Lecture sec2: Authentication

****
Review -- 1 min
****

Security mindset

- engineer v. security engineer
  - violate assumptions
  - Ken Thompson rootkit (machine is trustworthy)
  - Tenex passwords (interactions between subsystems; analog world side channels)
  - ATM bank->gas station (physical security)
- Why do computer systems fail?
- Broad principles
  - Robustness (Anderson)
  - Saltzer & Schroeder

****
Outline - 1 min
****

Authentication Basics

- principles: authentication, authorization, enforcement
- local authentication (passwords, etc.)
- distributed authentication (crypto)
- pitfalls: really hard to get right
-

****
Lecture - 1 min
****

1. Authentication

3 key components of security
Authentication – identify principal performing an action
Authorization – figure out who is allowed to do what

Enforcement – only allow authorized principals to perform specific actions

Principal – an entity associated with a security identifier; an entity authorized to perform certain actions

Authentication – an entity proves to a computer that it is particular principal

Basic idea – computer believes principle knows secret
entity proves it knows secret
→ computer believes entity is principal

1.1 Local authentication -- Passwords

common approach – passwords

advantage: convenient
disadvantage: not too secure

"Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations. (They are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations.)" – Kaufman, Perlman, and Speciner "Private communication in a public world" 1995

fundamental problem – Passwords are easy to guess

passwords must be long and obscure

paradox: short passwords are easy to crack;
long ones, people write down

technology → need longer passwords

Orig unix – 5 letter, lowercase password

how long to crack (exhaustive search) 26^5 = 10M
1975 – 10ms to check password → 1 day
1992 – 0.001 ms to check password → 10 seconds
2011 -- ??

Many people choose even simpler passwords
e.g. english words – Shakespeare's vocabulary 30K words
e.g. all english words, fictional characters, place names, person names, astronomy names, english words backwards, replace i with 1/e with 3, …

Implementation techniques to improve security

(1) Enforce password quality

e.g., >= 8 letters with mix of upper/lower case, number, special character
70^8 → 5x10^14 (10^7 times better than 6 lower case (?))

On-line check at password creation time (e.g., Require "at least X characters, mix of upper/lower case, include at least one number, include at least one punctuation, no substring in dictionary, …")

[Can do on-line check to get rid of really bad passwords. But if attacker is willing to spend 1 week cracking a password, do you want to wait a week before accepting a user password…]

Off-line checking …

BUT
except: people still pick common patterns (e.g. 7 lower case letters + 1 punctuation + 1 number)

(2) Don't store passwords

system must keep copy of secret to check against password. What if attacker gets access to this list of passwords? (design for robustness, right?)

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

solution: system stores only encrypted version, so OK even if someone reads the file!
When you type password, system encrypts it; compares encrypted versions

System believes principal knows secret
→ Store <principal> {Password}K

Entity proves it knows secret
→ Input password. System generates {Password}K and compare against stored value. If they match, input must have been password.

example: UNIX /etc/passwd file
    passwd → one-way transform → encrypted password

(3) Slow down guessing -- Interface

Passwords vulnerable to exhaustive search

Slow down rate of search
e.g.,

- Add pause after incorrect attempt
- Lock out account (or add really long delay) after k incorrect attempts

- **Slow down guessing – Internals**

Salt password file:

extend everyone's password with a unique number (stored in password file) so can't crack multiple passwords at a time (otherwise, takes 10sec to crack every account in the system; now have to do 1 at a time)

e.g., store <userID> <salt> <{password + salt}K>

(5) Think carefully about password reset protocol

(6) Implementation details matter…
-- e.g., tenex

1.2 Limits of passwords

These techniques help, and you should use them, but passwords remain vulnerable

- people still manage to pick poor ones (though seems to be getting better (anecdotal evidence; I don't have strong data)
- people re-use passwords across sites
- (some/enough) people give away passwords to "anyone" who asks
  - social engineering
  - phishing

1.3 2-factor authentication

Passwords limited by human capabilities

2 factor authentication:
Identify human by at least 2 of
(1) Something you know (secret e.g., password)
(2) Something you have (smart card, authentication token)
(3) Something you are (biometrics – fingerprint, iris scan, picture, voice, …)

Current state of the art – if you care about access control, you do something like this

- e.g., password + key fob

login: <username>
password: <password>
secureID: <number>
>

(Internally, key fob has a cryptographic key – think of it as a really long password + a clock; every k seconds compute f(key, time) → if you supply the right number for the current 30-second interval (+/- 30 seconds) then you must have the key fob)

Current state of art for authentication – 2 factor authentication

Key idea: Stealing key fob OR guessing password not enough

[Details:

Human knows password. Computer stores {password, salt}K1
Timer card and computer share secret key K2 and both have accurate clock and so know current time (30-second window). Card has a display window and displays {time}K2

