Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

COBRA
Component Based Routing Architecture

Amir Guindehi

Studentenarbeit SA-2001.30
Sommersemester 2001
Tutoren: Ralph Keller, Lukas Ruf
Supervisor: Prof. Dr. Bernhard Plattner

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

Einführung

Aktive Netzwerke (Active Networks) werden es in Zukunft ermöglichen, Netzwerke den eigenen Bedürfnissen nach zu programmieren, um neue Netzwerkdienste rasch und unkompliziert einführen zu können.

Die am TIK entwickelte Active Network Node Architektur erlaubt es, sogenannte *Plugins*, welche ausführbaren Code enthalten, zur Laufzeit dynamisch in den Betriebssystemkernel zu installieren. Damit wird ermöglicht, den Netzwerkknoten auf flexible Weise mit beliebiger Funktionalität zu erweitern. Zum Beispiel kann ein Benutzer ein Ver- und Entschlüsselungs-Plugin im Netz installieren, um zu erreichen, dass sein Datenverkehr verschlüsselt übertragen wird. Ein Netzwerkbenutzer hat somit die Möglichkeit, Plugins in Routers zu installieren, um einzelne Flows oder aggregierte Flows speziell nach seinen Bedürfnissen zu verarbeiten.

Linux bietet mit der Netfilter-Architektur eine elegante Möglichkeit, den Betriebssystemkernel zur Laufzeit mit beliebiger Netzwerk-Funktionalität zu erweitern. Spezifische Netzwerkkomponenten (wie NAT, SYN-Tracing etc.) wurden bereits erfolgreich mittels Netfilter implementiert.

Ziele

In dieser Arbeit geht es nun darum, eine Architektur basierend auf Netfilter zu entwickeln, die es erlaubt, beliebige Komponenten (Plugins) für aktive Netzwerke in den Betriebssystemkernel zu laden. Das Framework soll auf flexible Art und Weise ankommende Pakete einer entsprechenden Plugininstanz zuweisen können.

Dabei sollen die folgenden Kriterien erfüllt sein:
- **Flexibilität:** Pakete sollten flexibel einer Plugininstanz zugewiesen werden können
- **Effizienz:** Die Ausführung einer Plugininstanz sollte effizient gestaltet sein, der Overhead sollte vertretbar bleiben.
- **Integration:** Die Plugin-Architektur sollte optimal in Netfilter unter Linux integriert werden

Resultate

In dieser Semesterarbeit wurde ein Plugin Framework entworfen und implementiert das optimal in die bestehende Netfilter Architektur des Linuxkernels passt. Die Modularität des bestehenden Netfilter Frameworks ermöglichte eine gelungene Verschmelzung der beiden Frameworks.

Die an das Plugin Framework gesetzten Ziele wurden erreicht:
- Flows lassen sich flexibel an Plugins binden
- Das Plugin Framework ist gleich effizient wie das Netfilter Framework
- Die Integration in Linux / Netfilter scheint sehr gut gelungen zu sein
- Der Cobra Kernelcode ist übersichtlich und sehr modular
- Cobra Plugins sind einfach zu schreiben

Weiterführende Arbeit

Die nächsten Schritte sind die Implementation eines Plugin Servers und die Erweiterung des Cobra Frameworks um Remote Fähigkeiten, wie zum Beispiel die Möglichkeit, bei Bedarf Plugins aus dem Netz zu laden und von Remote-Rechnern aus die Flow / Plugin Assoziation zu verändern und zu konfigurieren.

Abstract

Introduction

In the future, Active Networks will allow users' requirements to be reprogrammed such that new network services can be implemented rapidly and efficiently.

The Active Network Node Architecture, designed and implemented at ETHZ by the Computer Engineering and Networks Laboratory (TIK), provides a framework which allows code plugins to be dynamically loaded and linked to the kernel code at runtime.

Network nodes can therefore be extended with a new functionality in a flexible way. For example, the user can install encryption and decryption plug-ins in a network for the encryption of data transfers. Using this new framework, the network user can install plug-ins in the routers such that individual or aggregate flows, can be processed according to pertinent needs.

Linux with its Netfilter architecture provides an elegant possibility for extending the operating system kernel during runtime with any number of network functionality. Specific network components (e.g. NAT, SYN tracing) have already been successfully implemented using Netfilter.

Aims & Goals

The goal of this work is to design and implement an active network framework based on the Linux Netfilter that will enable the dynamic loading of any number of active network plug-ins into the operating system kernel. The framework should allow the arriving packets to be assigned to the corresponding plug-in instance in a fast and flexible way.

The following criteria should be fulfilled:
- **Flexibility:** The routing of the packets to the assigned plug-in instance should be flexible.
- **Performance:** The execution of a plug-in instance should provide an efficient data path. The overhead of modularity should not seriously impact performance
- **Integration:** The plug-in architecture should be integrated optimally into Netfilter for Linux

Results

In this semester thesis, a plug-in framework which fits optimally into the existing Netfilter architecture for Linux, was designed and implemented. The modularity of the existing Netfilter framework allowed a permitted merger of both frameworks.

The targets set for the plugin framework were archived, namely:
- Flows can be bound flexibly to the plug-ins
- The Cobra plugin framework is as efficient as the Netfilter framework.
- The integration with Netfilter architecture and Linux seems to be very well successful
- The Cobra kernel code is clear and very modular
- Cobra plug-ins are easy to program and implement

Further Work

The next steps should be the implementation of a plug-in server and the extension of the Cobra framework to include remote features like loading plug-ins from network servers and configuring flows to plug-ins associations from remote servers.

Inhaltsverzeichnis

1. Einführung

1. 1. Motivation

Heutige Netzwerk-Infrastrukturen sind sehr kostspielig im Unterhalt. Netzwerkknoten (Roters) werden in rascher Folge installiert. Kaum installiert sollte die Software (bzw. die Betriebsysteme) dieser Knoten auch fliessend an die neuesten Protokolle angepasst werden.

Am Beispiel von IPv6, das 'IP Extentions', also Erweiterungen der Protokoll Headers schon im Protokollentwurf selbst vorsieht, sieht man, wie schnell und wie oft solche Protokollerweiterugen vorkommen können. Diese Erweiterungen können nicht vorhergesehen werden, sind also auch nicht in den aktuellen Softwareversionen und Betriebsystemversionen der installierten Router implementiert und werden somit von den aktuell installierten Knoten nicht unterstützt. Dies führt zwangsläufig zu veralteten Netzwerkknoten, die die neuesten Varianten der Protokolle nicht unterstützten.

Es sollte möglich sein, Netzwerkknoten zu entwerfen, die sich automatisch an die neuesten Protokolle anpassen können. Dies würde den Gesammtsupport von modernen Protokollen im Netzwerk stark verbessern.

Heutige Netzwerkknoten sind sehr monolitisch aufgebaut und aus diesem Grund nicht einfach erweiterbar. Es ist sehr aufwendig neue Protokolle zu implementieren oder alte Protokolle zu erweitern. Meistens muss die ganze Software bzw. das ganze Betriebsystem ausgewechselt werden.

Dies führt dazu, dass Protokollwechsel mehrere Jahre dauern, wie man auch beim Wechsel von IPv4 zu IPv6 gesehen hat!

1. 2. Problembeschreibung

In der Vergangenheit war die Hauptaufgabe eines Routers sehr einfach. Er musste auf Grund einer Zieladressentabelle Pakete auf die richtigen Interfaces weiterleiten.

Moderne Router haben allerdings mehrere Aufgaben zu erfüllen:
- Integrated / Differentiated Services [5]
- Erweiterte Routing Funktionalität: Level 3 und Level 4 Routing und Switching, QoS Routing, Multicast [5]
- Sicherheitsalgorithmen um VPN's - Virtuelle Private Netzwerke zu implementieren
- Erweiterungen zu existierenden Protokollen: RED - Random Early Detection
- Neue Kernprotokolle: IPv6 [5]

Die am TIK entwickelte Active Network Node Architektur [1] [2] [3] erlaubt es, sogenannte Plugins, welche ausführbaren Code enthalten, zur Laufzeit dynamisch in den Betriebsystemkernel zu installieren. Damit wird ermöglicht, den Netzwerkknoten auf flexible Weise mit beliebiger Funktionalität zu erweitern. Zum Beispiel kann ein Benutzer ein Ver- und Entschlüsselungs-Plugin im Netz installieren, um zu erreichen, dass sein Datenverkehr verschlüsselt übertragen wird. Ein Netzwerkbenutzer hat somit die Möglichkeit, Plugins in Routers zu installieren, um einzelne Flows oder aggregierte Flows speziell nach seinen Bedürfnissen zu verarbeiten.

Das in dieser Arbeit vorgeschlagene Framework wurde unter NetBSD entwickelt. Leider hat sich nun Linux in letzter Zeit viel schneller und viel weiter verbreitet als NetBSD.

Linux bietet seit der Kernel Version 2.4 mit der Netfilter-Architektur [4] eine elegante Möglichkeit, den Betriebsystemkernel zur Laufzeit mit beliebiger Netzwerk-Funktionalität zu erweitern. Spezifische Netzwerkkomponenten (NAT, Filtering, Mangeling, SYN-Tracing, Logging etc.) wurden bereits erfolgreich mit Netfilter implementiert.

1. 3. Ziel

In dieser Arbeit geht es nun darum, eine Architektur, basierend auf Netfilter [4] zu entwickeln, die es erlaubt, beliebige Komponenten (Plugins) für aktive Netzwerke in den Betriebsystemkernel zu laden. Das Framework soll auf flexible Art und Weise ankommende Pakete einer entsprechenden Plugininstanz zuweisen können.

Dabei sollen die folgenden Kriterien erfüllt sein:
- Flexibilität: Pakete sollen flexibel einer Plugininstanz zugewiesen werden können
- Effizienz: Die Ausführung einer Plugininstanz sollte effizient gestaltet sein, der Overhead sollte vertretbar bleiben
- Integration: Die Plugin-Architektur sollte optimal in Netfilter unter Linux integriert werden

Es soll ein allgemeingültiges Konzept entwickelt werden, welches es erlaubt, Plugins zu instanzieren und an Flows zu binden, so dass ankommende Pakete entsprechend verarbeitet werden können. Es soll eruiert werden, welche Erweiterungen an der bestehenden Netfilter-Architektur [4] für Linux nötig sind, um das Design umzusetzen. Das entworfene Design soll schlussendlich in Form eines oder mehrerer Netfilter-Modulen effizient und übersichtlich implementiert werden.

1. 4. Gliederung dieses Berichtes

Der vorliegende Bericht ist in 8 weitere Kapitel gegliedert.

Kapitel 2 führt auf einfache Art und Weise in das Konzept der Erweiterbaren Routerarchitektur ein, stellt die Netfilter Architektur [4] vor und hält fest, welche Funktionalitäten dem bestehenden Framework für unsere Belange fehlen. In Kapitel 3 wird die Netfilter Architektur genauer beleuchtet, die als Basis für die Implementation der Cobra Plugin Architektur verwendet wird. Kapitel 4 beschreibt dann die Cobra Architektur im Detail. In Kapitel 5 wird auf die Details der eigentlichen Cobra Implementation eingegangen. Kapitel 6 zeigt wie ein einfaches Cobra Plugin geschrieben wird. Daraufhin wird in Kapitel 7 die Benutzung des Cobra Frameworks am laufenden System demonstriert. Schlussendlich spricht dann Kapitel 8 über die Schlussfolgerungen, die aus dieser Entwicklung zu ziehen sind und zeigt weitere Ausblicke in zukünftige Weiterentwicklungen. In Kapitel 9 wurden die Anhänge zusammengefasst.

2. Erweiterbare Routerarchitektur

2. 1. Was sind erweiterbare Router?

Bis zum heutigen Tage besitzen Router typischerweise sehr monolitisch aufgebaute Betriebsysteme, die aus diesem Grund natürlich nicht einfach erweiterbar sind.

Mit der immer schnelleren Protokollentwicklung und der zunehmend schnelleren Markteinführung neuer Entwicklungen wird es immer wichtiger, dass Router Betriebsysteme dynamisch und zur Laufzeit erweitert und erneuert werden können.

Am TIK wurde in mehreren Arbeiten genau dieses Ziel verfolgt. Es wurde eine Active Network Node Architektur [1] [2] [3] entworfen und entwickelt. Das Hauptziel der vorgeschlagenen Architektur war es, ein modulares und erweiterbares System zu entwickeln, das das Konzept der Flows und die Fähigkeit bestimmte Komponenten abhängig von Flows auszuwählen besass.

