

Real Stateful TCP Packet Filtering in IP Filter

Guido van Rooij <guido@gvr.org>,
Origin IT, P.O.Box 218,
5600 MD Eindhoven, The Netherlands

Abstract

IP Filter is an Open Source packet filtering engine that is available for a number of operating systems, including Solaris and FreeBSD, OpenBSD and NetBSD.

IP Filter comes with so-called stateful packet filtering. In the case of TCP, the state engine not only inspects the presence of ACK flags, or looks at source and destination ports, but it includes sequence numbers and window sizes in its decision to pass or block packets. This greatly reduces the window of opportunity for malicious packets to be passed through the packet filter, even in the case when source and destination ports and addresses are known.

The original way of performing this in IP Filter produced a number of problems. The main problem being that IP Filter assumed, when it detected packets traversing through it, that the destination would also see the packets. In the case of packet loss this is not always true. Furthermore, it previously looked at window sizes symmetrically: the window size was treated as a relative upper bound for new data that was sent, and the negative window size was treated as a lower bound (for retransmissions). There is no apparent reason for doing this, and it will be proven to be invalid. Additionally, the window size taken as the upper bound for new data was the last window advertisement seen. In case of retransmissions this meant a fall back in allowed data which was also incorrect.

The new state engine does not have these problems. The main design criteria was to never assume anything, but to only take information for granted when it can be proven to be correct. When implementing this design in IP Filter, it turns out that some criteria must be relaxed in order to be compatible with other code within IP Filter. These consequences will be discussed. Furthermore, some hints will be given on how

to improve the implementation of the state timeouts.

The paper will conclude with experiences with the state code and list future work on the state code.

1. Introduction

In recent years, more and more networks with sensitive or even business critical data on them are being interconnected. Simultaneously, hacker activity has grown tremendously because of freely available hacker tools. In order to protect networks, so-called firewalls are deployed that protect against hacker activities. One of the ways to implement a firewall is to make use of so-called packet filters.

TCP/IP

In order to understand the following of this article a small introduction to TCP/IP is necessary. TCP is a protocol that runs on top of IP. IP takes care of delivering packets. For the remainder of this article, it is relevant that each IP packet contains two addresses: the address of the source of the packet and the address of the destination of the packet. TCP adds a reliable, connection oriented service to IP. It makes sure that there is a reliable data stream between source and destination. TCP avoids duplication of transmitted data and avoids delivering data out of order. In order to be able to setup multiple connections between two hosts, TCP adds so-called ports to IP addresses. These ports identify the connection endpoints on the source and destination. The combination of source port, source address, destination port and destination address are unique for every TCP connection.

TCP uses a three-way handshake to setup a connection. The source sends a packet with a special flag set called the SYN flag and with the so-called sequence number field (or for short

seq field) set to some initial value. These sequence numbers are used to uniquely identify each octet (the network lingo for 8 bits) of data in the connection. Counting starts at the initial sequence number. If the destination is willing to accept the connection, it sends a packet back with the SYN flag set as well. Furthermore another flag, called the ACK flag, is set. When the ACK flag is set, the value in the acknowledgement (or ack) field is equal to the number of the next unreceived octet. In this case the ACK flag acknowledges the first packet which only contains the SYN flag but has no payload octets. To be able to acknowledge this packet anyway, the SYN flag is also counted as one octet of data. The third packet then acknowledges the second packet by also having its ACK flag set and with the ack field set to the appropriate value.

There is an additional field in the TCP packet that is relevant in this article: the window size. The window size determines how much data a host is willing to accept. It thus serves as a flow control mechanism (no more than the window size of data will be in transit over the network). When a host wants to end the connection, another three-way handshake takes place. The difference with the initial one is that in this case not the SYN flag is used but the FIN flag. Similar to the SYN flag, the FIN flag also counts as one octet of data.

It should be noted that this is a very short introduction to TCP. The flags and fields relevant to the understanding of this article are covered but nothing more. In reality, the protocol has many more features and extensions. In order to get a more thorough understanding, the reader is referred to [RFC793], [RS1] and [RS2].

Packet Filtering

Packet filtering has proved to be a handy tool to put access controls to IP traffic. Packet filters can be used to block IP packets based on certain criteria such as the protocol used and various protocol characteristics. In early packet filters, filtering decisions were made based solely on the packet that is currently inspected. Data like the source and destination addresses and in UDP and TCP cases the source and destination ports could be used in the filtering decisions. Even the well known 'established' keyword, was based on static information (it inspected the presence of the ACK and RST flags in TCP return traffic [IOS12, access-list (extended)]). Such filtering could be very well used to protect

against spoofing attacks where the attacker would send packets that seem to originate from systems on the inside of the packet filter.

In recent years, the underworld has produced more and more tools exploiting the static nature of this generation of packet filters. A simple form of probing is to send packets with the ACK flag set (further referred to as an ACK) to a target host. This way one can determine if the target host is listening on a port without easy detection [UM, NMAP]. Denial of service tools exist that are able to bypass static filters [BR].

