Computer Science 322
Operating Systems
Mount Holyoke College
Spring 2010

Topic Notes: Protection and Security

The operating system is (in part) responsible for *protection* – to ensure that each object (hardware or software) is accessed correctly and only by those processes that are allowed to access it.

We have seen protection in a few contexts:

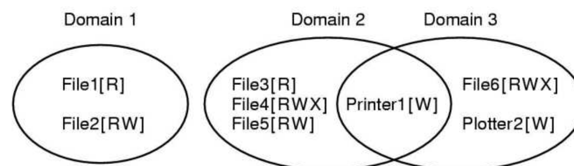
- memory protection
- file protection
- CPU protection

We want to protect against both malicious and unintentional problems.

Protection Domains

There are a number of ways that an OS might organize the *access rights* for its protection system. A *domain* is a set of access rights and the objects (devices, files, etc.) they refer to.

An abstract example from another text (Tanenbaum):



This just means that processes executing in, for example, domain 1, can read file1 and can read and write file2.

The standard Unix protection system can be viewed this way. A process' domain is defined by its user id (UID) and group id (GID).

Files (and devices, as devices in Unix are typically accessed through special files) have permissions for: user, group, and other. Each can have read, write, and/or execute permission.

A process begins with the UID and GID of the process that created it, but can set its UID and GID during execution. Whether a process is allowed a certain type of access to a certain file is determined by the UID and GID of the process, and the file ownership and permissions.

The *superuser* (UID=0) has the ability to set processes to any UID/GID. Regular users usually rely on *setuid* and *setgid* programs in the filesystem to change UID/GID.

An executable file with the setuid (setgid) bit set will execute with the effective UID (GID) of the file's ownership rather than the process that runs it.

See Example:
`/home/jteresco/shared/cs322/examples/setuid`

In this example, we see the use of several system calls related to user and group ids.

We can get our uid, effective uid, and gid with system calls.

The uid is the user who started the process and that user's default group.

The effective uid can be changed if the setuid bit is set on the program file.

We can set the uid and gid bits of the file with `chmod` and `chgrp` commands.

Example: see `/usr/X11R6/bin/xterm` on your favorite Unix system. Yes, each time you start an xterm, you're the superuser (!), at least for a few moments.

Recall that almost everything in the Unix world that requires protection by the OS uses the file system and its permissions:

- ttys (mesg/write)
- named pipes/semaphores
- process information (procfs)

Other possible protection mechanisms:

- protection matrix
- access control lists
- capabilities

See SG&G for details on these.

In any case, the OS must enforce the protection mechanism as appropriate.

Security

How can we define a system to be secure?

One possibility: all resources are used only exactly as intended

A secure system should not allow:

- unauthorized reading of information

- unauthorized modification of information
- unauthorized destruction of data
- unauthorized use of resources
- denial of service for authorized uses

Security: Authentication

Key aspect of security: Indentifying the user to the system.

Most common method: passwords

- passwords stored in an encrypted form - *cipher text*
- one-way encryption algorithm: easy to convert "clear text" password to cipher text, hard to convert back
- alternately, one-time passwords may be used – see crypto card, securid

In Unix, see `crypt(3)` for more information about the encryption used. See `/etc/passwd` or `/etc/shadow` (`/etc/master.passwd` in FreeBSD) for password files.

Security is compromised if passwords are compromised. There are many ways that this can happen:

- passwords written down
- typing simple passwords with someone watching
- run a "crack" program, encrpyting possible passwords, comparing to encrypted entries in the password file – many sysadmins do this periodically to look for guessable passwords on their systems
- packet sniffers – "watch" password get typed if clear text passwords are written to a shared network
- more recent problem: many web sites use passwords, meaning more chance that some passwords are stored in clear text or are even gathered by malicious web site operators or even more likely to be written down or stored somewhere
- trojan horse to gather passwords
- spoof/phishing web sites and e-mails

