ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Gabriel Mueller

Proxies in PromethOS

Semester Thesis SA-2006-16
10th October 2006

Supervisor: Lukas Ruf
Co-Supervisor: Daniela Brauckhoff
Professor: Bernhard Plattner

Abstract

When computers exchange data over networks (e.g. the internet) besides the data also information concerning sender and receiver and about the format of the data are exchanged. This data (called protocol header) is produced on the sending side and processed on the receiving side by so called protocol handlers.
Due to faulty implementations of such protocol handlers an adversary can attack computers by sending data with intentionally malformed protocol headers. When the computer processes these data (or more precisely the protocol headers) it will show an unintended behaviour which often enables the adversary to cause further harm to the computer.
One possibility to protect computers against these kind of attacks is to place a so called proxy between the computer and the rest of the network. Such a proxy is familiar with the format of protocol headers and therefore able to detect malformed protocol headers. By filtering and rebuilding the protocol headers (also called protocol interception) before forwarding them to the original receiver this type of threat can be eliminated.
In this semester thesis a proxy framework which can be used to easily implement proxies for different types of protocol headers was successfully designed and implemented. Its functionality was demonstrated by implementing a proxy using the proxy framework which is capable of understanding TCP protocol headers.

Zusammenfassung

Computer, welche Daten über ein Netzwerk (wie z.B. das Internet) austauschen, üebertragen neben den eigentlichen Daten auch Informationen betreffend Sender und Empfänger und über die Formatierung der Daten. Diese Daten (Protocol Header genannt) werden von so genannten Protocol Handlern auf der Senderseite generiert und auf der der Empfängerseite von diesen verarbeitet.
Auf Grund fehlerhafter Implementationen dieser Protocol Händler kann ein Angreifer einen Computer angreifen, indem dieser Daten mit absichtlich gefälschten Protocol Headern schickt. Wenn der Computer diese Daten (bzw. die dazugehörigen Protocol Header) verarbeitet, wird dieser ein nicht gewünschtes Verhalten zeigen, welches oft dem Angreifer ermöglicht, das System weiter zu schädigen.
Eine Möglichkeit, Computer vor solchen Attacken zu schützen besteht darin, einen Proxy zwischen dem Computer und dem restlichen Netwzerk zu plazieren. Ein solcher Proxy ist vertraut mit dem Format der Protocol Header und somit in der Lage, gefälschte Protocol Header zu erkennen. Werden nun diese Protocol Header herausgefiltert und neu aufgebaut (Protokolunterbruch), bevor sie an den ursprünglichen Empfänger weitergeleitet werden, kann diese Art von Bedrohung ausgeschaltet werden.
In dieser Semesterarbeit wurde erfolgreich ein Proxy Framework entworfen und implementiert, welches dazu benutzt werden kann, Proxies für die verschiedenen Arten von Protocol Headern einfach zu implementieren. Durch die Implementation eines Proxies, welcher in der Lage ist, TCP Protocol Header zu analysieren, wurde die Funktionalität des Proxy Frameworks demonstriert.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Outline . . . 2
- 1.3 Netfilter . . . 2
  - 1.3.1 Hooks in IPv4 . . . 2
  - 1.3.2 Hook Functions . . . 3
- 1.4 PromethOS . . . 4
  - 1.4.1 Properties of PromethOS Plugins . . . 4
  - 1.4.2 Functions of PromethOS Plugins . . . 4
  - 1.4.3 Usage of PromethOS Plugins . . . 6

2 Related Work **7**
- 2.1 Airlock . . . 7
- 2.2 Tetrade Secure Entry Server (SES) . . . 7
- 2.3 Squid . . . 7
- 2.4 Basic Modules for PromethOS Plugins . . . 7
- 2.5 Summery . . . 8

3 Architecture **9**
- 3.1 Overview . . . 9
- 3.2 Basic Ideas of the Proxy Framework Design . . . 9
  - 3.2.1 The PromethOS Plugin for the Proxy Framework . . . 10
  - 3.2.2 Proxy Framework Modules . . . 11
  - 3.2.3 Initializing the Proxy Framework . . . 11
  - 3.2.4 Creation and Configuration of a Proxy Instance . . . 11
  - 3.2.5 Incoming Packet for some Proxy Configuration . . . 12
  - 3.2.6 Design Discussion . . . 12
- 3.3 Summery . . . 13

4 Implementation **15**
- 4.1 Proxy Framework . . . 15
  - 4.1.1 Data Structures of the Proxy Framework . . . 15
  - 4.1.2 Initialization Phase . . . 16
  - 4.1.3 At Runtime . . . 17
  - 4.1.4 Miscellaneous . . . 20
- 4.2 TCP State Machine . . . 22
  - 4.2.1 Motivation . . . 22
  - 4.2.2 Requirements . . . 22
  - 4.2.3 Implementation . . . 24
- 4.3 HTTP Proxy . . . 27
  - 4.3.1 Design Aspects . . . 27
  - 4.3.2 Implementation Aspects . . . 28
- 4.4 HTTPS Proxy . . . 28
  - 4.4.1 Design . . . 28
- 4.5 Summary . . . 31

5 Evaluation **33**
5.1 Methodology . . . 33
5.1.1 Proxy Framework . . . 33
5.1.2 TCP State Machine Proxy Framework Module . . . 33
5.2 Testing . . . 33
5.2.1 Testing PPFP . . . 34
5.2.2 Testing TCP State Machine Proxy Framework Module . . . 34
5.3 Summary . . . 35

6 Summary and Conclusion **37**
6.1 Summary . . . 37
6.1.1 Improvements . . . 37
6.2 Conclusion . . . 38

A Usage **39**
A.1 Simple Example - modull . . . 39
A.1.1 Initialization . . . 39
A.1.2 Testing it . . . 39
A.1.3 Unloading . . . 40
A.2 TCP State Machine - module_tcp . . . 40
A.2.1 Initialization . . . 40
A.2.2 Validation for Working . . . 40
A.3 TCP State Machine - module_tcp2 . . . 40
A.3.1 Initialization . . . 41
A.3.2 Validation for Working . . . 41

B Components of PromethOS **43**
B.1 Kernel Space Parts of PromethOS . . . 43
B.1.1 iptable_promethos . . . 43
B.1.2 ipt_PROMETHOS . . . 43
B.1.3 promethos_<PLUGIN_NAME> . . . 43
B.2 User Space Parts of PromethOS . . . 44
B.2.1 libipt_PROMETHOS . . . 44

C Development Environment **45**
C.1 Host Machine . . . 45
C.2 Client Machine . . . 45
C.3 PromethOS . . . 46

D Aufgabenstellung **47**
D.1 Aufgaben: Proxies in PromethOS NP . . . 47
D.2 Vorgehen . . . 47

Chapter 1

Introduction

The number and also the usage of internet accounts is still increasing. Closely related to this growth is the increasing number of services provided via the internet mostly by companies to customers. Besides a few exceptions (such as peer to peer applications) the big majority of applications / services are still using the client server model where one entity (the server) provides services to a huge number of users (the clients).
What changed over the years is the usage of the internet. Still the internet is widely used for various academical fields but the commercial usage of the internet is rapidly growing. All in all it began with e-Commerce (such as online shops, E-banking) but more and more the internet is going to be used for different multimedia applications (such as telephone and television). Although it was already very painful for some E-Commerce companies when their services were attacked and could not be used as intended the issue of availability becomes even more important when thinking of internet services such as internet telephony.
This demand of high availability evokes additional motivation for some people to attack the backend infrastructure which provides these services. And because of the mentioned centralized structures (one server, many clients) and the public access, attacking the infrastructure is quite simple.
There are various methods and patterns of how a system can be attacked but what is common to all attacks is that they are all using some weakness of the target. The incorrect implementation of a protocol handler is such a weakness (a protocol is a description of how to exchange data between different entities, a protocol handler describes how to treat certain parts of data of a specific protocol).
An attacker can attack systems using a incorrect implemented protocol handler by sending packets with intentionally modified protocol headers (this is the part of the packet which will be in some way processed by the protocol handler). When processing such packets the system will show an unintended behavior. One way protecting system against this method of attack is to use a proxy.
Generally spoken a proxy of a specific protocol is an application which is able to understand this protocol (in contrast to proxies common firewalls[1] understand protocols out of a limited range). The intention is to use the proxy to detect these malformed protocol headers and modify the content of the header in a very secure manner so that the packet / header does not any longer represent a threat to the end system. This method of protection (called protocol interception) is not new and already implemented and used by several applications.

1.1 Motivation

The goal of this semester thesis is to design a modular proxy framework which can be used for any protocols starting at layer two of the ISO/OSI model[2]. A proxy for a specific protocol is realized as an additional module for the proxy framework (from now on called proxy framework module).
Because the proxy framework provides several generic functions and data structures the cost of

[1] A firewall is used to restrict access to computers over the network.
[2] ISO - Information Processing Systems - "Basic Reference Model for Open System Interconnection", IS 7498, 1983

developing proxies (realized as proxy framework modules) are minimized. Furthermore the modularity allows developers to implement new proxies after the proxy framework is implemented and installed on a system.
When using the proxy framework with several modules (that means several proxies for different protocols) incoming data is first processed by the proxy framework and then forwarded to the appropriate proxy framework module. Compared to systems on which several proxies applications for different protocols are running concurrently (and so all data is evaluated by every proxy application) the proxy framework approach also saves system resources.
The proxy framework will be implemented by extending PromethOS (see 1.4) which provides the necessary functionality to access network packets[3]. If the proxy framework can be successfully implemented using PromethOS this is also a proof that PromethOS can be used to implement proxy applications.

1.2 Outline

Chapter 1 The Proxy Framework extends PromethOS which in term relies on the netfilter framework. In the rest of this chapter the most important aspects of these applications concerning the development of the proxy framework are briefly described.

Chapter 2 This chapter names the most common, already existing proxy applications and shows why is desirable to develop a proxy framework.

Chapter 3 Describes the design of the proxy framework by naming the necessary functionality and data structures.

Chapter 4 The first part shows how the most important functions and data structures of the proxy framework are implemented. The second part describes the design and implementation of a proxy by implementing the TCP state machine as proxy framework module. Part three and four name the most important aspects which have to be considered when implementing a HTTP respectively a HTTPS proxy.

Chapter 5 Describes how tests were performed to evaluate the behaviour of the proxy framework and the proxy framework module of the TCP state machine and names encountered problems and their solutions.

Chapter 6 What has been achieved and what can still be improved, ending with a conclusion of this semester thesis.

1.3 Netfilter

Netfilter[1, 2, 3, 4] is a framework which operates in the kernel space. It allows access to raw network packets in a very convenient way. The framework can be separated into two parts:

- **Hooks**
- **Hook functions**

The knowledge about the different hooks and types of return values of the hook functionsis necessary to successfull design and implement the proxy framework.

1.3.1 Hooks in IPv4

For every network protocol (e.g. IPv4[15], IPv6[16]) hooks are defined. Each of these hooks can be seen as a well defined point in the traversal of a network packet through the linux network stack. For ipv4 the following five hooks are defined:

- NF_IP_PRE_ROUTING ①

[3] containing discrete portions of the network data

- NF_IP_LOCAL_IN ②
- NF_IP_FORWARD ③
- NF_IP_POSTROUTING ④
- NF_IP_LOCAL_OUT ⑤

(the numbers in the circles correspond to these ones in Figure 1.1)
Every network packet, depending on origin and destination, traverses different hooks (more exactly a network packet can be accessed at theses points while it traverses the linux network stack[4]). For example a network packet destinated to the local computer will traverse ① and ② (as can be seen in Figure 1.1).

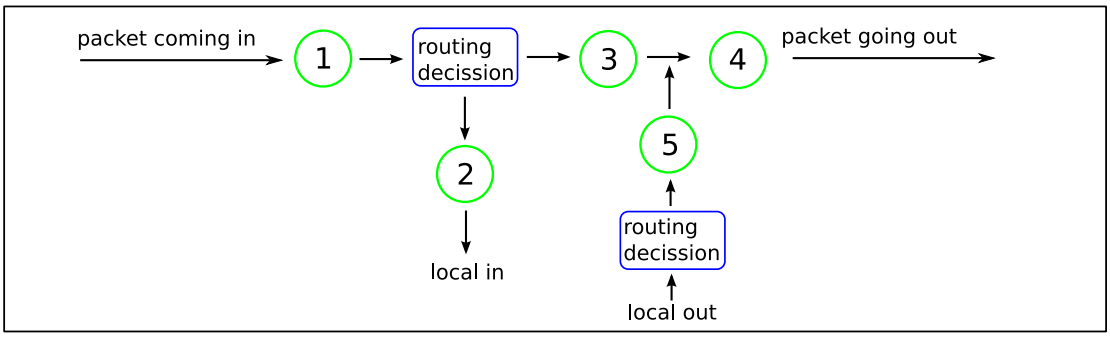

Figure 1.1: Netfilter Hooks

1.3.2 Hook Functions

Hook functions can register at the different hooks. When a network packet arrives at a hook the netfilter framework is activated. Netfilter will check if there are functions registered for this specific hook and if so will call these registered functions. The called hook functions now have the possibility to examine (and even alter) the network packet before they have to give back one of several predefined return values (see below).
To register at a hook the netfilter register function (`nf_register_hook()`) has to be used and a pointer to a struct (of type `nf_hook_ops`) has to be handled to the register function. The struct contains information about at which hook a certain hook function wants to register, a pointer to the hook function, the protocol and a priority value which is used by the netfilter framework to determine in which order different hook functions at one hook have to be called.
When a packet arrives at a hook the netfilter framework calls the hook functions registered at this hook in the order which is determined by the priority values. When the netfilter framework calls a hook function it handles a pointer to a struct (of type `sk_buff`) to the hook function besides other informations. With the pointer of type `sk_buff` the hook function has full access to the network packet.

Return Values

After a hook function has examined (and or altered) the network packet it has to return on of the return values specified by the netfilter framework:

- **NF_ACCEPT**
  Continue traversal of the linux network stack as normal.
- **NF_DROP**
  Drop the packet, do not continue traversal of the linux network stack.
- **NF_STOLEN**
  Packet was stolen by hook function, it will not continue traversal of the linux network stack.

[4]Describes the network architecture of the linux kernel.

- **NF_QUEUE**
  Queue the packet (usually for userspace handling).
- **NF_REPEAT**
  Call the hook functions of this hook again.

