Computer Security in the Real World

Butler W. Lampson[1]
Microsoft

Abstract

After thirty years of work on computer security, why are almost all the systems in service today extremely vulnerable to attack? The main reason is that security is expensive to set up and a nuisance to run, so people judge from experience how little of it they can get away with. Since there's been little damage, people decide that they don't need much security. In addition, setting it up is so complicated that it's hardly ever done right. While we await a catastrophe, simpler setup is the most important step toward better security.

In a distributed system with no central management like the Internet, security requires a clear story about who is trusted for each step in establishing it, and why. The basic tool for telling this story is the "speaks for" relation between principals that describes how authority is delegated, that is, who trusts whom. The idea is simple, and it explains what's going on in any system I know. The many different ways of encoding this relation often make it hard to see the underlying order.

1 Introduction

People have been working on computer system security for at least 30 years. During this time there have been many intellectual successes. Notable among them are the subject/object access matrix model [12], access control lists [19], multilevel security using information flow [6, 14] and the star-property [3], public key cryptography [16], and cryptographic protocols [1]. In spite of these successes, it seems fair to say that in an absolute sense, the security of the hundreds of millions of deployed computer systems is terrible: a determined and competent attacker could destroy most of the information on almost any of these systems, or steal it from any system that is connected to a network. Even worse, the attacker could do this to millions of systems at once.

The Internet has made computer security much more difficult than it used to be. In the good old days, a computer system had a few dozen users at most, all members of the same organization. It ran programs written in-house or by a few vendors. Information was moved from one computer to another by carrying tapes or disks.

Today half a billion people all over the world are on the Internet, including you. This poses a large new set of problems.

- *Attack from anywhere*: Any one on the Internet can take a poke at your system.
- *Sharing with anyone*: On the other hand, you may want to communicate or share information with any other Internet user.
- *Automated infection*: Your system, if compromised, can spread the harm to many others in a few seconds.

[1] blampson@microsoft.com, research.microsoft.com/lampson

- *Hostile code*: Code from many different sources runs on your system, usually without your knowledge if it comes from a Web page. The code might be hostile, but you can't just isolate it, because you want it to work for you.
- *Hostile physical environment*: A mobile device like a laptop may be lost or stolen and subject to physical attack.
- *Hostile hosts*: If you own information (music or movies, for example), it gets downloaded to your customers' systems, which may try to steal it.

All these problems cause two kinds of bad results. One is vandalism, motivated by personal entertainment or status-seeking: people write worms and viruses that infect many machines, either by exploiting buffer overrun bugs that allow arbitrary code to run, or by tricking users into running hostile code from e-mail attachments or web pages. These can disrupt servers that businesses depend on, or if they infect many end-user machines they can generate enough network traffic to overload either individual web servers or large parts of the Internet itself. The other bad result is that it's much easier to mount an attack on a specific target (usually an organization), either to steal information or to corrupt data.

On the other hand, the actual harm done by these attacks is limited, though growing. Once or twice a year an email virus such as "I love you" infects a million or two machines, and newspapers print extravagant estimates of the damage it does. Unfortunately, there is no accurate data about the cost of failures in computer security: most of them are never made public for fear of embarrassment, but when a public incident does occur, the security experts and vendors of antivirus software that talk to the media have every incentive to greatly exaggerate its costs.

Money talks, though. Many vendors of security products have learned to their regret that people may complain about inadequate security, but they won't spend much money, sacrifice many features, or put up with much inconvenience in order to improve it. This strongly suggests that bad security is not really costing them much. Firewalls and anti-virus programs are the only really successful security products, and they are carefully designed to require no end-user setup and to interfere very little with daily life.

The experience of the last few years confirms this analysis. Virus attacks have increased, and people are now more likely to buy a firewall and antivirus software, and to install patches that fix security flaws. Vendors like Microsoft are making their systems more secure, at some cost in backward compatibility and user convenience. But the changes have not been dramatic.

Many people have suggested that the PC monoculture makes security problems worse and that more diversity would improve security, but this is too simple. It's true that vandals can get more impressive results when most systems have the same flaws. On the other hand, if an organization installs several different systems that all have access to the same critical data, as they probably will, then a targeted attack only needs to find a flaw in one of them in order to succeed.

Of course, computer security is not just about computer systems. Like any security, it is only as strong as its weakest link, and the links include the people and the physical security of the system. Very often the easiest way to break into a system is to bribe an insider. This short paper, however, is limited to computer systems. It does not consider physical or human security. It also does not consider how to prevent buffer overruns. You might think from the literature that buffer overruns are the main problem in computer security, and of course it's important to eliminate them, especially in privileged code, but I hope to convince you that they are only a small part of the problem.

1.1 What is security?

What do we want from secure computer systems? Here is a reasonable goal:

Computers are as secure as real world systems, and people believe it.

Most real world systems are not very secure by the absolute standard suggested above. It's easy to break into someone's house. In fact, in many places people don't even bother to lock their houses, although in Manhattan they may use two or three locks on the front door. It's fairly easy to steal something from a store. You need very little technology to forge a credit card, and it's quite safe to use a forged card at least a few times.

Why do people live with such poor security in real world systems? The reason is that real world security is not about perfect defenses against determined attackers. Instead, it's about

- value,
- locks, and
- punishment.

The bad guys balances the value of what they gain against the risk of punishment, which is the cost of punishment times the probability of getting punished. The main thing that makes real world systems sufficiently secure is that bad guys who do break in are caught and punished often enough to make a life of crime unattractive. The purpose of locks is not to provide absolute security, but to prevent casual intrusion by raising the threshold for a break-in.

Well, what's wrong with perfect defenses? The answer is simple: they cost too much. There is a good way to protect personal belongings against determined attackers: put them in a safe deposit box. After 100 years of experience, banks have learned how to use steel and concrete, time locks, alarms, and multiple keys to make these boxes quite secure. But they are both expensive and inconvenient. As a result, people use them only for things that are seldom needed and either expensive or hard to replace.

Practical security balances the cost of protection and the risk of loss, which is the cost of recovering from a loss times its probability. Usually the probability is fairly small (because the risk of punishment is high enough), and therefore the risk of loss is also small. When the risk is less than the cost of recovering, it's better to accept it as a cost of doing business (or a cost of daily living) than to pay for better security. People and credit card companies make these decisions every day.

With computers, on the other hand, security is only a matter of software, which is cheap to manufacture, never wears out, and can't be attacked with drills or explosives. This makes it easy to drift into thinking that computer security can be perfect, or nearly so. The fact that work on computer security has been dominated by the needs of national security has made this problem worse. In this context the stakes are much higher and there are no police or courts available to punish attackers, so it's more important not to make mistakes. Furthermore, computer security has been regarded as an offshoot of communication security, which is based on cryptography. Since cryptography can be nearly perfect, it's natural to think that computer security can be as well.

