L7 Classification and Policing in the pfSense Platform

André Ribeiro, Helder Pereira
University of Minho, Department of Informatics
4710-057 Braga, Portugal
Email:{agentil,helderp}@di.uminho.pt

***Abstract*—The typical paradigm of identifying network traffic resorting to IP packet fields or to a set of well-known ports is highly limitative. Due to profound ongoing changes on the way applications try to hide their true nature by, for instance, using non default communication ports, a new challenge is presented to the way traffic classification and policing is accomplished. We argue and demonstrate that application layer inspection is a possible and convenient approach to derive the correct application protocol. This detection and classification process is of paramount importance to allow an efficient control of traffic entering the network. Taking pfSense as a case study, we extend its current layer 3 and 4 classification scheme with layer 7 capabilities, providing a powerful solution to control traffic based on application patterns. The user can easily create a set of rules for layer 7 inspection, which will drive lower level traffic control. In addition, we also provide a mechanism to create automatically useful application inspection scenarios.**

I. Introduction

The advent of services integration in a common network communication infrastructure as the Internet stresses the need to perform efficient traffic classification and policing, either due to resource management, charging or security concerns. The traditional way to classify traffic entering a network domain is usually based on network and transport data fields, e.g. service class marks, source and/or destination IP addresses and ports. Although in many cases this type of classification offers a good compromise between simplicity and efficacy, it fails to address particular cases where those fields are somehow inconclusive or unavailable. In fact, classification at layer 3 and 4 is hard to archive in presence of peer-to-peer (P2P) traffic as the applications use a range of random, non default ports. The typical cases of a HTTP server running on different ports from port 80 (port hoping) and encrypted connections are examples impairing a proper classification. In this context, performing traffic classification and policing at the application layer (layer 7 or L7 in short) can be a convenient solution to overcome these limitations. In L7 classification, user traffic can be identified based on an application pattern. An application pattern is a sort of signature used by an application during its communications. All applications use a specific application pattern or may share the pattern with other applications, as Pidgin or Google Talk.

A version of this paper has been submitted to the 21st International Teletraffic Congress (ITC 21), 2009, Paris, France.

Examples of related work on L7 classification include IPCop Firewall [1] and Bandwidth Arbitrator [2]. IPCop Firewall is a firewall platform that can be extended with L7 capabilities, while having other tools such as VPN, IDS, Proxy, Firewall, QoS and others. IPCop administration can be done through a simple web browser, with a secure and cryptographic connection. Although IPCop can support classification by application protocol, it does not allow the definition of shaping policies, only accepting blocking policies. This limits greatly the use of this feature, as one cannot assign different QoS parameters based on the application protocol.

In the present work, we study and tackle the L7 classification paradigm for the pfSense platform [3]. This free and open source platform is a variant of the FreeBSD operating system, specifically used as firewall and router. Although pfSense already includes support for traffic classification at application layer, it does not expose that capacity to the user. In this context, we have established the following goals: i) to develop mechanisms to control the classification component in the application protocol, integrating them in the platform through a graphical interface; ii) to define and implement user-friendly wizards to simplify the configuration of QoS rules; iii) to plan and develop a test platform which allows testing multiple patterns of applications simultaneously, and to measure the performance (e.g. response time) of the classification module based on the application layer.

The contents of this paper is organized as follows: the pfSense platform is described in Section II; the design and implementation issues regarding the classification solution are explained in Section III; the proof-of-concept and performance results are included in Section IV; and finally, the main conclusions are summarized in Section V.

II. The pfSense platform

A. A brief description

pfSense is a customized FreeBSD distribution, mainly oriented to be used as a firewall and router [3]. It started as a fork of the m0n0wall project. m0n0wall was mainly directed towards embedded hardware installations. pfSense, on the other hand, it is more focused on full PC installations, despite the fact that pfSense also offers solutions for embedded hardware. It includes many base features, and can be extended with the package system, including “one touch” installations.

pfSense is currently a viable replacement for commercial firewalling/routing packages, including many features found on commercial products (Cisco Pix, SonicWall, WatchGuard). The list of features, among others, include the following: firewall, routing, QoS differentiation, NAT, Redundancy, Load Balancing, VPN, Report and Monitoring, Real Time information, and a Captive Portal. It is fully prepared for high throughput scenarios (over 500 Mbps), as long as high end server class hardware is used. Common deployment scenarios include but are not limited to: Perimeter Firewall, LAN or WAN router, Wireless Access Point and several special purpose appliances, such as VPN, Sniffer, DHCP server, DNS and VoIP appliances. Adding to this, pfSense includes a powerful Web Configuration Interface, which allows the configuration of all pfSense capabilities.

pfSense uses a single XML file, called *config.xml*, that stores the configuration of all services available in the pfSense machine. This allows pfSense to be easily backed up, since *config.xml* is a self-contained configuration file, i.e. a newer machine can be fully constructed from scratch with a single file restore. The code responsible for the operation of the different pfSense services is essentially written in PHP, which makes easy to extend the current code base, improving existing features or adding new ones.