1.4 PromethOS

PromethOS is a framework which uses netfilter to gain access to network packets traversing the linux network stack and to forward the network packets to so called PromethOS plugins. The plugins can examine or alter the network packets in a nearly arbitrary way. Using PromethOS plugins is one possibility to gain access to network packets.
The PromethOS framework was original designed and developed in a semester thesis by Amir Guindehi (at that time called COBRA[6]). It was further developed and expanded by Lukas Ruf who also developed a derivative version supporting network processors.

1.4.1 Properties of PromethOS Plugins

PromethOS plugins are ordinary kernel modules which implement PromethOS specific functions. They can be loaded manually by the user (via the well known `modprobe` command) or PromethOS loads them automatically when needed. PromethOS creates a new instance of a PromethOS plugin by generating a unique PromethOS instance number and then invoking the config function of that PromethOS plugin handling over the new generated PromethOS instance number. Thus the responsibility for maintaining different instances of a PromethOS plugin lies by the plugin itself, which is a very important aspect.
What follows next is a short description of a part of the PromethOS functionality which is relevant in respect to the design of the Proxy Framework:

- Packet Selection
  Not all incoming or locally generated network packets are handled by PromethOS plugins but only those packets the user asked for (in fact the packets are already filtered by netfilter).
- Configuration / Reconfiguration
  Every instance of a plugin is configured by invoking the config function with an unique PromethOS instance number.
  Reconfiguration of a instance of a ProtmethOS plugin is possible via the PromethOS instance number.
- Network Packet Delivery
  All wanted packets are handled to the plugin by PromethOS by calling the plugin' s target function. Again it is task of the plugin to find out to which instance of the plugin the network packet belongs to.

1.4.2 Functions of PromethOS Plugins

An overview of the most important functions of a PromethOS plugin:

- load
  `static int __init load(void)`
  Called once when the PromethOS plugin is loaded into memory
- unload
  `static void __exit unload(void)`
  Called once when the PromethOS plugin is removed from memory
- config
  `unsigned int config(const char *config, unsigned long instance)`
  Called every time a new PromethOS instance is created and a new Plugin instance should be created (remember that this is the task of the PromethOS plugin itself).

- reconfig
  `unsigned int reconfig(const char *config, unsigned long instance)`
  Can be used to reconfigure a particular plugin instance, invoked by writing to a special file in the proc file system.
- target

```
unsigned int target(struct sk_buff **pskb,
                    unsigned int hooknum,
                    const struct net_device *in,
                    const struct net_device *out,
                    unsigned long instance)
```

  Called every time a packet for one of the instances of a PromethOS plugin arrives
- promethos_init

```
promethos_init("some name",
               load,
               target,
               config,
               re-config,
               NULL,
               NULL);
```

  Called when the PromethOS plugin is loaded into memory, handles function pointers of all above mentioned functions (besides unload) to PromethOS.

The concrete usage of a PromethOS plugin looks like this:

```
# iptables -t promethos <criteria for packet> -j PROMETHOS --plugin \
<plugin_name> --autoinstance --config '<config string>'
```

(for `iptables` which is part of the netfilter see [5])

This command will trigger PromethOS to create a new unique instance number (−−autoinstance). If the PromethOS plugin (<plugin_name>) was not already loaded into memory PromethOS will load it and execute the load and init function of the PromethOS plugin. Then the config function of the PromethOS plugin is called with function parameters containing the new generated PromethOS instance number and the config string. The intention of this call is that the PromethOS plugin now creates a new plugin instance which will be linked to the PromethOS instance number (Figure 1.2). Creation of this link is so important because later when PromethOS calls the target function to deliver a packet to a PromethOS plugin instance PromethOS will again present this instance number to enable the PromethOS plugin to find the PromethOS plugin instance with this PromethOS instance number (Figure 1.3).

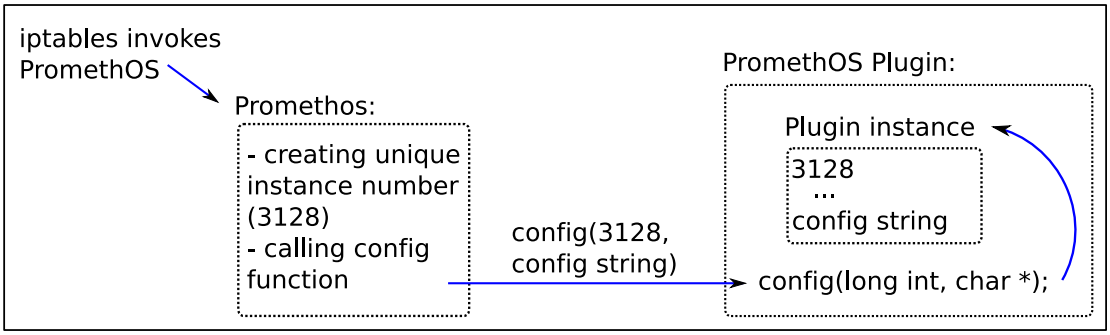

Figure 1.2: Creating a new plugin instance

After the command has been issued PromethOS will deliver every packet which matches the specified criteria (e.g. all packets with destination port 80) to the mentioned PromethOS plugin.

PromethOS does this by calling the target function of the plugin and besides other information handles over a pointer to that packet and the PromethOS instance number as function parameters (Figure 1.3).

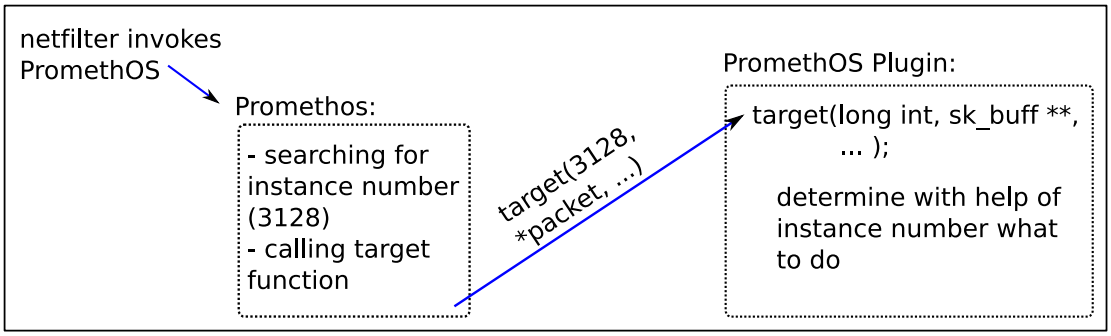

Figure 1.3: PromethOS calls the plugin target function

1.4.3 Usage of PromethOS Plugins

PromethOS plugins can be loaded and configured with the iptables tool by adding a rule to the PromethOS table, define PromethOS as target function and specify the desired PromethOS plugin. An example:

```
iptables -t promethos -A INPUT -s 192.168.1.1 -j PROMETHOS \
-plugin TEST -autoinstance -config 'some config string'
```

The relevant parts of PromethOS are:

- `-t promethos`
  Choosing the PromethOS table (see [1])
- `-j PROMETHOS`
  Jumping to the PromethOS target (see [1])
- `-plugin TEST`
  Using PromethOS plugin with name `TEST`
- `-autoinstance`
  Telling PromethOS to take care of the instance number management.
- `-config '...'`
  Optional config parameters which will be parsed to the PromethOS plugin

Chapter 2

Related Work

A short overview of products and approaches covering the issues of packet interception and proxies which demonstrates why it makes sense to design and implement a proxy framework.

2.1 Airlock

Airlock is a commercial software product developed and distributed by the Swiss company Seclutions AG[1]. At least according to the data sheet the product seems to be able to protect all variants of HTTP (Hypertext Transfer Protocol[9, 10] and HTTPS (HTTP Secured with SSL/TLS[12] applications completely against malicious attacks. It also supports hardware based en- and decryption of SSL traffic. The software is installed on a dedicated system and the hardware must be a SPARC[2] (based server from Sun or Fujitsu-Siemens.

2.2 Tetrade Secure Entry Server (SES)

SES[3] is also a commercial software product distributed by a company called United Security Providers. Installed on a dedicated hardware system and placed between network connection and web server the software protects web applications by examining all web-related traffic destinated to the server. The software has to be installed on a Sparc or HP Integrity server.

2.3 Squid

Squid[4] is a widely known opensource software which implements a full-featured web proxy cache. It supports proxying and caching of HTTP, proxying for SSL, cache hierarchies, transparent caching and much more. The functionality can further be extended by other software such as graphical interfaces, virus scanners and so on. The software is available for free as sourcecode for linux operating systems and runs as an application in userspace.

2.4 Basic Modules for PromethOS Plugins

A document[7] of the FAIN consortium (Future Active IP Networks) presents functions to process and modify HTTP specific data. The functions are designed to be used inside PromethOS plugins.

[1] www.seclutions.com
[2] Scalable Processor ARChitecture, big-endian RISC microprocessor instruction set architecture
[3] http://www.united-security-providers.ch
[4] http://www.squid-cache.org

2.5 Summery

Summarized Airlock, SES and Squid implement protocol interception but only for HTTP and/or HTTPS applications. The FAIN document only describes functions for HTTP data processing. But what about performing protocol interception on other protocols? To handle different protocols but also be able to reuse common parts of the protocols it would be nice to have one platform which allows to modular implement different applications performing protocol interception on different protocols.

Chapter 3

Architecture

This chapter presents the architecture of the proxy framework by describing the basic ideas of the framework and by naming the necessary functionality and data objects needed to successfully implement the proxy framework.

3.1 Overview

Figure 3.1 shows the overall structure in which the proxy framework is located: PromethOS connects itself to all five netfilter hooks and therefore is able to forward any network packets to PromethOS plugins which have been loaded and registered at PromethOS before. The proxy framework will be implemented as one PromethOS plugin. The functionality (and if needed additional data structures) of a proxy performing tasks on network packets of a specific protocol will be implemented in one or several proxy framework modules (functions implementing the neccessary functionality are called user functions). The terms introduced in figure 3.1 will be used in the remaining chapters.

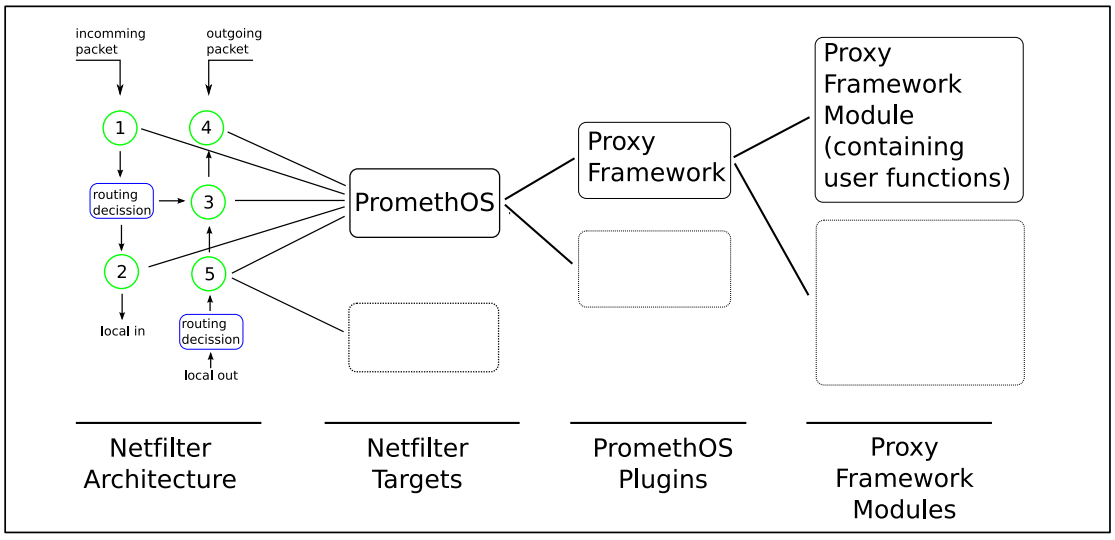

Figure 3.1: Architecture and Definitions

3.2 Basic Ideas of the Proxy Framework Design

The basic idea is that the proxy framework only provides data structures and functions necessary to maintain:

- proxies for diefferent protocols using user functions defined in proxy framework modules

- a generic data structure helping a proxy configuration to keep track of connections.

It is important to realize the difference between a proxy framework module and a instance of a proxy: The proxy framework module specifies how network packets (of a specific proxy) are processed or modified. A instance of a proxy is created at runtime. IT processes or modifies all received network packets as specified in proxy framework modules. So two proxy instances can use functions of the same proxy framework module or the user functions from different proxy framework modules. Information about the proxy instances are stored at the proxy framework.
Furthermore the proxy framework strictly differentiate packet flow from client to server and those from server to client (Figure 3.2). This helps to simplify design and implementation of the proxy framework.

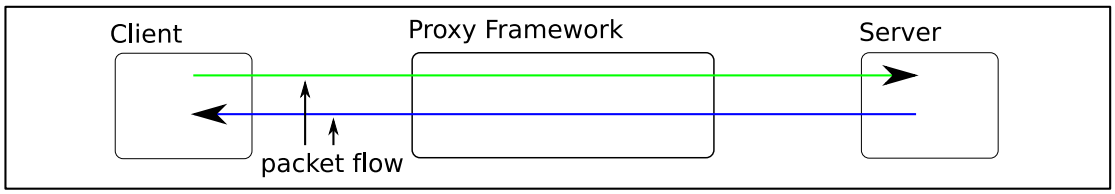

Figure 3.2: Different packet flows

3.2.1 The PromethOS Plugin for the Proxy Framework

The core of the proxy framework is designed as a single PromethOS plugin (from now on called PPFP (PromethOS Proxy Framework Plugin).
To configure one proxy instance two invocations of the PPFP are necessary, which will yield to two PromethOS instance numbers and two calls to the config function of the PPFP. Therefore every proxy instance is uniquely defined by two PromethOS instance numbers (Figure 3.3).

