



# Malicious Code/Logic



# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust



# Malicious Logic 定义

- ◆ Set of instructions that cause site security policy to be violated

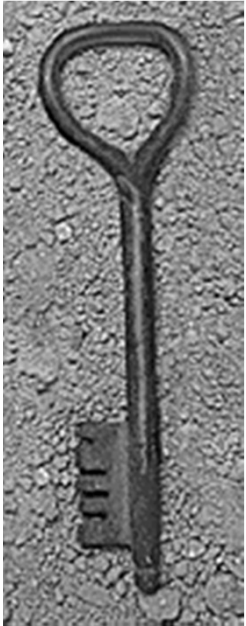


# Example

- ◆ Shell script on a UNIX system:

```
cp /bin/sh /tmp/.xyzzzy  
chmod u+s,o+x /tmp/.xyzzzy  
rm ./ls  
ls $*
```

- ◆ Place in program called “ls” and trick someone into executing it
- ◆ You now have a setuid-to-*them* shell!



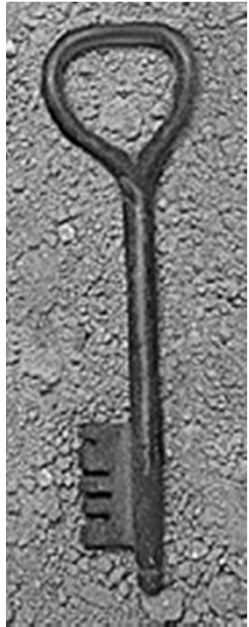
# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust



# Trojan Horse

- ◆ Program with an *overt* purpose (known to user) and a *covert* purpose (unknown to user)
  - Often called a Trojan
  - Named by Dan Edwards in 1974 Anderson Report
- ◆ Example: previous script is Trojan horse
  - Overt purpose: list files in directory
  - Covert purpose: create setuid shell



## Example: NetBus

- ◆ Designed for Windows NT system
- ◆ Victim uploads and installs this
  - Usually disguised as a game program, or in one
- ◆ Acts as a server, accepting and executing commands for remote administrator
  - This includes intercepting keystrokes and mouse motions and sending them to attacker
  - Also allows attacker to upload, download files



# Replicating Trojan Horse

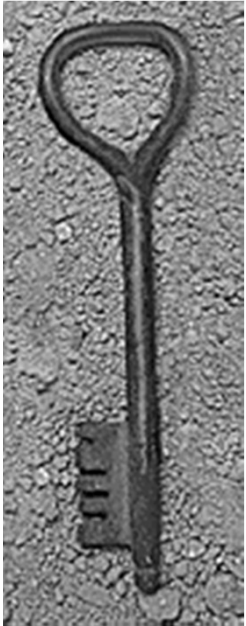
- ◆ Trojan horse that makes copies of itself
  - Also called *propagating Trojan horse*
- ◆ Hard to detect
  - 1976: Karger and Schell suggested modifying compiler to include Trojan horse that copied itself into specific programs including later version of the compiler
  - 1980s: Thompson implements this



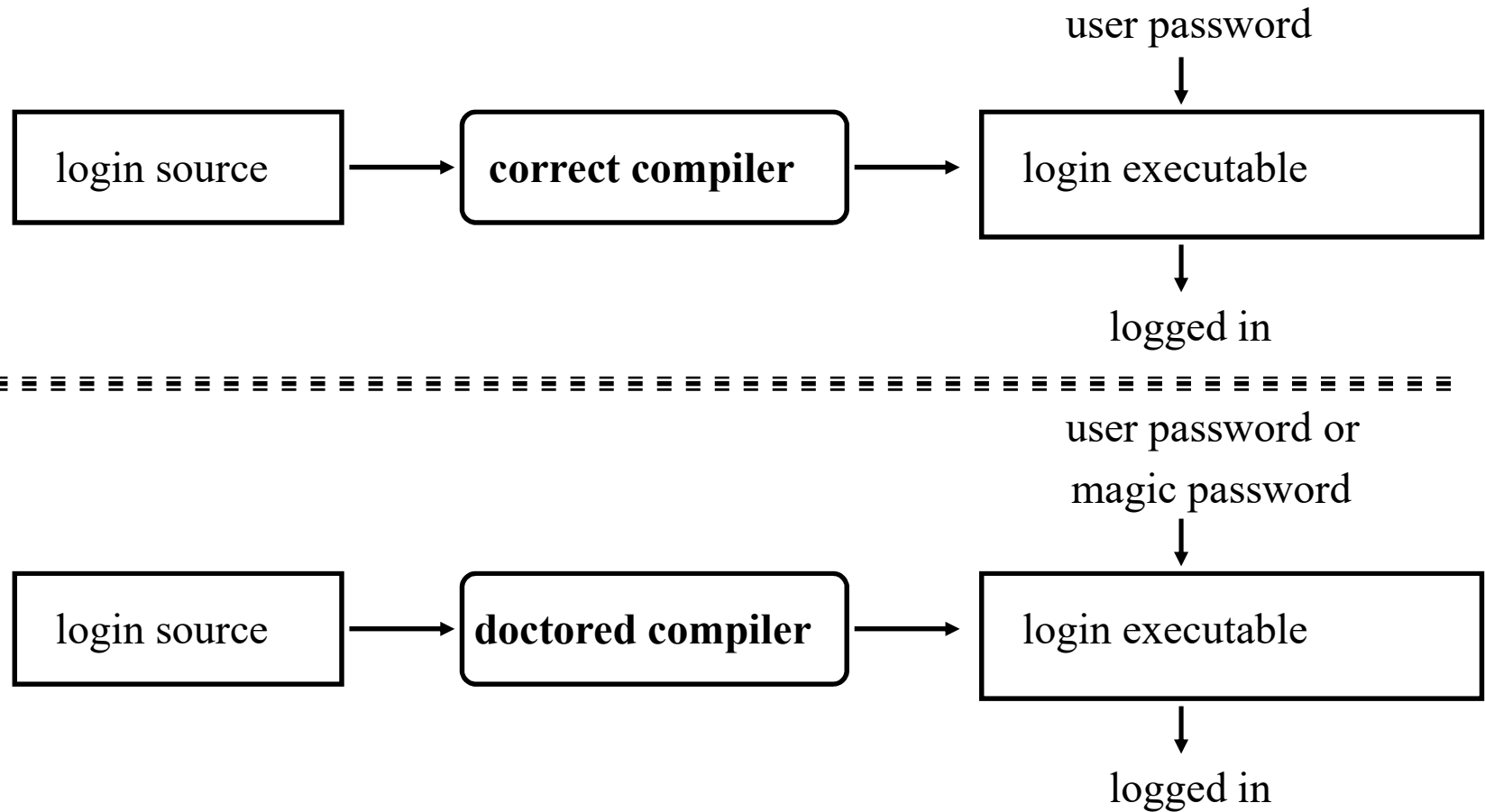


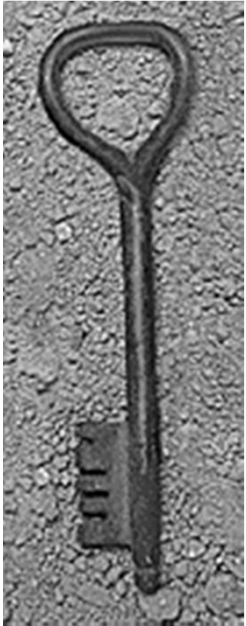
# Thompson's Compiler 带有自我传播的特洛伊木马

- ◆ Modify the compiler so that when it compiles *login* , *login* accepts the user's correct password or a fixed password (the same one for all users)
- ◆ Then modify the compiler again, so when it compiles a new version of the compiler, the extra code to do the first step is automatically inserted
- ◆ Recompile the compiler
- ◆ Delete the source containing the modification and put the undoctored source back