Die Active Network Node Architektur [1] [2] [3] besitzt die folgenden Eigenschaften:

- Dynamisches Laden und Freigeben von Plugins zum Betriebsystemkern zur Laufzeit des Systems. Plugins sind Codemodule die eine spezifische Routerfuktionalität implementieren (z.B: Ver- und Entschlüsselung von Paketen). Nachdem ein Plugin geladen wurde ist es nicht mehr von anderem Kernelcode zu unterscheiden.
- Instanzierung von beliebig vielen individuellen Instanzen eines Plugins. Eine Instanz ist eine spezifische Laufzeitkonfiguration eines individuellen Plugins. Es ist öfters sehr wünschenswert mehrere Instanzen desselben Plugins im Kernel zu haben, wie beispielweise beim Paketscheduling. Dort kann ein und derselbe Paket Scheduler in verschiedenen Konfigurationen (und somit in verschiedenen Instanzen) mit verschiedenen Interfaces arbeiten. State-of-the-art Paket Schedulers sind normalerweise hierarchisch aufgebaut, mit möglicherweise verschiedenen Modulen, die in verschiedenen Hierarchielevels arbeiten. Unter den Knoten auf dem selben Level sind die Module unterschiedlich konfiguriert, können aber als Plugininstanzen ein und desselben Plugins koexistieren. Um ein einfaches und einheitliches Interface für die Allokation mehrerer Instanzen ein und desselben Plugins zur Verfügung stellen zu können, muss das Plugin auf ein Set von standardisierten Nachrichten reagieren können. Durch die Standardisierung dieser Nachrichten und durch die Implementation in allen Plugins, wird die interoperabilität verschiedener Plugins garantiert.
- Effizientes Mapping der individuellen Daten Paket Flows und die Möglichkeit Flows an bestimmte Plugininstanzen zu binden. Sets aus speziellen Flows werden normalerweise durch Filter spezifiziert. Zum Beispiel kann ein Filter genau den TCP Datenverkehr vom Netzwerk 172.16.7.0/24 zum Host 172.16.0.65 spezifizieren. Filter können auch individuelle Ende zu Ende Applikationsflows spezifizieren. Filter werden immer durch 6-Tupels spezifiziert: <Source Adresse, Destination Adresse, Protokoll, Source Port, Destination Port, Interface>. Jedes Element des 6-Tupels kann auch als nicht relevant markiert sein. Für das vorherige Beispiel wäre das 6-Tupel also so anzugeben: <172.16.7.0/24, 172.16.0.65/32, TCP, *, *, *>. Natürlich hat ein Filter für einen Ende zu Ende Applikationsflow alle Felder (ausser eventuell dem Interface Feld) ganz spezifiziert.
- Hoher Durchsatz im ganzen Datenpfad. Der hohe Durchsatz ist zum einen Teil durch die vollständige Kernelimplementation, die kostbare Kontext Switches verhindert, gegeben. Zum anderen Teil ist die schnelle Paketklassifikation nötig, um genügend schnell Pakete an Filter und Filter an Plugins zuordnen zu können.

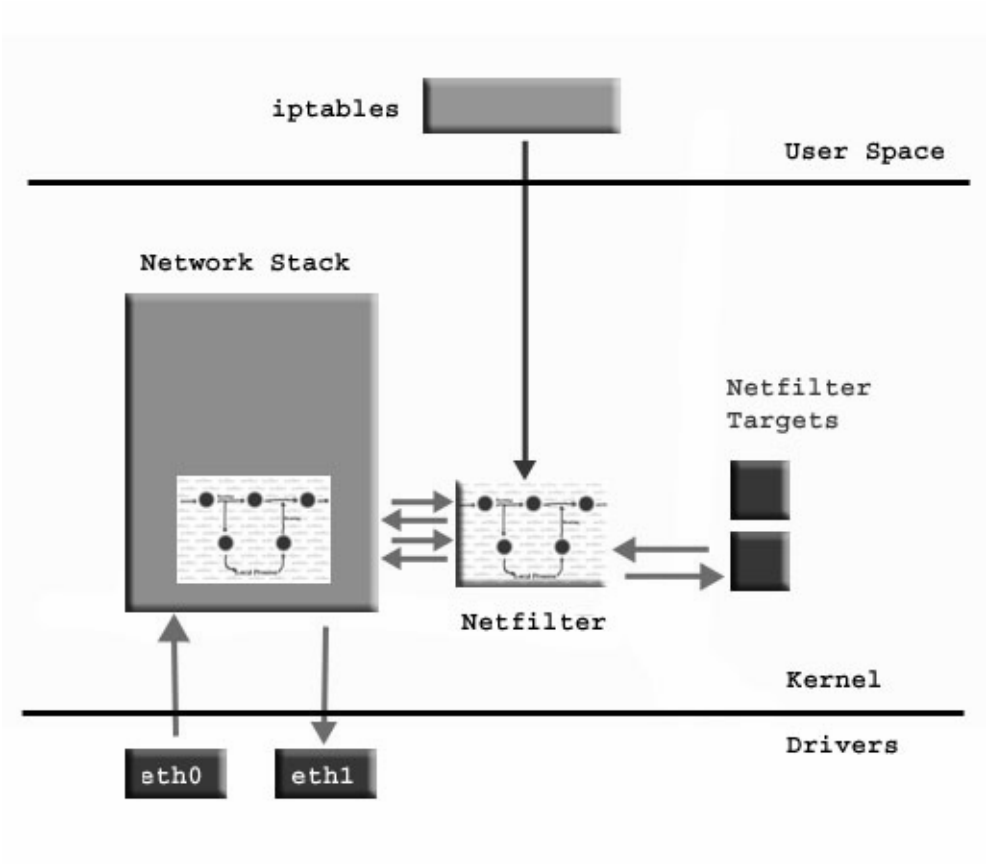
2. 2. Was ist die Netfilter Architektur?

Netfilter [4], entwickelt von Rusty Russel und seinem Netfilter Core Team, ist ein Framework zur Paketbehandlung, gehört aber nicht zu dem normalen Berkeley Socket Interface.

Jedes von Netfilter unterstützte Protokoll definiert ’Hooks’ (IPv4 definiert beispielsweise 5 Hooks), welches wohldefinierte Punkte auf dem Weg eines Pakets durch den Protokoll-Stack sind. An jedem dieser Punkte wird das Protokoll Netfilter mit dem Paket und der Hook-Nummer aufrufen. Teile des Kernels können sich an den verschiedenen Hooks für jedes Protokoll registrieren.

Wenn ein Paket also an Netfilter weitergereicht wird, wird überprüft, ob irgend jemand für dieses Protokoll oder für diesen Hook registriert ist; Wenn ja, bekommt jeder von ihnen der Reihe nach die Chance, das Paket zu untersuchen und es möglicherweise zu verändern, das Paket zu verwerfen, es durchzulassen oder Netfilter zu beauftragen, das Paket für den Userspace einzureihen. Die von Netfilter eingereihten Pakete werden gesammelt (von dem ip_queue-Treiber), um an den Userspace geschickt zu werden; diese Pakete werden asynchron behandelt.

Das Netfilter Framework [4] ist sehr modular aufgebaut.



Figur A: Netfilter Framework Gliederung

Zusätzlich zu diesem einfachen Rahmenwerk wurden verschiedene Module geschrieben, welche Funktionalitäten ähnlich zu früheren (pre-netfilter) Kernel bieten, im Besonderen ein erweiterbares NAT-System, und ein erweiterbares Paketfilter-System (iptables).

2. 3. Was fehlt?

Das Netfilter Framework [4] bildet einen wunderbaren Grundbaustein zur Implementation von dynamisch ladbaren Plugins. Da das Netfilter Framework eigentlich schon Plugins unterstützt, diese allerdings nicht zur Laufzeit zugefügt werden können und somit zur Compilierzeit schon vorhanden sein müssen, um statisch in das Framework eingebunden zu werden, sollte es sehr gut möglich sein, Netfilter dahingehend zu erweitern, dass es einen Dispatcher bekommt der genau diese Verwaltung und Verteilung zu zur Laufzeit geladenen Plugins übernimmt.

Um also dynamische, zur Laufzeit ladbare Plugins zu implementieren, müssen Netfilter Module geschrieben werden, die das Folgende implementieren:

- Es muss eine neue Netfilter Tabelle implementiert werden, in der man Filter registrieren kann, die es ermöglichen, Flows an geladene Instanzen eines Plugins zu binden
- Es muss ein neues Netfilter Target implementiert werden, das die Verwaltung und Verteilung der zur Laufzeit geladenen Plugins übernimmt. Dieses Target muss ankommende Pakete, die einem Filter entsprechen, an die entsprechende Plugininstanz weitergeben. Dieses Verteilen muss aus Performancegründen sehr schnell geschehen, da es für jedes Paket geschieht, das einem Plugin übergeben werden soll.
- Das Userspace Konfigurationstool ‘iptables’ muss erweitert werden, so dass die neuen Argumente des neuen Netfilter Targets verstanden werden.
- Es soll auch möglich sein abzufragen welche Plugins in wievielen Instanzen im Moment alloziert sind.

3. Netfilter Architektur

Das Netfilter Framework [4] besteht aus mehr als einer Serie von Hooks auf bestimmten Positionen in einem der Protokoll-Stacks von IPv4, IPv6 oder DECnet.

Die idealisierte IPv4 Paketreise sieht folgendermassen aus:

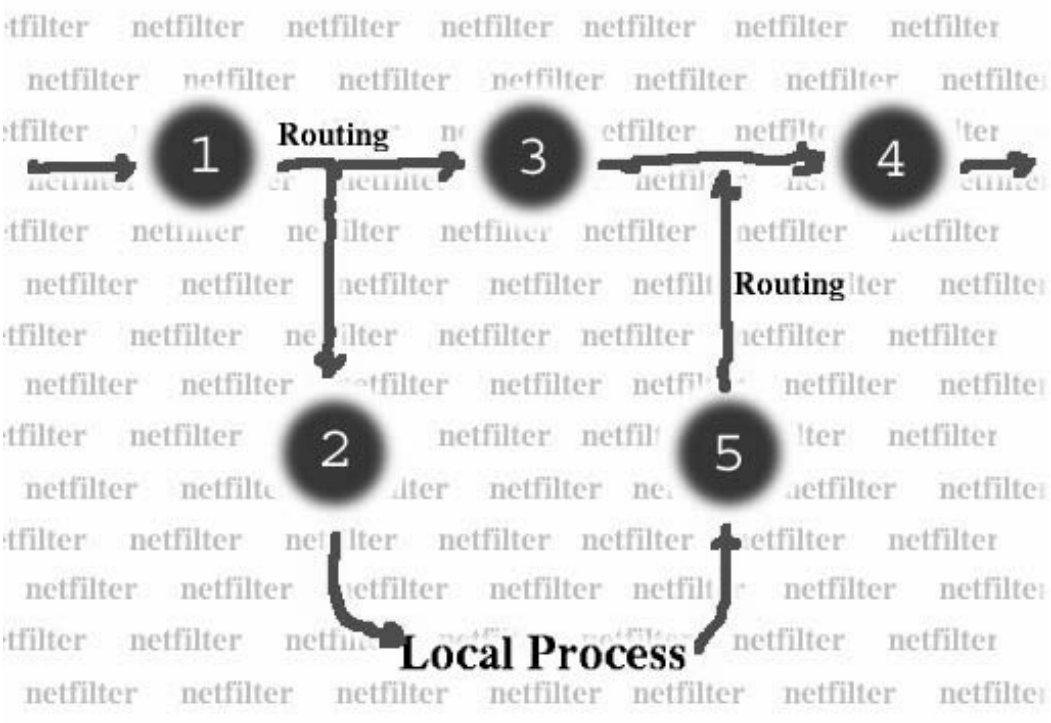


Bild B: Netfilter Paket Fluss

Das IPv4 Paket beginnt seine Reise von links. Den einfachen Sanity-Check überstanden (das heisst das Paket ist nicht verstümmelt, seine Checksumme ist OK und es ist kein Promiscuous Receive), werden sie an den NF_IP_PRE_ROUTING (1) Hook des Netfilter Frameworks weitergereicht.

Als nächstes werden die Pakete an den Routing-Code weitergereicht, welcher entscheidet, ob das Paket für eine andere Schnittstelle oder für einen lokalen Prozess bestimmt ist. Der Routing-Code kann unroutebare Pakete verwerfen.

Wenn das Paket für den lokalen Rechner selbst bestimmt ist, wird der NF_IP_LOCAL_IN (2) Hook das Netfilter Framework erneut aufrufen, bevor das Paket an den Prozess (falls es einen gibt) geschickt wird.

Wenn der Zielort stattdessen eine andere Schnittstelle ist, wird der NF_IP_FORWARD (3) Hook des Netfilter Frameworks aufgerufen.

Das Paket durchläuft dann einen letzten Netfilter Hook, den NF_IP_POST_ROUTING (4) Hook, bevor es wieder über ein Interface verschickt wird.

Für Pakete, die lokal generiert wurden, wird der NF_IP_LOCAL_OUT (5) Hook aufgerufen. Hier sieht man, dass das Routing erst dann einsetzt, wenn dieser Hook aufgerufen wurde: Tatsächlich wird zuerst der Routing-Code aufgerufen (um Angaben über Quellenadresse und IP-Optionen zu bestimmen), der dann erneut aufgerufen wird, falls das Paket geändert werden sollte.

3. 1. Netfilter Grundlagen

Die Essenz von Netfilter ist ganz klar die Aktivierung der Hooks beim Passieren der Pakete. Kernelmodule können sich registrieren, um an irgendeinem dieser Hooks zu lauschen. Wenn dieser Netfilter Hook dann vom Networking-Code aufgerufen wird, hat an diesem Punkt jedes registrierte Modul die Möglichkeit, das Paket zu verändern.