In order to be able to withstand these newer probes and attacks, filters must somehow keep state information on what is flowing through the filter. The filter can then check if an ACK actually belongs to a valid connection. Packet filters that keep such state information are called dynamic or stateful packet filters. Apart from being able to block more unwanted traffic than static filters, an important advantage of stateful packet filters is that one does not need to explicitly permit return traffic thus simplifying access list administration.

In some packet filtering engines, stateful packet filtering for TCP and UDP traffic is implemented in a very simple way: namely by storing the source and destination addresses and ports in a state table. While this is a lot safer than the older 'established' method where ACK's are always passed, this still is not as secure as can be. Ideally we only want those packets to pass the filtering engine that are absolutely necessary for the correct functioning of the TCP session. Such a way of filtering will at least reveal maliciously inserted packets and might protect against yet unknown vulnerabilities.

2. IP Filter's stateful packet filtering

IP Filter is a TCP/IP packet filtering engine that runs on a several Unix platforms. It has been used by the author for several years in a variety of systems. It has a very rich set of features from basic filtering to Network Address Translation in various forms and includes support for transparent proxying. For a complete description of all features see [IPF].

Ideally, the packet filter has a policy of blocking everything that is not specifically allowed. For TCP sessions, there is a rich set of information available on which to base the decision to block or pass a packet. We already described methods

that look at ACK flags and source and destination ports. In IP Filter, also the ack and seq fields are taken into account, further closing down the window of opportunity for abusive packets. In the remainder of the article the TCP protocol is assumed for all packets and all filtering.

Old stateful filtering description

Whenever a packet is allowed by the filter code to pass through the filtering host, the filter code allows for the creation of a state entry. New packets arriving at the filtering host are first matched against the state entry table. In case of a match, the state entry is updated if necessary and the packet is allowed through. First, the source address, source port, destination address and destination ports are matched. If a match is found, a special piece of code is executed that inspects if the ack and seq fields are valid for the given state entry. This code is the core of the TCP state engine. In Figure 1, a simplified

version of this particular code can be found; exception handling, e.g. packets not having the ACK flag set, and some initializing code, not relevant to the discussion below, are left out.

The code roughly works as follows: the pointer `is` points to the state entry that matches the source and destination addresses and ports. It contains the state of the connection as seen from the source of the packet that led to the creation of the `is` state entry. Among others it contains the last seen seq field from the source `is_seq` and the last seen ack field from the source `is_ack`. Both `is_ack` and `is_seq` are overloaded, however: if a packet is matched that travels from the destination to the source, `is_seq` is filled with the ack field of the packet and `is_ack` with the seq field of the packet. The state entry also contains the last seen window values, both from the source `is_swin` and the destination `is_dwin`.

```

/*
 * Find difference between last checked packet and this packet.
 */
seq = ntohl(tcp->th_seq);
ack = ntohl(tcp->th_ack);
source = (ip->ip_src.s_addr == is->is_src.s_addr);

if (source) {
    seqskew = seq - is->is_seq;
    ackskew = (ack - 1) - is->is_ack;
} else {
    ackskew = seq - is->is_ack;
    seqskew = (ack - 1) - is->is_seq;
}

/*
 * Make skew values absolute
 */
if (seqskew < 0)
    seqskew = -seqskew;
if (ackskew < 0)
    ackskew = -ackskew;

/*
 * If the difference in sequence and ack numbers is within the
 * window size of the connection, store these values and match
 * the packet.
 */
win = ntohs(tcp->th_win);
if ((seqskew <= is->is_dwin) && (ackskew <= is->is_swin)) {
    /* packet matches the state entry */
    if (source) {
        is->is_seq = seq;
        is->is_ack = ack;
        if (win != 0)
            is->is_swin = win;
    } else {
        is->is_seq = ack;
        is->is_ack = seq;
        if (win != 0)

```

```

        is->is_dwin = win;
    }
    do statistics;
    set timeout values;
    permit packet to pass;
}
/* packet does not match the state entry */
deny packet to pass;

```

Figure 1: Old state code implementation

When a packet comes in, the variables `seq` and `ack` are set to the respective fields in the TCP packet. Then two values are calculated: `seqskew` and `ackskew`. These represent the absolute value of the difference of the value of the `seq` respectively the `ack` fields in the packet with `is_seq` respectively `is_ack`. When `seqskew` is smaller than or equal to the last advertised destination window and `ackskew` is smaller than or equal to the last advertised source window the packet is matched and the state entry is updated accordingly.

3. Analysis of the old model

This section will give 2 examples of situations where the old state engine made some wrong decisions. In the examples, some packet traces will be shown. Packets will be shown in the following format, in the order they are sent:

```
A→B | 0:1000 win 2000 ack 1000 | N
```

This line identifies a packet that is sent from A to B. The packet contains data starting at sequence number 0 up to but not including 1000. The length of the data section is thus 1000. Furthermore, the packet contains a window advertisement of 2000 octets and it acknowledges all data sent by B up to but not including sequence number 1000. The number N at the end of the line means that this is the N-th packet that the filtering hosts sees. In case the ACK or WIN values are not relevant for the example, they are omitted. Finally, it is assumed that none of the packets are fragmented.

