Outline:

- Malware

- Thompson's "Reflections on Trusting Trust" (1984 Turing Award)
- Software bugs
- Hardware bugs
  - Shamir's mpqc mpq bug (related smart-card 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 \( L(pw) : \)
\[
\begin{align*}
&\text{if } \text{check}(pw) \\
&\quad \text{then allow-login()} \\
&\quad \text{else reject()}
\end{align*}
\]

evil login program \( L'(pw) : \)
\[
\begin{align*}
&\text{if } \text{pw} = "3YNQ748" \\
&\quad \text{or check}(pw) \\
&\quad \text{then allow-login()} \\
&\quad \text{else reject()}
\end{align*}
\]

but: someone may notice source has been modified

so: attacker can also modify compiler (!)

Let \( C = \) standard compiler

evil compiler \( C'(x) = \)
\[
\begin{align*}
&\text{if } x = L \text{ then output } C(L') \\
&\quad \text{else output } C(x)
\end{align*}
\]

now source for \( L \) is left alone, but source for compiler changed; it may be noticed.
so doubly-evil compiler:

\[
C''(x) = \begin{cases} 
  C(L') & \text{if } x = L \\
  C(C''') & \text{else if } x = C \\
  C(x) & \text{else}
\end{cases}
\]

note self-reference!

attacks treats sources as \( L, C, \ldots, x \)
but binaries as \( C(L'), C(C'''), \ldots, C(x) \)

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!"
Thompson's paper a good example of how you can "bug" the software to an adversary's advantage.

What else could be done by adversary, in this vein?

- kleptography
- random # generator with only 35 bits of entropy
- file system (send files away)
- configuration errors
- network (send copy of all packets elsewhere...)

this is too easy

- keylogger (sends copy of all keystrokes to adversary)
- botnet

Adversary's objectives:

- Bug should introduce **vulnerability** that is easy for adversary to exploit.
- Bug should be **hard** to discover without detailed examination of code or hardware.
- Deniability: Bug should look like **innocent mistake** rather than malicious attack.
- Bug should be hard for others to exploit, even if they know about it.
- Bug should be hard to **fix**.

(how does Thompson measure up here?)
What about putting bugs in hardware?

- in CPU:
  - undocumented opcodes
  - behavior
  
  (e.g. if register R1 contains OxFA23CB1C
  
  then following instruction is executed with no protection)

- system maintenance mode bug
  (what happens at boot time?)

- Shamir's bug:
  - target: crypto (pk) implementations (bignum)
  - bug in 64-bit mpy: \( r \cdot s \neq r \cdot s \mod 2^{64} \)

  for two particular values \( r, s \) (random-looking)

  - hard to discover, even though in all mpy proc
    \( 2^{-128} \) chance of hitting \( r, s \) per mpy
    \( \sqrt{2^{40}} \) processors on planet
    \( 2^{30} \) seconds (\( \approx 30 \) years)

    \( 2^{30} \) mpy/sec

    \( 2^{100} \) mpy, trying; \( \leq 2^{-28} \) of hitting bad one

\[
\begin{align*}
\text{RSA: } n &= p \cdot q \\
S &= M^d \\
\text{CRT: } \begin{cases} 
S_p &= M^{dp} \mod p \\
S_q &= M^{dq} \mod q 
\end{cases} \\
&\quad d = d \mod p^{-1} \\
&\quad d = d \mod q^{-1} \\
S &= a \cdot S_p + b \cdot S_q \\
&\quad a = 1 \mod p \\
&\quad a = 0 \mod q \\
&\quad b = 0 \mod p \\
&\quad b = 1 \mod q
\end{align*}
\]
How to utilize bug, as adversary?

Suppose \( p < \sqrt{n} < q \) (\( p, q \) unknown to adversary)

Let \( M = \sqrt{n} \) with least sig 2 64-bit words replaced by 0s

Then \( M \mod p \neq M \)

but \( M \mod q = M \)

\[
M^d \mod n \text{ using CRT}
\]

\[
d = d_k d_{k-1} \cdots d_1 d_0 \quad d_0 = 1s_6
\]

\[
A \leftarrow 1
\]

\[
x \leftarrow M
\]

\[
\text{for } i = 0, 1, \ldots, k
\]

\[
\text{if } d_i = 1 \text{ then } A \leftarrow A \cdot x \mod p
\]

\[
x \leftarrow x^2 \mod p \text{ (or } \mod q \text{)
}\]

needs to compute \( r \cdot s \)

gets it wrong \( \mod q \)

\( S \) is correct answer \( \text{ (both } \mod p \text{ & } \mod q \) \)

\( S' \) is \( \equiv \mod p \) but not \( \mod q \)

\[
gcd(S-S', n) = p
\]

(Note: easy to get \( S \), using self-reducibility (blinding))

Fix? \( \text{(no CRT!)} \) \( \text{(randomize)} \) \( \text{(check answer!)} \)
Related attack: (not "bug", but related...)  
power glitch or timing glitch

Smart card

\[
\begin{array}{c}
\text{data} \\
\text{clock} \\
\text{Vcc} \\
\text{gnd}
\end{array}
\]

\[\text{S. c.}\]

Smart card does RSA comp. using CRT

short clock cycle during mod g part of CRT can have same effect as bad rpy, \(\Rightarrow\) bad result mod g

power spike
More on keyloggers

- Could be - in OS
  - hypervisor (blue"pill")
  - hardware - in line - acoustic?
    - in keyboard
    - modified keyboard software
- How do they get data out?
  - pw causes data to be regurgitated
  - wireless

- How to detect?
  - Keyboard on screen (use mouse)?
  - move focus in and out of text entry areas?
  - use one-time password (SecureID)
    - changes every 40 secs
  - PRG based on AES
  - synchronization an issue...