Malware: Viruses, Worms, & Botnets

Reflections on Trusting Trust

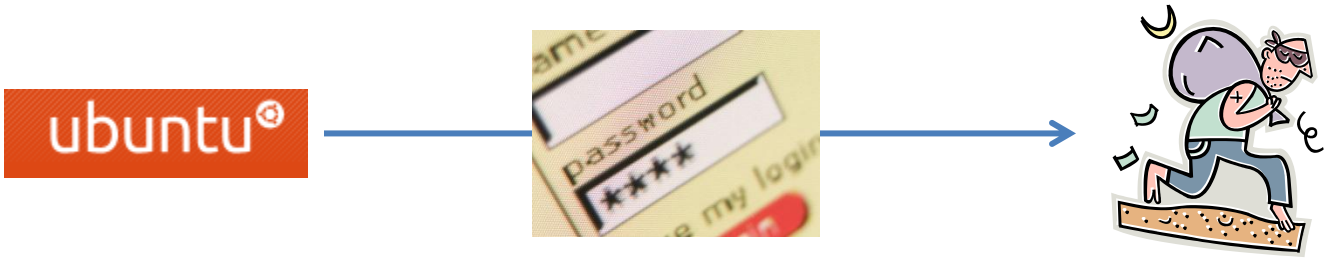
Security Can Be Really Tricky

- Check your assumptions
- Ken Thompson
  - Turing award lecture

Reflections on Trusting Trust,
http://www.acm.org/classics/sep95/

What Code Can We Trust?

- Alice downloads Ubuntu binary from web
  - Is Ubuntu binary trustworthy?
  - Does "login" have a backdoor that allows attacker to log in?



What Code Can We Trust?

- Since we cannot trust that the downloaded binary does not contain a backdoor that will allow the attacker to log-in, Alice proposed the following solution:
  - Download & read through the source code really carefully to ensure that there's no such backdoor and recompile the source code to get the binary.
- Will this solution solve the problem, i.e., ensure that the binary does not contain the backdoor?

  A. Yes. B. No.

Malicious Compiler

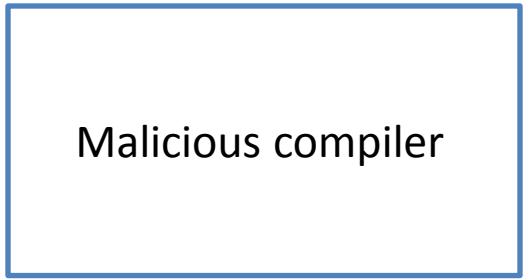
A malicious compiler can insert backdoor

- Check whether source code looks like *login* program
- If yes, insert login-backdoor which allows attacker to log in

```
compile(S) {
    if (match(S, "login-pattern")) {
        compile (login-backdoor);
        return;
    }
    .... /* compile as usual */
}
```

```
int login(char* password,
          char* user){
  ……
  authenticate(user, password);
  ……
  return 0;
}
```

