TOPIC: 6.857 - Computer & Network Security

DATE: 5/6/13

FILE UNDER:

PAGE: L23.1

Admin:

Presentation start this Wed.

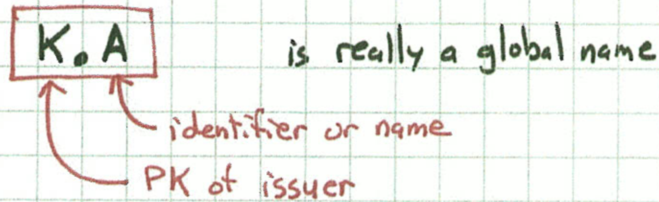
Papers due next Wed (5/15)

Today:

- Certificates: SPKI/SDSI
- Viruses (Virus detection is undecidable)
- Trusting Trust (The "evil compiler"!)

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)

cert = (K, A, X)

$K.A \Rightarrow X$

give as "rewrite rule"

K = issuer PK
A = identifier (name)
X = PK (key)
or PK.id (name)
or $PK.id_1.id_2 \cdots .id_k$ — extended name

Examples: (let K be my PK)

K . Bob ⟹ 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 . Bob ⟹ K_1 . Bob-Smith — "My Bob is the same as K_1's Bob Smith"

K . Bob ⟹ K . mit . Bob-Smith — "My Bob is same as mit's Bob Smith"

K . mit ⟹ K_2

Inference

K . Bob ⟹ K_2 . Bob-Smith — inferred rule

Groups: K . friends ⟹ K . Alice
K . friends ⟹ K . Bob
} group members

...

Let K_2 = mit PK

K_2 . faculty ⟹ K_2 . eecs . faculty

K_2 . faculty ⟹ K_2 . 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. 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] ⇒ K. 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

TOPIC:

DATE:

FILE UNDER:

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!

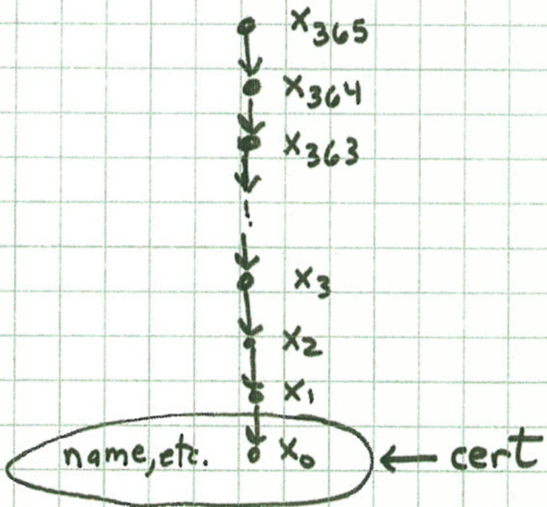
Method ②: CRL's (certificate revocation list)

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".

6.857 Rivest
L18.1 4/11/11

Admin:

Outline: Viruses

- ☐ Brief history of viruses
- ☐ Self-referential programs
- ☐ Halting Problem is undecidable
- ☐ Virus detection is undecidable
- ☐ Some practicalities of virus detection

Virus History

1981: first virus: "Elk Cloner" Apple II; Floppy disk xmit.

1983: Term "virus" coined by Adleman & Cohen (1986 PhD thesis)

1988: Internet worm (≈60,000 hosts on Internet; 10% infected.)
- fingerd buffer overflow
- sendmail debug mode left on – left access to shell
- dictionary attacks on passwords
- "trusted hosts" vulnerability

1990's: 1M hosts on web in '92
First "polymorphic" viruses (Tequila & Amoeba)
Start of anti-virus industry (Symantec)
late 90's: 30M hosts on web ('98)

2001: Code Red infects 360,000 hosts in 14 hours
(out of 125M hosts on web in '01)
buffer overflow in M.S. indexing service

2003: Slammer: infected M.S. SQL servers
doubled # infected every 8.5 seconds!
infected ≥ 75,000 hosts
caused network congestion (severe)
376 byte UDP packet
random scanning of 32-bit IP addresses

2006: 440M hosts on Internet (www.isc.org)

Jan '09
625M

Stuxnet

"Malware Theory"

6.857 Rivest
L18.3 4/11/11

- 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:

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

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.)

