SECURITY ARCHITECTURE OF MAIL TRANSFER AGENTS

BY

MUNAWAR HAFIZ

B.Sc., Bangladesh University of Engineering and Technology, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

To my family.

Acknowledgments

It is very difficult to find a finite subset of people to express my gratitude. In fact it turns out to be more difficult than writing a chapter on Postfix or going through the *qmail* code base. I have had support and assistance from a lot of people and the fear of not mentioning somebody's name remains to be prominent as I go ahead and mention some.

This thesis started as an independent study project with Professor Ralph E. Johnson in Fall 2003. I cannot thank him enough for his invaluable guidance, encouragement and support. Working with him was an enlightening and truly edifying experience.

I had the opportunity to meet Daniel J. Bernstein, the author of *qmail*, while he was visiting UIUC and I thank him for taking his time amidst a busy schedule to meet with me twice. I have emailed Wietse Zweitze Venema, the author of Postfix, many times and he was very patient in guiding me through the inner workings of Postfix. I have also had email exchanges with Claus Assmann, the maintainer of *sendmail X* design, on various architectural aspects. I take this opportunity to thank them for their help.

I have immense gratitude for my father, Mohammad Abdul Hafiz, and my mother, Nargis Hafiz, for their dedication and confidence in me. My father was particularly pushing me to finish my thesis for some unknown agenda that he has and that made me go through the writing phase. My kid sister, Fariha Sultana, always had for me words of encouragement.

I would thank Farhana Ashraf for her important influence in my life. I would like to thank my closest friends Shafquat Mahmud and Ehsanul Kabir. Interestingly, the impact that these three had over my thesis would feel as negative force to an observer, because it would seem that I am engaged in doing idle chit-chat with them for hours. Those sessions of merriment make me go through the stresses of life. I also thank Alok Sutradhar for being my elder brother at Champaign. His wife Mehruba Firdaus and daughter Ankita light up my life in Champaign. I would also like

to thank all my Bangladeshi friends in Champaign - Khalid, Ritu, Rumi, Tabassum, Shamsi and Sharif.

Raja Afandi was my companion at the start of this research and I thank him for being a wonderful mate. I would also like to thank Brian Foote of Software Architecture Group, UIUC for sharing valuable insights and experiences with me on various topics. I would like to thank all the people who helped me with the reviews of our *qmail* and Postfix paper. And my colleagues Paul Adamczyk, Tankut Baris Aktemur, Jeffrey Overbey, Spiros Xanthos, Alejandra Garrido, Federico Balaguer and Danny Dig - I thank them all for being wonderful friends.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CDB Content Data Base.

DFD Data Flow Diagram.

DNS Domain Name System.

EDB Envelope Data Base.

ESMTP Extended Simple Mail Transfer Protocol.

ETRN Extended Turn.

IMAP Internet Message Access Protocol.

MDA Mail Delivery Agent.

MTA Mail Transfer Agent.

MUA Mail User Agent.

NFS Network File System.

PCRE Perl Compatible Regular Expressions.

POP3 Post Office Protocol 3.

QMQP Quick Mail Queuing Protocol.

QMTP Quick Mail Transfer Protocol.

RBL Real-time Black-hole List.

RHSBL Right Hand Side Black-hole List.

SMTP Simple Mail Transfer Protocol.

UBE Unsolicited Bulk Email.

UCE Unsolicited Commercial Email.

UUCP Unix-to-Unix Copy.

VERP Variable Envelope Return Path.

Chapter 1

Introduction to Mail Transfer Agents

Electronic mail handling has strong analogies with traditional mail systems. The main institution for traditional mail handling is the Post Office. The post office handles the physical transfer and delivery of mail once an addressed envelope is dropped in the outgoing mail box. The senders and recipients are never concerned about the activities that go on behind the scene routing the mail.

For electronic mail, the component that handles the similar task of routing and delivery is the Mail Transfer Agent or the Mail Transport Agent(MTA). The MTA works behind the scenes, while the user usually interacts with another program, the Mail User Agent (MUA), which uses the MTA to deliver the mail. Mail User Agents are like postmen of the postal system; they are the interface between the post office and the end-user.

There are similarities in the functionality of a post office and an MTA. E-mail messages are contained in electronic envelopes just like traditional mail. If the recipient of a mail message resides in the same area, the same post office receiving the mail handles the delivery task. This is similar to local mail delivery of an MTA, where only one MTA is involved. In case of distant recipients, mail messages are forwarded from the local post office to a distant one. The lookup of the post office happens dynamically based on the address on the envelope. Also, post offices combine and collaborate with different services. Similarly MTAs can send and receive mails by communicating with different networks.

This chapter describes the basics of MTA architecture. This chapter also outlines a list of architectural requirements of an MTA.

1.1 Mail Transfer Agent and Mail User Agent

A mail transfer agent (also commonly called a mail server or a mail router or Internet Mailer) is a computer program that transfers electronic mail messages from one computer to another. An MTA receives incoming e-mail from local users (people within the same domain) and remote senders and forwards outgoing e-mail for delivery. A computer dedicated to running such applications is also called a mail server. The most popular MTA is *sendmail* written by Eric Allman. Other popular Unix compatible MTAs are D.J. Bernstein's *qmail*, Wietse Venema's Postfix, *exim* developed in Cambridge University, *MMDF* (Multi-channel Memo Distribution Facility) for SCO Unix, *Smail 3.x*, *Courier* and *ZMailer*.

MTAs remain transparent to the users. Normally, the users interact with a program called Mail User Agent (MUA). An MUA is used to read, reply to, compose or dispose of e-mail. Examples of MUAs include the original Unix mail program, */bin/mail*, the Berkeley *Mail* program, the System V *mailx* program, free software programs like *mush*, *elm*, *mh* and commercial software programs like *zmail*.

The most popular mail transfer protocol for MTAs is SMTP. Modern MTAs use ESMTP (Extended SMTP) for sending e-mail. MUAs generally use POP3 or IMAP protocol for fetching e-mail from the mailbox.

1.2 Architectural Requirements of an MTA

Other than the functional requirements, MTAs have a number of non-functional requirements. The two most important architectural requirements of an MTA are security and reliability.

Security. Security is the main driving force behind the architecture of MTA. An MTA must keep email secure against casual attacks, and must not allow attackers to break into the system. The 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 should have multiple layers of defense for security.

Reliability. An MTA must be reliable. It must never lose a message except in the case of hardware failures or software failures beyond MTA's control that occur after MTA has accepted

responsibility for an e-mail.

Other requirements of an MTA are listed as follows.

Robustness. In the event of resource shortages (memory, disk, etc) an MTA should degrade gracefully. It should be implemented in such a way that it can also deal with simple OS errors. In such cases, an MTA should log error messages and exit without causing havoc to the rest of the system.

Performance. An MTA must be efficient in both time and space, because it might process thousands of messages simultaneously. It must show good performance without loosing reliability. An MTA should be able to take advantage of changes in the underlying OS and hardware that modify the relative performance of the available components. If more processing power is available then the overall system should become faster. However, the system performance is limited by the speed of the slowest component.

Availability. An MTA must be available. It must be able to cope with DoS attacks. In case of unavailability, the system must be able to resume from the original state.

Extendability. An MTA must support existing mail transfer protocols, but it should be extensible so that it can incorporate future protocols. An MTA should provide options for performance tuning and replacement of components.

Maintainability. An MTA should be designed and implemented to be easily maintainable. This not only means adherence to standard coding practices, but also means simple and understandable design. Simplicity contributes a lot to the maintainability of the architecture.

Testability. It is important to automatically test the functionalities of an MTA. Having tests greatly simplifies maintainability because changes to the implementation can be automatically tested by doing regression testing. Tests should have good path coverage.

Portability. An MTA should be portable to different underlying OSs.

1.3 The Simplest Model of an MTA

The most basic tasks that an MTA does are,

- Communicate with the sender and receive outgoing mail from sender.

- Communicate with the receiver and deliver mail to the recipient.

From functional perspective, the simplest MTA should have two major components – one handling the mail receive task and one doing delivery. The DFD in figure 1.1 illustrates the simplest MTA.

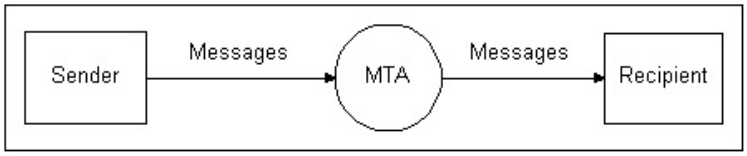

Figure 1.1: The Simplest MTA

DFD like diagrams are used to describe architecture of MTAs in the following chapters.

1.4 An Operational Model of an MTA

The mail senders and the mail recipients can be classified into two classes – local and remote. The functionalities of the simplest MTA are therefore updated to accommodate the following scenarios.

- MTA receives mail messages from a local sender and then transfers it to either a local or a remote receiver depending upon the mail address. Users use the abstraction of MUA to send messages to MTA so that they can be transferred to the recipient and similarly for receiving messages from MTA.

- MTA also receives messages from other MTA's over some predefined network protocol like SMTP. If the message is destined for the local user, then the message is stored in local mailbox or mail directory so that the user MUA can retrieve it. If the mail is not for this domain, then it is routed to other remote mail server.

Hence an operational MTA should have four functional components for mail receive and delivery. The structure of an MTA follows its context closely. There will be a component that handles messages from a MUA, a component that handles messages from another MTA, a component that sends mail to a local user, and a component that sends mail to a remote user.

Figure 1.1 also does not indicate the need for a mail queue, which turns out to have a big impact on the design of an MTA. Post offices do not deliver mail as soon as it is dropped in the mailbox.

