Developing Trust: Online Privacy and Security

MATT CURTIN

Apress™

Developing Trust: Online Privacy and Security

ISBN (pbk): 1-893115-72-0

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Karen Watterson

Technical Reviewer: Gary Ellison

Project Managers: Erin Mulligan, Alexa Stuart

Developmental Editor and Copyeditor: Kim Goodfriend

Compositor: David Kramer

Artist: Charles Bowen

Indexer: Ann Rogers

Marketing Manager: Stephanie Rodriguez

Cover Designer: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER;
`orders@springer-ny.com`; `http://www.springer-ny.com`
Outside the United States, contact `orders@springer.de`;
`http://www.springer.de`; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710

Phone: 510-549-5930; Fax: 510-549-5939; `info@apress.com`; `http://www.apress.com`

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to download the code successfully.

CHAPTER 6

Design Principles

THE STUDY OF COMPUTER SECURITY is by no means new. Principles of secure design are not unknown. Yet as discussed earlier, as technology moves forward in so many other areas, it seems that we are making no progress at all, or are actually falling behind when it comes to systems that are safe and trustworthy.

Interestingly, many people think that "security" is a synonym for "invincibility." This concept of security is deeply rooted. Consider the Russian word for security: *bezopasnost.* It literally means "without danger." When we talk about building trustworthy systems, we're not talking about systems in which there is no danger, but we're talking about systems that you can reasonably believe will not come back to get you after you've decided to place your trust in them.

In essence, the difference between computer system *bezopasnost* and "trustworthiness" is really one of perception. Correct or not, to many people, security is about the elimination of risk to whatever degree possible. Trustworthiness is more broad, but generally it's about the elimination of *surprises,* particularly unpleasant surprises. Unquestionably, security is a significant part of trustworthiness, but trustworthiness also includes things like reliability and manageability with requirements that go beyond security's "don't let attackers alter your state."

The Need for Secure Design Principles

Anyone who has ever developed anything knows that designing and implementing systems that operate correctly is difficult. When we're confronted with a specification, we look for ways to solve the problem, but will often do so in such a way that we enable some other functionality. For example, we might decide that if we're going to be able to do business, we need the ability to send and to receive Internet email. Few businesses with such a requirement would actually get connectivity that would enable *only* email—they'll get something that *includes* it, along with access to the Web, and any other service that runs atop Internet Protocol. Although this connectivity addresses the requirement, it does so by enabling a huge level of connectivity, not only for your machines to the outside world, but for the outside world to your machines.

Although in many cases, our bosses and our customers will see such systems and think that we're doing a good job, perhaps by solving multiple problems at once, the fact is that if we're providing functionality that has not been specified and analyzed, we're actually introducing a *risk* into the system. It's some other connection, some other element that can fail, or some way to use the system beyond what we've considered when thinking about how to keep from giving away the store.

We therefore need to concern ourselves not only with *minimal functionality* (solving the specified problem) but also with *maximal functionality*. Concern about maximal functionality means that we're not going to stop after we've asked the question, "Does this system do *X*?" but that we're also going to ask the question, "What else does it do?" Some of these design principles might seem redundant, but in reality, they're different perspectives on the same theme: deal with the system one feature at a time. Doing several things at once and otherwise being too clever are great ways to get ourselves into horrible trouble.

To illustrate, we'll return to an example raised in Chapter 2. We touched on an argument against absolute privacy made by Scott McNealy in a *Washington Post* article [125]. He suggested that sacrificing privacy can sometimes have significant appeal, for example, building cars that provide driver medical information and vehicle location data to authorities if an airbag is deployed.

Following the publication of McNealy's article, discussion ensued on the *Cryptography* mailing list. Security expert Win Treese raised the need to consider whether tradeoffs in privacy for the ability to get help are being made explicitly or implicitly. He further proposed three systems.

1. A system that tracks Scott at all times and calls for help if there's an accident. Scott pays for this service in the form of lost privacy. The service provider can also benefit from the system by making use of the information collected.

2. A system that is designed to help in the case of an accident, implemented in such a way that it tracks Scott all the time. Someone might later figure out that the system contains additional data that can be somehow used.

3. A system that is designed to help in the case of an accident, whose design explicitly collects only the information needed to perform its task.

In this example, we see three different approaches to exactly the same problem. In the first case, we're looking at a system explicitly designed to collect data in order to provide some benefit, perhaps to multiple parties. The second case is implemented such that such data are collected as a side effect. The final case is one explicitly designed not to collect more than it needs.

Looking at the systems all around us, we find many examples of the first and second cases. Systems do a lot of data collection and storage these days. Were we to perform an extensive study of systems in the wild, I believe we would find that most systems collect way more data than they need, and it is after such data are collected and stored that someone figures out how to use those data. The case studies that we consider in more detail in Chapter 8 tend to show that this is a very common way of viewing system design. Clearly, collecting additional data as a side effect is a quagmire of nasty problems.