Example 1

Suppose a connection is setup normally, and has entered the state table. Suppose further that the first packet below matches the state table and is passed through:

From	Content	Nr
B→A	win 2048 ack 0	1
A→B	0:1000	2
B→A	win 1048 ack 1000	7
A→B	1000:2000	3
B→A	win 2048 ack 2000	4
A→B	2000:3000	5
B→A	win 2048 ack 3000	6

Looking at this trace, one sees a normal TCP session where the first ACK sent by B is somehow delayed and arrives at the filtering host after the fourth ACK.

The state table now looks as follows during the above session (the state entry values shown are those before the packet has been matched by the state code):

nr	state entry		packet content			code seqskew
	is_seq	is_dwin	seq	ack	win	
1	<i>not relevant</i>			0	2048	-
2	0	2048	0			0
3	0	2048	1000			1000
4	1000	2048		2000	2048	999
5	2000	2048	2000			0
6	2000	2048		3000	2048	999
7	3000	2048		1000	1048	2001

Now either host A can send some data to B or host B can send a retransmit of packet 6:

```
A→B | 3000:4000 | 8a
B→A | win 2048 ack 3000 | 8b
```

These packets lead to the following state entries:

8a	1000	1048	3000			2000
8b	1000	1048		3000	2048	1999

Because `seqskew` is greater than `is_dwin`, both packet 8a and 8b will be blocked. The

connection can not proceed anymore: whenever A will send a packet, it believes it can send the data 3000:4000 and this will always be blocked. Packets from B will always ack 3000 and thus will also be blocked. This results in blocking packets that are part of a valid connection. In fact, the connection just hangs.

Example 2

Suppose that in an established TCP connection the following packets are sent:

From	Content	Nr
A→B	0:1000	1
B→A	win 4000 ack 1000	2
A→B	1000:2000	3
A→B	2000:3000	4
A→B	3000:4000	5
A→B	4000:5000	6
B→A	win 2000 ack 5000	8
B→A	win 4000 ack 5000	7

Suppose that all packets actually reach their destination in the order sent, except for the second and third ACK. The second ACK is delayed somehow between host B and the filtering host. Furthermore, both ACKs are dropped somewhere between the filtering host and A. Suppose also that the first 2 packets are such that they are passed through the filter.

The relevant fields in the state table look as follows during the above session (again, the state entry values shown are those before the packet has been matched by the state code):

nr	state entry		packet content			code
	is_seq	is_dwin	seq	ack	win	seqskew
1	<i>not relevant</i>		0			-
2	0			1000	4000	999
3	1000	4000	1000			0
4	1000	4000	2000			1000
5	2000	4000	3000			1000
6	3000	4000	4000			1000
7	4000	4000		5000	4000	999
8	5000	4000		5000	2000	1

Since `seqskew` is smaller than `is_dwin` in all cases, these packets will pass the filter.

Host B will only retransmit the ACK when:

1. It must send a window update

2. When it is piggy-backed on some data.
3. When it receives a retransmit.
4. When the TCP keep-alive timer goes of.

For the sake of the example we can assume that cases 1 and 2 are not applicable. Host A however will find that the data packet in 3 is not yet acknowledged. It will therefor retransmit it. The retransmit occurs because the acknowledgement did not arrive in time and thus host A will only try to retransmit the first unacknowledged packet. It is assumed here that host A complies to section 3.1 of the proposed standard [RFC2581].

This leads to the following packet:

A→B | 1000:2000 | 9

and corresponding state table values:

9 | 5000 2000 | 1000 | 4000

And thus, because `seqskew=4000` is greater than `is_dwin=2000`, the retransmit gets blocked unnecessarily. After a while, A will give up retransmitting the packet and drop the connection on his side.

Both examples show that the state engine is not coping well with out of order packets and packet loss. In order to fully understand the new state filtering engine that will be described in the next section, it is crucial to see why the old state engine was invalid: when the filter engine sees a packet, it adjusts its state administration accordingly. However, one can not be sure that the packet actually reaches its destination. Really, the only thing one can be sure of is that the sender did send the packet (given the assumption that a valid packet is considered to be originated from the sender and not spoofed)

4. Design of the new model

The following points made up the design strategy of the new code:

1. Never assume anything: the state administration should only be based on facts.
2. The state filtering should take all kinds of TCP behavior into account. This includes retransmissions and window probing. In fact every forwarded packet passes twice

through the filter code. The first pass is on the input queue and the second pass on the output queue. So every forwarded, outgoing packet looks to IP Filter as a retransmission!

3. Never block packets such that a TCP session can hang.
4. Make the window of opportunity for abuse as small as possible. Abuse is defined here as sending malicious data that will be accepted as valid data or sending malicious ACK's that will be accepted as valid ACK's.
5. Minimize the amount of blocked packets that belong to valid sessions because they will cause false alarms.

In general, when setting bounds on what constitutes valid packets, it will be possible that valid packets will be blocked. As an example: suppose that an ACK only packet is somehow delayed in transmission and pops up when the real connection has moved way forward. When this delayed ACK gets blocked, it will cause a false alarm but the blocking will not have any effect on the TCP session it belongs to.

