Chapter 4

Postfix ARCHITECTURE

Postfix has gained popularity as an MTA because of it has the same interface like *sendmail* , yet it does not have problems with security and reliability. The architecture of Postfix closely follows the design principles of *qmail*. A lot of security patterns of the *qmail* architecture are found in Postfix. Additionally, Postfix improves performance. It shows better performance than *qmail* and *sendmail* in benchmark tests [2] [3] [27]. In many cases Postfix uses the same security patterns like *qmail*. However, there are design areas where Postfix uses different patterns and these design choices are motivated by performance. Along with that, there are mechanisms adopted at different places in Postfix architecture to improve performance. Thus the Postfix architecture provides a good example of consideration of performance along with security and reliability.

4.1 History of Postfix

The primary goal of Postfix is security and performance. Another key goal is *sendmail* compatibility. Postfix acts as a direct and seamless replacement for *sendmail* with the users remaining transparent of the change.

Postfix was written by Wietse Zweitze Venema to provide an alternative MTA for standard Unix servers. Wietse Venema was in a sabbatical in IBM T. J. Watson Research center when he started the project of developing a secure and faster alternative of *sendmail*. Testing on Alpha version began in January 1998 and the beta release came out for public use in December that year.

Initially the MTA was named VMailer. Charles Palmer was the leader of the project and the main IBM contact. He proposed the name Postfix which was adopted eventually.

Although Postfix was intended to replace *sendmail*, another target was to get a faster version of *qmail*, yet retaining the security that *qmail* is famous for. Thus, the interfaces of Postfix is more similar to *sendmail* than *qmail*'s interface, but architecturally it is closer to *qmail*, because of the adoption of security patterns and principles used by *qmail* author Daniel Bernstein.

4.2 Postfix Architecture

Postfix is based on semi-resident, mutually-cooperating processes that perform specific tasks for each other, without any particular parent-child relationship. This gives better insulation than using one big program. In addition, the Postfix approach has the advantage that a service such as address rewriting is available to every Postfix component program, without incurring the cost of process creation just to rewrite one address.

Postfix is implemented as a resident master server that runs Postfix daemon processes on demand. These processes are created up to a configurable number, are re-used for a configurable number of times, and go away after a configurable amount of idle time. This approach reduces process creation overhead while still providing good insulation.

The Postfix architecture is based on programs, queues and lookup tables for configuration management. The core Postfix program is *master*, that runs in the background all the time. This spawns processes on demand to scan and process the queues. Instead of having one mail queue like *qmail*, Postfix has 5 different mail queues. Another interesting aspect of its architecture is the use of lookup tables for describing policies.

The architecture follows the structure of the generic MTA presented in chapter 1. The following three sections describe the processes, queues and tables in Postfix.

4.2.1 Postfix Processes

The default installation of Postfix has three daemon processes running - *master*, *qmgr* and *pickup*. The *pickup* program determines when messages are available for routing by scanning the *maildrop* queue. The central message routing for Postfix is handled by the *qmgr* program.

Email messages from remote hosts are received via SMTP by the *smtpd* process. *smtpd* hands the messages to *cleanup*. The *cleanup* process does the same tasks as *qmail-smtpd*. *qmail-*

smtpd delivers the message to *qmail-queue* to put in the queue. Postfix's *smtpd* process delivers messages to *cleanup* daemon. If not run from server (in stand-alone mode) *smtpd* deposits messages directly into the *maildrop* queue.