What's wrong with this reasoning? It ignores two critical facts:

- Secure systems are complicated, hence imperfect.
- Security gets in the way of other things you want.

Software is complicated, and it's essentially impossible to make it perfect. Even worse, security has to be set up, by establishing user accounts and passwords, access control lists on resources, and trust relationships between organizations. In a world of legacy hardware and software, networked computers, mobile code, and constantly changing relationships between organizations, setup is complicated. And it's easy to think up scenarios in which you want precise control over who can do what. Features put in to address such scenarios make setup even more complicated.

Security gets in the way of other things you want. For software developers, security interferes with features and with time to market. This leads to such things as a widely used protocol for secure TCP/IP connections that use the same key for every session as long as the user's password stays the same [22], or an endless stream of buffer-overrun errors in programs that are normally run with administrative privileges, each one making it possible for an attacker to take control of the system.

For users and administrators, security interferes with getting work done conveniently, or in some cases at all. This is more important, since there are lot more users than developers. Security setup also takes time, and it contributes nothing to useful output. Furthermore, if the setup is too permissive no one will notice unless there's an audit or an attack. This leads to such things as users whose password is their first name, or a large company in which more than half of the installed database servers have a blank administrator password [10], or public access to databases of credit card numbers [24, 25], or e-mail clients that run attachments containing arbitrary code with the user's privileges [4].

1.2 Real security?

The end result should not be surprising. We don't have "real" security that guarantees to stop bad things from happening, and the main reason is that people don't buy it. They don't buy it because the danger is small, and because security is a pain.

- Since the danger is small, people prefer to buy features. A secure system has fewer features because it has to be implemented correctly. This means that it takes more time to build, so naturally it lacks the latest features.
- Security is a pain because it stops you from doing things, and you have to do work to authenticate yourself and to set it up.

A secondary reason we don't have "real" security is that systems are complicated, and therefore both the code and the setup have bugs that an attacker can exploit. This is the reason that gets all the attention, but it is not the heart of the problem.

Will things get better? Certainly when security flaws cause serious damage, buyers change their priorities and systems become more secure, but unless there's a catastrophe, these changes are slow. Short of that, the best we can do is to drastically simplify the parts of systems that have to do with security:

- Users need to have at most three categories for authorization: me, my group or company, and the world.
- Administrators need to write policies that control security settings in a uniform way, since they can't deal effectively with lots of individual cases.
- Everyone needs a uniform way to do end-to-end authentication and authorization across the entire Internet.

Since people would rather have features than security, most of these things are unlikely to happen very quickly.

On the other hand, don’t forget that in the real world security depends more on police than on locks, so detecting attacks, recovering from them, and punishing the bad guys are more important than prevention.

Section 2.3 discusses the first two points in more detail, and section 3 explores the third. For a fuller account of real world security, see Bruce Schneier’s recent book [21].

1.3 Outline

The next section gives an overview of computer security, highlighting matters that are important in practice. Section 3 explains how to do Internet-wide end-to-end authentication and authorization.

2 Overview of computer security

Like any computer system, a secure system can be studied under three headings:

Common name	*Meaning*	*Security jargon*
Specification:	What is it supposed to do?	Policy
Implementation:	How does it do it?	Mechanism
Correctness:	Does it really work?	Assurance

In security it’s customary to give new names to familiar concepts; they appear in the last column.

Assurance, or correctness, is especially important for security because the system must withstand malicious attacks, not just ordinary use. Deployed systems with many happy users often have thousands of bugs. This happens because the system enters very few of its possible states during ordinary use. Attackers, of course, try to drive the system into states that they can exploit, and since there are so many bugs, this is usually quite easy.

This section briefly describes the standard ways of thinking about policy and mechanism. It then discusses assurance in more detail, since this is where security failures occur.

2.1 Policy: Specifying security

Organizations and people that use computers can describe their needs for information security under four major headings [17]:

- *Secrecy*: controlling who gets to read information.
- *Integrity*: controlling how information changes or resources are used.
- *Availability*: providing prompt access to information and resources.
- *Accountability*: knowing who has had access to information or resources.

They are usually trying to protect some resource against danger from an attacker. The resource is usually either information or money. The most important dangers are:

Vandalism or sabotage that	
—damages information	integrity
—disrupts service	availability
Theft	
—of money	integrity
—of information	secrecy
Loss of privacy	secrecy

Each user of computers must decide what security means to them. A description of the user’s needs for security is called a security policy.

Most policies include elements from all four categories, but the emphasis varies widely. Policies for computer systems are usually derived from policies for real world security. The military is most concerned with secrecy, ordinary businesses with integrity and accountability, telephone companies with availability. Obviously integrity is also important for national security: an intruder should not be able to change the sailing orders for a carrier, and certainly not to cause the firing of a missile or the arming of a nuclear weapon. And secrecy is important in commercial applications: financial and personnel information must not be disclosed to outsiders. Nonetheless, the difference in emphasis remains [5].

A security policy has both a positive and negative aspect. It might say, "Company confidential information should be accessible only to properly authorized employees". This means two things: properly authorized employees *should* have access to the information, and other people *should not* have access. When people talk about security, the emphasis is usually on the negative aspect: keeping out the bad guy. In practice, however, the positive aspect gets more attention, since too little access keeps people from getting their work done, which draws attention immediately, but too much access goes undetected until there's a security audit or an obvious attack,[2] which hardly ever happens. This distinction between talk and practice is pervasive in security.

This paper deals mostly with integrity, treating secrecy as a dual problem. It has little to say about availability, which is a matter of keeping systems from crashing and allocating resources both fairly and cheaply. Most attacks on availability work by overloading systems that do too much work in deciding whether to accept a request.

2.2 Mechanism: Implementing security

Of course, one man's policy is another man's mechanism. The informal access policy in the previous paragraph must be elaborated considerably before it can be enforced by a computer system. Both the set of confidential information and the set of properly authorized employees must be described precisely. We can view these descriptions as more detailed policy, or as implementation of the informal policy.

In fact, the implementation of security has two parts: the code and the setup or configuration. The code is the programs that security depends on. The setup is all the data that controls the operations of these programs: folder structure, access control lists, group memberships, user passwords or encryption keys, etc.

The job of a security implementation is to defend against vulnerabilities. These take three main forms:

1) Bad (buggy or hostile) *programs.*
2) Bad (careless or hostile) agents, either programs or *people*, giving bad instructions to good but gullible programs.
3) Bad agents tapping or spoofing *communications*.