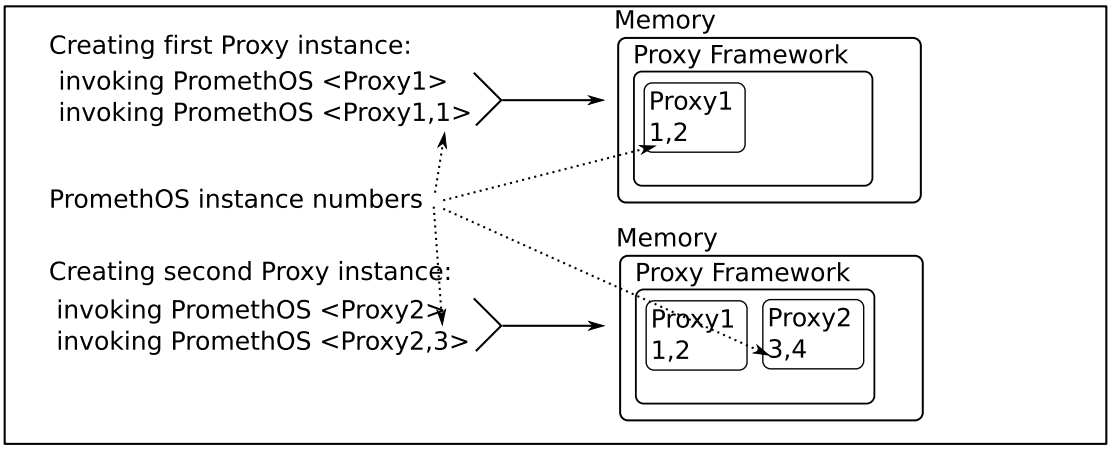

Figure 3.3: Definition: Proxy instance

The PPFP has to specify a function for the proxy framework modules to register user functions at the PPFP for the purpose of callback (in C this is realized with function pointers, see also [19]). In order to store these information about user functions the PPFP also needs some data object. Both aspects are depict in figure 3.4. The PPFP also offers a data structure for storing information about connections[1] and some functions for maintaining several instantiations of this data structure (mainly add / delete entry). Furthermore functionality to allow the proxy framework modules to link the start of a list of connection entries to a proxy instance for a fast access to connection information.

[1] virtual tunnel to transer network data

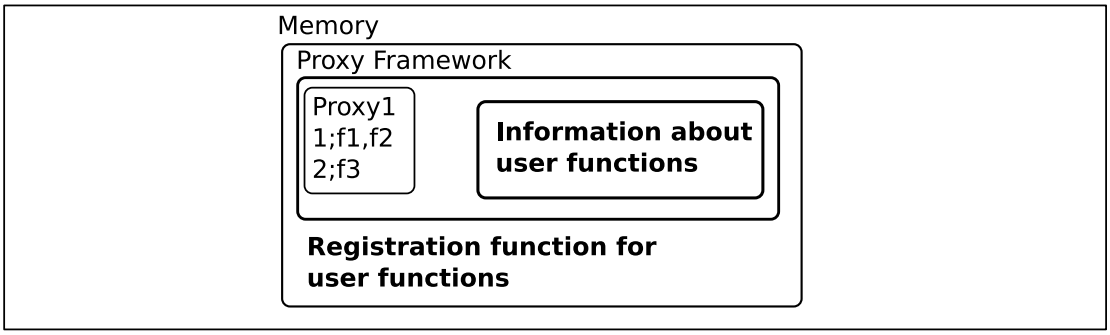

Figure 3.4: User functions at the Proxy Framework

3.2.2 Proxy Framework Modules

Proxy framework modules (PFMs) contain one or several user functions (and if necessary additional data objects needed by the user functions) which implement the functionality of a specific proxy application[2].
When a PFM is loaded it has to use the registration function provided by the PPFP to register its user functions at the PPFP for callback. Otherwise the PPFP will not be able to call the user functions of that PFM.

3.2.3 Initializing the Proxy Framework

This is done by loading the proxy framework kernel module (= PPFP). After that all PFMs should be loaded too. Now the user can start to create proxy instances and use all user functions of the loaded PFMs.

3.2.4 Creation and Configuration of a Proxy Instance

Each of the invocations of the PPFP creates one PromethOS instance number and with each invocation the user configures one packet flow (figure 3.2, client to server or server to client) of the proxy instance. Hence each packet flow (defining which kind of data is delegated to this proxy configuration) is binded to a PromethOS instance number (this binding is pictured in figure 3.5). Furthermore with each invocation of the PPFP the user has to specify which functions the

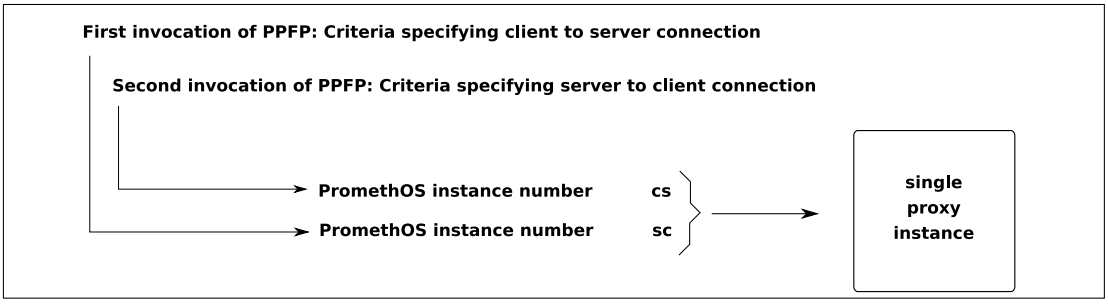

Figure 3.5: Binding of packet flow and PromethOS instance number

proxy framework should call later for that packet flow. With every second invocation of the PPFP the user has also to handle over the PromethOS instance number which was created at the first invocation of the PPFP. Otherwise the PPFP is not able to link the two PromethOS instance numbers together to one proxy instance. For an illustration see figure 3.6: The first invocation of the PPFP allocates space for a new proxy instance, tells the PPFP that this is the configuration for the direction client_to_server and that for this direction the user functions f1 and f2 should

[2]The functionality of a proxy application can also be implemented in several PFMs and one PFM can contain the functionality for several proxy applications

be called on the arrival of network packets. With the second invocation the user tells the PPFP the PromethOS instance number resulted out of the first invocation of the PPFP, specify the the direction for this configuration (server_to_client) and that the PPFP should call the user function f3 for arriving network packets.

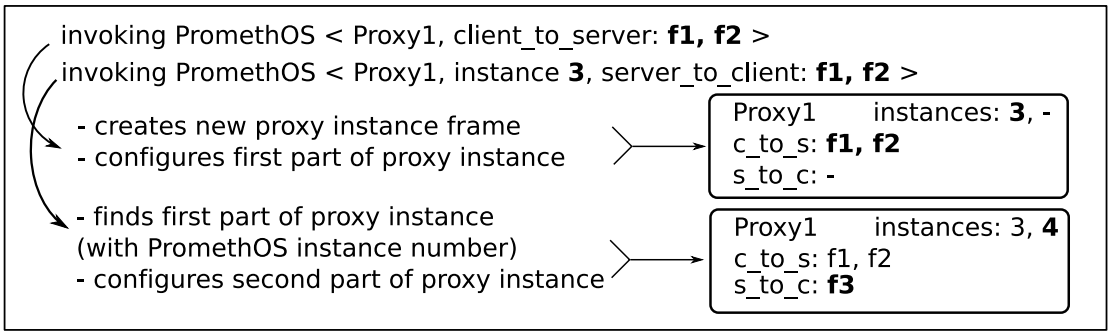

Figure 3.6: Creation and configuration of a proxy instance

3.2.5 Incoming Packet for some Proxy Configuration

After the configuration network packets which matches with one of the proxy instances, will be received and forwarded by PromethOS to the PPFP together with the PromethOS instance number. With the knowledge of the PromethOS instance number the PPFP now searches for the proxy instance which 'owns' this PromethOS instance number. When found the PPFP will execute all user functions defined in the proxy instance configuration for this PromethOS instance number. For a better understanding take a look at figure 3.7: The PPFP is invoked by PromethOS which handles over a link to the network packet to the PPFP together with the PromethOS instance number. PromethOS does this by calling a specific function of the PPFP (`promethos_target_function`). Now the PPFP searches for the matching proxy instance (Proxy4) and executes the previously configured user functions (f3 and f5).

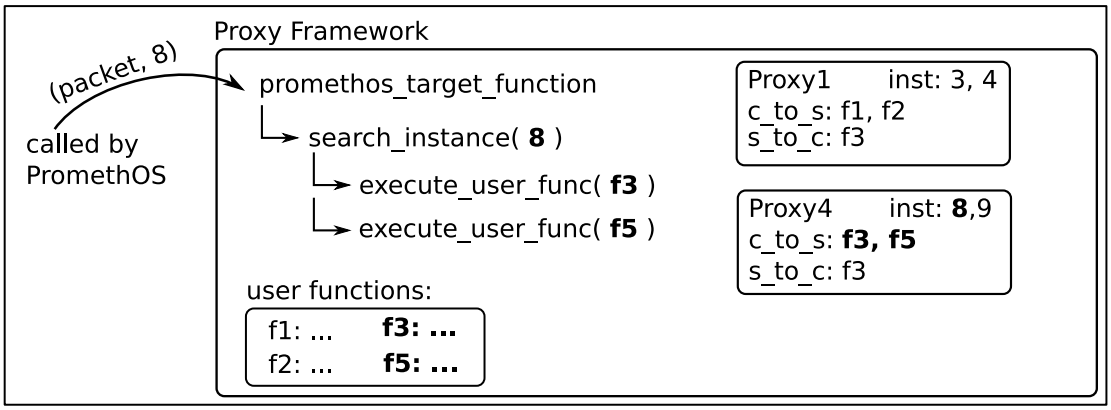

Figure 3.7: PromethOS calls the proxy framework target function

3.2.6 Design Discussion

What follows is a short argumentation of the most important design decisions.

Creating a proxy configuration

A proxy instance can be created by linking the proxy instance with

- one PromethOS instance number

- two or more PromethOS instance numbers

Linking the proxy configuration to only one PromethOS instance number would mean that the PPFP or a user function first has to find out to which direction the packet belongs to (client to server or server to client). This is due to the fact that the proxy framework is meant to be used for proxy applications. And in proxy applications the action very often strongly depends on the direction of a packet (client to server or server to client). E.g. think of a HTTP proxy which implements URL[3] rewriting. When the client sends its request (e.g. for www.ethz.ch) the proxy will rewrite the URL (e.g. to www.web.ethz.ch). When the server returns the requested data the proxy now has to do the inversion of the last action (www.web.ethz.ch -> www.ethz.chl). Otherwise the client would be confused when receiving the data.
In contrast to this with two PromethOS instance numbers finding the direction of the packet first in no longer necessary. And having two PromethOS instance numbers produces just a little overhead at implementation and no overhead at runtime.
These arguments also show that it makes no sense to use more than two PromethOS instance numbers. So the PPFP was designed to use two PromethOS instance number to uniquely describe a proxy instance.

PromethOS Proxy Framework Plugin and Proxy Framework Modules

Inspection or modification of network packets could be done by

- the PromethOS Proxy Framework Plugin
- user functions of the Proxy Framework Modules

In the first case the functions must be implemented 'proxy application independent' because the PPFP should be useable for various proxy applications.
Placing the functions into PFMs(the second case) and configuring the PPFP to call those functions is much easier. Users are able to define and implement new user functions in form of a PFM even after the PPFP has been implemented. Also this defines a clear separation of the responsibilities of the PPFP and the PFMs
So the second variant simplifies the implementation of both PPFP and PFMs and therefore was chosen for the proxy framework.

3.3 Summery

This chapter described the architecture of the proxy framework which is separated into the PromethOS Proxy Framework Plugin and several Proxy Framework Modules. Besides describing the mode of operation of the proxy framework mainly the necessary functionality and data structures were derived, which are summarized below. Chapter 4 will describe the implementation of these items:

- Functionality
  - Registering user functions at the proxy framework
  - Create proxy instances
  - Finding a proxy instance when given a PromethOS instance number
  - Execute user functions registered at the proxy framework
  - Free memory on exiting / unloading the proxy framework plugin
- Data Structures
  - Storing information about the user functions for callback
  - Storing information about the proxy instance:
    - Two PromethOS instance numbers
    - Two data structures for the user functions
    - Additional space for information about connections

[3] Uniform Resource Locator, see http://en.wikipedia.org/wiki/Uniform_Resource_Locator

Chapter 4

Implementation

The chapter starts with the description of the proxy framework implementation. The second part shows design and implementation of a Proxy Framework Module. In the third and fourth part basic ideas of how a HTTP respectively HTTPS proxy could be implemented as a Proxy Framework Module are presented.

4.1 Proxy Framework

The implementation details of the proxy framework (in the following also referenzed as PPFP) are described in the following subsections:

4.1.1 Most important data structures

4.1.2 Functions mainly used during the initialisation phase

4.1.3 Functions mainly used at runtime

4.1.4 Further data structures and functions mainly for supporting user functions. Also some generic information concerning memory allocation.

4.1.1 Data Structures of the Proxy Framework

A short overview of the implemented data structures of the proxy framework. The usage of the data structures will become clear later on when describing the functions of the proxy framework.

Data structures defined in promethos_PROXY.c

(reachable by all functions inside the PPFP)

- `struct proxy_generic *proxy_generic_first`
  used to point to the beginning of the linked list of proxy instances.
- `struct user_function *first_function`
  used to point to the beginning of the linked list of user functions.

Data structures defined in proxy_structures.h

- `struct proxy_generic`
  One instance of this struct is used to uniquely describe one proxy instance.
- `struct connection_data`
  Two instances of this struct are used to uniquely describe one bidirectional connection. Most parts of this struct are very similar to the data structure `sk_buff`[1]. The intention was to create a structure which can be used to store information contained in the protocol

[1] See `/include/linux/skbuff.h` in the linux kernel source code

headers of layer two, three and four of the ISO/OSI model. Furthermore the struct containts items making it possible to build a double linked list out of instances of this struct.

- `struct user_function`
  Every instance of this struct keeps information about the name of one user function and the corresponding function pointer.

4.1.2 Initialization Phase

When the PPFP is loaded into memory (either manually with `modprobe` or automatically by PromethOS when needed for the first time) the `promethos_init` function is called (provided by PromethOS) which will hand over a function pointer for each of the following functions:

- `load`
- `proxy_config`
- `proxy_reconfig`
- `proxy_target`

to PromethOS.
Next the two pointers (`proxy_generic_first and first_function`) are initialized to `NULL`.