Next, we will concentrate on QoS management within pfSense discussing its main features and limitations.

B. QoS Management

QoS management in pfSense is achieved through the use of the AltQ framework [4], [5]. AltQ is used to provide queuing disciplines and other QoS mechanisms in order to perform resource sharing and QoS control. Traffic schedulers available in AltQ, including Class Based Queuing (CBQ), Priority Queuing (PRIQ) and Hierarchical Fair Service Curve (HFSC), can be configured automatically through the use of a Traffic Shaper Wizard. In the newest builds (2.0 Alpha), pfSense includes an additional shaping mechanism, called Dummynet [6]. Dummynet was originally available to the ipfw firewall, but recently the pf firewall (which is used in pfSense) is able to use it. Despite the fact that it was originally thought as a tool to test network protocols, it is also currently used to manage bandwidth. It is able to simulate or enforce "queue and bandwidth limitations, delays, packet losses, and multipath effects", and uses Weighted Fair Queuing (WFQ2+) as scheduler. At present, QoS management in pfSense is carried out at the layer 3 and layer 4 of the OSI model. This means that applications and services are only recognized by IP packet fields or by a set of standard ports. This is a major limitation as there are many software applications that do not use well-known ports to communicate, suffer from "port hoping", or may change its communications port at any time in an unpredictable way.

As mentioned before, performing traffic classification at the application layer may be a prominent answer to address this problem. In fact, comparing flows from the application layer with a set of pre-defined pattern files, one can identify what application protocol is being used. We will address this issue and study how to integrate L7 inspection capabilities within pfSense.

III. L7 Traffic Control: Design and Implementation

A. Designing the Solution

The initial design goal behind our L7 traffic control solution was to balance appropriately the granularity and simplicity of the configuration process. When exposing L7 features to the end user, it is convenient to provide detailed configuration facilities in order to allow a fine tuning of control options, but also to give the opportunity to create a set of pre-defined parameters in a simple way, based on a set of pre-defined options. With these two goals in mind, the design options and components for the proposed solution are discussed next.

L7 classification is performed using an application called *ipfw-classifyd*. This application is able to: (i) produce blocking rules for incoming traffic; or (ii) perform traffic shaping by assigning IP packets of a specific flow to an AltQ queue or to a Dummynet pipe or queue. For simplicity, from now on, the term "Structure" refers either to an action, an AltQ queue or a Dummynet pipe or queue. The values of the structures themselves, such as "block", from now on will be called "Behaviors". A more in-depth explanation of how *ipfw-classifyd* works will be given in Section III-B.

An important goal is to provide a graphical user interface in order to allow the user to leverage the potentialities of *ipfw-classifyd*. Here, the trade-off between configuration simplicity and granularity has to be taken into consideration.

In sake of simplicity, the user does not need to be aware of the framework names that will be used in the Structures. Therefore, instead of displaying the structures to the user as AltQ, Dummynet pipe and Dummynet queue, a simple and intuitive terminology should be used, hiding the underlying framework. Thus, the user sees as available Structures: (i) Action; (ii) Queue (which contains AltQ queues); and (iii) Limiter (which contains Dummynet pipes and Dummynet queues). These simplified names were chosen in accordance with terminology of other pfSense shaping mechanisms.

As regards the Dummynet components in the Limiter structure, it was decided that the user does not need to know the difference between a pipe and a queue, being that difference automatically detected by the back-end code.

Establishing an arbitrary number of protocol definitions is a relevant option for the user. However, duped definitions, such as defining the same protocol twice, should not be allowed. When a dupe definition is found, it must be discarded and the user should have the opportunity to correct the situation.

When L7 rules definition is created, the user shall have a way to assign that rule to a pf rule. Due to a current *ipfw-classifyd* limitation, only TCP and/or UDP packets can be analyzed, which means that the user will not be allowed to apply L7 inspection rules to other traffic than TCP/UDP.

Other important design goal is to provide an easy way to create a set of pre-defined L7 rules. This means that during the creation of the rules, the user options are restricted to the selection of protocols, i.e. the user cannot select which structures and behaviors should be applied to the protocols. Those structures and behaviors should be provided automatically in accordance with the classification group to which a specific protocol belongs to. In this context, the decision of changing the existing traffic shaper, extending it with L7 rules was considered the best option.

B. Implementation

The steps toward the implementation of the proposed solution are detailed in this section. To better understand our proposal, some knowledge is required on how the pfSense framework exposes its features and how the necessary tools work.