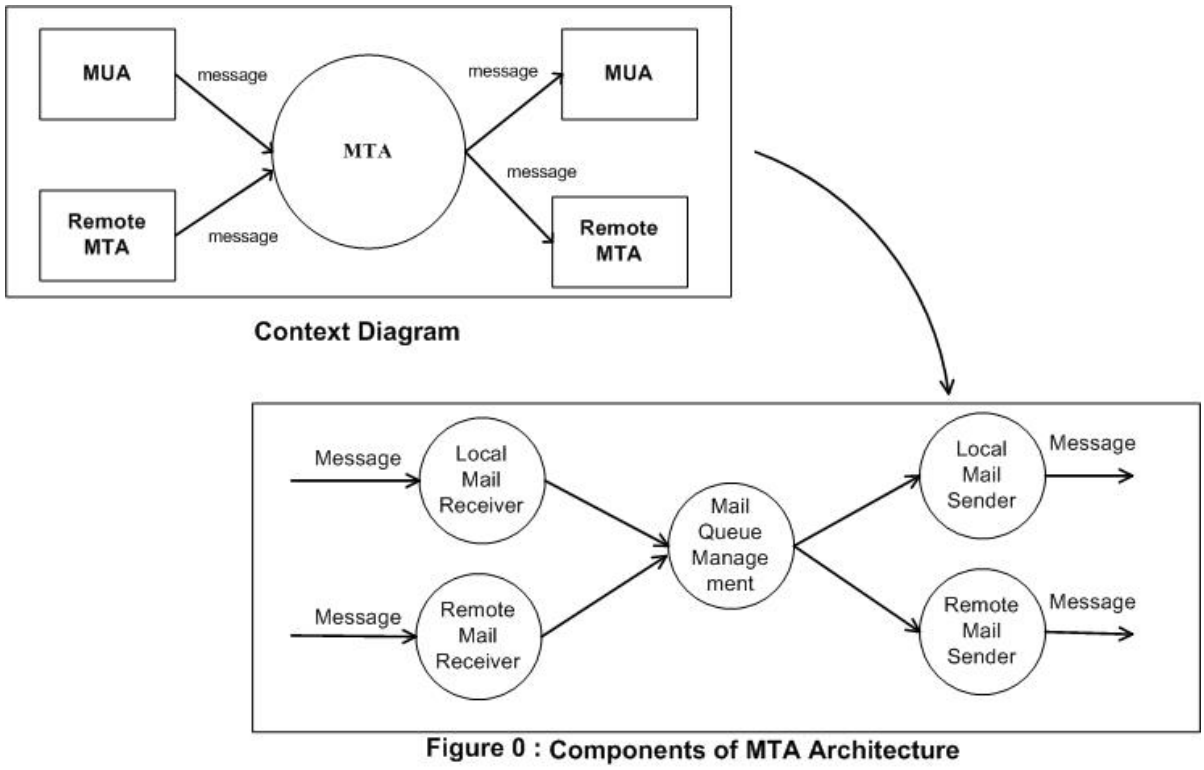

Figure 1.2: Architecture of an Operational MTA

They collect outgoing mail and process it periodically. An MTA does the same. It 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 can 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 will lose messages if its host fails before the messages are delivered. The addition of a mail queue management component completes the architecture of an operational MTA.

1.5 Conclusion

Figure 1.2 outlines the functional components of an MTA. The design decision of the implementation of the components depends more on the non-functional requirements. The influence of security and reliability over architecture is described in the following chapters where architecture of *sendmail*, *qmail* and Postfix are explained in detail from these perspectives. Finally, the architecture of *sendmail X* is described. *sendmail X* is an ongoing renovation project of the old *sendmail* and

its architecture is heavily influenced by the best practices of more secure and reliable *qmail* and Postfix architecture.

Chapter 2

sendmail ARCHITECTURE

The *sendmail* program pre-dates the internet evolution. It was the first widely used MTA. This chapter presents its architecture and also discusses the security issues associated with the architecture.

2.1 History of *sendmail*

sendmail was written by Eric Allman while he was at the University of California at Berkeley. Eric Allman wrote the pre-cursor of *sendmail*, named *delivermail* to handle mail routing requirements. The *delivermail* program was shipped in 1979 with 4.0 and 4.1 BSD Unix. But *delivermail* was not flexible enough to handle the changes in mail routing requirements. One of its major problems was that configuration was compiled-in.

In 1980, ARPAnet began converting from NCP (Network Control Protocol) to TCP (Transmission Control Protocol). This change increased the number of possible hosts from 256 to over one billion. SMTP (Simple Mail Transfer Protocol) was developed to transport mail. *delivermail* was insufficient to support new protocols and revisions of protocols. Eric Allman evolved *delivermail* to *sendmail* responding to the changing requirements.

The first *sendmail* program was shipped with 4.1c BSD (the first version of Berkeley UNIX to include TCP/IP). The current version of *sendmail* is 8.13.4 and it came out in March 2005.

Several other people contributed to *sendmail* development. In 1987, Lennart Lovstrand of the University of Linköping, Sweden, developed the IDA enhancements to BSD sendmail Version 5. IDA (which stands for "Institutionen för Datavetenskap") added a number of improvements to

sendmail (such as support for dbm files and separate rewriting of headers and envelopes) and fixed a number of bugs. Later developers from all around the world contributed to the project and it solved many complex mail routing problems.

Many vendors have modified *sendmail* to suit their particular needs. Sun Microsystems made many modifications and enhancements to *sendmail*, including support for nis and nisplus maps that contains host and password lookup functions. Hewlett Packard also contributed many fine enhancements including 8BITMIME support.

This explosion of *sendmail* versions has led to a great deal of confusion. Solutions that work for one version of *sendmail* fail miserably with others. Beyond this, configuration files are not portable, and some features cannot be shared.

This chapter describes the basic principles of *sendmail* architecture that has remained consistent, not the details of configuration changes that has occurred between different versions. Details of *sendmail* internals can be found in [13] and [24].

2.2 *sendmail* Architecture

sendmail runs as one big process with root privilege. The means the components of a generic MTA shown in figure 1.2 all run in the same address space. However, *sendmail* is more than one single program. One of the key parts of *sendmail* is the configuration file that defines the location and behavior of all the parts and includes rules for re-writing addresses. Another important part is the mail queue.

2.2.1 *sendmail* Program

The *sendmail* program listens to the network for incoming mail, and forwards mail messages to other machines. It handles local delivery by delivering mails to a local process. It also delivers mails by appending to mailbox files and piping through other programs.

Mail delivery can be an on-demand service. However, mail can only be received when the *sendmail* program runs as a daemon process listening to the TCP/IP network. As a daemon process, it remains independent of control from all terminals.

The *sendmail* program adopts different names for the different tasks it performs. For example,

the *sendmail* process that runs as a daemon listening for incoming mails is called *smtpd*. Again, the *sendmail* program that prints the queue is called *mailq*. All of these are actually the same *sendmail* program running with different command line parameters. Table 2.1 shows a list of some command line parameters.

Table 2.1: Command Line parameters of *sendmail* program

Name of Parameter	Description
-ba	Run using old ARPAnet protocols
-bD	Run as a daemon process without forking
-bd	Run as a daemon process (Also called *smtpd*)
-bH	Purge persistent host status (Also called *purgestat*)
-bh	Print persistent host status (Also called *hoststat*)
-bi	Rebuild alias database (Also called *newaliases*)
-bm	Become a mail sender
-bp	Print the queue (Also called *mailq*)
-bs	Run SMTP on standard input (Also called *bsmtpd*)
-bv	Resolve aliases for mail and also checks whether mail can be forwarded to the resolved address

2.2.2 Mail Queue

Mail Queue is an important part because it adds to the reliability of the MTA through persistence. The name of the queue directory is usually *mqueue*.

All the processing of email messages is done in the queue directory. After *sendmail* has processed the configuration file, it does a *chdir* into the queue directory and does all the rest of its work from there. However, this has an important side effect. If some program fault causes a core dump, the core image is left in the queue directory. This has security and reliability consequences.

The queue directory has very narrow permissions. It must be owned by root with a preferred privileged mode of 0700. Before *sendmail* version 8, such narrow permissions would cause C-shell scripts run from a `~/.forward` files to fail. *sendmail* version 8 lets specification of alternate directories in which to run programs.

As a further precaution, all the components of the path leading to the queue directory should be owned by root and be writable only by root.

When a message is stored in the queue, it is split into pieces. Each of those pieces is stored as a separate file in the queue directory. Six different types of files may appear in the queue directory.

The type of each is denoted by the first two letters of the filenames. Each filename begins with a single letter followed by an "f" character. This is shown in table 2.2.

Table 2.2: Queue File Types

File	Type of content
`df`	Data File (Body of the message)
`lf`	Lock File (Deprecated in latter versions)
`nf`	ID Creation File (Deprecated in latter versions)
`qf`	Queue control file (Header of the message)
`tf`	Temporary qf rewrite Image File
`xf`	Transcript File

The complete file name has one of the prefixes listed in table 2.2 followed by a unique queue identifier. Before version 8.6, *sendmail* used the process identification number (pid) of the *sendmail* process that is trying to create the file. *sendmail* forks a child process to process the queue. So the pid is likely to be unique. For V8 *sendmail* an extra letter is used as prefix of the unique identifier. It is an uppercase letter that corresponds to the hour that the identifier was created. The hour begins at midnight with A, 1 AM is B and so on. This also is helpful for auditing purposes, because the time is encrypted in alphabet format in file name.

Mail messages are composed of a header and a body. When queued, the body is stored in the `df` file. New versions of *sendmail* use 8 bits to represent the content, whereas the older versions use 7 bits with the MSB turned off.

Modern versions of *sendmail* does not use `lf` , but it was used in older versions. When *sendmail* processes a queued message it creates an empty lock file. That lock file is necessary to signal to other running *sendmail* processes that the mail message is busy. Current versions simply use *flock* or *fcntl* on the `qf` file.

Old versions of *sendmail* used an `nf` file when creating a message identifier to avoid race conditions. Newer versions create the queue identifier when first creating the `qf` file.

The `qf` file contains the message header. In addition to the header, it also contains information necessary for delivering the message, ordering and prioritizing the message delivery etc.

The `tf` file keeps a temporary image of the `qf` file as `qf` is being altered. This is done to ensure reliability.

A given mail message may be destined for many recipients, requiring different delivery agents.

While calling the necessary delivery agents, *sendmail* saves all the error messages it receives in a temporary `xf` file.

Messages stored in the queue directory are either processed periodically (when *sendmail* is running as a daemon) or on demand. First the queue directory is opened for reading. Then, the `qf` files are read to gather their priorities and times.

A single queued message has a single sender but may have many recipients. When processing a queued message, *sendmail* attempts to deliver it to all recipients before processing the next queued message. First it locks the files. Then the `qf` file is opened and read. For each recipient, delivery is attempted. If delivery is successful, that recipient's address is removed from the *sendmail* program's internal list of recipient addresses. If delivery fails, that address is either left in the list or bounced, depending on the nature of the error. *sendmail* re-examines the list after processing all the recipient addresses. In case the mail is delivered to all recipients, it is de-queued by the removal of all the corresponding contents in the queue directory. If any recipients are left, the `qf` file is rewritten with the list of remaining recipients. Finally, the `qf` file is closed, thus freeing its lock.

If the system crashes midway through the mail delivery to multiple recipients, mails are re-delivered. *sendmail* also has options for rewriting the `qf` files after processing n messages (the value of n is set using some external configuration parameter). This is an example of checkpointing [6] to keep the file reasonably up-to-date.

2.2.3 Local Mail Delivery Process

Local mail delivery is done by appending mail messages to a local user's mailbox. Each local user has a mailbox on the local file system.

Under Unix, a user's mailbox is a single file that contains a series of mail messages. The usual Unix convention is that each message in a mailbox begins with the five characters "From " (the fifth is a blank space) and ends with a blank line.

In general, *sendmail* does not put mail messages directly into files. Instead, it calls other programs to perform delivery. These programs are called Mail Delivery Agents (MDA).

The */bin/mail* program is generally used to append mail to the user's mailbox. Other programs

like *deliver* or *mail-local* can also be used. The structure of the user's mailbox is transparent to the *sendmail* program. It is only concerned about the name of the program to invoke.

The */bin/sh* program is generally used to run other programs that handle delivery. Other programs like the Korn shell (*/bin/ksh*) or the *sendmail* restricted shell (*smrsh*) are also used. Instead of running mail delivery programs directly, *sendmail* runs a shell and tells that shell to run the program.

