Lecture sec1: Intro to security: “The security mindset”

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

Distributed file systems
- Simple RPC
- Complexity
  - Performance
  - Caching, cache consistency
  - Fault tolerance
  - Security

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

Main point: you can’t trust computers
Goal: prevent misuse of computers

Outline
Today: Big picture
- o Why do computer systems fail
- o General principles
- o Key lesson: technical solutions alone insufficient; good designer needs to think about the big picture; need to consider how system will be developed, maintained, used
- o Other “outside of box” failures
  - o Can you trust your system?
  - o Can you trust your environment?

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

Real outline
- Your job is to think big picture. How your technology embeds in world

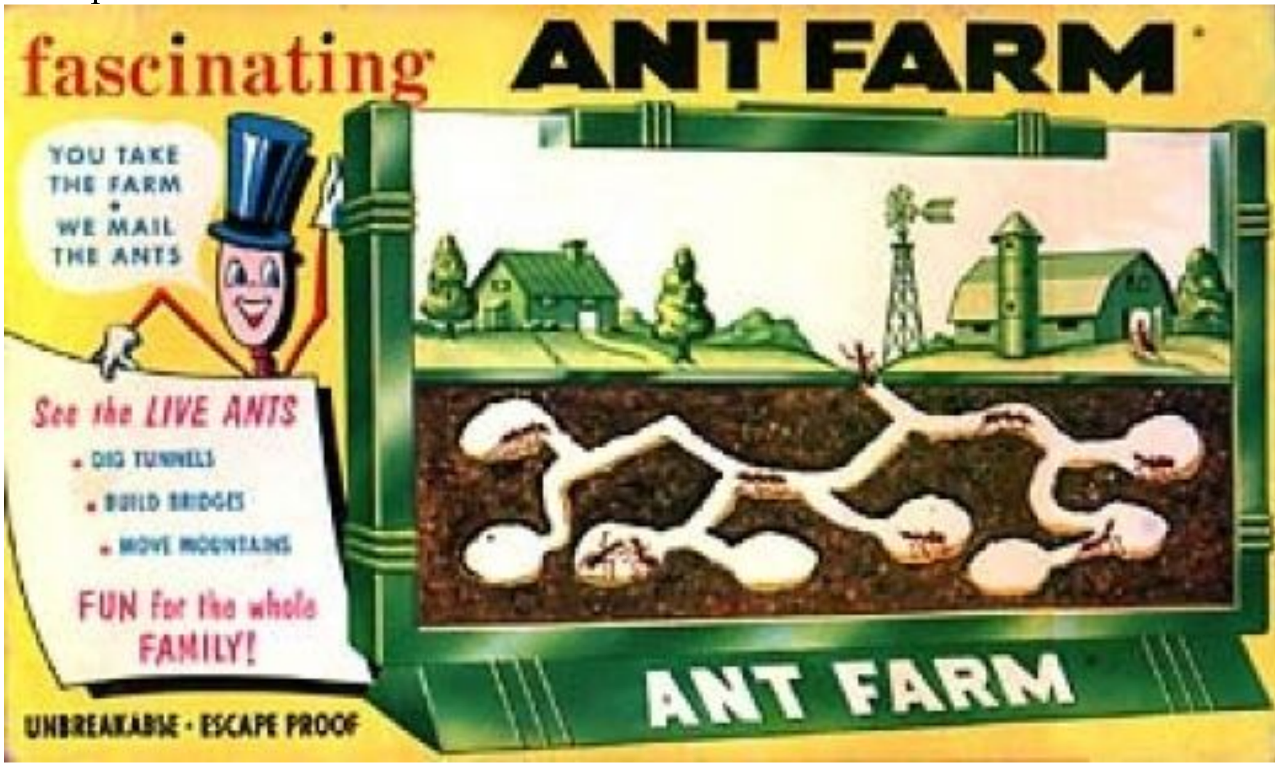
- Security is how to circumvent designers intent by violating designer's assumptions -- "Wow, I can send live ants to anyone in the country"
- Most security vulnerabilities are "outside" of technical design (or at least outside of any one module) but we spend lots of time worrying about technical stuff in our pieces)
- Business pressure to make insecure, unreliable systems (security, safety issues). Be responsible.

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

1. Security/reliability rant

Theme:
-- Security mindset

(1) Security = *break assumptions*
Example: Ant Farm



engineer: “What an elegant solution to the problem”
security engineer: "I can send live ants to anyone"

problem: Managing complexity in programming is all about abstraction layers

example: Tenex (we)hink of functions as black box; reality – they run on computers()

example: Tempest (we think of computers as things that process binary bits; reality – physics)

Many dimensions
E.g., for security "what parts of system can affect my security" (e.g., postal service/address change)

(2) Security/safety/good design = *how does technology embed in real world*

Many/most failures non-technical

- But technologists like to focus on technical stuff
-
- Anderson: top 3 reasons for ATM phantom withdrawals -- (1) background noise (big system, complex interactions), (2) postal interception, (3) theft by bank staff
- Anderson: Bank response (UK): Blame user
-