Case (2) can be cascaded through several levels of gullible agents. Clearly agents that might get instructions from bad agents must be prudent, or even paranoid, rather than gullible.

Broadly speaking, there are five defensive strategies:

[2] The modifier "obvious" is important; an undetected attack is much more dangerous, since the attacker can repeat it. Even worse, the victims won't know that they should take steps to recover, such as changing compromised plans or calling the police.

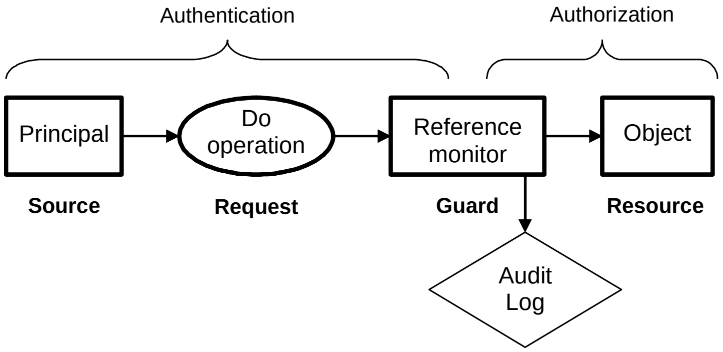

Figure 1: The access control model

1) *Coarse: Isolate*—keep everybody out. It provides the best security, but it keeps you from using information or services from others, and from providing them to others. This is impractical for all but a few applications.
2) *Medium: Exclude*—keep the bad guys out. It's all right for programs inside this defense to be gullible. Code signing and firewalls do this.
3) *Fine: Restrict*—Let the bad guys in, but keep them from doing damage. Sandboxing does this, whether the traditional kind provided by an operating system process, or the modern kind in a Java virtual machine. Sandboxing typically involves access control on resources to define the holes in the sandbox. Programs accessible from the sandbox must be paranoid; it's hard to get this right.
4) *Recover*—Undo the damage. Backup systems and restore points are examples. This doesn't help with secrecy, but it helps a lot with integrity and availability.
5) *Punish*—Catch the bad guys and prosecute them. Auditing and police do this.

The well-known *access control* model shown in figure 1 provides the framework for these strategies. In this model, a guard[3] controls the access of requests for service to valued resources, which are usually encapsulated in objects. The guard's job is to decide whether the source of the request, called a *principal*, is allowed to do the operation on the object. To decide, it uses two kinds of information: *authentication* information from the left, which identifies the principal who made the request, and *authorization* information from the right, which says who is allowed to do what to the object. As we shall see in section 3, there are many ways to make this division. The reason for separating the guard from the object is to keep it simple.

Of course security still depends on the object to implement its methods correctly. For instance, if a file's `read` method changes its data, or the `write` method fails to debit the quota, or either one touches data in other files, the system is insecure in spite of the guard.

Another model is sometimes used when secrecy in the face of bad programs is a primary concern: the *information flow control* model shown in figure 2 [6, 14]. This is roughly a dual of the access control model, in which the guard decides whether information can flow to a principal.

[3] a "reference monitor" in the jargon

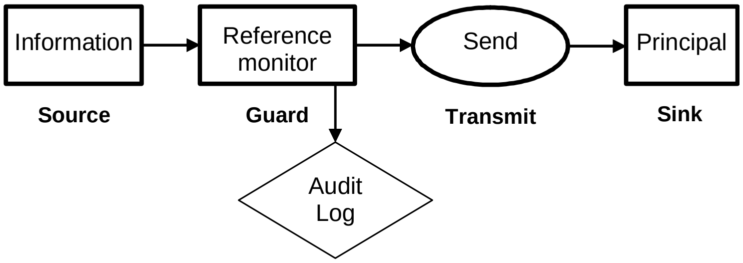

Figure 2: The information flow model

In either model, there are three basic mechanisms for implementing security. Together, they form the gold standard for security (since they all begin with Au):

- ***Au**thenticating* principals, answering the question “Who said that?” or “Who is getting that information?”. Usually principals are people, but they may also be groups, machines, or programs.
- ***Au**thorizing* access, answering the question “Who is trusted to do which operations on this object?”.
- ***Au**diting* the decisions of the guard, so that later it’s possible to figure out what happened and why.

2.3 Assurance: Making security work

The unavoidable price of reliability is simplicity. (Hoare)

What does it mean to make security work? The answer is based on the idea of a *trusted computing base* (TCB), the collection of hardware, software, and setup information on which the security of a system depends. Some examples may help to clarify this idea.

- If the security policy for the machines on a LAN is just that they can access the Web but no other Internet services, and no inward access is allowed, then the TCB is just the firewall (hardware, software, and setup) that allows outgoing port 80 TCP connections, but no other traffic.[4] If the policy also says that no software downloaded from the Internet should run, then the TCB also includes the browser code and settings that disable Java and other software downloads.[5]
- If the security policy for a Unix system is that users can read system directories, and read and write their home directories, then the TCB is roughly the hardware, the Unix kernel, and any program that can write a system directory (including any that runs as super-user). This is quite a lot of software. It also includes `/etc/passwd` and the permissions on system and home directories.

The idea of a TCB is closely related to the end-to-end principle [20]—just as reliability depends only on the ends, security depends only on the TCB. In both cases, performance and availability isn’t guaranteed.

[4] This assumes that there are no connections to the Internet except through the firewall.

[5] This assumes that the LAN machines don’t have any other software that might do downloads from the Internet. Enforcing this would greatly expand the TCB in any standard operating system known to me.

In general, it's not easy to figure out what is in the TCB for a given security policy. Even writing the specs for the components is hard, as the examples may suggest.

For security to work perfectly, the specs for all the TCB components must be strong enough to enforce the policy, and each component has to satisfy its spec. This level of assurance has seldom been attempted. Essentially always, people settle for something much weaker and accept that both the specs and the implementation will be flawed. Either way, it should be clear that a smaller TCB is better.

A good way to make defects in the TCB less harmful is to use *defense in depth*, redundant mechanisms for security. For example, a system might include:

- Network level security, using a firewall.
- Operating system security, using sandboxing to isolate programs. This can be done by a base OS like Windows or Unix, or by a higher-level OS like a Java VM.
- Application level security that checks authorization directly.

The idea is that it will be hard for an attacker to simultaneously exploit flaws in all the levels. Defense in depth offers no guarantees, but it does seem to help in practice.

Most discussions of assurance focus on the software (and occasionally the hardware), as I have done so far. But the other important component of the TCB is all the *setup* or configuration information, the knobs and switches that tell the software what to do. In most systems deployed today there is a lot of this information. It includes:

1) What software is installed with system privileges, and perhaps what software is installed that will run with the user's privileges. "Software" includes not just binaries, but anything executable, such as shell scripts or macros.
2) The database of users, passwords (or other authentication data), privileges, and group memberships. Often services like SQL servers have their own user database.
3) Network information such as lists of trusted machines.
4) The access controls on all the system resources: files, services (especially those that respond to requests from the network), devices, etc.
5) Doubtless many other things that I haven't thought of.

