TODO – source based routing

Note: we could use ipt_ROUTE's support of the –tee option (or ULOG) to copy all traffic into the virtual tunnel...)

TBD – The preferred way to impl. pinhole/ VIP to subnet mappings.

- Do nothing, just support mission defined specific pinholes and map based on FT ID
  con: requires more extensive mission config managment.
- route -net 192.168.1.0/24 will only work for one FT tunnel on the proxy server
- route -net 10.1.2.0/24 only works if the FT supports -j NETMAP or for a single dest. IP
  note: 10.X.X.0 could be defined by FT id in CT database.

  Idea: translate 10.X.X.0/24 to correct IP before placing in the tunnel? Note that this approach assumes a class C(24 bit) ipv4 subnet...

  How to support a non 24 bit subnet: specify the subnet in the mission so that the client can create the correct VIP to local IP network translation.

- Have mm generate a DNAT rule for each client on the subnet? - e.g. every LAN entry in /proc/net/arp? No, only works if the host was previously active... Generate an entry on the fly? e.g. send a telnet cmd over the tunnel to telnetd? Security?

Proxy & Pinhole production technical hurdles:

1. selectively bypass or override filter table
   solution: ipt_pfilter
2. source based routing on FT -
   iptables ROUTE table is an option... and we can ad it to a pmangle table to avoid any other system routes conflicts... perhaps we want to impl. other route commands this way too?
   See ipt_ROUTE target impl. - this must be performed in the mangle table:

   Ideas on integration of ipt_ROUTE iptables_mangle.c - without configuring from iptables:

- 1 requires registration of mangle hooks with the same priority as the mangle table in each hook:

  a) we would have to either expose a function from gf module so that we could determine a match
  or

  b) we could mark the packet

- 2 requires configuration of a “struct ipt_route_target_info” that is normally recieved from iptables as part of the target options
  a) we could hardcode a static config and expose a set__ipt_route_target_info accessor
  b) we could provide the config as part of the target action lookup (see opt 1a)

c) we're REALLY just dependent on openvpn configuring tun0

EXPORT_SYMBOL(ipt_do_table);

on Linux: the standard solution is iproute2/iproute
-/sbin/ip ~ 173K on FC5 iproute-2.6.15-1.2
-/sbin/tc ~ 216K, On WRT45G /usr/sbin/tc is 228.2k

from man iptables:

```
We have one classifier rule:

             tc filter add dev eth3 parent 1:0 protocol ip fw

       Earlier we had many rules just like below:

              iptables -t mangle -A POSTROUTING -o eth3 -d 192.168.5.2 -j
MARK --set-mark 0x10502

              iptables -t mangle -A POSTROUTING -o eth3 -d 192.168.5.3 -j
MARK --set-mark 0x10503

      Using IPMARK target we can replace all the mangle/mark rules with only
one:

             iptables -t mangle -A  POSTROUTING  -o  eth3  -j  IPMARK
--addr=dst  --and-mask=0xffff  --or-
             mask=0x10000
```

Best solution: just manipulate the kernel's routing table directly...

ipt_pfilter – allows for bypass of rules around filter table
A selective higher priority bypass of the filter table with a higher priority table
Concept: walk the packet around the filter table

1. disable filter hook by replacing hook function registration with dummy (debug funcs)
2. re-enable the filter table. find the filter table by iterating over pfilter's packet_filter.list and searching for the name
3. call the table's rules directly after processing local pfilter table

forward pinhole over a openvpn or socat tun interface.