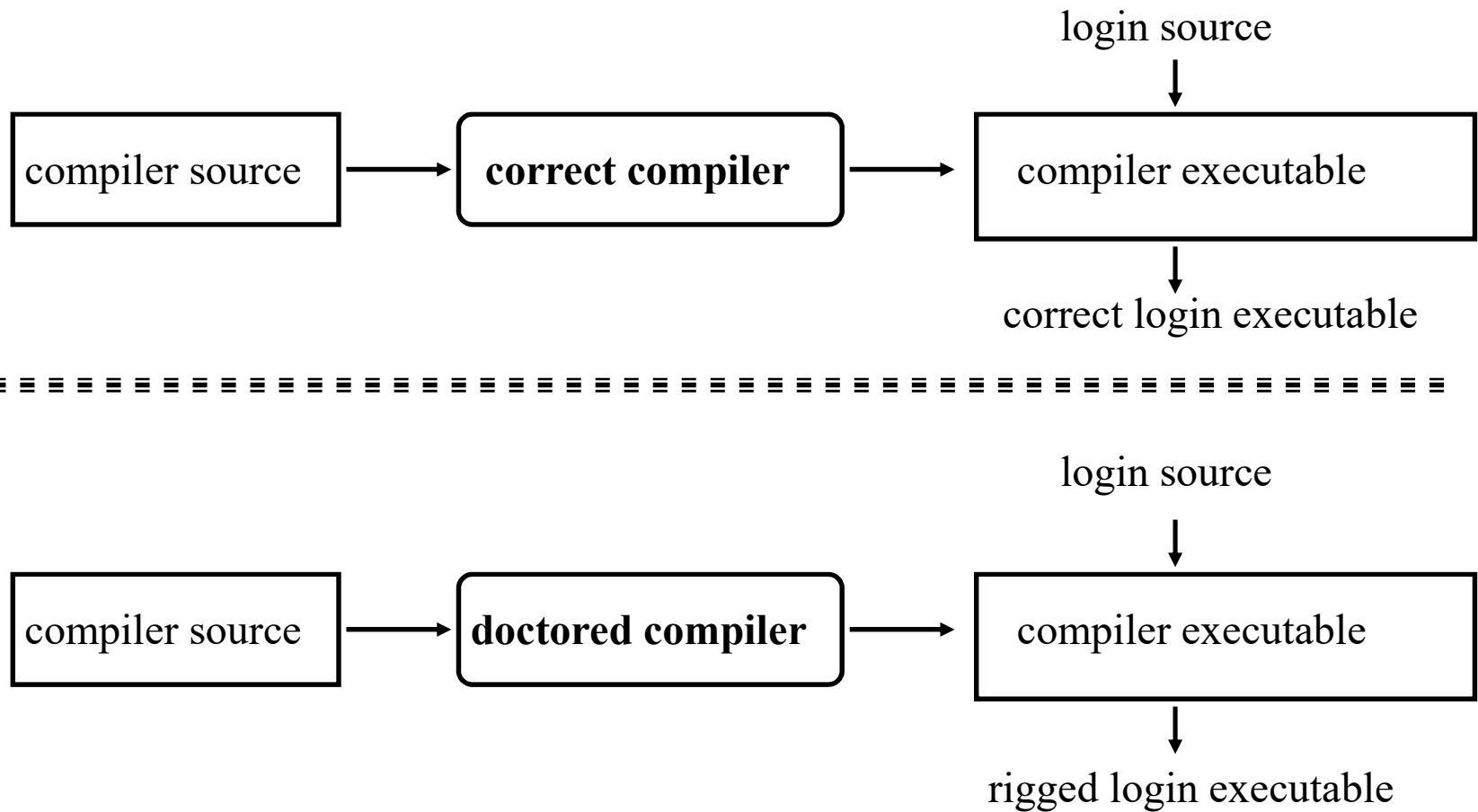


# The Login Program





# The Compiler





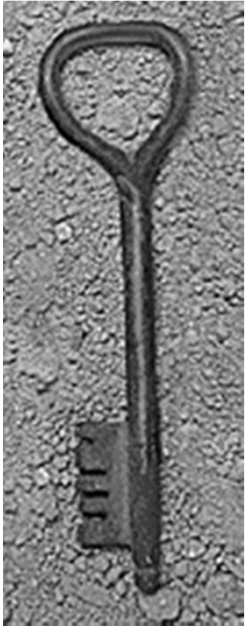
# Comments

- ◆ Great pains taken to ensure second version of compiler never released
  - Finally deleted when a new compiler executable from a different system overwrote the doctored compiler
- ◆ The point: *no amount of source-level verification or scrutiny will protect you from using untrusted code*
  - Also: having source code helps, but does not ensure you're safe



# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust



# Computer Virus

- ◆ Program that inserts itself into one or more files and performs some action
  - *Insertion phase* is inserting itself into file
  - *Execution phase* is performing some (possibly null) action
- ◆ Insertion phase *must* be present
  - Need not always be executed
  - Lehigh virus inserted itself into boot file only if boot file not infected



# Pseudocode

beginvirus:

if *spread-condition* then begin

for *some set of target files* do begin

if *target is not infected* then begin

*determine where to place virus instructions*

*copy instructions from beginvirus to endvirus into target*

*alter target to execute added instructions*

end;

end;

end;

*perform some action(s)*

*goto beginning of infected program*

endvirus:



# History

- ◆ Programmers for Apple II wrote some in 1980
  - Not called viruses; very experimental
- ◆ Fred Cohen
  - Graduate student who described them in 1983
  - Teacher (Adleman) named it “computer virus”
  - Tested idea on UNIX systems and UNIVAC 1108 system in 1984





# Cohen's Experiments

- ◆ UNIX systems: goal was to get superuser privileges
  - Max time 60m, min time 5m, average 30m
  - Virus small, so no degrading of response time
  - Virus tagged, so it could be removed quickly
- ◆ UNIVAC 1108 system (MAC): goal was to spread
  - Implemented simple security property of Bell-LaPadula
  - As writing not inhibited (no \*-property enforcement), viruses spread easily



# First Reports

- ◆ Brain (Pakistani) virus (1986)
  - Written for IBM PCs
  - Alters boot sectors of floppies, spreads to other floppies
- ◆ MacMag Peace virus (1987)
  - Written for Macintosh
  - Prints “universal message of peace” on March 2, 1988 and deletes itself



# More Reports

- ◆ Duff's experiments (1987)
  - Wrote a Bourne shell script virus
  - Small virus placed on UNIX system, spread to 46 systems in 8 days
- ◆ Highland's Lotus 1-2-3 virus (1989)
  - Stored as a set of commands in a spreadsheet and loaded when spreadsheet opened
  - Changed a value in a specific row, column and spread to other files



# Types of Viruses

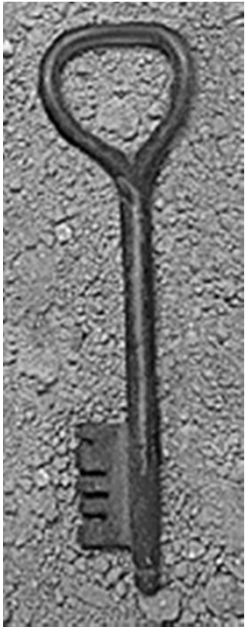
- ◆ Boot sector infectors
- ◆ Executable infectors
- ◆ Multipartite viruses
- ◆ TSR viruses
- ◆ Stealth viruses
- ◆ Encrypted viruses
- ◆ Polymorphic viruses
- ◆ Macro viruses



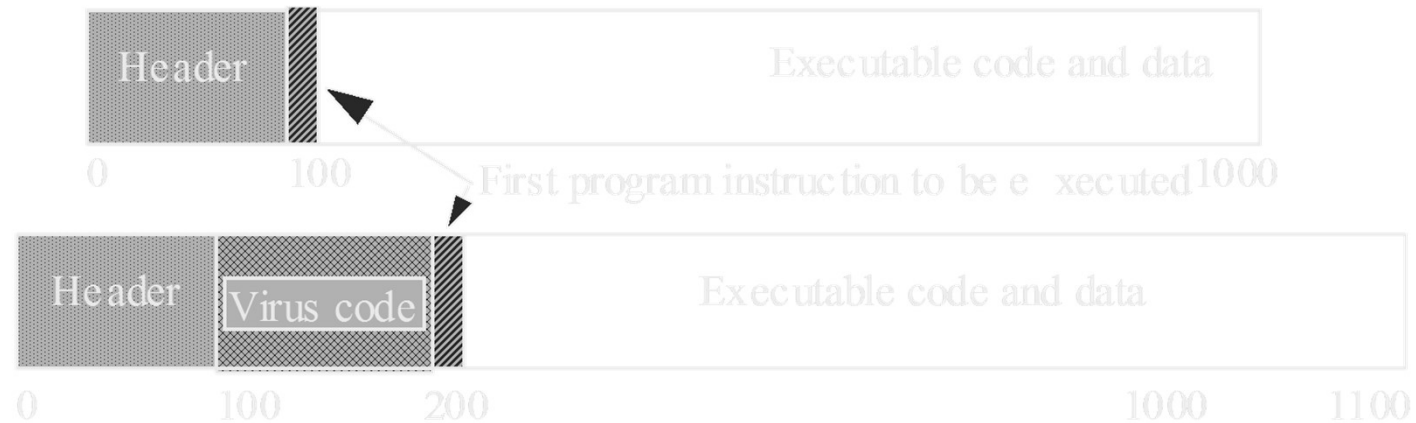
# Boot Sector Infectors

## 开机启动病毒

- ◆ A virus that inserts itself into the boot sector of a disk
  - Section of disk containing code
  - Executed when system first “sees” the disk
    - Including at boot time ...
- ◆ Example: Brain virus
  - Moves disk interrupt vector from 13H to 6DH
  - Sets new interrupt vector to invoke Brain virus
  - When new floppy seen, check for 1234H at location 4
    - If not there, copies itself onto disk after saving original boot block



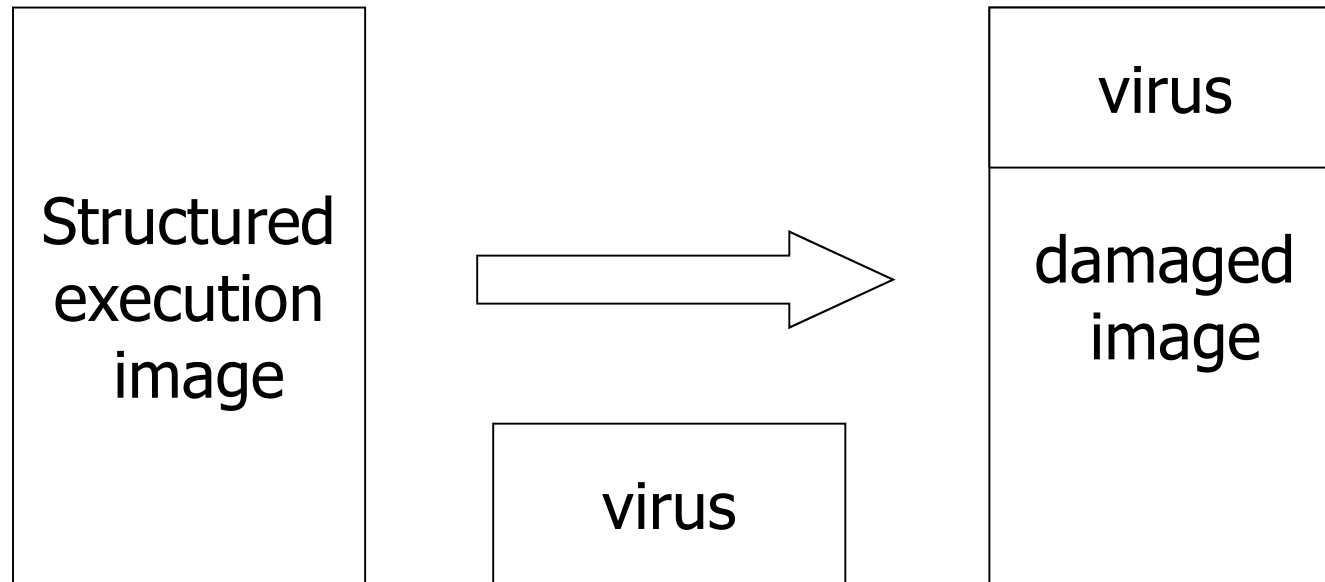
# Executable Infectors



- ◆ A virus that infects executable programs
  - Can infect either .EXE or .COM on PCs
  - May prepend itself (as shown) or put itself anywhere, fixing up binary so it is executed at some point

