OpenBSD PF 中文手册

PF:The OpenBSD Packet Filter

http://www.openbsd.org/faq/pf/

Power by xetex

2008.3.28

目 录

1 基本配置 **9**

1.1 开始 9

1.1.1 激活 9

1.1.2 配置 9

1.1.3 控制 10

1.2 列表与宏 10

1.2.1 列表 10

1.2.2 宏 11

1.3 表 11

1.3.1 简介 11

1.3.2 配置 12

1.3.3 用 pfctl 进行操作 12

1.3.4 指定地址 13

1.3.5 地址匹配 13

1.4 包过滤 13

1.4.1 简介 13

1.4.2 规则语法 14

1.4.3 默认拒绝 16

1.4.4 通过流量 16

1.4.5 quick 关键字 17

1.4.6 状态保持 17

1.4.7 UDP 状态保持 18

1.4.8 TCP 标记 18

1.4.9 TCP SYN 代理 20

1.4.10 阻塞欺骗数据包 20

1.4.11 被动操作系统识别 21

1.4.12 IP 选项 22

1.4.13 过滤规则实例 22

1.5 网络地址转换(NAT) 23

1.5.1 简介 . . . 23
1.5.2 NAT 如何工作 . . . 23
1.5.3 NAT 和包过滤 . . . 24
1.5.4 IP 转发 . . . 24
1.5.5 配置 NAT . . . 24
1.5.6 双向映射(1:1 映射) . . . 27
1.5.7 转换规则例外设置 . . . 27
1.5.8 检查 NAT 状态 . . . 28
1.6 重定向(端口转发) . . . 28
1.6.1 简介 . . . 28
1.6.2 重定向和包过滤 . . . 29
1.6.3 安全隐患 . . . 30
1.6.4 重定向和反射 . . . 30
1.6.5 TCP 代理 . . . 31
1.6.6 RDR 和 NAT 结合 . . . 31
1.7 规则生成捷径 . . . 31
1.7.1 简介 . . . 31
1.7.2 使用宏 . . . 32
1.7.3 使用列表 . . . 32
1.7.4 PF 语法 . . . 34
1.7.5 减少关键字 . . . 34
1.7.6 Return 简化 . . . 34
1.7.7 关键字顺序 . . . 35

2 高级配置 **37**
2.1 运行选项 . . . 37
2.1.1 set block-policy . . . 37
2.1.2 set debug . . . 37
2.1.3 set fingerprints file . . . 37
2.1.4 set limit . . . 37
2.1.5 set loginterface int . . . 38
2.1.6 set optimization . . . 38
2.1.7 set state-policy . . . 38
2.1.8 set timeout . . . 38
2.1.9 例子 . . . 38
2.2 流量整形(数据包标准化) . . . 39
2.2.1 简介 . . . 39

2.2.2 选项 . . . 39
2.3 锚定和命名规则集 . . . 40
2.3.1 简介 . . . 40
2.3.2 命名规则集 . . . 40
2.3.3 锚定选项 . . . 42
2.3.4 操作命名规则集 . . . 42
2.4 队列和优先级 . . . 42
2.4.1 队列 . . . 42
2.4.2 日程 . . . 43
2.4.3 基于类的队列 . . . 43
2.4.4 优先级队列 . . . 44
2.4.5 随机早期检测 . . . 45
2.4.6 外部拥塞告知 . . . 45
2.4.7 配置队列 . . . 46
2.4.8 为队列分配数据流 . . . 47
2.4.9 实例 1: 小型家庭网络 . . . 48
2.4.10 实例 2: 公司网络 . . . 51
2.5 地址池和负载均衡 . . . 54
2.5.1 简介 . . . 54
2.5.2 NAT 地址池 . . . 55
2.5.3 外来连接负载均衡 . . . 55
2.5.4 输出流量负载均衡 . . . 55
2.6 数据包标记 . . . 57
2.6.1 简介 . . . 57
2.6.2 给数据包打标记 . . . 57
2.6.3 检查数据包标记 . . . 59
2.6.4 策略过滤 . . . 59
2.6.5 标记以太网帧 . . . 60

3 附加主题 63
3.1 日志 . . . 63
3.1.1 简介 . . . 63
3.1.2 读取日志文件 . . . 63
3.1.3 导出日志 . . . 63
3.1.4 通过 Syslog 记录日志 . . . 64
3.2 性能 . . . 66
3.3 研究 FTP . . . 67

3.3.1 FTP 模式 67
3.3.2 工作在防火墙之后的 FTP 客户端 67
3.3.3 PF”自保护”FTP 服务器 68
3.3.4 使用 NAT 外部 PF 防火墙保护 FTP 服务器 68
3.4 pf 验证: 用 Shell 进行网关验证 69
3.4.1 简介 69
3.4.2 配置 69
3.4.3 配置加载的策略 70
3.4.4 访问控制列表 70
3.4.5 将 authpf 配置为用户 shell 70
3.4.6 查看登陆者 71
3.4.7 实例 71
3.5 实例：家庭和小型办公室防火墙 72
3.5.1 概况 72
3.5.2 网络 72
3.5.3 目标 73
3.5.4 准备 73
3.5.5 规则集 73
3.5.6 完整规则集 76

包过滤(以下简称 PF) 是 OpenBSD 系统上进行 TCP/IP 流量过滤和网络地址转换的软件系统。PF 同样也能提供 TCP/IP 流量的整形和控制，并且提供带宽控制和数据包优先集控制。PF 自 openbsd 3.0 以后作为内核的默认安装配置。以前版本的 openbsd 发行版使用一个不同的防火墙/NAT 软件包，现在已经不再被支持。

PF 最早是由 Daniel Hartmeier 开发的，现在的开发和维护由 Daniel 和 openbsd 小组的其他成员负责。

本文档对 openbsd 上运行的 PF 系统做一个简明的简介。本文可被用作是 man 手册页的补充，而不是它们的替代。本文档涵盖了 PF 的主要特性。要完整和深入的了解 PF 能做什么，请阅读 pf（4）man 手册页。

本 FAQ 文档是针对使用 openbsd 3.5 的用户的。由于 pf 是不断成长和开发中的，3.5 版本和当前版本之间存在软件的变化和功能增强。建议读者阅读正在运行系统的 man 手册页。

§1 基本配置

1.1 开始

1.1.1 激活

要激活 pf 并且使它在启动时调用配置文件，编辑/etc/rc.conf 文件，修改配置 pf 的一行：

```
pf=YES
```

重启系统让配置生效。你也可以通过 pfctl 程序启动和停止 pf

```
# pfctl -e
# pfctl -d
```

注意这仅仅是启动和关闭 PF，实际它不会载入规则集，规则集要么在系统启动时载入，要么在 PF 启动后通过命令单独载入。

1.1.2 配置

系统引导到在 rc 脚本文件运行 PF 时 PF 从/etc/pf.conf 文件载入配置规则。注意当/etc/pf.conf 文件是默认配置文件，在系统调用 rc 脚本文件时，它仅仅是作为文本文件由 pfctl（8）装入并解释和插入 pf（4）的。对于一些应用来说，其他的规则集可以在系统引导后由其他文件载入。对于一些设计的非常好的 unix 程序，PF 提供了足够的灵活性。

pf.conf 文件有 7 个部分:

1. 宏:用户定义的变量，包括 IP 地址，接口名称等等
2. 表: 一种用来保存 IP 地址列表的结构
3. 选项: 控制 PF 如何工作的变量
4. 整形: 重新处理数据包，进行正常化和碎片整理
5. 排队: 提供带宽控制和数据包优先级控制.
6. 转换: 控制网络地址转换和数据包重定向.

7. 过滤规则: 在数据包通过接口时允许进行选择性的过滤和阻止

除去宏和表，其他的段在配置文件中也应该按照这个顺序出现，尽管对于一些特定的应用并不是所有的段都是必须的。空行会被忽略，以 # 开头的行被认为是注释。

1.1.3 控制

引导之后，PF 可以通过 pfctl（8）程序进行操作，以下是一些例子：

```
# pfctl -f /etc/pf.conf 载入 pf.conf 文件
# pfctl -nf /etc/pf.conf 解析文件，但不载入
# pfctl -Nf /etc/pf.conf 只载入文件中的 NAT 规则
# pfctl -Rf /etc/pf.conf 只载入文件中的过滤规则

# pfctl -sn 显示当前的 NAT 规则
# pfctl -sr 显示当前的过滤规则
# pfctl -ss 显示当前的状态表
# pfctl -si 显示过滤状态和计数
# pfctl -sa 显示任何可显示的
```

完整的命令列表，参阅 pfctl 的 man 手册页。

1.2 列表与宏

1.2.1 列表

一个列表允许一个规则集中指定多个相似的标准。例如，多个协议，端口号，地址等等。因此，不需要为每一个需要阻止的 IP 地址编写一个过滤规则，一条规则可以在列表中指定多个 IP 地址。列表的定义是将要指定的条目放在 大括号中。

当 pfctl（8）在载入规则集碰到列表时，它产生多个规则，每条规则对于列表中的一个条目。例如：

```
block out on fxp0 from { 192.168.0.1, 10.5.32.6 } to any
```

展开后:

```
block out on fxp0 from 192.168.0.1 to any
block out on fxp0 from 10.5.32.6 to any
```

多种列表可以在规则中使用，并不仅仅限于过滤规则：

```
rdr on fxp0 proto tcp from any to any port { 22 80 } \
   -> 192.168.0.6
block out on fxp0 proto { tcp udp } from \
```

```
{ 192.168.0.1, 10.5.32.6 } to any port { ssh telnet }
```

注意逗号在列表条目之间是可有可无的。

1.2.2 宏

宏是用户定义变量用来指定 IP 地址，端口号，接口名称等等。宏可以降低 PF 规则集的复杂度并且使得维护规则集变得容易。

宏名称必须以字母开头，可以包括字母，数字和下划线。宏名称不能包括保留关键字如：pass, out, 以及 queue.

```
ext_if = "fxp0"
block in on $ext_if from any to any
```

这生成了一个宏名称为 ext_if. 当一个宏在它产生以后被引用时，它得名称前面以$字符开头。

宏也可以展开成列表，如：

```
friends = "{ 192.168.1.1, 10.0.2.5, 192.168.43.53 }"
```

宏能够被重复定义，由于宏不能在引号内被扩展，因此必须使用下面得语法：

```
host1 = "192.168.1.1"
host2 = "192.168.1.2"
all_hosts = "{" $host1 $host2 "}"
```

宏$all_hosts 现在会展开成 192.168.1.1, 192.168.1.2.

1.3 表

1.3.1 简介

表是用来保存一组 IPv4 或者 IPv6 地址。在表中进行查询是非常快的，并且比列表消耗更少的内存和 cpu 时间。由于这个原因，表是保存大量地址的最好方法，在 50，000 个地址中查询仅比在 50 个地址中查询稍微多一点时间。表可以用于下列用途：

1. 过滤，整形，NAT 和重定向中的源或者目的地址.
2. NAT 规则中的转换地址.
3. 重定向规则中的重定向地址.
4. 过滤规则选项中 route-to, reply-to, 和 dup-to 的目的地址.

表可以通过在 pf.conf 里配置和使用 pfctl 生成。

1.3.2 配置

在 pf.conf 文件中, 表是使用 table 关键字创建出来的。下面的关键字必须在创建表时指定。

1. constant - 这类表的内容一旦创建出来就不能被改变。如果这个属性没有指定，可以使用 pfctl（8）添加和删除表里得地址，即使系统是运行在 2 或者更高得安全级别上。
2. persist - 即使没有规则引用这类表，内核也会把它保留在内存中。如果这个属性没有指定，当最后引用它的规则被取消后内核自动把它移出内存。

实例:

```
table <goodguys> { 192.0.2.0/24 }
table <rfc1918> const { 192.168.0.0/16, 172.16.0.0/12, \
  10.0.0.0/8 }
table <spammers> persist

block in on fxp0 from { <rfc1918>, <spammers> } to any
pass in on fxp0 from <goodguys> to any
```

地址也可以用“非”来进行修改，如：

```
table <goodguys> { 192.0.2.0/24, !192.0.2.5 }
```

goodguy 表将匹配除 192.0.2.5 外 192.0.2.0/24 网段得所有地址。
注意表名总是在<>符号得里面。
表也可以由包含 IP 地址和网络地址的文本文件中输入：

```
table <spammers> persist file "/etc/spammers"
block in on fxp0 from <spammers> to any
```

文件/etc/spammers 应该包含被阻塞的 IP 地址或者 CIDR 网络地址，每个条目一行。以 # 开头的行被认为是注释，会被忽略。

1.3.3 用 pfctl 进行操作

表可以使用 pfctl（8）进行灵活的操作。例如，在上面产生的<spammer>表中增加条目可以这样写：

```
#pfctl -t spammers -T add 218.70.0.0/16
```

如果<spammer>这个表不存在，这样会创建出这个表来。列出表中的内容可以这样：

```
#pfctl -t spammers -T show
```

-v 参数也可以使用-Tshow 来显示每个表的条目内容统计。要从表中删除条目，可以这样：

```
#pfctl -t spammers -T delete 218.70.0.0/16
```

更多使用 pfctl 操作的信息可以参阅 pfctl（8）。

1.3.4 指定地址

除了使用 IP 地址来指定主机外，也可以使用主机名。当主机名被解析成 IP 地址时，IPv4 和 IPv6 地址都被插进规则中。IP 地址也可以通过合法的接口名称或者 self 关键字输入表中，这样的表会分别包含接口或者机器上（包括 loopback 地址）上配置的所有 IP 地址。

一个限制是指定地址 0.0.0.0/0 以及 0/0 在表中不能工作。替代方法是明确输入该地址或者使用宏。

1.3.5 地址匹配

表中的地址查询会匹配最接近的规则，比如：

```
table <goodguys> { 172.16.0.0/16, !172.16.1.0/24, 172.16.1.100 }
block in on dc0 all
pass in on dc0 from <goodguys> to any
```

任何自 dc0 上数据包都会把它的源地址和 goodguys 表中的地址进行匹配：

1. 172.16.50.5 - 精确匹配 172.16.0.0/16; 数据包符合可以通过
2. 172.16.1.25 - 精确匹配!172.16.1.0/24; 数据包匹配表中的一条规则，但规则是“非”（使用“!”进行了修改）；数据包不匹配表会被阻塞。
3. 172.16.1.100 - 准确匹配 172.16.1.100; 数据包匹配表，运行通过
4. 10.1.4.55 - 不匹配表，阻塞。

1.4 包过滤

1.4.1 简介

包过滤是在数据包通过网络接口时进行选择性的运行通过或者阻塞。Pf（4）检查包时使用的标准是基于的 3 层（IPV4 或者 IPV6）和 4 层（TCP, UDP, ICMP, ICMPv6）包头。最常用的标准是源和目的地址，源和目的端口，以及协议。

过滤规则集指定了数据包必须匹配的标准和规则集作用后的结果，在规则集匹配时通过或者阻塞。规则集由开始到结束顺序执行。除非数据包匹配的规则包含 quick 关键字，否则数据包在最终执行动作前会通过所有的规则检验。最后匹配的规则具有决定性，决定了数据包最终的执行结果。存在一条潜在的规则是如果数据包和规则集中的所有规则都不匹配，则它会被通过。

1.4.2 规则语法

一般而言，最简单的过滤规则语法是这样的：

```
action direction [log] [quick] on interface [af] [proto protocol] \
from src_addr [port src_port] to dst_addr [port dst_port] \
[tcp_flags] [state]
```

- action:数据包匹配规则时执行的动作，放行或者阻塞。放行动作把数据包传递给核心进行进一步出来，阻塞动作根据 block-policy 选项指定的方法进行处理。默认的动作可以修改为阻塞丢弃或者阻塞返回。
- direction 数据包传递的方向，进或者出
- log 指定数据包被 pflogd(8)进行日志记录。如果规则指定了 keep state, modulate state, or synproxy state 选项，则只有建立了连接的状态被日志。要记录所有的日志，使用 log-all
- quick 如果数据包匹配的规则指定了 quick 关键字，则这条规则被认为时最终的匹配规则，指定的动作会立即执行。
- interface 数据包通过的网络接口的名称或组。组是接口的名称但没有最后的整数。比如 ppp 或 fxp，会使得规则分别匹配任何 ppp 或者 fxp 接口上的任意数据包。
- af 数据包的地址类型，inet 代表 Ipv4，inet6 代表 Ipv6。通常 PF 能够根据源或者目标地址自动确定这个参数。
- protocol 数据包的 4 层协议:
  - tcp
  - udp
  - icmp
  - icmp6
  - /etc/protocols 中的协议名称
  - 0 ～ 255 之间的协议号
  - 使用列表的一系列协议.
- src_addr, dst_addr IP 头中的源/目标地址。地址可以指定为：