Login program



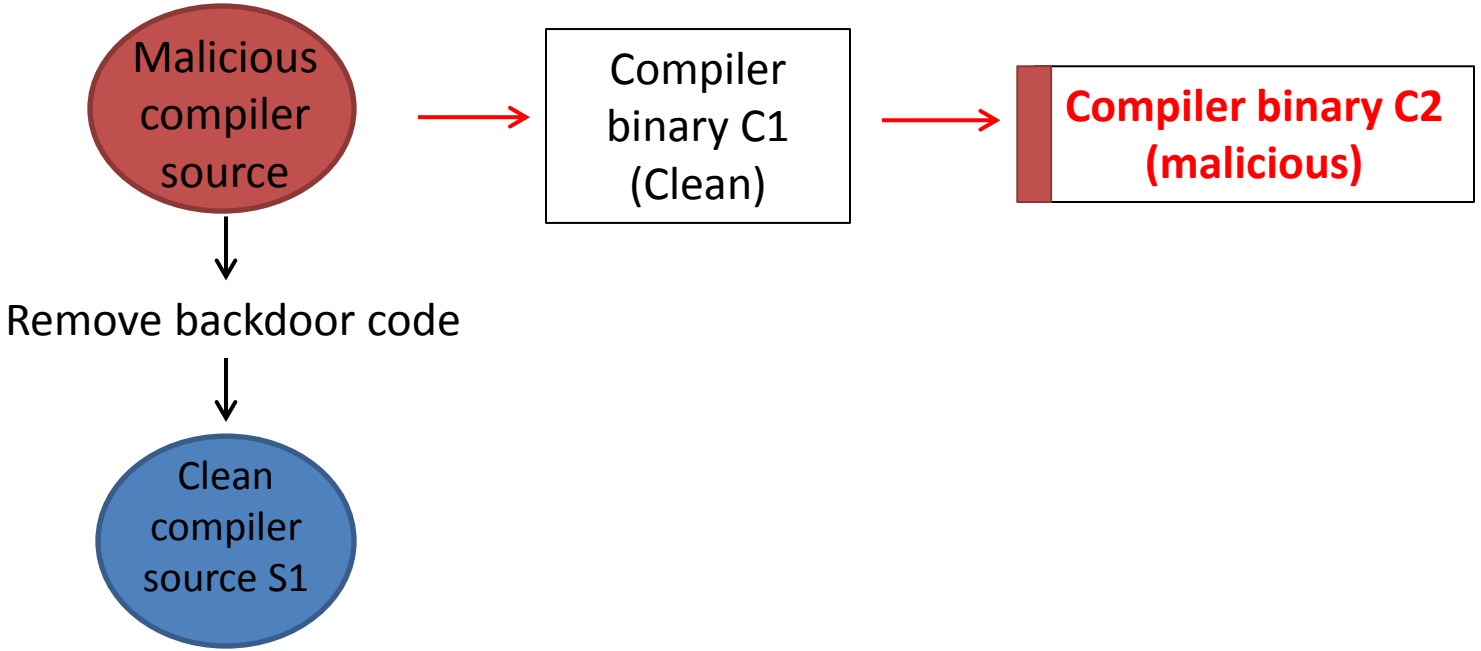
Compiler Backdoor

- How to solve this?
  - Alice: Inspect the compiler source
  - Will this work?
- C compiler is written in C & needs to be compiled
- Malicious compiler adds backdoor when compile clean compiler source S

```
compile(S) {
   if (match(S, "login-pattern")) {
        compile (login-backdoor);
        return;
   }
   if (match(S, "compiler-pattern")) {
        compile (compiler-backdoor);
        return;
   }
   .... /* compile as usual */
}
```

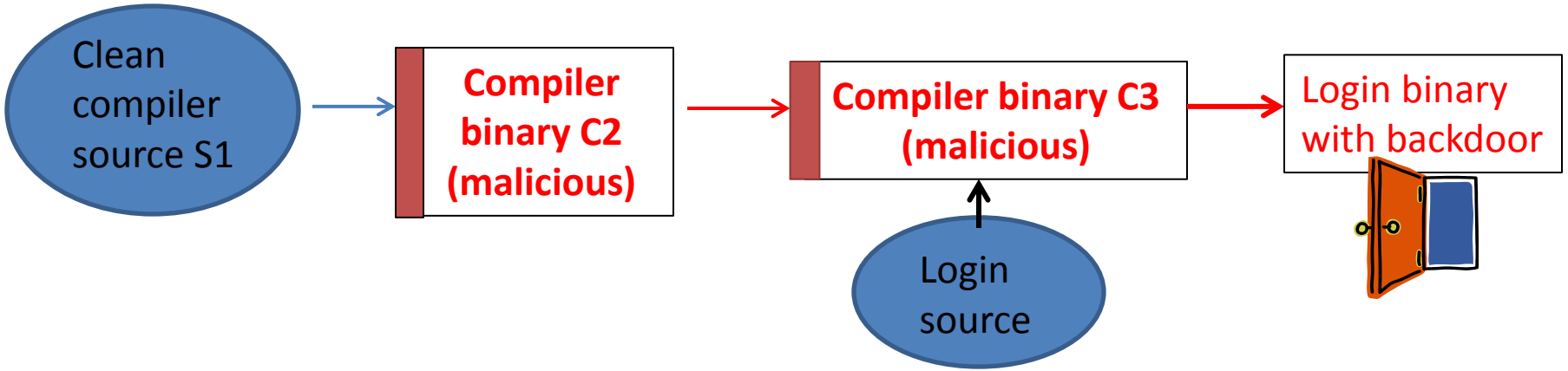
Creating Malicious Compiler Avoiding Detection