User enters <userID> <password> <{time}K2>

Computer checks <password salt>K1
Computer checks <{time}K2>
]

Other examples;

- password + ssh login key (I know my password; I have my laptop that has my ssh key on it…)
- smart card + pin to activate it
- password + text message sent to my phone
- password + cookie on my browser (old computer v. new computer login paths…)

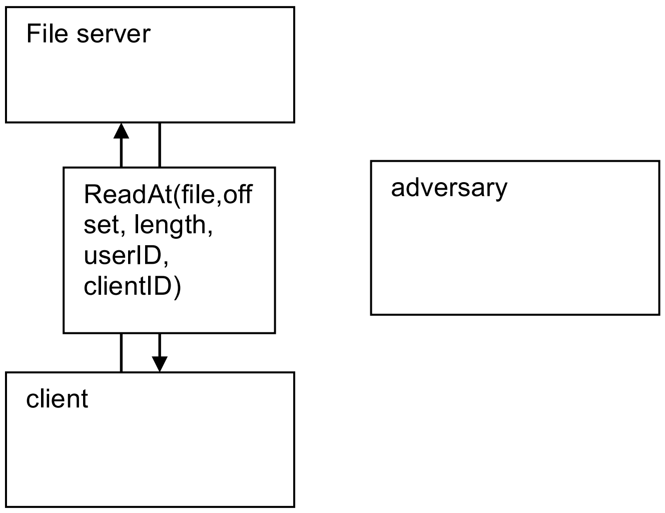
****
Admin
****

2. Authorization in distributed systems

Today, many/most services we rely on are supplied by remote machines (DNS, http, NFS, mail, ssh, …)

2.1 How not to do distributed authentication I

Consider authentication in distributed file system



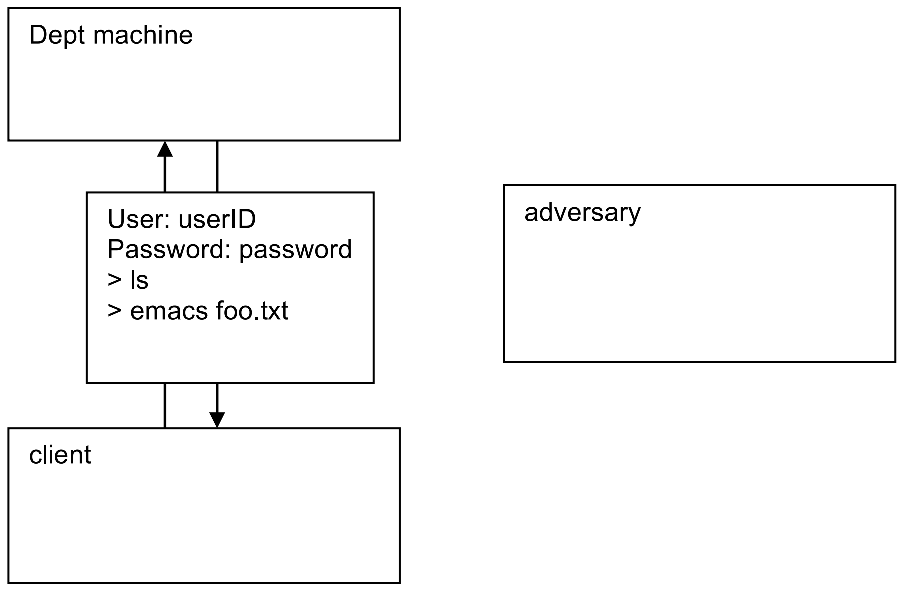
Adversary model
Typical assumption – we don't physically control the network so adversary can (a) see my packets, (b) change my packets, (c) insert new packets, (d) prevent my packets from being delivered

In some environments, this is a pretty good model of the adversary (I walk into a coffee shop that provides free wi-fi – their wifi router has nearly complete control over my network.) In other environments, we **hope** the adversary would have to work hard to get this much control (e.g., someone sitting next to me in a coffee shop might have to download some scripts to watch all of my network traffic and might even have to write some code to stomp on my wireless packets and replace them with their own if they want to modify my connection; e.g., department network – they might have to buy a ladder, a screwdriver, some cat-5 cable tools, and a $100 programmable router box)

Problems with the above protocol? Does it look familiar?

2.2 How not to do distributed authentication II

Consider remote login



Problems? Does it look familiar?

2.3 Solution: encryption

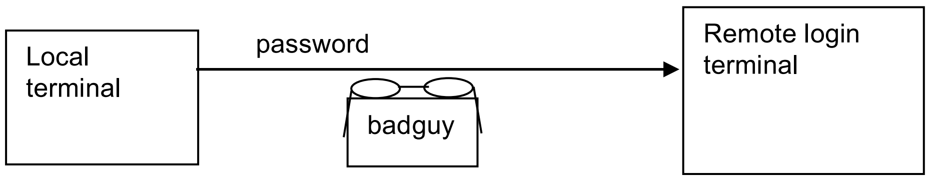
Two roles for encryption:

a) Authentication (+tamper resistance)
   Show that request was sent by someone that knows the secret w/o sending secret across the network
b) secrecy – I don't want anyone to know this data (e.g. medical records, etc.)

