SOFTWARE—PRACTICE AND EXPERIENCE

Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Evolution of MTA Architecture: The Impact of Security

Munawar Hafiz*,†, Ralph E. Johnson*,‡

Department of Computer Science, University of Illinois, 201 N Goodwin Avenue, Urbana, IL 61801, USA

SUMMARY

As the Internet matured, security became more important and formerly adequate designs became inadequate. One of the victims of the increased need for security was `sendmail`. This paper shows how its competitors improved on its design in response to the increased need for security. The designers of `qmail` and `Postfix` used well-known patterns to achieve better security without hurting performance; and these patterns can be used by designers of systems with an increased need for security.

KEY WORDS: Mail Transfer Agent (MTA), Security Patterns, Software Architecture.

*Correspondence to: University of Illinois, 201 N Goodwin Avenue, Urbana, IL 61801, USA

†E-mail: mhafiz@uiuc.edu

‡E-mail: johnson@cs.uiuc.edu

Received 13 April 1999

Revised 23 September 2002

1. Introduction

A Mail Transfer Agent (MTA) is a computer program that transfers electronic mail messages from one computer to another. The first standard MTA was `sendmail` [30]. `sendmail` made Simple Mail Transfer Protocol (SMTP) popular and is still the most widely used MTA. The early `sendmail` architecture was ideal for its time, because the key architectural requirement was flexibility with protocols and the architecture supported a lot of diverse protocols. The Internet introduced security and reliability as important requirements for an MTA. `sendmail` architecture failed to cope with these new requirements. `sendmail` developers have released several patches in response to the reported bugs, but people are still finding security vulnerabilities. These security vulnerabilities were addressed by Daniel Bernstein in his design of `qmail` [28]. `qmail` was designed as a replacement for `sendmail` with improved security. `qmail` had not revealed any security defects since its initial release in 1997. `Postfix` [27], another competitor of `sendmail`, was designed in 1999 by following many of the design decisions of `qmail`. It also retains a clean record of security. Both `qmail` and `Postfix` influenced the architecture of other MTAs that followed. In fact, the new generation of `sendmail`, `sendmail X`, is not an evolution of previous versions of `sendmail` (`sendmail` version 8). Instead, `sendmail X` follows the architecture of `Postfix` very closely.

The security of `qmail` and `Postfix` is the result of a set of architectural decisions. The same architecture that makes them secure also makes them more efficient than `sendmail` and easier to understand and maintain. This paper describes the design of `qmail` and `Postfix` as a sequence of decisions [26]. Many of the design decisions are examples of security patterns. These patterns are not new, but `qmail` and `Postfix` are examples of how to use them effectively. One

of the key principles of the `qmail` architecture is Defense in Depth [36], which means that a system adopts multiple layers of security tactics instead of a single security strategy. First, the way the system is divided into modules tends to decrease the damage caused by security break-ins, and ensures that many kinds of errors are not possible. The module decomposition also makes each module simpler, so it can be inspected for correctness. It makes multiprocessing more efficient. Second, the way that `qmail` uses the file system makes queuing and delivering the mail more reliable. Third, the low-level coding patterns in `qmail` eliminate important classes of errors such as buffer overflows. The result is an MTA architecture that is much superior to that of `sendmail`. The architecture of `Postfix` follows many of the same security patterns. However, `Postfix` adds some patterns that result in better performance than `qmail` and `sendmail` [2] [3] [33]. Thus the `Postfix` architecture illustrates how to consider performance along with security and reliability. Studying and comparing the architecture can help us make other systems secure, efficient, and easy to understand.

This paper narrates the story of the evolution of MTA architecture by describing four important MTAs. `sendmail` exemplifies how to make a monolithic architecture flexible. `qmail` shows how to modularize software architecture; modularity begets simplicity, security and reliability. By the time `Postfix` was written, performance became a key requirement because the MTAs had to be operational under heavy loads of traffic. The fourth MTA, `sendmail X` is chosen for two reasons. It is an example of the substantial influence of `qmail` and `Postfix` over the subsequent MTA architecture. But, it also is the final step of the architectural feedback cycle [24]; it completes the evolution of MTA architecture from "monolithic, insecure, unreliable and slow" to "modular, secure, reliable and fast". This paper also shows how different security

and reliability patterns are used to create a reference architecture for MTAs. Thus, this paper is another example of how patterns generate architecture [9].

The paper starts by outlining the requirements of a simple MTA architecture. Then it introduces the architecture of `sendmail` and discusses the problems. The architecture of `qmail` and `Postfix` are then described as a set of design decisions. Many of these design decisions come from security and reliability patterns and these patterns are mentioned. Finally, the paper demonstrates the impact of `qmail` and `Postfix` on the evolution of MTA architecture using `sendmail X` as an example.

2. Mail Transfer Agent

A *Mail Transfer Agent* [38] (*a.k.a.* mail server, mail router or Internet mailer) is a computer program that transfers electronic mail messages from one computer to another. MTAs are transparent to the users. Normally, the users interact with the *Mail User Agent* (MUA) [39] program. This program uses the MTA to deliver email. The most popular mail transfer protocol for MTAs is the Simple Mail Transfer Protocol (SMTP), although it has been extended with Extended SMTP (ESMTP). The most popular MTA is `sendmail`. Other Unix based ESMTP compatible MTAs are `qmail`, `Postfix`, `exim`, `Courier` and `ZMailer`. Another related system is a *Mail Delivery Agent* (MDA) [37]. An MDA accepts incoming email messages and distributes them to recipients' individual mailboxes. Heavy duty MTAs may delegate the local mail delivery task to the MDAs. On Unix systems, */bin/mail* is the most popular MDA.

Security is an important architectural requirement for an MTA. An MTA must keep email secure against casual attacks, and must not allow attackers to break into the system. The

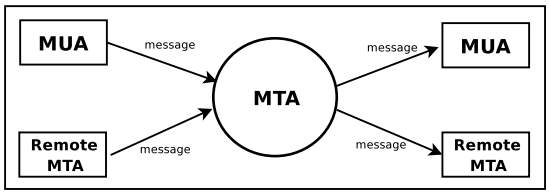

Figure 1. A Simple MTA

system must fail securely and should be recoverable to its original state, and it must also be able to track the cause of a security breach. An MTA must be reliable. It should never lose an email. An MTA must be efficient in both time and space, because it might process thousands of messages simultaneously. An MTA must have high availability. It must be able to cope with Denial of Service (DoS) attacks. Finally, an MTA must support existing mail transfer protocols, but it should be extensible for future protocols.

An MTA should accept email from a local user and deliver it to a local mailbox or to a remote MTA. It should also accept email from another MTA. Thus, an MTA has at least two ways to accept input and two ways to deliver output, as shown in figure 1.

The structure of an MTA follows its context closely. There is a process that handles incoming emails from an MUA, a process that handles incoming emails from another MTA, a process that sends email to a local user, and a process that sends email to a remote user. However, figure 1 does not indicate the need for a mail queue, which turns out to have a big impact on the design. Figure 2 shows an operational MTA with processes that handle input/output and a process for queue management.

Post offices do not deliver mail as soon as it is dropped in the mailbox. They collect outgoing mail and process it periodically. Likewise, an MTA cannot always deliver mail immediately because the destination MTA might be disconnected from the Internet, or might not be working. Often an MTA has a large number of messages to send at once, and it might take a long time to deliver them. It stores outgoing messages in a mail queue until they are delivered. An MTA without a mail queue loses messages if its host fails before the messages are delivered.

It is possible to design an MTA to have a single process for the mail queue. However, `qmail` has three processes, *qmail-queue*, *qmail-clean* and *qmail-send*. `Postfix` has the *cleanup* process responsible for placing email in the queue, and the *qmgr* process responsible for taking it out of the queue and sending it.

3. `sendmail` Architecture

`sendmail` runs as one big process with root privilege. This means that all the components of a generic MTA, shown in figure 2, run in the same address space. The `sendmail` process is configured through a complex configuration file.

`sendmail` dows not need root privilege to perform all its tasks. In fact, `sendmail` only runs with root privileges when it needs to satisfy legitimate needs of the user, e.g. accessing the configuration files for email forwarding (*.forward* files). When delivering mails to a local mailbox, it does not run as a root. In this case, `sendmail` spawns a child process lowering its privilege level to that of the local user. A program with lower privilege cannot go back to a higher privilege level. Therefore, `sendmail` must retain the highest privilege level and lower it whenever necessary.

