Computer Science 330
Operating Systems
Siena College
Fall 2020

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:

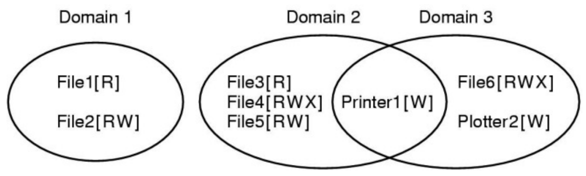
- memory protection
- file protection
- CPU protection

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.

`https://github.com/SienaCSISOperatingSystems/setuid-example`

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/bin/passwd` on FreeBSD. Yes, when you change your password, 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 the text for details on these. Not urgent for us.

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
- 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

There are constantly threats in our day-to-day use of technology. These can include:

- Stealing your system
  - vandalism
  - for-profit exploitation
- Stealing your information
  - identity theft
- Stealing your time
  - Spam – unsolicited email
  - denial of service attacks

Many common security threats involve “rogue code” or *malware* (malicious software) running on a computer.

- Infectious software
  - *viruses* and *worms*
  - spread by “making copies” of themselves like a real virus
- Concealment software
  - *trojans* – rogue software made up to look like something else – see example below
  - *rootkits* – change the OS to hide the rogue software’s existence
  - *backdoors* or *trap doors* – special “ways in” to the system left by programmers, compilers, or malicious users – consider “easter eggs” – could be malicious and be used to bypass your security
- For Profit
  - *Phishing* – spoofing you into believing you are going somewhere you aren’t
  - *Adware* – malicious advertisements on your browser
  - *Spyware* – log keystrokes and report over the net
  - *Scareware* – protection racket

How might the “bad guys” attack you?

- “Crack” your password
  - essentially keep trying possibilities until it works
  - How hard is it to crack yours?

  - How do you make a really strong password?

- Trick you into "opening the door"
  - Trojan "Games" "P2P" "Tool Bars"
  - Look alike web pages for Banks, Paypal, *etc.*
  - Device drivers from the net
  - Scam emails the Nigerian prince, webmail quota, *etc.*
  - Malicious attachments to email messages: programs disguised as images or movies

- Bugs (errors) in an operating system or system services
  - These would often run in a priviledged mode (*i.e.*administrator, superuser)
  - Attacker has an *exploit* for one of these bugs that allow the attacker to perform his own actions, now in that priviledged mode

Simple trojan horse example: 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

Stack and buffer overflows are the "way in" for many security problems over the last few decades.

- 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: `https://www.securityfocus.com/vulnerabilities`
  - US-CERT: `https://us-cert.cisa.gov/`
  - Smashing The Stack For Fun And Profit: `http://phrack.org/issues/49/14.html#article`

Until FreeBSD 6.x, the code in the last link could be used to exploit a setuid program. 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.

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:

`https://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

And what can you do to protect yourself?

- Filter out the threats
  - *Firewalls*
    - restrict the network services you accept
    - examine packets before they get "unpacked" and sent to their target application
    - can be done in hardware (routers block some ports or addresses) or software (OS kernel blocks some ports and/or addresses)
    - generally good to block all services except those you definitely need and use
  - *Spam filters*
    - standard on services such as GMail
    - what about false positives?
  - restrict browser capabilities from all but trusted sources
    - avoid accepting cookies
    - block scripts with tools like NoScript
    - delete personal information (history, autofill)
- Detect and Remove
  - Virus scan software
    - has to scan your whole file system

    * needs constant updates
  - Spyware/Adware detection and removal

- Keep operating system, browser, and other software up-to-date
  - "patch" to stop newly discovered "exploits"
- Good password practices
  - Use a "hard to crack" password that is not a dictionary word, using upper- and lower-case letters, numbers, punctuation
  - Don't use the same password everywhere – if one is compromised the attacker would have access to many other things
  - Too complex and you can't remember it and have to write down – even worse! How secure is that paper, or that file called `passwords` on your desktop?
  - Some ideas:
    * use the first letter of a sequence of words mixing in caps, numbers, punctuation
    * have a pattern that is similar for many of your accounts but which has some site-specific change you can make for each
    * have some "throwaway" passwords to use for registrations that do not require much security
    * be sure to use strong passwords for services like banking, email, *etc.*
  - Authentication using more than passwords to establish identity
    * a password is *something you know*
    * can also require *something you are*: *biometrics*
    * can also require *something you have*: *single-user password generators*

It is not only individual computers that present significant security risks.