2.2.4 Remote Mail Delivery Process

The *sendmail* program transports mail over TCP/IP network. *sendmail* version 8 and above also transports mail over ISO netowrk. The most popular mail transport protocol is SMTP.

Also *sendmail* supports UUCP protocol by running the */bin/uux* program.

There are many other kinds of network protocols that *sendmail* can use to transport email. For example, it can be used to send faxes (diverse functionality), or it can be used to transport e-mail over a DECnet network (diverse underlying network), or it can be used to send mails to Macintosh machines (diverse systems).

2.3 Security of the *sendmail* Architecture

Without proper configuration, *sendmail* can be used as a security hole and this can lead to an insecure and possibly compromised system. Since *sendmail* usually runs as a root process, it is a prime target for intrusion. If *sendmail* is not properly installed and configured, external probes over networks can gain information that can later be used o break into the system. Once that happens, improper file permissions can be used to trick *sendmail* into giving away root privilege.

There are some places where *sendmail* needs to run as root. One example is the binding and listening to the TCP/IP port. Again, access to the mailboxes and personal configuration files of users also requires root privilege. Contrarily, *sendmail* can run at a lower privilege level (as a local user) to access local information, but root privilege is necessary to change (lower) the privilege level.

sendmail only runs with root privilege when it needs to satisfy legitimate needs of the user. When delivering mails locally to a mailbox, it does not run as root. MDAs handle the local

delivery task and they run with their own privilege level. Again, the access to ~/.forward files of the users requires root privilege so that it can temporarily become the user to read and process the ~/.forward file. Also, users want programs that run on their behalf to run as themselves. This also requires root privileges.

Running *sendmail* as an un-trusted pseudo-user (such as 'nobody') does not work. It causes programs in users' ~/.forward files to be run as 'nobody', and it requires the queue to be owned by 'nobody'. Consequently, such a scheme allows any user to break into and modify the queue.

The change of privilege level is done by using *seteuid* or *setreuid* or *setuid*. *setreuid* is preferred over *seteuid* and *setuid* because it allows *sendmail* to temporarily give away both its real and effective root privilege, then to get it back again.

2.4 Problems of *sendmail* Architecture

The complexity of *sendmail* architecture comes from 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 problem is exacerbated because *sendmail* runs as root. So there has to be complex configuration to limit a malicious user from taking control of the system. The specification language is complex itself. The security holes that were found in *sendmail* were primarily of two categories.

- Security holes that exploit the root privilege.
- Security holes that come from mal-configuration.

A list of *sendmail* security and reliability problems from *sendmail* version 8.6 in 1993 to *sendmail* version 8.8.7 in 1997 is available in [5]. A list of *sendmail* vulnerabilities from *sendmail* version 4.1 to *sendmail* version 8.12.11 is listed in [23]. The most recent vulnerability of *sendmail* was reported in February, 2005.

The architecture of *sendmail* was not considered a problem when it was created. However, the long list of *sendmail* vulnerabilities and the series of patches to solve them have slowly put *sendmail* out-of-favor. Before 1990s, *sendmail* had almost 100% of the MTA market. In 1996, the figure was about 80% while in 1999, the figure became 60%.

This section describes some security bugs found in *sendmail* and explains the detail of the security hole. These vulnerabilities are because of design errors at different places of *sendmail* .

2.4.1 Problem with Super User Privilege

sendmail 8.12.0, in its default installation, is no longer using a setuid root binary to manipulate the mail queue and submit mail. This security enhancement is supposed to minimize the eventual impact of local *sendmail* vulnerabilities. The new *sendmail* binary is setgid 'smmsp', where 'smmsp' is a special group with read-write queue access permissions.

sendmail allows users to specify custom configuration files or configuration parameters. In the case of processing of untrusted configurations, *sendmail* was supposed to drop all extra privileges and continue to run at user level, causing no security risk. This mechanism worked fine in *sendmail* versions prior to 8.12.0. Because of a programming error, this inherited code fails to drop extra group privileges completely in new setgid conditions, leaving the saved gid value untouched. By calling the *setregid* function, the attacker will be able to regain dropped privileges [8]. This may be possible due to several bugs in the config file parser. Extra privileges expose a security risk to the mail subsystem and, in specific conditions, might lead to further privilege elevation. This was reported in 2001 by Michal Zalewski. The problem was solved in version 8.12.1 of *sendmail*.

Another vulnerability reported in *sendmail* version 8.12.9 allows local users to elevate their privilege level. Three *sendmail* scripts create insecure temporary files that might allow a local user to elevate their privileges [11]. The three scripts are - *expn*, *checksendmail* and *doublebounce.pl*. Successful exploitation would allow an attacker to gain the privileges of the user invoking the scripts. This as reported by Paul Szabo in May 2003. Debian has released patches to remove this vulnerability.

2.4.2 Buffer Overflow Attack

sendmail 8.12.7 has a buffer overflow in address parsing due to a char to int conversion problem which is potentially remotely exploitable [10]. The flaw is detected in the *prescan()* procedure which is used for processing addresses in the SMTP header. The goal of an attacker is to replace the two LSB of the saved frame pointer and make it point to an area controlled by the attacker.

The problem was reported by Michal Zalewski.

To provide partial protection to internal, un-patched *sendmail* MTAs, version 8.12.9 changes by default (char)0xff to (char)0x7f in headers etc. Also to prevent vulnerability performing 7 → 8 or 8 → 7 bit MIME conversions, the default parameter is re-calibrated.

2.4.3 File Locking Related Problems

File locking is used in *sendmail* for a variety of files. Any user who opens a file has a shared read lock on the file. This prevents other processes from getting an exclusive write lock on the file. This creates a local denial of service attack [9].

Version 8.12.4 of *sendmail* will change the existing permissions of *sendmail*- specific files to prevent access from unauthorized users.

2.4.4 Local User Mailbox Related Problems

Mailboxes have reliability problems as mail queues. It is difficult to modify existing files reliably, portably, and efficiently under Unix. *sendmail* follows a particular mailing directory format called the "mailbox" format. This format, along with a latter modification named "mbox", use a single file for all mail entry. So, when multiple sources are trying to put their mail into the file, there is some need for locking the resource. However, a machine may crash while it is delivering a message. For the traditional formats, this means that the message will be truncated. Even worse: for "mbox" format, if the message is truncated in the middle of a line, it will be silently joined to the next message. The MTA will try again later to deliver the message, but it is unacceptable that a corrupted message should show up at all.

2.4.5 Problems with Mishandling of Mail Messages

sendmail treated programs and files as addresses. That means mail messages may be passed as parameters to a program or they can be truncated to a file. This feature was exploited in one of the classic cases of Internet worm.

In November 1988, a worm was released in the Internet to launch a DoS attack against the infected. One of the loopholes that the worm abused was the debug function of the *sendmail*

program [14]. The *sendmail* feature of sending a mail to a program 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 recipient that includes command to strip off the headers and pass the remainder of the message to a shell. The body of the message consisted of a bootstrap shell script that, when executed, would connect back to the attacking machine via TCP and downloaded pre-compiled object code replication of the worm. It then tried to link the target code and execute it at which point the new machine gets infected.

2.5 Conclusion

The architecture of *sendmail* was considered a strength when it was created because it met the requirements for flexibility that were important at that stage. *sendmail* was the MTA that made SMTP popular. *sendmail* incorporates a lot of features for mail transfer and configuration management. A lot of this was necessary because *sendmail* ran as root and configurations were needed to prevent users from breaking into the system. The requirements of an MTA changed with the introduction of Internet and security and reliability became the major issue that an MTA should handle. The next chapter describes the architecture of *qmail* which was carefully thought out based on these new requirements. But the key principle to achieve security is keeping the architecture simple and *qmail* is a very good example of clear and simple architecture. The MTAs that followed were greatly influenced by *qmail*'s security principles. These will be described in the following chapters.

Chapter 3

qmail ARCHITECTURE

The architecture of *qmail* is motivated by a series of security breaches in *sendmail* . However, *qmail* is not only more secure than *sendmail*, it is also more efficient and easier to understand. Thus the architecture is able to accomplish several goals at once. The security of *qmail* is based on a few patterns and understanding its architecture can help people make other applications secure.

3.1 History of *qmail*

qmail was written by Daniel Julius Bernstein (also known as djb in the Unix world). D.J. Bernstein was in University of Berkeley when he started writing *qmail* in 1995. The first public release was the beta version 0.70 of *qmail* and it occurred on January 24, 1996. The first general release was version 1.00 and it was announced on February 20, 1997. The current version, *qmail* 1.03 was released on June 15, 1998.

A lot of users have moved to *qmail* and Postfix from *sendmail*. According to SMTP server statistics, the Internet had approximately 1791 SMTP servers running *qmail* in March 1997. In August 1997, this figure became 4078. This increased to 352000 from statistics taken in October 2000 and doubled to 717000 in October 2001. From statistics taken in October 2001, about 47% of the reachable servers were running *sendmail* under Unix, which dropped about 5% from the previous survey. Meanwhile, *qmail* user base increased form 9% to 17%.

qmail provides a secure alternative to *sendmail*. In March 1997, Daniel J. Bernstein declared a $500 prize for the first person to publish a verifiable security hole in *qmail* and the prize is still unclaimed. Only two minor bugs have been found in *qmail* since version 1.0.

3.2 *qmail* Architecture

The architecture of *qmail* comes from a few simple patterns. These patterns are not new for *qmail*, but *qmail* is an example of how to use them effectively. One of the key principles of *qmail*'s architecture is Defense in Depth [29], which means that *qmail* does not depend on any single pattern to achieve security, but has several layers of security. First, the way it is divided into modules tends to decrease the damage caused by security break-ins, and eliminates some kinds of errors. The module decomposition makes each module simpler, so it can be inspected for correctness. It makes multiprocessing more efficient. The way that *qmail* uses the file system makes queuing and delivering the mail more reliable. The low-level coding patterns eliminate important classes of errors such as buffer overflows. The result is an MTA architecture that is simpler and more reliable than that of *sendmail*.

Chapter 1 introduces the basic requirements and the generic components of an MTA. The architecture of *qmail* follows that closely by creating processes with separate user privileges for each of the tasks. The local mail handler (*qmail-inject*) receives local mail messages through reading a mail message from its standard input, adds appropriate information to the message header, and invokes *qmail-queue* to transfer them to mail queue. The remote mail receiver (*qmail-smtpd*) receives remote messages over SMTP from other MTA's and sends them to mail queue. *qmail-send* gets the messages from the central mail queue, formats address and handles it to either *qmail-local* or *qmail-remote* for local and remote delivery. These processes are spawned by *qmail-lspawn* and *qmail-rspawn*. If a message is temporarily undeliverable to one or more addresses, *qmail-send* leaves it in the queue and tries the addresses again later. *qmail-send* also communicates with *qmail-clean*, which removes delivered messages from the queue.