Create a virtual IP mapping VIP to pinhole mapping (does not need to be known to the end user)
e.g. 10.1.2.3

```
#on the server
#route traffic to the IP through the tun interface
route add -host 10.1.2.3 dev tun0
#idea may need to rout to tun ip endpoint, note that both are sent through
tunnel...
route add -host 10.1.2.3 gw 10.129.129.1

#on the client
#DEST NAT traffic to that IP to  the PIN_IP:PIN_PORT
#note: it should be possible to leave out the protocol option
iptables -t nat -A PREROUTING -p tcp -d 10.1.2.3 -j DNAT --to ${PIN_IP}

#note: don't forget to masq output to FT's LAN IP
# this step seems to work for inbound requests, but might be what is breaking
initial outbound connections from the PIN_IP. e.g. 192.168.1.128
# on 2.6 kernels/margarita
iptables -t nat -A POSTROUTING -o eth1 -s 10.129.66.1 -j SNAT --to-source
192.168.1.12
# on 2.4 FT(s)
iptables -t nat -A POSTROUTING -p tcp -s 10.129.66.1 -j MASQUERADE
iptables -t nat -A POSTROUTING -p udp -s 10.129.66.1 -j MASQUERADE

# iptables -t nat -A POSTROUTING -p udp -s 10.129.66.1 -o br0 -j MASQUERADE
# iptables -t nat -A POSTROUTING -p tcp -s 10.129.66.1 -o br0 -j MASQUERADE
# example of outbound
 client based rule
# iptables -t nat -A POSTROUTING -p tcp -s 192.168.1.128 -o tun0 -j MASQUERADE
# iptables -t nat -A POSTROUTING -p udp -s 192.168.1.128 -o tun0 -j MASQUERADE


#make sure the traffic isn't dropped by any other firewall rules
#TODO add interface or src criteria to rule to further lock down
iptables -t filter -I FORWARD 1 -d 192.168.1.128 -j ACCEPT
```

Access to FT local interface, from the proxy server

```
# on 2.6 kernels/margarita
iptables -t filter -I RH-Firewall-1-INPUT 1 -s 10.129.66.1 -j ACCEPT

# on 2.4 FT(s)
iptables -t filter -I INPUT 1 -s 10.129.66.1 -j ACCEPT





#the following is an alternative strategy that doesn't work yet
route add -host 10.1.2.1 gw 10.129.129.1
# on the FT
# this rule is not getting any packets, even without the -i tun0 option
iptables -t filter -I INPUT -i tun0 -s 10.129.66.1 -d 127.0.0.1 -j ACCEPT
#this rule is being applied, but packet lost
iptables -t nat -A PREROUTING -d 10.1.2.1 -j DNAT --to 127.0.0.1

#this rule doesn't quite work yet either
iptables -t nat -A PREROUTING -d 10.1.2.1 -j REDIRECT
```

Forward Pinhole Requirements and Options

Forwards an inbound packet sent to a configurable port on the WAN interface of the FT to an *internal address and port*.

This could be impl. as a network address translation (NAT) function mapping between a FT's *{external address, external port}* tuple and a specified *{internal address, internal port}* tuple.

As an alternative, TCP and UDP support could be implemented as a proxy socket that accepts connections on a FT's WAN interface:port and connects to a specified *internal address, internal port.*

Options, for each option letter we must choose at least one # to provide a working impl:

A) packet IP source validation

1. forward all inbound packets regardless of source IP
   or
2. only forward inbound packets that match a specific IP address/mask or range

B) NAT of inbound packet source IP

1. none (straight port forward does not hide source IP) – this is a more pure form
   or
2. change the source IP to the FT's LAN IP (essentially a dual NAT translation on the packet)
   - N/A to non-router AP's

C) pinhole duration

1. mission
   or
2. mission start + timeout
   or
3. window

- connection timeout (device/linux/impl. Specific keep alive workarounds?)

D) kernel (netfilter based) or application layer pinhole impl

1. netfilter based impl.
   Pros
   impl. could be as simple as applying current proxy code to inbound connections

   Cons
   ?

2. Application layer impl:
   Pros
   - Could be as simple as mission cmd exec support. e.g. exec netcat –args
   -simple impl. leads to control of alternative tools that may be available on some FTs

   Cons
   - not as reliable and as good of performance as NAT translation (marginal considering use cases)

- may be easier to provide an application layer impl on some devices
- even application layer impl. still requires firewall rule checking/override
- Note: we must override existing firewall config specified by a user on the FT-- which means that we'd have to either modify IP tables periodically or place our hooks in before iptables can filter out/drop the packets.

