Practical Principles for Computer Security[1]

Butler Lampson
Microsoft Research
Marktoberdorf, August, 2006

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.

Real security is about punishment, not about locks; about accountability, not access control

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.

Security is about risk management

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 (be-

[1] My colleagues Martin Abadi, Carl Ellison, Charlie Kaufman, and Paul Leach made many suggestions for improvement and clarification. Some of these ideas originated in the Taos authentication system [4, 6].

cause 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.

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.

1 Implementing security

The job of computer security 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:

4) *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.

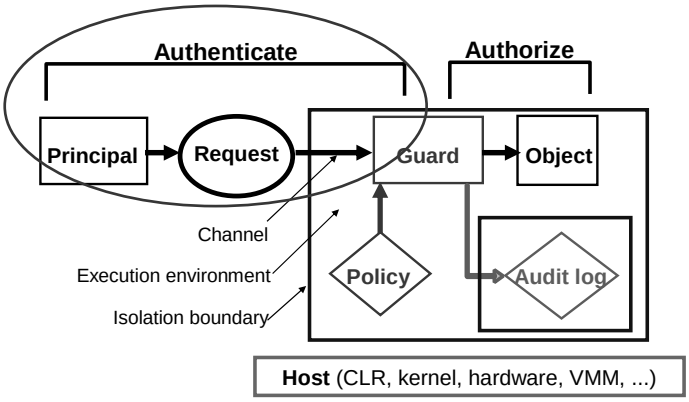

Figure 1: Access control model

5) *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.
6) *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.
7) *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.
8) *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 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. 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 [5]. This is roughly a dual of the access control model, in which the guard decides whether information can flow to a principal.

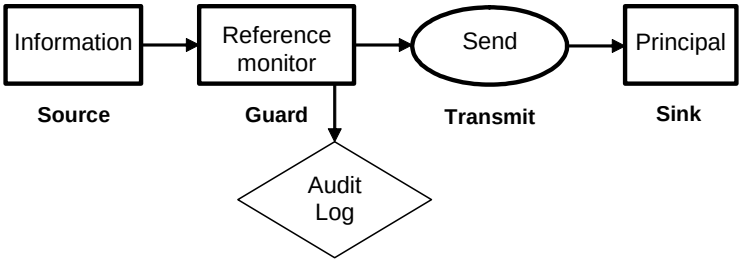

Figure 2: Information flow model

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 Access control

Figure 1 shows the overall model for access control. It says that *principals* make *requests* on *objects*; this is the basic paradigm of object-oriented programming or of services. The job of security is to decide whether a particular request is allowed; this is done by the *guard*, which needs to know who is making the request (the principal), what the request is, and what the target of the request is (the object). The guard is often called the *relying party*, since it relies on the information in the request and in policy to make its decision. Because all trust is local, the guard has the final say about how to interpret all the incoming information. For the guard to do its job it needs to see every request on the object; to ensure this the object is protected by an *isolation boundary* that blocks all access to the object except over a channel that passes through the guard. There are many ways to implement principals, requests, objects and isolation, but this abstraction works for all of them.

The model has three primary elements:

1. **Isolation**: This constrains the attacker to enter the protected execution environment via access-controlled channels.
2. **Access Control**: Access control is broken down into authentication, authorization, and auditing.
3. **Policy and User Model**: Access control policy is set by human beings—sometimes trained, sometimes not.

This paper addresses one piece of the security model: access control. It gives an overview that extends from setting authentication policy through authenticating a request to the

mechanics of checking access. It then discusses the major elements of authentication and authorization in turn.

2.1 What is access control

Every action that requires a security decision, whether it is a user command, a system call, or the processing of a message from the net, is represented in the model of as a request from a principal over a channel. Each request must pass through a guard or relying party that makes an access control decision. That decision consists of a series of steps:

1. Do *direct* authentication, which establishes the principal directly making the request. The most common example of this is verifying a cryptographic signature on a message; in this case the principal is the cryptographic key that verifies the signature. Another example is accepting input from the keyboard, which is the principal directly making the request.[2]
2. (optionally) Associate one or more other principals with the principal of step 1. These could be groups or attributes.
3. Do authorization, which determines whether any of these principals is allowed to have the request fulfilled on that object.

The boundary between authentication and authorization, however, is not clear. Different experts draw it in different places. It is also not particularly relevant, since it makes little sense to do one without the other.

3 Examples: Logon and cross-organization access control

This section gives two examples to introduce the basic ideas of access control.

3.1 Example: User and network logon

Figure 3 shows the basic elements of authentication and how they are used to log on a user, access a resource, and then do a network logon to another host. Note the distinction between the elements that are part of a single host and external token sources such as domain controllers and STS's. For concreteness, the figure describes the process of authenticating a user as logon to Windows, that is, as creating a Windows session that can speak for the user; in Windows a SID is a 128-bit binary identifier for a principal. However, exactly the same mechanisms can be used to log onto an application such as SQL Server, or to authenticate a single message, so it covers these cases equally well.

See the appendix for a sketch of what you need to know about cryptography.

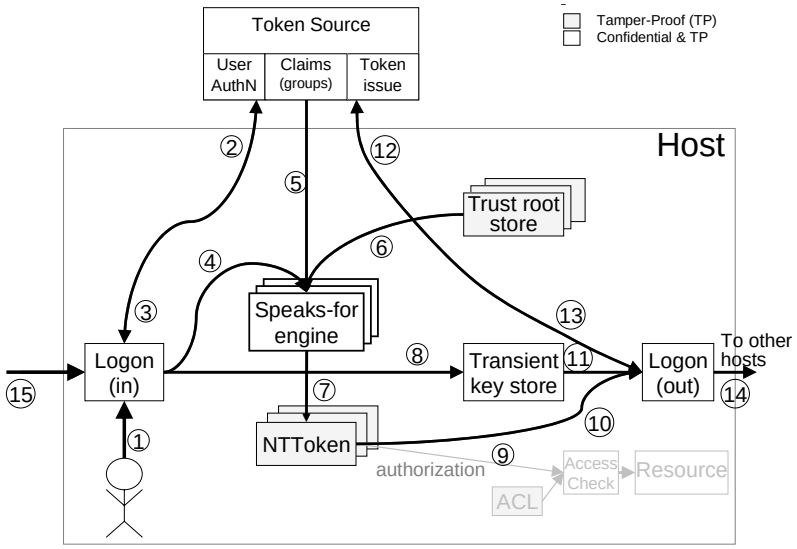

Figure 3: Core logon example

The numbers in the figure label the steps of the logon, which are as follows:

1. The user provides some input for logon (for example, user name and password).
2. The logon agent sends a logon validation request with the input (or something derived from it) to the domain controller (labeled "token source" in the figure),
3. which replies with the user's SID and a session key if logon succeeded, and an error if it didn't.
4. The token source provides the user's SID,
5. and uses it to provide the group SIDs.
6. The trust root says that the token source should be trusted to logon anyone, so
7. all the SIDs go into the NT token,
8. and the session key is saved in the transient key store.
9. When the process accesses some local resource the NT token is checked against the ACL, and with luck the access is granted.
10. When the process wants to access a remote resource, the NT token
11. and the session key are needed
12. to ask the token source to
13. issue a token that can be sent out
14. to the remote host,
15. which receives it (back on the left side of the figure) and does a net logon.

3.2 Example: Cross-organization 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 local systems that are well supported by Windows domains, as in the previous example) we need a way to treat uniformly all the information that contributes to the decision to grant or deny access. Consider the following example, illustrated in Figure 4:

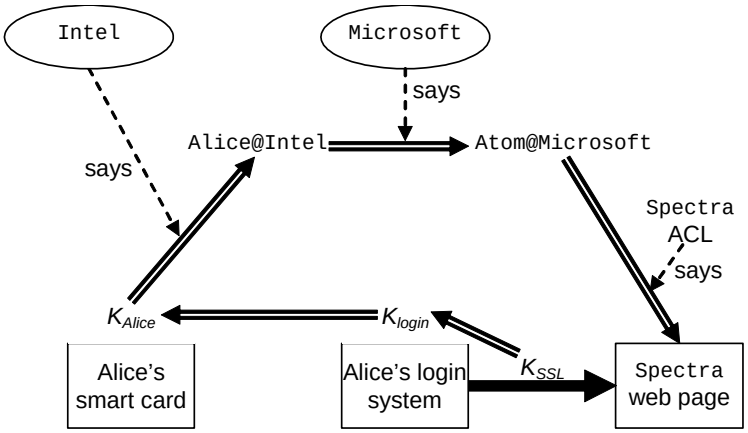

Figure 4: Speaks-for example

Alice at Intel is part of a team working on a joint Intel-Microsoft project called `Atom`. She logs in to her Intel workstation, 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 connection key K_{SSL} created using the Diffie-Hellman key exchange protocol.
2) To authenticate the SSL connection, Alice's workstation uses its temporary logon key K_{logon} to sign a statement certifying that requests secured by the connection key K_{SSL} come from the logon session.[3]
3) At logon time, Alice's smart card uses her key K_{Alice} certifies that requests signed by the logon session key K_{logon} come from Alice.
4) Intel certifies that K_{Alice} is the key for `Alice@Intel.com`.[4]
5) Microsoft's group database says that `Alice@Intel.com` is in the `Atom` group.
6) The ACL on the `Spectra` page says that `Atom` has read/write access.

In the figure, Alice's requests to `Spectra` travel over the SSL channel (represented by the fat arrow), which is secured by the key K_{SSL}. In contrast, the reasoning about trust that allows `Spectra` to conclude that it should grant the requests runs clockwise around the circle of double arrows; note that requests never travel on this path.

From this example we can see that 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

[3] Saying that the workstation signs with the public key K_{logon} 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. This is not the only way to authenticate an SSL connection, but it is the simplest to explain.

[4] Intel can do this with an X.509 certificate, or by responding to a query "Is K_{Alice} the key for `Alice@Intel.com`?", or in some other secure way.

- 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 logon 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.

4 Basic concepts

This section describes the basic concepts, informally but in considerable detail: principals and identifiers; speaks-for and trust; tokens; paths, security domains, attributes, and groups; global identifiers; how to choose identifiers and names, and freshness or consistency. Sections 5 and 6 describe the components of the architecture and how they use these concepts.

4.1 Principals and identifiers

A principal is the source of a request in the model of ; it is the answer to the questions:
- "Who made this request?" (authentication)
- "Who is trusted for this request?" (authorization—for example, who is on the ACL)

We say that the principal **says** the request, as in *P* **says do** `read report.doc`. In addition to saying requests, principals can also say speaks-for statements or claims, as explained in section 4.2.

Principals are not only people and devices. Executable code is a principal. An input/output channel and a cryptographic signing key are principals. So are groups such as `Microsoft-FTE` and attributes such as `age=32`. We treat all these uniformly because they can all be answers to the question "Who is trusted for this request?". Furthermore, if we interpret the question "Who made this request?" broadly, they can all be answers to this question as well: a request can be made directly only by a channel or key, but it can be made indirectly by a person (or device) that controls the key, or by a group that such a person is a member of.

It turns out to be convenient to treat objects or resources as principals too, even though they don't make requests.

Principals can be either simple or compound. Simple principals are denoted by *identifiers*, which are strings. Intuitively, identifiers are labels used for people, computers and other devices, applications, attributes, channels, resources, etc., or groups of these.[5] Compound principals are explained in section 5.8.

[5] Programs usually can deal only with identifiers, not with the real-world principals that they denote. In this paper we will ignore this distinction for the most part.

Channels are special because they are the only *direct* principals: a computer can tell directly that a request comes from a channel, without any other information. Thus any authentication of a request must start with a channel. A cryptographic signing key is the most important kind of channel.

An identifier is a string; often the string encodes a path, as explained below. The string can be meaningful (to humans), or it can be meaningless; for example, it can encode a binary number (Occasionally an identifier is something that is meaningful, but not as a string of characters, such as a picture.). This distinction is important because access control policy must be expressed in terms of meaningful identifiers so that people can understand it, and also because people care about the meanings of a meaningful identifier such as `coke.com`, but no one cares about the bit pattern of a binary identifier. Of course there are gray areas in this taxonomy; a name such as `davcdata.exe` is not meaningful to most people, and a phone number might be very meaningful. But the taxonomy is useful none the less.

Meaningless identifiers in turn can be direct or not. This leads to a three-way classification of identifiers:

- *name*: an identifier that is meaningful to humans.
- *ID*: a meaningless identifier that is not direct. In this taxonomy an identifier such as `xpz5914@hotmail.com` is probably an ID, not a name, since it probably isn't meaningful.
- *direct*: a meaningless identifier that identifies a channel. There are three kinds of direct identifiers:
    - *key*: a cryptographic key (most simply, a public key) that can verify a signature on a request. We view a signing key as a channel, and say that messages signed by the key arrive on the channel named by that key.[6]
    - *hash*: a cryptographic collision-free hash of data (code, other files, keys, etc.): different data is guaranteed to have different hashes. A hash *H* can say X if a suitable encoding of "This data **says** X" appears in the data of which *H* is the hash. For code we usually hash a *manifest* that includes the hash of each member file. This has the same collision-free property as a hash of the contents of all the files.
    - *handle*: an identifier provided by the host for some channel, such as the keyboard (Strictly speaking, the wire from the keyboard.) or a pipe.

