Active-Active Firewall Cluster Support in OpenBSD

David Gwynne

February 2009

to leese, who puts up with this stuff

Acknowledgements

I would like to thank Peter Sutton for allowing me the opportunity to do this work as part of my studies at the University of Queensland.

A huge thanks must go to Ryan McBride for answering all my questions about pf and pfsync in general, and for the many hours working with me on this problem and helping me test and debug the code.

Thanks also go to Theo de Raadt, Claudio Jeker, Henning Brauer, and everyone else at the OpenBSD network hackathons who helped me through this.

Abstract

The OpenBSD UNIX-like operating system has developed several technologies that make it useful in the role of an IP router and packet filtering firewall. These technologies include support for several standard routing protocols such as BGP and OSPF, a high performance stateful IP packet filter called pf, shared IP address and fail-over support with CARP (Common Address Redundancy Protocol), and a protocol called pfsync for synchronisation of the firewalls state with firewalls over a network link. These technologies together allow the deployment of two or more computers to provide redundant and highly available routers on a network.

However, when performing stateful filtering of the TCP protocol with pf, the routers must be configured in an active-passive configuration due to the current semantics of pfsync. ie, one host filters and routes all the traffic until it fails, at which point the backup system takes over the active role. It is possible to configure these computers in an active-active configuration, but if a TCP session sends traffic over one of the firewalls and receives the other half of the connection via the other firewall, the TCP session simply stalls due to a combination of pfs stateful filtering and pfsync being too slow to cope.

This report documents the protocol and implementation changes made to pfsync which allows stateful filtering with OpenBSD routers in active-active configurations.

Introduction

OpenBSD is becoming increasingly popular as a platform for network security services, in particular network routing and firewalling. Currently there is support for building redundant firewalls with OpenBSD using pf and pfsync in active-passive configurations, however, to scale throughput between networks protected by OpenBSD firewalls it should be possible to configure them in active-active configurations. This project is an attempt at supporting active-active OpenBSD firewalls using pf and pfsync.

This document tries to give some context to the problems with building active-active firewalls with OpenBSD currently, then goes on to explain the changes that were made to enable it.

It assumes some familiarity with the C programming language, concepts relating to the workings of operating system kernels, and the IP network protocols including TCP.

Background

OpenBSD

OpenBSD is a general purpose UNIX-like operating system. As the name implies, it is descended from the long heritage of the Berkeley Software Distribution (BSD for short), which began as a set of modifications to the original AT&T UNIX and eventually grew into a completely independent code base.

The OpenBSD Project was created in late 1995 by Theo de Raadt who wanted to create a free and open source distribution with a focus on "portability, standardisation, correctness, proactive security, and integrated cryptography". While it is usable for a variety of tasks ranging from a desktop system up to file and print services, it is this focus on correctness and security that has let OpenBSD evolve into a highly usable and secure operating system for network security services. OpenBSD ships with a variety of software and capabilities in the base system that allow administrators to build very capable routers and firewalls.

OpenBSD As A Router

OpenBSD can be installed on a wide variety of hardware and with some simple configuration changes can be used as a standard router on IPv4 and IPv6 networks. It also includes support for technologies such as IPsec, Ethernet VLAN (802.1Q) tagging, a wide variety of network tunnelling protocols, a high quality firewall, IP and Ethernet load balancing, and can interoperate with other routers using protocols such as OSPF and BGP.

pf - The OpenBSD Packet Filter

For the majority of it's existence, OpenBSD has shipped with some form of packet filtering software that can be used to enforce network security policy at the IP and TCP/UDP layers for traffic flowing through an OpenBSD box. Originally the codebase included in OpenBSD was Darren Reeds IPFilter. However, in 2001 an issue with the license on that code caused it to be removed from the OpenBSD source tree.

Fortunately, shortly after its removal a developer named Daniel Hartmeier began implementing a new packet filter simply called pf. The code was quickly imported into the OpenBSD system, and development continued rapidly. Since that starting point pf has developed into an extremely capable and fast firewall.

The network policy that pf is to enforce is described in a ruleset which is parsed and loaded into pf inside the OpenBSD. These rulesets are generally just a list of pass or block rules that let you match against packets based on characteristics such as their address family, protocol, and if available their port number. However, pf is also able to perform actions such as normalising fragmented IP packets, and network address translation (rewriting IP addresses or port numbers inside a packet). Once loaded into the kernel the ruleset is then evaluated for every packet that enters or leaves the system. Packets going through an OpenBSD based router will be evaluated by pf twice, once coming into the network stack and again when it is leaving the stack.