Whenever the filter sees a packet that it considers valid, it must assume that the sender sent the packet. If a valid packet arrives at the filtering system:

1. The filter clearly sees that some data is transmitted.
2. The filter sees the window advertisement done by the sender.
3. The filter can conclude that if the ACK flag is set, and the value of it is S, the sender has received all data up to at least sequence number S.

The real challenge lies in the decision what constitutes a valid packet. In order to determine the validity, lower and upper bounds will be derived both for the ack values in a packet and for the data the packet contains. Note that it is assumed that all packets within a certain TCP session are routed through the system where the packet filter is installed.

Boundaries for valid data

Suppose host A sends a packet to host B containing the data interval $[s, s+n)$ (meaning that it contains data starting with sequence number s and having length n). Between A and B there is a packet filtering system F that routes all packets sent between A and B.

The upper bound determines when data is allowed to be sent:

last octet in packet \leq maximum octet A is allowed to send

This is equivalent to:

$s + n \leq$ maximum octet A may send + 1

$$\leq \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \text{ack} + \text{win} \}$$

the right hand side meaning the maximum value of the sum of the ack and win fields from packets that are sent by host B to host A that are actually received by A.

Thus,

$$s + n \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by F}}} \{ \text{ack} + \text{win} \} \quad (\text{Ia})$$

because a packet received by host A must have been seen by F.

There is an exception to (Ia): in TCP, when host B advertises a zero window, host A will start the so-called persist timer that will cause it to reprobe the B's window persistently until it is non-zero. In doing so, A will send at least the first unacknowledged octet of data to B. This is the only accepted situation where data may be sent out of advertised window boundaries [RFC793, section 1.5].

On BSD systems, a window probing is always done with a packet containing one octet of data [GW, page 827]. This length was assumed for other systems as well and up to date, no problems seem to arise. So the real upper bound is:

$$s + n \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by F}}} \{ \text{ack} + \max(\text{win}, 1) \} \quad (\text{I})$$

This upper bound will prevent passing of data the recipient did not intend to receive. A packet that is blocked because of this boundary was sent by the sender at a time that it knew that the receiver would ignore it. In case the assumption about window probes containing one octet of data is wrong, the boundary can easily be adapted by replacing the '1' by the appropriate number of octets, or by contacting the vendor of the code to ask him not to waste bandwidth and use 1 octet window probes.

It is harder to find a suitable lower bound: When host A sends some data, it will only send unacknowledged data.

For short:

$$s \geq \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \text{ack} \} \quad (\text{i})$$

Looking at the derivation of the upper bound, it follows that equation (I) is also valid as seen from the senders point of view. Thus:

$$s+n \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \text{ack} + \max(\text{win}, 1) \}$$

$$\Rightarrow \max_{\text{packets sent by A}} \{ s+n \} \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \text{ack} \} + \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \max(\text{win}, 1) \} \quad (\text{ii})$$

and thus, combining (i) and (ii):

$$s \geq \max_{\text{packets sent by A}} \{ s+n \} - \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ \max(\text{win}, 1) \}$$

$$\geq \max_{\substack{\text{packets sent by A} \\ \text{seen by F}}} \{ s+n \} - \max_{\substack{\text{packets sent by B} \\ \text{seen by F}}} \{ \max(\text{win}, 1) \} \quad (\text{II})$$

The lower bound will prevent retransmission from data that is known to be already received. So if the receiver actually did send a packet containing such data it was somehow delayed during transit. Since the communication already moved on (otherwise we would not even be able to tell that the packet was an unnecessary retransmit) we can also be sure that the ack value on the blocked packet will at least have been duplicated (if not moved forward) in later packets from sender to receiver.

Boundaries for valid Acknowledgements

Packets cannot contain an ack value for data that was not sent. This implicates that we have a clear upper bound for the ack value sent: An ACK from host A, with ack value 'a,' can never acknowledge data that was not received by A.

Thus:

$$a \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by A}}} \{ s+n \} \leq \max_{\substack{\text{packets sent by B} \\ \text{seen by F}}} \{ s+n \} \quad (\text{III})$$

This upper bound will prevent sending of ACK's of data that could not have been received. A packet that is blocked because of this bound is known to be invalid.

A lower bound for the ack value is much harder. One might say that the last received ack value is a lower bound as ack values tend to move forward. If however, two packets both containing valid data are received out of order by the filter, then the last one received will be blocked. First of all, this is a false alarm and secondly, the sender will have to do a retransmission if the blocked packet contained valid data.

We might relax the above rule by saying:

$$a \geq \max_{\substack{\text{packet sent by A} \\ \text{seen by F}}} \{ \text{ack} \} \text{ or}$$

packet contains valid data (according to (i) and (ii))

But also in this case, packets might get blocked unnecessarily when dataless ACK's are received out of order. Furthermore, the presence of valid data suppresses checking the ack value at all, which is not necessary.

