

Some thoughts on security after ten years of qmail 1.0

Daniel J. Bernstein

Department of Mathematics, Statistics, and Computer Science (M/C 249)
University of Illinois at Chicago, Chicago, IL 60607-7045, USA
djb@cr.yp.to

ABSTRACT

The qmail software package is a widely used Internet-mail transfer agent that has been covered by a security guarantee since 1997. In this paper, the qmail author reviews the history and security-relevant architecture of qmail; articulates partitioning standards that qmail fails to meet; analyzes the engineering that has allowed qmail to survive this failure; and draws various conclusions regarding the future of secure programming.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*bug elimination, code elimination*; D.4.6 [Operating Systems]: Security and Protection; H.4.3 [Information Systems Applications]: Communications Applications—*electronic mail*

General Terms

Security

Keywords

Eliminating bugs, eliminating code, eliminating trusted code

1. INTRODUCTION

1.1 The bug-of-the-month club

Every Internet service provider runs an MTA (a “Mail Transfer Agent”). The MTA receives mail from local users; delivers that mail to other sites through SMTP (the Internet’s “Simple Mail Transfer Protocol”); receives mail from other sites through SMTP; and delivers mail to local users.

I started writing an MTA, qmail, in 1995, because I was sick of the security holes in Eric Allman’s “Sendmail” software. Sendmail was by far the most popular MTA on the

*Date of this document: 2007.11.01. Permanent ID of this document: acd21e54d527dabf29afac0a2ae8ef41.

Internet at the time; see, e.g., [4]. Here’s what I wrote in the qmail documentation in December 1995:

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.

Fourteen Sendmail security holes were announced in 1996 and 1997. I stopped counting after that, and eventually I stopped paying attention. Searches indicate that Sendmail’s most recent emergency security release was version 8.13.6 in March 2006; see [10] (“remote, unauthenticated attacker could execute arbitrary code with the privileges of the Sendmail process”).

After more than twenty years of Sendmail releases known to be remotely exploitable, is anyone willing to bet that the latest Sendmail releases are not remotely exploitable? The announcement rate of Sendmail security holes has slowed, but this fact doesn’t help the administrators whose systems have been broken into through Sendmail security holes.

1.2 The qmail release

I started serious code-writing for qmail in December 1995. I had just finished teaching a course on algebraic number theory and found myself with some spare time. The final kick to get something done was a promise I had made to a colleague: namely, that I would run a large mailing list for him. Sendmail didn’t offer me the easy list administration that I wanted, and it seemed to take forever to deliver a message (serially!) to a long list of recipients, never mind Sendmail’s reliability problems and security problems.

The 7 December 1995 version of qmail had 14903 words of code, as measured by

```
cat *.c *.h | cpp -fpreprocessed \
| sed 's/[a-zA-Z0-9][a-zA-Z0-9]*/*x/g' \
| tr -d '\012' | wc -c
```

(Other complexity metrics paint similar pictures.) The 21 December 1995 version had 36062 words. The 21 January 1996 version, qmail 0.70, had 74745 words. After watching this version run on my computer for a few days I released it to the public, starting the qmail beta test.

On 1 August 1996 I released qmail 0.90, 105044 words, ending the qmail beta test. On 20 February 1997 I released

qmail 1.00, 117685 words. On 15 June 1998 I released the current version, qmail 1.03, 124540 words. A slight derivative created by the community, netqmail 1.05, has 124911 words. I am aware of four bugs in the qmail 1.0 releases.

For comparison: Sendmail 8.7.5, released in March 1996, had 178375 words; Sendmail 8.8.5, released in January 1997, had 209955 words; Sendmail 8.9.0, released in May 1998, had 232188 words. The Sendmail release notes report hundreds of bugs in these releases. There are some user-visible feature differences between Sendmail and qmail, such as qmail's POP support, and Sendmail's UUCP support, and qmail's user-controlled mailing lists, and Sendmail's “remote root exploit” feature—just kidding—but these don't explain the complexity gap; most of the code in each package is devoted to core MTA features needed at typical Internet sites.

Fingerprinting indicates that more than a million of the Internet's SMTP servers run either qmail 1.03 or netqmail 1.05. The third-party qmail.org site says

A number of large Internet sites are using qmail: USA.net's outgoing email, Address.com, Rediffmail.com, Colonize.com, Yahoo! mail, Network Solutions, Verio, MessageLabs (searching 100M emails/week for malware), listserv.acsu.buffalo.edu (a big listserv hub, using qmail since 1996), Ohio State (biggest US University), Yahoo! Groups, Listbot, USWest.net (Western US ISP), Telenordia, gmx.de (German ISP), NetZero (free ISP), Critical Path (email outsourcing service w/ 15M mailboxes), PayPal/Confinity, Hypermart.net, Casema, Pair Networks, Topica, MyNet.com.tr, FSmail.net, Mycom.com, and vuurwerk.nl.

Several authors have written qmail books: [7], [20], [14], [22]. Comprehensive statistics are hard to collect, but samples consistently indicate that qmail sends and receives a large fraction of all of the legitimate email on the Internet.

1.3 The qmail security guarantee

In March 1997, I took the unusual step of publicly offering \$500 to the first person to publish a verifiable security hole in the latest version of qmail: for example, a way for a user to exploit qmail to take over another account. My offer still stands. Nobody has found any security holes in qmail. I hereby increase the offer to \$1000.

Of course, “security hole in qmail” does not include problems outside of qmail: for example, NFS security problems, TCP/IP security problems, DNS security problems, bugs in scripts run from `.forward` files, and operating-system bugs generally. It's silly to blame a problem on qmail if the system was already vulnerable before qmail was installed!

It's not as silly to blame qmail for failing to encrypt and authenticate mail messages sent through the network; maybe cryptography should be handled by applications such as qmail, rather than by TCP/IP. But cryptography is outside the scope of the qmail security guarantee.

Denial-of-service attacks are also specifically disallowed: they are present in every MTA, widely documented, and very hard to fix without a massive overhaul of several major protocols. One could argue, and I would agree, that Internet mail desperately needs this overhaul; but that's a topic for another paper.

What the qmail security guarantee *does* say is that users

can't exploit qmail to steal or corrupt other users' data. If other programs met the same standard, and if our network links were cryptographically protected, then the only remaining security problems on the Internet would be denial-of-service attacks.

1.4 Contents of this paper

How was qmail engineered to achieve its unprecedented level of security? What did qmail do well from a security perspective, and what could it have done better? How can we build other software projects with enough confidence to issue comparable security guarantees?

My views of security have become increasingly ruthless over the years. I see a huge amount of money and effort being invested in security, and I have become convinced that most of that money and effort is being wasted. Most “security” efforts are designed to stop yesterday's attacks but fail completely to stop tomorrow's attacks and are of no use in building invulnerable software. These efforts are a distraction from work that *does* have long-term value.

In retrospect, some of qmail's “security” mechanisms were half-baked ideas that didn't actually accomplish anything and that could have been omitted with no loss of security. Other mechanisms have been responsible for qmail's successful security track record. My main goal in this paper is to explain how this difference could have been recognized in advance—how software-engineering techniques can be *measured* for their long-term security impact.

Section 2 articulates three specific directions of progress towards invulnerable software systems. The remaining sections of the paper discuss qmail's successes and failures in these three directions.

Much of what I say has been said many times before. (This isn't the first paper on software security; it isn't even the first paper on qmail security.) I apologize for not having taken the time to locate original sources.

2. HOW CAN WE MAKE PROGRESS?

A software *bug* means, by definition, a software feature violating the user's requirements. A software *security hole* means, by definition, a software feature violating the user's security requirements. Every security hole is therefore a bug.

(Advocates of formal specifications, security policies, and so on will correctly point out that making a complete list of user requirements is quite difficult—especially in gray areas that the user hasn't thought through. This complexity has an effect on the difficulty of eliminating security holes, as discussed below. But the difficulty would exist even without the complexity: today's software doesn't even meet the most basic security requirements that come to mind!)

Suppose that there is, on average, 1 bug in every N words of code—but that there are $10000N$ words of code inside the computer. The unhappy conclusion is that the computer has about 10000 bugs. Many of those bugs are, presumably, security holes.

How can we make progress towards having *no* security holes? How can we *measure* the progress? This section gives three answers. This section also discusses three ways that the community has distracted itself from making progress.

2.1 Answer 1: eliminating bugs

The first answer, and surely the most obvious answer, is to reduce the bug rate.

We can estimate the bug rate of a software-engineering process by carefully reviewing the resulting code and tracking the number of bugs found as a function of the amount of code reviewed. We can then compare the bug rates of different software-engineering processes. We can meta-engineer processes with lower bug rates.

(Sometimes subtle bugs slip past a code review. However, experience suggests that the overall bug rate, taking account of all the user requirements, is only slightly larger than the not-so-subtle-bug rate. “Given enough eyeballs, all bugs are shallow,” Eric Raymond commented in [17, Section 4: “Release Early, Release Often”].)