E) Support to forward to FT LAN IP or localhost

1. Required
2. If it works for a FT without effort, great.
3. No support to forward to FT LAN IP or FT localhost

Assumptions:

- no validation or security on pinhole
  i.e. a packet from ANY IP would be forwarded through the firewall (option A-1)
  or
  the 'right' IP (option A-2)
- must be able to configure a FT's firewall to open up the hole without interfering with the FT's normal firewall configuration.

Reverse Pinhole

Note: We already have a form of reverse pinhole, but is for all outbound traffic for a specific IP. I don't think that it will be difficult to perform a NAT translation from a outbound *destination IP*, that corresponds to one of the following options, to a translated outside IP and port. Where the *destination IP is:*

1. the FT's LAN IP
   or
2. an arbitrary IP/network mask
   or
3. a domain, (requires looking into DNS requests).

Hole punching:

Technique of establishing and outbound TCP connection or UDP packet (or UDP handshake) in order to open up a hole in Firewalls between a non public addressable FT and an outside destination IP. This is useful in order to

Pinhole Estimate of Work

This estimate is for kernel/netfilter based impl (Option D-1) on FT's that already support HTTP Proxy (i.e. outbound NAT dest IP translation)

Firewall Manipulation:

We will have to open up the firewall to allow for inbound or outbound pinhole (packet with original IP destination must be allowed, regardless of user configured firewall).
As simple as programatically (preferred) or via an exec on the cmd line, adding an accept rule to every IP table. This estimate is for support on one device FT, FT's with a similar kernel would also likely be supported with little or no effort.

netfilter firewall configuration:

1-2 weeks

static/hardwired pinhole test for forward and reverse pinhole assuming options A-1, B-1, C-1:

2-5 days

UDP support (depends on previous TCP test working)

1 day

mission protocol configuration/support:

1-2 days

CW support:

target action/activated pinhole: + 5-10 days
static mission FW pinhole: 2-4 days if added to generic MissionProperties page

Note: sponsor has not requested Reverse Pinhole (RP)
domain-> IP pinholing / redirection (option RP-3): + 3-5 days
global/mission (option RP-1,2) : +2 days

Testing:

1 week minimum

Windex Connection Negotiation over HTTPS

Main Requirement: application layer proxy app (hereto referred as '**wxpx**') that servers as a HTTP connection proxy and can hand off a connection to windex. See white board for pseudo code impl. stages.

Significant Development tasks:

1. HTTP request and response inspection and domain lookup (already have kernel level functions for HTTP request and response inspection)
   or

   Including destination IP (port optional) in packet before sending over local interface or over NETLINK. If forwarding multiple connections from **gf**, local interface is preferred.

2. SSL support (many libs available, we need to choose a small footprint one that is portable to most Fts)

   - use libtomnet ... not mature enough...
   - netcat SSL patch... good example of patch to netcat that adds SSL support:

     patch_netcat_ssl-20040224.diff, based on OpenSSL support

       #include <openssl/err.h>

       #include <openssl/x509v3.h>

   -see http://xyssl.org/docs/ for alternative SSL impl.s, e.g.: peersec 50K

3. **wxpx** app communication IO with **gf**:

   - **wxpx** notify ready (after starting from signal to mm from **gf** after target acquisition)

   - **wxpx** notify success/failure (we could restrict **gf** to only forward one connection per client until success)

–

Transparent (TCP / UDP) Proxy

Proxy traffic from a client to remote proxy server.

TODO Research using tun interface to setup a forward pinhole

A) Traffic Types to proxy
Exclusions: Do not proxy TCP established connections and ICMP inbound (TTL traceroute replys)

1. All protocols
2. Mission configurable protocols and TCP or UDP ports
   Proxy all ports or a finite number of ports

B) Proxy transport
Assumption: There is no need to encrypt the traffic, it will only raise a red flag.