Although setup is much simpler than code, it is still complicated, it is usually done by less skilled people, and while code is written once, setup is different for every installation. So we should expect that it's usually wrong, and many studies confirm this expectation. The problem is made worse by the fact that setup must be based on the documentation for the software, which is usually voluminous, obscure, and incomplete at best.[6] See [2] for an eye-opening description of these effects in the context of financial cryptosystems, [18] for an account of them in the military, and [21] for many other examples.

The only solution to this problem is to make security setup much simpler, both for administrators and for users. It's not practical to do this by changing the base operating system, both because changes there are hard to carry out, and because some customers will insist on the fine-grained control it provides. Instead, take advantage of this fine-grained control by using it as a "machine language". Define a simple model for security with a small number of settings, and then compile these into the innumerable knobs and switches of the base system.

What form should this model take?

[6] Of course code is also based on documentation for the programming language and libraries invoked, but this is usually much better done.

Users need a very simple story, with about three levels of security: me, my group or company, and the world, with progressively less authority. Browsers classify the network in this way today. The corresponding private, shared, and public data should be in three parts of the file system: my documents, shared documents, and public documents. This combines the security of data with where it is stored, just as the physical world does with its public bulletin boards, private houses, locked file cabinets, and safe deposit boxes. It's familiar, there's less to set up, and it's obvious what the security of each item is.

Everything else should be handled by security policies that vendors or administrators provide. In particular, policies should classify all programs as trusted or untrusted based on how they are signed, unless the user overrides them explicitly. Untrusted programs can be rejected or sandboxed; if they are sandboxed, they need to run in a completely separate world, with separate global state such as user and temporary folders, history, web caches, etc. There should be no communication with the trusted world except when the user explicitly copies something by hand. This is a bit inconvenient, but anything else is bound to be unsafe.

Administrators still need a fairly simple story, but they need even more the ability to handle many users and systems in a uniform way, since they can't deal effectively with lots of individual cases. The way to do this is to let them define so-called security *policies*[7], rules for security settings that are applied automatically to groups of machines. These should say things like:

- Each user has read/write access to their home folder on a server, and no one else has this access.
- A user is normally a member of one workgroup, which has access to group home folders on all its members' machines and on the server.
- System folders must contain sets of files that form a vendor-approved release.
- All executable programs must be signed by a trusted authority.

These policies should usually be small variations on templates provided and tested by vendors, since it's too hard for most administrators to invent them from scratch. It should be easy to turn off backward compatibility with old applications and network nodes, since administrators can't deal with the security issues it causes.

Some customers will insist on special cases. This means that useful exception reporting is essential. It should be easy to report all the variations from standard practice in a system, especially variations in the software on a machine, and all changes from a previous set of exceptions. The reports should be concise, since long ones are sure to be ignored.

To make the policies manageable, administrators need to define *groups* of users and of resources, and then state the policies concisely in terms of these groups. Ideally, groups of resources follow the file system structure, but there need to be other ways to define them to take account of the baroque conventions in existing networks, OS's and applications.

To handle repeating patterns of groups, system architects can define *roles*, which are to groups as classes are to objects in Java. So each division in a company might have roles for 'employees', 'manager', 'finance, and 'marketing', and folders such as 'budget' and 'advertising plans'. The 'manager' and 'finance' roles have write access to 'budget' and so forth. The Appliance division will have a specific group for each of these: 'Appliance-members', 'Appliance-budget', etc., and thus 'Appliance-finance' will have write access to 'Appliance-budget'.

[7] This is a lower-level example of a security specification, one that a machine can understand, by contrast with the informal, high-level examples that we saw earlier.

Policies are usually implemented by compiling them into existing security settings. This means that existing resource managers don't have to change, and it also allows for both powerful high-level policies and efficient enforcement, just as compilers allow both powerful programming languages and efficient execution.

Developers need a type-safe virtual machine like Java or Microsoft's Common Language Runtime; this will eliminate a lot of bugs. Unfortunately, most of the bugs that hurt security are in system software that talks to the network, and it will be a while before system code is written that way.

They also need a development process that takes security seriously, valuing designs that make assurance easier, getting them reviewed by security professionals, and refusing to ship code with serious security flaws.

3 End-to-end access control

Any problem in computer science can be solved with another level of indirection. (Wheeler)

Secure distributed systems need a way to handle authentication and authorization uniformly throughout the Internet. In this section we first explain how security is done locally today, and then describe the principles that underlie a uniform end-to-end scheme. A number of papers [8, 11, 23] explain these ideas in more detail, and they are the basis for recent proposals for web services security [15].

3.1 Local access control

Most existing systems, such as Unix and Windows, do authentication and authorization locally. They have a local database for user authentication (usually by passwords) and group membership, and a local database of authorization information, usually in the form of an access control list (ACL) on each resource. They either rely on physical security or luck to secure the channel to the user, or use an encrypted channel protocol like PPTP. The system identifies a logged-in user by a "security identifier" or SID.

You might think that security on the web is more global or distributed, but in fact web servers work the same way. They usually use SSL to secure the user channel; this also authenticates the server's DNS name, but users hardly ever pay any attention. Authorization is primitive, since usually the only protected resources are the entire service and the private data that it keeps for each user. Each server farm has a separate local user database.

There is a slight extension of this strictly local scheme, in which each system belongs to a "domain" and the authentication database is stored centrally on a domain controller. To log in a user the local system, instead of doing the work itself, does an RPC to the controller (secured by a shared encryption key that is set up when the system joins the domain), passing in the user's password response. The controller does exactly what the login system did by itself in the earlier scheme, and returns the user's identity. It also returns a token that the login system can use later to reauthenticate the user to the controller. SIDs are no longer local to the individual system but span the domain.

Kerberos, Windows domains, and Passport all work this way. In Kerberos the reauthentication token is confusingly called a "ticket-granting ticket".

To authenticate the user to another system in the domain, the login system can ask the controller to forward the authentication to the target system (or it can forward the password response, an old and deprecated method). In Kerberos the domain controller does this by sending a message called a “ticket” to the target on the secure channel between them.[8]

Authentication to another domain works the same way, except that there is another level of indirection through the target domain’s controller. A shared key between the two domains secures this channel. The secure communication is thus login system to login controller to target controller to target. A further extension organizes the domains in a tree and uses this scheme repeatedly, once for each domain on the path through the common ancestor.

