechsel
HTTP	224	307	761
HTTPS	1057	700	1327

Abbildung 5.2: Mittlere Aufrufzeit in Millisekunden für HTTP und HTTPS Pakete im Vergleich

Um eine vollständige Vertraulichkeit herzustellen, müsste entweder für jede Domain oder benutzerdefinierter Domainklasse eine eigenständige L4Linux Instanz geladen, oder eine vertrauenswürdige Rendering Engine vollständig auf Basis von L4 entwickelt werden. Letztere Variante hätte den Vorteil, dass der Overhead durch Benutzung des Proxys entfallen würde.

5.2 Performance

Ein wichtiger Aspekt bei der Bewertung von Anwendungen, die höhere Sicherheitsanforderungen garantieren sollen, ist die Größe der Geschwindigkeitseinbuße. Es ist eine allgemein bekannte Tatsache, das Sicherheit Kosten verursacht, jedoch bedeutet ein stark spürbarer Leistungsverlust in der Regel, dass niemand das Programm in der Praxis nutzen wird.
Um die Leistung des Proxy Szenarios zu messen, wurde als Referenzpunkt eine L4Linuxinstanz mit direktem Zugriff auf die Netzwerkkarte gewählt, sprich die vormals benutzte unsichere Arbeitsumgebung. Gemessen wurde, wie lang eine Klartext und eine SSL verschlüsselte HTTP Anfrage einschließlich dem Empfang ihrer Antwort benötigt. Für die implementierte Lösung wurden zwei zusätzliche Werte ermittelt, um die Verzögerungen beim Wechsel von einer Linuxinstanz zur anderen bewerten zu können. Abbildung 5.2 zeigt die Messwerte in einer Tabelle.
Wie nicht anders zu erwarten ergibt sich, im Gegensatz zum direkten Zugriff, bei einem Klartext HTTP Request, der über den Proxy ausgeführt wird, eine geringfügige Verzögerung. Verwunderlich ist es jedoch, dass bei einer verschlüsselten Anfrage die Proxylösung schneller arbeitet, wobei hier die Verzögerung theoretisch noch viel größer sein müsste, da ja statt einer SSL Verschlüsselung zwei stattfinden, sowohl zwischen Proxy und Browser, als auch zwischen Webserver und Proxy. Dies lässt sich dadurch erklären, dass die für SSL Anfragen befugte L4Linuxinstanz in einer RAM-Disk laufen muss und somit schnellere Ladezeiten gegenüber der festplattennutzenden Instanz besitzt.
Ein merklicher Anstieg der Antwortzeiten erfolgt, wenn ein Instanzwechsel notwendig wird, wenn also entweder aus der unsicheren Umgebung eine verschlüsselte Verbindung aufgebaut werden soll, oder umgekehrt aus der sicheren eine unverschlüsselte. Der Anstieg ist zwar für die Nutzerin oder den Nutzer minimal, jedoch wurde für diese Messung mit wget ein sehr kleiner Webclient gewählt. Je größer der Client, desto erheblicher wird der Anstieg der Ladezeit. Um hier vorzubeugen, lässt sich eine Instanz des Webclient im voraus laden, noch bevor eine verschlüsselte Verbindung angefragt wird.

Abgesehen davon, wurde für die Messung auf eine geringe abzurufende Datenmenge zurückgegriffen. Je mehr Daten zu übertragen sind, desto weniger fällt die Verzögerung des Instanzenwechsels ins Gewicht und stellt sich für den Nutzer bzw. die Nutzerin als alltägliche minimale Schwankung beim Zugriff auf Ressourcen des Internet dar.

