Lecture 26: Protection and Security

26.0 Main Point

Why you shouldn't ever trust a computer system

Goal: Prevent misuse of computers

26.1 Definitions

Types of misuse:

1. Accidental

2. Intentional

Protection is to prevent either accidental or intentional misuse; security is to prevent intentional misuse.

This lecture and the next focuses on security.

This lecture: conceptual understanding of how to make systems more secure

Next lecture: 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
2. log in as self, do anything

3. can you trust software to make decisions about 1 + 2

26.3 Authentication in distributed systems

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

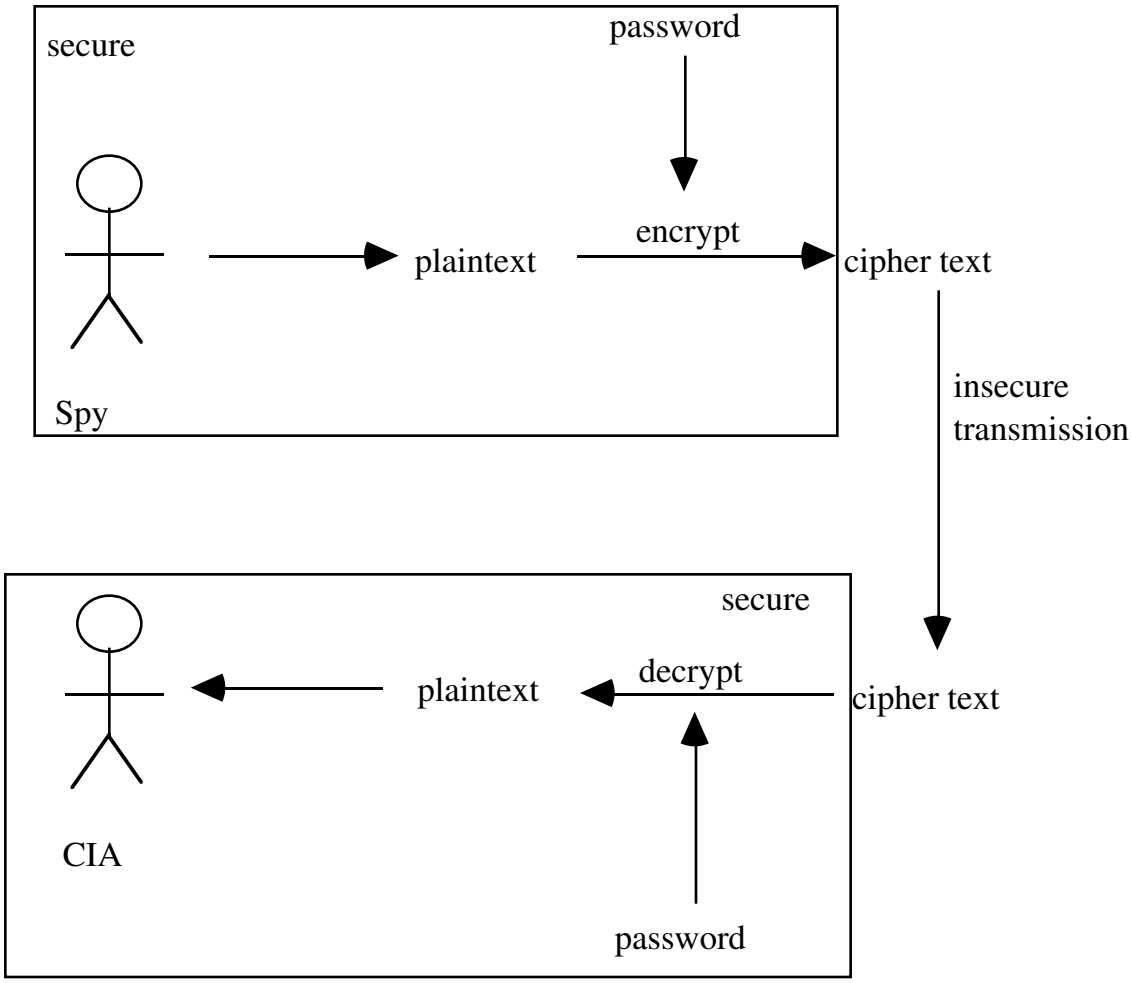
Two roles for encryption:

a. Authentication

b. Secrecy -- I don't want anyone to know this data (eg, medical records, etc.)