Iterating over the ruleset for each and every packet going into the system can be slow though, but fortunately pf is a stateful firewall. This means that pf will try to keep track of connections going through itself. States use characteristics of the packet such as the address family, IP addresses, port numbers, and in the case of TCP, the state and sequence numbers of the connection. These states are organised into a red-black tree inside the kernel, and packets entering pf are compared against existing states before the ruleset is evaluated. If a packet matches a state, pf doesn't evaluate the ruleset and simply allows the packet though.

This means there are three benefits to the pf state table.

Firstly, there is an O(log n) cost for state lookups for a packet (where n is the number of states) vs an O(n) cost for a ruleset evaluation (where n is the number of pf rules). Since most packets will belong to an existing state they will therefore tend to match existing states, providing a huge speedup in the per packet evaluation costs.

Secondly, because pf does have some state about a connection it is able to independently check that a packet does in fact belong to an existing connection, and is not some spoofed packet with similar characteristics (ie, IP or port numbers).

Thirdly, because the state table will allow packets through for existing connections, the ruleset only has to concern itself with matching packets that create connections. This is another security benefit since without the state, rules would have to be written that allow packets in both directions through a firewall. To highlight the difference, here is an example ruleset for TCP going through pf that does not take advantage of the state table:

```
pass in inet proto tcp from any port >1024 to $webserver port 80 no state
pass out inet proto tcp from $webserver port 80 to any port >1024 no state
```

As you can see this ruleset has to allow TCP replies from the web server back through the firewall otherwise the connection will fail. A ruleset taking advantage of the state table would look like this:

```
pass in inet proto tcp from any to $webserver port 80 flags S/SA
```

This rule only allows TCP packets with the SYN flag set (out of the SYN/ACK pair) to the webserver through the firewall. pf by default will create a state for this connection and will automatically allow replies just for that connection back through.

pf is quite scalable, meaning that with a suitable machine you can easily run firewalls with many network interfaces and with thousands of firewall rules. The number of states the firewall will manage is dependant largely on the amount of memory in the system. It is common for pf to handle tens of thousands of concurrent state entries.

pfsync

While it has been established that stateful firewalling is good and a necessary thing for pf to operate at speed, it is unfortunately not a good thing in the event of a firewall failing. For simple routing of packets it is trivial to configure two routers with the same configuration between networks. In the event of one failing you can simply replace the failed unit with the functional one. There are several systems and protocols to automate that failover, indeed, OpenBSD features several such as CARP (Common Address Redundancy Protocol), ospfd (a routing daemon implementing the OSPF protocol), and bgpd (a BGP implementation). These systems allow a router to take over from a failed peer within a matter of seconds, thereby minimising the impact of the failure.

However, if these routers are running pf then existing network connections through the redundant peer will no longer work since it has no knowledge of the state table the failed firewall had built. To compensate for that the pfsync protocol was developed.

pfsync simply sends messages between pf firewalls that notify each other other state inserts and deletions, and most importantly, state updates. Because pf keeps track of the TCP sequence numbers it is important for it to notify its peers when those sequence numbers progress so that in the event of a failure the peer knows where the TCP connection was up to.

pf with pfsync allows the deployment of fully redundant stateful firewalls. In the event of failure the redundant firewall will have a full knowledge of the current state table and will be able to continue forwarding packets for those existing connections. However, because of a semantic in the implementation of the pfsync protocol, it is currently not possible to build clusters where both firewalls are actively able to handle traffic.

Active-Active Routers and Firewalls

It is becoming more common to deploy multiple routers between networks to increase the forwarding performance between those networks. Two routers are obviously twice as capable as a single router at forwarding traffic.

Since OpenBSD can be deployed as a firewall and router, it would be nice to be able to increase performance of such a deployment by simply adding firewalls as necessary. However, as stated above, this is currently not possible. Why this is the case is described in detail below.

OpenBSD Kernel Semantics

pf and pfsync both operate entirely within the OpenBSD kernel, only taking configuration or monitoring requests from userland. It is therefore necessary to understand some of the semantics of the OpenBSD kernel to effectively develop software in that context.

Firstly, despite the recent development for support of multiple processors in the OpenBSD kernel, it still largely operates as if it were running on a single processor system. On a SMP system only one processor is able to be running the kernel at any point in time, a semantic which is enforced by a Big Giant Lock. There are portions of the kernel which are not protected by the Big Giant Lock, but they are not relevant to this discussion. The network stack and therefore pfsync still run under the Big Giant Lock.

There are two contexts for code execution in the kernel: interrupt context and process context. The differences between these two contexts affect how the locking of data structures and code are performed, and also affects which internal kernel APIs are available or appropriate for use.

The kernel has a process context when it has been asked to perform an action by a userland process via a system call. The OpenBSD kernel is non-preemptible, meaning that any code running in the kernel has to explicitly give up control before anything else in the kernel will run. If a critical path is only modified from a process context then simply having process context is sufficient to guarantee exclusive access to that section. However, if a critical path may sleep or yield control of the CPU, then additional locking is required. The

OpenBSD kernel provides reader/writer locks for processes in the kernel to lock against each other.