3.2.1 Compartmentalization

Figure 1.2 shows the components of a generic MTA architecture. An alternative to the *sendmail* way of implementing everything as a single process is to implement the components as separate processes. For example, SMTP server can be put in a separate address space from the rest of the MTA. Even if attackers discover a buffer overrun in the SMTP server, they can do little damage to *qmail*, because *qmail-smtpd* does nothing except call *qmail-queue* and give it a messages to

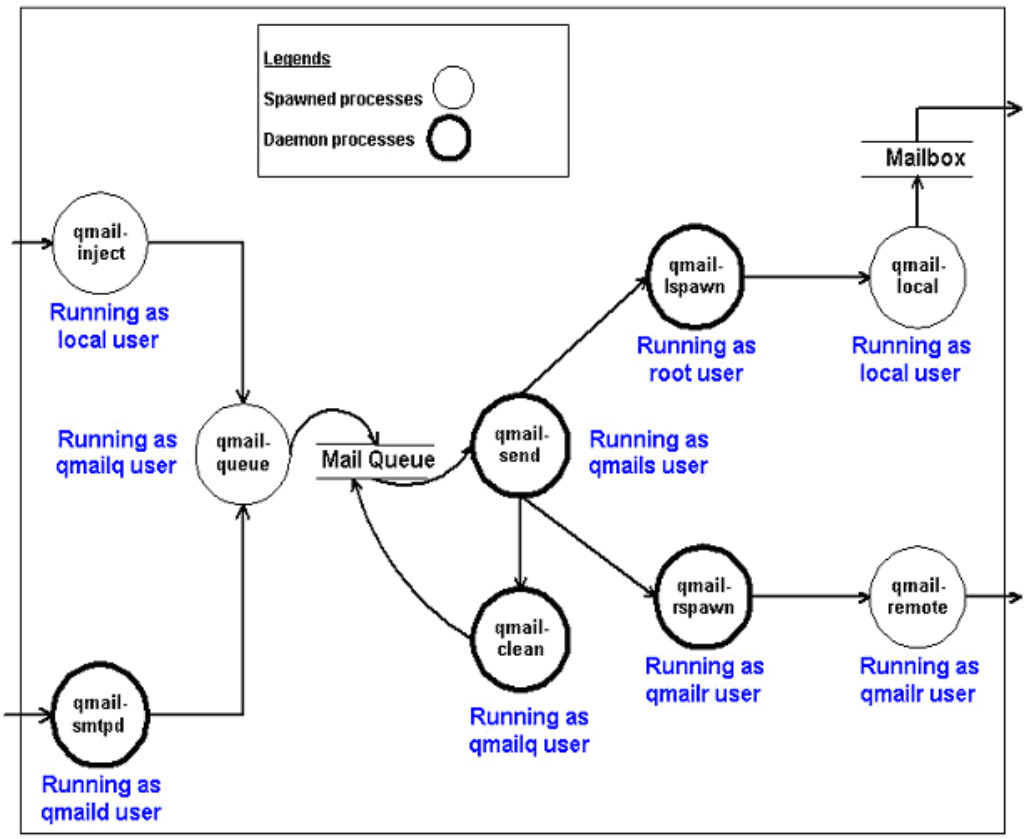

Figure 3.1: Major *qmail* Processes

put on the mail queue. It does not write on any files except a log file. This limits the extent of an error in the SMTP server of *qmail*. Moreover, the SMTP server is small and it is easy to read it and verify that it doesn't write to any file except the log file.

Separting the SMTP server from the rest of the MTA is an obvious way to improve security. This is done in a redesign of *sendmail* by Zhong and Edwards [30] and in Postfix [21].

This is an example of the first security pattern, compartmentalization. The name comes from Viega and McGraw [29], but the pattern has been described by many people, such as the Execution Domain pattern by Fernandez [15].

The key principle of compartmentalization is to have modules run as separate processes. This is very easy in Unix; it is part of the Unix way of reuse to try to make small, reusable processes. However, it is not part of the object oriented way of reuse. In Java, we would have made separate objects for the five modules and run them all in one process. Compartmentalization says to do it the Unix way, not the object oriented way.

With such decomposition, even if some intrusion happens in mail receiving part of the *qmail* system, the whole system is not compromised. The security critical operations of MTA are divided among basically three main portions of mail receiver, mail queue and mail sender modules, hence decreasing the security risk to the whole system.

Some of the modules of a system may be more vulnerable than others. As remote message handling modules are more prone to security breach, it is now easier to implement communication channel or resource specific security techniques for efficiently enhancing the security of the system. Similarly local mail sender can be implemented securely in order to prevent attackers from writing to security-critical targets. Compartmentalization enables adoption of custom security mechanism for different parts of the system.

Compartmentalization also makes the processes smaller and therefore simpler and easier to understand. The length of code of *sendmail* is over 60000 lines and all this is to handle one process that does everything. *qmail* has more than 20 separate modules (under the names '*qmail-XXX*'). Still the total length of all code for all modules is not greater than 15000 lines. The largest module of *qmail* is *qmail-send* and it is about 1612 lines. All the rest are less than 800 lines. This shows the impact of good design on the length of code.

The problem with compartmentalization is that it makes testing difficult. Because a lot of bugs occur in the way components interact, integration testing becomes more important than unit testing. It is difficult to correctly identify the problems and write tests when there are lots of small modules in a system. However, if the modules are simple and the contract of communication between them is clear then it can be proven that the modules are free of bugs and the integration would be bug-free as well. The goal of compartmentalization is therefore to achieve simplicity.

Table 3.1: Security Pattern - Compartmentalization

Problem A security failure in one part of a system allows another part of the system to be exploited.
Solution Put each part in a separate security domain. Even when the security of one part is compromised, the other parts remain secure.

3.2.2 Distributed Delegation

The main problem with *sendmail* is that it runs as super-user. An MTA needs to write in the directory of a local user to deliver mail. Either the MTA must run as the user or it must run as the super-user. *sendmail* runs as a single super-user process. When it writes a file, it checks to make sure that it is not abusing its privileges. A number of the *sendmail* errors resulted from not making sufficient checks.

Because *qmail* runs as many processes, 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 and so cannot write on either user files or system files. The *qmail-smtpd* process runs with 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 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 two processes, *qmail-lspawn* and *qmail-local*. *qmail-lspawn* runs as the super-user, but is short (less than 500 lines) and simple. First it looks up the target user to find the uid, then it runs *qmail-local* after becoming that user. It does not write any files, nor does it read any files once it decides on its new uid.

Table 3.2: Security Pattern - Distributed Delegation

Problem
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.

Solution
Partition responsibility across compartments such that compartments that are likely to fail do not have data that needs to be secure. Assign responsibilities in such a way that several of them need to fail in order for the system as a whole to fail.

This is called Distributed Delegation by [28].

Distributed Delegation extends the Compartmentalization pattern by outlining an efficient way to partition the processes. For this, the different communication resources are first identified. There are four main resources. These are the sent messages and received messages, each from the local or remote user. Efficient partitioning is done by partitioning along these resource boundaries. Compartmentalization is the pattern that suggests the partitioning of a monolithic process; the distributed delegation pattern shows an efficient way to do it.

The partitioned processes are also run with the lowest possible privilege level. *qmail* does this by creating a number of user groups and assigning processes to that group. Here is a list of *qmail* programs and their corresponding 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* spawns all 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 local user. Again, *qmail-queue* is the only setuid program. This is because, *qmail-queue* runs as qmailq and yet it has to insert messages in the queue.

3.2.3 Interfaces of *qmail* Processes

It can be argued that the reason of Compartmentalization of *qmail* processes is something other than security. 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 flows data 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* both 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 lspawn, 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 sender's address and the recipients' addresses. *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 in 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 have similar interfaces, they can be used as components in shell scripts. But the purpose of using separate processes in *qmail* is first compartmentalization and second multithreading. Reuse is not a purpose at all, as is proven by the idiosyncratic interfaces.

3.2.4 Reliable Mail Queuing

The mail queue is in the center of an MTA architecture. The reliability of *qmail* depends on the mail queue being reliable. In *qmail*, mail is placed in the mail queue only by *qmail-queue*. However, *qmail-queue* isn't a Unix process, but rather a Unix program that can be called simultaneously by many processes. For example, it can be called by the MUA (when it calls *qmail-inject*) and by *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.

The files are created by different processes. There should be some mechanism to ensure that multiple processes do not contend over a filename. This separation is very important for ensuring reliability. The key to ensuring that one *qmail-inject* process does not interfere with another is for each of them to give each message a unique name for its file in the mail queue directory. Each file represents a message, so this is the same as assigning a unique ID to each message. This is difficult to do efficiently and portably on Unix.

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 created the message as the message ID, but then changes the message ID to be the i-node number of the file. This solution is portable and efficient. Its only drawback is that it is not as simple as if a message had only one ID over its life.

This is an example of the pattern of ensuring non-interference of processes by unique entry of information.

Table 3.3: Reliability Pattern - Unique Location for each Write Request

Problem Many processes need to add information to a database concurrently. How do we ensure that multiple write operations are handled correctly and even if there is a crash no trace is left of the failure?
Solution Ensure that every write request is to a different location. Thus, different processes are never writing on the same file at the same time. This pattern is called Unique Atomic Chunks by [17].

The queue of *qmail* is managed by three processes. The queue is shared between *qmail-queue* (inserts messages into the queue), *qmail-send* (reads messages from the queue and takes appropriate sending actions) and *qmail-clean* (garbage collection).

The mail queue is divided into multiple directories and messages are moved from one directory to another. This transfer process from one directory to another resembles different states in a message queuing cycle. All the time the state information is retained.

qmail's queue consists of several directories. The directory structure is given in Table 3.4.

Table 3.4: Directory Structure of *qmail*'s mail queue

Subdirectory	**Contents**
`bounce`	Permanent delivery errors
`info`*	Envelope sender address
`intd`	Envelope under construction by *qmail-queue*
`local`*	Local envelope recipient addresses
`lock`	Lock files
`mess`*	Message files
`pid`	Used by *qmail-queue* to acquire an i-node number
`remote`*	Remote envelope recipient addresses
`todo`	Complete envelopes

Directories marked with an "*" contain a series of subdirectories named "0", "1", ..., up to (conf-split - 1), where conf-split is a compile-time configuration setting (usually a prime number). The newly created file is put in directory number (i-node mod conf-split).

To add a message to the queue, *qmail-queue* first creates a file in the `pid` directory with a unique name. The file system assigns that file a unique i-node number. Usually, the unique file names will come from the unique i-node numbers.

qmail-queue then divides the i-node number with conf-split value and puts the file in the appropriate subdirectory under `mess` directory. It then creates an entry in the appropriate subdirectory under `intd` directory and writes envelope information in that directory.

Finally *qmail-queue* creates a new link in the `todo` directory pointing to the entry in the `intd` directory. At that instant the message has been successfully queued, and *qmail-queue* leaves it for further handling by *qmail-send*.

qmail-send takes messages out of the mail queue and sends them to local and remote destinations. When *qmail-send* notices the entry in the `todo` directory, it creates one entry in the `info` directory and one entry in the appropriate subdirectory of either `local` or `remote` directory. Then it removes the entries first from the `intd` and then form the `todo` directory. At that instant the message has been successfully preprocessed.