Instead, a different approach was chosen. Currently, the following fixed boundary is chosen:

$$a \geq \max_{\substack{\text{packets sent by B} \\ \text{seen by F}}} \{ s+n \} - \text{MAXACKWINDOW} \quad (\text{IV})$$

In natural language: An ACK is allowed if it acknowledges data from host B that is not less than MAXACKWINDOW octets from the last octet of data seen by the filter. This last octet of data seen by the filter is of course larger than the last octet of data seen by host B, the sender of the ACK. However, MAXACKWINDOW is slightly larger than the maximum possible value of the TCP window field (66000) and thus it can be guaranteed that no valid ACK will ever get blocked. This boundary seems like a cheap deal after all the trouble that went into the sequence number boundaries. The observation is that an ACK of data that is already received will be ignored by the receiver of the ACK. Thus the window in which ACK's of received data are allowed can be made very large. The larger the window, the smaller the chance of a delayed ACK being blocked.

5. Implementation

Data structures

In order to implement checking of the derived boundaries, the following data structures are used:

```
struct tcpstate {
    u_short    ts_sport;
    u_short    ts_dport;
    tcpdata_t  ts_data[2];
    u_char     ts_state[2];
} tcpstate_t;
```

```
struct tcpdata {
    u_32_t     td_end;
    u_32_t     td_maxend;
    u_short    td_maxwin;
} tcpdata_t;
```

The meaning of the various fields is as follows:

In struct tcpstate:

```
ts_sport    source port
ts_dport    destination port
ts_data[0]  source struct tcpdata
ts_data[1]  destination struct tcpdata
ts_state[0] source state (used for state timing)
ts_state[1] destination state (used for state timing)
```

Source and destination are defined by the packet that leads to the new state entry.

Struct tcpdata contains:

```
td_end      maximum value of seq + len
             (boundaries II, III and IV)
td_maxend   maximum value of ack +
             max(win, 1) (boundary I)
td_maxwin   the maximum window seen
             (boundary II)
```

Initializations

The above boundaries are valid in the middle of connections, but special treatment should be given for initializations when a packet leads to creation of a new state entry.

First the normal case is examined: the sender sends a SYN packet to initiate a connection. The question is how should the state entry be initialized such that following packets are able to pass. The possibilities for the next packet in this session are retransmission of the SYN and the receiver sending a SYN/ACK.

Both packets will be matched against boundaries I-IV.

1. Retransmission of the SYN

When the initialization of the state entry is done as follows:

```

ts_data[0].td_end = SEQ + 1
ts_data[0].td_maxend = SEQ + 1
ts_data[1].td_end = 0
ts_data[1].td_maxend = 0
ts_data[1].td_maxwin = 1
ts_data[0].td_maxwin = max(WIN, 1)

```

Here SEQ and WIN are the values of the seq and win fields in the SYN packet. Note that `ts_data[1].td_end` and `ts_data[1].td_maxend` are not backed by actual data in the packet and have to be reinitialized once 'real' data is available.

Then clearly, for the retransmitted packet:

```

s+n = SEQ + 1
s = SEQ
s+n ≤ ts_data[0].td_maxend      (I)
s ≥ ts_data[0].td_end -
    ts_data[1].td_maxwin      (II)

```

where s and n are defined as in section 4.

Since the ACK flag is not set in the retransmission, just assuming that the ACK flag was set and setting the ack value to 0 will result in:

```

a = 0
a ≤ ts_data[1].td_end      (III)
a ≥ ts_data[1].td_end -
    MAXACKWINDOW      (IV)

```

Handling the absence of ACK flags in this way allows for an easier implementation since this exception is effectively eliminated.

2. Receiver sends a SYN/ACK

In this case, the falsely initialized fields can be set in the state entry:

```

ts_data[1].td_end = SEQ + 1
ts_data[1].td_maxend = SEQ + 1

```

Here SEQ is the value of the seq field in the SYN/ACK packet.

In this case,

```

s+n = SEQ + 1
s = SEQ
a = ACK

```

Clearly:

$$s+n \leq ts_data[1].td_maxend \quad (I)$$

$$s \geq ts_data[1].td_end - ts_data[0].td_maxwin \quad (II)$$

the latter because `ts_data[0].td_maxwin` ≥ 1

and:

$$a \leq ts_data[0].td_maxend \quad (III)$$

$$a \geq ts_data[0].td_end - MAXACKWINDOW \quad (IV)$$

because the ack field of this packet acknowledges SEQ+1 from the initial SYN packet.

The above analysis is correct for connections that enter the state table when being setup. However, IP Filter leaves the possibility for packets from the middle of a connection to lead to an entry in the state table. This can be handy when the system on which IP filter runs is rebooted and existing sessions need to be preserved. The above analysis is no longer correct in such a case. The reason is that the above scenario needs the history of the connection to be able to do its job. There are a number of ways to work around this lack of history. The first is to initialize the state variables such that the boundaries thus set will always include the boundaries that would have been set in case the history was known.

Another way of dealing with this is that the state code just pretends that it knows the history. When a packet comes in that would be blocked given the current boundaries but that would not have been blocked with the 'maximal' boundaries in the previous paragraph, then the current boundaries are stretched such that it would have just passed the packet. This scenario will make the state variables gently converge towards the values that they would have had in case the whole history was known. The advantage of this method is that extending the window is an explicit action that can be logged.