Many/most failures cross-layer

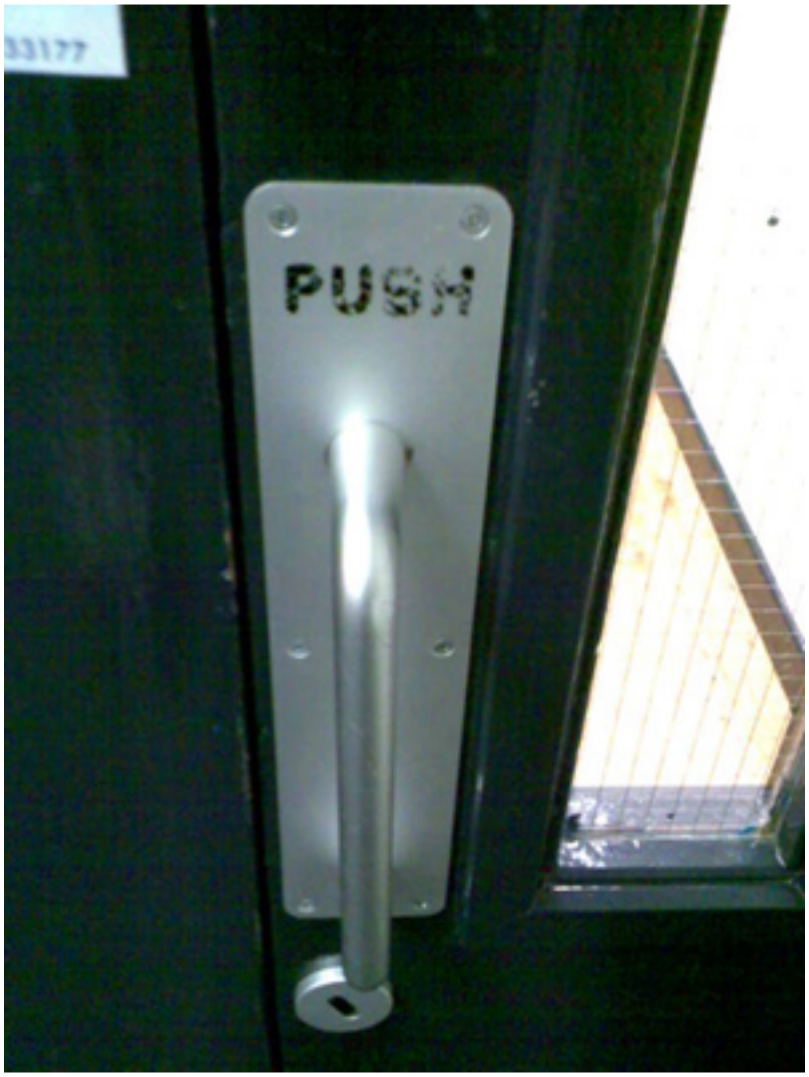
- But engineers like to focus on their little piece

(2b) end-to-end design includes user, environment, other programmers (API to your modules)

If security problems come when one layer breaks assumptions of another layer, then important to think about interfaces. Make it hard to misuse your layer; make it obvious how to use your layer correctly.

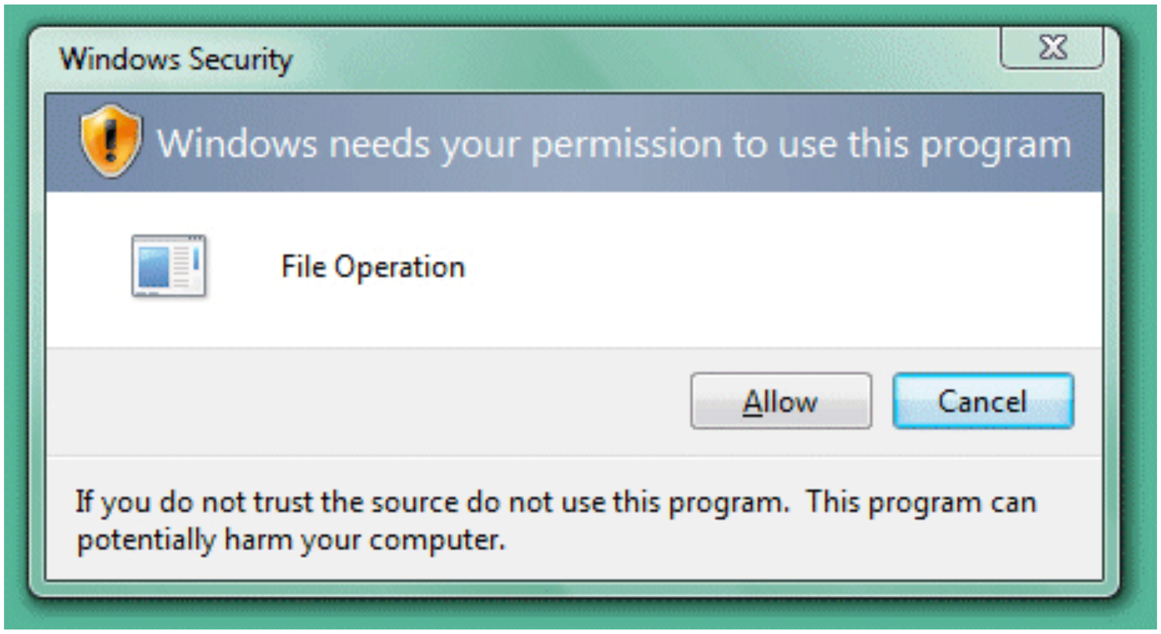
Not just software-software interfaces but also user/computer interfaces (blaming the user is not a security strategy)

non-example: A door with an instruction manual



example: How to label candy machine items

non-example: "Click here to get work done"
Jeff Atwood -- http://www.codinghorror.com/blog/2006/04/windows-vista-security-through-endless-warning-dialogs.html


Windows Security

Windows needs your permission to use this program

File Operation

If you do not trust the source do not use this program. This program can potentially harm your computer.

"The problem with the **Security Through Endless Warning Dialogs** school of thought is that *it doesn't work*. All those earnest warning dialogs eventually blend together into a giant "click here to get work done" button that nobody bothers to read any more."

