Project Number :	IST-2002-002154
Project Title :	Distributed Adaptive Security by Programmable Firewall

DIADEM Firewall

D11 – Integrated Prototype (revised version)

Deliverable Type :	Document
Dissemination:	Public
Contractual date :	January 2006

Editor:	Yannick Carlinet (FT)
File Name:	Diadem Firewall – D11 – Integrated Prototype - revised.doc
Contributors :	See list of authors
Version :	Revised version
Version Date :	February 2006
Deliverable Status:	

The DIADEM Firewall consists of:

	Partner	Short name	Country
1	France Telecom	FT	France
2	University of Tübingen	TU	Germany
3	IBM Research GmbH Zurich Research Laboratory	IBM ZRL	Switzerland
4	Imperial College London	Imperial	United Kingdom
5	Jozef Stefan Institute	JSI	Slovenia
6	Groupe des Ecoles des Télécommunications	GET	France
7	Polish Telecom	TP	Poland

Project Management:
Yannick Carlinet (FT)
Phone +33 2.96.05.03.25
Fax: +33 2 96 05 37 84
E-mail yannick.carlinet@francetelecom.com
France Telecom CORE/M2I
2 ave. Pierre Marzin,
22307 Lannion, France

List of authors:
Olivier Paul, GET
Sherif Yusuf, Imperial
Vryzlinn Thing, Imperial
Morris Sloman, Imperial
Dušan Gabrijelčič, JSI
Gerhard Muenz, TU
Ali Fessi, TU
Raimondas Sasnauskas, TU
Gero Dittmann, IBM ZRL
Piotr Piotrowski, TP
Paweł Tobiś, TP
Yannick Carlinet, FT

Executive summary

This document describes the integration work performed during period 4 (July-December 2005) of the project. For each high-level component, namely the Monitoring Element, the Violation Detection, the System Manager and the Firewall Element, the document describes how their sub-components work with each-other, and how they work with the other high-level components. The desciptions are given with an implementation viewpoint.

Acronyms

ACK	Acknowledge
ACL	Access Control List
API	Application Programming Interface
CE	Classification Engine
DDoS	Distributed Denial of Service
DMZ	Demilitarized Zone
DNS	Domain Name Service
DoS	Denial of Service
FD	Firewall Device
FE	Firewall Element
FM	Firewall Module
HTTP	Hyper Text Transfer Protocol
IPFIX	Internet Protocol Flow Information eXport
ISP	Internet Service Provider
ME	Monitoring Element
OSI	Open Systems Interconnection
RMI	Remote Method Invocation
PMA	Policy Management Agent
Seq	Sequence
SM	System Manager
SMTP	Simple Mail Transport Protocol
SSH2	Secure Shell Protocol version 2
SYN	Synchronize
TCP	Transmission Control Protocol
TFN	Tribe Flood Network
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VD	Violation Detection
VDF	Violation Detection Facility
WL	White List

Table of Contents

1 Introduction

The aim of this document is to describe all the integration issues that had to be solved in order to provide an integrated working prototype of the DIADEM Firewall system. For each high-level functional component, a list of the integration issues is given, and for each of these, a detailed description of how the issue was solved is given, including the engineering choices and the implementation choices. This revised version includes the description of the latest integration developments, made after the project plenary meeting in January 2006.

2 Monitoring Element

This section provides information about the integration of different components within the ME as well as about the integration of the ME into the whole DIADEM Firewall architecture. As described in D9 [5], two different MEs have been developed: the IPFIX/PSAMP monitoring probe VERMONT and the specialized ME for HTTP micro-session discovery called WSMon. In subsection 2.1, we present the implementation and integration progress that has been accomplished since the writing of D9. Subsection 2.2 describes the external communication with the VD.

2.1 Internal components

2.1.1 Integration of the HTTP Micro-session Discovery Function

The monitoring function was extracted from the RequIn architecture in order to work as a standalone tool called WSMon (for Web Server Monitor). As explained in [5], the HTTP aware monitor is built as an extension to IPfilter which gathers information about HTTP sessions. Which HTTP sessions have to be monitored is controlled through the IPfilter module configuration. As soon as HTTP sessions are terminated, monitoring information is exported to a new user space program through the kernel log device. This program is different from RequIn as it does not implement any of the parameter estimation, inference and detection functions but rather focuses on parameters extraction from kernel logs. Of particular interest was the decision whether we would associate parameter estimation functions with this software or whether we would keep parameter estimation functions as a part of the detection module. Finally we decided to follow the second option for the following reasons:

- Estimated parameters can carry less information than raw parameters. This additional information is necessary for some other functions. For example IP addresses are converted into country code before being used by the inference engine; however the IP address is also used by the notification module in order to identify attackers. Exporting both parameters would increase the size of exported records unnecessarily.
- Raw parameters are easier to fit to standardized IPFIX information elements. As a result exporting raw parameters allows us to reuse IPFIX standard information elements (see section 2.2.1) and limits the use of proprietary elements. This increases the openness of our tool which can provide information to other IPFIX compliant tools.
- Estimation functions can require configuration information. In order to keep management of the architecture as simple as possible and avoid duplication of configuration information it seems wiser to limit the configuration of our monitor as much as possible.

The bundle (IPFilter extension, user space program) is now available as a standalone tool.

2.1.2 Integration of the Configuration Component (Monitoring API)

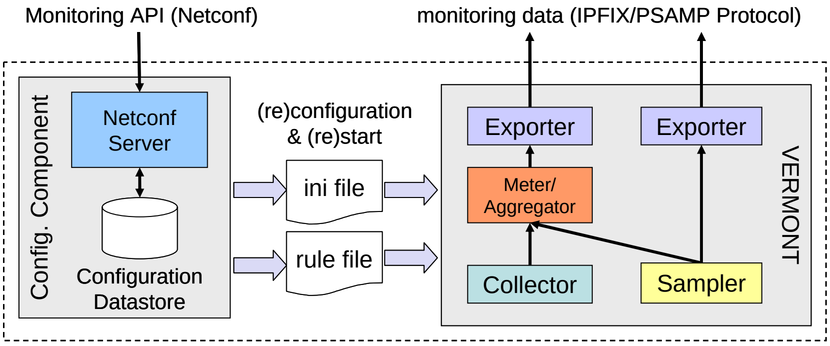

Figure 1: Configuration Component and VERMONT

The configuration component is responsible for controlling and configuring the ME. It is currently used to provide our IPFIX/PSAMP monitoring probe VERMONT with the correct configuration parameters.
As shown in Figure 1, the configuration component consists of a Netconf server that implements the Monitoring API based on the Netconf protocol. The Netconf server runs as a separate process that receives RPCs from one or more Netconf clients. The configuration datastore contains the different configuration settings. The running and candidate configurations are stored on the heap memory, whereas the startup configuration is saved to an XML file since it has to survive system crashes.
The monitoring probe VERMONT runs in a child process that is controlled and configured by the configuration component. Before starting VERMONT, the configuration component has to translate the configuration settings stored in the configuration datastore into valid VERMONT configuration files. Doing this, it also verifies the conformity of the configuration settings to VERMONT's capabilities.
The current version of VERMONT requires two configuration files, the ini-file with the configuration parameters, and a rule file containing the flow aggregation rules. After creating the two files, the configuration component terminates the currently running VERMONT process and starts a new one with the new configuration. If VERMONT fails to start-up with the new configuration, the configuration component performs a rollback to the previous configuration. The configuration component also watches over unexpected crashes of the VERMONT process. In this case, it tries to restart VERMONT with the last properly running configuration.
Even though a reconfiguration currently requires a restart of VERMONT, it takes less than a second until VERMONT is back running with the new configuration. Nevertheless, monitoring is disabled during a short period of time. On the other hand, running VERMONT as a child process of the configuration component increases robustness since an unlikely but not impossible crash of VERMONT would not result in a break-down of the entire ME.

2.2 Interface with Violation Detection

2.2.1 IPFIX Exporter

The transport of monitoring data from the ME to the VD is based on the IPFIX protocol [6]. Both ME implementations, WSMon and VERMONT, make use of the same IPFIX exporter library. This library provides a C API with the following functionalities:

- Specifying one or several collectors as recipients of the exported IPFIX records as well as communication characteristics (IP address, transport protocol, port number).
- Defining templates for the exported records using standard IPFIX Information Elements [7] and/or proprietary Information Elements.

- Building IPFIX records using previously defined templates. In order to limit network load, WSMon takes advantage of the ability provided by the library to concatenate records before sending them to the collector.
- Sending the records packaged into IPFIX packets to the collector(s).

The monitoring data exported by VERMONT contains standard IPFIX Information Elements only. On the other hand, WSMon exports additional HTTP micro-session specific information for which we had to specify proprietary Information Elements for the number of bytes exchanged within a HTTP micro-session and the micro-session identifier. The mapping between the original RequIn parameters and standardized and proprietary IPFIX Information Elements, as used by WSMon, are shown in Table 1.

Table 1. Mapping between RequIn and IPFIX parameters.

Original RequIn parameter	Mapped IPFIX parameter
Flow Start Timestamp (seconds)	flowStartSeconds
Flow Start Timestamp (micro seconds)	flowStartMicroSeconds
Flow End Timestamp (seconds)	flowEndSeconds
Flow End Timestamp (micro seconds)	flowEndMicroSeconds
Flow identifier	flowId
Source IP address	sourceIPv4Address
Destination IP address	destinationIPv4Address
Source TCP Port	sourceTransportPort
Destination TCP Port	destinationTransportPort
Flow termination cause	flowEndReason
Flow total number of bytes	octetTotalCount
Flow total number of packets	packetTotalCount
Flow Source to Destination TCP bytes	Proprietary
Flow Destination to Source TCP bytes	Proprietary
Flow session number	Proprietary

2.2.2 Monitoring API

As described in [5], the Monitoring API builds on a Netconf-over-SOAP implementation that we realized within the DIADEM project. This implementation implements many optional capabilities [8] of the Netconf protocol like candidate configuration, rollback on error, and distinct startup. We currently use a simple Netconf client program that offers shell for controlling and configuring the ME remotely. Configurations have to be given as XML documents that comply with the XML Schema definitions as shown in the annex of [5].

Even though the Monitoring API is ready for use, the VD is currently missing a module that makes use of it in order to dynamically adapt the ME configuration to the needs of the detection process. We intend to add this functionality to the VD later after having evaluated the detection performance of the VD without dynamic reconfiguration of the ME.

3 Violation Detection

The VD consists of multiple modules that run as different processes and communicate with each other using an event notification system. The progress of the implementation and integration of the modules within the VD is presented in subsection 3.1.

Externally, the VD communicates with the ME and the SM. Subsection 3.2 covers the communication with the SM while the interfaces to the ME have already been described in subsection 2.2

3.1 Internal components

3.1.1 Inter-Module Communication

As presented in [5], we use the open-source xmlBlaster MOM [9] for communication between the different modules within the VD. Furthermore, we decided to implement the Notify API upon xmlBlaster, too (see section 3.2.1).