Security: Threats

- other trojan horses:
  - if . is in your search path, someone (maybe me) can replace something like `ls` when you're in our directory
  - if a system is compromised, many system programs could be "trojaned" – things like `top`, `ps`, might be included to make it harder to detect an intruder
- trap doors: special "ways in" to the system left by programmers, compilers, or malicious users – consider "easter eggs" – could be malicious
- stack and buffer overflows:
  - for example, if a program does not do good array bounds checking, this can be exploited to overwrite part of a program's memory (most importantly, a setuid program) to gain root access to a system
  - these are very common – source of the majority of recent Unix security problems
  - if the program is a network daemon, you may be able to crash it or use it to gain a shell on the system – so even someone without an account on the system might be able to break in
  - if the program is a setuid root binary, a local user might be able to get in
  - See web sites:
    * SecurityFocus: `http://www.securityfocus.org`
    * CERT: `http://www.cert.org`
    * Smashing The Stack For Fun And Profit: `http://www.phrack.org/issues.html?id=14&issue=49`

See Example:
`/home/jteresco/shared/cs322/examples/bufferoverflow`

We try this on a FreeBSD 6.x system. Newer versions are not vulnerable.

Here, we read characters into a buffer that's local to main then call a function that copies it into a smaller buffer.

If the input we type is longer than 80 characters, it doesn't fit.

We can watch what's happening here by compiling with -g and running in the debugger.

If we put in enough spaces to clobber the return address for the function, we get a segfault.

Different numbers of spaces as input causes different behaviors: 79-87: success, 88: i gets corrupted. 89-90: program crashes on return, 91+ program crashes on copy.

This particular exploit can be foiled by a technique called ProPolice used by gcc versions 4.1 and up, described on this page in the FreeBSD documentation:

`http://www.freebsd.org/doc/en/books/developers-handbook/secure-bufferov`
`html`

- worms and viruses – worms are standalone programs that replicate themselves, viruses are typically embedded in other files or programs – read about Morris Internet Worm
- denial of service attacks – don't actually break in, but render a system or network unusable by overloading it with invalid requests

Discussion topic: Open Source vs. Proprietary systems for security?

Encryption

For privacy or security, some information may need to be *encrypted*.

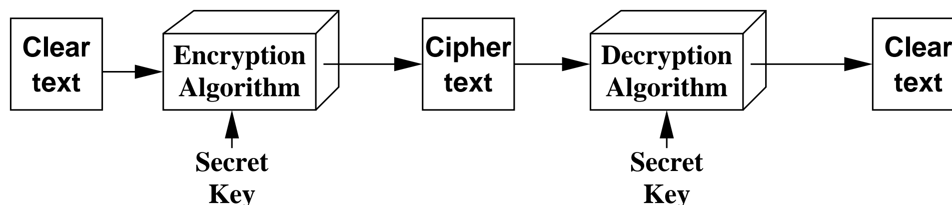
We have seen that Unix stores password entries in an encrypted form.

We won't talk in detail about networks here, that's a whole course, but one of the problems with communicating across many of today's networks is that the information on the network can be seen not only by its intended recipient, but also by many other computers. Encryption has been used for this purpose for centuries, often with the messages being military orders.

Even when a network is not involved, someone may want to encrypt files for privacy.

Original data is called *clear text*, encrypted version is called *cipher text*.

Conventional Encryption



The cipher text is a function of the clear text, the encryption algorithm, and the secret key. The algorithm is public! Or at least a good scheme should not rely on the secrecy of the algorithm. It's just the key that is kept secret.

The clear text is a function of the cipher text, the decryption algorithm, and the same secret key. Again, the algorithm is public. The decryption returns the original clear text.

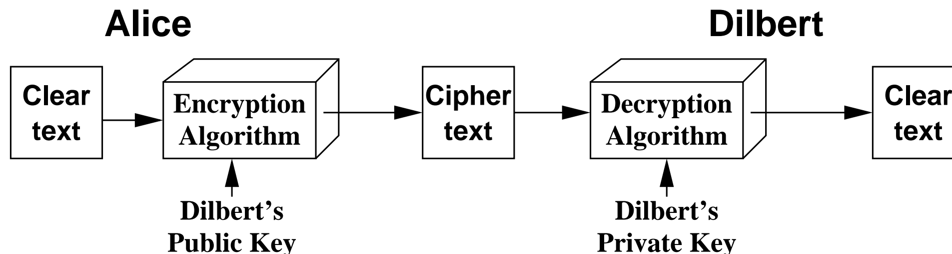
For the encryption to be strong enough, it must be very difficult to figure out the secret key, even given a bunch of cipher texts and the algorithm. Two approaches that an adversary may use are *cryptanalysis*, where properties of the clear text and the nature of the algorithm are examined to deduce the secret key, and a *brute-force attack*, where every possible key is tried until one works. For an n-bit key, this means up to 2^n keys must be tried, making brute-force attacks expensive. But modern hardware can break a 56-bit key in just a few hours.