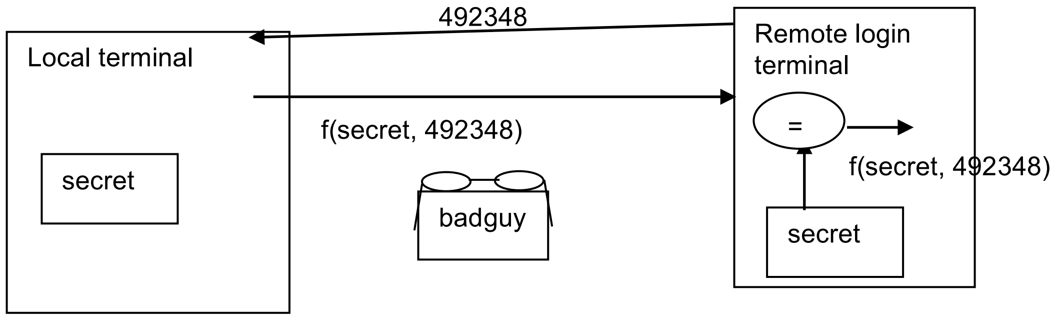
2.4 Network login

example: telnet login

sends password across the network!



solution: challenge/response



Compute function on secret and challenge

Common function: Cryptographic hash AKA 1-way hash (e.g., SHA-256)

Cryptographic hash easiest to understand under *random oracle* model

Random oracle cryptographic hash

- given any input, produce a truly random bit pattern of target length as output
- same input produces same output

properties h = H(x)

- Produce a **fixed length** array of bits h from variable-length input x
- given h and H, difficult to generate an x ;
- given x, H, and h, difficult to generate x' s.t. h' = H(x') == h;
- changing 1 bit of input "randomly" changes each bit of output
- → for above example, Can't learn secret from seeing network traffic; cannot predict correct response to a future challenge based on responses to past challenges

Example functions: MD5 (insecure), SHA-1 (borderline), SHA-256 (pretty good; current best practice)

NOTE: cheap to compute – 150MB/s SHA-1 on my 2GHz laptop (spring 2009)

Secret:

Typically, local terminal uses password to get secret

- Could use Unix approach – secret = encrypt 0 with password
    - Problem: dictionary attack via network
- Secret can be random string of 256 bits (much more random than password); encrypt secret with password and store on local terminal

Good news: Adversary doesn't learn my password
Bad news: Adversary can eavesdrop on my session
Bad news: Adversary can hijack my session (start sending what appear to be TCP packets from my session) and read or write any of my files!

Note: Above challenge/response protocol is simpler than typically used for login – generally have a stronger goal – login **and** establish encrypted connection

- not only do I need to send a token that proves I know a secret, I want to establish the ability to actually send new information (commands, data) to server

2.5 Encryption primitives

Cryptographic hash – see above
Secret key (symmetric) encryption
Public key (asymmetric) encryption

2.5.1 Private key encryption

encryption – transform on data that can easily be reversed given the correct key (and hard to reverse w/o key)

private key – key is secret (aka symmetric key)

(plaintext)^K → cipher text
(cipher text)^K → plaintext

from cipher text, can't decode w/o key
from plaintext, cipher text, can't derive key

Note, if A and B both know Kab, and A sends (X)^Kab, B just receives a random string of bits. How does B know which key to use? How does B know it got the right data?

- Low level protocol for (X)^Kab assumed to include sufficient redundancy for decrypter to know if it used a valid key on a valid message – magic number, checksum, cryptographic hash of message contents, ASCII text, …
- Typically, messages include a hint that helps receiver know what key to use (e.g., "A claims to have sent this message") Only a hint (if it is wrong, we might use wrong key and fail to decode the message (could try all of my keys) → impacts performance/liveness but not safety)

How big a key is needed?

56-bit DES key isn't big enough (was it ever?)
(DES – data encryption standard; federal standard 1976)
-- Michael Wiener 1993 built a search machine (CMOS chips)
$1M → 3.5 hours
$10M → 21 minutes
Key idea – easy to parallelize/build hardware – no per-key IO.
Just load each chip with "start key", "encrypted message", "plaintext message" an then GO

-- 2009 – assume costs halve every 2 years (conservative?)
$5K → 3.5 hours

$50K → 21 minutes

[[in fact 2006: FPGA COPACOBANA breaks DES in 9 days at $10K hardware cost; 2007 and 2009 improvements get this down to a day http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html]]

Worse: Don't throw the machine away after cracking one key!

