Atitit 代码之美 11章 第11章 安全通信：自由的技术

CHAPTER ELEVEN

**Secure Communication:**

**The Technology Of Freedom**

*Ashish Gulhati*

I speak of none other than the computer that is to come after me. A

computer whose merest operational parameters I am not worthy to calcu

late—and yet I will design it for you. A computer which can calculate the

Question to the Ultimate Answer, a computer of such infinite and subtle

complexity that organic life itself shall form part of its operational matrix.

*Deep Thought,* The Hitchhiker’s Guide to the Galaxy

**I**

**N MID**

**-1999 I FLEW TO COSTA RICA TO WORK WITH LAISSEZ FAIRE CITY,** a group that was work

ing to create software systems to help usher in a new era of individual sovereignty.

\*

The group at LFC was working primarily to develop a suite of software designed to protect

and enhance individual rights in the digital age, including easy-to-use secure email, online

dispute mediation services, an online stock exchange, and a private asset trading and

banking system. My interest in many of the same technologies had been piqued long ago

by the cypherpunks list and Bruce Schneier’s *Applied Cryptography* (Wiley), and I’d already

been working on prototype implementations of some of these systems.

The most fundamental of these were systems to deliver strong and *usable* communications

privacy to just about everybody.

When I stepped into LFC’s sprawling “interim consulate” outside San José, Costa Rica,

they had a working prototype of a secure webmail system they called MailVault. It ran on

Mac OS 9, used FileMaker as its database, and was written in Frontier. Not at all the mix

\* See *The Sovereign Individual: Mastering the Transition to the Information Age*, James Dale Davidson and

Sir William Rees Mogg, Free Press, 1999.**162**

CHAPTER ELEVEN

of technologies you’d want to run a mission-critical communications service on, but that’s

what the programmers had produced.

It was no surprise the system crashed early and often, and was extremely fragile. It could

hardly support two concurrent users. LFC was facing a credibility crisis with its investors,

as their software releases had been delayed many times, and their first beta of MailVault,

the flagship product, was no gem. So in the free time left over from my contract network

and system administration work at LFC, I started writing a new secure mail system from

scratch.

This system is now named Cryptonite and has been in constant off-and-on development

and testing since then, in between other projects.

The first functioning prototype of Cryptonite was licensed to LFC as MailVault beta 2, and

was open for testing in September 1999. It was the first OpenPGP-compatible webmail

system available for public use and was almost immediately put to the test by LFC’s inves

tors and beta testers. Since that time, Cryptonite has evolved in many ways through inter

action with users, the open source community, and the market. While not an open source

product itself, it has led to the development of numerous components I decided to release

as open source along the way.

**The Heart of the Start**

Developing Cryptonite and marketing and supporting associated services single-handedly

for many years (with unwavering support and many invaluable ideas from my wife,

Barkha) has been an incredibly interesting and rewarding journey, not only from a devel

opment perspective but also from an entrepreneurial one.

Before jumping into the nitty-gritty of the system, I thought I’d touch upon some points

that have impressed themselves strongly in my consciousness over the course of this

project:

•

My friend Rishab Ghosh once quipped that there’s a lot of hype about how the Internet

can enable wired hackers to work from anywhere, but most of the people who create

this hype live within a small area in California. The great thing about an independent

startup project is that it really can be done anywhere, and dropped and picked up again

when convenient. I’ve hacked on Cryptonite over many years on four continents, and

it may well be the first high-quality software application developed in large part in the

Himalayan mountains. (I used the word “wired” before loosely. In reality, five wireless

technologies facilitated our connectivity in the Himalayas: a VSAT satellite Internet

link, Wi-Fi, Bluetooth, GPRS, and CDMA.)

•

When working on a project as a single developer in your spare time, remember the old

hacker wisdom that “six months in the lab can save you ten minutes in the library.” It’s

critical to maximize your reuse of existing code libraries. For this reason, I elected to

develop the system in Perl, a popular and flexible high-level language with a rich

library of mature, free software modules, and the Perl hacker’s first virtue of laziness

informed every design decision.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**163**

•

Especially for end-user application software, ease of use is a critical issue. It is essential

to the function of such code to present a simple, accessible interface to the user. The

usability considerations of developing an end-user security application are even more

significant, and were in fact a key factor in making the Cryptonite system worth

developing.

•

To get off to a running start, it’s a good idea to implement a working prototype first,

and use a prototype-to-production path to move to production deployment after the

essential functionality is implemented. This can be a huge help in getting the basic

design and structure right before you unleash the code on hundreds or (hopefully!)

millions of users.

•

Keeping your system as simple as possible is always a great idea. Resist the urge to get

suckered into using the latest complex buzzword technology, unless the application

really demands it.

•

Processors are pretty fast now, and programmer time is generally more valuable than

processor time, but speed is still critical for application software. Users expect their

applications to be snappy. For web applications, which many users will use concur

rently, investing some time in optimizing for speed is a Very Good Thing.

•

A software application is a living entity, in constant need of attention, updating,

enhancement, testing, fixing, tweaking, marketing, and support. Its success and beauty

in an economic sense depends directly on the code being flexible enough to evolve over

time and meet the requirements of its users, and to do it again and again and again

over the course of many years.

•

It really does help if the problem you’re trying to solve is something that personally

interests you. This not only makes it possible to flip between user and developer roles

easily, but ensures you’ll still be interested in the project five years later—because

building and marketing a software application is generally quite a long-term

proposition.

The development of Cryptonite has been powered in large measure by my desire to create

tools to help individuals all over the world achieve practical liberty. And while developing

the system single-handedly has been difficult at times, I find that being a single-developer

project has also given the code a certain stylistic and structural unity that’s rare in code

developed by multiple programmers.

**Untangling the Complexity of Secure Messaging**

While bringing secure communications capabilities to the world is a whoppingly great idea

for the protection of individual human rights (more on this later), getting it right is a trick

ier task than it may seem. Public-key cryptosystems can, in principle, facilitate ad hoc

secure communications, but practical implementations are very often needlessly complex

and disconnected from on-the-ground realities concerning who will use such systems, and

how.**164**

CHAPTER ELEVEN

The fundamental problem to be solved in practical implementations based on public-key

cryptography is key authentication. To send an encrypted message to someone, you need

her public key. If you can be tricked into using the wrong public key, your privacy

vanishes.

There are two very different approaches to the key authentication problem.

