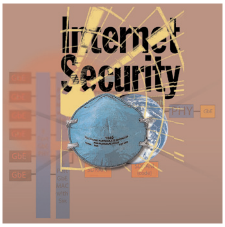
IEEE Computer, June 2004

Computer Security in the Real World



Most computers are insecure because security is expensive. Security depends on authentication, authorization, and auditing: the gold standard. The key to uniform security in the Internet is the idea of one security principal speaking for another.

Butler W. Lampson
Microsoft

Computer system security is more than 30 years old. It has had many intellectual successes, among them the subject/object access matrix model,[1] access control lists,[2] multilevel security using information flow[3,4] and the star property,[5] public-key cryptography,[6] and cryptographic protocols.[7] Despite these successes, in an absolute sense the security of the hundreds of millions of deployed computer systems remains terrible. A determined and competent attacker could steal or destroy most of the information on most of these systems. Even worse, the attacker could do this to millions of systems at once.

The Internet has made computer security much more difficult. Twenty years ago, a computer system had a few dozen users at most, all members of the same organization. Today, half a billion people all over the world connect to the Internet. Anyone can attack your system. Your system, if compromised, can infect others automatically. You face possibly hostile code that comes from many different sources, often without your knowledge. Your laptop faces a hostile physical environment. If you own content and want to sell it, you face hostile hosts. You can't just isolate yourself, because you may want to share information with anyone or run code from anywhere.

These vulnerabilities invite vandalism: worms and viruses. They also make it much easier to attack a specific target, either to steal information or to corrupt data. On the other hand, the actual harm these attacks cause is limited, though increasing. Unfortunately, there is no accurate data about the cost of computer security failures: Most are never made public for fear of embarrassment, but when a public incident does occur, security experts and vendors have every incentive to exaggerate its costs.

Money talks, though. Many companies have learned that although people may complain about inadequate security, they won't spend much money, sacrifice many features, or put up with much inconvenience to improve it. This strongly suggests that bad security is not really costing them much. Firewalls and antivirus 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 past 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 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. 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 to succeed.

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 any absolute standard. It's easy to break into someone's house; in fact, in many places people don't even bother to lock their doors. 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*.

Practical security balances the cost of protection and the risk of loss.

The bad guys balance 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. But these boxes 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), 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.

WHAT IS COMPUTER SECURITY?

With computers, 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 national security needs have dominated work on computer security has made this problem worse. Because the stakes are much higher and no police or courts are available to punish attackers, not making mistakes is more important. Further, 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. This reasoning ignores two critical facts.

First, software is complicated, and in practice it's impossible to make it perfect. Even worse, security must be set up, and in a world of legacy hardware and software, networked computers, mobile code, and constantly changing relationships between organizations, setup is complicated too.

Second, security gets in the way of other things you want. For software developers, security interferes with features and with time to market. For users and administrators, security interferes with getting work done conveniently or, in some cases, at all. This is more important, since there are more users than developers.

Security setup also takes time, and it contributes nothing to useful output. Furthermore, no one will notice that a setup is too permissive unless there's an audit or an attack. This leads to such things as users whose password is their first name, a company in which more than half of the installed database servers have a blank administrator password, public access to databases of credit card numbers,[8,9] or e-mail clients that run attachments containing arbitrary code with the user's privileges.[10]

The 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 must be implemented correctly. This means that it takes more time to build, so naturally it lacks the latest features.

A secondary reason we don't have "real" security is that systems are complicated, therefore both the code and the setup have bugs that an attacker can exploit, such as buffer overruns or other flaws that break the basic programming abstractions. 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.

Studying a secure system involves three aspects:

- *Specification/Policy*: What is the system supposed to do?
- *Implementation/Mechanism*: How does it do it?
- *Correctness/Assurance*: Does it really work?

The first name for each aspect is the one in general use throughout computing, while the second is the special name used in the security world.

POLICY: SPECIFYING SECURITY

Organizations and people that use computers can describe their needs for information security under four major headings:[11]

- *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.