26.3.1 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 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?

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:

Kxy is a key for talking between x and y.

(..)^K means, encrypt message (..) with key K.

A asks server for key:

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

Server gives back special "session" key encrypted in B's key:

S -> A (Use Kab (This is A! Use Kab)^Ksb)^Ksa

A gives B the ticket:

A -> B (This is A! Use Kab)^Ksb

Details:

1. add in timestamps to limit how long a key will be used, to prevent a machine from replaying messages later!

2. Also have to include encrypted checksums, to prevent malicious user from inserting stuff into 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 Ktemp-sa for the next 8 hours)^Ksa

Can now use Ktemp-sa in place of Ksa above.

26.3.2 Public key encryption

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

Each key is a pair: K-public, K-private

With private key system:

(text)^K = ciphertext

(ciphertext)^K = text

With public key system:

(text)^K-public = secret text

(secret text)^K-private = text

and

(text)^K-private = signed text

NOTE: not same secret text as above!

(signed text)^K-public = text

and:

(secret text)^K-public != text

(signed text)^K-private != 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:

(I'm tom!) ^K-private

Everyone can read it, but only I can send it (authentication)

(Hi!) ^ K-public

Anyone can send it, but only the target can read it (secrecy)

((I'm tom!)^K-private Hi!)^K-public

Only I can send it, only you can read it.

Problem: how do you trust dictionary of public keys?

26.5 Enforcement

Enforcer checks passwords, encryption keys, access control lists, etc.

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

In most OS's, lots of programs have to run with all the privilege of the kernel in order to work. If bug in any one of these programs, you're hosed!

Paradox:

a. 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.

26.6 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.

26.7 Classes of security problems

26.7.1 Abuse of privilege

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

nothing you can do about this

26.7.2 Imposter

Break into system by pretending to be someone else.

For example, can spoof a TCP connection by sending IP packets that appear to be from the sender.

Ex: if have remote login open, imposter can appear to type messages on behalf of the user.

Ex: if have open X windows connection over the network, can send message appearing to be keystrokes from window, but really is commands to allow imposter access.

rlogin and X have no way of encrypting its packets -- so no way to stop this!

26.7.3 Trojan horse

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

Trojan horse appears helpful, but really does something harmful

26.7.4 Salami attack

Superman 3 (terrible movie), but happened in real life.

Idea was to build up a hunk one bit at a time. What do you do with partial pennies from interest?

Bank keeps it! This guy 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!

26.7.5 Eavesdropping

Listener -- tap into serial line on the back of the terminal, or onto Ethernet. See everyting 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.

26.8 Concrete Examples

26.8.1 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, 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

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

a|aaaaaa

in memory| 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, rest on disk. Try all second characters, until one is slow.

Means takes 256 * 8 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?

26.8.2 Internet worm

A few years ago, broke into thousands of computers over Internet. Son of the person who wrote the paper in the reader!

Three attacks:

1. dictionary lookup
2. sendmail
   -- debug mode, if configured wrong, can let anybody log in
3. fingerd
   -- finger tea@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, could overwrite stack, 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.

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!

26.8.3 Mitnick

Two attacks:

1. Misdirection: Identify sysmgrs' 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 legitmate (eg, by replacing source machine 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.

26.8.4 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 for #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, 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 backdoor.

3. Buggy helper applications. As with fingerd attack, *any* bug in either Netscape, or its helper applications (ghostview), can potentially be exploited by creating a Web page that when viewed, will insert a trojan horse.

26.8.5 Thompson's self-replicating program

Bury trojan horse in binaries, so 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 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.

26.9 Lessons

1. Hard to resecure 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.