Bug elimination is one of the classic topics of software-engineering research. It is well known, for example, that one can drastically reduce the bug rate of a typical software-engineering process by adding coverage tests. It does not seem to be as well known that bug rates are affected by choices in earlier stages of the software-engineering process. See Section 3 of this paper for further discussion.

2.2 Answer 2: eliminating code

Consider again the computer system with $10000N$ words of code. Suppose that the bug rate is reduced far below 1 bug in every N words of code, but not nearly far enough to eliminate all the bugs in $10000N$ words of code. How can we make progress?

The second answer is to reduce the amount of code in the computer system. Software-engineering processes vary not only in the number of bugs in a given volume of code, but also in the volume of code used to provide the features that the user wants. Code elimination is another classic topic of software-engineering research: we can compare the code volumes produced by different software-engineering processes, and we can meta-engineer processes that do the job with lower volumes of code. See Section 4 of this paper for further discussion.

Rather than separately measuring the bugs-code ratio and the code-job ratio one could measure the product, the bugs-job ratio. However, the separate measures seem to highlight different techniques. Furthermore, the code-job ratio is of independent interest as a predictor of software-engineering time, and the bugs-code ratio is of independent interest as a predictor of debugging time.

2.3 Answer 3: eliminating trusted code

Suppose that, by producing a computer system with less code, and producing code with fewer bugs, we end up with a system having fewer than 1000 bugs. Presumably many of those bugs are still security holes. How can we make progress?

The third answer is to reduce the amount of *trusted* code in the computer system. We can architect computer systems to place most of the code into *untrusted* prisons. “Untrusted” means that code in these prisons—no matter what the code does, no matter how badly it behaves, no matter how many bugs it has—cannot violate the user’s security requirements. We can measure the amount of *trusted* code in our computer systems, and we can meta-engineer processes that produce systems with lower volumes of trusted code.

Of course, general techniques for reducing the amount of code in a system are also helpful for reducing the amount of trusted code. But additional techniques allow the amount of trusted code to be much smaller than the total amount

of code. There is a pleasant synergy between eliminating trusted code and eliminating bugs: we can afford relatively expensive techniques to eliminate the bugs in trusted code, simply because the volume of code is smaller.

Consider, for example, the portion of the `sendmail` code responsible for extracting an email address from the header of a mail message. According to [12], Mark Dowd discovered in 2003 that this code had a security hole:

Attackers may remotely exploit this vulnerability to gain “root” or superuser control of any vulnerable Sendmail server. . . . This vulnerability is especially dangerous because the exploit can be delivered within an email message and the attacker doesn’t need any specific knowledge of the target to launch a successful attack. . . . X-Force has demonstrated that this vulnerability is exploitable in real-world conditions on production Sendmail installations.

Suppose that the same address-extraction code is run under an interpreter enforcing two simple data-flow rules:

- the only way that the code can see the rest of the system is by reading this mail message;
- the only way that the code can affect the rest of the system is by printing one string determined by that mail message.

The code is then incapable of violating the user’s security requirements. An attacker who supplies a message that seizes complete control of the code can control the address printed by the code—but the attacker could have done this anyway without exploiting any bugs in the code. The attacker is incapable of doing anything else. See Section 5.2 for another example of the same idea.

(I am implicitly assuming that the user’s security requirements do not prohibit the creators of a mail message from controlling the address extracted from the message. But the requirements may be more complicated. Perhaps the user is creating a mail message by combining attachments from several sources; perhaps the user requires that each attachment be incapable of affecting other attachments, the header, etc. The interpreter then has to impose corresponding data-flow restrictions.)

By reducing the amount of trusted code far enough, and reducing the bug rate far enough, one can reasonably hope to produce a system where there are no bugs in the trusted code, and therefore no security holes. Presumably there are still bugs in the untrusted code, but those bugs cannot violate the user’s security requirements. Engineering a secure software system is easier than engineering a bug-free software system.

2.4 Distraction 1: chasing attackers

For many people, “security” consists of observing current attacks and changing something—anything!—to make those attacks fail. It is easy to understand the attraction of this reactive type of work: watching an attack fail, where earlier it would have succeeded, provides instant gratification for the defender.

Sometimes the changes fix the specific bugs exploited by the attacks. The Ubuntu distribution of the Linux operating system has issued more than 100 emergency security patches this year, fixing various bugs in a wide range of programs.

Sometimes the changes don't fix any bugs. "Firewalls" and "anti-virus systems" and "intrusion-detection systems" attempt to recognize attacks without patching the software targeted by the attacks.

Either way, the changes do nothing to fix the software-engineering deficiencies that led to the security holes being produced in the first place. If we define success as stopping yesterday's attacks, rather than as making progress towards stopping all possible attacks, then we shouldn't be surprised that our systems remain vulnerable to tomorrow's attacks.

2.5 Distraction 2: minimizing privilege

Many additional "security" efforts are applications of the "principle of least privilege." The principle is widely credited to Saltzer and Schroeder, who stated it as follows in [18]: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job."

These "security" efforts work as follows. We observe that program P has no legitimate need to access operating-system resource R . We then use (and possibly extend) operating-system controls to prevent P from accessing R . We prevent an image-displaying program from sending data through the network; we prevent a DNS-lookup program from reading disk files; etc. See, for example, [3], [21], [11], [2], [15], [1], [16], and [23]. Section 5.1 discusses some qmail examples.

I have become convinced that this "principle of least privilege" is fundamentally wrong. Minimizing privilege might reduce the damage done by some security holes but almost never fixes the holes. Minimizing privilege is not the same as minimizing the amount of trusted code, does not have the same benefits as minimizing the amount of trusted code, and does not move us any closer to a secure computer system.

Consider, as an example, [11]'s confinement of Netscape's "DNS helper" program, preventing the program from accessing the local disk. This confinement did not prevent the `libresolv` bug in [8] from being a security hole in Netscape: an attacker could use the bug to seize control of the "DNS helper," modify all subsequent DNS data seen by Netscape, and steal the user's web connections. The situation before [11] was that bugs in the "DNS helper" had the power to violate the user's security requirements and therefore needed to be fixed; the situation after [11] was that bugs in the "DNS helper" had the power to violate the user's security requirements and therefore needed to be fixed.

The defining feature of untrusted code is that it *cannot* violate the user's security requirements. Turning a "DNS helper" into untrusted code is necessarily more invasive than merely imposing constraints upon the operating-system resources accessed by the program. The "DNS helper" handles data from many sources, and each source must be prevented from modifying other sources' data.

2.6 Distraction 3: speed, speed, speed

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.

—Knuth in [13, page 268]

The most obvious effect of the pursuit of speed is that programmers put effort into low-level speedups, attempting to save time by tweaking small sections of code.

Programmers know when they're doing this. They see the increased programming time. They see the increased bug rate. They are generally quite happy to change their engineering process to skip almost all of this effort. Knuth's commentary seems to be aimed at novice programmers who don't understand how to use profiling tools.

Unfortunately, the pursuit of speed has other effects that are not as blatant and that are not as easy to fix.

Consider the address-extraction example in Section 2.3. Using an interpreter to impose simple data-flow restrictions on the address-extraction code would make bugs in the code irrelevant to security—a huge benefit. However, most programmers will say "Interpreted code is too slow!" and won't even try it.

Starting a new operating-system process could impose similar restrictions without an interpreter; see Section 5.2. However, most programmers will say "You can't possibly start a new process for each address extraction!" and won't even try it.

Anyone attempting to improve programming languages, program architectures, system architectures, etc. has to overcome a similar hurdle. Surely some programmer who tries (or considers) the improvement will encounter (or imagine) some slowdown in some context, and will then accuse the improvement of being "too slow"—a marketing disaster.

I don't like waiting for my computer. I really don't like waiting for someone else's computer. A large part of my research is devoted to improving system performance at various levels. (For example, my paper [6] is titled "Curve25519: new Diffie-Hellman speed records.") But I find security much more important than speed. We *need* invulnerable software systems, and we need them today, even if they are ten times slower than our current systems. Tomorrow we can start working on making them faster.

I predict that, once we all have invulnerable software systems, we'll see that security doesn't actually need much CPU time. The bulk of CPU time is consumed by a tiny fraction of our programs, and by a tiny fraction of the code within those programs; time spent on security verification will be unnoticeable outside these "hot spots." A typical hot spot spends millions of CPU cycles on data from a single source; modern compiler techniques, in some cases aided by proofs, will be able to hoist all the security verification out of the inner loops. The occasional hot spots with tricky security constraints, such as encryption of network packets, will be trusted bug-free code.

3. ELIMINATING BUGS

For many years I have been systematically identifying error-prone programming habits—by reviewing the literature, analyzing other people's mistakes, and analyzing my own mistakes—and redesigning my programming environment to eliminate those habits.