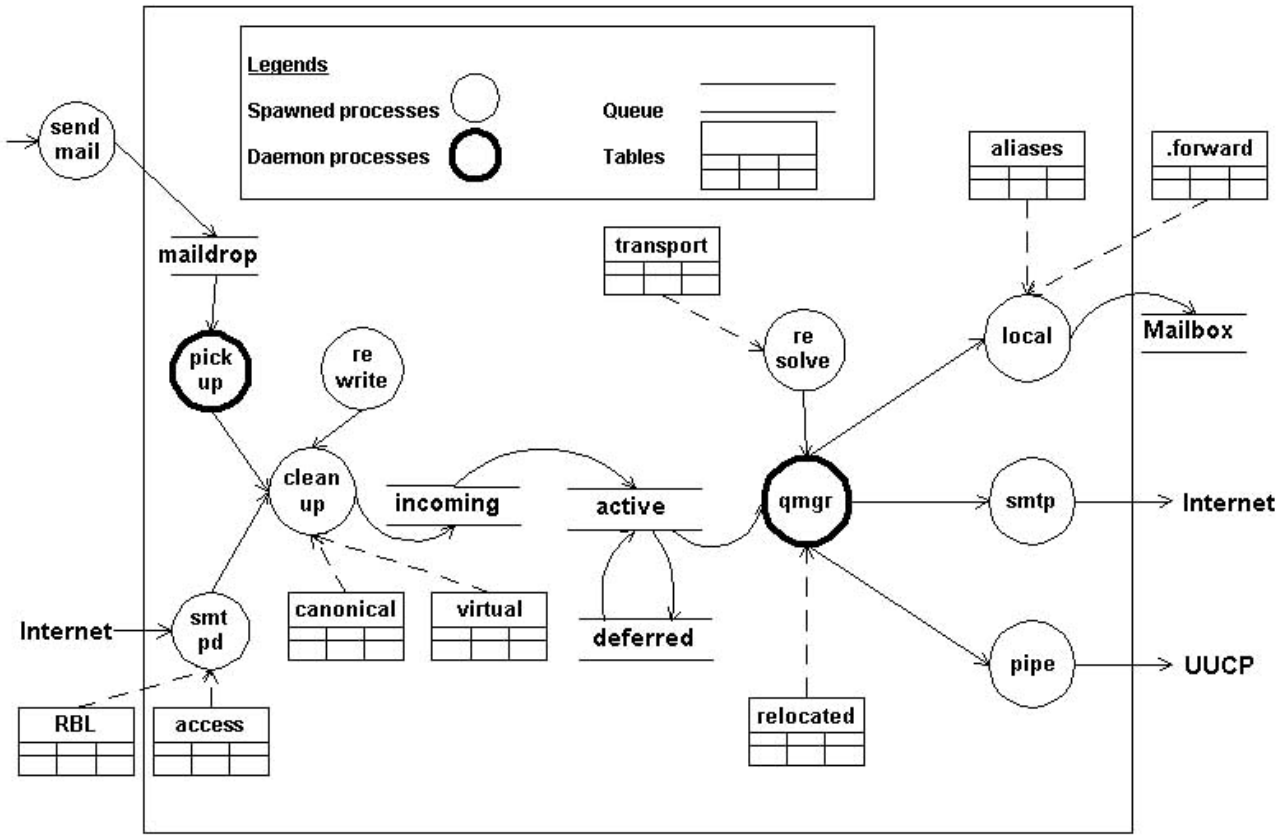

Figure 4.1: Postfix Architecture

For local delivery, Postfix has a program called *sendmail* that doubles as the original *sendmail* program to forward messages from local users to the mail queue (*qmail-inject* in *qmail*). However, instead of sending the message to *cleanup*, it writes in the world-writable *maildrop* queue.

qmail-smtpd handles messages to *qmail-queue* for writing in the queue. In Postfix the process that writes into queue is *cleanup*. However, unlike *qmail-queue* , *cleanup* processes the incoming mail headers and formats them with the help of *trivial-rewrite* and places them in the *incoming* queue. The input of the *cleanup* process comes from the *maildrop* queue via the *pickup*

program (local messages) or directly from the *smtpd* process (remote messages). The *cleanup* program checks the RFC 822 header fields to ensure that they conform to a specific format. For that, it inserts missing From:, Message-ID: and Date: fields and extracts and rewrites addresses of recipients in the To:, Cc: and Bcc: fields. For address re-writing it uses definitions in the *canonical* and *virtual* tables.

The *qmgr* daemon (similar to *qmail-send*) awaits the arrival of incoming mail and arranges for its delivery via Postfix delivery processes. Address rewriting and related tasks are done in this phase in *qmail*. However, in Postfix, the address-rewriting mechanism are re-factored and moved to the *trivial-rewrite* program. This makes *qmgr* simpler and smaller than *qmail-send*.

Postfix has separate processes for sending mails using different protocols. This is different from *qmail* architecture that has *qmail-local* doing local delivery and *qmail-remote* doing all types of remote delivery using different protocols. This complicates the architecture and incorporation of a new protocol is significantly difficult. When considering protocol extensibility, Postfix architecture is much scalable than *qmail*.