The conventional Public Key Infrastructure (PKI) approach, typically based on ISO stan

dard X.509, depends on a system of trusted third-party Certification Authorities (CAs),

and is in many ways fundamentally unsuited to meet the real needs of users in ad hoc net

works.\* PKI implementations have achieved significant success in more structured

domains, such as corporate VPNs and the authentication of secure web sites, but have

made little headway in the real-world heterogeneous email environment.

The other approach is exemplified by the most popular public-key-based messaging secu

rity solution in use today: Phil Zimmermann’s PGP and its descendants, now formalized as

the IETF OpenPGP protocol. OpenPGP preserves the flexibility and fundamentally decen

tralized nature of public-key cryptography by facilitating distributed key authentication

through “webs of trust” rather than depending on a centralized, hierarchical system of

CAs, as PKI approaches do (including OpenPGP’s primary competitor, S/MIME). Not sur

prisingly, S/MIME, which is almost ubiquitously available in popular email clients, enjoys

a vastly smaller user base than OpenPGP, despite email clients’ general lack of comprehen

sive support for OpenPGP.

But the web-of-trust approach, which relies on users to build their own chains of trust for

certifying and authenticating public keys, has its own issues. Prime among these are the

interrelated challenges of ensuring that users understand how to use the web of trust to

authenticate keys, and the need to achieve a critical mass of users in order to ensure that

any two users can easily find a trust path between each other.

As Figure 11-1 shows, in a web of trust implementation, no third parties are arbitrarily

designated as “trusted.” Each individual user is her own most trusted certifying authority,

and may assign varying levels of trust to others for the purpose of validating keys. You

consider a key valid if it is certified directly by you, by another person who is fully trusted

by you to certify keys, or by a user-definable number of people, each of whom is partially

trusted by you to certify keys.

Because the web-of-trust approach doesn’t attempt to outsource key authentication the

way PKI approaches do, users must play a central role in building their webs of trust and

ascertaining the authenticity of public keys. This puts usability considerations front and

center in the design of OpenPGP-based secure messaging systems.

\* The drawbacks of conventional PKI have been concisely summarized by Roger Clarke at *http://*

*www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html.*SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**165**

**Usability Is the Key**

Email privacy software often requires users to jump through too many hoops, so very few

bother to use it. Usability is critical to the success of any security solution, because if the

system isn’t usable, it will end up being bypassed or used in an insecure manner, in either

case defeating its whole purpose.

A case study of the usability of PGP conducted at Carnegie Mellon University in 1998

pointed out the specialized challenges of creating an effective and usable interface for

email encryption and found that of 12 study participants, all of whom were experienced at

using email, “only one-third of them were able to use PGP to correctly sign and encrypt an

email message when given 90 minutes in which to do so.”\*

I saw Cryptonite as an interesting project in terms of designing a secure, reliable, and effi

cient email system while achieving a very high level of usability. I set out to create a web

mail system that would embed OpenPGP security into the very structure of the email

experience, and help even casual users to effectively utilize OpenPGP to achieve commu

nications privacy. The webmail format was chosen specifically because it could bring pow

erful communications privacy technology to anyone with access to an Internet café, or a

cellphone with a web browser, not just to the few able to run desktop email encryption

software on powerful computers.

*FIGURE 11-1* . *How keys are validated through the web of trust*

\* “Usability of Security: A Case Study.” Alma Whitten and J. D. Tygar, Carnegie Mellon University.

*http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-155.pdf*.

CHAPTER ELEVEN

Cryptonite was designed to make encryption a normal part of everyday email, not by

masking the complexities of the public-key cryptosystems that it relies on, but rather by

making the elements of these systems clearer and more accessible to the user. Usability

considerations were thus central to Cryptonite’s design and development, as was mani

fested in a number of ways:

*Development of UI functionality from user feedback and usability studies*

The CMU user study provided many good ideas for the initial design, and many fea

tures evolved out of usability testing with Cryptonite itself by casual email users. The

interface was kept clean, minimalist, and consistent, with all important actions being at

most one or two clicks away at all times.

Significant insights gleaned from usability testing included the need to integrate key

management into the email client, the need to offer persistence for decrypted messages,

and the desirability of exposing message structure information in the message list view.

The final three-pane layout, similar to that found on desktop email programs, was

decided on after testing a simple single-pane HTML interface as well as an AJAX inter

face. The three-pane interface optimized the user’s experience by not forcing a page

reload every time one returned to the message list, as a single-pane design does, and a

simple three-pane HTML interface was both more portable and cleaner to implement

than an AJAX one, while not being much more bandwidth-intensive.

*Rich and meaningful exposure of OpenPGP objects to the user in an intuitive way*

All key operations are available to the user, including generating, importing and

exporting keys; checking key signatures and fingerprints; certifying keys and revoking

key certifications; and publishing keys to and retrieving them from a key server. This

puts the user in full control of her own web of trust. The validity and trust levels of

keys are visible explicitly in text, as well as by color-coding in the key list. Key trust val

ues are always kept updated with the latest state of the key ring and trust database.

The UI’s Key Ring view, illustrated in Figure 11-2, shows the validity of all user identi

ties for each key, both in text and by color-coding. It also shows the key type, using

icons, and owner trust values for each key (both in text and by color-coding). Full

details for any key are available through the “edit” link for the key.

*Warnings and feedback about security implications of user actions*

Giving users the power to manage keys brings the risk that they will use their abilities

in ways that weaken the security of the system. So, it is also the application’s job to

educate the user about security implications of actions such as certifying a key, altering

a key’s trust level, or signing a message.

All screens in Cryptonite that allow for actions with security implications contain short,

highlighted warnings about these implications. And they’re right on the same screen,

not in irritating pop-up boxes.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**167**

*Built-in associations*

Cryptonite’s concept of a user’s identity is strongly tied to the private keys in the user’s

key ring. When sending mail, users can use any “From” address that corresponds to a

private key in their key ring. This helps the user grasp in an intuitive and inescapable

way the idea of a private key. Public keys can be tied to contacts in the user’s address

book, so they can be picked up for automatic encryption whenever available.

*Full-featured email client*

Cryptonite is primarily an email client that just happens to have complete support for

OpenPGP-based security and key management built in. An important usability goal

was to provide the user with a full-featured email client without letting the security

functionality get in the way of its usability for email. This required not only providing

the full range of features a user would expect to find in an email client but, most signif

icantly, enabling users to search through their mail folders, including text within

encrypted messages, without much more complexity than a regular email client where

all messages are stored unencrypted.

**The Foundation**

Application software today, of course, is many levels removed from the bare hardware

and builds on top of many layers of existing code. So when starting a new project, getting

the foundation right has to be the crucial starting point.