Both the VD internal communication and the Notify API make use of IDMEF. The only difference is that IDMEF events exchanged between VD internal modules may contain additional information, such as state information, that needs to be exchanged in order coordinate the interworking between detection modules, investigation modules, and the PMA. The IDMEF messages sent to the SM do not include such internal state information but carry the detection results only.

We use different event *topics* in order to differentiate between internal and external events and to classify different types of detection and investigation results.

3.1.2 Integration of the Web Server Overload Detection Module

The web server module operation is controlled by the VD system. To the VD system the module appears as a separate process. Violation detection modules have a particular structure allowing detection operations to be controlled by the VD. To this mean generic detection module objects structures are provided for detection module developers. These structures carry three generic detection-related functions:

- Module configuration.
- Monitoring information collection.
- Detection operation scheduling.

The module configuration functions were extended with existing RequIn configuration functions. In order to provide the ability to configure the detection module dynamically at startup, configuration parameters are passed to VD process when executed.

As indicated in [1], the Web Server Overload Detection Module includes two main communication interfaces: an interface between the monitor and the detection module to obtain monitoring information and another interface between the notification engine and the user to provide the user with detection results. When integrating them into the VD, these two main interfaces have to be adapted to fit the interfaces provided by the detection system.

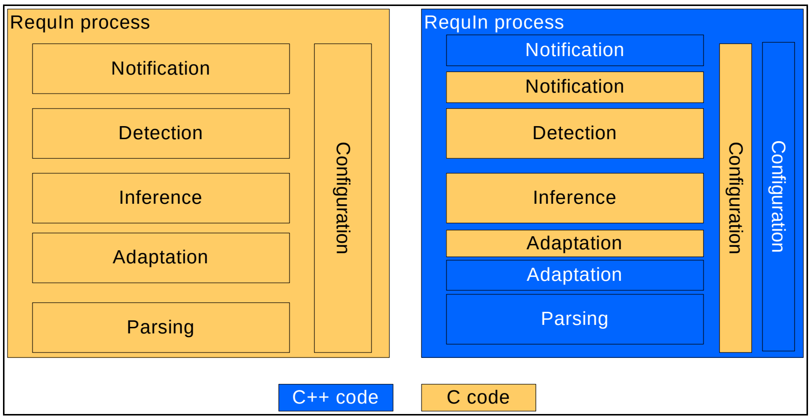

Figure 2. Integration work result on code nature (left: original RequIn code, right: Integrated RequIn detection module)

Integration between IPFIX collector and Web Server Overload Module
The generic detection module class provides methods allowing the detection module to subscribe to IPFIX messages with the IPFIX collector as well as to select information elements of interest. When receiving these elements generic methods are called so that the content of each element can be stored. Another generic function is called to signal when all the elements related to a particular message are received. The generic object also provides the ability to perform operations on a periodical basis. This ability was however not used in our case, detection operations being executed every time an IPFIX message is received in order to speedup attacks detection.
In order to use the original RequIn records parsing functions within a C++ detection module class, we ported them to C++ methods as shown on Figure 2 (Note that most C++ code originates from the generic detection module class). Other existing C functions including inference and detection functions are called as external methods.

Integration between the Web Server Overload Module and the PMA
The generic detection module object provides methods allowing the detection module to build, fill and send IDMEF events to the PMA. These methods permit a transparent communication management between the detection module and the PMA for the module developer. The address of the PMA is provided to the module at startup. When an attack is detected, a notification function is called with the detection results as parameters. These parameters are coded as IDMEF information elements. The notification is then sent to the PMA.

From an implementation standpoint, the main issue met during the integration was the communication between the C and C++ code. Our detection functions are implemented in C. However notification functions are implemented as C++ objects methods. Since C code cannot access C++ objects methods directly, we wrote a C wrapper function that calls the notification methods. This integration is shown in Figure 2.

3.1.3 Integration of the SYN Flood Detection Module

Initially, the implementation of SYN flood detection algorithm was an individual program which simultaneously performed traffic monitoring and attack detection by using a decision function. Then the application was ported to VD framework as a detection module, using the dedicated base classes.

The SYN flood detection module subscribes with the IPFIX collector and receives the monitoring data in IPFIX format. These data are then converted into an internal format for use with the detection functions.

The module can subscribe to the monitoring data in two different modes:

- The first mode subscribes to statistics of flow directed to chosen protected objects according to configuration read from text file. A protected object can be defined by a single IP address, an IP address with a netmask, or an IP address with TCP port number.
- In the second mode, the module subscribes to all statistics delivered by the IPFIX collector based on the assumption that the monitor or the collector already performed suitable filtering and aggregation in order to provide only information about the objects of interest.

The maximum number of protected objects is initially limited for both modes. The detection function is launched in specified time intervals and performs computation separately for each object. IDMEF are sent to the PMA when an attack is detected or optionally if an attack has finished.

3.1.4 Integration of the Traceback Module

The Traceback Module will be implemented as one of the Violation Detection modules. It will subscribe to the IPFIX collector to receive the monitoring data it requires for training its database during the learning phase. The data elements it will subscribe to include the source IP address, destination IP address and the next hop IP address of the original data packets (being sampled). The Monitoring Element's ID will also be extracted from the header of the IPFIX message for use as one of the parameters required to train the learning database. The learning process will be on-going throughout the module's lifetime till it is informed of the presence of an attack, which will be detected by another Violation Detection Module. In our case, the Traceback Module will subscribe to the xmlBlaster server to receive notification of the detection of a SYN Flood attack. Therefore, the tracing phase will be activated at this point. The sampled data from the IPFIX collector will then be used with the learning database's records to discover the Monitoring Elements carrying attack traffic. When the tracing of the Monitoring Elements carrying the attack traffic is completed, an IDMEF message will be generated with the ID of the Monitoring Elements. This message will then be sent to the PMA.

3.1.5 Integration of the Policy Management Agent

Just like any other module of the VD, the PMA connects to the xmlBlaster server to send and receive IDMEF events and messages. Since both the VD internal communication and the Notify API use xmlBlaster and IDMEF, the PMA is able to process events independently of if they were originated by a VD module, the SM, or another distant VD instance. Of course, the treatment of VD internal events differs from that of external events by providing different policies.

The integration of the PMA was delayed by the recent change to a new version. As event system, the new PMA uses xmlBlaster instead of Elvin [10], and it offers a very powerful environment for developing adapted policy-based control functions very different kinds of system components.

Currently, the PMA is able to send and receive IDMEF events to and from other modules within the VD. We are now developing specific functions and policies that are necessary to deploy the PMA in the context of the VD. In a first step, we are focusing on functions and policies that allow controlling the relaying of IDMEF messages to the SM. The goal is to inform the SM about new attacks as soon as possible, but to protect him from being overwhelmed by to many alerts concerning the same issue. In a second step, we will add more sophisticated policy rules and actions, such as triggering the correlation and aggregation of equal or similar IDMEF events or the investigation of suspicious observations that have not been classified as an attack yet.

3.1.6 Aggregation Module

The original RequIn code integrates aggregation functions in order to provide users with synthetic attack information. Aggregation functions aggregate alerts sharing common characteristics, such as similar targets, detection times in a close time range or attacks based on similar requests. These functions are however proprietary since they only apply to the web server overloading attack scenario. Since other attack detection module might take advantage of similar functions in order to present the system manager with more meaningful and synthetic information, the aggregation functions were rewritten as a separate module. This module receives notifications from the PMA and aggregates them according to a user defined aggregation policy indicating:

- Which notification elements have to be compared.
- How these elements have to be compared.
- How to aggregate them when a match is found.
- When to emit aggregated notifications back to the PMA.

Aggregated notifications are sent back to the PMA that can take further action depending on aggregation results. The new aggregation module is currently under test.

3.2 Interface with System Manager

3.2.1 Notify API

As specified in [11], the message format of the Notify API is based on IDMEF [12]. Message exchange between the VD and the SM is based on an event system. In contrast to the original plan to use the commercial Elvin [10] event notification system, we decided to change to the open-source xmlBlaster message-oriented middleware (MOM) [9]. The main advantage of xmlBlaster over Elvin is that xmlBlaster uses XML-encoded events. If the exchanged payload is also based on XML, as in case of IDMEF, XPath expressions [13] can be used to refine the subscription of events based on their content.
As xmlBlaster is already deployed for inter-module communication within the VD, the implementation of the Notify API does not require any new software components. We secure the Notify API by activating xmlBlaster support for authentication and encryption based on SSL.

3.2.2 Service API

The Service API is used to load and enable new detection and notification policies on the VD, as well as to disable and unload them afterwards again. As described in [14], the detection policies control the exchange of IDMEF events between the different modules within the VD. On the other hand, notification policies determine which IDMEF events trigger the sending of an alert to the SM.
The Service API is exported by the PMA of the VD. It is to be called by the SM via Java RMI (Remote Method Invocation). However, current implementation of the PMA only allows uploading XML-encoded policies via a direct connection to a specific port (see also subsection5.1.1).

4 System Manager

4.1 Internal components

4.1.1 Policy Management Agent

We have implemented a new Policy Management Agent (PMA), which will be made available in the public domain for use by a number of different projects. Policies are specified using an XML-based notation, although we intend to provide an easier to use notation similar to the original Ponder.
Policies are of the form:

```
on event (params)
do managedObject.action
when condition
```

The event could be an IDMEF notification from the VD which passes detection results as parameters used by the PMA, e.g. in the condition clause. The managed object could represent a FE and the action would be one of the operations defined in its API.

4.1.2 Example Policy

An example policy is as follows, if we show a typical response to Deliverable D2, Use Case 2 [1]. There are a few ways to deal with the event of a web service attack, and below is one step we might want to take.
The general format (Ponder syntax) for the policy is as follows:

```
type oblig webAttack(subject <PMA> s, target <FirewallElement> t) {
     on webAttack(<idmef> msg);
     do t.ratelimit(limited_rate);
     when msg.confidence = high;
}

inst oblig response = webAttack(/PolicyManager/SystemManager,
     /Firewall/FD4);
```

The current PMA uses XML-based notation, and the above policy would be written as follows:

```
<xml>
   <use name=”Policy”>
      <add name=”webAttack”>
         <use name=”/Template/policy”>
            <create type=”obligation” event=”/Event/webAttack”
active=”true”>
               <arg name=”msg”/>
               <condition>
                  <equal>!msg.confidence; <!--  --> “high” </equal>
                </condition>
               <action>
                  <use name=”/FirewallElement/FD4”>
                     <add>
                        <reconfigure rate=”!limited_rate”/>
                     </add>
                  </use>
               </action>
            </create>
         </use>
      </add>
   </use>
</xml>
```

When we receive an event of a web attack, with the confidence at high, then one possible response by the System Manager is to trigger rate limiting on the appropriate firewall device(s).

4.1.3 Integration with Notification Service

We decided not to use the Elvin Event Service, as indicated in D6 [3], as there were licensing problems for use by the industrial partners since Elvin is now a commercial product. In addition Elvin does not support subscription to XML encoded events, which is required for the IDMEF standard. We switched to using xmlBlaster, which is public domain software with source code readily available [9].

We have integrated the PMA with xmlBlaster so that it subscribes to receive events from the xmlBlaster server. In addition the managedObject.action could be an action to publish (e.g. an IDMEF notification) to the server.