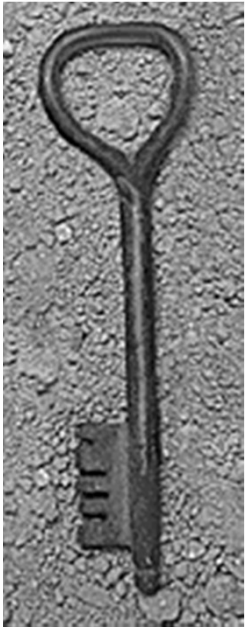


## Means of attaching: overwriting (virus *replaces* part of program)

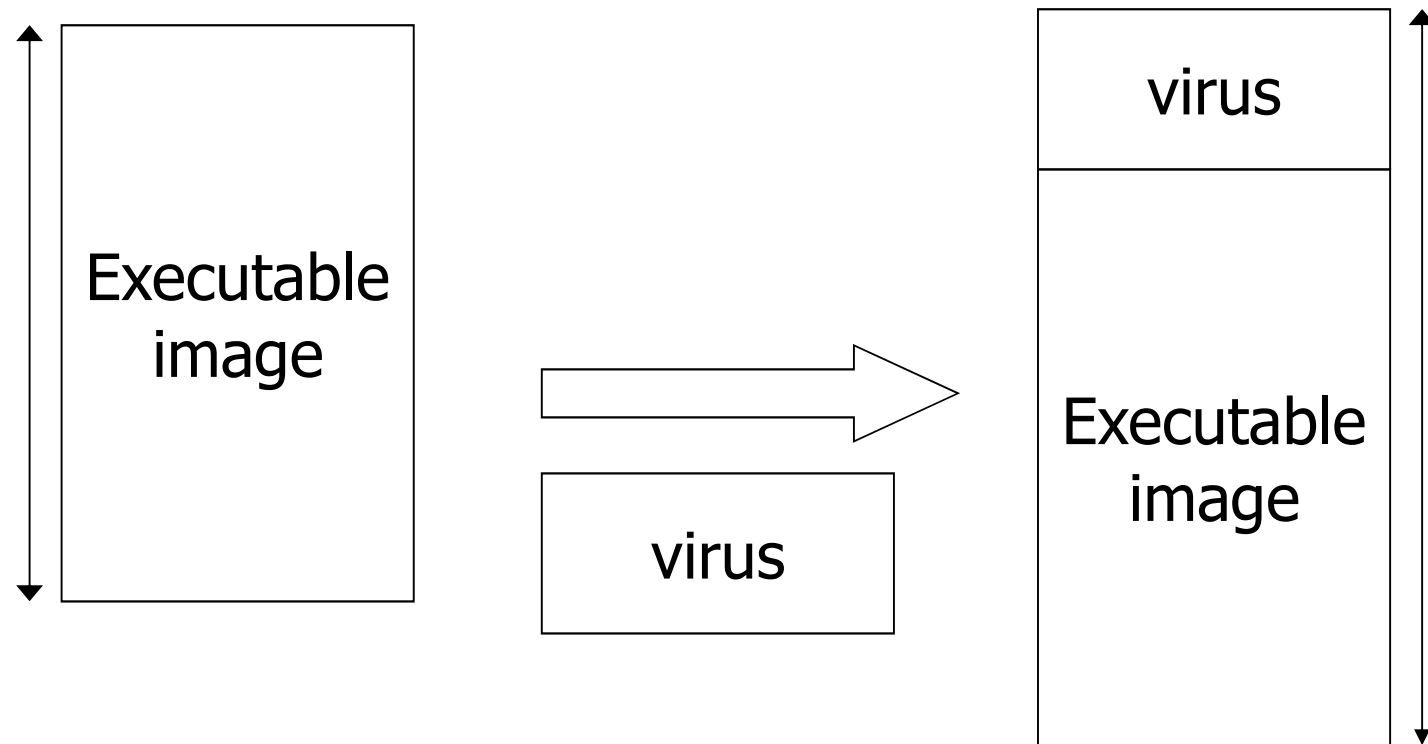


病毒替换程序的部分代码  
易被发现

- Virus overwrites an executable file
- Easiest mechanism
- Since original program is damaged easily detected

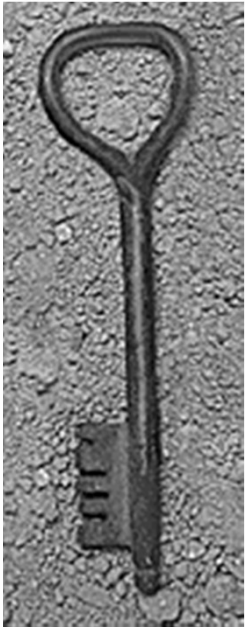


Means of attaching: at the beginning  
(virus is *appended* to program)

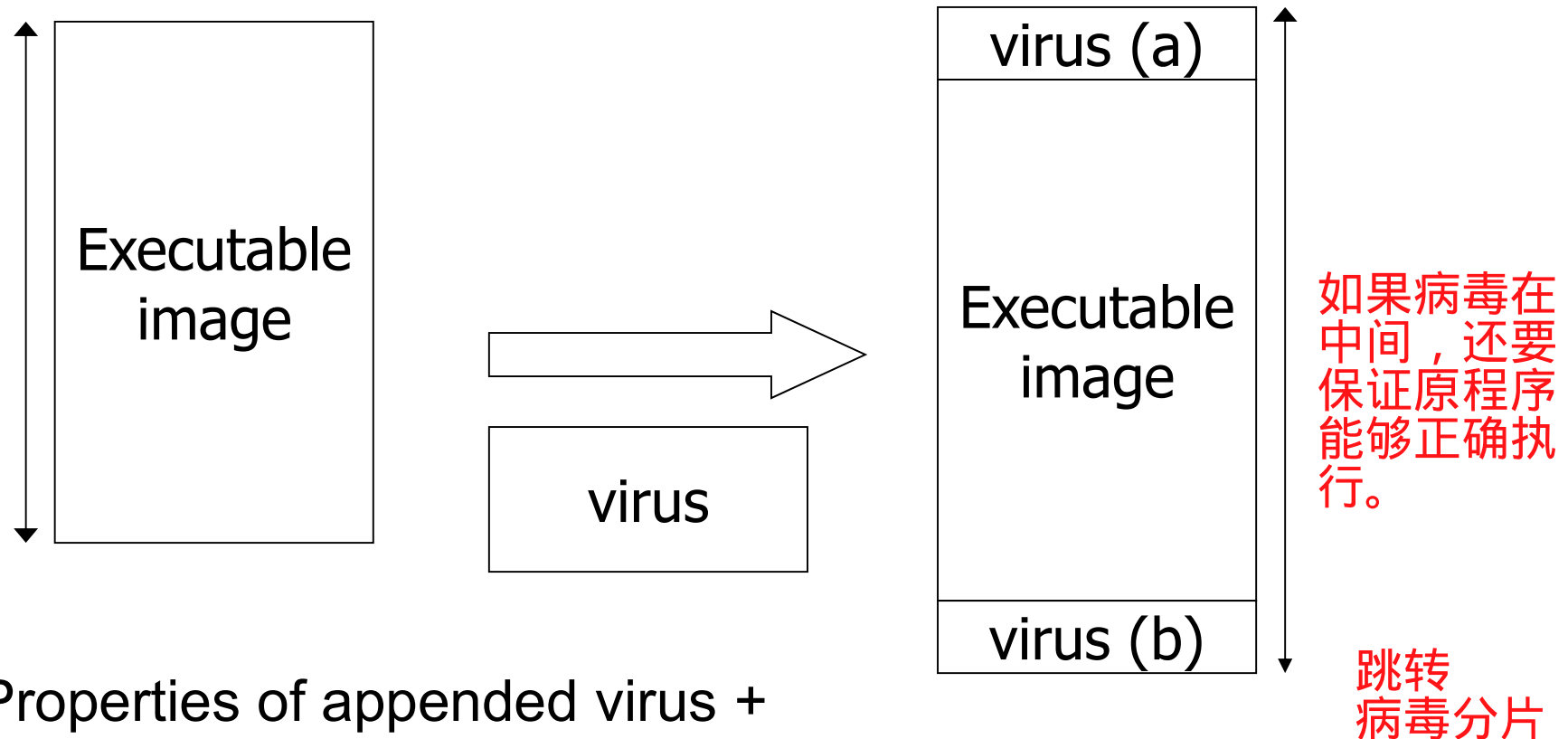


- Improved stealth because original program is intact
- If original program is large, copying it may be slow
- File size grows if multiple infections occur