*FIGURE 11-2* . *The Key Ring view exposes information on keys and trust***168**

CHAPTER ELEVEN

For a number of reasons, I chose to write Cryptonite in Perl. The rich pool of open source

reusable modules on CPAN (*http://www.cpan.org*) helped minimize the need to write new

code where existing solutions could be leveraged, and also allowed a great deal of flexibil

ity in interfaces and options. This was borne out well by prior experience with the lan

guage as well as by later experiences with the Cryptonite project.

The ability to interface to C and other libraries through Perl’s XS API allowed access to

even more libraries. Perl’s excellent portability and robust support for object-oriented pro

gramming were other important advantages. Cryptonite was intended to be easily modifi

able by licensees, which would also be facilitated by writing it in Perl.

So, the Cryptonite system is implemented entirely in object-oriented Perl. The project has

led to the creation of numerous open source Perl modules, which I have made available

on CPAN.

GNU/Linux jumped out as the obvious development platform, because code developed on

a Unix-like environment would be easiest to port to whatever deployment platform it

would be used on, which could only be another Unix-like platform. No Windows or Mac

system at the time (OS X was in pre-beta) had what it took to run mission-critical software

to be used concurrently by thousands of users. Linux was my preferred desktop environ

ment anyway, so it was also the default choice.

In 2001, development and deployment moved to OpenBSD, and since 2003, development

has proceeded on OS X and OpenBSD (as well as Linux). OS X was chosen for its out-of

box usability as a portable primary desktop, combined with its Unix-like underpinnings

and ability to run a wide variety of open source software. OpenBSD was chosen as a

deployment platform for its reliability, superlative security record, and focus on code qual

ity and code auditing.

The IDE used for development was Emacs, selected for its power, extensibility, and excel

lent portability, including portability to handheld and wearable devices that I often used

for development on the move. I also appreciated the availability of Emacs’s *cperl* mode,

which manages to offer pretty good auto-formatting for Perl code, even though “only *perl*

can parse Perl.”

**Design Goals and Decisions**

Cryptonite was envisioned as an OpenPGP-compatible webmail system designed to be

secure, scalable, reliable, and easy to use. Portability and extensibility were other impor

tant goals of the project.

A key decision made early on was to develop a fully independent core engine to facilitate

interface diversity and cross-platform access. It was important for interface specialists to be

able to build interfaces without needing to modify the core. Clean separation of the core

from the interface would allow experimentation with a variety of interface styles, which

could then be subjected to usability testing to help evolve the optimal interface. This sepa

ration is also the essential design feature that will enable a diversity of interfaces to be built

in the future, including interfaces designed for small devices such as cellphones and PDAs.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**169**

This design called for a client-server system, with a well-defined internal API and a clear

separation of functionality and privilege between the Cryptonite engine and the user

interface. Interfaces to the core could then be implemented in any language with any UI

framework. A reference interface would be developed to enable live usability testing.

Another consideration was to enable flexibility in deployment, by providing the option to

perform cryptographic operations either on the server or on the user’s own machine. Both

approaches have their advantages and drawbacks.

While in principle it is desirable to restrict cryptographic operations to the user’s machine,

these machines in practice are very often physically insecure and riddled with spyware.

The server, on the other hand, can benefit from both high physical security and dedicated

software maintenance by experts, making server-side cryptography (especially in conjunc

tion with hardware token authentication) a more secure option for many users. This was

another reason behind the choice of Perl as the implementation language: its high porta

bility would make it possible to run the application (or components of it) on both server

and user machines, as needed.

An object-oriented implementation would help keep the code easy to comprehend,

extend, maintain, and modify over many years. As the code would be available in source

form to licensees and end users, readability and accessibility of the code were themselves

important objectives.

**Basic System Design**

The initial design of Cryptonite is shown in Figure 11-3.

*FIGURE 11-3* . *The initial design of Cryptonite (C::M is shorthand for Cryptonite::Mail)*

**cryptonite**

(mod\_perl)

C::M::HTML

**cmaild**

C::M::Server C::M::Service

C::M::User C::M::Config Mail::Folder::Mbox

User DB Key Ring PGP Crypt::PGP MIME::\*

**User170**

CHAPTER ELEVEN

Most of the work is done by the Cryptonite::Mail::Service class, which defines a high-level

service object that implements all the core functionality of the Cryptonite system. The

methods of this class simply perform operations based on their arguments and return a

status code and the results of the operation, if any. All the methods are noninteractive,

and there is no user interface code in this class:

Cryptonite::Mail::Service encapsulates all the core functionality of the system, including

user creation and management; creating, opening and closing folders; sending, deleting

and copying mail; encryption, decryption and signature verification; and parsing multipart

MIME messages.

The Service class is used by Cryptonite::Mail::Server to implement a server that receives

serialized Cryptonite API calls and dispatches them to a Service object.

Serialization was initially achieved via SOAP calls, but the SOAP object parsing and han

dling added too much needless complexity and overhead. So, a simple home-brewed seri

alization scheme was implemented instead. (Seven years in, this looks like a really good

move, judging from *http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for*

*simple* and its comments.) This is the command dispatcher in Cryptonite::Mail::Server:

The Cryptonite Mail Daemon (*cmaild*) receives serialized method calls via Unix or TCP

sockets, calls the method on the service object, and returns a result code (+OK or -ERR)

along with a human-readable status message (e.g., “Everything is under control!”) and

optional return values (such as a list of messages in a folder, or the text of a message part).

If multiple lines of return values are being returned, the status message indicates how

many lines the client should expect to read.

The server forks a new process every time a new client connects, so Perl’s built-in *alarm*

function is used to send each new server process a SIGALRM *$timeout* seconds after the

last message received from the client, which causes the server to time out and disconnect

the client.**172**

CHAPTER ELEVEN

**The Test Suite**

Because automated testing is a crucial component of long-term development, I developed

a test suite simultaneously with the project code.

The clean separation of the core from the interface makes it easy to test both components

separately, as well as to quickly diagnose bugs and pinpoint where they are in the code.

Writing tests for *cmaild* is just a matter of calling its methods with valid (or invalid) inputs

and making sure that the return codes and values are as expected.

