Authentication in Distributed Systems

Butler W. Lampson
Huygens Systems Research Laboratory
Universiteit Twente
Enschede
Concerned with

- Allowing or denying access to data — access-control lists
- Guaranteeing access to data — prevent denial of service
- Controlling information flow — mandatory security policies

We shall only concern ourselves with access control
The Access-Control Model

Principal → Operation → Reference monitor → Object

Source → Request → Guard → Resource
The three ‘A’s

- **Authentication**: Who makes this request
- **Authorization**: Who is allowed to make this request
- **Auditing**: Who did what
<table>
<thead>
<tr>
<th></th>
<th>$O_j$</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$</td>
<td>$R_{ij}$</td>
<td></td>
</tr>
</tbody>
</table>

**Row:** Capability List

**Column:** Access Control List
Requests arrive on a *channel*

The channel can be a wire from a terminal, a network connection, a pipe, a system call or the successful decryption of an encrypted message.

The task of the monitor is to determine which principal is responsible for a request that arrives on a channel.

In other words, the monitor must authenticate the channel.
In a centralized system, authenticating a channel is easy: the operating system implements them all and knows the principal responsible for each process.

In a distributed system, we must deal with

- Autonomy
- Size
- Heterogeneity
- Fault tolerance
Distributed Systems Issues

- **Autonomy**: Not all machines are equally trusted. Want to express restricted access for trusted users on not-fully-trusted machines
- **Size**: The size of the system may make it necessary for there to be multiple sources of authority
- **Heterogeneity**: There may be different kinds of channels, protected in different ways (physically secure wires, encryption hardware, local communication)
- **Fault tolerance**: When parts of the system are broken, the system must remain available. Access may neither be unjustly permitted nor unjustly denied in the face of crashes
A theory allows

- Exact description of assumptions about authority and trust
- Exact description of the rules for access control
- Exact analysis of where that gets you

The theory is used to analyze everything except the channels based on encryption and security of hardware and local operating system
• **Principals**: Active entities that make requests
• **Statement**: Things principals say
<table>
<thead>
<tr>
<th><strong>Simple</strong></th>
<th><strong>People</strong></th>
<th>Lampson, Abadi</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machines</strong></td>
<td>VaxSN12648, 4thFloorPrinter</td>
<td></td>
</tr>
<tr>
<td><strong>Roles</strong></td>
<td>Manager, Secretary, NFS-Server</td>
<td></td>
</tr>
<tr>
<td><strong>Sets</strong></td>
<td>Services</td>
<td>SRC-NFS, X-Server</td>
</tr>
<tr>
<td><strong>Groups</strong></td>
<td>SRC, DEC-Employees</td>
<td></td>
</tr>
<tr>
<td><strong>Channels</strong></td>
<td>Wires or I/O Ports</td>
<td>Terminal 14</td>
</tr>
<tr>
<td><strong>Encrypted Channels</strong></td>
<td>DES encryption with key #574897</td>
<td></td>
</tr>
<tr>
<td><strong>Network addresses</strong></td>
<td>IP Address 16.4.0.32</td>
<td></td>
</tr>
<tr>
<td><strong>Compound</strong></td>
<td>Principals in roles</td>
<td>Abadi as Manager</td>
</tr>
<tr>
<td><strong>Delegations</strong></td>
<td>BurrowsWS for Wobber</td>
<td></td>
</tr>
<tr>
<td><strong>Conjunctions</strong></td>
<td>Lampson ∧ Wobber</td>
<td></td>
</tr>
</tbody>
</table>
A single channel can give rise to many:
IP address 16.4.0.32 may carry UDP channels to ports 2, 75 and 443
The name /com/dec/src multiplexes /com/dec/src/burrows, /com/dec/src/lampson, etc.
A subchannel is no more secure than the channel on which it is multiplexed
The (small amount of) hardware and software that security depends on

The rest of the system can misbehave without affecting security

Note that denial of service is not addressed: For access to be granted, encrypted statements may be needed from a database — if the database is down, access may be denied; yet, the database is not part of the TCB

With the right organization, only the checking algorithm, the encryption mechanisms and the keys are in the TCB
Your TCB

Everyone has a different TCB
A user’s TCB is usually a superset of the system’s
Many system services (such as the file system) belong to your TCB
Machines and operating systems on which you run your programs belong to your TCB. Your terminal belongs to your TCB
For machines, the rule is “Physical access is total control”
$A$ speaks for $B$ when the fact that $A$ makes a statement means we can believe that $B$ makes the same statement.