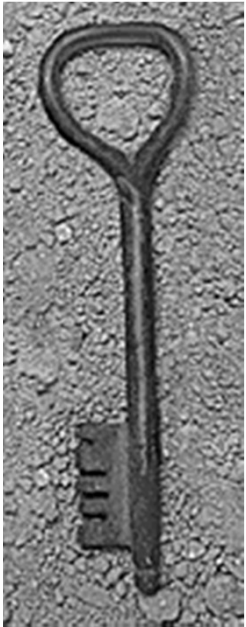




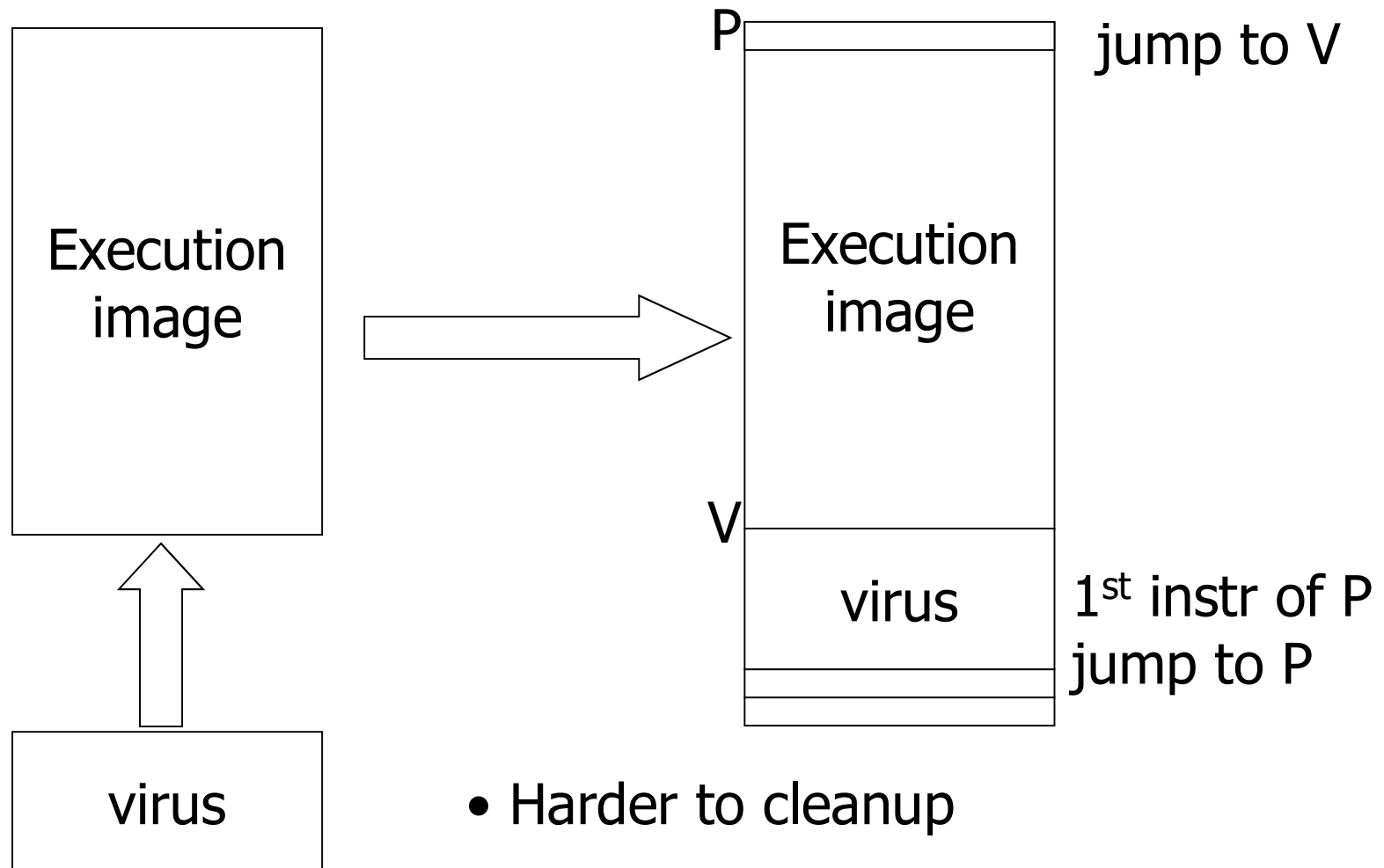
## Means of attaching: beginning and end (virus *surrounds* program)



- Properties of appended virus +
- Ability to clean up and avoid detection
- Example: attach to program that constructs file lists with sizes; modify after program has ended

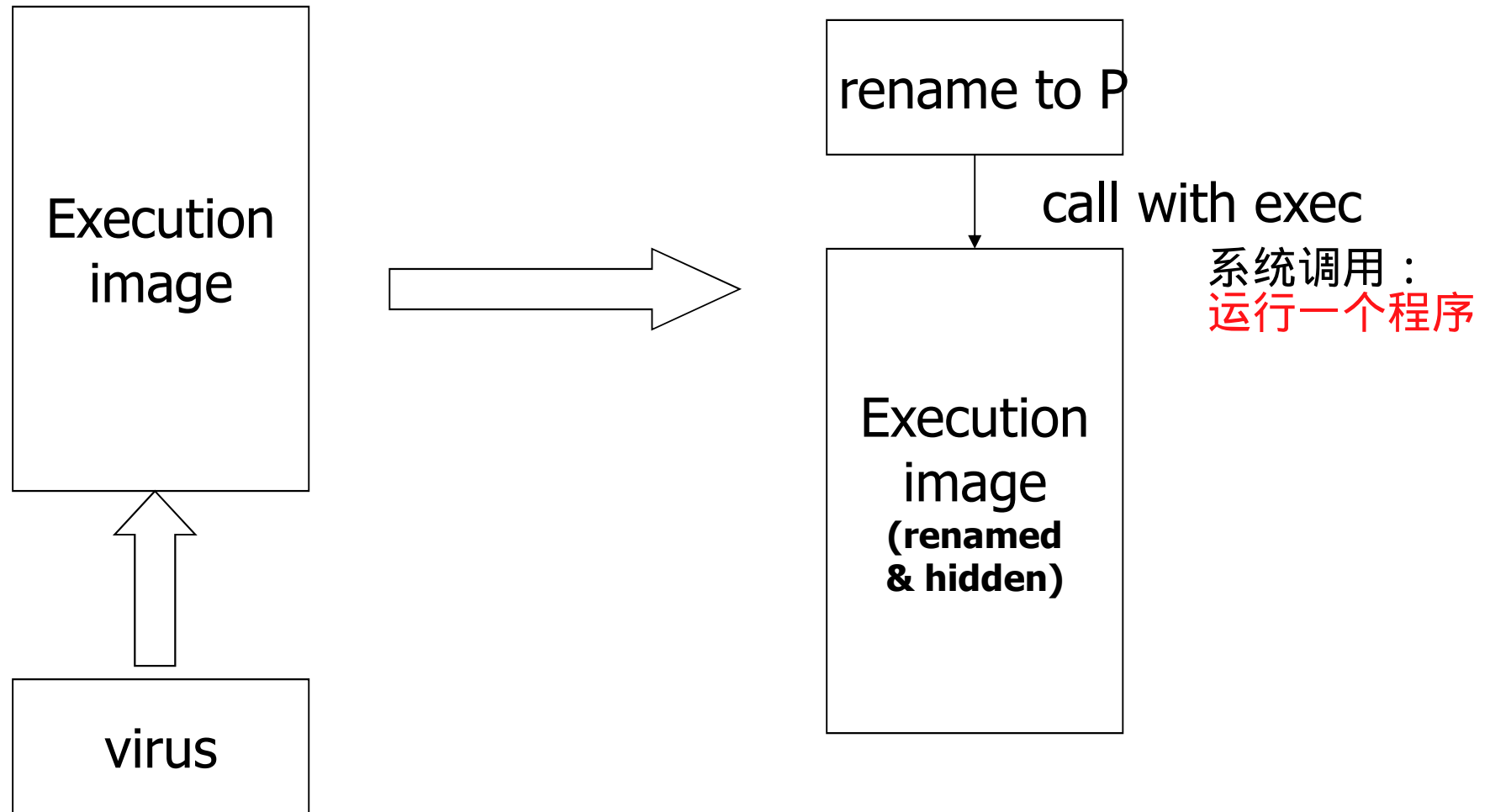


Means of attaching: intersperse  
(virus is *integrated* into program)





# Means of attaching: companions

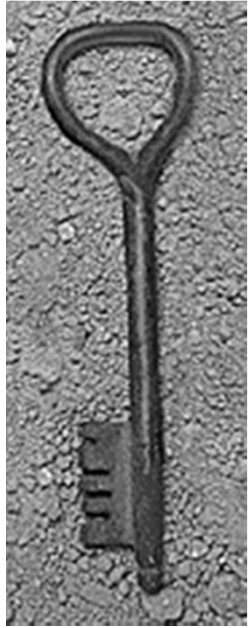




# Executable Infectors (*con't*)

## ◆ Jerusalem (Israeli) virus

- Checks if system infected
  - If not, set up to respond to requests to execute files
- Checks date
  - If not 1987 or Friday 13th, set up to respond to clock interrupts and then run program
  - Otherwise, set destructive flag; will delete, not infect, files
- Then: check all calls asking files to be executed
  - Do nothing for COMND.COM
  - Otherwise, infect or delete
- BUG!: doesn't set signature when .EXE executes
  - So .EXE files continually reinfected, size goes up quickly



# Multipartite Viruses

多体病毒：感染可执行程序，也可感染宏定义

- ◆ A virus that can infect either boot sectors or executables
- ◆ Typically, two parts
  - One part boot sector infector
  - Other part executable infector



# TSR Viruses

- ◆ A virus that stays active in memory after the application (or bootstrapping, or disk mounting) is completed
  - TSR is “**Terminate and Stay Resident**” 永远驻留在系统内存
- ◆ Examples: Brain, Jerusalem viruses
  - Stay in memory after program or disk mount is completed



# Stealth Viruses

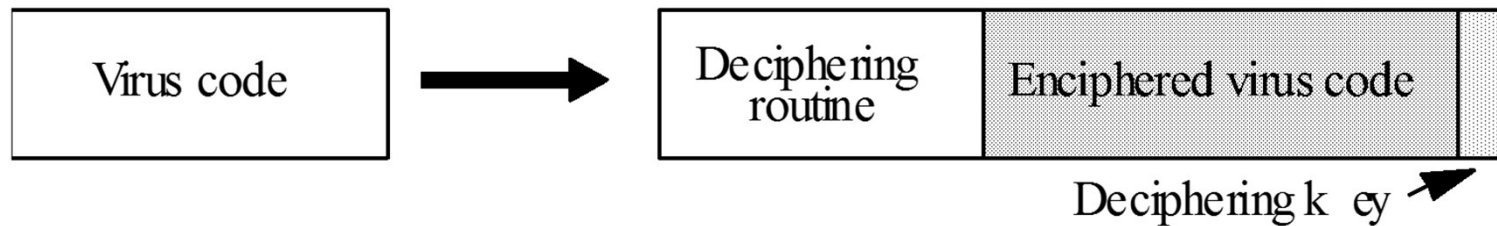
- ◆ A virus that conceals infection of files
- ◆ Example: IDF virus modifies DOS service interrupt handler as follows:
  - Request for file length: return length of *uninfected* file
  - Request to open file: temporarily disinfect file, and reinfect on closing
  - Request to load file for execution: load infected file



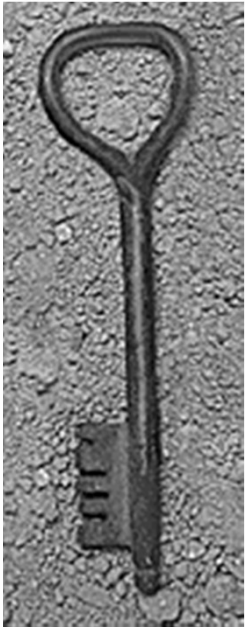
# Encrypted Viruses

- ◆ A virus that is enciphered except for a small deciphering routine
  - Detecting virus by signature now much harder as most of virus is enciphered