Unless the domains trust each other completely, each one should have its own space of SIDs and should only be trusted to assign its own SIDs to a user. Otherwise any domain can assign any SID to any user, so that the Microsoft subsidiary in Columbia can authenticate someone as Bill Gates. See section 3.5 for more on this point.

3.2 Distributed access control

A distributed system may involve systems (and people) that belong to different organizations and are managed differently. To do access control cleanly in such a system (as opposed to the strictly local systems of the last section) we need a way to treat uniformly all the items of information that contribute to the decision to grant or deny access. Consider the following example:

Alice at Intel is part of a team working on a joint Intel-Microsoft project called `Atom`. She logs in, using a smart card to authenticate herself, and connects using SSL to a project web page called `Spectra` at Microsoft. The web page grants her access because:

1) The request comes over an SSL connection secured with a session key K_{SSL}.
2) To authenticate the SSL connection, Alice’s smart card uses her key K_{Alice} to sign a response to a challenge from the Microsoft server.[9]
3) Intel certifies that K_{Alice} is the key for `Alice@Intel.com`.

Microsoft’s group database says that `Alice@Intel.com` is in the `Atom` group.

4) The ACL on the `Spectra` page says that `Atom` has read/write access.

For brevity, we drop the `.com` from now on.

To avoid the need for the smart card to re-authenticate Alice to Microsoft, the card can authenticate a temporary key K_{temp} on her login system, and that key can authenticate the login connection. (2) is then replaced by:

2a) Alice’s smart card uses her key K_{Alice} to sign a certificate for the temporary key K_{temp} owned by the login system.

2b) Alice’s login system authenticates the SSL connection by using K_{temp} to sign a response to a challenge from the Microsoft server.

From this example we can see that many different kinds of information contribute to the access control decision:

Authenticated session keys

[8] This means that the controller encrypts the ticket with the key that it shares with the target. It actually sends the ticket to the login system, which passes it on to the target, but the way the bits are transmitted doesn’t affect the security. The ticket also enables a secure channel between the target and the login system, by including a new key shared between them.

[9] Saying that the card signs with the public key K_{Alice} means that it encrypts with the corresponding private key. Through the magic of public-key cryptography, anyone who knows the public key can verify this signature.

User passwords or public keys
Delegations from one system to another
Group memberships
ACL entries.

We want to do a number of things with this information:

- Keep track of how secure channels are authenticated, whether by passwords, smart cards, or systems.
- Make it secure for Microsoft to accept Intel's authentication of Alice.
- Handle delegation of authority to a system, for example, Alice's login system.
- Handle authorization via ACLs like the one on the `Spectra` page.
- Record the reasons for an access control decision so that it can be audited later.

3.3 Chains of trust

What is the common element in all the steps of the example and all the different kinds of information? From the example we can see that there is a *chain of trust* running from the request at one end to the `Spectra` resource at the other. A link of this chain has the form

"Principal *P* speaks for principal *Q* about statements in set *T*."

For example, K_{SSL} speaks for K_{Alice} about everything, and `Atom@Microsoft` speaks for `Spectra` about read and write.

The idea of "speaks for" is that if *P* says something about *T*, then *Q* says it too; that is, *P* is trusted as much as *Q*, at least for statements in *T*. Put another way, *Q* takes responsibility for anything that *P* says about *T*. A third way: *P* is a more powerful principal than *Q* (at least with respect to *T*) since *P*'s statements are taken at least as seriously as *Q*'s, and perhaps more seriously.

The notion of principal is very general, encompassing any entity that we can imagine making statements. In the example, secure channels, people, groups, systems, program images, and resource objects are all principals.

The idea of "about subjects *T*" is that *T* is some way of describing a set of things that *P* (and therefore *Q*) might say. You can think of *T* as a pattern or predicate that characterizes this set of statements. In the example, *T* is "all statements" except for step (5), where it is "read and write requests". It's up to the guard of the object that gets the request to figure out whether the request is in *T*, so the interpretation of *T*'s encoding can be local to the object. for example, we could refine "read and write requests" to "read and write requests for files whose names match `~lampson/security/*.doc`". SPKI [8] develops this idea in some detail.

We can write this $P \overset{T}{\Rightarrow} Q$ for short, or $P \Rightarrow Q$ if *T* is "all statements". With this notation the chain for the example is:

$$K_{SSL} \Rightarrow K_{temp} \Rightarrow K_{Alice} \Rightarrow \texttt{Alice@Intel} \Rightarrow \texttt{Atom@Microsoft} \overset{\text{r/w}}{\Rightarrow} \texttt{Spectra}$$

Figure 3 shows how the chain of trust is related to the various principals. Note that the "speaks for" arrows are quite independent of the flow of bytes: trust flows clockwise around the loop, but no data traverses this path.

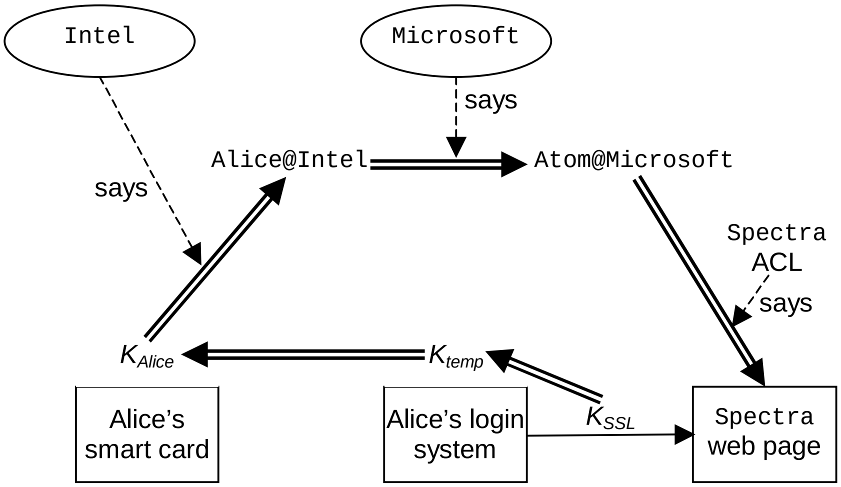

Figure 3: The chain of trust

The remainder of this section explains some of the details of establishing the chain of trust. Things can get a bit complicated; don't lose sight of the simple idea. For more details see [13, 23, 8, 11].

3.4 Evidence for the links

How do we establish a link in the chain, that is, a fact $P \Rightarrow Q$? Someone, either the object's guard or a later auditor, needs to see evidence for the link; we call this entity the "verifier". The evidence has the form "principal says delegation", where a delegation is a statement of the form $P \overset{T}{\Rightarrow} Q$.[10] The principal is taking responsibility for the delegation. So we need to answer three questions:

Why do we *trust* the principal for this delegation?
How do we know who *says* the delegation?
Why is the principal *willing* to say it?