I had already made some progress in this direction when I started writing qmail, and I made further progress during the development of qmail 1.0. This doesn't mean I'm happy with the bug rate of the programming environment I used for qmail; my error rate has continued to drop in the last decade, and I see many aspects of the qmail programming environment as hopelessly obsolete.

Fortunately, my bug rate in the mid-1990s was low enough that—given the low volume of qmail code, as discussed in Section 4—there were only a few bugs in qmail 1.0. None of those bugs were security holes. This is the explanation for qmail’s exceptional security record. Note that qmail’s privilege minimization didn’t help at all; see Section 5.1 for further discussion.

This section discusses several examples of *anti-bug meta-engineering*: modifying programming languages, program structures, etc. to reduce bug rates.

3.1 Enforcing explicit data flow

The standard argument against global variables is that they can create hidden data flow, often surprising the programmer.

Consider, for example, the following bug fixed in qmail 0.74: “`newfield_datemake` would leave `newfield_date` alone if it was already initialized, even though `qmail-send` calls `newfield_datemake` anew for each bounce.”

I originally wrote the `newfield_datemake` function as part of the `qmail-inject` program, which sends exactly one outgoing message and needs to create exactly one Date field, conveniently stored in a global variable `newfield_date`. But then I reused the same function in the `qmail-send` program, which sends many outgoing messages (“bounce messages,” i.e., non-delivery reports) and needs to create a new Date field for each outgoing message. I had forgotten that `newfield_datemake` didn’t reset an existing `newfield_date`; this global variable ended up transmitting information—specifically, the obsolete Date field—from the previous message to the current message.

Hidden data flow is also at the heart of buffer-overflow bugs. The statement `x[i] = m` in C might appear at first glance to modify only the `x` variable but—if `i` is out of range—might actually modify any variable in the program, including return addresses and memory-allocation control structures. Similarly, reading `x[i]` might read any variable in the program.

Several aspects of qmail’s design make qmail’s internal data flow easier to see. For example, large portions of qmail run in separate UNIX processes. The processes are connected through pipelines, often through the filesystem, and occasionally through other communication mechanisms, but they do not have direct access to each other’s variables. Because each process has a relatively small state, it has relatively few opportunities for the programmer to screw up the data flow. At a lower level, I designed various array-access functions for which the indices were visibly in range, and I avoided functions for which this was hard to check.

Nowadays I am much more insistent on programming language support for smaller-scale partitioning, sane bounds checking, automatic updates of “summary” variables (e.g., “the number of nonzero elements of this array”), etc. By “sane bounds checking” I don’t mean what people normally mean by “bounds checking,” namely raising an exception if an index is out of range; what I mean is automatic array extension on writes, and automatic zero-fill on reads. (Out of memory? See Section 4.2.) Doing the same work by hand is silly.

3.2 Simplifying integer semantics

Another surprise for the programmer is that `y` can be much smaller than `x` after `y = x + 1`. This happens if `x`

is the largest representable integer, typically $2^{31} - 1$; `y` will then be the smallest representable integer, typically -2^{31} .

The closest that qmail has come to a security hole was a potential overflow (pointed out by Georgi Guninski) of a 32-bit counter that I had failed to check. Fortunately, the counter’s growth was limited by the available memory, which in turn was limited by standard configuration; but the same 32-bit increment operation in another context could easily have caused a disastrous bug.

Similar comments apply to other integer operations. The operation semantics *usually* match the mathematical semantics that the programmer intends, but *occasionally* don’t. If I want to detect those occasions, I have to go to extra work to check for overflows. If I want to have sane mathematical semantics applied on those occasions—extending the integer range, and failing only if I run out of memory—I have to go to extra work to use a large-integer library.

Most programming environments are meta-engineered to make typical software easier to write. They should instead be meta-engineered to make *incorrect* software *harder* to write. An operation that is not exactly what I normally want should take *more* work to express than an operation that is exactly what I normally want. There are occasions when I really do want arithmetic modulo 2^{32} (or 2^{64}), but I am happy to do extra work on those occasions.

In some languages, `a + b` means exactly what it says: the sum of `a` and `b`. Often these languages are dismissed as being “too slow” for general use: an inner loop such as

```
for (i = 0; i < n; ++i) c[i] = a[i] + b[i];
```

suddenly involves `n` calls to an expensive high-precision integer-arithmetic function such as `gmp_add()`. But it is not rocket science for a compiler to generate code that keeps track of the locations of large integers and that replaces the `gmp_add()` operations by machine operations in the typical case that all integers are small. Perhaps some slowdowns are more difficult to address, but—as in Section 2.6—we should *first* get the code right and *then* worry about its speed.

3.3 Avoiding parsing

I have discovered that there are two types of command interfaces in the world of computing: good interfaces and user interfaces.

The essence of user interfaces is *parsing*: converting an unstructured sequence of commands, in a format usually determined more by psychology than by solid engineering, into structured data.

When another programmer wants to talk to a user interface, he has to *quote*: convert his structured data into an unstructured sequence of commands that the parser will, he hopes, convert back into the original structured data.

This situation is a recipe for disaster. The parser often has bugs: it fails to handle some inputs according to the documented interface. The quoter often has bugs: it produces outputs that do not have the right meaning. Only on rare joyous occasions does it happen that the parser and the quoter both misinterpret the interface in the same way.

I made these comments in the original qmail documentation, along with two examples of how parsing and quoting were avoided in qmail’s extremely simple internal file structures and program-level interfaces. But I didn’t say anything about how parsing bugs and quoting bugs could be avoided when external constraints prohibit better interfaces.

Consider, for example, the following bug, fixed in qmail 0.74: “`qmail-inject` did not check whether `USER` needed quoting.” What `qmail-inject` was doing here was creating a `From` line showing the user’s name:

```
From: "D. J. Bernstein" <djb@cr.yp.to>
```

Normally the user’s account name, in this case `djb`, can be inserted verbatim before the `@`; but the name has to be quoted in a particular way if it contains unusual characters such as parentheses. My tests didn’t check specifically for unusual characters in `USER`, so they didn’t distinguish verbatim insertion from proper encoding.

Consider, as another example, the following format-string bug in the UNIX `logger` program:

```
Presumably logger on your system is doing
  syslog(pri,buf)
instead of the correct
  syslog(pri,"%s",buf)
... It is my guess that this does not constitute a
security hole in logger beyond a denial-of-service
attack, since an attacker would have an awfully
difficult time encoding reachable VM addresses
into printable ASCII characters, but without seeing
disassembled object code I can't be sure. Better
safe than sorry.
```

Tests that don’t put `%` into `buf` won’t notice the difference between `syslog(pri,buf)` and `syslog(pri,"%s",buf)`. (The quote is from [5], four years before format-string bugs were widely appreciated; see [19, Introduction].)

Verbatim copying of “normal” inputs—with ad-hoc quoting required for “abnormal” inputs—seems to be a universal feature of user interfaces. One way to catch the resulting bugs, in situations where the interface cannot be improved, is to systematically convert each quoting rule into another test.

3.4 Generalizing from errors to inputs

The following bug was fixed in qmail 0.90: “Failure to `stat .qmail-owner` was not an error.”

If Bob puts several addresses into `~bob/.qmail-buddies` then qmail will forward `bob-buddies` mail to those addresses. Delivery errors are sent back to the original sender by default, but Bob can direct errors to a different address by putting that address into `.qmail-buddies-owner`.

To check whether `.qmail-buddies-owner` exists, qmail uses the UNIX `stat()` function. If `stat()` says that the file exists, qmail directs errors to `bob-buddies-owner`. If `stat()` says that the file doesn’t exist, qmail uses the original sender address.

The problem is that `stat()` can fail temporarily. For example, `.qmail-buddies-owner` could be on a network filesystem that is momentarily unavailable. Before version 0.90, qmail would treat this error the same way as nonexistence—which would be wrong if the file actually exists. The only correct behavior is for qmail to give up and try again later.

My test suite didn’t cover the temporary-failure case. I had some tests for common cases, and I had some tricky tests going to extra effort to set up various error cases, but I didn’t have any tests manufacturing network-filesystem errors.

The volume of error-handling code can be drastically reduced, as discussed in Section 4.2, but computer systems

always have *some* error-handling code. How can we prevent bugs in code that is rarely exercised?

Testing would have been much easier if I had factored the code into (1) a purely functional protocol handler that could talk to anything, not just the filesystem, and (2) a simple wrapper that plugged `stat()` into the protocol handler. Feeding a comprehensive set of test cases to the protocol handler would then have been quite easy—and would have immediately caught this bug.

3.5 Can we really measure anti-bug progress?

There is an obvious difficulty in modelling user-interface research in general, and bug-elimination research in particular. The goal is to have users, in this case programmers, make as few mistakes as possible in achieving their desired effects. How do we model this situation—how do we model human psychology—except by experiment? How do we even recognize mistakes without a human’s help?

