CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2002

Lecture 24: Protection and Security in Distributed Systems

24.0 Main Point

Why you shouldn't ever trust a computer system

Goal: Prevent misuse of computers

24.1 Definitions

Types of misuse:

1. Accidental
2. Intentional

Protection is to prevent either accidental or intentional misuse. **Security** is to prevent intentional misuse.

Two parts to this:

- Conceptual understanding of how to make systems more secure
- Some examples, to illustrate why providing security is really hard in practice.

Three pieces to security:

1. Authentication – who user is
2. Authorization – who is allowed to do what
3. Enforcement – make sure people do only what they are supposed to do

Loophole in any of these, problem: For example:

1. Log in as superuser and you've circumvented authentication
2. Log in as self and you can do anything you want to your own resources. What if you run some program that decides to erase all your files?
3. Can you trust software to correctly enforce decisions about 1 + 2

24.2 Authentication

Common approach: passwords. Shared secret between two parties. Since only I know password, machine can assume it is me.

Problem 1: system must keep copy of secret, to check against passwords. What if malicious user gains access to this list of passwords?

Encryption – transformation that is difficult to reverse without the right key

For example: UNIX `/etc/passwd` file

passwd -> one way transform (hash) -> encrypted passwd

System stores only encrypted version, so OK even if someone reads the file! When you type in your password, system compares encrypted versions.

Problem 2: Passwords must be long and obscure

Paradox: Short passwords are easy to crack
Long ones, people write down!

Technology means we have to use longer passwords: UNIX initially required only lowercase, 5 letter passwords

How long for an exhaustive search? 26^5 = 10 million

In 1975, 10 ms to check a password -> 1 day

In 1992, 0.001 ms to check a password -> 10 seconds

Many people choose even simpler passwords, such as English words – takes even less time to check for all words in the dictionary!

Some (partial) solutions:

a. Extend everyone's password with a unique number (stored in password file), so can't crack multiple passwords at a time. UNIX uses 12-bit "salt" (makes it 2^{12} or 4096 times harder).
   The salt is used to increase the cost of dictionary attacks. If a salt were not used, it would be possible to precompute a tape with all the words in the dictionary encrypted (hashed), the dictionary attack would then degenerate to simply streaming the pre-encrypted fields from the tape, and comparing them to any password files being attacked.

   A second reason for the use of salts, is that the way that the salt is combined in a first stage which permutes the password with the salt is designed to frustrate the use of off-the-shelf DES hardware.

   Without salts, it would take less than 10 seconds to crack every account on entire system!

b. Require more complex passwords. For example: 6 letters (uppercase and lowercase), numbers, and special characters:
   $70^6 \approx 600$ billion, or 6 days

   Except, people still pick common patterns (ex: 5 lower case letters, plus one number).

c. Make it take a long time to check each password. For example, delay every remote login attempt by 1 second.

d. Assign very long passwords. Give everyone a smart card (or ATM card) to carry around to remember the password. Requires physical theft to steal password.
   Long passwords or passphrases can have more entropy (randomness -> harder to crack).

Smart cards – generate pseudorandom number (client and server both have the same initial seed and accurate clocks). Can have a keyboard (for PIN code) or user can prepend/append PIN – helps prevent theft problems.

Problem 3: Can you trust the encryption algorithm? Example: one algorithm that was thought to be safe had a back door. If there is a back door, means you don't need to do complete exhaustive search.

Also, security through obscurity doesn't work (Example: GSM encryption algorithm was secret, accidentally released – Berkeley graduate students cracked it in a few hours! Also, found that the algorithm was purposefully weakened!)

