Some thoughts on security after ten years of qmail 1.0

D. J. Bernstein
University of Illinois at Chicago

The bug-of-the-month club

"Every few months CERT announces Yet Another Security Hole In Sendmail—something that lets local or even remote users take complete control of the machine. I'm sure there are many more holes waiting to be discovered; Sendmail's design means that any minor bug in 41000 lines of code is a major security risk. Other popular mailers, such as Smail, and even mailing-list managers, such as Majordomo, seem just as bad."

Source: qmail docs, 1995.12.

Solution: Write a secure MTA!

1995.12.07 version of qmail:
14903 words of code.

1995.12.21:	36062 words.
1996.01.21, 0.70:	74745 words.
1996.08.01, 0.90:	105044 words.
1997.02.20, 1.00:	117685 words.
1998.06.15, 1.03:	124540 words.
netqmail 1.05:	124911 words.

Total known bugs in the qmail 1.0 releases: 4.

Total known security holes: 0.

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!
qmail: POP support!

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!
qmail: POP support!
Sendmail: UUCP support!

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!
qmail: POP support!
Sendmail: UUCP support!
qmail: User-controlled lists!

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!
qmail: POP support!
Sendmail: UUCP support!
qmail: User-controlled lists!
Sendmail: Remote root exploits!

Compare to Sendmail:

1996.03, 8.7.5:	178375 words.
1997.01, 8.8.5:	209955 words.
1998.05, 8.9.0:	232188 words.

Hundreds of known bugs;
many security holes.

Different user-visible features!
qmail: POP support!
Sendmail: UUCP support!
qmail: User-controlled lists!
Sendmail: Remote root exploits!

But, in each MTA, most code
focuses on core MTA features.
Why the complexity gap?

1997.03: \$500 reward for
first qmail security hole.
Still unclaimed.

Four subsequent qmail books:
2000 Blum; 2002 Sill;
2004 Levine; 2007 Wheeler.

More than 1 million of
the Internet's SMTP servers
run qmail today.

2007.11: \$500 $\rightarrow$ \$1000;
qmail placed into public domain.

Mission: Invulnerable

Most “security” mechanisms are breakable, and are broken as soon as they become popular.

The conventional wisdom: “We’ll never build a serious software system without security holes.”

Why not? “It’s impossible.” Or: “Maybe it’s possible, but it’s much too expensive.”

The conventional wisdom:
“We’ll never build a tunnel
from England to France.”

Why not? “It’s impossible.”
Or: “Maybe it’s possible,
but it’s much too expensive.”

Engineer’s reaction:
How expensive is it?
How big a tunnel *can* we build?
How can we reduce the costs?

Eliminating bugs

Estimate bug rate of
software-engineering processes
by carefully reviewing code.
(Estimate is reliable enough;
"all bugs are shallow.")

Meta-engineer processes
that have lower bug rates.
Note: progress is *quantified*.

Well-known example:
Drastically reduce bug rate
of typical engineering process
by adding coverage tests.

Example where qmail did well:
“Don’t parse.”

Typical user interfaces
copy “normal” inputs and
quote “abnormal” inputs.
Inherently bug-prone:
simpler copying is wrong
but passes “normal” tests.
Example (1996 Bernstein):
format-string danger in `logger`.

qmail’s internal file structures
and program-level interfaces
don’t have exceptional cases.
Simplest code is correct code.

Example where qmail did badly: integer arithmetic.

In C et al., `a + b` *usually* means exactly what it says, but *occasionally* doesn't.

To detect these occasions, need to check for overflows. Extra work for programmer.

To guarantee sane semantics, extending integer range and failing only if out of memory, need to use large-integer library. Extra work for programmer.

The closest that qmail
has come to a security hole
(Guninski): potential overflow
of an unchecked counter.

Fortunately, counter growth
was limited by memory
and thus by configuration,
but this was pure luck.

Anti-bug meta-engineering:
Use language where `a + b`
means exactly what it says.

"Large-integer libraries are slow!"

That's a silly objection.
We need invulnerable systems,
and we need them today,
even if they are $10\times$ slower
than our current systems.
Tomorrow we'll make them faster.

Most CPU time is consumed
by a very small portion
of all the system's code.
Most large-integer overheads
are removed by smart compilers.
Occasional exceptions can be
handled manually at low cost.

