26 Protection and Security

The purpose of a protection system is to prevent accidental or intentional misuse of a system.

- Accidents: program mistakenly overwrites the file containing the command interpreter. Nobody else can log in. Problems of this kind are relatively easy to solve (can do things to make the likelihood small).
- Malicious abuse: high school brat breaks password for B of A accounting system and transfers $3 billion to his own account for kicks. Problems of this kind are very hard to completely eliminate (can't leave any loopholes, can't play on probabilities).

There are three aspects to a protection mechanism:

- Authentication: make sure we know who is doing what.
- Authorization determination: must figure out what the user is and isn't allowed to do. Need a simple database for this.
- Access enforcement: must make sure there are no loopholes in the system.

Even the slightest flaw in any of these areas may ruin the whole protection mechanism.

Authentication is most often done with *passwords*. This is a relatively weak form of protection.

- A password is a secret piece of information used to establish the identity of a user.
- Passwords should not be stored in a directly-readable form. One-way transformations should be used.
- Passwords should be relatively long and obscure. Paradox: short passwords are easy to crack, long passwords are easily forgotten and often written down.

Another form of identification: badge or key.

- Does not have to be kept secret.
- Should not be forgeable or copyable.
- Can be stolen, but the owner should know if it is.

Key paradox: key must be cheap to make, hard to duplicate. This means there must be some trick (i.e. secret) that has to be protected.

Once identification is complete, the system must be sure to protect the identity since other parts of the system will rely on it.

Authorization Determination: must indicate who is allowed to do what with what. In general, can be represented as an access matrix with one row per user, one column per file. Each entry indicates the privileges of that user on that object. In practice, a full access matrix would be too bulky, so it gets stored in one of two condensed ways: *access lists* or *capabilities.*

Access Lists: with each file, indicate which users are allowed to perform which operations.

- In the most general form, each file has a list of <user, privilege> pairs.
- It would be tedious to have a separate entry for every user, so they are usually grouped into classes. For example, in Unix there are three classes: self, group, anybody else (nine bits per file).
- Access lists are simple, and are used in almost all file systems.

Capabilities: with each user, indicate which files may be accessed, and in what ways.

- Store a list of <object, privilege> pairs with each user. This is called a *capability list.*
- Typically, capability systems use a different naming arrangement, where the capabilities are the only names of objects. You can't even name objects not referred to in your capability list. This is almost as if there were no root directory in Unix and no "..".

- In access-list systems, the default is usually for everyone to be able to access a file. In capability-based systems, the default is for no-one to be able to access a file unless they have been given a capability. There is no way of even naming an object without a capability.
- Capabilities have sometimes been used in systems that need to be very secure. However, capabilities can make it difficult to share information: nobody can get access to your stuff unless you explicitly give it to them.

Example of a simple capability-based protection scheme: page tables.

Access enforcement: some part of the system must be responsible for enforcing access controls and protecting the authorization and identification information.

- Obviously, this portion of the system must run protected. Thus it should be as small and simple as possible. Example: the portion of the system that sets up memory mapping tables.
- The portion of the system that provides and enforces protection is called the *security kernel*or *trusted computing base*. Most systems, like Unix, let the whole OS run in "God" mode. As a consequence, the systems aren't very secure.
- Paradox: the more powerful the protection mechanism, the larger and more complex must be the security kernel, and hence the more likely it is to have bugs!

In general, protecting a computer system is extremely difficult. There is no completely secure computer system in existence. Some common problems:

- Abuse of valid privileges. On Unix a super-user can do ANYTHING, many people usually know super-user password. Privileges aren't fine grained enough.
- Imposter or Trojan Horse. Leave a program lying around that looks like the login process. When people type passwords, remember them for your owner. Or, give someone a program that steals information, e.g. supposed editor that reads unauthorized files. Example: checking account system that credited fractional cents to the account of its creator.

- Listener. Eavesdrop on terminal wire, or listen in on local network traffic to steal information.
- Spoiler. (Denial of Service) Use up all resources and make system crash. E.g. grab all file space or create zillions of processes.
- Worm or virus: a Trojan Horse or similar program that is also capable of spreading itself from machine to machine.

Three examples of penetration:

- The Tenex page-fault caper: tiger team cracked passwords by aligning on page boundary, measuring time to check validity.
- Sendmail/finger worm: disabled thousands of computers on the internet in Fall of 1988. Two attacks:
  - Sendmail attack: take advantage of "debug" command left enabled to execute code as super-user, setting up Trojan Horses on machine.
  - Fingerd attack: give long name to fingerd, which overflows buffer, modifies stack in clever way, causing particular procedure to be executed with root privilege.
- How the Unix system designer could make an undetectable trojan horse: have login program recognize special login and make super-user, regardless of password file. But, login code would be giveaway, so change compiler to insert special code when compiling login (have special keyword in login file to do this). But, compiler code would be giveaway, so have special code for compiler too. The only difference is in the binaries!!!

Once the system has been penetrated, it may be impossible to secure it again: hooks could have been left around for the imposter to regain control.

It is not always possible to tell when the system has been penetrated, since the villain can clean up all traces behind himself.

If we can never be sure that there are no bugs, then we can never be sure that the system is secure, since bugs could provide loopholes in the protection mechanisms.

How are computers different from humans when it comes to security?

- Computer memory is volatile, humans don't forget.
- Anonymity is easy to come by in computers (can't really tell who is doing what).
- We are much more trusting of computers than of people: privileges are given away freely in huge doses: any program you run could conceivably modify any of your files.

Solutions: nothing works perfectly, but here are some possibilities:

- Logging: record all important actions and uses of privilege in an indelible file. Can be used to catch imposters during their initial attempts and failures. E.g. record all attempts to specify an incorrect password, all super-user logins. Even better is to get humans involved at key steps (this is one of the solutions for EFT).
- Caller identification: in telecommunications, it's traditional for callers to be anonymous, but not receivers. This makes it very difficult to catch electronic thieves. Callback is one way to get around this.
- Principle of minimum privilege ("need-to-know" principle): each piece of the system has access to the minimum amount of information, for the minimum possible amount of time. E.g. file system can't touch memory map, memory manager can't touch disk allocation tables. This reduces the chances of accidental or intentional damage. Capabilities are an implementation of this idea. It is very hard to provide fool-proof information containment: e.g. a trojan horse could write characters to a tty, or take page faults, in Morse code, as a signal to another process. (covert channels)
- Correctness proofs. These are very hard to do. Even so, this only proves that the system works according to spec. It doesn't mean that the spec. is necessarily right, and it doesn't deal with Trojan Horses.
- Type-safe languages and other modern programming tools. These help avoid bugs with potential security implications.