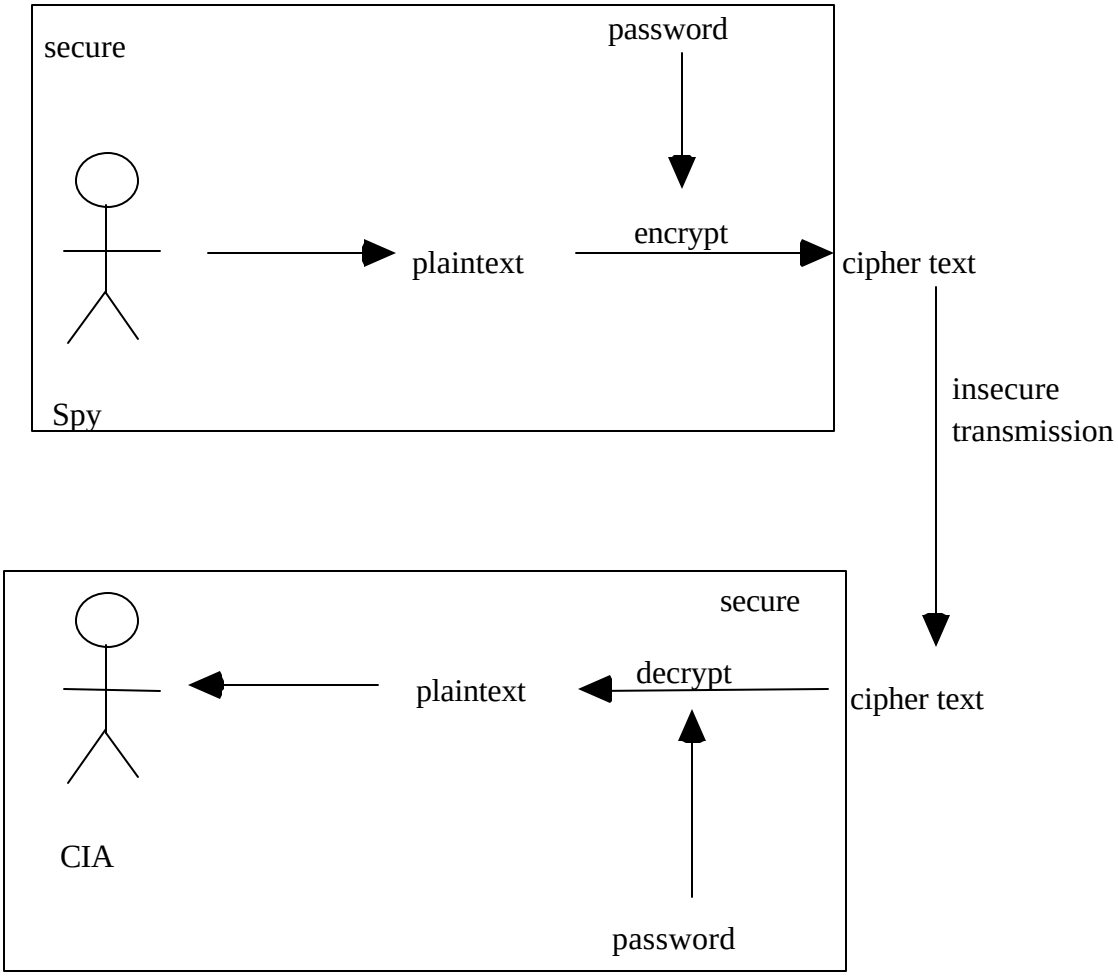
24.3 Authentication in distributed systems

Two roles for encryption:

a. Authentication
b. Secrecy – I don't want anyone to know this data (e.g., medical records, etc.)

24.3.1 Private key encryption

Private key: use an encryption algorithm that can be easily reversed, given the correct key (and hard to reverse without the key)



From cipher text, can't derive plain text (decode) without password.
From plain text and cipher text, can't derive password!

As long as password stays secret, get both secrecy and authentication.

But how do you get shared secret in both places? (e.g., key distribution)

Authentication server (example: Kerberos)

Server keeps list of passwords, provides a way for two parties, A, B to talk to one another, as long as they trust server.

Notation:

K_{xy} is a key for talking between x and y.

$(..)^K$ means, encrypt message (…) with the key K.

A asks server for key:

A -> S (Hi! I'd like a key for talking between A and B)

Server gives back special *session* key encrypted using B's key:

S -> A (Use K_{ab} (This is A! Use K_{ab})^{Ksb})^{Ksa}

A gives B the ticket:

A -> B (This is A! Use K_{ab})^{Ksb}

Lots of details:

1. Add in timestamps to limit how long a key will be used and to prevent a machine from replaying messages later!
2. Also have to include encrypted checksums (hashed version of message), to prevent malicious user from inserting stuff into the message or changing the message!
3. Want to minimize # of times password must be typed in, and minimize amount of time password is stored on machine. So initially ask the server for a temporary password, using the real password for authentication:

   A->S (Give me a temporary secret)

   S->A (Use $K_{temp-sa}$ for the next 8 hours)^{Ksa}

   Can now use $K_{temp-sa}$ in place of K_{sa} above.

24.3.2 Public key encryption

With a private key system you encrypt a message and decrypt the message with the same key. Private key systems are also called *symmetric* systems. Such systems require that parties share a trusted authentication server.

What if A and B don't share a trusted authentication server?