The channel from a terminal speaks for the person at the terminal.

A person speaks for the group of which he is a member.
• Primitive: “read file /etc/passwd”
• If $s$ and $s'$ are statements, then $s \land s'$, $s \supset s'$ and $s \equiv s'$ are statements
• If $A$ is a principal and $s$ is a statement, then $A$ says $s$ is a statement
• If $A$ and $B$ are principals, then $A \Rightarrow B$ ($A$ speaks for $B$) is a statement
When transmitted between programs, statements must be encoded in messages.

A Kerberos ticket is encoded as $E(K_{bs}, (T_s, K_{ab}, A))$.

The statement carried is $K_{bs}$ says $K_{ab} \Rightarrow A$
‘s is an axiom’ is denoted by ⊢ s

1. If s is an instance of a theorem of propositional logic, then
   ⊢ s, for example, ⊢ s ∧ s′ ⊃ s
2. If ⊢ s and ⊢ s ⊃ s′, then ⊢ s′
3. ⊢ (A says s ∧ A says (s ⊃ s′)) ⊃ A says s′
4. ∀ principal A: if ⊢ s then ⊢ A says s
5. ⊢ A says (s ∧ s′) ≡ (A says s) ∧ (A says s′) (follows from 1 and 4)
A says \( s \) does not mean that \( A \) actually makes the statement \( s \).

It means that we can act as if \( A \) has done so.

We say that \( A \) makes the statement \( s \) when \( A \) actually does something to let another principal infer \( A \) says \( s \).

For instance, \( A \) can make \( A \) says \( s \) by sending \( s \) on a channel that speaks for \( A \).
1. $A$ and $B$ say something when both say it
   $\vdash (A \land B) \text{ says } s \equiv (A \text{ says } s) \land (B \text{ says } s)$

2. $A$ quoting $B$ as saying $s$ (but $A$ may be lying)
   $\vdash (A | B) \text{ says } s \equiv A \text{ says } B \text{ says } s$

3. Definition of speaks for:
   $\vdash (A \Rightarrow B) \equiv (A = A \land B)$

4. $\vdash (A \Rightarrow B) \supset ((A \text{ says } s) \supset (B \text{ says } s))$

5. $\vdash (A = B) \equiv ((A \Rightarrow B) \land (B \Rightarrow A))$
1. **handoff axiom**: 
   \[ \vdash (A \text{ says } (B \Rightarrow A)) \supset (B \Rightarrow A) \]

2. **certification authority** for A: 
   \[ \vdash ((A' \Rightarrow A) \land A' \text{ says } (B \Rightarrow A)) \supset (B \Rightarrow A) \]

3. **joint authority**: 
   \[ \vdash ((A' \land B \Rightarrow A) \land (B \Rightarrow A')) \supset (B \Rightarrow A) \]

A proof that \( B \Rightarrow A \) is called B’s **credentials** for A
Statements from channels \((C \text{ says } s)\) can be taken as assumptions

A message arriving from IP address 130.89.181.105 is only interesting if we can deduce who sent it

If we know the possible senders on channel \(C\) then we say that \(C\) has integrity

If we know the possible receivers on channel \(C\) then we say that \(C\) has secrecy

Knowing the sender on \(C\) implies we can find an \(A\) such that \(C \Rightarrow A\)
How can we know that a channel $C \Rightarrow A$?