2006 Cost per key (assuming 10 year operational life) $10000/(1 key/10 days * 365 days/year * 10 years/machine) → $27 per key

How big is big enough?

- adding 1 bit doubles search space. 2^128 is a big search space
- Brute force not feasible for decades (even assuming 2x/year);
  - At some point key sizes get large enough that you start seeing claims like "if every atom in the universe were a computer capable of testing a billion keys per second, then it would take X billion years…"
- Look for flaws in algorithm to restrict search space ("differential cryptography" "integral cryptography", "back doors", …)
- AES-128 and AES-256 are current "best practice" and believed to be quite secure

o Performance pretty good: AES-128 is 48MB/s on my 2008 laptop; AES-256 is 35MB/s

2.5.2 Public key encryption

public key encryption is alternative to private key – separate authentication from secrecy

2.5.2.1 Definitions and basics

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

(text)^K-public = ciphertext
(ciphertext)^K-private) = text

(text)^K-private = ciphertext'
NOTE: not same ciphertext as above!
(ciphertext)^K-public) = text

and
(ciphertet)^K-public != text
(ciphertext')K-private != text

and can't derive K-public from K-private or vice versa

Idea – K-private kept secret; K-public put in telephone directory

For example:
(I'm mike)^K-private
- ♦ everyone can read it, but only I can send it (authentication)

(Hi)^K-public
- ♦ anyone can send it but only target can read it (secrecy)

((I'm mike)^K-mike-private Hi!)^K-you-public
- ♦ only mike can send it, only you can read it
- ♦ **QUESTION**: Should this message convince you that "mike says hi?"
- ~~♦ E.g., public key crypto is orders of magnitude slower than private key crypto, so often the goal of a public key protocol is to do a "key exchange" to establish a shared private key. Suppose you receive~~
  ~~((I'm mike)^K-mike-private Use Kx)^Kyou-public~~
  ~~Should you believe that Kx is a good key to use for communicating with mike?~~
- ~~♦ Problem 1: Got the secrecy and authentication backwards – we know Kmike-private said "I'm mike" but we don't know~~

~~that it said anything about Kx!~~
~~Should have been:~~
~~((Use Kx)^Kyou-public mike you)^Kmike_private~~

- ~~Problem 2: freshness~~
- ~~Problem 3: how do you know Kmike-public?~~

~~You can build the above protocol using these as well.~~
~~But can get rid of key server~~
~~Instead, publish a dictionary of public keys~~
~~If A wants to talk to B~~
~~A->B (I'm A (use Kab)^K-privateA) ^K-publicB~~

~~Problem – how do you trust dictionary of public keys?~~
~~Trusted authentication service S~~
~~(Dictionary)^K-privateS~~

~~Kpublic-S is distributed by hand (or pre-installed on your computer – internet explorer, netscape)~~

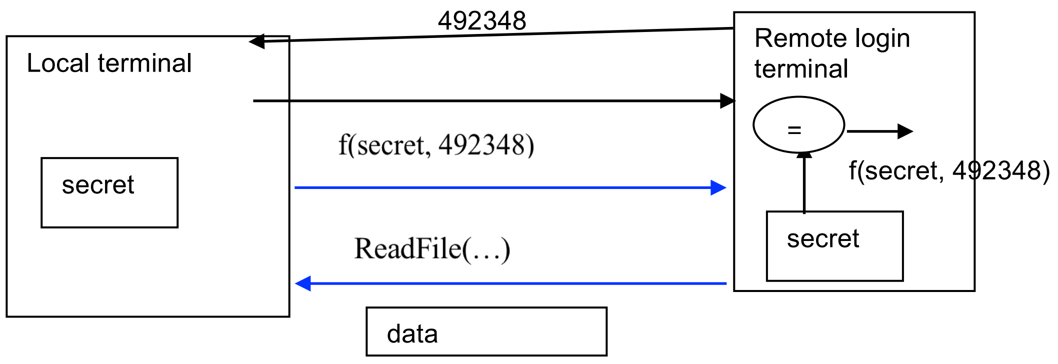
Performance is much worse than private key – RSA-1024 can do 170 sign/sec (about 5ms per sign) and 3827 verify/sec (about .3ms/verify) on my 2008 laptop

→ Often use public key crypto to set up shared, secret keys and then can have longer conversation using symmetric/private key encryption

2.6 Encrypted session

In distributed system, point is not just to prove "its me" but to issue some series of commands.