Each address in `local` and `remote` is marked either NOT DONE or DONE. DONE means that the message was successfully delivered, or the last delivery attempt met with permanent failure. Either way, *qmail-send* should not attempt further delivery to this address. NOT DONE means if there have been any delivery attempts, they have all met with temporary failure. Either way, *qmail-send* should try delivery in the future.

qmail-send may at its leisure try to deliver a message to a NOT DONE address. If the message is successfully delivered, *qmail-send* marks the address as DONE. If the delivery attempt meets with permanent failure, *qmail-send* first appends a note to `bounce` directory and then marks the address as DONE.

When all addresses in `local` or `remote` directory are DONE, *qmail-send* deletes the directory by calling *qmail-clean*. When `local` and `remote` directory entries are gone, entries in the `info` and `mess` directory are also removed and *qmail-send* goes back to stage 1. This whole process is shown in Figure 3.1.

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. 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 DONE, the new *qmail-send* sends the message again. This redundancy is not harmful in any way, but it is very crucial in ensuring the reliability of the whole system.

Unique Atomic Chunks pattern also takes care of some other reliability issues. In *qmail* queue, because all the messages are kept in the file system with unique i-nodes and named as i-node numbers, the messages survive crash because they are not in memory but permanently residing in file system. If the whole file system crashes, then the data would be lost but there is no expectation of retrieval in that case anyway. Besides the queuing process goes through separate states and there are different error handling paths for different states.

For example, if the computer crashes while *qmail-queue* is trying to queue a message, or while *qmail-send* is eliminating a message, the message may be left in state 2 or 3. When *qmail-send* sees a message in state 2 or 3, it deletes the entry in `intd` if that exists, then deletes the entry in

the `mess` directory. Any *qmail-queue* handling the message must be dead at that point.

3.2.5 Mailbox Management

The same technique used to get a reliable mail queue is used in *qmail* to eliminate reliability problems associated with traditional mailbox formats.

Ideally, every message should be guaranteed complete upon delivery. A machine may have two programs simultaneously delivering mail to the same user. The traditional formats require the programs to update a single central file. If the programs do not use some locking mechanism, the central file will be corrupted. There are several locking mechanisms, none of which work portably and reliably.

Many sites use Sun's Network File System (NFS). NFS makes the problems even worse. The "dotlocking" program was written to provide an alternative to NFS locking because ordinary file locking tends to create all sorts of problems with hung state and lock daemons. dotlock locks the file mailbox and runs command with the given arguments. If the command does not finish in 55 seconds, it is sent an ALRM signal. The command takes appropriate actions in response to this signal, for example it restores mailbox to the original state. After 60 seconds the lock is removed, even though the command is not finished. Even with the dotlocking mechanism, there are race conditions.

The solution to the 'mbox' format and the problems related to it is the 'Maildir' format in *qmail*. This format is used by other MTAs including Postfix, exim, Courier and Mac OS X Mail Application with the RCI mail server. A 'Maildir' is a structured directory that holds e-mail messages. *qmail*'s 'Maildir's are a simple data structure, nothing more than a single collection of e-mail messages.

A directory in 'Maildir' format has three subdirectories, all on the same filesystem. The subdirectories are named `tmp`, `new`, and `cur`. Each file in new is a newly delivered mail message. The modification time of the file is the delivery date of the message. Files in `cur` are like files in `new` except that they have been seen by the user's mail-reading program. The `tmp` directory is used to ensure reliable delivery using the following six steps.

1. *chdir*() to the 'Maildir' directory.

2. *stat*() the name `tmp/time.pid.host`, where time is the number of seconds since the beginning of 1970 GMT, pid is the program's process ID, and host is the host name.

3. If *stat*() returned anything other than ENOENT, the program sleeps for two seconds, updates time, and tries the *stat*() again, a limited number of times.

4. Create a file `tmp/time.pid.host`.

5. NFS-write the message to the file.

   (a) Check the number of bytes returned from each *write*() call.

   (b) Call *fsync*() and check its return value.

   (c) Call *close*() and check its return value.

6. *link*() 's the file to new/time.pid.host.

After step 6, the message has been successfully delivered. The delivery program is required to start a 24-hour timer before step 4, and to abort the delivery if the timer expires. Upon error, timeout, or normal completion, the delivery program may attempt to *unlink*() `tmp/time.pid.host`.

Typically the mail delivery system is a finite state machine. The key issue about both mail queue and mailbox management is that the mail delivery process follows a path to reach final state and therefore ensure reliable queuing or reliable delivery. Along with this 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 use of permanent file system storage between different states acts as a checkpoint for graceful recovery. This is an example of the Checkpointed System [6] Pattern.

The reason why the "mailbox" or "mbox" approach failed is they did not retain the atomicity in writing messages. File writing or appending is unreliable because the underlying file system does not provide atomic locking mechanism. However, the creation of a file is an atomic process enforced by the file system. Creating a new file is also in essence a write in the file system except for the fact the content is written in the directory hierarchy (the buffer cache). Therefore, when a new, unique file is created under a directory a lock on the directory is placed. NFS makes sure

Table 3.5: Reliability Pattern - Checkpointed System

Problem
A component failure can result in loss or corruption of state information maintained by the failed component. How can we design a system so that its state can be recovered and restored to a known valid state in case a component fails?

Solution
Design the system as a finite state machine. Make the state information persistent. Use a wide variety of configurations that provide the ability to restart the system from a known valid state, either on the same platform or different platforms.

that this locking and file creation under a directory is an atomic process. This mechanism is tidier than placing a lock on a file and ensuring the commit job is atomic.

3.2.6 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 it takes several seconds to decide that an MTA is not available. Therefore, an MTA will be multithreaded so it can send many messages at once and not allow unavailable MTAs to block delivery of mail to available ones.

Table 3.6: Performance Pattern - Small Processes

Problem
A program memory processes can be limited by the memory used by the processes. If the processes grow unbounded, then there is a potential DoS scenario. How can a program with many processes be made safe from resource exhaustion?

Solution
Make the processes small. Each process should perform one task. This will ensure that processes allocate limited memory.

This is called Small Processes by [18].

qmail has a small amount of multithreading just because *qmail-smtpd* and *qmail-send* run as separate processes. Moreover, *qmail-lspawn* and *qmail-rspawn* also run as separate 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. *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.

3.2.7 Flexibility

Although SMTP is the most common mail transfer protocol, it is not the only one. *qmail* supports two other mail transfer protocols, QMTP and QMQP, which are faster than SMTP. For each protocol, there will be a server that runs along with *qmail-smtpd* to deliver mail to *qmail-queue*. 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.

In contrast, a new protocol requires changing *qmail-remote*, because it has to decide which protocol to use.

3.2.8 Safe Data Structure

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 instead implements all of the libraries that it uses. This eliminates the chance that a defective library could introduce an error.

For example, 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". A way to avoid this problem is to dynamically allocate strings. However, that approach is vulnerable to DoS attacks. Details about buffer overflow attacks can be found in [1].

The string library, rewritten in *qmail*, eliminates buffer overruns. *qmail* strings are not null-terminated. They are encapsulated in a struct type 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 enough 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.

Table 3.7: Security Pattern - Safe Data Structure

Problem
Buffer overflow is a security threat that occurs from bad programming practice. If every string handling routine checked allocated memory and validated input beforehand, buffer overflow would not occur. However, in practice, they are not written to be safe. How can string routines be made safe from buffer overflow attacks?

Solution
Represent strings with data structure that includes length information and allocated memory information. All string routines should check for length and memory available before proceeding.

This pattern is called Safe Data Structure by [18].

Handling string functions at this level creates the last line of defense of security. This is an instance of Defense in Depth.

In a language with garbage collection and bounds checking like JAVA and Smalltalk, string

copy does not pose a problem. C needs string that carry information about its length and memory because the compiler does not check array bounds. This is an instance of the Safe Data Structure pattern.

3.2.9 Content Independent Processing

Some security breaches are caused by maliciously used features. *sendmail* treats programs and files as addresses. The situation is aggravated because *sendmail* runs as root. *sendmail* tries to prevent this by trying define policies in order to keep track of whether some local user was responsible for an address. But that never works out right because of the complexity it offers.

In *qmail*, programs and files are not addresses. *qmail-local* can run programs or write to files as directed by the ".qmail" configuration file in the local directory of the user. The local user is always less privileged than a root user. This follows the Content-Independent Processing pattern.

Table 3.8: Security Pattern - Content Dependent Processing

Problem
In an MTA, the body of a message should not be used for any other purposes. If message can be sent to files and programs, and the files are overwritten by message content or the programs execute with message content as parameter, then an abuser can send messages with malicious content to utilize this feature for his benefit. How can a mail program be made secure so that the message content cannot be used maliciously?

Solution
Treat the received contents as mail message only and do not perform any processing on them. Even when treating programs and files as addresses, minimize the impact by using less-privileged user to execute.

This patterns is called Content Independent Processing in [18].

3.2.10 Trust Partitioning

The fact that *qmail* refuses to send mails to programs and files manifests that *qmail* does not trust the message contents. This is important for security. Even tighter security is achieved by making the components in such manner that they do not trust the communication payload sent among

themselves. This ensures that a compromise of one component of the system remains limited in that section only.

In *qmail*, separate functions have been moved into mutually untrusting programs. *qmail* modules can run as separate programs and trust can easily be distributed among these programs. Trust partitioning is achieved by running these programs under a set of *qmail*-specific UIDs, so compartmentalizing their access. These programs always validate other program inputs before operating on them. The data sent between programs do not constitute special data structures but handles to physical resources containing them. The five separate user ids have control over different components and because they have limited privilege the effects of a compromise are limited. This is an example of the Trust Partitioning pattern [18].

Table 3.9: Security Pattern - Trust Partitioning

Problem In a system architecture that is compartmentalized, an intruder may get hold of a compartment with super user rights or a compartment with maximum responsibilities and crash the whole system. Many security violations are possible, because system developers extend trust unnecessarily. How can security be ensured even after some part of the program is compromised?
Solution Assign minimum privilege level to components. Classify the owners of processes into different trusted and un-trusted groups. Design the components to not trust inputs from other groups and to validate inputs.

3.3 Conclusion

One of the main reasons given for monolithic architectures is efficiency. Partitioning a system can result in inefficiencies in both time and space. It is easier to eliminate duplication when the entire program is in one place. However, *qmail* is smaller and more efficient than *sendmail*, in spite of being more modular.

sendmail is made up of a single Unix program and consists (*sendmail 8.11.7*) of 67936 lines in .c files and 5378 lines in .h files. *qmail* is made up of 24 separate Unix programs and consists (*qmail 1.03*) of 15542 total lines in .c files and 1075 lines in .h files. Thus, *sendmail* is about four

times larger than *qmail*. Moreover, *qmail* does not reuse any Unix libraries, while *sendmail* reuses the standard IO libraries. Although *qmail* does not implement every feature of *sendmail*, it supports some features (such as extended protocols support for QMTP and QMQP, support for mailing list using Variable Envelope Return Paths etc.) that *sendmail* does not. Thus, the *qmail* architecture allows it to be one fourth the size of *sendmail* with similar number of features.

Another advantage of the *qmail* architecture is that it is made from components that are small and easy to understand. The largest module of *qmail* is *qmail-send*, which is 1612 lines. All the rest are less than 800 lines. Except for *qmail-send*, we were able to understand each component in no more than a couple of hours. Understandability is especially important for security.