Because $A$ or a principal $A'$ speaking for $A$ tells us so

The handoff rule: $\vdash (A \text{ says } (C \Rightarrow A)) \supset (C \Rightarrow A)$

and the certification-authority rule $\vdash ((A' \Rightarrow A) \land A' \text{ says } (C \Rightarrow A)) \supset (C \Rightarrow A)$

are used to deduce that $C \Rightarrow A$
A sender on a channel $C$ can *make* $C$ *says* $X$ *says* $s$, for any $X$

This is the same as $C|X$ *says* $s$

If $C$ names the channel, $C|X$ names the subchannel

Note that, before you believe $X$ *says* $s$, you need proof that $C \Rightarrow X$
An encryption channel with the RSA public key $K$ will be denoted by RSA($K$), or simply by $K$

The channel speaks for the principal that knows $K^{-1}$

A public-key channel is a broadcast channel:

- A message on a public-key channel can be stored in a file
- You can forward a message to somebody else and the receiver will have the same assurance of its source as you
A statement signed with a secret key can be stored away and passed around without losing its meaning:
From the existence of $E(K^{-1}, s)$, anyone who knows $K$ can deduce that $K$ says $s$

We call a statement of the form $E(K^{-1}, s)$ a certificate, and we say that $K$ signs the certificate

Certificates must have a lifetime to be practical, so most certificates will say things like $K$ says $(T_i < t < T_e) \supset s$ where $t$ is the current time, $T_i$ the time of issue and $T_e$ the expiration time of the certificate
In order to establish authenticated communication or verify authority to carry out a request, the collection of certificates to make the necessary deductions must be gathered. They can be collected in two ways:

**Push**: The sender collects the credentials and transmits them with a request.

**Pull**: The receiver gets a request and consults a database to find the credentials.

Most protocols use pushing. The system presented in Chapter 21 uses pulling.
There is one certification authority $CA$
Everyone trusts $CA$: $\forall$ named $A : CA \Rightarrow A$
Everyone knows $CA$’s public key $K_{ca}$: $K_{ca} \Rightarrow CA$

From this, anybody can deduce $K_{ca} \Rightarrow A$ for every named $A$
For each named principal $A$ that $CA$ speaks for, $CA$ issues a certificate that says $K_{ca}$ says $K_a \Rightarrow A$
These certificates are stored in a database and are indexed by name
Authenticating Channels

\[ CA \text{ knows } K_{ca}^{-1}, \quad K_a \Rightarrow A, \quad K_b \Rightarrow B \]

\[ A \text{ knows } K_{a}^{-1}, \quad K_{ca} \Rightarrow CA, \quad CA \Rightarrow \text{Anybody} \]

\[ B \text{ knows } K_{b}^{-1}, \quad K_{ca} \Rightarrow CA, \quad CA \Rightarrow \text{Anybody} \]

\[ \text{Certificates} \quad K_{ca} \text{ says } K_{b} \Rightarrow B \]

\[ \quad K_{ca} \text{ says } K_{a} \Rightarrow A \]

\[ A \text{ learns } CA \text{ says } K_{b} \Rightarrow B, \quad K_{b} \Rightarrow B \]

\[ B \text{ learns } CA \text{ says } K_{a} \Rightarrow A, \quad K_{a} \Rightarrow A \]
Since certificates can be stored on files, it is not necessary for the certification authority to be on line.

The certification server can be a laptop computer that is kept in a safe.

Certificates can be carried to the system on floppy disks.

Thus, no security holes or operator stupidities can make $K_{ca}^{-1}$ public.
If certificates can be stored anywhere, you cannot *revoke* them by deleting them or not issuing them anymore. Short expiration times make an on-line certification server necessary. To get fast revocation with the security advantage of an off-line certification service, you add an on-line *revocation service*. The revocation server has much weaker authority but can be used for fast revocation by using short expiration times.
Let the certification authority be $CA$ and the revocation authority be $RA$

Then, instead of making certificates of the form $K_\text{ca} \text{ says } K_a \Rightarrow A$,

$CA$ makes a certificate that says

$K_\text{ca} \text{ says } (RA|K_a \land K_a) \Rightarrow A$ with a long expiration time

$RA$ countersigns by issuing $RA|K_a \text{ says } K_a \Rightarrow RA|K_a$ with a short expiration time

The TCB for access is just $CA$, that for revocation is both $CA$ and $RA$
Changing Keys