If someone can write a program to recognize a class of mistakes, great—we’ll incorporate that program into the user interface, eliminating those mistakes—but we still won’t be able to recognize the *remaining* mistakes. As a mathematician I’m bothered by this lack of formalization; I expect to be able to *define* a problem whether or not I can *solve* it.

Fortunately, research can and does proceed without models. We can observe humans to measure their programming-bug rates, even though we don’t know the algorithms that the humans are using. We can see that some software-engineering tools are bug-prone, and that others are not, without having any idea how to mathematically prove it.

4. ELIMINATING CODE

To this very day, idiot software managers measure ‘programmer productivity’ in terms of ‘lines of code produced,’ whereas the notion of ‘lines of code spent’ is much more appropriate.

—Dijkstra in [9, page EWD962-4]

This section discusses several examples of *code-volume-minimization meta-engineering*: changing programming languages, program structures, etc. to reduce code volume. As in Section 3, some of these examples were used in qmail and contributed to qmail’s low bug count, while other examples show that it is possible to do much better.

4.1 Identifying common functions

Here is a section of code from Sendmail (line 1924 of `util.c` in version 8.8.5):

```
if (dup2(fdv[1], 1) < 0)
{
    syserr("%s: cannot dup2 for stdout", argv[0]);
    _exit(EX_OSERR);
}
close(fdv[1]);
```

The `dup2()` function copies a file descriptor from one location to another; this `dup2()/close()` pattern moves a file descriptor from one location to another. There are several other instances inside Sendmail of essentially the same `dup2()/close()` pattern.

This particular pattern occurs only once in qmail, inside an `fd_move()` function called from a dozen other locations in the code:

```

int fd_move(int to,int from)
{
    if (to == from) return 0;
    if (fd_copy(to,from) == -1) return -1;
    close(from);
    return 0;
}

```

Most programmers would never bother to create such a small function. But several words of code are saved whenever one occurrence of the `dup2()`/`close()` pattern is replaced with one call to `fd_move()`; replacing a dozen occurrences saves considerably more code than were spent writing the function itself. (The function is also a natural target for tests.)

The same benefit scales to larger systems and to a huge variety of functions; `fd_move()` is just one example. In many cases an automated scan for common operation sequences can suggest helpful new functions, but even without automation I frequently find myself thinking “Haven’t I seen this before?” and extracting a new function out of existing code.

4.2 Automatically handling temporary errors

Consider the following excerpt from `qmail-local`:

```
if (!stralloc_cats(&dtline, "\n")) temp_nomem();
```

The `stralloc_cats` function changes a dynamically resized string variable `dtline` to contain the previous contents of `dtline` followed by a linefeed. Unfortunately, this concatenation can run out of memory. The `stralloc_cats` function then returns 0, and `qmail-local` exits via `temp_nomem()`, signalling the rest of the `qmail` system to try again later.

There are thousands of conditional branches in `qmail`. About half of them—I haven’t tried to count exactly—are doing nothing other than checking for temporary errors.

In many cases I built functions such as

```

void outs(s)
char *s;
{
    if (substdio_puts(&ss1,s) == -1) _exit(111);
}
```

to try an operation and exit the program upon temporary error. However, I didn’t—and don’t—like repeating the same work for each operation.

I could have pushed these tests into a relatively small number of bottom-level subroutines for memory allocation, disk reads, etc., exiting the program upon any temporary error. However, this strategy is unacceptable for long-running programs such as `qmail-send`. Those programs aren’t allowed to exit unless the system administrator asks them to!

I’ve noticed that many libraries and languages take the same strategy, rendering them similarly unacceptable for long-running programs. Maybe specialized types of programs should use specialized software-engineering environments, but I don’t view long-running programs as a specialized case; I question the wisdom of designing software-engineering environments that are unsuitable for building those programs.

Fortunately, programming languages can—and in some cases do—offer more powerful exception-handling facilities, aborting clearly defined subprograms and in some cases automatically handling error reports. In those languages I would be able to write

```
stralloc_cats(&dtline, "\n")
```

or simply

```
dtline += "\n"
```

without going to extra effort to check for errors. The reduced code volume would eliminate bugs; for example, the bug “if `ipme_init()` returned -1, `qmail-remote` would continue” (fixed in `qmail` 0.92) would not have had a chance to occur.

When I wrote `qmail` I rejected many languages as being much more painful than C for the end user to compile and use. I was inexplicably blind to the possibility of writing code in a better language and then using an automated translator to convert the code into C as a distribution language. Stroustrup’s `cfont`, the original compiler from C++ to C, is an inspirational example, although as far as I know it has never acquired exception-handling support.

4.3 Reusing network tools

UNIX has a general-purpose tool, `inetd`, that listens for network connections. When a connection is made, `inetd` runs another program to handle the connection. For example, `inetd` can run `qmail-smtpd` to handle an incoming SMTP connection. The `qmail-smtpd` program doesn’t have to worry about networking, multitasking, etc.; it receives SMTP commands from one client on its standard input, and sends the responses to its standard output.

Sendmail includes its own code to listen for network connections. The code is more complicated than `inetd`, in large part because it monitors the system’s load average and reduces service when there is heavy competition for the CPU.

Why does Sendmail not want to handle mail when the CPU is busy? The basic problem is that, as soon as Sendmail accepts a new message, it immediately goes to a lot of effort to figure out where the message should be delivered and to try delivering the message. If many messages show up at the same time then Sendmail tries to deliver all of them at the same time—usually running out of memory and failing at most of the deliveries.

Sendmail tries to recognize this situation by checking the load average. If the CPU is busy, Sendmail inserts new messages into a queue of not-yet-delivered messages. Sendmail has a background delivery mechanism that periodically runs through the queue, trying to deliver each message in turn. The background delivery mechanism never overloads the computer; if many messages arrive in the queue at once, the messages aren’t even noticed until the next queue run, and are then handled with highly limited parallelism.

The queue-run interval “is typically set to between fifteen minutes and one hour,” the documentation says; “RFC 1123 section 5.3.1.1 recommends that this be at least 30 minutes.” System administrators who set a very short queue-run interval, for example 30 seconds, find Sendmail trying each queued message thousands of times a day.

Why does Sendmail have a foreground delivery mechanism? Why does it not put *all* incoming messages into the queue? The answer is that—even when the CPU is *not* busy—queued messages are not delivered immediately. Putting all incoming messages into the queue would mean waiting for the next queue run before delivery; presumably users would sometimes see the delays and complain.

With `qmail`, a small amount of extra code notifies the background delivery mechanism, `qmail-send`, when a message is placed into the queue. The `qmail-send` program

instantly tries delivering the message (if it is not busy), and tries again later if necessary, on a reasonable schedule. Consequently, the motivation for a foreground delivery mechanism disappears; qmail has no foreground delivery mechanism. Furthermore, the motivation for checking the load average disappears; qmail is happy to run from `inetd`.

If I *did* want to check the load average, I would do so in a general-purpose tool like `inetd`, rather than repeating the code in each application.

4.4 Reusing access controls

I started using UNIX, specifically Ultrix, twenty years ago. I remember setting up my `.forward` to run a program that created a file in `/tmp`. I remember inspecting thousands of the resulting files and noticing in amazement that Sendmail had occasionally run the program under a uid other than mine.

Sendmail handles a user's `.forward` as follows. It first checks whether the user is allowed to read `.forward`—maybe the user has set up `.forward` as a symbolic link to a secret file owned by another user. It then extracts delivery instructions from `.forward`, and makes a note of them (possibly in a queue file to be handled later), along with a note of the user responsible for those instructions—in particular, the user who specified a program to run. This is a considerable chunk of code (for example, all of `safefile.c`, plus several scattered segments of code copying the notes around), and it has contained quite a few bugs.

Of course, the operating system already has its own code to check whether a user is allowed to read a file, and its own code to keep track of users. Why write the same code again?

When qmail wants to deliver a message to a user, it simply starts a delivery program, `qmail-local`, under the right uid. When `qmail-local` reads the user's delivery instructions, the operating system automatically checks whether the user is allowed to read the instructions. When `qmail-local` runs a program specified by the user, the operating system automatically assigns the right uid to that program.

I paid a small price in CPU time for this code reuse: qmail starts an extra process for each delivery. But I also avoided all the extra system calls that Sendmail uses to check permissions. Anyway, until someone shows me a real-world mail-delivery computer bottlenecked by qmail's `fork()` time, I'm certainly not going to spend extra code to reduce that time.

4.5 Reusing the filesystem

Suppose that the National Security Agency's SMTP server receives mail for the `efd-friends@nsa.gov` mailing list. How does the MTA find out that it is supposed to accept mail for `nsa.gov`? How does the MTA find the delivery instructions for `efd-friends`?