The obvious question that we must raise at this point, then, is why we don't just build systems that collect everything, figure out what to do with everything collected, and move forward from there. As already discussed, privacy has long been viewed as a basic human right. Privacy by definition—informational self-determination—is a matter that must be decided on an individual basis, for oneself. We as an industry cannot seriously claim that we're reasonable and professional if we're building systems that are designed to deny users an internationally recognized basic human right.

We're thus left with only one viable option: building systems not to collect more data than they need to do their jobs. This means that we must change our perceptions about system functionality; we must concern ourselves not only with minimal functionality, but we must be sure we understand the system's maximal functionality. Instead of merely meeting our requirements, we must ensure that we meet them exactly.

Without further ado, we'll move on to the secure design principles themselves.

Saltzer and Schroeder Secure Design Principles

The 1975 Saltzer and Schroeder paper [168] outlined design principles of secure systems. The principles they presented have shown themselves to be timeless principles that apply just as much in today's world of many devices talking to each other over networks as they were in the days of many users per machine. We're going to consider these principles here, in detail. As we go through each, we're going to consider what the principle means, some examples, and why we care. Our examples will draw heavily from our experiences in everyday life and from network-enabled computing systems. Our discussion will focus less on security, in favor of trustworthiness.

An advantage to using these principles is not only that they give thorough consideration to secure system design, but that they are widely cited, giving us a common vocabulary with which we can discuss secure systems meaningfully. Our discussion here is for the purpose of understanding how to build systems that are privacy-aware. Our examples and discussion of these ten principles, therefore, will be focused on privacy and enforcing it.

Economy of Mechanism

The first of the Saltzer and Schroeder secure design principles is *economy of mechanism.* This principle actually goes back much further than 1975.

In the fourteenth century, William Ockham[1] proposed the principle that has come to be called *Ockham's Razor:* "Pluralitas non est ponenda sine neccesitate." Literally, the phrase means "entities should not be multiplied unnecessarily," but the principle is most often taken to mean "keep it simple." In physics, this Razor is used to establish priority: given two theories for the same thing, study the simplest one first.

For our purposes, it is worth noting that there is a practical limit to how simple things should get. Albert Einstein warned, "Everything should be made as simple as possible, but not simpler." Simplicity and elegance go together well, but it does not necessarily follow that the more simple something is the more elegant it is. Taken together, simplicity and elegance form a balance that is admired everywhere, and for good reason.

Simplicity has many beneficial side effects. Simple solutions will tend to be:

- Easier to understand in their entirety;
- Easier to review from beginning to end;

[1] In Ockham's Latin, his name would be written *Occam.*

- Easier to analyze in their interactions with other things in the deployment environment; and
- Implementable in fewer lines of code, with fewer lines of configuration, and with fewer dependencies.

The result is that it's easier for us to ensure correctness and exact conformance to specification.

Here is an example. Suppose that we have a need to transfer a file from one system to another. The only interface that we have is serial, the kind of interface we'd use for a modem. We could implement a system that would work like this:

- Source calls the target on its modem.
- Target answers the incoming call.
- Source gives the target some identification and authentication credentials.
- Source then transfers the file to the target using some trivial protocol.
- Source tells the target that the transfer is complete.
- Source and target both hang up their modems.

This is something that could be written with relatively little code and that code would be straightforward for others to review. Even if we had a very high security requirement that made it necessary for us to analyze not only our scripts that would make this process work, but also the underlying code that supports it, there's relatively little code for us to analyze.

Contrast this with the amount of code that would be active to accomplish essentially the same objective over a typical Internet connection. In addition to the serial interface and the file transfer code, you would have to examine code that would handle:

- ISO layer 2 services (e.g., Ethernet drivers)
- Device drivers for the Ethernet cards
- The means of tying layer 2 services to layer 3 services (e.g., Ethernet to IP, using something like ARP)
- The IP stack in the operating system
- The TCP support in the operating system
- IP routing

That's an awful lot of code that comes into play in order solve a relatively simple problem. This, of course, conveniently neglects to point out that certain things that could be transferred over a typical Internet connection simply wouldn't be feasible to do for other reasons, including cost, in our simple manner.

Interactive media like the Web depend upon a constant connection for usability. Although you *could* send HTTP messages over our simple system, instead of taking seconds, it could take anywhere from minutes to days to get a single web page.

The point is that there are multiple, often conflicting, requirements that need to be considered. In our simple example of a need to transfer a file from one system to another, our solution *is* feasible, and therefore worth considering. But this is the reason that we specify our requirements; we might quickly decide that a trivial or "old" way of doing something isn't appropriate because it's not as flexible or "can't do as much" as some newfangled general-purpose solution. Keep in mind, though, that unless we've considered all of that additional functionality, we're only introducing unassessed risk into the equation.

