

Admin:

Presentations - aim for 8-8.5 mins
+ Q&A etc
10 min total

Today:

- Quines
- Decidability & A/V detection
- Trusting Trust
- Certificates
- SPKI/SDSI

"Malware Theory"

C.857 Rivest

L14.3 3/31/08

- We are used to programs that work on other programs:
E.g. interpreters, optimizers, compilers, byte-code verifier, virus detector...
- We are now interested in programs that work on themselves ("self-referential") - they have access to their own source code.

- Can you write a program that prints itself? ("Quine")
in C:

```
char #s = "char #s = %c %s %c; main() { printf(s, 34, s, 34); }";  
main() { printf(s, 34, s, 34); }
```

Note: 34 is decimal code for double-quote char.

- We can modify above pgm to save text in a variable, rather than printing it.
(Use "sprintf" in C; or modify s by substitution...)

Thus, we can have programs P of form:

P ≡ { s = <text for P>;
~~~~~  
} ← modification of above  
← other code, can operate on s, as if it were input.

- For example, we can write a program  $P$  that applies a routine  $A$  to text of  $P$ :

6.857 Rivest  
L14.4 3/31/08

$$P \equiv \begin{cases} s = \langle \text{text for } P \rangle \\ \text{define } A(x) ::= \\ A(s) \end{cases}$$

or (in high-level notation):

$$P \equiv A(P)$$

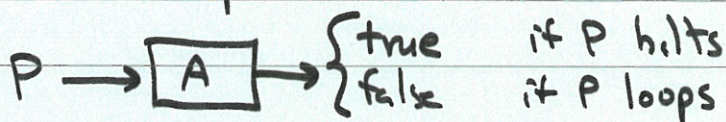
[ running  $P$  is same as applying  $A$  to source code for  $P$ . ]

(All this is called "Recursion Theorem" in theory of computation...)

- Def: The Halting Problem is: Given a program  $P$  that takes no inputs, decide if  $P$  halts, or loops forever when run.

- Thm: The Halting Problem is undecidable.

(I.e. there is no program  $A$  that takes as input a description of a program  $P$ , and always halts & outputs correctly whether  $P$  halts or loops.)




Proof: Assume such an  $A$  exists (& we have its code):

Let  $P \equiv$  [ if  $A(P)$  then loop else halt ]

What does  $A$  do on  $P$ ?

if  $A(P)$  then  $A$  says  $P$  halts  $\Rightarrow$   $A$  is wrong

if  $A(P)$  <sup>true</sup> false then  $A$  says  $P$  loops  $\Rightarrow$   $A$  is wrong

$\therefore A$  doesn't exist. 

6.857 Rivest  
L 18.5 4/11/11

- More generally, determining any nontrivial property of (output) behavior of a program is undecidable.

(Known as "Rice's Theorem")

Nontrivial means:  $\exists$  program  $X$  that exhibits behavior  
&  $\exists$  program  $X'$  that doesn't exhibit behavior.

Behavior = output behavior; not things that depend on source code  
or number of steps taken

e.g. "uses printer" or "halts" or "prints two zeros in a row"

Pf: Same as before.

Assume  $A$  can decide if program has property.

Consider  $P \equiv [ \text{if } A(P) \text{ then } X'() \text{ else } X() ]$

$A$  is wrong about  $P$ .  $\square$

- Thm: Virus detection is undecidable. (Cohen '87)

(Define virus to be a program that "spreads" (infects other programs).)

Pf: Same as for Rice's Theorem, etc.

Assume  $A$  can decide if input program is a virus.

Consider  $P \equiv [ \text{if } A(P) \text{ then halt else spread } () ]$