An identifier can be a *path*, which is a sequence of strings, just like a path name for a file such as `C:\program files\Adobe\Acrobat6`. It can be encoded as a single string using some syntactic convention. There are a number of different syntactic conventions for representing a path as a single string; the file name example uses `"\"` as a separator. The canonical form is left-to-right with / as the separator. A path can be rooted in a key, such

[6] For a symmetric key we can use a hash of it as the public name of the channel, though of course this is not enough to verify a signature.

as $K_{Verisign}$/andy@intel.com (or $K_{Verisign}$/com/intel/andy in the canonical form for paths); such a path is called *fully qualified*. A path not rooted in a key is rooted in self, the local environment interpreting the identifier; it is like a relative file name because its meaning depends on the context.

4.2 Speaks-for and trust

Authentication must start with a channel, for example, with a cryptographic signature key. But it must end up with access control policy, which has to be expressed in terms of names so that people can understand it. To bridge the gap between channels and names we uses the notion of "speaks-for". We say that a channel speaks for a user, for example, if we trust that every request that arrives on the channel comes from the user, in other words, if the channel is trusted to *speak for* the user.

But the notion of speaks-for is much more general than this, as the example of section 3 illustrates. What is the common element in all the steps of the example and all the different kinds of information? 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 *R*"

For example, K_{SSL} speaks for K_{Alice} about everything, and Atom@Microsoft speaks for Spectra about read and write. We write "about *R*" as shorthand for "about statements in set *R*". Often *P* is called the *subject* and *R* is called the *rights*.

The idea of "*P* speaks for *Q* about *R*" is that

if *P* says something about *R*, then *Q* says it too

That is, *P* is trusted as much as *Q*, at least for statements in *R*. Put another way, *Q* takes responsibility for anything that *P* says about *R*. A third way: *P* is a more powerful principal than *Q* (at least with respect to *R*) since *P*'s statements are taken at least as seriously as *Q*'s (and perhaps more seriously). Thus *P* has all of *Q*'s authority about *R*.

The notion of principal is very general, encompassing any entity that we can imagine making statements or being trusted. Secure channels, people, groups, attributes, systems, program images, and resource objects are all principals. The notion of speaks-for is also very general; some examples are:

Binding a key to a user name.
Binding a program hash to a name for the program.
Allowing an authority to certify a set of names.
Making a user a member of a group.
Assigning a principal an attribute.
Granting a principal access to a resource by putting it on the resource's ACL.

The idea of "about *R*" is that *R* is some way of describing a set of things that *P* (and therefore *Q*) might say. You can think of *R* as a pattern or predicate that characterizes this set of statements, or you can think of it as some rights that *P* can exercise as much as *Q* can. In the example of section 3, *R* 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 *R*, so the interpretation of *R*'s encoding can be local to the object. For example, we could refine "read and write requests" to "read and write re-

quests for files whose names match `/users/lampson/security/*.doc`". In most ACEs today, *R* is encoded as a bit vector of permissions, and you can't say anything as complicated as the previous sentence.

We can write this $P \Rightarrow_R Q$ for short, or just $P \Rightarrow Q$ without any subscript if *R* is "all statements". With this notation the chain for the example is:

$K_{SSL} \Rightarrow K_{logon} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`

A single speaks-for fact such as $K_{Alice} \Rightarrow$ `Alice@Intel` is called a *claim*. The principal on the left is the *subject.*

The way to think about it is that $\Rightarrow$ is "greater than or equal": the more powerful principal goes on the left, and the less powerful one on the right. So `role=architect` $\Rightarrow$ `Slava` means that everyone in the architect role has all the power that Slava has. This is unlikely to be what you want. The other way, `Slava` $\Rightarrow$ `role=architect`, means that Slava has all the power that the architect role has. This is a reasonable way to state the implications for security of making Slava an architect.

Figure 4 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. The example shows that claims can abstract from a wide variety of real-world facts:

- A key can speak for a person ($K_{Alice} \Rightarrow$ `Alice@Intel`) or for a naming authority ($K_{Intel} \Rightarrow$ `Intel.com`).
- A person can speak for a group (`Alice@Intel` $\Rightarrow$ `Atom@Microsoft`).
- A person or group can speak for a resource, usually by being on the ACL of the resource (`Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`). We say that `Spectra` makes this claim by putting `Atom` on its ACL.

4.2.1 Establishing claims: Delegation

How does a claim get established? It can be built in; such facts appear in the trust root, discussed in section 5.1. Or it can be derived from other claims, or from statements made by principals, according to a few simple rules:

(S1) Speaks-for is transitive: if $P \Rightarrow Q$ and $Q \Rightarrow R$ then $P \Rightarrow R$.

(S2) A principal speaks for any path rooted in itself: $P \Rightarrow P/N$. This is just like a file system, where a directory controls its contents. Section 4.10 discusses paths.

(S3) Principals are trusted to *delegate* their authority, privileges, rights, etc.: if Q **says** $P \Rightarrow Q$ then $P \Rightarrow Q$. (There are restricted forms of speaks-for where this rule doesn't hold.)

From the definition of $\Rightarrow$, if Q' **says** $P \Rightarrow Q$ and $Q' \Rightarrow Q$ then Q **says** $P \Rightarrow Q$, and it follows from (S3) that $P \Rightarrow Q$. So a principal is trusted to delegate the authority of any principal it speaks for, not just its own authority. Frequently a delegation is restricted so that the delegate *P* speaks for *Q* only for requests (this is the usual interpretation of an X.509

end-entity certificate, for example, or membership in a group) or only for further delegation (an X.509 CA certificate, or `GROUP_ADD/REMOVE_MEMBER` permission on the ACL for a group).

4.2.2 Validity period

A claim usually has a validity period, which is an interval of real time during which it is valid. When applying the rules to derive a claim from other claims and tokens, intersect their validity periods to get the validity period of the derived claim. This ensures that the derived claim is only valid when all of the inputs to its derivation are valid. A claim can be the result of a query to some authority *A*. For example, if the result of a query "Is *P* in group *G*" to a database of group memberships is "Yes", that is an encoding of the claim $P \Rightarrow G$. The validity period of such a statement is often just the instant at which the response is made, although the queryer might choose to cache it and believe it for a longer time.

4.3 Tokens

A claim made by a principal is called a *token* (not to be confused with a user authentication token such as a SecurID device). Many tokens are called certificates, but this paper uses the more general term except when discussing X.509 certificates specifically. The rule (S3) tells you whether or not to believe a token; section 4.5 on global identifiers gives the most important example of this.

Examples of tokens:

1. X.509 certificate [K_I **says** $K_S \Rightarrow$ name, (optionally K_I **says** name $\Rightarrow$ attribute)]
2. Authenticode certificate [K_V **says** H(code) $\Rightarrow$ publisher/program]
3. Group memberships [K_D **says** $S_U \Rightarrow S_G$]
4. Signed SAML attribute assertion [K_I **says** name $\Rightarrow$ attribute]
5. ISO REL (XrML) license

where K_I is the issuer key, K_S is the subject key, "name" is the certified name, K_V is Verisign's key, H(code) is the hash value of the code being signed, "publisher" is the name of the code's publisher, K_D is the key of the domain controller, S_U is the SID of the user and S_G is the SID of the group of which the user is a member. XrML tokens can do all of these things, and more besides.

A token can be signed in several different ways, which don't change the meaning of a token to its intended recipient, but do affect how difficult it is to forward:

- A token signed by a public key, like a X.509 certificate, can be forwarded to anyone without the cooperation of the third party. From a security point of view it is like a broadcast.
- A token signed by a symmetric key, like a Kerberos ticket, can be returned to its sender for forwarding to anyone with whom the sender shares a symmetric key.
- A token that is just sent on an authenticated channel cannot be forwarded, since there's no way to prove to anyone that the sender said it.

In a token the principals on both sides of the ⇒ must be represented by identifiers, and it's important for these identifiers to be unambiguous. A fully qualified identifier (one that starts with a key or hash) is unambiguous. Other identifiers depend on the context, that is, on some convention between the issuer and the consumer of the token.

Like a claim, a token usually has a validity period; see section 4.2.2. For example, a Kerberos token is typically valid for eight hours.

A token is the most common way for a principal to communicate a claim to others, but it is not the only way. You can ask a principal *A* "Do you say $P \Rightarrow Q$?" or "What principal does *P* speak for?" and get back "*A* **says** 'yes'" or "*A* **says** '*Q*'". Such a statement only makes sense as a response to the original query; to be secure it must not only be signed by (some principal that speaks for) *A*, but also be bound securely to the query (for example, by a secure RPC protocol), so that an adversary can't later supply it as the response to some other query.

4.4 Organizing principals

There are several common ways to impose structure on principals in addition to the path identifiers introduced in section 4.1: security domains, attributes, and groups.

4.4.1 Security domains

A security domain is a collection of principals (users, groups, computers, servers and other resources) to which a particular set of policies apply, or in other words, that have common management. Usually we will just say "domain". It normally comprises:

- A key K_D.
- A namespace based on that key.
- A trust root—a set of claims of the form $K_{j1} \wedge K_{j2} \ldots \Rightarrow$ identifier-pattern
- ACLs for the trust root and the accounts, which define the administrators of the domain.
- A set of *accounts*—statements of the form K_D **says** $K_i \Rightarrow K_D/N$ for principals with names in its namespace.
- A set of resources and policies for those resources

The essential property of paths is that namespaces with different roots are independent, just as different file system volumes are independent. In fact, namespaces with different *prefixes* are independent, just as file system directories with different names are independent. This means that anybody with a public key *K* can create a namespace rooted in that key. Such a namespace is the most important part of a security domain. Because of (S2), *K* speaks for the domain. Because of (S3), if you know K^{-1} you can delegate authority over any part of the domain, and since *K* is public, anyone can verify these delegations. This means that authentication can happen independent of association with any domain controller. Of course, you can also rely on a third party such as a domain controller to do it for you, and this is necessary if *K* is a symmetric key.

For example, an application such as SQL Server can create its own domain of objects, IDs, names and authorities that has no elements in common with the Windows domain of

objects, IDs, names and authorities for the machine on which SQL Server is running. However, the SQL Server can use part or all the Windows security domain if that is desired. That use is controlled by policy, in the form of trust root contents and issued tokens.

Here are some other examples of operating in multiple security domains:

1. A user takes a work laptop home and connects to the home network, which has no connection to the work security domain.

2. A consultant has a laptop that is used in working with two competing companies. For each company, the consultant has a virtual machine with its own virtual disk. Each of those virtual machines joins the Windows domain of its respective company. The host OS, however, is managed by the consultant and has its own local domain.

Sometimes we distinguish between resource domains and account domains, depending on whether the domain mostly contains resources or objects, or mostly contains users or subjects.

Domains can be nested. A child domain has its own management, but can also be managed by its parent.

4.4.2 Attributes

An attribute such as `age=32` is a special kind of path, and thus is a principal like any other. This one has two components, the *name* `age` and the *value* `32`; they are separated by "=" rather than "/" to emphasize the idea that `32` is a value for the attribute name `age`, but this is purely syntactic.[7] The claim `Paul` ⇒ `age=32` expresses the fact that Paul has the attribute `age=32`. Like any path, an attribute should be global if it is to be passed between machines: K_{oasis}/`age=32`. However, unlike file names or people, we expect that most attributes with the same name in many different namespaces will have the same intended meaning in all of them. A claim can translate the attribute from one namespace to another. For example, `WA/dmv/age` ⇒ `NY/rmv/age` means that New York trusts `WA/dmv` for the `age` attribute. Translation can involve intermediaries: `WA/dmv/age` ⇒ `US/age` and `US/age` ⇒ `NY/rmv/age` means that New York trusts US for `age`, and US in turn trusts Washington (presumably US trusts lots of other states as well, but these claims don't say anything about that). Locally, of course, it's fine to use `age=32`; it's a local name, and if you want to translate `US/age=32` to `age=32` you need a trust root entry `US/age` ⇒ `age`. In fact, from the point of view of trust `age=32` is just like a nickname. The difference is that we expect lots of translations, because we expect lots of principals to agree about the meaning of `age`, whereas we don't expect wide agreement about the meaning of Bob.

[7] Sometimes people call `age=32` an "attribute-value" or an "attribute-value pair", and call `age` an "attribute". This is perfectly good English; it might even be better English than calling `age=32` an attribute. But it is confusing to have both meanings for "attribute" floating around. In this paper, "attribute" means the pair `age=32`, and `age` is the attribute name. Sometimes we say "the `age` attribute", meaning an attribute whose name is `age`.

