Connection tracking

What is connection tracking?

Connection tracking refers to the ability to maintain state information about a connection in memory tables, such as source and destination ip address and port number pairs (known as *socket pairs*), protocol types, connection state and timeouts. Firewalls that do this are known as *stateful*. Stateful firewalling is inherently more secure than its "stateless" counterpart .... simple packet filtering.

Connection tracking is accomplished with the state option in iptables. From the iptables manpage:

state
This module, when combined with connection tracking, allows access to the connection tracking state for this packet.

--state state
Where state is a comma separated list of the connection states to match. Possible states are INVALID meaning that the packet is associated with no known connection, ESTABLISHED meaning that the packet is associated with a connection which has seen packets in both directions, NEW meaning that the packet has started a new con nection, or otherwise associated with a connection which has not seen packets in both directions, and RELATED meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error.

Connection tracking is done either in the PREROUTING chain, or the OUTPUT chain for locally generated packets.
Connection tracking defragments all packets before tracking their state. This explains why there is no ip_always_defrag switch as there was in the 2.2 kernel.

The state table for udp and tcp connections is maintained in */proc/net/ip_conntrack*. We will discuss what its contents look below.

The maximum number of connections the state table can contain is stored in */proc/sys/net/ipv4/ip_conntrack_max*. This value is determined initially by how much physical memory you have (on my 128Mb machine, ip_conntrack_max = 8184 by default).

How does connection tracking work?

A quick overview

Lets get our bearings first with respect to the whole netfilter framework before we delve deeper. For a packet forwarded between interfaces the sequence of chain negotiation would be:

1) PREROUTING chain - DNAT the packet if necessary. Mangle the packet if necessary. Connection tracking now defragments and tracks (classifies) the packet in some way:

If the packet matches a entry in the state table it is part of an ESTABLISHED connection. If it is icmp traffic it might be RELATED to a udp/tcp connection already in the state table. The packet might be starting a NEW connection, or it might be unrelated to any connection in which case it is deemed INVALID.

2) FORWARD chain - Compare the packet state against the ruleset in the filter table until the first match, or until the default policy of the chain is executed.

3) POSTROUTING chain - SNAT the packet if necessary.

Note that all packets are compared against the ruleset in the filter table. This is easily proved - If you have entries in the state table and you change

the rules to deny all traffic, then although the entries in the state table remain, all traffic is indeed denied as it should be.

More detail

We will consider each of the three protocols, udp, tcp and icmp in turn.

UDP

Because it lacks sequence numbers, udp is known as a "stateless" protocol . However, this does not mean we can't track udp connections. There is still other useful information we can utilize. Here is an example state table entry for a newly formed udp connection:

udp 17 19 src=192.168.1.2 dst=192.168.1.50 sport=1032 dport=53 [UNREPLIED] src=192.168.1.50 dst=192.168.1.2 sport=53 dport=1032 use=1

This state table entry can only be made if there is an iptables filter rule specifying NEW connections, something like the following ruleset, which allows NEW connections outbound only (as is often wise):

iptables -A INPUT -p udp -m state --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p udp -m state --state NEW,ESTABLISHED -j ACCEPT

Things we can tell from the state table entry are as follows:

- The protocol is udp (IP protocol number 17).
- The state table entry has 19 seconds until it expires.
- Source and destination addresses and ports of original query.
- Source and destination addresses and ports of expected reply. The connection is marked UNREPLIED so this has not been received yet.

Udp timeouts are set in */usr/src/linux/net/ipv4/netfilter/ip_conntrack_proto_udp.c* at compile time.
Here is the relevant section of code:

#define UDP_TIMEOUT (30*HZ)
#define UDP_STREAM_TIMEOUT (180*HZ)

A single request will enter into the state for 30*HZ (generally 30 seconds). In the example above, where we have 19 seconds left, 11 seconds have already elapsed without a reply being received. Once a reply is received, and allowed by a rule permitting ESTABLISHED connections, the timeout is reset to 30 seconds and the UNREPLIED mark is removed. Here we see the connection a couple of seconds after this has taken place:

udp 17 28 src=192.168.1.2 dst=192.168.1.50 sport=1032 dport=53 src=192.168.1.50 dst=192.168.1.2 sport=53 dport=1032 use=1

If multiple requests and replies occur between the same socket pairs, the entry is considered to be a stream and the timeout changes to 180 seconds. At this point the entry is marked ASSURED (once connections become ASSURED they are not dropped under heavy load). Here we see the connection a few of seconds after this has taken place:

udp 17 177 src=192.168.1.2 dst=192.168.1.50 sport=1032 dport=53 src=192.168.1.50 dst=192.168.1.2 sport=53 dport=1032 [ASSURED] use=1

There is no absolute timeout for a udp connection (or a tcp connection for that matter), provided traffic keeps flowing.