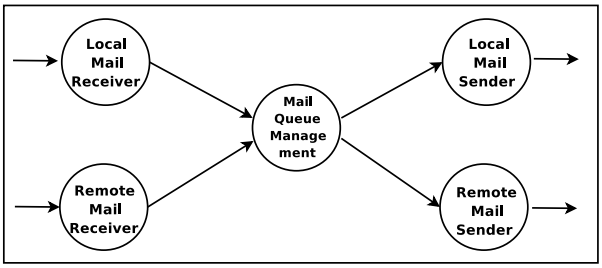

Figure 2. Main Components of an MTA

Running `sendmail` as an untrusted pseudo-user (such as 'nobody') does not work. It causes the local user's email forwarding scripts to be run as 'nobody', and it requires the queue to be owned by 'nobody'. Such a scheme would allow any user to break into the queue and modify its contents.

The monolithic architecture means that if one malicious user gets control of the `sendmail` process, he can use the root privilege to get control of the whole system. Without proper configuration, `sendmail` is a security hole that can be used to compromise the operating system. Configuring `sendmail` is complex because of its flexibility. The simplest mail transfer scenario in `sendmail` (between one local user to another) is almost trivial. The complexity arises as `sendmail` adds support for multiple protocols and diverse services. The specification language of `sendmail` is very complex itself. The security holes found in `sendmail` are primarily of two categories.

- Security holes that exploit the root privilege.
- Security holes that come from misconfiguration.

`sendmail` vulnerabilities have been listed in Daniel Bernstein's website [6] and in the Bugtraq database [29]. The most recent vulnerability of `sendmail` (Buqtraq ID 19714) was reported in August, 2006 [11]. Table I lists different `sendmail` releases and vulnerabilities reported for these versions. Some of the vulnerabilities affect many versions of `sendmail`, e.g. `sendmail` vulnerability no. 19714 [11] affects all the `sendmail` releases except version 8.13.8. This is included in the vulnerability count for all the `sendmail` versions.

The architecture of `sendmail` was considered a strength when it was created because it met the flexibility requirements. However, the long list of `sendmail` vulnerabilities and the series of patches to solve them are slowly putting `sendmail` out of favor. Before 1990s, `sendmail` controlled almost 100% of the MTA market. In 1996, `sendmail` had about 80% share while in 1999, it had about 60% [31] share.

The following sections describe the architecture of `qmail` and `Postfix`, which were designed to meet the new requirements for security.

4. `qmail` Architecture

`qmail` does not have a monolithic architecture like `sendmail`. Instead, it is divided into processes performing mail acceptance, storage and delivery. Figure 3 shows the main `qmail` processes. The partitioning of processes is done based on functionality outlined in figure 2. The local mail handler *qmail-inject* receives local mail messages, adds appropriate information to the message header, and invokes *qmail-queue* to transfer them to the mail queue. The remote mail receiver *qmail-smtpd* receives remote email messages over SMTP from other MTAs and sends them to *qmail-queue*. *qmail-send* gets the messages from the mail queue, formats

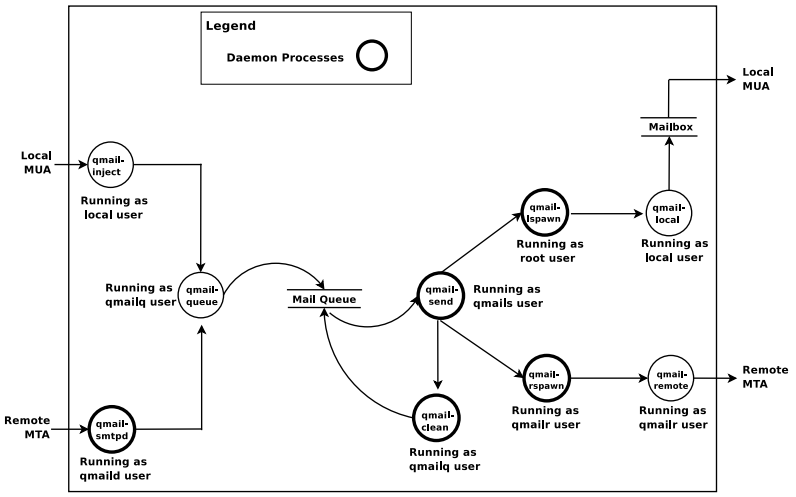

Figure 3. Major `qmail` Processes

addresses and hands it over to either *qmail-local* or *qmail-remote* for local or remote delivery. These processes are spawned by *qmail-lspawn* or *qmail-rspawn*. If a message is temporarily undeliverable to one or more addresses, *qmail-send* leaves it in the queue and tries to deliver it later. *qmail-send* also communicates with *qmail-clean*, which removes delivered messages from the queue.

4.1. Defense In Depth in `qmail`

Most secure systems apply Defense in Depth: they adopt a variety of security solutions at different layers of system architecture. The architecture of `qmail` is a classic example. At the highest level, the monolithic process is partitioned into several processes. This is formulated

by finding the requirements of the system, distributing the responsibilities, and aligning the partitions along this distribution.

MTAs have to be secure from vulnerabilities that are associated with data management. Buffer overflow is a vulnerability for languages without bounds checking (like C). `qmail` implements safe data structures by associating length and memory allocation information with each data buffer. Another classic MTA vulnerability that plagued `sendmail` spawns from treating mail messages as parameters passed to a program or data written to a file. `qmail` avoids this by treating data content as mail messages only. Finally, the partitioned processes do not trust the data in the shared channel. They always validate their inputs. This is done to prevent the propagation of a security compromise. This also prevents an attacker from launching a Denial of Service attack on internal processes by sending garbage data.

One of the key security principles is to keep the architecture simple [36]. Simplicity contributes to flexibility and better performance. `qmail` spawns multiple processes for fast mail delivery. These processes do not perform any other task than mail delivery and are very simple. This also means that the processes do not need a lot of resource and do not exhaust the system resource limit. Moreover, the simple architecture of `qmail` contributes to its extensibility.

Another crucial aspect of MTA architecture is reliable mail storage management. A related requirement is graceful recovery from failure without loosing any mail. `sendmail` queue and user mailbox are implemented as a single file. This creates problems when multiple processes try to write to the file. `qmail`'s queue and mailbox are implemented as directories; mail messages are written as separate files. The mail delivery process is implemented as a state machine.

Each of the states are persisted in the file system. This ensures the reliable management of mail messages.

4.1.1. Compartmentalization

The `qmail` approach is to split the monolithic process into many small processes. The SMTP server runs in a separate address space from the rest of the MTA. Even if an attacker discovers a buffer overrun in the SMTP server, he can do little damage to `qmail`, because *qmail-smtpd* only calls *qmail-queue* and gives it a message to put on the mail queue. It does not write on any files except a log file. This contains the error in *qmail-smtpd*. Moreover, *qmail-smtpd* is small and it is easy to read it and verify that it doesn't write to any file except the log file.

Separating the SMTP server (*qmail-smtpd*) from the rest of the MTA is an obvious way to improve security. The separation in security domains protects the compartments even if the security of one part is compromised. `qmail` takes the idea further because it divides the functions of `sendmail` in twenty two programs. This separation was also done in a redesign of `sendmail` by Zhong and Edwards [42] and in `Postfix`. This is an example of the *Compartmentalization* [36] pattern.

4.1.2. Distributed Responsibility

`sendmail` is a prime target for intrusion because it runs as a super-user. If `sendmail` is not properly installed and configured, external probes over networks can gain information that can later be used to break the system. Once that happens, improper file permission can be used to trick `sendmail` into giving away root privilege. Michael Zalewski reported one such

vulnerability in `sendmail` 8.12.0 in 2001 [12]. `sendmail` 8.12.0, in its default installation does not use a *setuid* root binary to process the mail queue. The binary is *setgid* 'smmsp', where 'smmsp' is a special group with read/write permissions on the mail queue. `sendmail` allows users to specify custom configuration files or configuration parameters. When it is processing untrusted information, `sendmail` is supposed to drop all extra privileges and continue to run at user level. This mechanism worked fine in `sendmail` versions prior to 8.12.0. Because of a programming error, `sendmail` 8.12.0 code fails to drop extra group privileges completely in new *setgid* conditions, leaving the saved gid value untouched. By calling the *setregid* function, an attacker can regain dropped privileges. This is possible due to several bugs in the configuration file parser. The problem was solved in `sendmail` 8.12.1.