每次病毒感染时代码是一样的，容易被发现，所以病毒进行加密。

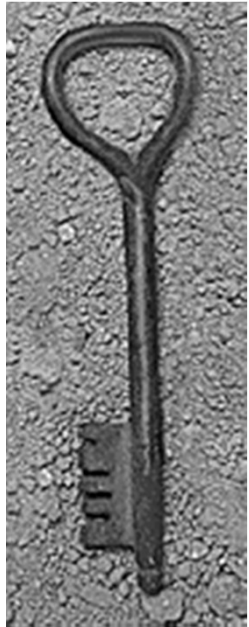






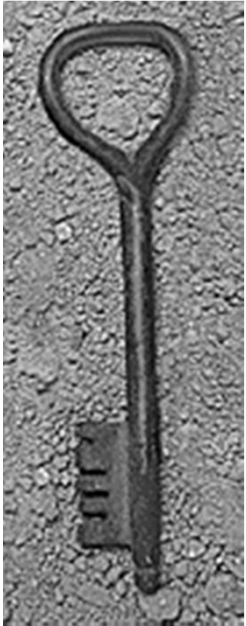
# Example

```
(* Decryption code of the 1260 virus *)
(* initialize the registers with the keys *)
rA = k1; rB = k2;
(* initialize rC with the virus; starts at sov, ends at eov *)
rC = sov;
(* the encipherment loop *)
while (rC != eov) do begin
    (* encipher the byte of the message *)
    (*rC) = (*rC) xor rA xor rB;
    (* advance all the counters *)
    rC = rC + 1;
    rA = rA + 1;
end
```

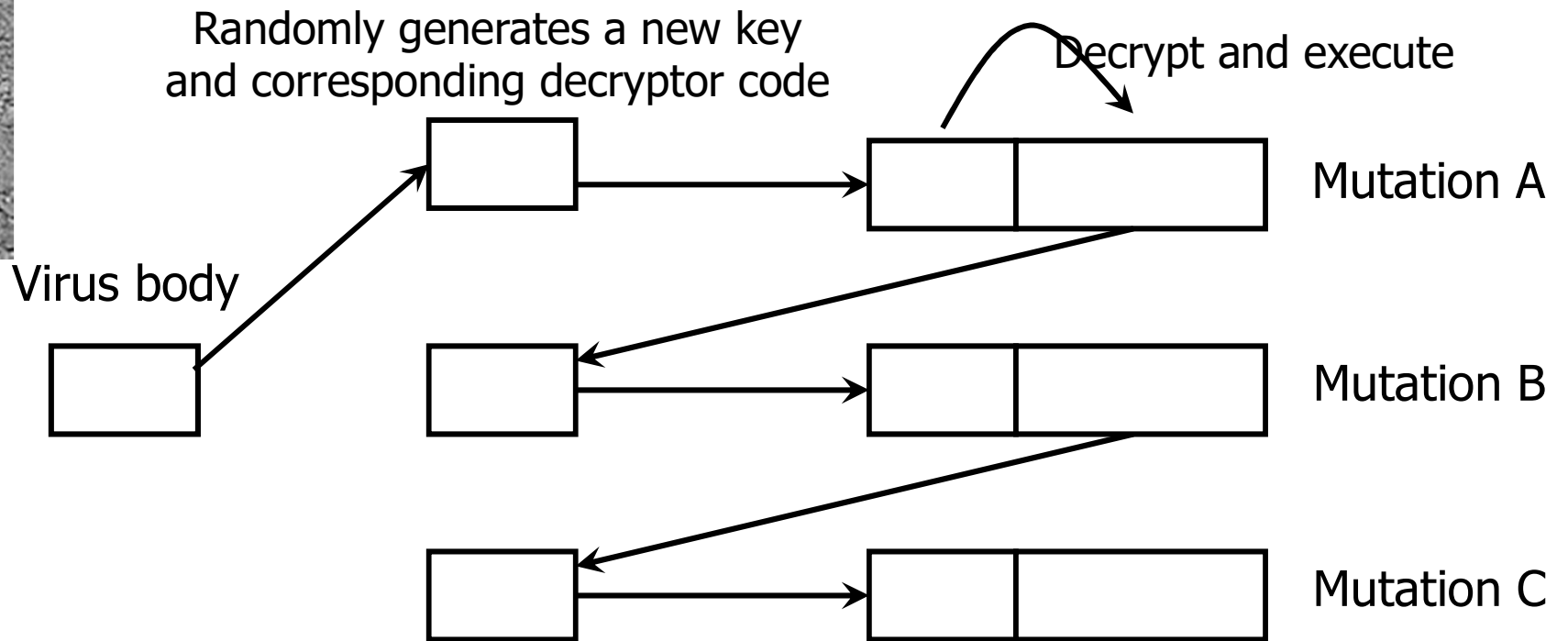


## Anti-virus tools' answer to encryption

- ◆ Select the signature from the unencrypted portion of the code, i.e. the decryption engine
- ◆ Problems:
  - Anti-virus tools usually want to determine which virus is present, not just determine that some virus is present (in order to “disinfect”).
    - Can emulate the decryption then further analyze the decrypted code.
  - virus writers have responded by obscuring the encryption engine through mutations
- ◆ It's a game of cat and mouse



# Virus Emulation





# Polymorphic Viruses

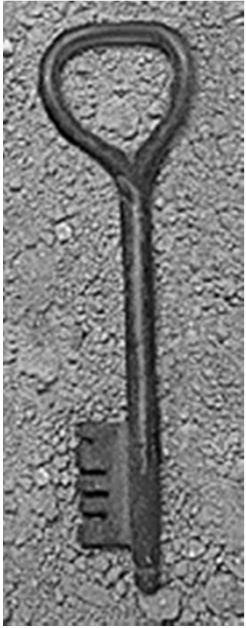
- ◆ Polymorphic = “many forms”. A virus that changes its form each time it replicates
- ◆ Prevent signature detection by changing the “signature” or instructions used for deciphering
- ◆ At instruction level: substitute instructions    指令级别
- ◆ At algorithm level: different algorithms to achieve the same purpose  
    算法级别的多态，使病毒每次演化产生不同机器码，杀病毒软件很难发现。
  - Given two code segments, evaluating their semantic equivalency is an undecidable problem!



# Example

- ◆ These are different instructions (with different bit patterns) but have the same effect:
  - add 0 to register
  - subtract 0 from register
  - xor 0 with register
  - no-op
- ◆ Polymorphic virus would pick randomly from among these instructions

# Example



```
(* Polymorphic code of the 1260 virus *)
(* initialize the registers with the keys *)
rA = k1; rB = k2;
rD = rD + 1; (* random code*)
(* initialize rC with the virus;  starts at sov, ends at eov *)
rC = sov;
rC = rC + 1; (* random code*)
(* the encipherment loop *)
while (rC != eov) do begin
    rC = rC - 1; (* random code*)
    (* encipher the byte of the message *)
    (*rC) = (*rC) xor rA xor rB;
    (* advance all the counters *)
    rC = rC + 2;
    rD = rD + 1; (* random code*)
    rA = rA + 1;
end
While ( rC != sov) do begin (* random code*)
    rD = rD - 1; (* random code*)
    rC = rC - 1; (* random code*)
end (* random code*)
```



# Macro Viruses

- ◆ A virus composed of a sequence of instructions that are interpreted rather than executed directly
- ◆ Can infect either executables (Duff's shell virus) or data files (Highland's Lotus 1-2-3 spreadsheet virus)
- ◆ Independent of machine architecture
  - But their effects may be machine dependent

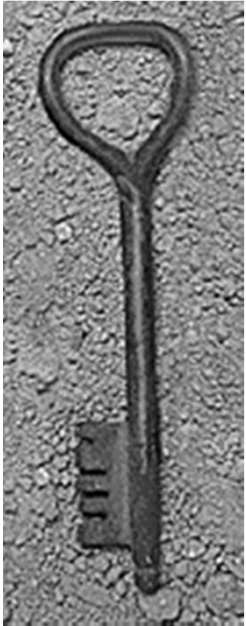


# Example

## ◆ Melissa

- Infected Microsoft Word 97 and Word 98 documents
  - Windows and Macintosh systems
- Invoked when program opens infected file
- Installs itself as “open” macro and copies itself into Normal template
  - This way, infects any files that are opened in future
- Invokes mail program, sends itself to everyone in user’s address book





# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust



# Viruses vs. Worms

## VIRUS

一定要感染

- ◆ Propagates by infecting other programs
- ◆ Usually inserted into host code (not a standalone program)

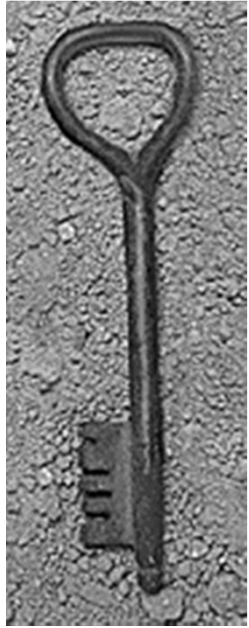


## WORM

可独立存在

- ◆ Propagates automatically by copying itself to target systems
- ◆ Is a standalone program





# Computer Worms Origins: distributed computations

- ◆ Schoch & Hupp (1982): animations, broadcast messages
- ◆ Segment: part of program copied onto workstation
- ◆ Segment processes data, communicates with worm's controller
- ◆ Any activity on workstation caused segment to shut down



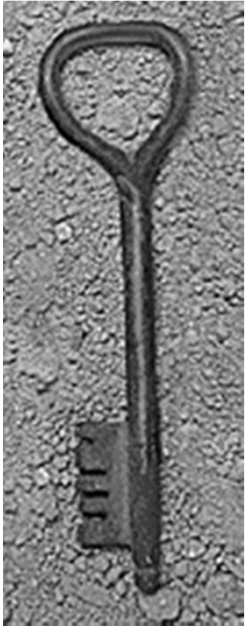
# Example: Christmas Worm

- ◆ Distributed in 1987, designed for IBM networks
- ◆ Electronic letter instructing recipient to save it and run it as a program
  - Drew Christmas tree, printed “Merry Christmas!”
  - Also checked address book, list of previously received email and sent copies to each address
- ◆ Shut down several IBM networks
- ◆ Really, a macro worm
  - Written in a command language that was interpreted
  - Can cross different system platform!