TCP

A tcp connection is initiated via a three-way handshake involving a synchronization request from the client, a synchronization and an acknowledgement from the server, and finally an acknowledgement from the client. Subsequent traffic flowing between server and client is acknowledged in all cases. The sequence looks like

Client		Server
SYN	--->	
	<---	SYN+ACK
ACK	--->	
	<---	ACK
ACK	--->	
	.....	
	.....	

SYN and ACK refer to flags set in the tcp header. There are also 32 bit sequence and acknowledgement numbers stored in the tcp header which are passed back and forth and updated during the session.

To get connection tracking to work for a tcp connection you need a ruleset like this:

```
iptables -A INPUT     -p tcp -m state --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp -m state --state NEW,ESTABLISHED -j ACCEPT
```

Walkthrough

What we are going to do now is walk and talk through the establishment of a normal tcp connection and look at the state table at each stage:

1) Once an initial SYN is sent in the OUTPUT chain, and accepted by out rule that allows the NEW connection, the connection table entry may look something like:

```
tcp      6 119 SYN_SENT src=140.208.5.62 dst=207.46.230.218 sport=1311 dport=80 [UNREPLIED] src=207.46.230.218 dst=140.208.5.62 sport=80 dport=1311 use=1
```

The tcp connection status is SYN_SENT and the connection is marked UNREPLIED.

2) We are now waiting for a SYN+ACK to arrive at which point the tcp connection state changes to SYN_RECV and the UNREPLIED disappears:

```
tcp      6 57 SYN_RECV src=140.208.5.62 dst=207.46.230.218 sport=1311 dport=80 src=207.46.230.218 dst=140.208.5.62 sport=80 dport=1311 use=1
```

3) We are now waiting for the final part of the handshake, an ACK. When it arrives, we check that it's sequence number matches the ACK of the handshake from the server to the client. The tcp connection state now becomes ESTABLISHED and the state table entry is marked ASSURED (ASSURED connections are not dropped from the state table when the connection is under load). Here we see the ESTABLISHED connection:

```
tcp      6 431995 ESTABLISHED src=140.208.5.62 dst=207.46.230.218 sport=1311 dport=80 src=207.46.230.218 dst=140.208.5.62 sport=80 dport=1311 [ASSURED] use=1
```

Connection tracking's perspective on the state table

We just talked a lot about tcp connection states. Now let's think about this from the perspective of the connection tracking:

Connection tracking only knows about NEW, ESTABLISHED, RELATED and INVALID, classified as described above and in the iptables manpage. To quote Joszef Kadlecsik, who helped me out with a confusion I had initially about this very subject:

When a packet with the SYN+ACK flags set arrrives in response to a packet with SYN set the connection tracking thinks: "I have been just seeing a packet with SYN+ACK which answers a SYN I had previously seen, so this is an ESTABLISHED connection."

The important point here is that the conntrack states are *not* equivalent to tcp states. We have already seen that a connection doesn't achive the tcp connection status of ESTABLISHED until the ACK after the SYN+ACK has been received.

The representation of the tcp connection states in the state table is purely for timeouts. You can prove this to yourself by sending an ACK packet through your firewall to a non-existent machine (so that you don't get the RST back). It will create a state table entry no problem because it it is the first packet of a connection and so is treated as NEW (the entry will not be marked as ASSURED though). Checkpoint's Firewall-1 version 4.1 SP1 allows connection initiation by ACK packets too (see Lance Spitzner's whitepaper for details).

In the light of the fact that ACK packets can create state table entries, the following contribution from Henrik Nordstrum is insightful: To make sure that NEW tcp connections are packets with SYN set, use the following rule:

```
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

Note that doing this will prevent idle sessions from continuing once they have expired from the conntrack table. In the normal "relaxed" view such connections initiated from the correct direction (i.e. the direction you allow NEW packets through) can normally continue even if expired from conntrack, provided that the first data/ack packet that resumes the connection comes from the correct direction.

If you want real stateful filtering that requires correct connection initiation and tracks sequence numbers, apply the tcp-window-tracking patch from patch-o-matic. A very detailed paper describing the patch can be found here.

Timeouts

Something to note is that timeouts are reset to the maximum each time a connection sees traffic. Timeouts are set in */usr/src/linux/net/ipv4/netfilter /ip_conntrack_proto_tcp.c* at compile time. Here is the relevant section of code:

```
static unsigned long tcp_timeouts[]
= { 30 MINS,   /*      TCP_CONNTRACK_NONE,     */
    5 DAYS,    /*      TCP_CONNTRACK_ESTABLISHED,      */
    2 MINS,    /*      TCP_CONNTRACK_SYN_SENT, */
    60 SECS,   /*      TCP_CONNTRACK_SYN_RECV, */
    2 MINS,    /*      TCP_CONNTRACK_FIN_WAIT, */
    2 MINS,    /*      TCP_CONNTRACK_TIME_WAIT,        */
```