Public key encryption is an alternative to private key; separates authentication from secrecy.

With a public key system, each key is a pair: K_{public}, $K_{private,}$ such that:

$(text)^{Kpublic} = ciphertext$

$(ciphertext)^{Kprivate} = text$

and

$(text)^{Kprivate} = ciphertext'$

NOTE: not same ciphertext as above!

$(ciphertext')^{Kpublic} = text$

and:

$(ciphertext)^{Kpublic} != text$

$(ciphertext')^{Kprivate} != text$

and: can't derive K_{public} from $K_{private}$ or vice versa.

Idea is: $K_{private}$ kept secret, K_{public} put in a telephone directory.

For example, assume $KF_{private}$ and KF_{public} are Fred's keys and $KJ_{private}$ and KJ_{public} are Joe's keys:

$(I'm\ Fred!)^{KFprivate}$

Everyone can read it, but only Fred can send it (*authentication*)

$(Hi!)^{KFpublic}$

Anyone can send it, but only Fred can read it (*secrecy*)

$((I'm\ Fred!)^{KFprivate}\ Hi!)^{KJpublic}$

Only Fred can send it, only Joe can read it.

Problem: how do you trust dictionary of public keys?

24.4 Authorization

Authorization: who can do what.

Basic concepts covered in a previous lecture – quick review:

Access control matrix: formalization of all the permissions in the system

Objects	file1	file2	file3	...
users				
A	Rw	r		
B		rw		
C			r	
...				

For example, one box represents C can read file3.

Potentially huge # of users, operations, so impractical to store all of these

Two approaches:

1. Access control list – store all permissions for all users with each object

   Still, might be lots of users! UNIX addresses this by having each file store: r, w, x for owner, group, world. More recent systems provide way of specifying groups of users, and permissions for each group.

2. Capability list – each process, stores all objects the process has permission to touch

   Lots of capability systems built in the past; idea out of favor today. But page tables are an example. Each process has list of pages it has access to; not each page has list of processes that are permitted to access it.

The real problem: how fine-grained should authorization be?

Example of the problem:

> Suppose you buy a copy of a new game from "Joe's Game World" and then run it.
>
> It's running with your userid.

It removes all the files you own, including the project due the next day!

How can you prevent this?

- Have to run the program under *some* userid. Could create a second *games* userid for the user, which has no write privileges.
  Like the nobody userid in UNIX – can't do much.
- But what if the game needs to write out a file recording scores? Would need to modify your *games* userid to have write privileges to one particular file (or directory).
- But what about non-game programs you want to use, such as Quicken? Now you need to create your own private *quicken* userid, if you want to make sure that the copy of Quicken you bought can't corrupt non-Quicken-related files.
- What about word processor programs, which need to have read/write access to entire categories of files?

One semi-satisfactory way to deal with this problem is to only use software from sources you trust, thereby dealing with the problem by means of a form of authentication.

That's fine for big, established firms, like Microsoft. But what about new start-ups? Who "validates" them?

Can establish validation agencies. But how easy is it to fool them? We have no real experience with this yet.
An even bigger risk these days:

- "Programs" can appear on your machine in the form of macros attached to your documents (as is the case with Microsoft Word and Excel)
- Java applets that are part of Web pages!

Macros (typically) run with full privileges and may get automatically invoked as part of initially accessing a document, or as part of saving the document later on.

Macros can be used as virus vectors – replicating themselves when documents are opened or copied.

Java applets are normally *sand-boxed*: the Java Virtual Machine inside a Web browser runs them with no privileges except the ability to send and retrieve data from the server that the Web page they are part of is from.

However, as Web page designers have created ever more sophisticated applets, they have started demanding that Java allow limited access to the resources of the client machine that the Web browser is being run on.

The general problem:

- How do I specify the exact privileges that something running on my behalf should have?
- How to avoid making this specification task so onerous that no one will put up with it?

24.5 Enforcement

Enforcer checks passwords, access control lists, etc.

Any bug in enforcer means: way for malicious user to gain ability to do anything!

In UNIX, superuser has all the powers of the UNIX kernel – can do anything. Because of coarse-grained access control, lots of stuff has to run as superuser in order to work. If there's a bug in any one of these programs, you're hosed!

Paradox:

a. Bullet-proof enforcer
   Only known way is to make enforcer as small as possible.
   Easier to make correct, but simple-minded protection model
b. Fancy protection – only minimal privilege necessary
   Hard to get right.

24.6 State of the world in security

Authentication – encryption

But almost nobody encrypts!