- 单个的 Ipv4 或者 Ipv6 地址.
- CIDR 网络地址.
- 能够在规则集载入时通过 DNS 解析到的合法的域名，IP 地址会替代规则中的域名。
- 网络接口名称。网络接口上配置的所有 ip 地址会替代进规则中。
- 带有/掩码（例如/24）的网络接口的名称。每个根据掩码确定的 CIDR 网络地址都会被替代进规则中。
- 带有（）的网络接口名称。这告诉 PF 如果网络接口的 IP 地址改变了，就更新规则集。这个对于使用 DHCP 或者拨号来获得 IP 地址的接口特别有用，IP 地址改变时不需要重新载入规则集。
- 带有如下的修饰词的网络接口名称：

network 替代 CIDR 网络地址段(例如, 192.168.0.0/24)

broadcast 替代网络广播地址(例如，192.168.0.255)

peer 替代点到点链路上的 ip 地址。

另外，:0 修饰词可以附加到接口名称或者上面的修饰词后面指示 PF 在替代时不包括网络接口的其余附加（alias）地址。这些修饰词也可以在接口名称在括号（）内时使用。例如：fxp0:network:0

- 表.
- 上面的所有项但使用!（非）修饰词
- 使用列表的一系列地址.
- 关键字 any 代表所有地址(关键字 all 是 from any to any 的缩写)。

• src_port, dst_port 4 层数据包头中的源/目标端口。端口可以指定为：

- 1 到 65535 之间的整数
- /etc/services 中的合法服务名称
- 使用列表的一系列端口
- 一个范围:
  + != (不等于)
  + < (小于)
  + > (大于)
  + <= (小于等于)
  + >= (大于等于)
  + >< (范围) 元操作符,需要 2 个参数，在范围内不包括参数。
  + <> (反转范围) 元操作符,需要 2 个参数，在范围内不包括参数。
  + (inclusive range) 也是二元操作符但范围内包括参数。

- tcp_flags 指定使用 TCP 协议时 TCP 头中必须设定的标记。标记指定的格式是：flags check/mask. 例如: flags S/SA -这指引 PF 只检查 S 和 A(SYN and ACK)标记，如果 SYN 标记是“on”则匹配。
- state 指定状态信息在规则匹配时是否保持。
  - keep state - 对 TCP, UDP, ICMP 起作用
  - modulate state - 只对 TCP 起作用. PF 会为匹配规则的数据包产生强壮的初始化序列号。
  - synproxy state - 代理外来的 TCP 连接以保护服务器不受 TCP SYN FLOODs 欺骗。这个选项包含了 keep state 和 modulate state 的功能。

1.4.3 默认拒绝

按照惯例建立防火墙时推荐执行的是默认拒绝的方法。也就是说先拒绝所有的东西，然后有选择的允许某些特定的流量通过防火墙。这个方法之所以是推荐的是因为它宁可失之过于谨慎（也不放过任何风险），而且使得编写规则集变得简单。

产生一个默认拒绝的过滤规则，开始 2 行过滤规则必须是：

```
block in all
block out all
```

这会阻塞任何通信方在任何方向上进入任意接口的所有流量。

1.4.4 通过流量

流量必须被明确的允许通过防火墙或者被默认拒绝的策略丢弃。这是数据包标准如源/目的端口，源/目的地址和协议开始活动的地方。无论何时数据包在被允许通过防火墙时规则都要设计的尽可能严厉。这是为了保证设计中的流量，也只有设计中的流量可以被允许通过。

实例:

```
# 允许本地网络 192.168.0.0/24 流量通过 dc0 接口进入访问 openbsd 机器的 IP 地址
#192.168.0.1，同时也允许返回的数据包从 dc0 接口出去。
pass in on dc0 from 192.168.0.0/24 to 192.168.0.1
pass out on dc0 from 192.168.0.1 to 192.168.0.0/24

# Pass TCP traffic in on fxp0 to the web server running on the
# OpenBSD machine. The interface name, fxp0, is used as the
# destination address so that packets will only match this rule if
# they ‘re destined for the OpenBSD machine.
pass in on fxp0 proto tcp from any to fxp0 port www
```

1.4.5 quick 关键字

前面已经说过，每个数据包都要按自上至下的顺序按规则进行过滤。默认情况下，数据包被标记为通过，这个可以被任一规则改变，在到达最后一条规则前可以被来回改变多次，最后的匹配规则是“获胜者”。存在一个例外是：过滤规则中的 quick 关键字具有取消进一步往下处理的作用，使得规则指定的动作马上执行。看一下下面的例子：

错误:

```
block in on fxp0 proto tcp from any to any port ssh
pass in all
```

在这样的条件下，block 行会被检测，但永远也不会有效果，因为它后面的一行允许所有的流量通过。

正确:

```
block in quick on fxp0 proto tcp from any to any port ssh
pass in all
```

这些规则执行的结果稍有不同，如果 block 行被匹配，由于 quick 选项的原因，数据包会被阻塞，而且剩下的规则也会被忽略。

1.4.6 状态保持

PF 一个非常重要的功能是“状态保持”或者“状态检测”。状态检测指 PF 跟踪或者处理网络连接状态的能力。通过存贮每个连接的信息到一个状态表中，PF 能够快速确定一个通过防火墙的数据包是否属于已经建立的连接。如果是，它会直接通过防火墙而不用再进行规则检验。

状态保持有许多的优点，包括简单的规则集和优良的数据包处理性能。PF is able to match packets moving in either direction to state table entries meaning that filter rules which pass returning traffic don‘t need to be written. 并且，由于数据包匹配状态连接时不再进行规则集的匹配检测，PF 用于处理这些数据包的时间大为减少。

当一条规则使用了 keep state 选项，第一个匹配这条规则的数据包在收发双方之间建立了一个状态。现在，不仅发送者到接收者之间的数据包匹配这个状态绕过规则检验，而且接收者回复发送者的数据包也是同样的。例如：

```
pass out on fxp0 proto tcp from any to any keep state
```

这允许 fxp0 接口上的任何 TCP 流量通过，并且允许返回的流量通过防火墙。状态保持是一个非常有用的特性，由于状态查询比使用规则进行数据包检验快的多，因此它可以大幅度提高防火墙的性能。

状态调整选项和状态保持的功能在除了仅适用于 TCP 数据包以为完全相同。在使用状态调整时，输入连接的初始化序列号（ISN）是随机的，这对于保护某些选择 ISN 存在问题的操作系统的连接初始化非常有用。从 openbsd 3.5 开始，状态调整选项可以应用于包含非 TCP 的协议规则。

对输出的 TCP, UDP, ICMP 数据包保持状态，并且调整 TCP ISN:

```
pass out on fxp0 proto { tcp, udp, icmp } from any \
to any modulate state
```

状态保持的另一个优点是 ICMP 通信流量可以直接通过防火墙。例如，如果一个 TCP 连接使用了状态保持，当和这个 TCP 连接相关的 ICMP 数据包到来时，它会自动找到合适的状态记录，直接通过防火墙。

状态记录的范围被 state-policy runtime 选项总体控制，也能基于单条规则由 if-bound, group-bound, 和 floating state 选项关键字设定。这些针对单条规则的关键字在使用时具有和 state-policy 选项同样的意义。例如：

```
pass out on fxp0 proto { tcp, udp, icmp } from any \
 to any modulate state (if-bound)
```

状态规则指示为了使数据包匹配状态条目，它们必须通过 fxp0 网络接口传递。

需要注意的是，nat，binat，rdr 规则隐含在连接通过过滤规则集审核的过程中产生匹配连接的状态。

1.4.7 UDP 状态保持

大家都听说过，“不能为 UDP 产生状态，因为 UDP 是无状态的协议”。确实，UDP 通信会话没有状态的概念（明确的开始和结束通信），这丝毫不影响 PF 为 UDP 会话产生状态的能力。对于没有开始和结束数据包的协议，PF 仅简单追踪匹配的数据部通过的时间。如果到达超时限制，状态被清除，超时的时间值可以在 pf.conf 配置文件中设定。

1.4.8 TCP 标记

基于标记的 TCP 包匹配经常被用于过滤试图打开新连接的 TCP 数据包。TCP 标记和他们的意义如下所列：

- F : FIN - 结束; 结束会话
- S : SYN - 同步; 表示开始会话请求
- R : RST - 复位;中断一个连接
- P : PUSH - 推送; 数据包立即发送
- A : ACK - 应答
- U : URG - 紧急
- E : ECE - 显式拥塞提醒回应
- W : CWR - 拥塞窗口减少

要使 PF 在规则检查过程中检查 TCP 标记，flag 关键需按如下语法设置。

```
flags check/mask
```

mask 部分告诉 PF 仅检查指定的标记，check 部分说明在数据包头中哪个标记设置为“on”才算匹配。

```
pass in on fxp0 proto tcp from any to any port ssh flags S/SA
```

上面的规则通过的带 SYN 标记的 TCP 流量仅查看 SYN 和 ACK 标记。带有 SYN 和 ECE 标记的数据包会匹配上面的规则，而带有 SYN 和 ACK 的数据包或者仅带有 ACK 的数据包不会匹配。

注意：在前面的 openbsd 版本中，下面的语法是支持的：

```
. . . flags S
```

现在，这个不再支持，mask 必须被说明。

标记常常和状态保持规则联合使用来控制创建状态条目：

```
pass out on fxp0 proto tcp all flags S/SA keep state
```

这条规则允许为所有输出中带 SYN 和 ACK 标记的数据包中的仅带有 SYN 标记的 TCP 数据包创建状态。

大家使用标记时必须小心，理解你在做什么和为什么这样做。小心听取大家的建议，因为相当多的建议时不好的。一些人建议创建状态“只有当 SYN 标记设定而没有其他标记”时，这样的规则如下：

```
. . . flags S/FSRPAUEW
```

糟糕的主意!!

这个理论是，仅为 TCP 开始会话时创建状态，会话会以 SYN 标记开始，而没有其他标记。问题在于一些站点使用 ECN 标记开始会话，而任何使用 ECN 连接你的会话都会被那样的规则拒绝。比较好的规则是：

```
. . . flags S/SAFR
```

这个经过实践是安全的，如果流量进行了整形也没有必要检查 FIN 和 RST 标记。整形过程会让 PF 丢弃带有非法 TCP 标记的进入数据包（例如 SYN 和 FIN 以及 SYN 和 RST）。强烈推荐总是进行流量整形：

```
scrub in on fxp0
.
.
.
pass in on fxp0 proto tcp from any to any port ssh flags S/SA \
  keep state
```

1.4.9 TCP SYN 代理

TCP SYN 代理通过当客户端向服务器初始化一个 TCP 连接时，PF 会在二者直接传递握手数据包。然而，PF 具有这样的能力，就是代理握手。使用握手代理，PF 自己会和客户端完成握手，初始化和服务器的握手，然后在二者之间传递数据。这样做的优点是在客户端完成握手之前，没有数据包到达服务器。这样就消沉了 TCP SYN FLOOD 欺骗影响服务器的问题，因为进行欺骗的客户端不会完成握手。

TCP SYN 代理在规则中使用 synproxy state 关键字打开。例如：

```
pass in on $ext_if proto tcp from any to $web_server port www \
 flags S/SA synproxy state
```

这样，web 服务器的连接由 PF 进行 TCP 代理。由于 synproxy state 工作的方式，它具有 keep state 和 modulate state 一样的功能。

如果 PF 工作在桥模式下，SYN 代理不会起作用。

1.4.10 阻塞欺骗数据包

地址欺骗是恶意用户为了隐藏他们的真实地址或者假冒网络上的其他节点而在他们传递的数据包中使用虚假的地址。一旦实施了地址欺骗，他们可以隐蔽真实地址实施网络攻击或者获得仅限于某些地址的网络访问服务。

PF 通过 antispoof 关键字提供一些防止地址欺骗的保护。

```
antispoof [log] [quick] for interface [af]
```

- log 指定匹配的数据包应该被 pflogd（8）进行日志记录
- quick 如果数据包匹配这条规则，则这是最终的规则，不再进行其他规则集的检查。
- interface 激活要进行欺骗保护的网络接口。也可以是接口的列表。
- af 激活进行欺骗保护的地址族，inet 代表 Ipv4，inet6 代表 Ipv6。

实例:

```
antispoof for fxp0 inet
```

当规则集载入时，任何出现了 antispoof 关键字的规则都会扩展成 2 条规则。假定接口 fxp0 具有 ip 地址 10.0.0.1 和子网掩码 255.255.255.0（或者/24），上面的规则会扩展成：

```
block in on ! fxp0 inet from 10.0.0.0/24 to any
block in inet from 10.0.0.1 to any
```

这些规则实现下面的 2 个目的：

1. 阻塞任何不是由 fxp0 接口进入的 10.0.0.0/24 网络的流量。由于 10.0.0.0/24 的网络是在 fxp0 接口，具有这样的源网络地址的数据包绝不应该从其他接口上出现。

2. 阻塞任何由 10.0.0.1 即 fxp0 接口的 IP 地址的进入流量。主机绝对不会通过外面的接口给自己发送数据包，因此任何进入的流量源中带有主机自己的 IP 地址都可以认为是恶意的!

注意：antispoof 规则扩展出来的过滤规则会阻塞 loopback 接口上发送到本地地址的数据包。这些数据包应该明确的配置为允许通过。例如：

```
pass quick on lo0 all
antispoof for fxp0 inet
```

使用 antispoof 应该仅限于已经分配了 IP 地址的网络接口，如果在没有分配 IP 地址的网络接口上使用，过滤规则会扩展成：

```
block drop in on ! fxp0 inet all
block drop in inet all
```

这样的规则会存在阻塞所有接口上进入的所有流量的危险。

1.4.11 被动操作系统识别

被动操作系统识别是通过基于远端主机 TCP SYN 数据包中某些特征进行操作系统被动检测的技术。这些信息可以作为标准在过滤规则中使用。

PF 检测远端操作系统是通过比较 TCP SYN 数据包中的特征和已知的特征文件对照来确定的，特征文件默认是/etc/pf.os。如果 PF 起作用，可是使用下面的命令查看当前的特征列表。

```
# pfctl -s osfp
```

在规则集中，特征可以指定为 OS 类型，版本，或者子类型/补丁级别。这些条目在上面的 pfctl 命令中有列表。要在过滤规则中指定特征，需使用 os 关键字：

```
pass in on $ext_if any os OpenBSD keep state
block in on $ext_if any os "Windows 2000"
block in on $ext_if any os "Linux 2.4 ts"
block in on $ext_if any os unknown
```

指定的操作系统类型 unknow 允许匹配操作系统未知的数据包。

注意以下的内容:

1. 操作系统识别偶尔会出错，因为存在欺骗或者修改过的使得看起来象某个操作系统得数据包。

2. 某些修改或者打过补丁得操作系统会改变栈得行为，导致它或者和特征文件不符，或者符合了另外得操作系统特征。

3. OSFP 仅对 TCP SYN 数据包起作用，它不会对其他协议或者已经建立得连接起作用。

1.4.12 IP 选项

默认情况下，PF 阻塞带有 IP 选项得数据包。这可以使得类似 nmap 得操作系统识别软件工作困难。如果你的应用程序需要通过这样的数据包，例如多播或者 IGMP，你可以使用 allow-opts 关键字。。

```
pass in quick on fxp0 all allow-opts
```

1.4.13 过滤规则实例

下面是过滤规则得实例。运行 PF 的机器充当防火墙，在一个小的内部网络和因特网之间。只列出了过滤规则，queueing, nat, rdr,等等没有在实例中列出。

```
ext_if = "fxp0"
int_if = "dc0"
lan_net = "192.168.0.0/24"

# scrub incoming packets
scrub in all

# setup a default deny policy
block in all
block out all

# pass traffic on the loopback interface in either direction
pass quick on lo0 all

# activate spoofing protection for the internal interface.
antispoof quick for $int_if inet

# only allow ssh connections from the local network if it ‘s from the
# trusted computer, 192.168.0.15. use "block return" so that a TCP RST is
# sent to close blocked connections right away. use "quick" so that this
# rule is not overridden by the "pass" rules below.
block return in quick on $int_if proto tcp from ! 192.168.0.15 \
to $int_if port ssh flags S/SA

# pass all traffic to and from the local network
pass in on $int_if from $lan_net to any
pass out on $int_if from any to $lan_net

# pass tcp, udp, and icmp out on the external (Internet) interface.
```

```
# keep state on udp and icmp and modulate state on tcp.
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state

# allow ssh connections in on the external interface as long as they‘re
# NOT destined for the firewall (i.e., they‘re destined for a machine on
# the local network). log the initial packet so that we can later tell
# who is trying to connect. use the tcp syn proxy to proxy the connection.
pass in log on $ext_if proto tcp from any to { !$ext_if, !$int_if } \
port ssh flags S/SA synproxy state
```