The only exception to the kernel's non-preemptibility is for interrupts. Interrupt handlers may run at any time unless their interrupt source is masked and therefore prevented from running. Interrupt handlers are established at an interrupt priority level (IPL). To prevent interrupt handlers you simply mask interrupts at that level before executing your critical path. IPLs, as their name implies, are levelled. A high IPL prevents interrupt handlers established at lower priorities running at the same time. These IPLs range from IPL_NONE (no interrupts are masked) up to IPL_HIGH (all interrupts are masked). Interrupt handlers are run at their IPL, meaning that if you're currently executing code in a hardware network interrupt handler established at IPL_NET, you are guaranteed that no other interrupt handler at IPL_NET will run at the same time.

This ability to mask interrupts is the fundamental locking primitive in the OpenBSD kernel for data structures used in interrupt handlers. When a userland process enters the kernel, the effective interrupt mask level is at IPL_NONE, meaning no interrupts are masked at all. If this code wishes to modify structures that are also handled during interrupts, it must raise the IPL to prevent those interrupt handlers from running.

The only other large difference between interrupt and process contexts is that you cannot yield in an interrupt handler. The code must return, it is not able to sleep since sleeping relies on being able to switch from the current process to a new one.

The network stack in OpenBSD runs at two IPL levels: IPL_NET and IPL_SOFTNET. IPL_NET is used by drivers for network hardware and protects their data structures. The majority of the network stack runs at IPL_SOFTNET which is a lower priority than IPL_NET. Packets move between the network stack and the hardware via queues protected with IPL_NET. This allows packet processing to occur at the same time as packets are moved on and off the hardware. Packets may be received and queued for processing while processing of other packets is still in progress. It is therefore possible for many packets to be processed in a single call to the softnet handlers.

There are several softnet interrupt handlers implemented in the OpenBSD kernel, generally one per address family. Each of these handlers can be run from software by generating a software interrupt anywhere in the kernel. pf is run when the softnet interrupt handlers for the IPv4 and IPv6 protocols are run. Each handler de-queues a packet and passes it to pf for testing. If pf passes the packet it is then allowed to go down the rest of that particular address families stack, otherwise the packet is dropped. For a packet coming out of the stack, it is tested by pf before being put on an interfaces send queue. If pf makes a change to its state table because of that packet, it in turn notifies pfsync with that state. All of this occurs at IPL_SOFTNET.

Another relevant detail about the OpenBSD kernel is that there are no readily usable high resolution timers that can be used to schedule events in the future with. The most usable timer generally has a resolution of 100 ticks a second. This is important to know if you're trying to mitigate some activity within the kernel.

pfsync In Depth

pfsync can be thought of as being made up of two parts: the protocol and the implementation.

The current version of the pfsync protocol is version 4. The protocol is surprisingly simple. Each message is an IP (the protocol allows either IPv4 or IPv6) encapsulated datagram that contains a small header prefixing a series of messages. The IP protocol ID for pfsync packets is 240, it does not get encapsulated inside a UDP or TCP. The header describes the type of these messages, and the number of them to expect in the packet. Each packet may only contain messages of the one type specified in the header, and it may only include as many messages as may be contained in a single IP datagram without needing to perform fragmentation. The size of the IP datagram is determined by the medium over which it is transmitted.

An exact knowledge of the layout of the header and the messages is generally unnecessary, however, some details are useful to know.

While a pf state may be uniquely identifiable by the characteristics of the packets it represents (ie, the address family, protocol, port numbers, etc), it can be inefficient to exchange these details between pfsync peers.

To address this inefficiency, an arbitrary state must be uniquely identifiable by all peers exchanging pfsync messages by a 96 bit key made up of a 32 bit "creator id" and a 64 bit "state id". Each peer in a pfsync cluster randomly generates a 32 bit creator id on boot which uniquely identifies that particular peer within the cluster while it is alive. Each state that the peer creates must be uniquely identifiable within the system by a state id. The combination of those two values allows pfsync to uniquely refer to any state amongst all its peers.

The message types within the pfsync protocol are:

PFSYNC_ACT_CLEAR	Clear pf state table A host will send this message when an administrator on the system has requested that the state table be flushed. On receiving this message the host must flush its state table.
PFSYNC_ACT_INS	Insert a new state into the state table When pf has processed a packet and decided to create state for it, it will request pfsync send a message of this type to describe that state to its peers A peer receiving this message creates a new state for the traffic described in the message and inserts into its local state table.
PFSYNC_ACT_UPD	Update a state When a packet causes a state to be modified, eg, a TCP packet will move the sequence numbers within the state along, it will generate a message of this type describing the state. A peer receiving this message attempts to locate an existing state and updates it with the new details. If no state has been found it may create and insert a new state with the specified details.