Because of the broad scope of many attribute names such as `age`, the name of an attribute can change as it is expressed in different languages and even different scripts. Therefore it is often necessary to use an ID rather than a name for the attribute in policy. For example, an X.509 object identifier or OID is such an ID. Sections 4.5 and 4.6 discuss the implications of this; what they say applies to attributes as well.

A Boolean-valued attribute (one with a value that is true or false), such as `over21`, defines a *group*; we normally write it that way rather than as `over21=true`. The next section discusses groups.

4.4.3 Groups and conditions

A condition is a Boolean expression over attribute names and values, such as "`microsoft.com/division == 'sales' & microsoft.com/region == 'NW'`". A condition is a principal; every principal that speaks for attributes whose values cause the expression to evaluate "`true`" speaks for the condition. In the preceding example, every Microsoft employee in the northwest sales region would speak for it.

For use in conditions, identifiers are considered to be Boolean-valued attributes that evaluate true for the principals that speak for them. Hence the condition `paul@microsoft.com | carl@microsoft.com` is true for `paul@microsoft.com` and `carl@microsoft.com`. It is also true for the key `K` if `K => paul@microsoft.com`.

In addition, there are special attributes, such as `time`, that may be used in conditions; every principal is considered to speak for them. For example, "`time >= 0900 & time <= 1500 & shift == 'day' & jobtitle == 'operator'`" would be true for all day-shift operators between 9am and 5pm.

If *C* is a condition, and a principal *P* has attributes whose values cause *C* to evaluate true, then we write:

$$P \Rightarrow C$$

We can give a condition an identifier (a name or an ID) by saying that the condition speaks for the identifier:

$$C \Rightarrow \textit{identifier}$$

We call such an identifier a *group*.[8] A group is thus a principal with zero or more other principals that speak for it. If a principal speaks for the group, we say that it is a *member* of the group. Today's groups are defined by a condition that is just the "or" of a list of members. In such a case, it's possible to provide a complete list of all the group members, but this is not always true. The distinction is important for a principal with the authority to define members, but it is invisible to access control, which only cares about a requestor *P* presenting a claim $P \Rightarrow G$ and $G \Rightarrow \textit{resource}$ being on the ACL.

Such an authority will only issue such a claim if it:

[8] This is not the only meaning of 'group' in English, in computing, or in security, but it is the usual meaning and the one we adopt.

- Has access to a complete list of the group members (such as Paul, Carl, Charlie), and *P* is in it, or
- Has access to a partial list of the group members and *P* is on its partial list; there may be several such lists, each accessible to a different issuer, or
- Knows that *P* satisfies the condition that defines the group (such as `age>=21`).

The question of who is trusted to assert $P \Rightarrow G$, that is, who can define the members of a group, is part of authorization.

4.5 Global identifiers

To avoid confusion, identifiers communicated between computer systems should be global. If a set of systems doesn't communicate with the rest of the world, they only need to agree among each other. However, when these systems suddenly do need to share identifiers (perhaps because they merge with another set of systems), collisions of identifiers can occur, requiring a massive renaming of entities. To avoid such problems, all identifiers that might travel between computers should be global, except perhaps names intended to communicate to a human being.

An identifier is global if everyone agrees on its meaning, that is, when presented with a request and some supporting evidence, everyone either agrees on whether the identifier is the principal that made the request or doesn't know. A key or hash is automatically global; cryptography makes it so. Other identifiers are paths (perhaps of length one).

A path rooted in a key, such as K_{intel}`/andy@intel.com`, is called *fully qualified*. Such identifiers are global, because K_{intel} is global, and according to rule (S2) above it can say what other keys can speak for identifiers rooted in itself. For example, K_{intel} can establish that Andy's key K_{andy} speaks for the name K_{intel}`/andy@intel.com`, by signing a certificate (token):
(C1) K_{intel} **says** $K_{andy} \Rightarrow K_{intel}$`/andy@intel.com`
Paths not rooted in keys are rooted in `self`, the local environment interpreting the identifier. They are not global and therefore should not be sent outside the local environment.

We would like to treat an identifier like `andy@intel.com` (or /`com`/`intel`/`andy` in the canonical form) as global, even though it is not rooted in a key, because we want to keep keys out of most policy. This is a *conventionally* global identifier: we make it very likely that almost everybody agrees about what speaks for it, by making it very likely that everyone agrees that $K_{andy} \Rightarrow$ `andy@intel.com`. We do that by getting the same agreement that K_{intel} `=> intel.com`; then everyone will accept K_{intel}'s certificate (C1). Of course this is the same problem, and we can solve it in the same way: agree that $K_{verisign} \Rightarrow$ `com`, and get a certificate
(C2) $K_{verisign}$ **says** $K_{intel} \Rightarrow$ `intel.com`

This recursion has to stop somewhere, and it stops in a special part of the security policy called the *trust root*, where some of these facts are built in. The essential idea is:

Provided their trust roots agree and they have the same tokens, two parties will agree on what keys speak for a conventionally global identifier.

One case in which the parties might disagree is while a key is being rolled over or replaced, but only if they have different tokens—one has heard about the key change and the other one hasn't.

Section 5.1 discusses the trust root in detail, and section 5.1.1 explains how to make it likely that two trust roots agree.

Although any kind of path could be a conventionally global identifier, the ones that people cares most about are DNS names (see section 4.7). Email names are important too, but they usually don't require special attention because there's a single DNS name that authenticates a given email name.

4.6 Choosing identifiers for access policy

There are three conflicting requirements on identifiers:

- *Meaningful* (to humans): When security policy such as group definitions, access control lists, etc is displayed to humans, identifiers must be meaningful, since people must be able to understand the policy. Only names are meaningful. Another consequence is that only names are controversial: no one cares what bit pattern your public key has, or what domain ID your SID uses, but people do care who controls `microsoft.com` or `mit.edu`.

- *Long-lived*: The identifier doesn't need to change when encryption keys or names change. This is desirable, because much security policy is long-lived: the identifier may appear on ACLs for objects that last for decades, and that are scattered over the internet or written on DVDs. Neither names nor direct identifiers can be guaranteed to be long-lived, since people get married, join a new organization, or otherwise change their minds about names, and keys can be compromised and need to change.

- *Direct*: some identifiers must be direct, since only direct identifiers can actually make requests. Direct identifiers are neither meaningful nor long-lived.[9]

The following table summarizes the choices:

Property	Meaningful	Long-lived	Direct
Identifier type			
Name	**yes**	no	no
ID	no	**possibly**	no
Direct (keys etc.)	no	no	**yes**

We can distinguish three main places where an identifier may appear:

- As the direct source of a request, where it must be direct, since all the machine directly knows about the source of a request is the channel it arrives on.

[9] The hash of some data is long-lived in the sense that it won't change. However, the hashes that are important for access control are hashes of code, and the hash of code that you care about changes frequently, because of patches and new versions. So in practice a hash has a much shorter lifetime than many keys.

- In the user's view of access control policy, where it must be meaningful, in other words, a name.
- In access control policy stored in the system, where it's desirable for it to be long-lived, but it could have none of these properties as long as there is extra machinery to make up the lack.

As peer-to-peer operation grows—both personal P2P and corporate P2P—identifiers for principals will show up in access control policy far and wide. An identifier might be on ACLs on machines and DVDs all over the world, with no record of where those machines are. It might also be in tokens such as XrML licenses, SAML or XACML tokens, certificates in various forms, etc., which are another way to express access control policy. These signed statements can be carried anywhere, can be backed up, can be transferred from one machine to another. Again, there is no requirement that each such statement have its location registered in any central place. Hence it's often desirable for the identifiers in access control policy to be long-lived.

Since no identifiers satisfy all the requirements, there have to be ways of mapping among them:

- When a request or a token comes in, it can only be authenticated as coming from a direct principal, that is, a channel C, so there must be a mapping $C \Rightarrow P$ to a stored principal.
- When a user wants to examine or edit policy they need to see a meaningful principal M, so there must be mappings in both directions $M \Rightarrow P$ and $P \Rightarrow M$.

Any kind of identifier can appear in stored access control policy. As we have seen, however, it's often important for stored identifiers to be long lived, so that the policy doesn't have to change when the identifiers change. It's therefore advantageous to use a particular kind of ID called a SID for stored policy, because SIDs are carefully constructed to be long-lived; see section 4.7. There has to be a reliable correspondence between SIDs and names so that policy can be read and written by people, but this correspondence can change with time. There also has to be a reliable SID↔key correspondence so that requests can get access.[10]

The preferred approach to keys is complementary to this one: the only long-term place to store keys should be the trust root (see section 5.1), which contains facts about principals that are installed manually and accepted on faith in reasoning about authentication.

[10] Preferring names would also work, and it would be simpler since there would be no need for the SID↔name correspondence, but it leads to inconvenience when a name changes, and to insecurity when a name is reused.

Preferring keys seems appealing at first, since although it needs a key↔name correspondence, it doesn't need anything else. Unfortunately, it's insecure when a key is compromised, unless the key in policy is no longer treated as a direct identifier but rather as something that can be mapped reliably to a key that is currently valid. Doing this makes it harder to handle than a SID. Since you can't tell by looking at it whether a key has been compromised, you have to do this work every time.

4.6.1 Anonymity

Sometimes people want to avoid using the same identifier for all their interactions with the world, because they want to preserve their anonymity. A variation on this is that they don't want their actions at one web site, for example, to be correlated with their actions at another site; this kind of correlation is called tracking.

Since there is no shortage of encryption keys or identifiers, it's easy for a computer to generate as many identifiers for me as I want, for example, a different one for every web site I interact with. The computer can keep track of which identifier to use at which site. If you are really paranoid, you can use a different identifier each time you go to the *same* site.

In many case, this by itself is sufficient. Sometimes, however, a web site or other party may want to know something about me: that I am over 18, or have a decent credit rating, or whatever. For this purpose a mutually trusted third party such as Live or *Consumer Reports* can authenticate one of my identifiers, certifying, for example, $K_{bwl\text{-}amazon} \Rightarrow$ `over18`. The protocol for this is simple: I authenticate to Live, I give $K_{bwl\text{-}amazon}$ to Live and ask for a certificate, and I get back K_{live} **says** $K_{bwl\text{-}amazon} \Rightarrow$ `over18`.

4.7 SIDs

SIDs contain a 96 bit domain identifier plus a 32 bit relative identifier within the domain. Thus the structure is D/R. To distinguish SIDs from other identifiers we prefix `SID`, so the full identifier is `SID`/D/R, but we will usually omit the `SID`/ prefix here. Roughly speaking, D corresponds to something like `microsoft.com`, and R to `blampson` or the server `red-msg-70`, so D/R corresponds to `blampson@microsoft.com` or `red-msg-70.microsoft.com`.

These SIDs have the following useful properties:

1. They are not meaningful to humans, unlike names. No one will care which numbers are assigned to which domains or which principals.
2. They are not direct identifiers, as keys are, so that policy expressed in terms of SIDs remains the same when keys change. Only the SID↔key correspondence needs to change.
3. There are plenty of them, so they don't have to be rationed (except to prevent denial of service attacks on ID services that map SIDs to keys).
4. They are (two part) paths D/R, so that a key that speaks for a domain D can speak for lots of SIDs in that domain.

Because of (1) and (2) a SID is a long-lived identifier that is suitable for long-lived policy such as ACLs.

Since there are plenty of domain identifiers, you can get a new one just by choosing a 96 bit random number; this is reasonable because one D is as good as another. The chance of an accidental collision is very small (once every 8,000 years if there are a thousand new domains per second); we consider collisions caused by malice shortly. Some domains will have only a few SIDs (that is, a few values of *R* for one *D*), for example, a domain

for a person, family, or small organization. But most SIDs will probably be in large domains belonging to corporations or to Internet services such as Live or Yahoo.

As we saw in the previous section, we need to know $K \Rightarrow D/R$ so that we can authenticate a statement signed by K as coming from D/R. We also need to know $name \Rightarrow D/R$ and $D/R \Rightarrow name$ so that users can read and change policy that is stored in terms of SIDs. These mappings could be strictly local if the local administrator takes responsibility for setting up and maintaining them, but in general it will come from someone who speaks for D/R (for example, someone who speaks for D) or for *name* (for example, `microsoft.com` if *name* is `billg@microsoft.com`).

Note that joining a Windows domain is quite different from learning $K_D \Rightarrow D$. A machine can only be joined to one domain, and a domain joined machine trusts its domain controller for *any* SID, and also for various management functions. A machine or session can know about lots of domains, and it trusts each one *only* for its own SIDs.

4.7.1 Domain ID service

To simplify the handling of domain key changes and malicious (as opposed to accidental) conflicts for domain identifiers, it's desirable to have one or more domain ID services, which are intended to issue tokens K_{DR} **says** $K_D \Rightarrow D$. Then instead of having a trust root entry for each D that you encounter, you only need one that says $K_{DR} \Rightarrow$ `SID`/* for each ID service that you want to trust. For greater security, you could configure your trust root with n domain ID services and a requirement that k of them agree on $K_D \Rightarrow D$ before it is believed; see section 5.1.2 for more on this. As with other kinds of trust root entries, an entry $K_D \Rightarrow D$ for a specific domain takes precedence, or disagreement is referred to the administrator; see section 5.1. For this to work well, there should not be too many ID services and the scope of each one should be wide.