Note that the first principle we're considering isn't strictly simplicity, it's economy. Economy means that we need to make efficient use of our resources. Our trivial file transfer mechanism solves the problem that was originally stated. Adding new functionality could become difficult and costly. Economy of mechanism will therefore mean that we strike a balance between simplicity and functionality.

Fail-Safe Defaults

Imagine this design for a nuclear weapon launch mechanism: an electrical circuit is responsible for keeping the weapon from launching. A break in the circuit or a loss of power will launch the weapon. Would this be a good design?

It might be said that a nuclear weapon station would have plenty of backup power and that an interruption in service would mean that the site had been attacked successfully.

The problem with this, obviously, is that there could be numerous other reasons for an interruption in power. Even if we do assume that the station was attacked, it doesn't stand to reason that the pre-programmed destination of the weapon is that of the attacker. Imagine that a developing country takes issue with the United States. If this country were to attack another nuclear power such as Russia, particularly the sites that are most likely aiming their weapons at the U.S, it would be an effective means to get Russia to do their dirty work. Simply take out the power on the Russian stations and wait for them to assume that they're under attack, and off the Russian missiles go.

Consider a bank vault. If its door is open and the building loses power, should the vault use its remaining backup power to close the door in a locked position or would it be alright for there to be no additional power, leaving no means to lock the door?

The question of *default behavior* is an important one, because the default behavior is typically what we'll see a system do whenever it fails.

And systems do fail [146].

The question is, will they fail into a position that's safe or into a position that's unsafe? Although we can easily see how this is applicable in the case of nuclear weapons or bank vaults, do we consider the same kinds of principles when we're designing systems that hold information?

If we're running a search engine, unless we look at every submission to the search engine by hand and peruse the sites included in our indices regularly, we can't really be sure to what we're linking. If we have been given a directive from management that we need to make our search engine "safe for kids"—i.e., no links to anything highly controversial or illegal to give to minors—a simple way to do that would be to have one of the options of our search be whether to limit our results to "safe for kids" links.

In this context, we're looking at the second of our design principles, *fail-safe mechanisms*. So what do we do by default? Do we make our searches include everything by default or do we limit our results to "safe for kids" links unless the user specifies otherwise?

Perhaps we want to make the user's choice persistent, so that he doesn't need to choose a non-default option every time that he conducts a search. The obvious solution to this problem in a Web context is to use a persistent cookie that will tell our search engine what the user's preference is every time that a search is submitted. The question arises again: which behavior is default, and which behavior do we use if the cookie is not present?

The choice for a fail-safe system is to fall into the position of absolutely greatest possible safety if the correct safety level cannot be determined. In this case, it means that our searches are going to need to be "safe for kids" unless the user affirmatively specifies otherwise. A user whose choice is not to restrict search results has, therefore, made an explicit choice, not had the choice made for him. If the user's cookie that stores the preference is lost—perhaps because he's using a new browser or new computer account—the user will have to assert the choice to be less safe again.

A great debate is raging at present with regard to the default behavior associated with such things as cookies and data collection practices on the public Web, particularly in connection with online advertising. The debate is opt-in versus opt-out. We discuss this in significantly more detail in Chapter 10, but this is something worth considering in this context. If we're going to be able to protect consumer privacy online, given the principle of fail-safe defaults, should we collect and profile the information by default, forcing consumers to assert that they do not want to have their data included in this kind of thing? Or does it make more sense not to collect the data by default and to require that users explicitly state that they want to participate in the data collection and analysis?

Complete Mediation

Complete mediation means that every time we request access to any object in the system—whether some piece of memory, a file on the disk, or a device—permission to access the device must be checked. By requiring that we include access control as part of the process of accessing any object, we're forcing the system to have the notion of access control included in the system's core, rather than as an add-on. This is extremely advantageous, since add-ons are often straightforward enough to circumvent.

One place to which we can look as an example of complete mediation is the Unix filesystem. Each device on the system, file on the disks, and even (in some systems) each process on the process table has an entry somewhere on the filesystem. Each entry includes several important properties, namely:

- Ownership by both user and group, with more granular specification available in many, particularly commercial, implementations
- Permissions, specifiable in such things as access to read, to write, and to execute at the very least

Thus, whenever a process in the system attempts to access a file or even a device, the ownership and permissions of that object's entry on the filesystem will perform the mediation and determine whether the requested action should be taken. If permission is granted by configuration, the action proceeds. If permission is not granted, the system will return an error (EX_NOPERM).

Having the ability to perform complete mediation on the filesystem is extremely useful and, particularly in the case of implementations that allow for highly granular permission specification, effective.

However, having all of the security features in the world won't do a system much good if the system administrators don't take advantage of those features. When we're writing applications, we do well to understand what features the operating system can provide us and to take advantage of all of those features that will help us.