6 Zusammenfassung und Ausblick

Zusammenfassend lässt sich sagen, dass die sichere Nutzung von Webbrowsern, ohne Modifikation derselben, mittels des HTTP Proxy möglich ist. Die derzeitigen Performanceeinbußen sind zu verkraften und lassen sich durch das Hinzufügen einer Verwaltung für persistente Verbindungen im Proxy mindern. Ein noch höherer Gewinn ließe sich mit der Entwicklung eines minimalen Webbrowsers auf Basis von L4 erzielen. Damit fiel die zusätzliche SSL Ver- und Entschlüsselung zwischen Proxy und Webbrowser weg, die maßgeblich für den Performanceanstieg verantwortlich ist, allerdings auf Kosten der freien Browserwahl. Eine vergleichbare Komfortabilität wie sie beispielsweise Mozilla Firefox aufweist, ließe sich bei einer derartigen L4 Anwendung durch eine wohldefinierte Schnittstelle für sichere wie unsichere Plugins erreichen.
Ausgangspunkt einer solchen Lösung wäre eine minimale, vertrauenswürdige Rendering Engine, die die einfachsten HTML Konstrukte realisieren kann, sowie eine Komponente, die die erforderlichen HTTP Funktionalitäten umsetzt. Alternativ ließe sich ein minimaler Webbrowser oder Teile davon portieren. Anwärter hierfür wurden in Abschnitt 2.8 vorgestellt. Die Anwendung wäre dank ihrer geringen Komplexität leicht verifizierbar und könnte vollständig auf den Einsatz des Proxys verzichten, da ihr als vertrauenswürdige Anwendung zugestanden werden kann, direkt auf das Netzwerk zuzugreifen. Um im Hinblick auf den Komfort nicht hinter heutigen Browsern zurückzustehen, könnte die Anwendung gewisse Teile der Seite, Bilder, einzelne Frames oder ganze Seiten, die z.B auf Javascript aufbauen, von Plugins innerhalb einer Linuxumgebung rendern lassen. Die vertrauenswürdige Kernkomponente müsste lediglich den Speicherbereich des entsprechenden Bildschirmrechtecks dem Plugin übergeben und dafür Sorge tragen, dass der Bereich aussen herum mit einem Rahmen makiert wird, der die NutzerInnen über den eingeschränkt vertrauenswürdigen Inhalt in Kenntnis setzt. Sofern per HTTPS übertragene Daten vorliegen, müssten die Plugins abgeschaltet, oder anlog zur Proxy-Lösung domainspezifisch in einem SandboxLinux betrieben werden.
Abschliessend sollen die fehlenden Funktionalitäten des L4 HTTP Proxy beschrieben werden, die eine ausgereifte Anwendung notwendigerweise zur Verfügung stellen müsste. Zunächst ist die Implementation des Domain Name Service für L4 zu nennen, ohne die eine domainspezifische Rechteverwaltung nicht umsetzbar ist. Der Hintergrund dazu wurde bereits in Abschnitt 4.2 und 5.1 erläutert. Für eine ausgereifte Lösung mangelt es derzeit weiterhin an einem sicheren Speicherkonzept zur persistenten Verwaltung von Zertifikaten, sowie an vertrauenswürdigen Authentifikationsmechanismen zwischen einzelnen L4 Servern und einer sicheren Booting Infrastruktur.
Darüberhinaus wurde ein Problem in der hier vorliegenden Arbeit bisher vernachlässigt: Bei einem Instanzwechsel vom WorkingLinux zu einem SandboxLinux oder umgekehrt, oder von einem SandboxLinux zum anderen, wird aus der ursprünglichen Anfrage die URI extrahiert und in der jeweils anderen Instanz ein Webbrowser mit der entspre-

chenden URI als Argument aufgerufen. Dabei wird stillschweigend davon ausgegangen, dass bei der ursprünglichen Anfrage die GET-Methode benutzt wurde, zumindest wird der neugestartete Webbrowser vermittels dieser Methode versuchen, die von der URI referenzierte Ressource zu erhalten. Es ist jedoch durchaus denkbar, dass eine unverschlüsselt übertragene Webseite ein Formular beinhaltet, dessen Daten, nach Betätigung eines Button, mit der Methode POST verschlüsselt an den Webserver gesendet werden. In einem solchen Fall würde der Proxy nicht korrekt arbeiten. Allerdings wäre die Anwendung in der Lage diesen Fall zu Erkennen und dem Nutzer bzw. der Nutzerin den Umstand mitzuteilen. Da eine Webanwendung, die erst verschlüsselt überträgt, wenn die Daten mittels POST gesendet werden sollen, sowieso nicht vertrauenswürdig arbeitet, da bereits die ursprüngliche Formularseite gefälscht sein kann, könnte in einem solchen Fall der Zugriff, nach Rückfrage bei den NutzerInnen, von der unsicheren WorkingLinux Seite gestattet werden.
Ein anderes, derzeit nicht behandeltes Problem stellen Cookies dar, die auch bei einem Instanzenwechsel verfügbar sein sollten. Hier muss der Proxy Sorge tragen, dass wenn eine mit einem Cookie versehene Anfrage an eine andere Instanz umgeleitet werden soll, der neu gestartete Webbrowser das Cookie setzt.
Das vorliegende Konzept einer Application Firewall auf Basis eines vertrauensvollen Proxy lässt sich im Übrigen auf andere Internetdienste übertragen. Als Beispiele seien die verbindungsverschlüsselte Übertragung von E-Mail, diverse Chatprotokolle, oder SSH genannt.

Glossar

DOpE Desktop Operating Environment

DoS Denial of Service

GTK+ Gimp Toolkit

HTTP Hyper Text Transfer Protocol

HTTPS HTTP Secure

IPC Inter Process Communication

LOC Lines of Code

OWM Overlay Window Management

SSL Secure Sockets Layer

TCB Trusted Computing Base

TLS Transport Layer Security

URI Universal Resource Identifier

Literaturverzeichnis

[FH04] Feske, Norman und Christian Helmuth: *Overlay Window Management: User interaction with multiple security domains.* Technischer Bericht TUD-FI04-02-März-2004, TU Dresden, 2004.

[Här02] Härtig, Hermann: *Security Architectures Revisited.* In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[HPHS04] Hohmuth, Michael, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro: *Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors.* In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[pre05] *Afterburning and the accomplishment of virtualization.* Technical report, University of Karlsruhe, University of New South Wales and National ICT Australia, 2005.

[Res01] Rescorla, Eric: *SSL and TLS, Designing and Building Secure Systems.* Addison Wesley, 2001.

[Sch05] Schneier, Bruce: *Website of bruce schneier - software complexity and security.* URL: `http://www.schneier.com/crypto-gram-0003.html`, 2005.

[SH05] Singaravelu, Lenin and Christian Helmuth: *Making security-sensitive applications sensitive to security.* Technical report, Georgia Institute of Technology / TU-Dresden, 2005. Unveröffentlichtes Manuskript.

[WC] W3-Consortium: *Rfc2616 - http 1.1 specification.* URL: `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.