Neither of these methods have been implemented yet.

Actual Implementation

In this section, the actual implementation is given in heavily annotated form. First the initialization part (which is done when a packet leads to a state entry:

```

is->is_tcp.ts_data[0].td_end = ntohl(tcp->th_seq) + ip->ip_len -
                               fin->fin_hlen - (tcp->th_off << 2) +
                               ((tcp->th_flags & TH_SYN) ? 1 : 0) +
                               ((tcp->th_flags & TH_FIN) ? 1 : 0);

```

The right hand side is an ugly way of specifying the TCP payload length of the packet. This is somewhat different from the initialization mentioned earlier. It is a generalization thereof that will allow the state engine to work in the T/TCP case [RFC1644, RS2].

```

is->is_tcp.ts_data[0].td_maxend = is->is_tcp.ts_data[0].td_end;
is->is_tcp.ts_data[0].td_end + 1;
is->is_tcp.ts_data[1].td_end = 0;
is->is_tcp.ts_data[1].td_maxend = 0;
is->is_tcp.ts_data[1].td_maxwin = 1;
is->is_tcp.ts_data[0].td_maxwin = ntohs(tcp->th_win);
if (is->is_tcp.ts_data[0].td_maxwin == 0)
    is->is_tcp.ts_data[0].td_maxwin = 1;

```

These are directly taken from the earlier paragraph on initializations.

Figure 2: New state code initialization

The state matching code looks as follows:

```

source = (ip->ip_src.s_addr == is->is_src.s_addr);
fdata = &is->is_tcp.ts_data[!source];
tdata = &is->is_tcp.ts_data[source];

```

By setting fdata and tdata the code below can be the same, regardless of the direction of the packet. fdata represents the state variables for the sender of the packet that is investigated and tdata represents its receiver.

```

seq = ntohl(tcp->th_seq);
ack = ntohl(tcp->th_ack);
win = ntohs(tcp->th_win);
end = seq + ip->ip_len - fin->fin_hlen - (tcp->th_off << 2) +
      ((tcp->th_flags & TH_SYN) ? 1 : 0) +
      ((tcp->th_flags & TH_FIN) ? 1 : 0);

```

Again the length of the payload of the TCP packet is determined and added to seq.

```

if (fdata->td_end == 0) {
    /*
     * Must be a (outgoing) SYN-ACK in reply to a SYN.
     */
    fdata->td_end = end;
    fdata->td_maxwin = 1;
    fdata->td_maxend = end + 1;
}

```

When td_end equals 0, we assume that we have to do the initialization described in 'Initializations: 2. Receiver sends a SYN/ACK'.

```

if (!(tcp->th_flags & TH_ACK)) { /* Pretend an ack was sent */
    ack = tdata->td_end;

```

In case the packet does not have its ACK flag set, just pretend it was set by setting ack such that it will match the ack boundaries. Also set its window value to 1

```

} else if (((tcp->th_flags & (TH_ACK|TH_RST)) == (TH_ACK|TH_RST)) &&

```

```

        (ack == 0)) {
        /* gross hack to get around certain broken tcp stacks */
        ack = tdata->td_end;
    }

```

The code above is necessary because there seem to be TCP implementations that set the ACK flag in RST packets but always leave the value of the ack field 0. In such a case, pretend the ACK is valid.

```

if (seq == end)
    seq = end = fdata->td_end;

```

In case the packet contains no data at all, assume it is valid and only look at the ack value below. Passing this packet when the ack field is valid poses absolutely no threat. This code is meant to prevent false (or harmless) blocked packets.

```

maxwin = tdata->td_maxwin;
ackskew = tdata->td_end - ack;

if ((SEQ_GE(fdata->td_maxend, end)) &&
    (SEQ_GE(seq, fdata->td_end - maxwin)) &&
    /* XXX what about big packets */)
#define MAXACKWINDOW 66000
    (ackskew >= -MAXACKWINDOW) &&
    (ackskew <= MAXACKWINDOW)) {

```

SEQ_GE and later SEQ_GT implement sequence number comparison with modular arithmetic (see also [RFC793, section 3.3])

```

    /* if ackskew < 0 then this should be due to fragmented
     * packets. There is no way to know the length of the
     * total packet in advance.
     * We do know the total length from the fragment cache though.
     * Note however that there might be more sessions with
     * exactly the same source and destination parameters in the
     * state cache (and source and destination is the only stuff
     * that is saved in the fragment cache). Note further that
     * some TCP connections in the state cache are hashed with
     * sport and dport as well which makes it not worthwhile to
     * look for them.
     * Thus, when ackskew is negative but still seems to belong
     * to this session, we bump up the destinations end value.
     */

```

The comment above explains why boundary (III) cannot be used.

```

if (ackskew < 0)
    tdata->td_end = ack;