The test suite for *cmaild* uses the client API calls cmdopen (to open a connection to the Cryp

tonite Mail Daemon), cmdsend (to send an API call to the daemon), and cmdayt (to send an

“Are you there?” ping to the server):

use strict;

use Test;

BEGIN { plan tests => 392, todo => [] }

use Cryptonite::Mail::HTML qw (&cmdopen &cmdsend &cmdayt);

$Test::Harness::Verbose = 1;

my ($cmailclient, $select, $sessionkey);

my ($USER, $CMAILID, $PASSWORD) = 'test';

my $a = $Cryptonite::Mail::Config::CONFIG{ADMINPW};

ok(sub { # 1: cmdopen

my $status;

($status, $cmailclient, $select) = cmdopen;

return $status unless $cmailclient;

1;

}, 1);

ok(sub { # 2: newuser

my $status = cmdsend('test.pl', $a, $cmailclient, $select,

'newuser', $USER);

return $status unless $status =~ /^\+OK.\*with password (.\*)$/;

$PASSWORD = $1;

1;

}, 1);

...

**The Functioning Prototype**

For the first prototype, I used a simple object persistence module, Persistence::Object::Sim

ple (which my friend Vipul had written for a project we’d worked on earlier) to whip up a

basic user database. Using persistent objects helped keep the code clean and intuitive, and

also provided a straightforward upgrade path to production database engines (simply cre

ate or derive a compatible Persistence::Object::\* class for the database engine).SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**173**

In late 2002, Matt Sergeant created another simple prototype-to-production path for Perl

hackers, DBD::SQLite module, a “self-contained RDBMS in a DBI driver,” which can be

used for rapid prototyping of database code without the need for a full database engine

during development. Personally, though, I prefer the elegance and simplicity of persistent

objects to having my code littered with SQL queries and DBI calls.

Mail received into the Cryptonite system was saved to regular *mbox* files, which worked

fine for the prototype. Of course, a production implementation would have to use a more

sophisticated mail store. I decided to use PGP itself as the encryption backend, to avoid

rewriting (and maintaining) all the encryption functionality already contained in PGP.

GnuPG was coming along, and I kept in mind that I might want to use it for cryptography

support in the future. So, I wrote Crypt::PGP5 to encapsulate the PGP5 functionality in a

Perl module. This module is available from CPAN (though I haven’t updated it in ages).

For the cryptographic core of Crypt::PGP5, I could have used the proprietary PGPSDK

library, but I would have had to create a Perl interface to it, which would likely have been

more work than just using the PGP binary. So, with a healthy dose of Perlish laziness and

keeping in mind that TMTOWTDI,\* I decided to use the Expect module to automate inter

actions with the PGP binary, using the same interface that’s available to human users of

the program. This worked well enough for the first prototype.

A basic web interface was developed, using the Text::Template module, to populate HTML

templates. The Cryptonite::Mail::HTML module contained all web-interface-related code,

including session handling.

The prototype system was ready after just three months of part-time coding. It imple

mented a full web interface, basic MIME support, OpenPGP encryption, decryption, sign

ing and signature verification, online new user registration, and a new and interesting

alternative to login passwords for authentication: PassFaces from ID Arts.

**Clean Up, Plug In, Rock On...**

After developing the initial prototype of Cryptonite in Costa Rica, I continued working on

it independently. After a much needed cleanup of the code (prototype development had

been hectic and had left not much time to refactor or test the code), I worked on a number

of Perl modules and components that would be needed next, to make the jump from a

simple prototype to a scalable product. These included Crypt::GPG (with an interface

almost identical to that of Crypt::PGP5, so that switching to GnuPG for the crypto opera

tions in Cryptonite involved little more than a single-line change to the code), and Persis

tence::Database::SQL and Persistence::Object::Postgres (which provide object persistence

in a Postgres database, with a similar interface to Persistence::Object::Simple, making the

backend database switch quite seamless as well).

\* “There’s More Than One Way To Do It,” a central tenet of the Perl way of life.**174**

CHAPTER ELEVEN

Persistence::Object::Postgres, like Persistence::Object::Simple, uses a blessed reference\* to a

hash container to store key-value pairs, which can be committed to the database with a

commit method call. It also uses Perl’s Tie mechanism to tie Postgres’ large objects (BLOBs)

to filehandles, enabling natural filehandle-based access to large binary objects in the data

base. One of the major benefits of Persistence::Database::SQL over Persistence::Object::

Simple, of course, is that it enables proper queries into a real database. For example, with

Persistence::Object::Simple, there’s no clean way to quickly search for a particular user’s

record, whereas with Persistence::Database::SQL, getting a specific user record from the

database is straightforward:

sub \_getuser { # Get a user object from the database.

my $self = shift; my $username = shift;

$self->db->table('users'); $self->db->template($usertmpl);

my ($user) = $self->db->select("WHERE USERNAME = '$username'");

return $user;

}

With Persistence::Object::Simple one would have to either iterate over all the persistent

objects in the data directory or resort to a hack such as directly grepping the plaintext per

sistence files in the data directory.

In most respects, the interface of Persistence::Object::Postgres is very similar to that of Per

sistence::Object::Simple. To modify an object with either module, the code is identical:

my $user = $self->\_getuser($username);

return $self->cluebat (EBADLOGIN) unless $user and $user->timestamp;

$user->set\_level($level);

$user->commit;

The switch from a plaintext database to a real DBMS was made after most of the prototype

code was basically working well, and marked the second stage of Cryptonite development:

getting the system ready for real-world deployment. For prototype development, Persis

tence::Object::Simple was great, as it didn’t require a database server to be available for

development, and objects were stored in plaintext files so they could be easily examined

for debugging.

The use of homomorphic interfaces for Crypt::GPG and Persistence::Object::Postgres

allowed these major changes (of the encryption and the database backends) to be made

with very minor edits to the code in Cryptonite::Mail::Service.

**Revamping the Mail Store**

Storing user mail in plain *mbox* files worked for the first prototype, but a production sys

tem needed to be able to access and update individual messages more efficiently than a

single flat file mailbox allowed. I also wanted to move toward the very important objective

of providing mail store replication for fault-tolerance.

\* In Perl, a reference becomes an object when associated to a class by bless, so “blessed reference” is

just a Perlish term for an object.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**175**

A usability consideration also imposed some requirements on the mail store. In Crypto

nite, unlike most email clients, information about MIME structures of messages would be

made visible to users in the message list. This would make it possible for a user to visually

identify which messages were encrypted and/or signed, directly in the message list. Avail

ability of information about message parts in the message list would also enable the user

to open a message subpart directly. The message parts are visible as icons in the rightmost

column of the message list view, as shown in Figure 11-4.

To enable such visual feedback, the mail store would need to efficiently provide accurate

information about the MIME structure of a list of messages. A further complication was

the fact that the OpenPGP/MIME spec allows for MIME parts to be nested within signed

and/or encrypted parts, so only an OpenPGP/MIME-aware mail store could return accu

rate information about MIME structures of encrypted or signed messages.