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

• Damage to information	integrity
• Disruption of service	availability
• Theft of money	integrity
• Theft of information	secrecy
• Loss of privacy	secrecy

Each computer user must decide what security means. A description of the user's needs for security is called a *security policy.*

Computer security policies usually derive from policies for real-world security. The military is most concerned with secrecy, ordinary businesses with integrity and accountability, and 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, cause the firing of a missile, or arm a nuclear weapon. Secrecy is important in commercial applications as well: Financial and personnel information must not be disclosed to outsiders. Nonetheless, the difference in emphasis remains.[12]

A security policy has both a positive and a 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 guys. In practice, however, the positive aspect gets more attention, since too little access keeps people from getting their work done, which draws attention immediately. However, too much access goes undetected until there's a security audit or an obvious attack, which rarely happens. This distinction between talk and practice pervades the security field.

> ***The unavoidable price of reliability is simplicity.***
> **—Hoare**

MECHANISM: IMPLEMENTING SECURITY

One man's policy is another man's *mechanism*. Before a computer system can enforce it, the informal access policy in the previous section must be expanded to precisely describe both the set of confidential information and the set of properly authorized employees. We can view these descriptions as more detailed policy or as implementation of the informal policy.

Security implementation has two parts: code and setup. The code is the programs that security depends on. The setup is all the data that controls the programs' operations: folder structure, access control lists, group memberships, user passwords or encryption keys, and so on.

A security implementation must defend against vulnerabilities, which take three main forms: bad—buggy or hostile—programs; bad—careless or hostile—agents, either programs or people, giving bad instructions to good but gullible programs; and bad agents that tap or spoof communications. Careless or hostile agents can cascade 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:

- *Isolate*—keep everybody out. This coarse-grained strategy provides the best security, but it keeps users from sharing information or services. This is impractical for all but a few applications.
- *Exclude*—keep the bad guys out. This medium-grained strategy makes it all right for programs inside this defense to be gullible. Code signing and firewalls do this.
- *Restrict*—let the bad guys in, but keep them from doing damage. This fine-grained strategy, also known as sandboxing, can be implemented traditionally with an operating system process or with a more modern approach that uses 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, and it's hard to get this right.

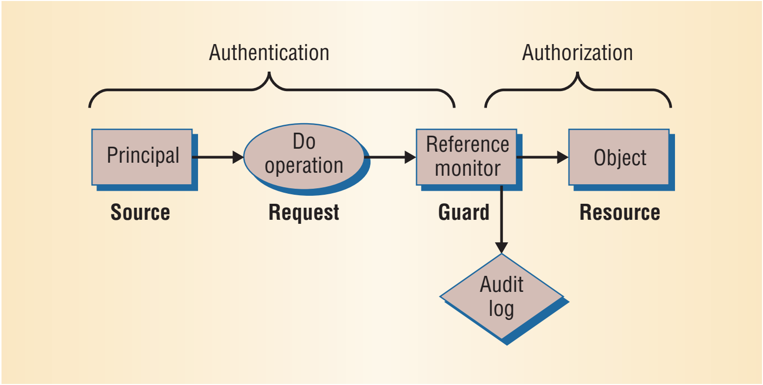


Figure 1. Access control model. A guard controls access to valuable resources, deciding whether the source of the request, called a principal, is allowed to do the operation on the object.

- *Recover*—undo the damage. This strategy, exemplified by backup systems and restore points, doesn't help with secrecy, but it does help with integrity and availability.
- *Punish*—catch the bad guys and prosecute them. Auditing and police do this.

Figure 1 shows the well-known *access control model* that provides the framework for these strategies. In this model, requests for service arrive at valuable resources, which usually are encapsulated in objects. A guard decides whether the source of the request, called a *principal*, is allowed to do the operation on the object.

To decide, the guard 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 states who is allowed to do what to the object. The guard is separate from the object to keep the guard simple and therefore more likely to be correct.

Security is mainly up to the guard, but it still depends on the object to implement its methods correctly. For example, 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 becomes insecure despite the guard.

Another model, called *information-flow control*, works better when secrecy in the face of bad programs is a primary concern.[3,4] This is roughly a dual of the access control model: The guard decides whether information can flow to a principal.

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

- *authenticating principals*—determines who made a request; principals usually are people, but they also can be groups, channels, or programs;
- *authorizing access*—determines who is trusted to do which operations on an object; and
- *auditing the guard's decisions*—makes it possible to determine later what happened and why.

ASSURANCE: MAKING SECURITY WORK

Making security work requires establishing a *trusted computing base*. The TCB is the collection of hardware, software, and setup information on which a system's security depends. For example, if the security policy for a LAN's machines mandates that they can access the Web but no other Internet services, and no inward access is allowed, the TCB is just the firewall that allows outgoing port 80 TCP connections but no other traffic. If the policy also states that no software downloaded from the Internet should run, the TCB also includes the browser code and settings that disable Java and other software downloads.