Most of the `qmail` processes do not run with super-user privilege. The process that delivers local mail (*qmail-local*) runs as the user receiving the mail. Most of the other processes run as `qmail` specific user IDs; they cannot write on either user files or system files. The *qmail-smtpd* process runs with user ID (UID) 'qmaild' while *qmail-queue* runs with UID 'qmailq' and *qmail-send* runs with UID 'qmails'. This means that it is impossible for the SMTP server (*qmail-smtpd*) to write on the mail queue or on user files, even if attackers can completely change its program. The worst they can do is to have it generate bad messages to *qmail-queue*. Similarly, *qmail-send* is not able to add or remove messages from the mail queue. It can only read them and mark them as sent. No other part of `qmail` can even read them; much less modify them. `qmail` ensures that local mail delivery is secure by breaking it into *qmail-lspawn* and *qmail-local*. *qmail-lspawn* runs as the super-user, but it is short (less than 500 lines) and simple. First it looks up the target user to find the UID, then drops its super-user privilege to

that of local user and then it runs *qmail-local*. It does not write any files, nor does it read any files once it decides on its new UID.

`qmail` uses five different user groups:

- *qmail-smtpd* runs as 'qmaild',
- *qmail-rspawn* and *qmail-remote* run as 'qmailr',
- *qmail-send* runs as 'qmails',
- *qmail-queue* and *qmail-clean* run as 'qmailq',
- *qmail-start* and *qmail-lspawn* are the only programs that run as 'root'.

Although *qmail-start* and *qmail-lspawn* run as root, they perform minimal tasks. *qmail-start* runs as root because it spawns other processes during startup. *qmail-lspawn* is root because it creates a *qmail-local* process and then has to set its user ID to that of the local user. *qmail-queue* is the only *setuid* program. This is because *qmail-queue* has to run as 'qmailq' to insert messages in the queue but it is spawned by another process. The limited privilege of the 'qmailq' user means it is not as dangerous as a program that runs as *setuid* root.

A security failure in a compartment can change any data in that compartment. A compartment has both an interface that is at risk of a security failure and data that needs to be secure. In order to protect the data, the *Distributed Responsibility* [35] pattern partitions responsibility across compartments such that compartments that are likely to fail do not have data that needs to be secure. Responsibilities are assigned in such a way that several of the compartments need to fail in order for the system as a whole to fail.

Someone might object that we are reading too much into the fact that `qmail` is implemented as a set of processes. Perhaps this wasn't done to make `qmail` more secure, but was done for some other reason. After all, Unix programmers often implement a system as a set of processes communicating by pipes. They often do this so that the processes are reusable, or to build their system out of pre-existing processes. Each process will take a standard input and a standard output, each of which is a pipe along which data flows in the form of ASCII text. But few of the interfaces in `qmail` are like this. This implies that the decomposition of `qmail` is not done for reuse, but for security.

For example, *qmail-lspawn*, *qmail-rspawn* and *qmail-clean* take one input pipe and one output pipe, but both the pipes go to *qmail-send*. *qmail-send* uses an input pipe to send commands to a component and the component's output pipe to read the result. *qmail-send* thus has two pipes each to communicate with *qmail-lspawn*, *qmail-rspawn* and *qmail-clean*. The pipes are used like a remote procedure call mechanism, except that results are not returned in the same order that commands are given. Every time *qmail-send* issues a command, *lspawn* or *rspawn* will spawn a new process, and each process returns a result when it is finished. Since some processes are faster than others, results return in a different order than their commands.

A less strange example is given by *qmail-queue*. It takes two input pipes. File descriptor 0 (normally standard input) is the message, while file descriptor 1 (normally standard output) is the envelope for the message. The envelope describes the addresses of the sender and the recipients. *qmail-queue* reads the message and the envelope and uses them to create an entry in the mail queue. It communicates back to the program that invoked it only by its exit code.

The only core `qmail` process with a normal Unix interface is *qmail-inject*, which reads a message on its standard input. It is also the only process that would be called by a Unix program that is not part of `qmail`, so it is good that its interface is natural for Unix application programmers.

qmail-inject runs with the UID of the process that invokes it. It invokes *qmail-queue*, but *qmail-queue* must be able to write to the mail queue. So, *qmail-queue* changes its uid to be 'qmailq', which is the owner of the mail queue. It does this because its suid bit is set in the file system. It can be dangerous to allow a program to change its uid, but it is safe in this case because the 'qmailq' user is not powerful, and is able only to add and remove messages from the mail queue.

Because most Unix processes read from standard input and write to standard output, they can be used as components in shell scripts. But the purpose of using separate processes in `qmail` is first compartmentalization and second multi-threading. Reuse is not a purpose at all, as is proven by the idiosyncratic interfaces.

4.1.3. Reliable Mail Queuing

The mail queue is the center of an MTA architecture. The reliability of `qmail` depends on mail being reliably inserted and removed from the mail queue. A mail is placed in the mail queue only by *qmail-queue*. However, *qmail-queue* is not a Unix process, but rather a Unix program that can be run simultaneously by many processes. For example, it can be run by *qmail-inject* and *qmail-smtpd* at the same time. Thus, it is important that several messages can be placed

in the mail queue simultaneously. Mail delivery is sometimes interrupted and often takes a long time. Remote MTAs can die in the process of receiving mail.

If all the incoming messages are written to a single file, *qmail-inject* and *qmail-smtpd* processes are in a potential race condition to get a lock on the file. This can be avoided if for all incoming messages, the processes create a separate file in the mail queue directory. Each filename is unique, so this is the same as assigning a unique ID to each message. The mechanism of determining this unique ID has to be efficient and portable. Ideally, `qmail` could use an ID created automatically by Unix, but none of the standard Unix IDs are perfect. Each process has a unique process ID, but a process ID is unique only as long as the process is running. Messages can last long enough for process IDs to be recycled. Once a file is created, its i-node number could be used as a unique ID, but this only works after it is created. So, `qmail` first uses the ID of the process that creates the message as the message ID, and then changes the message ID to be the i-node number of the file. This solution is portable and efficient.

This ensures non-interference of processes by maintaining that the target of every write request is a different location. Thus different processes are never writing on the same file at the same time, enabling processes to add information to the queue concurrently. This is an example of *Unique Location for each Write Request* [21] pattern. The pattern ensures that the system fails securely [36]. Because of the unique resource location, there is no trace left of the failure.

The key to ensuring that messages are not lost is to keep track of the various states of a message in the mail queue. The state of a message is determined by the existence (or non-

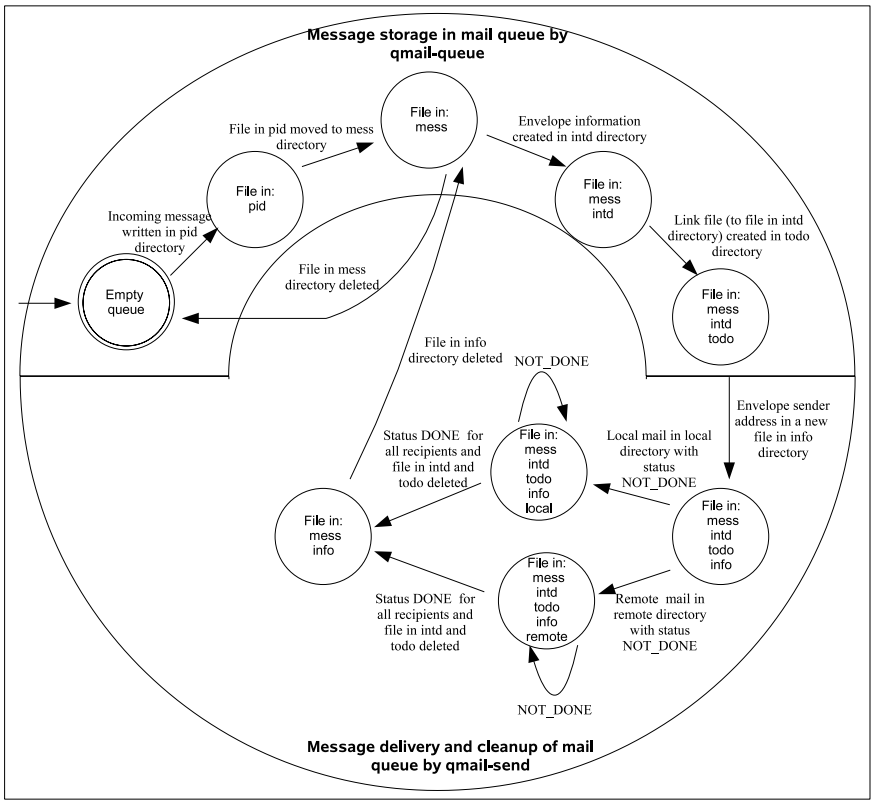

Figure 4. Mail Storage and Delivery as a State Machine

existence) of a file in a queue directory. *qmail-queue* creates a message by first creating a file in the *pid* directory, then moving it to the *mess* directory and creating the envelope in the *intd* directory. When it is finished, a link to the envelope is placed in the *todo* directory. At this point, *qmail-queue* has successfully stored the message in the mail queue.

qmail-send takes messages out of the mail queue and sends them to local and remote destinations. *qmail-send* creates two files - one in the *info* directory and the other in either *local*

or *remote* directory. *qmail-send* spawns *qmail-local* or *qmail-remote* process to send messages in the *local* or *remote* directory respectively. At this point the entries in the *intd* and *todo* directory are deleted. Once the message is sent, the files in the *info* directory and *mess* directory are deleted to complete the delivery. The different states of mail delivery process are shown in figure 4.