Evidently the MTA will look up the names `nsa.gov` and `efd-friends` in a database. Maybe the database isn't called a "database"; maybe it's called an "associative array" or something else; whatever it's called, it is capable of giving back information stored under names such as `efd-friends` by the system administrator and the mailing-list manager.

With Sendmail, names such as `nsa.gov` and various other configuration options are listed inside a "configuration file" having a fairly complicated format. Mailing lists such as `efd-friends` are listed inside an "aliases file." Looking up `nsa.gov` and `efd-friends` inside these files requires a considerable amount of parsing code.

Of course, the operating system already has code to store chunks of data under specified names and to retrieve the chunks later. The chunks are called "files"; the names are called "filenames"; the code is called "the filesystem." Why write the same code again?

With qmail, the delivery instructions for `efd-friends` are the contents of a file named `.qmail-efd-friends`. Finding or modifying those instructions is a simple matter of opening that file. Users already have tools for creating and managing files; there is no need for qmail to reinvent those tools.

I should have, similarly, put the `nsa.gov` configuration into `/var/qmail/control/domains/nsa.gov`, producing the same simplicity of code. I instead did something requiring slightly more complicated code: `nsa.gov` is a line in a file rather than a file in a directory. I was worried about efficiency: most UNIX filesystems use naive linear-time algorithms to access directories, and I didn't want qmail to slow down on computers handling thousands of domains. Most UNIX filesystems also consume something on the scale of a kilobyte to store a tiny file.

In retrospect, it was stupid of me to spend code—not just this file-parsing code, but also code to distribute message files across directories—dealing with a purely hypothetical performance problem that I had not measured as a bottleneck. Furthermore, to the extent that measurements indicated a bottleneck (as they eventually did for the message files on busy sites), I should have addressed that problem at its source, fixing the filesystem rather than complicating every program that uses the filesystem.

5. ELIMINATING TRUSTED CODE

5.1 Accurately measuring the TCB

"Even if all of these programs are completely compromised, so that an intruder has control over the `qmaild`, `qmails`, and `qmailr` accounts and the mail queue, he still can't take over your system," I wrote in the qmail documentation. "None of the other programs trust the results from these five." I continued in the same vein for a while, talking about privilege minimization as something helpful for security.

Let's think about this for a minute. Suppose that there *is* a bug in `qmail-remote` allowing an attacker to take control of the `qmailr` account. The attacker can then steal and corrupt the system's outgoing mail, even if network connections are completely protected by strong cryptography. This is a security disaster, and it needs to be fixed. The only way that qmail avoids this disaster is by avoiding bugs.

Similarly, I shouldn't have highlighted the small amount of code in qmail capable of affecting files owned by `root`. Almost all of the code in qmail is capable of affecting files owned by normal users—either on disk or in transit through the mail system—and is therefore in a position to violate the users' security requirements. The only way that this code avoids security holes is by avoiding bugs.

Programmers writing word processors and music players generally don't worry about security. But users expect those programs to be able to handle files received by email or downloaded from the web. Some of those files are prepared by attackers. Often the programs have bugs that can be exploited by the attackers.

When I taught a "UNIX security holes" course in 2004, I asked the students to find new security holes for their

homework. I ended up disclosing 44 security holes found by the students. Most of those were in programs that are usually—incorrectly—viewed as being outside the system’s trusted code base.

For example, Ariel Berkman discovered a buffer overflow in xine-lib, a movie-playing library. Users were at risk whenever they played movies downloaded from the web: if the movies were supplied by an attacker then the attacker could read and modify the users’ files, watch the programs that the users were running, etc.

“Secure” operating systems and “virtual machines” say that their security is enforced by a small base of trusted code. Unfortunately, a closer look shows that this “security” becomes meaningless as soon as one program handles data from more than one source. The operating system does nothing to stop one message inside a mail-reading program from stealing and corrupting other messages handled by the same program, for example, or to stop one web page inside a browser from stealing and corrupting other web pages handled by the same browser. The mail-reading code and browsing code are, in fact, part of the trusted code base: bugs in that code are in a position to violate the user’s security requirements.

5.2 Isolating single-source transformations

The `jpegtopnm` program reads a JPEG file, a compressed image, as input. It uncompresses the image, produces a bitmap as output, and exits. Right now this program is trusted: its bugs can compromise security. Let’s see how we can fix that.

Imagine running the `jpegtopnm` program in an “extreme sandbox” that doesn’t let the program do anything other than read the JPEG file from standard input, write the bitmap to standard output, and allocate a limited amount of memory. Existing UNIX tools make this sandbox tolerably easy for `root` to create:

- Prohibit new files, new sockets, etc., by setting the current and maximum `RLIMIT_NOFILE` limits to 0.
- Prohibit filesystem access: `chdir` and `chroot` to an empty directory.
- Choose a uid dedicated to this process ID. This can be as simple as adding the process ID to a base uid, as long as other system-administration tools stay away from the same uid range.
- Ensure that nothing is running under the uid: `fork` a child to run `setuid(targetuid)`, `kill(-1, SIGKILL)`, and `_exit(0)`, and then check that the child exited normally.
- Prohibit `kill()`, `ptrace()`, etc., by setting gid and uid to the target uid.
- Prohibit `fork()`, by setting the current and maximum `RLIMIT_NPROC` limits to 0.
- Set the desired limits on memory allocation and other resource allocation.
- Run the rest of the program.

At this point, unless there are severe operating-system bugs, the program has no communication channels other than its initial file descriptors.

Suppose an attacker supplies a rogue JPEG file that exploits a bug in `jpegtopnm` and succeeds in seizing complete control of the program. The attacker can then generate any output bitmap he wants—but this is what he could have

done without exploiting any bugs. The attacker cannot do anything else. At this point `jpegtopnm` is no longer trusted code; bugs in `jpegtopnm` are no longer capable of violating the user’s security requirements.

As in Section 2.3, I am implicitly assuming that the user’s security requirements allow anyone who influences the JPEG file to control the resulting bitmap. This is a constraint on the security requirements, but I think it is a reasonable one; I have never heard anyone ask for pieces of a JPEG to be protected from each other. More importantly, I see no reason to believe that variations in the security requirements cannot be accommodated by analogous variations in the partitioning of code into sandboxes.

I am also assuming that the CPU does not leak secret information from one process to another. This assumption is debatable. Perhaps access to CPU instructions needs to be limited, for example with the interpreter discussed in Section 2.3. See Section 2.6 for comments on performance.

Ariel Berkman, following my suggestions, has reengineered the standard UNIX `xloadimage` picture-viewing tool, modularizing essentially the entire program into a series of filters such as `jpegtopnm`. Each filter is easily imprisoned by the techniques described above, leaving a much smaller amount of trusted code.

5.3 Delaying multiple-source merges

Consider a mail reader displaying messages from a user’s mailbox. Single-source transformations—for example, uncompressing an attached JPEG file—can be imprisoned as described in Section 5.2. But what about transformations that combine data from multiple sources?

A displayed list of message subjects combines information from several messages. Each message is allowed to control its own entry in the list, but it is not allowed to affect other entries. Even if the list is created correctly in the first place, bugs in subsequent transformations of the list can violate the required separation between list entries.

The situation is quite different if the subjects of separate messages are kept in separate locations, transformed independently, and *then* merged. The transformations are now single-source transformations. Delaying the merge reduces the volume of trusted code.

Stopping cross-site scripting, for example, currently means paying careful attention in every piece of code that merges data from multiple sources into one file, and in every piece of code helping transform that file into a web page. If data from different sources were instead kept separate, and merged only at the last moment by the browser, then care would be required only in the final merging code.

5.4 Do we really need a small TCB?

I failed to place any of the qmail code into untrusted prisons. Bugs anywhere in the code could have been security holes. The way that qmail survived this failure was by having very few bugs, as discussed in Sections 3 and 4.

Perhaps continued reductions of overall code volume and of bug rate will allow the same survival to scale to much larger systems. I’ve heard reports of systems that are believed to be bug-free despite having half a million lines of code. There are many more lines of code in my laptop computer—many more lines of code in a position to violate my security requirements—but it’s *conceivable* that the world could eliminate all the bugs in that code.

However, Section 5.2 shows that a large chunk of code can be eliminated from the TCB at very low cost—surely lower cost, and higher confidence, than eliminating bugs from the code. I’m optimistic about the scalability of this example; I don’t know exactly how small the ultimate TCB will be, but I’m looking forward to finding out. Of course, we still need to eliminate bugs from the code that remains!