So I decided to implement, based on the Mail::Folder module, an SQL-based mail storage

backend with most of the abilities of an IMAP4rev1 server. The core of this system is the

Mail::Folder::SQL class, based on Mail::Folder and using Persistence::Object::Postgres. This

was back when IMAP had not yet gained much traction. I opted not to use an existing

IMAP server as a mail store because I anticipated needing some features that most IMAP

servers didn’t support well, such as mail store replication and the ability to retrieve

detailed information about the structure of a MIME message without having to retrieve

and parse the entire message.

Even though some IMAP servers might have suited my needs, I also didn’t want Crypto

nite to be dependent on and tied down to the capabilities of any specific IMAP server

implementation. All in all, this turned out to be a good decision, even though it did lead to

a lot of effort being expended on code that was later demoted to a less central role in the

system.

Mail store replication was hacked up using two Perl modules I wrote: Replication::Recall

and DBD::Recall, which used Eric Newton’s Recall replication framework (*http://www.*

*fault-tolerant.org/recall*) to replicate databases across multiple servers. The idea was to use

this as a prototype and to custom-build a new database replication system in the future.

With the encryption, database, and mail store backends revamped, and with a new,

cleaner theme, the first internal beta of Cryptonite went online in October 2001. It was

tested by many users of varying skill levels, some of whom even used it as their primary

*FIGURE 11-4* . *Message list with parts***176**

CHAPTER ELEVEN

mail client. Usability testing during the internal beta indicated that novice users were able

to successfully generate and import keys, and to send and read encrypted and signed mes

sages without much trouble.

**Persistence of Decryption**

An essential feature for an encrypted mail client is the ability to keep decrypted messages

available in decrypted form for the duration of the user’s session. A secure mail client that

lacks this facility can get very irritating and inefficient to use, as it would require typing in

long passphrases and waiting for decryption every time you want to read an encrypted

message or search within encrypted messages.

Persistence for previously decrypted messages in Cryptonite was accomplished by creating

a new Mail::Folder class, based on Mail::Folder::SQL. Mail::Folder::Shadow would dele

gate mailbox accesses to a *shadow folder* if the message had a counterpart in the shadow

folder; otherwise, it would access the underlying (or *shadowed*) folder.

By this means, decrypted messages could be kept in the shadow folder while a session was

alive, and little modification of the code was necessary to add persistent decrypts, other

than to plug in the Mail::Folder::Shadow module everywhere Mail::Folder::SQL was used.

Mail::Folder::Shadow implements its magic with a simple, tweakable delegation table:

my %method =

qw (get\_message 1 get\_mime\_message 1 get\_message\_file 1 get\_header 1

get\_mime\_message 1 mime\_type 1 get\_mime\_header 1 get\_fields 1

get\_header\_fields 1 refile 1 add\_label 2 delete\_label 2

label\_exists 2 list\_labels 2 message\_exists 1 delete\_message 5

sync 2 delete 2 open 2 set\_header\_fields 2 close 2 DESTROY 2

get\_mime\_skeleton 1 get\_body\_part 1);

Mail::Folder::Shadow delegates method calls as appropriate to the shadow folder, the orig

inal shadowed folder, or to both. Perl’s powerful AUTOLOAD feature, which provides a mech

anism to handle methods that are not explicitly defined in a class, is a simple way to

accomplish this delegation, while also providing a simple mechanism to tweak at runtime

how different methods are handled.

Methods that have to check the shadow store, such as get\_message and get\_header, are del

egated to the shadow if the message concerned exists in the shadow folder; otherwise,

they are delegated to the original shadowed folder. Other methods, such as add\_label and

delete (which deletes a folder), need to be dispatched to both the shadow and the shad

owed folder, as these messages must change the state of the original folder, as well as that

of the shadow folder.

Yet other methods, such as delete\_message, can accept a message list through an array ref

erence. Some of the messages in the message list may be shadowed, and others may not.

Mail::Folder::Shadow’s AUTOLOAD handles such methods by building two lists from the mes

sage list passed to it, one of shadowed messages and one of nonshadowed messages. It

then calls the method on both the shadowed and shadow folder for messages that are

shadowed, and only on the shadowed folder for messages that aren’t.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**177**

The practical upshot of all of this is that *cmaild* can continue to use folders just as it did

before, and stash decrypted messages in the shadow folder for the duration of a session.

There are a few extra methods in Mail::Folder::Shadow to enable this, including

update\_shadow, which is used to save the decrypted message in the shadow folder;

delete\_shadow, used to delete individual shadowed messages at user request; and unshadow,

used to delete all messages in shadow folders before session termination.

Mail::Folder::Shadow makes it possible to offer persistence of decrypted messages for a

session and to implement search within encrypted messages—both essential features from

a user’s perspective, but rarely implemented in current-generation OpenPGP-compliant

email systems.

**Hacking in the Himalayas**

Through 2000 and 2001 I was able to work on Cryptonite only intermittently, both

because of other commitments and because the project needed peace and quiet, which

was in limited supply when I was traveling around and living in chaotic, cacophonous,

polluted Indian cities.

In the summer of 2002, my wife and I took a vacation in the Himalayas, where I finally

managed to get the time to finish writing major chunks of the code, including adding

important key management abilities to Crypt::GPG, and creating an integrated interface

for key management, which is a critical part of the whole web-of-trust mechanism. The

core of this management interface, the Edit Key dialog, is shown in Figure 11-5. It enables

fingerprint verification, the viewing and creation of user identity certifications, and the

assigning of trust values to keys.

I also ported the system over to OpenBSD, which would be the ultimate deployment

platform.

We already had all the other major components for a secure email service in place, and as

it would still take some time to get Cryptonite ready for public use, we decided to go ahead

and launch a commercial secure email service right away. This would enable me to spend

more time on Cryptonite development, and to begin building a community of testers

immediately.

So in mid-2003, we launched the Neomailbox secure IMAP, POP3, and SMTP email

service. In the following years, this proved to be an excellent move that would help fund

development, freeing me from the need to take on other contract work and simulta

neously keeping me in close touch with the market for secure, private messaging.

In the fall of 2003, we set up a semi-permanent development base in a small Himalayan

hamlet, about 2000 meters above sea level, and this is primarily where development has

progressed since then. This kept our cash burn low, which is critical for a bootstrapping

startup, and gave me lots of time and peace to work on Neomailbox and Cryptonite.**178**

CHAPTER ELEVEN

Even though we had our share of trials working on mission-critical high-tech systems

from a remote Himalayan village that was, for the most part, still stuck in the 19th cen

tury, the words of Nikolai Roerich, the prolific Russian artist, writer, and philosopher who

lived in the same part of the Himalayas for many years, did to a large extent hold true for

us, too: “In truth, only here, only in the Himalayas, exist the unique, unprecedented, calm

conditions for achieving results.”