Authorization – access control

But many systems provide only very coarse-grained access control (ex: UNIX – means, need to turn off protection to enable sharing)

Enforcement – kernel mode

Hard to write a million line program without bugs, and any bug is a potential security loophole.

24.7 Classes of security problems

24.7.1 Abuse of privilege

If the superuser is evil, we're all in trouble

Nothing you can do about this

24.7.2 Imposter

Break into system by pretending to be someone else.

For example, in UNIX, can set up an `.rhosts` file to allow logins from one machine to another, without having to re-type password.

Also allows "rsh" – command to do an operation on a remote node.

Combination means: send rsh request, pretending to be from the trusted user, to install .rhosts file granting imposter full access!

Similarly, if you have open X windows connection over the network, an imposter can send messages appearing to be keystrokes from a window, but really they are commands to give the imposter access.

Currently, X has no way of encrypting its packets – so no way to stop this!

24.7.3 Trojan horse

One army gave another a present of a wooden horse, army hidden inside.

Trojan horse appears helpful, but really does something harmful

24.7.4 Salami attack

Idea: steal or corrupt something a little bit at a time.

For example: What do you do with all those partial pennies from bank interest?

> Bank keeps it! Hacker re-programmed it so that the partial pennies would go into his account. Doesn't seem like much, but if you are Bank of America, with a few million customers, adds up pretty quickly!

24.7.5 Eavesdropping

Listener – tap into serial line on the back of the terminal, or onto Ethernet. See everything typed in; almost everything goes over network unencrypted. For instance, rlogin to remote machine, your password goes over the network unencrypted!

Spoiler – not stealing information, just making system unusable. Just chews up system resources; electronic equivalent of vandalism.

How do you prevent these? Hard to build system that is both useful, and prevents misuse.

24.8 Concrete Examples

24.8.1 Tenex – early 70's, BBN

Most popular system at universities before UNIX

Thought to be very secure. To demonstrate it, created a team to try to find loopholes. Gave them all the source code and documentation (want code to be publicly available, as in UNIX); gave them a normal account.

In 48 hours, they had every password in the system.

Here's the code for the password check:

```
for (i = 0; i < 8; i++)
     if (userPasswd[i] !=
     realPasswd[i])
          go to error
```

Looks innocuous, like you'd have to try all combinations. 256^8

Wrong!

Tenex also used virtual memory, and it interacts badly with the above code.

Key idea: force page faults at inopportune times; can break passwords quickly.

Arrange first character in string to be the last character in page, rest to be on the next page. Arrange for the page with the first character to be in memory, and rest to be on disk (for example, by referencing lots of other pages, then referencing the first page).

```
                a|aaaaaa
                 |
page in memory| page on disk
```

By timing how long the password check takes, can figure out whether the first character is correct!

- If fast, first char is wrong
- If slow, first char is right, page fault, one of the others was wrong

So try all first characters, until one is slow. Then put first two characters in memory, and the rest on disk. Try all second characters, until one is slow.

Means takes only a maximum of 256 * 8 attempts to crack passwords.

Fix is easy, don't stop until you look at all the characters.

But how do you figure this out in advance?

24.8.2 Internet worm

Ten years ago, the worm broke into thousands of computers over Internet.

Three attacks:

1. Dictionary lookup-based password cracking
2. sendmail
   – Debug mode, if configured wrong, can let anybody log in
3. fingerd
   – finger adj@cs

   fingerd didn't check for length of string, but only allocated a fixed size array for it on the stack. By passing a (carefully crafted) really long string, a program could overwrite fingerd's stack and get the program to call arbitrary code!

Got caught because the idea was to launch attacks on other systems from whatever systems were broken into; so ended up breaking into same machine multiple times, dragged CPU down so much that people noticed (was a bug in the code).

Variant of this problem: kernel checks system call parameters to prevent anyone from corrupting it by passing bad arguments.

So kernel code looks like:

    Check parameters; if ok, use arguments

But what if application is multithreaded? Can change contents of arguments after check and before use!

24.8.3 Kevin Mitnick

Two attacks:

1. Misdirection: Identify system managers' machines, then loop, requesting TCP connections to those machines, until no more connections are permitted. Freezes those machines.

Meanwhile:

2. Imposter: forge packets to appear as if legitimate (e.g., by replacing source address in packet header), but really from Mitnick.

If notice an open, idle rlogin connection, for example, can send packets as if user typed command to add Mitnick to .rhosts.

24.8.4 Netscape follies

Netscape claimed to provide secure communication, for example, so you could send a credit card # over the Internet.