  P → [A] → { true if P halts; false if P loops }

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

Let P ≡ [if A(P) then loop else halt

What does A do on P?

if A(P) true then A says P halts ⇒ A is wrong

if A(P) false then A says P loops ⇒ A is wrong

∴ A doesn't exist. ▨

- More generally, determining any nontrivial property of (output) behavior of a program is undecidable.
(known as "Rice's Theorem")
Nontrivial means: ∃ program X that exhibits behavior
& ∃ 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 ≡ [if A(P) then X'() else X()]
A is wrong about P. ∎

- 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 ≡ [if A(P) then halt else spread()]
A is wrong on P.
(Contradiction, so A doesn't exist.)

6.857 Rivest
L 18.6 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

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

But it is true that

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

P is polymorphic
P uses randomization
P spreads as a different form P'

P = if A(P) then halt else spread( )
↖ randomized

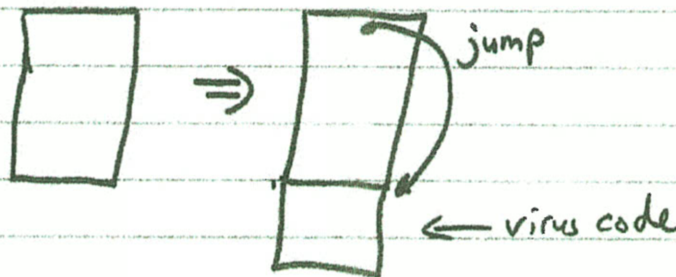
P' = if A'(P) then halt else spread.
↑ random A/V scanner

∃ one member of family that defeats each A...

6.857 Rivest
L 18.7 4/11/11

(Material from Carey Nachenberg slides "Virus/Antivirus Co-evolution")

① simple virus: insert code at end for virus
insert jump to beginning of virus code

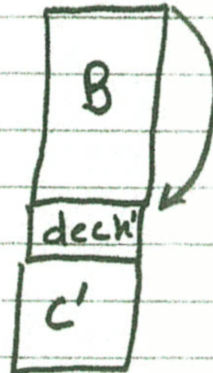


② A/V responses:
"signatures": for each virus, find static pattern
that occurs in virus code, but not in
any common good code (e.g. μsoft word)
build table of signatures
scan files for signatures

③ virus writer response: polymorphic code
"poly" = many
"morphic" = shape
virus code never looks the same

6.857 Rivest
L 18.8 4/11/11

infected pgm



dec = decryption routine

Use encryption: when infected pgm A runs, it generates a new key K', so that ciphertext C' = E(K', virus) looks random. Only common part is small decryption routine.

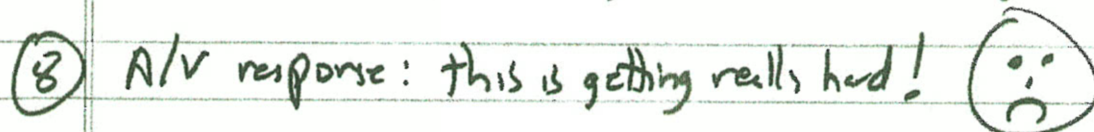
(4) A/V response:
scan for small decryption routine

(5) Virus writer response:
change enc/dec algorithm when spreading;
generate at random a new enc/dec pair.
∴ no static decryption routine!

⑥ A/V response:
emulate code in "sandbox" until it decrypts itself, <u>then</u> run static signature check.
(Hard to know exactly when to stop, though...)

⑦ Virus writer's response:

- randomly decrypt, or not
- don't decrypt at beginning of execution, but later
- "recompile" virus code in a way that preserves semantic equivalence, but destroys all "signatures"



6.857 Rivest
L19.1 4/13/11

Admin:

Outline: Malware

- ☐ Thompson's "Reflections on Trusting Trust" (1984 Turing Award)
- ☐ Software bugs
- ☐ hardware bugs
  - Shamir's μproc mpy bug (& related smartcard attack) ↓
- ☐ keyloggers

- 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): [if check(pw)
then allow_login()
else reject()]

evil login program L'(pw): [if pw = "34NQ74B"
or check(pw)
then allow_login() else reject()]

but: someone may notice source has been modified

so: attacker can also modify compiler (!)
Let C = standard compiler

evil compiler C'(x) = if x = L then output C(L')
else output C(x)

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

- so doubly-evil compiler:

$C''(x)$ = if $x = L$ then output $C(L')$
else if $x = C$ then output $\underline{C(C'')}$ ← note self-reference!
else output $C(x)$

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

all sources look clean!
situation is <u>stable</u>: recompiling any program yields same binary!

<u>Ouch!!!</u>

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