- Compile malicious compiler source with clean compiler binary
- Delete backdoor tests from source



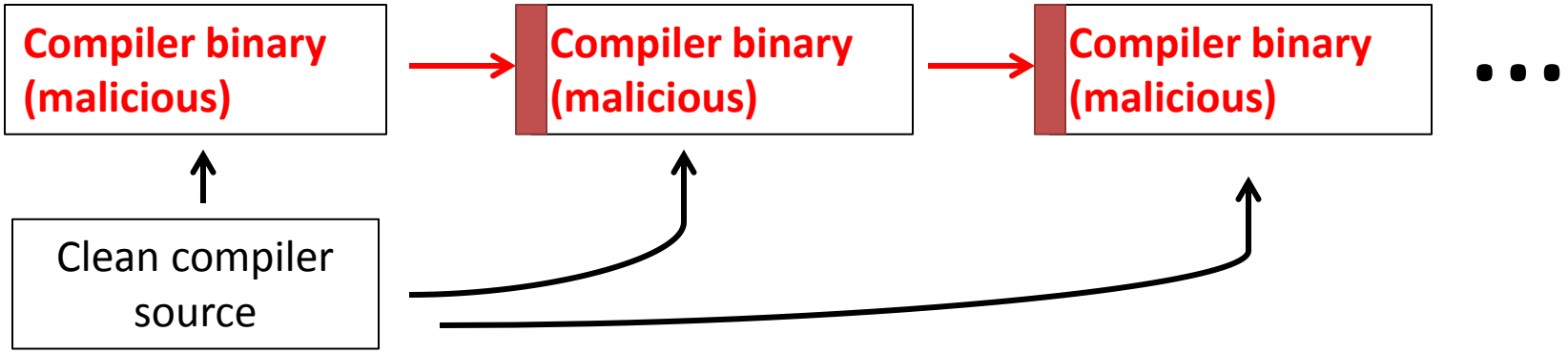
Malicious Compiler Injecting Backdoor

- Using C2 to compile S1: C3
- Using C3 to compile login source: login binary w. backdoor



Malicious Compiler Avoid Deletion

- Possible to make code for compiler backdoor output itself
  - Can you write a program that prints itself? Recursion theorem

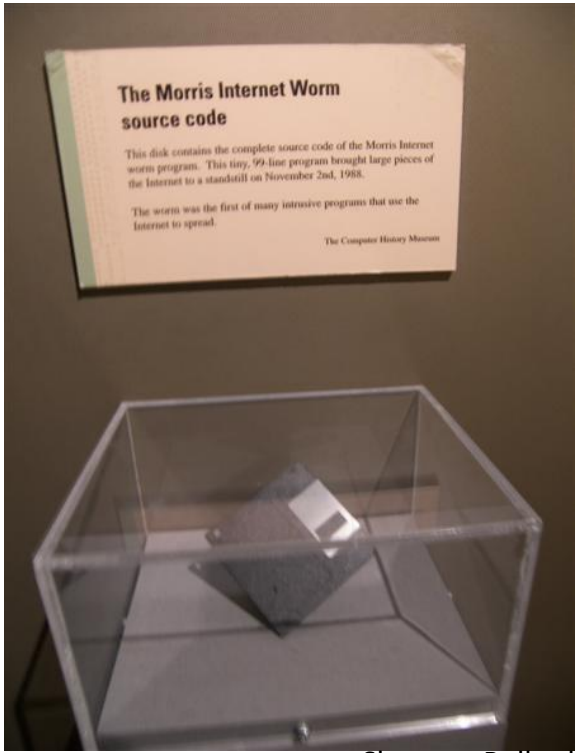


Lessons Learned

- Know what you are trusting
- Reduce trusted code base
  - No need to trust compiler if analyze binary directly

Worms

The Morris Worm: Nov. 1988



Shannon Bullard

(First large-scale worm)

Targeted VAX, Sun Unix systems

6-10% of all Internet hosts infected

Spread By:

- *Scanning* the local subnet
- Mining /etc/passwd, /etc/hosts.equiv/ .rhosts for targets
- Exploiting a *fingerd* buffer overflow
- Exploiting *sendmail*'s DEBUG mode (not a bug!)

