The Security Architecture of qmail

Munawar Hafiz, Ralph E Johnson, Raja Afandi
University of Illinois at Urbana-Champaign, Department of Computer Science
{mhafiz, rjohnson, afandi}@uiuc.edu

Abstract

The architecture of qmail was carefully designed to overcome the security problems of 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 us make other applications secure.

1. Introduction

Daniel Bernstein designed qmail as a replacement for sendmail with improved security[DB97]. The security of qmail is the result of a set of architectural decisions. The same architecture that makes it secure also makes it more efficient than sendmail and easy to understand and maintain. This paper describes the design of qmail by treating it as a sequence of decisions. [CP93] Many of the design decisions are examples of security patterns. Thus, this paper is another example of how patterns generate architecture. [BJ94]

Security is the main driving force behind the architecture of qmail. qmail is a mail transfer agent (MTA), a computer program that transfers electronic mail messages from one computer to another. 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 must be reliable. It must never lose a message.

An MTA must be efficient in both time and space, because it might process thousands of messages simultaneously.

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

An MTA must support existing mail transfer protocols, but it should be extensible so that it can incorporate future protocols.

Although sendmail's architecture was not considered a problem when it was created, qmail's architecture is clearly superior. sendmail was the program that made SMTP popular, and is still the most widely used MTA. Its security problems are no worse than many other widely used systems, but qmail has had no security defects detected since the release of qmail 1.0 in 1997. Similarly, sendmail is efficient enough for many organizations running large mailing lists, but qmail takes less memory and has several times the throughput of sendmail. Moreover, qmail is smaller, simpler, and easy to extend.

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 [VM02], which means that qmail does not depend on any single idea 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 breakins, and ensures that many kinds of errors are not possible. The module decomposition also makes each module simpler, so it can be inspected for correctness. It makes multiprocessing more efficient. The way that qmail uses the file system makes queing 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 much superior to that of sendmail. Studying the architecture of qmail can help us to make other systems secure, effeicient, and easy to understand.

2. Mail Transfer Agent

A **mail transfer agent** or **MTA** (also commonly called a 'mail server') is a computer program that transfers electronic mail messages from one computer to another. 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. The most popular mail transfer protocol for MTAs is SMTP. The most popular MTA is sendmail. Other Unix compatible and SMTP compatible MTAs are qmail, Postfix, Exim, Courier and ZMailer.

An MTA will accept mail from a local user and deliver it either locally or by using SMTP to transfer it to another MTA. It must also accept e-mail from another MTA for local delivery or to relay to a third MTA. Thus, an MTA has at least two ways to accept input and two ways to handle output, as shown in Figure 1, the context diagram of a leveled Data Flow Diagram (DFD).

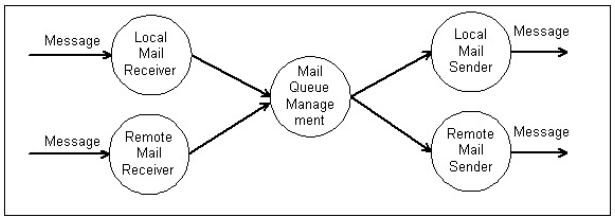

Figure 1: An Operational MTA

3. The mail queue

The structure of an MTA follows its context closely. There will be a process that handles messages from a MUA (which in qmail is qmail-inject), a process that handles messages from another MTA (qmail-smptd), a process that sends mail to a local user (qmail-local), and a process that sends mail to a remote user (qmail-remote). However, figure 1 does not indicate the need for a mail queue, which turns out to have a big impact on the design of qmail.

Post offices do not deliver mail as soon as it is dropped in the mailbox. 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.

It would be possible to design an MTA to have a single process for the mail queue. However, qmail has two processes, qmail-queue and qmail-send. The first is responsible for placing mail in the queue, and the second is responsible for taking it out of the queue and sending it. This leads to the design of figure 2.

4. Security problems

A variety of security problems have plagued sendmail. One of the most common targets of attack is the SMTP server, since it is the main interface to users outside the host. Even though Figure 2 might make it seem that an MTA is a collection of independent processes, sendmail implements them all with a single Unix program. The "processes" share an address space. This makes communication between the processes efficient, but means that once someone breaks into the SMTP server, they have all the powers of sendmail.

One solution is to put the SMTP server 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 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 was also done in a redesign of sendmail by Zhong and Edwards[ZE98] and in Postfix[].

This is an example of the first security pattern, compartmentalization. The name comes from Viega and McGraw [VM02], but the pattern has been described by many people, such as the Execution Domain pattern by Fernandez [F02].