```

This is necessary to 'synchronize' td_end when indeed fragments were passed and the total length is unknown

```

    /* update max window seen */
    if (fdata->td_maxwin < win)
        fdata->td_maxwin = win;
    if (SEQ_GT(end, fdata->td_end))
        fdata->td_end = end;
    if (SEQ_GE(ack + win, tdata->td_maxend)) {
        tdata->td_maxend = ack + win;
        if (win == 0)
            tdata->td_maxend++;
    }

```

This is the update of the relevant state variables with the information from the inspected packet.

```
        ret = 1;
    } else {
        ret = 0;
    }
    return(ret);
```

Figure 3: New state code implementation

The largest compromise that had to be made when implementing the new design was that boundary (III) cannot be checked. IP Filter does not fully reassemble fragmented packets before they are passed (this is supposed to be done by the final destination [RFC1812, section 5.2.1.1] anyway). It does use a limited fragment cache but it can not always give back the total length from packets of which it is known that all fragments have been forwarded by IP Filter. This is noticed by the state engine code in that packets might arrive that acknowledge data that seems not to have been sent. In case this is detected and the acknowledgement is within MAXACK-WINDOW from what was perceived as the last octet sent, it is assumed the ack value is valid and it acknowledges data that was sent.

6. Testing

Testing was first done on a system that did not route packets but that sniffed a network and collected state information from all sniffed packets. IP Filter had to be modified slightly to do this. Whenever a packet was seen that would have been blocked by the state engine, it was logged on this machine.

Such a setup allows for testing the state engine without actually disrupting network traffic because no real filtering is performed. Thus it is possible to test on operational networks which saves the trouble of producing test network traffic.

On the actual (operational) network on which the tests were done, an enormous amount and variety of connections could be tested as the network is used to monitor and administer machines using connectivity from high speed low latency links to low bandwidth high latency links.

7. Timeouts

In order not to fill up kernel memory with state entries it is necessary to add a timeout to each

entry. When the timeout expires, the entry is removed. Of course the value of the timeout can be adjusted according to the (TCP) state the connection is in.

When testing the above stateful filtering design, it turned out that in a number of cases, packets would get blocked because the state entry did time out where it should not have. This was particularly true in the case of TCP half-closed connections as often seen with browsers. It would be easy to set a large timeout on every state entry but on the other hand, state entries should be removed as quickly as possible to avoid unnecessary memory use. This section contains some thoughts on how to reimplement TCP state entry timeouts.

The state timeout code in IP Filter has a state machine for each half of a connection. This state machine more or less, uses the same states as the TCP stack does [RFC793, section 3.2].

Having two state machines is a nice idea because it gives the possibility to look at the status of each half of the connection in the state table. The timeouts however are not always set correctly. As an example: when one half of the connection is in the ESTABLISHED state while the other half has sent a FIN, the resulting timeout should be the same as if the connection would have been fully established since one side of the connection might still be sending data.

Furthermore, when one side of a connection sends a FIN and the other side responds with a FIN/ACK, a constant timeout is used for the so-called 2MSL period. Over time, this timeout has been increased to prevent blocking of retransmits. This has the unwanted side effect that for those connections where the connection ended okay, the state entries linger around unnecessarily long. Especially in HTTP intensive setups, this poses a heavy burden on used memory. As a solution to this problem, a variable timeout should be used. When both ends of a connection have sent a FIN, a relatively small timeout

should be set. This timeout should be such that a possible first retransmit of a FIN will be done within the timeout period. An exponential back-off should then be used to increase the timeout value upon reception of retransmits of one of the FIN packets. Both [RFC1122, section 4.2.3.1] and [RFC2581] specify how to implement retransmission. However given that there are quite a number of incorrect implementations of SYN retransmissions [RS2, section 14.7] and that SYN retransmissions should use the same algorithm as data segments, further study is probably necessary to determine optimal values for the initial value of the timeout as well as for the actual exponential backoff implementation.

8. Conclusions

The new state code has been in operation by quite a number of IP Filter users and seems to work as expected. The most remarkable block of a packet was seen in a HTTP session from the author's home system running FreeBSD (host A) to a Windows NT system (host B). The blocking seemed to point to a bug in the filtering code. Closer examination revealed that this was not true. The relevant packets of the session are depicted below (in tcpdump [TCP] format with line numbers). The last packet was blocked by IP Filter.

```
1 B.80 > A.1102: . 153993:155453
2 B.80 > A.1102: . 155453:156913
3 A.1102 > B.80: . ack 156913 win 8760
4 B.80 > A.1102: . 156913:158373
5 A.1102 > B.80: . ack 158373 win 8760
6 B.80 > A.1102: . 161293:162753
7 A.1102 > B.80: . ack 158373 win 8760
8 B.80 > A.1102: . 162753:164213
9 A.1102 > B.80: . ack 158373 win 8760
10 B.80 > A.1102: . 164213:165673
11 A.1102 > B.80: . ack 158373 win 8760
12 B.80 > A.1102: . 165673:167133
13 A.1102 > B.80: . ack 158373 win 8760
14 B.80 > A.1102: . 158373:159833
15 B.80 > A.1102: . 167133:168593
```

Looking carefully at these packets, we see that packet no 14 seems to be a retransmission of a packet that seems to be lost between packet no 4 and 6. Looking carefully at packet 15, its sequence number and the advertised window of host A (packet 13), it turns out that host A was sending out of window data. It is unclear if this violates the TCP protocol specification [RFC793], but at least it seems like a waste of bandwidth. This situation does not seem to occur often, though it was at least reported by one other IP Filter user [CS]. In fact the old state code would have completely blocked the connection when too much data was sent out of the

advertised window and an earlier packet was lost.