When we're building applications that run locally, for local users, it's generally a lot easier to take advantage of the features of the operating system. When we're running services that are for use by network users, we seem to forget that we're dealing with a completely different environment, one over which no one has control. When we get a connection from a machine, for example, we can't really be sure that the machine is what it claims to be, since IP provides service for identification, but not authentication. When we get a packet that looks like it came from our application running on another machine, we really can't be sure that it wasn't really generated by a hostile application. When data are being directed to a dæmon on our server, we can't be sure that an attacker hasn't initiated the connection in an attempt to exploit some bug in our software or the operating system underneath.

We could enumerate examples all day. The point that we need to remember is that building applications that work on local, controlled environments is fundamentally different from building applications in the large, for potentially any user anywhere on the network. We must recognize the difference between what we can believe about our environment when running on a local machine and when offering our service to the world.

A concrete example we can use for driving this point home would be a web site that needs connectivity to a back-end database. Many system designers will look at ways to integrate web functionality with the database system itself. Others will look at ways to provide the web application access to the database through some standardized mechanism like SQL.[2] Each of these might seem reasonable upon first thought, but when looking at the secure design principle of complete mediation, we see that both approaches fail to consider security.

Perhaps the easiest way to see the folly of integration would be to consider the system in a mode of failure. A bug in the dæmon that services the request could influence the behavior of the database itself. If, on the other hand, we have a mechanism in the middle the mediates the connection, we can focus on ensuring that mediator's correctness and provide sanity checking, thus preventing hostile code from being able to reach the dæmon where a vulnerability could be exploited to undermine the system.

Standard access mechanisms, like SQL, have an additional risk. Rather than running only the risk of providing potentially hostile forces direct access to the dæmon, it's also possible that a data-borne attack could cause the exposure of data. Introducing the mediating program would allow us another point for enforcing policy. Here, we can ensure that the only kinds of SQL statements allowed to make it to the database for evaluation are valid by our policy. The advantage we have in a case like this is that if an attacker manages to break into the web server that talks to the database, requiring such access to go through a mediator would prevent the attacker from formulating queries that would reveal information that the web server would not need. Thus, even though an attacker has broken through a layer or two of protection (such as a firewall and the host security), there is still no advantage. It isn't until the mediator itself is broken that the attacker could send the database a `SELECT * FROM *`.

[2] By the way, that's "S-Q-L," not to be confused with SEQUEL, which is another, different, database access language.

Open Design

When we discuss *open design,* what we're talking about is a design that is not secret. A system with open design is one where the design is published and widely available for review and comment, just as open source code is published and widely available for review and comment. Let's take a look at how well this principle is understood in very general terms.

Some computer software vendors, Microsoft most notably among them, loudly and proudly proclaim that their offerings are "more secure" because their implementations are trade secrets, protected by the law of intellectual property and lawyers willing to chase after anyone violating the owner's intellectual property rights. Some computer security system vendors have even taken up this side of the argument. Many people buy this line (along with the proverbial hook and sinker) and look with disdain at offerings like Linux and BSD Unix. The question remains: does secrecy of implementation result in higher or lower security? Microsoft isn't the only organization in the world that seems to think that secrecy of source code means greater security.

Law in the U.S.—namely the Digital Millennium Copyright Act (DMCA)—codifies this mindset, making it illegal even to attempt to circumvent a security feature. People foolishly imagine themselves protected by this sort of thing. Whether you're talking about the secrecy of source code or outlawing reverse engineering, you're after the same thing: security through obscurity.

In most security circles, you'll hear, quite rightly, that security through obscurity is no security at all.

Of all of the areas of computer security where the goodness of open design is proclaimed, cryptography is especially noteworthy. It's easy to develop a cipher that one cannot break oneself. It's hard to develop a cipher that others cannot break. Therefore, instead of pursuing a comparatively simple business model of inventing a cipher, building it into a product, and taking it to market, responsible cryptosystem developers follow a much longer, harder path to market.

A cipher is first developed to address some sort of specification: that it should be good for *these* applications, that it should have *this* level of flexibility, that it should be *this* fast, that it should be easy to implement in software, and so on. Developers of the cipher will then apply known cryptanalytic techniques to look at their own work from the perspective of an attacker. Ciphers are going to have certain properties: that they process their data in blocks of *so many* bits, that they use keys of *this* length, that they run through *this many* rounds before spitting out the results.

After a cryptologist has implemented a cryptosystem and has analyzed its performance against well-known attacks, he might decide that it seems to show some promise. At this point, he'll publish the algorithm in its entirety so his peers in the scientific community can study the algorithm. Algorithms that seem to be the best candidates for actual use or that are novel in some way are most likely to get the attention of other cryptologists. Others might find new avenues of attack. Some might try new variations of already well-known attacks. Some might find successful results on weaker versions of the algorithm, perhaps one with a reduced number of rounds.