Dialogs train user to hit "OK" (security bugs, Therac)

Not OK to say "My part of system worked as designed, you're just using my system wrong." No -- you need to fix your system to make it harder to misuse.

Many dimensions
-- UI (user v. system)
-- API (programmer v. module)
-- Control of physical environment (ATM spy cameras; fake ATM machines, ...)

-- Operations (training, monitoring, response, diagnose, refine design)
    -- When you get bitten by a bug, figure out how not to get bitten again (e.g., fix bug)
    -- When you get bitten by a bug, figure out how not to build a similar bug into future systems (e.g., programmer, fix thyself)
-- Human resources (background check, firing process, ...)
-- Insider attack (auditing, division of responsibility, ...)
...

Not just "does my code protect against stack smashing attacks" and "do I encrypt data on the network"

(2c) Ethical engineer

You will be pressed to
-- ship with bugs
-- ship with security bugs
-- ship with safety bugs

Hard to stay "stop the presses"
Not always right to say "stop the presses"
    -- Your job is to work to find a solution not just to say no
(But also hard to estimate risks when systems embedded in real world)

With these caveats in mind, there are techniques we can use to make it more likely the systems we build are sufficiently secure

Next couple weeks, discuss some of them.

Tiny subset – even a full semester security class can't cover everything…

- today: big picture
- Tuesday: authentication
- Wednesday: TBD

2. Security – problem definition

Types of misuse
1) accidental
2) intentional

protection is to prevent either accidental or intentional misuse
security is to prevent intentional misuse

So far, class has focused on protection. Today focus on security

Security v. reliability
Reliability – system does what is supposed to do
Security – system only does what it is supposed to do (nothing else)

"Why Cryptosystems fail", Ross Anderson
Plug: **Security Engineering** by Ross Anderson
Lots of fun

- Standard stuff like Chapter 2 Protocols, Chapter 3 Passwords, Chapter 4 access control; but also…
- Chapter 11 – Nuclear command and control
- P 19 – the MIG in the middle attack
- P 267 Fingerprint identification in crimes "Even if the probability of a false match .. is one in ten billion as claimed by police, once many prints are compared against each other, probability theory starts to bite. A system that worked well in the old days, whereby a crime scene print would be compared manually with the records of 57 known local burglars, breaks down once thousands of prints are compared every year with a database of millions."
- P 291-295 "How to hack a smartcard (1-7)"
- Chapter 17 – telecom security (phone phreaking and mobile phones)

- … Lots of fun…

Lesson: A good computer security designer needs to be broad
(1) Learn from world around you
(2) Your systems must exist in the real world (payTV, road tolls, medical records, … not just login to workstation)

3. Problem

3.1 "Why cryptosystems fail"

basic story: technologists designed ATM system and focused on technical attacks – 'break crypto; intercept and insert network messages'

Essentially no recorded security breaches used this mode of attack (OK, 2 did; but tens of thousands of "phantom withdrawals" from other sources)

i.e., focus on complex sophisticated crypto attacks

-- Only 2 documented attacks on network messaging/crypto;

"High-tech threats were the ones that most exercised the cryptographic equipment industry, and the ones that their products were explicitly designed to prevent. But these products are often so difficult to integrate into larger systems that they can contribute significantly to the blunders that caused the actual losses."

Conclusion
"Our research showed that the organizational problems of building and managing secure systems are so severe that they will frustrate any purely technical solution."

3.2 Real reasons for failures (ATM networks)

"Almost all attacks on banking systems involved blunders, insider involvement, or both."

QUESTION: What were the three main causes?

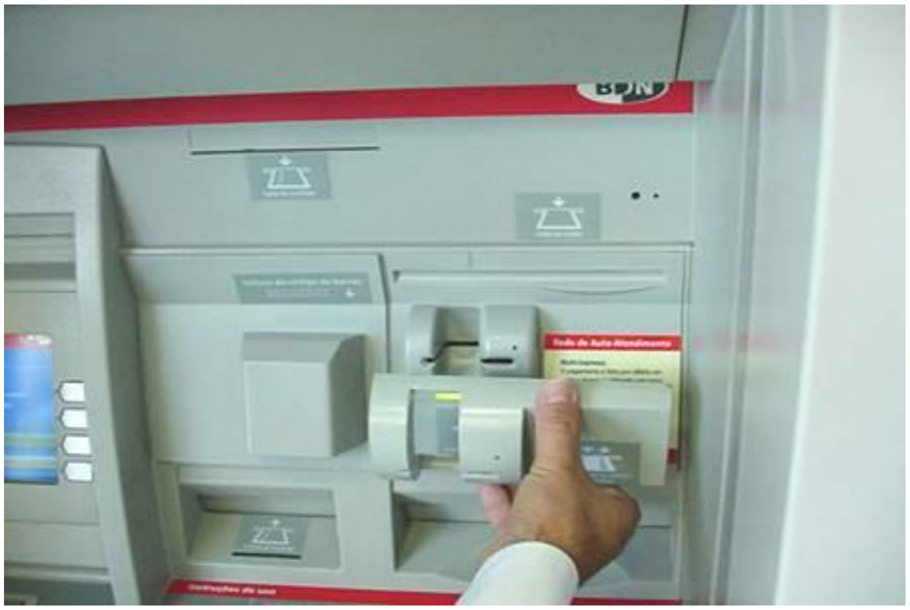
(1) Program bugs
  Lower bound on transaction error rate 1/10,000 due to program bugs for large, heterogeneous transaction processing system [note: cites a 1991 study here…has this number changed?]
  → 600K "phantom withdrawals" in US
  [Banks' response: phantom withdrawals clustered near residences of cards]
(2) Postal interception of cards
(3) Thefts by bank staff
  "British banks dismiss about 1% of their staff each year for disciplinary reasons…"