Security Pattern: Compartmentalization [VM02]

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.

5. Distributed delegation

Another problem with sendmail is that it runs as super-user. A 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 superuser. sendmail runs as a single super-user process. When it writes a file, it checks to make sure that it is not abusing its priviledges. 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.

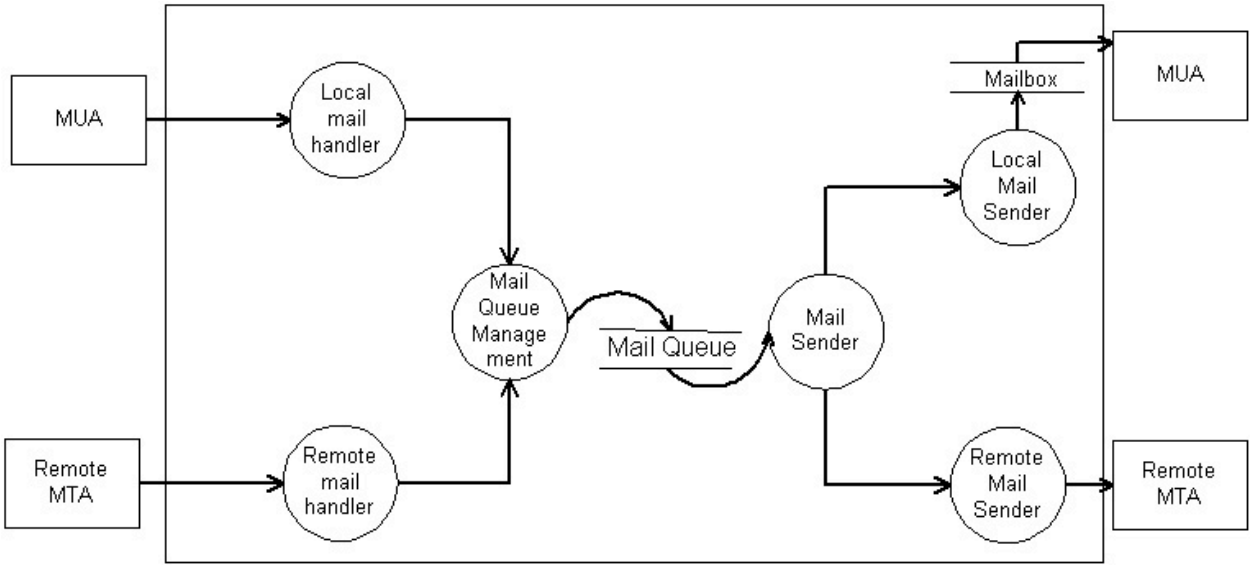

Figure 2: An MTA with a mail queue

Security Pattern: Distributed Responsibility

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 Varyard and Ward. [VW01]

Following are the qmail programs and their corresponding user groups:

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

6. Interfaces

The most common way for a Unix programmer to interpret a flow between two DFD processes is as a pipe between two Unix processes. Almost none of the flows in Figure 3 should be interpreted in this way. The interfaces to most of the qmail programs are idiosyncratic and do not follow Unix standards.

For example, qmail-lspawn, qmail-rspawn and qmail-clean both take one input pipe and one output pipe, but both of them go to qmail-send. qmail-send uses one pipe to send commands to lspawn or rspawn, and the other 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 streams. 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.

A common reason to make something a separate Unix process is reuse. 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.