$A$  is wrong on  $P$ .

(Contradiction, so  $A$  doesn't exist.)

6.857 Rivest  
 L.B.G 4/11/11

- Every virus scanner must make mistakes on some inputs, and/or fail to halt on some inputs.
- Even worse: (Chess & White)

What we have shown is

$(\forall A)(\exists P)$  A is wrong about whether P is a virus.

But it is true that

$(\exists P)(\forall A)$  A is wrong about whether P is a virus

P is polymorphic

P uses randomization

P spreads as a different form P'

$P =$  if  $A(P)$  then halt else spread ( )  
 ↑ randomized

$P' =$  if  $A'(P)$  then halt else spread.  
 ↑ random  
 A/v  
 scanner

$\exists$  one member of family that defeats each  $A_{i.e.}$

6.857 Rivest

L14.5 3/31/08

- More generally, determining any property of behavior of a program is undecidable (as long as it is nontrivial.) ("Rice's theorem")  
halt  $\Rightarrow$  exhibit behavior  
false  $\Rightarrow$  don't exhibit behavior

- Ken Thompson's "Reflections on Trusting Trust" (1984)  
example of nasty malware: can't even find it by looking at source code (!)

let  $L =$  login program  $L(pw):$   $\begin{cases} \text{if check}(pw) \\ \text{then allow\_login}() \\ \text{else reject}() \end{cases}$

evil login program  $L'(pw):$   $\begin{cases} \text{if } pw = "3YNQ74B" \\ \text{or check}(pw) \\ \text{then allow\_login}() \text{ else reject}() \end{cases}$

but: someone may notice source has been modified

so: attacker can also modify compiler (!)

Let  $C =$  standard compiler

evil compiler  $C'(x) = \begin{cases} \text{if } x = L \text{ then output } C(L') \\ \text{else output } C(x) \end{cases}$

now source for  $L$  is left alone, but source for compiler changed; it may be noticed.

• so doubly-evil compiler:

6.857 Rivest  
L14.6 3/31/08

$C''(x) =$  if  $x = L$  then output  $C(L')$   
else if  $x = C'$  then output  $C(C'')$   
else output  $C(x)$

← note self-reference!

Attacks leaves sources as  $L, C, \dots, x$   
but binaries as  $C(L'), C(C''), \dots, C(x)$

all sources look clean!

situation is stable: recompiling any program yields same binary!

Ouch!!!

Moral (Thompson): "You can't trust any code you did not totally create yourself!"

## Key management / Key distribution

- themes: crypto, keys, names, individuals, trust, identity, scaling, usability, certificates, PKI, TTP's
- Crypto keys need to be shared/distributed to be useful.
- How is such sharing/distribution to be arranged?

### ① Directly - by physical meeting



Meet in private ⇒ no eavesdroppers (privacy)

Recognize each other ⇒ authentication

Share/exchange PK's or symmetric keys

Save in DB:

| Alice | name | key             | Bob | name  | key             |
|-------|------|-----------------|-----|-------|-----------------|
|       | -    | -               |     | -     | -               |
|       | -    | -               |     | Alice | PK <sub>A</sub> |
|       | Bob  | PK <sub>B</sub> |     | -     | -               |
|       | -    | -               |     | -     | -               |