This should be done periodically.
A principal chooses a new key and tell the CA its new public key.
The CA issues a new certificate and install it.
Old and new keys must co-exist for a while.
The old key automatically expires as all extant certificates expire.
For auditing purposes it is useful to keep a log of all past certificates.
Multiple Authorities

![Diagram of Multiple Authorities]

- University
  - Bologna
    - Ozalp
    - Sape
  - Twente
  - Tromsø
    - Tore
    - Terje

- Industry
  - DEC
    - SRC
    - Burrows
  - IBM
    - Abadi
• Tage trusts /University to authenticate Sape
• Tage does not trust /University to authenticate Otto

Certification via shortest path, through lowest common ancestor

• An authentication server lets its parent authentication server *speak for* anything except itself
• An authentication server lets its child authentication servers *speak for* everything below them
We formalize this through a new compound principal constructed with the `except` operator

- $\vdash P \text{ except } M \Rightarrow P$
- $\vdash M \neq N \supset (P \text{ except } M)|N \Rightarrow P/N \text{ except } '..$
  /University/Tromsø except Tage quoting Otto can speak for /University/Tromsø/Otto except '..', so that /University/Tromsø/Otto cannot speak for /University/Twente/Otto
- $\vdash M \neq '..\supset (P/N \text{ except } M)| '..\Rightarrow P \text{ except } N$
  /University/Tromsø except Tage quoting '..' can speak for /University except Tromsø so that /University cannot speak for /University/Tromsø
Example: Tage Authenticates Sape

Tage trusts his personal CA, /University/Tromsø/Tage, and knows a channel $C_{Tage} \Rightarrow /University/Tromsø/Tage$ except $\emptyset$

The following series of statements is then generated by the CAs:

1. $C_{Tage} | '..' \text{ says } C_{Tromsø} \Rightarrow /University/Tromsø$ except $Tage$
2. $C_{Tromsø} | '..' \text{ says } C_{University} \Rightarrow /University$ except $Tromsø$
3. $C_{University} | Twente \text{ says } C_{Twente} \Rightarrow /University/Twente$ except '..'
4. $C_{Twente} | Sape \text{ says } C_{Sape} \Rightarrow /University/Twente/Sape$ except '..'
Initial assumptions:
\( \vdash \text{/University/Tromsø/Tage} \Rightarrow \text{Anyone} \)
\( \vdash C_{Tage} \Rightarrow \text{/University/Tromsø/Tage except } \emptyset \)

We see the statement
\( C_{Tage}\mid \text{..'} \text{ says } C_{Tromsø} \Rightarrow \text{/University/Tromsø except } \text{Tage} \)

We deduce
\( C_{Tromsø} \Rightarrow \text{/University/Tromsø except } \text{Tage} \)

and from the definition of \text{except}
\( C_{Tromsø} \Rightarrow \text{/University/Tromsø} \)
We can add special authentication paths to avoid less trusted CAs
For example, we can avoid the /University CA and create a cross link from node 17 to node 48 so that /University/Tromsø can directly authenticate /University/Tromsø/Twente
We do this by adding the certificate $C_{Tromsø}$ says $C_{Twente} \Rightarrow /University/Tromsø/Twente$ except ‘.’
If the topology of the tree changes, or if names in the authentication tree change (e.g., we switch from ASCII to UNICODE so that Tromsoe can become Tromsø), all certificates will have to be re-issued.

To avoid this, we give each authentication node a unique identifier and, instead of certificates of the form $C \Rightarrow P$ we create pairs of certificates $C \Rightarrow id$ and $id \Rightarrow P$.

The sender *pushes* $C \Rightarrow id$, the receiver pulls $id \Rightarrow P$. 