The domain ID service can work as a simple web service with no human operator involvement only because what it records has no intrinsic value. The ID service is designed specifically and only to meet the needs of authentication. It offers only one public query: "Is $K_D \Rightarrow D$ a registered claim?"[11] It is intentionally not a general purpose directory. It is intentionally limited never to become a general purpose directory. Nothing stops people from making more general directories, but those are not domain ID services.

In addition to the query, there is one operation for registering new values of D. The input parameters are D, a public key K_D, and an optional password PW encrypted by K_{DR} that can be used for resetting K_D. The request is signed by K_D^{-1}. There is no other authentication. In particular, there is no linkage to any PII or to any other information that would require human operators at the domain ID service. After success, $K_D \Rightarrow D$ is a registered claim.

Windows domains today implement a highly simplified version of this scheme, since a domain joined machine trusts its domain controller for any SID.

[11] Or perhaps "What are the keys that speak for D?"

4.8 Names

The purpose of a name is to be meaningful to a human. Most useful names are paths, and the preferred (conventionally) global names are DNS and email names such as `research.microsoft.com` or `billg@microsoft.com`. As we did with SIDs, to distinguish DNS names from other identifiers we prefix `DNS`, so the full identifier is `DNS/com/microsoft/research`, but we will usually omit the `DNS/` prefix here and use the standard DNS syntax.

The crucial security questions about a name are what real world entity it identifies, and what key or SID speaks for it. To answer the second question, you consult the trust root, together with any tokens that are relevant. Thus the trust root might contain

$K_{Verisign}$ ⇒ `DNS/*`; K_{billg} ⇒ `billg@microsoft.com`

Here the second name is written in its conventional email form; as a canonical path name it would be `DNS/com/microsoft/email/billg`. The rule for trust roots (see section 5.1) is that the more specific entry governs, so that what Verisign or Microsoft have to say about `billg@microsoft.com` will be ignored.

Today's X.509 trust roots usually grant a certificate authority such as Verisign authority over all DNS names; that is what the $K_{Verisign}$ ⇒ `DNS/*` claim in the example says. Although there are ways to limit the names that such a key can speak for, today they are obscure. Such limits are of fundamental importance, and need to be easy to set and understand.

Adding an entry for a name to the trust root must be a human decision, so the procedure by which the human decides that it's the right thing to do, called a *ceremony*, must be carefully designed. A ceremony is like a network protocol but includes human components as well as computers. for more on this topic.

4.9 Freshness

Secure communication requires more than assurance that a message came from a known source; it also requires *freshness*, a guarantee that the message is sufficiently recent. Without freshness, a bad guy can make trouble by replaying old messages, which might well be misinterpreted in the current context. For example, consider a request to a service to write a check for $10,000. Replaying this request should not result in a second check. Or consider a request that asks "Does key *K* speak for `microsoft.com`?" and expects a yes or no answer. If a previous request that asked "Does key $K_{microsoft}$ speak for `microsoft.com`?" got a "yes" answer, it should not be possible to replay this answer and get the requester to accept it as the answer to the later request.

There are many ways to ensure freshness. In a request-response protocol like the second example above, you tag the request with a sequence number and demand the same sequence number in the response. Such a tag is called a *nonce* or *challenge*. To ensure that an incoming message is fresh, in particular that it was generated since you chose a nonce, you insist that it contain some evidence that the sender received that nonce.

The essential property of a nonce is that it is not reused; nonces may be ordered, but this is usually unimportant. If you want to prevent the responder from precomputing the response, a nonce must be unpredictable; frequently this is not a requirement. Often there are two layers of freshness. For example, a sequence of requests might be carried on a channel that is secured with a fresh key. Then the nonces need only be unique within that sequence, since a different sequence of requests will be secured with a different key. In this example the sequence numbers on the messages don't need to be unpredictable.

To ensure that a key is fresh, generate it by hashing some data that includes a newly generated random number. For two party two-way communication, each party should generate its own random number to be included in the hashed data; this gives each party assurance of freshness, and also ensures a good key even if one of the parties is not good at generating random numbers.

For broadcast communication such as a certificate signed with a public key, nonces don't work because the receivers don't send anything to the broadcaster beforehand. Instead, we usually rely on a timestamp in the certificate for freshness. The validity period in a token is an example of such a timestamp. You might also want to use a timestamp to avoid a round trip, for instance when sending email. It's not as conclusive as a nonce because of clock skew (and perhaps because it's predictable).

4.9.1 Consistency vs. availability

Availability and consistency: choose one

There is a fundamental tradeoff between consistency (or freshness) and availability. *A* is consistent with *B* if *A*'s view of *B*'s state agrees with *B*'s actual state.[12] The only way to ensure this is for *A* to hold a lock on *B*'s state, but this means that *A* has to communicate with *B* to acquire the lock, and after that *B* can't change its state until *A* releases the lock. This is usually unacceptable in a distributed system because it hurts availability too much: if *A* and *B* can't communicate, one of them is going to be stuck.[13]

The alternative is for *A* to settle for a view of *B*'s state at some time in the past; often this is cached information. Now there is a tradeoff among freshness (how far in the past?), availability, and performance (how often does *A* check for changes in *B*'s state?). This tradeoff is fundamental; no cleverness in the implementation can avoid it. The choice is between acting on old (perhaps cached) information, and getting stuck when you can't communicate. This is a management decision and it must be exposed to management control. At least two parameters must be settable by the relying party (perhaps taking account of hints in the token):

1. How old data can be and still be acted on (the tradeoff between freshness and availability).

[12] More precisely, the view is some function v of *B*'s state s_B, and A knows $v(s_B^{past})$, where s_B^{past} is some past value of s_B. *A* is consistent with *B* if $v(s_B^{past}) = v(s_B)$.

[13] Sometimes a special kind of lock called a *lease* is acceptable; this is a lock that times out. A lease prevents its issuer from changing the state until either the leaseholder releases it, or the lease times out. People usually don't use leases for security information, but they could.

2. How frequently data should be refreshed (the tradeoff between freshness and performance).

The way to get the freshest information is for *A* to ask *B* for its state right now. This still doesn't guarantee perfect consistency, since *B*'s state can change between the time that *B* sends its reply and the time that *A* receives and acts on it, but it's the best you can do for consistency without a lock. The way to get the greatest availability and the least communication cost is for *A* to act on any view it has of *B*'s state, no matter how old.

This issue shows up most often for authentication in the validity period of a token. A short validity period means that the token is fresh, but also that new tokens must be issued and distributed frequently. A long validity period means that once you have the token you're good to go, but the token's issuer might have changed its mind about the claims in it. Note that there's nothing to stop a relying party from using a different validity period from the one in the token.

4.9.2 Revoking claims

If you have issued a token and you want to cancel it, is there any alternative to letting the validity period expire? Well, yes and no. Yes, because you may be able to revoke the token. No, because the revocation is just another kind of token, with a shorter validity period.

The idea behind revocation is that you need *two* tokens to justify a claim: the original token *Tk* that is "issuer **says** subject ⇒ ... as long as revoker confirms", and another *confirmation* token "revoker **says** *Tk* is still valid" that has a much shorter validity period than *Tk*. This is better than simply issuing *Tk* with a short validity period because the revoker is optimized for issuing confirmation tokens cheaply, quickly, and with high availability. It can't grant any access by itself, and it doesn't need any detailed information about the principals involved. Its database consists simply of tokens revoked by their issuers. When queried about *Tk*, it checks that database and issues a confirmation token if the database doesn't say that *Tk* is revoked.

To add an entry to the revoker's database, the original issuer writes a token "issuer **says** the token identified by *TkId* has been revoked" and sends it to the revoker. *TkId* could be a hash of the original token or a serial number embedded in the original token. The revoker puts (issuer, *TkId*) in its database. Since issuers can only revoke their own tokens, the revoker doesn't need to know anything about the issuers (unless it wants them to pay). The only harm the revoker can do is to revoke tokens without instructions, that is, mount a denial of service attack.

Because it is much simpler than most issuers and because it can't grant any access by itself, the revoker can afford to issue confirmation tokens with short validity periods, and it can be replicated for high availability. It's important to understand, however, that this is a difference of degree and not of kind. The tradeoffs described in section 4.9.1 still apply; only the parameters are different. For systems that are expected to be connected to the Internet, it's reasonable to use a validity period of a few minutes (or the length of a ses-

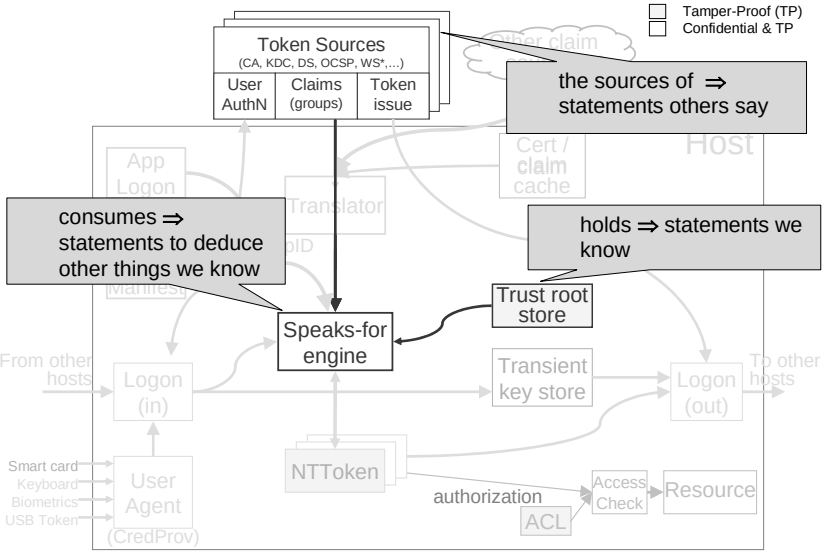

Figure 5: Core authentication components (see section 3.1 for a walk-through)

sion, if that is greater). Policy might say that if you can't contact a revoker, you should accept the token anyway.

There are several schemes for revocation. The original X.509 standard specifies a method called a Certificate Revocation List (CRL), but this has fallen out of favor. The revocation scheme usually used for X.509 certificates is the Internet standard OCSP; see [3]. It's undecided what revocation scheme should be used for other tokens; currently there is none.

5 Authentication

This section describes the core components of authentication, highlighted in Figure 5: the trust root, token sources, and the speaks-for engine. Then it touches briefly on other components: user logon, device and app authentication, compound principals, and capabilities.

Access control is based on checking that the principal making a request is authorized to access the resource, in other words, that the principal speaks for the resource. This check typically involves a trust chain like the one in the example of section 3.2:

$K_{SSL} \Rightarrow K_{logon} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`

Where do these claims come from? They can be *known*, (that is, built in), or they can be *deduced* from other claims or from tokens, which are claims made by known principals. The trust root holds the built in claims, token sources supply tokens, and the speaks-for engine makes the deductions. Thus these components are the core of authentication:

1. The trust root holds claims that we *know*, such as $K_{Verisign} \Rightarrow$ Verisign. All trust is local, so the trust root is the basis of all trust.
2. Token sources provide claims that *others say*, such as $K_{Verisign}$ **says** $K_{Amazon} \Rightarrow$ Amazon.

3. The speaks-for engine consumes claims and tokens to *deduce* other things we may need to know, such as what tokens to believe, nested group memberships, impersonation, etc.

5.1 Trust root

All trust is local.

The trust root is a local store, protected from tampering, that holds things that a system (a machine, a session, an application) *knows* to be true. Everything that a system knows about authentication is based on facts held in its trust root. The trust root needs to be tamper-resistant because attackers who can modify it can assign themselves all the power of any principal allowed on the system.

The trust root is a set of claims (speaks-for facts) that say what keys (or other identifiers) are trusted and what identifiers (names, SIDs) they can speak for. Typical trust root entries are:

$K_D \Rightarrow$ `SID`/D	key K_D speaks for domain identifier D
$K_{\texttt{Microsoft}} \Rightarrow$ `microsoft.com`	key $K_{\texttt{Microsoft}}$ speaks for the name `microsoft.com`
$K_{\texttt{Verisign}} \Rightarrow$ `DNS/*`	the key $K_{\texttt{Verisign}}$ speaks for any DNS name
$K_{DR} \Rightarrow$ `SID/*`	key K_{DR} speaks for all domain identifiers

Because all trust is local, the trust root is local, and it must be set up manually. It must also be protected, like any other local store whose integrity is important. Because manual setup is expensive and error-prone, a trust root usually delegates a lot of authority to some third party such as a domain controller or certificate authority. The third claim example above, $K_{\texttt{Verisign}} \Rightarrow$ `DNS/*`, is such a delegation. It says that Verisign's key is trusted for any DNS name. Another example of such a delegation is the first one above, $K_D \Rightarrow$ `SID`/D, which delegates authority over the domain identifier D to the key K_D.