The implementation phase can be generically divided in four steps: (i) L7 classification - involves understanding the inner working components of *ipfw-classifyd*, how to leverage it, understanding as well the AltQ framework and the Dummynet application; (ii) traffic redirection - involves redirect traffic caught by a pf rule to *ipfw-classifyd* in order to enforce the behavior defined by L7 rules; (iii) definition of rules- involves the creation of a graphical web interface where L7 rule definitions can be built, specifying what to do when there is a match with a specific application protocol. These rule definitions, from now on will be known as L7 *containers*; (iv) definition of wizards - involves the implementation of a wizard where a L7 container with a pre-defined set of L7 rules can be easily created, through few simple steps, in an automated and transparent way to the end user.

1) ipfw-classifyd: As stated before, *ipfw-classifyd* is the application responsible for L7 classification. Despite the fact that it includes "ipfw" in the name, it was modified to work with the pf firewall as well. *ipfw-classifyd* has a straightforward configuration. Essentially, it allows there different types of operations to be applied to an identified application protocol, namely, defining an action to apply, typically a block action, or assigning traffic to an AltQ queue, a Dummynet pipe or a Dummynet queue. This allows some granularity, however, knowing that AltQ queues and Dummynet pipes and queues are defined in the pf rules file, it would not be easy to specify them clearly within the *ipfw-classifyd* file in a non automatized way. Furthermore, Dummynet uses numbers to identify pipes and queues, dynamically assigned every time the pf rules file is rebuilt by the pfSense framework, while those same pipes and queues are identified by common names in the pfSense platform.

A way to expose the *ipfw-classifyd* features to the user, while abstracting the QoS mechanisms and their implementation details, is therefore recommended. This abstraction means that the user is not required to know the difference between a Dummynet pipe and a queue, i.e. the difference can be automatically detected by the back-end code. In order to expose *ipfw-classifyd* capabilities, the concept of *L7 containers* was introduced and implemented. A L7 container is a structure that contains multiple definitions for different application protocols. It allows to specify what to do with each protocol the user intends to configure. Those protocol patterns (typically identified by an *application protocol name.pat*) are stored in a specific location inside the filesystem, and are used to dynamically provide the list of available application protocol patterns to the user. This means that, somehow, the user should also be allowed to add non default protocols to the ones already defined.

As mentioned before, there are three different types of operations that can be applied to an identified protocol. These type of operations are a miscellaneous of *Structures* and *Behaviors*. A *Structure* can be an "Action", a "Limiter" or a "Queue". A "Limiter" corresponds to a Dummynet Structure and a "Queue" to an AltQ queue. A *Behavior* is related to a structure. In the case of the Structure Action, the only available behavior is "block"; for the Limiter, it is a set of Dummynet pipes and queues; and for the Structure Queue, the behavior is a set of AltQ queues. The behaviors for Queue and Limiter must be present in the system (this means they should have been already created in the right places). After the user defines a L7 container, all data is already there to produce a *ipfw-classifyd* configuration file successfully. However a way to send the traffic to *ipfw-classifyd* is still needed. In this context, pf recently acquired one more option, called divert sockets.

2) Divert sockets: Divert sockets are, essentially, a way for the kernel to send network traffic to the user context. As regards *ipfw-classifyd*, diverting actually interrupts the normal flow of packets and sends them to a listening socket (in this case, *ipfw-classifyd*), or sends data directly to the IP processing routine. This is a context switch, in which packets abandon kernel land and go to userland. Clearly, this context switch has an overhead attached to it. To keep it small and controlled to minimize the impact on performance, *ipfw-classifyd* takes some actions. First, if pf is taught previously about the actions to take, when *ipfw-classifyd* daemon decides that something should change from its normal workflow, the effects of context switching can be minimized. The overhead can also be reduced with the definition of a pf rule option to limit the number of packets that are diverted from the kernel to the userland software application. That option is called "max-packets" and can be defined inside the "keep state" option. In a first approach, and for test purposes, a maximum of five packets are diverted (max-packets=5). This is purely an experimental value and might be subject to change. In the future, this parameter is planed to be offered as a user option to allow a fine tuning, whenever packets are not correctly identified. This may occur when *ipfw-classifyd* has insufficient packets to analyze the application protocol conveniently.

The need to tell pf in advance what actions to take, also requires that pf knows the corresponding structures (action, AltQ, Dummynet) to overload. This indication is given through the "keep state" option of a pf rule, and written as "over-

