# Lecture 19
# November 8, 2023

# Michigan Terminal System

- General-purpose OS running on IBM 360, 370 systems
- Class exercise: gain access to terminal control structures
  - Had approval and support of center staff
  - Began with authorized account (level 3)

# Step 1: Information Gathering

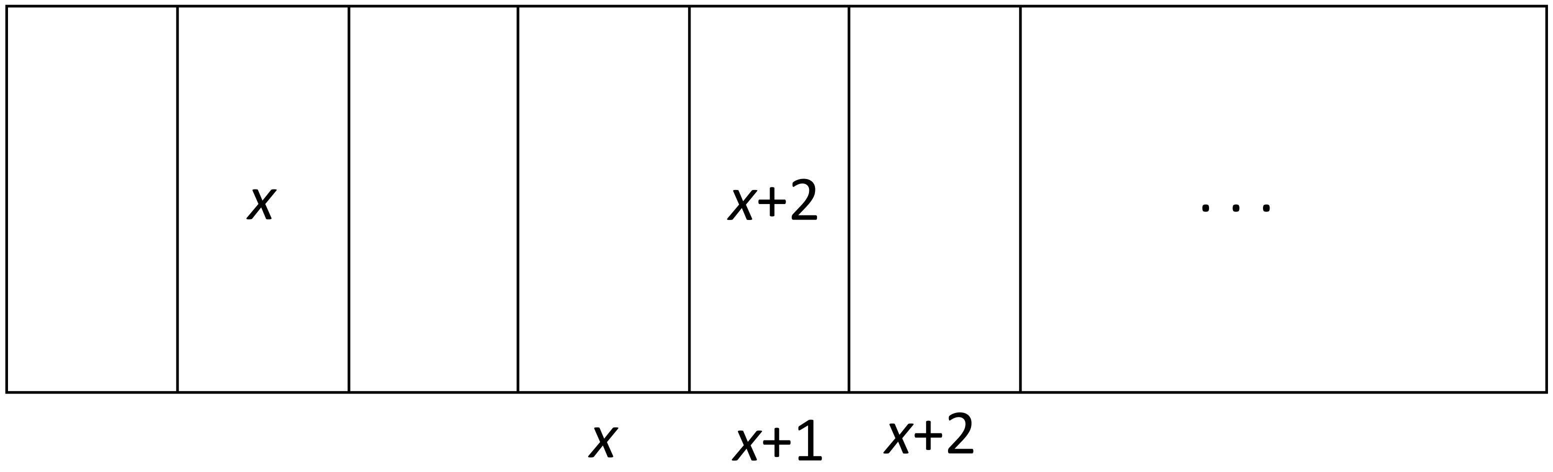
- Learn details of system's control flow and supervisor
  - When program ran, memory split into segments
  - 0-4: supervisor, system programs, system state
    - Protected by hardware mechanisms
  - 5: system work area, process-specific information including privilege level
    - Process should not be able to alter this
  - 6 on: user process information
    - Process can alter these
- Focus on segment 5

# Step 2: Information Gathering

- Segment 5 protected by virtual memory protection system
  - System mode: process can access, alter data in segment 5, and issue calls to supervisor
  - User mode: segment 5 not present in process address space (and so can't be modified)
- Run in user mode when user code being executed
- User code issues system call, which in turn issues supervisor call