4.2 Interface with Firewall Element

The FE provides a common API for implementing response actions on all types of firewall devices. The current state of implementation supports:

1. The manipulation of firewall rules - IPTables in Linux or the access lists in a Cisco PIX firewall – in order to, for example add, delete or otherwise modify rules.
2. The redirection of 'attack' traffic, for example to a specified IP address or a blackhole, using Linux. This element is still undergoing full functional testing with the Response Module.
3. The limiting of traffic rate using Linux. Similarly, this is still also undergoing full functional testing with the Response Module.

4.2.1 Integration with Traceback

The Traceback facility has been simulated using the NS2 simulator. The attacker, legitimate client, victim, router and learning agent NS2 modules were developed. Five scenarios of simulations were carried out (e.g. running with/without legitimate traffic during attack, extension of topology or/and number of clients) to proof the feasibility of the scheme. Work on implementing the learning engine on an actual testbed has just started. The learning engine will be implemented as a detection module within the Violation Detection System. However, Traceback responses to an alarm have not yet been integrated into the system manager.

4.2.2 Integration between PMA and Service Module

The Service Module will offer an RMI (Remote Method Invocation) interface for the SM to use for communication with the PMA within the FE. See section 5.1.3 for the list of the methods that can be invoked in the Service Module. Currently we have not implemented an interface that is required by the Service Module, for the PMA to allow the loading/removing of policies externally, nor the enabling/disabling of such policies. This will be provided by RMI calls for the required methods, and the Service Module will control the access to the methods.

4.2.3 Integration between PMA and Response Module

The Response Module defines specific actions that can be performed as the result of a notification being received by the PMA. When the PMA receives the notification, the SM examines it to ascertain which action to undertake, based on the appropriate policy. Once the action is determined, the necessary information (such as the source of the attack, its destination and the protocol being used) is filtered out from the notification, and then a response is decided upon and communicated to the Response Module or an event is generated by the SM to be received by the PMA in the FE, see Figure 5-1 in [2]. This then determines where the necessary action(s) needs to be performed, either locally or on a remote device.

To carry out a local response in our current implementation, the local device, which is a Linux machine, is accessed by attaching the machine as the Firewall Module then use the Firewall API to perform the required action(s). Whereas, for a remote response, the PMA uses the RMI calls provided by the Response Module to connect to and execute the appropriate action(s) on the remote device. The actions that have already been implemented in the Response Module are manipulation of firewall rules, rate limiting and redirection.

5 Firewall Element

The Firewall Element as introduced in the project deliverable D2 [1] and revised in deliverables D5 [2] and D6 [3] is an element in the DIADEM Firewall system that enables enforcement of the firewall related security policy and provides response mechanisms to tackle security threats. In this section, we describe the implementation progress related to particular Firewall Element sub systems and provide the status and issues in their integration.

5.1 Internal components

5.1.1 Integration between Policy Management Agent and Service Module

The service module provides the functionality of loading and removing policies from the PMA. Since the migration to the new version of the PMA, we have not yet provided an interface for the service API to use. The previous version of the PMA provided an RMI interface that is used to load/unload policies as well as enable/disable those policies.

Currently there is no formal API yet for the new version of the PMA, however there is a shell interface. A connection to port 13570 (or any other port the PMA shell is running on) can be opened and some XML sent down it. The XML that represents the policies would need to be encapsulated within <xml> and </xml> tags.

5.1.2 Firewall Module implementing the Firewall API

Initial implementation report on Firewall Module (FM) implementing the Firewall API was given in the deliverable D8 [4]. The implementation has been improved and extended in many ways, most notably in a complete rewriting of the Firewall Element (FE) and Devices (FD) capabilities, supported actions of redirection and rate limiting, support for issuing multiple commands on a FD as a result of one applied rule and initial implementation of the time policy. Integration progress with other sub systems will be described in subsections dedicated to this topic. Similar to the report in D8 [4], we present these achievements through the programming example shown in the Figure 3. The output of the test run is presented in the Appendix (section 8). The implementation size has grown from 3000 lines of source code as reported in D8 to more than 5000 lines of code.

```
public void testSSH2Usage() throws Exception {
  FirewallModule fm = FirewallModule.getFirewallModule();
  //    fm.throwable = true;
  Credential c = new Credential("root","dinjajutro");
  if (fm.attachDevice("193.138.1.100","ssh2",c) != 0) {
    _log.debug(fm.lastCommand.error);
  } else {
    // Collect some variables to compare them later
    FirewallDevice fd = fm.getCurrentDevice();

    // Create some groups
```

```
        if (fm.groupCreate("tcpstart") != 0)
                _log.debug(fm.lastCommand.error);
        if (fm.groupCreate("web_servers") != 0)
                _log.debug(fm.lastCommand.error);

        // Redirect all tcp traffic to port 80 to group web_servers
        int r1 = fm.ruleAppend("selector:\"intf=*;src=*;dst=*;proto=tcp;dstport=80\" action:\"redirect group web_servers\"");
        if (r1 < 0)
                _log.debug(fm.lastCommand.error);

        // Redirect all tcp traffic with state new to group
        // tcpstart. Order of rules matter, web traffic is redirected
        // prior to this rule.
        int r2 = fm.ruleAppend("selector:\"intf=*;src=*;dst=*;proto=tcp;state=new\" action:\"redirect group tcpstart\"");
        if (r2 < 0)
                _log.debug(fm.lastCommand.error);

        //Select tcpstart group and insert rules there
        if (fm.groupSelect("tcpstart") != 0)
                _log.debug(fm.lastCommand.error);

        //Insert rules
        int r3 = fm.ruleAppend("selector:\"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;srcport=*;dstport=139\" action:drop");
        if (r3 < 0)
                _log.debug(fm.lastCommand.error);
        // Specifically we don't like this host. All its TCP traffic is
        // redirected to the blackhole.
        int r31 = fm.ruleAppend("selector:\"intf=*;src=193.138.1.34;dst=*;proto=tcp;srcport=*;dstport=*\" action:\"redirect blackhole\"");
        if (r31 < 0)
                _log.debug(fm.lastCommand.error);
        // Traffic to the port 1111 is redirected to the blackhole
        // prohibited and the ICMP admin prohibited is returned.
        int r32 = fm.ruleAppend("selector:\"intf=*;proto=tcp;srcport=*;dstport=1111\" action:\"redirect prohibited\"");
        if (r32 < 0)
                _log.debug(fm.lastCommand.error);
        // A specific mail server in the net is not accessible from the
        // outside. Network prohibited error is returned.
        int r33 = fm.ruleAppend("selector:\"intf=*;src=!193.138.1.0/24;dst=*;proto=tcp;srcport=*;dstport=25\" action:\"redirect
netunreachable\"");
        if (r33 < 0)
                _log.debug(fm.lastCommand.error);
        int r4 = fm.ruleAppend("selector:\"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;srcport=*;dstport=22\" action:pass");
        if (r4 < 0)
                _log.debug(fm.lastCommand.error);
        // Again order matters. Previous rule selector traffic passes,
        // this gets redirected ... Redirect the new tcp traffic with
        // selector targeting ssh protocol to the host with tcpsynflood
        // reaction function for example
        int r5 = fm.ruleAppend("selector:\"intf=*;proto=tcp;dstport=22\" action:\"redirect address 193.138.1.100\"");
        if (r5 < 0)
                _log.debug(fm.lastCommand.error);
        // Redirect also the UDP traffic, add another reference to the
        // 193.138.1.100 routing table
        int r51 = fm.ruleAppend("selector:\"intf=*;proto=udp;dstport=22\" action:\"redirect address 193.138.1.100\"");
        if (r51 < 0)
                _log.debug(fm.lastCommand.error);
        // Create another routing table, 22
        int r52 = fm.ruleAppend("selector:\"intf=*;proto=tcp;dstport=20:21\" action:\"redirect address 193.138.1.101\"");
        if (r52 < 0)
                _log.debug(fm.lastCommand.error);

        // ICMP traffic is redirected to the user space
        int r7 = fm.ruleAppend("selector:\"intf=*;src=*;dst=193.138.1.100;proto=icmp\" action:\"redirect queue\"");
        if (r7 < 0)
                _log.debug(fm.lastCommand.error);


        //now try to put the some rules to web_servers group
        if (fm.groupSelect("/193.138.1.100/web_servers") != 0)
                _log.debug(fm.lastCommand.error);
        // Prevent acces to the web server from certain network
        int r8 = fm.ruleAppend("selector:\"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;state=new\" action:drop");
        if (r8 < 0)
                _log.debug(fm.lastCommand.error);
        // Allow access from some other network
        int r9 = fm.ruleAppend("selector:\"intf=*;src=193.138.2.0/24;dst=193.138.1.2\" action:pass");
        if (r9 < 0)
                _log.debug(fm.lastCommand.error);
        // Redirect some of the web traffic towards selected server to
        // the second routing table.
        int r91 = fm.ruleAppend("selector:\"intf=*;dst=193.138.1.80\" action:\"redirect address 193.138.1.101\"");
        if (r91 < 0)
                 _log.debug(fm.lastCommand.error);

        // Ratelimit, all web trafic redirected to web_servers before
        // other syn traffic is redirected to tcpstart
```

```
                  int r92 =               fm.ruleAppend("selector:\"intf=*;proto=tcp;dstport=80;state=new\" action:\"ratelimit 10kbps\"");
                  if (r92 < 0)
                          _log.debug(fm.lastCommand.error);
100
                  // for logging purposes
                  _log.debug("+++ List iptables:");
                  _log.debug(fd.session.protocol.sendCommand("/sbin/iptables -L"));
                  _log.debug("");
105               _log.debug("+++ List mangle table where the redirect rule is inserted:");
                  _log.debug(fd.session.protocol.sendCommand("/sbin/iptables -L -t mangle"));
                  _log.debug("");
                  _log.debug("+++ List ip rules and fwmark:");
                  _log.debug(fd.session.protocol.sendCommand("/sbin/ip rule list"));
110               _log.debug("");
                  _log.debug("+++ List tc qdisc:");
                  _log.debug(fd.session.protocol.sendCommand("/sbin/tc qdisc show"));

                  // Delete some rules, we are still in web_server group
115               if (fm.ruleDelete(r8) != 0)
                           _log.debug(fm.lastCommand.error);
                  // Relative path
                  //      if (fm.ruleDelete("../tcpstart/" + r5) != 0)
                  //          _log.debug(fm.lastCommand.error);
120               // Absolute path
                  if (fm.ruleDelete("/193.138.1.100/tcpstart/" + r4) != 0)
                           _log.debug(fm.lastCommand.error);

                  // Flush the groups ...
125               int z = fm.groupFlush("/193.138.1.100/tcpstart");
                  if (z != 0)
                           _log.debug(fm.lastCommand.error);
                  if (fm.groupFlush("/193.138.1.100/web_servers") != 0)
                           _log.debug(fm.lastCommand.error);
130               if (fm.groupFlush("/193.138.1.100") != 0)
                           _log.debug(fm.lastCommand.error);

                  // Detach the device
                  if (fm.detachDevice("193.138.1.100") != 0)
135                        _log.debug(fm.lastCommand.error);
                 }
               }
```

Figure 3: Firewall API test case

The Firewall API test case as presented in Figure 3 is one method of the Junit[1] tests testing the functionality of the Firewall API. The Firewall Device in this test case is Linux as an Open Firewall Device accessed by the Firewall Module via the SSH2 protocol. The device is attached, line 5, and detached, line 134, in the manner as we have described in D8 [4].