1. Application Layer/User Space Tunnel
   Pros: unencrypted ppp tunnel over TCP 80 shouldn't raise too many flags, and avoid FW issues between the FT and the proxy server/router.
   Note: unencrypted ppp can run over a telnet tty, see

   Cons: why not just use something like a app layer PPP proxy?
   http://www.tldp.org/HOWTO/ppp-ssh/forwarding.html
   http://www.netfilter.org/documentation/HOWTO/NAT-HOWTO-6.html#ss6.1
   Q: General Problem: How should DNS requests be handled?
   A: We may have to map or translate all outbound DNS request dest IPs to the proxy server's nameserver.
   Q: What about DHCP requests from the client?
   A: We should try to prevent them from being sent through the VPN. From the openvpn howto:

   "Many OpenVPN client machines connecting to the internet will periodically interact with a DHCP server to renew their IP address leases. The **redirect-gateway** option might prevent the client from reaching the local DHCP server (because DHCP messages would be routed over the VPN), causing it to lose its IP address lease."

   On FT/ProxyServer (note: socat generally has support for TCP4, SSL, or UDP channels...)
   telnet and pppd / telnetd and pppd
   http://www.imonk.com/jason/hacks/
   socat and pppd / socat and pppd
   socat and pppd / socat and slirp
   http://www.unix-tutorials.com/go.php?id=466
   ssh and pppd / sshd and pppd
   socat (requires kernel support for TUN socket/network interface, deps: openssl)
   openvpn (deps: lzo openssl)

http://www.dest-unreach.org/socat/doc/socat-tun.html

binary sizes:
openvpn on i386 358k, lzo 63k
socat 261 k

Note: if we roll our own client and server we could auth with CT protocols, and support either clear text or encrypted comms. e.g. CT accepts connection, authenticates, determines proxy type (socat, openvpn), and then pipes data to protected proxy server (either local or remote).

Note: openvpn doesn't allow –ifconfig-pool option on server without --mode server -> --tls-server.... could try –ifconfig-push

Socat build:
http://www.openembedded.org/repo/org.openembedded.dev/packages/socat/socat_1.3.2.1.bb

From http://www.linuxjournal.com/article/1174
"However, although both are available under Linux, I highly recommend using PPP instead of SLIP, for the following reasons:

- PPP is an Internet Standard Protocol—this means that it has undergone a standardization process approved by the Internet Architecture Board (IAB) and is an official part of the Internet Protocol Suite. SLIP, by contrast, is an "Internet non-standard" and is not on the standard track.
- PPP will work over some connections that are not 8-bit-transparent; SLIP will not.
- PPP can support authentication, peer address negotiation, packet header compression, and point-to-point error correction; SLIP can support none of these (although Compressed SLIP, or CSLIP, does support packet header compression).

"

2. VPN tunnels (e.g. route all outbound client traffic through a IPSEC, authenticated header tunnel)
   VPN kernel support likely limited on some FTs, may require a significant amount of image space.

   IPSEC – requires pre-shared key or cert, or radius server auth

   PPTP – sends regular PPP session with GRE, requires two network sessions
   "The system uses TCP (i.e., port 1723) to send the PPTP control channel packets. On the data channel, PPTP uses a protocol called Generic Routing Encapsulation (GRE—IP protocol number 47) to securely encapsulate the Point-to-Point Protocol (PPP) packets in an IP packet."

   `pptpclient.sourceforge.net`

```
# OpenWRT notes
### Allow PPTP control connections from WAN
iptables -t nat -A prerouting_rule -i $WAN -p tcp --dport 1723 -j
ACCEPT
```

```
iptables        -A input_rule      -i $WAN -p tcp --dport 1723 -j
ACCEPT
### Allow GRE protocol (used by PPTP data stream)
iptables        -A output_rule             -p 47               -j
ACCEPT
iptables        -A input_rule              -p 47               -j
ACCEPT
```

Note: we need to do IP src bassed routing, i.e. Iptables support for the ROUTE target in order to direct traffic to the pptpd localip. Note: any firewall between the FT and the proxy server must allow GRE packets (protocol 47).

General Problem:

1) any protocol that embeds the client's IP or negotiates subsequent connections on a new port will require a special proxy server impl. (and firewall reconfiguration on the FT) or it will fail.
2) True IP Transparency can only be achieved when the proxy server is a MITM between the client and the destination server (the FT is, but the remote proxy server likely isn't).