The idea of a TCB is closely related to the end-to-end principle—just as reliability depends only on the ends, security depends only on the TCB.[13] In both cases, performance and availability aren't guaranteed. Unfortunately, it's hard to figure out what is in the TCB for a given security policy. Even writing the specs for the components is hard.

Defense in depth through redundant security mechanisms is a good way to make defects in the TCB less harmful. For example, a system might include

- network-level security, using a firewall;
- operating system or virtual machine security that uses sandboxing to isolate programs; and
- application-level security that checks authorization directly.

An attacker must find and exploit flaws in all the levels. Defense in depth offers no guarantees, but it does seem to help in practice.

Although most discussions of assurance focus on the software, there is another important TCB component: 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, including

- what installed software has system or user privileges—not just binaries, but anything executable, such as shell scripts or macros;
- the database of users, passwords, privileges, and group memberships; services like SQL servers often have their own user database;
- network information such as lists of trusted machines; and
- the access controls on all system resources: files, devices, services.

Setup is much simpler than code, but it is still complicated and usually is done by less skilled people. Worse, while code is written once, setup is different for every installation, and it is based on documentation that is usually voluminous, obscure, and incomplete. Therefore, we should expect that the setup usually is wrong, and many studies confirm this. Ross Anderson[14] gives an eye-opening description of insecure setup in financial cryptosystems, the National Research Council[15] does the same for the military, and Bruce Schneier[16] gives many other examples.

To solve this problem, security setup must be much simpler for both administrators and users. They need a simple model for security with a small number of settings. What form should this model take?

Users need a simple story with about three levels of security—me, my group or company, and the world—each with progressively less authority. Browsers classify the network this way today. The corresponding data should be in three separate 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 safe deposit boxes. Vendors or administrators should handle everything else.

In particular, the system should classify all programs as trusted or untrusted based on how they are signed, unless the user explicitly says otherwise. It can either reject or sandbox untrusted programs. Sandboxed programs must run in a completely separate world with a separate global state: user and temporary folders, history, Web caches, and so on. There should be no communication with the trusted world except when the user explicitly copies something by hand, or by network file sharing. This is a bit inconvenient, but anything else is bound to be unsafe.

Administrators still need a fairly simple story, but even more they need the ability to handle many users and systems uniformly because they can't deal effectively with numerous individual cases. The way to do this is with *security policies*, rules for security settings that are applied automatically to groups of machines. These rules should say things like:

- Each user has read and write access to a 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.
- A trusted authority must sign all executable programs.

While code is written once, setup is different for every installation.

Since it's too hard for most administrators to invent them from scratch, such policies usually should be small variations on templates that vendors provide and test. Backward compatibility should be off by default because administrators can't deal with its complex security issues.

Because some customers will insist on special cases, it should be easy to report all the exceptions 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 because long ones will surely be ignored.

To make the policies manageable, administrators must define *groups* of users and resources, then state the policies in terms of these groups. Ideally, resource groups follow the file system structure, but the baroque conventions in existing networks, systems, and applications require other options as well.

To handle repeating patterns of groups, system architects can define *roles*, which are to groups as classes are to objects in Java. Thus, 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 on. The Appliance division has a specific group for Appliance-members, Appliance-budget, and so forth; thus, Appliance-finance will have write access to Appliance-budget.

The most practical way to implement policies is to compile them into existing security settings, treating the settings as a machine language. This means that existing resource managers don't have to change. It also allows for both powerful high-level policies and efficient enforcement, just as compilers allow for both powerful programming languages and efficient execution.

Developers also need help with security. A type-safe virtual machine like Java or Microsoft's .NET framework will eliminate many bugs automatically. Unfortunately, many security bugs are in system software that talks to the network, and it will be a while before developers write this code in a type-safe world. Developers also need a process that takes security seriously, values designs that make

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

assurance easier, gets those designs reviewed by security professionals, and refuses to ship code with serious security flaws.

END-TO-END ACCESS CONTROL

Secure distributed systems need a way to handle authentication and authorization uniformly throughout the Internet. Several reports explain in detail how to do this,[17-20] and they are the basis for recent Web services security proposals.[21]

Local access control

Most existing systems, such as Unix and Windows, do authentication and authorization locally. They have local databases for user authentication—usually a password file—and for authorization—usually an access control list (ACL) on each resource. They rely on physical security or luck to secure the channel to the user, or they use an encrypted channel protocol like the Point-to-Point Tunneling Protocol. Web server security works the same way. Servers usually use Secure Socket Layer (SSL) to secure the user channel. Each server farm has a separate local user database.