The message delivery requests from the *qmgr* process are handled by the SMTP client, i.e., the *smtp* process. Each request specifies a queue file, a sender address, a domain or host to deliver to, and recipient information. *smtp* runs from the master process manager. It looks up a list of mail exchanger addresses for the destination host, sorts the list by preference, and connects to each listed address until it finds a server that responds. When a server is not reachable, or when mail delivery fails due to a recoverable error condition, the SMTP client will try to deliver the mail to an alternate host.

Local delivery is handled by the *local* process. Mails delivered in standard *sendmail* "mailbox" or "mbox" format mailboxes or *qmail* style "maildir" mailboxes. This process is operationally similar to *qmail-local*.

The *pipe* process forwards messages from *qmgr* to some external program (UUCP, etc).

4.2.2 Postfix Queues

Postfix uses several different queues for message storage and management at different points of delivery.

The *maildrop* queue stores messages that have been submitted via the *sendmail* command of Postfix. This queue is drained by the *pickup* process. The formatted messages end up in the *incoming* queue.

Another important queue is the *active* queue. This is somewhat analogous to an operating system's process run queue. Messages in the active queue are ready to be sent (runnable), but are not necessarily in the process of being sent (running). *qmgr* is a delivery agent scheduler; it works to ensure fast and fair delivery of mail to all destinations within designated resource limits. *active* queue is not maintained physically in the disk unlike other queues. It is maintained as a data structure in the address space of *qmgr*. The reliability issue comes to the forefront in this case. However, the mail delivery process is implemented as a state machine and the data is safely resident in physical drive for the states. So, the system can restart gracefully from a crash.

When all the deliverable recipients for a message are delivered, and for some recipients delivery failed temporarily for a transient reason, the message is placed in the *deferred* queue. *qmgr* scans the queue periodically. The scan interval is controlled by the *queue_run_delay* parameter. The queue manager alternates between bringing a new message from the *incoming* and the *deferred* queue for delivery and therefore there is no starvation.

Recent versions of Postfix have the *flush* message queue to improve performance during SMTP ETRN delivery. ETRN is an extension of SMTP that enables aa MTA to request a second MTA to forward it outstanding mail messages. The ETRN operation is useful for intermittently connected mail servers.

4.2.3 Postfix Tables

Postfix does not have a policy specification language. Instead it uses several lookup tables that are created by the email administrator. Each table defines parameters that control the delivery of mail within the Postfix system.

- ***access.*** The *smtpd* process uses the *access* table. The table maps remote SMTP hosts to an accept/deny table for spam filtering.
- ***aliases.*** The mapping of alternative recipients to local mailboxes is stored in *aliases* table.

- ***canonical.*** The *canonical* table maps alternative mailbox names to real mailboxes for message headers.
- ***relocated.*** The *relocated* table maps an old mailbox name to a new name.
- ***transport.*** The *transport* table maps domain names to delivery methods for remote connectivity and delivery.
- ***virtual.*** The task of the *virtual* table is to map and manage virtual domains.

The mail administrator creates each lookup table as a plain ASCII text file. Once the text file is created, a binary database file is created using the *postmap* command. Postfix uses the binary database file when searching for lookup tables for better performance.

4.2.4 Interfaces of Postfix Processes

Postfix processes communicate between themselves using Internet sockets (inet), Unix sockets (unix) and Unix named pipes (fifo). In a typical configuration, *smtpd* uses internet sockets (inet) as communication option. *pickup* and *qmgr* uses fifo. Other processes, namely *cleanup*, *trivial-rewrite*, *smtp*, *local* etc., use the Unix sockets option.

Each transport method has its own underlying mechanism for initiating and terminating connections. The Postfix software handles all of the low-level details required for those communications. The availability of channels to outside processes is specified by a special field ('private') in the configuration file. Postfix system uses two subdirectories, `public` and `private`, to contain the named pipes needed by each service. The `private` subdirectory contains the pipes for processes marked as private, while the `public` sub-directory contains the pipes for processes marked as public.

For privacy reasons Postfix uses Named pipes or Unix sockets that live in a protected directory. Postfix processes do not trust the communication payload and keeps them limited in size. In many cases, the information passed between processes is just a file name or some status information. This is an example of the Trust Partitioning pattern.