An algorithm that has gotten this kind of attention and has generally fared well in the face of serious analysis will begin to get a reputation as a worthy challenge. The longer that an algorithm stands in the face of analysis, the greater its reputation will be and the greater the level of trust that will be placed in it.

The reason for all of this work is straightforward: no matter how big your organization, no matter how smart your staff, there are more smart people working outside of your group than there are in it. In general, the more analysis that something receives, the greater the likelihood that its problems will be discovered and fixed. This is true in cryptography, and it is true in all areas of security.

Anyone still inclined to believe that hiding the details of implementation and its source code results in greater security would do well to consider the example of Microsoft. Despite being the largest software company in the world, with pockets deeper than virtually any other organization known and having so much riding on its software, the company that made Internet email-based worms possible doesn't seem able to stem the tide of security advisories about its products.

Reasons for this situation are many, but we're picking on Microsoft here merely to demonstrate a point: keeping source code a secret hasn't kept Microsoft out of CERT advisories.

Separation of Privilege

In Real Life (whatever that is), when we give someone a piece of paper, we're granting him essentially complete control over the paper. The ability to read it, to modify it, to copy it, even to destroy it. When dealing with computers, we have a nice feature that we don't have when using paper: the ability to separate privilege. Rather than giving someone "ownership" or "stewardship" of an object, we can grant the necessary access on a very granular basis. The well-known Unix permissions of "read," "write," and "execute" provide a useful means of sharing information to certain sets of users. Newer Unix implementations that include the concept of filesystem Access Control Lists (ACLs) give even greater separation of privilege.

What this means in practice is that if we are developing a system that will grant access to certain data, we need to specify what kinds of access the system will support. The more granular we are, the better. With no separation of privilege, we would need to grant each user with need to use an object for any reason complete stewardship of the object. As far as our security is concerned, we're back to using paper.

With a high separation of privilege, we'd have the ability to specify which users may and may not

- Read: learn a datum and its meaning;
- Copy: duplicate a datum;
- Modify: change a datum;
- Add: create a datum;
- Delete: mark a datum to be removed; or
- Expunge: destroy a datum that has been marked for removal.

There are probably many more that would make sense. The point is that when a user needs the ability to add records to the system, it isn't necessary also to grant him the ability to read the records in bulk after the fact or to remove them from the system. Someone (or something) responsible for making backups might have the need to copy everything in the system, but not necessarily to read anything or to to delete anything.

This leads us into our next issue . . .

Least Privilege

Least privilege is an extremely important principle. In practice, this means don't give any more privilege to someone than he needs to accomplish the specific task at hand.

As an example, let's consider the use of credit cards. Our requirement is to provide a simple, convenient mechanism for one party to transfer funds to another party without needing to handle cash. In general, this kind of thing can be done by check, though this can be something of a hassle, and the relative ease with which one can produce bogus checks is fairly high. It'd be nice to be able to do this in a way that one could actually verify the funds' availability and transfer the money. Credit cards are nice because it's possible for the merchant to put the transaction through immediately. If the funds are unavailable or there is reason to believe that an unauthorized person is attempting to make the payment, the transaction can be rejected on the spot.

If we then ask the question, "Does the system satisfy its requirements?" we're quite likely to answer, "Yes." If, however, we step back and ask the question, "What else does it do?" we're going to find some very interesting answers.

Use of a relatively constant and fairly easy-to-learn token—the credit card number itself—introduces some interesting possibilities, namely reuse of that token. For “security reasons,” completion of the transaction will generally be dependent upon another token, one that changes every two to three years—the card’s expiration date. Additionally, cards where a transaction is attempted with the wrong expiration date attached are shut down after a few unsuccessful tries—we’ll say three tries for the sake of this discussion.

One might think that we’ve been able to address our requirement without introducing too much risk into the system. Since there are twelve months in the year, and cards are good for three years from their date of issue, we’re looking at a maximum of 36 possible expiration dates. With only three attempts to guess the expiration date before the card will be shut down, we might feel pretty safe about the way that our cards are being used. Even if someone is able to get ahold of the number, the attacker has a one in 13 (that’s three in 36) chance of guessing the valid expiration date, which would be required to use the card.

The problem is that the attacker doesn’t have to try 13 different attacks on a single card to reach his goal since chances are that one attack against each of 13 different cards will get the same result—without triggering an alarm that something fishy is happening. People who would illegally use a credit card don’t generally care *whose* card it is—as long as it isn’t one that can be tied back to them.

Return to what we’re talking about: not giving someone more information than he needs to do his job. If what we’re trying to do is provide a means to pay for something, if we’re following the principle of least privilege, we’re going to give only information that will work for the present transaction. Trying to use the same token again would not work. One possible means of doing this would be to transfer money into a specific account for a specific purpose and to give the merchant the token to that account, which holds only enough money for that transaction, and which will be closed as soon as the transaction has been completed.

In many situations with computers, however, this principle isn’t as cumbersome to implement.