Cleanups are not necessary if the computer crashes while *qmail-send* is delivering a message. At worst a message may be delivered twice. There is no way for a distributed mail system to eliminate the possibility of duplication. What if an SMTP connection is broken just before the server acknowledges successful receipt of the message? The client must assume the worst and send the message again. Similarly, if the computer crashes just before *qmail-send* marks a message as delivered, the new *qmail-send* must assume the worst and send the message again. This redundancy is not harmful, but it is crucial in ensuring the reliability of the whole system.

4.1.4. Mailbox Management

Mailboxes have similar reliability problems as mail queues. The most popular mailbox format used by `sendmail` and other MTAs is 'mailbox' or 'mbox'. This format uses a single file for all mails received by a user. The problem with 'mailbox' format is that if the system crashes while some process is writing a message into the mailbox file, the file is truncated. The next message is appended with the truncated message.

Another problem with the 'mailbox' format arises when two or more programs simultaneously try to deliver mail to the same user. The 'mailbox' format requires the programs to update a single central file. If the programs do not use some locking mechanism, the central

file will be corrupted. Sun's Network File System (NFS), which is typically used in this case, does not work reliably. There are some reliable Unix locking mechanisms, but they are not widely available.

Problems related to the 'mailbox' format are solved by `qmail`'s 'maildir' format for mailboxes. This format is used by other MTAs including `Postfix`, `exim`, `Courier` and `Mac OS X Mail Application` with the `RCI` mail server.

In the 'maildir' format, email messages are stored as separate files under a directory structure in the same file system. This follows the Unique Location for each Write Request pattern (section 4.1.3). A directory in 'maildir' format has three subdirectories named *tmp*, *new* and *cur*.

Each file in *new* is a newly delivered mail message. Files are moved to the *cur* directory after they have been seen by the user's mail-reading program. The *tmp* directory is used to ensure reliable delivery.

The reliable mail management in both mail queue and mailbox is ensured by treating the delivery process as a finite state machine. All the states are defined in such a manner that even if some crash occurs, the system can restart gracefully without losing any messages. The mail messages are persistently stored in the disk between different states. This would act as a checkpoint that is used for graceful recovery. This is an example of the *Checkpointed System* [10] pattern.

4.1.5. Multi-threading

Delivering mail using SMTP can take a long time. It takes only a fraction of a second to send a short message to a lightly loaded MTA. However, it can take a long time to send a long message to an MTA on a slow network, and the MTA may become unavailable after part of the message is sent. Therefore, an MTA will be multi-threaded so it can send many messages at once and not allow unavailable MTAs to block delivery of mail to available ones.

`qmail` has a small amount of multi-threading. *qmail-smtpd* and *qmail-send* run as singleton, daemon processes. *qmail-lspawn* and *qmail-rspawn* also run as daemon processes. However, most of the concurrency in `qmail` comes from the fact that *qmail-rspawn* will repeatedly run *qmail-remote*. There can be hundreds or thousands of copies of *qmail-remote* running, most of them waiting for a response from a remote MTA. It is important that these processes not take much memory, because the number of processes is limited by the memory they take. If the processes grow unbounded, there is a potential DoS scenario. The *Small Processes* [23] pattern makes processes safe from resource exhaustion by making the processes small and having them perform a single task. *qmail-remote* is small, so it takes little memory, and it is easy to run many copies. This is one of the reasons why `qmail` has high performance on small machines.

4.1.6. Flexibility

Although SMTP is the most common mail transfer protocol, it is not the only one. `qmail` supports two other mail transfer protocols, Quick Mail Transfer Protocol (QMTP)[§] and Quick Mail Queuing Protocol (QMQP)[¶], that are faster than SMTP. For each protocol, there will be a server that runs along with *qmail-smtpd* to deliver mail to *qmail-queue*. The *qmail-qmtpd*

process delivers mail following the QMTP protocol and the *qmail-qmqpd* process uses the QMQP protocol for mail delivery. Thus, it is easy to configure `qmail` to support other mail transfer protocols. None of the `qmail` input programs must be changed. Instead, a new server is written and only the configuration files must be changed.

However, a new protocol requires changing *qmail-remote*, because it has to decide which protocol to use. The `qmail` design does not have separate processes for remote mail delivery using different protocols. This is because the choice of protocols would have to be included in the *qmail-rspawn* process. *qmail-rspawn* runs as a daemon process and it is important for the sake of security to make this process as simple as possible. The complexity of selecting appropriate protocol is moved to the ephemeral *qmail-remote* process.

4.1.7. Coding Standards

There are a variety of coding standards followed in `qmail` that reduce the likelihood of security problems. It does not use the standard Unix I/O libraries, but provides its own I/O library that eliminates buffer overrun and makes string operations mor ereliable.

The string library in C uses null termination to identify the end of a string. As such, a string function like *strcpy* blindly copies all characters starting at the address of the source string into the destination until it finds a null. This opens up to potential buffer-overflow attacks like "Smashing the stack" or "Overrun screw" [1]. A way to avoid this problem is to dynamically allocate strings. However, that approach is vulnerable to DoS attacks.

[§]QMTP protocol - http://cr.yp.to/proto/qmtp.txt
[¶]QMQP protocol - http://cr.yp.to/proto/qmqp.html

The rewritten string library in `qmail` eliminates buffer overruns. `qmail` strings are not null-terminated. They are encapsulated in a data structure (*stralloc*) along with information about the length. The structure has three fields.

```
typedef struct stralloc{
                char *s;
                unsigned int len;
                unsigned int a;
                }
```

s is a pointer to the string or 0 if it is not allocated. *len* is the number of bytes in the string, if it is allocated. *a* is the count of allocated bytes for the string. An unallocated *stralloc* variable is initialized to 0.

The string copy routines are re-written as *stralloc_copy*, *stralloc_cat*, *stralloc_append* etc. They use the underlying routines *stralloc_ready* and *stralloc_readyplus* for dynamic memory allocation.

stralloc_ready(stralloc sa, int len)* makes sure *sa* has sufficient memory allocated for *len* characters. *stralloc_readyplus (stralloc* sa , int len)* makes sure that *sa* has enough space allocated for *len* characters more than its current length. This defensive mechanism makes the string copy function safe.

In a language with garbage collection and bounds checking like Java and Smalltalk, string copy does not create a vulnerability. Strings carrying information about length and memory are needed in C because the compiler does not check array bounds. The inclusion of length and allocated memory in a data buffer is an instance of the *Safe Data Structure* [23] pattern.

4.1.8. Content Dependent Processing

Some security breaches are caused by maliciously used features. In November 1988, Morris worm, the first of its kind, was released in the Internet to launch a DoS attack. One of the loopholes that the worm abused was the debug function of the `sendmail` program [15]. `sendmail` has a feature to send mails to a program, such that the program receiving the mail executes with the body of the mail message as input parameters. This feature is not generally allowed for incoming connections except when the debugging mode is on. Unfortunately, the `sendmail` program packaged with 4.3BSD and pre-4.1 versions of SunOS had this mode turned on by default. The worm used this feature to connect to a `sendmail` daemon and send a message to a shell program of the recipient. The body of the message is passed as a parameter to the shell. The body consisted of a bootstrap shell script that, when executed, would connect back to the attacking machine via TCP and download a pre-compiled object code replication of the worm. It then tried to link the target code and execute it. At this point, the machine became infected. The worm then sent the same mail message from the infected machine to other `sendmail` servers.

`sendmail` treats programs and files as addresses. A message sent to a program means that the program is executed with the content of the mail massage as its parameter; whereas a message sent to a file means that the content of the message is appended to the file. The situation is aggravated because `sendmail` runs as root. `sendmail` tries to prevent these attacks by defining policies in order to keep track of whether some local user was responsible for an address. But that sometimes does not work out right because of the complexity of the configuration language.