Why trust? The answer to the first question is always the same: We trust Q for $P \Rightarrow Q$, that is, we believe it if Q says it. When Q says $P \overset{T}{\Rightarrow} Q$, Q is delegating its authority for T to P, because on the strength of this statement anything that P says about T will be taken as something that Q says. We believe the delegation on the grounds that Q, as a responsible adult or the computer equivalent, should be allowed to delegate its authority.

Who says? The second question is: How do we know that Q says $P \overset{T}{\Rightarrow} Q$? The answer depends on how Q is doing the saying.

1) If Q is a key, then "Q says X" means that Q cryptographically signs X, and this is something that a program can easily verify. This case applies for $K_{temp} \Rightarrow K_{Alice}$. If K_{Alice} signs it, the verifier believes that K_{Alice} says it, and therefore trusts it by the delegation rule above.
2) If Q is the verifier itself, then $P \overset{T}{\Rightarrow} Q$ is probably just an entry in a local database; this case applies for an ACL entry like `Atom` ⇒ `Spectra`. The verifier believes its own local data.

[10] In [13, 23] this kind of delegation is called "handoff", and the word "delegate" is used in a narrower sense.

These are the only ways that the verifier can *directly* know who said something: receive it on a secure channel or store it locally.

To verify that any other principal says something, the verifier needs some reasoning about "speaks for". For a key binding like $K_{Alice} \Rightarrow$ `Alice@Intel`, the verifier needs a secure channel to some principal that can speak for `Alice@Intel`. As we shall see later, `Intel` $\overset{\text{delegate}}{\Rightarrow}$ `Alice@Intel`. So it's enough for the verifier to see $K_{Alice} \Rightarrow$ `Alice@Intel` on a secure channel from `Intel`. Where does this channel come from?

The simplest way is for the verifier to simply know $K_{Intel} \Rightarrow$ `Intel`, that is, to have it wired in (case (2) above). Then encryption by K_{Intel} forms the secure channel. Recall that in our example the verifier is a Microsoft web server. If Microsoft and Intel establish a direct relationship, Microsoft will know Intel's public key K_{Intel}, that is, know $K_{Intel} \Rightarrow$ `Intel`.

Of course, we don't want to install $K_{Intel} \Rightarrow$ `Intel` explicitly on every Microsoft server. Instead, we install it in some Microsoft-wide directory. All the other servers have secure channels to the directory (for instance, they know the directory's public key K_{MSDir}) and trust it unconditionally to authenticate principals outside Microsoft. Only K_{MSDir} and the delegation

> "$K_{MSDir} \Rightarrow$ `*` except `*.Microsoft.com`"

need to be installed in each server.

The remaining case in the example is the group membership `Alice@Intel` $\Rightarrow$ `Atom@Microsoft`. Just as `Intel` $\overset{\text{delegate}}{\Rightarrow}$ `Alice@Intel`, so `Microsoft` $\overset{\text{delegate}}{\Rightarrow}$ `Atom@Microsoft`. Therefore it's Microsoft that should make this delegation.

Why willing? The third question is: Why should a principal make a delegation? The answer varies greatly. Some facts are installed manually, like $K_{Intel} \Rightarrow$ `Intel` at Microsoft when the companies establish a direct relationship, or the ACL entry `Atom` $\overset{\text{r/w}}{\Rightarrow}$ `Spectra`. Others follow from the properties of some algorithm. For instance, if I run a Diffie-Hellman key exchange protocol that yields a fresh shared key K_{DH}, then as long as I don't disclose K_{DH}, I should be willing to say

> "$K_{DH} \Rightarrow$ me, provided you are the other end of a Diffie-Hellman run that yielded K_{DH}, you don't disclose K_{DH} to anyone else, and you don't use K_{DH} to send any messages yourself."

In practice I do this simply by signing $K_{DH} \Rightarrow K_{me}$; the qualifiers are implicit in running the Diffie-Hellman protocol.[11]

For a very different example, consider a server S starting a process from an executable program image file `SQLServer.exe`. If S sets up a secure channel C from this process, it can safely assert $C \Rightarrow$ `SQLServer`. Of course, only someone who trusts S to run `SQLServer` (that is, believes $S \overset{\text{delegate}}{\Rightarrow}$ `SQLServer`) will believe S's statement. Normally administrators set up such delegations. We discuss this in more detail in section 3.6.

3.5 Names

In the last section we said without explanation that `Intel` $\overset{\text{delegate}}{\Rightarrow}$`Alice@Intel`. Why is this a good convention to adopt? Well, someone has to speak for `Alice@Intel`, since the only alternative is to install facts about it manually, which is tedious and error-prone. Who should it be? The

[11] Another way, used by SSL, is to send my password on the K_{DH} channel. This is all right if I know from the other scheme that the intended server is at the other end of the channel. Otherwise I might be giving away my password.

obvious answer is that the parent of a name should speak for it, at least for the purpose of delegating its authority. Formally, we have the axiom $P \overset{\text{delegate}}{\Rightarrow} P/N$[12] for any principal P and simple name N. Using this repeatedly, P can delegate from any path name that starts with P. This is the whole point of hierarchical naming: parents have authority over children.

The simplest form of this is $K \overset{\text{delegate}}{\Rightarrow} K/N$, where K is a key. This means that every key is the root of a name space. This is simple because you don't need to install anything to use it. If K is a public key, it says $P \Rightarrow K/N$ by signing a certificate with this contents. The certificate is public, and anyone can verify the signature and should then believe $P \Rightarrow K/N$.

Unfortunately, keys don't have any meaning to people. Usually we will want to know $K_{Intel} \Rightarrow$ `Intel`, or something like that, so that from K_{Intel} says "$K_{Alice} \Rightarrow$ `Alice@Intel`" we can believe it. How do we establish this? One way, as always, is to install $K_{Intel} \Rightarrow$ `Intel` manually; we saw in the previous section that Microsoft might do this if it establishes a direct relationship with Intel. The other is to use hierarchical naming at the next level up and believe $K_{Intel} \Rightarrow$ `Intel.com` because K_{com} says it and we know $K_{com} \Rightarrow$`com`. Taking one more step, we get to the root of the DNS hierarchy; secure DNS [7] would let us take these steps if it were ever deployed.

This is fine for everyday use. Indeed, it's exactly what browsers do when they rely on Verisign to authenticate the DNS names of web servers, since they trust Verisign for any DNS name. It puts a lot of trust in Verisign or the DNS root, however, and if tight security is needed, people will prefer to establish direct relationships like the Intel-Microsoft one.