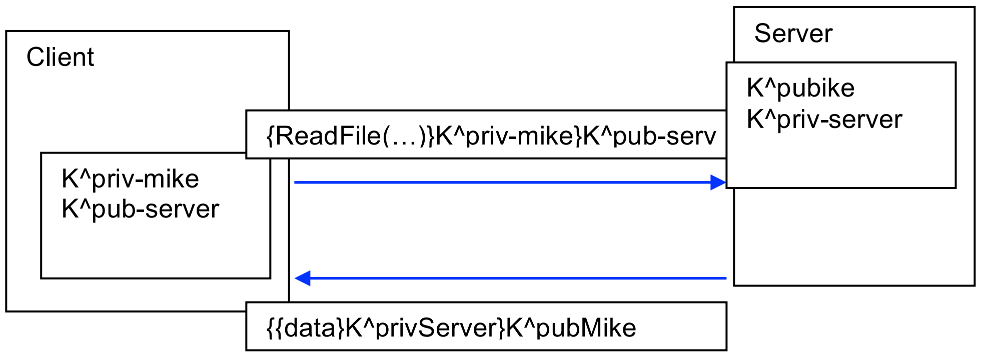
The above protocol can prove it is me. But then what?



What's wrong?

2.6.1 Example protocol (simplified)

I know K^private-mike and K^public-server and server knows K^public-mike and K^private-server


Client
K^priv-mike
K^pub-server
{ReadFile(…)}K^priv-mike}K^pub-serv
{{data}K^privServer}K^pubMike
Server
K^pubike
K^priv-server

2.6.2 Issues

3 problems with above protocol

(1) Initialization – how do I know K_public_server and how does server know K_public_mike?

a. Walk or pre-install list of all public keys on all machines
b. Certificate Authority can bind names to keys (pre-install certificate authority key on machines)

{BIND Mike Dahlin K_public_mike}K_private_CA

(2) Slow – public key operations **slow**

a. **Authentication**: Sign hash of message not message
   {mike says [longwinded msg]}K_private_mike
   =
   mike says [longwinded msg] {H(mike says [longwinded msg])}K_private_mike
b. **Authentication + secrecy**: Use public keys to set up symmetric secret key (much faster) [**see below**]

(3) Freshness -- Vulnerable to replay attacks

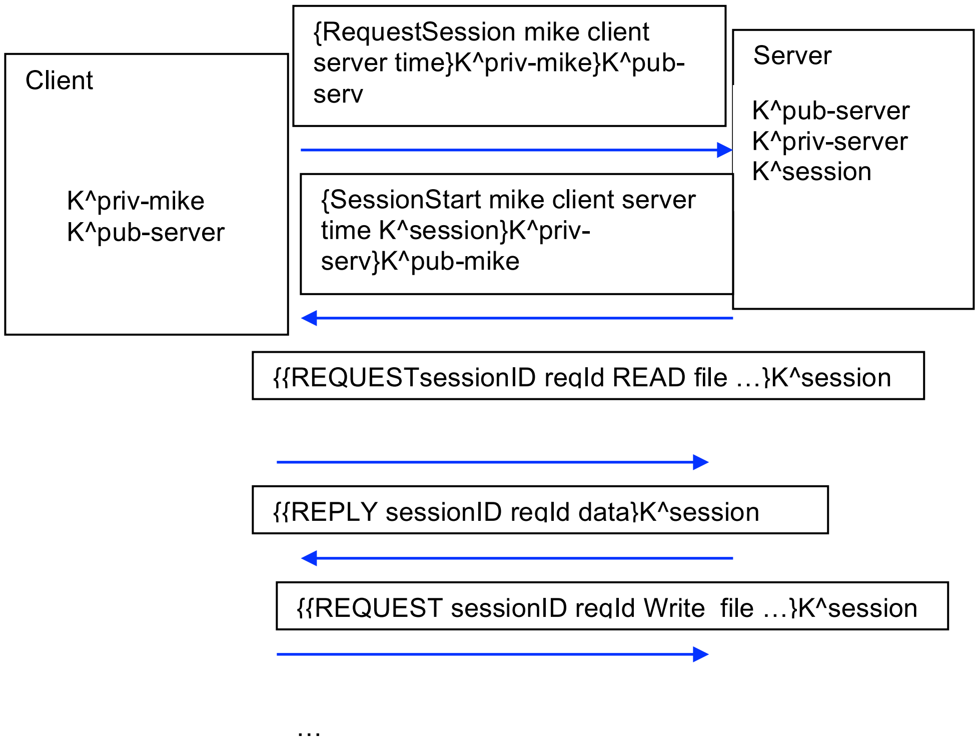
- attacker can resend old read request (for read, limited effect. What about command "buy 100 shares of IBM"?)
- attacker can send old read reply (how does client match requests to replies?}
- → Include timestamps or nonces in messages, expiration times in certificates