6. REFERENCES

- [1] Anurag Acharya, Mandar Raje, *MAPbox: using parameterized behavior classes to confine untrusted applications*, 9th USENIX Security Symposium (2000). URL: <http://www.usenix.org/publications/library/proceedings/sec2000/acharya.html>. Citations in this document: §2.5.
- [2] Vinod Anupam, Alain Mayer, *Security of web browser scripting languages: vulnerabilities, attacks, and remedies*, 7th USENIX Security Symposium (1998). URL: <http://www.usenix.org/publications/library/proceedings/sec98/anupam.html>. Citations in this document: §2.5.
- [3] M. Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, Sheila A. Haghight, *A domain and type enforcement UNIX prototype*, 5th USENIX Security Symposium (1995). URL: <http://www.usenix.org/events/security95/badger.html>. Citations in this document: §2.5.
- [4] Daniel J. Bernstein, *Internet host SMTP server survey*, posting to comp.security.unix, comp.mail.misc, comp.mail.sendmail (1996). URL: <http://cr.yp.to/surveys/smtpsoftware.txt>. Citations in this document: §1.1.
- [5] Daniel J. Bernstein, *Re: Logging question* (1996). URL: <http://www.ornl.gov/lists/mailings-lists/qmail/1996/12/msg00314.html>. Citations in this document: §3.3.
- [6] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in [24] (2006), 207–228. URL: cr.yp.to/papers.html#curve25519. Citations in this document: §2.6.
- [7] Richard Blum, *Running qmail*, Sams Publishing, 2000. ISBN 978-0672319457. Citations in this document: §1.2.
- [8] Computer Emergency Response Team, *CERT advisory CA-2002-19: buffer overflows in multiple DNS resolver libraries* (2002). URL: <http://www.cert.org/advisories/CA-2002-19.html>. Citations in this document: §2.5.
- [9] Edsger W. Dijkstra, *Introducing a course on mathematical methodology*, EWD962 (1986). URL: <http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD962.PDF>. Citations in this document: §4.
- [10] Jeff Gennari, *Vulnerability Note VU#834865: Sendmail contains a race condition* (2006). URL: <http://www.kb.cert.org/vuls/id/834865>. Citations in this document: §1.1.
- [11] Ian Goldberg, David Wagner, Randi Thomas, Eric Brewer, *A secure environment for untrusted helper applications (confining the wily hacker)*, 6th USENIX Security Symposium (1996). URL: <http://www.usenix.org/publications/library/proceedings/sec96/goldberg.html>. Citations in this document: §2.5, §2.5, §2.5, §2.5.
- [12] Internet Security Systems, *Remote Sendmail header processing vulnerability* (2003). URL: <http://www.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=21950>. Citations in this document: §2.3.
- [13] Donald Knuth, *Structured programming with go to statements*, Computing Surveys 6 (1974), 261–301. Citations in this document: §2.6.
- [14] John R. Levine, *qmail*, O’Reilly, 2004. ISBN 978-1565926288. Citations in this document: §1.2.
- [15] Nimisha V. Mehta, Karen R. Sollins, *Expanding and extending the security features of Java*, 7th USENIX Security Symposium (1998). URL: <http://www.usenix.org/publications/library/proceedings/sec98/mehta.html>. Citations in this document: §2.5.
- [16] David S. Peterson, Matt Bishop, Raju Pandey, *A flexible containment mechanism for executing untrusted code*, 11th USENIX Security Symposium (2002). URL: <http://www.usenix.org/events/sec02/peterson.html>. Citations in this document: §2.5.
- [17] Eric Raymond, *The cathedral and the bazaar* (1997). URL: <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>. Citations in this document: §2.1.
- [18] Jerry H. Saltzer, Mike D. Schroeder, *The protection of information in computer systems*, Proceedings of the IEEE 63 (1975), 1278–1308. Citations in this document: §2.5.
- [19] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, David Wagner, *Detecting format string vulnerabilities with type qualifiers*, 10th USENIX Security Symposium (2001). URL: <http://www.usenix.org/publications/library/proceedings/sec01/shankar.html>. Citations in this document: §3.3.
- [20] Dave Sill, *The qmail handbook*, Apress, 2002. ISBN 978-1893115408. Citations in this document: §1.2.
- [21] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, Karen A. Oostendorp, *Confining root programs with domain and type enforcement*, 6th USENIX Security Symposium (1996). URL: <http://www.usenix.org/events/sec96/walker.html>. Citations in this document: §2.5.
- [22] Kyle Wheeler, *Qmail quickstarter: install, set up and run your own email server*, Packt Publishing, 2007. ISBN 978-1847191151. Citations in this document: §1.2.
- [23] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman, *Linux security modules: general security support for the Linux kernel*, 11th USENIX Security Symposium (2002). URL: <http://www.usenix.org/events/sec02/wright.html>. Citations in this document: §2.5.
- [24] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, Berlin, 2006. ISBN 978-3-540-33851-2. See [6].

The Protection of Information in Computer Systems

JEROME H. SALTZER, SENIOR MEMBER, IEEE, AND MICHAEL D. SCHROEDER, MEMBER, IEEE

Invited Paper

Abstract—This tutorial paper explores the mechanics of protecting computer-stored information from unauthorized use or modification. It concentrates on those architectural structures—whether hardware or software—that are necessary to support information protection. The paper develops in three main sections. Section I describes desired functions, design principles, and examples of elementary protection and authentication mechanisms. Any reader familiar with computers should find the first section to be reasonably accessible. Section II requires some familiarity with descriptor-based computer architecture. It examines in depth the principles of modern protection architectures and the relation between capability systems and access control list systems, and ends with a brief analysis of protected subsystems and protected objects. The reader who is dismayed by either the prerequisites or the level of detail in the second section may wish to skip to Section III, which reviews the state of the art and current research projects and provides suggestions for further reading.

GLOSSARY

THE FOLLOWING glossary provides, for reference, brief definitions for several terms as used in this paper in the context of protecting information in computers.

Access	The ability to make use of information stored in a computer system. Used frequently as a verb, to the horror of grammarians.	Authorize	To grant a principal access to certain information.
Access control list	A list of principals that are authorized to have access to some object.	Capability	In a computer system, an unforgeable ticket, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket.
Authenticate	To verify the identity of a person (or other agent external to the protection system) making a request.	Certify	To check the accuracy, correctness, and completeness of a security or protection mechanism.
		Complete isolation	A protection system that separates principals into compartments between which no flow of information or control is possible.
		Confinement	Allowing a borrowed program to have access to data, while ensuring that the program cannot release the information.
		Descriptor	A protected value which is (or leads to) the physical address of some protected object.
		Discretionary	(In contrast with <i>nondiscretionary</i> .) Controls on access to an object that may be changed by the creator of the object.
		Domain	The set of objects that currently may be directly accessed by a principal.
		Encipherment	The (usually) reversible scrambling of data according to a secret transformation key, so as to make it safe for transmission or storage in a physically unprotected environment.
		Grant	To authorize (<i>q.v.</i>).
		Hierarchical control	Referring to ability to change authorization, a scheme in which the record of

Manuscript received October 11, 1974; revised April 17, 1975. Copyright © 1975 by J. H. Saltzer.

The authors are with Project MAC and the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. 02139.

	each authorization is controlled by another authorization, resulting in a hierarchical tree of authorizations.	User	Used imprecisely to refer to the individual who is accountable for some identifiable set of activities in a computer system.
List-oriented	Used to describe a protection system in which each protected object has a list of authorized principals.		
Password	A secret character string used to authenticate the claimed identity of an individual.		
Permission	A particular form of allowed access, e.g., permission to READ as contrasted with permission to WRITE.		
Prescript	A rule that must be followed before access to an object is permitted, thereby introducing an opportunity for human judgment about the need for access, so that abuse of the access is discouraged.		
Principal	The entity in a computer system to which authorizations are granted; thus the unit of accountability in a computer system.		
Privacy	The ability of an individual (or organization) to decide whether, when, and to whom personal (or organizational) information is released.		
Propagation	When a principal, having been authorized access to some object, in turn authorizes access to another principal.		
Protected object	A data structure whose existence is known, but whose internal organization is not accessible, except by invoking the protected subsystem (<i>q.v.</i>) that manages it.		
Protected subsystem	A collection of procedures and data objects that is encapsulated in a domain of its own so that the internal structure of a data object is accessible only to the procedures of the protected subsystem and the procedures may be called only at designated domain entry points.		
Protection	1) Security (<i>q.v.</i>). 2) Used more narrowly to denote mechanisms and techniques that control the access of executing programs to stored information.		
Protection group	A principal that may be used by several different individuals.		
Revoke	To take away previously authorized access from some principal.		
Security	With respect to information processing systems, used to denote mechanisms and techniques that control who may use or modify the computer or the information stored in it.		
Self control	Referring to ability to change authorization, a scheme in which each authorization contains within it the specification of which principals may change it.		
Ticket-oriented	Used to describe a protection system in which each principal maintains a list of unforgeable bit patterns, called tickets, one for each object the principal is authorized to have access.		

I. BASIC PRINCIPLES OF INFORMATION PROTECTION

A. Considerations Surrounding the Study of Protection

1) *General Observations:* As computers become better understood and more economical, every day brings new applications. Many of these new applications involve both storing information and simultaneous use by several individuals. The key concern in this paper is multiple use. For those applications in which all users should not have identical authority, some scheme is needed to ensure that the computer system implements the desired authority structure.