The *qmail* architecture provides an outstanding level of security by using compartmentalization and distributed delegation to minimize the danger of security holes and by using simplicity and coding standards to eliminate security holes. This architecture leads to a MTA that is efficient and robust. The MTAs that were written after *qmail* were greatly influenced by the philosophy of this architecture. The following chapters on Postfix and *sendmail X* are great examples of this thesis.

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.

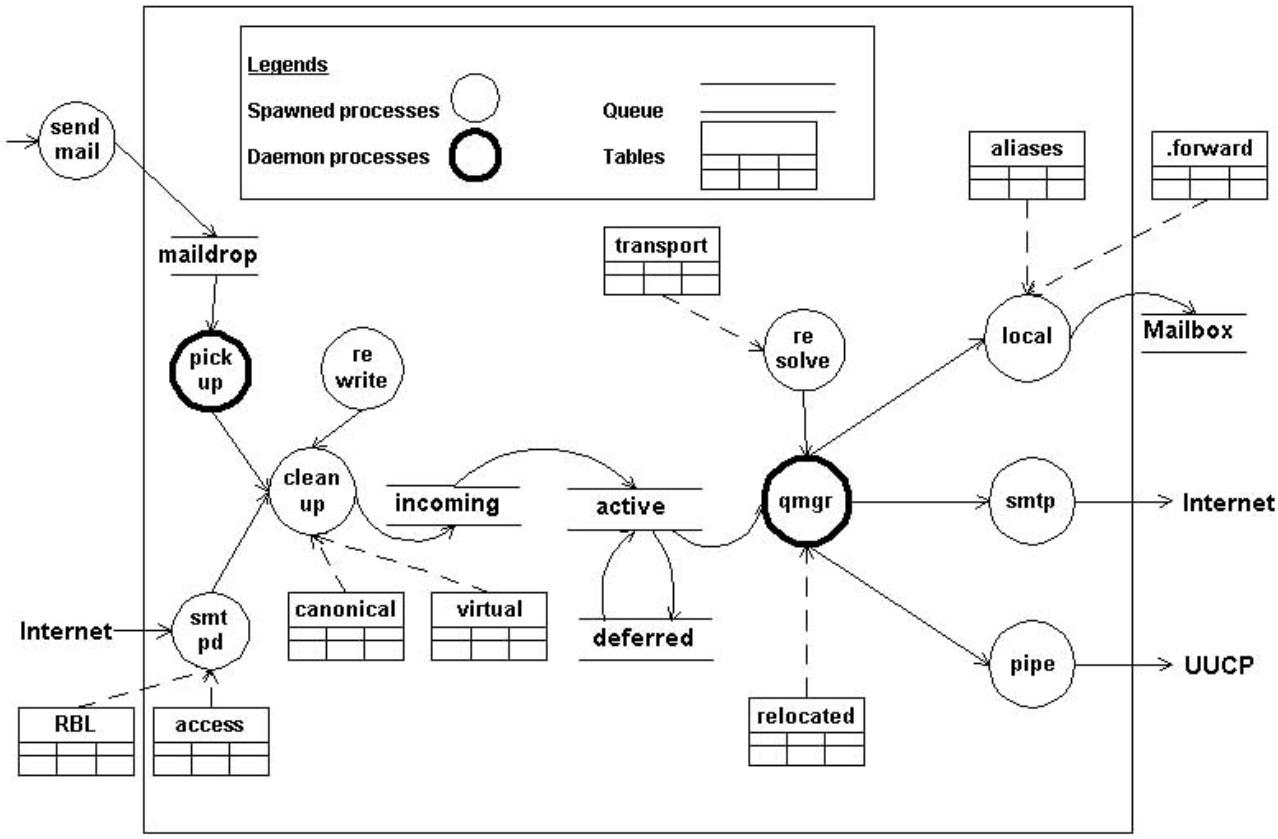

Figure 4.1: Postfix Architecture

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.

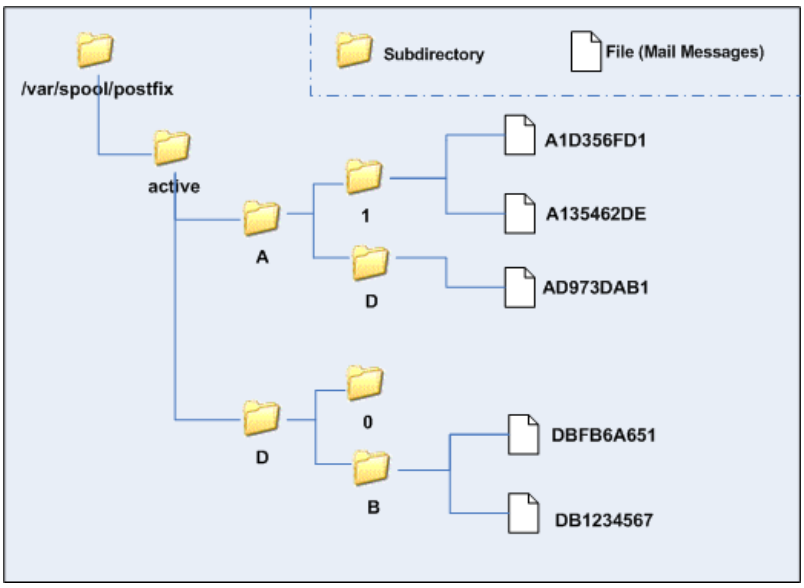

Figure 4.2: Postfix queue in underlying file system

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.

Chapter 5

sendmail X ARCHITECTURE

The old *sendmail* architecture has a long history of security compromises [5] [23]. *qmail* and Postfix were written to provide a simpler and more secure alternative to *sendmail* . *sendmail X* is targeted to provide a *sendmail* system which has all the required architectural features like security and reliability. The development of *sendmail X* is still under way. This chapter describes the architecture of *sendmail X* from its design documents [4].

5.1 History of *sendmail X*

Eric Allman, the author of *sendmail* , had plans to write a new version of *sendmail* with a modified architecture and fresh architectural perspectives to satisfy emerging architectural requirements like security and reliability [13].

This new generation of *sendmail* is called *sendmail X* . Initially it was called *sendmail* 9. *sendmail X* is a completely new design and is not an evolution of previous versions of *sendmail* (*sendmail* version 8).

sendmail X is still under development. Most of the design is completed, but there are still some unresolved issues [4]. The development version of *sendmail X* has been running since January 2004 in production mode on two mail servers. It was made available for Alpha testing with the release of version smX-0.0.Alpha2.0 on May 31, 2005.

5.2 *sendmail X* Architecture

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 [29] 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 the 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 5.1 shows the major *sendmail X* processes.

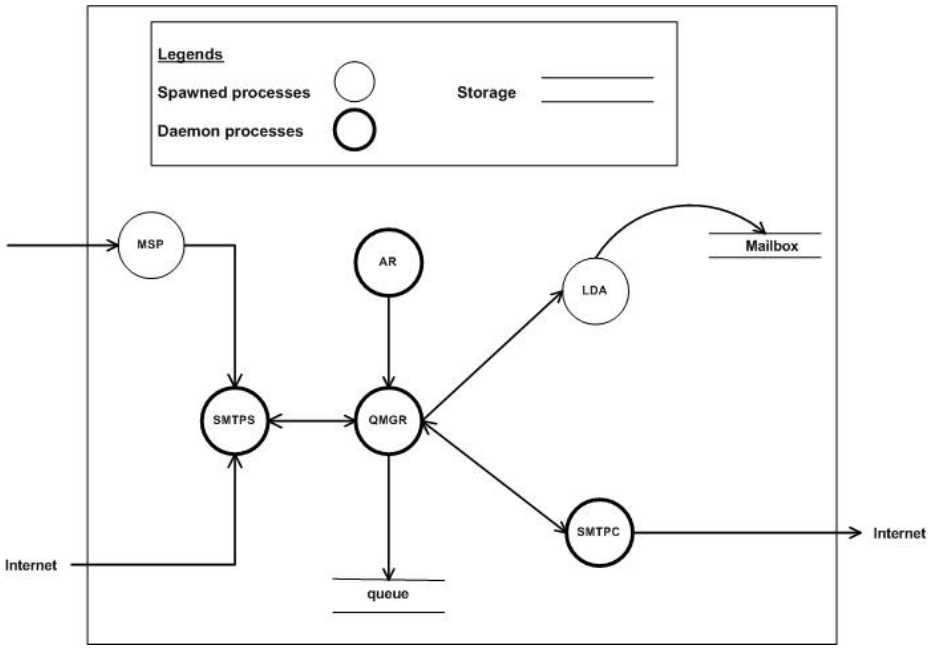

Figure 5.1: *sendmail X* Architecture

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

Incoming messages arrive via SMTP. The connection attempts are received by the SMTPS (SMTP Server). SMTPS communicates with the QMGR (Queue Manager). These two collab-

oratively decide on whether to accept or reject the message. Local mail sending is done by the command line message submission tool called the MSP (Message Submission Program). For each transaction, a new envelope is created and the commands are communicated to QMGR and AR (Address Resolver) for validation and information. Other processes can be involved too (through an API called milters) and the commands are either accepted or rejected based on the feedback of all the processes involved. When the email is received it is either persisted or an appropriate delivery agent is informed to deliver it immediately. The acknowledgement to remote sender is sent after successful termination of either of these two actions. The process that schedules delivery is QMGR. Local Delivery is handled by LDA (Local Delivery Agents) and remote delivery is handled by SMTPC (SMTP Client). Delivery scheduling is done per recipient and not per envelope as previous *sendmail* versions did. This makes it simpler to reuse open connections. Delivery agents receive sender and recipient information from QMGR. It informs QMGR the delivery status and QMGR updates the EDB (Endelope DataBase) accordingly. When mail is delivered to all recipients, the corresponding data is removed from EDB and the content is removed from CDB.

5.2.1 Supervisor Process

Some of the tasks require root privileges (like port binding), and the processes performing them have to have root privilege, or they dynamically upgrade their privilege level. *sendmail X* does nar ot have any program running as *setuid* root for security reasons. MCP performs these tasks.

MCP starts and supervises all the processes. It deals with failures like crashes either by restarting or by logging and reporting. MCP also performs the shutdown.

The configuration of MCP is similar to the *master* process in postfix. The configuration file specifies the user id/group id of each started process. It also has configuration of failure actions. This is similar to *inetd*, but processes are started at startup rather than on demand.

MCP runs as root and therefore it has to be carefully written. Input from other sources must be carefully examined for security implications.

MCP doesn't have an external interface, except for starting/stopping it. Reconfiguration is done via stop/start. Future versions would have the feature of enabling/disabling parts of *sendmail X* by sending it HUP signal.

The supervisor starts and controls several processes. As such, it has a control connection with them. In the simplest case these are just *fork* or *wait* system calls, in more elaborate case it may be a socket over which status and control commands are exchanged.

5.2.2 Mail Submission

Mail submission is done by two processes. Local mail submission is done by the command line process called MSP. Remote mail submission is done by SMTPS.

The basic control flow of an incoming SMTP connection is described here.