The main difference in between both procedures is the new capabilities implementation. They are defined in a programmable way as stand alone classes. Programmable implementation provides a way to detect, test, initialize and finalize the capabilities. In similar way as we have described in D8 [4], this implementation uses dynamical loading of classes. This feature provides separation between abstract definition of capabilities and device dependent implementation and tests. The capabilities can be easily added and new devices seamlessly supported. Till now we have defined 33 FD capabilities related to selector as defined in D2 [1] and D6 [3], actions, routing and rate limiting. In the test case output as presented in Appendix section 8.1, we can observe in lines 13 to 22 the determination of the device capabilities and their initialization. In this case the FD didn't have a module environment and could not match strings in the packet. The lines 14 to 19 show the initialization of the capabilities related to redirection. The routing tables needed for blackhole, sinkhole, etc. routing are initialized with predefined values. When the device is detached, these routing tables are deleted and capabilities finalized as can be seen in same output, lines 546 to 551.

The redirection as specified in D6 [3] enables redirection to various targets. We have extended the implementation of this functionality according to the following list:

- to user space via queue argument,
  <selector> redirect queue

[1] See http://www.junit.org for details.

- to certain internet address,
  <selector> redirect address IP_ADDRESS
- to certain group in firewall element,
  <selector> redirect group GROUP
- to the blackhole,
  <selector> redirect blackhole
- to the sinkhole,
  <selector> redirect sinkhole
- to the prohibited,
  <selector> redirect prohibited
- to the unreachable,
  <selector> redirect unreachable
- to the netunreachable,
  <selector> redirect netunreachable

While some targets are obvious, prohibited means a blackhole that returns an ICMP admin prohibited message, unreachable is a blackhole that returns an ICMP unreachable message and the netunreachable returns an ICMP net unreachable message. In contrast, the blackhole and sinkhole are silent.

The Linux implementation supports all specified targets. It uses iptables for marking the selected packets and the 'ip' command of the iproute2[2] package to achieve the desired functionality. The sinkhole target is the same as blackhole or a specific IP address. Examples of the target implementation, which specify the part of the command related to the redirect action:

- queue,
  <selector> redirect QUEUE is defined as:
  -j QUEUE
  and can be removed as
  -j QUEUE
- IP address is implemented with marking in iptables and rerouting via Linux routing tables. Since iproute2 supports only up to 255 routing tables, this is a hard limit for the number of available destinations. Some marks in this example text use the default values (mark 12 and table 12), while the other are unique, random generated mark values in the scope of the single FD.
  <selector> redirect address IP_ADDRESS is defined as:
  --table mangle -j MARK --set-mark 12; /sbin/ip rule add fwmark 12 table 12; /sbin/ip route add default via 193.138.1.100 table 12
  and can be removed as
  --table mangle -j MARK --set-mark 12; /sbin/ip rule del fwmark 12 table 12; /sbin/ip route del default via 193.138.1.100 table 12
- group is implemented via iptables as:
  <selector> redirect group GROUP
  -j GROUP
  and can be removed as
  -j GROUP
- blackhole is implemented as a special routing table with reserved number 5. In similar way also the rest of targets are implemented, thus the routing tables below 20 are reserved for special purposes. All the marked traffic, which is redirected to the blackhole with iptables command, is silently dropped. The default route to drop the traffic is defined only once via determination of device capabilities as:
  /sbin/ip route add blackhole all table 5
  The command sets the route in the default LINUX_BLACKHOLE_ROUTING_TABLE which contains the route:

[2] See http://linux-net.osdl.org/index.php/Iproute2 page for details.

blackhole default
and is removed when the device is detached with a command:
/sbin/ip route del blackhole all table 5
The rest of the blackhole is implemented as:
<selector> redirect blackhole
--table mangle -j MARK --set-mark 402493064; /sbin/ip rule add fwmark 402493064 table 5
and can be removed as
--table mangle -j MARK --set-mark 402493064; /sbin/ip rule del fwmark 0x17fd8e88 table 5

- sinkhole is currently implemented as the blackhole,
  <selector> redirect sinkhole
- prohibited uses routing table 7
  <selector> redirect prohibited
- unreachable uses routing table 8
  <selector> redirect unreachable
- netunreachable uses routing table 9
  <selector> redirect netunreachable

The usage of the redirection can be seen in the test case in Figure 3, lines 17, 25, 39, 44, 49, 59, 64, 68, 73 and 91. Consequently the output of the appended rules is presented in the Appendix section 8.1. An example of blackhole routing is presented in the output, lines 86 to 91. We can see that in this case a single rule can result in two related commands issued on this particular FD. The current implementation supports now such actions with an unlimited number of commands that can be executed on a certain device and even rolled back if needed.

The rate limit action in the case of Linux FD is again implemented using iptables and the ‘tc’ command from iproute2 package. We have used iptables to mark the selected packets and tc and qdisc less traffic shaping on ingress to implement the required functionality as stated in D2 [1] and D6 [3]. In the test case in Figure 3, we can see the appended rule that rate limits the selected traffic in line 96. The resulting output is shown in Appendix section 8.1 on lines 191 to 196. We tend to make the rate limiting as simple as possible since tc functionality is very complex and changes very often.

The consequences of the applied commands can be seen in the Appendix section 8.1 in lines 191 to 252. These lines present the control output of the configuration of iptables, ip rules and mark (in this case fwmark) values and the output of the tc configuration. We can note here that in difference to the implementation described in D8 [4], we have tried to make the use of groups as defined in D6 [3] implementation-independent and thus abstract. Now, the predefined groups as in case of Linux iptables are not supported any more in the current implementation. Of course, these groups are still used internally in the implementation.

In addition, the current implementation includes the implementation of the time policy, as stated in D6 [3]. As we have explained in D8 [4], we have used the command pattern to implement the Firewall API functions. The implementation of the time policy has required an extension of the command manager. The command manager is an entity through which all commands on the FE are passed. This has enabled us to implement the time policy in an efficient way. Besides, using the command manager has additional features, the roll and unroll of the commands can be easily implemented, the issued commands on the FE can be tracked and the entity can be used to implement authorization service efficiently.

5.1.3 Integration between Service Module and Code Module

Both the Service Module and the Code Module are implemented in Java. The Service Module acts as an RMI server; it binds the name "serviceAPI" to the class that implements the functions of the Service API in the RMI registry. The functions of the Service API were defined in deliverable D6 [3]. These functions are:

- OperationResult loadPolicy(Policy p)

- OperationResult removePolicy(Policy p)
- OperationResult activatePolicy(Policy p)
- OperationResult disablePolicy(Policy p)
- OperationResult loadModule(Module m)
- OperationResult removeModule(Module m)
- ListCapabilities listCapabilities()
- boolean hasModuleEnvironment()
- void addCapability(Module m)
- HashSet listLoadedPolicies()
- HashSet enforcedPolicies()
- HashSet listLoadedModules()
- ElementConfig getConfig()

OperationResult is a class that contains a Boolean variable to indicate if the operation was successful, and a String which contains the error message if a problem was encountered during the operation. An object of type Policy is a Java representation of a policy (specified in XML). The class Module represents a module that can be loaded in the Module Environment and it contains a field that points to the URL where the code for this module is available. The class ListCapabilities contains a list of the capabilities available at a particular Firewall Element, depending on the functionalities offered by the underlying Firewall Devices attached to this Firewall Element. The capabilities are of type String, and they contain the textual description of the functionality offered. For instance in the case where the Firewall Device is a Cisco router, the list of capabilities includes a String such as "redirection", which indicates that the Firewall Element has the capability to instruct the Firewall Device to redirect some traffic to a particular destination. The list of capabilities also includes the list of the modules loaded in the Module Environment, since they add functionalities to the Firewall Device. HashSet is a Java class that is used to store a list of objects. Finally, ElementConfig is a class that contains information about the Firewall Element, such as its IP address, its name, its network name, the attached devices, and so on.

The Code Module is a class that implements all the functions related to the management of the modules loaded in the attached devices. This includes communicating with the Module Environment in order to load/remove modules, and also keeping trace of the loaded modules.

Objects of type Service Module and Code Module are part of the Firewall Element and are run in the same Java Virtual Machine (JVM) therefore their integration is straightforward; the Service Module object calls methods on the Code Module object. These methods are:

- OperationResult loadModule(Module m)
- OperationResult removeModule(Module m)
- HashSet listLoadedModules()

5.1.4 Integration between Response Module and Firewall Module

The response API as defined in D6 [3] provides three functions to access the basic Firewall Device response mechanisms: kill session, redirection and rate limiting. The last two can be easily implemented on top of the current Firewall API implementation with an addition of stopping the specific redirection or rate limiting. By design, as explained in D5 [2], the Response API can be remotely accessed by outside entities like the System Manager. The API implementation has to support this requirement.

We have implemented four API functions related to rate limiting and redirection with usage of Firewall Module functionality. To enable remote access, we have used RMI.[3] The test case that shows how the functionality of the API can be used is presented in

```
public void testResponseModuleClient() throws Exception {
    _log.debug("\nClient -> Starting the Response Module client");
    ResponseIntf ri = ResponseModuleClient.getIntfReference();
    ri.rateLimit("tcp","192.168.1.5","192.168.1.1","","80","","1mbps","");
    ri.redirect("icmp","192.168.1.7","","","","193.138.1.100","","");
    ri.redirect("udp","192.168.1.7","","","","","blackhole","");
```

[3] See http://java.sun.com/products/jdk/rmi/ for details.

```
      Utility.sleep(2);
      ri.stopRateLimit("tcp","192.168.1.7","192.168.1.1","","80","","1mbps","");
      ri.stopRedirect("icmp","192.168.1.7","","","","193.138.1.100","","");
      ri.stopRedirect("udp","192.168.1.7","","","","","blackhole","");
20 }
```

Figure 4. In the Figure, two methods of two different JUnit tests are presented. The first starts the RMI server that exports the Response API and waits for 20 seconds for the client. The Response Module is implemented as a singleton so only one instance of the class is loaded in one Java virtual machine.

```
1  public void testResponseModuleRMI() throws Exception {
     _log.debug("\nServer -> Starting the Response Module");
     ResponseModule rm = ResponseModule.getReference();
     _log.debug(rm.toString());
5    _log.debug("Server -> Will wait for 20 seconds ...");
     Utility.sleep(20);
     rm.stop();
   }

10 public void testResponseModuleClient() throws Exception {
      _log.debug("\nClient -> Starting the Response Module client");
      ResponseIntf ri = ResponseModuleClient.getIntfReference();
      ri.rateLimit("tcp","192.168.1.5","192.168.1.1","","80","","1mbps","");
      ri.redirect("icmp","192.168.1.7","","","","193.138.1.100","","");
15    ri.redirect("udp","192.168.1.7","","","","","blackhole","");
      Utility.sleep(2);
      ri.stopRateLimit("tcp","192.168.1.7","192.168.1.1","","80","","1mbps","");
      ri.stopRedirect("icmp","192.168.1.7","","","","193.138.1.100","","");
      ri.stopRedirect("udp","192.168.1.7","","","","","blackhole","");
20 }
```

Figure 4: Response API test case, server and the client

The second method starts the Response Module client. The test that implements this is executed in another process in another Java virtual machine, delayed for a few seconds which allows the start of the Firewall Element invoked in the first method. The output of both test cases is presented in the Appendix section 8.2, intermixed, since both processes write to the same standard output, the second make command is not echoed to the terminal.