For example, in an airline seat reservation system, a reservation agent might have authority to make reservations and to cancel reservations for people whose names he can supply. A flight boarding agent might have the additional authority to print out the list of all passengers who hold reservations on the flights for which he is responsible. The airline might wish to withhold from the reservation agent the authority to print out a list of reservations, so as to be sure that a request for a passenger list from a law enforcement agency is reviewed by the correct level of management.

The airline example is one of protection of corporate information for corporate self-protection (or public interest, depending on one's view). A different kind of example is an on-line warehouse inventory management system that generates reports about the current status of the inventory. These reports not only represent corporate information that must be protected from release outside the company, but also may indicate the quality of the job being done by the warehouse manager. In order to preserve his personal privacy, it may be appropriate to restrict the access to such reports, even within the company, to those who have a legitimate reason to be judging the quality of the warehouse manager's work.

Many other examples of systems requiring protection of information are encountered every day: credit bureau data banks; law enforcement information systems; time-sharing service bureaus; on-line medical information systems; and government social service data processing systems. These examples span a wide range of needs for organizational and personal privacy. All have in common controlled sharing of information among multiple users. All, therefore, require some plan to ensure that the computer system helps implement the correct authority structure. Of course, in some applications no special provisions in the computer system are necessary. It may be, for instance, that an externally administered code of ethics or a lack of knowledge about computers adequately protects the stored information. Although there are situations in which the computer need provide no aids to ensure protection of information, often it is appropriate to have the computer enforce a desired authority structure.

The words "privacy," "security," and "protection" are frequently used in connection with information-storing systems. Not all authors use these terms in the same way. This paper uses definitions commonly encountered in computer science literature.

The term "privacy" denotes a socially defined ability of an individual (or organization) to determine whether, when, and

to whom personal (or organizational) information is to be released.

This paper will not be explicitly concerned with privacy, but instead with the mechanisms used to help achieve it.¹

The term "security" describes techniques that control who may use or modify the computer or the information contained in it.²

Security specialists (e.g., Anderson [6]) have found it useful to place potential security violations in three categories.

1) Unauthorized information release: an unauthorized person is able to read and take advantage of information stored in the computer. This category of concern sometimes extends to "traffic analysis," in which the intruder observes only the patterns of information use and from those patterns can infer some information content. It also includes unauthorized use of a proprietary program.

2) Unauthorized information modification: an unauthorized person is able to make changes in stored information—a form of sabotage. Note that this kind of violation does not require that the intruder see the information he has changed.

3) Unauthorized denial of use: an intruder can prevent an authorized user from referring to or modifying information, even though the intruder may not be able to refer to or modify the information. Causing a system "crash," disrupting a scheduling algorithm, or firing a bullet into a computer are examples of denial of use. This is another form of sabotage.

The term "unauthorized" in the three categories listed above means that release, modification, or denial of use occurs contrary to the desire of the person who controls the information, possibly even contrary to the constraints supposedly enforced by the system. The biggest complication in a general-purpose remote-accessed computer system is that the "intruder" in these definitions may be an otherwise legitimate user of the computer system.

Examples of security techniques sometimes applied to computer systems are the following:

- 1) labeling files with lists of authorized users,
- 2) verifying the identity of a prospective user by demanding a password,
- 3) shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,
- 4) enciphering information sent over telephone lines,
- 5) locking the room containing the computer,
- 6) controlling who is allowed to make changes to the computer system (both its hardware and software),
- 7) using redundant circuits or programmed cross-checks that maintain security in the face of hardware or software failures,
- 8) certifying that the hardware and software are actually implemented as intended.

It is apparent that a wide range of considerations are pertinent to the engineering of security of information. Historically, the

¹A thorough and scholarly discussion of the concept of privacy may be found in [1], and an interesting study of the impact of technology on privacy is given in [2]. In 1973, the U.S. Department of Health, Education, and Welfare published a related study [3]. A recent paper by Turn and Ware [4] discusses the relationship of the social objective of privacy to the security mechanisms of modern computer systems.

²W. Ware [5] has suggested that the term *security* be used for systems that handle classified defense information, and *privacy* for systems handling nondefense information. This suggestion has never really taken hold outside the defense security community, but literature originating within that community often uses Ware's definitions.

literature of computer systems has more narrowly defined the term *protection* to be just those security techniques that control the access of executing programs to stored information.³ An example of a protection technique is labeling of computer-stored files with lists of authorized users. Similarly, the term *authentication* is used for those security techniques that verify the identity of a person (or other external agent) making a request of a computer system. An example of an authentication technique is demanding a password. This paper concentrates on protection and authentication mechanisms, with only occasional reference to the other equally necessary security mechanisms. One should recognize that concentration on protection and authentication mechanisms provides a narrow view of information security, and that a narrow view is dangerous. The objective of a secure system is to prevent all unauthorized use of information, a negative kind of requirement. It is hard to prove that this negative requirement has been achieved, for one must demonstrate that every possible threat has been anticipated. Thus an expansive view of the problem is most appropriate to help ensure that no gaps appear in the strategy. In contrast, a narrow concentration on protection mechanisms, especially those logically impossible to defeat, may lead to false confidence in the system as a whole.⁴

2) *Functional Levels of Information Protection*: Many different designs have been proposed and mechanisms implemented for protecting information in computer systems. One reason for differences among protection schemes is their different functional properties—the kinds of access control that can be expressed naturally and enforced. It is convenient to divide protection schemes according to their functional properties. A rough categorization is the following.

a) Unprotected systems: Some systems have no provision for preventing a determined user from having access to every piece of information stored in the system. Although these systems are not directly of interest here, they are worth mentioning since, as of 1975, many of the most widely used, commercially available batch data processing systems fall into this category—for example, the Disk Operating System for the IBM System 370 [9]. Our definition of protection, which excludes features usable only for mistake prevention, is important here since it is common for unprotected systems to contain a variety of mistake-prevention features. These may provide just enough control that any breach of control is likely to be the result of a deliberate act rather than an accident. Neverthe-

³Some authors have widened the scope of the term "protection" to include mechanisms designed to limit the consequences of accidental mistakes in programming or in applying programs. With this wider definition, even computer systems used by a single person might include "protection" mechanisms. The effect of this broader definition of "protection" would be to include in our study mechanisms that may be deliberately bypassed by the user, on the basis that the probability of accidental bypass can be made as small as desired by careful design. Such accident-reducing mechanisms are often essential, but one would be ill-advised to apply one to a situation in which a systematic attack by another user is to be prevented. Therefore, we will insist on the narrower definition. Protection mechanisms are very useful in preventing mistakes, but mistake-preventing mechanisms that can be deliberately bypassed have little value in providing protection. Another common extension of the term "protection" is to techniques that ensure the reliability of information storage and computing service despite accidental failure of individual components or programs. In this paper we arbitrarily label those concerns "reliability" or "integrity," although it should be recognized that historically the study of protection mechanisms is rooted in attempts to provide reliability in multiprogramming systems.

⁴The broad view, encompassing all the considerations mentioned here and more, is taken in several current books [6]–[8].

less, it would be a mistake to claim that such systems provide any security.⁵

b) All-or-nothing systems: These are systems that provide isolation of users, sometimes moderated by total sharing of some pieces of information. If only isolation is provided, the user of such a system might just as well be using his own private computer, as far as protection and sharing of information are concerned. More commonly, such systems also have public libraries to which every user may have access. In some cases the public library mechanism may be extended to accept user contributions, but still on the basis that all users have equal access. Most of the first generation of commercial time-sharing systems provide a protection scheme with this level of function. Examples include the Dartmouth Time-Sharing System (DTSS) [10] and IBM's VM/370 system [11]. There are innumerable others.

c) Controlled sharing: Significantly more complex machinery is required to control explicitly who may access each data item stored in the system. For example, such a system might provide each file with a list of authorized users and allow an owner to distinguish several common patterns of use, such as reading, writing, or executing the contents of the file as a program. Although conceptually straightforward, actual implementation is surprisingly intricate, and only a few complete examples exist. These include M.I.T.'s Compatible Time-Sharing System (CTSS) [12], Digital Equipment Corporation's DECsystem/10 [13], System Development Corporation's Advanced Development Prototype (ADEPT) System [14], and Bolt, Beranek, and Newman's TENEX [15].⁶

d) User-programmed sharing controls: A user may want to restrict access to a file in a way not provided in the standard facilities for controlling sharing. For example, he may wish to permit access only on weekdays between 9:00 A.M. and 4:00 P.M. Possibly, he may wish to permit access to only the average value of the data in a file. Maybe he wishes to require that a file be modified only if two users agree. For such cases, and a myriad of others, a general escape is to provide for user-defined *protected objects* and *subsystems*. A *protected subsystem* is a collection of programs and data with the property that only the programs of the subsystem have direct access to the data (that is, the protected objects). Access to those programs is limited to calling specified entry points. Thus the programs of the subsystem completely control the operations performed on the data. By constructing a protected subsystem, a user can develop any programmable form of access control to the objects he creates. Only a few of the most advanced system designs have tried to permit user-specified protected subsystems. These include Honeywell's Multics [16], the University of California's CAL system [17], Bell Laboratories' UNIX system [18], the Berkeley Computer Corporation BCC-500 [19], and two systems currently under construction: the CAP system of Cambridge University [20], and the HYDRA system of Carnegie-Mellon University [21]. Ex-

⁵One can develop a spirited argument as to whether systems originally designed as unprotected, and later modified to implement some higher level of protection goal, should be reclassified or continue to be considered unprotected. The argument arises from skepticism that one can successfully change the fundamental design decisions involved. Most large-scale commercial batch processing systems fall into this questionable area.