A slight extension is to put each system into a *domain* and store the authentication database centrally on a domain controller. To log in a user, the local system sends the controller a message that includes the user's password or challenge response. The controller does exactly what the local system used to do. Kerberos, Windows domains, and Passport all work this way. To authenticate the user to another system, the login system can ask the controller to forward the authentication; Kerberos calls this a *ticket*.[22] Shared keys between machines secure the communication. The entire domain is under the same management.

Distributed access control

A distributed system can involve systems and people that belong to different organizations and are managed differently. Consider the following example.

Alice, an Intel employee, belongs to a team working on a joint Intel-Microsoft project called Atom. She logs in, using a smart card to authenticate herself, and uses SSL to connect to a project Web page at Microsoft called Spectra. The Web page grants her access according to a five-step process:

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 cryptographically sign a response to a challenge from the Microsoft server.
3. Intel certifies that K_{Alice} is the key for Alice@Intel.com.
4. Microsoft's group database says that Alice@Intel.com is in the Atom@Microsoft group.
5. The ACL on the Spectra page says that Atom has read/write access.

In this example many different kinds of information contribute to the access control decision: authenticated session keys, user passwords or public keys, delegations from one system to another, group memberships, and ACL entries. They are all different cases of a single mechanism.

Chains of trust

A *chain of trust* runs from the SSL channel at one end of the example 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_{Alice} speaks for Alice@Intel 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 for statements in T. Put another way, Q takes responsibility for anything P says about T. The notion of principal is very general, encompassing any entity that makes statements. In the example, keys, people, groups, systems, program images, and resource objects are all principals.

The idea of "about subjects T" is that T is a way to describe a set of things that P might say. We can think of T as a pattern that characterizes these statements. In the example, T is "all statements" except for Step 5, where it is "read and write requests." It's the object's guard that decides whether the request is in T, so different objects can have different encodings for T. For example, for file access, T could be "read and write requests for files whose names match ~lampson/security/*.doc." SPKI develops this idea in some detail.[17]

We can abbreviate "P speaks for Q about T" as $P \overset{T}{\Rightarrow} Q$, or just $P \Rightarrow Q$ if T is "all statements." Here, $\Rightarrow$ is short for "speaks for." With this notation, the chain of trust for the example is: $K_{SSL} \Rightarrow K_{Alice} \Rightarrow \text{Alice@Intel} \Rightarrow \text{Atom@Microsoft} \overset{r/w}{\Rightarrow} \text{Spectra}$.

Figure 2 shows how the chain of trust relates the various principals. Note that the speaks-for arrows

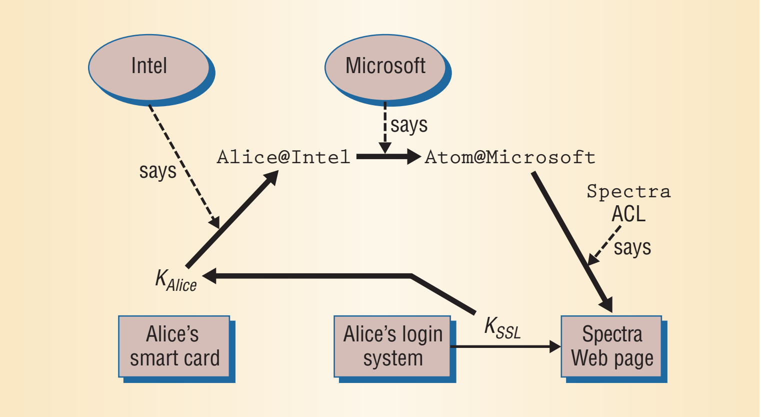


Figure 2. Chain of trust. The chain relates the various principals: Trust flows clockwise around the loop, but data flows on K_{SSL} from Alice's login system to Spectra.

are independent of the data flow: Trust flows clockwise around the loop, but no data traverses this path.

Evidence for the links

What establishes a link in the chain, that is, a fact $P \Rightarrow Q$? Some *verifier*, either the object's guard or a later auditor, needs to see evidence for the link. The evidence has the form "principal says delegation," where a *delegation* is a statement of the form $P \overset{T}{\Rightarrow} Q$ that delegates Q's authority for T to P: Anything that P says about T will be taken as something that Q says.