**Tree Topology Changes**
Groups have no public keys or channels of their own

Member principals speak for the group

Looking up a group name $G$ yields a set of certificates:

$K_{ca}$ says $P_1 \Rightarrow G$, $K_{ca}$ says $P_2 \Rightarrow G$, ..., $K_{ca}$ says $P_n \Rightarrow G$

With $K_{ca} \Rightarrow G$, we can deduce $P_i \Rightarrow G$

Groups are useful for putting on ACLs
A Trojan horse is an enemy masquerading as a friend
The PC world has been plagued by them
How can you limit the destructive capabilities of a program you don’t trust?
In conventional systems, when a principal $P$ runs a program $X$, we essentially get $X \Rightarrow P$
We need something like $X \Rightarrow P'$ where $P'$ is a lot less powerful than $P$
For this, we introduce roles: $P$ as GamePlayer
If $A$ is a principal and $R$ is a role, we write $A \text{ as } R$ for $A$ acting in role $R$

A role *limits* authority, hence $A \text{ as } R$ is less powerful than $A$

We use quoting to define *as*

- $\text{Roles}$ is a subset of the simple principals
- $\vdash \forall R \in \text{Roles} : A \text{ as } R = A|R$
- $\vdash \forall R \in \text{Roles} : A \Rightarrow A \text{ as } R$

Thus, $A \text{ as } R \text{ says } s$ is the same as $A \text{ says } R \text{ says } s$
Roles and programs are similar

If $A$ executes a program whose image is $I$, the program can be thought of as $A \text{ as } I$

Node $N$ will $\text{make } N \text{ as } I \text{ says } s$ for a statement $s$ made by a process executing $I$ in $N$

Program binaries are not suitable to be used as names, so, instead, we use a digest $D$ of the program $I$ in its place

Node $N$ will then make $N \text{ as } D \text{ says } s$ instead

The digest speaks for the role. For a named program $P$, we can introduce $D \Rightarrow P$ to indicate that the image whose digest is $D$ implements the program $P$
Node $N$ executes $B$’s request to run a program $P$
If $N$ trusts the file system, it fetches $P$ and starts it up in a process $pr$
It hands off to $pr$ the right to speak for $B$ as $P$
If $N$ does not trust the file system, it computes the digest $D$ of the program it loads and tries to find credentials for $D \Rightarrow P$
We can create a name for the role *trusted software*, e.g.,
/University/Twente/TSW

Anyone who trusts the node $N$ might put $N$ as /University/Twente/TSW
on the ACL of its objects

If $P$ is a trusted piece of software, the CA signs a certificate

\[ K_{ca} \textbf{ says } P \Rightarrow /University/Twente/TSW \]

When $P$ is run on $N$, $N$ will allow it to speak for $N as P$

Since we believe $P \Rightarrow /University/Twente/TSW$, $N$ as $P$ says $s$
implies $N$ as /University/Twente/TSW says $s$

\[(N \textbf{ as } P \textbf{ says } s \supset N|P \textbf{ says } s \supset N \textbf{ says } P \textbf{ says } s)\]
When a machine $M$ is booted with program $P$, the result will be a node that can speak for $M$ as $P$

A machine can only grant this if it first learns to ‘speak for itself’

On installation, a machine is given a private key $K_{m}^{-1}$ in its non-volatile memory; the bootstrap ROM uses this when it loads an operating system

The system administrator then gets the CA to sign $K_{ca}$ says $K_{m} \Rightarrow M$

When $M$ is booted up with an operating system image $P$, all of its resources come under the control of the process $N$ executing $P$ — $M$ is no longer around to sign things

$M$ constructs a new key pair $\{K_{n}^{-1}, K_{n}\}$, gives $K_{n}^{-1}$ to $N$ and signs $K_{m}$ says $K_{n} \Rightarrow M$ as $P$
We don’t want to bootstrap just any program as operating system

Today’s network boot protocols are not trustworthy

However, if $M$ signs $K_m \text{ says } K_n \Rightarrow M \text{ as } D$, where $D$ is the digest of the program loaded, the operating can’t do any harm outside $M$ if it turns out that the program was a Trojan horse — there won’t be a certificate $D \Rightarrow P$ where $P$ is a legitimate program

But the program $P$ does, of course take over $M$’s resources, such as $M$’s local disks and $M$’s keyboard and screen

Preventing the wrong program from being booted can be done by giving $M$ a set of legitimate digests in non-volatile memory
A principal can hand its authority off to another: *A says B ⇒ A*

A principal can limit its authority: *A as R*

We combine these to allow a principal to hand off *part* of its authority, *B acting on behalf of A*, or *B for A*