2.6.3 Example protocol (realistic)

(1) Exchange certificates

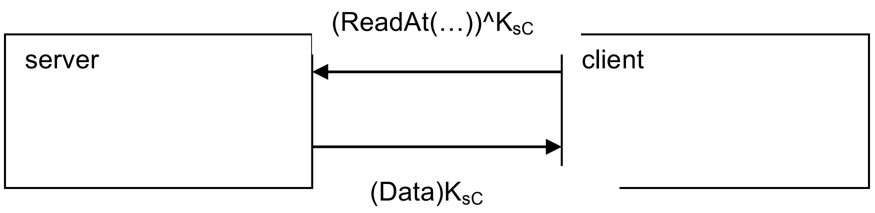
Client->server: {CA, K_pub-mike, mike, expires}K_priv-CA
Server->client: {CA, K_pub-server, server, expires} K_priv-CA

(2) Exchange private key


Client
K^priv-mike
K^pub-server
{RequestSession mike client server time}K^priv-mike}K^pub-serv
Server
K^pub-server
K^priv-server
K^session
{SessionStart mike client server time K^session}K^priv-serv}K^pub-mike
{{REQUESTsessionID reqId READ file …}K^session
{{REPLY sessionID reqId data}K^session
{{REQUEST sessionID reqId Write file …}K^session
…

~~2.7 Private key encryption~~

~~As long as key stays secret, get both secrecy and authentication~~



~~How do you get shared secret to both sender and receiver~~
~~Send over network? Not secret any more~~
~~Encrypt it? With what?~~

2.7.1 Authentication server (example: kerberos)

We can do something similar without public/private keys and certificate authority; do require trusted authentication server;

Authentication server -- server keeps list of passwords, provides a way for two parties, A and B, to talk to one another (as long as they trust server)

e.g., Kerberos (and varients) widely used (Microsoft, nfs, …)

Notation:

Kxy is key for talking between x and y

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

Results

Each client machine still needs to know a key for communicating with authentication server But no longer need to know a key for each service

This "master key" distributed out of band (e.g., sneaker-net or at machine installation time)

- master key plays same role as certificate authority did in public-key crypto
-

Store master key Ksa locally at A encrypted with A's password

→ only A can get Kab from S

[[Same for Ksb, B]]

Example: **Needham Schroeder Protocol** (precursor to Kerberos)

Step 1: A->S: A, B, N_A // *N_A is a nonce*

Step 2: S->A: {N_A, B, K_AB, {K_AB, A}K_BS}K_AS

Step 3: A->B: {K_AB, A}K_BS

Step 4: B->A: {N_B}K_AB

Step 5: A->B: {N_B – 1}K_AB

Step 1: A ask server for key to talk to B

Step 2: Server sends key to A (encrypted with K_AS) and ticket that B can use to get key

→ Now A believes it has the key
Step 3: A send the ticket to B
→ Now B believes it has the key (?)
Step 4 – 5: A and B handshake nonces to make sure they are both currently talking to each other (?)

Claim: At end of protocol, A knows it is talking to B and vice versa

Q: What's the problem with this?
A: Message 3 is not protected by nonces → no way for B to conclude that the K_AB it receives is the current key
→ Example attack: I am a disgruntled employee. Before I get fired, I run the first few steps of the protocol a bunch of times, gathering up a bunch of tickets {K_AB, A}K_BS for all of the servers B in our system (mail server, file server, database, …). → After I get fired, I can continue to log into all of the company's servers

Whoops.

Needham and Schroeder are really smart. They were quite experienced and careful system builders. They had lots of smart colleages. This protocol got published and lots of attention from experts. Several years later, the flaw was discovered.

[[We started security with "be afraid"; we end the same way…]]

Conclude: Arm waving by whiteboard is not enough for authentication protocols

Right answer: Formal analysis

- Needham and Schroder were not happy that they could have missed a bug. → Burrows Abadi Needham (BAN) logic provides better, formal way to reason about these protocols.
- Improvements since then by others with new logics
- No time to teach BAN logic or successor formal analysis tools today (but not too hard – if I had a full day I would…)

- Basic intuition:
  - {msg}K_x → I believe X said msg
  - {msg nonce}K_x → I believe x believes msg
  - x has authority over msg + above → I believe x
  - [[step through belief progression in protocol]

Additional answer: Informal analysis, prudent engineering
Hint for reading crypto protocols
(1) Ignore the "X → Y" part – a hint only; but you are assuming that adversary can forge headers, intercept communication, etc, so the meaning of a message can only depend on the contents not on who (claims to have) sent it
(2) Interpret "{X}^Ky" as "y (the holder of key Ky) once said X" (then you need to decide if the message is *fresh* (y recently said X) and whether you believe X (y has authority over X)

See Burrows, Abadi, Needham "A logic of authentication" http://www.cs.utexas.edu/users/dahlin/Classes/GradOS/papers/p18-burrows.pdf
(3) Always include **everything** needed to interpret message **in** message (don't rely on "previous messages" in protocol b/c adversary might reorder them and/or use messages from previous round of protocol (e.g., above – suppose we get rid of "A" and "B" in ticket)