Registration functions for user written functions

After the PPFP has been loaded into memory and before the user starts to create proxy instances the user functions which are intended to be called by the PPFP for the different proxy instances should register at the PPFP for callback. User functions are embedded in PFMs. A PFM is an ordinary kernel module containing a call to a registration function (of the PPFP) in its init section (`register_user_function`, if several user functions should be registered at the PPFP the registration function has to be called for every user function) . The registration function is invoked via the init function of the PFM kernel modul when the module is loaded and handles over a function pointer and the name of a user function to the proxy framework. The function pointer enables the proxy framework to call the this user function at runtime. The type definition of the function pointer is shown below.

```
typedef struct sk_buff* (*pt_to_user_function)
(
          struct proxy_generic*,       // 1 current proxy
          unsigned int,                // 2 direction
          struct sk_buff**,            // 3 skb** (received)
          struct sk_buff*,             // 4 skb_to_send
          const struct net_device*,    // 5 in,
          const struct net_device*,    // 6 out
          unsigned int                 // 7 hooknum
);
```

At runtime the PPFP will call each user function with the seven function parameters seen above:

1 Points to the current proxy instance.

2 Indicates the direction (client to server or server to client).

3 Points to the received packet.

4 Points to the outgoing packet (this can be `NULL` if no outgoing packet has been created so far).

5 Points to a struct with more or less hardware specific information about the input interface.

6 As in 5, here about the outgoing interface.

7 The hook number the packet pointed to by 3 was received at.

For registering a user function the PPFP provides the function:

```
void register_user_function
     (
          const char *name, pt_to_user_function user_function
     );
```

The PPFP also provides a function for unregistering user functions:

```
void unregister_user_function
     (
          const char *name, pt_to_user_function user_function
     );
```

The information about the user functions will be stored in a data structure called `user_function`:

```
struct user_function
{
        char *function_name;
        pt_to_user_function user_function; //function pointer
        struct user_function *next
};
```

This struct contains a pointer pointing to a char string which contains the name of the user function, a function pointer containing the address of the user function (which is necessary in order to call the user function) and a pointer pointing to the next element of this struct.
When `register_user_unction` is called the PPFP creates a new instance of struct `user_function`, fills it with name and pointer of the user function and append it to the linked list of user functions (see figure 4.1, lower part of the graphic). The pointer `first_function` points to the first element of this list. When `unregister_user_function` is called the PPFP searches for and removes the appropriate entry and frees the according memory. To use the functions the user must include `proxy_structures.h` in the PFM. An very simple example how such a PFM look like can be found in /proxy/ipv4/modull.c.

4.1.3 At Runtime

Creating a Proxy Instances

The following **two** commands will crate a new proxy instance:

```
# iptables ... -t promethos <criteria> -j PROMETHOS -plugin \
   PROXY -autoinstance -config 'cs 3 func1,func3.'
# iptables ... -t promethos <criteria> -j PROMETHOS -plugin \
   PROXY -autoinstance -config 'sc <instance-nr cs> func2.'
```

Two invocations of PromethOS (via the iptables command) are necessary to create one proxy instance. The criteria of the first command must specify the network packets of this proxy instance moving from the client to the server (e.g. TCP, destination port 80) and the criteria of the second command must specify the network packets of the proxy instance moving from the server to the client (e.g. TCP, source port 80). Both commands will result in a call to the config function of the PPFP. The actions performed by the config function depend on the first two letters of the config string.

- cs:
  The config function generates a new proxy instance by creating a new instance of type struct `proxy_generic` (by calling `register_proxy_instance` which returns a pointer to the new created struct). The structs are maintained in a link list, `proxy_generic_first` points to the beginning of this linked list. Then the PromethOS

instance number, the layer number (3 in the above example) and the char string with the names of the user functions to call are stored in the new generated struct of type `proxy_generic` (Figure 4.1).

- sc:
  The config function does NOT create a new struct instance but searches for the already existing struct instance of type `proxy_generic` by using <instance-nr cs> (see note below how to find out these value). The search is done by `find_proxy` which will return a pointer to the struct instance or `NULL` if it was not able to find a struct instance with a matching PromethOS instance number cs. Having found the instance of the struct the config function copies the PromethOS instance number and the names of the functions to call to the existing struct instance (Figure 4.1).

The sequence of function calls is as follows:

➥ `promethos_config_proxy` (called by PromethOS)

- If string == cs
  - ➥ `register_proxy_instance`
  - ➥ copy data to new created instance of type struct `proxy_generic`
- If string == sc
  - ➥ `find_proxy`
    - If returned pointer == NULL: Error
    - Else: Copy data to existing instance of struct `proxy_generic` found with `find_proxy`

Note: To find out the PromethOS instance number of the first invocation one can look at the shell output of PromethOS, use the `dmesg` or the command

```
iptables -t promethos -L
```

Note: The function names specified in the config string must have the same name as the char string of the `register_user_function` function parameter (compare with figure 4.1, func1, func2 and func3). Otherwise the PPFP will be unable to detect which functions it should call. Several functions in the config string must be separated by comma (',') and the string must be terminated with a dot ('.').

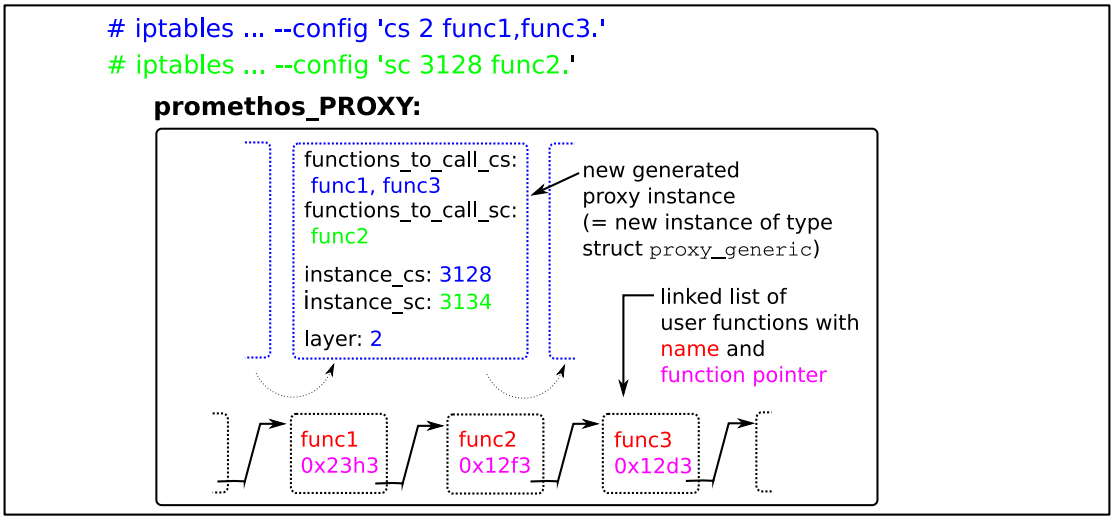

Figure 4.1: Creating and configuring an new proxy instance

Processing a Received Packet

The description of how a network packet is processed by PromethOS and the PPFP is (only for the purpose of a better readability) separated into four steps:

1. Network packet arrives at the system
   When a network packet arrives which matches one of the criteria specified at the time of the creation of a proxy instance the netfilter framework forwards it to PromethOS. PromethOS in turn calls the target function of the PPFP handling over a pointer to that packet and the PromethOS instances number which is related with the matching criteria.

2. Proxy Framework Searches Proxy Instance
   Provided with the PromethOS instance number the PPFP searches for the `proxy_generic` entry which contains this PromethOS instance number (with the function `find_proxy` which returns a pointer to the entry). Then the PPFP determines the direction of the packet by checking if the instance number presented by PromethOS matches `instance_cs` or `instance_sc`.

3. Preparing to Call the User Functions
   Now the PPFP calls `handle_packet`, handing over most information received from PromethOS, a pointer to the entry of type `proxy_generic` and the direction as function parameters. `handle_packet` now finds out which user functions it has to call (the one specified for client to server or the one specified for server to client direction). It then examines the appropriate char string (`functions_to_call_cs` or `functions_to_call_sc`) of the `proxy_generic` entry.

4. Calling the User Functions
   For every user function named in this char string `handle_packet` the PPFP tries to find a matching entry in the list of registered user functions (by searching for a matching name string). If successful, `handle_packet` executes the user function with the help of the function pointer (`pt_to_user_function`, see figure 4.1, lower part). This procedure is repeated for every user function name found in the char string.

The following call sequence shows how the different functions get called by the PPFP (`proxy_target` is called by PromethOS):

```
➥ promethos_target_proxy(**skb,instance-number,...)
    ➥ find_proxy(instance)
        ➥ handle_packet(*proxy_instance,**skb,functions_to_call,...)
            ➥ pt_to_user_function(**skb, *proxy_instance,...)
```

The PPFP expects the user functions and the function `handle_packet` to return a pointer of type `sk_buff` which points to the packet which should be send out. Note that the PPFP will allways drop the received packet and only send out the packet pointed to by `skb_to_send`. If one only want to modify the received packet (and does not want to implement protocol interception) he can simply let `skb_to_send` point to `skb`. Depending on what `skb_to_send` is pointing to one of the two folling actions will be performed:

- returned pointer is `NULL`:
  The PPFP gives out a error message and just drop the original (received) packet by returning NF_DROP to PromethOS.

- returned pointer not `Null`:
  The PPFP prepares the packet for sending out by calling `ip_route_output_key`. If the preparation was successful the return value will be unequal to `NULL` and the PPFP calls the `(skb_out)>dst->output` function to send out the packet (the mentioned functionality is provided by the `sk_buff` structure). And again after successfully sending out the new packet the PPFP will return NF_DROP to PromethOS to discharge the original packet.

4.1.4 Miscellaneous

Handling of Struct connection_data

Besides the two functions for registering and unregistering user functions the PPFP also provides two functions to add and delete objects of type struct `connection_data` for the proxy framework modules (figure 4.2). The idea here is to offer the PFMs an easy way to add connection information to a specific proxy instance.

```
static struct connection_data* create_connection_entry
(
          struct proxy_generic *proxy, unsigned int hooknum
);
```

will instantiate two empty objects of type struct `connection_data`, link them against each other and link them to the Proxy instance pointed to by `*proxy` (see figure 4.2.

Figure 4.2: Creating and linking a new pair pair of `connection_data` to a proxy instance

```
static void delete_connection_entry
(
          struct proxy_generic *current_proxy,
          struct connection_data *entry
);
```

will remove such a pair of `connection_data` from the list and free the memory which is occupied by this pair.
Inside the struct `connection_data` the following data structures for the different OSI layers are defined (for more details please look at `proxy_structures.h`):

- Layer 2:
  As in the `sk_buff` structure only a pointer of type 'unsigned char' is provided for layer 2 protocols.
- Layer 3:
  For layer 3, the user should use the struct `net_h` which contains:
  - union `net_header`:
    Used to point to different layer 3 protocol structures created by the user.
  - enum type_layer3:
    This enum is used to store the type of layer 3 protocol structure the union `net_header` points to. It is of crucial importance that the PFM as soon as it initializes the `net_header` pointer also stores the type of pointer in this enum structure (e.g. if `net_header` points to a structure of type `iphdr` the enum should be set to `t_ipv4`). This is necessary because when the PPFP is unloaded from memory it will delete all entries of type `connection_data`. Therefore it needs to know which item of the union `net_header` was initialized. If the enum does not contain the right value the kernel will crash; if the PFM uses `delete_connection_entry` at runtime, otherwise after unloading `promethos_PROXY`.
- Layer 4:
  For layer 4, the PFM should use the struct `trans_h` which contains:
  - union `trans_header`:
    This can be used to point to different layer 4 protocol structures created by the user.
  - enum `type_layer4`:
    This enum is used to store the type of layer 4 protocol structure the union `trans_header` points to. The same said for enum `type_layer3` holds also for this enum!

Not all layer 3/4 protocols are implemented so far in the struct `connection_data`. When additional layer 3/4 protocols are added also the the corresponding enum types have to be defined in `proxy_structures.h` and the function `free_connection_data_entry` in `promethos_PROXY.c` has to be adapted.

See 'Allocating new memory for structures' and 'Copy data to connection_data' of how to create objects which are referenced by the above mentioned unions.

Allocating new memory for structures

To be able to create instances of structs like `connection_data` one first has to allocate space in memory which will hold the data of the struct. Because the PPFP runs in kernel space the `malloc` function for allocating memory cannot be used. Instead for the kernel space the two functions

```
void * kmalloc(size_t size, int flags)  //defined in linux/slab.h
void * vmalloc(unsigned long size)      //defined in linux/vmalloc.h
```

are provided by C. The functions return a pointer pointing to the beginning of the reserved memory space. To free memory space which was reserved by `kmalloc` or `vmalloc` use the the functions:

```
void kfree(const void *ptr) //for kmalloc
void vfree(void *addr)      //for vmalloc
```

`ptr` / `addr` are pointers pointing to the beginning of the occupied memory. Make sure that the pointer points to the right place and not for example to `NULL`. Otherwise the usage of `kmalloc` / `vmalloc` will result in a kernel crash (as I noted sometimes it could take several minutes until the system crashes!). Further information about `kmalloc / vmalloc` can be found in [8].
For the purpose of illustration an example how to use `kmalloc` to allocate memory for data structures.

```
struct connection_data * temp =
       (struct connection_data*) kmalloc(sizeof(struct  \
                        connection_data), GFP_ATOMIC);
```

will reserve space for an structure of type `connection_data` in memory an return an pointer pointing to the beginning of the memory which will be stored in `temp`. Now to create for example space for a structure of type `tcphdr`

```
temp->trans.trans_header.th =
      (struct tcphdr*) kmalloc(sizeof(struct tcphdr),GFP_ATOMIC);
temp->trans.t_layer4 = t_tcp;             //DO NOT forget this!!!
```

If you now wish to free the memory space occupied by temp (and temp->trans.trans_header.th you can do this by

```
# kfree(temp->trans.trans_header.th);
# kfree(temp);
```

Note: You first have to free `temp->trans.trans_header.th` before you free `temp`. Also here the kernel will crash if you do not obey this. For creation of a new structure of type `connection_data` you could also use `create_connection_entry` provided by the PPFP.
Note: For all data structures provided by the PPFP (specified in `proxy_structures.h`) use `kmalloc` for memory allocation because the PPFP will use `kfree` to free the memory (at unloading the module). For more examples for the usage of `kmalloc` lock at `promethos_PROXY.c` and `module_tcp.c`.