1. SMTPS receives an incoming connection attempt and contacts QMGR with that information. QMGR decides whether to accept or reject the connection and returns a session id.

2. After the connection has been made, SMTPS sends the list of features returned from QMGR.

3. If SMTP commands are used that change the status of a session (e.g. STARTTLS, AUTH), those are executed by SMTPS and their effects are stored in the session context. The data is also sent to QMGR.

4. For each transaction SMTPS creates a new envelope and communicates the commands to QMGR and AR for validation and information. Other processes might be involved too and the commands are either accepted or rejected based on the feedback from all involved processes. *sendmail* v7 introduced use of external processes through an API called milters for validation and filtering. *sendmail X* also uses milters API.

The SMTP server must bind to port 25, which is done by MCP before the server is actually started. *sendmail* 8 closes the socket if the system becomes overloaded which requires it to be reopened later on, which in turn requires root privileges again.

SMTPS may need access to an authentication database, which contains secret information (e.g. passwords). Access to this information is usually restricted to the root user. To minimize the exposure of the root account, access to this data should be done via daemons, which are contacted via protected communication means, e.g., local socket, message queues.

In some cases it might be sufficient to make secret information only available to the user id under which the SMTP server is running, e.g., the secret key for a TLS certificate. This is especially true if the information is only needed by the SMTP server and not shared with other programs.

5.2.3 Queue Management

QMGR controls the flow of email throughout the system. It implements the policies (with the help of AR) and it coordinates sending and receiving processes.

QMGR handles several queues. The queues of *sendmail X* are similar to the queues of Postfix. There are two types of queue. The contents of mail message are stored in the CDB (Content Data Base) queue. The envelope routing related information is kept in the EDB (Envelope Data Base) queue. There are different EDB queues listed in the design documents.

- ***incoming.*** This queue stores the envelope data of currently incoming mails. This memory-resident queue is called IQDB, the backed-up version is called IBDB and both together are referred to as INCEDB.
- ***active.*** The primary task of this queue is to schedule deliveries. This queue is also called AQ or ACTEDB.
- ***deferred.*** This queue is also called the *delayed* queue or DEFEDB. It stores deliveries with temporary problems.
- ***bounce.*** This queue stores information about permanently failed delivery attempts.
- ***hold.*** This queue stores envelope data based on policy decisions.
- ***ETRN.*** This stores envelopes that are only scheduled on ETRN requests.
- ***corrupt.*** This stores envelope that are somehow corrupted.

The *deferred* queue is the main queue and it subsumes the *hold*, *ETRN* and the *bounce* queue. That way the status of all the messages that are stored for later delivery are available in one place. To achieve the effect of having different queues it might be sufficient to build different indices to access parts of the queue.

The *corrupt* queue is technically different because it stores data that will no longer be scheduled for delivery. Current implementation does not have this queue.

The *active* and *incoming* queues are memory resident. The *incoming* queue is backed up on a stable storage in a form that allows faster modifications. The *active* queue is not directly backed up on disk, other queues collaborate to act as backup for this. The *active* queue is a restricted size cache of entries in other queues. The *deferred* queue stores items that are removed from *active* queue for various reasons, like policy, delivery at a certain time, quarantine due to milter feedback, delays or as a result of failed delivery attempts. An envelope must be either in the *incoming* queue or in the *deferred* queue at any given time.

The implementation of *sendmail X* has three queues for envelope data - the *active* and *incoming* queue in memory and the *main* or *delayed* queue in which envelope data is stored which refers to recipients that could have not been delivered due to problems or queueing.

SMTPS stores envelope data in IQDB via QMGR. SMTPS also stores the contents in the CDB. SMTPS is the only process that has write privileges on CDB files. The envelope recipients are stored in IQDB and IBDB. The CDB information is stored in the *incoming* queue (IQDB) when the transaction is closed; this also causes the transaction data to be written to IBDB. Before mail reception is acknowledged the entire transaction data is committed to IBDB and appropriate function like *fsync* is called to make sure the data is safely written to persistent storage.

Data comes into the *active* queue from two sources.

1. Data in IQDB is copied into the *active* queue (AQ) when an SMTP server transaction is closed.

2. Entries are read from the *deferred* queue (DEFEDB) based on some criteria determined by the scheduler.

QMGR runs as an un-privileged user. The communication channels between the various modules (esp. between the QMGR and other modules) must be protected. Even if they are compromised, the worst that is allowed to happen is a local DoS attack and the loss of e-mail. Getting write access to the communication channel must not result in enhanced privileges. There is one possible way to protect the communication even if an attacker can get write access to them and that is by the

use of cryptography. However, this is most likely not worth the overhead. It could be considered in design if the communication is done over otherwise unsecured channels.

5.2.4 Address Resolution

The address resolver (AR) plays an important role in the architecture. It rewrites the addresses into a canonical form and determines the delivery information for a recipient address. These two tasks are performed respectively by the Postfix processes *trivial-rewrite* and *resolve*. AR might also perform anti-spam checks.

The amount of data the AR has to return might be rather large, at least if it is used to expand aliases. Using IPC for that could cause a significant slowdown compared to intra-process communication. So maybe the AR should be a library that is linked into QMGR. However, the AR may need to be re-used by other modules and in that case that would cause a security flaw, as that module would also run with same privileges like QMGR, because AR is running with QMGR privileges. Again, AR performs blocking calls and is therefore unusable with QMGR.

There are advantages in separation of routing and address rewriting tasks. Postfix does that by separating two processes *trivial-rewrite* and *resolve*. However, in many cases both effects (routing and rewriting) are required together and one process is sufficient. *sendmail X* follows this perspective.

5.2.5 Mail Delivery

There are several types of mail delivery agents in *sendmail X* similar to *sendmail* version 8. One of them acts as SMTP client. Another important one is the local delivery agent.

Instead of having a fixed set of delivery agents and an address resolver that knows about all of them, the delivery agents are provided as modules, which provide their own address rewriting functions. This follows the approach of Exim and Courier-MTA. These are called in some specified order and the first that returns that it can handle the address will be selected for delivery.

sendmail 8 used a centralized approach. All delivery agents must be specified in the .cf file and the address rewriting must select the appropriate delivery agent.

sendmail X provides a simpler way to add custom delivery agents and to select them. It seems

best to hook them into AR, that's the module that selects a delivery agent.

A local delivery agent must run as a non-privileged user. It needs to change its user id to that of the recipient. *sendmail X* does not have a setuid root program.

Alternatively, *sendmail X* considers using a group-writable mailstore can be used as it is done in most System V based Unix systems. A unique group id must be chosen that is only used by the local delivery agent. It must not be shared with MUAs as it is done in some OSs. There is one problem with this approach. A user mailbox must exist before the first delivery can be performed. That requires that the mailbox is created when the user account is created and no MUA must remove the mailbox when it is empty. There could be a helper program that creates an empty mailbox for a user, which however must run as root and hence will have security implications.

Another option is to use a daemon process. *sendmail X* runs as a daemon and speaks LMTP. By default, it uses root privileges and changes its user id to that of a recipient before writing to a mailbox.

The LDA does not deliver mail to a mailbox owned by root. This is because such a mail would require the LDA to assume root privilege for writing in a directory owned by the root. Instead there is an alias that redirects mail to another account. The LDA should also not read files owned by root.

Remote mail sending is handled by SMTPC. Similar to SMTPS there are several architectures possible for SMTPC for example pre-forked processes, multi-threaded, event-driven, or using state-threads. The approach that the writers of *sendmail X* are taking is to write simple prototypes to find a suitable design. However, the SMTPS and SMTPC are planned to have same implementation alternative to make the implementation process easier.

There are basically two different situations for a SMTPC. Open a new connection (new session) or reuse an existing connection (new transaction).

The steps of opening a new connection are.

1. The SMTPC receives a connection request from the QMGR containing the required information. If the connection attempt fails, it informs QMGR.

2. SMTPC sends EHLO (or HELO) message and reads list of features as returned from the server. It checks whether required features are supported and if it is unsupported it informs

QMGR. The QMGR may decide to use the connection for something else, otherwise the connection will be closed.

3. QMGR sends request to end a session. SMTPC sends QUIT message to terminate the session.

For each transaction the envelope information and the identifier for the CDB is received from the QMGR. The SMTPC sends MAIL, RCPT, and DATA as allowed by the SMTPS and reads the replies. If a command fails, the QMGR is informed about the problem. If the transaction is done, the QMGR is informed about the status returned by the server in response to the final dot. Finally the SMTPC cleans up the CDB.

SMTPC runs without root privileges. It needs only access to the body of an e-mail that it is supposed to deliver. However, it may need access to authentication data. For these reasons an SMTPC uses a different user id than other *sendmail X* programs, and achieves access to shared data (mail body, interprocess communication) via group rights.

5.2.6 Security

One of the primary goals of *sendmail X* is to be more secure than the old *sendmail* . Many architectural decisions in *sendmail X* design were taken because of this. *qmail* and Postfix were written to provide a secure alternative to *sendmail* . The design of *sendmail X* has a lot of design decisions that were proven to be effective in the architecture of *qmail* and Postfix.

The primary design decision of *sendmail X* is to revert from the old *sendmail* design of a single root process and compartmentalize the modules. This provides the insulation of processes. Therefore compromise of one process does not affect other parts. The modules run with a lower privilege level and this limits the exploits of the attacker. It has not yet been decided whether the initial mail submission program will be *setgid* . No program in *sendmail X* runs as *setuid* root. Root owned files are not written to. For example, mail is not sent to the mailbox that belongs to the root. Reading from files that are owned and accessible by root only is also avoided.

A major problem of old *sendmail* was the complexity of the configuration file. *sendmail X* configuration would be defined by a new configuration language. The primary goal for designing this language is simplicity and lack of ambiguity. However, one problem with this language is it is designed to be too feature-oriented. This falls back to the case of old *sendmail* where the main

problem was the flexibility. One of the primary goals of *sendmail* was flexibility, it incorporated a lot of features that were used by few people. That added to the complexity of configuration and many security holes resulted from configuration scripts that failed to take into account all the usage scenarios.

sendmail X uses new string routines that follow the safe string pattern [18]. This is to prevent the buffer overflow attacks that occur because of unsafe string routines. The string buffers are defined by a structure called *sm_str_S* . The buffer stores both data and length information in the data structure. The members of *sm_str_S* include,

sm_str_base . This field is an unsigned character pointer representing the data. The string buffers do not have to be null terminated.

sm_str_size . This field denotes the total bytes allocated for the buffer. It is of type *size_t* .

sm_str_len . This is a *size_t* type data structure representing total number if characters in the buffer.

sm_str_max . This specifies the maximum number of characters in buffer. It is of type *size_t* .

sm_str_rpool . This is an *sm_rpool_P* type data structure denoting the resource pool to allocate from.

Safe routines are written to make the string operations safe. For example, the basic routine for string copying is *sm_str_cpy* and the basic routine for string concatenation is *sm_str_cat* . This follows the defense in depth principle by providing protection at multiple levels of architecture.

5.2.7 Reliability

The mail queue of *sendmail X* is split into directories like the mail queue of *qmail* and Postfix. The mail delivery process is implemented as a state machine and information of all the states are stored explicitly. This is an example of the Checkpointed system pattern [6].