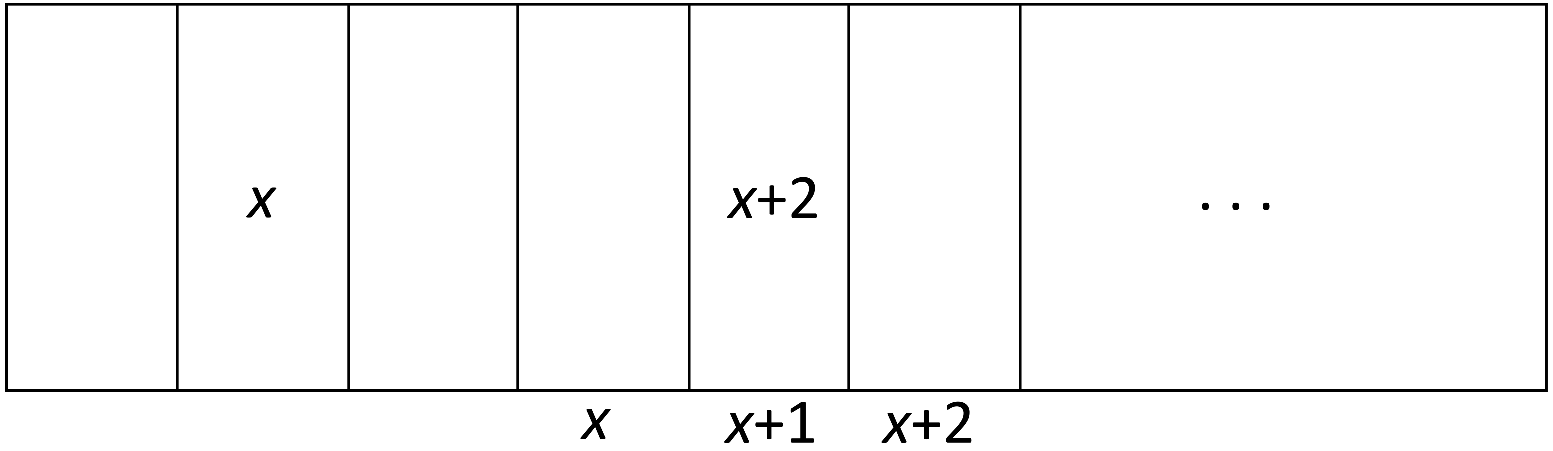
# How to Make a Supervisor Call

- System code checks parameters to ensure supervisor accesses authorized locations only
  - Parameters passed as list of addresses (*x*, *x*+1, *x*+2) constructed in user segment
  - Address of list (*x*) passed via register



# Step 3: Flaw Hypothesis

- Consider switch from user to system mode
  - System mode requires supervisor privileges
- Found: a parameter could point to another element in parameter list
  - Below: address in location $x$+1 is that of parameter at $x$+2
  - Means: system or supervisor procedure could alter parameter's address *after* checking validity of old address



# Step 4: Flaw Testing

- Find a system routine that:
  - Used this calling convention;
  - Took at least 2 parameters and altered 1
  - Could be made to change parameter to any value (such as an address in segment 5)
- Chose line input routine
  - Returns line number, length of line, line read
- Setup:
  - Set address for storing line number to be address of line length

# Step 4: Execution

- System routine validated all parameter addresses
  - All were indeed in user segment
- Supervisor read input line
  - Line length set to value to be written into segment 5
- Line number stored in parameter list
  - Line number was set to be address in segment 5
- When line read, line length written into location address of which was in parameter list
  - So it overwrote value in segment 5

# Step 5: Flaw Generalization

- Could not overwrite anything in segments 0-4
  - Protected by hardware
- Testers realized that privilege level in segment 5 controlled ability to issue supervisor calls (as opposed to system calls)
  - And one such call turned off hardware protection for segments 0-4 …
- Effect: this flaw allowed attackers to alter anything in memory, thereby completely controlling computer

# Burroughs B6700

- System architecture: based on strict file typing
    - Entities: ordinary users, privileged users, privileged programs, OS tasks
        - Ordinary users tightly restricted
        - Other 3 can access file data without restriction but constrained from compromising integrity of system
    - No assemblers; compilers output executable code
    - Data files, executable files have different types
        - Only compilers can produce executables
        - Writing to executable or its attributes changes its type to data
- Class exercise: obtain status of privileged user

# Step 1: Information Gathering

- System had tape drives
  - Writing file to tape preserved file contents
  - Header record indicates file attributes including type
- Data could be copied from one tape to another
  - If you change data, it’s still data

# Step 2: Flaw Hypothesis

- System cannot detect change to executable file if that file is altered off-line

# Step 3: Flaw Testing

- Write small program to change type of any file from data to executable
  - Compiled, but could not be used yet as it would alter file attributes, making target a data file
  - Write this to tape
- Write a small utility to copy contents of tape 1 to tape 2
  - Utility also changes header record of contents to indicate file was a compiler (and so could output executables)

# Creating the Compiler

- Run copy program
  - As header record copied, type becomes "compiler"
- Reinstall program as a new compiler
- Write new subroutine, compile it normally, and change machine code to give privileges to anyone calling it (this makes it data, of course)
  - Now use new compiler to change its type from data to executable
- Write third program to call this
  - Now you have privileges

# Corporate Computer System

- Goal: determine whether corporate security measures were effective in keeping external attackers from accessing system
- Testers focused on policies and procedures
  - Both technical and non-technical

# Step 1: Information Gathering

- Searched Internet
  - Got names of employees, officials
  - Got telephone number of local branch, and from them got copy of annual report
- Constructed much of the company’s organization from this data
  - Including list of some projects on which individuals were working

# Step 2: Get Telephone Directory

- Corporate directory would give more needed information about structure
  - Tester impersonated new employee
    - Learned two numbers needed to have something delivered off-site: employee number of person requesting shipment, and employee’s Cost Center number
  - Testers called secretary of executive they knew most about
    - One impersonated an employee, got executive’s employee number
    - Another impersonated auditor, got Cost Center number
  - Had corporate directory sent to off-site “subcontractor”

