1 Overview

In the last lecture we discussed Access Control Lists (represented column by column) and Capabilities (represented row by row).

In this lecture we discussed the Confused Deputy Problem and implementations of security levels vs. integrity levels.

2 Access Control Systems

Very few systems use Access Control Systems directly, because they are very sparse. Commonly, we represent these more economically by accessing list capabilities. Each capability represents all the objects a subject can access. Therefore, there is an additional requirement on the capabilities: the user can't edit the capabilities.

There are three common ways to make capabilities non-forgeable.

• Cryptography
  Encryption of the rights mechanism, using Signature or Message Authentication. Example: If you wish to send a link to friend to view photos and they don’t have an account, the private key must contain a path to the photo and a symmetric key (MAC) that only the server can generate.

• Programming Languages
  Java can make objects to represent capabilities, since you can’t forge objects in Java (or C++). Only system can create capabilities.

• Rights-restricted file system
  Store the rights mechanism in a control block that require privileges to access.

Are there any advantages of capabilities over access control lists?
- Lets you change all the permissions of a user at once (but conversely, NOT all the permissions of the object). As another example, photo sharing is impossible for access control lists if the friend doesn’t have an account.

3 The "Confused Deputy" Problem

In the 1960s, when we had mainframes (multiple user systems), computing was very expensive. Because of this, users were charged based on the amount of CPU time used. On a particular system, a compiler program must write to two files – the billing file, and the executable file. How do we implement access control on these two files?
The billing file is writeable ONLY by the compiler. The compiler should also be able to write its output into any object file. When the user invokes the compiler, they give it input and output file names. The compiler then writes to the billing file and the output file.

This opens the system up to attack: a malicious user could use the location of the billing file as one of the output file names. This causes the compiler to write the compilation output to the billing file, which renders it unusable.

The compiler could attempt to maintain the integrity of the billing file by storing its path location. However, this strategy runs into the Canonicalization problem -- there are multiple paths to the same file, and it is difficult to determine whether two paths lead to the same file.

The solution is to use capabilities, which bundle together the designation of the file and the permission to access it.

Consider the following two commands:

```
cp a b cat <a >b
```

These two different commands accomplish the same thing: copying the contents of file a into file b. But the permissions involved are completely different. A user may have the capability to modify any file on the system, even the kernel image. Thus, cp needs permission to open any file. On the other hand, cat does not open any files. The user must read a and write to b, and cat merely receives and outputs the file information. So in effect, cp needs permission for everything and cat needs permission for nothing.

We will use this idea to solve the Confused Deputy Problem. Instead of letting the user supply the file name, we let the user open the file and give the file descriptor -- just like cat. Thus, with capabilities, you can solve the confused deputy problem.

4 More on Capabilities

4.1 Ambient Capability

Reconsider the Confused Deputy Problem. Imagine that operations derive the capabilities of the invoking user. When the program invokes the operation, the system checks all of the program's capabilities to see if it is allowed. This is insecure because we inevitably give programs more capabilities than they need, in order to guarantee successful execution. Additionally, this still does not solve the Confused Deputy problem, as the user can still give the program the name of the billing file and access it through the program's capabilities.

This implementation of security is known as "Ambient Capability". This is analogous to a custodian having a ring of RFID keys. Each key is intended to open a different door. When the custodian walks to the door, he doesn't have to pick which key to use, since the door will automatically open as long as one of the keys matches. This is not safe, and does not avoid the Confused Deputy problem.

4.2 Photo Example

Recall our example of sharing a link to a photo. Whoever has the link can access the photo. So the link itself serves as an identifier of the object. But it also serves as an authorization to access the file. In this way, the designation and the authority are hand in hand. You cannot designate a file without the authority to access it, or vice versa. This makes designing capability systems easier, since the authority is embedded in the naming of the object.
4.3 Android Example

Consider mobile operating systems. In android, for example, you can send messages. The messages can be interlapsed, and used for performing operations between different components, functions, and apps. Additionally, android apps have different privileges. When developers are not careful, they open up certain interfaces intended for inter-app components to everyone. This allows less privileged apps to access these components by sending messages to the insecure app. Imagine that malicious app M asks app A to perform some operation. M is not allowed to perform this operation, but the system will execute it anyway because A requested it.

5 Access Control Models

At the beginning of the last lecture, we discussed an access control model where the owner can determine who can access the file. We call this type of access control "Discretionary Access Control" (DAC). This is the most commonly used model of access control. In some settings, however, this is not sufficient. For example, Edward Snowden was given access to secret military information, so he was able to it and share it with the world.

Need-to-know is one method that attempts to solve this problem. In these systems, the system determines who can access what according to its own rules (usually access levels) – NOT the owner of the resources. This is called "Mandatory Access Control" (MAC).

5.1 Bell Lapadula Model - Security guarantee

We define a set of security levels. \[ L = \text{UC} < \text{C} < \text{S} < \text{TS} \]. For example, we can define 4 security levels (unclassified, confidential, secret, top-secret). \( L(S) \) defines the Security clearance, and \( L(O) \) defines the Security classification.

1. Subject can read an object iff \( L(S) \geq L(O) \). We should make sure that the security clearance of the subject is no lower than the security classification of the object. This is called "No read up".

2. Subject can write object iff \( L(S) \leq L(O) \). We should make sure that the security clearance of the subject is no higher than the security classification of the object. This is called "No write down".

Imagine that a subject could read an object iff \( L(S) \geq L(O) \). Then you could read a file whose security classification is above you -- so there are no top secret files. Additionally, imagine that a subject could write an object iff \( L(S) \geq L(O) \). Now, a subject with a high security clearance (Snowden!) could write highly sensitive information into an unclassified file. \( L(S) = L(O) \) also cannot work, because then it is impossible for subjects with lower security clearances to communicate with higher ones.

5.2 Biba's model - Integrity guarantee

1. Subject can read object iff \( L(S) \leq L(O) \). We should only read from sources that are no less trustworthy than the subject. This is called "No read down".

2. Subject can write object iff \( L(S) \geq L(O) \). We should make sure that a write operation makes a file no less trustworthy. This is called "No write up".

Note that if \( L(S) \geq L(O) \), we can read from sources that are no MORE trustworthy than you. This means that everyone can launch nuclear missles!
6 Login problem

On a UNIX based system (for example, SELinux), a user must enter a username and password in order to log in to the system. If they want to sign in as a different user, they must open a shell and run login. Normally, running a program gives it the same privileges you have, but login needs permission to check the password for any possible user. Therefore, login needs to be run as its owner (root). This is accomplished by setting the setUID bit. There is a problem with this: once you’ve launched login, since it’s owned by root, you no longer have the permission to kill it. How do we get around this? We will discuss in the next lecture.