In `qmail`, programs and files are not addresses. `qmail` treats the the incoming contents as mail messages only and does not perform any processing on them. Even when it treats programs and files as addresses, the process runs as a less-privileged user to minimize the impact. *qmail-local* can run programs or write to files as directed by the *.qmail* configuration file in the local directory of the user; but the local user is always less privileged than a root user. This follows the *Content Dependent Processing* [21] pattern.

4.1.9. Trust Partitioning

`qmail` partitions trust in many levels of its architecture. The *Content Dependent Processing* pattern illustrates that `qmail` refuses to trust the message content, therefore refusing to send mails to programs and files. This is important for security.

Even tighter security is achieved by making the components in such a manner that they do not trust the communication payload sent between themselves. This ensures that a compromise of one component of the system remains limited in that section only.

At the process level, `qmail` processes run under different `qmail`-specific UIDs. These programs do not trust the input data from other programs and always validate before using the input. The data sent between programs do not constitute special data structures but only names of message files.

In a compartmentalized system architecture, an intruder may get hold of a compartment and crash the whole system by sending malicious payload. The solution is to design the components not to trust inputs from other sources and validate inputs. This follows the *Trust Partitioning* [23] pattern.

5. `Postfix` Architecture

The `Postfix` architecture is in many cases similar to the architecture of `qmail`. Comparison between the `Postfix` architecture in figure 5 and the `qmail` architecture in figure 3 illustrates that the partitioning and responsibility distribution among processes are inspired by `qmail`. One difference is that `Postfix` uses lookup tables extensively for configuration management. Another difference is the mail queue. `Postfix` uses in-memory queues for better performance, but most of the queues are implemented as subdirectories in the directory hierarchy. `Postfix` considers these subdirectories as separate queues; but `qmail` does not make this distinction.

`Postfix` is implemented as a resident master server that runs `Postfix` daemon processes on demand. The core program is *master*, that runs in the background. It spawns processes on demand to scan and process the queue.

The default installation of `Postfix` has 3 daemon processes - *master*, *qmgr* and *pickup*. The *pickup* program determines when messages are available for routing by scanning the *maildrop* queue. The *qmgr* program is responsible for the central message routing system.

The *smtpd* process receives email messages from remote hosts via SMTP (like *qmail-smtpd*). The *cleanup* process writes in the mail queue (like *qmail-queue*). However, unlike *qmail-queue*, *cleanup* processes the incoming mail headers, formats them with the help of *trivial-rewrite*, and places them in the *incoming* queue. The *trivial-rewrite* process handles three types of client service requests, *rewrite*, *resolve* and *verify*. It is split into two processes, namely *rewrite* and *resolve*. The *rewrite* process is used by *cleanup* process while the *resolve* process is used by *qmgr*. Both the processes perform almost the same tasks (address rewriting, message filtering etc). However, in `Postfix`, these tasks are done in multiple phases in both *cleanup* and *qmgr*.

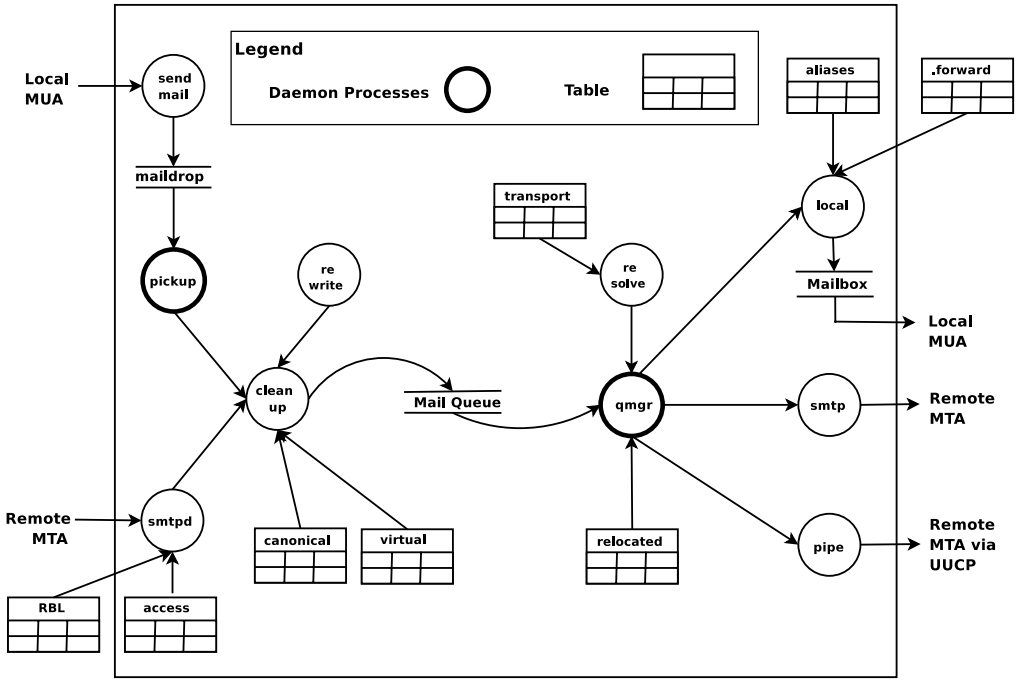

Figure 5. `Postfix` Architecture

The partitioning of *trivial-rewrite* process is done to keep the separation between *cleanup* and *qmgr* clean and secure.

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.

The queue in `qmail` is implemented by several subdirectories. `Postfix` has a separate queue for each subdirectory. Hence the queues of `qmail` and `Postfix` are not conceptually

different, they are only expressed differently. The *qmgr* daemon (similar to *qmail-send*) awaits the arrival of incoming mail and arranges for its delivery using `Postfix` delivery processes. Like *qmail-send*, it rewrites addresses. However, address-rewriting is delegated 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`, which has *qmail-local* handling local delivery and *qmail-remote* handling all types of remote delivery using different protocols. This complication makes incorporating a new protocol more difficult in `qmail`. `Postfix` architecture is more scalable than that of `qmail` in terms of protocol extensibility.

One of `sendmail`'s main problems is its complex configuration language. `Postfix` and `qmail` do not have a configuration language. `qmail`'s configuration management is file based. `Postfix` 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. The *smtpd* process uses the *access* table. This table maps remote SMTP hosts to an accept/deny decision for spam filtering. The *cleanup* process uses the *canonical* and *virtual* table. The *canonical* table maps alternative mailbox names to canonical names. The *virtual* table is used to rewrite recipient addresses for all local, virtual and remote destinations. The *relocated* table maps old mailbox names to new names. The *transport* table maps domain names to delivery methods for remote connectivity and delivery. This is used by the *resolve* part of the *trivial-rewrite* process. The *aliases* table is used by the *local* process to map alternative recipient names (aliases) to local mailboxes.

`Postfix` also uses external tables like *RBL* and *.forward* file of `sendmail`. *RBL* (Real-Time Black-Hole List) is a list of malicious IP addresses maintained in the Internet. `Postfix` consumes this list and the *smtpd* process matches the sender MTA's address with this list. If the email comes from a suspicious domain, then the mail is rejected as a spam. `Postfix` maintains `sendmail` compatibility as one of its architectural goals and hence it supports *.forward*-file oriented `sendmail` configuration.

`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). *pickup* and *qmgr* use fifo. Other processes, namely *cleanup*, *trivial-rewrite*, *bounce*, *flush*, *smtp*, *showq*, *error*, *local* etc, use Unix sockets.

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` 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. Following the Defense in Depth principle, `Postfix` processes do not trust the communication payload and keep them limited in size. In many cases, the information passed between processes is just a file name or some status information.

5.1. Defense in Depth in `Postfix`

Postfix follows many of the same design decisions as `qmail`. The main difference between qmail and Postfix is that the `Postfix` architecture also considers performance as one of its design goals.

At the process level, `Postfix` uses the Compartmentalization pattern like qmail; but unlike `qmail`, `Postfix` processes run under only one user. This creates a security vulnerability because if an attacker manages to compromise a process, he can also corrupt other processes running at the same privilege level. `Postfix` mitigates this by running the processes in a *chroot* jail.

`Postfix` improves performance by pre-forking and resource pooling. The processes in the resource pool are not implemented as daemons; an attacker, after compromising one daemon process, would have had infinite time to attack other processes in that scenario. Instead, `Postfix` associates a lifetime to the processes, and renews the processes in the resource pool. Thus Postfix prioritizea security over performance.