Let’s consider a second example, a computing example. For the sake of “ease,” a system administrator might decide to login to his own workstation with a privileged account, like `root` on Unix machines or `Administrator` on Windows machines. A sysadmin would have a much easier time of going about his work this way. Configuration changes, software installation, and other routine administrative tasks could be done directly, without the need to go through hoops to get additional privilege.

Windows NT is best written “Windowsn’t.”

In practice, though, few people do this, because everyone—generally, even the people who still insist on doing it—knows that working that way puts the system at risk. In multiuser systems, users generally are protected from each other, and users who get themselves into trouble tend only to be able to hose their own data. To destroy someone else’s data or to make the system unstable will often require additional levels of privilege. Thus, if a user is running a stupid piece of software, it will be limited in what damage it can unleash on the system. If, however, the program is being run with the privileges of a superuser, there will be essentially no limit to the amount of damage that something poorly written (or an ill-considered command) can do to other users and even to the system itself.

A third example might be in the issue of system configuration. Suppose that you have a web site that you're running. You need for your HTTP server to be able to read the content off of the filesystem in order to serve it to the clients. In many cases, people will make the ownership of the files on the filesystem match the user of the HTTP server process. This, however, violates the principle of least privilege. The HTTP server does not need the ability to write, to delete, or to remove any of those files. It needs merely to have the ability to read those files. A web server with a bug that can be exploited by a hostile client can only have its HTML files overwritten if the HTTP server that's being tricked into executing a command will have the authority to write over top of that file.

Following this principle will force us to ask the fundamental question: what, exactly, does this piece of the system need to be able to do? Knowing the answer to that question will then allow us to provide exactly that level of functionality—and not a single bit more. This one obstacle is extremely effective and terribly under-used.

Least Common Mechanism

Another principle that will help us to avoid building systems that can be used against us is *least common mechanism.* This means that we shouldn't create components that run exactly the same for everyone and provide everyone with exactly the same thing. This is the security-conscious way of saying what other software experts will tell you to do for other reasons: build modular code.

An operating systems example of this principle in practice would be in the development of a general-purpose utility. In a Unix system, one might implement a feature in the kernel, in a library against which "userland" programs may link, or in a function that is local to the application itself.

Addition of the functionality in the kernel might be desirable for performance reasons, perhaps for convenience, or just because the user wants to be able to say that he's written "kernel code." This is problematic, however, because code that runs in the kernel cannot be readily seen by a running system. A function that goes out to lunch when invoked will cause the kernel to fail. A function with a memory leak will cause the system to run out of RAM.

Kernel failure would be bad.

These kinds of problems are limited to some degree if they're implemented as library functions because a program that uses the function in user space can be killed by the system, thus reclaiming resources or at the very least, stopping the waste of resources. If this is true, one might argue that the obvious thing to do to limit the damage further would be to make the function local to one's own program, rather than providing the functionality to the rest of the system. This would be carrying the idea to the extreme, however, because although it's true that it would make the reach of bugs in the function even more localized, the amount of damage that would be caused by such a bug is roughly the same: a process would be goofed up. Its means for recovery would be the same: kill the hosed process.

Thus, we have no real "win" in the case of implementing the function locally in the program itself. We do have several losses, however, by comparison to the provision of the function as a generic feature that's available as a library. By increasing the potential for dependency on the code, we're increasing its importance, providing greater economic justification—whether we're talking about real money, funny money, or just plain ol' time doesn't matter here—for taking the time to build it correctly and to review it for defects.

One more example that we can use would be the case of connecting to a database. As we're building web-based applications, we see the need to build them such that they'll be used by many different persons, each of whom would have his own identity in the system. In such a system, if we rely on a single interface to a database, one where all connections to all parts of the database are handled through a single interface with the same privilege for every request, we have provided a common mechanism for functions like "add harmless record to database," "download every record in the database," and "scramble everything." Least common mechanism would dictate that we provide interfaces to the system such that if one were subverted, the attacker would not be able to do any more than that interface would allow. The compromised mechanism can't be used as a mechanism to do other things to the target.

Of course, we're advocating principles that complement each other well; such a design would allow us to enforce other principles like separation of privilege and complete mediation. We're building in layer after layer of functionality, rather than providing all of our functionality in a single component that can be compromised. Not all of our secure design principles, however, come down to the question of technology.

Psychological Acceptability

In general, the biggest problem that we have in building and deploying secure systems is our user base. Many just don't want to know anything more about any technology than they need to avoid getting fired from their jobs. Others would take more of an active interest in how all of this stuff works if there were some way to educate them reasonably. In either case, we end up with a user base that is clueless.

Even beyond dealing with the "clueless user" problems, we have to ensure that we build systems that people will not view as obstacles to getting things done. If our users don't accept the systems we build, no amount of "security" that we build in will prevent compromise, because they'll just work around whatever safeguards we put in place. Security can be viewed as a spectrum, just as we saw with nymity in Figure 2-1. On one end of the spectrum, we have complete security and on the other, we have complete access.