Included code to:

- Crack passwords
- Detect co-resident worm processes
- Die off if magic global is set
- Phone home to *ernie.berkeley.edu* (buggy)

Large-Scale Malware

- Worm = code that self-propagates/replicates across systems by arranging to have itself immediately executed
  - Generally infects by altering running code
  - No user intervention required

- Botnet = set of compromised machines ("bots") under a common command-and-control (C&C)
  - Attacker might use a worm to get the bots, or other techniques; orthogonal to bot's use in botnet

Worms

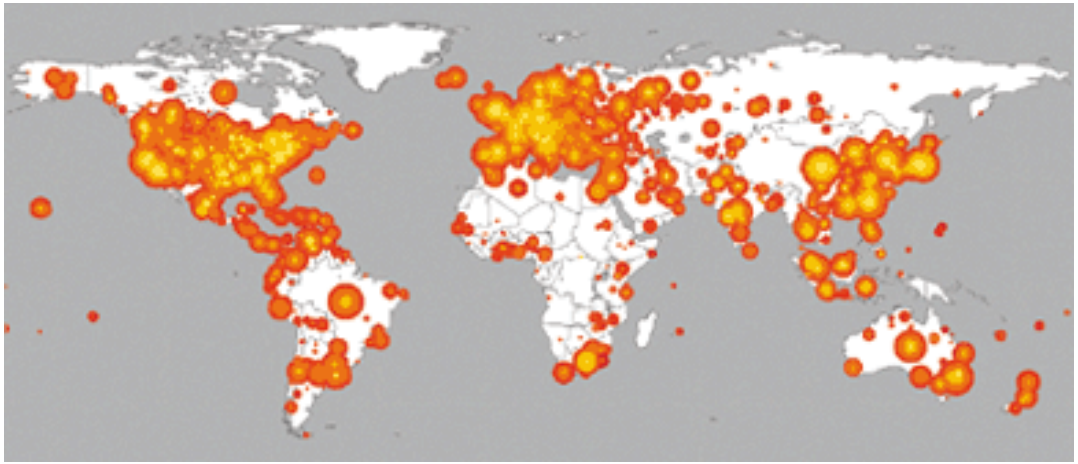
- **Propagation**
  - exponential growth, different propagation mode
- **Observation**
  - backscatter
- **Defense**
  - detection, filter, sig generation

Worm Propagation

- Worm-spread often well described as *infectious epidemic*
- Propagation is often faster than human response
- Persistence: worms stick around
  - E.g. Nimda & Slammer still seen in 2011!

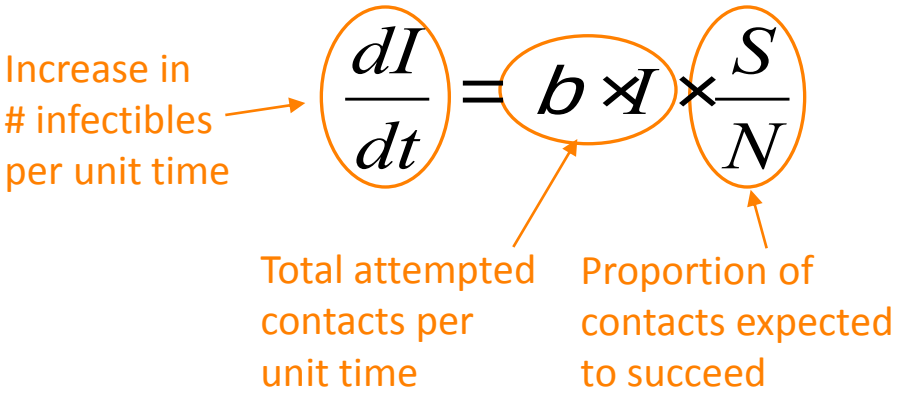
Example: Code Red

- Released July 13, 2001
- Exploits buffer overflow vulnerability inside IIS
- More than 2000 new hosts were infected each minute at peak propagation