⁶An easier-to-implement strategy of providing shared catalogs that are accessible among groups of users who anticipate the need to share was introduced in CTSS in 1962, and is used today in some commercial systems.

ploring alternative mechanisms for implementing protected subsystems is a current research topic. A specialized use of protected subsystems is the implementation of protection controls based on data content. For example, in a file of salaries, one may wish to permit access to all salaries under \$15 000. Another example is permitting access to certain statistical aggregations of data but not to any individual data item. This area of protection raises questions about the possibility of discerning information by statistical tests and by examining indexes, without ever having direct access to the data itself. Protection based on content is the subject of a variety of recent or current research projects [22]–[25] and will not be explored in this tutorial.

e) Putting strings on information: The foregoing three levels have been concerned with establishing conditions for the release of information to an executing program. The fourth level of capability is to maintain some control over the user of the information even *after* it has been released. Such control is desired, for example, in releasing income information to a tax advisor; constraints should prevent him from passing the information on to a firm which prepares mailing lists. The printed labels on classified military information declaring a document to be "Top Secret" are another example of a constraint on information after its release to a person authorized to receive it. One may not (without risking severe penalties) release such information to others, and the label serves as a notice of the restriction. Computer systems that implement such strings on information are rare and the mechanisms are incomplete. For example, the ADEPT system [14] keeps track of the classification level of all input data used to create a file; all output data are automatically labeled with the highest classification encountered during execution.

There is a consideration that cuts across all levels of functional capability: the *dynamics of use*. This term refers to how one establishes and changes the specification of who may access what. At any of the levels it is relatively easy to envision (and design) systems that statically express a particular protection intent. But the need to change access authorization dynamically and the need for such changes to be requested by executing programs introduces much complexity into protection systems. For a given functional level, most existing protection systems differ primarily in the way they handle protection dynamics. To gain some insight into the complexity introduced by program-directed changes to access authorization, consider the question "Is there any way that O'Hara could access file *X*?" One should check to see not only if O'Hara has access to file *X*, but also whether or not O'Hara may change the specification of file *X*'s accessibility. The next step is to see if O'Hara can change the specification of who may change the specification of file *X*'s accessibility, etc. Another problem of dynamics arises when the owner revokes a user's access to a file while that file is being used. Letting the previously authorized user continue until he is "finished" with the information may not be acceptable, if the owner has suddenly realized that the file contains sensitive data. On the other hand, immediate withdrawal of authorization may severely disrupt the user. It should be apparent that provisions for the dynamics of use are at least as important as those for static specification of protection intent.

In many cases, it is not necessary to meet the protection needs of the person responsible for the information stored in the computer entirely through computer-aided enforcement. External mechanisms such as contracts, ignorance, or barbed

wire fences may provide some of the required functional capability. This discussion, however, is focused on the internal mechanisms.

3) *Design Principles*: Whatever the level of functionality provided, the usefulness of a set of protection mechanisms depends upon the ability of a system to prevent security violations. In practice, producing a system at any level of functionality (except level one) that actually does prevent all such unauthorized acts has proved to be extremely difficult. Sophisticated users of most systems are aware of at least one way to crash the system, denying other users authorized access to stored information. Penetration exercises involving a large number of different general-purpose systems all have shown that users can construct programs that can obtain unauthorized access to information stored within. Even in systems designed and implemented with security as an important objective, design and implementation flaws provide paths that circumvent the intended access constraints. Design and construction techniques that systematically exclude flaws are the topic of much research activity, but no complete method applicable to the construction of large general-purpose systems exists yet. This difficulty is related to the negative quality of the requirement to prevent *all* unauthorized actions.

In the absence of such methodical techniques, experience has provided some useful principles that can guide the design and contribute to an implementation without security flaws. Here are eight examples of design principles that apply particularly to protection mechanisms.⁷

a) **Economy of mechanism**: Keep the design as simple and small as possible. This well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths). As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.

b) **Fail-safe defaults**: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965,⁸ means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situa-

⁷Design principles b), d), f), and h) are revised versions of material originally published in *Communications of the ACM* [26, p. 398]. © Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission.

⁸In this paper we have attempted to identify original sources whenever possible. Many of the seminal ideas, however, were widely spread by word of mouth or internal memorandum rather than by journal publication, and historical accuracy is sometimes difficult to obtain. In addition, some ideas related to protection were originally conceived in other contexts. In such cases, we have attempted to credit the person who first noticed their applicability to protection in computer systems, rather than the original inventor.

tion, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

c) **Complete mediation**: Every access to every object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It implies that a fool-proof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.

d) **Open design**: The design should not be secret [27]. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose.⁹ Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

e) **Separation of privilege**: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extensible protected data types usually depend on separation of privilege for their implementation.

f) **Least privilege**: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale

⁹We should note that the principle of open design is not universally accepted, especially by those accustomed to dealing with military security. The notion that the mechanism not depend on ignorance is generally accepted, but some would argue that its design should remain secret. The reason is that a secret design may have the additional advantage of significantly raising the price of penetration, especially the risk of detection.

for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.

g) Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users [28]. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it.

h) Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.

Analysts of traditional physical security systems have suggested two further design principles which, unfortunately, apply only imperfectly to computer systems.

a) Work factor: Compare the cost of circumventing the mechanism with the resources of a potential attacker. The cost of circumventing, commonly known as the "work factor," in some cases can be easily calculated. For example, the number of experiments needed to try all possible four-letter alphabetic passwords is $26^4 = 456\,976$. If the potential attacker must enter each experimental password at a terminal, one might consider a four-letter password to be adequate. On the other hand, if the attacker could use a large computer capable of trying a million passwords per second, as might be the case where industrial espionage or military security is being considered, a four-letter password would be a minor barrier for a potential intruder. The trouble with the work factor principle is that many computer protection mechanisms are *not* susceptible to direct work factor calculation, since defeating them by systematic attack may be logically impossible. Defeat can be accomplished only by indirect strategies, such as waiting for an accidental hardware failure or searching for an error in implementation. Reliable estimates of the length of such a wait or search are very difficult to make.

b) Compromise recording: It is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss. For example, if a tactical plan is known to have been compromised, it may be possible to construct a different one, rendering the compromised version worthless. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism. Although the information stored inside may be easy to obtain, the cabinet will inevitably be damaged in the process and the next legitimate user will detect the loss. For another example, many computer systems record the date and time of the most recent use of each file. If this record is tamperproof and re-

ported to the owner, it may help discover unauthorized use. In computer systems, this approach is used rarely, since it is difficult to guarantee discovery once security is broken. Physical damage usually is not involved, and logical damage (and internally stored records of tampering) can be undone by a clever attacker.¹⁰

As is apparent, these principles do not represent absolute rules—they serve best as warnings. If some part of a design violates a principle, the violation is a symptom of potential trouble, and the design should be carefully reviewed to be sure that the trouble has been accounted for or is unimportant.

4) *Summary of Considerations Surrounding Protection:* Briefly, then, we may outline our discussion to this point. The application of computers to information handling problems produces a need for a variety of security mechanisms. We are focusing on one aspect, computer protection mechanisms—the mechanisms that control access to information by executing programs. At least four levels of functional goals for a protection system can be identified: all-or-nothing systems, controlled sharing, user-programmed sharing controls, and putting strings on information. But at all levels, the provisions for dynamic changes to authorization for access are a severe complication.

Since no one knows how to build a system without flaws, the alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws: Economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability.

Finally, some protection designs can be evaluated by comparing the resources of a potential attacker with the work factor required to defeat the system, and compromise recording may be a useful strategy.

¹⁰An interesting suggestion by Hollingsworth [29] is to secretly design what appear to be compromisable implementation errors, along with monitors of attempted exploitation of the apparent errors. The monitors might then provide early warning of attempts to violate system security. This suggestion takes us into the realm of counterintelligence, which is beyond our intended scope.