All trust is partial.

For convenience people tend to delegate a great deal of authority in the trust root. For example:

- A domain-joined machine trusts its domain controller for any SID.
- Most trust root entries for X.509 certificate authorities trust the authority for any DNS name.
- Today Microsoft Update is trusted by default to change entries in a Windows X.509 trust root.

This is not necessary, however. In a speaks-for claim, a delegation can be as specific as desired. Existing encodings of claims are not completely general, but for example, name constraints in a X.509 certificate can either allow or forbid any set of subtrees of the DNS or email namespace.

A very convenient way of limiting the authority of the delegation in the trust root is the rule that "most specific wins". According to this rule, a trust root with the two entries

$K_{\texttt{Verisign}} \Rightarrow$ `DNS/*;` $K_{\texttt{MS}} \Rightarrow$ `microsoft.com`

means that $K_{Verisign}$ speaks for every DNS name except those that start with `microsoft.com`. It may also be desirable to find out what key $K_{Verisign}$ says speaks for `microsoft.com`, and notify an administrator if that key is different from K_{MS}.

5.1.1 Agreeing on conventionally global identifiers

As we saw in section 4.5, we would like to use names such as `microsoft.com` as global identifiers. Since this name doesn't start with a key and therefore is not fully qualified, however, and since all trust is local, this can only be done by convention. There is nothing except convention to stop two different trust roots from trusting two different keys to speak for `microsoft.com`, or from delegating authority over `*.com` to two different third parties that have different ideas about what PKI speaks for `microsoft.com`.

Our goal is that "normal" trust roots should agree on conventionally global identifiers (SIDs and DNS names). We can't force them to agree, but we can encourage them to consult friends, neighbors and recognized authorities, and to compare their contents and notify administrators of any disagreements.

As long as trust roots delegate authority to the same third parties they will agree. If they delegate to two different third parties that agree, the trust roots will also agree. So it is desirable to systematically detect and report cases where recognized authorities disagree.

5.1.2 Replacing keys

The cryptographic mechanisms used in distributed authentication merely take the place, in the digital world, of human authentication processes. These are not just human-scale scenarios performed faster and more accurately, however; they are scenarios that are too complex for unaided humans. Therefore it's important that human intervention be needed as seldom as possible.

It's simple to roll over a cryptographic key automatically, which is fortunate since good cryptographic hygiene demands that this be done at regular intervals. The owner of the old key simply signs a token K_{old} **says** $K_{new} \Rightarrow K_{old}$. Both keys will be valid for some period of time. The main use of these tokens is to persuade each authority that issued a certificate for K_{old} to issue an equivalent certificate for K_{new}.

When a cryptographic key is stolen or otherwise compromised, or the corresponding secret key is lost, things are not so simple. If the key is compromised but not lost, often the first step is to revoke it with a revocation certificate K_{old} **says** "K_{old} is no longer valid"; by a slight extension of (S3), everyone believes this. See section 4.9.2.

The lost or compromised key must now be replaced with a new key. That replacement process requires authentication. In the simplest case, there is an authority responsible for asserting that the key speaks for a SID or name, for example, a trust root (the base case), Verisign or a domain ID service. This authority must have a suitable ceremony for replacing the key. Here are five examples of such a ceremony:

- You sign a replacement request with a backup key.
- You visit the bank in person.

- You give your mother's maiden name.
- You call up your associates in a P2P system on the phone and tell them to change their trust roots.
- Microsoft takes out full page ads in every major newspaper announcing that the Microsoft Update key has been compromised and explaining what you should do to update the trust root of your Windows systems.

5.2 Token sources

Recall that a token is a signed claim (speaks-for statement): issuer **says** $P \Rightarrow Q$. In today's Windows, the sources of tokens are highly specialized to particular protocols. For example, a domain controller provides Kerberos tokens, and the SSL protocols obtain server and client certificates. Any entity that obeys a suitable protocol (like the STS protocol for Web Services) can be a source of tokens.

The same host may get tokens from many sources, and any kind of token source can be local, remote, or both. In addition to coming from domain controllers, protocols such as SSL and IPSec, and Web Services Security Token Services, tokens can come from public key certificate authorities, from peer machines, from searches over web pages or online databases that contain tokens, from Personal Trusted Devices such as smart cards or (trusted) cellphones, and from many other places. In corporate scenarios most if not all tokens will probably come from the corporate authentication authority, but in P2P scenarios they will often come from peer machines as well as from services such as Live. This means that a standard Windows machine needs to be a token source.

The simplest kind of token to manage is signed by a key, and therefore can be stored anywhere since its security depends only on the signature and not on where it is stored. If the token is signed by a public key, anyone can verify it. However, a token can also be signed by a symmetric key, and in this case it usually must come from a trusted online source that shares the symmetric key with the recipient of the token.

5.3 Speaks-for engine

The job of the speaks-for engine is to derive conclusions about what principals are trusted, starting from claims and adding information derived from tokens. The starting claims are:

- The ones in the trust root.
- If you are checking access to a resource that has an ACL, the claims in the ACL. Recall that we view an ACL entry as a claim of the form SID $\Rightarrow_{\text{permissions}}$ resource.

Today this reasoning is done in a variety of different places. For example, in Windows:

- Logon, both interactive and network, derives the groups and privileges that a user speaks for; this is called group expansion. Part of this work is done in the host, part in the domain controller.

- X.509 certificate chain validation, which is used to authenticate SSL connections, for example, derives the name that a public key speaks for. In Windows it also does group expansion and optionally maps a certified name to a local account.

- `AccessCheck` uses an NT token, which asserts that a thread speaks for every SID in a set, and an access control list, which asserts that every SID in a set speaks for a resource, to check that a thread making a request has the necessary access (that is, speaks for) the resource.

- A Web Services STS takes authentication tokens supplied as input and a query, and produces new tokens that match the query. It can do this in any way it likes, but in many cases it has a database that encodes a set of claims (for example, associating keys with users or users with attributes), and the tokens it produces are just the ones that the speaks-for engine would produce from those claims and the inputs.

Although some or all of these specialized reasoning engines may survive for reasons of performance or expediency, or because they implement specialized restrictions, every conclusion about trust should be derived from a set of input claims and tokens using a few simple rules.

The implementation of this tenet is a *speaks-for engine*, a piece of code that takes a set of claims and tokens as input and produces all of the claims that follow from this input. More practically, it produces all of the claims that match some query. In general, the query defines a set of claims. For example, for an access to a resource, the query is "Does this request speak for this resource about this operation". For group expansion, the query is "What are all the groups that this principal speaks for".

The speaks-for engine produces one or more chains of trust demonstrating that principal P speaks for resource T about access R. For example, in section 3 we saw how to demonstrate that $K_{SSL} \Rightarrow_{r/w}$ `Spectra` by deriving the chain of trust

$K_{SSL} \Rightarrow K_{logon} \Rightarrow K_{Alice} \Rightarrow$ `Alice@Intel` $\Rightarrow$ `Atom@Microsoft` $\Rightarrow_{r/w}$ `Spectra`

Each link in this chain corresponds to a claim, either already in the trust root or derived from a token. For example, we derive $K_{Alice} \Rightarrow$ `Alice@Intel.com` from the token K_{Intel} **says** $K_{Alice} \Rightarrow$ `Alice@Intel.com`, using the claim $K_{Intel} \Rightarrow$ `Intel.com`. This fact comes either from the trust root or from another token $K_{Verisign}$ **says** $K_{Intel} \Rightarrow$ `Intel.com`, using the claim $K_{Verisign} \Rightarrow$ `*.com`. So the main chain of trust has auxiliary chains hanging off it to justify the use of tokens. The entire structure forms a *proof tree* for the conclusion $K_{SSL} \Rightarrow_{r/w}$ `Spectra`.

When P is a set of SIDs in an NT token, R is a permission expressed in the bit mask form used in Windows and Unix ACLs and T has an ACL, this is a very simple, very efficient computational proof.

The full speaks-for calculus extends the flexibility and power of this statement. P can be a principal other than SIDs. T can be the name of a resource or a named group of resources. Rights R can be expressed as names and as named groups of rights. A principal P can delegate to Q its right R to T by the token P **says** $Q \Rightarrow_R T$ (if P has the right to do this).

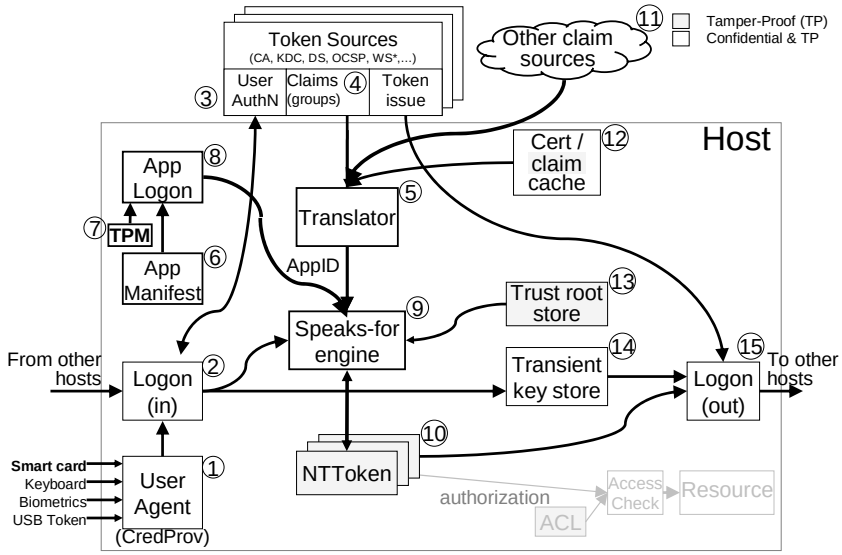

Figure 6: Authentication: The full story

For example, what can be delegated in an X.509 certificate chain is the permission to speak for some portion of the namespace for which the chain's root key can speak. This does not include the ability to define groups, for example, because group definition is outside the X.509 certificate scope. For that, one can use another encoding of a speaks-for statement (perhaps in SAML, XACML or ISO REL). From the speaks-for engine deduction we can establish that some key (bound to an ID by X.509) speaks for some group (defined by the other encoding—e.g., SAML), and establish that without having to teach SAML to understand X.509 or teach X.509 to understand SAML.

5.4 Additional components

Figure 6 shows all the components of authentication. They are (starting in the lower left corner of the figure and roughly tracing the arrows in the figure, which follow the walk-through in section 3.13.1; * marks components already discussed):

1. **User Logon Agent:** a module that is responsible for gathering authentication information from human users.
2. **Logon (in):** a module that takes logon requests (currently user, network, batch or service), interacts with token sources, and collects the principals that the user speaks for.
3. **Token Sources (User Authentication):** a source, whether local or remote, such as the Kerberos KDC or an STS, that verifies a logon and provides SIDs or other identifiers to represent the logged-on principal.
4. ***Token Sources (Claims (groups), Token issue):** a source of group and attribute information. This information may either be obtained over a secure channel, or issued as a token.
5. **Translator:** a dispatcher and a collection of components, each of which verifies the signature on a token and translates that token into an internal claim.
6. **App Manifest:** a data structure that completely specifies an application (listing the modules of the application and the hash of each module).

7. **TPM:** hardware support for strong verification of application manifests and of the entire stack on which the application runs.
8. **App Logon:** code that compares an application being loaded into a process against the manifest for that application and, when the two agree, assigns an appropriate SID to that process.
9. ***Speaks-for Engine:** the module that derives claims according to the speaks-for calculus—of primary use in authorization but used in authentication to deduce group memberships.
10. **NT Token:** the existing Windows NT Token—of which there is at least one per session—containing a collection of SIDs identifying the system on which the logon initiated, the user, groups to which this process belongs and the application ID of the process application. In other applications of the architecture this will be a general security context, that is, a principal. Authentication verifies that the user and app speak for this principal.
11. **Other claim sources:** token or claim sources that do not fit the model of Token Sources—tokens or claims can come from anywhere.
12. **Cert / claim cache:** a local cache of certificates or claims (in general, tokens)—in either external or internal form.
13. ***Trust root:** a protected store of speaks-for statements representing things that this session knows**.**
14. **Transient key store:** a protected and confidential store of cryptographic keys (symmetric keys and private keys) by which this session authenticates (proves) itself to remote entities.
15. **Logon (out):** the module with which this session authenticates (proves) itself to a remote entity, including both protocols for authentication with negotiation and the user interface that allows a human operator to decide what information to release to the remote system (the CardSpace Identity Selector).

5.5 User agent and logon

User logon (often called *interactive* logon) does two things:

- It authenticates the user to the host, giving the host evidence that the user is typing on the keyboard and viewing the screen.
- It optionally also makes it possible for the host to convince others that it is acting on behalf of the user without any more user interaction. This process of convincing others is called *network logon*.