**Securing the Code**

Originally the code was designed as a prototype, and I didn’t worry about securing it too

much. But as time to make the system available as a public beta came around, it was time

to lock down the code with, at least:

•

Complete privilege separation

•

Paranoid input validation

•

Security audit of Crypt::GPG

•

Documentation of any potential security issues

Privilege separation was already built in from the ground up, by running *cmaild* as a privi

leged user and interacting with it via its API. This allowed *cmaild* to perform privileged

operations such as modifying system configuration files and performing cryptographic

operations in a controlled manner, without giving the web server process access to sensi

tive resources. Only a few areas required cleanup of the separation between the core and

the interface.

*FIGURE 11-5* . *The Edit Key dialog*SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**179**

One of these was the composition of MIME messages with binary attachments. When the

code was built using Persistence::Object::Simple, the *cmaild* protocol had been circum

vented for binary MIME message composition. Attachments uploaded by the user were

saved in a temporary directory, which both *cmaild* and the web server process had access

to. Thus, it was necessary to run *cmaild* and the Cryptonite web interface on the same

server.

With the move to Persistence::Object::Postgres, it became possible to easily pass binary

objects between the frontend and the backend via the database, without relying on direct

filesystem operations. This was important because the interface, the database, and the

Cryptonite engine were all intended to run on their own independent servers or in load

balancing clusters.

Input validation (to check the validity of user-supplied inputs, such as folder and message

identifiers) was straightforward to add. The Params::Validate module, very slightly modi

fied, was used to add input validation to every method of Cryptonite::Mail::Service. The

mvmsgs method, for example, validates its inputs with:

sub mvmsgs { # Move a list of messages to some other mailbox.

my ($self, $username, $key, $dest, $copy, @msgnums) =

(shift, lc shift, shift);

my ($user, $session, $err) = $self->validateuser($username, $key);

return $err if $err;

return $self->cluebat(@{$@}) unless eval {

($dest, $copy, @msgnums) = validate\_with ( params => \@\_,

extra => [$self], spec = [

{ type => SCALAR, callbacks =>

{ 'Legal Folder Name' => $self->legal\_foldername } },

{ type => SCALAR, callbacks =>

{ 'Boolean Flag' => $self->opt\_boolean }, optional => 1 },

({ type => SCALAR, callbacks =>

{ 'Legal Message Number' => $self->legal\_msgnum } })

x (@\_ - 2) ]

)

};

The acceptability of user-supplied input for each type of input field is specified via callback

subroutine references stored in a hash in the Cryptonite::Mail::Config module:

LGL\_FOLDERNAME => sub { $\_[0] =~ /$\_[1]->{"VFOLDER"}/i

or die (['EBADFOLDER', $\_[0]]) },

OPT\_BOOLEAN => sub { $\_[0] eq '' or $\_[0] eq 0 or $\_[0] eq 1

or die (['EBADBOOL', $\_[0]]) },

LGL\_MSGNUM => sub { $\_[0] =~ /$\_[1]->{"VMSGNUM"}/

or die (['EBADMSGNUM', $\_[0]]) },

Similar subroutines are invoked whenever an input parameter is validated. The regular

expressions for validity are stored separately in Cryptonite::Mail::Config.

Even though most of the validation subroutines are essentially the same, they are all dis

tinct, to enable each one to be tweaked as necessary without affecting the others or sacri

ficing clarity in this part of the code. The validation regular expressions and error strings

are stored in a table as well, to enable localization in the future.**180**

CHAPTER ELEVEN

Persistence::Object::Postgres also performs its own input sanity checks, to protect against

SQL injection attacks.

**Auditing Crypt::GPG**

Crypt::GPG had been written to be a working prototype and needed complete auditing to

eliminate any potential security issues before public testing of the system.

Crypt::GPG had been freely available on CPAN since 2001, and I’d received much valuable

feedback from its users. While many users said that they really liked the module’s clean

and simple interface, some had trouble getting it to run on certain platforms, where the

Expect module it used to interact with GnuPG didn’t work right. (Expect uses Unix

pseudoterminals [ptys] as its IPC mechanism, and that doesn’t work on Windows, for

example.)

The Expect module’s interface and syntax were also somewhat convoluted, which made

the code a little difficult to read, as can be seen from this section of the sign method:

my $expect = Expect->spawn ($self->gpgbin, @opts, '-o-', '--sign',

@extras, @secretkey, $tmpnam);

$expect->log\_stdout($self->debug);

$expect->expect (undef, '-re',

'-----BEGIN', 'passphrase:', 'signing failed');

if ($expect->exp\_match\_number == 2) {

$self->doze; print $expect ($self->passphrase . "\r");

$expect->expect (undef, '-re', '-----BEGIN', 'passphrase:');

if ($expect->exp\_match\_number == 2) { # Passphrase incorrect

$self->doze;

print $expect ($self->passphrase . "\r");

$expect->expect (undef, 'passphrase:'); $self->doze;

print $expect ($self->passphrase . "\r");

$expect->expect (undef);

unlink $tmpnam;

return;

}

}

elsif ($expect->exp\_match\_number == 3) {

unlink $tmpnam; $expect->close;

return;

}

$expect->expect (undef);

my $info = $expect->exp\_match . $expect->exp\_before;

Using the Expect-based module also caused *Heisenbugs*—failures that weren’t easily repro

ducible, and that I discovered were the result of sending input to *gpg* too fast. The calls to

doze in the previous code are a workaround for this: they introduce a few milliseconds of

delay before sending the next bit of input to *gpg*. This generally worked, but failures would

still occur on heavily loaded systems.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**181**

All these issues pointed to moving away from Expect and using another mechanism to

interact with the GnuPG binary. I considered the idea of writing a pure Perl implementa

tion of OpenPGP, but decided against it for basically the same reasons that I had decided to

use GnuPG in the first place: Cryptonite is primarily an email client, with integrated Open

PGP support. A full OpenPGP implementation would at least double the complexity of the

code I would have to maintain.\*

After a little experimenting, it looked like IPC::Run by Barrie Slaymaker might do the trick

for communication with GnuPG. With IPC::Run, the previous code became:

my ($in, $out, $err, $in\_q, $out\_q, $err\_q);

my $h = start ([$self->gpgbin, @opts, @secretkey, '-o-',

'--sign', @extras, $tmpnam],

\$in, \$out, \$err, timeout( 30 ));

my $i = 0;

while (1) {

pump $h until ($out =~ /NEED\_PASSPHRASE (.{16}) (.{16}).\*\n/g or

$out =~ /GOOD\_PASSPHRASE/g);

if ($2) {

$in .= $self->passphrase . "\n";

pump $h until $out =~ /(GOOD|BAD)\_PASSPHRASE/g;

last if $1 eq 'GOOD' or $i++ == 2;

}

}

finish $h;

my $d = $detach ? 'SIGNATURE' : 'MESSAGE';

$out =~ /(-----BEGIN PGP $d-----.\*-----END PGP $d-----)/s;

my $info = $1;

IPC::Run works reliably without the mini-delays needed with Expect, is much clearer to

read, and works perfectly on most platforms.