Das Modul kann Netfilter veranlassen, eine von mehreren Sachen zu tun:
1. NF_ACCEPT: Die Reise wie gewöhnlich fortsetzen.
2. NF_DROP: das Paket verwerfen, die Reise nicht fortsetzen.
3. NF_STOLEN: Das Modul hat das Paket übernommen, die Reise wird nicht fortgesetzt.
4. NF_QUEUE: Das Paket soll für den Userspace eingereiht werden.
5. NF_REPEAT: Dieser Hook soll erneut aufgerufen werden.

Auf diesem Fundament können ziemlich komplexe Paketfilter-Manipulationen aufgebaut werden.

3. 2. Paket Selektion: IP Tables

Ein System zur Paketauswahl mit dem Namen IP-Tables wurde über das Netfilter Framework gebaut. Es ist ein direkter Abkömmling von ipchains (welches von ipfwadm abstammt, das wiederum von BSD’s ipfw abstammt), mit Erweiterungen. Kernelmodule können eine neue Tabelle registrieren und von einem Paket verlangen, eine vorgegebene Tabelle zu durchwandern. Diese Methode zur Paketauswahl wird für Paketfilter (die ’Filter’-Tabelle), Network Address Translation (die ’NAT’-Tabelle) und allgemeine Behandlung von pre-routing Paketen (die ’mangle’-Tabelle) verwendet.

3. 3. Die Netfilter-Tabellen

3. 3. 1. Paket Filtering: Die ‘filter’ Tabelle

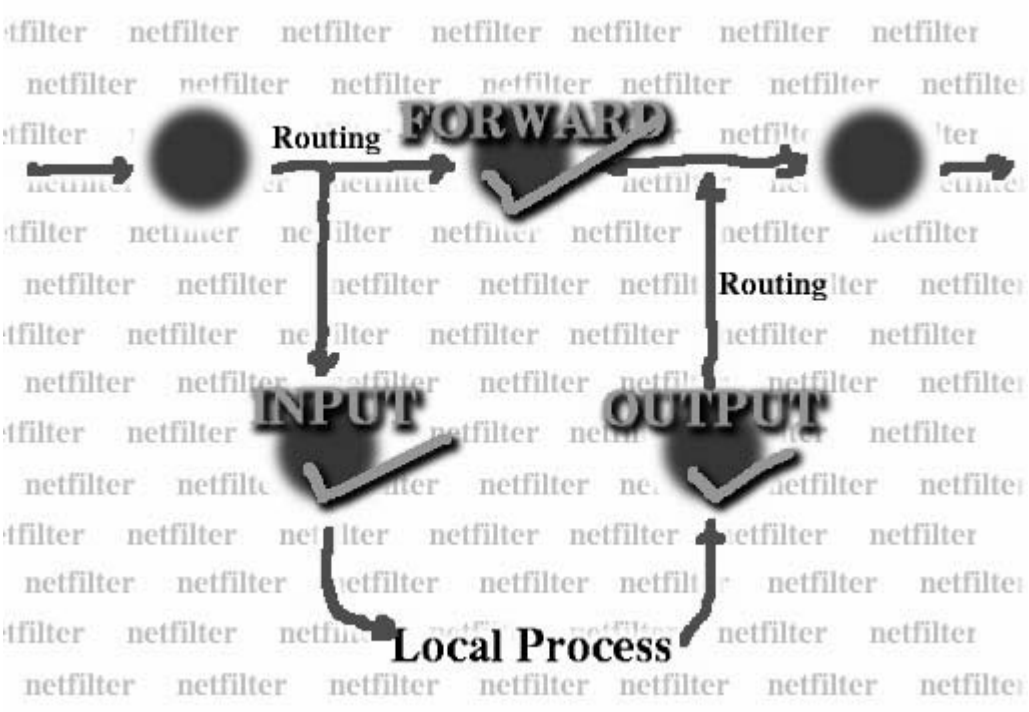


Bild C: Hooks der ‘filter’ Tabelle

Die ’filter’ Tabelle sollte die Pakete niemals verändern: sie soll sie nur filtern.

Einer der Vorteile von iptables Filtern gegenüber ipchains ist, dass sie klein und schnell sind, und die Netfilter-Hooks NF_IP_LOCAL_IN, NF_IP_FORWARD und NF_IP_LOCAL_OUT verwenden.

Das bedeutet, dass es für jedes gegebene Paket einen (und nur einen) möglichen Ort gibt, um es zu filtern. Dies vereinfacht die Dinge ungemein. Ausserdem bedeutet der Fakt, dass das Netfilter Framework beides, eingehende und ausgehende Schnittstellen für den NF_IP_FORWARD Hook bietet, dass viele Arten des Filterns weitaus einfacher werden.

3. 3. 2. Network Address Translation: Die ‘nat’ Tabelle

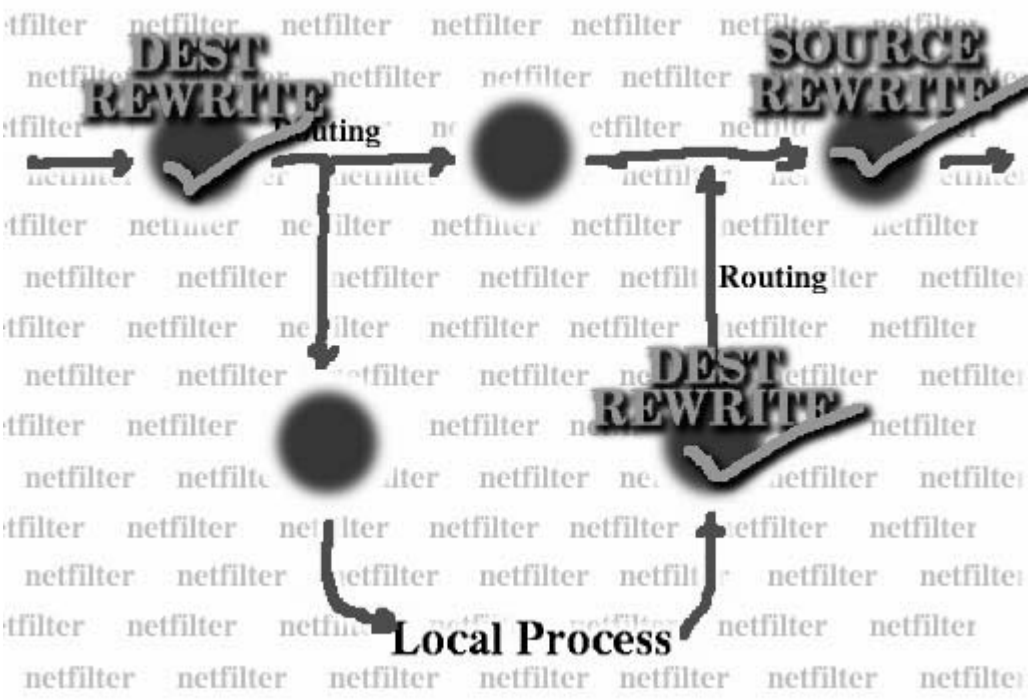


Bild D: Hooks der ‘nat’ Tabelle

Die NAT-Tabelle wird mit Paketen von drei Netfilter-Hooks gespiesen: Bei nicht-lokalen Pakete wird für Quell- und Zielveränderungen jeweils der NF_IP_PRE_ROUTING und der NF_IP_POST_ROUTING Hook benutzt. Um das Ziel von lokalen Paketen zu verändern, wird der NF_IP_LOCAL_OUT Hook verwendet.

Diese Tabelle unterscheidet sich insofern leicht von der ’filter’-Tabelle, als dass nur das erste Paket einer neuen Verbindung die Tabelle durchwandern wird: Das Resultat dieser Untersuchung wird dann auf alle weiteren Pakete derselben Verbindung angewandt.

Man unterteilt NAT in Source NAT (wo die Quelle des ersten Pakets verändert wird) und in Destination NAT (wo das Ziel des ersten Pakets verändert wird). Masquerading ist eine spezielle Form von Source NAT: Port-Forwarding und transparente Proxys sind eine spezielle Form von Destination NAT.

3. 3. 3. Paket Mangling: Die ‘mangle’ Tabelle

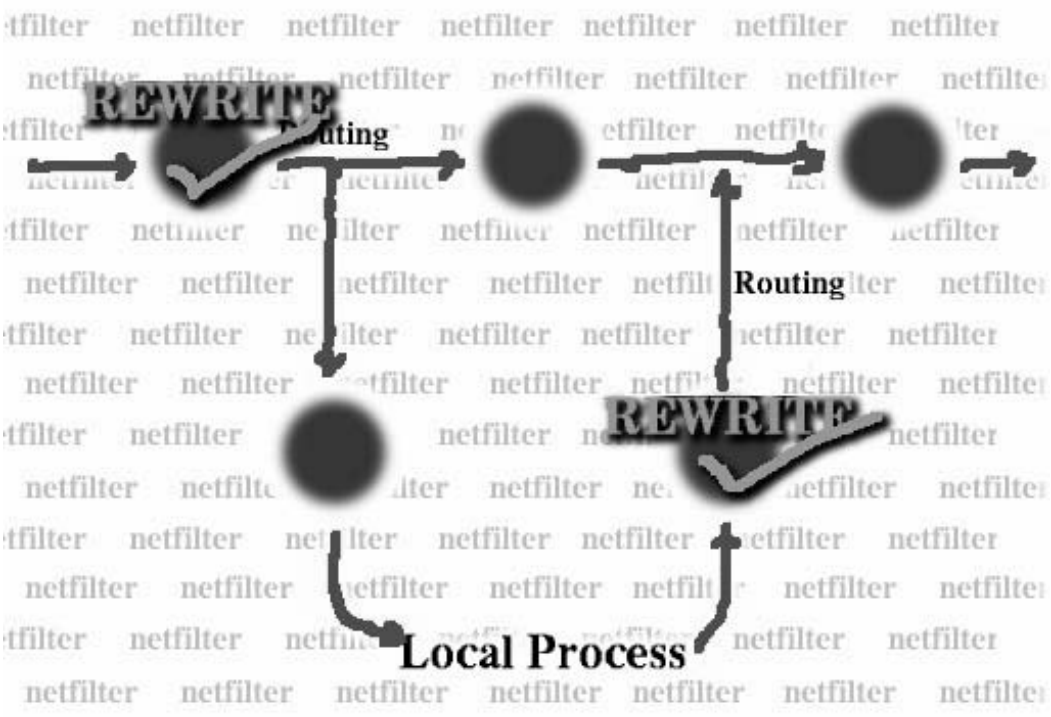


Bild E: Hooks der ‘mangle’ Tabelle

Die Mangle-Tabelle wird benutzt um beliebige Paket Informationen zu ändern.

Sie benutzt die Hooks NF_IP_PRE_ROUTING und NF_IP_LOCAL_OUT und sieht somit jedes Packet, das durch den Stack wandert, egal ob lokal generiert oder von aussen kommend.

Es gibt Bestrebungen die Mangle-Tabelle in alle 5 Hooks einzufügen und es ist auch ein Patch vorhanden, um genau dies zu tun. Aus Performance-Gründen ist dieser Patch allerdings nur separat verfügbar, da es in den meisten Fällen genügt, die Mangle-Tabelle nur in den zwei oben genannten Hooks zu verwenden.

3. 3. 4. Connection Tracking

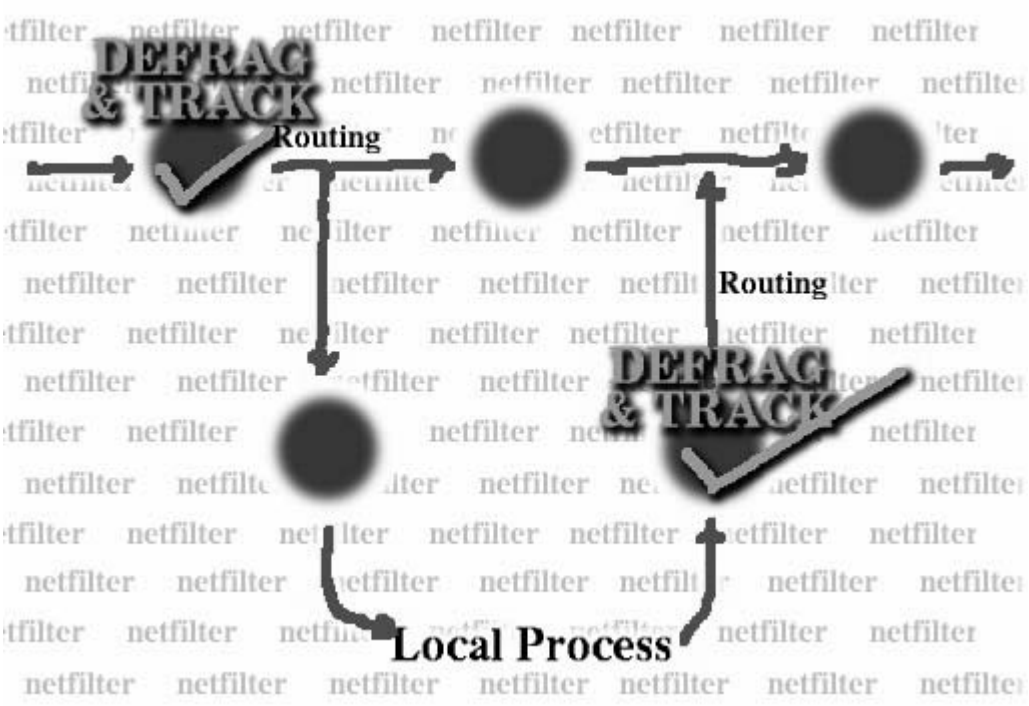


Bild F: Hooks des Connection Trackings

Connection Tracking ist ein fundamentaler Bestandteil der Network Address Translation (NAT), wurde aber als separates Modul implementiert. Dies erlaubt allfälligen Erweiterungen des Paket Filter Codes das Connection Tracking in einer sauberen und einfachen Art und Weise zu nutzen, wie man beispielsweise am State Modul sieht.