PFSYNC_ACT_DEL	Delete a state When a peer has received a packet that terminates a state, or a state times out, it will send a message of this type. A peer receiving this message will attempt to locate this state and remove it from its local state table.
PFSYNC_ACT_UPD_C	"compressed" state update This is semantically the same as the PFSYNC_ACT_UPD message, except instead of exchanging all the state details it only contains the creator and state ids to identify which state the update is for.
PFSYNC_ACT_DEL_C	"compressed" state deletion This is semantically the same as the PFSYNC_ACT_UPD message, except instead of exchanging all the state details it only contains the creator and state ids to identify which state to delete.
PFSYNC_ACT_INS_F	insert fragment pf may maintain a cache of fragmented packets and reassemble them to allow proper stateful filtering on those fragments. This message is intended to share the contents of the fragment cache between peers.
PFSYNC_ACT_DEL_F	delete fragment this request is intended to remove a fragment from the shared fragment cache.
PFSYNC_ACT_UREQ	request "uncompressed" state update If a pfsync peer receives a compressed update for a state it cannot locate by the creator and state ids, it can request an uncompressed state update so it can enter the current state details into its local table. A peer may request a a bulk update by sending a message with the creator and state ids set to 0. A peer receiving this message will search its local state table for an entry identified by the creator and state ids and will send a message of type PFSYNC_ACT_UPD in response. If the peer receives a request with the creator and state ids set to zero it will initiate a send of update messages for all of its states.

PFSYNC_ACT_BUS	bulk update status A peer will send this message after completing a bulk send of update messages. A peer receiving this message may be confident that it has a usable copy of the pf state table and may actively participate in the cluster.
PFSYNC_ACT_TDB_UPD	TDB replay counter update If the peer is acting as an IPsec gateway, it will send updates to the replay counters on the IPsec security associations to its peers. A peer receiving these messages will search for an IPsec security association with the same parameters and update the replay counters with the values provided by the pfsync message. This can be used to provide failover for IPsec gateways

pfsync doesn't recognise individual peers in the cluster, it only deals with communicating with the cluster as a whole. pfsync by default sends packets to a multicast group and receives packets destined for that multicast group. If a peer receives a compressed update it knows nothing about, it will send an update request to the multicast group, not to the specific peer the compressed update came from.

The current implementation of pfsync is implemented as a pseudo network interface driver. This is largely to allow for the creation and configuration of pfsync via ifconfig, which is a familiar administration tool for network services in UNIX-like systems. Despite being a network interface it does not provide an endpoint for traffic to be routed in or out of in the kernel.

A pfsync interface requires very minimal configuration to become usable. It is enough to define which interface the pfsync packets are to be exchanged on, and then bring the pfsync interface up. If pf is active on the system then it will begin to notify pfsync of inserts, updates, deletions, and clears of states in the pf state table.

Internally pfsync builds pfsync packets on the fly when pf issues notifications to it. The first notification from pf will cause pfsync to allocate memory for a packet to be built in, and will write the message for that notification into the packet. A second notification from pf will cause pfsync to search that packet it is building for a message about that same state. If the state was found in the packet already, the message is updated with the new parameters, otherwise a new message is added to the packet about the state.

pfsync will eventually transmit the packet it is building on several events.

The first is from a timeout that is scheduled one second from when the packet was first allocated. When that timeout is hit the packet is sent out with whatever updates were made during that 1 second.

The second condition is if any particular state described in a pfsync packet is updated more than a certain number of times. The maximum number of updates a state in a packet may receive is 128 by default, but that value is configurable by the user.

Both of these conditions are there to try and mitigate against pfsync generating a single packet for each and every state update that pf notifies it of. Without this mitigation the pfsync traffic between peers in a cluster could conceivably exceed the traffic pf is firewalling.

The third condition a packet is transmitted on is caused when pfsync has to switch message types for the packet it is building. Because pfsync packets may only contain messages of one type, if pfsync is notified of a different type of action to the one it was building a packet for then it will simply transmit the current packet immediately. It will then allocate a new packet for messages of the new type and will start filling it in again.

The fourth condition a packet will be sent out on is if it is in response to an action that requires peers know about the change immediately. For example, if pf notifies pfsync of a state table clear, pfsync will build that packet and send it immediately. Also, if a peer requests an uncompressed update about a state, that message is also built and sent out immediately.

Lastly, if an additional message will cause the packet to grow larger than the MTU of the physical interface specified for the pfsync traffic, the current packet will be transmitted immediately.