|       | ...  |                 |     | ...   |                 |

Note use of names to identify entries

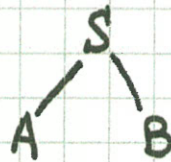
Privacy not needed if keys are PK's.



Alice or Bob could be a computer (e.g. Alice installs key in computer Bob, or computer Alice gives user Bob her public key...)

Such direct meetings are necessary foundation of any approach to key mgt, as remote meetings don't provide either authentication or privacy naturally...

## ② Indirect / Two-link using TTP/server



Alice & Bob can't meet directly, but they have both previously met with  $S$  & exchanged keys.

They can use  $S$  to broker a "key setup" operation so that  $A$  &  $B$  end up sharing a key, more-or-less as if they had met directly.

But: they need to trust  $S$  to behave properly and setup protocol ("key exchange" protocol) needs to be well designed (tricky!)

Needham & Schroeder proposed two protocols:

one for symmetric keys & one for public keys.

## Needham-Schroeder Symmetric Key Exchange Protocol

### Already shared:

$K_{AS}$  - symmetric key shared between A & S

$K_{BS}$  - " " " " B & S

### Goal:

$K_{AB}$  - " " " " A & B (and S)

### Notations: (standard)

•  $N_A$  nonce generated by A

$N_B$  nonce generated by B

"nonce" = a "use once" value (never repeats)

could be counter, or long random value

Nonces can protect against "replay attacks"

- $\{M\}_K$  = message M encrypted & authenticated with key K (e.g. derive  $K_1$  &  $K_2$  from K, encrypt M using AES in suitable mode, then append  $MAC_{K_2}$  of ciphertext).

≡ "authenticated encryption"

Literature often vague about properties of  $\{ \}_K$

Symmetric Protocol:

①  $A \rightarrow S: A, B, N_A$   $\equiv$  "Hi, I'm Alice & I want to talk with Bob.  $N_A$  is my "request nonce" "

②  $S \rightarrow A: \{N_A, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$   
"blob"  $\downarrow$   $\equiv$  "OK. Here's  $K_{AB}$  for you & a blob to give to B."

③  $A \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$   $\equiv$  "Hi B. I'm A. Here's a blob" (Bob checks & decrypts blob)

④  $B \rightarrow A: \{N_B\}_{K_{AB}}$   $\equiv$  "Hi A. Here's a challenge to prove you know  $K_{AB}$ ."

⑤  $A \rightarrow B: \{N_B - 1\}_{K_{AB}}$   $\equiv$  "That's easy. Here's  $N_B - 1$  encrypted with  $K_{AB}$ ."

Notes: A, B, and S know  $K_{AB}$

S must be trusted! (S can pretend to be A to B or vice versa...)

Note roles of names: handles to identify parties, addresses to send messages to, text strings to put in messages

If no nonces (or perhaps timestamps), Adversary could replay earlier protocol to Alice or Bob...

Kerberos

Public-Key Protocol:

$K_{Px}, K_{Sx} \equiv$  public, secret key of party X

Assume: S knows  $K_{PA}, K_{PB}$

A & B know  $K_{PS}$

①  $A \rightarrow S: A, B \equiv$  "request from A for  $K_{PB}$ "

②  $S \rightarrow A: \{K_{PB}, B\}_{K_{SS}} \equiv$  signed "cert" for B's PK

③  $A \rightarrow B: \{N_A, A\}_{K_{PB}} \equiv$  "Hi! I'm A, with nonce  $N_A$ "