It has been said so often that it's almost a cliché now—but not often enough outside of security circles, so I can get away with it here—that a completely secure system is one where all of the data on its disks are encrypted before it's unplugged from the network, turned off, unplugged from its power supply, locked in a safe, and thrown to the bottom of the ocean. Even then, given enough time and money (that's "dedication" when we're talking about the resources of an attacker), that system might not *really* be secure. The problem with this kind of system is that it's not very usable. What's the point of having a computer if it isn't usable to *anyone?* We need to make the system do something useful for its intended user base.

On the other extreme of the spectrum is complete access. The computer will happily do anything that anyone tells it to do. Give me a listing of all of the users on the system. "Here you go." Dump the contents of the company's customer database and give it to me in a nice pretty format. "No problem." Tell me the salary and employment history for each of the employees. "Here you are." Reformat your drives and reboot. "Sure thing." The problem, of course, with complete access is that it does nothing to prevent the system from being abused by people who should not have the authority to get at the information it contains or even to protect itself against the most trivial of attacks. Neither does it prevent any accidental damage, such as that which might come from a poorly implemented program.

This might sound pretty silly, but in fact, complete access has been granted in a wide variety of production systems. Take, for example, single-user computer operating systems; no concept of access control. If you're on the machine, you must have authorization to do absolutely anything you want. This isn't something that was limited to the early days of "home computing." Even "business" microcomputers had this problem—look at DOS and Windows as examples. All you needed was the ability to type and you could make the computer do anything that it was capable of doing. Of course, with the advent of voice-recognition technology, it wasn't even necessary to have access to the keyboard. Simply shouting loud enough at the right time would be enough to make the computer do something stupid.

This isn't merely a theoretical problem. A great example appeared in a 1999 issue of *Computing* [6]. A representative of a company with a voice recognition product prepared to demonstrate their product and asked the crowd gathered to see the demonstration to be quiet. Someone in the back of the room shouted, "Format C Colon Return!" Someone else shouted, "Yes, Return!" The software worked perfectly, reformatting the primary disk on the demonstration unit, requiring that the machine finish its format and have all of its software and data reinstalled. Another great example of complete access was the IBM PS/1, with the infrared keyboard. Aim your keyboard at your office mate's machine, hit Control, Alt, Delete, and watch his machine reboot.

Work Factor

The cost of circumventing a security system is sometimes called its *work factor.* This is most commonly seen in cryptosystems. The fact is that all ciphers are ultimately susceptible to the problem of someone guessing the right key. The way that we prevent that attack from being a reasonable one is by making the number of possible keys so big that it's not practical for someone to try every single combination.

A simpler example of the same principle is a bicycle combination lock. Imagine such a lock having one tumbler with ten positions. Would the lock be secure? The amount of time that it would take to try each position to see if it will open the lock is not significant. Thus, its work factor is very low—you only need to try ten possible options. At the rate of one per second, that's probably a ten second job.

Consider the same combination lock, having two tumblers with ten positions each. Instead of having $10^1 = 10$ possible combinations, there are now $10^2 = 100$ possible combinations. By adding another tumbler, we've increased the work factor by an order of magnitude.[3] Now instead of taking ten seconds, the job will take 100 seconds. Add another tumbler and we get $10^3 = 1000$ possible combinations, etc.

This principle is exactly what we deal with when we're working with cryptosystems and we're trying to define how secure a cipher is. By the time we're working with a cipher in a production system, it should have undergone significant peer review and commentary to be sure that the brute force attack is the most effective attack. This is also why it's unwise to deploy a cryptosystem that depends on an algorithm that hasn't been studied widely and hasn't been able to stand the test of time. *Open Design!*

Algorithms like the Data Encryption Standard (DES) [140] have proved to be quite strong. After decades of inspection, the best avenues of attack are brute force. There are now variations of DES that increase its key length from the original 56 bits. Once such variant is 3DES, which will increase the effective key

[3] Since there are ten positions on each tumbler, each subsequent order of magnitude is *ten times* the previous. Combinatorics is the friend of people who want to address such attacks.

I'm not spelling out 2^{168}. That's much too silly.

length to 112 bits[4] or 168 bits, depending on how it's implemented. However, it is notable that at 56 bits, the key length of DES is short enough that computers today can break DES keys by trying every single possible combination. The first public crack of a DES key by brute force was by Rocke Verser's DESCHALL team in 1997 [46]. Since that time, other, faster DES-key–cracking machines have been built [121].