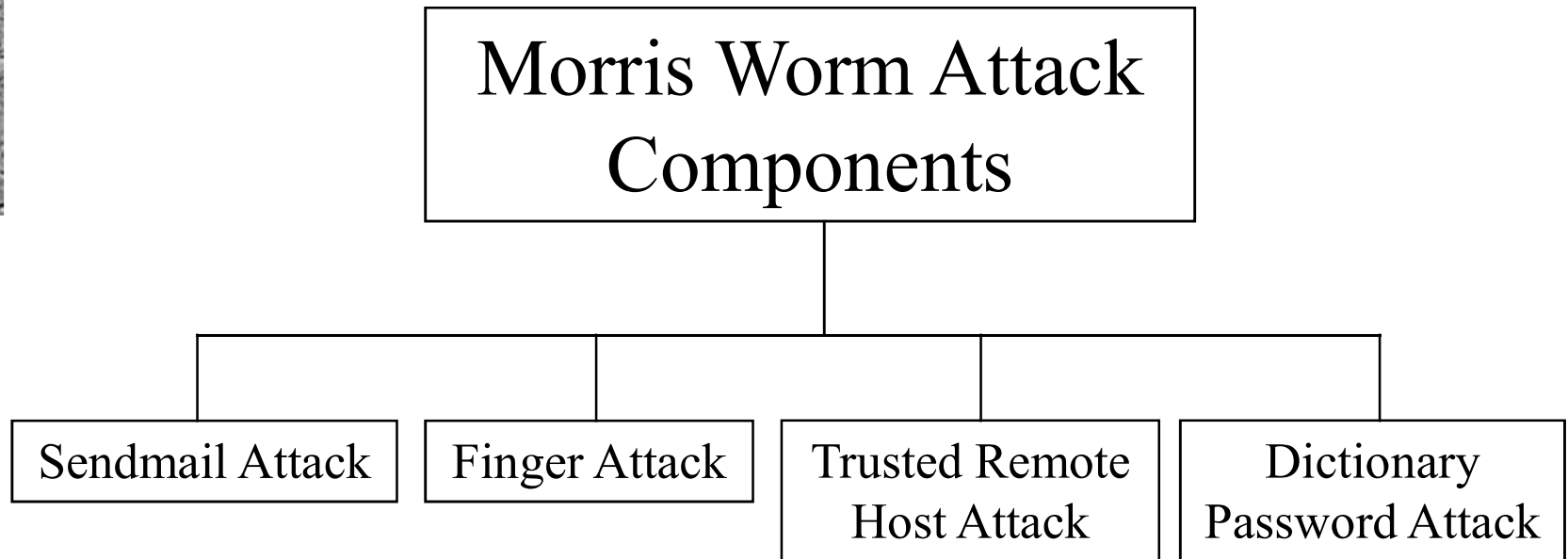


# Case study: 1988 Internet Worm

- ◆ Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now a computer science professor at MIT
- ◆ Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- ◆ Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- ◆ Disabled 3000--4000 machines, or 5% of the machines on the Internet then, \$10-100M damage



# Components of Worm





# Overview of the Morris Worm

- ◆ Worm selects host for infection.
- ◆ Place hook (grappling hook) on host.
- ◆ Causes hook to compile and run.
- ◆ Executing hook, the hook will copy remainder of code over.



# Step 1: About to Infect

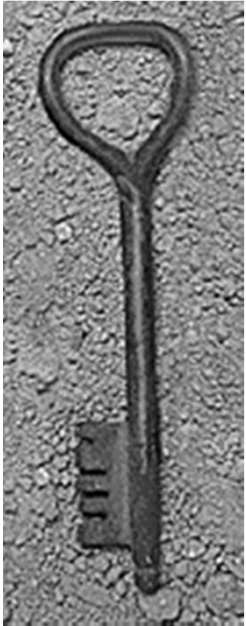
- ◆ Status: infecting machine has found a victim

Server Worm

Socket established on victim





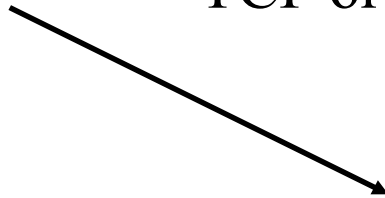


## Step 2: Placing Grappling hook

- ◆ Status: Send a small hook code across

Server Worm

TCP or SMTP connection





## Step 3: Pull in code from server

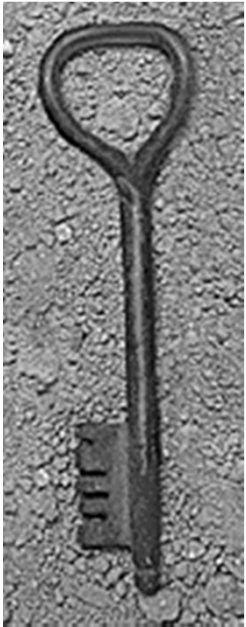
- ◆ Status: Send challenge string back, transfer Sun, VAX binaries and vector Server Worm

Server Worm

Check challenge  
string; transmit  
files

Send challenge;  
Copy files back;  
Exec shell and  
maintain socket





## Step 4: Get further compiler commands from server worm

- ◆ Status: Server sends code to compile

Server Worm

Send shell commands  
to attempt compile  
of binaries  
(one after the other)

Execute compile commands  
as sent by Server Worm; only  
the appropriate binary will  
link.





## Step 5: Hide and gather information

- ◆ Status: New worm generates victim lists





## Step 6: Attempt infection

- ◆ Read through `hosts.equiv`, `/.rhosts`, `.forward` (not user `.rhosts`!)
- ◆ Attempt to break each user password ...
  - with simple guesses
  - with internal dictionary
  - with UNIX online dictionary
- ◆ For any successful password compromise
  - look for remote machines where the user has an account (`.forward .rhosts`)
  - For each, attempt a remote shell (`rexec`)
- ◆ Loop “forever” trying to infect new hosts



# Major Tricks

- ◆ Sendmail attack
- ◆ Finger attack
- ◆ Trusted remote host attack
- ◆ Dictionary-based password attack



# Sendmail Attack

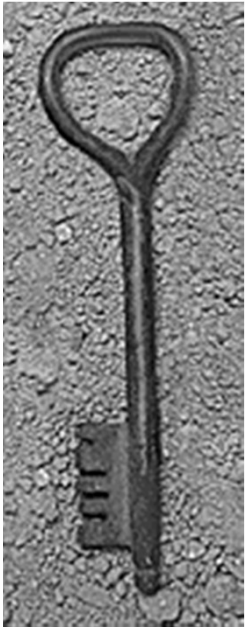
- ◆ DEBUG command in Sendmail (early ver.)
  - Possible to execute a command on a remote machine by sending an SMTP message
  - Intended for in-house debugging
- ◆ Simply put, the worm did the following:
  - Put commands in the body of a mail message
  - While running, deleted mail header
  - Body was run through an interpreter, and it:
    - Stored small 99-line program in a file (grappling hook)
    - Compiled program
    - Started it executing



# Fingerd Attack

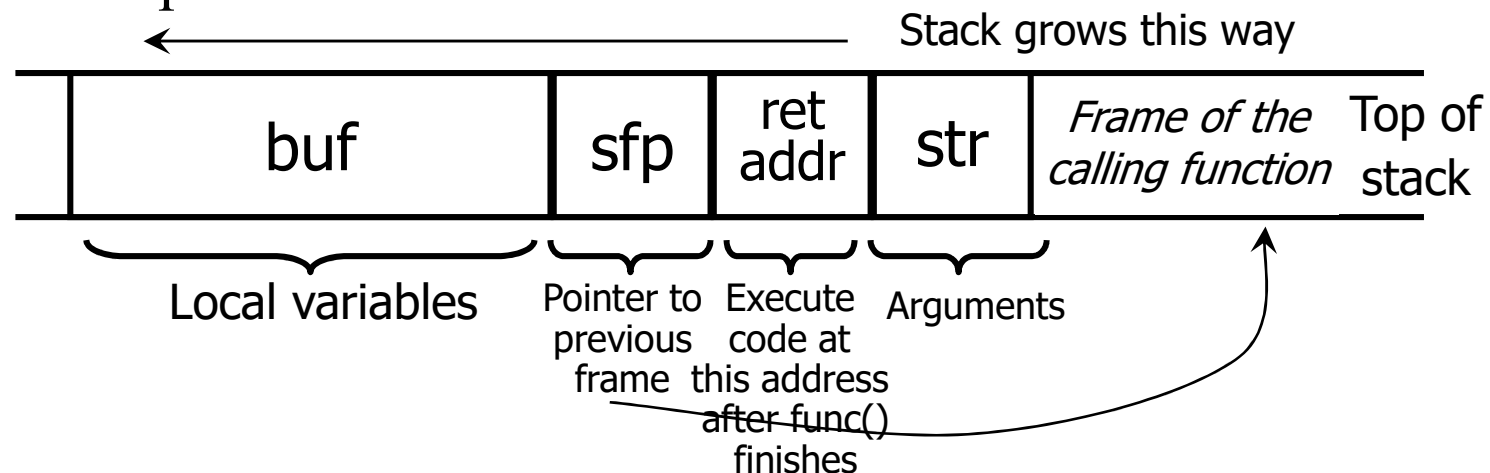
- ◆ Buffer Overflow attack against a vulnerable version of fingerd on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
  - Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy)

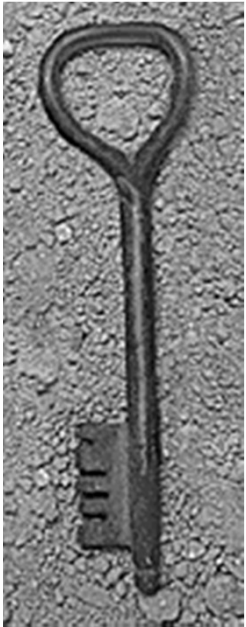




# Buffer Overflow Attack

- ◆ A kind of buffer overflow attack
  - Returned information was stored in a large buffer. Bounds weren't checked on this buffer.
  - Worm sent "too many" bytes, with the extra bytes actually being program code. This code over-wrote the original code
  - PC of the routine, after return from obtaining data, now pointed to the new code

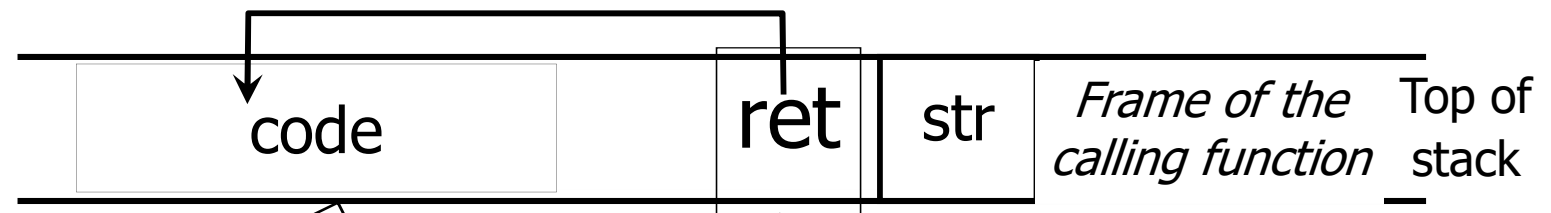




# Buffer Overflow Attack



- ◆ Suppose buffer contains attacker-created string



Attacker puts actual assembly instructions into his input string, e.g., binary code of `execve("/bin/sh")`