1.5 网络地址转换(NAT)

1.5.1 简介

网络地址转换是映射整个网络（或者多个网络）到单个 IP 地址的方法。当你的 ISP 分配给你的 IP 地址数目少于你要连上互联网的计算机数目时，NAT 是必需的。NAT 在 RFC1631 中描述。

NAT 允许使用 RFC1918 中定义的保留地址族。典型情况下，你的内部网络可以下面的地址族：

- 10.0.0.0/8 (10.0.0.0 - 10.255.255.255)
- 172.16.0.0/12 (172.16.0.0 - 172.31.255.255)
- 192.168.0.0/16 (192.168.0.0 - 192.168.255.255)

担任 NAT 任务的 openbsd 系统必须至少要 2 块网卡，一块连接到因特网，另一块连接内部网络。NAT 会转换内部网络的所有请求，使它们看起来象是来自进行 NAT 工作的 openbsd 主机系统。

1.5.2 NAT 如何工作

当内部网络的一个客户端连接因特网上的主机时，它发送目的是那台主机的 IP 数据包。这些数据包包含了达到连接目的的所有信息。NAT 的作用和这些信息相关。

- 源 IP 地址(例如, 192.168.1.35)
- 源 TCP 或者 UDP 端口(例如, 2132)

当数据包通过 NAT 网关时，它们会被修改，使得数据包看起来象是从 NAT 网关自己发出的。NAT 网关会在它的状态表中记录这个改变，以便将返回的数据包反向转换，确保返回的数据包能通过防火墙而不会被阻塞。例如，会发生下面的改变：

- 源 IP: 被网关的外部地址所替换(例如，24.5.0.5)
- 源端口:被随机选择的网关没有在用的端口替换(例如，53136)

内部主机和因特网上的主机都不会意识到发生了这个转变步骤。对于内部主机，NAT 系统仅仅是个因特网网关，对于因特网上的主机，数据包看起来直接来自 NAT 系统，它完全不会意识到内部工作站的存在。

当因特网网上的主机回应内部主机的数据包时，它会使用 NAT 网关机器的外部地址和转换后的端口。然后 NAT 网关会查询状态表来确定返回的数据包是否匹配某个已经建立的连接。基于 IP 地址和端口的联合找到唯一匹配的记录告诉 PF 这个返回的数据包属于内部主机 192.168.1.35。然后 PF 会进行和出去的数据包相反的转换过程，将返回的数据包传递给内部主机。

ICMP 数据包的转换也是类似的，只是不进行源端口的修改。

1.5.3 NAT 和包过滤

注意：转换后的数据包仍然会通过过滤引擎，根据定义的过滤规则进行阻塞或者通过。唯一的例外是如果 nat 规则中使用了 pass 关键字，会使得经过 nat 的数据包直接通过过滤引擎。

还要注意由于转换是在过滤之前进行，过滤引擎所看到的是上面“nat 如何工作”中所说的经过转换后的 ip 地址和端口的数据包。

1.5.4 IP 转发

由于 NAT 经常在路由器和网关上使用，因此 IP 转发是需要的，使得数据包可以在 openbsd 机器的不同网络接口间传递。IP 转发可以通过 sysctl（3）命令打开：

```
# sysctl -w net.inet.ip.forwarding=1
# sysctl -w net.inet6.ip6.forwarding=1 (if using IPv6)
```

要使这个变化永久生效，可以增加如下行到/etc/sysctl.conf 文件中：

```
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

这些行是本来就存在的，但默认安装中被 # 前缀注释掉了。删除 #，保存文件，IP 转发在机器重启后就会发生作用。

1.5.5 配置 NAT

一般的 NAT 规则格式在 pf.conf 文件中是这个样子：

```
nat [pass] on interface [af] from src_addr [port src_port] to \
 dst_addr [port dst_port] -> ext_addr [pool_type] [static-port]
```

- nat 开始 NAT 规则的关键字。
- pass 使得转换后的数据包完全绕过过滤规则。
- interface 进行数据包转换的网络接口。

- af 地址类型，inet 代表 Ipv4，inet6 代表 Ipv6。PF 通常能根据源/目标地址自动确定这个参数。
- src_addr 被进行转换的 IP 头中的源（内部）地址。源地址可以指定为：
  - 单个的 Ipv4 或者 Ipv6 地址.
  - CIDR 网络地址.
  - 能够在规则集载入时通过 DNS 解析到的合法的域名，IP 地址会替代规则中的域名。
  - 网络接口名称。网络接口上配置的所有 ip 地址会替代进规则中。
  - 带有/掩码（例如/24）的网络接口的名称。每个根据掩码确定的 CIDR 网络地址都会被替代进规则中。
  - 带有（）的网络接口名称。这告诉 PF 如果网络接口的 IP 地址改变了，就更新规则集。这个对于使用 DHCP 或者拨号来获得 IP 地址的接口特别有用，IP 地址改变时不需要重新载入规则集。
  - 带有如下的修饰词的网络接口名称：

network 替代 CIDR 网络地址段(例如, 192.168.0.0/24)

broadcast 替代网络广播地址(例如，192.168.0.255)

peer 替代点到点链路上的 ip 地址。

另外，:0 修饰词可以附加到接口名称或者上面的修饰词后面指示 PF 在替代时不包括网络接口的其余附加（alias）地址。这些修饰词也可以在接口名称在括号（）内时使用。例如：fxp0:network:0

  - 表.
  - 上面的所有项但使用!（非）修饰词
  - 使用列表的一系列地址.
  - 关键字 any 代表所有地址
  - 关键字 all 是 from any to any 的缩写。
- src_port 4 层数据包头中的源端口。端口可以指定为：
  - 1 到 65535 之间的整数
  - /etc/services 中的合法服务名称
  - 使用列表的一系列端口
  - 一个范围:
    + != (不等于)
    + < (小于)
    + > (大于)
    + <= (小于等于)
    + >= (大于等于)

+ >< (范围) 元操作符,需要 2 个参数，在范围内不包括参数。
+ <> (反转范围) 元操作符,需要 2 个参数，在范围内不包括参数。
+ (inclusive range) 也是二元操作符但范围内包括参数。

- Port 选项在 NAT 规则中通常不使用，因为目标通常会 NAT 所有的流量而不过端口是否在使用。
- dst_addr 被转换数据包中的目的地址。目的地址类型和源地址相似。
- dst_port 4 层数据包头中的目的目的端口，目的端口类型和源端口类型相似。
- ext_addr NAT 网关上数据包被转换后的外部地址。外部地址可以是：
  - 单个的 Ipv4 或者 Ipv6 地址.
  - CIDR 网络地址.
  - 能够在规则集载入时通过 DNS 解析到的合法的域名，IP 地址会替代规则中的域名。
  - 网络接口名称。网络接口上配置的所有 ip 地址会替代进规则中。
  - 带有/掩码（例如/24）的网络接口的名称。每个根据掩码确定的 CIDR 网络地址都会被替代进规则中。
  - 带有（）的网络接口名称。这告诉 PF 如果网络接口的 IP 地址改变了，就更新规则集。这个对于使用 DHCP 或者拨号来获得 IP 地址的接口特别有用，IP 地址改变时不需要重新载入规则集。
  - 带有如下的修饰词的网络接口名称：

  network 替代 CIDR 网络地址段(例如, 192.168.0.0/24)

  broadcast 替代网络广播地址(例如，192.168.0.255)

  peer 替代点到点链路上的 ip 地址。

  另外，:0 修饰词可以附加到接口名称或者上面的修饰词后面指示 PF 在替代时不包括网络接口的其余附加（alias）地址。这些修饰词也可以在接口名称在括号（）内时使用。例如：fxp0:network:0

  - 表.
  - 上面的所有项但使用!（非）修饰词
  - 使用列表的一系列地址.
- pool_type 指定转换后的地址池的类型
- static_port 告诉 PF 不要转换 TCP 和 UDP 数据包中的源端口

这条规则最简单的形式如下：

```
nat on tl0 from 192.168.1.0/24 to any -> 24.5.0.5
```

这条规则是说对从 tl0 网络接口上到来的所有 192.168.1.0/24 网络的数据包进行 NAT，将源地址转换为 24.5.0.5。

尽管上面的规则是正确的，但却不是推荐的形式。因为维护起来有困难，当内部或者外部网络有变化时都要修改这一行。比较一下下面这条比较容易维护的规则：（tl0 时外部，dc0 是内部）：

```
nat on tl0 from dc0:network to any -> tl0
```

优点是相当明显的，你可以任意改变 2 个接口的 IP 地址而不用改变这条规则。

象上面这样在地址转换中使用接口名称时，IP 地址在 pf.conf 文件载入时确定，并不是凭空的。如果你使用 DHCP 还配置外部地址，这会存在问题。如果你分配的 IP 地址改变了，NAT 仍然会使用旧的 IP 地址转换出去的数据包。这会导致对外的连接停止工作。为解决这个问题，应该给接口名称加上括号，告诉 PF 自动更新转换地址。

```
nat on tl0 from dc0:network to any -> (tl0)
```

这个方法对 IPv4 和 IPv6 地址都用效。

1.5.6 双向映射(1:1 映射)

双向映射可以通过使用 binat 规则建立。Binat 规则建立一个内部地址和外部地址一对一的映射。这会很有用，比如，使用独立的外部 IP 地址用内部网络里的机器提供 web 服务。从因特网到来连接外部地址的请求被转换到内部地址，同时由（内部）web 服务器发起的连接（例如 DNS 查询）被转换为外部地址。和 NAT 规则不过，binat 规则中的 tcp 和 udp 端口不会被修改。

例如:

```
web_serv_int = "192.168.1.100"
web_serv_ext = "24.5.0.6"

binat on tl0 from $web_serv_int to any -> $web_serv_ext
```

1.5.7 转换规则例外设置

使用 no 关键字可以在转换规则中设置例外。例如，如果上面的转换规则修改成这样：

```
no nat on tl0 from 192.168.1.10 to any
nat on tl0 from 192.168.1.0/24 to any -> 24.2.74.79
```

则除了 192.168.1.10 以外，整个 192.168.1.0/24 网络地址的数据包都会转换为外部地址 24.2.74.79。

注意第一条匹配的规则起了决定作用，如果是匹配有 no 的规则，数据包不会被转换。No 关键字也可以在 binat 和 rdr 规则中使用。

1.5.8 检查 NAT 状态

要检查活动的 NAT 转换可以使用 pfctl（8）带-s state 选项。这个选项列出所有当前的 NAT 会话。

```
# pfctl -s state
fxp0 TCP 192.168.1.35:2132->24.5.0.5:53136->65.42.33.245:22 TIME_WAIT:TIME_WAIT
fxp0 UDP 192.168.1.35:2491->24.5.0.5:60527->24.2.68.33:53 MULTIPLE:SINGLE
```

解释(对第一行):

- fxp0 显示状态绑定的接口。如果状态是浮动的，会出现 self 字样。
- TCP 连接使用的协议。
- 192.168.1.35:2132 内部网络中机器的 IP 地址(192.168.1.35)，源端口（2132）在地址后显示，这个也是被替换的 IP 头中的地址。of the machine on the internal network. The source port (2132) is shown after the address. This is also the address that is replaced in the IP header.
- 24.5.0.5:53136 IP 地址(24.5.0.5) 和端口(53136) 是网关上数据包被转换后的地址和端口。
- 65.42.33.245:22 IP 地址(65.42.33.245) 和端口(22) 是内部机器要连接的地址和端口。
- TIME_WAIT:TIME_WAIT 这表明 PF 认为的目前这个 TCP 连接的状态。

1.6 重定向(端口转发)

1.6.1 简介

如果在办公地点应用了 NAT，内部网所有的机器都可以访问因特网。但如何让 NAT 网关后面的机器能够被从外部访问？这就是重定向的来源。重定向允许外来的流量发送到 NAT 网关后面的机器。

看个例子：

```
rdr on tl0 proto tcp from any to any port 80 -> 192.168.1.20
```

这一行重定向了 TCP 端口 80（web 服务器）流量到内部网络地址 192.168.1.20。因此，即使 192.168.1.20 在网关后面的内部网络，外部仍然能够访问它。

上面 rdr 行中 from any to any 部分非常有用。如果你明确知道哪些地址或者子网被允许访问 web 服务器的 80 端口，你可以在这部分严格限制：

```
rdr on tl0 proto tcp from 27.146.49.0/24 to any port 80 -> \
 192.168.1.20
```

这样只会重定向指定的子网。注意这表明可以重定向外部不同的访问主机到网关后面不同的机器上。这非常有用。例如，如果你知道远端的用户连接上来时使用的 IP 地址，你可以让他们使用网关的 IP 地址和端口访问他们各自的桌面计算机。

```
rdr on tl0 proto tcp from 27.146.49.14 to any port 80 -> \
 192.168.1.20
rdr on tl0 proto tcp from 16.114.4.89 to any port 80 -> \
 192.168.1.22
rdr on tl0 proto tcp from 24.2.74.178 to any port 80 -> \
 192.168.1.23
```

1.6.2 重定向和包过滤

注意：转换后的数据包仍然会通过过滤引擎，根据定义的过滤规则进行阻塞或者通过。唯一的例外是如果 rdr 规则中使用了 pass 关键字，会使得重定向的数据包直接通过过滤引擎。

还要注意由于转换是在过滤之前进行，过滤引擎所看到的是在匹配 rdr 规则经过转换后的目标 ip 地址和端口的数据包。

192.0.2.1 - 因特网上的主机
24.65.1.13 - openbsd 路由器的外部地址
192.168.1.5 - web 服务器的内部 IP 地址

重定向规则:

```
rdr on tl0 proto tcp from 192.0.2.1 to 24.65.1.13 port 80 \
 -> 192.168.1.5 port 8000
```

数据包在经过 rdr 规则前的模样：

- 源地址: 192.0.2.1
- 源端口: 4028 (由操作系统任意选择)
- 目的地址: 24.65.1.13
- 目的端口: 80

数据包经过 rdr 规则后的模样：

- 源地址: 192.0.2.1
- 源端口: 4028
- 目的地址: 192.168.1.5
- 目的: 8000

过滤引擎看见的 IP 数据包时转换发生之后的情况。

1.6.3 安全隐患

重定向确实存在安全隐患。在防火墙上开了一个允许流量进入内部网络的洞，被保护的网络安全潜在的受到了威胁！例如，如果流量被转发到了内部的 web 服务器，而 web 服务器上允许的守护程序或者 CGI 脚本程序存在漏洞，则这台服务器存在会被因特网网上的入侵者攻击危险。如果入侵者控制了这台机器，就有了进入内部网络的通道，仅仅是因为这种流量是允许通过防火墙的。

这种风险可以通过将允许外部网络访问的系统限制在一个单独的网段中来减小。这个网段通常也被称为非军事化区域（DMZ）或者私有服务网络（PSN）。通过这个方法，如果 web 服务器被控制，通过严格的过滤进出 DMZ/PSN 的流量，受影响的系统仅限于 DMZ/PSN 网段。

1.6.4 重定向和反射

通常，重定向规则是用来将因特网上到来的连接转发到一个内部网络或者局域网的私有地址。例如：

```
server = 192.168.1.40
rdr on $ext_if proto tcp from any to $ext_if port 80 -> $server \
 port 80
```

但是，当一个重定向规则被从局域网上的客户端进行测试时，它不会正常工作。这是因为重定向规则仅适用于通过指定端口（$ext_if,外部接口，在上面的例子中）的数据包。从局域网上的主机连接防火墙的外部地址，并不意味着数据包会实际的通过外部接口。防火墙上的 TCP/IP 栈会把到来的数据包的目的地址在通过内部接口时与它自己的 IP 地址或者别名进行对比检测。那样的数据包不会真的通过外部接口，栈在任何情况下也不会建立那样的通道。因而，PF 永远也不会在外部接口上看到那些数据包，过滤规则由于指定了外部接口也不会起作用。

指定第二条针对内部接口的也达不到预想的效果。当本地的客户端连接防火墙的外部地址时，初始化的 TCP 握手数据包是通过内部接口到达防火墙的。重定向规则确实起作用了，目标地址被替换成了内部服务器，数据包通过内部接口转发到了内部的服务器。但源地址没有进行转换，仍然包含的是本地客户端的 IP 地址，因此服务器把它的回应直接发送给了客户端。防火墙永远收不到应答不可能返回客户端信息，客户端收到的应答不是来自它期望的源（防火墙）会被丢弃，TCP 握手失败，不能建立连接。

当然，局域网里的客户端仍然会希望象外部客户一样透明的访问这台内部服务器。有如下的方法解决这个问题：

1. 水平分割 DNS 存在这样的可能性，即配置 DNS 服务器使得它回答内部主机的查询和回答外部主机的查询不一样，因此内部客户端在进行名称解析时会收到内部服务器的地址。它们直接连接到内部服务器，防火墙根本不牵扯。这会降低本地流量，因为数据包不会被送到防火墙。

2. 将服务器移到独立的本地网络增加单独的网络接口卡到防火墙，把本地的服务器从和客户端同一个网段移动到专用的网段（DMZ）可以让本地客户端按照外部重定向

连接的方法一样重定向。使用单独的网段有几个优点，包括和保留的内部主机隔离增加了安全性；服务器（我们的案例中可以从因特网访问）一旦被控制，它不能直接存取本地网络因为所有的连接都必须通过防火墙。

1.6.5 TCP 代理

一般而言，TCP 代理可以在防火墙上设置，监听要转发的端口或者将内部接口上到来的连接重定向到它监听的端口。当本地客户端连接防火墙时，代理接受连接，建立到内部服务器的第二条连接，在通信双方间进行数据转发。

简单的代理可以使用 inetd（8）和 nc（1）建立。下面的/etc/inetd.conf 中的条目建立一个监听套接字绑定到 lookback 地址（127.0.0.1）和端口 5000。连接被转发到服务器 192.168.1.10 的 80 端口。