"More exotic attacks"
  Variations on: Acquire PIN and card #

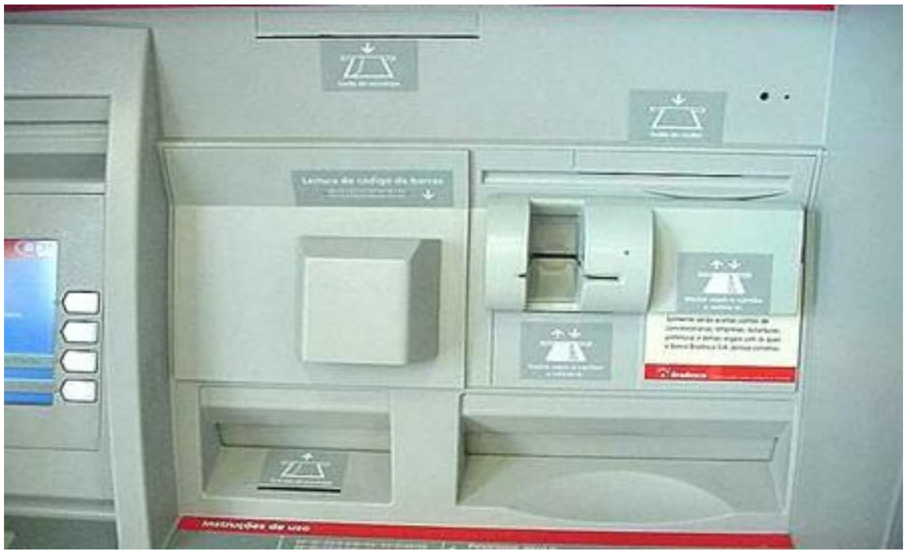
- Blunders: bad PIN generation, shoulder surfing + "telephone card" bug [invalid card makes ATM think old card was just re-inserted], encrypted PIN on magnetic strip not tied to account #, off-line ATM operation, test-mode password (→spits out 10 bills)

- Acquire card + pin: shoulder surfing + receipt, camera + scanner [recent attacks at UT], theft of card + PIN written down, bank staff gives replacement card and PIN, set up fake ATM machine file:///C:/Documents%20and%20Settings/dahlin/My%20Documents/research/notes/2005/4/atm.html

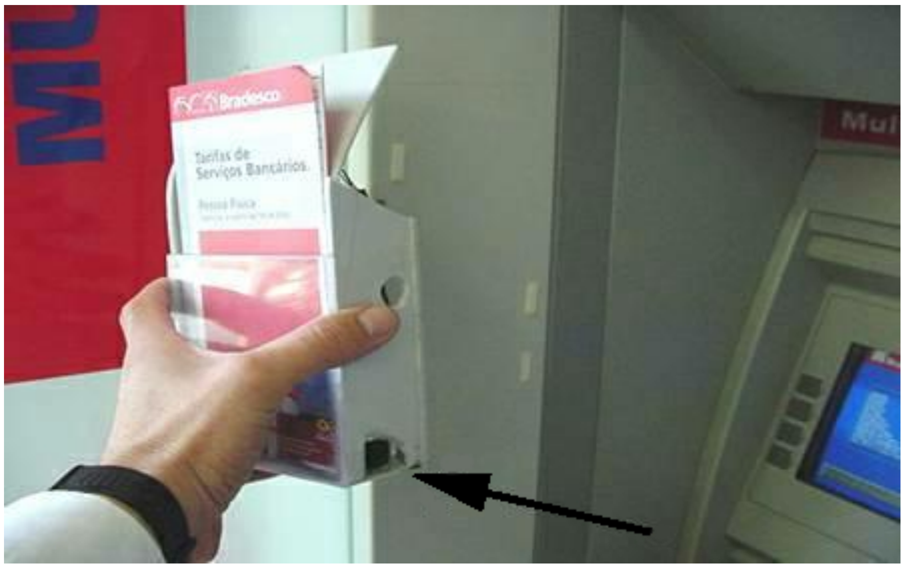
Example from UT (2005)



Equipment being installed on front of existing bank card slot.



The equipment as it appears installed over the normal ATM bank slot.



The PIN reading camera being installed on the ATM is housed in an innocent looking leaflet enclosure.

The camera shown installed and ready to capture PINs by looking down on the keypad as you enter your PIN.

-
- Possible help: Smart cards (but Anderson skeptical …) QUESTION: Why do smart cards help? Which of these attacks thwarted

3.3 Technical problems are hard too

Engineer testing: does system behave as expected over intended/expected operating conditions/workloads/input

Security engineer testing: Does system behave as expected over **all possible** operating conditions/workloads/input

Asymmetry attacker v. defender

Defender needs to find and fix all bugs. Attacker needs to find and exploit one bug.

Doomed if you do, doomed if you don't:

"Designers…have suffered from a lack of feedback about how their products fail in practice, as opposed to how they might fail in theory. This has led to a false threat model being accepted; **designers focused on what could possibly go wrong rather than what was likely to, and many of their products ended up being so complex and tricky to use, they caused implementation blunders that led to security failures.**" (emphasis added)

-- try to cover every possible failure → system complex; implementation errors will make system insecure

-- focus on common failures → missed attacks will be exploited

Black box system

How do I know this program I just downloaded will do what it says it will.

For that matter, how do I know that this phone isn't secretly recording every conversation I have?

How does the government know that the software in the plane will work as intended? What about the hardware?