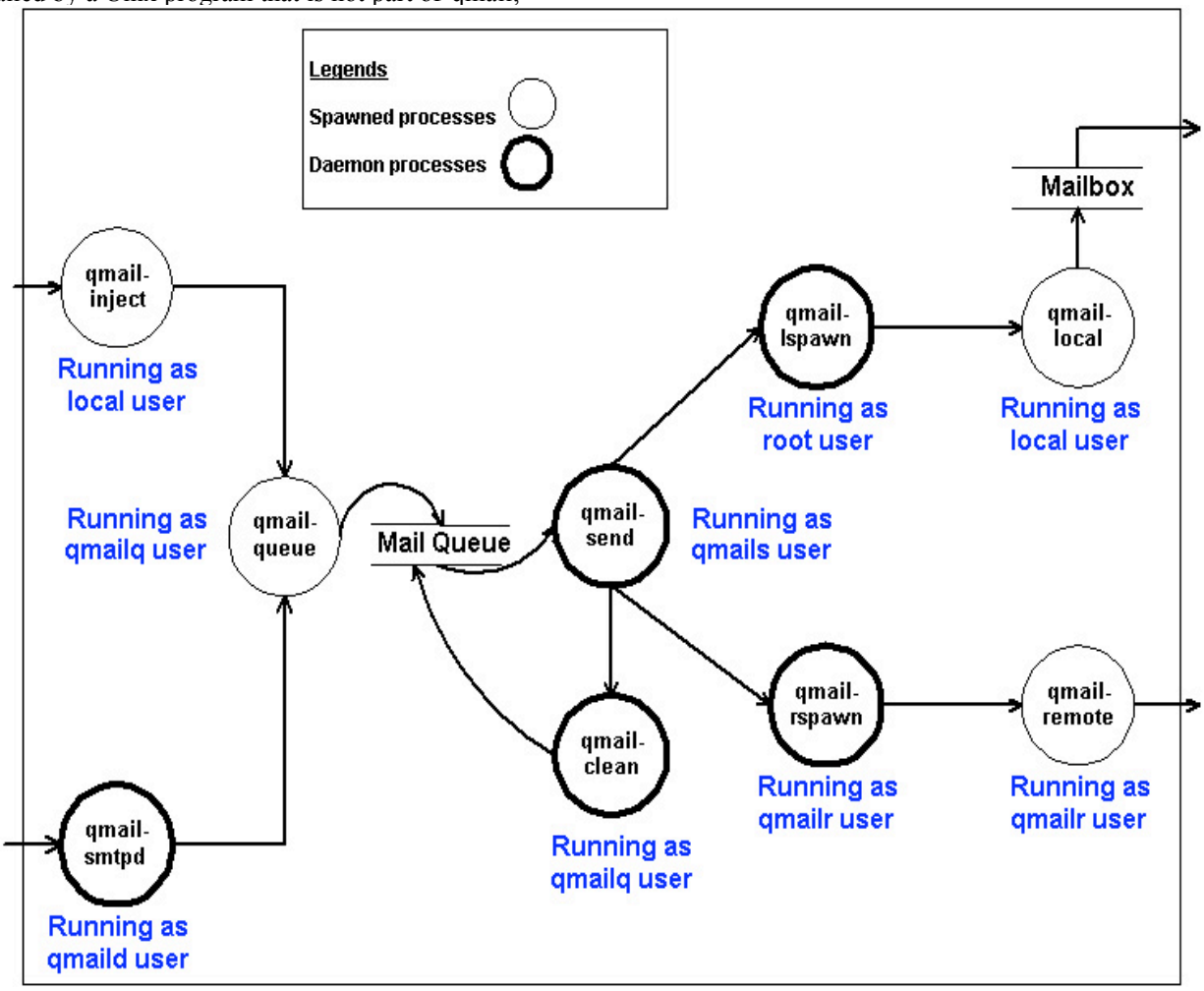

Figure 3: Major qmail processes

7. Reliable Mail Queuing

The mail queue is in the center of qmail. The reliability of qmail depends on the mail queue being reliable. 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-inect) 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 key to ensuring that one qmail-inject process does not interfere with another is for each 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.

Reliability Pattern: Unique Entry of Information

Problem
Many proesses 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
Create unique entries for each write request. Thus, different processes are never writing on the same file at the same time.

The key to ensuring that messages are not lost is to keep track of the various states of a message. In particular, a message is created by writing the message in one directory and the envelope in another. The message is scheduled by a program that reads it and decides what should be done with it. If a message is to be delivered locally, its local envelope recipient addresses are written into the mail queue, but if it is to be delivered remotely then its remote envelope recipient addresses are written. Each address is first marked "NOT DONE". Once the message is delivered to the address, the address is marked "DONE". Once all the addresses for a message are marked "DONE", the message has been delivered and so is removed.

qmail's mail queue consists of several directories. qmail-queue creates a message in the "pid" directory with its name being the ID of the process that created it. Then the message is moved to the "mess" directory and its name is changed to the message's i-node number. The envelope for the message is created in the "intd" directory, and when it is finished a link to the envelope is placed in the "todo" directory. At this point, the message is officially in the mail queue.

qmail-send takes messages out of the mail queue and sends them to local and remote destinations. It creates files in the "info", "local" and "remote" directories, all named after the i-node number of the message in the "mess" directory. The file in the local directory contains all the local destinations for the message, while the file in the remote directory contains all the remote destinations.

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

Reliability Pattern: Recoverable Component [BB02]

Problem
How to structure a component so that its state can be recovered and restored in case the component fails?

Solution

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.

Reliability Pattern: Checkpointed System [BB02]

Problem
Can we structure a system that can be "rolled back" to a known valid state when its state becomes corrupt?