Protocols that would break under a simple proxy:
FTP, SIP (most VOIP) requires a specialized proxy

Best solution: allow for configurable ports to proxy out and in (inbound ports require altering firewall) allow for ability to save port configs and name: e.g. FTP proxy, VOIP proxy, etc. One way of looking at this is as a control message sent from the proxy server to the FT that registers pinouts on the fly.

Therefore, the best temporary solution might be to only proxy specific types of traffic or ports in a static mission configuration (pinouts).

```
# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.129.66.1     *               255.255.255.255 UH    0      0        0 tun0
192.168.1.0     *               255.255.255.0   U     0      0        0 br0
10.1.1.0        *               255.255.255.0   U     0      0        0 vlan1
192.12.16.0     10.1.1.1        255.255.255.0   UG    0      0        0 vlan1
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
default         10.129.66.1     0.0.0.0         UG    0      0        0 tun0
```

VPN Testing Barebone Notes/Howto:

```
#on client/FT
# 5.4.16.104 is the IP of the proxy / VPN server
#10.129.66.0/24 is an arbitrary virtual IP for the server, but chosen to hopefully not conflict with any other private network
insmod tun.o
openvpn --proto tcp-client --remote 5.4.16.104 8080 --dev tun0 --ifconfig 10.129.129.1 10.129.66.1 --verb 5 --ping 30

#setup up client to use tunnel as default gw
route add -net 5.4.16.0 netmask 255.255.255.0 gw 10.1.1.1
route add default gw 10.129.66.1

#note: iptables v1.2.7a doesn't support SNAT and MASQ only works for tcp or udp
# udp entry will forward DNS requests
#this doesn't work  since MASQ only takes the –to-ports arg and not a dest IP
iptables -t nat -A POSTROUTING  -o tun0 -j SNAT --to 10.129.129.1
#or for a single target on a FT
iptables -t nat -I POSTROUTING 1 -s 192.168.1.128 -o tun0 -j MASQUERADE
#TBD does this work? Yes... but if not applied before a client creates an existing ct_contrack entry then it won't work!
#therefore, this rule must be applied before ipt_ROUTE_gfint module loads and starts forwarding connections
 iptables -t nat -I POSTROUTING 1 -o tun0 -j MASQUERADE

# 1-2-08 testing: (simplified)
# iptables -t nat -A POSTROUTING -s 10.129.66.1 -o br0 -j MASQUERADE
# iptables -t nat -I POSTROUTING 1 -s 192.168.1.128 -o tun0 -j MASQUERADE


#Troubleshooting:  make sure that your firewall allows the traffic (this is typically not a problem on FTs since they are setup to route anyway)
# e.g. where 128 is your client/target
#note: do not confuse this example of a desktop that normally doesn't forward with a FT that does not
# normally need such a rule (should already be allowed)
#iptables -t filter -I FORWARD 1 -s 192.168.1.128 -j ACCEPT

#on proxy server (5.4.16.104)
#note –remote required since we aren't in multi-client mode... --mode=server requires TLS
sudo /usr/sbin/openvpn --remote 5.4.16.62 --proto tcp-server --port 8080 --dev tun --ifconfig 10.129.66.1 10.129.129.1 --ping 30 --user cbuser --group cbgroup --persist-key --verb 4

#enable forwarding
echo "1" > /proc/sys/net/ipv4/ip_forward
```

```
# enable NAT for TUN traffic
iptables -t nat -A POSTROUTING -s 10.129.0.0/16 -o eth0 -j SNAT --to 5.4.16.104
```

12-26-07 notes:
I had to apply the routing rules and then restart openvpn before I could ping both VIPs from the FT's cmd line. I can only ping 10.129.66.1 from 5.4.16.104 (the server).
Since “-j MASQUERADE” doesn't support icmp, outbound pings do not have their SRC IP altered on the server, and a reply will never be returned. Ideally, we would just use a -j DNAT rule with newer versions of iptables.

```
#socat notes

#client/FT, note: add up to 4 -d's for more debug
insmod tun.o
./socat -d tcp:5.4.16.104:11443 tun:10.0.0.2/24,up &

#server
./socat -d -d TCP-LISTEN:11443,reuseaddr TUN:10.0.0.1/24,up &
```

Proxy Tunnel Uses

1. proxy target traffic
2. provide forward pinhole from server
3. provides a routable virtual IP or port to FT from the proxy server (in the case where the FT does not have a public IP.
4. provide means to access FT web interface to reflash
5. provide means to access telnetd (if started from mm and there is a iptables rule that only allows access to port 23 from the localhost traffic)
6. provide a means to transfer additional tools and libraries to the FT. e.g. libssl, dropbear, Jeremy's routing app.