Copy data to connection_data

The PPFP provides a function which copies the data from the pointer **`skb` (which is handled to the Proxy Framework and 'forwarded' to the user functions (pointing to the received network packet)) to a structure of type `connection_data`. This function can be used for example when one wants to store information about a new connection.

```
struct connection_data* copy_data_from_skb
        (
          struct sk_buff **skb,
          struct connection_data *c_data,
          unsigned short int mode
        )
```

The function finds out which protocols are used on the different layers and depending of the value of `mode` will copy data from layer 2, 3 and 4 to the `connection_data` structure. More details about the usage can be found in `promethos_PROXY.c`. At the time of writting the function only supports the protocols IPv4, TCP (Transmission Control Protocol[17] and UDP (User Datagram Protocol[18]).

Unloading the PPFP

This means calling

```
# rmmod promethos_PROXY
```

This invokes the unload function of the PPFP. This function in turn calls several functions to free the occupied memory space.

Error Messages

Error messages are implemented as `printk` messages, error messages can be seen with invoking `dmesg`. For more information on the `printk` command see [4] or [8]

4.2 TCP State Machine

This section describes the architecture and implementation of a simple TCP state machine as a PFM. The section is divided into three parts:

4.2.1 Motivation for a TCP state machine PFM.

4.2.2 Description of required functionality in the TCP state machine PFM

4.2.3 Shows how functions and data structures are implemented

4.2.1 Motivation

The data of many application protocols (like HTTP) is transfered reliable using TCP. So if the PPFP should be used to operate on application protocols it is likely that the PPFP has to deal with TCP issues. Implementing TCP related functionality inside a proxy framework module enables the user to create a proxy instance performing TCP related operations. What makes the situation concerning TCP inside the PPFP a little bit tricky is that the PPFP does not present an endpoint of a TCP connection[2]. So there exist no sockets[3] in the kernel space and therefore the TCP state machine of the Linux kernel cannot be used because it only works with sockets.
The TCP state machine PFM also implements protocol interception: For security reasons for every processed network packet a new network packet structure is created and the data of the original network packet is copied controlled to the new network packet structure (see figure 4.3).

4.2.2 Requirements

The sequence of necessary processing steps which must be implemented inside the PFM of the TCP state machine:

- ➥ packet arrives

[2] uniquely defined by source and destination IP and port
[3] for access of network packets in the user space, see `include/net/sock.h`

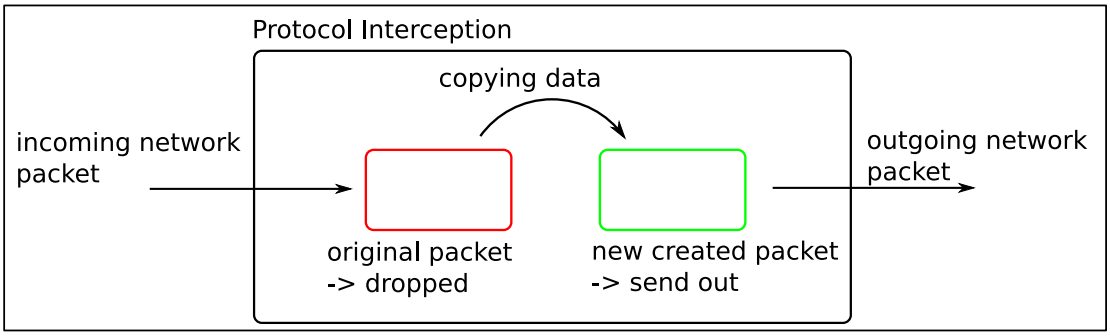

Figure 4.3: Protocol interception

- check if this packet belongs to a already existing connection (this is the case when the SYN flag is not set):
  - Yes ⇒ search information about this connection (stored in some data structure)
  - No ⇒ create new data structure for the new connection
- create new packet
- copying data from original packet to new created packet
- update information of TCP connection

Regarding protocol interception (figure 4.3) there are two different possibilities (called Modell A and B in figure 4.4) how to handle TCP sequences. By implementing method A the error handling (concerning out of the order packets, duplicates and so on) is nearly completely delegated to the client respectively to the server and so does not have to be done by the proxy. To illustrate this an simple example: Imagine the client wants to connect to the server `www.ethz.ch:23`. Using method B the proxy would reply and finish the TCP handshake pretending to be `www.ethz.ch`. Next the proxy sends a SYN packet to `www.ethz.ch:23` which will immediately answered with a RST packet. Of course now the Proxy could send a RST packet to the client but the proxy has already sent a lot of packets out for nothing. This behavior of the server (simulated by the proxy) looks strange to the client (first agreeing in the connection by finishing the TCP handshake and then directly RST this connection) and would not have happened when using method A (for general information regarding TCP handshake and SYN and RST packets see [17]).
Also (by using method B) the proxy would introduce a bigger delay (it takes some time until the client receives the RST packet).
Because of the big advantages Model A was chosen to be implemented inside the TCP state machine PFM.

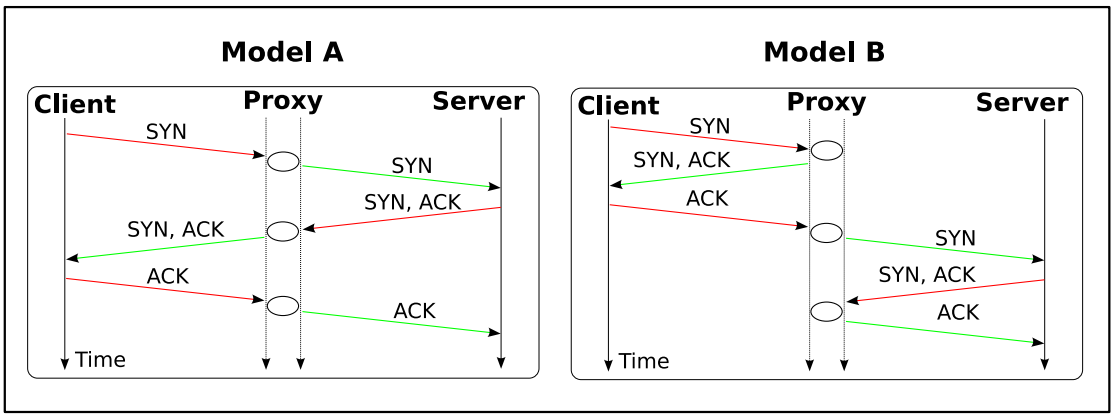

Figure 4.4: Two methods how to handle a TCP handshake

4.2.3 Implementation

Out of the aspects mentioned in 4.2.2 the TCP state machine PFM must provide functions and data structures for:

- Saving information about TCP states inside the PPFP
- Copying data from the received (original) network packet to the new created network packet
- Create new network packets

Storing TCP States

To be able to remember states of TCP connections the data structure `connection_data` also provides structures which can be used for storing TCP states. These structures are:

- union `conn_track`

```
union {
       struct tcp_states cp; //client->proxy
       struct tcp_states sp; //server->proxy
      } conn_track;
```

- variable `timestamp`

```
unsigned long timestamp;
```

where the struct `tcp_states` is defined as follows:

```
struct tcp_states
{
        __u8
        syn_received:1,
        syn_sent:1,
        syn_ack_received:1,
        syn_ack_sent:1,
        conn_established:1,
        fin_received:1,
        fin_sent:1,
        conn_closed:1;
};
```

Because there might be scenarios where TCP control packets (as SYN, ACK and FIN, see [17]) are not received and forwarded by the proxy at the same time the above struct provide two 'states' (`*_received` and `*_sent`).
Information concerning the TCP header (like TCP sequence number ...) can be saved in a struct `tcphdr` which can be linked to the data structure `connection_data`. The above union is used to be able to separate one bidirectional TCP connection between two communication parties (e.g. server and client) into the two unidirectional TCP connections (e.g. server to client and client to server). This is because the TCP protocol specifies the possibility that a TCP communication between two parties can be unidirectional (also in most cases it is bidirectional). This requires to use the `tcp_states` union twice, once initiated as `cp` and once as `sp` (Figure 4.5). Therefore two instances of type `connection_data` are necessary. These are created with `create_connection_entry` (one invocation!) which will create two instances of type `connection_data`, link the two instances against each other and add the entries into the doubled linked list which can be accessed via the pointer `connections_client` and `connections_server` contained in the proxy instance of type `proxy_generic` (figure 4.5). Information contained in the IP and TCP header can again easily copied to the data structure `connection_data` by using `copy_data_from_skb`.

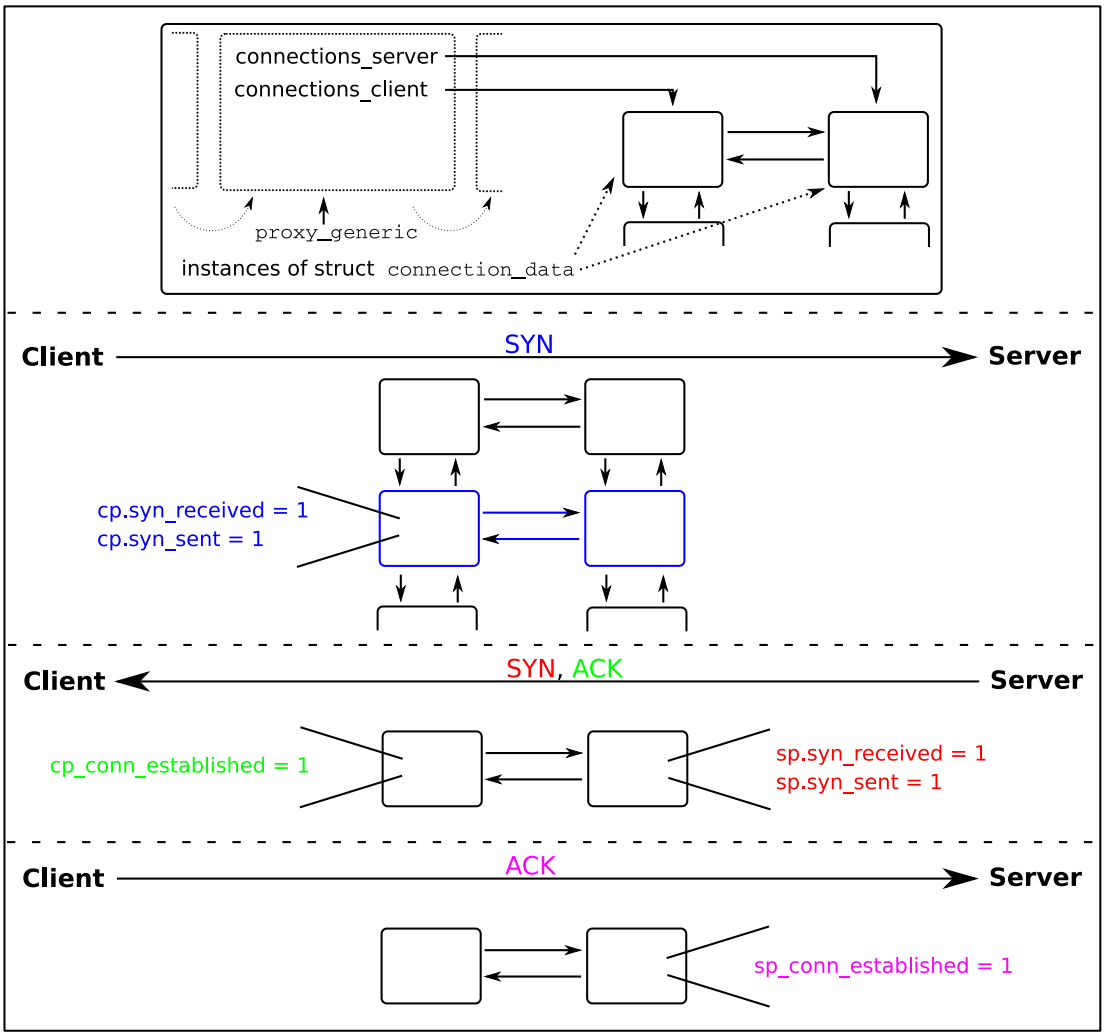

Figure 4.5: Example for usage of union conn_track

Function handle_tcp_packet

The main function `handle_tcp_packet` is registered at the PPFP as user function and is called for every packet matching the specified criteria at configuration time. When the function `handle_TCP_packet` is called the PPFP has already determined the direction of the packet (client to server or server to client) and the target function of the PPFP delivers this information to the `handle_TCP_packet` function. Because depending on the direction TCP state information has to be stored at different places and because the code is already complex the implementation was divided into two parts, one handling the direction client to server and one handling the direction server to client. Depending on the direction one of the two parts will be processed. The TCP state machine PFM uses the four part tuple (consisting of source and destination IP and port) to separate different TCP connections. Regardless of the direction of the packet the following operations are performed every time the `handle_TCP_packet` function is called:

- Searching if there is already an entry which matches the four parts of the tuple of the the received packet (by using `has_TCP_entry`).
  - NO ⇒ This must be a new connection and the function `create_connection_entry` will be used to create a new object for storing connection information (but only when only the syn flag of the received packet is set). Necessary information about the received network packet is copied to the new created data structs (of type connection_data). Next the corresponding TCP state values are set and the other values of the new created object are initialized (set to '0', in the `tcp_states` struct).

  - YES ⇒ By evaluating the stored TCP information and the TCP flags of the received packet the module updates the TCP states (if possible).

- By calling `set_up_TCP_packet` a new nework packet (containing IPv4 and TCP header) is is created and the information of the received network packet is transfered to the new network packet. This new packet is linked to `skb_to_send` and is sent out by the PPFP after the PPFP has called all configured user function for this proxy instance.
- The module returns the new created packet to the PPFP (as pointer) or `NULL` if it was not able to perform a TCP update operation.