When a pfsync packet is received by the network stack, it is processed immediately to the pfsync input routine. The input routine simply iterates over the messages that were specified by the header and takes action accordingly. Generally the implementation follows the actions that the protocol specifies that were described above. However, some aspects of the implementation of the pfsync receive code are smarter about their actions than what the protocol would imply.

For example, it is possible that a host will receive an update for a state that has been modified locally as well as by another peer, which is quite likely when you are moving from a passive to active role in a cluster. In that situation it will try to take the most current details from both the state and the update it just received and merge them together. If it has determined that any part of the peers version is stale, it will immediately send an update of the state based on its merged information.

Also, pfsync attempts to keep track of how current it thinks it is with respect to its peers, and it feeds that state back into the system. When the pfsync interface is first brought up it marks itself as "not ok" and sends out a request for a bulk update by transmitting a PFSYNC_ACT_UREQ with 0s set for the creator and state ids. It is only after it has successfully received a PFSYNC_ACT_BUS message that it will move itself to "ok". If the firewall is using carp as part of an active-passive cluster, the pfsync ok state is used to demote the carp interfaces so it will not receive traffic until it is ready to continue forwarding traffic with the existing states. If pfsync does not receive such a message, it will request a bulk update another 12 until it times out and moves to the ok state under the assumption that it is the only peer still present.

It is worth noting that the current implementation makes no effort to mitigate against the generation of packets for actions requiring "immediate" transmission. ie, if a peer requests uncompressed updates for 8 states, the current pfsync code will generate 8 separate packets, one for each of the uncompressed updates it generates.

Also, because pfsync only builds one type of packet at a time, it is susceptible to generating excessive traffic simply because pf notifies it of different events all the time. It is uncommon for a state insert to be followed immediately by another state insert. The same is true for state deletions. It is common for pfsync to be able to build packets for several updates and mitigate against frequent sends of those packets, but that mitigation is offset by the inserts and deletes that occur too.

It is also worth noting that this implementation makes no attempt to detect if the traffic associated with a particular state is split over two of the systems in the cluster. For traffic going through a pf firewall that requires up to date state information to proceed, the lack of this detection will prevent the traffic from moving forward.

For example, because pf tracks a TCP sessions sequence numbers, and because TCP uses packets in both directions to move those sequence numbers forward, a pf firewall needs frequent updates from its peers if it can only see half the TCP packets. Without an update from a peer pf will not move the states sequence numbers forward. pf will drop TCP packets that move beyond the TCP sequence number windows (plus a bit of fuzz).

Because it is almost impossible for a TCP session to receive 128 updates (ie, 128 new packets from the wire) without moving the session forward, it is in the worst case only the 1 second timeout which causes pfsync to send its update out. Therefore an active-active pf cluster may only allow new packets in a TCP session through for a small portion of every second.

In response to this the TCP endpoints will back off the speed at which they send and attempt retransmission. The TCP session will move forward, but only at a small fraction of the speed that a single pf firewall would forward at.

This over mitigation of messages proves to be the fatal flaw that prevents pfsync and pf to be usable in active-active clusters.

Currently pfsync only generates and receives IPv4 datagrams.

Approach and Execution

As described in detail above, the big problem with the current implementation is that it mitigates sending of pfsync packets too much, ie, in a firewall cluster with traffic split over two peers, updates aren't exchanged rapidly enough for the states on each firewall to move forward fast enough to keep up with the actual traffic. This is especially (or perhaps only) problematic with TCP traffic, which requires extremely current information from both sides of the connection to move the TCP sequence numbers forward.

Two attempts were made to try to solve the active-active problem in pfsync, firstly simple modifications to the existing implementation, and then as a result of that an almost full blown rewrite of the code.

Changes To The pfsync v4 Implementation

A large number of different approaches at dealing with states with traffic split over two peers in an active-active firewall cluster were evaluated and tested as part of the initial problem solving. This stage could be considered the exploratory surgery and was required so I could gain familiarity with the problem and the current implementation. However, all of the solutions except the final solution presented here were rejected as being unsuitable.

These solutions ranged from allowing packets for split TCP states to accept packets if the sequence numbers are on the edge of the window as stored in the state, to decreasing the maximum timeout on pfsync packets from 1 second down to small fractions of a second. All of these solutions either compromised the security provided by pf, or hurt the performance too much in existing use cases to be feasible.

Despite the long road to the changes made to the v4 code, the final changes were actually quite minimal. After a lot of trial and error it was decided that it was necessary for each firewall involved in a split path for TCP sessions to be notified of updates to that state as soon as possible. This in turn required the ability to detect when two peers were involved in a split path.

Detecting a split path turned out to be simple. Whenever an update to a state is received via the pfsync protocol, we record the time the update arrived at in the pf state structure. Then, when an update to the same state arrives via pf, we simply find the difference between the current time and the last time the state was updated via pfsync and assume the state is split if the difference is less than some arbitrary value, 4 in our case.

