6.826 Principles of Computer Systems — Fall Semester, 1993

Lecture 22: November 29, 1993

Lecturer: Butler Lampson — *Scribe: Kevin Lew*

1 Handouts

- Handout 51, "Authentication in Distributed Systems: Theory and Practice" by Butler Lampson, *et al.*
- Handout 52, "Why Cryptosystems Fail" by Ross Anderson

2 Overview

- Finish configuration control
- Security
- Information Flow Model
- Access Control Model
- Principals
- Security Firewalls

3 Configuration Control

The last lecture ended with a discussion of the following issues relevant to computer system configuration:

1. naming and access
2. installation and repair

We now finish the discussion on installation and repair, then move on to discuss the following:

3. replication
4. load balancing

3.1 Installation and Repair

Dependencies and keeping track of state are relevant to installation and repair. Dependencies are important and complex. A *dependency* is a requirement that one component must work in order for another component to work. For example, $\TeX$ requires that a lot of other files are available in order to produce a $\TeX$ document. Because a computer system has dependencies, installation and repair of the system may mess up dependencies.

There are two ways to keep track of system state. One way is to store the ideal configuration in a database, and to have a process enforce this ideal configuration. A second way is to have a process determine the system state and report it.

3.2 Replication

Replication involves having a fault-tolerant system, backups, and distribution of replicas. When a computer system uses replication, it is fault-tolerant. However, a disadvantage is that configuring the system is more complicated because the system now has more components. Because replication involves making backups, we must worry about keeping all backups consistent, and we must worry about recovery and replication time. Finally, replication requires that we distribute the replicas somehow. One way to do this is to make changes to a master copy, then distribute this master copy to other sites. This is what is done with the UNIX `/etc/passwd` file. Because changes propagate to slave copies of the password file, this replication scheme is loosely consistent: a user may see different versions of the password file on different machines if a set of changes has not finished propagating to all machines. As another example, some distribution scheme is required for a bank's computer system that consists of 25,000 PCs, each of which needs a particular software package installed on it. Distribution of the software package becomes more difficult when each PC requires site-specific customization.

3.3 Load Balancing

Load balancing is the process of scheduling demands on resources in some way. Load balancing involves determining the current load, then handling it gracefully. When people talk about load balancing, they use it to describe scheduling computational tasks on multiple processors and scheduling files on multiple disks. Normally, load balancing across multiple processors is not done because most people run computational tasks on one processor. However, in parallel computing research, load balancing is important because distributing the load evenly across processors minimizes execution time. Scheduling files on multiple disks is a bin packing problem. In general, spreading one file system's files across multiple disks is not a good idea because the consequences of a failure are hard to determine: when a failure occurs, an arbitrarily large portion of a file system may become unavailable.

4 Security

Security provided by computers is only a fraction of the security problem (for an elaboration of this point, see Anderson's paper, Handout 52). For example, a bunch of security experts at AT&T Bell

Labs were hired to provide a better security system at AT&T's headquarters in New York. One night, the experts entered the headquarters and found janitors cleaning the room with all the computers that needed better security. The janitors let the experts in the room, and what's more, the experts found a PostIt with a password taped on one of the computer screens! Thus, the security problem also deals with making sure only authorized people are able to use the computers that need to be kept secure. In general, intruders go after the weakest point in a system.

Security is difficult to provide on a large scale. In addition, most people don't bother to provide security for their computer systems because doing so isn't cost-effective. Normally, people provide less and less security according to some exponential-backoff scheme, and only stop backing off when an intruder breaches their system. Security breaches can be quite serious. For example, a computer virus that erases hard drives can cause massive destruction if it finds its way to many computers.

A *vulnerability* is a weakness in a system. A *threat* is the fact that someone will exploit a vulnerability.