Another similar design decision is taken in thread management. Multi-threading makes the processes complex, often leading to a security hole. `Postfix` implements multi-threading whenever possible except for processes that communicate with external processes. This is done to have better protection.

`Postfix` implements resource management routines at every process. `qmail` and `Postfix` authors share different views on resource exhaustion attacks [34, 8]. The `qmail` author relies on operating system's resource management mechanisms. But the `Postfix` author considers this a vulnerability and implements specific checks.

`Postfix` stores mails like `qmail`. However, instead of implementing the mail queue as one single directory with subdirectories like `qmail`, `Postfix` has several queues to handle mail storage task. Some of the queues are maintained in memory for better performance.

Some performance hacks in `Postfix` may create a reliability problem [5]. `Postfix` uses *softupdates* [25] for file system updates that makes the file-based queue management as fast as memory-based queue management. But this can create a problem if the file system crashes at the wrong time. However, even traditional file system based queues have this vulnerability and the risk is too minute to be practical.

A new functional requirement for `Postfix` is spam handling. Spam affects performance, availability and security. `Postfix` implements spam handling techniques at the system entry points. This is done by specification and enforcement of policies.

Some of the design decisions in `Postfix` are taken purely to improve performance. For example, the `qmail` remote mail delivery process does not reuse the open connections with remote MTAs. `Postfix` improves performance by sending all the mails to a recipient in a batch. Performance is not the main requirement in `qmail`, although the simplicity of `qmail` architecture and flexibility to adopt specific protocols (QMTP and QMQP) makes it fast. On the contrary, most of the design decisions of `Postfix` are influenced by performance or a performance-security tradeoff. These show that performance and security can both be achieved in an architecture.

5.1.1. Mail Storage Management

Mail storage is of two types. Pre-processed mail is stored in the mail queue. Mail intended for local recipients is stored in the mailbox.

`Postfix` delivers mail by following a sequence of steps. In each of these steps, mail is stored in the mail queue. The `Postfix` mail queues and the `qmail` queues are both implemented as subdirectories in the file system. The only difference is that `Postfix` treats each of the subdirectories as a separate queue and uses the name of the subdirectory to identify the queue; whereas `qmail` treats all the subdirectories as part of one queue.

Some of the mail queues are memory resident, but `Postfix` keeps a backup copy of these queues on the disk and periodically updates them. The disk-based queue structure provides persistent storage for mail messages. The state of mail delivery for each mail can be identified by the subdirectory it is currently in. The storage of data and state ensures that the system can restart gracefully from a crash. This follows the Checkpointed System [10] pattern.

The *maildrop* queue stores messages that have been submitted by the *sendmail* process. The *pickup* process drains the queue. The formatted messages end up in the *incoming* queue. Another important queue is the *active* queue. Messages in the *active* queue are ready to be sent, but are not necessarily in the process of being sent. The *active* queue is implemented as a data structure in the address space of *qmgr*. A backup copy in the disk is periodically synchronized with the memory based queue. Memory based queue management is faster but unreliable because `Postfix` might lose messages if the system fails between synchronization.

When `Postfix` fails to deliver a message to some recipients for a transient reason (it might succeed later), the message is placed in the *deferred* queue. *qmgr* scans the *deferred* queue

periodically. It fetches messages alternatively from the *deferred* and the *incoming* queue to the *active* queue. This round-robin strategy prevents starvation of either the *incoming* or the *deferred* queue.

For mailbox management, `Postfix` uses the 'maildir' format of `qmail`. However, it also has support for the 'mbox' format. 'mbox' is unreliable, but `Postfix` uses it to retain compatibility with `sendmail`. The rationale here is compatibility, not reliability of message storage.

5.1.2. chroot Jail

Configuration management of `qmail` is difficult because of the number of user and group IDs in the `qmail` architecture. `Postfix` simplifies this by using only one user. The 'postfix' user owns all the queue directories (*active*, *bounce*, *corrupt*, *defer*, *deferred*, *flush* and *incoming*). This 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. *sendmail* process can directly add new messages to the queue. This poses a security problem because the *sendmail* process communicates with outside processes. As an alternative, *sendmail* uses the *postdrop* process to write to 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. It is a *setgid*-helper that helps unprivileged *sendmail* process to write to the *maildrop* queue.

`sendmail` uses Unix *setuid* to grant its program root privileges. `Postfix` does not use *setuid*. It uses *setgid* in *postdrop* but it changes the group ID to a lesser privilege level. That is why it does not affect the overall security.

Compartmentalization [36] suggests breaking up the task into smaller processes. This does not prevent attackers from compromising one process and then using it as a base to compromise others. Processes with shared resources (files etc) are more vulnerable. Running the processes under separate users and groups reduces this vulnerability, but `Postfix` uses minimal number of users and groups in its design. Instead, `Postfix` runs the programs/processes in a controlled environment with limited access to system files. This limits the exploits of an attacker. This security pattern is called *chroot jail* [21].

chroot jail limits the exploits of an attacker in a specific directory. This saves the important system files. A *chroot*() [13] 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 jailed process.

All of the `Postfix` processes (except *local* and *pipe*) can run using the *chroot* environment. `Postfix` processes do not run in a *chroot* environment by default. The master configuration file (*master.cf*) has to be modified to include `Postfix` processes that would run in the *chroot* environment. `Postfix` *chroot* script sets '`/var/spool/postfix`' as the default root directory. This requires a modification of the '`/var/spool/postfix`' directory so that it contains copies of required system files and libraries and has a specific directory structure to pass as a fake root. Programs that communicate with remote hosts, such as *smtpd* and *smtp* are the most susceptible to attacks. 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 [17]. Therefore, the jailed process must run with a less-privileged user ID. Also, writing privileges are removed from the *chroot* directory, because

in that way an attacker can dump malicious files in the *chroot* directory that can be accessed by processes running outside the jail.

`Postfix` 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.

5.1.3. Secure Pre-forking

`Postfix` uses pre-forking to improve performance. In `qmail`, processes are forked on demand and their lifetime is limited for the duration of serving the request. `Postfix` tries to improve on this by avoiding the process creation overhead. The *master* daemon runs other daemons on demand. It spawns a number of processes in a resource pool and hands over the task to one of these processes when a request is made. The pre-forking mechanism is widely used in Apache server for performance improvement [18].

There are a lot of trade offs with pre-forking architecture. The most critical issue is the vulnerability associated with pre-forking. In `qmail`, a compromised process has a limited lifetime. With pre-forked processes that run as daemons, a malicious user has more time because the process never dies. *chroot* jail partially solves the problem by limiting the exploits during a security breach.

A more secure solution would be to avoid daemon processes and set up a life limit for all pre-forked processes. A monitor process creates, monitors and kills child processes in the

resource pool. Some of the process parameters that are monitored are process lifetime, idle time in the resource pool, the number of requests served etc. After a pre-defined time period (or a pre-defined number of requests served), the monitor process kills the child processes and replenishes the resource pool with newly forked processes. The size of the `Postfix` process pool is specified through command line. The life limit metric of processes is also specified through command line arguments. This is an example of the *Secure Pre-forking* [20] pattern.

In `qmail`, most of the processes are not daemon processes. Even if some process is compromised, after some time it dies and is garbage collected. The attacker cannot use this process to try to compromise other processes. Even if some attacker does not try to infect other processes, he can send garbage payload to other processes and launch an internal Denial of Service attack. This is the primary benefit of Secure Pre-forking. It adds to the security of the overall system by limiting the lifetime of the daemon processes so that the effect of a process compromise remains transient.

Secure pre-forking improves performance because processes do not have to be spawned for each incoming request. If the pre-forked processes run as daemons, the system would have shown even better performance because the cost of process creation is entirely eliminated. Instead, this pattern balances the trade-off between security and performance. The limited lifetime of processes adds some overhead for forking; but this contributes to the entire system being secure.

The major bottleneck 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.

5.1.4. Multi-threading

Multi-threading improves performance, but it requires careful resource management and synchronization. If this complexity is introduced to the processes that communicate with the outside environment, then the system becomes more vulnerable. The processes in the system perimeter should be made simple in architecture, and therefore should be single-threaded. This security pattern is called *Single Threaded Façade* [21]

`Postfix` processes use multi-threading wherever possible to improve performance with one exception. The *pickup* process manages the *maildrop* queue. Because *pickup* lies in the outer boundary of the program and communicates with outer processes, it is designed to be single-threaded. Thus performance is given lower priority in comparison to security.