If we never get an update about a state via pfsync it means no other peer was involved in handling that state, therefore the timestamp in the state will always be 0 (the default value). The comparison between it and the current time will always be greater than 4. A peer that is handling packets for that session will send pfsync packets out about it though, so that comparison will evaluate to true and we know the state is split at that point.

The other feature of this mechanism is if the paths in both directions for this state merge onto a single firewall, that firewall will no longer receive updates for the state via pfsync. The timestamp on the state will no longer be updated, and the comparison between it and the system time will eventually fail as time moves forward. This means the same mechanism for detecting split states also allows us to detect when a state is no longer split.

With that mechanism in place it was trivial to modify the code to immediately send a pfsync state update about our own state changes to the peer.

These changes were successful, ie, if you had a pair of firewalls called A and B between two networks x and y, you could configure the route from hosts on network x to network y to go via firewall A, and the routes from hosts on network y to network x on firewall B, you could then successfully run pf as a fully stateful firewall with traffic for the same session split between those two firewalls.

The problem with these code changes is that they caused pfsync to become extremely chatty. Every single packet involved in a split session going through a firewall would generate a corresponding update from pfsync. Since packets are evaluated by pf twice (once coming into the system and once leaving the system), two updates were being generated for every forwarded packet. Additionally, for every single update for that split session we received from the firewall peers, we hit the merge state case in the pfsync update parsing code which cause us to send an update out again. Because of this the majority of the tests we did with asynchronous paths for traffic through firewalls showed the pfsync traffic between two firewalls was several times the traffic of the actual traffic we were forwarding over the firewalls. Obviously causing more load than what we're attempting to forward is unacceptable.

One of the discoveries made during the tinkering with the v4 code was that there was a race between the forwarding of a packet that caused the creation of a state, the pfsync packet it generates, and when the reply from the host the packet was for is seen by a peer.

If we forward that packet on to the host, and that host sends a reply through another firewall, it is likely that the 1 second timeout on the pfsync packet describing that state has not hit that second firewall yet. Because pf is stateful, it will probably reject or drop that reply rather than forward it on like it should.

In response to this problem a new pfsync message was created called PFSYNC_ACT_IACK. When a firewall creates a state, instead of forwarding the network packet that created the state on immediately, we delay transmission for a short period. While that first packet is delayed we immediately send a pfsync state insertion message. Peers that receive that state insertion message then send an insert acknowledgement message to the first firewall, which in turn uses that to trigger the transmission of the packet that was delayed. If no firewall is there to acknowledge the insert, a timeout on the packet fires and causes it to be transmitted anyway.

It was at about this point that it was decided that the code required significant surgery to avoid transmitting too many pfsync packets. Since the code was going to have to be heavily modified to fix its behaviour, slipping an update to the wire protocol was also allowed, especially if it would help mitigate the number of packets pfsync intended to transmit.

To summarise, it was determined that not only does the pfsync code mitigate sending of pfsync packets too much, it also doesn't mitigate them enough.

pfsync v5

The only really major flaw with version 4 of the pfsync protocol was its inability to contain multiple types of messages within the same frame. It could only contain packets with state inserts, or updates, or deletions, or so on, but not a mix of those message types. This becomes a problem if you're trying to mitigate the number of packets sent, but needed to send a lot of messages of different types.

Therefore the only real change between pfsync v4 and pfsync v5 was the removal of the message type and counter fields in the pfsync header, and the introduction of a sub-header. Several different messages can now be placed in a pfsync packet, all prefixed by different sub-headers.

A new message type was added to the protocol as part of the new version too. To cope with the race between forwarding a packet that creates a state, and it's reply hitting a peer before the state was received by that peer, it was envisaged that the first peer forwarding the first packet would defer sending of that packet until it received an acknowledgement of the insert from a peer. That acknowledgement is represented by a new PFSYNC_ACT_IACK message type that simply contains the creator and state ids of the state that was just inserted.

To summarise, pfsync v5 is largely the same as pfsync v4, except for a new message type (IACK) and the ability to send multiple types of messages inside a single pfsync packet due to the addition of a sub-header in the frame format.

The following changes to the OpenBSD kernel were made to address the inadequacies discovered by the previous implementation.

Firstly, the pfsync packet parsing code in the kernel has been split up to avoid the use of switch statements to pick which code is used to parse which message types. Switch statements in C have an O(n) cost where n is the number of options you're switching between. Instead, the parser is picked by using the pfsync action in the sub-header id as an index into a table of function pointers. This moves the cost of picking a parser for a message to an O(1) cost.

Next, pfsync packet generation was moved from being done "on the fly" when pf notified us of updates, to being done only when a packet was ready to be transmitted. This in turn required that the state of a pf state with respect to pfsync be stored somewhere other than the packet currently being generated.