Now for the computer-related stuff about security. First, we need a *security policy*, which is an informal spec. A policy includes the following components:

1. secrecy, also known as non-disclosure and confidentiality;
2. integrity, which means that data in a system is not altered when it shouldn't be;
3. availability, also known as "working on demand" or "preventing the demand of a service".

Once we have a security policy, we must organize our computer system by giving a *model*, which is a precise description of how a system is organized. We will consider two models: the information flow model and the access control model.

5 Information Flow Model

One model is the *information flow model*, also known as the *mandatory security model*. Modeling information flow involves specifying the following set of rules:

1. A partially ordered set C of classifications that always has a maximum "most top secret" level. Such a set is called a *semi-lattice* since a maximum can always be determined. If a partially ordered set does not have a maximum, we can always add one. As an example of a partial order, consider company files. Personnel and medical files should have higher security than company-confidential files.
2. Every datum has a classification.
3. $f(x)$ has a security level that is the maximum of f and x.
4. Every storage cell or communication channel has a label that is in C.
5. Only people authorized to see a computer screen with a certain label can see the screen.

The third rule is hard to enforce because if f is "return 3 no matter what x is," then f should be unclassified.

Most people worry about integrity; they are more concerned with data getting altered when it's not supposed to than the data being made known to someone.

6 Access Control Model

The access control model[1] takes the following view of a computer system:

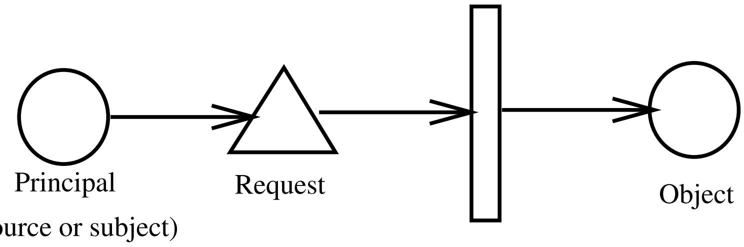

Figure 1: Access Control Model

In this model, a *principal* sends a *request* to access or modify an *object*. Each object has a *guard*, which grants or denies access to the object according to access information the guard maintains about the principal and the object.

In UNIX, access control to files is accomplished by associating permissions with the owner of a file, a group of users, and all other users.

We can represent an access control function by using the following matrix:

Object	**Principal A**	**Principal B**	**Principal C**
File1	none	read	read&write
File2	...	...	...
...	...	...	...

If access control is organized by the columns of such a matrix, then this is access control via *capability lists*. Each principal has a list of objects that can be accessed. Each item in this list is called a *capability*. For example, if the principal is a process and an object is a page, then a capability list would include all those pages in the process' virtual memory space. As another example, UNIX file descriptors also use capability lists. With capability lists, it is cheap to check if a principal is authorized to access a certain object, but it's hard to change a capability because it's hard to find where a capability is stored.

If access control is organized by the rows of the above matrix, then this is access control via *access control lists*. With this scheme, associated with each object is a list of principals that can access the object.

[1] Read Handout 51 for a more thorough discussion of the theory of the access control model. Some of the definitions and statements in this section are taken from that paper.

7 Principals

If a request is sent to a database, the principal may be *one or more* of the following:

1. user
2. application
3. OS
4. hardware

It is possible for the user, the application, or the OS to send bogus requests. How do we determine who or what the principal is (this process is called *authentication*)? NFS uses a login on a workstation in order to determine the principal. But in general, in a distributed system, it's hard to determine the principal. First, we need assurance that a request came from the place we think. Second, we need more details than the information that can be gleaned from a simple centralized system.

We treat a *principal* as the abstraction for a source of a request. We say that principal P *says* statement S. Generally, we want statements to have Boolean values. For example, one statement can be "Butler Lampson's password is `xyz`."

Simple principals include the following:

1. a user
2. a program
3. a machine
4. a communication channel

Why can a communication channel be a principal? Well, the only thing a guard can be sure about a request is that the request came in on an incoming channel. Thus, if a guard receives a message on a channel, then the channels says the message. In addition, a principal can make a statement only by making it on some channel that speaks for that principal.