Example: Kerberos (simplified)
A asks server for key
Step 1: A → S: A B time // Hi! I'd like key for AB

Server gives back special "session" key encrypted in B's key:
// S says to A "use Kab for communication between
// A and B {A B Kab}^Ksb"
Step 2: S → A: {A B time Kab {A B time Kab}^Ksb}^Ksa

A gives B the ticket
// S says to B "use Kab for communication between
// A and B"

Step 3: A → B: {A B time Kab}^Ksb

Details

1) Add in timestamp to limit how long a key will be used
(to prevent a machine from replaying messages later)

2) want to minimize # of times password must be typed in, and minimize amount of time password stored on machine → initially ask server for temp password, using real passwd for authentication

A→S (give me temp secret)
S→A (A use Ktemp-sa for next 8 hours)^Ksa

Can now use Ktemp-sa in place of Ka above

3. Authorization

authorization – who can do what?

Access control matrix: formalization of all permissions in the system

	file1	file2	file3	…
userA	rw	r	--	
userB	--	rw	--	
userC	rw	rw	rw	

potentially huge # users, objects → impractical to store all of these

2 approaches

1) access control lists – store all permissions for all users with each object
still – might be lots of users! Unix approach - have 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

Example – page tables – each process has list of pages it has access to (not each page has list of processes that are peritted to access it)

4. Enforcement

enforcer checks psswords, access control lists, etc

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

In UNIX, superuser has all powers of the kernel - can do anything. Because of coarse-grained access control, lots of stuff has to run as superuser in order to work. If a bug in any of thse 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 needed
   hard to get right

…

****

Admin - 3 min

****

-

****

Lecture - 25 min

****

5. State of the world in security

ugly

Authentication – encryption
  but almost nobody encrypts

Authorization – access control

but many systems provide only coarse-grained access contrl (e.g. UNIX file – 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

6. Classes of security problems

6.1 abuse of privilege

if superuser is evil, we're all in trouble

no hope

6.2 imposter

break into system by pretending to be someone else

example – 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

6.3 trojan horse

one army gives another a present of a wooden horse, army hidden inside

trojan horse appears to be helpful, but really does something harmful

e.g. "click here to download this plugin"

6.4 Salami attack

superman 3 (terrible movie) but happened in real life

idea was to build up hunk one bit at a time – what do you do with partial pennies of interest?

Bank keeps it! This guy re-programmed it so that partial pennies would go into his account. Doesn't seem like much, but if you are Bank of America, add up pretty quickly.

This is part of why people are so worried about credit cards on internet. Today – steal credit card, charge $1000 – credit card company, merchant, owner notice
Tomorrow – steal 1000000 credit cards, charge $1; no one notices

6.5 Eavesdropping

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

…

7. Examples

7.1 Tenex – early '70s BBN

Most popular systems at universitives before Unix

Thought to be v. secure. To demonstrate it, created a team to try to find loopholes. Give them all source code and documentation (want code to be publicly available as in Unix). Give them a normal account

in 48 hours, had every password in the system

Here's the code for the password check in the kernel:

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

Looks innocuous – have to try all combinations – 256^8

But! Tenex also had virtual memory and it interacts badly with above code

Key idea – force page fault at carefully designed times to reveal password

Arrange first character in string to be last character in page, rest on next page. Arrange that the page with first character in memor, and rest on disk

    a|aaaaaa

Time how long password check takes

    if fast – first character is wrong
    if slow – first character is right; page fault; one of 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
…

    → takes 256 * 8 to crack password

Fix is easy – don't stop until you look at all characters
But how do you figure that out inadvance?

Timing bugs are REALLY hard to avoid!!

7.2 internet worm

1990 - broke into thousands of computers over internet

Three attacks

1. dictionary lookup
2. sendmail

--debug mode – if configured wrong, can let anyone log in

3. fingerd

-- finger dahlin@cs

Fingerd didn't check for length of string, but only alocated 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!

Go caught b/c idea was to launch attacks on other systems from whatever systems were broken into; so ended up breaking into sae machine multiple times, dragging down CPU so much that peopl noticed

variant of 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 but before use!

7.3 Mitnick

Two attacks:

1) misdirection: identify system mgrs machines, then loop, requesting TCP connections to those machies until no more connections are permitted → freeze machine

2) Imposter: forge packets to appear as if legit (e.g. replace source machine in packet header) but really from Mitnick

   hijack open, idle rlogin connection. E.g. send packets as if user typed command to add mitnick to .rhosts file

7.4 Netscape follies

1995-6

Netscape wants to provide secure communication so you can send credi card number over internet

3 problems

1) algorithm for picking session keys was predictable (used time of day). Brute force allows someone to break key in a few hours

2) netscape makes new version to fix #1; make available over internet (unencrypted). Modify netscape executable w/ 4-byte patch to make it always use specific key – so can insergt backdoor by mangling packets containiing executable as they fly by on internet