4.2.5 Compartmentalization and Distributed Delegation

The partitioning of Postfix processes are examples of Compartmentalization and Distributed Delegation Pattern. The Postfix processes are similar to corresponding *qmail* processes in terms of

functionality. The exception is the *trivial-rewrite/resolve* process that acts as a library for various tasks like spam filtering and address rewriting. In *qmail*, spam filtering and address rewriting are done by the *qmail-send* process. Creating a separate library simplifies the structure of the mail sending process.

Effective partitioning comes from categorizing the resources and running the processes with least privilege users. *qmail* has a number of users that serve different purposes. Configuration management of *qmail* is difficult because of the number of user and group ids in the *qmail* architecture. Postfix simplifies this scenario by having only one user.

Postfix has only one user for all these processes. During installation, one has to create a user and a group named 'postfix'. The 'postfix' user owns all the queue directories. The user has limited privileges - it does not even need a home directory or a login shell.

The default installation of Postfix creates a *maildrop* queue that is world-writable and *sendmail* program can directly write new messages in the queue. This poses a security problem. As an alternative, *sendmail* program uses the *postdrop* program to write into the queue. A special user group ('maildrop') is created that owns the *maildrop* queue. *postdrop* runs as that user group and writes messages to the queue. The *postdrop* program is a setgid-helper that helps un-privileged *sendmail* program to write into the *maildrop* queue.

sendmail uses Unix *setuid* to grant its program root privileges when they run. Postfix does not use *setuid*. It uses *setgid* in *postdrop* but it *setgid*'s to a lesser privilege level. That is why it does not affect the overall security.

4.2.6 *chroot* Security

The fact that Postfix uses minimal number of user ids and groups in its design intuitively suggests that it has to adopt something else to be secure. This follows the Defense in Depth principle. Because all the processes are running under same user id, compromise in one process means that the attacker can attack other processes and the resources that they work on. To limit this, the processes run inside *chroot* jail.

chroot jail provides an added level of security by limiting the exploits of an attacker in a specific directory. This saves the important system files. A *chroot*() [12] call with a pathname as

parameter sets up the *chroot* jail. After the call, the pathname becomes the root directory ('/') for the process. Thus files outside the specified directory structure are considered 'safe' from the *chroot*'d program.

Table 4.1: Security Pattern - *chroot* Jail

Problem Compartmentalization is a high level pattern that suggests breaking up the task into smaller processes. It does not eliminate the problem of compromise in one process affecting other processes because processes communicate. Distributing responsibility among processes reduces the vulnerability. However, processes having shared resources (files etc.) are still not secure from attack. How can we design a system that is secure in a manner that compromise in one process does not affect another? **Solution** Run the processes under separate least privilege user ids. Also, the programs/processes should be run in a controlled environment with limited access to system files. This will limit the exploits of an attacker. In UNIX, this is achieved by running the processes in a *chroot* jail.

All of the Postfix programs (except *local* and *pipe*) can run using the *chroot* environment. Postfix *chroot* script sets `/var/spool/postfix` as the root directory by default. This would require a modification of the `/var/spool/postfix` directory in a manner that it contains copies of specific system files and libraries and has a specific directory structure to pass as a fake root. Postfix processes do not run in a *chroot* environment by default. The master configuration file (master.cf) of Postfix has to be modified to include Postfix programs that would run in the *chroot* environment. Programs that communicate with remote hosts, such as *smtpd* and *smtp* are the most susceptible to attacks by malicious attackers. So they are almost always run in a *chroot* jail.

A number of things have to be considered prior to the setup of *chroot* jail. A process with root privileges can break the *chroot* jail [16]. Therefore, the process must run with a less-privileged user id. Also, writing privileges are removed from the *chroot* directory, because in that case an attacker can dump malicious files in the *chroot* directory that can be accessed by processes running outside the *chroot*'d environment.

Postfix architecture is designed to run under *chroot* jail. It is broken up into many small programs that are specialized into specific tasks. This way, setting up the *chroot* environment

is easy. This setup involves only toggling the postfix daemon's *chroot* options in the main configuration (main.cf) file. Some binary-package distributions (like in SuSe LINUX) toggle the appropriate daemons to *chroot* automatically during postfix installation.

4.2.7 Pre-forking

Postfix is up to three times as fast as its nearest competitor. Postfix uses web server tricks to reduce process creation overhead and uses other tricks to reduce file system overhead, without compromising reliability. Pre-forking is used in web servers like Apache for better performance.

