Authentication in distributed systems: theory and practice

CS86D; Mike Dahlin
April 3, 2002

1 Preliminaries

1.1 Review

1.2 Outline

1.3 Preview

2 Motivation

2.1 Theme

- Lots of interlocking subsystems
  - channel, machine, server, secure boot, ...
  - All have implications on security and trust one another to do certain things

- Current approach

  Expert makes sure they all fit together; probably no written spec of how different assumptions/keys interlock

  (Actually, in 1992 when this paper is written, a bunch of the pieces were missing and the paper described how to build them.
  Today, we have many of the pieces, but we mostly hope they fit together.)

- Proposed approach

  A single framework for reasoning about all subsystems

  - make sure all things fit together (or the proofs won’t go through)
  - full articulating information

2.2 Case for fine-grained access control

- Today (at best) e.g., coarse-grained access control (e.g., ssh or keycars) log in and give my full authority to make connections as “me” using my key.

  Why add the complexity to delegate the authority for another key to “speak for” me?

- Goal: More complex delegations

  Restricted roles, longer (and/or explicit) delegation chains

  Mobile code (downloaded from net, run locally) → run with less than my full authority (e.g., “can read from/write to market windows on screen and can access files in /tmp/PID/*”)

  Untrusted devices think of world with 110 computing devices.

  Does your $9.95 connected watch that you bought from a street vendor have the same rights as your $2000 Dell desktop? E.g., your watch should be able to read and write your calendar, but it can’t talk to anything else (privacy) and it can’t read or write other files.

  Telecommuting should someone’s home PC have all the rights that someone’s office PC has?

  Mobile code agents: if you send an agent into the network, should you send your private key with it?

  Ubiquitous computing e.g., “embedded PC” walk into airport lounge and start using an assembled computer: keyboard, mouse, Sanyo video display in the lounge + the disk cache provided by your ISP + the updates you’ve entered into your Palm.

  What principal is issuing commands? Does this thing have the same rights as your desktop PC in your office?

  Smart card log in system will not have your private key ever always add one more level of indirection

  Will these types of things (especially in combination) lead us to commonly having more complex delegation patterns?
• Question: Are these practical problems or just academic
  
  Task is complex
  Arguably, we survive w/o this complexity today
  Arguably, more need for this complexity as environments get more complex?
  Can we structure tasks so that things are simple the vast majority of the time, but we can express more fine-grained controls when we want to do so?

• Question: End-to-end v. compound principles + delegation
  (Or maybe the continuum is "little demultiplexing" v. "lots of multiplexing"

  End-to-end: requests by A signed by KA
  Delegation: requests by A signed by something else + chain of delegation

  User-level stuff uh, hehems roughly end-to-end
  * More precisely, the delegation is "simple enough" that the delegation is easy to reason about implicitly, e.g., session key for each user/server interaction.

IPSec more similar to Taos

Section 8.4 of paper discuss trade-offs (argues for lots of multiplexing)

* "When acting for a sender, the agent has to respond to asynchronous calls from receivers. Although the sending process could expect the agent interface, this is a lot of machinery to have in every process."

* "An agent in the operating system can optimize the common case of authentication between two processes on the same node. This is especially important for handing off an authority from a parent to a child process which is very common in Unix. All the agent has to do is check that the parent speaks for a and add the child to the set of processes that speak for a. This can be implemented almost exactly like the standard Unix mechanism for handing off a file descriptor from a parent to a child."

• "The agent must deal with encryption keys, and cryptographic religion says that key handling should be localized as much as possible. Of course we could have put just this service in the operating system, at some cost in complexity."

• "Process to process encryption channels mean many more keys to establish and keep track of."

• "The operating system must be trusted anyway, so we are not missing a chance to reduce the size of the trusted computing base."

• Bottom line: is this a solution looking for a problem or a paper ahead of its time?

3 Admin

Midterm next Wednesday
Next Monday office hours 4:30-5:30 here or in office?

4 Major components

• Principal

• Channel

• Naming hierarchy for CA
  
  Online and offline agent
  
  Upward and downward delegation

• Compound principles

  - joint authority useful to have a long-lived signature + periodic short term renewals
  
  - delegation useful to have someone else "speak for" you
  
  - role useful to grant subset of my rights

• Their protocols

  - boot a machine transfer rights from hardware to OS
5 Principal and naming

(See section 5)

- 2 differences with SDSI
  
  They seem similar, but SDSI seemed simpler. Help me nail down exactly what the differences are. When is fundamental, what (if anything) can Taos do that SDSI can’t (e.g., where exactly did they add complexity and what did they get for it?)

- Names appear more fundamental “when users refer to principals, they must do so by names that make sense to people, since users can’t understand alternatives like unique host files or keys.”
  
  V, w, z: only keys make sense to computers

- Hierarchical global namespace v, SDSI local namespaces
  
  What are principals?

- Hierarchical global namespace v, SDSI local namespaces
  
  In both Taos and SDSI, “anyone can be a CA” and can delegate “down”
  
  - In Taos, $K_A$ can say “$K_B$ speaks for (any path) $A/B$ (except ‘)”
  
  - In SDSI, $K_A$ can say “my local name for $K_B$ is ‘B’

- SDSI statement seems creepier, Taos statement is poser: “$C_A$ says that $CA_B$ can be a CA under me, SDSI says “anyone can be a CA (duh, I can’t stop them)” Note that the bytes in the certificate are not really the same, just the concept utilized of them (I think.)

- In Taos, a node can also delegate “up”
  
  - “$CA_B$ can delegate for $B$ except host owners”
  
  What’s the difference?