- *Denial of service* (DoS) attacks, often directed against web servers
  - one method: direct a fast stream of bogus requests to a server so it cannot service legitimate requests either at all or at least not in a timely fashion
- *Botnets* – networks of computers controlled by a central coordinator
  - can be legitimate: SETI@Home and similar distributed computing projects
  - often attackers gain control of computers (turning them into "robots", hence the name) and use them to perform a variety of malicious acts
  - launch *distributed denial of service* (DDos) attacks
  - generate large volumes of spam email

- botnet "owner" may simply "rent out" the botnet for use
- July 2010: operator of 12-million node botnet Mariposa arrested

So who are these "bad guys" using these techniques to steal data and access?

- Professional hackers
  - organized crime
  - former government agents
  - fired former employees
  - often are very skilled programmers
- "Script Kiddies"
  - often bored teenagers who find hacker scripts and try them
  - usually essentially vandals
- Marketers

What are the targets for theft, and what good is each to the theives?

- Credit/Debit card numbers
  - to make fraudulent charges
- Social Security Numbers and similar personal identifiers
  - to establish bogus accounts, make fraudulent charges
- Access to computers
  - use *your* computer, not their own, to commit fraud or break the law
  - pass through to cover tracks
- Web surfing/online shopping data
  - improve targeted marketing

Unfortunately, loss of personal information is very common and does lead to *identity theft*.

Many attacks are not strictly electronic – they have a physical component as well:

- Stealing physical hardware

  - laptops, flash memory sticks, mobile devices

- Office break-ins
  - steal passwords from desks
- Dumpster diving

A few more important terms with examples:

- *Intrusions*: the hacker uses electronic or physical means to compromise a system
  - an effective way to compromise a system is to steal the username and password of a legitimate user
  - easy to cover your trail
  - password cracking (guessing) is a common example
  - programs can try millions of combinations very quickly before being noticed
- *Phishing*: techniques for getting a user to disclose information, like passwords
  1. Hacker creates a website that looks just like your bank (*e.g.*`http://www.citi.com.IPAddress.net`)
  2. Send spam email to 20 million people telling them that they have to visit the site and log in for whatever reason
  3. Even if 0.001% of the people respond, this steals 20,000 bank passwords!

  Also, create the fraudulent site on someone else's server (perhaps in a botnet), to make it harded to catch.
- *Eavesdropping*: techniques for intercepting passwords
  - One possibility: create a public wireless access point with a nice, tempting name like "FreePublicWifi"
  - Users connect, and this access point monitors everything they do
  - Lesson: **always** assume that your network traffic is being monitored, especially when connecting to unknown wireless access points!

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

Encryption

Recall that Internet traffic of all kinds is broken down into packets, and sent through a series of routers from source to destination. 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. Each router along the way and possibly other computers on the same networks as the source and destination will be able to see these packets of information. We must assume that any packet that gets sent onto the network will not only be readable by the destination computer but by pretty much anyone. Tools such as *packet sniffers* can turn any computer into a packet spy on the network.

Combine this fact with the fact that we send information over the Internet all the time that we do not want made public, and it's clear that something must be done to keep our Internet traffic a bit more private.

Any data that should not be exposed to the public needs to be *encrypted*.

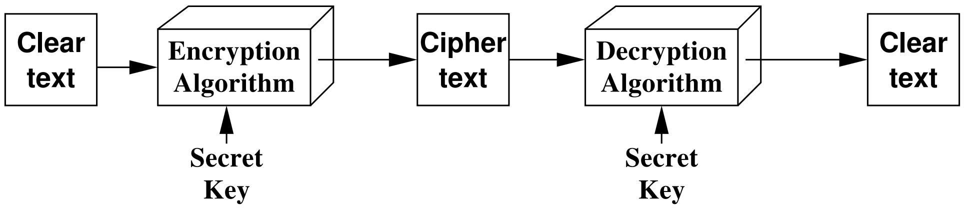
We have seen that Unix stores password entries in an encrypted form.

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.

Encryption allows some data, which we will call *clear text* to be modified into encrypted and seemingly unintelligible *cipher text*. This cipher text can then be sent across a network or stored on a device, and if anyone sees it, they will be unable to ascertain its meaning. Of course, to be useful, we will also need some way to convert the cipher text back to the original clear text (the process of *decryption*).

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.