Example: First rootkit

3.4 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 in 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
```

idea 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:

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.

Lessons

- Abuse of privilege is a hard problem
- Once system compromised, you are in BIG trouble (not robust!)

3.5 Tenex – early '70s BBN (discussed above)

Most popular systems at universities 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 – 26^8

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!!

Broad principle: computer scientists think in digital world – what bits go back and forth;

- Attackers succeed when they violate designer's model of world
- Analog v. digital – timing, tempest
- Environment – ATM designed in '70's to be deployed by banks in banks; now deployed by ??? in gas stations; reasonable trade-offs in closed, mainframe environment may no longer work in portable laptop environment…

3.6 See notes for other examples…really hard to build secure system

4. Solutions (principles)

Be broad; understand big picture;
Valuable to be able to know technical means well but also be able to apply them to business problems (easy to make a system secure. **Hard** to make a system usable and secure.)

4.1 Broad principles (Anderson)

(1) Don't lose sight of the big picture and focus just on sophisticated technical means of attack; know why real systems get compromised and recognize that a system that is more complicated to design or use may be significantly less secure in practice than a simpler system with "weaker" technological safeguards

(2) "Moral hazard" -- The entity responsible for verifying authentication should be the one that pays the penalty for authentication errors.

Special cast/higher-level application of "(2a) e2e design"

-- societal/incentives

"cost should be borne by party that is in a position to prevent/fix problems"

Very general principle –
example: accepting a signature (nod, handshake, click, fax, sign, notarize, guarantee)

Traditional business practice for handwritten signatures "In general, if someone wishes to enforce a document against you on the basis that you signed it, and you deny that you signed it, then it

is for them to prove that the signature was made by you or authorized by you." [_Security Engineering_ p 483]
→ balance convenience v. security [continuum: handshake deal, "press 9 to agree", faxed signature, original signature, compare original signature against reference [e.g., on back of card], witnessed signature, notarized signature, bank signature guarantee]

example: ATM (customer v. bank)

"If…the system operator carries the risk, as in the United States [for ATM transactions], then the public-interest issue disappears, and security becomes a straightforward engineering problem for the bank (and its insurers and equipment suppliers)."

example: credit card

(why no "chip and PIN" in US?

example: Diebold voting machine v Diebold casino machine

Note: Legal/policy issue here – natural for entity to try to transfer the legal risk to the other party

- Efforts to create a technical digital signature standard s.t. signatures are presumptively genuine
- Phone model v. credit card model – you are generally liable for all charges made to your phone (→ GSM security is primarily there to prove that a call made by a customer **not** to protect your account from having minutes stolen…)
- …

QUESTION: How does incorrect moral hazard definition in Britain contribute to three main causes of ATM phantom withdrawals.

4.2 Suggested solution/approach (Anderson)

"Caveat…problems…are so severe that they will frustrate any purely technical solution."

Robustness and Explicitness

Robustness

Goal: Robustness – provide resilience against minor errors in design and operation and component failure

"Aircraft engineers are also aware that accidents usually have multiple causes, while security researchers tend to use threat and risk models in which only one thing goes wrong at a time. Yet in the majority of ATM frauds, the cause was a combination: carelessness by insiders plus an opportunistic attack by outsiders (or by other insiders.)"

Explicitness

- o Explicit goals
- o Protocols that make their requirements and assumptions explicit (and "that cannot be fitted together in unsafe ways")
- o Techniques from safety-critical systems (formal methods, sw engineering, ….)