Why not always have direct relationships? They are a nuisance to manage, since each one requires exchanging a key in some manual way, and making some provisions for changing the key in case it's compromised.

Naming is a form of multiplexing, in which a principal P is extended to a whole family of subprincipals P/N_1, P/N_2, etc.; such a family is usually called a "name space". Other kinds of multiplexing, like running several TCP connections over a host-to-host connection secured by IPSec, can use the same "parent delegates to child" scheme. The quoting principals of [11, 13] are another example of multiplexing. SPKI [8] is the most highly developed form of this view of secure naming.

3.6 Authenticating Systems

As we saw in section 3.4, we can treat a program image, represented by its secure hash, as a principal; the hash plays the same role as an encryption key. But a program can't make statements; to do so, it must be *loaded* into a *host* H. Booting an operating system is a special case of loading. A loaded program depends on the host it runs on; if you don't trust the host, you certainly shouldn't trust the running program.

There are four steps in authenticating a system S:

1) H needs to know something about the program image. It could simply rely on the image's file name, but to be conservative, H can compute a cryptographically secure hash or *digest* D_{SQL} of the image. If H runs the image with digest D_{SQL} as S, then $S \Rightarrow D_{SQL}$.
2) A digest, however, has the same drawback as a key: it's not meaningful to a person. So just as we bind a key to a user name with K_{Intel} says $K_{Alice} \Rightarrow$ `Alice@Intel`, we bind a digest to a program name with $K_{Microsoft}$ says $D_{SQL} \Rightarrow$ `Microsoft/SQLServer`. Now we have $S \Rightarrow D_{SQL}$

[12] `Alice@Intel.com` is just a variant syntax for `com/Intel/Alice`.

$\Rightarrow$ `SQLServer`. The host may also have an ACL of programs that it's willing to run, perhaps just `Microsoft/SQLServer`, perhaps `Microsoft/*`.

3) The host must authenticate a channel C from S to the rest of the world. The simplest way to do this is to make up a key pair (K_S, K_S^{-1}), give S the private key K_S^{-1}, and authenticate the public key K_S with H says $K_S \Rightarrow$ `SQLServer`. Now K_S acts as the channel C.
4) No one will believe this, however, unless they trust H to run `SQLServer`. So a third party needs to know H says $K_S \overset{\text{delegate}}{\Rightarrow}$ `SQLServer`.

There are four principals here: the executable file, the digest D_{SQL}, the running SQL server S, and the channel K_S to S. The chain of trust is $K_S \Rightarrow S \Rightarrow D_{SQL} \Rightarrow$ `SQLServer`. Of course only K_S actually transmits requests and only S actually generates them.

The NGSCB system [9] is an example of how to implement these ideas. Its aim is to provide a way to run newly written software on a PC with fairly high confidence that its execution isn't corrupted by a bad guy. Since existing operating systems are much too complicated to provide such confidence, the first step is to provide what amounts to a physically separate machine: hardware mechanisms ensure that this machine is isolated from the main OS. On this separate machine you run a new software stack, whose base is a new, small OS kernel called the *nexus*. More hardware stores a private key K_H for the machine and uses this key to sign a certificate for the nexus: K_H says $K_{nexus} \Rightarrow D_{nexus}$. In addition, the host uses its private key to encrypt data on behalf of the nexus, which it will decrypt only for a program with the same digest. The nexus in turn loads applications and provides the same services for them.

3.7 Variations

There are many variations in the details of setting up a chain of trust:

- How secure channels are implemented.
- How bytes are stored and moved around.
- Who collects the evidence.
- Whether evidence is summarized.
- How big objects are and how expressive T is.
- What compound principals exist other than names.

We pass over the complicated details of how to use encryption to implement secure channels. They don't affect the overall system design much, and problems in this area are usually caused by over-optimization or sloppiness [1]. We touch briefly on the other points, each of which could fill a paper by itself.

Handling bytes. The details of how to store and send around the bytes that represent messages are not directly relevant to security as long as the bytes are properly encrypted, but they make a big difference to the system design and performance. In analyzing security, it's important to separate the secure channels (usually recognizable by encryption at one end and decryption at the other) from the ordinary channels. Since the latter don't affect security, you can choose the flow and storage of encrypted bytes to optimize simplicity, performance, or availability.

The most important point here is the difference between public and shared key encryption. Public key allows a secure off-line *broadcast* channel. You can write a certificate on a tightly secured offline system and then store it in an untrusted system; any number of readers can fetch and verify it. To do broadcast with shared keys, you need a trusted on-line relay system. There's nothing wrong with this in principle, but it may be hard to make it both secure and highly available.

Contrary to popular belief, there's nothing magic about certificates. The best way to think of them is as secure answers to pre-determined queries (for example, "What is Alice's key?"). You can get the same effect by querying an on-line database that maps users to keys, as long as you trust the database server and the secure channel to it.

Caching is another aspect of where information is stored. It can greatly improve performance, and it doesn't affect security or availability as long as there's always a way to reload the cache if gets cleared or invalidated. This last pint is often overlooked.

Collecting evidence. The verifier (the guard that's granting access to an object) needs to see the evidence from each link in the chain of trust. There are two basic approaches to collecting this information:

Push: The client gathers the evidence and hands it to the object.

Pull: The object queries the client and other databases to collect the evidence it needs.

Most systems use push for authentication, the evidence for the identity of the client, and pull for authorization, the evidence for who can do what to the object. The security tokens in Windows are an example of push, and ACLs an example of pull. Push may require the object to tell the client what sort of evidence it needs; see [8, 11] for details.

If the client is feeble, or if some authentication information such as group memberships is stored near the object, more pull may be good. Cross-domain authentication in Windows is a partial example: the target domain controller, rather than the login controller, discovers membership in groups that are local to the target domain.

Summarizing evidence. It's possible to replace several links of a chain like $P \Rightarrow Q \Rightarrow R$ with a single link $P \Rightarrow R$ signed by someone who speaks for R. In the limit a link signed by the object summarizes the whole chain; this is usually called a capability. An open file descriptor is a familiar example; it summarizes the access rights of a process to a file, which are checked when the process opens the file. The advantages are savings in space and time to verify, which are especially important for feeble objects such as computers embedded in small devices. The drawbacks are that it's harder to do setup and to revoke access.

Expressing sets of statements. Traditionally an object groups its methods into a few sets (for example, files have read, write, and execute methods), permissions for these sets of requests appear on ACLs, and there's no way to restrict the set of statements in other delegations. SPKI [8] uses "tags" to define sets of statements and can express unions and intersections of sets in any delegation, so you can say things like "`Alice` ⇒ `Atom` for reads of files named `*.doc` and purchase orders less than $5000". This example illustrates the tradeoff between the size of objects and the expressiveness of T: instead of separate permissions for each `.doc` file, there's a single one for all of them.