In *qmail*, processes are forked on demand and their lifetime is limited for the duration of servicing the request. Postfix tries to improve this by avoiding the process creation overhead. The *master* daemon is resident and it runs other daemons on demand. It spawns a number of processes beforehand and handles them the task when a request is made. The pre-forking mechanism is widely used in Apache server for performance improvement. However, Apache has sophisticated modules for resource pooling and load balancing unlike Postfix. Postfix only pre-forks up a pool with a size that has been specified through command line. The pre-forked processes serve a fixed number of requests also specified through command line arguments. After the process dies, the parent process forks off another in replacement.

Table 4.2: Security Pattern - Secure Pre-forking

Problem The consequences of security compromise are worse in case of daemon processes because they have a long lifetime. How can the vulnerability associated with daemon processes be minimized? **Solution** Limit the lifetime of daemon processes and fork them again after a configurable, short lifetime. Limit the number of requests handled by daemons. Run the daemons in a contained environment to minimize the exploits.

There are a lot of trade offs with pre-forking architecture. The most critical issue is the vulnerability it has associated with it. In *qmail*, even though some malicious user manages to control a process, that process only has a limited lifetime. In case of pre-forked processes, the malicious user has more time before the process dies. That is another reason why *chroot* jail is used to limit the

exploits during a security breach.

Another issue with this architecture is the complexity. This means more bugs, less portability, and a bigger binary. The performance improvement of pre-forking only becomes evident in case of heavy load. In case of light load, the pre-forked processes occupy memory and become a bottleneck instead.

4.2.8 Reliable Mail Queuing and Mailbox Management

Postfix uses the same patterns from *qmail* for reliable mail queuing. The Postfix process writing in the central mail queue is *cleanup*. Instead of implementing the mail queue as one single directory with subdirectories, Postfix has several queues to handle mail storage task. This is only semantically different because the underlying file system implementation of the queue is the same.

Like *qmail* , the queue directory is split into sub-directories for improved search performance. Each of the queue directories is split into two levels of subdirectories. Messages are placed into a sub-directory based on the first two characters of its filename.

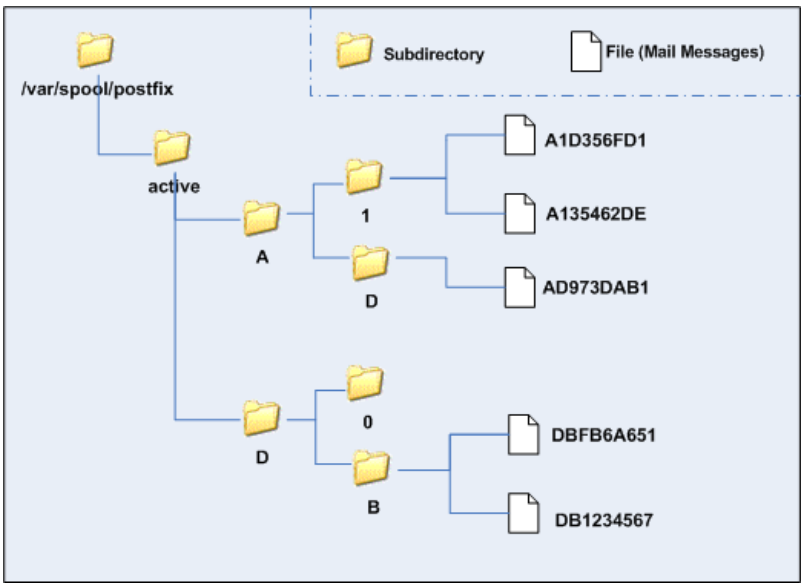

Figure 4.2: Postfix queue in underlying file system

As new messages are received in the message queues, corresponding sub-directories are created. As files are retrieved from the directories, other messages use the sub-directories.

For mailbox management, Postfix uses the 'Maildir' format of *qmail*. However, it also has

support for 'mbox' format. 'mbox' is unreliable but Postfix uses it to retain compatibility with *sendmail*. The rationale here is compatibility not reliability of message storage.

4.2.9 Multi-threading

Postfix processes use multi-threading wherever possible to improve performance. However, there is one exception that is made to achieve security. The *maildrop* queue is processed by single-threaded *pickup* service. Because, this process lies in the outer boundary of the program and communicates with outer processes it is safer if it is created single-threaded. Thus performance is given lower priority in comparison to security.