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

- ◆ When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root



# Trusted Remote Host Attack

- ◆ Many Unixes have a way to indicate that some hosts can be trusted 'just as much' as the current host. Use rsh to remote execute a command.
- ◆ The names of these hosts are stored in world-readable places, and can be network-specific or user-specific.
- ◆ Worm targeted:
  - hosts in this list of network-specific trusted hosts
  - hosts in user-specific list whenever a password was compromised



# Dictionary-based password attack

- ◆ The worm used a very simple form:
  - Scan password file to get user names
  - Try common passwords using a quick implementation of crypt(3).
    - Account name
    - Account name reversed
    - First and last name
    - Short standard online dictionary

*A desire to help system administrators quickly identify such easily guessable passwords was behind Moffet's development of Crack.*



# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust



# Others: Rabbits, Bacteria

- ◆ A program that absorbs all of some class of resources
- ◆ Example: for UNIX system, shell commands [Dennis 1979]:

```
while true
do
    mkdir x; chdir x    文件描述的空间被用掉
done
```
- ◆ Exhausts either disk space or file allocation table (inode) space, may crash the system. But very easy to trace back!
- ◆ Countermeasures: Quote for each user



# Others: Logic Bombs

## 逻辑炸弹

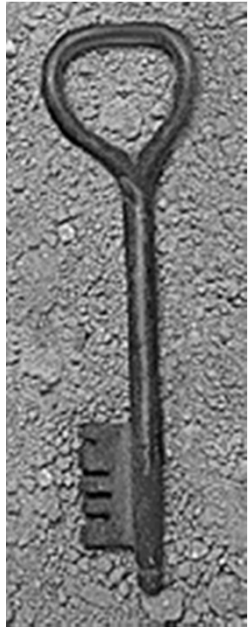
- ◆ A program that performs an action that violates the site security policy when some external event occurs (Ex. Special Holiday, Special Name,...)
- ◆ Example: program that deletes company's payroll records when one particular record is deleted
  - The “particular record” is usually that of the person writing the logic bomb
  - Idea is if (when) he or she is fired, and the payroll record deleted, the company loses *all* those records
  - =rand(100,2) in Microsoft Word
  - 联通 in Microsoft Notebook



# Overview

- ◆ Defining malicious logic
- ◆ Types
  - Trojan horses
  - Computer viruses and worms
  - Other types
- ◆ Defenses
  - Properties of malicious logic
  - Trust





# How Hard Is It to Write a Virus?

- ◆ 2268 matches for "virus creation tool" in CA's Spyware Information Center
  - Including dozens of poly- and metamorphic engines
- ◆ OverWritting Virus Construction Toolkit
  - "The perfect choice for beginners"
- ◆ Biological Warfare Virus Creation Kit
  - Note: all viruses will be detected by Norton Anti-Virus
- ◆ VBS Worm Generator (for Visual Basic worms)
  - Used to create the Anna Kournikova worm
- ◆ Many others



# Theory of Virus Detection

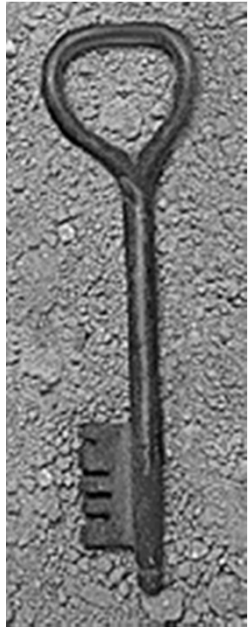
- ◆ There is No algorithm to detect all possible virus/malicious code.
- ◆ Proved by Cohen in “Computers and Security” in 1989.



# No perfect solutions : Now what?

## ◆ Signature-based antivirus

- Look for known patterns in malicious code
- Always a battle with the attacker
- *Great business model!*

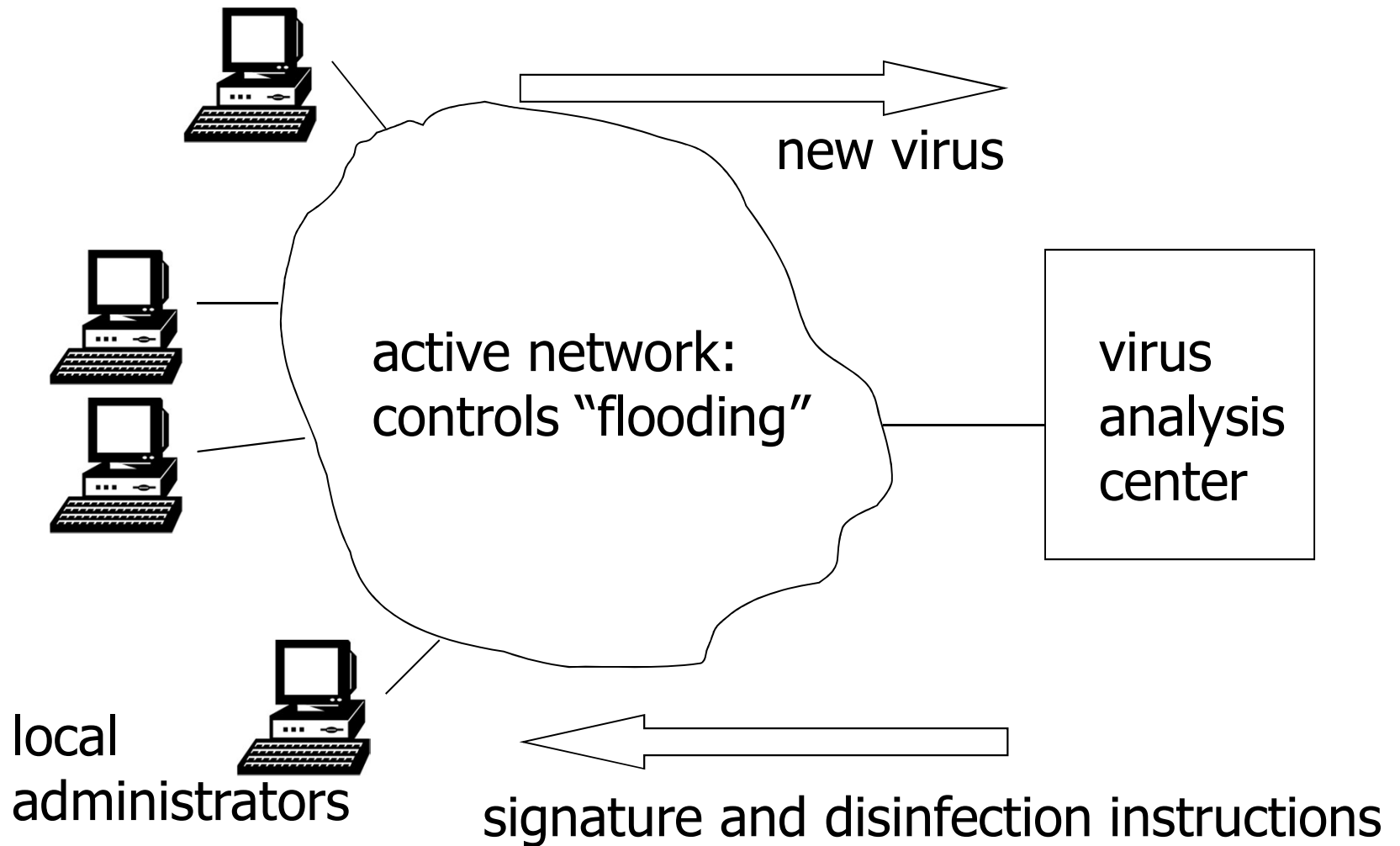


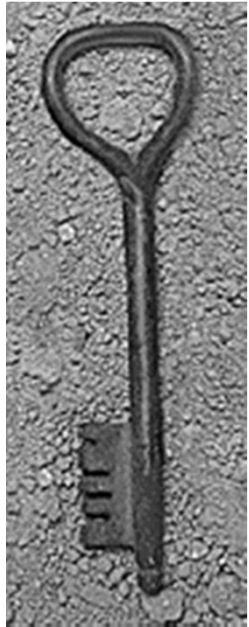
# Virus Analysis

- ◆ Analysis of virus by human expert
  - slow: by the time signature has been extracted, posted to AV tool database, downloaded to users, virus may have spread widely.
    - pre-1995: 6 months to a year for virus to spread world-wide
    - mid-90's: a few months
    - now: days or hours
  - labor-intensive: too many new viruses
    - currently, 8-10 new viruses per day
  - can't handle epidemics:
    - queue of viruses to be analyzed overflows
    - heavy demand on server that posts signatures & fixes
- ◆ Automated analysis, e.g. “Immune System”
  - developed at IBM Research
  - licensed to Symantec



# Immune System Architecture





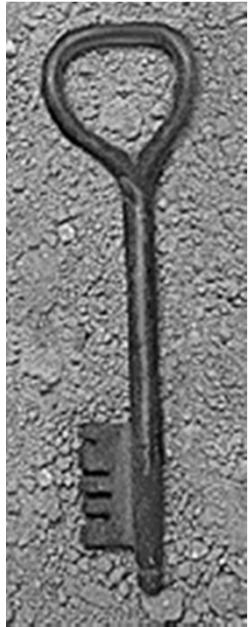
# Signature Extraction at VAC

- ◆ Virus allowed (encouraged) to replicate in controlled environment in immune center
- ◆ This yields collection of infected files
- ◆ In addition, a collection of “clean” files is available
- ◆ Machine learning techniques used to find strings that appear in most infected files and in few clean files, e.g.:
  - search files for candidate strings
    - add points if found in infected file
    - subtract points if found in clean file



# Defenses

- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



# Clear Distinction between Data and Executable

- ◆ Virus must write to program
  - Write only allowed to data
- ◆ Must execute to spread/act
  - Data not allowed to execute
- ◆ Auditable action required to change data to executable





# Example: LOCK

- ◆ Logical Coprocessor Kernel
  - Designed to be certified at TCSEC A1 level
- ◆ Compiled programs are type “data”
  - Sequence of specific, auditable events required to change type to “executable”
- ◆ Cannot modify “executable” objects
  - So viruses can’t insert themselves into programs (no infection phase)