- ⊢ A ∨ B|A ⇒ B for A
- ⊢ for is monotonic and distributes over ∨
1. A delegates to B: A says $B|A \Rightarrow B$ for $A$
2. B accepts: $B|A$ says $B|A \Rightarrow B$ for $A$

We can deduce that $B|A \Rightarrow B$ for $A$, or B can speak for $B$ for $A$ by quoting $A$

Timeouts are used to let delegations expire
User $U$ logs in to workstation $W$

Complication is that $K_u$ is only available at login time

Solution is to generate a session key $K_l$ to which the user’s authority is delegated for a limited period and that is discarded on logout

- $U$ delegates: $K_u$ says $(K_w \land K_l)|K_u \Rightarrow K_w$ for $K_u$
- $W$ accepts, so $(K_w \land K_l)|K_u \Rightarrow K_w$ for $K_u$
- $|$ distributes over $\land$: $K_w|K_u \land K_l|K_u \Rightarrow K_w$ for $K_u$
- $K_l$ signs a short-term certificate $K_l$ says $K_w \Rightarrow K_l$
- Therefore $K_w|K_u \Rightarrow K_l|K_u$
- and, with a short lifetime, $K_w|K_u \Rightarrow K_w$ for $K_u$
**Putting it All Together**

**WS as Nemesis ⇒ UTnode**

\[ K_{Sape} ⇒ Sape \quad K_{WS} ⇒ WS \]
The DBase process speaks on the channel $K_n|pr$ or $C|pr$

Assume $pr$ issues a request to read file foo; $C|pr$ is the channel carrying the request. This channel speaks for $K_{ws}$ as Nemesis as DBase for Sape

The credentials of $C|pr$ are:

1. $K_{ws}$ says $K_n \Rightarrow K_n$ as Nemesis 
   from booting
2. $K_{Sape}$ says $(K_n \land K_l)|K_{Sape} \Rightarrow K_n$ for $K_{Sape}$
   from login
3. $K_l$ says $K_n \Rightarrow K_l$
   from login
4. $K_n|K_{Sape}$ says $C|pr \Rightarrow ((K_{ws}$ as Nemesis) as DBase) for $K_{Sape}$
   sent on $C|K_{Sape}$
5. $WS$ as Nemesis $\Rightarrow$ UTnode
   from database
In the operating system is a trusted agent that operates the authentication and credentials machinery.

The agent acts on behalf of the operating system and its user and system processes — since the OS is part of any process’ TCB that runs on it, this is not a restriction in security.

The agent maintains a database of credentials and caches credentials obtained from elsewhere.
The Agent’s State

- **Key pair** — of the node $K_n, K_n^{-1}$
- **Principal Cache** — maps channels $C_a = C_s | pr_s | a$ to principal $A$
  An entry is made when proof has been given that $C_a \Rightarrow A$; entries expire
- **Authorities** — A table mapping an agent $a$ to a principal $A$ it speaks for, the credentials that prove this, the set of processes that speak for $a$ and the set of channels $C_b$ that speak for $a$
The Agent’s Interface

Self(): A
Discard(a:A)
And(a:A, b:A): A \quad a \land b \text{ says } \text{result} \Rightarrow a \land b
As(a:A, r:R): A \quad a | r \text{ says } \text{result} \Rightarrow a \text{ as } r
Handoff(a:A, b:C) \quad a \text{ says } b \Rightarrow a
Claim(a:C, b:A): A \quad \text{Retrieve } a \text{ says } b \Rightarrow a; b \text{ says } \text{result} \Rightarrow a
For(a:A, b:C) \quad a \text{ says } b | a \Rightarrow b \text{ for } a
Accept(a:C, b:A): A \quad \text{Retrieve } a \text{ says } b | a \Rightarrow b \text{ for } a; b | a \text{ says } b | a \Rightarrow b \text{ for } a \land \text{result} \Rightarrow b | a
CheckAccess(acl:ACL, b:C, op:O): boolean
\quad \text{Does } acl \text{ grant } b \text{ the right to do } op?
Necessary if you want a secure system. You must be able to detect attempts to breach security. You must be able to detect compromised keys.

The theory gives proofs for every control decision. These can be stored in a log (called the *audit trail*)