Previously the code determined if a state was already scheduled to be sent out on the wire by iterating over the packet it was building in memory. This is another O(n) cost where n is the number of states already scheduled in the current packet.

Because we also wanted to be able to send multiple types of messages in the same packet, it is now also necessary to mark what type of message the state had associated with it. Several queues for holding the states were created inside pfsync, one for each type of message that the state could appear in on the wire. The pf state structure was extended to record which of these queues it was currently held on. Now it is an O(1) cost to determine where a state is with respect to pfsync.

When the packet is eventually scheduled for transmission, the pfsync code walks all these queues and serialised the states into messages within the packet. As it does this iteration it simply marks the pf states as not being queued within pfsync anymore.

Another side effect of moving to queues of states was that it is now easy to move states between queues in response to pf notifications or requests from other peers. For example, pf itself could schedule a compressed update for a state which would leave it on the compressed update queue and marked as such. A peer can then request an uncompressed update for it. Where the previous implementation would have sent the previous message out immediately so it could begin a new packet with the uncompressed message type, the new code now can trivially figure out that it should simply remove the state from the compressed update queue and place it on the uncompressed update queue and mark it as such.

Next, a mechanism to mitigate against having to send "immediate" packets out immediately was developed. Since there are no general purpose high resolution timers available in the OpenBSD kernel, it was decided that a new softnet interrupt handler be created specifically to flush the pfsync message queues into a packet for transmission.

Because all the events in pfsync are generated by code that is running at softnet, ie, the pf tests for network packets on both the systems input and output queues and the processing of pfsync packets, it is possible to queue updates for all the states touched during that processing and schedule a softnet interrupt for the pfsync packet generator. Because that code is running at softnet it masks the pfsync packet generator scheduled at softnet and prevents from running until after all the current packet processing is finished.

Additionally, on systems with busy network interfaces it is typical that you process several dozen packets per call to the softnet interrupt handlers. Any updates requiring immediate transmission of a pfsync packet can bundle all those updates into a single update before the packet transmission code is run.

Finally, the features from the pfsync v4 reworking were brought over to the new pfsync v5 code. The time at which a pf state was last updated by a pfsync packet is stored in the pf state. If an update from pf for a state occurred within two seconds of the update from the pfsync system, it is determined that the traffic for that state is now split over two peers in the cluster and it is marked for "immediate" transmission by the softnet handler.

The IACK handling made against the pfsync v4 implementation was also brought over, however, instead of the pointer to the packet and its timeout being stored in the state, a separate queue for deferred packets was added to the pfsync code. This was done because the space required for the packet pointer and the timeout was considered excessively wasteful for every state to store when it was only to be used for an extremely short period of time. This was weighed against the extra cost in terms of CPU time of handling that separate queue, which was considered worth it for the memory savings.

Because of the changes to the wire protocol, tools outside the kernel that parsed pfsync packets must be updated to understand the new packet format. The only tool in the OpenBSD source tree that parses those packets is tcpdump. This program was updated to handle the new packet format as part of this work.

Overall the changes to the OpenBSD system resulted in a unified diff to the source tree touching 10 separate files and totalling over 4000 lines of changes.

Results

During the implementation of the new version of the pfsync protocol, several problems in the OpenBSD kernel were uncovered.

Insufficient splsoftnet() Protection In The Network Stack

The pfsync code assumed that all paths into it from pf would hold splsoftnet, which was an assumption that was necessary to guarantee that the pfsync data structures were sufficiently locked.

Testing of the code during development kept showing corrupt data structures in pfsync and in the mbuf (network packet handling structures in the OpenBSD kernel) pools. These corruption's inevitably led to panics in the OpenBSD kernel. Because of this corruption, calls to splassert (a function that checks if the current CPU interrupt mask is at least as high as the level required) were added to the entry points into the pfsync driver to check if softnet was actually held.

It was discovered that there were cases when pfsync was being called without softnet being held.

When a normal network interface is brought up, ie, configured to send and receive packets, the IPv6 protocol is also enabled on that interface and immediately generates IPv6 duplicate address detection packets. These packets are built and sent up the network stack without the spl being raised

This meant that large portions of the network stack, including pf and pfsync, were being used without appropriate protection. If the system received a packet during this time it was likely that the state in these systems would become corrupted. This was indeed the case we discovered with pfsync.

As a result of this discovery several code paths from the IPv6 stack had the necessary calls to splsoftnet() added to provide the appropriate protection required by the network stack.

This fix was developed by Ryan McBride and committed to the OpenBSD source tree. Further splasserts have also been added to other parts of the network stack to try and catch any further problems.

ip_output() Panic

After pfsync generates a packet to be transmitted, it hands it to the ip_output function for it to be sent up the stack and on to the wire. A combination of three factors caused some of these packets to generate a panic.