There are many subtleties in user authentication that are beyond the scope of this paper. Here are the steps of user authentication in its most straightforward form:

1. The user agent in the host collects some evidence that it interacted with the user, called *credentials*: a nonce signed by a key or password, biometric samples (the output of a biometric reader: measurements of fingerprints, irises, or whatever), a one time password, etc.. Modularity here is for the data collection, which is likely to depend on the type of evidence, and often on the particular hardware device that provides it.

2. It passes this evidence to logon along with the user name.

3. Logon sends the evidence, together with a temporary logon session key K_{logon}, over a secure channel to a user authentication service that understands this kind of evidence; the service may be local, like the Windows SAM (Security Accounts Manager), or may be remote (as in the figure) like a domain controller. Modularity here is for the protocol used to communicate with the service.[14]

4. The authentication service evaluates the evidence, and if it is convinced it returns "yes, this evidence speaks for this user name".

5. In addition, to support single sign-on it returns tokens *authority* **says** $K_{logon} \Rightarrow$ *user name* and *authority* **says** $K_{logon} \Rightarrow$ *user SID*. It may also return additional information such as $K_{logon} \Rightarrow$ *authentication method* or $K_{logon} \Rightarrow$ *logon location*.

Single sign-on works by translating the user's interactive authentication to cryptographic authentication. Logon generates a cryptographic key pair for the user's logon session. The new key K_{logon} is certified by a more permanent key (on the user's smartcard, in the computer's hardware security module, sealed by a password, from a domain controller, or whatever): $K_{permanent}$ **says** $K_{logon} \Rightarrow$ *user*. It is then used for that one logon session. Since today there are protocols that insist on secret key such as Kerberos, and others that use public key such as SSL, logon should certify one of each.

5.6 Device authentication

Device authentication is more subtle than you think. As much as possible, computers and other digital devices should authenticate to each other cryptographically with tokens of the form *K* **says** ... As we have seen, for these to be useful the key *K* must speak for some meaningful name. This section explains how such names get established, using the example of very simple devices such as a light switch or a thermostat. More powerful devices with better I/O, such as PCs, can use the same ideas, but they can be much more chatty.

It is a fundamental fact of cryptographic security that keys must be established initially by some out of band mechanism. There are several ways to do this, but two of them seem practical and are unencumbered by intellectual property restrictions: a pre-assigned meaningful name and a key ferry. This section describes both of them.

You might think that this is a lot of bother over nothing, but consider that lots of wireless microphones and even cameras are likely to be installed in bedrooms in apartments. Some neighbors will certainly be strongly motivated to eavesdrop on these devices. Be-

[14] You might think that one protocol could work for any kind of authentication factor. There are two reasons for using different protocols. One is purely historical: existing services used particular protocols. The other is that some protocols, such as Kerberos, depend on the fact that the workstation has a key that it can use to communicate secrets to the service. In Kerberos, for example, the user's password is the source for such a key. Biometric samples don't work. Other protocols, such as SSL, create a secure channel to the service and authenticate it starting with nothing but a trust root entry for a generic authority such as Verisign. As far as I know, SSL secure channel setup together with conventions for finding the service to use, encapsulating the evidence, and allowing for interaction between the user and the service would be a universal protocol.

cause the wireless channel is a broadcast channel, the neighbor can mount a "man-in-the-middle" attack that intercepts the messages passing between the device and your computer, and pretends to be the device to the computer and the computer to the device.

5.6.1 Device authentication by name

For device authentication, the simplest such mechanism is for the manufacturer to install a key K^{-1} in the device, give it a name *dn*, and provide a certificate *manufacturer* **says** $K \Rightarrow dn$, for example, `Honeywell` **says** $K \Rightarrow$ `thermo524XN12.Honeywell.com`. In this example the out of band channel is a piece of paper with the name `thermo524XN12` printed on it that comes in the box with the thermostat. After installing the thermostat in the living room, the user goes to a computer, asks it to look around for a new device, reads the name off the screen, compares it with the name on the paper, and assigns the thermostat a meaningful name such as `LivingRoomThermostat`. Of course a hash of the device's key would do instead of a name, but it may be less meaningful to the user (not that `524XN12` is very meaningful). This protocol only authenticates the device to the computer, not the other way around, but now the computer can "capture" the device by sending it a "only listen to this key" message.

In many important cases this assignment needs to be done only once, even though many different people and computers will interact with the device. For example, a networked projector installed in Microsoft conference room 27/1145 might be given the name `projector.27-1145.microsoft.com` by the IT department that installs it. When you walk into the conference room and ask your laptop to look around for available projectors, seeing one that can authenticate with that name should be good enough security for almost anyone. Because this name is very meaningful, authenticating to it is just like authenticating to any other service such as a remote file system.

In many other important cases this assignment only needs to be done very rarely because the device belongs to one computer, which is the device's exclusive user until the computer is replaced. This is typical for an I/O device such as a scanner or keyboard.

5.6.2 Device authentication by key ferry

There are three disadvantages to pre-assigned names that might make you want to use a different scheme:

- You might lose the piece of paper, in which case the device becomes useless.
- You might not trust the manufacturer to assign the name correctly and uniquely.
- You might not trust the user to compare the displayed name with the printed one correctly (or at all, since users like to just click OK)

The alternative to a pre-assigned name as an out of band channel is some sort of physical contact. What makes this problem different from peer-to-peer user authentication is that the device may have very little I/O, and does not have an owner that you can talk to. There are various ways to solve this problem, but the simplest one that doesn't assume a cable or other direct physical connection is a "key ferry". This is a special gadget that can communicate with both host and device using channels that are *physically* secure. This communication can be quite minimal: upload a key from host into ferry at one end;

download the key out of ferry into device at the other end. The simplest ferry would plug into USB on the host and the device.

5.7 App ID

This section explains how to authenticate applications. While it's also important to understand how apps are isolated so that it makes sense to hold an app responsible for its requests, this out of scope here.

The basic idea is that apps are principals *just like users*:

- An app is registered in a domain, with an AppSID and a name. This domain is typically the publisher's domain.
- An app is authenticated by the hash of a binary image, just as a user is authenticated by a key.
- When a host makes a new execution environment (process, app domain, etc.) and loads a binary image into it, the new environment gets the hash of the image (and everything that the hash speaks for) as its ID.
- User, machine, and app identifiers can all appear on ACLs or as group members.

Also like users, apps can be put into groups, but this is even more important for apps than it is for users because groups are the tool for managing multiple versions of apps. Like any group membership, the fact that an app is a member of the group can be recorded in AD, or it can be represented in a certificate that is digitally signed by an appropriate authority. Like groups containing users, groups containing apps can nest to make management easier. For example, the `GoodApps` group might have members `GoodOffice`, `GoodAcrobat`, etc.

AppSIDs are probably assigned from the same space as user, group, and machine SIDs, though frequently the AppSIDs are from a "foreign" domain, that of the software publisher (e.g. Microsoft). The assignment is encoded in a signed certificate (usually in the manifest) that associates the binary image with an AppSID and a name in the publisher's domain.

AppSIDs can also be assigned locally by a domain or machine administrator. This must always be done for locally generated applications, and can be done for third party applications (where the AppSID is assigned as part of some approval process). The application is identified by a hash just as in the published case. The local administrator can sign a manifest just like the publisher, or can define a group locally or in AD.

ACLs list the users, machines, and applications that are allowed to access the resource. Sensitive resources might only be accessible through applications in the `GoodApps` group. Specialized resources might only be accessible to specific applications (plus things like backup and restore utilities).

5.7.1 AppSIDs and versions

A certificate for an app is a signed statement that says something like "hash 743829 => `MS/Word12.3.1`, `s-msft-word12.3.1`. Applications contain many files; a *manifest* is a

data structure that defines the entire contents of the application. The manifest includes hashes of all the component files, and it's the hash of the manifest that defines the app.

The manifest can reference system components that are not distributed with the app (e.g. system .dlls). Such a component is considered to be part the platform on which the app is running, not part of the app; see section 5.7.2, and it is referred to by a name, which need not change if the component is patched. There are many complications having to do with side-by-side execution that are not relevant here; it's the platform's job to ensure that the name gets bound appropriately for both security and compatibility. In this respect an app treats a platform component just like a kernel call.

The way this is normally encoded is that the publisher includes the principals that the app speaks for (such as `MS/Word12.3.1`, `s-msft-word12.3.1`) in the manifest, and then simply signs the hash of the manifest. This is just a useful coding trick. Of course, the signer of the manifest (or other app certificate) must be authoritative for the domain of the SID and for the name, just as for any other speaks-for statement.

If the system trusts its file store, it can verify the manifest at install time and cache it. This also covers cases where installation includes updates to registry settings and such.

There may be good reasons not to change AppSIDs with each small version change such as a patch. Changing the AppSID requires updating all policy that references it. Some admins will want to do so; others will not. An admin can avoid having to update lots of policy by adding a level of indirection, defining a group and putting the AppSID for each new version into the group; this gives the admin complete control. Publishers can make the admin's life easier by including multiple AppSIDs in a manifest. For example, the manifest for a version of Word might say that it is `Word`, `Word12`, and `Word12SP2` as well as `Word12.3.1`. In SP3, the first two SIDs remain the same. Then Contoso ITG can say `MS/Word12`, `MS/Word11.7.3` => `Contoso/GoodWord`. Since all trust is local, the structure of the name space for an app is in the end up to the administrator of the machine that runs it. The job of a publisher like Microsoft is to provide some versions and names that are useful to lots of customers, not to meet every conceivable need.

5.7.2 The AppID stack

The only assertions an app can make directly are ones encoded in its manifest. When the app is running it depends on its *host environment* to provide the isolation that is needed for an app identity to make any sense. Typically the host environment is itself hosted, so the entire app identity is actually a stack:

```
StockChart
IE 7.0.1
Vista + patch44325
Viridian hypervisor + patch7654
MachineSID
```

At the bottom, the machine gets its identity from a key it holds. Ideally this key is protected by the TPM.

We could describe the identity of the app by hashing together the hashes of all the things below it on the stack, just as we hashed all the files of the app together in the manifest.

This is probably not a good idea, however, because if there are ten versions of each level in the stack there will be 100,000 different versions—hard to manage. It's better to manage each level separately.

Access control of course sees the whole stack. Taking account of plausible group memberships, an ACL might say `GoodApp` **on** `GoodOS` **on** `GoodMachine` gets access, where "on" here is an informal operator that makes a single principal out of an app running on a host. This makes it easy for the administrator to decide independently which apps, which OS's, and which machines are good. Going further, the administrator might define `GoodApp` on `GoodOS` on `GoodMachine` ⇒ `GoodStuff` and just put `GoodStuff` on ACLs.

Note that the policy for what stacks are acceptable might come from the app rather than user or administrator. The main example of this is DRM, in which some remote service that the app calls, such as the license server, demands some kind of evidence that it is running on a suitably secure hardware and OS. The app's manifest might even declare its requirements, but of course an untrustworthy host could ignore them, so the license server has to check the evidence itself.[15]

When a running program loads some new code into itself (a dll, a macro, etc.), it has a number of options about the appID of the resulting execution environment. It can:

1. Use the new code's appID to decide not to load it at all.
2. Trust the code and keep the same AppID the host had before. This is typically what happens at an extensibility point, or in general when an app calls `LoadLibrary`.
3. Downgrade its own AppID to reflect less trust in the new code.
4. Sandbox the new code and add another level to the stack. Of course the credibility of the resulting AppID is only as good as the isolation of the sandbox.

ACL entries on the operation of loading code can express this choice. Note that when an app calls `CreateProcess`, for example, it is not loading new code into itself, but asking its host OS to create a sibling execution environment, and it's the host's job to assign the appID for the new process, which might have different, even greater rights that the app that called `CreateProcess`.

5.8 Compound principals

Simple principals that appear in access control policy are usually human beings, devices or applications. In many cases, two or three of these will actually provide proof (authenticate a request). Today only one principal typically provides proof—either a human being or a computer system. Multiple proofs of origin can be used to strengthen security. One important example of this is combining a user identifier and an appID. There are two main ways this can be done:

1. **Protected subsystem**: access is granted only to the combination of two principals, not to either of them alone—for example, opening of a file for backup can be

[15] The app itself could also demand properties from its host, but since the host has complete control over the app, this demand could not be enforced very securely. Ideally the evidence for the license server is a chain of certificates rooted in the hardware TPM's key.

allowed to a registered backup operator, but only when that operator is also running a registered backup application.

2. **Restricted Process**: the desired access is granted only if each of the two or more principals qualify for that access individually[16]—for example, an applet downloaded from a web page at `xyz.com` might be allowed to access things on `xyz.com` but not on the user's local machine, and the user running that applet might have access only to objects that the user and the applet both can access.

These two ways of combining principals correspond to **and** and **or**. The principal `billg` **and** `HeadTrax` is `billg` running the `HeadTrax` protected subsystem; Windows doesn't currently have a way to add such an appID to a security context. The principal `billg` **or** `MyDoom` is `billg` running the `MyDoom` virus; in Windows today this is a `billg` process with a `MyDoom` restricted token.