MM enhancements

1. add mission exec capability
2. add attachement capability
   allows mm to exec attachement after written

Post test dump to iptables on FT2:

```
# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  0.0.0.0/0            192.168.1.0/24
DNAT       icmp --  0.0.0.0/0            10.1.1.123          to:192.168.1.2
TRIGGER    all  --  0.0.0.0/0            10.1.1.123          TRIGGER type:dnat
match:0 relate:0
DNAT       tcp  --  0.0.0.0/0            10.1.2.3            to:192.168.1.128

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  0.0.0.0/0            0.0.0.0/0
MASQUERADE  all  --  192.168.1.0/24       192.168.1.0/24
MASQUERADE  tcp  --  192.168.1.128        0.0.0.0/0
MASQUERADE  udp  --  192.168.1.128        0.0.0.0/0
MASQUERADE  udp  --  10.129.66.1          0.0.0.0/0
MASQUERADE  tcp  --  10.129.66.1          0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  10.129.66.1          0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0           state INVALID
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state
RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state NEW
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state NEW
DROP       icmp --  0.0.0.0/0            0.0.0.0/0
DROP       2    --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  0.0.0.0/0            192.168.1.128
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0           state INVALID
TCPMSS     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp flags:0x06/0x02
tcpmss match 1461:65535TCPMSS set 1460
TRIGGER    all  --  0.0.0.0/0            0.0.0.0/0           TRIGGER type:in
match:0 relate:0
trigger_out  all  --  0.0.0.0/0            0.0.0.0/0
lan2wan    all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state
RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           state NEW
DROP       all  --  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain advgrp_1 (0 references)
target     prot opt source               destination
```

```
Chain advgrp_10 (0 references)
target     prot opt source               destination

Chain advgrp_2 (0 references)
target     prot opt source               destination

Chain advgrp_3 (0 references)
target     prot opt source               destination

Chain advgrp_4 (0 references)
target     prot opt source               destination

Chain advgrp_5 (0 references)
target     prot opt source               destination

Chain advgrp_6 (0 references)
target     prot opt source               destination

Chain advgrp_7 (0 references)
target     prot opt source               destination

Chain advgrp_8 (0 references)
target     prot opt source               destination

Chain advgrp_9 (0 references)
target     prot opt source               destination

Chain grp_1 (0 references)
target     prot opt source               destination

Chain grp_10 (0 references)
target     prot opt source               destination

Chain grp_2 (0 references)
target     prot opt source               destination

Chain grp_3 (0 references)
target     prot opt source               destination

Chain grp_4 (0 references)
target     prot opt source               destination

Chain grp_5 (0 references)
target     prot opt source               destination

Chain grp_6 (0 references)
target     prot opt source               destination

Chain grp_7 (0 references)
target     prot opt source               destination

Chain grp_8 (0 references)
target     prot opt source               destination

Chain grp_9 (0 references)
target     prot opt source               destination

Chain lan2wan (1 references)
```

```
target     prot opt source               destination

Chain logaccept (0 references)
target     prot opt source               destination
LOG        all  --  0.0.0.0/0            0.0.0.0/0          state NEW LOG flags 7
level 4 prefix `ACCEPT '
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0

Chain logdrop (0 references)
target     prot opt source               destination
LOG        all  --  0.0.0.0/0            0.0.0.0/0          state NEW LOG flags 7
level 4 prefix `DROP '
DROP       all  --  0.0.0.0/0            0.0.0.0/0

Chain logreject (0 references)
target     prot opt source               destination
LOG        all  --  0.0.0.0/0            0.0.0.0/0          LOG flags 7 level 4
prefix `WEBDROP '
REJECT     tcp  --  0.0.0.0/0            0.0.0.0/0          tcp reject-with tcp-
reset

Chain trigger_out (1 references)
target     prot opt source               destination
```