Paper has more examples of
anti-bug meta-engineering:
automatic array extensions;
partitioning variables
to make data flow visible;
automatic updates of
“summary” variables;
abstraction for testability.

“Okay, we can achieve
much smaller bug rates.
But in a large system
we’ll still have many bugs,
including many security holes!”

Eliminating code

Measure code rate of
software-engineering processes.

Meta-engineer processes
that spend less code
to get the same job done.
Note: progress is *quantified*.

This is another classic topic
of software-engineering research.
Combines reasonably well
with reducing bug rate.

Example where qmail did well:
reusing access-control code.

A story from twenty years ago:
My `.forward` ran a program
creating a new file in `/tmp`.
Surprise: the program was
sometimes run under another uid!

How Sendmail handles `.forward`:
Check whether user can read it.
(Prohibit symlinks to secrets!)
Extract delivery instructions.
Keep track (often via queue file)
of instructions and user.

Many disastrous bugs here.

OS already tracks users.
OS already checks readability.
Why not reuse this code?

How qmail delivers to a user:
Start `qmail-local`
under the right uid.

When `qmail-local` reads
the user's delivery instructions,
the OS checks readability.
When `qmail-local` runs a
program, the OS assigns
the same uid to that program.
No extra code required!

Example where qmail did badly: exception handling.

qmail has thousands of conditional branches. About half are simply checking for temporary errors.

Easy to get wrong: e.g., “if `ipme_init()` returned −1, `qmail-remote` would continue” (fixed in qmail 0.92).

Easily fixed by better language.

Paper has more examples of
small-code meta-engineering:
identifying common functions;
reusing network tools;
reusing the filesystem.

“Okay, we can
build a system with less code,
and write code with fewer bugs.
But in a large system
we’ll still have bugs,
including security holes!”

Eliminating trusted code

Can architect computer systems to place most of the code into *untrusted* prisons.

Definition of "untrusted": no matter what the code does, no matter how badly it behaves, no matter how many bugs it has, it cannot violate the user's security requirements.

Measure *trusted* code volume, and meta-engineer processes that reduce this volume.
Note: progress is *quantified*.

Warning: “Minimizing privilege”
rarely eliminates trusted code.

Every security mechanism,
no matter how pointless,
says it’s “minimizing privilege.”
This is not a useful concept.

qmail did very badly here.
Almost all qmail code is trusted.
I spent considerable effort
“minimizing privilege”; stupid!
This distracted me from
eliminating trusted code.

Example: `jpegtopnm`,
converting JPEG into bitmap.

Easy to run `jpegtopnm`
in an "extreme sandbox"
that allows nothing but
(1) reading the JPEG file,
(2) writing the bitmap, and
(3) allocating limited memory.
Then `jpegtopnm` is untrusted.

Oops: cache-timing attacks etc.
violate memory "protection."
Solution: restrict CPU access,
for example with an interpreter.

Warning: Trusted code
is much more than the kernel.

(Orange Book screwed up:
defined TCB much too narrowly,
confusing many people.
Lampson/Abadi/Burrows/Wobber
give correct definition.)

Web-browser code is trusted.
But replace web browser's
built-in JPEG decompression
by a sandboxed `jpegtopnm`
and then that code is untrusted.
Can replace many components.

Analogy:
Give someone an account.
Allow him to upload a JPEG,
log in, run `jpegtopnm`,
show you the final bitmap.

The upload and `jpegtopnm`
can't touch *your* files!

Same for every transformation
that handles single-source data.
This is a huge amount of code.
We can make all of it untrusted.

What code remains trusted?
What types of code handle
data from multiple sources?

One common pattern:
merge data into "summary";
then transform the summary.

Example: merge mail messages
into a list of subjects.
Transformations of list
are then trusted.

Usually can delay the merge,
transforming messages separately.
Transform is then single-source.

Always have some trusted code.

Have to identify data sources
(local users, URLs, etc.);
copy this identification
from inputs to outputs;
cryptographically protect
network connections; etc.

But I see no reason that
trusted code has to be
a large fraction of all code
on the computer system.

The future

Architect systems so that
most functions are untrusted.
Minimize volume of code
providing those functions.
Minimize bug rate in that code.

My prediction: We *will* have
invulnerable software systems,
with no bugs in trusted code.
We will be confident that
these systems enforce the
user's security requirements.