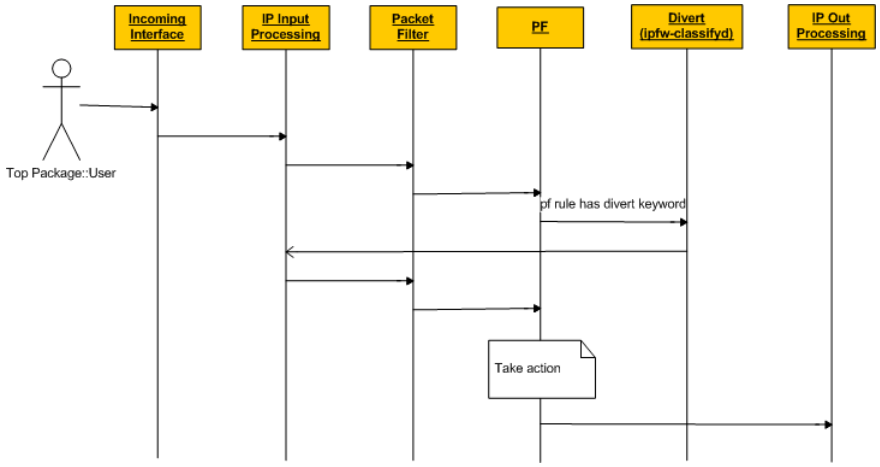

Fig. 1. IP Traffic entering and exiting ipfw-classifyd

load structure diverttag". After knowing what structures to overload, the set of parameters to successfully produce a pf rule to divert the traffic to *ipfw-classifyd* is almost complete. However, a key component is still missing, which is the divert port. Again, for test purposes, ports in the 40000-60000 range were considered for *ipfw-classifyd*. This decision is not critical as divert sockets exist in their own domain, and do do not correlate to non divert-aware applications.

In order to easily assign a divert port to a L7 container, a random unused port is created every time the rules are reloaded. Once the port is defined and validated, the L7 container is ready. In practice, a L7 container is ready when *ipfw-classifyd* is running in the specified divert port, the set of rules is loaded and there is a pf rule that diverts the traffic to *ipfw-classifyd*.

The typical pf<=>ipfw-classifyd flow sequence is illustrated in Figure 1.

3) L7 container interface: This section describes the proposed graphical interface which allows the user to create and change L7 containers in a simple and intuitive way, allowing to control the operation of *ipfw-classifyd*. A view of the graphical web interface for creating the rule containers is depicted in Figure 2.

As regards the interface design and implementation, several relevant decisions are highlighted.

Each container may have more than one application protocol specified, however, application protocols cannot have dupe specifications. In each rule that is created, the "protocol", "structure" and "behavior" must be specified. The "protocol" field allows the user to choose the protocol application to create a rule for; the "structure" and "behavior" are an aggregate pair conditioning what to do with the detected traffic. If an "action" is chosen, the behavior will be "block", i.e. the traffic will be blocked. If "queue" is the option, the behavior is defined by selecting one of the AltQ queues configured in the system. On the other hand, if "limiter", is chosen the behavior is defined by selecting one of the Dummynet queues or pipes available to the user. As explained, the difference between is transparent to the user, being assured by the back-end code. The "structure" field is also dynamically filled only with structures having, at least, one queue or pipe defined. For example, if the definition just includes AltQ queues and none Dummynet limiters, the available options are restricted to "action" and "queue" on the structure field, and "limiter" is hidden. The inclusion of "behaviors" is determined accordingly to the selected "structure".

The level of control implemented, using javascript code, also has the objective of minimizing user configuration errors.

4) Assigning a L7 container to a traffic flow: For each container that is built, the user may decide to assign it to a firewall rule. To cover this facility, a specific field was inserted into the Firewall Rules Edit page so that the user may take that option. As shown in Figure 3, if a L7 container is chosen to be applied, all traffic from matching rule will be analyzed by *ipfw-classifyd*.

Due to a current limitation of *ipfw-classifyd*, only UDP and/or TCP streams can be diverted to *ipfw-classifyd*. Improvements to *ipfw-classifyd* are underway as it is currently a work-in-progress. In addition, for each firewall rule only one L7 container can be assigned. This is a pf limitation since only one divert can be done per pf rule.

5) Creating L7 aware wizards: At this point, a straightforward way allowing a user to create a standard set of rules was still missing. As pfSense platform had already wizards to configure the shaper, our first thought was to create an explicit L7 configuration wizard. After intense discussion within pfSense developers forum, the resulting decision was that this could not be the best route. Instead, the preferable option was to use the current wizards, extending them to include L7 capabilities, in a completely transparent way to the user.

All the AltQ queues created by the wizard are used in the L7 configuration file in order to mimic non L7 shaper behaviors