The following call sequence should show how `handle_TCP_packet` processes a TCP packet: (`handle_TCP_packet` is called by `handle_packet`)

➥ `handle_TCP_packet`

  ➥ `has_TCP_entry`

  – `has_TCP_entry` returns `NULL`:

    ➥ `create_connection_entry`
    ➥ `copy_data_from_skb` // for client side data
    ➥ `copy_data_from_skb` // for server side data
    ➥ `set_up_TCP_packet` // new packet
      ➥ `build_TCP_packet`
      ➥ `checksum_TCP`
      ➥ `checksum_ip`
    ➥ `copy_TCP_data` // copy user data (if necessary)
    ➥ `checksum_TCP` // (if necessary)
    ➥ `checksum_ip` // (if necessary)

  – `has_TCP_entry` finds a connection entry:

    ➥ `copy_data_from_skb` // update TCP data
    ➥ `set_up_TCP_packet` // new packet
      ➥ `build_TCP_packet`
      ➥ `checksum_TCP`
      ➥ `checksum_ip`
    ➥ `copy_TCP_data` // copy user data (if necessary)
    ➥ `checksum_TCP` // (if necessary)
    ➥ `checksum_ip` // (if necessary)

Comments

The described TCP state machine PFM does not implement the whole functionality needed for handling TCP traffic completely. But also this simple, incomplete implementation was able to show that it is possible to keep track of TCP states and also that it is possible to do the protocol interception. Some features which were not implemented:

- Sequence Numbers:
  The TCP state machine PFM does not check the correctness of the sequence numbers (but this can perfectly done at the connection end point). Also the PFM does not exchange the sequence numbers (it just reuses the one of the received packet by copying them over to the new generated packet.
- Time Outs:
  The TCP state machine PFM only shows first approaches of how to handle TCP connection time outs (see [17]. The idea is to set a timestamp in the `connection_data` structure and then (directly or delayed) add the `connection_data` structures of connections

which have been correctly been closed or timed out to a list of removal candidates. An other function regularly checks the 'age' of the timestamps of the removal candidates. If it is to 'aged' the connection entry will be removed from the list of connections. Implemented functions for this are:

  - `add_connection_entry_to_removal_candidates`
  - `check_removal_candidates`
  - `delete_removal_candidate_entry`
    (which uses `delete_connection_entry` offered by the PPFP)

- Unidirectional Connections:
  These are not supported so far concerning the needed functionality which detects such connections and updates the TCP states. As said before the implemented data structures fully supports this TCP connection type.

Creating TCP packets

- `build_TCP_packet`
  Creates a new `sk_buff` structure with the necessary formation for a TCP/IP packet. This function very well demonstrates how to build a `sk_buff` structure in general and can easily be adapted for other protocols (like UDP...).

- `set_up_TCP_packet`
  Copies the data of struct `connection_data` into the new created packet and updates checksums (IP and TCP).

Miscellaneous

The function `search_for_ip_output_interface` tries to find the IP address of the outgoing interface of a packet.
Note: You can only use this function on packets of type `**sk_buff` which was already be processed by the routing machinery of the kernel. Figure 1.1 shows where this is the case.

4.3 HTTP Proxy

This section describes the basic ideas of how a Hypertext Transfer Protocol (HTTP) proxy can be realised as a PFM for the PPFP.[4]

4.3.1 Design Aspects

Aspects which have to be considered for the design of the HTTP PFM.

HTTP Protocol

The HTTP PFM must be familiar with the HTTP protocol to be able to detect malicious HTTP protocol headers which presents harm to the end system (where the HTTP server is located). By replacing those malicious data packets with new created, inoffensive ones the HTTP server can be secured. Today most applications uses HTTP/1.1**??** but for reasons of downward compatibility the proxy module should also be capable to understand HTTP/1.0[9].

[4] Due to a lack of time the HTTP PFM was not implemented

Stateless

The HTTP protocol is stateless. When a request arrives at a HTTP server, the server will respond to that request and directly after sending the response will forget about the request (when not using additional objects such as cookies[5]). That means that a succeeding request / response can not relay on information sent in the last request / response. Also the HTTP protocol is stateless and it is not necessary to keep track of connections as it must be done for TCP, the possibility exists that the response of a single request is larger than what one network packet is capable to transport. In this case the content of the response is split and transferred in several successive packets. Therefore the proxy module must implement functions and data structures enabling the HTTP PFM to detect and remember these splitted pieces of a response. The same also applies to large HTTP post requests[6].

Reliable Connection

HTTP presumes a reliable connection for sending and receiving data. This means error handling (e.g. retransmission, out of order data) is assumed to be handled by some protocol below the HTTP protocol. Therefore it is not possible that a request / response will be sent twice (or more often) without seeing also the corresponding response / request twice (or more often). So the proxy module should be act on that.

4.3.2 Implementation Aspects

Aspects which have to be considered during the implementation of the HTTP PFM:

- Before starting with the implementation determine the weaknesses of the HTTP protocol handler of the system / application which should be protected and define the exact task of the proxy (only partial support of the HTTP protocols or full)
- Next study the HTTP protocols 1.0 and 1.1 very deeply, with focus on what is really necessary in the HTTP header.
- The final implementation then consists of defining data structures and writing functions which for every received HTTP packet creates a new HTTP packet, copy the HTTP data of the received packet controlled to the new one (=protocol interception) and keep track of splitted requests / responses

4.4 HTTPS Proxy

One method to secure HTTP traffic is to use the SSL (Secured Socket Layer) or the TLS (Transport Layer Security) protocol (see [12, 13, 14] for further information on SSL/TLS). Both protocols are implemented on top of the transport layer of the OSI/ISO reference model (see 4.6) and encrypt data their receive from the upper application layer before the data is sent away. Securing HTTP that way is also denoted by HTTPS (HTTP Secured). Because the data is encrypted and because of the design of SSL/TLS a possible proxy must represent a SSL/TLS endpoint in order to decrypt the data.
This section describes how a HTTPS proxy can be realized for PromethOS in form a PFM[7]. It is assumed that the reader is familiar whith the SSL/TLS protocols.

4.4.1 Design

Assuming that an HTTP PFM already exists the task of the HTTPS PFM can be reduced to:

- The HTTPS PFM can represent a SSL/TLS connection endpoint (the PFM is able to establish and maintain several SSL/TLS connections) and can en/decrypt data for these connections.

[5] data object to circumvent the stateless of HTTP
[6] usually used to transfer data from the client to the server
[7] Due to a lack of time the HTTPS PFM was not implemented

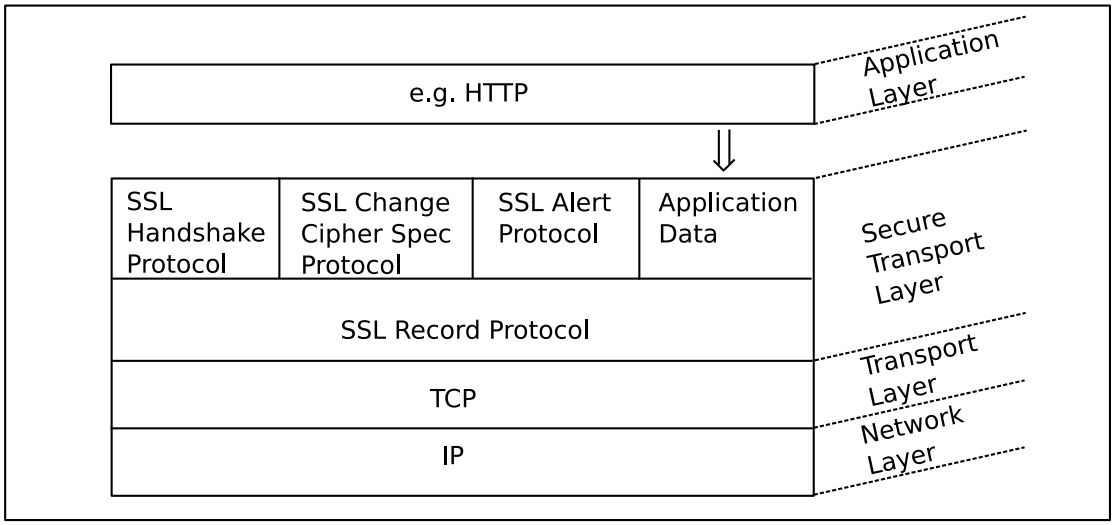

Figure 4.6: Location of the SSL/TLS protocol inside the ISO/OSI model

- The HTTPS PFM is able to setup a new network packet containing the unencrypted part of the network packed received before and the encrypted part of the received network packet after decrypted (see right side of figure 4.7).
- The HTTPS PFM is able to setup a new network packet containing the just encrypted data of a packet and the part of the old packet which was not encrypted (see left side of figure 4.7).

The HTTPS PFM does not have to implement any HTTP proxy functionality. Instead the HTTP PFM can be used for HTTP proxy related tasks:

- When the HTTPS PFM receives an encrypted network packet on a SSL/TLS connection, it decrypts it, builds a new, 'decrypted' packet and forwards this to the HTTP PFM (see figure 4.9).
- When the HTTPS PFM receives a unencrypted network packet the it encrypts the sensitive part of the part of the network packet and builds a new network packet. This new packet is then sent out (see figure 4.9).

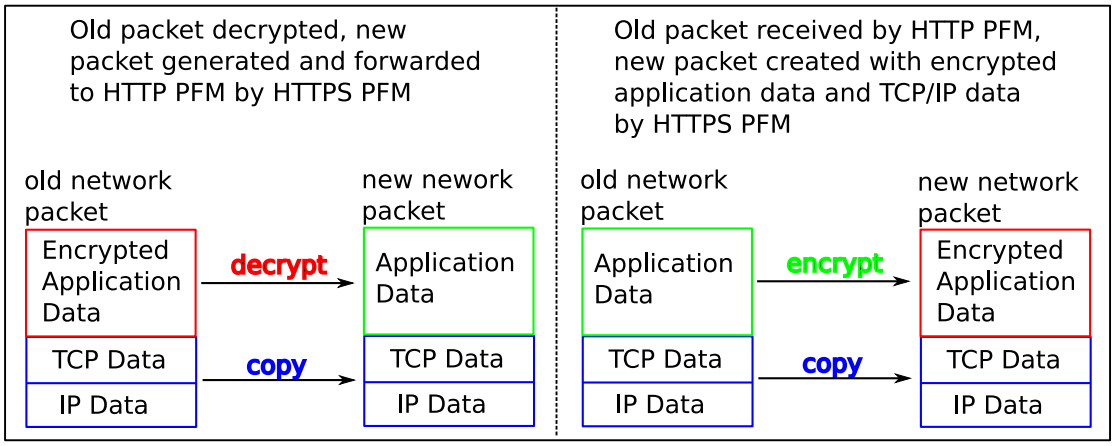

Figure 4.7: Composition of an unencrypted (left) and encrypted packet (right)

Setup

The following setup is assumed: The proxy is placed directly in front of the server and instead at the server the encryption tunnel is already terminated at the proxy (figure 4.8). Encryption is

only done between client and proxy, data between Proxy and server is not protected.

Data Structures

- Server Certificate
  In the most common applications today the client does not own a copy of the server's certificate and so this certificate, containing the public key of the server, must be transmitted in the server hello message. So a data structure to store one or more of such server certificates is needed

- Server Secret
  Close related to the certificate is a secret normally owned by the server which is used to decrypt data which was encrypted by the client with some information contained in the server's certificat. So a data structure to store one or more secrets must be provided. Furthermore attention must be paid to protect the secret because keeping the server secret secret is extremely essential.

- Session Keys
  After client and SSL/TLS server (= proxy) have exchanged the pre-master-secret both sides calculate the master secret which contains keys for encryption, decryption and MAC calculation. These keys will only be usable for one connection between a dedicated client and the SSL/TLS server. So the HTTPS PFM needs a data structure to store several instances of these key 'pairs'.

- Supported CipherSuites
  Depending on which encryption algorithms will be implemented in the HTTPS PFM a data structure for storing supported ciphers is needed. The list will not be send to the client but when the client sends several proposals for the chipher suite (which the client wants to use in the following session) the HTTPS PFM must be able to see if there is a matching cipher suite.

- Connection Tracking
  To be able to keep track of the different connections to the HTTP PFM the HTTPS PFM needs a data structure which is capable of storing all the information needed for uniquely describing different connections (see figure 4.8).

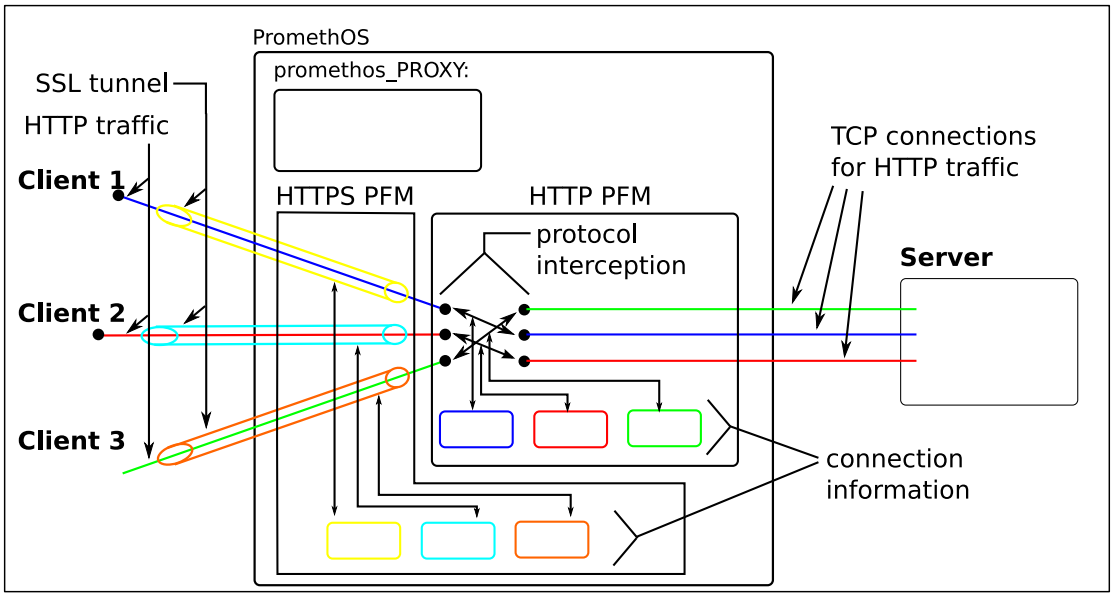

Figure 4.8: Architecture of the HTTPS Proxy

Functionality

In the assumed setup (described above) the HTTPS PFM has only to perform server side related SSL/TLS operations. These server side related functionality is listed below. If also a secure connection should be setup from the proxy to the web server the HTTPS PFM must also implement client specific SSL/TLS functionality (which is not listed below):

1. Evaluate received client hello.
2. Send out appropriate sever hello to client (after generated random values and created new instance for storing connection information).
3. Decrypt client key exchange, depending on cipher suite examine client certificate. Generate secret values out of pre master secret and storing these values.
4. Expect change chipher spec message from client.
5. After receiving cipher spec message check and reply to this mesage with cipher spec message containing information which was sent and received before.
6. After successfully established the secure channel with the client open a TCP connection to the server and link the connection to the client connection in the instance of the data structure used to describe the client connection.
7. Change Cipher Spec Protocol
   The proxy module must be able to detect this kind of protocol and react appropriate to it by creating new key material (for encryption, decryption and MAC calculations). Therefore functions for implementing the related algorithms must be provided and functions for read / write operations to the data structures storing these values.
8. Alert Protocol
   Alert messages will be received from the Client or generated by the proxy module. Most commonly these alert messages are sent when one party wants to close the encrypted channel. When received by the HTTPS PFM from the client the HTTPS PFM must inform the HTTP PFM. On the other side when the server closes the TCP connection to the HTTP PFM the HTTP PFM must inform the HTTPS PFM. So functions for evaluating and acting appropriately to the alert protocol messages and for the communication to the HTTP PFM must be implemented.
9. Processing Upper Layer Data
   When the HTTPS PFM receives encrypted data for the server it must decrypt it, create a new network packet, copy the data to that new network packet and hand it over to the HTTP PFM. The HTTP PFM now can perform HTTP related proxy operations on the network packet and send it to the a web server. Analog procedure for the other direction: When the HTTP PFM receives a network packet from the web server, it performs the proxy related operations on the network packet and forwards the network packet to the HTTPS PFM. The HTTPS PFM will create a new network packet with the HTTP data encrypted and send it out to the client. This procedure is depicted in figure 4.9.