SMTPS writes into the CDB queues. There are multiple SMTP processes running at one time. To ensure separation of entries, the message contents are written under different name space. This follows the Unique Atomic Chunks pattern [17].

QMGR handles the EDB queues. The *active* queue is the most delicate because it is not persisted. However, entries in the *active* queue are formulated from the entries in *incoming* and *deferred* queue. So the worst case scenario for a crash would be that a mail is delivered twice, which is not very bad.

5.2.8 Performance

SMTPS is implemented as an event-driven state machine. Pipelining is implemented in SMTPS for better performance. In *sendmail* version 8, the SMTP requests are handled sequentially and I/O buffering is used to emulate pipelining. *sendmail X* processes several SMTP commands concurrently. This speeds up the overall SMTP dialogue.

The SMTP servers should have load control features. These features are triggered by request form the QMGR process. The rejection of requests should be done gradually in various steps. The worst case scenario is the rejection of all incoming connections.

QMGR is like a pipeline between sending and receiving modules. This gives an additional responsibility that it does not slow the system down. QMGR is a multi-threaded program to allow scalability and fast response.

Having a number of queues has the following advantages. Fewer entries provide provision for faster access. So it increases performance. Again, special storage of ETRN and hold requests ensures that they are not retried for schedule at every scan. Different disks can be used for different queues. A slow disk can be used for ETRN or hold queues, whereas a faster disks can be used for persisting incoming queue. This ensures efficient usage of resources.

However, in *sendmail X* design the QMGR program is the single interface to the mail queue and this might cause performance problems. In *qmail* the queue is handled by three processes - *qmailqueue* puts mail in queue, *qmail* send gets mail form the queue and *qmail* clean performs cleanup of the queue. In Postfix, the *cleanup* program is responsible for putting mails in the queue and the *qmgr* program drains the queue. In *sendmail X* , QMGR puts mails in the queue, schedules and moves mails between queue directories and also gets mails off the queue. It also communicates with AR process for format checking. The queue has to be persisted in disks to ensure reliability and this is a major overhead. Because the QMGR has to deal with a lot of things, it can become

a performance choke point. However, result shows, that persistence limits the performance of the overall mail delivery and the QMGR process does not cause performance overhead.

When data is sent to AR from QMGR for address formatting, it also is potential to performance overhead. However, performance tests show that a normal machine can perform about 10000 communications per second. That's at least one order of magnitude (usually two) faster than the entire system can process mail. Hence, at current processing speed, this issue does not become an overhead.

SMTPS checks for valid local recipients. If the checking is delegated to the local delivery agents, then the junk mail would only be discarded during local delivery and it would sit in the system causing overhead. Again, this might result in double bounces and may clog up the system.

The AR module is used for address re-writing and formatting related tasks. The advantage of doing these tasks up-front is if the mal-formed mails are identified and deleted before they are stored in the queue, then it adds to the overall performance of the system.

5.2.9 Spam Handling Policy

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. If the anti-spam checks are done by an outside module, all these data need to be sent to it. However, most anti-spam checks are blocking calls and it is undesirable to block SMTP server. It might be interesting to parallelize the requests by starting several requests and collect the data later on. A clean API has to be defined and it may be available as library which can be linked into the SMTPS or the AR or another module.

sendmail X provides support for additional third-party programs to access mail messages and examine and modify the content and meta level information. The Content Management API of *sendmail X* is called milter. The configuration file contains specification of policies and these policies are implemented by filters that are conforming to milter configuration.

The milter API is designed such that the the filter processes do not need to run as root. Filters run as separate processes, outside of the sendmail address space. The milter API provides separation of filter processes in such a way that the inner processes are not coupled to filter processes and an

error in a milter process does not impact the functionality of any of the *sendmail X* processes.

The milter library acts as the bridge between MTAs and filters. It accepts connections from various MTAs, passes the relevant data to the filter through callbacks, then makes appropriate responses based on return codes.

5.3 Conclusion

sendmail X is an improvement of the old *sendmail* architecture. The main problem of the old *sendmail* is the single process architecture. *qmail* and Postfix improved the MTA architectural viewpoint. A lot of the design decisions in those MTAs are instances of security patterns. *sendmail X* architecture can be attributed as a lesson learned from the mistakes of previous architecture and from the best practices of peers in the community.

Chapter 6

Conclusion

The architecture of Mail Transfer Agents have been influenced by security. The evolution of *sendmail* architecture is a good reference to track the evolution of MTA architecture. The architecture of *qmail* was inspired by the poor security record of *sendmail* . Postfix was written to get better performance without compromising security. Finally, in *sendmail X* architecture, the evolution goes full circle by adopting the best practices in *qmail* and Postfix.

The goal of making a software secure can be better achieved by making the design simple and easier to understand and verify. Tony Hoare said in his Turing Award Lecture in 1980, "I conclude that there are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies." A perfect example is the contrast between the feature envy early *sendmail* architecture implemented as one process and the simple, modular architecture of *qmail*. The security of *qmail* comes from its compartmentalized simple processes that perform one task only and are therefore testable for security. However, simplicity is a difficult goal. Even in *qmail*, *qmail-send* has been implemented as a complex process because it does more than one task. In Postfix the similar module (the *qmgr* daemon) is made simpler by re-factoring the address rewriting tasks in another module (the *trivial-rewrite/resolve* module).

The architectural quality requirements are not always orthogonal and the final design decision is a tradeoff between the requirements [19]. Again, 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 of *qmail* and Postfix.

Another major idea is that security cannot be retrofitted into an architecture. The problems of

old *sendmail* could not be solved by writing patches as vulnerabilities were found in the attempted fixes. An absolute design overhaul was needed and *qmail* is the result of a security up-front design approach. For legacy software, however, security is often retrofitted. Nevertheless, the key thing to remember while designing some secure architecture is to push for inclusion of security considerations in the early design phase.

The early *sendmail* architecture was perfect in its time. As requirements changed and security became important, the problems of *sendmail* architecture were outlined by the constant detection and fix of the vulnerabilities. The future MTAs, similarly, might face some new requirements that would become important eventually. These evolving requirements would require either a new design approach or adaptation of the existing design. Currently, one of the major threats in public email system is handling spam and phishing type attacks. *qmail* architecture does not have spam handling support built in it, it was added as separate feature later. This is because in 1997, when *qmail* was written, spam handling was not a major issue. Postfix which came later uses internal and external tables to detect and remove spam. Recent mail systems like *sendmail X* has elaborate internal and external mechanisms built in for handling spam. This again illustrates the adaptation of MTA architecture in response to feedback generated by usage scenarios and evolving requirements. Evolution of architecture based on feedbacks is absolutely vital to ensure that it withstand challenges imposed by the new requirements.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, Volume Seven, Issue Forty-Nine. <http://www.phrack.org/phrack/49/P49-14>

[2] Andree, M. (2001). MTA Benchmark. <http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/bench2.html>.

[3] Andree, M. (2002). Postfix vs. *qmail* - Performance. <http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/vsqmail.html>.

[4] Assmann, C. (2004). *sendmail X*: Requirements, Architecture, Functional Specification, Implementation, and Performance. *Sendmail Inc.*, September 2004. <http://www.sendmail.org/~ca/email/sm-X/index.html>.

[5] Bernstein, Daniel J. (1997). *sendmail* Disasters. <http://cr.yp.to/maildisasters/sendmail.html>

[6] Blakley, B. and Heath C. (2004). Security Design Patterns Technical Guide - Version 1. *Open Group (OG), led by Blakley, B. and Heath, C.* <http://www.opengroup.org/security/gsp.htm>.

[7] Blum, R. (2001). Postfix. *Sams Publications*, 1st Edition, May 2001.

[8] BugTraq ID 3377. (2001). *sendmail* Inadequate Privilege Lowering Vulnerability. <http://www.securityfocus.com/bid/3377/info>

[9] BugTraq ID 4822. (2002). *sendmail* File Locking Denial Of Service Vulnerability. <http://www.securityfocus.com/bid/4822/info>

[10] BugTraq ID 7230. (2003). *sendmail* Address Prescan Memory Corruption Vulnerability. <http://www.securityfocus.com/bid/7230/info>

[11] BugTraq ID 7614. (2003). *sendmail* Insecure Temporary File Privilege Escalation Vulnerability. <http://www.securityfocus.com/bid/7614/info>

[12] chroot(). *Unix man pages*. <http://unixhelp.ed.ac.uk/CGI/man-cgi?chroot+2>.

[13] Costales, B. and Allman, E. (2002). *sendmail*. *O'Reilly*, 3rd Edition, December 2002.

[14] Elchin, M.W. and Rochlis, J.A. (1989). With Microscope and Tweezers: An analysis of the Internet Virus of November 1988. *Massachussets Institute of Technology.*

[15] Fernandez, Eduardo B. (2002). Patterns for Operating Systems Access Control. *PLOP 2002*. <http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings>.

[16] Friedl, S. (2002). Go Directly to Jail: Secure Untrusted Applications with Chroot. *Linux Magazine*, December 2002. <http://www.linux-mag.com/2002-12/chroot_01.html>.

[17] Hafiz, M. (2004). Unique Atomic Chunks: A Pattern for Security and Reliability. *PLoP 2004*. <http://hillside.net/plop/2004/papers/mhafiz0/PLoP2004_mhafiz0_0.doc>.

[18] Hafiz, M., Johnson, R., and Afandi, R. (2004). The Security Architecture of qmailn. *PLoP 2004*. <http://hillside.net/plop/2004/papers/mhafiz1/PLoP2004_mhafiz1_0.pdf>.

[19] Kazman, R., Klein, M., and Clements, P. (2000). ATAM: Method for Architecture Evaluation. *Technical Report, CMU/SEI-2000-TR-004*, August 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>

[20] Levine, John R. (2004). *qmail*. *O'Reilly*, 1st edition, February, 2004.

[21] Postfix Home Page. *Maintained by Wietse Zweitze Venema*. <http://www.postfix.org>.

[22] *qmail* Home Page. *Maintained by Daniel Julius Bernstein*. <http://cr.yp.to/qmail.html>.

[23] Security Focus. *sendmail* Vulnerabilities. <http://www.securityfocus.com/bid>.

[24] *sendmail* Home Page. *Maintained by Sendmail Consortium*. <http://www.sendmail.org>.

[25] Sill, D. (2001). The *qmail* Handbook. *Apress*, 2nd edition, October, 2001.

[26] Sill, D. (2004). Life with *qmail*. <http://www.lifewithqmail.org/lwq.html>.

[27] Venema, Wietse Z. Postfix Performance Results. <http://www.porcupine.org/postfix/performance.html>.

[28] Veryard, R. and Ward, A. (2001). Trusting Components and Services. <http://www.antelopes.com/trusting_components.html>.

[29] Viega, J. and McGraw, G. (2002). Building Secure Software - How to Avoid Security Problems the Right Way. *Addison-Wesley*.

[30] Zhong, Q. and Edwards, N. (1998). Security Control for COTS Components. *IEEE Computer* vol. 31, no. 6, pp. 67-73, June 1998.