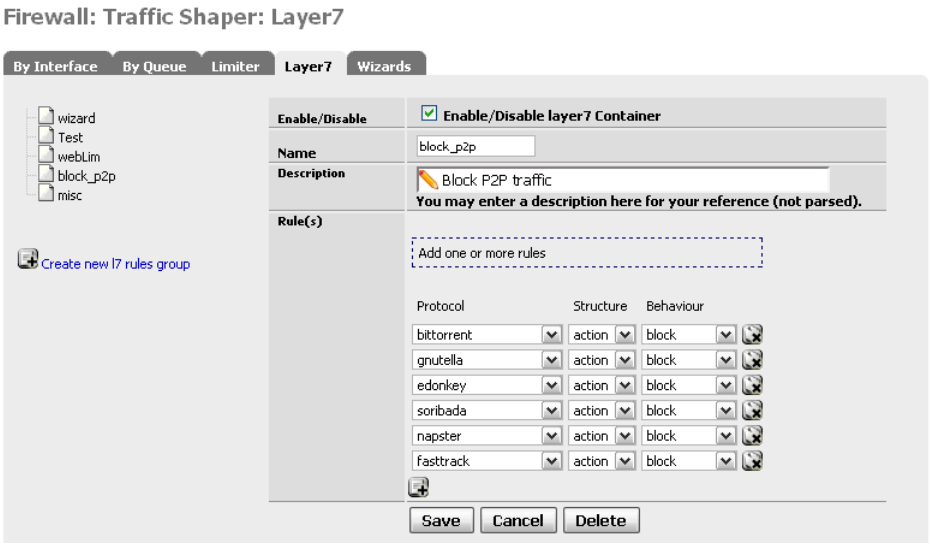

Fig. 2. Graphical interface for creating rule containers

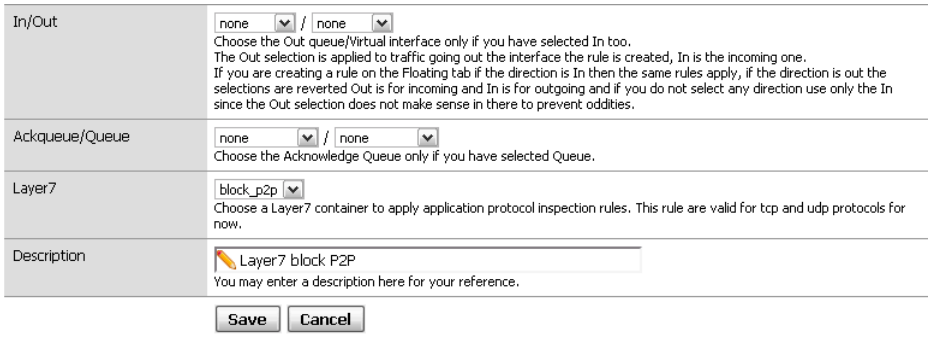

Fig. 3. Redirecting all traffic in the rule to ipfw-classifyd daemon instance

in *ipfw-classifyd*. The problem with this was that, not every protocol had a direct correspondence with an application protocol pattern. As a consequence, not all selections included in the wizard are translated into application protocol shaping.

As regards VoIP services and applications, it was decided that the most common application protocols related to VoIP would be assigned to a VoIP queue, without showing this to the end user. This is also completely transparent to the user, to whom the only concern is selecting what application protocols to shape.

The wizard creation for relevant L7 traffic, such as Peer-to-Peer and Network Games, is illustrated in Figures 4 and 5. If the traffic to shape is integrated in Peer-to-Peer applications, the select box on top of the page can be enabled and the related protocols or applications can be selected (blocked) one by one. By default , the wizards already comprise a comprehensive set of application protocols, however, new patterns can be uploaded to upgrade the interface.

Although the application protocol verification is currently performed by port and pattern, the purpose is to become only pattern-based. Using the application protocol inspection