Um das Connection Tracking zu implementieren, müssen natürlich alle neu in den Stack eintretenden Pakete defragmentiert werden. Erst mit vollständig zusammengesetzen Paketen ist es möglich, die Zustände der einzelnen Verbindungen zuverlässig mit zu protokollieren.

4. Cobra Plugin Architektur

4. 1. Netfilter Erweiterungen - Cobra Plugins

Um das Netfilter Framework [4] um Cobra Plugins zu erweitern, müssen einige neue Module geschrieben.

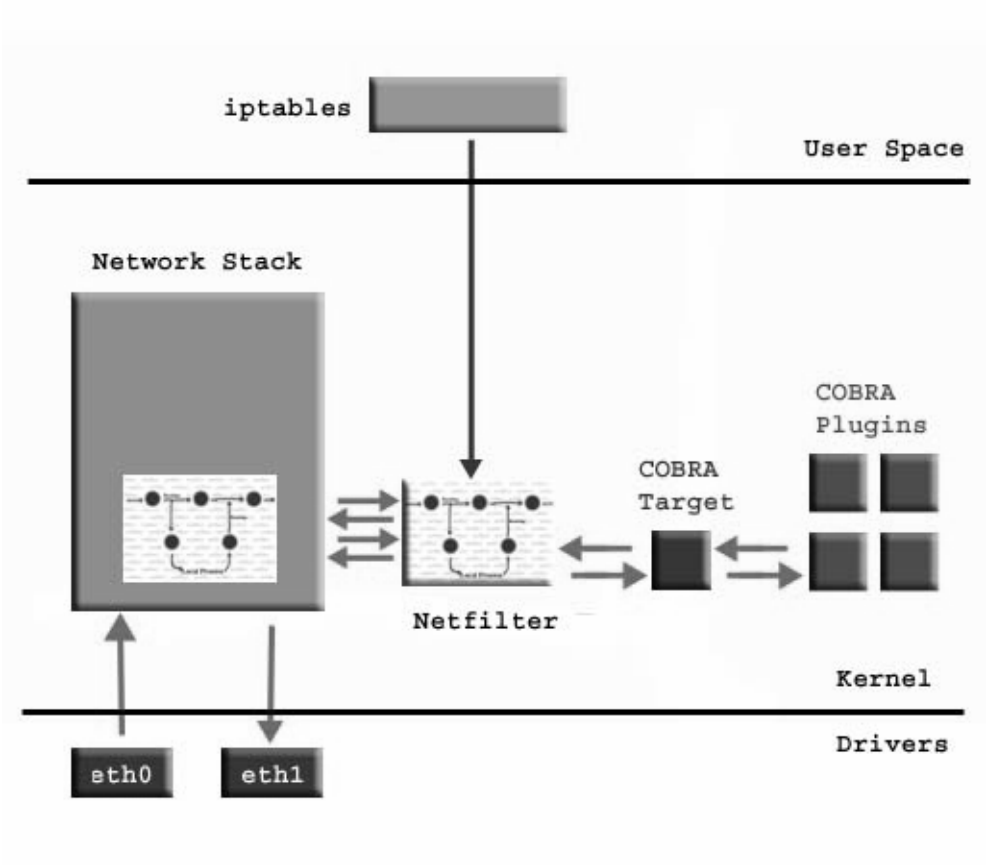


Bild G: Cobra Framework Gliederung

Im Speziellen muss sowohl ein Paket Dispatcher und Plugin Verwalter als neues Netfilter Target, wie auch eine neue Netfilter Tabelle mit den Filtern für die Flows als Modul implementiert werden. Ein Framework für das Design der Cobra Plugins muss ebenfalls entworfen werden.

Zusätzlich muss das Userspacetool ‘iptables’ noch um die Optionen für die neuen Funktionalitäten erweitert werden, was durch Hinzufügen einer Shared Library möglich ist.

4. 1. 1. Cobra Netfilter-Tabelle

Um Cobra Flows und normale Filter sauber zu trennen und sie auch in allen 5 IPv4 Hooks registrieren zu können, implementieren wir eine neue Netfilter-Tabelle mit dem Namen ‘cobra’ in Netfilter.

Die ‘cobra’ Tabelle muss sich beim Laden des Modules im Netfilter Framework anmelden und beim Entfernen auch wieder abmelden.

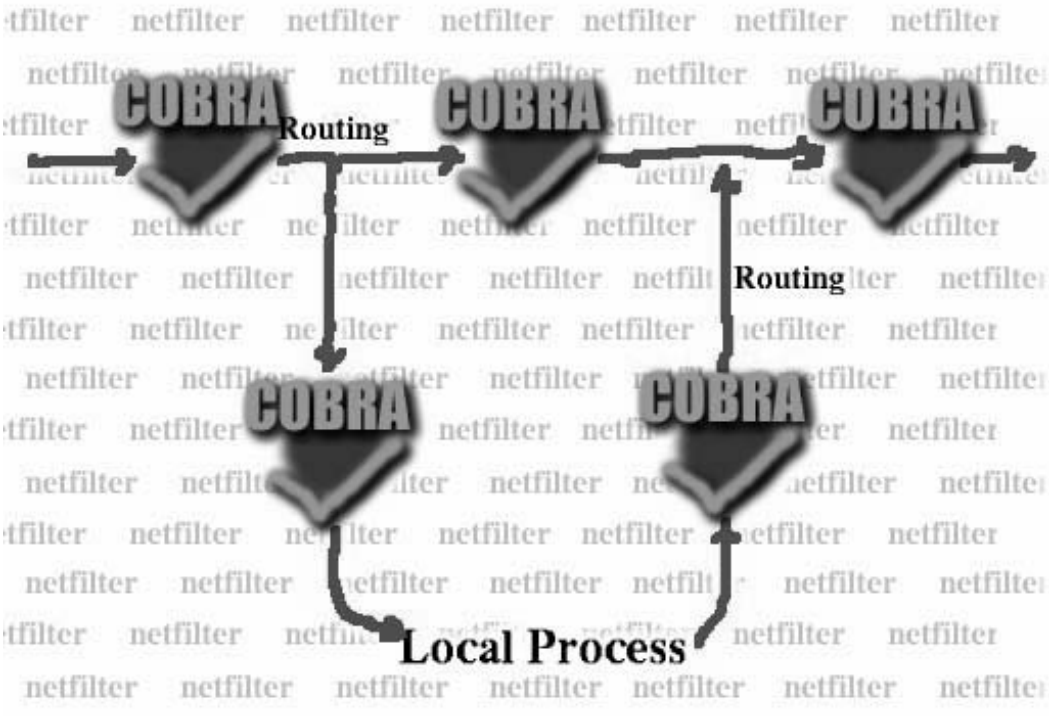


Bild H: Die Hooks der ‘cobra’ Tabelle

Da es Cobra Plugins möglich sein soll, an jeder Stelle im Netzwerk Stack aktiv zu werden, muss sich die neue Tabelle beim Initialisieren in allen 5 IPv4 Netfilter Hooks registrieren. Des weiteren muss die Tabelle diese fünf Hooks auch wieder löschen, wenn das Tabellen Module wieder aus dem Kernel entfernt wird.

Sollte das Cobra Framework einmal performancemässig den Anforderungen nicht standhalten können, so ist es möglich, die in dieser Anwendung unbenutzten Hooks zu entfernen und so das Framework durch Ausschalten der unnötigen Aufrufe zu entlasten und zu beschleunigen.

Die ‘cobra’ Tabelle wird als separates Netfilter-Tabellen Kernel Modul implementiert, das auch die nötigen Datenstrukturen zur Verwaltung der Tabelle registriert.

4. 1. 2. Cobra Netfilter-Target

Flow Einträge in der ‘cobra’ Tabelle zeigen auf ein spezielles Netfilter-Target, das ‘COBRA’ heisst.

Dieses Target implementiert die Verwaltungstabellen, die nötig sind, um zu vermerken, welche Cobra Plugins geladen sind und wieviele Instanzen von ihnen initialisiert wurden.

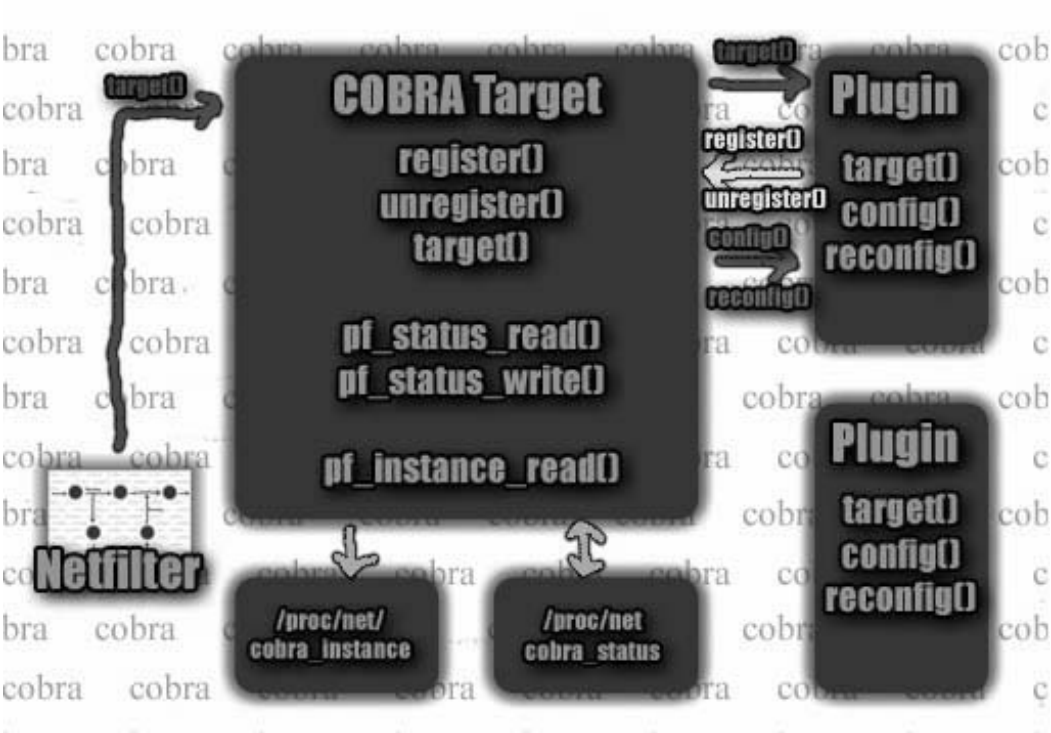


Bild I: Das ‘COBRA’ Target

Dazu implementiert das Target zwei exportierte Funktionen. Die erste dieser Funktionen wird von jedem sich initialisierenden Cobra Plugin zur Registrierung desselben aufgerufen, die zweite wird von jedem beendenden Cobra Plugin zur Deregistrierung aufgerufen. Auf diese Weise ist sichergestellt, dass das Cobra Target jederzeit über alle geladenen und instanzierten Cobra Plugins informiert ist.

Das Target erstellt eine Instanzdatei im /proc Dateisystem des Kernels. Dies ist nötig, da sichergestellt werden muss, dass neue Instanzen eines Plugins auch eine garantiert eindeutige Instanznummer erhalten. Um dies trotz Concurrent Programming garantieren zu können, wurde diese Aufgabe dem nur einmal im ganzen System existierenden Cobra Target übertragen. Ausser in Ausnahmefällen wird das Target die nächste freie Instanznummer ermitteln, und dies dem Userspacetool ‘iptables’ über die Instanzdatei im /proc Dateisystem mitteilen. Der Ausnahmefall ist der Fall, bei dem eine Plugin-Instanz Pakete von zwei verschiedenen Flows verarbeiten soll. Dann soll natürlich beim erstellen des zweiten Flow Eintrags in der ‘cobra’ Flow Tabelle keine neue Instanz des Plugins erstellt werden, sondern es soll die schon existierende Instanz des Plugins auch für die Pakete dieses neuen Flows aufgerufen werden.

Bei der Registrierung eines Cobra Plugins teilt das Plugin dem Target alle Informationen mit, die nötig sind, damit der Dispatcher Teil des Targets allfällige ankommende Pakete für eine Instanz eines Plugins an das Plugin übergeben kann.