```
127.0.0.1:5000 stream tcp nowait nobody /usr/bin/nc nc -w \
 20 192.168.1.10 80
```

下面的重定向规则转发内部接口的 80 端口到代理：

```
rdr on $int_if proto tcp from $int_net to $ext_if port 80 -> \
  127.0.0.1 port 5000
```

1.6.6 RDR 和 NAT 结合

通过对内部接口增加 NAT 规则，上面说的转换后源地址不变的问题可以解决。

```
rdr on $int_if proto tcp from $int_net to $ext_if port 80 -> \
 $server
no nat on $int_if proto tcp from $int_if to $int_net
nat on $int_if proto tcp from $int_net to $server port 80 -> \
  $int_if
```

这会导致由客户端发起的初始化连接在收到内部接口的返回数据包时转换回来，客户端的源 ip 地址被防火墙的内部接口地址代替。内部服务器会回应防火墙的内部接口地址，在转发给本地客户端时可以反转 NAT 和 RDR。这个结构是非常复杂的，因为它为每个反射连接产生了 2 个单独的状态。必须小心配置防止 NAT 规则应用到了其他流量，例如连接由外部发起（通过其他的重定向）或者防火墙自己。注意上面的 rdr 规则会导致 TCP/IP 栈看到来自内部接口带有目的地址是内部网络的数据包。

一般而言，上面提到的解决方法可以互相替代。

1.7 规则生成捷径

1.7.1 简介

PF 提供了许多方法来进行规则集的简化。一些好的例子是使用宏和列表。另外，规则集的语言或者语法也提供了一些使规则集简化的捷径。首要的规则是，规则集越简单，就越容易理解和维护。

1.7.2 使用宏

宏是非常有用的，因为它提供了硬编码地址，端口号，接口名称等的可选替代。在一个规则集中，服务器的 IP 地址改变了？没问题，仅仅更新一下宏，不需要弄乱你花费了大量时间和精力建立的规则集。

通常的惯例是在 PF 规则集中定义每个网络接口的宏。如果网卡需要被使用不同驱动的卡取代，例如，用 intel 代替 3com，可以更新宏，过滤规则集会和以前功能一样。另一个优点是，如果在多台机器上安装同样的规则集，某些机器会有不同的网卡，使用宏定义网卡可以使的安装的规则集进行最少的修改。使用宏来定义规则集中经常改变的信息，例如端口号，IP 地址，接口名称等等，建议多多实践！

```
# define macros for each network interface
IntIF = "dc0"
ExtIF = "fxp0"
DmzIF = "fxp1"
```

另一个惯例是使用宏来定义 IP 地址和网络，这可以大大减轻在 IP 地址改变时对规则集的维护。

```
# define our networks
IntNet = "192.168.0.0/24"
ExtAdd = "24.65.13.4"
DmzNet = "10.0.0.0/24"
```

如果内部地址扩展了或者改到了一个不同的 IP 段，可以更新宏为：
IntNet = ” 192.168.0.0/24, 192.168.1.0/24 ”

当这个规则集重新载入时，任何东西都跟以前一样。

1.7.3 使用列表

来看一下一个规则集中比较好的例子使得 RFC1918 定义的内部地址不会传送到因特网上，如果发生传送的事情，可能导致问题。

```
block in quick on tl0 inet from 127.0.0.0/8 to any
block in quick on tl0 inet from 192.168.0.0/16 to any
block in quick on tl0 inet from 172.16.0.0/12 to any
block in quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 127.0.0.0/8
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 10.0.0.0/8
```

看看下面更简单的例子：

```
block in quick on tl0 inet from { 127.0.0.0/8, 192.168.0.0/16, \
 172.16.0.0/12, 10.0.0.0/8 } to any
block out quick on tl0 inet from any to { 127.0.0.0/8, \
 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }
```

这个规则集从 8 行减少到 2 行。如果联合使用宏，还会变得更好：

```
NoRouteIPs = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, \
 10.0.0.0/8 }"
ExtIF = "tl0"
block in quick on $ExtIF from $NoRouteIPs to any
block out quick on $ExtIF from any to $NoRouteIPs
```

注意虽然宏和列表简化了 pf.conf 文件，但是实际是这些行会被 pfctl（8）扩展成多行，因此，上面的例子实际扩展成下面的规则：

```
block in quick on tl0 inet from 127.0.0.0/8 to any
block in quick on tl0 inet from 192.168.0.0/16 to any
block in quick on tl0 inet from 172.16.0.0/12 to any
block in quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 10.0.0.0/8
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 127.0.0.0/8
```

可以看到，PF 扩展仅仅是简化了编写和维护 pf.conf 文件，实际并不简化 pf(4)对于规则的处理过程。

宏不仅仅用来定义地址和端口，它们在 PF 的规则文件中到处都可以用：

```
pre = "pass in quick on ep0 inet proto tcp from "
post = "to any port { 80, 6667 } keep state"

# David ‘s classroom
$pre 21.14.24.80 $post

# Nick ‘s home
$pre 24.2.74.79 $post
$pre 24.2.74.178 $post
```

扩展后:

```
pass in quick on ep0 inet proto tcp from 21.14.24.80 to any \
port = 80 keep state
pass in quick on ep0 inet proto tcp from 21.14.24.80 to any \
```

```
port = 6667 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
port = 80 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
port = 6667 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.178 to any \
port = 80 keep state
pass in quick on ep0 inet proto tcp from 24.2.74.178 to any \
port = 6667 keep state
```

1.7.4 PF 语法

PF 的语法相当灵活，因此，允许编写非常灵活的规则集。PF 能够自动插入某些关键字因此它们不必在规则中明确写出，关键字的顺序也是随意的，因此不需要记忆严格的语法限制。

1.7.5 减少关键字

要定义全部拒绝的策略，使用下面 2 条规则：

```
block in all
block out all
```

这可以简化为:

```
block all
```

如果没有指定方向，PF 会认为规则适用于数据包传递的进、出 2 个方向。

同样的，”from any to any” 和”all” 子句可以在规则中省略，例如

```
block in on rl0 all
pass in quick log on rl0 proto tcp from any to any port 22 keep state
```

可以简化为:

```
block in on rl0
pass in quick log on rl0 proto tcp to port 22 keep state
```

第一条规则阻塞 rl0 上从任意到任意的进入数据包，第二条规则允许 rl0 上端口 22 的 TCP 流量通过。

1.7.6 Return 简化

用于阻塞数据包，回应 TCP RST 或者 ICMP 不可到达的规则集可以这么写：

```
block in all
```

```
block return-rst in proto tcp all
block return-icmp in proto udp all
block out all
block return-rst out proto tcp all
block return-icmp out proto udp all
```

可以简化为：:

```
block return
```

当 PF 看到 return 关键字，PF 可以智能回复合适应答，或者完全不回复，取决于要阻塞的数据包使用的协议。W

1.7.7 关键字顺序

在大多数情况下，关键字的顺序是非常灵活的。例如，规则可以这么写：

```
pass in log quick on rl0 proto tcp to port 22 \
  flags S/SA keep state queue ssh label ssh
```

也可以这么写：

```
pass in quick log on rl0 proto tcp to port 22 \
 queue ssh keep state label ssh flags S/SA
```

其他类似的顺序也能够正常工作。

§2 高级配置

2.1 运行选项

运行选项是控制 pf 操作的选择。这些选项在 pf.conf 中使用 set 指定。

2.1.1 set block-policy

设定过滤规则中指定的 block 动作的默认行为。

- drop - 数据包悄然丢弃.
- return - TCP RST 数据包返回给遭阻塞的 TCP 数据包，ICMP 不可到达数据包返回给其他。注意单独的过滤规则可以重写默认的响应。

2.1.2 set debug

设定 pf 的调试级别。

- none - 不显示任何调试信息。
- urgent - 为严重错误产生调试信息，这是默认选择。
- misc - 为多种错误产生调试信息。（例如，收到标准化/整形的数据包的状态，和产生失败的状态）。.
- loud - 为普通条件产生调试信息（例如，收到被动操作系统检测信息状态）。

2.1.3 set fingerprints file

设定应该装入的进行操作系统识别的操作系统特征文件来，默认是/etc/pf.os.

2.1.4 set limit

- frags - 在内存池中进行数据包重组的最大数目。默认是 5000。
- src-nodes - 在内存池中用于追踪源地址（由 stick — address 和 source-track 选项产生）的最大数目，默认是 10000。
- states - 在内存池中用于状态表（过滤规则中的 keep state）的最大数目，默认是 10000。

2.1.5 set loginterface int

设定 PF 要统计进/出流量和放行/阻塞的数据包的数目的接口卡。统计数目一次只能用于一张卡。注意 match, bad-offset, 等计数器和状态表计数器不管 loginterface 是否设置都会被记录。

2.1.6 set optimization

为以下的网络环境优化 PF：

- normal - 适用于绝大多数网络，这是默认项。
- high-latency - 高延时网络，例如卫星连接。
- aggressive - 自状态表中主动终止连接。这可以大大减少繁忙防火墙的内存需求，但要冒空闲连接被过早断开的风险。
- conservative - 特别保守的设置。这可以避免在内存需求过大时断开空闲连接，会稍微增加 CPU 的利用率。

2.1.7 set state-policy

设定 PF 在状态保持时的行为。这种行为可以被单条规则所改变。见状态保持章节。

- if-bound - 状态绑定到产生它们的接口。如果流量匹配状态表种条目但不是由条目中记录的接口通过，这个匹配会失败。数据包要么匹配一条过滤规则，或者被丢弃/拒绝。
- group-bound - 行为基本和 if-bound 相同，除了数据包允许由同一组接口通过，例如所有的 ppp 接口等。
- floating - 状态可以匹配任何接口上的流量。只要数据包匹配状态表条目，不管是否匹配它通过的接口，都会放行。这是默认的规则。

2.1.8 set timeout

- interval - 丢弃过期的状态和数据包碎片的秒数。
- frag - 不能重组的碎片过期的秒数。

2.1.9 例子

```
set timeout interval 10
set timeout frag 30
set limit { frags 5000, states 2500 }
set optimization high-latency
set block-policy return
set loginterface dc0
```

```
set fingerprints /etc/pf.os.test
set state-policy if-bound
```

2.2 流量整形(数据包标准化)

2.2.1 简介

流量整形是将数据包标准化避免最终的数据包出现非法的目的。流量整形指引同时也会重组数据包碎片，保护某些操作系统免受某些攻击，丢弃某些带有非法联合标记的 TCP 数据包。流量整形指引的简单形式是：

```
scrub in all
```

这会对所有接口上到来的数据包进行流量整形。

一个不在接口上进行流量整形的原因是要透过 PF 使用 NFS。一些非 openbsd 的平台发送（和等待）奇怪的数据包，对设置不分片位的数据包进行分片。这会被流量整形（正确的）拒绝。这个问题可以通过设置 no-df 选项解决。另一个原因是某些多用户的游戏在进行流量整形通过 PF 时存在连接问题。除了这些极其罕见的案例，对所有的数据包进行流量整形时强烈推荐的设置。

流量整形指引语法相对过滤语法是非常简单的，它可以非常容易的选择特定的数据包进行整形而不对没指定的数据包起作用。

更多的关于数据整形的原理和概念可以在这篇论文中找到：
Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics.

2.2.2 选项

流量整形有下面的选项:

- no-df 在 IP 数据包头中清除不分片位的设置。某些操作系统已知产生设置不分片位的分片数据包。尤其是对于 NFS。流量整形（scrub）会丢弃这类数据包除非设置了 no-df 选项。某些操作系统产生带有不分片位和用 0 填写 IP 包头中分类域，推荐使用 no-df 和 random-id 联合使用解决。

- random-id 用随机值替换某些操作系统输出数据包中使用的可预测 IP 分类域的值这个选项仅适用于在选择的数据包重组后不进行分片的输入数据包。

- min-ttl num 增加 IP 包头中的最小存活时间（TTL）。

- max-mss num 增加在 TCP 数据包头中最大分段值（MSS）。

- fragment reassemble 在传递数据包到过滤引擎前缓冲收到的数据包碎片，重组它们成完整的数据包。优点是过滤规则仅处理完整的数据包，忽略碎片。缺点是需要内存缓冲数据包碎片。这是没有设置分片选项时的默认行为。这也是能和 NAT 一起工作的唯一分片选项。

- fragment crop 导致重复的碎片被丢弃，重叠的被裁剪。与碎片重组不同，碎片不会被缓冲，而是一到达就直接传递。
- fragment drop-ovl 跟 fragment crop 相似，除了所有重复和重叠的碎片和其他更多的通信碎片一样被丢弃。
- reassemble tcp TCP 连接状态标准化。当使用了 scrub reassemble tcp 时，方向（进/出）不用说明，会执行下面的标准化过程：
    1. 连接双方都不允许减少它们的 IP TTL 值。这样做是为了防止攻击者发送数据包到防火墙，影响防火墙保持的连接状态，使数据包在到达目的主机前就过期。所有数据包的 TTL 都为了这个连接加到了最大值。
    2. 用随机数字调整 TCP 数据包头中的 RFC1323 时间戳。这可以阻止窃听者推断主机在线的时间和猜测 NAT 网关后面有多少主机。

实例:

```
scrub in on fxp0 all fragment reassemble min-ttl 15 max-mss 1400
scrub in on fxp0 all no-df
scrub on fxp0 all reassemble tcp
```

2.3 锚定和命名规则集

2.3.1 简介

除了主要的规则集，PF 还可以载入子规则集。由于子规则集可以使用 pfctl（8）操作，这提供了一个动态修改活动规则集的方法。正如表被用来保存动态地址列表，子规则用来保存动态过滤设定，nat，rdr 和 binat 规则。

子规则集通过使用锚定附加到主规则集中。有 4 种类型的锚定规则：

1. anchor name - 检测锚定名称中的所有规则
2. binat-anchor name - 检测锚定名称中的 binat 规则。
3. nat-anchor name -检测锚定名称中的 nat 规则。
4. rdr-anchor name -检测锚定名称中的 rdr 规则。

只有主规则集可以包含锚定规则。

2.3.2 命名规则集

命名规则集是被配置了名称一组过滤和/或转换规则。一个锚定点可以包含多于一个的类似规则集。当 PF 在主规则集中碰到锚定规则，它会按字母顺序检测附加到锚定点的所有规则集。处理过程会在主规则集中继续直到匹配了带 quick 的规则，或者在锚定中匹配了认为是结束的转换规则，锚定规则和主规则才不再继续执行。

例如:

```
ext_if = "fxp0"

block on $ext_if all
pass out on $ext_if all keep state
anchor goodguys
```

这条规则集设定了 fxp0 接口上默认拒绝进出的所有流量的策略。然后通过的流量保持状态，后面是一个锚定规则集名称是 goodguys。锚定可以通过 2 个方法替换为规则：

- 使用可载入的规则
- 使用 pfctl(8)

可载入的规则使 pfctl 通过读入一个文本文件代替指定的锚定和命名规则集。例如：

```
load anchor goodguys:ssh from "/etc/anchor-goodguys-ssh"
```

当主规则集载入时，/etc/anchor-goodguys-ssh 文件中列出的规则中命名为 ssh 的规则集会被载入到名称为 goodguys 的锚定点。

要使用 pfctl 为锚定点增加规则，可以使用下面这样的命令：

```
# echo "pass in proto tcp from 192.0.2.3 to any port 22" \
  | pfctl -a goodguys:ssh -f -
```

这增加了一条防线规则到 goodguys 锚定点并命名为 ssh。PF 在主规则中到达 goodguys 锚定点时会检测这条规则（包括其他加入的过滤规则）。

规则也可以通过文本文件载入和保存：