Some operations with *gpg* didn’t require any interaction, and earlier versions of the mod

ule had used Perl’s backtick operator for such cases. Because the backtick operator invokes

a shell, it’s a security risk. With IPC::Run, it was easy to replace the use of the backtick

operator with a tiny secure backtick function, thereby bypassing the shell. This made it

possible to eliminate all shell invocations in Crypt::GPG.

sub backtick {

my ($in, $out, $err, $in\_q, $out\_q, $err\_q);

my $h = start ([@\_], \$in, \$out, \$err, timeout( 10 ));

local $SIG{CHLD} = 'IGNORE';

local $SIG{PIPE} = 'IGNORE';

finish $h;

return ($out, $err);

}

\* A pure-Perl OpenPGP implementation, Crypt::OpenPGP, was written by Ben Trott in 2001–2002,

and is available from CPAN. I’m looking forward to using it in future versions of Cryptonite that

will support multiple cryptographic backends.**182**

CHAPTER ELEVEN

Some users had also pointed out that using temporary files to store plaintext could be

insecure. This problem could be easily overcome without touching the code, simply by

using temporary files on a RAM disk with encrypted swap (such as OpenBSD provides) or

an encrypted RAM disk, so plaintext would never be written to disk unencrypted.

Of course, it would be nice to modify the code to avoid writing plaintext to temporary files

at all, but as there already existed a practical workaround, eliminating temporary files

went on the to-do list rather than being implemented immediately.

The new IPC::Run-based Crypt::GPG was uploaded to CPAN at the end of 2005. It now

worked on a larger range of operating systems, and was more reliable and secure than its

Expect-based predecessor.

**The Invisible Hand Moves**

By mid-2004, Neomailbox was a year old and had attracted quite a few paying customers.

Cryptonite development was put on hold for a bit while I worked on developing various

aspects of the Neomailbox service as well as on a few other projects I just couldn’t wait to

get started on.

But being out in the market was great, as it brought market forces, from competition to

user feedback, to bear on the development process, and helped sharpen and clarify priori

ties. Customer requests and queries helped keep me intimately connected to what the

users and the market wanted. Meeting the market’s demands is how application code

becomes beautiful in a commercial sense, after all, so interaction with the market became

an integral and critical component of the development process.

Cryptonite was designed to be easy to maintain and modify, precisely because I knew that

at some point it would have to start to evolve in new ways, both in response to and in

anticipation of what the customer wanted. Being in the market enabled me to see that

emerging demand: it was clear that IMAP was the future of remote mailbox access.

IMAP has a lot of attractive features that make it a very powerful and practical mail access

protocol. One of the most important of these is the ability to access the same mailbox

using multiple clients, which becomes increasingly important with the proliferation of

computing devices. The typical user now has a desktop, a laptop, a PDA, and a cellphone,

all capable of accessing her mailbox.

This posed a slight problem, as I’d already implemented a full mail store for Cryptonite, and

it was not IMAP-based. There were two ways forward: either implement a full IMAP server

based on the Cryptonite mail store (a big job), or modify Cryptonite to enable it to use an

IMAP mail store as a backend. In fact, the second would have to be done either way.

Again, opting to reduce complexity of the system, and focusing on its primary purpose, I

decided not to develop the Cryptonite mail store into a full-blown IMAP server. Instead, I

modified it into a caching mechanism, which caches MIME skeletons (just the structure

information, without the content) of multipart MIME messages listed by the user, and alsoSECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**183**

entire messages read by the user, so that the next time a user opens a message she’s read

before, Cryptonite doesn’t need to go back to the IMAP server to fetch it again.

This gave me the best of both worlds. Cryptonite could reflect the contents of an IMAP

mailbox, while simultaneously posessing full information of each message’s exact MIME

structure, as well as being able to keep decrypted messages available in the shadow folders

the Cryptonite mail store supported.

The modifications to the code were straightforward. Whenever the user clicks to read a

message that isn’t in the cache, Cryptonite caches it in the corresponding Mail::Folder::

Shadow folder:

my $folder = $session->folder; # The folder name

my $mbox = \_opencache($username, $folder); # The M::F::Shadow cache

unless ($msgnum and grep { $\_ == $msgnum } $mbox->message\_list) {

# Message is not in cache. Fetch from IMAP server and cache it.

my $imap = $self->\_open\_imapconn($username, $user->password)

or sleep(int(rand(3))+2), return $self->cluebat (EBADLOGIN);

$imap->select($folder) or return $self->cluebat (ENOFOLDER, $folder);

$imap->Uid(1);

my ($tmpfh, $tmpnam) =

tempfile( $self->tmpfiles, DIR => "$HOME/tmp",

SUFFIX => $self->tmpsuffix, UNLINK => 1);

$imap->message\_to\_file($tmpfh, $msgnum);

$imap->logout;

my $parser = new MIME::Parser; $parser->output\_dir("$HOME/tmp/");

$parser->filer->ignore\_filename(1); # Do NOT use suggested filename

seek ($tmpfh,0,0);

my $mail = $parser->parse($tmpfh);

return $self->cluebat (ENOSUCHMSG, 0 + $msgnum) unless $mail;

$mbox->update\_message($msgnum,$mail);

}

In a similar manner, MIME skeletons are cached for all messages listed by the user

through the message list view. The rest of the code continues to work as before, by operat

ing on the cache for all read operations. Now we have IMAP compatibility, without com

promising the features afforded by my mail store or modifying the main code much.

Mail store replication would need to be worked in again because the switch from Mail::

Folder::SQL to an IMAP server for the mail store meant Replication::Recall couldn’t be

used for replication. But in any case, Replication::Recall wasn’t the most elegant or easy to

implement replication system, and the Recall library had been rewritten in Python, mak

ing my Perl interface to the earlier C++ implementation obsolete anyway.**184**

CHAPTER ELEVEN

In hindsight, I spent a lot of time on the replication functionality, which had to be

scrapped, and I probably would have been better off not bothering with replication at that

stage. On the other hand, it did teach me a lot that will come in handy when I get down to

implementing replication again.