Ebenfalls mitgeteilt wird welche Funktionen des Plugins zur Initalkonfiguration und zur Laufzeitkonfiguration vom Target aufgerufen werden sollen. Das Target initialisiert, nach der Registrierung eines neuen Moduls, dasselbe mit den, dem Userspacetool angegebenen Daten. Das Cobra Target erstellt auch eine Status Datei im /proc Dateisystem des Kernels. Über diese Status-Datei ist es möglich, ein Plugin zur Laufzeit neu zu konfigurieren, beziehungsweise dem Plugin beliebigste Daten zukommen zu lassen. Durch Schreiben in die Status-Datei im /proc Dateisystem kann, durch Angabe der Instanznummer und der zu übergebenden Daten, ein Plugin jederzeit neu konfiguriert werden.

Des weiteren ist es möglich, durch Lesen dieser Status Datei im /proc Dateisystem beim Cobra Target nachzufragen, welche Plugins in wievielen Instanzen alloziert sind.

Das neue Cobra-Target wird ebenfalls als separates Netfilter-Target Kernel Modul implementiert.

4. 1. 3. Shared Library für das Userspace Tool ‘iptables’

Das Userspacetool ‘iptables’ muss erweitert werden. Die neuen Flags für das neue Cobra Target wie auch ihre Hilfe-Texte müssen implementiert werden.

Des weiteren ist es Aufgabe des Userspacetools das entsprechende Cobra Plugin zu Proben und gegebenfalls zu Laden.

Ebenfalls Aufgabe des Userspacetools ist es, bei entsprechenden Kommandozeilen-Parametern die Instanzdatei des Cobra Targets im /proc Dateisystem zu lesen und eine neue, im ganzen System einheitliche Instanznummer für die Instanzierung der neuen Plugininstanz zu verwenden.

Diese Erweiterungen des Userspacetools erfolgen durch Hinzufügen einer Shared Library.

4. 2. Design Ziele

4. 2. 1. Flexibilität

Ziel unseres Designs ist es, so flexibel wie möglich zu sein.

Das Design des Cobra Frameworks soll die Entwicklung zukünftiger Cobra Plugins nirgends einengen und es sollte einem Plugin alles möglich sein, was ihm möglich wäre, wenn es als eigenständiges Kernel Modul geschrieben worden wäre.

Dies wird durch folgende Punkte gewährleistet:

- Cobra implementiert zwei neue Kernel Module init() und exit() Makros, die das An- und Abmelden beim Cobra Target kapseln. Trotz dieser neuen Makros ist es möglich, eigene init() und exit() Funktionen zu implementieren und diese den Makros mitzugeben. Sie werden vom Cobra Framework zur richtigen Zeit ausgeführt.
- Cobra Plugins sind ganz normale Kernel Module. Sie enthalten lediglich 3 spezielle Funktionen: Eine target() Funktion, die aufgerufen wird, wenn Pakete einen auf eine Instanz dieses Plugins zeigenden Flow erfüllen, eine config() Funktion, die bei der Initialisierung einer neuen Instanz des Plugins aufgerufen wird und eine reconfig() Funktion, die bei Laufzeitkonfiguration über die Status Datei des Cobra Targets im /proc Dateisystem aufgerufen wird.
- Cobra Plugins können jederzeit auch unabhängig vom Cobra Framework geladen und entfernt werden, solange beim Laden des Plugins auch das Cobra Target schon geladen ist und beim Entfernen keine Flows mehr auf Instanzen des Plugins zeigen. Sollten noch Flows auf Instanzen des Plugins zeigen wenn es entfernt werden soll, so wird der Kernel sich weigern das Plugin zu entfernen.
- Cobra Plugins haben somit einen sehr simplen und klaren Aufbau.
- Dadurch dass garantiert wird, dass Cobra Plugins immer das gleiche einfache Interface besitzen, sind sie universell einsetzbar und können von System zu System migriert werden, solange die Systeme Binärkompatibel sind.
- Cobra Plugins können zur Laufzeit einem Cobra Kernel hinzugefügt werden. Dies ist für eine eventuell erfolgende spätere Ausbaustufe nötig, bei der die Cobra Plugins dann über das Netzwerk von einem Cobra Pluginserver installiert werden.
- Cobra Plugininstanzen haben garantiert eindeutige Instanz-Nummern im ganzen System. Dies ermöglicht die eindeutige Identifizierung eines Plugins und ist auch Voraussetzung für allfällige Remote-Operationen auf den Plugins.

4. 2. 2. Effizienz

Das Cobra Framework muss effizient implementiert werden. Da das Framework zum Router Design verwendet werden soll, muss garantiert werden, dass die höchstmögliche Performance erreicht wird.

Dies wird durch folgende Überlegungen gewährleistet:
- Die Verwaltung jeglicher Tabellen wird mittels Hash Datenstrukturen implementiert. Dies garantiert eine sehr effiziente Suche nach den Einträgen der Tabelle. Sollten die Hashes noch mehr beschleunigt werden, so wäre es noch möglich, sie durch doppeltes Hashing noch schneller zu machen.
- Das Netfilter Framework wurde schon unter Betrachtung der Effizienz entwickelt.
- Das Cobra Framework führt lediglich einen Hash Tabellen Lookup und einen einzigen weiteren Funktionsaufruf, der in das Plugin verzeigt, pro Paket ein. Dies sollte vertretbar sein da das Netfilter Framework bedeutend mehr Funktionsaufrufe pro Paket hinzufügt. Man könnte also eigentlich sagen, dass das Cobra Framework in etwa gleich schnell ist wie das Netfilter Framework.
- Alle anderen Features des Cobra Frameworks liegen nicht auf dem Datenpfad eines Paketes und beeinflussen somit nicht die Overall Performance des Frameworks.

4. 2. 3. Integration

Die Integration der Cobra Plugins in das schon bestehende Netfilter Framework ist von grösster Wichtigkeit. Je besser und je vollständiger sie gelingt, desto mehr ist ihre Langlebigkeit garantiert. Die Migration zu neuen Kernel Versionen ist so ebenfalls gesichert.

Die folgenden Merkmale sind Zeichen der optimalen Integration des Cobra Frameworks:
- Alle Features des Netfilter Frameworks [4] werden im Cobra Framework optimal verwendet.
- Das Cobra Framework besteht aus einem Netfilter Target Modul, einem Netfilter-Tabellen Modul und einer Netfilter Shared Library Extention zum Userspacetool ‘iptables’. Dies garantiert eine vollständige Integration in das Netfilter Framework und seine Philosophie.
- Cobra Plugins sind ganz normale Kernel Module. Sie fügen sich nahtlos in das bestehende Linux Kernel Modul Konzept ein. Ihre Abhängigkeiten können mit den Standardtool ‘depmod’ ermittelt werden und Cobra Plugins können mit dem jedem Linux beiliegenden Standardtool ‘modprobe’ geladen werden. Die Plugins können auch jederzeit (falls nicht mehr durch die Flow Tabelle referenziert) mit dem Standardtool ‘rmmod’ wieder aus dem Kernel entfernt werden. Einen Überblick über die geladenen Module (nicht ihrer Instanzen) kann mittels des Tools ‘lsmod’, wie bei allen Kernel Modulen, gewonnen werden.
- Cobra Plugins sind unabhängig von der aktuellen Kernel Version und vom eigentlichen Kernel und können somit frei von System zu System kopiert werden, falls der Zielkernel mit der Option ‘Set version information on all module symbols’ kompiliert wurde. Diese Option ist im Menu ‘Loadable module support’ des Linux Kernels zu finden. Sie garantiert die Kernelversionsunabhängigkeit von Kernel Modulen.

5. Cobra Plugin Implementation

5. 1. Cobra Netfilter-Tabelle: iptable_cobra.c

Die ‘cobra’ Tabelle wird im File linux/net/ipv4/netfilter/iptable_cobra.c implementiert. Im weiteren Text wird deren Implementation im Detail erläutert.

Das Define COBRA_VALID_HOOKS definiert die Hooks, von denen die Tabelle aufgerufen werden soll. Da Cobra möglichst flexibel sein soll und somit den Plugins alle Möglichkeiten offen lassen soll, registriert sich die Tabelle bei allen 5 IPv4 Netfilter Hooks:

```
#define COBRA_VALID_HOOKS ((1<<NF_IP_PRE_ROUTING) | (1 << NF_IP_POST_ROUTING)
| (1 << NF_IP_LOCAL_IN) | (1 << NF_IP_FORWARD) | (1 << NF_IP_LOCAL_OUT))
```

Die Struktur packet_cobra beinhaltet alle Informationen, die das Netfilter Framework braucht, um eine neue Tabelle zu registrieren. Sie definiert den Namen der Tabelle, die erlaubten Hooks die durch COBRA_VALID_HOOKS definiert werden, wie auch den Initialzustand der ‘cobra’ Tabelle in Form eines Pointers auf die Datenstruktur.

```
static struct ipt_table packet_cobra = {
                                        { NULL, NULL },
                                        "cobra",
                                        &initial_table.repl,
                                        COBRA_VALID_HOOKS,
                                        RW_LOCK_UNLOCKED,
                                        NULL
                                      };
```

Diese Struktur wird beim Registrieren der Tabelle mit der Funktion ipt_register_table() in der init() Funktion benutzt, um dem Netfilter Framework die benötigten Informationen mitzuteilen.

```
ret = ipt_register_table(&packet_cobra);
```

In der Struktur ipt_ops[] werden die aufzurufenden Funktionen für jeden Hook, wie auch die einzelnen Hook Prioritäten definiert.

```
static struct nf_hook_ops ipt_ops[] =
{
  { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_PRE_ROUTING, NF_IP_PRI_COBRA },
  { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_POST_ROUTING, NF_IP_PRI_COBRA },
  { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_LOCAL_IN, NF_IP_PRI_COBRA },
  { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_FORWARD, NF_IP_PRI_COBRA },
  { { NULL, NULL }, ipt_hook, PF_INET, NF_IP_LOCAL_OUT, NF_IP_PRI_COBRA }
};
```

Diese Struktur wird in der init() Funktion des ‘cobra’ Tabellen Modules verwendet, um sie bei den einzelnen Hooks zu registrieren.

```
ret = nf_register_hook(&ipt_ops[0]);
```

In der exit() Funktion des Modules werden sowohl die Hooks wie auch die ‘cobra’ Tabelle selbst wieder abgemeldet.

```
for (i = 0; i < sizeof(ipt_ops)/sizeof(struct nf_hook_ops); i++)
     nf_unregister_hook(&ipt_ops[i]);
ipt_unregister_table(&packet_cobra);
```

Schlussendlich werden die beiden Makros module_init() und module_exit() mit den eigentlich init() und exit() Funktionen des ‘cobra’ Tabellen Modules als Parameter aufgerufen, wie es für jedes Kernel Modul nötig ist.

```
module_init(init);
module_exit(fini);
```

Die eigentliche Verarbeitung ankommender, von den Hooks an die Tabelle weitergegebenen Pakete erfolgt in der Funktion ipt_hook(). In dieser Funktion wird das ankommende Paket über die Funktion ipt_do_table() an das Netfilter Framework weitergereicht, das dann anhand der vorhandenen Tabelleneinträge entscheidet durch welches Target das Paket weiterverarbeitet werden soll.

```
static unsigned int
ipt_hook(unsigned int hook,
         struct sk_buff **pskb,
         const struct net_device *in,
         const struct net_device *out,
         int (*okfn)(struct sk_buff *))
{
  return ipt_do_table(pskb, hook, in, out, &packet_cobra, NULL);
}
```

5. 2. Cobra Netfilter-Target: ipt_COBRA.c

Das ‘COBRA’ Target wird in der Datei linux/net/ipv4/netfilter/ipt_COBRA.c implementiert. Im weiteren Text wird dessen Implementation im Detail erklärt.

Das Cobra Target Modul implementiert zwei separate Hash Tabellen.

```
static struct target_table cobra_table;
static struct instance_table cobrai_table;
```

Die zugehörigen Strukturen werden aus der Header Datei ipt_COBRA.h importiert, das sie wiederum über Defines aus der Linux Header Datei ghash.h importiert.

```
DEF_HASH_STRUCTS(target, COBRA_PLUGIN_HASH_MAX_ENTRIES, struct
cobra_target);
DEF_HASH_STRUCTS(instance, COBRA_INSTANCE_HASH_MAX_ENTRIES, struct
cobra_instance);

DEF_HASH(static, target, COBRA_PLUGIN_HASH_MAX_ENTRIES, struct cobra_target,
ptrs, char*, cobra_name, strcmp, !strcmp, hashfn_plugin);
DEF_HASH(static, instance, COBRA_INSTANCE_HASH_MAX_ENTRIES, struct
cobra_instance, ptrs, unsigned long, cobra_instance, hashcmp_instance,
hasheq_instance, hashfn_instance);
```

Diese beiden Hashes werden verwendet, um sich in der Funktion cobra_register() die geladenen Cobra Plugins anhand ihrer Namen (cobra_table), beziehungsweise in der Funktion checkcon-

fig() anhand ihrer Instanz Nummer (cobrai_table) zu merken, da die Instanznummer erst in der checkconfig() Funktion bekannt ist.