5.1.5. File System Update

The performance of an MTA is limited by the file system since both `qmail` and `Postfix` delivers messages by transferring them from one directory to another. Poor file system performance would result in long end-to-end message delivery time.

`Postfix` improves performance by using *softupdates*, which is an implementation technique that uses delayed writes for metadata updates [25]. *softupdates* increases disk activity speed and decreases disk I/O through 'trickle sync' facility that is incorporated into the kernel for

more efficient disk synchronization operations. With *softupdates*, performance asymptotically approaches that of a memory-based file system.

`qmail` does not use *softupdates*. The issue against *async* or *softupdates* filesystems is that if the system crashes at the wrong moment, it will lose mail [5]. 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 updated right before the failure. This corresponds to losing emails. In case of such a failure, the feature can be turned off easily with the $tunefs$ command. On the other hand, even a *sync* mount can become corrupt in the event of hardware failure, although it is much less likely.

5.1.6. End-to-End Message Delivery Time

Process creation and context switching overhead affects the performance of a system. If a system has several small jobs, then this overhead becomes significant. The performance of a system that handles similar tasks can be improved by using the *Batch Transaction* [21] pattern. A batched system groups related tasks and performs them at a time to avoid task switching and process creation overhead.

`qmail` creates a connection for each email rather than sending them as a batch. Messages are processed one at a time. Hence, if message A is targeted to enough people to flood the outbound connections, message A uses all the available connections. Message B has to wait until message A has been delivered to all its recipients. `qmail` does have parallel delivery feature, but if a

single message has more recipients than the number of available connections, it uses up all of the available connections. `Postfix` avoids this problem by using batch processing.

`Postfix` delivers mail in the same way as `sendmail` - multiple emails to different recipients under 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.

5.1.7. Resource Management

Denial of Service attack on system resources is tackled by adopting several network-based strategies. However, the system should not only depend on a network-based strategies, but also on resource management strategies in the software architecture. Different techniques have been adopted by MTAs for resource management. Disk space is managed by partitioning and limiting the space with quotas. File size can be limited by OS directives (like *rlimit*). Similar ideas apply for program memory segments, e.g. data segment and stack segment.

Some programs would 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.

The authors of `qmail` and `Postfix` take different approaches to resource limitation [34, 8]. `qmail` uses general resource limitation tools like *softlimit*. But `Postfix` includes resource limit checks into each program. Therefore, Bernstein doesn't consider this a bug, but Venema does [7]. The fact is that an LWQ-style `qmail` installation (installation procedure described in the 'Life with *qmail*' website [32]) using *softlimit* on the *qmail-smtpd* processes is not vulnerable to these attacks.

Explicit setting of resource limit follows the *DoS Safety* [21] pattern. This can be implemented by using per-process resource management or by adopting operating system's resource management features.

5.1.8. 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. This is because when `qmail` was written spam was not a major issue. `Postfix` has policies built in a number of places to handle spam. Spam handling became an important architectural requirement at the time `Postfix` was designed and coded.

`Postfix` uses a number of maps and filters for spam handling. The filter files are in `${installation-folder}/maps` directory. These files are written to be compatible with regular expressions, e.g. Perl Compatible Regular Expression (PCRE).

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 target for filtering 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 record of the sender domain. This check will verify that the address is valid and is capable of receiving email. However, this requires connection to another mail server every time `Postfix` receives an email; so this is done sparingly.

`Postfix` also uses services from lists available via Domain Name System (DNS) like Real-time Black-hole List (RBL) or Right Hand Side Black-hole List (RHSBL). 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 checks the messages to ensure proper formatting and spam handling.

Malicious attackers try to exploit the perimeter processes of a system. If the number of processes communicating with the outside environment grows large, the attack can come through various points of access. The solution is to channel all outside communication through one point of the system and enforce authentication and other security checks at that point by defining security policies. This security pattern is called *Policy Enforcement Point* [10] (*a.k.a. CheckPoint* [41]).

In `qmail`, all the incoming messages pass through the queue. Locally generated messages are handled by *qmail-inject* while remotely generated messages are handled by *qmail-smtpd*. The spam check can be added at different points in `qmail`, depending on the target messages. Incoming messages can be scanned by the *qmail-smtpd* process as soon as 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 added to the queue, wasting a lot of CPU time. Similar checking will happen in case of bounce messages. These can be handled by using complex wrappers. However, adding the functionality to *qmail-smtpd* would involve moving/copying the functionalities from *qmail-local* to *qmail-smtpd*. This would require *qmail-smtpd* to have access to home directories like *qmail-local*. This is insecure. Again in many cases (aliasing and virtual domains), *qmail-smtpd* would be unable to determine conclusively whether an address is valid or not.

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 [40], adds unnecessary complexity to the SMTP daemon.

6. Evolution of MTA architecture: `sendmail X`

In the architectural life cycle, experience and best practices impact future architecture [24]. The security architecture of `qmail` has significant influence on the evolution of MTA architecture in response to changing requirements [22]. Eric Allman, the author of `sendmail`, planned a new

version of `sendmail` with a modified architecture and fresh architectural perspectives to satisfy emerging architectural requirements like security and reliability [14]. This new generation of `sendmail` is called `sendmail X`. `sendmail X` is a completely new design and is not an evolution of previous versions of `sendmail` (`sendmail` version 8).

`sendmail X` architecture is completely different from its predecessors. It does not have a single root process. Instead, it follows the architecture of `Postfix` very closely.

`sendmail X` processes are compartmentalized according to their functionality. A central queue manager controls SMTP servers and SMTP clients to receive and send e-mails, an address resolver provides lookups in various maps including DNS for mail routing, and a main control program starts the others processes and watches over their execution. The queue manager organizes the flow of messages through the system and provides measures to avoid overloading of local or remote systems.

There are direct parallels of `Postfix` processes in `sendmail X`. The `Postfix` process *master* becomes MCP, *smtpd* becomes SMTPS, *local* becomes LDA, *smtp* becomes SMTPC, *trivial-rewrite* becomes AR and *qmgr* becomes QMGR. Figure 6 shows the major `sendmail X` processes.

The supervisor process that starts the `sendmail X` processes is called MCP (Master Control Process). It is a daemon process with root privilege.

`sendmail X` has two types of queues. The message content is stored in the Content Database (CDB) queue. The envelope content is kept in the Envelope Database (EDB) queue. There are different EDB queues in `sendmail X`. All of the queues are implemented in the file system; but, like `Postfix`, some of the queues are maintained in memory for better performance.

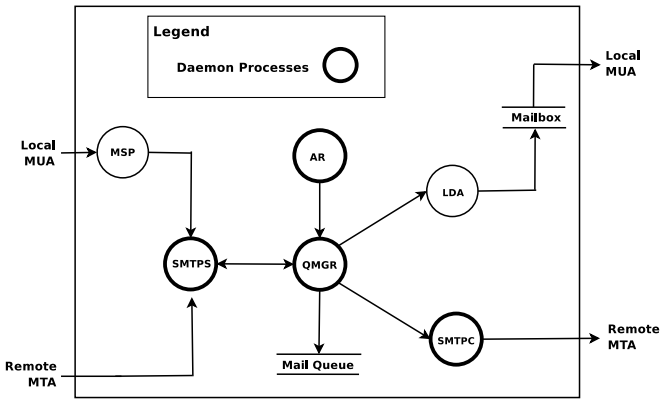

Figure 6. `sendmail X` Architecture

Incoming messages arrive via SMTP. The connection attempts are received by the SMTP Server (SMTPS). SMTPS communicates with the Queue Manager (QMGR). These two collaboratively decide whether to accept or reject the message. Local mail sending is done by the command line message submission tool called the Message Submission Program (MSP). For each transaction, a new envelope is created and the commands are communicated to the QMGR and the Address Resolver (AR) for validation. When the email is received, it is either stored to disk or an appropriate delivery agent is informed to deliver it immediately. The acknowledgement to the remote sender is sent after successful termination of either of these two actions. QMGR schedules the delivery. Local Delivery is handled by the Local Delivery Agents (LDA) and remote delivery is handled by the SMTP Clients (SMTPC). Delivery agents receive sender and recipient information from the QMGR. QMGR gets the delivery status from

the delivery agents and updates the EDB accordingly. When a mail is delivered to all recipients, the corresponding data is removed from the EDB and the content is removed from the CDB.