Items 1 to 5 are the server side parts of the SSL/TLS handshake(see 4.9), 7 and 8 are parts of SSL/TLS(see [12])

4.5 Summary

This chapter described the implementation of the PromethOS Proxy Framework Plugin and of a simple TCP state machine as a Proxy Framework Module. Section 4.3 and 4.4 described how HTTP/HTTPS Proxy can be realized with the PPFP inside PFMs.
The successful implementation of the PPFP showed that a proxy can be realized inside PromethOS. The functioning TCP state machine proofed that protocol interception can be done with the PPFP and PromethOS.

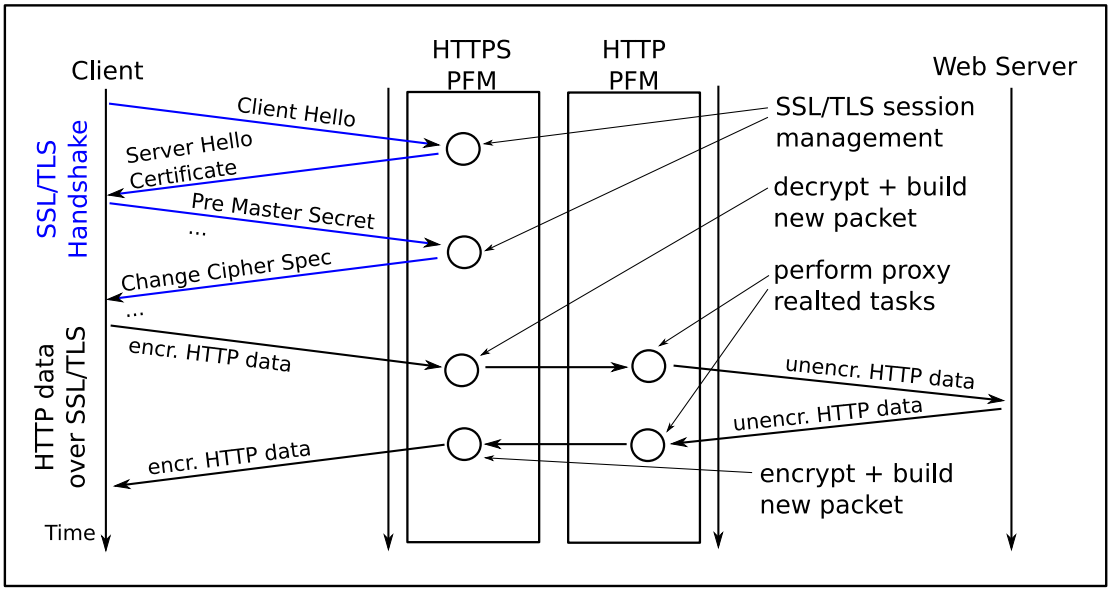

Figure 4.9: Interaction of HTTP and HTTPS PFM

Chapter 5

Evaluation

This chapter shows how the PPFP and the TCP state machine PFM were tested to ensure an error free operatin. Especially because PromethOS and therefore also the PPFP and its PFMs are running in kernel space it is of crucial importance to ensure an error free operation. This is due to the fact that most erroneous behavior will lead to a kernel crash which forces a reboot of the operating system.

5.1 Methodology

Different testing procedures for the PPFP and the TCP state machine PFM were used to verify the correct implementation. The methodology and the reasons for the different testing procedures are described in 5.1.1 and 5.1.2.

5.1.1 Proxy Framework

The PPFP mostly consists of instantiations of different data structures and pointers which links different instances with each other (via pointers). Furthermore the PPFP provides functions to create and modify these links and fill the instances with information. So it is necessary to ensure that the links are created and modified as intended and that the right information is stored at the right location. Furthermore instances of data structures which are no longer used must be removed properly. Because the complexity of the data structures and of the functions is still manageable the tests was done by looking at the concatenation of the instances and their content at various stages.

5.1.2 TCP State Machine Proxy Framework Module

This PFM is a first application for the PPFP. It extensively uses one data structure of the PPFP (`connection_data`) and the corresponding functions provided by the PPFP. Furthermore it creates new network packets and copies data from the original network packets into them. This is more complicated (compared to the PPFP) and it would cost to much time to test all the functionality in the way it was done for the PPFP. Therefore the tests for the TCP state machine PFM concentrated on looking at the network packets before and after their were processed by the PFM. This was mainly done with ethereal[1].

5.2 Testing

In 5.2.1 and 5.2.2 the test routines are described which were used to test the PPFP and the TCP state machine PFM.

[1] A tool to analyze network packets, `http://www.wireshark.org/`

5.2.1 Testing PPFP

The test procedure was as follows: After having added a new data structure and functions for creating / modifying this data structure some simple test functions were written which used the new added structures and functions and (because nearly every data structure was used in linked list) printed out the content of the linked lists. The printouts were done with the `printk` command. Furthermore a lot of mistakes 'showed up' quickly in form of a kernel panic (this means the kernel has crashed).

5.2.2 Testing TCP State Machine Proxy Framework Module

By looking at the network packets before and after their were processed by the PFM most errors concerning network packet creation and modification could be detected. By using unencrypted user data (HTTP data) it was possible to check if the user data were correctly transfered to the new network packet. During one of the tests it figured out that when TCP state machine PFM was used about 2 times more packets where used to transfer the test file. This was because the TCP options were not copied into the new packet (and among other informations the TCP options also specified the Maximum Transfer Unit (MTU)[11]. Because the web server which was sending the file had gotten no information of the the MTU by the client it used the minimum default value which is about 500byte. By copying the missing MTU value into the new created packet the erroneous behaviour was eliminated.

Performance

Also some performance tests were performed by sending data through the proxy while the TCP state machine PFM was loaded and configured to handle this connection. The test setup is depicted in figure 5.1: The web server and the system running PromethOS and the PPFP/PFM were running inside a VMware environment. The client (outside the VMware environment) used wget to fetch an about 20Mbyte file from web server. The values below are calculated out of 2000 Packets and show different values concerning the processing time of one packet:

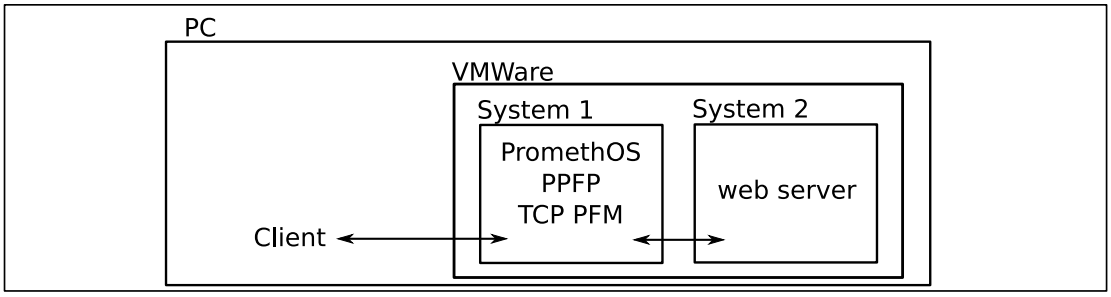

Figure 5.1: Test Setup

- Time values: Processing time for one packet
  (# Samples = 2000 packets)
  - mean: $80 * 10^{-6} sec$
  - standard deviation: $300 * 10^{-6} sec$
  - max value: $5.5 * 10^{-3} sec$
  - min value: $5 * 10^{-6} sec$
- Throughput:
  - PromethOS only: 13Mbyte/sec
  - PromethOS with PPFP and TCP state machine PFM: 10.5Mbyte/sec

The values of standard deviation and maximum are outstanding high which is most probably due to the fact that the proxy ran inside a VMware environment and therefore had not continuous access to the CPU.
The throughput decrease at about 20 percent which appears a lot at a first glance but one must obey that for every network packet the TCP state machine PFM has to create a new packet and transfer all data out of the original packet into the new one. Therefore the throughput of the TCP state machine is still passable.

5.3 Summary

The TCP state machine PFM heavily uses the `connection_data` data structure and its maintain functions and no errors occurred. This indicates that the PPFP implements this data structures and the functions correctly.
The TCP state machine PFM was always able to find an appropriate entry for every connection which shows that the module correctly updates and evaluate the TCP states.
With over 10Mbyte/sec the the throughput of the TCP state machine is still passable.

Chapter 6

Summary and Conclusion

The chapter describes what has been achieved in this semester thesis and mentions what still needs to be improved. The last part explains the meaning of the successful implementation of the PPFP and the TCP state machine PFM.

6.1 Summary

In this semester thesis the following was achieved:

- A proxy framework for PromethOS was designed which is able to maintain several proxy instances of the same or different types. The functionality for a specific type of proxy is specified inside a Proxy Framework Module.
- The correctness of the design was shown by a implementation of the proxy framework as a single PromethOS Plugin, the PromethOS Proxy Framework Plugin.
- Design of a simple TCP state machine which also performs protocol interception. The TCP state machine was implemented as a single PFM.
- The design descriptions of the HTTP and HTTPS proxies showed how these types of proxies can be realized as PFMs for the PPFP respectively for PromethOS.

6.1.1 Improvements

Concerning the Proxy Framework

The core of the proxy framework (functions and data structures) works well and so improvements more concern the usability:

- Improving the config function for an easier configuration handling.
- Automating loading of PFMs so that the user does not have to take care of loading PFMs.
- Add functions which provide information about single proxy instances. For example this could be done via the proc filesystem which is easy accessable by users and other processes.
- Add functions for calculating / searching hash values of arbitrary data out of the `connection_data` struct to speed up search processes.

Concerning the TCP state machine

The TCP state machine has not been fully implemented to cover all aspects of the TCP protocol. The most important missing items are:

- Complete implementation for connection timeouts to avoid that terminated TCP connections are stored inside the PPFP/PFM longer than necessary.

- Check and manage TCP sequence numbers. So far the TCP sequence number are not checked and the protocol interception part reuses the TCP sequence numbers by copying them from the original to the new created packet.
- Define valid and invalid combinations of TCP flags to detect attacks based on invalid TCP flag combinations.

6.2 Conclusion

The successfull implementation of the PPFP showed that it is possible to realize a proxy framework as a PromethOS plugin. This together with the successfull implementation of the TCP state machine PFM proofs that PromethOS can be used for proxy related tasks including protocol interception.
The design and implementation of the proxy framework inside PromethOS has several advantages:

- Different proxy related tasks can be implemented in different PFMs and be performed by the PPFP at the same time.
- New proxy applications can easily be implemented after the implementation and installation of the PPFP on a system.
- Also existing proxy applications realized as PFMs can easily be modified an reloaded.The modifications can concern proxy functionality but it is also possible to use a single PFM to store data e.g. types of malicious HTTP headers. When new malicious header types are found these can easily be added to a existing PFM which then only needs to be reloaded.
- Adding or modifying PFMs can be done at runtime of PromethOS/PPFP. Reloading PFMs and eventually needed reconfiguration of the PPFP can be done within seconds via the PromethOS extended Iptables command.

Compared to user space applications PromethOS and the PPFP running inside the kernel space are also more performant. But there also disadvantages regarding kernel space environment:

⇒ Limited re-usability of existing code:
This is because a lot of network code is written for user space and/or using sockets.

⇒ Debugging process:
The written code cannot be run inside a debugging software. A lot of mistakes (such as dereferencing a NULL pointer) result in a kernel crash, and even when using a virtual environment for testing the code it takes some time until the environment has recovered from the kernel crash.