```
# cat >> /etc/anchor-goodguys-www
pass in proto tcp from 192.0.2.3 to any port 80
pass in proto tcp from 192.0.2.4 to any port { 80 443 }

# pfctl -a goodguys:www -f /etc/anchor-goodguys-www
```

这从/etc/anchor-goodguys-www 文件载入规则到 goodguys 锚定点命名为 www。

过滤和转换规则也可以使用同样的语法和选项载入命名规则集，类似主规则集载入普通的规则。一个警告：命名规则集中使用的宏必须同时被定义，在主规则集中定义的宏对命名规则集不可见。

每一个命名规则集，和主规则集一样，和其他规则集独立存在的。对于某个规则集的操作，比如删除一个规则集，不会影响其他的规则集。另外，在主规则集中移动一个锚定点的位置不会影响这个锚定点和附属于这个锚定点的命名规则集。一个命名规则集不会被破坏，除非使用 pfctl（8）删除所有规则。如果一个锚定点没有附属的命名规则集，它也就被破坏了！

2.3.3 锚定选项

锚定规则可以随意指定接口，协议，源和目的地址，标记等等，使用的是和过滤规则同样的语法。如果这些信息存在，锚定规则仅处理匹配锚定规则定义的数据包。例如：

```
ext_if = "fxp0"

block on $ext_if all
pass out on $ext_if all keep state
anchor ssh in on $ext_if proto tcp from any to any port 22
```

锚定规则 ssh 仅检测来自 fxp0 目标为端口 22 的 tcp 数据包。规则可以用如下方法添加：

```
# echo "pass in from 192.0.2.10 to any" | pfctl -a ssh:allowed -f -
```

因此，尽管过滤规则中没有指定接口，协议，端口，由于锚定规则的定义，主机 192.0.2.10 只允许使用 ssh 连接。

2.3.4 操作命名规则集

命名规则集的操作是通过 pfctl 实现的。可以不用重新载入主规则集就在规则集中删除和增加规则。要列出附属于 ssh 锚定规则集中的规则，可以使用：

```
# pfctl -a ssh:allowed -s rules
```

要删除这个规则集中所有过滤规则：

```
# pfctl -a ssh:allowed -F rules
```

如果规则集名称省略，这个动作应用于锚定里的所有规则。命令的详细列表，可以查看 pfctl（8）。

2.4 队列和优先级

2.4.1 队列

使用队列是为了按顺序保存一些等待处理的数据。在计算机网络中，当数据包由一台主机发出后，他们将进入一个等待队列，由操作系统决定哪个队列或者某个队列中的哪些包将被处理。操作系统选取包进行处理的顺序将影响网络性能。例如，一个用户打开了两个网络程序：SSH 和 FTP，由于 SSH 的时效性要求，在理想情况下 SSH 数据包将先于 FTP 数据包被处理。当用户在 SSH 客户端敲入一条命令时，需要有及时的反馈信息，但是 FTP 数据传输延时一段时间不会引起太大注意是可以忍受的，但是如果系统在处理 SSH 连接之前正在处理大量的 FTP 数据包，这时会怎样呢？SSH 的数据包会留在队列中（或者由于队列长度限制被丢弃），导致 SSH 会话滞后。通过定义队列策略，网络带宽可以公平地在不同应用程序、用户和计算机之间分配。

注意队列只是对流出外部接口的数据包起作用。当数据包流入内部接口时再做队列将是非常迟的，因为当内部接口收到这些数据包时他们已经耗用了带宽。唯一的解决办法是在相邻的路由器启用队列，或者，如果接受到数据包的主机被当作是个路由器，那么在数据包流出该路由器的接口上启用队列。

2.4.2 日程

日程用来规定执行哪条队列和执行的顺序。默认情况下，OpenBSD 使用先进先出（FIFO）队列。先进先出队列类似于在超市或者银行排队，先进入队列将被优先处理。新到的包被加在队列尾。如果队列满了，新到的包将被丢弃，这就是所谓的“弃尾”。

OpenBSD 支持另外两种日程：

- 基于类的队列
- 优先级队列

2.4.3 基于类的队列

基于类的队列（CBQ）是一种排序算法，它将网络连接带宽在很多队列和类中分摊，每个队列都拥有基于源、目的地址，端口号，协议等信息分配的网络流量。一个队列可以被随意配置借用它父队列的带宽，前提是父队列没有占用该带宽。队列也有优先级，例如 SSH，它的数据包将提前于含有大量数据流的队列（例如 FTP）被处理。

CBQ 队列以分等级的方式部署，最高一级是根队列，用来定义全部带宽。子队列建立在根队列之下，每一个子队列可以分配根队列所规定的部分带宽。例如,定义如下队列：

```
Root Queue (2Mbps)

Queue A (1Mbps)
Queue B (500Kbps)
Queue C (500Kbps)
```

这里总带宽被设置为 2Mbps，然后由子队列分割。

在子队列中仍然可以定义下一级策略。用来在不同用户中平等地分配带宽，同时对他们的流量进行分类，以便使某个协议不会抢占其他协议的带宽。具体队列结构定义如下：

```
Root Queue (2Mbps)

UserA (1Mbps)

ssh (50Kbps)
bulk (950Kbps)

UserB (1Mbps)
```

```
  audio (250Kbps)
  bulk (750Kbps)

  http (100Kbps)
  other (650Kbps)
```

注意每一层分配给各个队列的带宽之和不能超过赋予父类队列的带宽。

当父类队列由于它的一个子队列没用占用完原本分配给它的带宽时，该父类队列中的另一个子队列可以占用这部分带宽。看下列配置：

```
Root Queue (2Mbps)

UserA (1Mbps)

ssh (100Kbps)
ftp (900Kbps, borrow)

UserB (1Mbps)
```

如果 UserA 队列实际占用的带宽小于 1Mbps（比如 ssh 队列没用完全占用 100Kbps），则 ftp 队列的流量超过 900Kbps 时，ftp 子队列将占用 UserA 父队列中定义的剩余的带宽。通过这种途径可以使 ftp 子队列超过它所定义的负荷时得到更多的带宽。随着 ssh 子队列流量的增加，被占用的这些带宽将被返还。

CBQ 为每个队列分配一个优先级。高优先级的队列在负荷重的情况下将早于低优先级的队列被处理，同样情况应用在同父类的子队列。同优先级的队列间通过抢占方式轮流执行。例如：

```
Root Queue (2Mbps)

UserA (1Mbps, priority 1)

ssh (100Kbps, priority 5)
ftp (900Kbps, priority 3)

UserB (1Mbps, priority 1)
```

CBQ 将以轮流抢占模式处理 UserA 和 UserB 队列，任何一个队列不能优先于另一个被处理。当 UserA 队列执行的时候，CBQ 同时会处理它的子队列，这时，如果网络拥塞，ssh 子队列会被优先处理，因为它的优先级高于 ftp 子队列。注意为什么 ssh 和 ftp 子队列不与 UserA 和 UserB 队列比较优先级，因为他们不在一个层上。

2.4.4 优先级队列

优先级队列（PRIQ）为一个网络接口上的多个队列逐一分配唯一的优先级。拥有高优先级的队列总是在低优先级队列前被处理。

PRIQ 中的队列结构是平面的，你不可以在一个队列中定义子队列。根队列用来定义全部带宽，其他队列定义在根队列之后。请看如下实例：

```
Root Queue (2Mbps)

Queue A (priority 1)
Queue B (priority 2)
Queue C (priority 3)
```

上面定义了一个有 2Mbps 带宽的根队列和 3 个子队列。最高优先级的队列被先处理，该队列中的所有包均被处理完后，或者该队列是空，PRIQ 将进一步处理下一优先级的队列。在一个队列中，各个包是按照先进先出原则进行处理。

需要注意的是当使用 PRIQ 时必须非常谨慎地进行设计，因为 PRIQ 的工作机制是先高后低，如果一个高优先级的队列收到的数据包是持续的流，那么它将延时处理低优先级的队列，更有甚者导致丢包。

2.4.5 随机早期检测

随机早期检测(RED)是一种避免网络拥塞的算法，它通过确认队列没有超长来避免网络拥塞。实现方法是不停的计算队列的平均大小并与两个阈值比较，如果计算出的平均值低于小阈值将不会丢弃任何包；如果在两个阈值之间将通过计算概率丢掉一些包；换言之，如果计算的平均值越接近大阈直则被丢弃的包越多。当丢掉一些包时，RED 随机选择从哪些连接丢包，占用大带宽的连接被丢包的几率高。

RED 的用处非常大，因为它可以避免一种被称为全体同步的状态，也可以调整突发流量。全体同步指多个连接的数据包在同一时间被丢弃导致的吞吐量全部消失的情况。例如，如果承载 10 个 FTP 连接流量的一台路由器出现拥塞，大部分包被丢弃，总的流量将迅速下降，这并不是最好的处理方法，因为所有的 FTP 连接都降低了流量，换句话说，这个网络将不会再次发挥最大潜能。RED 通过只在随机挑选的连接上丢包来避免上述情况。占用大带宽的连接被丢包的几率高，这样，占用大带宽的连接将受到节制，避免了拥塞，同时总流量迅速降低的现象也不会出现。另外，RED 可以处理突发流量，因为它在队列装满之前就开始丢弃数据包，当突发流量到来时，队列中有足够的空间保存新发来的数据包。

RED 只能被用在传输协议有能力反馈拥塞指示的情况。在大多数情况下，这也就是说 RED 被用来处理 TCP 数据流而不是 DUP 或 ICMP 数据流。

2.4.6 外部拥塞告知

外部拥塞告知(ECN)与 RED 协作来发现两台主机网络通讯路径上的任何拥塞现象。原理是使 RED 在头包中设置一个标志位并返回而不是丢弃该包。假设发送端主机支持 ECN，它将读到这个标志位信息并减少发出网络流量。

更多关于 ECN 的信息请参考 RFC 3168

2.4.7 配置队列

自 OpenBSD3.0 后交互队列（ALTQ）就成为基本系统的一部分。到了 OpenBSD3.3，ALTQ 被集成到了 PF 中。ALTQ 支持 CBQ、PRIQ，也支持 RED 和 ECN。

既然 ALTQ 被集成到了 PF，那么 PF 就必须使队列工作。在开始一章有配置 PF 的介绍。

队列定义在 pf.conf 中，有两种指令模式：

- altq on - 在某个接口上开启队列，定义使用哪些日程，建立根队列
- queue - 定义子队列的属性

altq 的语法为：

```
altq on interface scheduler bandwidth bw qlimit qlim \
  tbrsize size queue { queue_list }
```

- interface - 开启队列的网络接口。
- scheduler - 使用的队列日程。可能的取值是 cbq 和 priq。某个接口在某个时间只能启动 1 个日程。
- bw - 日程所能用到的所有带宽。可以使用后缀 b，Kb,Mb 和 Gb 表示，可以是具体的数值也可以是当前接口带宽的百分比。
- qlim - 队列保存数据包的最大数量。该值是可选的，默认 50。
- size - 承载容量的大小，单位 bytes。如果没有指定，将基于接口带宽自动设置。
- queue_list - 在根队列下建立的一个子队列列表。

实例:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ssh, ftp }
```

这条策略在接口 fxp0 启用 CBQ，总带宽设置为 2Mbps，定义了 3 个子队列：std，ssh 和 ftp。

队列指令的语法是：

```
queue name [on interface] bandwidth bw [priority pri] [qlimit qlim] \
 scheduler ( sched_options ) { queue_list }
```

- name - 队列的名称，必须与 altq 中定义的队列列表中的某个队列名称一致。对于 cbq，还可以与以前定义的队列列表中的某个名称一致。长度不能超过 15 个字符。
- interface - 队列所生效的网络接口，是可选项，如果没指定，将在所有接口生效。
- bw - 队列可以使用的总带宽。可以是具体数值加上后缀 b，Kb，Mb 和 Gb 来表示，也可以是总带宽的百分比。该参数只有在使用 cbq 日程时可用。

- pri - 队列的优先级。对于 cbq，优先级的范围是 0 ～ 7，对于 priq，范围是 0 ～ 15。0 是最低的优先级，如果没定义，默认优先级为 1。
- qlim - 队列可以保存的最大包数，默认 50。
- scheduler - 日程，cbq 或者 priq。必须和根队列一致。
- sched_options - 控制日程行为的更多选项：
  - default - 定义默认队列，当包不匹配其他队列时的默认值。
  - red - 在当前队列启用 RED。
  - rio - 对进/出启用 RED。在这种状态下，RED 会维护多个平均队列长度和平均门限值，每个 IP 服务质量级别对应一个。
  - ecn - 在当前队列启用 ECN。
  - borrow - 队列可以向父类借用带宽值，只被用于 cbq 日程模式。
- queue_list - 在当前队列下建立的一系列子队列。只有在 cbq 日程模式下有效。

继续上述实例:

```
queue std bandwidth 50% cbq(default)
queue ssh { ssh_login, ssh_bulk }
queue ssh_login priority 4 cbq(ecn)
queue ssh_bulk cbq(ecn)
queue ftp bandwidth 500Kb priority 3 cbq(borrow red)
```

这里将设置前面已经定义的队列。Std 队列分配了根队列的 50%带宽，也就是 1Mbps，并被设置为默认队列。Ssh 队列定义了两个子队列，ssh_login 和 ssh_bulk。前者被分配了高于后者的优先级，并且都开启了 ECN。ftp 队列分配了 500Kbps 带宽，拥有第 3 优先级，当总带宽富余时它可以借用，同时也开启了 RED。

2.4.8 为队列分配数据流

通过在 PF 过滤策略中增加 queue 关键字为队列分配流量。例如，假设某个策略集包含下面一条策略：

```
pass out on fxp0 from any to any port 22
```

符合上述策略的数据流可以通过 queue 关键字赋予一个特定的队列：

```
pass out on fxp0 from any to any port 22 queue ssh
```

当 queue 关键字应用到 block 时，任何 TCP RST 或者 ICMP Unreachable 数据包将被分配到特定对列。

注意队列除了可以在 altq 语句生效外，还可以在接口上生效:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ftp }
queue std cbq(default)
queue ftp bandwidth 1.5Mb

pass in on dc0 from any to any port 21 queue ftp
```

fxp0 接口上指定了队列，但是这个队列却可以在 dc0 上生效。如果数据包匹配了 fxp0 接口外的那条 pass 策略，它们将被应用到 ftp 队列。这种用法在路由器上用处很广。

一般情况下 queue 关键字后面只有一个队列，如果有另外一个，这个队列将被用处理低延时的包和 TCP ACK 的包，这种包没有有效数据载荷。例如，当使用 SSH 时，SSH 的登录会话将设置为低延时性，然而 SCP 和 SFTP 会话则不会。PF 可以通过这些信息对属于登录会话的数据包取出并重新排列，这对于赋予登录会话的数据包高优先级非常有用。

```
pass out on fxp0 from any to any port 22 queue(ssh_bulk, ssh_login)
```

这条策略将属于 SSH 登录连接的数据包分配给 ssh_logind 队列，属于 SCP 和 SFTP 连接的数据包分配给 ssh_bulk 队列。由于前者拥有比后者高的优先级，所以登录连接的数据包被优先处理。

在非对称连接情况下给 TCP ACK 包分配一个高优先级的队列将是很有用的，比如，ADSL 就是一种非对称连接，因为它的上行和下行带宽不一致。如果上行通道已满，而恰在这时开始了下载动作，下载过程的启动需要客户端返回 TCP ACK 包，但此时该包由于上行阻塞不能及时到达下载服务器，下载过程将会进入等待状态。试验证明，为了取得好的效果，上行队列的带宽最好设置比实际带宽小一些。例如，一个 ADSL 线路最大上行带宽 640Kbps，那么设置根队列 600Kbps 的带宽效果会很好。

使用 keep state 的队列策略：

```
pass in on fxp0 proto tcp from any to any port 22 flags S/SA \
 keep state queue ssh
```

PF 将在状态表中记录 ssh 队列，当数据从 fxp0 接口流出时如果符合表中的记录它将被放入 ssh 队列。注意，尽管 queue 关键字应用到了过滤规则中，目的在于影响流出的数据包，上述策略中队列不会对流入的数据包起作用。

2.4.9 实例 1: 小型家庭网络

```
[ Alice ]   [ Charlie ]
   |             |                                        ADSL
---+-----+-------+------ dc0 [ OpenBSD ] fxp0 -------- ( Internet )
         |
      [ Bob ]
```

在这个例子中，OpenBSD 充当一个家庭网的网关，提供包过滤和 NAT 服务，家庭网中有 3 个客户端，因特网连接是通过 ADSL 实现，并有 2Mbps 下行和 640Kbps 上行的带宽。

该网络的队列策略：

- 为 Bob 保留玩在线游戏的 80Kbps 下行带宽，以减少另外两人对他的影响，并且总带宽富余的情况下可以超出该限制。
- 交互的 SSH 和即时信息流量要有高于其他流量的优先级。
- DNS 请求和反馈数据流要有第二高的优先级。
- 流出的 TCP ACK 数据包的优先级要高于其他流出数据包的优先级。

下面是对应的策略（省略了其他部分策略，如 rdr、nat 等）：