# Example: Duff and UNIX

- ◆ Observation: users with execute permission usually have read permission, too
  - So files with “execute” permission have type “executable”; those without it, type “data”
  - Executable files can be altered, but type immediately changed to “data”
    - Implemented by turning off execute permission
    - Certifier can change them back
      - So virus can spread only if run as certifier



# Defenses

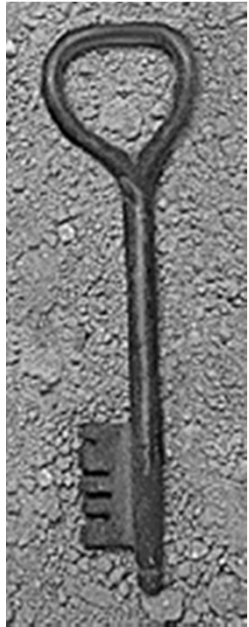
- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



# Limiting Accessibility

## ◆ Information Flow

- Malicious code usurps authority of user
- Limit information flow between users
  - If  $A$  talks to  $B$ ,  $B$  can no longer talk to  $C$
- Limits spread of virus
- Problem: Tracking information flow



# Application of principle of least privilege

- ◆ Basic idea: remove rights from process so it can only perform its function
  - Warning: if that function requires it to write, it can write anything
  - But you can make sure it writes only to those objects you expect



# Karger's Scheme

- ◆ Base it on attribute of subject, object
- ◆ Interpose a knowledge-based subsystem to determine if requested file access reasonable
  - Sits between kernel and application
- ◆ Example: UNIX C compiler
  - Reads from files with names ending in “.c”, “.h”
  - Writes to files with names beginning with “/tmp/ctm” and assembly files with names ending in “.s”
- ◆ When subsystem invoked, if C compiler tries to write to “.c” file, request rejected



# Lai and Gray

- ◆ Implemented modified version of Karger's scheme on UNIX system
  - Allow programs to access (read or write) files named on command line
  - Prevent access to other files
- ◆ Two types of processes
  - Trusted (no access checks or restrictions)
  - Untrusted (valid access list – VAL controls access)
    - VAL initialized to command line arguments plus any temporary files that the process creates



# File Access Requests

1. If file on VAL, use effective UID/GID of process to determine if access allowed
2. If access requested is read and file is world-readable, allow access
3. If process creating file, effective UID/GID controls allowing creation
  - Enter file into VAL as NNA (new non-argument); set permissions so no other process can read file
4. Ask user. If yes, effective UID/GID controls allowing access; if no, deny access





# Example

- ◆ Assembler invoked from compiler

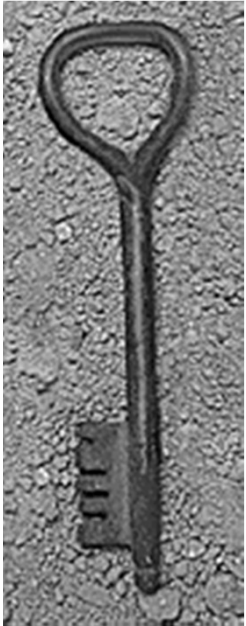
```
as x.s /tmp/ctm2345
```

and creates temp file /tmp/as1111

- VAL is

```
x.s /tmp/ctm2345 /tmp/as1111
```

- ◆ Now Trojan horse tries to copy x.s to another file
  - On creation, file inaccessible to all except creating user so attacker cannot read it (rule 3)
  - If file created already and assembler tries to write to it, user is asked (rule 4), thereby revealing Trojan horse



# Trusted Programs

- ◆ No VALs applied here
  - UNIX command interpreters
    - csh, sh
  - Program that spawn them
    - getty, login
  - Programs that access file system recursively
    - ar, chgrp, chown, diff, du, dump, find, ls, restore, tar
  - Programs that often access files not in argument list
    - binmail, cpp, dbx, mail, make, script, vi
  - Various network daemons
    - fingerd, ftpd, sendmail, talkd, telnetd, tftpd

They are ideal targets for virus!



# Guardians, Watchdogs (1)

- ◆ System intercepts request to open file
- ◆ Program invoked to determine if access is to be allowed
  - These are *guardians* or *watchdogs*
- ◆ Effectively redefines system (or library) calls



## Guardians, Watchdogs (2) Trust

- ◆ Trust the user to take explicit actions to limit their process' protection domain sufficiently
  - That is, enforce least privilege correctly
- ◆ Trust mechanisms to describe programs' expected actions sufficiently for descriptions to be applied, and to handle commands without such descriptions properly
- ◆ Trust specific programs and kernel
  - Problem: these are usually the first programs malicious logic attack



# Sandbox / Virtual Machine

- ◆ Run in protected area
- ◆ Libraries / system calls replaced with limited privilege set



# Defenses

- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



# Inhibit Sharing

- ◆ Use separation implicit in integrity policies
- ◆ Example: LOCK keeps single copy of shared procedure in memory
  - Master directory associates unique owner with each procedure, and with each user a list of other users the first trusts
  - Before executing any procedure, system checks that user executing procedure trusts procedure owner



# Use Multilevel Security Mechanisms

- ◆ Use Multi-Level Security Mechanisms
  - Place programs at lowest level
  - Don't allow users to operate at that level
  - *Prevents writes by malicious code*
- ◆ Example: DG/UX system
  - All executables in “virus protection region” below user and administrative regions





# Defenses

- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



# Detect Alteration of Files (1)

- ◆ Compute manipulation detection code (MDC) to generate signature block for each file, and save it
- ◆ Later, recompute MDC and compare to stored MDC
  - If different, file has changed
- ◆ Example: tripwire
  - Signature consists of file attributes (size, owner id, protection mode, inode no.), cryptographic checksums chosen from among MD4, MD5, HAVAL, SHS, CRC-16, CRC-32, etc.)



## Detect Alteration of Files (2)

- ◆ Assumptions: Files do not contain malicious logic when original signature block generated
- ◆ Pozzo & Grey: implement Biba's model on distributed LOCUS O.S. to make assumption explicit
  - Credibility ratings assign trustworthiness numbers from 0 (untrusted) to  $n$  (signed, fully trusted)
  - Subjects have risk levels
    - Subjects can execute programs with credibility ratings  $\geq$  risk level
    - If credibility rating  $<$  risk level, must use the special command "run-untrusted" to run program
- ◆ Cons: performance, cryptographic key management



# Antivirus Programs

- ◆ Look for specific sequences of bytes (called “virus signature” in file
  - If found, warn user and/or disinfect file
- ◆ Each agent must look for known set of viruses
- ◆ Cannot deal with viruses not yet analyzed
  - Due in part to undecidability of whether a generic program is a virus



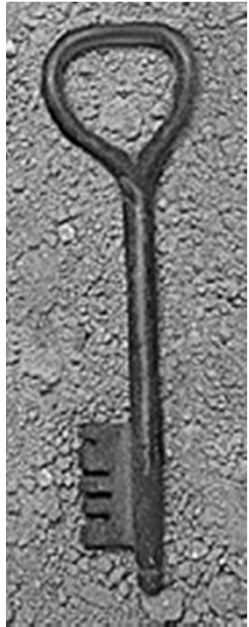
# Defenses

- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



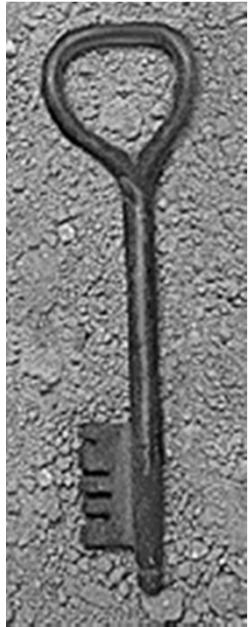
# Detect Actions Beyond Spec

- ◆ Treat execution, infection as errors and apply fault tolerant techniques
- ◆ Example: break program into sequences of nonbranching instructions
  - Checksum each sequence, encrypt result
  - When run, processor recomputes checksum, and at each branch co-processor compares computed checksum with stored one
    - If different, error occurred



# N-Version Programming

- ◆ Implement several different versions of algorithm
- ◆ Run them concurrently
  - Check intermediate results periodically
  - If disagreement, majority wins
- ◆ Assumptions
  - Majority of programs not infected
  - Underlying operating system secure
- ◆ Conflicts:
  - In order to control the propagation of the virus, all the algorithms must vote for each file accesses.
  - They have to be almost the same!



# Proof-Carrying Code

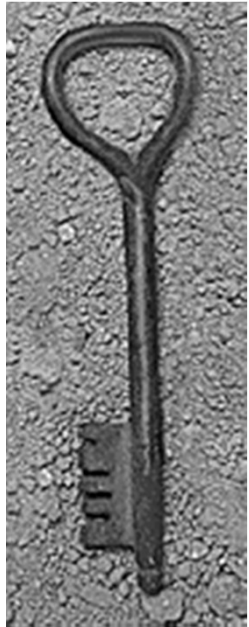
- ◆ Code consumer (user) specifies safety requirement
- ◆ Code producer (author) generates proof code meets this requirement
- ◆ Binary (code + proof) delivered to consumer
- ◆ Consumer validates proof
  - Changing the code will make the validation process failed, so that the consumer will reject the code
- ◆ Example statistics on Berkeley Packet Filter [Necula & Lee 1996]: proofs 300–900 bytes, validated in 0.3 –1.3 ms
  - Startup cost higher, runtime cost considerably shorter





# Defenses

- ◆ Distinguish between data, instructions
- ◆ Limit objects accessible to processes
- ◆ Inhibit sharing
- ◆ Detect altering of files
- ◆ Detect actions beyond specifications
- ◆ Analyze statistical characteristics



# Detecting Statistical Changes

- ◆ Based on: each programmer has his own coding style. This style can be collected by statistical data
- ◆ Example: application had 3 programmers working on it, but statistical analysis shows code from a fourth person—may be from a Trojan horse or virus!
  - Source code level: format, comments, etc
  - Binary code level: data structure, algorithm, etc
- ◆ Application Statistical
  - High/low number of files read/written
  - Unusual amount of data transferred
  - Abnormal usage of CPU time
  - Denning: use intrusion detection system to detect these
  - *Only works after the damage is done!*



# Summary – the Key Points

- ◆ A perplexing problem
  - How do you tell what the user asked for is *not* what the user intended?
- ◆ Strong typing leads to separating data, instructions
- ◆ File scanners most popular anti-virus agents
  - Must be updated as new viruses come out