When the Response Module is started as a server, it attaches the local device as a Firewall Device to the Firewall Module, see Appendix section 8.2, lines 9 to 52. This part of the output is similar to that in the section 8.1. At that point the server is ready and waits for the client to connect. When the client connects to the server and gets the Response API implementation object (line 12 in the 10 public void testResponseModuleClient() throws Exception {

```
      _log.debug("\nClient -> Starting the Response Module client");
      ResponseIntf ri = ResponseModuleClient.getIntfReference();
      ri.rateLimit("tcp","192.168.1.5","192.168.1.1","","80","","1mbps","");
      ri.redirect("icmp","192.168.1.7","","","","193.138.1.100","","");
15    ri.redirect("udp","192.168.1.7","","","","","blackhole","");
      Utility.sleep(2);
      ri.stopRateLimit("tcp","192.168.1.7","192.168.1.1","","80","","1mbps","");
      ri.stopRedirect("icmp","192.168.1.7","","","","193.138.1.100","","");
      ri.stopRedirect("udp","192.168.1.7","","","","","blackhole","");
20 }
```

Figure 4) the method ‘rateLimit’ is called in the next line. This method is performed on the server and translates the call into the Firewall API rule as can be seen in the output, Appendix section 8.2, line 58. This rule is applied on the device as is shown in the same output, lines 60 to 65. After two seconds, the same rule is stopped; see 10 public void testResponseModuleClient() throws Exception {

```
      _log.debug("\nClient -> Starting the Response Module client");
      ResponseIntf ri = ResponseModuleClient.getIntfReference();
      ri.rateLimit("tcp","192.168.1.5","192.168.1.1","","80","","1mbps","");
      ri.redirect("icmp","192.168.1.7","","","","193.138.1.100","","");
15    ri.redirect("udp","192.168.1.7","","","","","blackhole","");
      Utility.sleep(2);
      ri.stopRateLimit("tcp","192.168.1.7","192.168.1.1","","80","","1mbps","");
      ri.stopRedirect("icmp","192.168.1.7","","","","193.138.1.100","","");
      ri.stopRedirect("udp","192.168.1.7","","","","","blackhole","");
20 }
```

Figure 4, line 15. After 20 seconds, the server is stopped, Firewall Device detached and the RMI process terminates.

The kill session function of the API is still work in progress. The current Firewall Devices do not support such response mechanism directly so it is not available as part of the Firewall API. But when the functionality of the call will be available, it can be made a part of both APIs. In addition, the Response API can be extended with some new functionality that is added to the Firewall API, if needed.

5.2 Interface with Firewall Devices

5.2.1 Integration between Code Module and Module Environment

For prototyping the Diadem Firewall system, we have chosen the OSGi technology for implementing the Module Environment. OSGi is an independent corporation that is defining a specification to deliver services over wide-area networks to local area networks and devices. OSGi has over 70 participants and is currently enjoying industry-wide support. The reasons behind this choice are that OSGi is a standard supported by many influential industrials and there are mature open-source implementations available.

In OSGi, components are called bundles. A bundle is a JAR file containing the code of a service. In our case, Firewall Modules are implemented as OSGi bundles. The Module Environment is containing a shell bundle that provides the commands necessary to manage the bundles, and a telnet bundle, to allow remote access.

5.2.2 Integration between Firewall Module and Commercial Firewall

In the DIADEM deliverable D8 [4], we described and demonstrated the integration of the commercial Cisco PIX firewall as a firewall device. As a short summary it can be said that Firewall API function calls are translated into the corresponding Cisco IOS commands. After, an SSHv1 connection is used to log on the firewall and execute the commands.

Since the writing of [4] we have implemented the support of the Group concept. This allows organizing firewall rules in groups which correspond to different ACLs. However, due to the differing concepts of Groups and ACLs, the Cisco firewall checks every incoming packet against the rules of all active groups. Furthermore, it is not possible to use rules for passing a packet to a specific group.

The second open issue concerns the support of specific kinds of selectors that cannot be translated into a single command of the Cisco IOS. Some of them like the selectors of packets that do not match a given IP address could not be implemented.

Finally, we still have to cope with the fact that newly added rules do not affect existing connections, i.e. existing states within the Cisco firewall are not deleted. In order to block existing connections, the *shun* command can be used. However, the syntax of this command differs from the definition of usual filtering rules and does not offer the same flexibility for specifying the affected connections.

5.2.3 Integration between Firewall Module and Commercial Router

The operating system on the Cisco 7200 router is IOS 12.6 and on the Cisco 3600 is IOS 12.3. They both can be accessed remotely through telnet. In consequence, the firewall module includes a component that uses telnet to log on the routers and issue the appropriate commands.

As an example, the following commands are used in order to perform the redirection of a flow:

```
ip access-list extended ACLNAME
 permit tcp any host 10.194.113.42 eq 22
route-map ROUTEMAPNAME permit 10
 match ip address ACLNAME
 set ip next-hop 10.194.113.35
ip policy route-map ROUTEMAPNAME
```

The first command creates an ACL (Access Control List):

```
ip access-list extended ACLNAME
 permit tcp any host 10.194.113.42 eq 22
```

The ACL defines a flow. In this example, all TCP packets destined to 10.194.113.42 port 22 will match the ACL.

The second command creates a route-map:

```
route-map ROUTEMAPNAME permit 10
 match ip address ACLNAME
 set ip next-hop 10.194.113.35
```

A route-map is a route definition for all packets matching a certain ACL. The keyword 'permit' means that the packets will be allowed to be forwarded by the router and 10 represents the priority of the route, so that the router knows in which order the different routes must be applied. In our example, the route-map specifies that the packets matching the ACL must be forwarded to IP address 10.194.113.35.

Finally the command `'ip policy route-map ROUTEMAPNAME'` activates the route-map so that packets are effectively re-routed.

The commands issued to the routers will of course depend on the policies enforced in the Firewall Element, and the events received. For each possible action that results from a policy, the appropriate set of commands is coded in the Firewall Module so that the action is actually enforced by the Firewall Device, which is in this case the router.

5.2.4 Integration between Firewall Module and Classification Engine

Implementation status of the integrated classification engine

The implementation of the integrated Classification Engine (CE) experienced a significant set back when we had to abandon the FPGA board we intended to use, the Amirix AP130. The documentation, development environment, and reliability turned out to be inadequate to such an extent that it would have been impossible to complete the project based on that board as we constantly encountered new problems and shortcomings. We shifted the development effort to a less advanced but reliable board with appropriate documentation, the Spyder-Virtex-X2E, equipped with a Xilinx Virtex XCV2000E FPGA.

In setting up the new environment we adapted our Diadem hardware driver to support the Spyder board. As the available Spyder software runs only on Linux 2.4, we had to extend our Diadem kernel module, the Classifier Module, to program the FPGA on a Linux platform with kernel version 2.6. This step required a different path from the user space into the kernel than our existing software. We had relied on passing strings via the new sysfs file system whereas the Spyder software requires *ioctl* calls on a traditional device node in the `/dev/` directory. We ported the X2E FPGA programming tool to run in the new environment and use the *ioctl* hooks of the Classifier Module.

In the process, it became evident that for the communication between the Classifier API in the user space and the Classifier Module in the kernel the direct passing of binary data types by ioctls is safer than the indirection of string conversions in the sysfs file system. Therefore, we migrated the path between user space and kernel from sysfs to ioctls. In implementing the ioctls, we have established communication between the Classifier Module and the FPGA which completes the path from the user space through the kernel module to the hardware.

We have also completed porting our prototype hardware to the Spyder board. We had to abandon the plan to extend the maximum number of 128 rules from the initial prototype as there was no time left to develop a new, larger TCAM emulation due to the implementation delay with the first FPGA board. However, the new prototype now supports classification based on IP source and destination address, transport-layer protocol, source and destination ports, and a TCP flag.

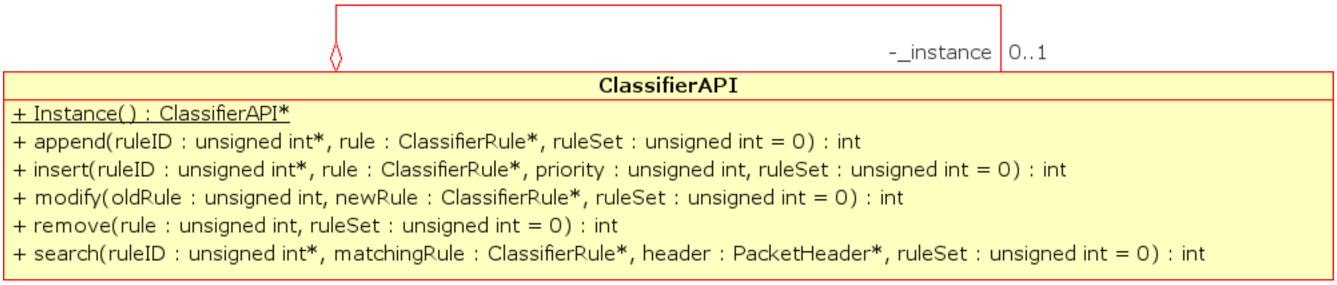
-_instance
0..1
ClassifierAPI
+ Instance() : ClassifierAPI*
+ append(ruleID : unsigned int*, rule : ClassifierRule*, ruleSet : unsigned int = 0) : int
+ insert(ruleID : unsigned int*, rule : ClassifierRule*, priority : unsigned int, ruleSet : unsigned int = 0) : int
+ modify(oldRule : unsigned int, newRule : ClassifierRule*, ruleSet : unsigned int = 0) : int
+ remove(rule : unsigned int, ruleSet : unsigned int = 0) : int
+ search(ruleID : unsigned int*, matchingRule : ClassifierRule*, header : PacketHeader*, ruleSet : unsigned int = 0) : int

Figure 5: Class diagram of the Classifier API.

To complete the integration of all CE components we have implemented a new version of the Classifier API that presents cleaner API prototypes and provides the required bookkeeping of the active rule set. Figure 5 shows the public methods of the C++ class. The class is a singleton, only accessible through `Instance()` to prevent inconsistencies in the rule-set bookkeeping between different instances. The remaining methods follow the specification in [3]. The setup functionality is implemented in the class constructor.