In fact, because of demand, had dozen mirror sites (including Berkeley, ..) to redistribute new version. So anyone with root access to any machine at Berkeley CS could insert backdoor to netscape

3) buggy helper applications

As with fingerd attack – any bug in either netscape or in helper application (ghostview, mplay, …) can potentially be exploited by creating a web page that when viewd will insert a trojan horse

e.g. postscript is a full-featured language, including commands to write to disk!! So send a postscript file that says "write(dahlin, rhosts)

7.5 Timing, environment

Computer designers design to make sure that software interfaces are secure. But software runs on hardware in the real world…

(a) smart card power supply analysis
(b) Tempest – your monitor (and keyboard) is also a radio transmitter – relatively easy to build a device that can receive radio broadcast and display what your monitor is displaying from several feet away
(High end attack: irradiate the subject machine at resonance frequency of keyboard cable → pick up keystrokes from 50-

100yards. Some speculate this is why the USSR constantly beamed radar at the US embassy in Moscow for a while…)

(c) Traffic analysis – e.g., you encrypt your web traffic over network so know one knows what you are browsing. But they see 14321 bytes, pause, 29140 bytes, pause, 2341 bytes, pause… Pretty quickly they can match what pages you are viewing to a suspect website with high confidence

(d) …

7.6 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 on new platforms. Almost invisible

gave Ken thompson the ability to log into any Unix system I the world

2 parts
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
```

ida is: hide change so no one can see it

step 2: modify C compiler

instead of having code in login, put it in compiler:

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

Whenever the compiler sees a trigger /* gobbleygook */,
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 problem in the compiler

step 3: modify compiler to have

```
C:
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 compiler with C present

- now in binary for compiler

step 5: replace code with trigger2

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

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

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

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

8. Lessons

1. Hard to resecure after penetration

What do you need to do to remove the backdoor?
Remove all the triggers?
What if he left another trigger in the editor—if you ever see anyone removing the trigger, go back ad re-insert it!

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

Toggle in everything from scrtch every time you log in?

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 has happened; can't get rid of it.

****

Summary - 1 min

****

9. Major Topics

1) Memory management & address spaces ; virtual memory/paging to disk

Excellent example of "any problem can be solved with a level of indirection" -- virtual memory system allows you to interpose on each memory reference – translation, protection, relocation, paging, automatically growing stack, …

A bunch of data structures with funny names (base&bounds, paging, segmentation, combined, TLBs) but beyond the jargon – a few basic concepts, simple data structures (hash, tree, array, …)

Cache replacement – power tool: identify ideal algorithm – even if not realizable in practice – (1) improve understanding/help design good algorithms, (2) basis for evaluation

2) **Threads**: state, creation, dispatching; synchronization

Basic mechanism: per thread state v. shared state
Basic attitude: assume nothing about scheduler; have to design programs that are safe no matter what the scheduler does

Power tool: monitors (locks and condition variables) provide a "cookbook" approach for writing safe multithreaded programs. Don't cut corners

Open question: liveness – deadlocks, etc. Global structure of program (as opposed to modular safety)

Scheduling: shortest job first, round robin – specific policies not so important. Gain insight on trade-offs so you can develop your own.
Power tools: (1) Know your goals, (2) Analyze optimal case

3) **File systems:**
disk seeks, file headers, directories, transactions

Finding data on disk – again lots of jargon, but it comes down to arrays and trees and hash tables…
2 step process
name->ID/header
header->blocks of file

Reliability: transactions, undo/redo log
Power tool: Transactions are definitely a power tool!

4) **Networks**, distributed systems
RPC: It's simple…
Issues
Reliability: Lost messages, partitions, crashed machines
→ retry, 2-phase commit (distributed transaction)
Power tool: 2-phase commit

Performance: Caching, replication
Consistency/coherence across replicas – callbacks, polling, leases

5) **Security**:
attitude – robustness, big picture
access control, authentication, pitfalls

10. OS as Illusionist

Physical Reality	**Abstraction**
single CPU	infinite # of CPUs (multiprogramming)
interrupts	cooperating sequential threads
limited memory	unlimited virtual memory
no protection	each address space has its own machine
unreliable, fixed-size messages	reliable, arbitrary messages and network services

11. Problem Areas

1) Performance

- abstractions like threads, RPC are not free
- caching doesn't work when there is little locality
- predicting the future to do good resource mgmt

2) Failures
How do we build systems that continue to work when parts of the system break?

3) Security
Basic tradeoff between making computer system easy to use v. hard to misuse

Admin: Course evaluation
(1) Paper for javeetha
(2) Snafu: electronic for me

This year, an experiment
(1) Harder homeworks + HW discussion section
(2) Easier labs (related to (1))
(3) Lab discussion section (get rid of?)
(4) Changing name of class
(5) Topics – more networks, security, memory management