Invulnerable software

D. J. Bernstein
University of Illinois at Chicago

Public goal of
computer-security research:
Protection of
the average home computer;
critical-infrastructure computers;
and everything in between.

Public goal of
computer-security research:
Protection of
the average home computer;
critical-infrastructure computers;
and everything in between.

Secret goal of
computer-security research:
Lifetime employment
for computer-security researchers.

ECRYPT is a consortium of European crypto researchers.

eSTREAM is the "ECRYPT Stream Cipher Project."

2004.11: eSTREAM calls for submissions of stream ciphers. Receives 34 submissions from 97 cryptographers around the world.

2008.04: After two hundred papers and several conferences, eSTREAM selects portfolio of 4 SW ciphers and 4 HW ciphers.

eSTREAM says: The HW ciphers are aimed at "deployment on passive RFID tags or low-cost devices such as might be used in sensor networks. Such devices are exceptionally constrained in computing potential . . . [Keys are] 80 bits which we believe to be adequate for the lower-security applications where such devices might be used."

Obviously these ciphers will be built into many chips over the next 5 or 10 years.

Iain Devlin, Alan Purvis,
"Assessing the security
of key length," 2007:
Buy FPGAs for US$3 million;
break 80-bit keys in 1 year.
Or: US$165 million; 1 week.

Cost will continue to drop.
Will come within reach
of more and more attackers.

Same story in public-key crypto.
1024-bit RSA will be broken.
160-bit ECC will be broken.

So users will pay us for
96-bit ciphers and 192-bit ECC.
And then those will be broken.
Continue for decades.

So users will pay us for
96-bit ciphers and 192-bit ECC.
And then those will be broken.
Continue for decades.

Success: Lifetime employment!

This is not a new pattern.
Consider, e.g., 56-bit DES,
or 48-bit Mifare CRYPTO1,
or MD5, or Keeloq.
All breakable by brute force,
and often by faster methods.

Naive conclusion
from all these attacks:
“There is no such thing as
100% secure cryptography!
Crypto breaks are inevitable!”

Naive conclusion
from all these attacks:
"There is no such thing as
100% secure cryptography!
Crypto breaks are inevitable!"

This conclusion is unjustified
and almost certainly wrong.

Nobody has found patterns
in output of 256-bit AES.
Most cryptographers think
that nobody ever will.

“AES software leaks keys
to cache-timing attacks!”

True, but we know how
to eliminate this problem
by (for example) bitslicing.

“Maybe secret-key crypto is okay,
but large quantum computers will
kill public-key cryptography!”

If large quantum computers are
built, they’ll break RSA and ECC,
but we have replacements.
See PQCrypto workshops.

Enough crypto for this talk.
How about all the rest
of computer security?

Flood of successful attacks,
even more than in crypto.

Conventional wisdom:
We'll never stop the flood.

Viega and McGraw: "Because
there is no such thing as 100%
security, attacks will happen."

Schneier: "Software bugs
(and therefore security flaws)
are inevitable."

Analogy:
"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?

Eventually a tunnel *was* built
from England to France.

Here's what I think:

Invulnerable software systems
can and *will* be built,
and will become standard.

Most "security" research today
doesn't aim for invulnerability,
doesn't contribute to it,
and will be discarded once
we have invulnerable software.

Eliminating bugs

Every security hole is a bug.

Everyone agrees that
we *can* eliminate all bugs
in *extremely small* programs.

What about larger programs?

Space-shuttle software:
"The last three versions of the
program—each 420000 lines
long—had just one error each.
The last 11 versions of this
software had a total of 17 errors."

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 my qmail software 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× 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.

More anti-bug meta-engineering
examples in my qmail paper:
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.

Kernel already tracks users.
Kernel 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 kernel checks readability.
When `qmail-local` runs a
program, the kernel 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.

More small-code meta-engineering examples in my qmail paper:
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.

What *are* the user's
security requirements?

My fundamental requirement:
The system keeps track
of sources of data.

When the system is asked
for data from source X,
it does not allow
data from source Y
to influence the result.

Example: When I view an
account statement from my bank,
I am not seeing data from other
web pages or email messages.

There is no obstacle to
centralization and minimization
of source-tracking code.
Can and should be small enough
to eliminate all bugs.

Classic military TCBs
used very few lines of code
to track (e.g.) Top Secret data.
VAX VMM Security Kernel
had < 50000 lines of code.

Minor programming problem to
support arbitrary source labels
instead of Top Secret etc.

“Doesn’t the UNIX/Linux kernel
already track sources?”

If I log into a system,
the kernel copies my uid
to my login process,
to other processes I start,
to files I create, etc.

But if I transfer data
to another user’s processes—
through the network or a file—
the kernel doesn’t remember
that I’m the source.

Source tracking today
is implemented by programmers
writing web browsers, mail clients,
PHP scripts, etc.

All of this code is trusted.
All other code in these programs
is also trusted, thanks to
nonexistent internal partitioning.

Your laptop has tens of millions
of lines of trusted code written by
thousands of novice programmers.
A screwup anywhere in that code
can violate security policy.

“Teach every programmer
how to write secure code.”

No, no, no!
If every programmer
is writing trusted code
then the system has
far too much trusted code.

We need new systems
with far less trusted code.
Enforce security in TCB
so that typical programmers
don’t have to worry about it.

Imagine a TCB tracking sources.

Alice's process reads Bob's file.
TCB automatically labels
process as /Alice/Bob.

Process creates a file.
TCB automatically labels
file as /Alice/Bob
(and refuses to touch a file
labelled only /Alice).

Joe's process reads the file
and another file from Charlie.
Process then has two labels:
/Joe/Alice/Bob; /Joe/Charlie.

A web-browsing process
that reads from mbna.com
and from nytimes.com
will have both labels.
TCB won't allow process
to write under just one label.

Solution 1: Track sources
separately for each variable
inside the web-browsing process.
Doable with compiler support.

Solution 2: Rewrite browser
in the classic UNIX style,
one process for each page.
More work but smaller TCB.

Quantitative questions:

How much code is required
for a TCB that enforces
source-tracking policy
against all other code?

How many bugs do we expect
in a TCB of this size?
Note: Can afford expensive
techniques to reduce bug rate.

If code volume is small enough,
and bug rate is small enough,
then we will be confident
that sources are tracked.