Given the above simple principals, we build *compound principals*:

- **A** $\wedge$ **B**: This means **A** *and* **B**. For example, "**A** $\wedge$ **B** say **S**" means "**A** says **S** and **B** says **S**."
- **A** $|$ **B**: This means **A** *quoting* **B**. "**A** $|$ **B** says **S**" means "**A** says (**B** says **S**)." This does not necessarily mean that **B** said **S**. **A** can be confused or can be lying.
- **A as R**: This means **A** has *role* **R**.
- **A for B**: This means **A** *acts for* **B**. This is also known as *delegation*; **A** delegates authority to **B**. For example, a workstation acts for a user.
- **A** $\Rightarrow$ **B**: This means, "if **A** says **S**, then **B** says **S**." This is known as the "speaks for" relationship. Using this construct means that **A** is stronger than **B**.

How are these compound principals related to security? Consider the following example. Suppose

1. An access control list says that Lampson can read the object `File1`.

2. A request coming in on terminal line 3 says "Read `File1`."

What we need now is that Terminal 3 speaks for Lampson in order to make any progress.

8 Security Firewalls

We assume that authentication schemes are not foolproof and that we don't have good control over a group of internal machines. How do we deal with security in this case? Well, we can build firewalls that only allow certain types of requests to be passed to a guard. For example, since a lot of workstations in DEC are poorly managed, we can build a firewall that allows only mail to enter or leave DEC. We do not allow any other kinds of requests to enter or leave DEC.

What are some ways to do authentication?

One old way is to hardwire terminals to a central machine. Suppose Terminal A is hardwired to port 4 on the central machine. If a message comes into the central machine on port 4, then we know that the message came from Terminal A. We can also assure to an acceptable degree the "speaks for" relationship, "Terminal A speaks for Lampson," if the door to the office that contains Terminal A is kept locked and only Lampson has the key to the door.

A second way is to use the *Handoff Axiom*, which says, "If **A** says **B** $\Rightarrow$ **A**, then **B** $\Rightarrow$ **A**." This means that every principal **A** has the authority to allow another principal **B** to speak for **A**. For example, suppose you live in a condominium complex, and there's a meeting you can't make, and at this meeting owners are going to vote on which color to paint the laundry room. You decide to let your neighbor vote for you. What you've done is to let your neighbor speak for you. That is, in the handoff axiom, **A** = you, and **B** = your neighbor.

A third way is to use passwords. A password says that channel $\Rightarrow$ Lampson. However, when using passwords, we need a decoding scheme to unambiguously parse messages containing a password, a channel identifier, and a user.

A rule for system security is to "write down what you mean." In other words, one should write down what messages mean and check that each message is what one wanted to say.

How are simple and compound principals used to explain how authentication by using passwords works? Consider the following statement:

If **A'** $\Rightarrow$ **A** $\wedge$ **A'** says **B** $\Rightarrow$ **A**, then **B** $\Rightarrow$ **A**.

We prove this statement by using two things: **A** says **B** $\Rightarrow$ **A** and the handoff axiom.

The "if" part of the statement states that **A'** speaks for **A**, and **A'** says **B** $\Rightarrow$ **A**. But by the definition of the "speaks for" relationship, **A** can also say **B** $\Rightarrow$ **A** since **A'** speaks for **A**. Now by applying the handoff axiom to "**A** says **B** $\Rightarrow$ **A**," we can conclude that **B** $\Rightarrow$ **A**.

Let **A'** = password, **B** = channel, and **A** = Lampson.

Then the above statement says that if the password speaks for Lampson, and the password says that the channel speaks for the Lampson, then the channel speaks for Lampson. In UNIX, establishing the "password speaks for user A" involves asking a user for a password, then verifying the password by looking into the `/etc/passwd` file.