The principal is taking responsibility for the delegation. So we must answer three questions.

Why trust the principal? We trust Q for $P \Rightarrow Q$, that is, we believe it if Q says it. This *delegation rule* is justified because Q, as a responsible adult or the computer equivalent, should be allowed to delegate its authority.

Who says? Second, we must establish how we know that Q says $P \overset{T}{\Rightarrow} Q$. The answer depends on how Q does the saying. If Q is a key, then "Q says X" means that Q cryptographically signs X, something that a program can easily verify.[6] This case applies for $K_{SSL} \Rightarrow K_{Alice}$. If K_{Alice} signs it, the verifier believes that K_{Alice} says it, and therefore trusts it by the delegation rule. If, on the other hand, 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. These are the only ways the verifier can directly know who said something: by receiving it on a secure channel or by storing 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 $\overset{delegate}{\Longrightarrow}$ 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 store $K_{Intel} \Rightarrow$ Intel locally. Then signing by the key K_{Intel} forms the secure channel. If Microsoft and Intel establish a direct relationship, Microsoft will know Intel's public-key K_{Intel}. We don't want to install $K_{Intel} \Rightarrow$ Intel explicitly on every Microsoft server, so we install it in a Microsoft-wide directory MSDir. All the other servers have secure channels to the directory and trust it unconditionally to authenticate principals outside Microsoft. We only need to install the public-key K_{MSDir} and the delegation "$K_{MSDir} \Rightarrow$ * except *.Microsoft.com" in each server.

The remaining case is the group membership Alice@Intel ⇒ Atom@Microsoft. Just as Intel $\overset{delegate}{\Longrightarrow}$ Alice@Intel, so Microsoft $\overset{delegate}{\Longrightarrow}$ Atom@Microsoft. Therefore, Microsoft should make this delegation.

Why is the principal willing? Third, we must know why a principal should make a delegation. The reasons vary greatly. Some facts are installed manually, such as $K_{Intel} \Rightarrow$ Intel at Microsoft, when the companies establish a direct relationship. Others follow from the properties of some algorithm. For example, if a principal P runs a Diffie-Hellman key exchange protocol that yields a fresh shared-key K_{DH}, and P doesn't disclose K_{DH}, then P should be willing to say "$K_{DH} \Rightarrow P$, provided you are on 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 to yourself." In practice, P does this simply by signing $K_{DH} \Rightarrow K_P$; the qualifiers are implicit in running the Diffie-Hellman protocol.

Names

Why did we say that Intel $\overset{delegate}{\Longrightarrow}$ Alice@Intel? Someone must speak for Alice@Intel unless we want to install facts about it manually, which is tedious and error prone. The parent of a name is the most natural principal to delegate its authority. This is the point of hierarchical naming: Parents have authority over children. Formally, we have the axiom $P \overset{delegate}{\Longrightarrow} P/N$ for any principal P and simple name N; Alice@Intel is just a variant syntax for Intel/Alice.

The simplest case is when P is a key. It is simple because you don't need to install anything to use it. This means that every key is the root of a name space. If K is a public key, it says $Q \Rightarrow K/N$ by signing a certificate with this content. The certificate is public, and anyone can verify the signature and should then believe $Q \Rightarrow K/N$.

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 if K_{Intel} says "$K_{Alice} \Rightarrow$ Alice@Intel" we can believe it. As always, one way

If you don't trust the host, you certainly shouldn't trust the running program.

to establish this is to install $K_{Intel} \Rightarrow$ Intel manually, a direct relationship with Intel. Another way uses hierarchical naming at the next level up—we believe that $K_{Intel} \Rightarrow$ 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[23] would let us take these steps if it were ever deployed. Indeed, this is exactly what browsers do when they trust Verisign to authenticate a Web server's DNS name.

This 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 because each one requires exchanging a key manually and making some provisions for changing the key in case it's compromised.

Authenticating systems

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* by 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* running on a host *H*:

1. *H* needs to know something about the program image, preferably 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 $S \Rightarrow D_{SQL} \Rightarrow$ SQLServer. The host also can have an ACL of programs that it's willing to run, perhaps just Microsoft/SQLServer, perhaps Microsoft/*.
3. The host must authenticate a channel 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 is the channel.
4. A third party won't believe this, however, unless it trusts *H* to run SQLServer. So a third party needs to know $H \overset{delegate}{\Rightarrow}$ 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.