new { ③'  $B \rightarrow S: B, A \equiv$  "request from B for  $K_{PA}$ "

③''  $S \rightarrow B: \{K_{PA}, A\}_{K_{SS}} \equiv$  signed "cert" for A's PK

④  $B \rightarrow A: \{N_A, N_B\}_{K_{PA}} \equiv$  "hi" (note these are bound together, non malleable.)

⑤  $A \rightarrow B: \{N_B\}_{K_{PB}} \equiv$  "yep, I'm here!"

Note: ③' & ③'' could be removed by having cert ③ inside ②

At end: only A & B know  $N_A$  &  $N_B$ ; eavesdropper doesn't

Is this secure! How do you tell?

## Attack found by automated analysis!

(Gavin Lowe found it in 1995 - 17 years later!)

- ① Intruder  $I$  gets  $A$  to initiate communication with  $B$
- ②  $I$  passes knock  $\{N_A, A\}$  on to  $B$  (after reencrypting with  $K_{PB}$ )
- ③  $B$  responds with  $\{N_A, N_B\}_{K_{PA}}$ , which  $I$  sends to  $A$
- ④  $A$  sends  $\{N_B\}_{K_{PI}}$  to  $I$ .  $I$  decrypts & gets  $N_B$
- ⑤  $I$  sends  $\{N_B\}_{K_{PB}}$  to  $B$

Now  $B$  thinks he is sharing  $N_A$  &  $N_B$  only with  $A$ ,  
but  $I$  knows  $N_B$ . This is wrong.

Fix: ④  $B \rightarrow A: \{N_A, N_B, \underline{B}\}_{K_{PA}}$

Morals: Be explicit in protocols!

(e.g. give session id both ways, & identities...)

Give hash of shared transcript in each message  
(i.e. of all previous messages)...

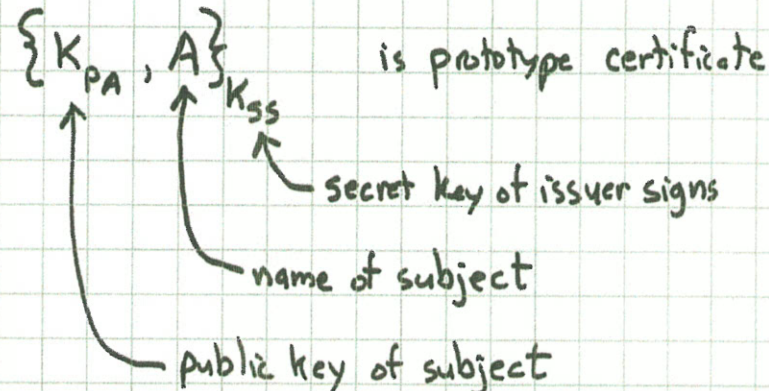
Automated analysis works!

Huge literature on key establishment protocols.

Certificates:

• Typically: a signed statement from an "issuer" about a "subject" & his public key

• In Needham-Schroeder:



S certifies that Alize's PK is  $K_{PA}$

• Others can get this cert from S, from A, or from some online database.

(Kohnfelder's MIT bachelor's thesis, 1978)

• Scaling: how to scale from 100 users to  $10^{10}$ ?

Everything starts breaking:

- there is no one server everyone trusts
- one server can't handle load
- what are names?

Names:

- How does Alice know Bob's name?
- Can Alice trust Bob when he says his name is ....
- Who ensures that names are unique?

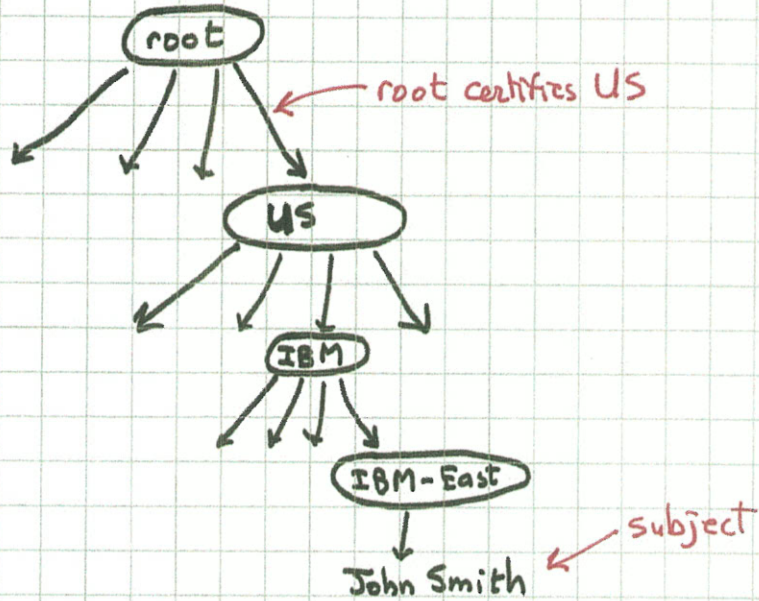
How is this done? Can it be manipulated/compromised?

[Compare: email addresses...]

- If Alice can get Bob's (name or email address) correctly already, why can't she get his PK the same way?

One approach: Hierarchical naming system

X.509 hierarchy



DN = "distinguished name"

= "CO=US/ORG=IBM/DIV=IBM-EAST/CN=John-Smith"

subject's DN

These DN's become unwieldy for people to use.



## Certificate fields:

Version #

Cert serial #

Signature algorithm

Issuer DN

Subject DN

Validity period (not before, not after)

Subject PK (alg & key)

Issuer unique #

Extensions: type & critical / non-critical flag

↳ key usage (encs, sigs, certs)

cert policies

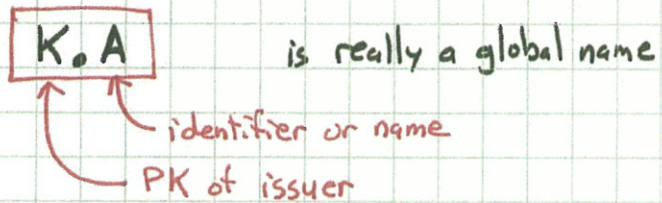
subject alternate name

path constraints

Used by TLS (SSL)

## SPKI/SDSI

- alternate framework by Ellison, Lampson, Rivest & others
- No global names (sort of...)
- Each PK is a CA & has its own name space



- Two types of certificates:

name cert = associates name with key  