Advice

(1) Explicitly list failure modes
"The specification should list all possible failure modes of the system. This should include every substantially new accident or incident that has ever been reported and that is relevant to the equipment being specified."

(2) Explicitly state how failure modes addressed
"[the specification] should explain what strategy has been adopted to prevent each of these failure modes, or at least make them acceptably unlikely."

(3) Explicitly state implementation plan – both technical and managerial
"[the specification] should then spell out how each of these strategies is implemented, including the consequences when each single component fails. This explanation must cover not only technical factors, but training and management issues too. If the procedure when an engine fails is to continue flying with the other engine, then what skills does a pilot need to do this and what are the procedures whereby these skills are acquired, kept current, and tested?"

(4) Explicitly test specification; analyze failures
"The certification program must include a review by independent experts, and test whether the equipment can in fact be operated by people with the stated level of skill and experience. It must also include a monitoring program whereby all incidents are reported to both the equipment manufacturer and certification body."

5. Basic principles – Saltzer and Shroeder

Saltzer and Schroeder: "Protection of information in computer systems"

Broadly used checklist for computer security design

1. economy of mechanism (keep design simple; integrate pieces into a whole)
   small TCB – trusted computing base
   -- volume (TCB is simple; few lines of code; simple design…)
   -- surface area (interfaces are narrow and simple)

   e.g., hypervisor + domain 0 OS may be safer than OS alone (more volume but less surface area)

2. Fail-safe defaults – default should be no access; explicitly grant access

3. Complete mediation – all requests should be checked

4. Open design – security is a function of explicit secrets not on obscurity of design or algorithm; systems is secure even against someone who knows its design

v. security by obscurity

(But, still a place for obscurity as added barrier)

5. Separation of privilege – "Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key"

Common approach – "authentication from a combination of **something they have** [smart card, fingerprint, retinal scan, trusted machine] and **something they know** [password, pin, …]"

6. Lease privilege – every program and user should operate using the least set of privileges needed to do the job

7. Least common mechanism – minimize mechanism depended on by all users (avoid single point of failure, allow flexibility to customize mechanism to different requirements)

8. Psychological acceptability

QUESTIONS
(1) Are there conflicts between any of these? (→ engineering judgement/trade-offs)
(2) Compare these issues to Anderson robustness theme (tolerate minor failures or avoid misuse/misdesign) how do principles relate to robustness?

6. Other case studies

Some classic attacks
How do they relate to robustness themes?
How well would 4 strategies from Anderson or 8 rules from Saltzer address these issues?

6.1 abuse of privilege

if superuser is evil, we're all in trouble

no hope…

Problem magnified by requiring lots of programs to have superuser privilege (more programs → more opportunities for insider; more exploitable bugs; more misconfigurations)

- Backup – needs to read all users' files
- Mail – need to copy data from protected shared mail file/socket to protected per-user mail file (→ sendmail follies)

- …

QUESTIONS:
How violate principles?
Solutions?

- Fine grained access control: Mail user with access to specific files, backup user that can copy but not read or write, … Evaluate from standpoint of robustness: Which of Saltzer's principles is helped? Which hurt?
- Auditing: legal/technical combination to combat abuse of privilege by insiders
- …

6.2 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 open this attachment/download this plugin"

How relate to robustness, principles…
"Social engineering" – psychological acceptability, robustness issues
Least privilege?
…

6.3 internet worm

1990 - broke into thousands of computers over internet
2001 – code red – millions of machines compromised
…

1990 worm:
Three attacks

1. dictionary lookup
2. sendmail

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

3. fingerd – buffer overflow
   -- 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!

Got caught b/c idea was to launch attacks on other systems from whatever systems were broken into; so ended up breaking into same machine multiple times, dragging down CPU so much that people 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!

Interesting paper "How to own the internet in your spare time" – exponential growth of aggressive worm → can take over all vulnerable machines on internet in minutes!

7. Conclusions

Lots of conflicting goals – no silver bullet

- Need to understand big picture – not just technical issues but also how system will be designed, maintained, used
- Learn why systems really fail, don't just guess (design, audit, feedback)
- Consider moral hazard when designing system

- Design for robustness
- Lots of (conflicting) goals/principles

Monday – discuss some technical issues; but remember that's not the whole story!

Review

- Understand how system will be used – psychological acceptability (for designer and user) is key
- Think about (and measure) how system might/does fail
- Design for robustness

Outline: Authentications
Local – passwords and pitfalls
Remote -- encryption

8. 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

8.1 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

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…

some (partial) solutions

b) require more complex passwords – example: 6 letter w/ upper, lower case, and number, and special character:
    70^6 ~600B → 6 days

except: people still pick common patterns (e.g. 5 lower case letters + 1 number)

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

d) assign very long passwords – give everyone a calculator (or smartcard) to carry around to remember the password. Requires physical theft to steal password

This is state of the art -- if you care about security you do this

Implementation techniques to improve security

(1) Enforce password quality

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 …

(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 after k incorrect attempts

(4) 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…

8.2 Tenex – early '70s BBN

Most popular systems at universities 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!!

8.3 2-factor authentication

Passwords limited by human capabilities

Current state of art for authentication – 2 factor authentication

Identify human by
(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, …)

Example: timer-card authentication

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>

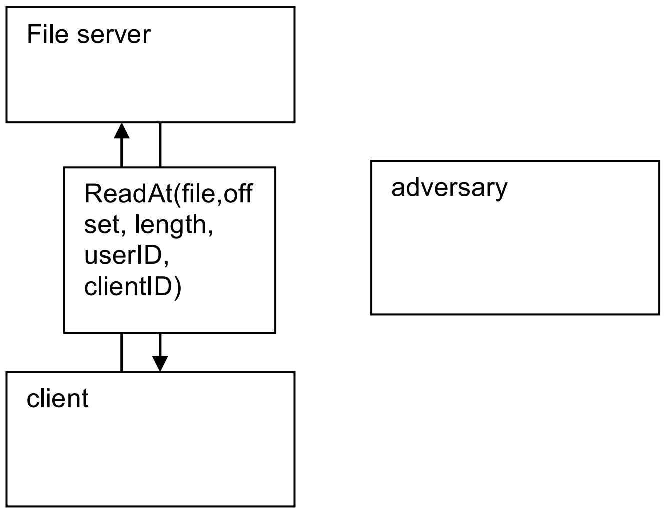
****
Lecture - 20 min
****

9. Authorization in distributed systems

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

9.1 How not to do distributed authentication I

Consider authentication in distributed file system



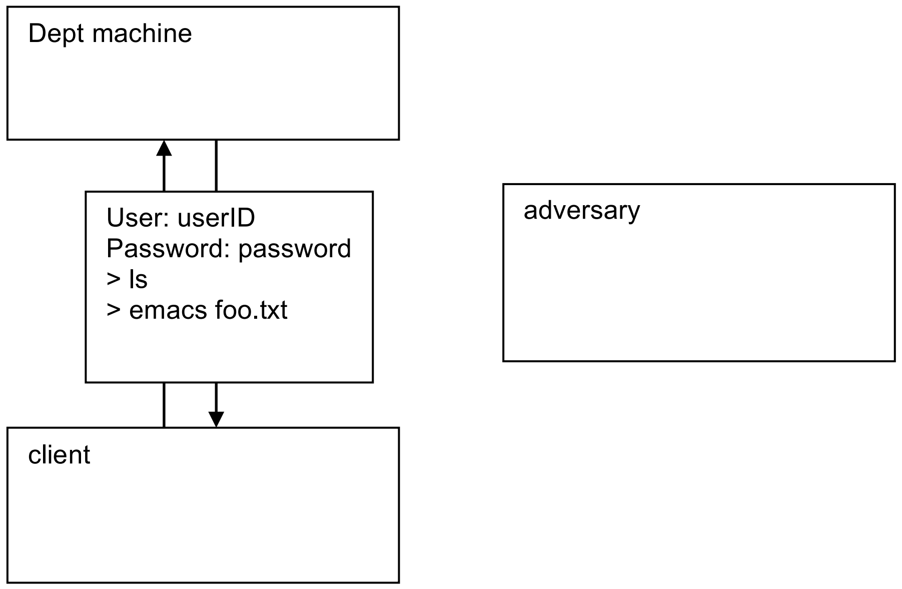
Adversary model
Typical assumption – we don't physically control the network so adversary can (a) see my packets, (b) change my packets, (c) insert new packets, (d) prevent my packets from being delivered

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?

9.2 How not to do distributed authentication II

Consider remote login



Problems? Does it look familiar?

9.3 Solution: encryption

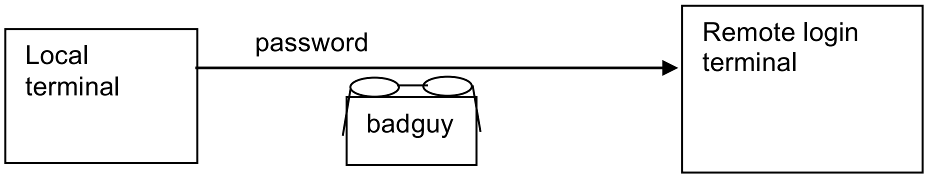
Two roles for encryption:

a) Authentication (+tamper resistance)
   Show that request was sent by someone that knows the secret w/o sending secret across the network

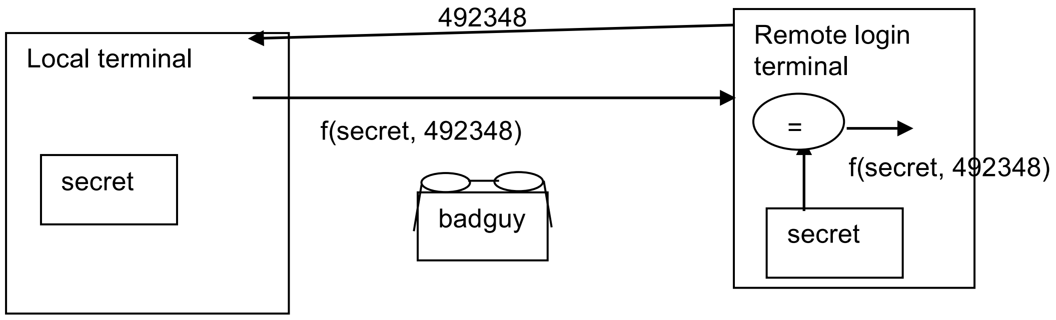
b) secrecy – I don't want anyone to know this data (e.g. medical records, etc.)

9.4 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

9.5 Encryption primitives

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

9.5.1 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?)
-- 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
Don't throw the machine away after cracking one key!
Cost per key (assuming 10 year operational life)
$5000/(8 keys/day * 365 days/year * 10 years/machine) → $0.17 per key

- adding 1 bit doubles search space. 2^128 is a big search space
- Brute force not feasible
- Look for flaws in algorithm to restrict search space ("differential cryptography" "integral cryptography", …