```
# enable queueing on the external interface to control traffic going to
# the Internet. use the priq scheduler to control only priorities. set
# the bandwidth to 610Kbps to get the best performance out of the TCP
# ACK queue.

altq on fxp0 priq bandwidth 610Kb queue { std_out, ssh_im_out, dns_out, \
tcp_ack_out }

# define the parameters for the child queues.
# std_out - the standard queue. any filter rule below that does not
# explicitly specify a queue will have its traffic added
# to this queue.
# ssh_im_out - interactive SSH and various instant message traffic.
# dns_out - DNS queries.
# tcp_ack_out - TCP ACK packets with no data payload.

queue std_out priq(default)
queue ssh_im_out priority 4 priq(red)
queue dns_out priority 5
queue tcp_ack_out priority 6

# enable queueing on the internal interface to control traffic coming in
# from the Internet. use the cbq scheduler to control bandwidth. max
# bandwidth is 2Mbps.

altq on dc0 cbq bandwidth 2Mb queue { std_in, ssh_im_in, dns_in, bob_in }

# define the parameters for the child queues.
# std_in - the standard queue. any filter rule below that does not
# explicitly specify a queue will have its traffic added
# to this queue.
# ssh_im_in - interactive SSH and various instant message traffic.
# dns_in - DNS replies.
```

```
# bob_in - bandwidth reserved for Bob ‘s workstation. allow him to
# borrow.

queue std_in cbq(default)
queue ssh_im_in priority 4
queue dns_in priority 5
queue bob_in bandwidth 80Kb cbq(borrow)

2007-1-11 15:40 linuxaihao
# ... in the filtering section of pf.conf ...

alice = "192.168.0.2"
bob = "192.168.0.3"
charlie = "192.168.0.4"
local_net = "192.168.0.0/24"
ssh_ports = "{ 22 2022 }"
im_ports = "{ 1863 5190 5222 }"

# filter rules for fxp0 inbound
block in on fxp0 all

# filter rules for fxp0 outbound
block out on fxp0 all
pass out on fxp0 inet proto tcp from (fxp0) to any flags S/SA \
keep state queue(std_out, tcp_ack_out)
pass out on fxp0 inet proto { udp icmp } from (fxp0) to any keep state
pass out on fxp0 inet proto { tcp udp } from (fxp0) to any port domain \
keep state queue dns_out
pass out on fxp0 inet proto tcp from (fxp0) to any port $ssh_ports \
flags S/SA keep state queue(std_out, ssh_im_out)
pass out on fxp0 inet proto tcp from (fxp0) to any port $im_ports \
flags S/SA keep state queue(ssh_im_out, tcp_ack_out)

# filter rules for dc0 inbound
block in on dc0 all
pass in on dc0 from $local_net

# filter rules for dc0 outbound
block out on dc0 all
pass out on dc0 from any to $local_net
pass out on dc0 proto { tcp udp } from any port domain to $local_net \
```

```
queue dns_in
pass out on dc0 proto tcp from any port $ssh_ports to $local_net \
queue(std_in, ssh_im_in)
pass out on dc0 proto tcp from any port $im_ports to $local_net \
queue ssh_im_in
pass out on dc0 from any to $bob queue bob_in
```

2.4.10 实例 2: 公司网络

```
( IT Dept ) [ Boss ‘s PC ]
  |          |                                              T1
--+----+-----+---------- dc0 [ OpenBSD ] fxp0 -------- ( Internet )
                                | fxp1
                          [ WWW ] /
```

这个例子中 OpenBSD 作为公司网络的防火墙，公司内部在 DMZ 区运行了 WWW 服务器，用户通过 FTP 上传他们的网站。IT 部门有自己的子网，老板的电脑主要用来收发电子邮件和网页冲浪。防火墙通过 1.5Mbps 双向带宽的 T1 电路连接因特网，其他网段均使用快速以太网（100Mbps）。

实现上述要求的策略如下：

- 限制 WWW 服务器到因特网之间的双向流量——500Kbps。
- WWW 服务器和内部网络之间没有流量限制。
- 赋予 WWW 服务器和因特网间的流量高于其他流量的优先级（例如 FTP 上传流量）。
- 为 IT 部门保留 500Kbps 的带宽使他们可以下载到最新的软件，同时如果总带宽富余，他们可以借用。
- 为老板访问因特网的流量赋予比其他访问因特网流量高的优先级。

下面是对应的策略（省略了其他部分策略，如 rdr、nat 等）：

```
# enable queueing on the external interface to queue packets going out
# to the Internet. use the cbq scheduler so that the bandwidth use of
# each queue can be controlled. the max outgoing bandwidth is 1.5Mbps.

altq on fxp0 cbq bandwidth 1.5Mb queue { std_ext, www_ext, boss_ext }

2007-1-11 15:40 linuxaihao
# define the parameters for the child queues.
# std_ext - the standard queue. also the default queue for
# outgoing traffic on fxp0.
```

```
# www_ext - container queue for WWW server queues. limit to
# 500Kbps.
# www_ext_http - http traffic from the WWW server
# www_ext_misc - all non-http traffic from the WWW server
# boss_ext - traffic coming from the boss ‘s computer

queue std_ext cbq(default)
queue www_ext bandwidth 500Kb { www_ext_http, www_ext_misc }
queue www_ext_http priority 3 cbq(red)
queue www_ext_misc priority 1
queue boss_ext priority 3

# enable queueing on the internal interface to control traffic coming
# from the Internet or the DMZ. use the cbq scheduler to control the
# bandwidth of each queue. bandwidth on this interface is set to the
# maximum. traffic coming from the DMZ will be able to use all of this
# bandwidth while traffic coming from the Internet will be limited to
# 1.0Mbps (because 0.5Mbps (500Kbps) is being allocated to fxp1).

altq on dc0 cbq bandwidth 100% queue { net_int, www_int }

# define the parameters for the child queues.
# net_int - container queue for traffic from the Internet. bandwidth
# is 1.0Mbps.
# std_int - the standard queue. also the default queue for outgoing
# traffic on dc0.
# it_int - traffic to the IT Dept network.
# boss_int - traffic to the boss ‘s PC.
# www_int - traffic from the WWW server in the DMZ.

queue net_int bandwidth 1.0Mb { std_int, it_int, boss_int }
queue std_int cbq(default)
queue it_int bandwidth 500Kb cbq(borrow)
queue boss_int priority 3
queue www_int cbq(red)

# enable queueing on the DMZ interface to control traffic destined for
# the WWW server. cbq will be used on this interface since detailed
# control of bandwidth is necessary. bandwidth on this interface is set
# to the maximum. traffic from the internal network will be able to use
# all of this bandwidth while traffic from the Internet will be limited
```

```
# to 500Kbps.

altq on fxp1 cbq bandwidth 100% queue { internal_dmz, net_dmz }

# define the parameters for the child queues.
# internal_dmz - traffic from the internal network.
# net_dmz - container queue for traffic from the Internet.
# net_dmz_http - http traffic.
# net_dmz_misc - all non-http traffic. this is also the default queue.

queue internal_dmz # no special settings needed
queue net_dmz bandwidth 500Kb { net_dmz_http, net_dmz_misc }
queue net_dmz_http priority 3 cbq(red)
queue net_dmz_misc priority 1 cbq(default)

2007-1-11 15:41 linuxaihao
# ... in the filtering section of pf.conf ...

main_net = "192.168.0.0/24"
it_net = "192.168.1.0/24"
int_nets = "{ 192.168.0.0/24, 192.168.1.0/24 }"
dmz_net = "10.0.0.0/24"

boss = "192.168.0.200"
wwwserv = "10.0.0.100"

# default deny
block on { fxp0, fxp1, dc0 } all

# filter rules for fxp0 inbound
pass in on fxp0 proto tcp from any to $wwwserv port { 21, \
> 49151 } flags S/SA keep state queue www_ext_misc
pass in on fxp0 proto tcp from any to $wwwserv port 80 \
flags S/SA keep state queue www_ext_http

# filter rules for fxp0 outbound
pass out on fxp0 from $int_nets to any keep state
pass out on fxp0 from $boss to any keep state queue boss_ext

# filter rules for dc0 inbound
pass in on dc0 from $int_nets to any keep state
```

```
pass in on dc0 from $it_net to any queue it_int
pass in on dc0 from $boss to any queue boss_int
pass in on dc0 proto tcp from $int_nets to $wwwserv port { 21, 80, \
> 49151 } flags S/SA keep state queue www_int

# filter rules for dc0 outbound
pass out on dc0 from dc0 to $int_nets

# filter rules for fxp1 inbound
pass in on fxp1 proto { tcp, udp } from $wwwserv to any port 53 \
keep state

# filter rules for fxp1 outbound
pass out on fxp1 proto tcp from any to $wwwserv port { 21, \
> 49151 } flags S/SA keep state queue net_dmz_misc
pass out on fxp1 proto tcp from any to $wwwserv port 80 \
flags S/SA keep state queue net_dmz_http
pass out on fxp1 proto tcp from $int_nets to $wwwserv port { 80, \
21, > 49151 } flags S/SA keep state queue internal_dmz
```

2.5 地址池和负载均衡

2.5.1 简介

地址池是提供 2 个以上的地址供一组用户共享的。地址池可以是 rdr 规则中的重定向地址；可以是 nat 规则中的转换地址；也可以是 route-to, reply-to, 和 dup-to filter 选项中的目的地址。

有 4 种使用地址池的方法：

- bitmask - 截取被修改地址（nat 规则的源地址；rdr 规则的目标地址）的最后部分和地址池地址的网络部分组合。例如：如果地址池是 192.0.2.1/24，而被修改地址是 10.0.0.50，则结果地址是 192.0.2.50。如果地址池是 192.0.2.1/25，而被修改地址是 10.0.0.130，这结果地址是 192.0.2.2。
- random - 从地址池中随机选择地址.
- source-hash - 使用源地址 hash 来确定使用地址池中的哪个地址。这个方法保证给定的源地址总是被映射到同一个地址池。Hash 算法的种子可以在 source-hash 关键字后通过 16 进制字符或者字符串来指定。默认情况下，pfctl（8）在规则集装入时会随机产生种子。
- round-robin - 在地址池中按顺序循环，这是默认方法，也是表中定义的地址池唯一的方法。

除了 round-robin 方法，地址池的地址必须表达成 CIDR（Classless Inter-Domain Routing ）的网络地址族。round-robin 方法可以接受多个使用列表和表的单独地址。

sticky-address 选项可以在 random 和 round-robin 池类型中使用，保证特定的源地址始终映射到同样的重定向地址。

2.5.2 NAT 地址池

地址池在 NAT 规则中可以被用做转换地址。连接的源地址会被转换成使用指定的方法从地址池中选择的地址。这对于 PF 负载一个非常大的网络的 NAT 会非常有用。由于经过 NAT 的连接对每个地址是有限的，增加附加的转换地址允许 NAT 网关增大服务的用户数量。

在这个例子中，2 个地址被用来做输出数据包的转换地址。对于每一个输出的连接，PF 按照顺序循环使用地址。

```
nat on $ext_if inet from any to any -> { 192.0.2.5, 192.0.2.10 }
```

这个方法的一个缺点是成功建立连接的同一个内部地址不会总是转换为同一个外部地址。这会导致冲突，例如：浏览根据用户的 ip 地址跟踪登录的用户的 web 站点。一个可选择的替代方法是使用 source-hash 方法，以便每一个内部地址总是被转换为同样的外部地址。，要实现这个方法，地址池必须是 CIDR 网络地址。

```
nat on $ext_if inet from any to any -> 192.0.2.4/31 source-hash
```

这条 NAT 规则使用地址池 192.0.2.4/31 (192.0.2.4 - 192.0.2.5)做为输出数据包的转换地址。每一个内部地址会被转换为同样的外部地址，由于 source-hash 关键字的缘故。

2.5.3 外来连接负载均衡

地址池也可以用来进行外来连接负载均衡。例如，外来的 web 服务器连接可以分配到服务器群。

```
web_servers = "{ 10.0.0.10, 10.0.0.11, 10.0.0.13 }"
rdr on $ext_if proto tcp from any to any port 80 -> $web_servers \
 round-robin sticky-address
```

成功的连接将按照顺序重定向到 web 服务器，从同一个源到来的连接发送到同一个服务器。这个 sticky connection 会和指向这个连接的状态一起存在。如果状态过期，sticky connection 也过期。那个主机的更多连接被重定向到按顺序的下一个 web 服务器。

2.5.4 输出流量负载均衡

地址池可以和 route-to 过滤选项联合使用，在多路径路由协议（例如 BGP4）不可用是负载均衡 2 个或者多个因特网连接。通过对 round-robin 地址池使用 route-to，输出连接可以平均分配到多个输出路径。

需要收集的附加的信息是邻近的因特网路由器 IP 地址。这要加入到 route-to 选项后来控制输入数据包的目的地址。

下面的例子通过 2 条到因特网的连接平衡输出流量：

```
lan_net = "192.168.0.0/24"
int_if = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

pass in on $int_if route-to \
 { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
 from $lan_net to any keep state
```

route-to 选项用来在收到流量的内部接口上指定平衡的流量经过各自的网关到输出的网络接口。注意 route-to 选项必须在每个需要均衡的过滤规则上出现。返回的数据包会路由到它们出去时的外部接口（这是由 ISP 做的），然后正常路由回内部网络。

要保证带有属于$ext_if1 源地址的数据包总是路由到$ext_gw1（$ext_if2 和$ext_gw2 也是同样的），下面 2 行必须包括在规则集中：

```
pass out on $ext_if1 route-to ($ext_if2 $ext_gw2) from $ext_if2 \
 to any
pass out on $ext_if2 route-to ($ext_if1 $ext_gw1) from $ext_if1 \
 to any
```

最后，NAT 也可以使用在输出接口中：

```
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($ext_if2)
```

一个完整的输出负载均衡的例子应该是这个样子：

```
lan_net = "192.168.0.0/24"
int_if = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

# nat outgoing connections on each internet interface
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($ext_if2)

# default deny
block in from any to any
block out from any to any
```

```
# pass all outgoing packets on internal interface
pass out on $int_if from any to $lan_net
# pass in quick any packets destined for the gateway itself
pass in quick on $int_if from $lan_net to $int_if
# load balance outgoing tcp traffic from internal network.
pass in on $int_if route-to \
{ ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
proto tcp from $lan_net to any flags S/SA modulate state
# load balance outgoing udp and icmp traffic from internal network
pass in on $int_if route-to \
{ ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
proto { udp, icmp } from $lan_net to any keep state

# general "pass out" rules for external interfaces
pass out on $ext_if1 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if1 proto { udp, icmp } from any to any keep state
pass out on $ext_if2 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if2 proto { udp, icmp } from any to any keep state

# route packets from any IPs on $ext_if1 to $ext_gw1 and the same for
# $ext_if2 and $ext_gw2
pass out on $ext_if1 route-to ($ext_if2 $ext_gw2) from $ext_if2 to any
pass out on $ext_if2 route-to ($ext_if1 $ext_gw1) from $ext_if1 to any
```

2.6 数据包标记

2.6.1 简介

数据包标记是给数据包打内部标记的方法，以后可以在过滤和转换规则中使用。使用标记，有可能做这样的事情，比如在接口间产生信任关系，或者确定数据包是否已经经过了转换规则处理。也可能从基于规则的过滤中移出，开始执行基于策略的过滤。

2.6.2 给数据包打标记

要给数据包打标记，使用 tag 关键字：

```
pass in on $int_if all tag INTERNAL_NET keep state
```

标记 INTERNAL_NET 会增加到任何匹配上述规则的数据包中。注意 keep state 的使用; keep state (或者 modulate state/synproxy state) 在标记数据包通过的规则中使用。

标记也可以通过宏来打，比如：

```
name = "INTERNAL_NET"
pass in on $int_if all tag $name keep state
```

有一组预先定义的宏也可以被使用。

- $if - 接口
- $srcaddr - 源 IP 地址
- $dstaddr - 目的 IP 地址
- $srcport - 源端口
- $dstport - 目的端口
- $proto - 协议
- $nr - 规则号

这些宏在规则集装入时扩展，而不是运行时。

标记遵循以下规则：

- 标记是粘性的。一旦一个标记被匹配的规则打到一个数据包，就不能被删除。但它可以被不同的标记替换。
- 由于标记的粘性，打了标记的数据包会一直保持，即使所有的规则都没有使用这个标记。
- 一个数据包一次最多只能打一个标记。
- 标记是内部标识符，标记不会被送到网上。

看看下面的例子：