In der Funktion cobra_unregister() werden Cobra Plugins, die wieder aus dem Kernel entfernt werden, auch aus den beiden Hashes entfernt. Die beiden Funktionen werden von den Makros cobra_init() und cobra_exit(), die in der Header Datei include/linux/netfilter_ipv4/cobra.h definiert werden, und mit denen sich jedes Cobra Plugin initialisiert, aufgerufen. ipt_COBRA.c definiert in der Header Datei include/linux/netfilter_ipv4/ipt_COBRA.h die beiden Funktionen als extern verfügbar, so dass sie jedem Plugin zur Verfügung stehen.

```
extern unsigned int cobra_register(char *cobrap, cobra_target_func_t t,
cobra_config_func_t c, cobra_config_func_t rc);
extern unsigned int cobra_unregister(char *cobrap);
```

Über den cobrai_table Hash kann mit der Instanznummer als Schlüssel der Name des zugehörigen Plugins ermittelt werden. Über den cobra_table Hash können mit dem Plugin Namen als Schlüssel die zum Plugin gespeicherten Informationen gefunden werden. Diese bestehen aus Pointer auf drei Funktionen pro Plugin: Eine target() Funktion, eine config() Funktion und eine reconfig() Funktion werden in einer Struktur von Typ cobra_target gespeichert.

```
typedef unsigned int (*cobra_target_func_t)(struct sk_buff **, unsigned int,
const struct net_device *, const struct net_device *, unsigned long);
typedef unsigned int (*cobra_config_func_t)(const char *config, unsigned
long);

struct cobra_target
{
  char cobra_name[COBRA_PLUGIN_NAME_MAX_LEN];
  cobra_target_func_t cobra_target_func;
  cobra_config_func_t cobra_config_func;
  cobra_config_func_t cobra_reconfig_func;
  struct target_ptrs ptrs;
};
```

Die Funktion target() des Tabellen Moduls wird vom Netfilter Framework aufgerufen und verteilt die ankommenden Pakete an die einzelnen Instanzen der geladenen Cobra Plugins. Dies geschieht durch die im cobra_table Hash gespeicherte Target Funktion.

```
n = find_target_hash(&cobra_table, cobrainfo->plugin);
if (!n)
  return IPT_CONTINUE;
return n->cobra_target_func(pskb, hooknum, in, out, cobrainfo->instance);
```

Die init() Funktion des Cobra Target Modules registriert die Tabelle.

```
if (ipt_register_target(&ipt_cobra_reg))
  return -EINVAL;
```

Des weiteren registriert die init() Funktion zwei Dateien im /proc Dateisystem des Kernels. Die erste Datei heisst /proc/net/cobra_instance und die zweite Datei heisst /proc/net/cobra_status.

```
/* register /proc/net/cobra_instance */
entry = create_proc_entry(PF_INSTANCE, S_IFREG |S_IRUGO |S_IWUGO, proc_net);
```

```
if (entry)
  entry->read_proc = pf_instance_read;
else
  return -EINVAL;

/* register /proc/net/cobra_status */
entry = create_proc_entry(PF_STATUS, S_IFREG | S_IRUGO | S_IWUGO, proc_net);
if (entry)
{
  entry->read_proc = pf_status_read;
  entry->write_proc = pf_status_write;
}
else
  return -EINVAL;
```

Die read und write Funktionen für die beiden /proc Dateien sind in den Funktionen pf_status_read() und pf_status_write(), sowie in der Funktion pf_instance_read() implementiert.

pf_status_read() visualisiert den aktuellen Zustand der beiden Hashes cobra_table und cobrai_table.

```
LOCK_BH(&ip_cobra_status_read_lock);
len = hashprint(buffer, &cobra_table, &cobrai_table);
*start = buffer;
UNLOCK_BH(&ip_cobra_status_read_lock);
return len;
```

pf_status_write() ruft die Reconfig Funktion des entsprechenden Cobra Plugins mit den übergebenen Daten auf. Die Statusdatei muss entweder mit “<Pluginname>#<Plugininstanz> <Konfigdaten>\n” oder mit “<Plugininstanz> <Konfigdaten>\n” beschrieben werden. Im nächsten Codefragment ist ersichtlich, wie die Reconfig Funktion des entsprechenden Plugins aufgerufen wird.

```
LOCK_BH(&ip_cobra_status_write_lock);
[...]
n = find_target_hash(&cobra_table, name);
if (n)
  n->cobra_reconfig_func(val, instance);
[...]
UNLOCK_BH(&ip_cobra_status_write_lock);
```

pf_instance_read() implementiert die Verwaltung der garantiert eindeutigen Instanznummern.

Die exit() Funktion des Target Moduls meldet das Cobra Target beim Netfilter Framework ab und entfernt die beiden /proc Dateien.

Um die Cobra Plugins beim Laden zu konfigurieren wird in der checkconfig() Funktion die im cobra_table Hash gespeicherte Config Funktion des Cobra Plugins mit den mitgegebenen Konfigdaten aufgerufen.

```
if (!cobrainfo->config_done)
{
  cobrainfo->config_done = 1;
```

```
  n = find_target_hash(&cobra_table, cobrainfo->plugin);
  if (!n)
    return IPT_CONTINUE;
  ret = n->cobra_config_func((const char *)cobrainfo->config, cobrainfo->in-
stance);
}
```

5. 3. Instanz Nummern: Lesen von /proc/net/cobra_instance

Das Cobra Framework ist darauf angewiesen, dass einzelne Plugininstanzen garantiert eindeutige Instanznummern zugewiesen bekommen. Andernfalls ist die eindeutige Identifikation der entsprechenden Plugininstanz nicht gewährleistet.

Um dies garantieren zu können, wurde die Verwaltung der Plugininstanzen dem Kernel, und somit dem Cobra Target, überlassen. Das Cobra Target erstellt die Datei /proc/net/cobra_instance, wie oben im letzten Kapitel erwähnt. Bei einem Lesezugriff auf die Datei wird die nächste freie Instanznummer zurückgegeben. Der Code im Kernel ist durch einen Semaphoren Lock geschützt. Somit kann garantiert werden, dass die Instanznummern eindeutig im ganzen System sind.

Dies wird auch in einer allfälligen Fortsetzungsarbeit sehr wichtig sein, denn falls Cobra Plugininstanzen Remote, von anderen Rechnern oder Routern aus gesteuert, geladen werden sollen, so müssen diese eindeutig identifizierbar sein, und die Remote Steuerinstanz weiss ja nicht welche Instanznummer auf dem zu steuernden System schon verwendet worden sind.

5. 4. Status Information: Lesen von /proc/net/cobra_status

User und Userprozesse, die gerne wissen möchten, welche Plugins in wievielen Instanzen in einem gewissen Moment vom Cobra Framework geladen und verwaltet werden, können durch Lesen der Datei /proc/net/cobra_status den aktuellen Zustand abfragen.

```
# cat /proc/net/cobra_status

Registered Router Plugins: 2
Registered Router Plugins Instances: 5

        Plugin TEST     Module COBRA_TEST
        Instances 1,2,3

        Plugin LOG      Module COBRA_LOG
        Instances 4,5
```

Im obigen Beispiel sind insgesamt 5 Instanzen geladen und zwar 3 Instanzen, mit den Instanznummern 1, 2 und 3, des Plugins COBRA_TEST und 2 Instanzen, mit den Instanznummern 4 und 5, des Plugins COBRA_LOG.

5. 5. Runtime Konfiguration: Schreiben auf /proc/net/cobra_status

User und allfällige Userprozesse, die eine geladene Plugininstanz gerne neu konfigurieren möchten beziehungsweise ihr beliebige Daten zukommen lassen möchten, können durch Schreiben auf die Datei /proc/net/cobra_status der entsprechenden Instanz die Daten zukommen lassen.

Der Syntax ist einfach. Zuerst wird die empfangende Plugininstanz Nummer in die Datei geschrieben. Danach folgt ein Space (ASCII Code 32) und daraufhin beliebige Daten, die die entsprechende Plugininstanz dann in ihrer Reconfig Funktion erhält. Es ist auch möglich vor der Plugininstanz noch den Plugin Namen zuzufügen.

```
xcore:/ # cat > /proc/net/cobra_status
4 new configuration for instance 4

oder

xcore:/ # cat > /proc/net/cobra_status
LOG#4 new configuration for instance 4
```

Beide Varianten bewirken dasselbe.

5. 6. Cobra Erweiterung von iptables: libipt_COBRA.c

Das Userspacetool ‘iptables’ musste erweitert werden, um die neuen Optionen für das neue Cobra Target zu implementieren. Dies geschah, indem dem iptables Programm eine weitere Shared Library zugefügt wurde. Diese Shared Library wird in durch die Datei iptables/extentions/libipt_COBRA.c implementiert. Im weiteren Text wird deren Implementation im Detail erläutert.

Die Shared Library implementiert prinzipiell zwei Teile: Zum einen überprüft sie allfällige Komandozeilenparameter auf ihre Korrektheit, zum Anderen lädt sie allfällig noch nicht geladene Cobra Plugins, die sich dann automatisch beim Cobra Target registrieren und somit dem Cobra Framework zur Verfügung stehen.

Die Prüfung der Kommandozeilenparameter geschieht in den Funktionen parse() und final_check(). Das laden allfälliger noch nicht geladener Cobra Plugins geschieht am Ende der Funktion parse().

In der _init() Funktion registriert sich die Shared Library als neues Target bei dem ‘iptables’ Hauptcode. Es wird eine Struktur mit den Zeigern auf die in der Shared Library implementierten Funktionen und Strukturen übergeben.

5. 7. Kompilieren eines Linux Kernels mit Cobra Support

Einen um Cobra Plugins erweiterten Kernel zu kompilieren ist einfach. Das Cobra Framework fügt dem Standart Kernelkonfig neue Kerneloptionen zu, die im Untermenü ‘Networking options / IP: Netfilter Configuration’ erscheinen. Nach Wahl der richtigen Optionen kann der Kernel ganz normal kompiliert werden.

```
<M>   COBRA plugins (EXPERIMENTAL)
[*]     COBRA plugin debug
<M>     TEST COBRA plugin support (EXPERIMENTAL)
<M>     LOG COBRA plugin support (EXPERIMENTAL)
<M>     DEMO COBRA plugin support (EXPERIMENTAL)
```

6. Entwicklung von Cobra Plugins

6. 1. Plugin Grundlagen

Beim Design des Cobra Plugin Frameworks wurde grossen Wert darauf gelegt, dass das Schreiben von Cobra Plugins so einfach wie möglich ist.

Ein Cobra Plugin muss lediglich eine Cobra Header Datei durch eine Include Anweisung einbinden und einige spezielle Funktionen implementieren. Diese Funktionen werden in zwei Makros angegeben und damit dem Cobra Framework bekannt gemacht.

Folgende Funktionen müssen in einem Cobra Plugin implementiert werden:
- load / unload
- target
- config / reconfig

Im den weiteren Kapitel werden die Aufgaben der einzelnen Funktionen im genauer erklärt.

6. 1. 1. Funktion ‘load’

In der ‘load’ Funktion des Plugins können weitere Initialisierungen für die Plugin Implementation vorgenommen werden. Diese Funktion wird genau einmal beim Laden des Plugin Moduls aufgerufen und ist somit geeignet, allfällige globale Initialisationen für das ganze Plugin vorzunehmen.

6. 1. 2. Funktion ‘unload’

Die ‘unload’ Funktion des Plugins wird genau einmal beim Entfernen des Plugins aus dem Speicher aufgerufen. Hier können allen Instanzen gemeinsame Ressourcen freigegeben werden.

6. 1. 3. Funktion ‘target’

Die ‘target’ Funktion wird für jedes Paket, das den entsprechenden Flow Definitionen in der Cobra Tabelle der Instanz entspricht, aufgerufen. Die Funktion erhält sowohl den sk_buff des Paketes wie auch die Hook Nummer, bei der das Paket gesehen wurde, das Input, wie auch das Output Interface und die Instanz Nummer der empfangenden Plugininstanz.

6. 1. 4. Funktion ‘config’

Die ‘config’ Funktion wird für jede Instanz eines Cobra Plugins mit dem entsprechenden Konfigurationsstring und der Instanznummer aufgerufen. Hier können Plugininstanzen allfällige Initialisationen, die jeder Instanz eigen sind, vornehmen.

6. 1. 5. Funktion ‘reconfig’

Die ‘reconfig’ Funktion wird aufgerufen, wenn eine User Prozess in die /proc/net/cobra_status Datei schreibt und die entsprechende Plugininstanz angibt. Diese Funktion ist geeignet, einer Instanz zu ermöglichen, zur Laufzeit neu konfiguriert zu werden. Es können auch andere Laufzeitfunktionalitäten implementiert werden.

6. 2. Beispiel: Ein Cobra Plugin erklärt

Im Folgenden wird eine Beispielimplementation im einzelnen erklärt.

