Secure communication and computation

Hardware reality: insecure networks

- attacker can eavesdrop on data going over the wire
- attacker can modify data
- attacker can insert new data or messages
- attacker can delete data

- attacker can replay old messages (eavesdrop, then insert later)
- attacker can spoof identity, by pretending to send a message from your IP address
- man-in-the-middle attack: eavesdrop and delete the original message, insert a new message that pretends to be from the original sender

Secure communication

- confidentiality: attacker should not be able to understand data the sender sends
- authentication: assure receiver that the message is from the right sender
- freshness: attacker should not be able to replay an old request
- no denial-of-service (we don't know how to provide this yet)

Encryption

Encryption is the main tool used to provide secure communication

Basic idea

- encrypt(clear text, e-key) = cipher text
- decrypt(cipher text, d-key) = clear text

- encrypt and decrypt algorithms are usually public
- shouldn't be able to deduce d-key from (clear text, cipher text) pairs

Symmetric key encryption

e-key = d-key (i.e. symmetric)

- only sender and receiver know the key (sometimes this is called “secret key” encryption)

Analogous to writing data on a floppy and placing the floppy inside a box with a padlock, then sending the box to the receiver via an untrusted courier. When the receiver gets the box and opens it, he/she knows:

- the true sender wrote the data on the floppy (authentication)
- nobody other than the sender has read the floppy (confidentiality)

Symmetric-key encryption algorithms are fast

E.g. I send message to registrar with a student’s grade

- encrypt(“B”, key)

- can someone modify the message?

How do sender and receiver get a shared secret key in the first place?

Public-key (asymmetric) encryption

e-key != d-key

Typically, encrypt() == decrypt() (we'll just call it crypt())

- crypt(clear text, e-key) = encrypted-text-1
- crypt(encrypted-text-1, d-key) = clear text
- crypt(clear text, d-key) = encrypted-text-2
- crypt(encrypted-text-2, e-key) = clear text

Note that

- encrypted-text-1 != encrypted-text-2
- crypt(encrypted-text-2, d-key) != clear text
- crypt(encrypted-text-1, e-key) != clear text

e-key is called the "public key"

- everyone knows the value of everyone's public keys

d-key is called the "private key"

- only the sender knows his/her own private key

Difficult to guess private key, even if you know the public key, crypt(), and lots of encrypted pairs

Using public-key encryption to authenticate sender

- "from pmchen" crypt(message, pmchen-private)
- anyone can read this message (no confidentiality)
- only pmchen can generate this message; others can verify that pmchen generated the message by decrypting with pmchen-public
- why include "from pmchen"

- this is called a "digital signature". Can detect any change to the data.

Using public-key encryption for privacy

- crypt(message, receiver-public)
- anyone can send this message (no authentication)
- only receiver can read it

Using public-key encryption for authentication and privacy

- crypt("From pmchen" crypt(message, pmchen-private), receiver-public)
- only receiver can read this; only pmchen can send it
- does the following work?
  "From pmchen"
  crypt(crypt(message, receiver-public), pmchen-private)

Public-key encryption used in

- SSL (secure sockets layer, used in web https)
- ssh (secure shell)
- pgp (secure mail)

Problems with public-key encryption

- more computationally expensive than symmetric-key encryption. Solve by using public-key to exchange a short-lived symmetric key (session key)
- how to change my public key?
- how to trust authenticity of published public keys?

E.g. A wants to communicate with B, so A and B must learn each other's public keys (A-public and B-public). Villain has two public keys V-public1 and V-public2.

- what if villain manages to convince A that B's public key is V-public1? And what if villain can convince B that A's public key is V-public2?
- A sends signed and sealed message with the wrong key: crypt("From A" crypt(message, A-private), V-public1)

How to authenticate the published public key?

pgp: verify the "fingerprint" of a public key via the telephone or a trusted web server

SSL example: your web browser wants to communicate with e-trade. You want to ensure that only e-trade can see your messages; e-trade wants to be sure that you are really who you say you are.

step 1: assure you that your messages can be read only be e-trade

- e-trade has public key, but how to learn this securely?
- certification authority (e.g. verisign) vouches for the authenticity of e-trade's public key
- e-trade sends you their public key, digitally signed by verisign: crypt("e-trade's public key is X", verisign-private)
- I decrypt with verisign's public key and see that verisign is vouching for e-trade's public key
- once I have e-trade's public key, e-trade and I can set up a shared session key (could be secret key)
- any problem with this?

step 2: assure e-trade that you are really who you say you are

- most clients don't have a certified public key from verisign
- you send your password (encrypted with the secret session key)
- e-trade decrypts with your password to verify that this session key is really from you.

Replay attacks

Example using symmetric-key encryption (same is possible with public-key encryption)

- I send message to bank using symmetric-key encryption encrypt("transfer $100 to U-M", key)
- evil U-M administrator eavesdrops and saves the encrypted message, then replays it later. Bank transfers **another** $100 to U-M.

How to defend against this attack?