Table 4.3: Security Pattern - Single Threaded Facade

Problem Even though multi-threading improves performance, it requires careful resource management and synchronization. The processes communicating with outside environment are more vulnerable. Therefore the internal design of these processes should be simple. How can this simplicity be achieved? **Solution** The processes in the perimeter of the system should be such that they perform a single task. Again, they should be single-threaded because multi-threading involves complex resource management.

4.2.10 File System Update

The performance of an MTA is limited by the file system since both *qmail* and Postfix transfers messages from one directory to another during its delivery phase. Poor file system performance would result in long end-to-end message delivery time.

Postfix improves performance by using *softupdates*. *softupdates* is an implementation technique that uses delayed writes for meta-data updates. With *softupdates* the cost of retaining integrity is low and performance asymptotically approaches that of a memory-based file system. Also, additional update sequencing methodology can be added with little loss of performance. This improves security and integrity.

softupdates increases disk activity speed and decreases disk I/O through 'trickle sync' facility

that is incorporated into the kernel for more efficient disk synching operations. *softupdates* will not cause file system corruption, but, if they begin causing difficulties, then can be turned off easily with the *tunefs* command.

qmail does not use *softupdates*. The issue against *async* or *softupdates* file systems is that if the system crashes at the wrong moment, the system will lose mail. Under Linux, all mail-handling file systems are mounted sync for *qmail*.

In reality, *softupdates* generally survives hardware failure without corruption, although in a few cases it loses files that were saved right before the failure. This corresponds to losing emails. However, even a sync mount can become corrupt in the event of hardware failure, although it is much less likely.

4.2.11 End-to-end Message Delivery Time

People have complained about *qmail*'s way of sending emails in that it creates a connection for each email rather than bunching them up like *sendmail*. *qmail* processes the message one at a time. Thus, if message A is targeted to enough people to flood the outbound connections, message A uses the connections as they become available until it gets finished. Message B has to wait until it is completed. *qmail* does do parallel delivery but if a single message has more recipients than the number of available connections it "hogs" all of them until it no longer requires that many. Postfix does not do this.

Postfix sends the new messages as they arrive (or connections become available). Thus, a message to a very large number of people has only a minor impact on new messages.

Postfix, in fact, can do either, depending on the configuration. By default, Postfix does the same as *sendmail* - multiple emails to different recipients at the same domain will be sent in one SMTP session. Actually, Postfix performs much better than *sendmail* in this instance because for any given message with multiple recipients, Postfix will open multiple connections to different servers in parallel, whereas, for the same message, *sendmail* will open only one connection to each server in turn. Postfix and *sendmail* follows the Batch Transaction pattern.

Table 4.4: Performance Pattern - Batch Transaction

Problem
Process creation and context switching overhead affects the performance of a system. If a system has several small jobs, then this overhead becomes significant. How can we improve performance of a system that handles similar tasks?

Solution
Batch transactions to eliminate overhead. Group related tasks and perform them at a time to avoid task switching and process creation overhead.

4.2.12 Resource Management

Different strategies are adopted by MTAs for resource management. Disk space is managed by partitioning and limiting the disk space with quotas. File size can be limited by OS directives (like *rlimit*). Similar ideas apply for program memory segments like data segment and stack segment.

Some programs will spawn an arbitrary number of children, using up all memory even though per-process memory is limited. For example, *inetd* can start any number of children to handle TCP connections; its $fork-per-second$ limit does not provide effective protection against a flood of long-term TCP connections. One can control the number of processes per uid with the *maxproc rlimit*, but this is useless for root daemons. *qmail* provides a solution by replacing *inetd* with *tcpserver*, which provides a concurrency limit.

Bernstein and Venema take different approaches to resource limitation. Bernstein prefers to use general resource limitation tools like *softlimit*. Venema prefers to build resource limitation into each program that uses resources. Therefore, Bernstein doesn't consider this a bug, but Venema does. The fact is that an LWQ-style installation using *softlimit* on the *qmail-smtpd* processes is not vulnerable to these attacks.

4.2.13 Spam Handling Policy

Spam handling is not directly built into *qmail* architecture. Patches of *qmail* have been written to add this to the base architecture. Postfix, however, has policies built in a number of places to handle spam.