Jedes Cobra Plugin muss eine spezielles Cobra Header Datei durch eine Include Anweisung einbinden.

```
#include <linux/netfilter_ipv4/cobra.h>
```

Als Nächstes implementiert ein Cobra Plugin ‘load’ und ‘unload’ Funktionen.

```
static int __init load(void)
{
  DEBUGP("COBRA_TEST: load() called...\n");

  /* do plugin initialisations */

  DEBUGP("COBRA_TEST: load() succeded...\n");
  return 0;
}

static void __exit unload(void)
{
  DEBUGP("COBRA_TEST: unload() called...\n");

  /* do plugin cleanup */

  DEBUGP("COBRA_TEST: unload() succeded...\n");
}
```

Die eigentliche Paketverarbeitung eines Cobra Plugins erfolgt in seiner ‘target’ Funktion. Sie wird für jedes Paket, das den Flow Definitionen der entsprechenden Plugininstanz in der Cobra Tabelle entspricht, aufgerufen.

```
unsigned int cobra_target_test(struct sk_buff **pskb,
                           unsigned int hooknum,
                           const struct net_device *in,
                           const struct net_device *out,
                           unsigned long instance)
{
  DEBUGP("COBRA_TEST: target() called for plugin TEST#%lu for hock %u\n",
         instance, hooknum);

  /* process packet */
  return IPT_CONTINUE;
}
```

Der Rückgabewert der Funktion entspricht dem Rückgabewert der Netfilter Target Funktion und kann die folgenden Werte annehmen:

- IPT_CONTINUE Falls das Traversieren der Tabellen weitergehen soll
- NF_ACCEPT Falls das Traversieren der Tabellen weitergehen soll
- NF_DROP Falls das Paket weggeworfen werden soll
- NF_STOLEN Falls das Paket vom Plugin übernommen wird
- NF_QUEUE Falls das Paket für den Userspace eingereiht werden soll
- NF_REPEAT Falls der Hook nochmals aufgerufen werden soll

Die Initialisierung der einzelnen Plugininstanzen erfolgt in der ‘config’ Funktion. Sie wird für jede Instanz einmal aufgerufen. Hier initialisiert sich die entsprechende Instanz und erhält auch den auf der Kommandozeile angegebenen Konfigurationsstring.

```
unsigned int cobra_config_test(const char *config, unsigned long instance)
{
 DEBUGP("COBRA_TEST: config() called for plugin TEST#%lu with config '%s'\n",
        instance, config);
  /* configure plugin instance */
  return 1;
}
```

Um eine Plugininstanz zur Laufzeit neu konfigurieren zu können implementiert jedes Plugin eine ‘reconfig’ Funktion. Sie wird aufgerufen, wenn ein Userprozess auf die Datei /proc/net/cobra_status schreibt.

```
unsigned int cobra_reconfig_test(const char *config, unsigned long instance)
{
  DEBUGP("COBRA_TEST: reconfig() called for plugin TEST#%lu with '%s'\n",
         instance, config);
  /* reconfigure plugin instance */
  return 1;
}
```

Alle diese im obigen Text erklärten Funktionen müssen dem Cobra Framework bekannt gegeben werden. Dies erfolgt durch zwei spezielle Makros, die in der Cobra Header Datei definiert werden.

Mit dem cobra_init Makro werden Plugin Name wie auch ‘load’, ‘target’, ‘config’ und ‘reconfig’ Funktion definiert.

```
cobra_init("TEST", load, cobra_target_test,
           cobra_config_test, cobra_reconfig_test);
```

Mit dem cobra_exit Makro wird die ‘unload’ Funktion definiert.

```
cobra_exit(unload);
```

6. 3. Cobra Plugin Design für Fortgeschrittene

Cobra Plugins sind ganz normale Kernel Module. Dies ermöglicht den Plugins alle Möglichkeiten wahrnehmen zu können, die einem Code im Kernel offenstehen.

Unter anderem ist es einem Cobra Plugin somit möglich, auch direkt auf die Datenstrukturen des Linux Netzwerkkerns zuzugreifen und so beispielsweise die Routing Tabelle zu verändern.

7. Demonstration

7. 1. Kommandozeile - Erklärungen

In diesem Kapitel wird ein Beispieldurchlauf, der alle implementierten Features von Cobra zeigt, erklärt.

Auf einem frisch gebooteten System wird als erstes eine neue Plugininstanz des Plugins TEST initialisiert und auch gleich an einen einfachen Filter gebunden. Der Filter wird im PREROUTING Hook erstellt und leitet alle Pakete die eine IP Source Adresse 172.16.7.4 besitzen an die neu erstellte Plugininstanz weiter.

```
xcore:~ # iptables -t cobra -A PREROUTING -s 172.16.7.4 -j COBRA --plugin TEST --
autoinstance --config 'initial config string for first instance'
```

Das System antwortet mit einigen Debug Nachrichten, lädt das benötigte Plugin und gibt die eindeutige Instanznummer, in diesem Fall die Nummer 1, des Plugins aus.

```
libipt_COBRA: init() called...
libipt_COBRA: init() succeded...
libipt_COBRA: parse() trying to load module ipt_COBRA
libipt_COBRA: autoinstance, requesting unique instance from kernel.
libipt_COBRA: got unique instance 1 from kernel.
Cobra plugin instance is 1
libipt_COBRA: load_module() trying to load module cobra_TEST
```

Als nächstes erstellen wir eine weitere Instanz des selben Plugins TEST und binden diese an einen weiteren Filter im PREROUTING Hook, der alle Pakete mit IP Source Adresse 172.16.7.5 an die neue Plugininstanz weiterleitet.

```
xcore:~ # iptables -t cobra -A PREROUTING -s 172.16.7.5 -j COBRA --plugin TEST --
autoinstance --config 'initial config string for second instance'
```

Auch hier antwortet das System mit den üblichen Debug Meldungen und der nächsten freien Plugininstanznummer 2.

```
libipt_COBRA: init() called...
libipt_COBRA: init() succeded...
libipt_COBRA: parse() trying to load module ipt_COBRA
libipt_COBRA: autoinstance, requesting unique instance from kernel.
libipt_COBRA: got unique instance 2 from kernel.
Cobra plugin instance is 2
libipt_COBRA: load_module() trying to load module cobra_TEST
```

Nun wollen wir ein zweites Plugin namens LOG laden und an einen weiteren Filter binden. Auch hier wird das System angewiesen, selbstständig die nächste freie eindeutige Instanznummer zu wählen.

```
xcore:~ # iptables -t cobra -A PREROUTING -s 172.16.7.6 -j COBRA --plugin LOG --au-
toinstance --config 'initial config string for first instance'
```

Das System weist der neuen Plugininstanz die Instanznummer 3 zu.

```
libipt_COBRA: init() called...
```

```
libipt_COBRA: init() succeded...
libipt_COBRA: parse() trying to load module ipt_COBRA
libipt_COBRA: autoinstance, requesting unique instance from kernel.
libipt_COBRA: got unique instance 3 from kernel.
Cobra plugin instance is 3
libipt_COBRA: load_module() trying to load module cobra_LOG
```

Es muss allerdings nicht immer eine eins zu eins Beziehung zwischen Plugininstanz und Filter bestehen. Mit dem nächsten Kommando binden wir einen weiteren Filter an die als allererstes erstellte Plugininstanz mit der Nummer 1.

```
xcore:~ # iptables -t cobra -A PREROUTING -s 172.16.7.7 -j COBRA --plugin TEST --
instance 1
```

Wie man sieht bindet das System den neuen Filter an die alte Plugininstanz und gibt keine neue Plugininstanznummer aus.

```
libipt_COBRA: init() called...
libipt_COBRA: init() succeded...
libipt_COBRA: load_module() trying to load module cobra_TEST
```

Mit dem folgenden Kommando inspizieren wir die momentan konfigurierten Filter Tabellen. Da wir alle Filter im PREROUTING Hook eingefügt haben, sind die anderen Hooks leer. Die beiden Filter die beide auf die selbe Plugininstanznummer 1 zeigen, sind klar ersichtlich.

```
xcore:~ # iptables -t cobra -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
COBRA      all  --  172.16.7.4           anywhere           COBRA TEST#1
COBRA      all  --  172.16.7.5           anywhere           COBRA TEST#2
COBRA      all  --  172.16.7.6           anywhere           COBRA LOG#3
COBRA      all  --  172.16.7.7           anywhere           COBRA TEST#1

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Durch lesen der Datei /proc/net/cobra_status kann der Status des Cobra Frameworks auch direkt abgefragt werden.

```
xcore:~ # cat /proc/net/cobra_status

Registered COBRA plugins: 2
Registered COBRA plugins Instances: 3

        Plugin TEST     Module COBRA_TEST
        Instances 1,2

        Plugin LOG      Module COBRA_LOG
        Instances 3
```

Um nun auch noch die zur Laufzeit mögliche Rekonfiguration eines Plugins zu demonstrieren, senden wir der Plugininstanz des LOG Plugins mit der Nummer 3 einen neuen Konfigurationsstring.

```
xcore:~ # cat > /proc/net/cobra_status
LOG#3 new configuration string
```

Um die Demonstration zu beenden, werden nun alle Filter wieder gelöscht. Und daraufhin alle Cobra Module wieder aus dem Speicher entfernt.

```
xcore:~ # iptables -t cobra -D PREROUTING 1
xcore:~ # iptables -t cobra -D PREROUTING 1
xcore:~ # iptables -t cobra -D PREROUTING 1
xcore:~ # iptables -t cobra -D PREROUTING 1

xcore:~ # lsmod
Module                  Size  Used by
cobra_LOG               4324   0  (unused)
cobra_TEST              1756   0  (unused)
ipt_COBRA               7992   1  [cobra_LOG cobra_TEST]
iptable_cobra           3552   0  (autoclean) (unused)
ip_tables              14648   2  [iptable_cobra ipt_COBRA]
ne2k-pci                5436   1  (autoclean)
8390                    7180   0  (autoclean) [ne2k-pci]

xcore:~ # rmmod cobra_TEST cobra_LOG iptable_cobra ipt_COBRA ip_tables
```

7. 2. Debug Ausgabe - Erklärungen

Im nun folgenden Kapitel werden die Debug Meldungen des Cobra Frameworks des im vorherigen Kapitel gezeigten Beispielsdurchlaufs erklärt. Das Cobra Framework loggt über den klogd direkt in den syslogd des Systems.

Mit dem Laden des ersten Plugins und dessen Instanzierung werden die Module ip_tables, ipt_COBRA, iptable_cobra, wie auch das eigentliche Plugin cobra_TEST geladen. Dies wird durch die Modulabhängigkeiten erzwungen und automatisch vom System erledigt.

Als erstes wird ip_tables geladen.

```
Sep  4 15:20:38 xcore kernel: ip_tables: (c)2000 Netfilter core team
```

Da die Kommandozeile auf das Cobra Target Bezug nimmt, wird das ipt_COBRA Modul geladen. Bei seiner Initialisierung erstellt es die zwei Dateien /proc/net/cobra_instance und /proc/net/cobra_status.

```
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: init() called...
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: init() - create_proc_entry('/proc/net/
cobra_instance') succeeded...
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: init() - create_proc_entry('/proc/net/
cobra_status') succeeded...
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: init() succeded...
```

Das Userspacetool iptables erfragt daraufhin beim Cobra Target die nächste freie Plugininstanznummer an und erhält die Nummer 1.

```
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: pf_instance_read - returning instance 1
```

Das cobra_TEST Plugin wird nun geladen und es wird eine Instanz desselben erzeugt die die eindeutige Instanznummer 1 erhält.

```
Sep  4 15:20:38 xcore kernel: COBRA: cobra_init(TEST) called...
Sep  4 15:20:38 xcore kernel: COBRA_TEST: load() called...
Sep  4 15:20:38 xcore kernel: COBRA_TEST: load() succeded...
```

Die Instanz des geladenen Plugins registriert sich nun beim Cobra Target.

```
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: cobra_register(TEST) called...
Sep  4 15:20:38 xcore kernel: COBRA: cobra_init(TEST) succeded...
```

Der neue Filter muss nun in der Cobra Tabelle eingetragen werden, was bedingt, dass das Cobra Tabellen Modul iptable_cobra geladen wird und sich in allen 5 IPv4 Hooks registriert.

```
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: init() called...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering table cobra...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering ipt_ops[0]...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering ipt_ops[1]...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering ipt_ops[2]...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering ipt_ops[3]...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: registering ipt_ops[4]...
Sep  4 15:20:38 xcore kernel: COBRA_TABLE: init() succeded
```

Als letztes wird nun die neue Plugininstanz konfiguriert.

```
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: configuring plugin TEST instance 1.
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: config() dispatching to plugin TEST#1
Sep  4 15:20:38 xcore kernel: COBRA_TEST: config() called for plugin TEST#1 with config
'initial config string for first instance'
Sep  4 15:20:38 xcore kernel: COBRA_TARGET: checkentry(TEST): adding plugin instance 1
```

Das Erstellen einer weiteren Instanz desselben Plugins ist für das Framework nun eine kleine Sache. Das Userspacetool iptables erfragt die nächste freie Instanznummer, erstellt die entsprechenden Tabelleneinträge und konfiguriert die neue Instanz.