- “Anyone can be a CA” I think this is a crisper philosophy and makes the system easier to describe but I don’t think it changes how the system actually behaves.

- Taos assumes one global root, I think this is superficial, I think all of this works with multiple roots (as long as the roots have different “names”)

- Taos lets A and B agree on a common “root” (or parent)
  
  - Perhaps makes it easier for 2 nodes to agree on the same name for the same key?
  
  - Example SDSI: burrow’s dec’s root’s mill’s clerk v, clerk’s mill’s root’s mill’s clerk
  
  - Taos: root’s mill’s clerk for both
  
  - How can two nodes agree they are talking about the same clerk?

- What can we do (Monday I argued that SDSI doesn’t need “global names” this means that I think that two nodes can still agree that they are talking about the same root (or in fact the same principal)

- Principal (key) is a global name
  
  - Burrows and clerk can figure out that they are using the same root and from that see that they are talking about the same root’s mill’s clerk
  
  - In fact, burrows and clerk can figure out they are using the same clerk directly

- What (if anything) is lost (or is more complex) by establishing commorality of names through (a) pre-agreed roots, (b) principal = global name principle?

- Is there some other use for speaks delegation that I’m missing?

- Other issues

  - The joint authority seems nice,

  - “The TCB for granting access is just CA, because O acting on its own cannot do anything, but CA speaks for A, the TCB for revocation in CA and O, since either one can prevent access from being revoked.”
One more question about "down", what do you really get if a child for its delegation "up"? The parent can just give a different child your name right?

6 Channels

Section 4

- Main complexity seems to come from multiplexing
  
  One encrypted channel per pair of nodes (note: hardware encryption)
  
  A number of users may send requests to a process
  
  A number of processes may run under an OS
      (Virtual machines)
      A number of OS’s may run on a node
  
  Node should run with a temporary key rather than the real HW key
  ...

- Is an alternative: “encrypt all messages on behalf of a user with the user’s key” feasible?
  
  Maybe (sub, well sub negotiates a per-user session key,)
  
  Maybe not what about requests I send through a server process?
  (and other motivations for delegation above?)

- Other questions

  Footnote 13 says “Alternatively, the node could directly authenticate the shared key K by making K晟 examines for K晟. This prevents channel setup from changing K on its own, which is a significant loss of functionality.” What is the problem?

  The last paragraph on page 21 says “In practice, application programs normally use Kerberos to authenticate network connections, which applications then rather unrealistically treat as secure channels.” What’s the problem?

  Point of clarification (hopefully), I think the idea of delegating to “A B” (a quoting B) (e.g., B says A B speaks for B) is that in multiplexing, it is prudent for a multiplexing entity like A to always explicitly say when they are acting on behalf of some particular principle. That is, if both B and C delegate to A, and A says something, you can only assume that this statement has B’s authority if A explicitly states that statement has B’s authority. Of course, A can be malicious or buggy and mix up B’s authority and C’s authority, but this delegation prevents you from implicitly assuming something you shouldn’t assume.

  Another motivation — backwards compatibility. See Fig 7, NFS uses (used) timestamp to get authenticated.

  So, this picture “node for user” really does reflect the trust model.

  • But this is just how broken NFS is under this model, I can’t use my [untrusted] laptop to access my files, Kerberized NFS a much better solution, right? Does this say that we don’t need all of this delegation?

7 Compound principals

8 Secure boot

A nice subproblem...

Difficult to have security if you don’t trust machines, But we know that once someone gains control of a machine they can become root, change all the binaries, and you are doomed (until you update the hard disk and format it on a unaffected machine.) See “Reflections on trusting trust.”

Suppose you only worry about software failures (e.g., the adversary will not change your hardware). Then we can ensure that when we reboot the machine it is no longer compromised,

- Solution part 1: generate a new OS key that speaks for machine key K晟 at each reboot. Ensure that software cannot get at key between reboots (requires HW support) put K晟 in boot ROM and ensure that boot ROM can’t be read after “Done booting” (e.g., this bit cannot be cleared except by reboot)

- Solution part 2: ensure that kernel code cannot be modified by software

  Option 1: put it on a separate disk. Set the jumpers for that disk to “read only.”
Option 2: Store hash of good kernel (or store public key of trusted "signer of good kernels") in PROM, Boot sequence verifies hash before transferring control to OS.

- Roughly the above was done in Athena, give all MIT CS students "root" on all CS dept machines; reboot between users.
- What if user can open the box?
  - In the limit, all bets are off...
  - In practice, there may be a lot you can do smarter card instead of PROM ...

9 Criticisms/future work

- Bottom line: is this a solution looking for a problem or a paper ahead of its time?

- Organization: What are components and interfaces among components? Separate the specific implementations where possible. (E.g., we should be able to replace the strange TARS channel setup with IPsec, Kerberos or SSH, right?)

- Simplicity: do we really want to know "machine for user X" or is it enough to say "user X" with the implicit assumption that ("user trusts any machine he gives key to")? The latter is roughly what we do today. Will we want to do more in the future (with web terminals, $10 abqu computing devices, etc? Even if we want to do more than today, is it just too complex to try to do so?

- If I use Tars or something like it, can I write simple applications that make coarse-grained statements about security and only get into more complex, fine-grained distinctions if I care a lot, (E.g., suppose I want to make the assumption that any machine a user uses is trusted by that user. Does that simplify the protocols? Assuming false default, such an assumption needs to be explicit. How do I make that explicit assumption?)

- Research question: does any of this help you in a world like the web where a user accesses thousands of mutually distrustful servers?