1-2-08 test results:

Post test dump of iptables on marg:

```
-bash-3.1# iptables -t nat -L -n -v
Chain PREROUTING (policy ACCEPT 423K packets, 35M bytes)
 pkts bytes target     prot opt in     out     source               destination
   20  1200 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'
    2   120 DNAT       tcp  --  *      *       0.0.0.0/0            10.1.2.3
to:192.168.1.128

Chain POSTROUTING (policy ACCEPT 17230 packets, 1035K bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'
   38  2270 SNAT       all  --  *      tun0    192.168.1.128        0.0.0.0/0
to:10.129.129.1
    2   120 SNAT       all  --  *      eth1    10.129.66.1          0.0.0.0/0
to:192.168.1.12

Chain OUTPUT (policy ACCEPT 17446 packets, 1051K bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'
-bash-3.1# iptables -t filter -L -n -v
```

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    4   204 LOG        tcp  --  *      *       0.0.0.0/0            127.0.0.1
tcp dpt:8443 LOG flags 0 level 4
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.1
LOG flags 0 level 4
    0     0 LOG        all  --  *      *       10.129.66.1          127.0.0.1
LOG flags 0 level 4
   41  6407 LOG        all  --  *      *       10.129.66.1          0.0.0.0/0
LOG flags 0 level 7 prefix `cb'
    0     0 ACCEPT     all  --  tun0   *       10.129.66.1          127.0.0.1
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'
4352K 1089M RH-Firewall-1-INPUT  all  --  *      *       0.0.0.0/0
0.0.0.0/0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
  120  9648 ACCEPT     all  --  *      *       10.129.66.1          0.0.0.0/0
11584 8414K LOG        all  --  *      *       192.168.1.128        0.0.0.0/0
LOG flags 0 level 7 prefix `cb'
 5487  898K LOG        all  --  *      *       0.0.0.0/0            192.168.1.128
LOG flags 0 level 7 prefix `cb'
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'
  524 42932 ACCEPT     all  --  *      *       192.168.1.128        0.0.0.0/0
16660 9284K RH-Firewall-1-INPUT  all  --  *      *       0.0.0.0/0
0.0.0.0/0

Chain OUTPUT (policy ACCEPT 4317K packets, 1162M bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 LOG        all  --  *      *       0.0.0.0/0            10.1.2.3
LOG flags 0 level 7 prefix `cb'

Chain RH-Firewall-1-INPUT (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 LOG        all  --  *      *       10.129.66.1          127.0.0.1
LOG flags 0 level 4
    2   120 LOG        all  --  *      *       10.129.66.1          10.1.2.1
LOG flags 0 level 4
    5   300 LOG        all  --  *      *       0.0.0.0/0            10.1.2.1
LOG flags 0 level 4
   42  6431 ACCEPT     tcp  --  *      *       0.0.0.0/0            0.0.0.0/0
tcp dpt:8443
2570K  825M ACCEPT     all  --  lo     *       0.0.0.0/0            0.0.0.0/0
1275K  107M ACCEPT     icmp --  *      *       0.0.0.0/0            0.0.0.0/0
icmp type 255
    0     0 ACCEPT     esp  --  *      *       0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     ah   --  *      *       0.0.0.0/0            0.0.0.0/0
   97 21668 ACCEPT     udp  --  *      *       0.0.0.0/0            224.0.0.251
udp dpt:5353
    0     0 ACCEPT     udp  --  *      *       10.1.1.0/24          0.0.0.0/0
udp dpt:161
    0     0 ACCEPT     udp  --  *      *       0.0.0.0/0            0.0.0.0/0
udp dpt:631
    0     0 ACCEPT     tcp  --  *      *       0.0.0.0/0            0.0.0.0/0
tcp dpt:631
 489K  165M ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0
```