```
Sep  4 15:21:19 xcore kernel: COBRA_TARGET: pf_instance_read - returning instance 2
Sep  4 15:21:19 xcore kernel: COBRA_TARGET: configuring plugin TEST instance 2.
Sep  4 15:21:19 xcore kernel: COBRA_TARGET: config() dispatching to plugin TEST#2
Sep  4 15:21:19 xcore kernel: COBRA_TEST: config() called for plugin TEST#2 with config
'initial config string for second instance'
Sep  4 15:21:19 xcore kernel: COBRA_TARGET: checkentry(TEST): adding plugin instance 2
```

Erst das Erstellen einer Instanz eines neuen Plugins erfordert das erneute registrieren des Plugins beim Cobra Target. Danach läuft der selbe Vorgang wie bei den beiden vorherigen Instanzen ab.

```
Sep  4 15:22:10 xcore kernel: COBRA_TARGET: pf_instance_read - returning instance 3
Sep  4 15:22:10 xcore kernel: COBRA: cobra_init(LOG) called...
Sep  4 15:22:10 xcore kernel: COBRA_LOG: load() called...
Sep  4 15:22:10 xcore kernel: COBRA_LOG: load() succeded...
```

```
Sep  4 15:22:10 xcore kernel: COBRA_TARGET: cobra_register(LOG) called...
Sep  4 15:22:10 xcore kernel: COBRA: cobra_init(LOG) succeded...
Sep  4 15:22:10 xcore kernel: COBRA_TARGET: configuring plugin LOG instance 3.
Sep  4 15:22:10 xcore kernel: COBRA_TARGET: config() dispatching to plugin LOG#3
Sep  4 15:22:10 xcore kernel: COBRA_LOG: config() called for plugin LOG#3 with config
'initial config string for first instance'
Sep  4 15:22:10 xcore kernel: COBRA_TARGET: checkentry(LOG): adding plugin instance 3
```

Das Erstellen eines weiteren Filters für eine schon existierende Plugininstanz gibt keine weiteren Meldungen aus.

Im Folgenden wurde dann noch die Status Datei /proc/net/cobra_status gelesen.

```
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: hashprint(TEST)
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: hashprint(TEST) - Instance 1
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: hashprint(TEST) - Instance 2
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: hashprint(LOG)
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: hashprint(LOG) - Instance 3
Sep  4 15:23:53 xcore kernel: COBRA_TARGET: pf_status_read - returning 158 bytes
```

Das Rekonfigurieren der Plugininstanz mit der Nummer 3, durch schreiben auf die Status Datei /proc/net/cobra_status, löst die Rekonfiguration der Instanz aus.

```
Sep  4 15:24:42 xcore kernel: COBRA_TARGET: pf_status_write - accepting 31 bytes
Sep  4 15:24:42 xcore kernel: COBRA_TARGET: reconfiguring plugin LOG instance 3.
Sep  4 15:24:42 xcore kernel: COBRA_LOG: reconfig() called for plugin LOG#3 with config
'new configuration string'
```

Nun wurden alle Filter im PREROUTING Hook gelöscht und dann alle Module in der Reihenfolge ihrer Abhängigkeiten wieder aus dem Speicher entfernt. Wie man sieht, deregistrieren sich alle Plugininstanzen beim Cobra Target, bevor dieses ebenfalls aus dem Speicher entfernt wird, und die Cobra Tabelle deregistriert alle 5 IPv4 Hooks.

```
Sep  4 15:25:50 xcore kernel: COBRA: cobra_exit(TEST) called...
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: cobra_unregister(TEST) called...
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: cobra_unregister(TEST): removing instance
1
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: cobra_unregister(TEST): removing instance
2
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: cobra_unregister(LOG): removing instance 3
Sep  4 15:25:50 xcore kernel: COBRA_TEST: unload() called...
Sep  4 15:25:50 xcore kernel: COBRA_TEST: unload() succeded...
Sep  4 15:25:50 xcore kernel: COBRA: cobra_exit(TEST) succeded...
Sep  4 15:25:50 xcore kernel: COBRA: cobra_exit(LOG) called...
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: cobra_unregister(LOG) called...
Sep  4 15:25:50 xcore kernel: COBRA_LOG: unload() called...
Sep  4 15:25:50 xcore kernel: COBRA_LOG: unload() succeded...
Sep  4 15:25:50 xcore kernel: COBRA: cobra_exit(LOG) succeded...
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: fini() called...
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering ipt_ops[0]..
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering ipt_ops[1]..
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering ipt_ops[2]..
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering ipt_ops[3]..
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering ipt_ops[4]..
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: unregistering cobra table...
Sep  4 15:25:50 xcore kernel: COBRA_TABLE: fini() succeded
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: fini() called...
Sep  4 15:25:50 xcore kernel: COBRA_TARGET: fini() succeded...
```

8. Schlussfolgerungen und Ausblick

Das Resultat dieser Arbeit ist ein komplettes Plugin Framework, das die gesetzten Ziele in allen Punkten erfolgreich implementiert.

Das Netfilter Framework des Linux Kernels hat die Implementation stark vereinfacht und ermöglichte eine sehr gelungene Verschmelzung der Cobra Anforderungen mit den schon vorhandenen modularen Netfilter Konzepten. Infolgedessen konnte der Cobra Kernel Code sehr übersichtlich und modular implementiert werden.

Durch die Benutzung von Codemakros und der Verlagerung von Initalisierungroutinen in die Header Dateien wurde es möglich, ein sehr einfaches und übersichtliches Plugin Konzept zu entwerfen. Dies ermöglicht es, Cobra Plugins auf einfachste Art und Weise zu entwerfen und zu schreiben.

Flows lassen sich flexibel und in beliebiger Menge dynamisch zur Laufzeit an beliebig viele Plugininstanzen binden, ohne die Effizienz des Cobra Frameworks massgeblich im Vergleich zum Netfilter Framework, zu verändern.

In Zukunft wird es möglich sein, Plugins über das Netz auf speziellen Plugin Servern bereitzustellen. Einfache oder mehrstufige Plugin Server Infrastrukturen mit oder ohne Caching der Plugins wären denkbar.

Die Flow zu Plugin Assoziation könnte Remote erfolgen. So wären sowohl zentral gesteuerte, wie auch dynamisch anpassungsfähige globale Szenarien, sowohl manueller wie auch automatischer Art, denkbar.

All dies wird natürlich ganz neue Anforderungen an die Sicherheit und Authentisierung der entsprechenden Parteien stellen.

9. Anhang

9. A. Offizielle Problemstellung

Aufgabenstellung:	Prof. Dr. B. Plattner, Ralph Keller
Titel:	**Router Plugins für Linux**
Beginn der Arbeit:	12. März 2001
Abgabetermin:	15. Juni 2001
Betreuung:	Ralph Keller, Lukas Ruf
Arbeitsplatz:	ETZ G69
Umgebung:	Linux, C

9. A. 1. Einleitung

Aktive Netzwerke (Active Networks) werden es in Zukunft ermöglichen, Netzwerke den eigenen Bedürfnissen nach zu programmieren, um neue Netzwerkdienste rasch und unkompliziert einführen zu können.
Die am TIK entwickelte Active Network Node (ANN) Architektur [2] [3] erlaubt es, sogenannte *Plugins*, welche ausführbaren Code enthalten, zur Laufzeit dynamisch in den Betriebssystemkernel zu installieren. Damit wird ermöglicht, den Netzwerkknoten auf flexible Weise mit beliebiger Funktionalität zu erweitern. Zum Beispiel kann ein Benutzer ein Ver- und Entschlüsselungs-Plugin im Netz installieren, um zu erreichen, dass sein Datenverkehr verschlüsselt übertragen wird. Ein Netzwerkbenutzer hat somit die Möglichkeit, Plugins in Routers zu installieren, um einzelne Flows oder aggregierte Flows speziell nach seinen Bedürfnissen zu verarbeiten.

Linux bietet mit der Netfilter-Architektur [4] eine elegante Möglichkeit, den Betriebssystemkernel zur Laufzeit mit beliebiger Netzwerk-Funktionalität zu erweitern. Spezifische Netzwerkkomponenten (wie NAT, SYN-Tracing etc.) wurden bereits erfolgreich mittels Netfilter implementiert.

In dieser Arbeit geht es nun darum, eine Architektur basierend auf Netfilter zu entwickeln, die es erlaubt, beliebige Komponenten (Plugins) für aktive Netzwerke in den Betriebssystemkernel zu laden. Das Framework soll auf flexible Art undWeise ankommende Pakete einer entsprechenden Plugininstanz zuweisen können.

Dabei sollen die folgenden Kriterien erfüllt sein:
- Flexibilität: Pakete sollten flexibel zu einer Plugininstanz zugewiesen werden können
- Effizienz: Die Ausführung einer Plugininstanz sollte effizient gestaltet sein, der Overhead sollte vertretbar bleiben.
- Integration: Die Plugin-Architektur sollte optimal in Netfilter unter Linux integriert werden .

9. A. 2. Aufgabenstellung

Entwickeln Sie ein allgemeingültiges Konzept, welches es erlaubt, Plugins zu instanzieren und an Flows zu binden, sodass ankommende Pakete entsprechend verarbeitet werden. Bestimmen Sie, welche Erweiterungen an der bestehenden Netfilter-Architektur [4] für Linux notwendig sind, um ihr Design umzusetzen.Implementieren Sie schliesslich ihr Design in Form von einem oder mehreren Netfilter Modulen.

9. A. 2. 1. Ziel

Ziel dieser Arbeit ist ein lauffähiges, flexibles and effizientes Framework, welches es erlaubt, Flows so zu konfigurieren, dass sie durch entsprechende Plugininstanzen verarbeitet werden. Demonstrieren Sie die Funktionsweise, indem Sie zeigen, wie ein bestimmter Flow mittels eines entsprechenden Plugins verarbeitet werden kann.

9. A. 2. 2. Vorgehen

- Lesen Sie sich in die Literatur zum Thema ein ([1], [2] und [3]).
- Machen Sie sich mit der bestehenden ANN-Plattform sowohl von NetBSD als auch Linux vertraut ([7]).
- Erstellen Sie sich einen Zeitplan.
- Betrachten Sie die bestehende Netfilter-Architektur [4] und erstellen Sie ein Design, welches erlaubt, Flows an Plugininstanzen zu binden.
- Bestimmen Sie die Komponenten, welche für die Konfiguration (im Userspace) und Ausführung von Plugins (im Kernel) notwendig sind.
- Implementieren Sie die notwendigen Komponenten unter Linux. Achten Sie auf ein modulares Design, und vermeiden Sie Modifikationen am Standard-Kernel wenn möglich.
- Demonstrieren Sie die Lauffähigkeit ihres System indem Sie zeigen, wie ein sehr einfaches Plugin in den Kernel geladen werden kann, eine Instanz des Plugin an einen Filter gebunden wird und ankommende Pakete durch die entsprechende Plugininstanz verarbeitet werden. Das Plugin kann dabei sehr einfach gestaltet sein, es muss nur die Funktionsweise ihres Frameworks gezeigt werden.
- Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte ausführlich schriftlich festzuhalten.

9. A. 2. 3. Bemerkungen

- Mit dem Betreuer sind wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll mündlich über den Fortgang der Arbeit berichtet und anstehende Probleme diskutiert werden.
- Es ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt.
- Die Dokumentation ist vorzugsweise mit dem Textverarbeitungsprogramm *FrameMaker* oder *Latex* zu erstellen.

9. A. 3. Ergebnisse der Arbeit

Neben einem mündlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein Bericht (in 3 Exemplaren). Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Liste der gelösten und ungelösten Probleme. Eine kritische Würdigung der gestellten Aufgabe und des vereinbarten Zeitplanes runden den Bericht ab.
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht und Implementationsbeschreibung,
- Eine Sammlung aller zum System gehörender Software.

- Eine englischsprachige Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen Überblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

9. B. Liste aller Bilder

9. C. Cobra Sourcen

Die Sourcen des Cobra Frameworks wie auch die Vorstellungspresentation und Dokumentation können auf der folgenden Website bezogen werden:

http://cobra.digital-impact.ch/

Für allfällige Fragen und Anregungen bezüglich des Cobra Frameworks ist der Author unter den folgenden Email Adressen ereichbar:

amir@digital-impact.ch

und

amir@guindehi.ch

9. D. Referenzen

[1] ANN - A Scalable, High Performance Active Network Node. http://www.arl.wustl.edu/arl/projects/ann/ann.html

[2] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins - A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers”, Washington University Tech Report WUCS-98-08, February 1998

[3] Decasper, D., Parulkar, G., Choi, S., DeHart, J.,Wolf, T., Plattner, B., “A Scalable, High Performance Active Network Node”, In IEEE Network, January/February 199

[4] Netfilter Project, IP Tables. http://netfilter.samba.org/

[5] Amir Guindehi: Ein durch Quality of Services (QoS) erweitertes Internet. http://www.amir.ch/papers/QoS-Internet-V2.pdf

[6] Prof. Dr. Burkhard Stiller: Protokolle für Multimediakommunikation: WS 2000/2001, Eidgenössische Technische Hochschule Zürich, Departement für Elektrotechnik und Departement für Informatik

[7] Router Plugins Toolkit. http://www.tik.ee.ethz.ch/~crossbow/rp/

[8] Shreedhar M., Varghese G., “Efficient Fair Queueing using Deficit Round Robin”, Proc. ACM SIGCOMM, August/September, 1995.