A Windows security context (or NT token) is a set of SIDs that defines a principal: the **and** of all those SIDs. This principal can exercise all the power that any of those SIDs can exercise. Thus when a security context makes a request, the interpretation is that each of the SIDs independently makes that request; if any of them is on the resource's ACL, the request is granted. So *security context* **says** *request* is SID1 **says** *request* **and** SID2 **says** *request* ..., which is another way of saying that *security context* = SID1 **and** SID2 **and** ....

There are other uses for compound principals made with **and**. Financial institutions often demand what they call dual control: two principals have to make a request in order for it to get access to an object such as a bank account. In speaks-for terms, this is P_1 **and** $P_2 \Rightarrow$ object. The method for making long-term keys fault-tolerant described in section 5.1.2 is another example of this, which generalized **and** to *k*-of-*n*.

There are also other uses for compound principals made with **or**. In fact, an ACL is such a principal. It says that (ACE_1 **or** ... **or** ACE_n) $\Rightarrow$ object.

5.9 Capabilities

A capability for an object is a claim that some principal speaks for the object immediately, without any indirection. A familiar example in operating systems is a file descriptor or file handle for an open file. When a process opens the file, the OS checks that it speaks for some principal on the file's ACL, and then creates a handle for the open file. The handle encodes the claim that the process speaks directly for reads and writes of the file, without any further checking; this claim is encoded in the OS data structure for the handle. A capability is thus a *summary* of a trust chain. Usually it has a quite limited period of validity, in order to avoid the need to revoke it if the trust chain becomes invalid.

[16] This kind of access is provided today in Windows by the *restricted token*, in which one has effectively two NTtokens, one for the user's principals and one for a service ID. `AccessCheck` is called with each of those tokens and the Boolean results of those calls are then **and**ed.

For a capability to work without a common host such as an OS, it must be in a token of the form *object* **says** $P \Rightarrow$ *object* that the object issues after evaluating a trust chain. Later *P* can make a request along with this token, and the object will grant access without having to examine the whole chain. Such a token doesn't have to be secret, since it only grants authority to *P*.

6 Authorization

The main problem with authorization is management. Products usually have enough raw functionality to express the customer's intent, but there is so much detail to master that ordinary mortals are overwhelmed. The administrator (or user) needs a way to build a **model** of the system that drastically reduces the number of items they need to configure. The model needs to not only handle enterprise level security, but also "scale down" to small businesses and homes where there is no professional IT administrator, to peer-to-peer systems, and to mobile platforms and small devices.

Authorization also needs to be feasible to implement. It needs to **scale up** to the Internet, avoiding algorithms and data structures that only work for intranet-sized systems or that depend on having a single management authority for the whole system. Everything that works locally should work on the Internet. Authorization needs to support **least privilege**, by taking account of application as well as user identity, so that trusted apps can get more privileges and untrusted ones fewer; this must work even though apps come in many versions and are extensible. And it needs to be **efficient**: fast in the common case and reasonable in complex cases, even in a large system; it needs to identify problem cases so that people setting policy can avoid them.

6.1 Overview

The underlying semantics of authorization is the notion of "speaks-for": there is a chain of principals, starting with the principal making a request (typically a channel on which the request is transmitted or an encryption key that signs the request) and ending with the resource. For example:

$K_{session}$ => K_{Paul} => Paul@microsoft.com => Zeno@microsoft.com => http://winsecurity/sites/strategy

We call the part of this chain closer to the user "authentication", and the part closer to the resource "authorization". This division is somewhat arbitrary, since there is no sharp dividing line.

In order to make authorization more manageable, you can build a model that collects resources into *scopes* and defines *roles*, each with a set of predefined permissions to execute operations on the resources in the scope. In addition, you can build a *template* for a scope and its roles, and then instantiate the template multiple times for different collections of resources that have the same pattern of authorization policy. Figure 7 is an overview that shows the main steps in specifying and checking authorization.

This model-based access control (MBAC) organizes resources into **scopes** and principals making requests into **roles**.

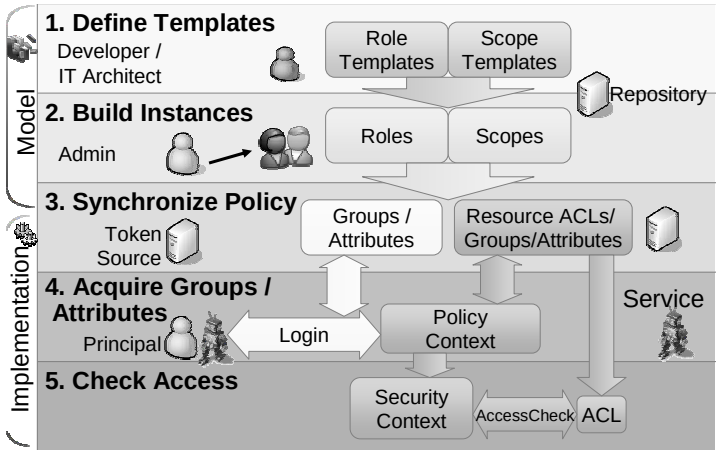

Figure 7: Authorization architecture overview.

1. The developer or IT architect defines *templates* for scopes and roles that can be used repeatedly in similar situations.

2. The administrator or owner makes *instances* of these templates, groups resources into scopes, and assigns principals to roles.

The remainder of the picture shows how to implement the policy that the model defines.

3. The system compiles or *synchronizes* the model's policy into groups, claims, and ACLs on resources used to do access checks efficiently. When a service starts it acquires its own identity and resource groups, along with those of its enclosing execution environments (OS, device, etc.)

4. The user logs in to a service and acquires groups and claims from the directory or STS to add to the identifiers she already has. The system combines these with resource manager claims and service trust policy to obtain a set of principals that the service thinks the user speaks for.

5. Finally, the set of principals is checked against the ACL for the resource the user is trying to access.

The templates and instances are part of MBAC. The acquisition and access check are part of implementation. The model and implementation are connected when the policy is synchronized.

6.2 Model-Based Access Control (MBAC)

The idea of MBAC is to make authorization policy accessible to ordinary mortals; think of it as Excel for authorization. The main customer pain point is that security management is too hard. There are thousands of security knobs (individual ACLs, privileges, resource names, etc.) on each computer, and in a large installation there are thousands of computers. No human can keep that number of separate objects in mind. The model con-

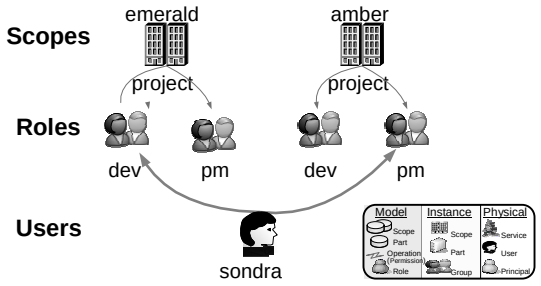

Figure 8: The admin sees two scopes, `emerald` and `amber`; both are instances of a `project` repository template. A project has two roles, `dev` and `pm`. Sondra is a `dev` for emerald and a `pm` for `amber`.

ceals the complexity of the underlying implementation from users and administrators (though they can dive down into individual groups and ACLs if they really need to).

MBAC shines when complex policies apply to multiple objects. It reduces repetitive manual effort by the administrator, and makes it easy to find out what the policy is after a long history of incremental changes. Our examples are necessarily contrived, since something simple enough to put in this paper is simple enough to do manually. So use your imagination to see how the reduction in administrative work is actually substantial for real world scenarios.

Figure 8 shows the administrator's view of a model for part of a system—two project repositories that are scopes for resources, one for the `emerald` project and one for the `amber` project. Each project has two roles: one for PMs and one for devs. When deploying a project repository you create a group for each role, containing the users who are in that role for that project. Thus a scope is a collection of resources, and a role is a collection of principals.

This is a simple model—the admin just puts a user, such a Sondra, into the correct group, and all the permissions and memberships are created as a consequence. The actual situation might be messier, as Figure 9 shows. Administering this manually would be quite difficult, but with MBAC the administrator doesn't have to worry about the mess when configuring authorization policy.

Someone has to worry, of course, and that person is the designer of the template, typically a developer or an IT architect. Figure 10 shows the `SharePoint` template and the `emerald.specs` scope that is an instance of it. Such a leaf scope corresponds to an instance of a service along with (a subset of) its resources. The developer of the service, in addition to coding the service, creates a **scope template** that defines the roles for the service. A role determines the permissions for a user in that role. Each role is tailored to enable a user to perform some task—like being a teller, or an HR benefits clerk, or in this example, a contributor or viewer of documents on a SharePoint server. A viewer can read documents; a contributor can edit documents, and also is a viewer (this is an example of role nesting). These predefined roles determine the combination of permissions that get tested, to make sure that they correctly enable the desired tasks. Thus the developer or IT

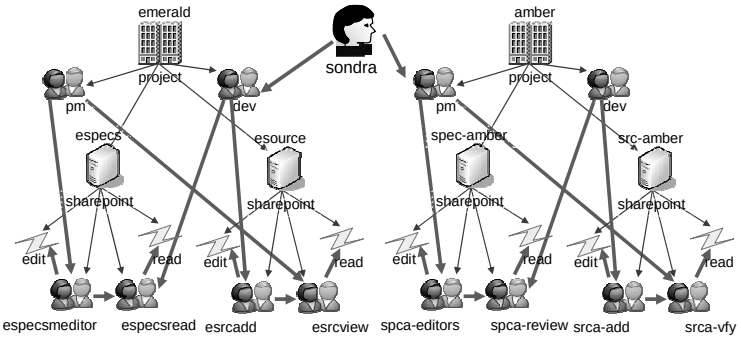

Figure 9: Manual administration gets messy

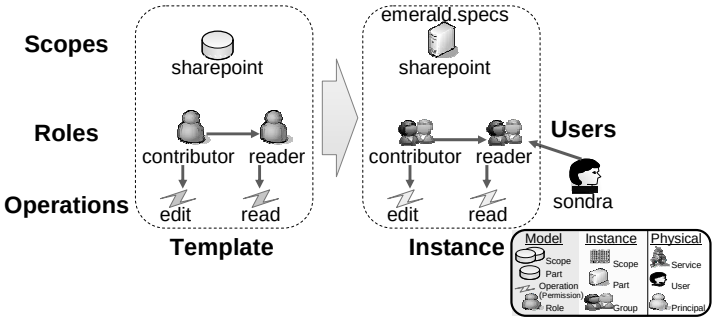

Figure 10: A template and an instance `emerald.specs` for `Sharepoint`; Sondra is a `viewer`.

architect is responsible for all the details of authorization policy within the scope. From the point of view of the administrator, all the ACLs are immutable.

The administrator instantiates the scope template to create a scope. The same template can be used to create many scopes. Figure 10 shows one of these, in which the contributor and viewer roles have the same permissions for the SharePoint resource in the scope that the corresponding role templates had in the template. The administrator has put Sondra into the viewer role for the `emerald.specs` scope. Each scope precisely mirrors the scope template and has the resources, roles, and permissions defined in the template, just as each instance of a class in an object oriented programming language precisely mirrors the class definition.

An IT architect can create higher level templates. In Figure 11 SharePoint is used to create the project repository we described earlier. The `project` has two subparts, called `specs` and `source`. The `PM` role is assigned to the `contributor` role in the `specs` server, and the `viewer` role in the `source` server. A part's roles constitute the interface that it exports to containing scopes. The smallest parts are actual services such as `SharePoint`; composite parts such as `project` contain subparts. The architect can nest these as deeply as necessary. We expect that there will be a market for templates that are useful to more than one organization.

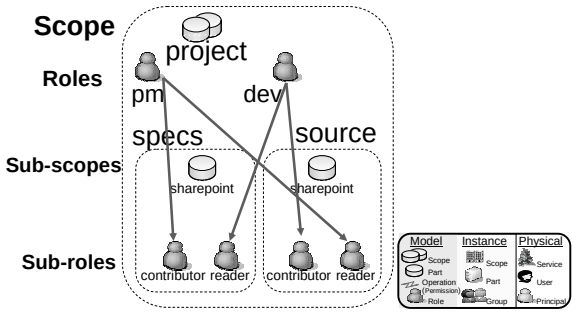

Figure 11: Build bigger parts from smaller ones. The `specs` and `source` scope templates are `SharePoint` scope templates that are parts of the outer `project` scope template, and the inner `contributor` and `viewer` role templates are populated from the outer `pm` and `dev` ones.

Because the IT architect defines this for all project repositories, all the admin has to do is instantiate the model; she no longer needs to understand all of the details. Two instances of the project template called `emerald` and `amber` would get us back to Figure 8.

6.3 The model and the real world

This section explains how the model is connected to the code and data in the real world that it is modeling. Although usually we ignore the distinction between the model and the real world, in this section we need to be clear about it, so we call the real world thing that corresponds to an object in the model its *entity*.