```
state RELATED,ESTABLISHED
   60  3592 ACCEPT     tcp  --  *      *       0.0.0.0/0            0.0.0.0/0
state NEW tcp dpt:22
    0     0 ACCEPT     tcp  --  *      *       10.1.1.14            0.0.0.0/0
state NEW tcp dpt:8086
    0     0 ACCEPT     tcp  --  *      *       10.1.1.15            0.0.0.0/0
state NEW tcp dpt:8086
    0     0 REJECT     tcp  --  *      *       10.1.1.3             0.0.0.0/0
state NEW tcp dpt:9443 reject-with icmp-port-unreachable
  201 12060 ACCEPT     tcp  --  *      *       192.12.16.26         0.0.0.0/0
state NEW tcp dpt:9443
   77  3696 ACCEPT     tcp  --  *      *       192.12.16.47         0.0.0.0/0
state NEW tcp dpt:9443
    0     0 ACCEPT     tcp  --  *      *       192.12.16.48         0.0.0.0/0
state NEW tcp dpt:9443
    0     0 ACCEPT     tcp  --  *      *       192.12.16.74         0.0.0.0/0
state NEW tcp dpt:9443
 2820  135K ACCEPT     tcp  --  *      *       192.12.16.77         0.0.0.0/0
state NEW tcp dpt:9443
    0     0 ACCEPT     tcp  --  *      *       192.12.16.81         0.0.0.0/0
state NEW tcp dpt:9443
   28  1680 ACCEPT     tcp  --  *      *       192.12.16.104        0.0.0.0/0
state NEW tcp dpt:9443
  212 12778 ACCEPT     tcp  --  *      *       10.1.1.0/24          0.0.0.0/0
state NEW tcp dpt:9443
    0     0 REJECT     tcp  --  *      *       10.1.1.1             0.0.0.0/0
state NEW tcp dpt:3690 reject-with icmp-port-unreachable
    0     0 REJECT     tcp  --  *      *       10.1.1.3             0.0.0.0/0
state NEW tcp dpt:3690 reject-with icmp-port-unreachable
   22  1320 ACCEPT     tcp  --  *      *       10.1.1.0/24          0.0.0.0/0
state NEW tcp dpt:3690
  290 17400 ACCEPT     tcp  --  *      *       10.1.1.0/24          0.0.0.0/0
state NEW tcp dpt:8180
    0     0 ACCEPT     tcp  --  *      *       10.1.1.14            0.0.0.0/0
state NEW tcp dpt:8080
 1172 70440 ACCEPT     tcp  --  *      *       10.1.1.0/24          0.0.0.0/0
state NEW tcp dpt:8443
    0     0 ACCEPT     tcp  --  *      *       10.1.1.104           0.0.0.0/0
state NEW tcp dpt:3306
    0     0 ACCEPT     tcp  --  *      *       10.1.1.12            0.0.0.0/0
state NEW tcp dpt:3306
    0     0 ACCEPT     udp  --  *      *       0.0.0.0/0            0.0.0.0/0
state NEW udp dpt:500
28923 1011K REJECT     all  --  *      *       0.0.0.0/0            0.0.0.0/0
reject-with icmp-host-prohibited
```

[1m4.2.1.3. New Tables[0m

You can create a new table for your specific purpose if you wish. To
do this, you call `ipt_register_table()', with a `struct ipt_table',
which has the following fields:

```
[1mlist[0m
   This field is set to any junk, say `{ NULL, NULL }'.


[1mname[0m
   This field is the name of the table function, as referred to by
   userspace.  The name should match the name of the module (i.e.,
   if the name is "nat", the module must be "iptable_nat.o") for
   auto-loading to work.


[1mtable[0m
   This is a fully-populated `struct ipt_replace', as used by
   userspace to replace a table.  The `counters' pointer should be
   set to NULL.  This data structure can be declared `__initdata'
   so it is discarded after boot.


[1mvalid_hooks[0m
   This is a bitmask of the IPv4 netfilter hooks you will enter the
   table with: this is used to check that those entry points are
   valid, and to calculate the possible hooks for ipt_match and
   ipt_target `checkentry()' functions.


[1mlock[0m
   This is the read-write spinlock for the entire table; initialize
   it to RW_LOCK_UNLOCKED.


[1mprivate[0m
   This is used internally by the ip_tables code.
```