It's important to remember that brute-force attacks aren't necessarily the best attacks against a system. In practice, it's extremely difficult to build a system where a brute-force attack is the best. Just as we can use bolt cutters to defeat a bicycle combination lock of any number of tumblers in under a second, we can often apply other methods to defeat cryptosystems with huge key lengths. Key length is important, but only if the cryptosystem can last through all of the other attacks that we'll apply to it. Most cryptosystems can't, even in theory. Essentially, none can in practice, thanks to such modern wonders as virtual memory (read: "swap space might have an image of some pages of memory with the clues we need to break the encrypted file"), temporary files, and other goodies that are enabled by the operating systems on which we're trying to run these things.

So, in practice, calculating work factor can be difficult, because we need to identify each of the avenues of attack and then figure out how much it would cost (in terms of time, money, and effort) to circumvent that attack. If we can make it necessary for an attacker to spend more resources than the target is worth, our job is finished. Work factor is what helps us to determine that.

Compromise Recording

In some cases, it might not be as important to prevent an attacker from obtaining something as it would be to make it readily apparent that an attacker has done his deed, or to record the act of compromising the system. The reason this is useful is because cost is something that must be considered not only by attackers, but also by defenders.

If we have some physical documents that we want to store safely, we might do so by keeping them in a building that has some basic security systems in place: locks on the outside doors, other locks on inside doors, and perhaps a lock on the file cabinet where we keep the documents.

[4] Since bits are binary units, our "tumblers" have two possible positions; a 112 bit cipher has $2^{112} = 5{,}192{,}296{,}858{,}534{,}827{,}628{,}530{,}496{,}329{,}220{,}096$ possible keys. For the curious, that's "five decillion, one hundred ninety-two nonillion, two hundred ninety-six octillion, eight hundred fifty-eight septillion, five hundred thirty-four sextillion, eight hundred twenty-seven quintillion, six hundred twenty-eight quadrillion, five hundred thirty trillion, four hundred ninety-six billion, three hundred twenty-nine million, two hundred twenty thousand, ninety-six."

In such a case, we might decide that we're going to defend against someone trying to get into the building, against someone in the building trying to gain access to the floor where our office is, and against someone on our floor gaining entry to our office. But if such a person can gain entry into our office, we might figure at that point, it's all over: there's no stopping the attacker now. This is a time when we might want to have one final mechanism that will not stop the attacker as much as it will leave an audit trail.

One choice available to us might include a closed-circuit video camera pointed at the door. The attacker can destroy the camera, but not until after he goes through the door and gets his picture taken and has the image saved somewhere else. An excellent example of this principle can be found as a result of the 1982 Tylenol tragedy. Drugs on store shelves were vulnerable to being taken, tampered with, and returned to the shelf. Making the containers tamper-proof was too expensive, but the same benefit could be derived by a simple foil seal. If the drugs had been reached, the seal would be broken, giving us evidence of compromise.

No system is immune to all avenues of attack—even theoretically "perfect" systems—so the idea of recording a compromise in some way is very helpful for us to reconstruct what's happened, perhaps to make the recovery easier, and at least so that we don't continue to believe that a compromised system is safe. Sometimes, we can even use this as a means of exercising some legal recourse.

A commonplace technological example is one of the newest and most popular tools in data security: the intrusion detection system. By recording what's happened, we can have some evidence to suggest how the attacker broke through—thus giving us a feedback channel to improve the system—and giving us the heads-up we need to react appropriately.

One thing to remember is that the mechanisms that we use for compromise recording are themselves imperfect. An attacker can write raw traffic to a network to make it look like something that isn't happening is happening. The systems to record the compromise can crash or otherwise fail. Sophisticated attackers might even be able to render the data recorded by the system useless, making it impossible to tell which—if any—data can be believed.

Putting the Design Principles to Use

Use of secure design principles can help us make tremendous steps forward in building systems that resist failure. We do well to keep these in mind any time that we're involved in design or implementation—irrespective of whether management realizes that security must be built in. Once again, our ten principles are the following:

Memorize this list. That's an order.

- Economy of mechanism
- Fail-safe defaults
- Complete mediation
- Open design
- Separation of privilege
- Least privilege
- Least common mechanism
- Psychological acceptability
- Work factor
- Compromise recording

Consideration of secure design principles is unfortunately not common in undergraduate curricula. We therefore have a large number of folks working on building systems without ever having given serious consideration to the construction of systems that would not be vulnerable to attack. Hopefully, our tour of the Saltzer and Schroeder principles has proved eye-opening. We have gone a long time building systems without giving serious attention to the issue of security. Exceptions to our ability to avoid security has largely been confined to specific industries like banking. Secure design principles have long been necessary where security is a requirement. What we're discussing now isn't new; but the requirement for security is now present in more systems, affecting more systems developers.

As the Internet has become ubiquitous, the nature of the applications that we have built has changed dramatically. Most of the software that we've been building over the years has been confined to individual machines and used by users in whom we have some level of trust. Our deployment environments are no longer the desktop; we're deploying into the Internet, where users of our systems aren't always people we can trust.

We're now going to consider the Internet as a deployment environment.