We have already integrated the new Classifier API with the Firewall Module. Currently, we are working on the integration of the new API with the rule compiler and the new ioctl calls before finally activating the Netfilter hooks

Stand-alone classification engine

The stand-alone Classification Engine (CE) is intended to act as a stand-alone packet classifier and processor, requiring no support from a host PC during packet processing. The initial hardware platform will be the Celoxica RC300 platform, which consists of a central Xilinx xc2v6000e Virtex-2 FPGA that provides processing functionality, connected to on-board devices including four independent banks of low-latency SRAM and an Intel Gigabit MAC. The MAC provides the FPGA with access to two network ports, allowing the FPGA to directly control and process two independent full-duplex streams of packets. There is no CPU on the board (unless one is instantiated within the FPGA logic), and no OS. Rule changes will be applied by communicating with a Linux host, which pre-processes the rules into a binary form usable by the FPGA, then transfers the rules to the firewall over USB or through the network. The Linux host acts as a Firewall Device, allowing the stand-alone firewall to be managed and queried by other modules within the Diadem system. Figure 6 shows the structure of the Linux and FPGA firewall, while Figure 7 shows that of the stand-alone hardware firewall, including the Linux host used to provide the Firewall Element interface. The bold outlined API block indicates common API.

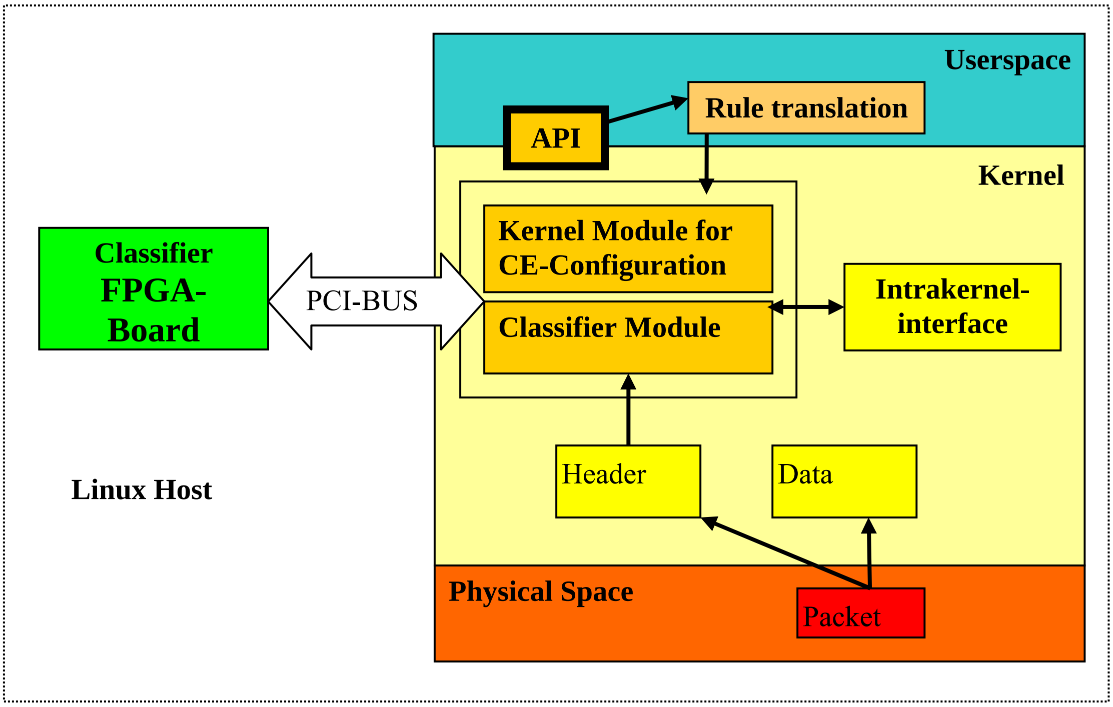

Figure 6: Integrated Classifier Engine architecture

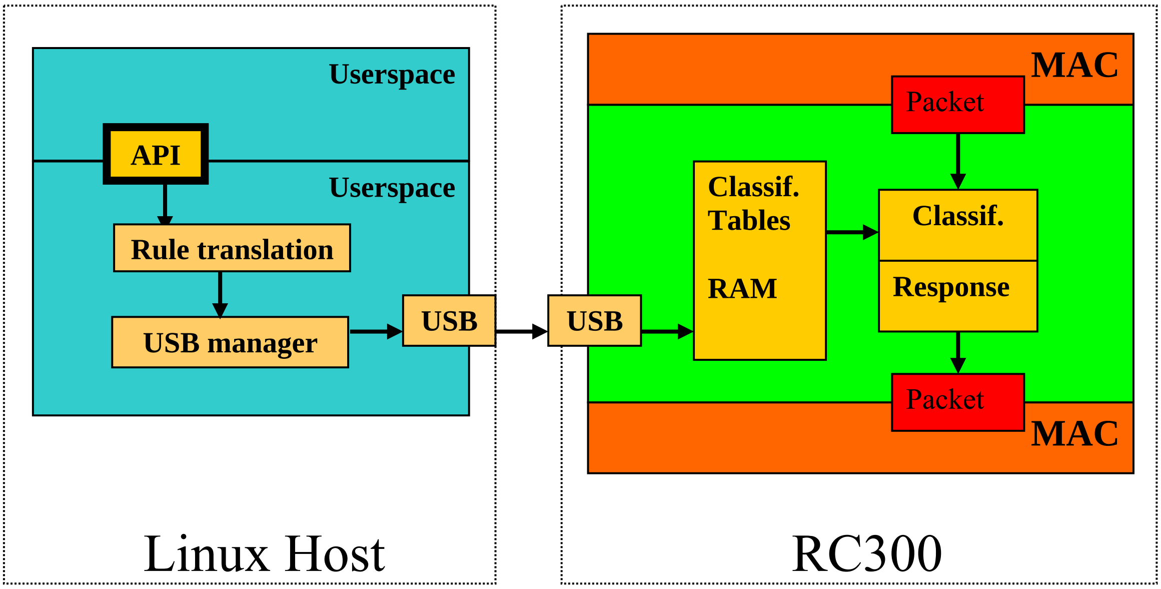

Figure 7: Stand-alone Classifier Engine architecture

The functionality of the stand-alone CE will be a subset of that provided by the existing Integrated CE. The same basic process of packet classification and processing is used in both CEs, but with some differences:

- The stand-alone CE operates at full line-speed, providing guaranteed full Gigabit processing speed in all conditions, while the Integrated CE is limited by the speed of the processor and OS stack performance.
- The functionality of the stand-alone CE is limited by the fixed size of the FPGA, reducing the number and complexity of the rules that can be applied, while the Integrated CE provides greater flexibility in terms of rule-set size and complexity.
- The Integrated CE performs full protocol handling, including packet fragmentation and identifying individual data streams, while the stand-alone firewall will be purely packet-based, ignoring all high-level network aspects.

In practise the two CEs will act together, with the stand-alone CE providing initial high-speed filtering to traffic entering the network, before further processing by the Integrated CE. Under an attack the Integrated CE can apply simple rules to reject the majority of known-bad attack traffic, allowing the Integrated CE to apply more complex rules at a slower rate to the remaining traffic.

Calling the Classifier API from the Firewall Module

For the integration of the FM with the CE, we have focused on two aspects: first, access from the FM implementation in Java to the CE via the Classifier API in C/C++, and second, the division in between the rules that can be applied on the CE and those that can be applied via iptables. Both hardware classification engines as presented in the previous two sections support the same Classifier API they are both treated from Java based Firewall API implementation in the same way. Both devices are different only in few capabilities that are implementation and design dependent. For a solution of the first issue we have used SWIG,[4] which can generate wrapped interfaces for many languages, including Java, to C or C++ libraries. Moreover, since the high speed classification engine hardware is not accessible to all the partners, we have implemented a dummy Classifier API for testing the integration. Via the presented capabilities this device can mimic both hardware implementations. We have wrapped the Classifier API implementation as a library and both tests of the dummy CE, in C++ and in Java, run successfully.

The second issue of limited CE capabilities can be solved in two ways: we can treat the CE as a firewall with limited capabilities and therefore on such a device only the suitable rules will be applied. An alternative solution would be to treat such a device as an ordinary Linux firewall but internally divide the rules in between those that can be applied on the CE and those that can be applied via iptables only. Both CEs can support both solutions if the CE is paired with a suitable Linux Firewall Device.

6 Conclusion

This document has described the integration issues and how they were solved. At the moment some issues are still unresolved because integration is still ongoing work.

7 References

[1] Initial interface specification, DIADEM Firewall deliverable D2, July 2004.
[2] Architecture Specification, DIADEM Firewall deliverable D5, January2005.
[3] Revised interface specification, DIADEM Firewall deliverable D6, January 2005.
[4] Initial Firewall Element Prototype, DIADEM Firewall deliverable D8, July 2005.

[4] See Simplified Wrapper and Interface Generator home page, http://www.swig.org for details.

[5] Münz, G. et al, “Initial Violation Detection Prototype”, DIADEM Firewall deliverable D9, July 2005.

[6] Claise, B. et al., “IPFIX Protocol Specifications”, Internet Draft, Work in Progress, draft-ietf-ipfix-protocol-19.txt, 2005.

[7] Quittek, J. et al, “Information Model for IP Flow Information Export”, Internet Draft, Work in Progress, draft-ietf-ipfix-info-11, 2005.

[8] Enns, R. et al., “NETCONF Configuration Protocol”, Internet Draft, Work in Progress, draft-ietf-netconf-prot-10, 2005.

[9] xmlBlaster Homepage: http://www.xmlblaster.org/.

[10] Elvin Homepage: http://elvin.dstc.edu.au/.

[11] Gabrijelčič, D. et al, “Revised Interfaces Specification”, DIADEM Firewall deliverable D6, January 2005.

[12] Debar, Hervé et al, “The Intrusion Detection Message Exchange Format”, Internet Draft, Work in Progress, draft-ietf-idwg-idmef-xml-14, 2005.

[13] Clark, J., DeRose, S, “XML Path Language (XPath) Version 1.0”, W3C Recommendation, 1999.

[14] Carlinet, Y. et al, “Architecture Specifications”, DIADEM Firewall deliverable D5, January 2005.