```
pass in on $int_if tag INT_NET keep state
pass in quick on $int_if proto tcp to port 80 tag \
   INT_NET_HTTP keep state
pass in quick on $int_if from 192.168.1.5 keep state
```

- 按照规则 1，$int_if 接口上收到的数据包会打上 INT_NET 标记。
- $int_if 接口上收到的目标端口 80 的数据包根据规则 1 首先打上 INT_NET 标记，然后根据规则 2，被 INT_NET_HTTP 标记替代。
- $int_if 接口上收到的来自 192.168.1.5 的数据包根据规则 3 会方向，由于这是最终匹配规则，因此如果它们的目标端口是 80，则标记是 INT_NET_HTTP ，否则标记是 INT_NET 。

标记除了适用于过滤规则以外，nat, rdr, binat 转换规则也可以用 tag 关键字使用标记。

2.6.3 检查数据包标记

要检查先前已经打的标记，可以使用 tagged 关键字：

```
pass out on $ext_if tagged INT_NET keep state
```

在$ext_if 输出的数据包为了匹配上述规则必须打上 INT_NET 标记。反转匹配也可以使用！操作：

```
pass out on $ext_if tagged ! WIFI_NET keep state
```

2.6.4 策略过滤

过滤策略提供了编写过滤规则集的不同方法。定义的策略设定规则，说明哪种流量放行，哪种流量阻塞。数据包被基于传统的标准如源/目的 IP 地址，协议等等分配到不同的策略。例如，检查下面的防火墙策略：

- 自内部 LAN 到 DMZ 的流量是允许的(LAN_DMZ)。
- 自因特网到 DMZ 的服务器流量是允许的。(INET_DMZ)
- 自因特网被重定向到 spamd（8）是允许的(SPAMD)
- 其他所有流量阻塞。

注意策略是如何覆盖所有通过防火墙的流量的。括号里面的项目指示这个策略项目将使用的标记。

需要过滤和转换规则来把数据包分配到不同的策略。

```
rdr on $ext_if proto tcp from <spamd> to port smtp \
  tag SPAMD -> 127.0.0.1 port 8025

block all
pass in on $int_if from $int_net tag LAN_INET keep state
pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ keep state
pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ keep
state
```

现在要设置定义策略的规则。

```
pass in quick on $ext_if tagged SPAMD keep state
pass out quick on $ext_if tagged LAN_INET keep state
pass out quick on $dmz_if tagged LAN_DMZ keep state
pass out quick on $dmz_if tagged INET_DMZ keep state
```

现在要建立整个规则集，修改分类规则。例如，如果 pop3/SMTP 服务器增加到了 DMZ 区，就需要增加针对 POP3 和 SMTP 流量的分类，如下：

```
mail_server = "192.168.0.10"
...
pass in on $ext_if proto tcp to $mail_server port { smtp, pop3 } \
tag INET_DMZ keep state
```

Email 流量会作为 INET-DMZ 策略的条目被放行。完整的规则:

```
# macros
int_if = "dc0"
dmz_if = "dc1"
ext_if = "ep0"
int_net = "10.0.0.0/24"
dmz_net = "192.168.0.0/24"
www_server = "192.168.0.5"
mail_server = "192.168.0.10"

table <spamd> persist file "/etc/spammers"

# classification -- classify packets based on the defined firewall
# policy.
rdr on $ext_if proto tcp from <spamd> to port smtp \
tag SPAMD -> 127.0.0.1 port 8025

block all
pass in on $int_if from $int_net tag LAN_INET keep state
pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ keep state
pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ keep state
pass in on $ext_if proto tcp to $mail_server port { smtp, pop3 } \
tag INET_DMZ keep state

# policy enforcement -- pass/block based on the defined firewall policy.
pass in quick on $ext_if tagged SPAMD keep state
pass out quick on $ext_if tagged LAN_INET keep state
pass out quick on $dmz_if tagged LAN_DMZ keep state
pass out quick on $dmz_if tagged INET_DMZ keep state
```

2.6.5 标记以太网帧

打标记可以在以太网级别进行，如果执行标记/过滤的机器同时做为网桥。通过创建使用 tag 关键字的网桥过滤规则，PF 可以建立基于源/目的 MAC 地址的过滤规则。网桥规则可以由 brconfig（8）命令产生，例如：

```
# brconfig bridge0 rule pass in on fxp0 src 0:de:ad:be:ef:0 \
```

```
tag USER1
```

然后在 pf.conf 文件中:

```
pass in on fxp0 tagged USER1
```

§3 附加主题

3.1 日志

3.1.1 简介

PF 的包日志是由 pflogd（8）完成的，它通过监听 pflog0 接口然后将包以 tcpdump（8）二进制格式写入日志文件（一般在/val/log/pflog）。过滤规则定义的日志和 log-all 关键字所定义的日志都是以这种方式记录的。

3.1.2 读取日志文件

由 pflogd 生成的二进制格式日志文件不能通过文本编辑器读取，必须使用 Tcpdump 来查看日志。

使用如下格式查看日志信息:

```
# tcpdump -n -e -ttt -r /var/log/pflog
```

使用 tcpdump(8)查看日志文件并不是实时的，若要实时查询日志信息需加上 pflog0 参数：

```
# tcpdump -n -e -ttt -i pflog0
```

注意:当查看日志时需要特别注意 tcpdump 的详细协议解码（通过在命令行增加-v 参数实现）。Tcpdump 的详细协议解码器并不具备完美的安全历史，至少在理论上是这样。日志记录设备所记载的部分包信息可能会引发延时攻击，因此推荐在查询日志文件信息之前先将该日志文件从防火墙上移走。

另外需要注意的是对日志文件的安全访问。默认情况下，pflogd 将在日志文件中记录 96 字节的包信息。访问日志文件将提供访问部分敏感包信息的途径（就像 telnet（1）或者 ftp（1）的用户名和密码）。

3.1.3 导出日志

由于 pflogd 以 tcpdump 二进制格式记录日志信息，因此当回顾这些日志时可以使用 tcpdump 的很多特点。例如，只查看与特定端口匹配的包：

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80
```

甚至可以限定具体的主机和端口：

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80 and host 192.168.1.3
```

同样的方法可以应用到直接从 pflog0 接口读取的信息：

```
# tcpdump -n -e -ttt -i pflog0 host 192.168.4.2
```

注意这与包被记录到 pflogd 日志文件不相冲突；上述语句只以包被记录的形式显示。

除了使用标准的 tcpdump（8）过滤规则外，OpenBSD 的 tcpdump 过滤语言为读取 pflogd 而被扩展：

- ip -IPv4 版本地址。
- ip6 - IPv6 版本地址。
- on int - 包通过 int 接口。
- ifname int - 与 on int 相同.
- rulenum num - 包匹配的过滤规则编号为 num。
- action act - 对包的操作。可能是 pass（通过）或者 block（阻断）。
- reason res - 执行对包操作的原因。可能的原因是 match（匹配），bad-offset, fragment, short, normalize（规格化）, memory（内存）。
- inbound -入栈包。
- outbound - 出栈包。

举例:

```
# tcpdump -n -e -ttt -i pflog0 inbound and action block and on wi0
```

这将以实时方式显示被 wi0 接口阻断的入栈包的日志信息。

3.1.4 通过 Syslog 记录日志

很多情况下需要将防火墙的日志记录以 ASCII 代码格式存储，或者（同时）把这些日志存到远程的日志服务器上。这些可以通过两个小的脚本文件实现，是对 openbsd 配置文件和 syslogd（8），日志守护进程的少许修改。Syslogd 进程以 ASCII 格式存储日志，同时可以将日志存储到远程日志服务器。

首先我们必须建立一个用户，pflogger，使用/sbin/nologin shell.最简单的建立用户的方法是使用 adduser（8）。

完成后建立如下两个脚本:

```
/etc/pflogrotate
FILE=/home/pflogger/pflog5min.$(date "+%Y%m%d%H%M")
kill -ALRM $(cat /var/run/pflogd.pid)
```

```
if [ $(ls -l /var/log/pflog | cut -d " " -f 8) -gt 24 ]; then
mv /var/log/pflog $FILE
chown pflogger $FILE
kill -HUP $(cat /var/run/pflogd.pid)
fi

/home/pflogger/pfl2sysl
for logfile in /home/pflogger/pflog5min* ; do
tcpdump -n -e -ttt -r $logfile | logger -t pf -p local0.info
rm $logfile
done
```

编辑 root 的 cron 任务：

```
# crontab -u root -e
```

增加如下两行:

```
# rotate pf log file every 5 minutes
0-59/5 * * * * /bin/sh /etc/pflogrotate
```

为用户 pflogger 建立一个 cron 任务：

```
# crontab -u pflogger -e
```

增加如下两行:

```
# feed rotated pflog file(s) to syslog
0-59/5 * * * * /bin/sh /home/pflogger/pfl2sysl
```

将下行增加到/etc/syslog.conf:

```
local0.info /var/log/pflog.txt
```

如果需要日志记录到远程日志服务器，增加：

```
local0.info @syslogger
```

确定主机 syslogger 已在 hosts（5）中定义。
建立文件/var/log/pflog.txt 使 syslog 可以向该文件写入日志：

```
# touch /var/log/pflog.txt
```

重启 syslogd 使变化生效：

```
# kill -HUP $(cat /var/run/syslog.pid)
```

所有符合标准的包将被写入/var/log/pflog.txt. 如果增加了第二行，这些信息也将被存入远程日志服务器 syslogger。

脚本/etc/pflogrotate 将执行，然后删除/var/log/pflog ，因此 rotation of pflog by newsyslog(8) 不再必需可以被禁用。然而, /var/log/pflog.txt 替代/var/log/pflog and rotation of it 要被启用。改变/etc/newsyslog.conf 如下:

```
#/var/log/pflog 600 3 250 * ZB /var/run/pflogd.pid
/var/log/pflog.txt 600 7 * 24
```

PF 将日志以 ASCII 格式记录到/var/log/pflog.txt. 如果这样配置/etc/syslog.conf, 系统将把日志存到远程服务器。存储过程不会马上发生，但是会在符合条件的包出现在文件中前 5 — 6 分钟实现。

3.2 性能

“PF 可以处理多少带宽？”“我需要多少台计算机处理因特网连接？”

这个问题没有简单的答案。对于一些应用程序来说，一台 486/66 主机，带有 2 个比较好的 ISA 网卡在做过滤和 NAT 时接近 5Mbps，但是对于其他应用程序，一个更快的计算机加上更有效的 PCI 网卡也会显得能力不足。真正的问题不是每秒处理的位数而是每秒处理的包数和规则集的复杂程度。

体现 PF 性能的几个参数:

- 每秒处理包的数量. 一个 1500 字节的包和一个只有 1 个字节的包所需要的处理过程的数目几乎是一样的。每秒处理包的数量标志着状态表和过滤规则集在每秒内被评估的次数，标志着一个系统最有效的需求（在没有匹配的情况下）。

- 系统总线性能. ISA 总线最大带宽 8MB/秒, 当处理器访问它时, 它必须降速到 80286 的有效速度，不管处理器的真实处理速度如何，PCI 总线有更有效的带宽，与处理器的冲突更小。

- 网卡的效率. 一些网卡的工作效率要高于其他网卡。基于 Realtek 8139 (rl(4)) 的网卡性能较低，而基于 Intel 21143 (dc(4)) 的网卡性能较好。为了取得更好的性能，建议使用千兆网卡，尽管所连接的网络不是千兆网，这些千兆网卡拥有高级的缓存，可以大幅提高性能。

- 规则集的设计和复杂性。规则越复杂越慢。越多的包通过 keep 和 quick 方式过滤，性能越好。对每个包的策略越多，性能越差。

- 值得一提: CPU 和内存。由于 PF 是基于内核的进程，它不需要 swap 空间。所以，如果你有足够的内存，它将运行很好，如果没有，将受影响。不需要太大量的内存。32MB 内存对小型办公室或者家庭应用足够，300MHz 的 cpu 如果配置好网卡和规则集，足够满足要求。

人们经常询问 PF 的基准点。唯一的基准是在一个环境下系统的性能。不考虑环境因素将影响所设计的防火墙的系统性能。

PF 曾经在非常大流量的系统中工作，同时 PF 的开发者也是它的忠实用户。

3.3 研究 FTP

3.3.1 FTP 模式

FTP 是一种协议，它可以追溯到因特网发展初期，那时的因特网规模小，联网的计算机彼此友好，过滤和严格安全性在那时不是必须的。FTP 设计之初就没有考虑包过滤、穿透防火墙和 NAT。

FTP 的工作模式分为被动（passive）和主动（active）两种。通常这两种选择被用来确定哪边有防火墙问题。实际上，为了方便用户你应该全部支持这两种模式。

在 active 模式下,当用户访问远程 FTP 服务器并请求一个文件信息时，那台 FTP 服务器将与该用户建立一个新的连接用来传输请求的数据，这被称为数据连接。具体过程为：客户端随机选择一个端口号，在该端口监听的同时将端口号传给服务器，由服务器向客户端的该端口发起连接请求，然后传递数据。在 NAT 后的用户访问 FTP 服务器的时候会出现问题，由于 NAT 的工作机制，服务器将向 NAT 网关的外部地址的所选端口发起连接，NAT 网关收到该信息后将在自己的状态表中查找该端口对应的内部主机，由于状态表中不存在这样的记录，因此该包被丢弃，导致无法建立连接。

在 passive 模式下（OpenBSD 的 ftp 客户端默认模式），由客户端请求服务器随机选择一个端口并在此端口监听，服务器通知客户端它所选择的端口号，等待客户端连接。不幸的是，ftp 服务器前的防火墙可能会阻断客户端发往服务器的请求信息。OpenBSD 的 ftp（1）默认使用 passive 模式；要强制改为 active 模式，使用-A 参数，或者在“ftp>”提示符下使用命令“passive off”关闭 passive 模式。

3.3.2 工作在防火墙之后的 FTP 客户端

如前所述，FTP 对 NAT 和防火墙支持不好。

包过滤机制通过将 FTP 数据包重定向到一个 FTP 代理服务器解决这一问题。这一过程将引导 FTP 数据包通过 NAT 网关/防火墙。OpenBSD 和 PF 使用的 FTP 代理是 ftp-proxy（8），可以通过在 pf.conf 中的 NAT 章节增加下列信息激活该代理：

```
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
 port 8021
```

这条语句的解释为：在内部接口上的 ftp 数据包被重定向到本机的 8021 端口

显然该代理服务器应该已在 OpenBSD 中启动并运行。配置方法为在/etc/inetd.conf 中增加下列信息：

```
127.0.0.1:8021 stream tcp nowait root /usr/libexec/ftp-proxy \
 ftp-proxy
```

重启系统或者通过下列命令发送一个‘HUP’标记来生效：

```
kill -HUP ‘cat /var/run/inetd.pid‘
```

ftp 代理在 8021 端口监听，上面的 rdr 语句也是将数据包转发到这一端口。这一端口号是可选的，因为 8021 端口没有被其他应用程序占用，因此不失为一个好的选择。

请注意 ftp-proxy（8）是用来帮助位于 PF 过滤器后的 ftp 客户端传递信息；并不用于 PF 过滤器后的 ftp 服务器。

3.3.3 PF"自保护"FTP 服务器

当 PF 运行在一个 FTP 服务器上，而不是单独的一台防火墙。这种情况下处理 passive 模式的 FTP 连接请求时 FTP 服务器将随机取一个较大的 TCP 端口接收数据。默认情况下 OpenBSD 的本地 FTP 服务器 ftpd（8）使用 49152 ～ 65535 范围内的端口，显然，必须要有对应的过滤规则放行这些端口的数据：

```
pass in on $ext_if proto tcp from any to any port 21 keep state
pass in on $ext_if proto tcp from any to any port > 49151 \
 keep state
```

如果需要可以调整上述端口范围。在 OpenBSD 的 ftpd（8）环境下，可以通过 sysctl（8）进行调整 net.inet.ip.porthifirst 和 net.inet.ip.porthilast。

3.3.4 使用 NAT 外部 PF 防火墙保护 FTP 服务器

这种情况下，防火墙必须将数据重定向到 FTP 服务器。为了讨论方便，我们假设该 FTP 服务器使用标准的 OpenBSD ftpd（8），并使用默认端口范围。

这里有个例子

```
ftp_server = "10.0.3.21"

rdr on $ext_if proto tcp from any to any port 21 -> $ftp_server \
 port 21
rdr on $ext_if proto tcp from any to any port 49152:65535 -> \
 $ftp_server port 49152:65535

# in on $ext_if
pass in quick on $ext_if proto tcp from any to $ftp_server \
 port 21 keep state
pass in quick on $ext_if proto tcp from any to $ftp_server \
 port > 49151 keep state