or another name (!)

auth cert = gives permission (authorization)  
to subject(s)

- Name certs:

Provide elegant & flexible "algebra" of names

Allows one to describe groups (multivalued names)

$$\text{cert} = (K, A, X)$$

$$K.A \Rightarrow X$$

give as "rewrite rule"

K = issuer PK

A = identifier (name)

X = PK

or PK.id

or PK.id<sub>1</sub>.id<sub>2</sub>...id<sub>k</sub>

(key)  
(name)

extended name

Examples: (let K be my PK)

$K_0 \text{ Bob} \Rightarrow K_0$  "I say Bob's PK is  $K_0$ "

(Note: the name "Bob" is my choice & arbitrary. It doesn't need to correlate with anything else...)

$K_0 \text{ Bob} \Rightarrow K_1 \text{ Bob-Smith}$  "My Bob is the same as  $K_1$ 's Bob Smith"

$K_0 \text{ Bob} \Rightarrow K_{\text{mit}} \text{ Bob-Smith}$  "My Bob is same as mit's Bob Smith"

$K_{\text{mit}} \Rightarrow K_2$

$K_0 \text{ Bob} \Rightarrow K_2 \text{ Bob-Smith}$  inferred rule

Inference

Groups:  $K_0 \text{ friends} \Rightarrow K_0 \text{ Alice}$   
 $K_0 \text{ friends} \Rightarrow K_0 \text{ Bob}$  } group members

...

Let  $K_2 = \text{mit PK}$

$K_2 \text{ faculty} \Rightarrow K_2 \text{ eecs faculty}$

$K_2 \text{ faculty} \Rightarrow K_2 \text{ math faculty}$

⋮

(group defined as union of other groups)

## Authorization certs & ACL's

ACL = access control list

Can put name on ACL for a resource:

"Only individuals in  $K_0$  friends may read files in this directory"

### Authorization cert:

Issuer key  $K$

Rights being granted [read directory foo]

Delegatable or not

Subject key or name

$K$  [read "/foo"; delegatable]  $\Rightarrow K_0$  friends

There are polynomial-time algorithms for determining whether a given collection of name & auth certs implies that a given key is authorized to perform a given action

Certificate revocation

- Why?
- key compromise
  - change of affiliation
  - change of authorization
  - change of name (e.g. merger)

Fairly high "churn rate"

If certificate says "good until 2020-12-01"

who decides if that is good enough?

issuer? (current practice)

relying party? ← Should be relying party!  
(they are taking the risk...)

Helpful to think about it this way:

- Issuer maintains authoritative DB
- Certificates are merely "snapshots" of items
- Note that DB may not fully reflect key compromises, etc...

Method ①: On-line check:

Relying party asks issuer if cert still good

Issuer gives signed response (new cert?)

≡ OCSP (online certificate status protocol)

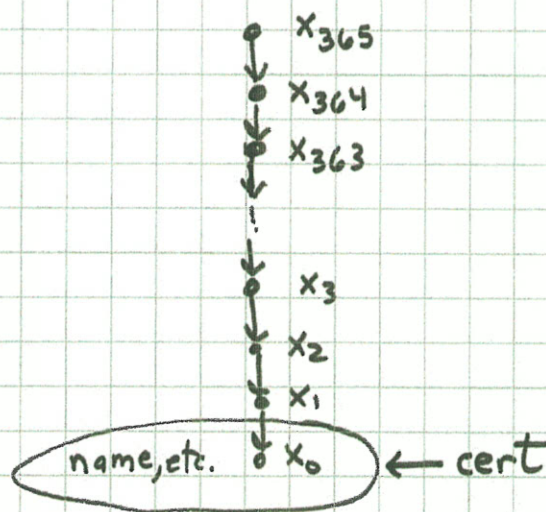
Puts heavy load on server!

Method ②: CRL's (certificate revocation list)

Server periodically issues CRL,  
giving list of revoked cert serial #'s  
(signed, of course)  
CRL can get long!

Method ③: (due to Micali)

cert contains end point of a hash chain



On day  $d+i$ , where  $d$  = cert issuing date,  
you need  $x_i$  to validate cert. Server can  
give  $x_i$  to keyholder, or to anyone else.

Relying party hashes  $i$  times & checks that result =  $x_0$ .

If no  $x_i$  given out, cert "expires".