8 Appendix

8.1 Firewall API test case output

```
cd make; make -f firewall.make testapi
        make[1]: Entering directory `/home/dusan/delo/src/firewall/make'

        ***** running firewall API test
        /opt/sun-jdk-1.4.2.06//bin/java -classpath
"/home/dusan/delo/src/firewall/src:/home/dusan/delo/src/firewall/lib/ccl.jar:/home/dusan/delo/src/firewall/lib/javancss.jar:/home/dusan/delo/src/firewall
/lib/jhbasic.jar:/home/dusan/delo/src/firewall/lib/jsch.jar:/home/dusan/delo/src/firewall/lib/jssh.jar:/home/dusan/delo/src/firewall/lib/junit.jar:/home/dusa
n/delo/src/firewall/lib/log4j-
1.2.8.jar:/home/dusan/delo/src/firewall/lib/xercesImpl.jar:/home/dusan/delo/src/firewall/src:/home/dusan/delo/src/firewall/code/classifier.jar:/home/dus
an/delo/src/firewall/code/firewall.jar:/home/dusan/delo/src/firewall/code/response.jar" -Djava.security.egd=file:/dev/urandom
org/diadem_firewall/api/firewall/FirewallAPITest
        .
        Session with /193.138.1.100 over protocol ssh2 established
        Session: 193.138.1.100
        Address: /193.138.1.100
        Protocol: ssh2
        Connection time: 3

        Removed cap: ModuleEnvironmentjava.lang.Exception: The device of type Linux at /193.138.1.100 has no "ModuleEnvironment"
capability.
        Initializing the capability: NetUnreachable with command "/sbin/ip route add throw all table 9"
        Initializing the capability: Unreachable with command "/sbin/ip route add unreachable all table 8"
        Initializing the capability: Prohibited with command "/sbin/ip route add prohibit all table 7"
        Initializing the capability: PolicingOnIngress with command "/sbin/tc qdisc add dev eth0 ingress"
        Initializing the capability: Blackhole with command "/sbin/ip route add blackhole all table 5"
        Initializing the capability: Sinkhole with command "/sbin/ip route add blackhole all table 6"
        While executing the command: '/bin/ls /lib/iptables/libipt_string.so' the command has failed.
        Removed cap: MatchStringjava.lang.Exception: Execution of command '/bin/ls /lib/iptables/libipt_string.so' failed with exit status: 1
        Capabilities for firewall device at /193.138.1.100 of type Linux over protocol ssh2 determined!
        Firewall device of type Linux created and initialized!
        Added among firewall devices: 193.138.1.100

        !Command to be executed: groupSelect with an argument /
        Executed through: org.diadem_firewall.api.firewall.GroupSelect

        Selecting group: /


        !Command to be executed: groupCreate with an argument 193.138.1.100
        Executed through: org.diadem_firewall.api.firewall.GroupCreate


        !Command to be executed: groupSelect with an argument 193.138.1.100
        Executed through: org.diadem_firewall.api.firewall.GroupSelect

        Selecting group: 193.138.1.100

        Linux Firewall Device defaults set
        Firewall device stup finished

        !Command to be executed: groupCreate with an argument tcpstart
        Executed through: org.diadem_firewall.api.firewall.GroupCreateLinux

        Requesting a command: /sbin/iptables --new-chain tcpstart

        !Command to be executed: groupCreate with an argument web_servers
        Executed through: org.diadem_firewall.api.firewall.GroupCreateLinux

        Requesting a command: /sbin/iptables --new-chain web_servers

        !Command to be executed: ruleAppend with an argument selector:"intf=*;src=*;dst=*;proto=tcp;dstport=80" action:"redirect group
web_servers"
        Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

        Requesting a command:
        1.)/sbin/iptables --append INPUT --protocol TCP --destination-port 80 -j web_servers

        Reference added to the group: web_servers


        !Command to be executed: ruleAppend with an argument selector:"intf=*;src=*;dst=*;proto=tcp;state=new" action:"redirect group tcpstart"
        Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

        Requesting a command:
        1.)/sbin/iptables --append INPUT --protocol TCP -m state --state NEW -j tcpstart
```

```
Reference added to the group: tcpstart


!Command to be executed: groupSelect with an argument tcpstart
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: tcpstart


!Command to be executed: ruleAppend with an argument selector:"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;srcport=*;dstport=139" action:drop
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP --destination-port 139 -j DROP



!Command to be executed: ruleAppend with an argument selector:"intf=*;src=193.138.1.34;dst=*;proto=tcp;srcport=*;dstport=*" action:"redirect blackhole"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --source 193.138.1.34 --protocol TCP --table mangle -j MARK --set-mark 672970456
2.) /sbin/ip rule add fwmark 672970456 table 5



!Command to be executed: ruleAppend with an argument selector:"intf=*;proto=tcp;srcport=*;dstport=1111" action:"redirect prohibited"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --protocol TCP --destination-port 1111 --table mangle -j MARK --set-mark 501949570
2.) /sbin/ip rule add fwmark 501949570 table 7



!Command to be executed: ruleAppend with an argument selector:"intf=*;src=!193.138.1.0/24;dst=*;proto=tcp;srcport=*;dstport=25" action:"redirect netunreachable"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --source  ! 193.138.1.0/24 --protocol TCP --destination-port 25 --table mangle -j MARK --set-mark 523687883
2.) /sbin/ip rule add fwmark 523687883 table 9



!Command to be executed: ruleAppend with an argument selector:"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;srcport=*;dstport=22" action:pass
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP --destination-port 22 -j ACCEPT



!Command to be executed: ruleAppend with an argument selector:"intf=*;proto=tcp;dstport=22" action:"redirect address 193.138.1.100"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

New routing table for address 193.138.1.100, table number 21, count 1.
Requesting a command:
1.)/sbin/iptables --append tcpstart --protocol TCP --destination-port 22 --table mangle -j MARK --set-mark 545041447
2.) /sbin/ip rule add fwmark 545041447 table 21



!Command to be executed: ruleAppend with an argument selector:"intf=*;proto=udp;dstport=22" action:"redirect address 193.138.1.100"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Got routing table for address 193.138.1.100, table number 21, count 2.
Requesting a command:
1.)/sbin/iptables --append tcpstart --protocol UDP --destination-port 22 --table mangle -j MARK --set-mark 1194497590
2.) /sbin/ip rule add fwmark 1194497590 table 21



!Command to be executed: ruleAppend with an argument selector:"intf=*;proto=tcp;dstport=20:21" action:"redirect address 193.138.1.101"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

New routing table for address 193.138.1.101, table number 22, count 1.
Requesting a command:
1.)/sbin/iptables --append tcpstart --protocol TCP --destination-port 20:21 --table mangle -j MARK --set-mark 1228547868
2.) /sbin/ip rule add fwmark 1228547868 table 22
```

```
!Command to be executed: ruleAppend with an argument selector:"intf=*;src=*;dst=193.138.1.100;proto=icmp" action:"redirect queue"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append tcpstart --destination 193.138.1.100 --protocol ICMP -j QUEUE



!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/web_servers


!Command to be executed: ruleAppend with an argument selector:"intf=*;src=193.138.1.0/24;dst=193.138.1.2;proto=tcp;state=new" action:drop
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append web_servers --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP -m state --state NEW -j DROP



!Command to be executed: ruleAppend with an argument selector:"intf=*;src=193.138.2.0/24;dst=193.138.1.2" action:pass
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append web_servers --source 193.138.2.0/24 --destination 193.138.1.2 -j ACCEPT



!Command to be executed: ruleAppend with an argument selector:"intf=*;dst=193.138.1.80" action:"redirect address 193.138.1.101"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Got routing table for address 193.138.1.101, table number 22, count 2.
Requesting a command:
1.)/sbin/iptables --append web_servers --destination 193.138.1.80 --table mangle -j MARK --set-mark 1259135406
2.) /sbin/ip rule add fwmark 1259135406 table 22



!Command to be executed: ruleAppend with an argument selector:"intf=*;proto=tcp;dstport=80;state=new" action:"ratelimit 10kbps"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append PREROUTING --protocol TCP --destination-port 80 -m state --state NEW --table mangle -j MARK --set-mark 1082920409
2.) /sbin/tc filter add dev eth0 parent ffff: protocol ip prio 50 handle 1082920409 fw police rate 10kbps burst 1000k drop flowid :1


+++ List iptables:
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
web_servers  tcp  --  anywhere             anywhere            tcp dpt:www
tcpstart   tcp  --  anywhere             anywhere            state NEW
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
Chain tcpstart (1 references)
target     prot opt source               destination
DROP       tcp  --  193.138.1.0/24       kekec.e5.ijs.si     tcp dpt:netbios-ssn
ACCEPT     tcp  --  193.138.1.0/24       kekec.e5.ijs.si     tcp dpt:ssh
QUEUE      icmp --  anywhere             firewall.e5.ijs.si
Chain web_servers (1 references)
target     prot opt source               destination
DROP       tcp  --  193.138.1.0/24       kekec.e5.ijs.si     state NEW
ACCEPT     all  --  193.138.2.0/24       kekec.e5.ijs.si

+++ List mangle table where the redirect rule is inserted:
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
MARK       tcp  --  anywhere             anywhere            tcp dpt:www state NEW MARK set 0x408c0dd9
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
Chain tcpstart (0 references)
target     prot opt source               destination
```

```
MARK       tcp  --  kamra-win.e5.ijs.si  anywhere            MARK set 0x281cb6d8
MARK       tcp  --  anywhere             anywhere           tcp dpt:1111 MARK set 0x1deb2482
MARK       tcp  -- !193.138.1.0/24       anywhere           tcp dpt:smtp MARK set 0x1f36d7cb
MARK       tcp  --  anywhere             anywhere           tcp dpt:ssh MARK set 0x207cac27
MARK       udp  --  anywhere             anywhere           udp dpt:ssh MARK set 0x47329636
MARK       tcp  --  anywhere             anywhere           tcp dpts:ftp-data:ftp MARK set 0x493a271c
Chain web_servers (0 references)
target     prot opt source               destination
MARK       all  --  anywhere             ucilna.e5.ijs.si    MARK set 0x4b0ce1ae

+++ List ip rules and fwmark:
0:          from all lookup local
32759:      from all fwmark 0x4b0ce1ae lookup 22
32760:      from all fwmark 0x493a271c lookup 22
32761:      from all fwmark 0x47329636 lookup 21
32762:      from all fwmark 0x207cac27 lookup 21
32763:      from all fwmark 0x1f36d7cb lookup 9
32764:      from all fwmark 0x1deb2482 lookup 7
32765:      from all fwmark 0x281cb6d8 lookup 5
32766:      from all lookup main
32767:      from all lookup default

+++ List tc qdisc:
qdisc pfifo_fast 0: dev eth0 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc ingress ffff: dev eth0 ----------------

!Command to be executed: ruleDelete with an argument 10
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete web_servers --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP -m state --state NEW -j DROP



!Command to be executed: groupSelect with an argument //193.138.1.100/tcpstart
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: //193.138.1.100/tcpstart


!Command to be executed: ruleDelete with an argument 50
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP --destination-port 22 -j ACCEPT



!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/web_servers


!Command to be executed: groupFlush with an argument /193.138.1.100/tcpstart
Executed through: org.diadem_firewall.api.firewall.GroupFlushLinux


!Command to be executed: groupSelect with an argument /193.138.1.100/tcpstart
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/tcpstart

References: /193.138.1.100/20

!Command to be executed: groupSelect with an argument //193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: //193.138.1.100


!Command to be executed: ruleDelete with an argument 20
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete INPUT --protocol TCP -m state --state NEW -j tcpstart



!Command to be executed: groupSelect with an argument /193.138.1.100/tcpstart
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/tcpstart


!Command to be executed: ruleDelete with an argument 10
```

```
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --source 193.138.1.0/24 --destination 193.138.1.2 --protocol TCP --destination-port 139 -j DROP



!Command to be executed: ruleDelete with an argument 30
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --protocol TCP --destination-port 1111 --table mangle -j MARK --set-mark 501949570
2.) /sbin/ip rule del fwmark 0x1deb2482 table 7



!Command to be executed: ruleDelete with an argument 70
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Removed reference to routing table for address 193.138.1.100, table number 21, count 1.
Requesting a command:
1.)/sbin/iptables --delete tcpstart --protocol UDP --destination-port 22 --table mangle -j MARK --set-mark 1194497590
2.) /sbin/ip rule del fwmark 1194497590 table 21



!Command to be executed: ruleDelete with an argument 20
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --source 193.138.1.34 --protocol TCP --table mangle -j MARK --set-mark 672970456
2.) /sbin/ip rule del fwmark 0x281cb6d8 table 5



!Command to be executed: ruleDelete with an argument 90
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --destination 193.138.1.100 --protocol ICMP -j QUEUE



!Command to be executed: ruleDelete with an argument 40
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete tcpstart --source  ! 193.138.1.0/24 --protocol TCP --destination-port 25 --table mangle -j MARK --set-mark 523687883
2.) /sbin/ip rule del fwmark 0x1f36d7cb table 9



!Command to be executed: ruleDelete with an argument 60
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Removed reference to routing table for address 193.138.1.100, table number 21, count 0.
Removed routing table for address 193.138.1.100, table number 21, count 0.
Requesting a command:
1.)/sbin/iptables --delete tcpstart --protocol TCP --destination-port 22 --table mangle -j MARK --set-mark 545041447
2.) /sbin/ip rule del fwmark 545041447 table 21



!Command to be executed: ruleDelete with an argument 80
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Removed reference to routing table for address 193.138.1.101, table number 22, count 1.
Requesting a command:
1.)/sbin/iptables --delete tcpstart --protocol TCP --destination-port 20:21 --table mangle -j MARK --set-mark 1228547868
2.) /sbin/ip rule del fwmark 1228547868 table 22


Requesting a command: /sbin/iptables --delete-chain tcpstart

!Command to be executed: groupSelect with an argument null
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Removing the group tcpstart


!Command to be executed: groupFlush with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupFlush


!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
```

```
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: /193.138.1.100/web_servers

References: /193.138.1.100/10

!Command to be executed: groupSelect with an argument //193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: //193.138.1.100


!Command to be executed: ruleDelete with an argument 10
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete INPUT --protocol TCP --destination-port 80 -j web_servers



!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/web_servers


->

!Command to be executed: groupFlush with an argument /193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupFlushLinux


!Command to be executed: groupSelect with an argument /193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100

->

!Command to be executed: groupSelect with an argument /193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100


!Command to be executed: groupSelect with an argument /
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /


!Command to be executed: groupSelect with an argument 193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: 193.138.1.100


!Command to be executed: groupFlush with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupFlushLinux


!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/web_servers

References: /193.138.1.100/10

!Command to be executed: groupSelect with an argument //193.138.1.100
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: //193.138.1.100


!Command to be executed: ruleDelete with an argument 10
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Exception thrown ... java.lang.Exception: The requested rule at position + 10 does not exist!

!Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /193.138.1.100/web_servers

Exception thrown ...
```

```
        !Command to be executed: groupSelect with an argument /193.138.1.100/web_servers
        Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

        Selecting group: /193.138.1.100/web_servers


        !Command to be executed: ruleDelete with an argument 30
        Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

        Removed reference to routing table for address 193.138.1.101, table number 22, count 0.
        Removed routing table for address 193.138.1.101, table number 22, count 0.
        Requesting a command:
        1.)/sbin/iptables --delete web_servers --destination 193.138.1.80 --table mangle -j MARK --set-mark 1259135406
        2.) /sbin/ip rule del fwmark 1259135406 table 22



        !Command to be executed: ruleDelete with an argument 20
        Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

        Requesting a command:
        1.)/sbin/iptables --delete web_servers --source 193.138.2.0/24 --destination 193.138.1.2 -j ACCEPT



        !Command to be executed: ruleDelete with an argument 40
        Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

        Requesting a command:
        1.)/sbin/iptables --delete PREROUTING --protocol TCP --destination-port 80 -m state --state NEW --table mangle -j MARK --set-mark
1082920409
        2.) /sbin/tc filter del dev eth0 parent ffff: protocol ip prio 50 handle 1082920409 fw


        Requesting a command: /sbin/iptables --delete-chain web_servers

        !Command to be executed: groupSelect with an argument null
        Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

        Removing the group web_servers
        Flushing the 193.138.1.100 configuration

        !Command to be executed: groupSelect with an argument /
        Executed through: org.diadem_firewall.api.firewall.GroupSelect

        Selecting group: /


        !Command to be executed: groupFlush with an argument 193.138.1.100
        Executed through: org.diadem_firewall.api.firewall.GroupFlush


        !Command to be executed: groupSelect with an argument 193.138.1.100
        Executed through: org.diadem_firewall.api.firewall.GroupSelect

        Selecting group: 193.138.1.100


        Finalizing the capability: NetUnreachable with command /sbin/ip route del throw all table 9
        Finalizing the capability: Unreachable with command /sbin/ip route del unreachable all table 8
        Finalizing the capability: Prohibited with command /sbin/ip route del prohibit all table 7
        Finalizing the capability: PolicingOnIngress with command /sbin/tc qdisc del dev eth0 ingress
        Finalizing the capability: Blackhole with command /sbin/ip route del blackhole all table 5
        Finalizing the capability: Sinkhole with command /sbin/ip route del blackhole all table 6
        Session with /193.138.1.100 over protocol ssh2 dissconected

        Time: 6.653

        OK (1 test)
```