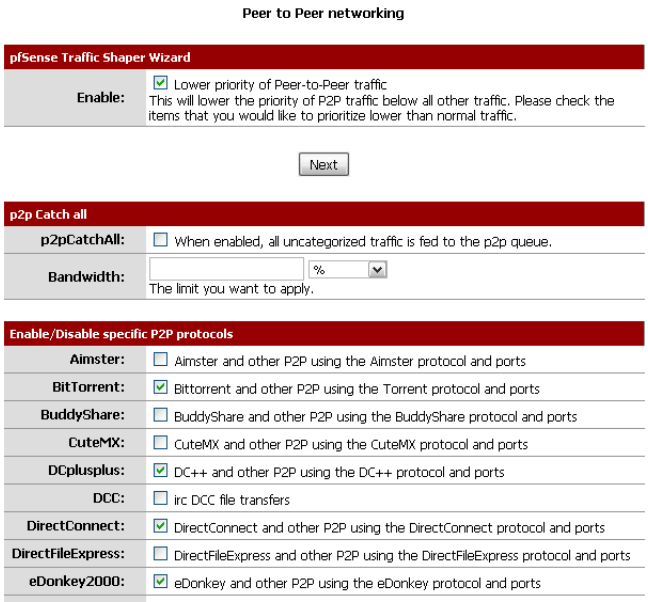

Fig. 4. Wizard creation - Peer-to-Peer step

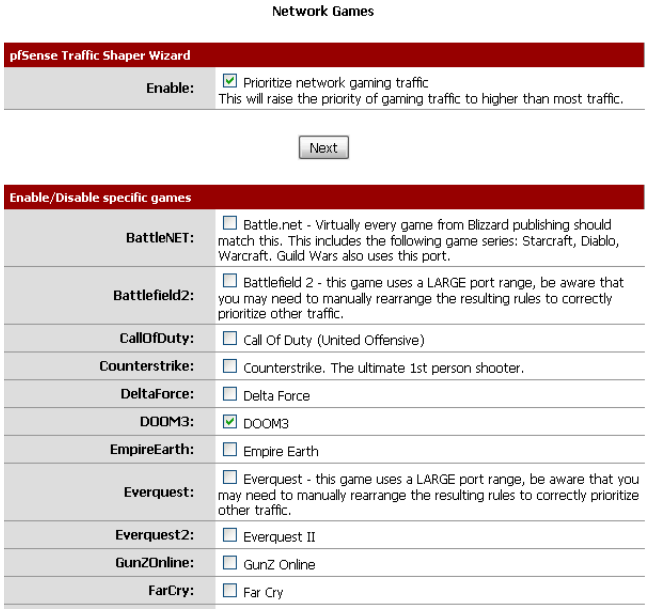

Fig. 5. Wizard creation - Network Games step

turns the port verification unnecessary and not so granular as application protocol inspection. This improvement will be done in the near future.

6) Upgrading or adding new L7 pattern files: A relevant feature to improve the support for L7 inspection is the possibility of allowing the user to upgrade the platform with new application protocol patterns.

Figure 6 illustrates how the user can upload new application patterns to the system. This feature is important when the user wants to block an application that uses a protocol pattern that is not defined in the system. If a new pattern is uploaded to the system, it only appears in the list of protocols when a container is created or modified. This does not affects the wizard, which remains unchanged. If a pattern that already exists is uploaded, it will be replaced. Therefore, the user must be very careful when uploading new application protocol patterns as previous definitions are overwritten. This flexibility is crucial since L7 inspection can have a wider use than simple application protocols' inspection. As example is the detection of specific URLs for HTTP traffic (providing the right pattern file), which is not directly connected to the detection of application protocol patterns.

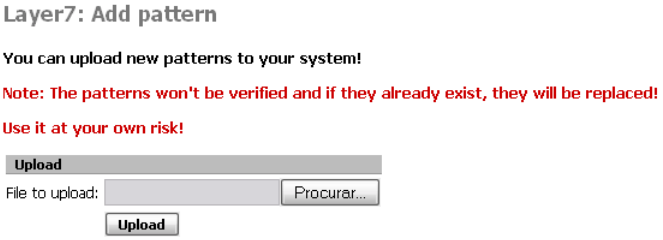

Fig. 6. Adding new application patterns

IV. Results

In this section, the process of L7 classification and policing is discussed from a practical perspective, highlighting representative configuration steps and obtained results.

First, we will show the result of creating a simple L7 container with a strict block policy (block_p2p) and how it is stored in *config.xml*. Figure 2 illustrates a L7 container and Figure 7 exemplifies how that container is stored in *config.xml*.

```
<container>
    <name>block_p2p</name>
    <enabled>on</enabled>
    <description>Block P2P traffic</description>
    <divert_port>47244</divert_port>
    <l7rules>
        <protocol>bittorrent</protocol>
        <structure>action</structure>
        <behaviour>block</behaviour>
    </l7rules>
    <l7rules>
        <protocol>gnutella</protocol>
        <structure>action</structure>
        <behaviour>block</behaviour>
    </l7rules>
```

Fig. 7. Piece of block_p2p container in XML

As shown, the definitions from the L7 container GUI are easily stored in *config.xml*. As there is only a block policy, the translation from the *config.xml* definition to the *ipfw-classifyd* configuration file[1] is somewhat strict, as illustrated in the following *ipfw-classifyd* configuration file extract:

```
bittorrent = action block
gnutella = action block
edonkey = action block
fasttrack = action block
```

Next, this container is assigned to a specific Firewall Rule, as shown in Figure 3. As there is only a single block policy, the resulting pf rule is also simple:

```
pass  in  quick  on $LAN  proto
{ tcp udp }  from {  192.168.160.2 }
to 192.168.87.2 divert 47244 keep
state ( max-packets 5,
overload action diverttag )
label "USER_RULE: Layer7 block P2P"
```

It is clear that the overload option is correctly created, since only one action needs to be overloaded.

Now, we will show how containers with Dummynet structures are handled. As explained before, Dummynet structures are exposed to the user with current names, but when defining them in pf configuration file they only have numbers. Thus, a translation mechanism is required in order to allow the end user to still be able to choose the Dummynet structures by their names. In addition, an automatic detection mechanism

[1] As *ipfw-classifyd* is currently work-in-progress and is evolving at a good pace, some of the files or configurations included in this paper may suffer minor changes.

was created in order to differentiate Dummynet pipes and Dummynet queues in a totally transparent way. Figure 8 exemplifies a container with several Dummynet structures. “Web” and “Others” are Dummynet queues and “Lim_2mb” is a Dummynet pipe. The *ipfw-classifyd* configuration file for this L7 container is as follows:

```
http = dnpipe 2
pop3 = dnqueue 2
smtp = dnqueue 2
cvs = dnqueue 1
```