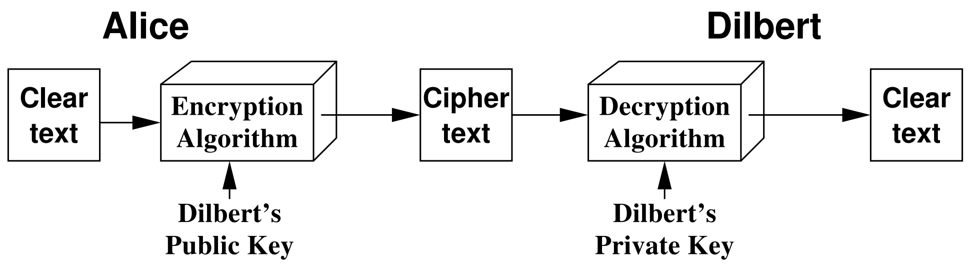
For the encryption to be strong enough, it must be very difficult to figure out the secret key, even given a bunch of cipher texts and the algorithm. Two approaches that an adversary may use are *cryptanalysis*, where properties of the clear text and the nature of the algorithm are examined to deduce 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)* – key sizes of 128, 192, or 256 bits. A competition was held, and *Rijndael* was selected in 2001 as the new standard. See more at `http://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

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

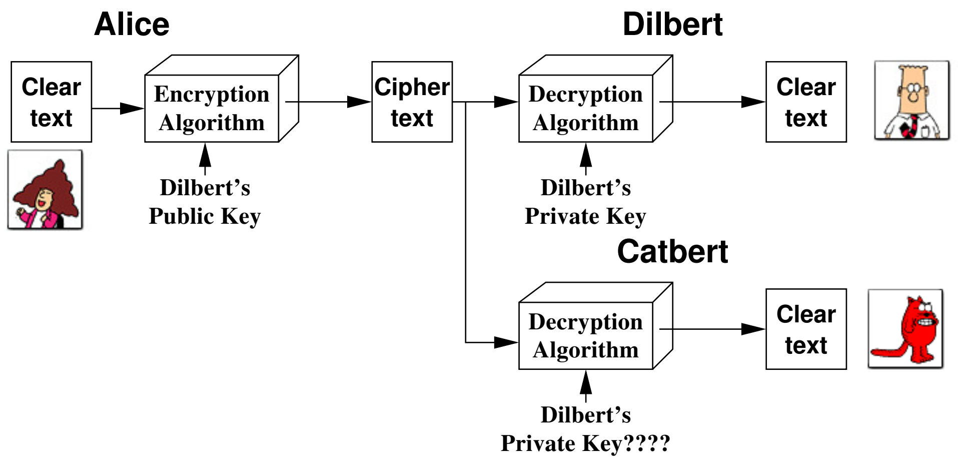


Instead of a single key, we have a *public key* and a *private key*. The public key is, well, public – anyone who wants to have it, can have it. But the private key is never shared. This idea was proposed in 1976 by Diffie and Hellman.

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 *Rivest-Shamir-Adleman (RSA) Algorithm* public-key algorithm 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. This is known as a *man in the middle attack*.

We need a way for everyone involved to ensure that the others with whom they are communicating are who they say they are.

The mechanism most commonly used involves *digital signatures* or *certificates of authority*. A secure certificate authority (a popular one is called Verisign) gives a "certificate" to a computer that it can present to any other computer to prove that it is who it says it is. Sort of an electronic identication card.

When starting a connection with a server, we would request its certificate and compare to the certificate for that server from the certificate authority. If they match, we can be confident that we're exchanging information with the computer we think we are exchanging information with. Once we've done that, we can exchange public keys and continue on and have a safely encrypted conversation.

Your web browser does this all the time. If you are going to a web site where you are going to pass sensitive information across the network, you will likely connecting using `https`, the *secure http*, instead of standard `http`.

When you connect to a web server using `https`, your browser will make sure that the server is who it says it is by making sure the certificate matches one we have seen from that server in the past, or by checking with the certificate authority if we have never visited that server before. It can then use the *secure socket layer (SSL)* connection to communicate securely using a random encryption key (good for just the one session).

Note that modern browsers indicate that a connection is using `https` by putting a little padlock on the screen. Any time you are going to send information over the Internet that you would not want to be seen by everyone in the world, make sure it is going over an `https` encrypted connection!

There is a lot more to discuss about encryption, but most of it does not really fit into this course.