# out on $int_if
pass out quick on $int_if proto tcp from any to $ftp_server \
 port 21 keep state
pass out quick on $int_if proto tcp from any to $ftp_server \
 port > 49151 keep state
```

FTP 的更多信息过滤 FTP 和 FTP 如何工作的更多信息可以参考白皮书《ftp reviewed》

3.4 pf 验证: 用 Shell 进行网关验证

3.4.1 简介

Authpf(8)是身份认证网关的用户 shell。身份认证网关类似于普通网关，只不过用户必须在网关上通过身份验证后才能使用该网关。当用户 shell 被设置为/usr/sbin/authpf 时（例如，代替了 ksh（1），csh（1）等），并且用户通过 SSH 登录，authpf 将对 pf（4）策略集做必要的修改以便该用户的数据包可以通过过滤器或者（和）使用 NAT、重定向功能。一旦用户退出登录或者连接被断开，authpf 将移除加载到该用户上的所有策略，同时关闭该用户打开的所有会话。因此，只有当用户保持着他的 SSH 会话进程时他才具备透过防火墙发送数据包的能力。

Authpf 通过向附加到锚点的命名策略集增加策略来改变 pf（4）的策略集。每次用户进行身份验证，authpf 建立一个新的命名策略集，并将已经配置好的 filter、nat、binat 和 rdr 规则加载上去。被 authpf 所加载的策略可以被配置为针对单独的一个用户相关或者针对总体。

Authpf 可以应用在：

- 在允许用户访问因特网之前进行身份验证。
- 赋予特殊用户访问受限网络的权利，例如管理员。
- 只允许特定的无线网络用户访问特定的网络。
- 允许公司员工在任何时候访问公司网络，而公司之外的用户不能访问，并可以将这些用户重定向到特定的基于用户名的资源（例如他们自己的桌面）。
- 在类似图书馆这样的地方通过 PF 限制 guest 用户对因特网的访问。Authpf 可以用来向已注册用户开放完全的因特网连接。

Authpf 通过 syslogd（8）记录每一个成功通过身份验证用户的用户名、IP 地址、开始结束时间。通过这些信息，管理员可以确定谁在何时登陆，也使得用户对其网络流量负责。

3.4.2 配置

配置 authpf 的基本步骤大致描述如下。详细的信息请查看 man 手册。

将 authpf 连入主策略集

通过使用锚点策略将 authpf 连入主策略集：

```
nat-anchor authpf
rdr-anchor authpf
binat-anchor authpf
anchor authpf
```

锚点策略放入策略集的位置就是 PF 中断执行主策略集转为执行 authpf 策略的位置。上述 4 个锚点策略并不需要全部存在，例如，当 authpf 没有被设置加载任何 nat 策略时，nat-anchor 策略可被省略。

3.4.3 配置加载的策略

Authpf 通过下面两个文件之一加载策略：

- /etc/authpf/users/$USER/authpf.rules
- /etc/authpf/authpf.rules

第一个文件包含只有当用户$USER（将被替换为具体的用户名）登录时才被加载的策略。当特殊用户（例如管理员）需要一系列不同于其他默认用户的策略集时可以使用每用户策略配置。

第二个文件包含没定义自己的 authpf.rules 文件的用户所默认加载的策略。如果用户定义的文件存在，将覆盖默认文件。这两个文件至少存在其一，否则 authpf 将不会工作。

过滤器和传输策略与其他的 PF 策略集语法一样，但有一点不同：authpf 允许使用预先定义的宏：

- $user_ip - 登录用户的 IP 地址
- $user_id - 登录用户的用户名

推荐使用宏$user_ip，只赋予通过身份验证的计算机透过防火墙的权限。

3.4.4 访问控制列表

可以通过在/etc/authpf/banned/目录下建立以用户名命名的文件来阻止该用户使用 authpf。文件的内容将在 authpf 断开与该用户的连接之前显示给他，这为通知该用户被禁止访问的原因并告知他解决问题联系人提供了一个便捷的途径。

相反，有可能只允许特定的用户访问，这时可以将这些用户的用户名写入/etc/authpf/authpf.allow 文件。如果该文件不存在或者文件中输入了”*”，则 authpf 将允许任何成功通过 SSH 登录的用户进行访问（没有被明确禁止的用户）。

如果 authpf 不能断定一个用户名是被允许还是禁止，它将打印一个摘要信息并断开该用户的连接。明确禁止将会使明确允许失效。

3.4.5 将 authpf 配置为用户 shell

authpf 必须作为用户的登录 shell 才能正常工作。当用户成功通过 sshd（8）登录后，authpf 将被作为用户的 shell 执行。它将检查该用户是否有权使用 authpf，并从适当的文件中加载策略，等等。

有两种途径将 authpf 设置为用户 shell：

1. 为每个用户手动使用 chsh(1), vipw(8), useradd(8), usermod(8),等。
2. 通过把一些用户分配到一个登录类，在文件/etc/login.conf 中改变这个登录类的 shell 属性

3.4.6 查看登陆者

一旦用户成功登录，并且 authpf 调整了 PF 的策略，authpf 将改变它的进程名以显示登录者的用户名和 IP 地址：

```
# ps -ax | grep authpf
```

在这里用户 chalie 从 IP 地址为 192.168.1.3 的主机登录。用户可以通过向 authpf 进程发送 SIGTERM 信号退出登录。Authpf 也将移除加载到该用户上的策略并关闭任何该用户打开的会话连接。

```
# kill -TERM 23664
```

更多信息，请查询 man 手册。

3.4.7 实例

OpenBSD 网关通过 authpf 对一个大型校园无线网的用户进行身份验证。一旦某个用户验证通过，假设他不在禁用列表中，他将被允许 SSH 并访问网页（包括安全网站 https），也可以访问该校园的任一个 DNS 服务器。

文件/etc/authpf/authpf.rules 包含下列策略：

```
wifi_if = "wi0"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
 port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
 https } flags S/SA keep state
```

管理员 charlie 除了网页冲浪和使用 SSH 外还需要访问校园网的 SMTP 和 POP3 服务器。下列策略被配置在/etc/authpf/users/charlie/authpf.rules 中:

```
wifi_if = "wi0"
smtp_server = "10.0.1.50"
pop3_server = "10.0.1.51"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
 port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to $smtp_server \
 port smtp flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to $pop3_server \
 port pop3 flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
 https } flags S/SA keep state
```

定义在/etc/pf.conf 中的主策略集配置如下：

```
# macros
wifi_if = "wi0"
ext_if = "fxp0"

scrub in all

# filter
block drop all

pass out quick on $ext_if proto tcp from $wifi_if:network flags S/SA \
modulate state
pass out quick on $ext_if proto { udp, icmp } from $wifi_if:network \
keep state

pass in quick on $wifi_if proto tcp from $wifi_if:network to $wifi_if \
port ssh flags S/SA keep state

anchor authpf in on $wifi_if
```

该策略集非常简单，作用如下：

- 阻断所有（默认拒绝）。
- 放行外部网卡接口上的来自无线网络并流向外部的 TCP,UDP 和 ICMP 数据包。
- 放行来自无线网络，目的地址为网关本身的 SSH 数据包。该策略是用户登录所必须的。
- 在无线网络接口上为流入数据建立锚点“authpf”。

设计主策略集的主导思想为：阻断任何包并允许尽可能小的数据流通过。在外部接口上流出数据包是允许的，但是默认否策略阻断了由无线接口进入的数据包。用户一旦通过验证，他们的数据包被允许通过无线接口进入并穿过网关到达其他网络。

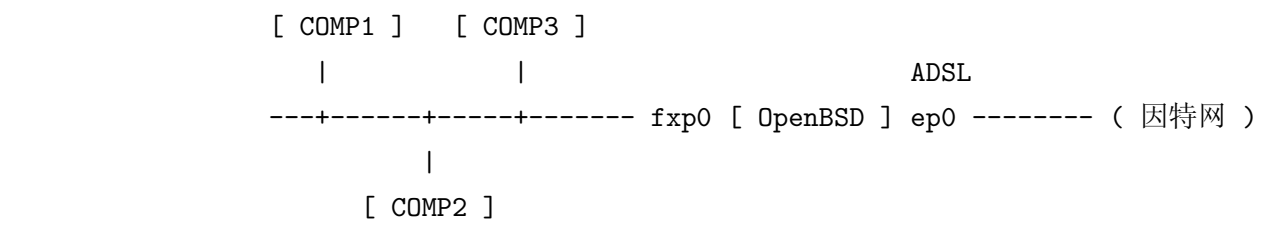
3.5 实例：家庭和小型办公室防火墙

3.5.1 概况

在这个例子中，PF 作为防火墙和 NAT 网关运行在 OpenBSD 机器上，为家庭或办公室的小型网络提供服务。总的目标是向内部网提供因特网接入，允许从因特网到防火墙的限制访问。下面将详细描述：

3.5.2 网络

网络配置如下:



内部网有若干机器，图中只划出了 3 台，这些机器除了 COMP3 之外主要进行网页冲浪、电子邮件、聊天等；COMP3 运行一个小型 web 服务器。内部网使用 192.168.0.0/24 网段。

OpenBSD 网关运行在 Pentium 100 计算机上，装有两块网卡：一个 3com 3c509B（ep0），另一个 Intel EtherExpress Pro/100（fxp0）。该网关通过 ADSL 连接到因特网，同时通过 NAT 向内网共享因特网连接。外部网卡的 IP 地址动态分配。

3.5.3 目标

1. 向内部网络的每台计算机提供无限制的因特网接入。
2. 启用一条“默认拒绝”策略。
3. 允许下列来自因特网的请求访问防火墙：
   - SSH (TCP 端口 22): 用来远程维护防火墙。
   - Auth/Ident (TCP 端口 113): SMTP 和 IRC 服务用到的端口。
   - ICMP Echo Requests: ping（8）用到的 ICMP 包类型。
4. 重定向访问 80 端口（访问 web 的请求）的请求到计算机 COMP3，同时，允许指向 COMP3 计算机的 80 端口的数据流过防火墙。
5. 记录外部网卡接口上的过滤日志。
6. 默认为阻断的包返回一个 TCP RST 或者 ICMP Unreachable 信息。
7. 尽量保持策略集简单并易于维护。

3.5.4 准备

这里假设作为网关的 OpenBSD 主机已经配置完成，包括 IP 网络配置，因特网连接和设置 net.inet.ip.forwarding 的值为 1。

3.5.5 规则集

下面将逐步建立策略集以满足上诉要求。

宏

定义下列宏以增强策略集的可维护性和可读性：

```
int_if = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"

comp3 = "192.168.0.3"
```

前两行定义了发生过滤的网络接口。第 3、4 行定义了向因特网开放的服务端口号（SSH 和 ident/auth）和允许访问防火墙的 ICMP 包类型。第 5 行定义了回送地址段和 RFC1918 定义的私有地址段。最后一行定义了主机 COMP3 的 IP 地址。

注意：如果 ADSL 接入因特网需要 PPPoE，则过滤和 NAT 将发生在 tun0 接口上而不是 ep0 接口。

选项

下列两个选项用来设置阻断后的默认操作为反馈，并在外部接口设置开启日志记录：

```
set block-policy return
set loginterface $ext_if
```

流量整修

没有理由不起用对所有进入防火墙的所有包进行规格化，因此只需要简单的一行：

```
scrub in all
```

NAT（网络地址转换）

为所有内部网启用 NAT 可以通过下列策略：

```
nat on $ext_if from $int_if:network to any -> ($ext_if)
```

由于外部网卡的 IP 地址是动态获得的，因此在外部网卡接口处增加小括号以使当 IP 地址发生变化时 PF 可以自适应。

重定向

第一个需要重定向策略的是 ftp-proxy（8），只有这样内部网上的 FTP 客户端才可以访问因特网上的 FTP 服务器。

```
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 port 8021
```

注意这条策略只捕获到 21 端口的数据包，如果用户通过其他端口访问 FTP 服务器，则在定义目的端口时需要使用 list（列表），例如：from any to any port 21, 2121 。

第二个重定向策略捕获因特网上的用户访问防火墙 80 端口的数据包。用户试图访问网络的 web 服务器时将产生合法的访问该端口的数据包，这些连接请求需要重定向到主机 COMP3:

```
rdr on $ext_if proto tcp from any to any port 80 -> $comp3
```

过滤规则

过滤规则第一行是默认否规则：

```
block all
```

这时没有任何数据包可以流过防火墙，甚至来自内部网络的数据包。下面的规则将逐个依据上面提到的目标开启防火墙上的虚拟接口。

每个 Unix 系统都有一个"loopback（回送）"接口，它是用于系统中应用程序间通信的虚拟网络接口。在 OpenBSD 中，回送接口是 lo（4）。

```
pass quick on lo0 all
```

下一步，由 RFC 1918 定义的私有地址将在外部网卡接口的进和出方向被阻断。这些地址不应该出现在公网上，通过阻断这些地址可以保证防火墙不向外部网泄漏内网地址，同时也阻断了来自外部网中源地址为这些私有地址的数据包流入内网。

```
block drop in quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets
```

这里 block drop 用来通知 PF 停止反馈 TCP RST 或者 ICMP Unreachabel 数据包。因为 RFC 1918 规定的地址不会存在于因特网上，发往那些地址的数据包将没有意义。Quick 选项用来通知 PF 如果这条规则匹配则不再进行其他规则的匹配操作，来自或流向$priv_nets 的数据包将被立即丢弃。

现在将打开因特网上的一些服务所用到的端口：

```
pass in on $ext_if inet proto tcp from any to ($ext_if) \
 port $tcp_services flags S/SA keep state
```

通过在宏$tcp_services 中定义服务端口可以更方便的进行维护。开放 UDP 服务也可一模仿上述语句，只不过改为 proto udp。

已经有了一条 rdr 策略将 web 访问请求转发到主机 COMP3 上，我们必须建立另一条过滤规则使得这些访问请求可以通过防火墙：

```
pass in on $ext_if proto tcp from any to $comp3 port 80 \
 flags S/SA synproxy state
```

考虑到安全问题，我们使用 TCP SYN Proxy 保护 web 服务器——synproxy state。

现在将允许 ICMP 包通过防火墙：

```
pass in inet proto icmp all icmp-type $icmp_types keep state
```

类似于宏$tcp_services，当需要增加允许进入防火墙的 ICMP 数据包类型时可以容易地编辑宏$icmp_types。注意这条策略将应用于所有网络接口。

现在数据流必须可以正常出入内部网络。我们假设内网的用户清楚自己的所作所为并且确定不会导致麻烦。这并不是必然有效的假设，在某些环境下更具限制性的策略集会更适合。

```
pass in on $int_if from $int_if:network to any keep state
```

上面的策略将允许内网中的任何计算机发送数据包穿过防火墙；然而，这并没有允许防火墙主动与内网的计算机建立连接。这是一种好的方法吗？评价这些需要依靠网络配置的一些细节。如果防火墙同时充当 DHCP 服务器，它需要在分配一个地址之前 ping 一下该地址以确认该地址没有被占用。允许防火墙访问内部网络同时也允许了在因特网上通过 ssh 控制防火墙的用户访问内网。请注意禁止防火墙直接访问内网并不能带来高安全性，因为如果一个用户可以访问防火墙，他也可以改变防火墙的策略。增加下列策略可以使防火墙具备访问内网的能力：

```
pass out on $int_if from any to $int_if:network keep state
```

如果这些策略同时存在，则 keep state 选项将不是必须的；所有的数据包都可以流经内网接口，因为一条策略规定了双向放行数据包。然而，如果没有 pass out 这条策略时，pass in 策略必须要有 keep state 选项。这也是 keep state 的有点所在：在执行策略匹配之前将先进行 state 表检查，如果 state 表中存在匹配记录，数据包将直接放行而不比再进行策略匹配。这将提高符合比较重的防火墙的效率。

最后，允许流出外部网卡接口的数据包通过防火墙

```
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```

TCP, UDP, 和 ICMP 数据包将被允许朝因特网的方向出防火墙。State 信息将被保存，以保证反馈回来的数据包通过防火墙。

3.5.6 完整规则集

```
# macros
int_if = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"
```

```
comp3 = "192.168.0.3"

# options
set block-policy return
set loginterface $ext_if

# scrub
scrub in all

# nat/rdr
nat on $ext_if from $int_if:network to any -> ($ext_if)
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
port 8021
rdr on $ext_if proto tcp from any to any port 80 -> $comp3

# filter rules
block all

pass quick on lo0 all

block drop in quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets

pass in on $ext_if inet proto tcp from any to ($ext_if) \
port $tcp_services flags S/SA keep state

pass in on $ext_if proto tcp from any to $comp3 port 80 \
flags S/SA synproxy state

pass in inet proto icmp all icmp-type $icmp_types keep state

pass in on $int_if from $int_if:network to any keep state
pass out on $int_if from any to $int_if:network keep state

pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```