Appendix A

Usage

A.1 Simple Example - modull

This is just a very simple example which should demonstrate how user functions should be registered at the PPFP:

A.1.1 Initialization

```
# modprobe promethos_PROXY // if not already loaded
# modprobe modull
```

Using `dmesg` one can see what happened:

```
PROMETHOS: init() called...
...
PROXY:  Function 'test' registered at table
```

Creating (first part of) new proxy instance (to be able to invoke the registered function 'test'):

```
iptables -t promethos -A OUTPUT -p icmp -j PROMETHOS --plugin PROXY \
--autoinstance --config 'cs 2 test.'
```

Again by using `dmesg` see what happened:

```
...
PROXY:  proxy_generic instance cs:5 sc:0
PROXY:  Functions CS test
...
```

You can see how the proxy instance is maintained by `iptables` with the following comand:

```
# iptables -t promethos -L
```

The output should look similar to this:

```
...
Chain OUTPUT (policy ACCEPT)
target       prot opt source         destination
PROMETHOS    icmp -  anywhere       anywhere      PROMETHOS PROXY\#6
...
```

A.1.2 Testing it

Now just ping some IP. You will most probably get some error mesage, ignore it. Stop pinging and again watch at the output of `dmesg`:

```
...
TEST_FUNCTION called
Proxy: Unable to process c2p packet
...
```

A.1.3 Unloading

To unload the proxy module 'modull' use `rmmod`. But before that use iptables to clean the promethos chain:

```
# iptables -t promethos -F
# rmmod modull
```

When the module is unloaded it will deregister from the PPFP:

```
...
PROXY:  Function 'test' removed
```

A.2 TCP State Machine - module_tcp

A.2.1 Initialization

Load the modules:

```
# modprobe promethos_PROXY // if not already loaded
# modprobe module_tcp
```

Configure the proxy instance:
(Assuming server has IP 192.168.4.2)

```
//for the packets client -> server
# iptables -t promethos -A PREROUTING -p tcp -d 192.168.4.2 \
-j PROMETHOS --plugin PROXY --autoinstance --config 'cs 2 tcp.'
//for the packets server -> client
# iptables -t promethos -A PREROUTING -p tcp -s 192.168.4.2 \
-j PROMETHOS --plugin PROXY --autoinstance \
--config 'sc <instance-nr> tcp.'
```

After the first `iptables`-command has been entered PromethOS will print out some messages. Look for something like:

```
...
got unique instance 7 from kernel.
...
```

In this example 7 would be the instance number for the second command (<instance-nr>).
NOTE: The number in the first config string ('cs textbf2 tcp.') was intended for different operation modes of the PPFP, but this was not implemented finally.

A.2.2 Validation for Working

If every thing was correctly configured it is now possible to transfer data over the PPFP including protocol interception. To check if every thing works fine check the ID field in the IP header. This should be '0x0000' after the packet has been processed by the TCP state machine.

A.3 TCP State Machine - module_tcp2

This version will replace the source address from the client with the one of the outgoing interface of the PPFP. On the way back the destination IP/port will be corrected and the packet will be send to the client (Figure A.1).

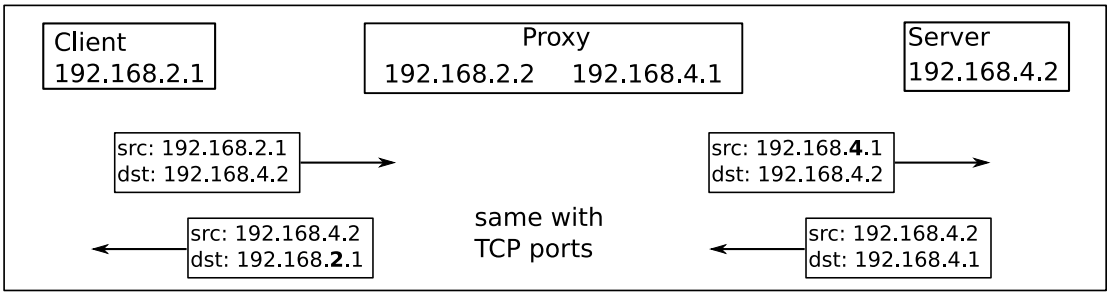

Figure A.1: Exchanging IPs

A.3.1 Initialization

Load the modules:

```
# modprobe promethos_PROXY // if not already loaded
# modprobe module_tcp2
```

Configure the proxy instance:
(Assuming IPs as above in graphic)

```
//for the packets client -> server
# iptables -t promethos -A FORWARD -d 192.168.4.2 -p tcp \
-j PROMETHOS --plugin PROXY3 --autoinstance --config 'cs 2 tcp2.'
//for the packets server -> client
# iptables -t promethos -A PREROUTING -s 192.168.4.2 -p tcp \
 -j PROMETHOS --plugin PROXY3 --autoinstance \
--config 'sc <instance-nr> tcp2.'
```

(for <instance-nr> see section above)
This configuration assumes that the Client will initiate the connection. If the server will do it with this configuration the kernel will crash. The reason for this is the function `inet_select_addr` used in this module. The function can only be used for packets which have already been traversed the PREROUTING hook and passed the routing algorithm of the kernel (this is not the case for the above configuration for packets from server to client!).

A.3.2 Validation for Working

If every thing was correctly configured it is now possible to transfer data over the PPFP including protocol interception. To check if every thing works fine check the ID field in the IP header. This should be '0x0000' after the packet has been processed by the TCP state machine.

Appendix B

Components of PromethOS

The PromethOS framework consists of two kernel modules, a shared library for the user space tool iptables and the PromethOS plugins (which are also kernel modules).
After the PromethOS kernel modules are loaded and the Framework is configured via the iptables user space tool PromethOS delivers the corresponding network packets to the desired plugin.

B.1 Kernel Space Parts of PromethOS

B.1.1 iptable_promethos

The module defines the netfilter hooks Promethos wants to use. Because the requirements of the PromethOS plugins are not known at this point the module registers at all IPv4 hooks of netfilter.
First a new table `promethos` is defined with a struct of type `ipt_table`. This table is registered with `ipt_register_table`.
Second a array of struct type `nf_hook_ops` is build which is be used to register the PromethOS framework on all five netfilter hooks of IPv4. The registration is done with the function `nf_register_hook`.
The registration of table and hooks is done when the module is loaded (`modprobe iptable_promethos`). Furthermore there a functions defined which deregisteres the table and the hooks when the module is removed (`rmmod iptable_promethos`).

B.1.2 ipt_PROMETHOS

This is the main part of PromethOS. Besides defining the target function for netfilter/iptables and providing a registration function for the PromethOS plugins this part also implements a structure for the administration of all PromethOS plugins and their instances. Furthermore it provides in interface for reconfiguration of PromethOS plugins.
When the PromethOS target function is executed by netfilter its task is to determine which PromethOS plugin was configured to handle the specific network packet. After the PromethOS framework has figured out the appropriate plugin it calls the target function of that plugin.
Because one plugin can be used in several instances with may be different configurations PromethOS assigns a unique instance number to every instance of a plugin making it possible for PromethOS plugins to distinguish between different instances of a plugin.

B.1.3 promethos_<PLUGIN_NAME>

Every PromethOS plugin must at least define the following three functions:

- `target function`
  This function is called by the PromethOS framework when a packet has arrived which is intended for that specific plugin. When a target function of a plugin is called the PromethOS

framework also handles a pointer of type `sk_buff` to the plugin so that the plugin access the network packet.

- `config function`
  The config function is called every time a new instance of the plugin is created. For example when one adds a second netfilter rule with the iptables tool and both rules corresponds to the same plugin two instances of the plugin is be generated by the PromethOS framework. So the config function is called two times, once for every instance of the plugin.

- `reconfig function`
  The reconfig function can be used to reconfigure a instance of a plugin by writing the configuration options into `/proc/promethos/net/management`. The plugin instance is referenced via the unique PromethOS i

There is also a plugin configuration function which is called once when the plugin is loaded and which can be used for plugin wide configurations/initializations.
Each plugin registers at the PromethOS framework by calling the `promethos_init function`. With this function the plugin handles function pointers of the above mentioned functions to the framework enabling PromethOS to call the functions of the plugin later on when necessary.

B.2 User Space Parts of PromethOS

Only one PromethOS part is necessary in the user space to add the functionality provided by PromethOS to the iptables user interface. This is done by a shared library which is called libipt_PROMETHOS for IPv4.

B.2.1 libipt_PROMETHOS

In this library first a struct of type `iptables_target` is configured with informations about the PromethOS target. By including two header files of the PromethOS framework the functions needed by iptables can handled over to it with function pointers in the struct. The second part consists of an init section which is called when iptables load the shared library and which registers the PromethOS target to iptables by returning the previously mentioned struct (`with iptables_register`).
Provided with the information out of the struct iptables can now parse PromethOS specific configuration information to the PromethOS framework for the purpose of sanity checks and configuring of the PromethOS plugins. Furthermore the PromethOS framework loads further PromethOS plugins if necessary.

Appendix C

Development Environment

Just for the sake of completeness a short description the development setup for the proxy. Because the PromethOS plugins are kernel modules a virtual Linux system was setup, booted via NFS and the PromethOS plugins were tested within this virtual system. So when the virtual system crashed no reboot of the computer was neccessary.

C.1 Host Machine

Gentoo linux with a 2.6 version kernel was installed, also a graphical system and VMware server and VMware client for the virtual system. Besides the normal network stuff support for NFS, NAT and the tun/tap interface were compiled into the kernel. Because VMware Server was still in beta phase and hence not in the portage tree, portage overlay was used to install VMware Server out of some packages created by Mike Auty [1] (this was gentoo specific).
With the tun/tap driver of the kernel a virtual network interface `tap0` was created and a IP and netmask assigned to it. Then `dhcpcd` and `tftp-hpa` were installed and configured to provide the virtual system with the necessary information and data for network booting over the `tap0` interface. NFS was configured to export the necessary directories over the `tap0` interface. NAT was activated for the `tap0` interface to give virtual system access to the outer network.

C.2 Client Machine

Because the whole virtual system was booted via NFS nothing had to be installed inside the VMWare. A very simple VMware configuration was created just with memory, processor and two network interfaces. The BIOS was set to boot via network. On startup the virtual system got its very first information from the dhcp server telling the virtual system the location of `pxelinux.0` (some bootloader). The virtual system then fetched this file with `tftp` and executed it. `Pxelinux.0` now loaded the kernel image with the configured append line (telling the kernel to use dhcp for configuring its IP and to mount its root directory via NFS.
The virtual system was set up as a very small gentoo linux system created out of a stage3-gentoo-package (this is again gentoo specific). The installation was done by first downloading all necessary data into a directory, unpacking it, mounting /proc and /dev onto the directories of the virtual system and then chrooting to the directory for further configuration of the system. Also later when installing additional software the chrooting method was very nice (compared with using the same system booted in the VMware environment where the whole filesystem was accessed via NFS which would result in a much lower performance.
The kernel was configured to support network booting and module load/unload support. After compiling the kernel and copied to /boot (of the virtual system) the last thing was to edit the configuration file of pxelinux (position of kernel image and append options).

[1] Gentoo Bugzilla Bug #122500
http://bugs.gentoo.org/show_bug.cgi?id=122500

C.3 PromethOS

After some small adjustments to the makefiles compiling and installing the PromethOS kernel modules and the shared library for iptables was just a matter of calling two makefiles provided together with the PromethOS framework by Lukas Ruf.

Appendix D

Aufgabenstellung

D.1 Aufgaben: Proxies in PromethOS NP

Während das weitverbreitete NAT eine Uebersetzung der TCP/IP Adressen vornimmt, ist Ziel dieser Semesterarbeit, ein modulares Proxy-Framework für PromethOS NP zu realisieren, das prinzipiell für beliebige Protokolle ab Layer 2 des ISO/OSI-Modells eingesetzt werden kann. Basierend auf dieser Grundlage soll es mit entsprechend geringem Aufwand möglich sein, neue Proxies zu erstellen.
Verschiedene, protokoll-spezifische Proxies sollen die Funktionsweise des zu entwickelnden Proxy- Frameworks demonstrieren. Als geeignete Kandidaten, werden neben anderen Proxies für HTTP, HTTP over TLS[20, 21] in Betracht gezogen.

D.2 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum bestehenden PromethOS NP Framework.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Analysieren Sie verschiedene Protokolle und definieren Sie Funktionsmodule, die in verschiedenen Protokollen benötigt werden.
- Entwickeln Sie eine modulare PromethOS NP Service-Architektur für das zu entwickelnde Proxy- Framework.
- Implementieren Sie die Komponenten Ihrer Service-Architektur.
- Implementieren Sie die protokoll-spezifischen Proxy-Komponenten.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte durch geeignete Beispielapplikationen.
- Dokumentieren Sie die Resultate ausf¨hrlich.
- optional Entwickeln Sie geeignete Komponenten für Paketprozessoren.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

Bibliography

[1] Rusty Russel, Harald Welte: Linux netfilter Hacking HOWTO, 2002, Revision: 1.14 http://www.netfilter.org/documentation/

[2] Fabrice Marie: Netfilter Extensions HOWTO, 2002, Revision: 1.31 http://www.netfilter.org/documentation/

[3] Victor Castro: Roll Your Own Firewall with Netfilter, 2003 http://www.linuxjournal.com/article/7184

[4] Wehrle, Paehlke, Ritter, Mueller, Bechler: Linux Netzwerkarchitektur, 2002, ISBN 3-8273-1509-3,Addison-Wesley http://www.addison-wesley.de

[5] Rusty Russell: Linux 2.4 Packet Filtering HOWTO, 2002, Revision 1.26 http://www.netfilter.org/documentation/

[6] Amir Guindehi: COBRA Component Based Routing Architecture, 2001, Studienarbeit SA-2001.30 TIK ETH Zurich

[7] Peter Graubmann, Cornel Klein, Leopold Mandl: Basic Modules for PromethOS Plugins, 2002 http://www.promethos.org/Publications/cgiDownload.cgi?filename=Promethos_Support_Modules_Siemens_-_V1.0.pdf

[8] Robert Love: Linux-Kernel-Handbuch, 2005, ISBN 3-8273-2204-9, Addison-Wesley http://www.addison-wesley.de

[9] T. Berners-Lee and other: Hypertext Transfer Protocol – HTTP/1.0, 1996, RFC 1945

[10] R. Fielding and other: Hypertext Transfer Protocol – HTTP/1.1, 1999, RFC 2616

[11] Larry L. Peterson, Bruce s. Dave: Computer Networks, 2000 (second edition), ISBN 1-55860-577-0, Morgan Kaufman Publishers

[12] William Stalling: Network Security Essential, 2003, ISBN 0-13-035128-8, Prentice Hall

[13] Charlie Kaufman, Radia Perlman, Mike Speciner: Network Security, 2002, ISBN 0-13-046019-2, Prentice Hall PTR

[14] Eric Rescorla: SSL and TLS, 2001, ISBN 0-201-61598-3, Addison-Wesley

[15] By University of Southern California, 1981, RFC 791

[16] S. Deering, R. Hinden: Internet Protocol, Version 6 (IPv6), 1998, RFC 2460

[17] By University of Southern California: Transmission Control Protocol, 1981, RFC 793

[18] J. Postel: User Datagram Protocol, 1980, RFC 768

[19] Lars Haendel: The Function Pointer Tutorials, 2005, http://www.newty.de

[20] T. Dierks and Allen C. The TLS Protocol Version 1.0. RFC 2246, Network Working Group, Jan. 1999.

[21] E. Rescorla. HTTP over TLS. RFC 2818, Network Working Group, May 2000.