Modeling Worm Spread

- Worm-spread often well described as *infectious epidemic*
  - Classic SI (Susceptible-Infectable) model: homogeneous random contacts



Susceptible-Infectable model

Model dynamics:

$$\frac{dI}{dt} = b \times I \times \frac{S}{N}$$

Model parameters:

$$N = S(t) + I(t)$$
$$S(0) = S_0, \quad I(0) = I_0$$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*
  - How many hosts each infected host communicates with per unit time

Modeling Worm Spread

Susceptible-Infectable model

- Rewriting by using i(t) = I(t)/N, S = N - I:

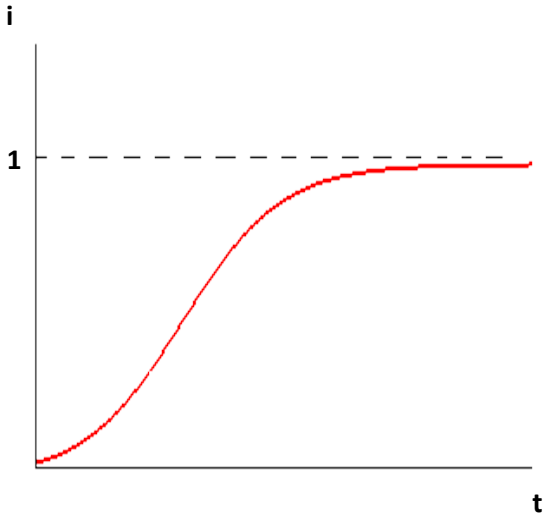
$$\frac{di}{dt} = \beta i(1 - i)$$

$$\Rightarrow$$

$$i(t) = \frac{e^{\beta t}}{\left(\frac{1}{i_0} - 1\right) + e^{\beta t}}$$

$$i_0 = \frac{I_0}{N}$$

Fraction infected grows as a *logistic*



Model dynamics:

$$\frac{dI}{dt} = \beta \times I \times \frac{S}{N}$$

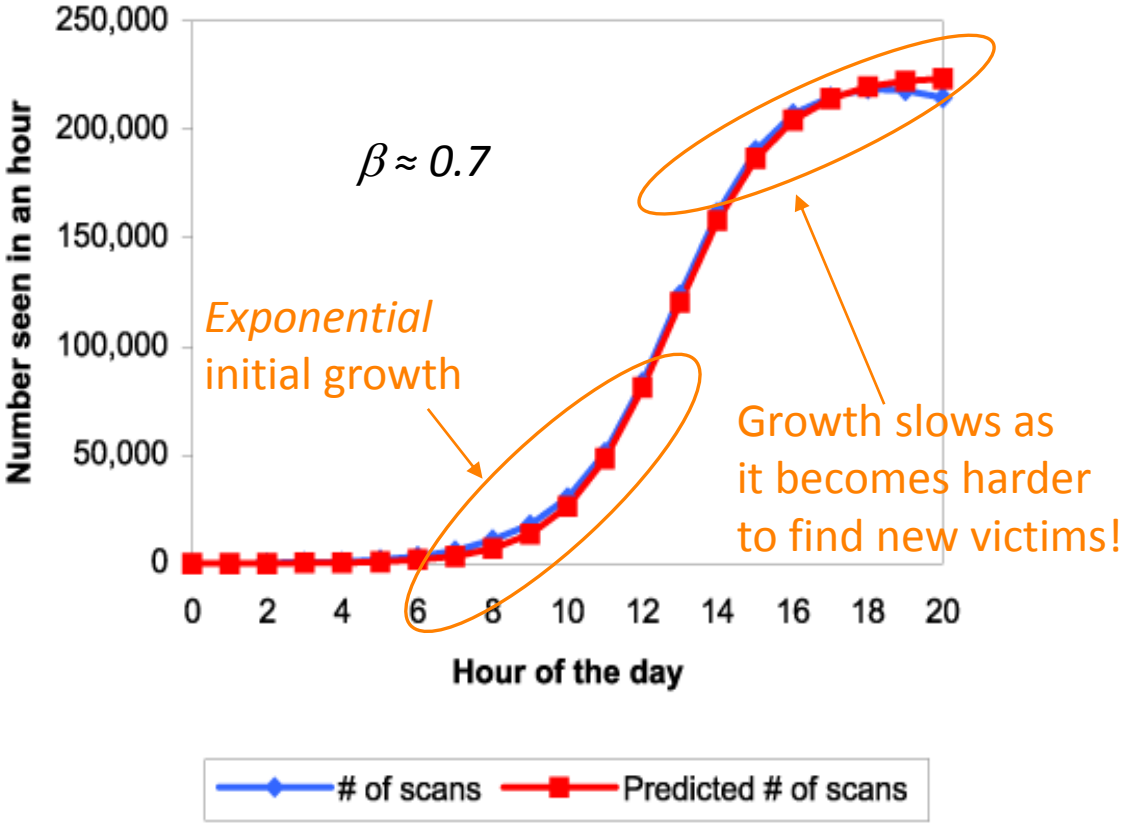
Model parameters:

N = S(t) + I(t)

$S(0) = S_0$, $I(0) = I_0$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*
  - How many hosts each infected host communicates with per unit time

Fitting the Model to Code Red



Susceptible-Infectable model

Model dynamics:

$$\frac{dI}{dt} = \beta \times I \times \frac{S}{N}$$

Model parameters:

$N = S(t) + I(t)$

$S(0) = S_0,\ I(0) = I_0$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*

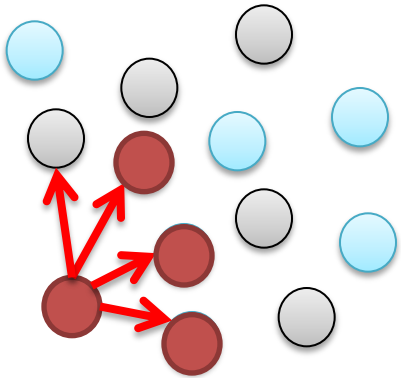
Solution:

$$i(t) = \frac{e^{\beta t}}{\left(\frac{1}{i_0} - 1\right) + e^{\beta t}}, \quad i_0 = \frac{I_0}{N}$$

Propagation Methods

Scanning Worm

- Randomly and blindly choose a host to target
- $\beta \approx N/2^{32}$



susceptible hosts
non-susceptible hosts
infected hosts

Susceptible-Infectable model

Model dynamics:

$$\frac{dI}{dt} = \beta \times I \times \frac{S}{N}$$

Model parameters:

$N = S(t) + I(t)$

$S(0) = S_0,\ I(0) = I_0$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*

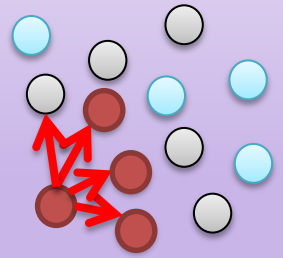
Solution:

$$i(t) = \frac{e^{\beta t}}{\left(\frac{1}{i_0} - 1\right) + e^{\beta t}}$$

Propagation Methods

Scanning Worm

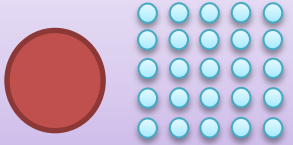
- Randomly and blindly choose a host to target
- $\beta \approx N/2^{32}$



Hitlist Worm

- I_0 is large

Hitlist:



Susceptible-Infectable model

Model dynamics:

$$\frac{dI}{dt} = \beta \times I \times \frac{S}{N}$$

Model parameters:

$$N = S(t) + I(t)$$

$$S(0) = S_0,\ I(0) = I_0$$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*

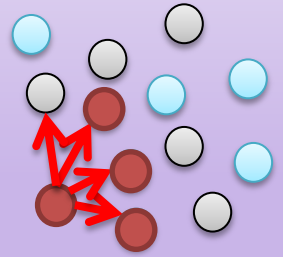
Solution:

$$i(t) = \frac{e^{\beta t}}{\left(\frac{1}{i_0} - 1\right) + e^{\beta t}}$$

Propagation Methods

Scanning Worm

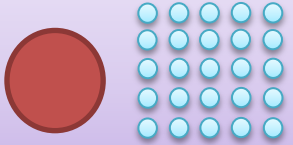
- Randomly and blindly choose a host to target
- $\beta \approx N/2^{32}$