Compound principals. We discussed named or multiplexed principals in section 3.5. Here are some other examples of compound principals:

- Conjunctions: `Alice and Bob`. Both must make a statement for the conjunction to make it. This is very important for commercial security, where it's called "separation of duty" and is intended to make insider fraud harder by forcing two insiders to collude. SPKI has a generalization to threshold principals or "*k* out of *n*" [8].
- Disjunctions: `Alice or FlakyProgram`. An object must grant access to both for this principal to get it. In Windows this is a "restricted token" that makes it safer for Alice to run a

flaky program, since a process with this identity can only touch objects that explicitly grant access to `FlakyProgram`, not all the objects that Alice can access.

Each kind of compound principal has axioms that define who can speak for it. See [13] for other examples.

3.8 Auditing

As is typical for computer security, we have focused on how end-to-end access control works and the wonderful things you can do with it. An equally important property, though, is that the chain of trust collects in one place, and in an explicit form, all the evidence and rules that go into making an access control decision. You can think of this as a *proof* for the decision. If the guard records the proof in a reasonably tamper-resistant log, an auditor can review it later to establish accountability or to figure out whether some unintended access was granted, and why.

Since detection and punishment is the primary instrument of practical security, this is extremely important.

4 Conclusion

We have outlined the basic ideas of computer security: secrecy, integrity, and availability, implemented by access control based on the gold standard of authentication, authorization, and auditing. We discussed the reasons why it doesn't work very well in practice:

- Reliance on prevention rather than detection and punishment.
- Complexity in the code and especially in the setup of security, which overwhelms users and administrators.

We gave some ways to reduce this complexity.

Then we explained how to do access control end-to-end in a system with no central management, by building a chain of trust based on the "speaks for" relation between principals. Each link in the chain is a delegation of the form "`Alice@Intel` speaks for `Atom.Microsoft` about reads and writes of files named `*.doc`". The right principals (Intel, Microsoft, and the file owner in this example) has to assert each link, using a secure channel. Every kind of authentication and authorization information fits into this framework: encrypted channels, user passwords, groups, `setuid` programs, and ACL entries. The chain of trust is a sound basis for logging and auditing access control decisions.

Principals with hierarchical names are especially important. A parent can delegate for all of its children. Rooting name spaces in keys avoids any need for a globally trusted root.

There are many ways to vary the basic scheme: how to store and transmit bytes, how to collect and summarize evidence for links, how to express sets of statements, and what is the structure of compound principals.

References

1. Abadi and Needham, Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Engineering* **22**, 1 (Jan 1996), 2-15, dlib.computer.org/ts/books/ts1996/pdf/ e0006.pdf or gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-25.html
2. Anderson, Why cryptosystems fail. *Comm. ACM* **37**, 11 (Nov. 1994), 32-40, www.acm.org/pubs/citations/ proceedings/commsec/168588/p215-anderson
3. Bell and LaPadula, Secure computer systems. ESD-TR-73-278 (Vol. I-III) (also Mitre TR-2547), Mitre Corporation, Bedford, MA, April 1974

4. CERT Coordination Center, CERT advisory CA-2000-04 Love Letter Worm, www.cert.org/advisories/CA-2000-04.html
5. Clark and Wilson, A comparison of commercial and military computer security policies. *IEEE Symp. Security and Privacy* (April 1987), 184-194
6. Denning, A lattice model of secure information flow. *Comm. ACM* **19**, 5 (May 1976), 236-243
7. Eastlake and Kaufman, *Domain Name System Security Extensions*, Jan. 1997, Internet RFC 2065, www.faqs.org/ rfcs/rfc2065.html
8. Ellison et al., *SPKI Certificate Theory*, Oct. 1999, Internet RFC 2693, www.faqs.org/rfcs/rfc2693.html
9. England et al., A trusted open platform, *IEEE Computer* **36**, 7 (July 2003), 55-62.
10. Gray, J., personal communication
11. Howell and Kotz, End-to-end authorization, *4th Usenix Symp. Operating Systems Design and Implementation*, San Diego, Oct. 2000, www.usenix.org/publications/library/ proceedings/osdi00/howell.html
12. Lampson, Protection. *ACM Operating Systems Rev.* **8**, 1 (Jan. 1974), 18-24, research.microsoft.com/lampson/09-Protection/Abstract.html
13. Lampson et al, Authentication in distributed systems: Theory and practice. *ACM Trans. Computer Systems* **10**, 4 (Nov. 1992), pp 265-310, www.acm.org/pubs/citations/ journals/tocs/1992-10-4/p265-lampson
14. Myers and Liskov, A decentralized model for information flow control, *Proc. 16th ACM Symp. Operating Systems Principles*, Saint-Malo, Oct. 1997, 129-142, www.acm.org/ pubs/citations/proceedings/ops/268998/p129-myers
15. Oasis, Web services security, oasis-open.org/committees/wss.
16. Rivest, Shamir, and Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* **21**, 2 (Feb., 1978), 120-126, theory.lcs.mit.edu/~rivest/ rsapaper.ps
17. National Research Council, *Computers at Risk: Safe Computing in the Information Ag*e. National Academy Press, Washington D.C., 1991, books.nap.edu/catalog/1581.html
18. National Research Council, *Realizing the Potential of C4I: Fundamental Challenges*. National Academy Press, Washington D.C., 1999, books.nap.edu/catalog/6457.html
19. Saltzer, Protection and the control of information sharing in Multics. *Comm. ACM* **17**, 7 (July 1974), 388-402
20. Saltzer et al., End-to-end arguments in system design. *ACM Trans. Computer Systems* **2**, 4 (Nov. 1984), 277-288, web. mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf
21. Schneier, Secrets and Lies: Digital Security in a Networked World, Wiley, 2000.
22. Schneier and Mudge, Cryptanalysis of Microsoft's point-to-point tunneling protocol. *5th ACM Conf. Computer and Communications Security*, San Francisco, 1998, 132-141, www.acm.org/pubs/citations/proceedings/commsec/ 288090/p132-schneier
23. Wobber et al., Authentication in the Taos operating system. *ACM Trans. Computer Systems* **12**, 1 (Feb. 1994), pp 3-32, www.acm.org/pubs/citations/journals/tocs/1994-12-1/p3-wobber
24. ZDNet, Stealing credit cards from babies. ZDNet News, 12 Jan. 2000, www.zdnet.com/zdnn/stories/news/0,4586, 2421377,00.html
25. ZDNet, Major online credit card theft exposed. ZDNet News, 17 Mar. 2000, www.zdnet.com/zdnn/stories/news/0, 4586,2469820,00.html