- AES-128 and AES-256 are current "best practice" and believed to be quite secure

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

9.5.2 Public key encryption

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

9.5.2.1 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~~

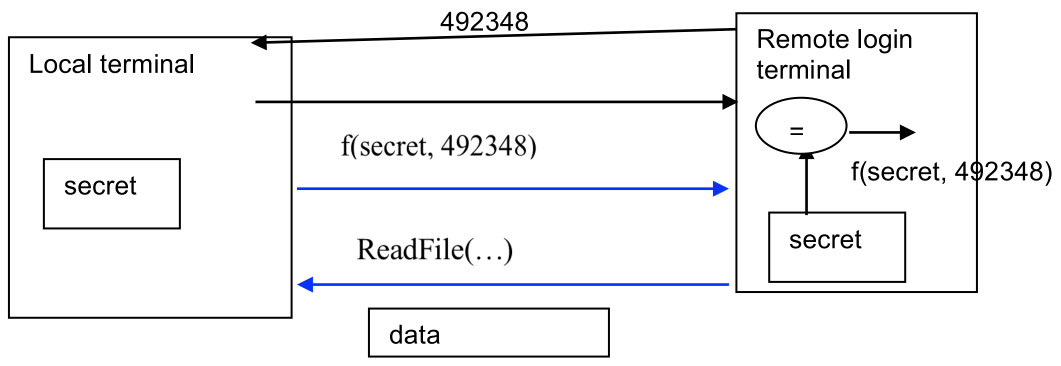
~~Kpublic-S is distributed by hand (or pre-installed on your computer – internet explorer, netscape)~~

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

9.6 Encrypted session

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

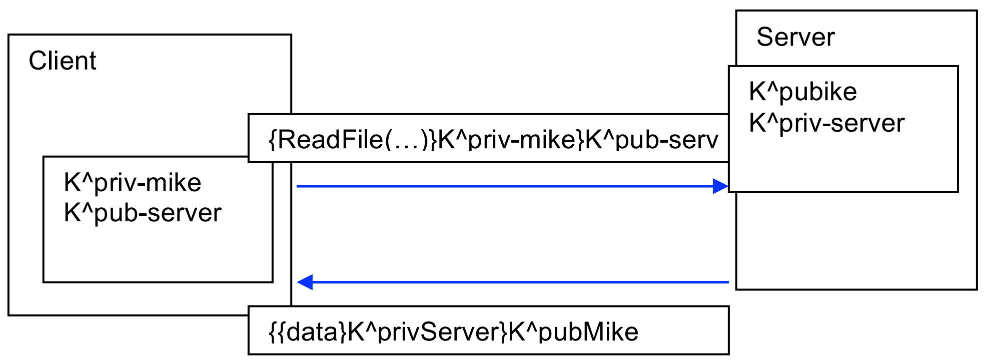
The above protocol can prove it is me. But then what?



What's wrong?

9.6.1 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
Server
K^pubike
K^priv-server
{{data}K^privServer}K^pubMike

9.6.2 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 **very 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

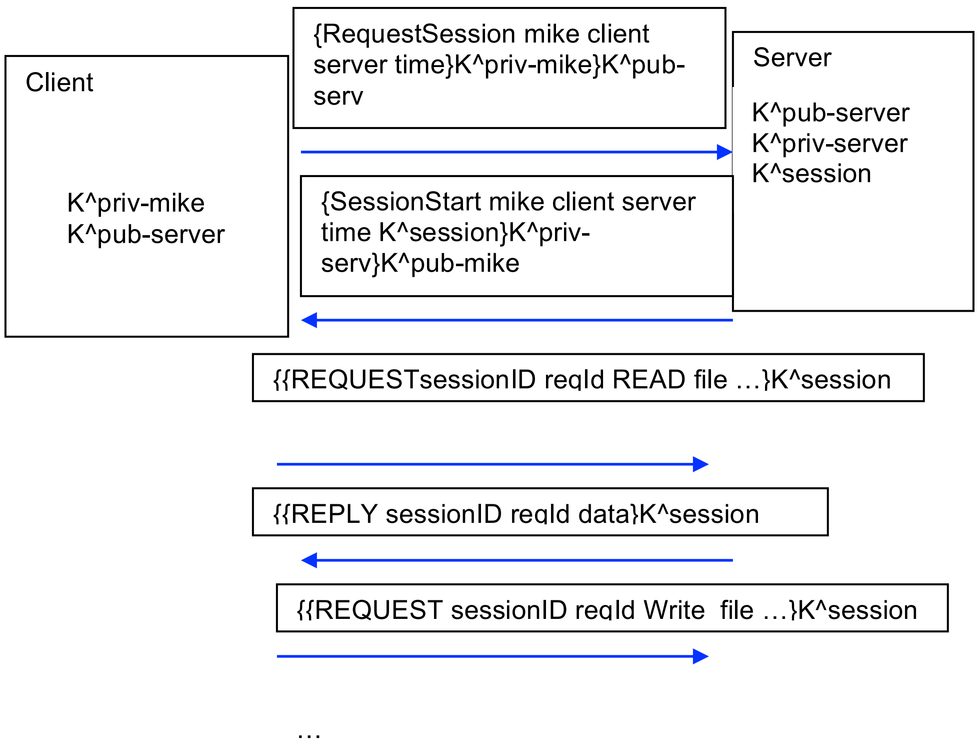
- attacker can resend old read request (for read, limited effect. What about command "buy 100 shares of IBM"?)
- attacker can send old read reply (how does client match requests to replies?}
- → Include timestamps or nonces in messages, expiration times in certificates

9.6.3 Example protocol (realistic)

(1) Exchange certificates

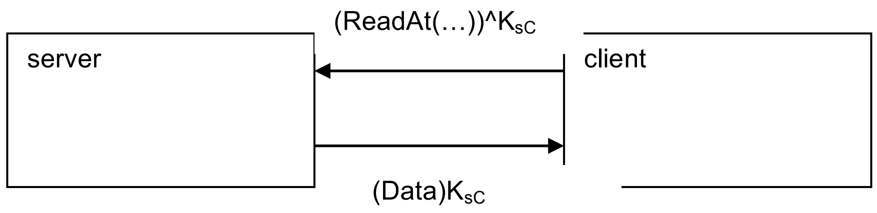
Client->server: {CA, K_pub-mike, mike, expires}K_priv-CA
Server->client: {CA, K_pub-server, server, expires} K_priv-CA

(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
…

9.7 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?

9.7.1 Authentication server (example: kerberos)

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)

Notation:

Kxy is key for talking between x and y

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

Example (simplified[1] kerberos)

A asks server for key

A → S: A B // 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"
S → A: {A B Kab {A B Kab}^Ksb}^Ksa

A gives B the ticket

// S says to B "use Kab for communication between
// A and B"
A → B: {A B Kab}^Ksb

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

[1] For simplicity, I have omitted timestamps from this description, so the protocol sketched above may be vulnerablele to "replay attacks" where an adversary stores messages from a previous execution of the protocol and uses them to fool you during a new execution of the protocol. Technically, the above protocol is sufficient to convince A and B that S *once said* that Kab is a good key for communication between A and B, but not to convince them that S *believes* Kab is a good key and therefore not enough to convince A or B to believe it either.

(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)

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)

Store master key Ksa (or Ksb) locally at A (or B) encrypted with A's (or B's) password
→ only A (or B) can get Kab (Ksb) from S

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

10. 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)

11. 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

****

12. 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

13. Classes of security problems

13.1 abuse of privilege

if superuser is evil, we're all in trouble

no hope

13.2 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

13.3 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"

13.4 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

13.5 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!

…

14. Examples

14.1 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|aaaaaaa

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!!

14.2 internet worm

1990 - broke into thousands of computers over internet

Three attacks

4. dictionary lookup
5. sendmail

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

6. 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!

14.3 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

14.4 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)

14.5 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) …

14.6 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
3) make it possible (easy)
4) 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.

15. 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

****

16. 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

17. 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

18. 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