The goal is to keep the model and the real world synchronized, so that changes in entities (and especially creation of new entities) are reflected in the model, and the access control policy set by the model is reflected in its entities. There are three basic issues in synchronization:

1. **Naming**: An object in the model and its entity in the real world are not necessarily named in the same way.
2. **Delay**: An object and its entity are supposed to be in sync, but there may be some delay.
3. **Aggregation**: When entities change, how are the changes aggregated for notifying the model.

6.3.1 Naming: Paths and handles

Objects are named by paths: sequences of field names and queries (for selecting an object from a set-valued field). Entities are named by *handles*, which are opaque from the viewpoint of the model. The handle must have enough information to enable secure communication with the root entity.

Because paths and handles are different in general, there has to be a way to map between them. In particular, if the model wants to refer to an object's entity, it needs the entity's handle. Similarly, if an entity wants to refer to its object, it needs the object's path. We

take the view that MBAC should work without any changes to entities, as long as they have some sort of interface that is adequate for implementing the `get`, `set`, and `enum` methods described below. Thus the model needs to keep track of each object's handle, which it can do by storing it as part of the object.

In some cases a path may itself be a suitable handle. For example, the model for a file system has objects that correspond to directories and files with isomorphic names. Thus a directory object `do` has a set-valued `contents` field whose elements are the files and directories in `do`, each with a `name` field. So a file with pathname `a\b` corresponds to the object whose path is `contents?{.name="a"}.contents?{.name="b"}`. As this example illustrates, a path may include queries, and hence to use a path as a handle the entities have to be able to understand a query well enough to follow a path. The simplest kind of query has the form `[.name = "foo"]`, where `name` is a primary key, and this shouldn't be too hard for an entity.

6.3.2 The model is in charge

The model can read, and perhaps change, the abstract fields of an entity that correspond to fields of the model by invoking the `get` and `set` methods of a corresponding object: `obj.get(f)` allows the model to read the value of field `f` in the entity, and `obj.set(f, value)` allows the model to set the access control policy of the entity. If `f` is an object, `get` returns a handle to that object; see below. If a field is a large set, these methods are not suitable, so set fields have a different method: `obj.enum(f, i)` returns a handle to the `i`th element of the set, or `nil` if it has fewer elements (along with a generation number that increases every time something happens to change the object numbering). To change the membership of the set you use operations on the containing scope, such as `create`. Using these APIs a model can fully explore its entity (as long as the entity isn't changing too fast), learn the handles of all the entities, fill in all the fields of the model, and tell the entity the values of any fields that are determined by the model (normally roles).

In order to use MBAC, an entity must implement these APIs. It may also need to implement `query` and `assign` APIs to deal efficiently with large sets of objects. To reflect changes to the entity in the model more efficiently than by polling we may also want a change log. Entries in this log are `(h, f)` pairs, meaning that field `f` of entity `h` has changed.

6.3.3 Notification and aggregation

With these APIs the only way for the model to find out about changes in the entities is to do a *crawl*, that is, read out the entire state again with `get` and `enum`. This seem impractical for models of any size, so it's necessary to have some kind of change notification. Notification has three issues:

1. It has to be extremely reliable, since if any changes are missed the model's state will diverge from reality, and the only way to get it back in sync is do to a crawl.
2. The entity's name space is handles, so it can only report changes in terms of handles. These have to be mapped to paths.

3. It might be desirable to aggregate all the notifications below some point in the tree.

6.4 Scale Up

Current OS authorization mechanisms can scale quite well to enterprises (one Windows AD installation exists that holds 6 million users, for example). They need some work, however, if they are to scale to the Internet, both because things can get much bigger on the Internet, and because there's no single management authority that is universally trusted.

There are some basic features of access control that are important for scaling up:

1. All authentication and authorization statements (speaks-for statements) can be represented in three different ways:
- They can be stored locally (for example, in the trust root).
- They can be held in a database on the network (for example, active directory) and delivered over a secure authenticated connection.
- They can be expressed in a digitally signed certificate (for example, X.509 or SAML tokens), which can be stored and forwarded among the various parties in the transaction.

The first and third ways permit offline operation and offload of online services (caching). The third way means that claims can be transmitted via untrusted parties.

2. All principal identifiers that are passed from one system to another are globally unique. This means that there's no ambiguity about the meaning of an identifier.

3. Any system or domain can make use of statements from any other domain. It is trust policy, rather than domain boundaries, that distinguishes friend from foe.

4. There is an unavoidable tradeoff among freshness, availability, and performance. If you want the latest information about whether a key is revoked, for example, you cannot proceed if the source of that information is unavailable, and you must pay for the communication to get it. This tradeoff should be controlled by policy, rather than being baked in. For example, here are two possible policies for key revocation:
   - Fail without a fresh OCSP for every access.
   - If OCSP isn't available, treat all cached statements as valid for some period.

Neither one is unconditionally better than the other; it's a matter for administrators' judgment to choose the appropriate one.

In addition to these general principles, there are two topics that require special attention in scaling to the Internet:
- Trust in attribute claims made by other authorities.
- Handling groups, because both the number of groups that a principal belongs to and the total size of a group can become extremely large.

6.4.1 Scale Up: Attribute Claims

An attribute differs from a group in two ways:

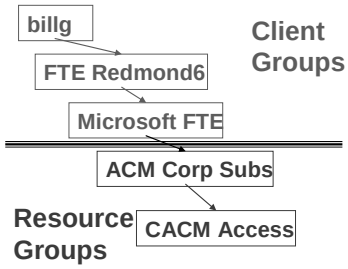

Figure 12: Corporate subscribers can access CACM online. The arrows are group membership.

- It can have a value associated with it, for example, birthdate.
- There may not be a single authority responsible for its definition. For example, birthdates may be certified by any one of 50 state driver's license issuing authorities.

For scaling up, only the second point is important. The first one is handled by conditions.

It is a system's trust policy that handles attributes from other authorities. For example, consider using a driver's license from another state to verify date of birth at a bar in New York. It's convenient for states to agree on the string name of this property. `Oasis.org` is a standards organization, and we will use `oasis.org/birthdate` as the standard name.

The first step is for the bar's trust policy to say what the primary authority is for this property:

K_{NY} ⇒ oasis.org/birthdate

Then the primary authority says which other sources to trust:

K_{NY} says K_{WA}/oasis.org/birthdate ⇒ oasis.org/birthdate

This says that New York believes Washington about birth dates. If they have a broader agreement, New York might believe Minnesota about all properties defined by `oasis`.

K_{NY} says K_{MN}/oasis.org/* ⇒ oasis.org/*

Name translation can be done, too. Suppose Illinois doesn't adopt the `oasis` name:

K_{NY} says K_{IL}/DOB ⇒ oasis.org/birthdate

6.4.2 Scale Up: Group Claims

Group membership is a scaling problem today, at least in large organizations. The reason is that a user can be a member of lots of groups, and a group can have lots of members. Today Windows manages this problem in two ways:

- By distinguishing *client* and *resource* groups (also called domain global and domain local groups in Windows), and imposing restrictions on how they can be used.
- By allowing only administrators to define groups used for security.

Figure 12 illustrates the problem. Imagine that ACM creates a group of corporate subscribers to its online digital library. There are 1000 corporate members, each with 10-1,000,000 employees, for a total of millions of individual members. Furthermore, every Microsoft employee may implicitly be a member of thousands of such groups, since Mi-

crosoft subscribes to lots of services. Thus a client may be in too many groups to list, and a resource may define a group with too many members to list.

In addition, there may be a privacy problem: the client may not want to disclose all its group memberships, and the server may not want to disclose all the groups that it's using for access control.

This is the group expansion, or path discovery, problem. The solution that Windows adopts today, and that we generalize, is to distinguish two kinds of groups:

- **Client** groups (also called *push* groups), which the client is responsible for asserting when it contacts the resource. An individual identifier is a special case of a client group. Thus in Figure 12, the client groups are green: `billg`, `FTE-Redmond6`, and `MicrosoftFTE`. A requestor's client groups are thus known to all resources (subject to privacy constraints), but there can only be a limited number of them.
- **Resource** groups (also called *pull* groups), which the resource is responsible for keeping track of and expanding as far as client group members. In Figure 12 the resource groups are blue: `ACMCorpSubs` and `CACMAccess`. The resource thus knows all the client groups that are members, but there can be only a limited number of them.

A client group can only have other client groups as members. This means that there can be only one transition from green to blue in the figure. The client asserts all its client group memberships, and the resource expands its resource groups to the first level of client groups. Consequently, if there is *any* path from the client to the resource, what the client presents and what the resource knows will intersect and the resource will know it should grant access.

Client groups are a generalization of today's domain global groups in AD. Unlike domain global groups, client groups can have members from other domains, but the client *must* know all the client groups it belongs to so that it can assert them, because the resource won't try to expand client groups.

Resource groups are a generalization of today's domain local groups in AD. Unlike domain local groups, resource groups can be listed on the ACL of any resource so long as the resource has permission to read the group membership. It's the resource administrator's job to limit the total size of the group, measured in first-level client groups. The resource *may* cache the membership of third party resource groups.

An added complication is that today Windows eagerly discovers all the resource groups in a domain a client belongs to when the client connects to any resource in the domain. This makes subsequent access checks efficient, and the protocols allow the client and the resource to negotiate at connection time, but if the domain is big (for example, if it contains lots of big file servers) there might be too many resource groups. To handle this, resources may use smaller resource scopes than an entire domain – for example, a service.

To sum up, the way to handle large-scale group expansion is by distinguishing client and resource groups. This extends what Windows does today in five ways:

1. The client and resource can negotiate what group memberships (or other attributes) are needed.

2. Both client and resource can query selected third parties for groups.
3. Both client and resource can cache third party groups. The client must do this, since it must assert all its client groups.
4. The resource can use a smaller scope to limit the number of resource groups that get discovered when the client connects.
5. The client can be configured to know which groups the resource requires.

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. Internet X.509 Public Key Infrastructure: Certificate and Certificate Revocation List (CRL) Profile, RFC 3280, http://www.ietf.org/rfc/rfc3280.txt
3. Internet X.509 Public Key Infrastructure: Online Certificate Status Protocol – OCSP, RFC 2560, http://www.ietf.org/rfc/rfc2560.txt
4. 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
5. 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
6. 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

Appendix: Basic facts about cryptography

Distributed computer security depends heavily on cryptography, since that is the only practical way to secure communication between two machines that are not in the same room. You can describe cryptography at two levels:

- Concrete: how to manipulate the bits
- Abstract: what the operations are and what properties they have

This section explains abstract cryptography; you can take it on faith that there are concrete ways to implement the abstraction, and that only experts need to know the details.

Cryptography depends on keys. The essential idea is that if you don't know the key, you can't do X, for various values of X. The key is the only thing that is secret; everything about the algorithms and protocols is public. There are two basic kinds of cryptography: public key (for example, RSA or elliptic curve) and symmetric (for example, RC4, DES, or AES). In public key (sometimes called asymmetric) cryptography, keys come in pairs, a *public* key K and a *secret* key K^{-1}. The public key is public, and the secret key is the only thing that is kept secret. In symmetric crypto there is only one key, so $K = K^{-1}$.

Cryptography is useful for two things: signing and sealing. Signing provides integrity: an assurance that signed data hasn't changed since it was signed. Sealing provides secrecy: only the intended recipients can learn any of the bits of the original data even if anyone can see all the bits of the sealed data.

For signing, the primitives are `Sign`(K^{-1}, *data*), which returns a signature, and `Verify`(K, *data*, *signature*), which returns `true` if and only if *signature* = `Sign`(K^{-1}, *data*). The essential property is that to make a signature that verifies with K requires knowing K^{-1}, so if

you verify a signature, you know it was made by someone that knew K^{-1}. With public key, you can verify without being able to sign, and everyone can know K, so the signature is like a network broadcast. With symmetric crypto, anyone who can verify can also sign, since $K = K^{-1}$, so the signature is basically from one signer to one verifier, and there's no way for the verifier to prove just from the signature that the signature came from the signer rather than from the verifier itself.

For sealing, the primitives are `Seal`(*K*, *data*), which returns sealed data, and `Unseal`(K^{-1}, *sealedData*), which returns *data* if and only if *sealedData* = `Seal`(*K*, *data*). The essential property is that you can't learn any bits of *data* (other than its length) from *sealedData* unless you know K^{-1}. With public key, anyone can seal data with K (since K is public) so that only one party can unseal it; thus lots of people can send different secrets to the same place. With symmetric crypto, the sealing is basically from one sealer to one unsealer.

There's a trick that uses public key sealing to get the effect of a signature in one important case; it's the usual way of using a certificate to authenticate an SSL session. Suppose you have made up a symmetric key K (usually a session key) and you want to know $K \Rightarrow P$, That is, any messages signed with K that you don't sign yourself come from another party P. Suppose you have a certificate for P, that is, you know $K_P \Rightarrow P$. This means that only P knows K^{-1}. The usual way to authenticate K is to get a signed statement K_P **says** $K \Rightarrow P$ from P. Instead, you can compute SK = `Seal`(K_P, K) and send it to P in the clear. Only P can unseal SK, so only P (and you) can know K.