To understand this configuration file, one also needs the configuration section that defines the Dummynet structures in the pf configuration file:

```
dnpipe 1 bandwidth 1Mb
dnqueue 1 dnpipe 1 weight 1
dnqueue 2 dnpipe 1 weight 3

dnpipe 2 bandwidth 2Mb
```

In this configuration, it is clear that “Lim_2mb” is dnpipe 2, “Web” is dnqueue 2 and “Others” is dnqueue 1. These queues belong to dnpipe 1, which has “Lim_1mb” as internal. Looking at the resulting *ipfw-classifyd* configuration file and to the L7 container definition, one can easily conclude that the name translation is successfully accomplished, and dnpipes and dnqueues are correctly identified. The corresponding Dummynet structures are also correctly identified as dnpipes and dnqueues, without the user indicating it explicitly. As for the pf rule, it is also correctly created, overloading uniquely the Dummynet structure:

```
pass  in  quick  on $LAN  proto
{ tcp udp }  from {  192.168.160.1 }
to 192.168.87.2 divert 51391 keep
state ( max-packets 5,
overload dummynet diverttag )
label "USER_RULE: Layer7 webLim"
```

Now, we illustrate how this solution handles a container with the three possible structures configured (see Figure 9). The resulting *ipfw-classifyd* configuration file is:

```
sip = queue qVoIP
bittorrent = action block
http = dnpipe 2
```

and the corresponding pf rule:

```
pass  in  quick  on $LAN  proto
{ tcp udp }  from {  192.168.160.10 }
to 192.168.87.2 divert 53363 keep
state ( max-packets 5,
overload action diverttag
overload dummynet diverttag
overload altq diverttag )
label "USER_RULE: Layer7 Miscellaneous"
```

The important part in this rule is showing that the structures to be overloaded were correctly identified. Three different type of structures were created in the *ipfw-classifyd* configuration file and were also correctly identified. As for the Dummynet translation, it was already concluded that it is being correctly applied. For AltQ queues, no special attention has to be taken, since the names defined in the L7 container are already the final names, unlike Dummynet pipes and queues.

Next, it will be shown how the shaper wizard was implemented. As stated before, the rules produced by the wizard are completely transparent to the final user. The user only selects the applications where shaping is due to be applied, and the wizard is smart enough to understand which application protocol is related with to each particular application. A part of the wizard L7 container is represented in Figure 10, showing the configured protocols.

As illustrated, since this is a shaping policy, only AltQ queues are assigned. Several different queues with specific QoS parameters adapted to the type of traffic they are going to receive can be observed. The QoS parameters for these queues are automatically derived from the link bandwidth and the type of traffic they are going to receive. Part of the wizard *ipfw-classifyd* configuration file is shown below:

```
sip = queue qVoIP
rtp = queue qVoIP
...
bittorrent = queue qP2P
edonkey = queue qP2P
...
doom3 = queue qGames
xboxlive = queue qGames
...
rdp = queue qOthersHigh
vnc = queue qDefault
msnmessenger = queue qOthersLow
...
```