The Next-Generation Secure Computing Base system[24] is one way to implement these ideas. NGSCB aims to provide a way to run newly written software on a PC with fairly high confidence that a malicious intruder doesn't corrupt its execution. Since existing operating systems are too complicated to provide such confidence, the first step is to provide what amounts to a physically separate machine: Hardware mechanisms keep this machine isolated from the main OS. This separate machine runs a new software stack, whose base is a small virtual machine monitor called a nexus or *hypervisor*. More hardware stores a private-key K_M for the machine and uses this key to sign a certificate for the hypervisor: K_M says $K_{hypervisor} \Rightarrow D_{hypervisor}$. In addition, the machine uses its private key to encrypt data on behalf of the hypervisor, which it will decrypt only for a program with the same digest. The hypervisor in turn loads applications and provides the same services for them, just like any other operating system.

Variations

A chain of trust can vary in many details, including how to implement secure channels, how to store and transmit bytes, who collects the evidence, whether to summarize evidence, how expressive *T* is, and what compound principals exist other than names. Encryption is the usual way to implement a secure channel. Martin Abadi and Roger Needham[7] explain how to do it properly and give references to the existing literature.

Handling bytes. In analyzing security, it's important to separate the secure channels—usually recognizable by encryption at one end and decryption at the other—from ordinary channels. The latter don't affect security, so we can choose the flow and storage of encrypted bytes to optimize simplicity, performance, or availability. The most important choice is between public-key and shared-key encryption.

Public-key encryption allows a secure offline *broadcast* channel. You can write a certificate on a tightly secured offline system, then store it in an untrusted system so that any number of readers can fetch and verify it. Doing broadcast with shared keys requires a trusted online relay. 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 public-key certificates. The best way to think of them is as secure answers to predetermined queries. You can get the same effect by querying an online database as long as you trust the database server and the secure channel to it. Kerberos works this way.[22]

Caching is another aspect of information storage. 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 point is often overlooked.

Collecting evidence. The verifier needs to see the evidence from each link in the chain of trust. In the *push* approach, the client gathers the evidence and hands it to the object. In the *pull* approach, the object queries the client and other databases to collect the evidence it needs.

Most systems use push for authentication and pull for authorization. Security tokens in Windows are an example of push, access control lists are an example of pull. Push may require the object to tell the client what sort of evidence it needs.[17,18]

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 an example: The target domain controller, rather than the login controller, discovers membership in groups 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, the object signs a link that summarizes the whole chain; this is usually called a capability. An open file descriptor is a familiar example that summarizes the access rights of a process to a file, which are checked when the process opens the file. Capabilities save space and time to verify, which are especially important for feeble objects such as computers embedded in small devices, at the expense of more complicated setup and revocation of access.

Expressing sets of statements. Traditionally, an object groups its methods into a few sets, such as read, write, and execute operations on files. ACLs hold permissions for these sets, but other delegations are unlimited. SPKI[17] uses tags to define sets of statements and can express unions and intersections of sets in any delegation so that we can say things like "Alice ⇒ Atom for reads of files named *.doc and purchase orders less than $5,000."

Compound principals. Names are compound principals; other examples are *conjunctions* and *disjunctions*.[19] Conjunctions such as Alice and Bob consist of two principals, and the conjunction makes a statement only if both of them 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.

The chain of trust collects all the evidence and rules that go into making an access control decision.

Disjunctions such as Alice or FlakyProgram also consist of two principals. An object must grant access to both for the disjunction to get it. In Windows, this is a *restricted token* that makes it safer for Alice to run a flaky program, because a process with this identity can only touch objects that explicitly grant access to FlakyProgram, not all the objects that Alice can access.

Auditing

In addition to implementing end-to-end access control, the chain of trust also collects in one place, in an explicit form, all the evidence and rules that go into making an access control decision. This data serves 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 determine whether the system granted some unintended access and why. Since detection and punishment are the primary instruments of practical security, this is extremely important.

Most computers today are insecure because security is costly in terms of user inconvenience and foregone features, and people are unwilling to pay the price. Real-world security depends more on punishment than on locks, but it's hard to even find network attackers, much less punish them. The basic elements of security are authentication, authorization, and auditing: the gold standard. The idea of one principal speaking for another is the key to doing these uniformly across the Internet.