Hitlist Worm

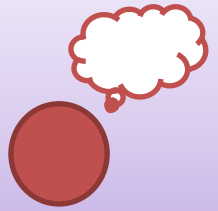
- I_0 is large

Hitlist:



Topological Worm

- Uses info on the infected host to find the next target



Susceptible-Infectable model

Model dynamics:

$$\frac{dI}{dt} = \beta \times I \times \frac{S}{N}$$

Model parameters:

$N = S(t) + I(t)$

$S(0) = S_0,\ I(0) = I_0$

- N: size of vulnerable population
- S(t): susceptible hosts at time t.
- I(t): infected hosts at time t.
- β: *contact rate*

Solution:

$$i(t) = \frac{e^{\beta t}}{\left(\frac{1}{i_0} - 1\right) + e^{\beta t}}$$

Propagation Methods

	# Infected Hosts	Start Date
Code Red	359,000	2001
Nimba	450,000	2001
Witty	12,000	2004
Sapphire/Slammer	75,000	2003

Stuxnet

- Discovered July 2010. (Released: Mar 2010?)
- Multi-mode spreading:
  - Initially spreads via USB (virus-like)
  - Once inside a network, quickly spreads internally using Windows RPC
- Kill switch: programmed to die June 24, 2012
- Targeted *SCADA* systems
  - Used for industrial control systems, like manufacturing, power plants
- Symantec: infections geographically clustered
  - Iran: 59%; Indonesia: 18%; India: 8%

Stuxnet, con't

- Used four *Zero Days*
  - Unprecedented expense on the part of the author
- "Rootkit" for hiding infection based on installing Windows drivers with valid digital signatures
  - Attacker stole private keys for certificates from two companies in Taiwan
- Payload: do nothing …
  - … unless attached to particular models of frequency converter drives operating at 807-1210Hz
  - … like those made in Iran (and Finland) …
  - … and used to operate centrifuges for producing enriched Uranium for nuclear weapons

Stuxnet, con't

- Payload: do nothing …
  - … unless attached to particular models of frequency converter drives operating at 807-1210Hz
  - … like those made in Iran (and Finland) …
  - … and used to operate centrifuges for producing enriched Uranium for nuclear weapons
- For these, worm would slowly increase drive frequency to 1410Hz …
  - … enough to cause centrifuge to **fly apart** …
  - … while sending out fake readings from control system indicating everything was okay …
- … and then drop it back to normal range

Israel Tests on Worm Called Crucial in Iran Nuclear Delay

By WILLIAM J. BROAD, JOHN MARKOFF and DAVID E. SANGER
Published: January 15, 2011

*This article is by **William J. Broad, John Markoff** and **David E. Sanger.***

Enlarge This Image

Nicholas Roberts for The New York Times

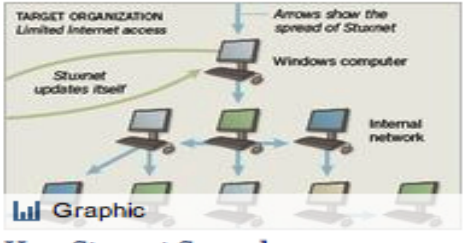
Ralph Langner, an independent computer security expert, solved Stuxnet.

The Dimona complex in the Negev desert is famous as the heavily guarded heart of Israel's never-acknowledged nuclear arms program, where neat rows of factories make atomic fuel for the arsenal.

Over the past two years, according to intelligence and military experts familiar with its operations, Dimona has taken on a new, equally secret role — as a critical testing ground in a joint American and Israeli effort to undermine Iran's efforts to make a bomb of its own.

Behind Dimona's barbed wire, the experts say, Israel has spun nuclear centrifuges virtually identical to Iran's at Natanz, where Iranian scientists are struggling to enrich uranium. They say Dimona tested the effectiveness of the Stuxnet computer worm, a destructive program that appears to have wiped out roughly a fifth of Iran's nuclear

Multimedia



Graphic

How Stuxnet Spreads

Detection & Defense

- Hardening programs
- Detecting scanning
- Rate limiting
- Signature-based detection
  - Exploit-based signatures
  - Vulnerability-based signatures