Postfix uses a number of maps and filters for this purpose. The filter files are located in

Table 4.5: Reliability Pattern - DoS Safety

Problem Denial of Service attack is tackled by adopting several network-based strategies. However, Defense in Depth principle suggests that the system architecture should be resilient to such attacks as well. How can we design a system that provides internal DoS safety? **Solution** Protect against Denial of Service attacks by setting resource limit. This can be done by using per-process resource management or by adopting operating systems resource management features.

`/${installation-folder}/maps` directory. These files are written as regular expression compatible. They can also be PCRE (Perl Compatible Regular Expression).

Spam checking is done in several phases. Header and body checks are useful to identify and discard spam. The Subject header is the most popular to reject for based on words or phrases. The X-Mailer header can be used to identify some software or mail clients that are used almost exclusively for spam. Body checks are done to scan content for phrase patterns and discard email based on that information. Also this is useful for virus filtering.

A number of internal and external lists of clients/hosts/senders are used to scan and filter based on sender addresses and domains. Mails coming from someone in these lists would be discarded. Postfix has a number of access lists that it keeps internally. These lists are used to keep the *access* table up-to-date.

Another way to fight spam is to verify users and domains by using 'verify' lists. In this case, Postfix checks on the MX host responsible for the domain portion of the sender's mail address. This check will verify that the address is valid and is capable of receiving email. However, this connects out to another mail server, every time Postfix receives an email, so this is done sparingly.

Postfix also uses services from lists available via DNS like RBL (Real-time Black-hole List) or RHSBL (Right Hand Side Black-hole List). These list the addresses of mail servers known (or believed) to send spam.

The *smtpd* process filters spam using RBL and RHSBL lists when the mail is accepted and rejects mail if it is a spam. The header and body checks are done by *cleanup* and *qmgr* process

with the help of *trivial-rewrite/resolve* daemon. All the incoming traffic goes through *cleanup* and it works on the messages to ensure proper formatting and spam handling. This is an example of Policy Enforcement Point [6] pattern.

Table 4.6: Security Pattern - Policy Enforcement Point

Problem Malicious attackers attack a system through processes of the system that communicates with outsiders. If the number of processes communicating with outside environment grows large, then it is very difficult to maintain security because the attack can come through various points of access. How can security be achieved in this scenario? **Solution** Channel all outside communication through one point of the system. Use authentication and other security mechanism at that point by defining security policies.

In *qmail*, no matter where the messages come from, they all pass from the queue. A message can be received either by a local process or by a remote process through the SMTP protocol. Locally generated messages are handled by *qmail-inject* while remotely generated are handled by *qmail-smtpd*.

The spam check can be added in different points, depending on the target messages. In order to scan incoming messages, the solution is to wrap the *qmail-smtpd* process in order to get it to check the messages as soon the SMTP transaction is finished (like Postfix). This is a better option but it has some consequences. Because all the messages are checked before insertion, forwarded messages that are re-queued by *qmail-local* will be checked by the spam checker each time they will be put the queue, wasting a lot of CPU time. Similar checking will happen in case of bounce messages. These are handled by using complex wrappers.

Adding this functionality to *qmail-smtpd* would involve moving/copying the functionalities from *qmail-local* to *qmail-smtpd*. This would necessitate *qmail-smtpd* to have access to home directories as *qmail-local* has. This is insecure. Again, in many cases, *qmail-smtpd* would be unable to determine conclusively whether an address is valid or not. (For example, consider the case where ~`alias/.qmail-default` exists or a virtual domain has a `.qmail-default`.)

Validating recipients during the SMTP session makes it trivial for spammers to determine which

addresses are deliverable. A spammer can quickly run through a dictionary of possible recipients and the MTA will obligingly validate them. The standard fix to this problem, tarpitting, adds unnecessary complexity to the SMTP daemon.

4.3 Conclusion

Although the design of *qmail* and Postfix are not the same, they use common security patterns. Their designs are closer to each other than to the design of *sendmail* because security was a more important part of their original requirements than it was of *sendmail*'s. Some of the differences between *qmail* and Postfix are because performance is a more important requirement for Postfix. But both of them are evidence that security does not have to come at the cost of performance. A good design can provide security, reliability, performance, and understandability. Although sometimes these software qualities conflict, often they support each other, and a good design can simultaneously achieve them all.