In the future, type-safe programming systems such as Java or Microsoft's .NET framework and more careful attention to secure programming will continue to reduce low-level security bugs. Setting up security is still much too complicated for ordinary users, but this too is gradually improving.

Fundamentally, better security requires punishing malefactors. This is not mainly a matter of laws, but of being able to track them down. To make that possible, anything that tries to enter your computer should be rejected unless it comes from a real-world source that you can hold accountable. The Internet is likely to evolve in this direction. ■

References

1. B. Lampson, "Protection," *ACM Operating Systems Rev.*, vol. 8, no. 1, 1974, pp. 18-24.
2. J.H. Saltzer, "Protection and the Control of Information Sharing in Multics," *Comm. ACM*, July 1974, pp. 388-402.
3. D.E. Denning, "A Lattice Model of Secure Information Flow," *Comm. ACM*, May 1976, pp. 236-243.
4. A.C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," *Proc. 16th ACM Symp. Operating Systems Principles*, ACM Press, 1997, pp. 129-142; www.acm.org/pubs/citations/proceedings/ops/268998/p129-myers.
5. D.E. Bell and L.J. LaPadula, *Secure Computer Systems*, tech. report M74-244, Mitre Corporation, 1974.
6. R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, Feb. 1978, pp. 120-126.
7. M. Abadi and R.M. Needham, "Prudent Engineering Practice for Cryptographic Protocols," *IEEE Trans. Software Eng.*, vol. 22, no. 1, 1995, pp. 2-15.
8. ZDNet, "Stealing Credit Cards from Babies," *ZDNet News*, 12 Jan. 2000; www.zdnet.com/zdnn/stories/news/0,4586,2421377,00.html.
9. ZDNet, "Major Online Credit Card Theft Exposed," *ZDNet News*, 17 Mar. 2000; www.zdnet.com/zdnn/stories/news/0,4586,2469820,00.html.
10. CERT Coordination Center, "CERT Advisory CA-2000-04 Love Letter Worm," 2000; www.cert.org/advisories/CA-2000-04.html.
11. National Research Council, *Computers at Risk*, National Academies Press, 1991; http://books.nap.edu/catalog/1581.html.
12. D.D. Clark and D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 1987, pp. 184-194.
13. J.H. Saltzer et al., "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, Nov. 1984, pp. 277-288; http://Web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf.
14. R. Anderson, "Why Cryptosystems Fail," *Comm. ACM*, Nov. 1994, pp. 32-40.
15. National Research Council, *Realizing the Potential of C4I*, National Academies Press, 1999; http://books.nap.edu/catalog/6457.html.
16. B. Schneier, *Secrets and Lies: Digital Security in a Networked World*, Wiley, 2000.
17. C. Ellison et al., "SPKI Certificate Theory," Internet RFC 2693, Oct. 1999; www.faqs.org/rfcs/rfc2693.html.
18. J. Howell and D. Kotz, "End-to-End Authorization," *Proc. 4th Usenix Symp. Operating Systems Design and Implementation*, Usenix Assoc., 2000; www.usenix.org/publications/library/proceedings/osdi2000/howell.html.
19. B. Lampson et al., "Authentication in Distributed Systems: Theory and Practice," *ACM Trans. Computer Systems,* vol. 10, no. 4, ACM Press, 1992, pp. 265-310; www.acm.org/pubs/citations/journals/tocs/1992-10-4/p265-lampson.
20. E. Wobber et al., "Authentication in the Taos Operating System," *ACM Trans. Computer Systems*, Feb. 1994, pp. 3-32; www.acm.org/pubs/citations/journals/tocs/1994-12-1/p3-wobber.
21. Oasis Web Services Security TC; oasis-open.org/committees/wss.
22. B.C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Comm.*, Sept. 1994, pp. 33-38; gost.isi.edu/publications/kerberos-neuman-tso.html.
23. D. Eastlake and C. Kaufman, "Domain Name System Security Extensions," Internet RFC 2065, Jan. 1997; www.faqs.org/rfcs/rfc2065.html.
24. P. England et al., "A Trusted Open Platform," *Computer*, July 2003, pp. 55-62.

***Butler W. Lampson** is a distinguished engineer at Microsoft Research, where he works on systems architecture, security, and advanced user interfaces. He received the ACM's Turing Award in 1992, the IEEE's von Neumann Medal in 2001, and the NAE's Draper Prize in 2004. Lampson received a PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at blampson@microsoft.com.*