Market forces and changing standards mean that application software is always evolving,

and much of the beauty of such code from the programmer’s point of view is certainly in

how easy it is to adapt the code to ever-changing requirements. Cryptonite’s object

oriented architecture makes it possible to implement major revisions with ease.

**Speed Does Matter**

With the Cryptonite mail store, performance had been quite snappy, and most mail store

operations had been independent of mailbox size. But with the switch to IMAP, I noticed

some major slowdowns with large mailboxes. A little profiling revealed that the perfor

mace bottleneck was the pure-Perl Mail::IMAPClient module, which I’d used to imple

ment the IMAP capability.

A quick benchmark script (written using the Benchmark module) helped me check

whether another CPAN module, Mail::Cclient, which interfaces to the UW C-Client

library, was more efficient than Mail::IMAPClient. The results showed clearly that I’d have

to redo the IMAP code using Mail::Cclient:

Rate IMAPClientSearch CclientSearch

IMAPClientSearch 39.8/s -- -73%

CclientSearch 145/s 264% --

Rate IMAPClientSort CclientSort

IMAPClientSort 21.3/s -- -99%

CclientSort 2000/s 9280% --

I probably should have thought of benchmarking the different modules before writing the

code with Mail::IMAPClient. I’d originally avoided the C-Client library because I wanted

to keep the build process as simple as possible, and Mail::IMAPClient’s build process is

definitely simpler than that of Mail::Cclient.

Fortunately, the switch from the former to the latter was generally quite straightforward.

For some operations, I noticed that IMAPClient could do the job better than C-Client

without much of a performance penalty, so Cryptonite::Mail::Service now uses both

modules, each to do whatever it’s better at.

A program like Cryptonite is never “finished,” of course, but the code is now mature,

robust, full of features, and efficient enough to serve its purpose: to provide thousands of

concurrent users a secure, intuitive, and responsive email experience while helping them

effectively protect the privacy of their communications.SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM

**185**

**Communications Privacy for Individual Rights**

I mentioned at the beginning of this chapter that making secure communications technol

ogy widely available is a very effective means of protecting individual rights. As this recog

nition is the basic motivation behind the Cryptonite project, I’d like to end with a few

observations on this point.

Cryptographic technology can, among other things:\*

•

Provide strong life-saving protection for activists, NGOs, and reporters working in

repressive countries.†

•

Preserve the communicability of censored news and controversial ideas.

•

Protect the anonymity of whistle-blowers, witnesses, and victims of domestic abuse.

•

For dessert, catalyze the geodesic society by enabling the free and unfettered exchange

of ideas, goods, and services globally.

The motley crew of hackers known as the Cypherpunks has been developing privacy

enhancing software for years, with the intent of enhancing personal freedom and individ

ual sovereignty in the digital age. Some cryptographic software is already a cornerstone of

how the world works today. This includes the Secure SHell (SSH) remote terminal soft

ware, which is essential to securing the Internet’s infrastructure, and the Secure Sockets

Layer (SSL) encryption suite, which secures online commerce.

But these systems target very specific needs: secure server access and secure online credit

card transactions, respectively. Both are concerned with securing human-machine interac

tions. Much more cryptographic technology specifically targeted at human-to-human inter

actions needs to be deployed in the coming years to combat the advancing menace of

ubiquitous surveillance (which “leads to a quick end to civilization”‡).

An easy-to-use, secure webmail system is an enabling technology—it makes possible, for

the first time in history, secure and private long-distance communications between indi

viduals all over the globe, who never need meet in person.

**Hacking the Civilization**

This computer of such infinite and subtle complexity that it includes organic life itself in its

operational matrix—the Earth, and the civilizations it hosts—can be reprogrammed in

powerful ways by simple pieces of code that hack human culture right back, and rewire

the operating system of society itself.

\* See *http://www.idiom.com/~arkuat/consent/Anarchy.html* and *http://www.chaum.com/articles/Security\_*

*Wthout\_Identification.htm.*

† See *http://philzimmermann.com/EN/letters/index.html.*

‡ Vernor Vinge, *A Deepness in the Sky.* Tor Books, 1999.**186**

CHAPTER ELEVEN

Code has changed the world many times. Consider the medical advances made possible by

genetic sequencing software, the impact of business software on large enterprises and

small businesses alike, the revolutions enabled by industrial automation software and

computer modeling, or the multiple revolutions of the Internet: email, the Web, blogs,

social networking services, VoIP.... Clearly, much of the history of our times is the story of

software innovation.

Of course, code, like any technology, can cut both ways, either increasing or decreasing

the “returns to violence”\* in society, increasing the efficacy of privacy-violating technol

ogy and giving tyrants more effective tools of censorship, on the one hand, or enhancing

and promoting individual rights on the other. Code of either sort hacks the very core of

human society itself, by altering fundamental social realities such as the tenability of free

speech.

Interestingly, even with a specific technology such as public key cryptography, the imple

mentation chosen can significantly alter cultural realities. For example, a PKI-based

implementation reimposes authoritarian properties such as centralized hierarchies and

identification requirements on a technology whose entire value arguably lies in its lack of

those properties. Despite this, PKI approaches deliver weaker key authentication than

does a web-of-trust implementation (which also doesn’t dilute other important features of

public-key cryptography, such as distributed deployment).

I think that as the weavers of code, it is to a large extent the ethical responsibility of pro

grammers to seek not only that our code be beautiful in its design and implementation,

but also that it be beautiful in its results, in a larger social context. This is why I find free

dom code so beautiful. It puts computing technology to the most sublime use: protecting

human rights and human life.

Laws and international human rights treaties can go only so far in protecting individual

rights. History has shown that these are far too easy to bypass or ignore. On the other

hand, the mathematics of cryptosystems can, when implemented carefully, provide practi

cally impenetrable shields for individual rights and open expression, and can finally set

people across the world free to communicate and trade in privacy and liberty.

Actualizing this global, mathematically protected open society is largely up to us, the gods

of the machine.

\* By “returns to violence,” I refer to the social and economic incentives for violating individual

rights. As the authors of *The Sovereign Individual* point out, “The key to understanding how societies

evolve is to understand factors that determine the costs and rewards of employing violence.”**187**

Chapter 12

CHAPTER TWELVE

**Growing Beautiful Code in BioPerl**

*Lincoln Stein*