Examples:

- **The Data Encryption Standard (DES)** – 56-bit secret key. Selected by US Gov't in 1977. Broken in 1998.
- New **Advanced Encrpytion Standard (AES)** – 168-bit key – competition was held, and **Rijndael** was selected in 2000 as the new standard. See more at `http://csrc.nist.gov/encryption/aes/`.

Problem: how do we tell the intended recipient of our messages what our secret key is, without telling all the world what our secret key is? Perhaps this can be sent securely by some other means, but perhaps the only communication channel is the one we do not trust that led us to employ encryption in the first place.

Public-Key Encryption

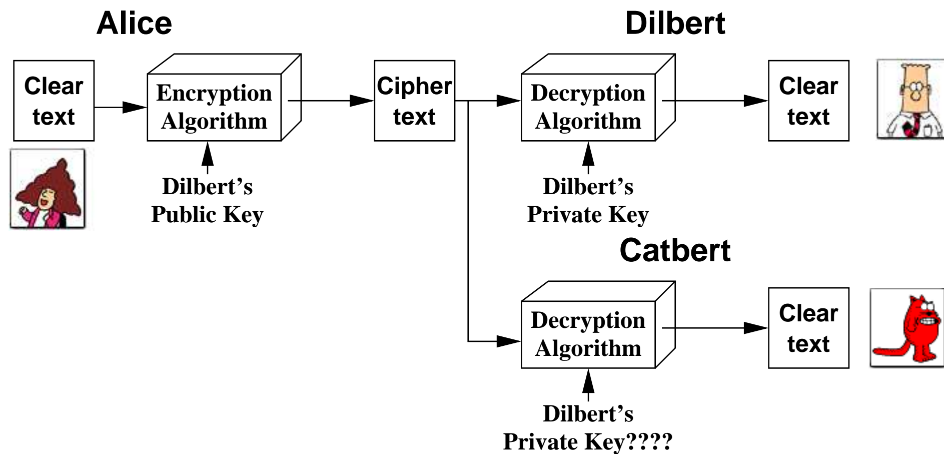


Instead of a single key, we have a *public key* and a *private key*. The public key is, well, public – anyone who wants to have it, can have it. But the private key is never shared. This idea was proposed in 1976 by Diffie and Hellman.

To transmit a message securely from Alice to Dilbert:

1. Alice and Dilbert each compute their public key-private key pair. Each publishes its public key for all the world to see.
2. Alice looks up Dilbert's public key, and uses it to encrypt the message.
3. Dilbert receives the message, and uses his private key to decrypt.

But what if Catbert intercepts the message?



Everything is just fine – even though Catbert, the evil director of Human Resources, has the cipher text and he can have Dilbert's public key, he does not have Dilbert's private key. So he has no way to intercept the message.

The most popular public-key algorithm is the *Rivest-Shamir-Adleman (RSA) Algorithm*. It uses the fact that it is relatively easy to compute numbers that are products of large primes, but very difficult to factor the number into those primes.

Secure shell works this way – when you set up ssh, your computer computes its public key and private key. When another computer wants to communicate securely with yours, they exchange public keys and they're off.

There's still a potential problem with the distribution of public keys. Suppose Alice decides to send Dilbert a message for the first time, so she needs his public key. When she makes that request, maybe Dilbert is out of the office, but Catbert pretends to be Dilbert (spoofs his address) and sends his own public key instead. Since Alice didn't know it came from Catbert instead of Dilbert, she gladly encrypts messages intended for Dilbert using the bad public key, and Catbert sits in his office decrypting, soon to fire Alice for what she said about him..

There is a lot more to discuss about encryption, but most of it does not really fit into this course. Some links to visit for more information can be found on the lecture page for this topic.