8.2 Response API test case output

```
firewall:~/delo/src/firewall:{1319*}> make response_test
        cd make; make -f response.make testapi
        make[1]: Entering directory `/home/dusan/delo/src/firewall/make'

        ***** running response API test
        /opt/sun-jdk-1.4.2.06//bin/java -classpath
"/home/dusan/delo/src/firewall/src:/home/dusan/delo/src/firewall/lib/ccl.jar:/home/dusan/delo/src/firewall/lib/javancss.jar:/home/dusan/delo/src/firewall
/lib/jhbasic.jar:/home/dusan/delo/src/firewall/lib/jsch.jar:/home/dusan/delo/src/firewall/lib/jssh.jar:/home/dusan/delo/src/firewall/lib/junit.jar:/home/dusa
n/delo/src/firewall/lib/log4j-
1.2.8.jar:/home/dusan/delo/src/firewall/lib/xercesImpl.jar:/home/dusan/delo/src/firewall/src:/home/dusan/delo/src/firewall/code/classifier.jar:/home/dus
an/delo/src/firewall/code/firewall.jar:/home/dusan/delo/src/firewall/code/response.jar" -Djava.security.egd=file:/dev/urandom -
Djava.security.policy=/home/dusan/delo/src/firewall/lib/policy org/diadem_firewall/api/response/ResponseAPITest &
        .
```

```
Server -> Starting the Response Module
Session with localhost/127.0.0.1 over protocol ssh2 established
Session: localhost
Address: localhost/127.0.0.1
Protocol: ssh2
Connection time: 3

Removed cap: ModuleEnvironmentjava.lang.Exception: The device of type Linux at localhost/127.0.0.1 has no "ModuleEnvironment" capability.
Initializing the capability: NetUnreachable with command "/sbin/ip route add throw all table 9"
Initializing the capability: Unreachable with command "/sbin/ip route add unreachable all table 8"
Initializing the capability: Prohibited with command "/sbin/ip route add prohibit all table 7"
Initializing the capability: PolicingOnIngress with command "/sbin/tc qdisc add dev eth0 ingress"
Initializing the capability: Blackhole with command "/sbin/ip route add blackhole all table 5"
Initializing the capability: Sinkhole with command "/sbin/ip route add blackhole all table 6"
While executing the command: '/bin/ls /lib/iptables/libipt_string.so' the command has failed.
Removed cap: MatchStringjava.lang.Exception: Execution of command '/bin/ls /lib/iptables/libipt_string.so' failed with exit status: 1
Capabilities for firewall device at localhost/127.0.0.1 of type Linux over protocol ssh2 determined!
Firewall device of type Linux created and initialized!
Got firewall device ...
Added among firewall devices: localhost

!Command to be executed: groupSelect with an argument /
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: /


!Command to be executed: groupCreate with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupCreate


!Command to be executed: groupSelect with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: localhost

Linux Firewall Device defaults set
Firewall device setup finished

!Command to be executed: groupSelect with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: localhost
Response module at host: localhost with an address 127.0.0.1

Server -> Will wait for 20 seconds ...

make[1]: Leaving directory `/home/dusan/delo/src/firewall/make'
firewall:~/delo/src/firewall:{1320*}> .
Client -> Starting the Response Module client
Server -> Rule:      selector:"src=192.168.1.5;dst=192.168.1.1;dstport=80;proto=tcp" action:"ratelimit 1mbps"

!Command to be executed: ruleAppend with an argument selector:"src=192.168.1.5;dst=192.168.1.1;dstport=80;proto=tcp" action:"ratelimit 1mbps"
Executed through: org.diadem_firewall.api.firewall.RuleAppendLinux

Requesting a command:
1.)/sbin/iptables --append PREROUTING --source 192.168.1.5 --destination 192.168.1.1 --protocol TCP --destination-port 80 --table mangle -j MARK --set-mark 1417121587
2.) /sbin/tc filter add dev eth0 parent ffff: protocol ip prio 50 handle 1417121587 fw police rate 1mbps burst 1000k drop flowid :1


Server -> Rule to be removed:      selector:"src=192.168.1.5;dst=192.168.1.1;dstport=80;proto=tcp" action:"ratelimit 1mbps"

!Command to be executed: ruleDelete with an argument 10
Executed through: org.diadem_firewall.api.firewall.RuleDeleteLinux

Requesting a command:
1.)/sbin/iptables --delete PREROUTING --source 192.168.1.5 --destination 192.168.1.1 --protocol TCP --destination-port 80 --table mangle -j MARK --set-mark 1417121587
2.) /sbin/tc filter del dev eth0 parent ffff: protocol ip prio 50 handle 1417121587 fw



Time: 2.486

OK (1 test)


firewall:~/delo/src/firewall:{1320*}>
!Command to be executed: groupSelect with an argument /localhost
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /localhost
```

```

!Command to be executed: groupSelect with an argument /
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /

!Command to be executed: groupSelect with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: localhost

Flushing the 127.0.0.1 configuration

!Command to be executed: groupSelect with an argument /
Executed through: org.diadem_firewall.api.firewall.GroupSelectLinux

Selecting group: /


!Command to be executed: groupFlush with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupFlush


!Command to be executed: groupSelect with an argument localhost
Executed through: org.diadem_firewall.api.firewall.GroupSelect

Selecting group: localhost


Finalizing the capability: NetUnreachable with command /sbin/ip route del throw all table 9
Finalizing the capability: Unreachable with command /sbin/ip route del unreachable all table 8
Finalizing the capability: Prohibited with command /sbin/ip route del prohibit all table 7
Finalizing the capability: PolicingOnIngress with command /sbin/tc qdisc del dev eth0 ingress
Finalizing the capability: Blackhole with command /sbin/ip route del blackhole all table 5
Finalizing the capability: Sinkhole with command /sbin/ip route del blackhole all table 6
Session with localhost/127.0.0.1 over protocol ssh2 dissconected
Server -> FirewallDevice localhost detached!
Server -> Kill the service
```