Other blocked packets seen are mostly due to timeouts of state entries and are thus unrelated to the state code itself. Packets that appear lost and were already retransmitted but that are actually not lost might sometimes also result in blocks.

Some minor issues were discovered in the implementation of the new code. The most notable one being an invalid initialization causing retransmits of SYN packets not to match the state entry.

9. Future work

In order to complete the state code, at least one additional feature should be added. Currently, for window advertisements, only the TCP window field is taken into account. For connections involving the TCP window scale option [RFC1323], the results are thus incorrect. The old IP Filter state engine had the same problem so the new state engine did not make things worse in this respect. Still, this omission needs to be corrected in a future version. When this is done, also the timestamp option is to be taken into account so that the state engine will be able to handle wrapped sequence numbers within high speed connections.

Of course, sessions that enter the state table, when they are already established, should be handled better. This was already discussed in Section 5.

Furthermore, the workarounds in the implementation for dealing with fragments should be eliminated. This has to be done in the fragment handling. What is needed is that the state engine is called when it is known that all fragments of a fragmented packet are forwarded by the IP Filter host. In that case, the state engine must also be passed the total length of the packet.

Another useful addition would be to enhance IP Filter such that when a packet comes in that is not yet in the state table, it can verify that the packet actually belongs to an existing TCP connection.

An idea to achieve this, as used by Checkpoint's Firewall-1, is when a packet comes in with the ACK flag set but not the SYN flag, to dynamically probe the receiver of the packet to determine if it is part of a valid connection. This can be done in the following way:

Suppose a packet comes in with the ACK flag set but not the SYN flag, that would be blocked. In case a rule exists that would have led to a state entry that would have allowed this packet to pass, the filter strips the payload from the packet, changes the seq field and forwards the packet. If the packet did not belong to a valid session, the receiver will return an RST. When the filter sees the RST it will drop the RST (thus not giving out any information about the receiver). But if it did belong to a valid session, the receiver will reply with an ACK. This ACK then leads to a state entry after which the connection can go on.

Last but not least, facilities should be added to the IP Filter TCP state code such that it can function in a redundant (high availability) setup. Passing all state changes from one IP Filter host to another one results in too much traffic so something smarter must be designed. Passing the static part of state entries (`struct tcpstate`) on state additions and removals might be a valid option. Once the active IP Filter setup fails and an inactive one takes over it knows the port numbers and addresses of valid existing sessions. The method described in Section 5 can then be used to determine on the fly suitable values for the `struct tcpdata` entries.

Availability

The new state code is available within the IP Filter distribution since version 3.3.0. The distribution can be obtained via the IP Filter Homepage [IPF].

Acknowledgements

The author would like to thank Darren Reed for his IP Filter package that has proven to be one of the most flexible filter solutions available. Thanks go also to the reviewers for their valuable comments.

References

- [BR] brkill, Basement Research,
<http://deep.ee.siue.edu/br/brkill/brkill.html>
- [CS] Constantine Sapuntzakis, personal communication, October 1999.

[GW] Gary R. Wright and W. Richard Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1995.

[IOS12] Cisco IOS Release 12.0, "IP Services Commands".
http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgr/np1_r/1rprt2/1rip.htm

[IPF] IP Filter Homepage,
<http://coombs.anu.edu.au/~avalon>

[NMAP] nmap, Fyodor,
<http://www.insecure.org/nmap/>

[RFC793] J. Postel, "Transmission Control Protocol", STD 7, RFC 793, September 1981.
<http://www.ietf.org/rfc/rfc0793.txt>

[RFC1122] R.T. Braden, "Requirements Internet hosts - communication layers", STD 3, RFC1122, October 1989.
<http://www.ietf.org/rfc/rfc1812.txt>

[RFC1323] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance", RFC1323,
<http://www.ietf.org/rfc/rfc1323.txt>

[RFC1644] R. Braden, "TCP Extensions for Transactions Functional Specification", July 1994.
<http://www.ietf.org/rfc/rfc1644.txt>

[RFC1812] F. Baker, "Requirements for IP Version 4 Routers", RFC1812, June 1995.
<http://www.ietf.org/rfc/rfc1812.txt>

[RFC2581] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", RFC2581, April 1999.
<http://www.ietf.org/rfc/rfc2581.txt>

[RS1] W. Richard Stevens, "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994.

[RS2] W. Richard Stevens, "TCP/IP Illustrated, Volume 3: TCP for Transactions, ...", Addison-Wesley, 1996.

[TCP] tcpdump Homepage,
<http://www.tcpdump.org/>

[UM] Uriel Maimon, "Port Scanning without the SYN flag", Phrack 49, November 1996,
<http://www.2600.com/phrack/p49-15.html>