Firstly, pfsync generates packets with the DF (don’t fragment) flags set in the IP header. This means that the network stack should not break the packet up into multiple IP frames if it is too large to transmit.

Secondly, pfsync sends to a multicast address by default. This changes how ip_output behaves internally in several ways, but most relevant here is that packets sent to multicast addresses don’t necessarily result in a route lookup.

Lastly, due to an accounting error the pfsync code would generate network packets that were too large to be transmitted. When ip_output is asked to deal with a packet larger than the interfaces MTU that has the DF flag set, it takes that opportunity to check if the route to the destination address needs to be updated with a smaller MTU.

Because the pfsync packet was being sent to a multicast address the local variable inside ip_output holding the destinations route was not set. When ip_output tried to use that variable to update the route's MTU it generated a system trap caused by an access to an invalid memory address.

The fix to this problem was developed by Claudio Jeker. A simple check to see if the route variable was not NULL was added before any attempt to use or modify the route was done. This fix was also committed to the OpenBSD source tree.

Functional Results

Unfortunately the time taken to implement the new protocol and its handling in the OpenBSD kernel and the debug it left little time for testing and evaluation of performance. However, despite this the initial results are much better than expected.

The new code results in less pfsync messages being exchanged between pf firewalls when compared to the old code in the exact same setups. Users with active-passive setups will gain a benefit from the new code because of this, and may also gain a small amount of CPU time back due to the relative efficiency of the new implementation.

Even better, the new code makes active-active firewalls actually work.

This code was developed with the idea that async paths through pf firewalls was a worst case situation for a cluster of firewalls, and that the code modifications to support it were simply to make that case tolerable rather than unusable like the current code. Because of this it was predicted that traffic forwarded over async paths would be at a rate noticeably slower than the same traffic going over a single firewall.

This was indeed the case with the simple modifications to the pfsync v4 code base. There was always a significant slow down with any traffic over async paths compared to the traffic sent over a single firewall.

However, it appears that the new pfsync protocol and implementation scales a lot better. Relatively slow TCP connections, (in my test setup it was TCP sessions with less than 10 thousand packets per second) do not experience any slow down when split across async paths. At this rate it is trivial for the pfsync traffic to keep up with the rate at which the TCP session window moves forward. As the packets per second of the forwarded connection increases above that threshold, the pfsync updates begin to struggle with keeping each firewalls view of the sequence numbers in sync. As a result the TCP state matching code in pf begins to drop packets that have moved beyond what it thinks the windows should be.

This result is in line with the expectation stated above. However, compared to the behaviour of the old implementation where async traffic simply stalls, this is a massive improvement.

Experience has shown that the majority of connections through a firewall tends toward lots of relatively slow streams, rather than one massively fast stream. From that point of view it is possible that the worst case behaviour with the new code will not be noticeable in practice.

The characteristics of the new protocol are also heavily dependant on the behaviour of the hardware that it is running on. The quality of the interrupt mitigation and the choice on network cards has a heavy influence on how many packets are processed at softnet. It is unfortunate that more time was not available to gain some understanding of these interactions.

Conclusion

Summary and Conclusion

The new protocol and the rewrite of the pfsync kernel code is a success. Not only does it allow active-active firewall clusters to be built, but it also improves the performance for currently supported active-passive configurations by reducing the network load of the associated pfsync traffic.

With this code it is now possible to increase the throughput between two networks by adding firewalls, rather than having to scale the performance of a single active firewall which takes all the load. Each peer in such an active-active cluster will be able to act as an independent gateway from the point of view of the client systems, but the network administrator will still have the ability to apply policy with the pf firewall and not have to give up security for the performance gained by running multiple gateways. Effort should be spent attempting to engineer the network so both the send and receive path will travel over the one firewall, but if that engineering fails it is possible that the service will degrade rather fail.

The new pfsync implementation has been committed to the OpenBSD source tree in time for the 4.5 release. However, several features of the rewrite described here were disabled for the release which prevent it from being usable in active-active setups. 4.5 will only allow users to take advantage of the other performance improvements made with the new protocol and code.

Future Work

Two features relating to async paths need to be fixed before it can be usable in active-active firewall setups.

Firstly, the merging pfsync state updates with the local pf state needs to be rewritten. This is confused by the synproxy functionality in pf which abuses the TCP state fields. That case has to be handled correctly before the code can be integrated.

Secondly, the currently implementation does not handle deferrals of initial packets correctly. Deferred packets appear to be only sent via the timeout on those packets, the peers in a cluster do not generate the IACK messages necessary to have that packet sent sooner. This leads to a noticeable delay for new sessions created through a firewall which is unacceptable in practice.

Working through those issues should be a relatively trivial task given more time and a better test environment.