6.1. Defense in Depth in `sendmail X`

The primary design decision of `sendmail X` is to revert from the old `sendmail` design of a single root process and compartmentalize the modules. The modules run with a lower privilege level. No program in `sendmail X` runs as *setuid* root. `sendmail X` processes run under separate user IDs. The SMTPS process runs as an 'smxs' user, the QMGR process runs as an 'smxq' user, the SMTPC process runs as an 'smxc' user, and the AR process runs as an 'smxm' user. There is also an 'smx' user with limited privilege that acts as 'nobody'. This partitioning is done based on the resource usage of the processes following the Distributed Responsibility pattern.

`sendmail X` architecture implements the reliability patterns that are in `qmail` and `Postfix` architecture. The mail queue of `sendmail X` is split into directories like the mail queue of `qmail` and `Postfix`. SMTPS writes into the CDB queues. There are multiple SMTP processes running at one time. The mails are entered as separate files in the mail queue. This follows the Unique Location for Each Write Requests pattern. The mail delivery process is implemented as a state machine and information of all the states are stored in the disk. This is an example of the Checkpointed System pattern.

`sendmail X` uses new string routines that follow the Safe Data Structure pattern. The string buffers are defined by a structure called *sm_str_S*. The buffer stores both data and length information in the data structure. `sendmail X` also follows the Trust Partitioning pattern. The

processes are partitioned into five mutually untrusting user groups. The processes validate the input from processes in a different user group.

Like `Postfix`, performance is an important quality requirement in `sendmail X` architecture. `sendmail X` follows the Batch Transaction pattern by reusing open connections and sending mails to a recipient in a batch. The delivery agents and QMGR collaboratively perform this task. *sendmail X* also keeps in-memory queues for better performance.

`sendmail X` balances the tradeoff between performance and security like `Postfix`. The QMGR process is like a pipeline between sending and receiving modules. It must not slow the system down. QMGR is a multi-threaded program to allow scalability and fast response. But the processes that communicate with external processes are all single-threaded. This follows the Single Threaded Façade pattern.

Handling spam is an important security requirement of current MTAs. The anti-spam checking in `sendmail X` is handled by the SMTP servers. It has access to all the necessary data for spam checking like client connection and authentication information, sender and recipient addresses etc (provided by QMGR and AR). This acts as a Policy Enforcement Point.

`qmail` and `Postfix` architecture showed how security and performance can be integrated into the architecture of MTAs. `sendmail X` architecture can be attributed as a lesson learned from the mistakes of previous architecture and from the best practices of the peers in the community.

7. Conclusion

Although the architecture 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. Some of the differences between `qmail` and `Postfix` are because `Postfix` is also designed to have better performance. 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.

The security architecture of `qmail` has significant influence on the evolution of MTA architecture in response to changing requirements. The architectural quality requirements are not always orthogonal and the final design decision is a tradeoff between the requirements. In the architectural life cycle, experience and best practices impact future architecture. This is clearly evident in the `sendmail X`'s adoption of best practices in `qmail` and `Postfix`.

REFERENCES

1. Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996.
2. M. Andree. MTA benchmark. http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/bench2.html.
3. M. Andree. `Postfix` vs. `qmail` - Performance. http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/vsqmail.html.
4. C. Assmann. `sendmail X`: Requirements, architecture, functional specification, implementation, and performance. Technical report, *Sendmail Inc.*, September 2004.
5. D. J. Bernstein. Reliability. http://cr.yp.to/qmail/faq/reliability.html#filesystems, 1997.

6. D. J. Bernstein. `sendmail` disasters. http://cr.yp.to/maildisasters/sendmail.html, 1997.
7. D. J. Bernstein. The `qmail` security guarantee. http://cr.yp.to/qmail/guarantee.html, 2005.
8. D. J. Bernstein. Time for a reality check. `qmail` mailing list. http://www.ornl.gov/lists/mailing-lists/qmail/1997/06/msg00500.html, Jun. 13, 1997.
9. K. Beck and R. Johnson. Patterns generate architectures. *Lecture Notes in Computer Science*, 821:139–??, 1994.
10. B. Blakley and C. Heath. Security design patterns technical guide - version 1. *Open Group (OG)*, led by Bob Blakley and Craig Heath. 2004. http://www.opengroup.org/security/gsp.htm.
11. BugTraq ID 19714 `sendmail` long header Denial of Service vulnerability. http://www.securityfocus.com/bid/19714, 2006.
12. BugTraq ID 3377 `sendmail` inadequate privilege lowering vulnerability. http://www.securityfocus.com/bid/3377, 2001.
13. chroot(). *Unix man pages*.
14. Bryan Costales and Eric Allman. *Sendmail*. O'Reilly & Associates, Inc., 2003.
15. M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *IEEE Security and Privacy*, pages 326–343, 1989.
16. E. B. Fernandez. Patterns for operating systems access control. In Proceedings of the 9th Conference on Patterns Language of Programming (PLoP'02).
17. S. Friedl. Go directly to jail: Secure untrusted applications with chroot. *Linux Magazine*, December 2002. http://www.linux-mag.com/2002-12/chroot_01.html.
18. B. Gröne, A. Knöpfel, R. Kugel, and O. Schmidt The Apache modeling project. http://www.f-m-c.org/projects/apache/html/Apache_Modeling_Project.html, 2004.
19. M. Hafiz. Unique atomic chunks: A pattern for security and reliability. In Proceedings of the 11th Conference on Patterns Language of Programming (PLoP'04).
20. M. Hafiz. Secure pre-forking: A pattern for security and performance. In Proceedings of the 12th Conference on Patterns Language of Programming (PLoP'05).
21. M. Hafiz. Security architecture of mail transfer agents. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

22. M. Hafiz. Security patterns and evolution of MTA architecture. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 142–143, New York, NY, USA, 2005. ACM Press.
23. M. Hafiz, R. Johnson, and R. Afandi. The security architecture of `qmail`. In Proceedings of the 11th Conference on Patterns Language of Programming (PLoP'04).
24. R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation, 2000.
25. M. K. McKusick and G. R. Ganger Soft Updates: A technique for eliminating most synchronous writes in the fast filesystem . In *Proceedings of the FREENIX Track (FREENIX-99)*, pp. 1-18, USENIX Association, 1999.
26. D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2: Colloquium on Software Engineering (CSE)*, pages 80–100, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
27. `Postfix` home page. *Maintained by Wietse Zweitze Venema*. `http://www.postfix.org`.
28. `qmail` home page. *Maintained by Daniel Julius Bernstein*. `http://cr.yp.to/qmail.html`.
29. Security Focus. `sendmail` vulnerabilities. `http://www.securityfocus.com/bid`
30. `sendmail` home page. *Maintained by Sendmail Consortium*. `http://www.sendmail.org`.
31. D. Shearer. MTAs - Choose the tool for the job `http://shearer.org/en/writing/mtacomparison.html`, June, 2001.
32. D. Sill Life with *qmail*. `http://www.lifewithqmail.org/lwq.html`, 2006.
33. W. Z. Venema. `Postfix` performance results. `http://www.porcupine.org/postfix/performance.html`.
34. W. Z. Venema. Denial of service (*qmail-smtpd*). `qmail` mailing list. `http://www.ornl.gov/lists/mailing-lists/qmail/1997/06/msg00317.html`, Jun. 11, 1997.
35. R. Veryard and A. Ward. Trusting components and services. `http://www.interscience.wiley.com/jpages/0038-0644`, 2001.
36. J. Viega and G. McGraw. *Building secure software - How to avoid security problems the right way.* Addison-Wesley, 2001.
37. Wikipedia contributors, Mail Delivery Agent. Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Mail_Delivery_Agent`.

38. Wikipedia contributors, Mail Transfer Agent. Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Mail_transfer_agent.

39. Wikipedia contributors, E-mail client. Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Mail_user_agent.

40. Wikipedia contributors, Tarpit (networking). Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Tarpit_%28networking%29.

41. J. Yoder and J. Barcalow. Architectural patterns for enabling application security, 1997.

42. Q. Zhong and N. Edwards. Security control for COTS components. *Computer*, 31(6):67–73, 1998.

Table I. Reported Vulnerabilities for Different Versions of `sendmail`

`sendmail` Version	Release Date	Reported Vulnerabilities
`sendmail` 4.1	Jul 1983	2
`sendmail` 5.61	Dec 1988	3
`sendmail` 8.9.0	May 1998	11
`sendmail` 8.10.0	Mar 2000	11
`sendmail` 8.11.0	Jul 2000	14
`sendmail` 8.12.0	Sep 2001	14
`sendmail` 8.13.0	Jun 2004	3
`sendmail` 8.13.6	Mar 2006	2
`sendmail` 8.13.7	Jun 2006	1
`sendmail` 8.13.8	Aug 2006	0