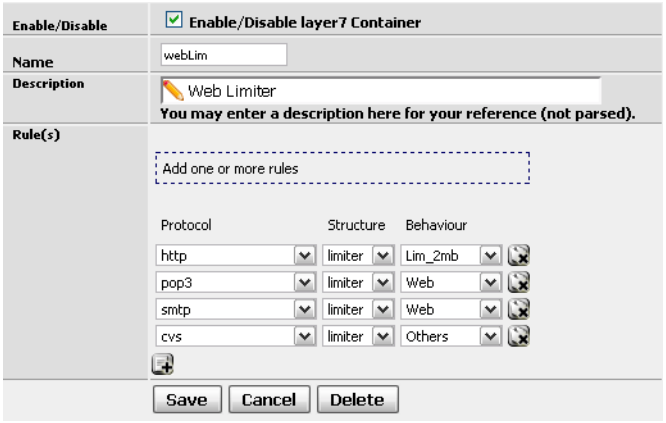

Fig. 8. Web_Lim container creation

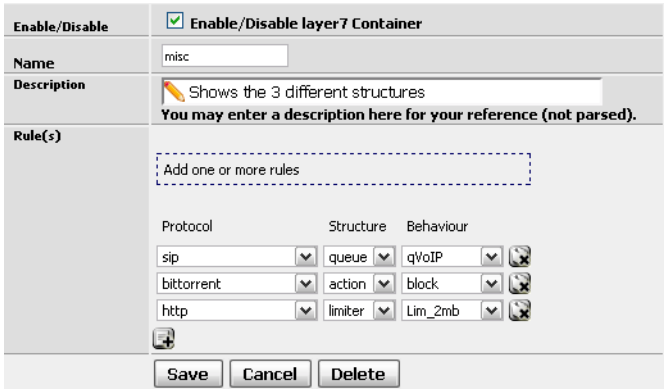

Fig. 9. Miscellaneous container creation

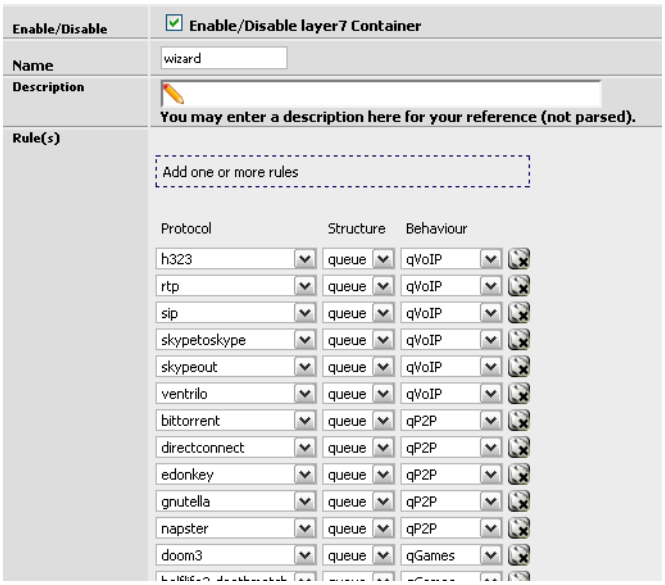

Fig. 10. Extract of wizard container creation

The pf rule responsible for diverting traffic to *ipfw-classifyd* is slightly different from the other ones. This is a directionless rule, that is automatically created by the wizard, and known in the pfSense terminology as a "Floating Rule". As the wizard only assigns AltQ queues to the different application protocols, only AltQ needs to be overloaded in the pf rule. The resulting pf rule for this L7 container is the following:

```
pass   out  proto { tcp udp }
from any to any divert 50476
keep state ( max-packets 5,
overload altq diverttag )
label "USER_RULE: Layer7 wizard"
```

Once again, the results are consistent with the envisioned system design, where this rule will try to enforce traffic shaping going through that particular pfSense box.

V. Conclusions

We consider our work to be a success! pfSense has now another shaping mechanism, that puts it on par with a great amount of commercial solutions, and it also has now a fully integrated GUI that allows the end user to easily leverage the layer 7 capabilities that *ipfw-classifyd* provides to the pfSense platform. This means that pfSense users are no longer limited to traffic shaping and classification only by IP packet fields of by a set of standard ports and is fully prepared to address the challenges that lie ahead, as the port hoping issue. The only current drawback in that *ipfw-classifyd* is not currently fully operational due to ongoing improvements. As soon as it is fully operational, all the necessary structure for it to work seamlessly in the pfSense platform is already built.

As future work, we think there is still some room for improvement. In particular, performing L7 inspection directly in kernel land would be very important and should be faced as a top goal. This would avoid the overhead introduced by the context switch between kernel and user land, that is necessary to divert IP packets from the kernel to *ipfw-classifyd* or to other application for that purpose. We also would like to point out that QoS auto-configuration could be an important improvement to the platform. Through auto-configuration, the platform could receive a set of input parameters (for example, the available bandwidth and the expected number of VoIP phones), and then could generate the required QoS parameters and provide feedback to the user about expected behavior for the service. In addition, self-configuration features could also be added in order the self-adapt the platform when new equipment is added to the infrastructure. As regards L7 protocol pattern files, a way to auto-detect application patterns [7], and automatically create application signatures would be an welcome add on.

Acknowledgments

The authors would like to thank to Ermal Luçi all the precious help he gave during the course of this work, specially, regarding ipfw-classifyd and pf. The authors also would like to thank Scott Ulrich and Chris Buechler, founders of pfSense, for their feedback and guidance during the course of the project, as well as, the active pfSense developers for their ideas and opinions.

References

[1] IPCop Firewall. URL: http://www.ipcop.org, 2003.
[2] Bandwidth Arbitrator. URL: http://www.bandwidtharbitrator.com/, 2002.
[3] pfSense Project. URL: http://www.pfsense.com/, 2004.
[4] K. Cho. Managing Traffic with ALTQ. In *USENIX 1999 Annual Technical Conference: FREENIX Track*, pages 121–128, Monterey, California, USA, June 1999.
[5] G. Quadros, A. Alves, E. Monteiro, and F. Boavida. How Unfair can Weighted Fair Queuing be? *IEEE Network*, 17, 2000.
[6] A. Seddik-Ghaleb, Y. Ghamri-Doudane, and S.-M. Senouci. Emulating End-to-End Losses and Delays for Ad Hoc Networks. *Communications, 2007. ICC apos; 07. IEEE International Conference*, 24-28:3224–3231, June 2007.
[7] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. *SIGCOMM'05*, August 2005.