```
    10 SECS,    /*     TCP_CONNTRACK_CLOSE,    */
    60 SECS,    /*     TCP_CONNTRACK_CLOSE_WAIT,       */
    30 SECS,    /*     TCP_CONNTRACK_LAST_ACK, */
    2 MINS,     /*     TCP_CONNTRACK_LISTEN,   */
};
```

There is no absolute timeout for a connection.

Connection termination

Connection termination occurs in two ways. Natural termination at the end of a session occurs when the client sends a packet with the FIN and ACK flags set. The closure proceeds as follows:

```
    Client              Server
                .........
                .........
  FIN+ACK   --->
                <---      ACK
                <---    FIN+ACK
      ACK       --->
```

Sometime during, or at the end of this sequence the state table connection status changes to TIME_WAIT and the entry is removed after 2 minutes by default.

Another way for connection termination to occur is if either party sends a packet with the RST (reset) flag set. RST's are not acknowledged. In this case the state table connection status changes to CLOSE and times out from the state table after 10 seconds. This often happens with http entries, where the server sends an RST after a period of inactivity.

ICMP

In iptables parlance, there are only four types of icmp that can be categorized as NEW, or ESTABLISHED:

1) Echo request (ping, 8) and echo reply (pong, 0).
2) Timestamp request (13)and reply (14).
3) Information request (15) and reply (16).
4) Address mask request (17) and reply (18).

The request in each case is classified as NEW and the reply as ESTABLISHED.
Other types of icmp are not request-reply based and can only be RELATED to other connections.

Let us consider a sample ruleset and a few examples:

```
iptables -A OUTPUT -p icmp -m state --state NEW,ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT     -p icmp -m state --state ESTABLISHED,RELATED  -j ACCEPT
```

1) An icmp echo request is NEW and so is allowed in the OUTPUT chain.

2) An icmp echo reply, provided it is in response to an echo request, is ESTABLISHED and so is allowed in the INPUT chain. An echo reply cannot be allowed in the OUTPUT chain for the rules above because there is no NEW in the INPUT chain to allow echo requests and a reply has to be in response to a request.

3) An icmp redirect, because it is not request-reply based, is RELATED and so can be allowed in both the INPUT and the OUTPUT chains provided there is already a tcp or udp connection in the state table already that it can be matched against.

Connection tracking and ftp

Firstly, you need to load the ip_conntrack_ftp module.

Assuming you have a single-homed box, a simple ruleset to allow an ftp connection would be:

```
iptables -A INPUT     -p tcp --sport 21 -m state --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp --dport 21 -m state --state NEW,ESTABLISHED -j ACCEPT
```

(Please note, I am assuming here you have a separate ruleset to allow any icmp RELATED to the conection. Please see my example ruleset for this).

This is not the whole story. An ftp connection also needs a data-channel, which can be provided in one of two ways:

1) Active ftp

The ftp client sends a port number over the ftp channel via a PORT command to the ftp server. The ftp server then connects from port 20 to this port to send data, such as a file, or the output from an ls command. The ftp-data connection is in the *opposite sense* from the original ftp connection.

To allow active ftp without knowing the port number that has been passed we need a general rule which allows connections from port 20 on remote ftp servers to high ports (port numbers > 1023) on ftp clients. This is simply too general to ever be secure.

Enter the *ip_conntrack_ftp* module. This module is able to recognize the PORT command and pick-out the port number. As such, the ftp-data connection can be classified as RELATED to the original outgoing connection to port 21 so we don't need NEW as a state match for the connection in the INPUT chain. The following rules will serve our purposes grandly:

```
iptables -A INPUT     -p tcp --sport 20 -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -p tcp --dport 20 -m state --state ESTABLISHED -j ACCEPT
```

2) Passive ftp

A PORT command is again issued, but this time it is from the server to the client. The client connects to the server for data transfer. Since the connection is in the *same sense* as the original ftp connection, passive ftp is inherently more secure than active ftp, but note that this time we know even less about the port numbers. Now we have a connection between almost arbitrary port numbers.

Enter the *ip_conntrack_ftp* module once more. Again, this module is able to recognize the PORT command and pick-out the port number. Instead of NEW in the state match for the OUTPUT chain, we can use RELATED. The following rules will suffice:

```
iptables -A INPUT     -p tcp --sport 1024: --dport 1024:  -m state --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp --sport 1024: --dport 1024:  -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Prepared by James C. Stephens
(jns@gfdl.noaa.gov)
Last updated:Thu Apr 5 EST 2001