Three problems (reported in NYT):

1. Algorithm for picking session keys was predictable (used time of day). Brute force allowed someone to break a session key in a matter of hours.
2. Made new version of Netscape to fix #1, available to users over Internet (unencrypted!). Four byte patch to Netscape executable can make it always use a specific session key – so can insert backdoor by mangling packets containing executable as they fly by on the Internet.

   In fact, because of demand, they had a dozen mirror sites (including Berkeley) to redistribute new version. So anyone with root access to any machine on LAN at mirror site could insert the backdoor.
3. Buggy helper applications. As with fingerd attack, *any* bug in either Netscape, or its helper applications (e.g., ghostview), can potentially be exploited by creating a Web page that when viewed, will insert a Trojan horse.

Can you trust an application that was preloaded on your computer at the factory?

- Not really, a major computer manufacturer just shipped several thousand computers with the CIH virus.
- Software companies, PR firms, and others routinely release software that contains viruses.

24.8.5 Ken Thompson's self-replicating program

Bury Trojan horse in binaries, so there's no evidence in the source

Replicates itself to every UNIX system in the world, and even to new UNIX's on new platforms. No visible sign.

Gave Ken Thompson the ability to log into any UNIX system in the world.

Two steps:

1. Make it possible (easy)
2. Hide it (tricky)

Step 1. Modify `login.c`

```
A:
     if (name == "ken")
          don't check password
          log in as root
```

Idea is: hide change, so no one can see it.

Step 2. Modify the C compiler

Instead of having the code in login, put it in the compiler:

```
B:
     if see trigger,
          insert A into input stream
```

Whenever the compiler sees a trigger (/* gobbledygook */), puts A into input stream of the compiler

Now, don't need A in login.c, just need the trigger.

Need to get rid of the problem in the compiler

Step 3. Modify compiler to have:

```
if see trigger2
     insert B + C into input stream
```

This is where self-replicating code comes in! Question for reader: can you write a C program that has no inputs, and outputs itself?

Step 4. Compile the compiler with C present

– Now it is in the binary for compiler

Step 5. Replace code with trigger2

Result is – all this stuff is only in the binary for the compiler. Inside the binary there is C, inside that, code for B, inside that code for A. But source code only needs trigger2!

Every time you recompile login.c, the compiler inserts the backdoor. Every time you recompile the compiler, the compiler re-inserts the backdoor.

What happens when you port to a new machine? Need a compiler to generate code; where does that compiler run?

On the old machine – C compiler is written in C! So everytime you go to a new machine, you infect the new compiler with the old one.

24.9 How to Fix Security

Lots of examples of problems. How do we fix them?

Start with people!

1. Security education is critical because people:
   - Write passwords down
   - Share passwords

     Guess what, it's not how you think!

     Classic attack: Company hires hackers to break into new system. Hackers break in two hours later!

     How? Social engineering
   - Are fired/laid off

     Worker who was laid off said, "I'll show them I'm important"

     Encrypted warehouse inventory database!

     Millions of items, where are they?

     Tried to blackmail company, failed because company went public.

   Solution is to provide training and proper safeguards (e.g., smart cards, limited access). Easy conceptually, hard in practice.
2. Secure distribution of software:
   - Tight source access controls and code reviews.

     Using source revision controls allows company to monitor changes and helps recovery.
   - Use antiviral software before releasing software.
   - Sign software distribution with company's public key (use hash to detect changes).

     How do you get the key? You don't.

     Instead, software preloaded on your computer contains the key of a trusted Certificate Authority. CA uses their private key to encrypt the company's public key and information.
3. User's machine determines sender and asks user for permission

   Has to validate key, verify contents haven't been tampered with while in transit to user.
4. Same process can be used for updates

In practice, most companies don't use these mechanisms. And, when they do, it still doesn't work if there are any holes.

Alternative is Java approach: Use sandboxing and careful control of access points. Very difficult to get correct!

24.10 Lessons

1. Hard to re-secure after penetration
   What do you need to do remove the backdoor? Remove all the triggers?

   What if he left another trigger in the editor – if you ever see anyone removing this trigger, go back and re-insert it!

   Re-write the entire OS in assembler? Maybe the assembler is corrupted!

   Toggle in everything from scratch every time you log into the computer?

2. Hard to detect when system has been penetrated. Easy to make system forget

3. Any system with bugs has loopholes (and every system has bugs!)
   Summary: can't stop loopholes, can't tell if it's happened, can't get rid of it.