Solution
Create a set of states and make the system follow the state sequences in its life cycle. Keep state information all the time.

Reliability Pattern: Hot Standby [BB02]

Problem
Can we structure a system which permits state updates to originate from multiple components, preserves the state of the overall system and of each transaction in the face of failures, and guards against loss of integrity due to incomplete application of transactions or changes?

Solution
Monitor the transactions between different states. Transfer from one state to another must be atomic.

8. 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.

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.

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.

9. Mailbox management

Mailboxes have similar reliability problems as mail queues. It is difficult to modify existing files reliably, portably, and efficiently under Unix. NFS has better control over ensuring atomixity for new file creation than it does for writing to an existing file. So, qmail provided a new mailbox format that is easier to implement reliably under Unix.

The most popular mailbox format used by sendmail and other MTAs is 'mailbox' or 'mbox'. This format uses a single file for all mail received by a user, both old and new. The problem with mbox format is that a crash can truncate the message. Even worse, the truncated message will be joined to the next message.

Another problem with mbox is that there may be two programs simultaneously delivering mail to the same user. The 'mbox' format requires the programs to update a single central file. If the programs do not use some locking mechanism, the central file will be corrupted. Sun's Network File System (NFS), which is typically used in this regard, does not work

reliably. There are other Unix locking mechanisms that do, though they are not as widely available.

The solution to the 'mbox' format and the problems related to it is the 'maildir' format in qmail. This format is now 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) NSF-write the message to the file.

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.

NFS-writing in step five means

(1) check the number of bytes returned from each write() call

(2) call fsync() and check its return value

(3) call close() and check its return value

10. 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.

11. Coding standards

There are a variety of coding standards followed in qmail that reduce the likelihood of security problems. It does not use the standard Unix I/O libraries, but 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.

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.

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. String carrying information about its length and memory allocation is needed in C because of the underlying mechanism of the

compiler. This is an instance of the Safe Data Structure pattern.

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.

12. Content-independent processing

Some security breaches are caused by maliciously used features. 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 [ER89]. sendmail has a feature to send mails to a program, such that the program receiving the mail executes with the body of the mail message as input. This feature is not generally allowed for incoming connections except when the debugging mode is on. Unfortunately, the sendmail program packaged with 4.3BSD and pre-4.1 versions of SunOS had this mode turned on by default. The worm used this feature to connect to a sendmail daemon and send a message to a 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.

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.

Security Pattern: Content–independent 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

Do not treat programs and files as addresses and therefore do not use the message content for anything other than message storage. Minimize the impact by using less-privileged user to execute.

13. 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. We believe that other applications could also make use of this architecture, and are looking for examples.

References

[BB02] Initial Draft of Security Design Patterns, Open Group (OG), led by Bob Blakley, http://www.opengroup.org/security/gsp.htm

[BJ94] Patterns Generate Architecture, Kent Beck, Ralph Johnson, *Lecture Notes in Computer Science, 1994*

[BS00] Secrets and Lies: Digital Security in a Networked World, John Wiley, 2000, by Bruce Schneier

[BY97] Architectural patterns for enabling application security, Joseph Yoder, Jeffrey Barcalow, *PLoP 1997* http://citeseer.nj.nec.com/yoder98architectural.html

[CP93] A Rational Design Process: How and why to fake it, Parnas, D.L., Clements, Paul C., *IEEE Transactions on Software Engineering, vol. 19, no. 2, pp. 251- 257, February 1993*

[DB97] D. J. Bernstein, qmail Author. http://cr.yp.to/djb.html

[DSL] Online article "Life with qmail" by Dave Sill and D. J. Bernstein http://www.lifewithqmail.org/

[DSL01] The qmail Handbook by Dave Sill, *Apress, October 15, 2001*

[F02] Patterns for Operating Systems Access Control, Eduaardo B. Fernandez, *PLOP 2002*. http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings

[SAL75] The protection of information in computer systems, J. H. Saltzer and M. D. Schroeder, *In Proceedings of the IEEE, volume 63(9), pages 1278–1308. IEEE, September 1975*

[VW01] Trusting Components and Services, Richard Veryard and Aidan Ward, http://www.antelopes.com/trusting_components.html

[VM02] Building Secure Software - How to Avoid Security Problems the Right Way, J. Viega and G. McGraw, *Addison-Wesley, September 2002.*

[ZE98] Security Control for COTS Components, Q. Zhong and N. Edwards, IEEE Computer vol. 31, no. 6, pp. 67-73, June 1998.