How to pick a nonce that doesn't require anyone to keep any state (allow only probabilistic guarantee of freshness)?

General security

Hardware reality:

- collection of processor, memory, disks, network interfaces that can be used by anyone to do anything
- or could turn it off, leaving you with hardware that won't do anything for anyone

OS abstraction: **controlled** access to hardware

What primitives does hardware provide for controlling access?

OS will provide two abstractions on top of these primitives

- identity (authentication): who are you?
- security policy (authorization): what are you allowed to do?

Hardware already provides these two abstractions in a primitive way:

Authentication: who are you?

Authentication is the process of you proving your identity to the operating system. It may also include the operating system proving its identity to you.

Many ways to do authentication

Passwords

- a shared secret between the user and the OS
- what happens if villain gains access to the list of passwords?

  instead of storing the password, the OS can store a one-way function of the password

- what's the weakest link in a password system?

Authenticate based on a physical token (that can't be easily forged)

- e.g. your ticket to the football game
- but what if your token is stolen?

Authenticate based on both a physical token and a password

- e.g. your ATM card plus your PIN
- PIN is small so it's easy to remember. Limit guessing by disabling card after small number of guesses

Authenticate based on biometric token

- e.g. retina, thumbprint, signature

How do companies authenticate customers?

Authorization: what can you do?

Access control matrix

	file1	file2	file3
user1	rw	rw	rw
user2	-	r	rw

Two approaches for how to store this information: access control lists (ACLs) and capabilities

Access control lists

- at each object, store a list of who can access the object and in what ways they can access it
- e.g. at file2, store <user1 rw; user2 r>
- on each access, check that the user (whose identity is authenticated a login time) has permissions to access the file
- can make things more convenient by having user groups. e.g. aprakash, pmchen both belong to the "faculty" group, and file3 could have ACL <faculty, rw>
- villain can attack ACL systems by fooling the system into thinking he is someone else. E.g. sendmail runs as root. Attacker can subvert sendmail and get it to run attack code. System allows arbitrary access, because the system thinks this code is root's code.

Capabilities

- at each user, store a list of objects the user is allowed to access and how they are allowed to access it
- e.g. at user2, store <file2 r, file3 rw>
- on each access, check that the user has a capability for this type of access
- possession of the capability gives the power to access the file
- capabilities are like car keys. If you possess the door key to a car, you have the power to enter the car. If you possess the ignition key to a car, you have the power to drive the car.
- villain can attack a capability system by forging a capability (especially since capabilities are stored at the user).
- solution is to make the capabilities self-authenticating (unforgeable). e.g. the capability might include encrypt(file name, system key).

How does the owner of an object revoke permissions for a user in an ACL system?

How does the owner of an object revoke permissions for a user in a capability system?

Principles for secure systems

Least privilege

- only give users the minimal privilege they need to work
- e.g. when running MS Outlook, the attachment viewer should not have the ability to send out e-mails to everyone in my address book

Minimize the trusted computing base (TCB)

- in every system, some part is "trusted". E.g. the code that checks the ACL
- the TCB should be as small and simple as possible (minimizes bugs and loopholes)
- in current operating systems, the entire kernel is trusted. A special user (root or administrator) is also trusted. PCs typically trust most users with more privileges.

Defense in depth

- have multiple levels of defense, with each level checking the others. For a villain to break in, they have to break all levels
- some levels might serve to log the actions of other levels

Common attacks

Hidden channels

- attacker gets the system to communicate information in ways that the designers didn't anticipate
- e.g. Tenex thought to be very secure. To demonstrate, give a "red team" all source code and a normal account. 48 hours later, red team had all passwords!

```
password checker
   for (i=0; i<8; i++) {
      if (input[i] != password[i]) {break;}
   }
```

- goal is to force attacker to try 256^8 passwords
- how to break this?

- another hidden channel is power consumption

Buffer overflow

- program reads input into an on-stack buffer. Program fails to check the length of that input
- villain can give a long input and corrupt your stack. If they corrupt the return address on the stack, they can force the program to jump to their code.

Trojan horse

- give somebody that is apparently useful, but have it do something evil
- e.g. replace the login program to e-mail your password to the villain
- e.g. send someone a Word document (or an e-mail attachment) with a macro that runs when the document is opened (it runs with the user's identity)

Example

Ken Thompson (creator of Unix) wrote a self-replicating piece of attack code

goal: put a backdoor into the login program to allow "ken" to login as root without knowing password

1. make it possible (easy)
2. hide it (tricky)

Step 1: modify login.c

```
(code A) if (name == "ken") login as root
```

But this is really obvious to anyone looking at login.c How to hide the attack code?

Step 2: modify C compiler

```
(code B) if (compiling login.c) compile code A
         into binary
```

Now you can remove code A from login.c, and still have a backdoor. But this is now obvious in the compiler. How to hide the compiler attack code?

Step 3: distribute a buggy C compiler binary

```
(code C) if (compiling C compiler), compile
         code B into binary
```