# Step 3: Flaw Hypothesis

- Controls blocking people giving passwords away not fully communicated to new employees
  - Testers impersonated secretary of senior executive
  - Called appropriate office
  - Claimed senior executive upset he had not been given names of employees hired that week
  - Got the names

# Step 4: Flaw Testing

- Testers called newly hired people
  - Claimed to be with computer center
  - Provided “Computer Security Awareness Briefing” over phone
  - During this, learned:
    - Types of computer systems used
    - Employees’ numbers, logins, and passwords
- Called computer center to get modem numbers
  - These bypassed corporate firewalls
- Success

# Step 5: Flaw Generalization

- Other human (social engineering) methods would get more information
- Problem here is human
  - Inadequate training
  - Inadequate validation of claims to be in the company
  - Not checking where information is sent
  - Not checking where information is came from

# Debate

- How valid are these tests?
  - Not a substitute for good, thorough specification, rigorous design, careful and correct implementation, meticulous testing
  - Very valuable *a posteriori* testing technique
    - Ideally unnecessary, but in practice very necessary
- Finds errors introduced due to interactions with users, environment
  - Especially errors from incorrect maintenance and operation
  - Examines system, site through eyes of attacker

# Problems

- Flaw Hypothesis Methodology depends on caliber of testers to hypothesize and generalize flaws
- Flaw Hypothesis Methodology does not provide a way to examine system systematically
  - Vulnerability classification schemes help here

# Malware

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

# Example

- Shell script on a UNIX system:

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

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

# 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 Anderson Report
- Example: previous script is Trojan horse
  - Overt purpose: list files in directory
  - Covert purpose: create setuid shell

# Example: Gemini

- Designed for Android cell phones
- Placed in several Android apps on Android markets, forums
- When app was run:
  - Gemini installed itself, using several techniques to make it hard to find
  - Then it connected to a remote command and control server, waited for commands
  - Commands it could execute included delete SMS messages; send SMS messages to remote server; dump contact list; dump list of apps

# Rootkits

- Trojan horse corrupting system to carry out covert action without detection
- Earliest ones installed back doors so attackers could enter systems, then corrupted system programs to hide entry and actions
  - Program to list directory contents altered to not include certain files
  - Network status program altered to hide connections from specific hosts

# Example: Linux Rootkit IV

- Replaced system programs that might reveal its presence
  - *ls*, *find*, *du* for file system; *ps*, *top*, *lsof*, *killall* for processes; *crontab* to hide rootkit jobs
  - *login* and others to allow attacker to log in, acquire superuser privileges (and it suppressed any logging)
  - *netstat*, *ifconfig* to hide presence of attacker
  - *tcpd*, *syslogd* to inhibit logging
- Added back doors so attackers could log in unnoticed
- Also added network sniffers to gather user names, passwords
- Similar rootkits existed for other systems

# Defenses

- Use non-standard programs to obtain the same information that standard ones should; then compare
  - *ls* lists contents of directory
  - *dirdump*, a program to read directory entries, was non-standard
    - Compare output to that if *ls*; if they differ, *ls* probably compromised
- Look for specific strings in executables
  - Programs to do this analysis usually not rigged, but easy enough to write your own
- Look for changes using cryptographically strong checksums
- These worked because they bypassed system programs, using system calls directly

# Next Step: Alter the Kernel

- Rootkits then altered system calls using kernel-loadable modules
  - Thereby eliminating the effectiveness of the earlier defenses
- Example: Knark modifies entries in system call table to involve versions in new kernel-loadable module; these hide presence of Knark
  - Defense: compare system call table in kernel with copy stored at boot time
- Example: SucKIT changes variable in kernel that points to system call table so it points to a modified table, defeating the Knark defense
- Example: adore-ng modifies virtual file system layer to hide files with rootkit's UID or GID; manipulates /proc and other pseudofiles to control what process monitoring programs report
  - Takes advantage of the ability to access OS entities like processes through file system

# Oops …

- Sony BMG developed rootkit to implement DRM on a music CDs
  - Only worked on Windows systems; users had to install a proprietary program to play the music
  - Also installed software that altered functions in Windows OS to prevent playing music using other programs
  - This software concealed itself by altering kernel not to list any files or folders beginning with “$sys$” and storing its software in such a folder
  - On boot, software contacted Sony to get advertisements to display when music was played
  - Once made public, attackers created Trojan horses with names beginning with “$sys$ (like “$sys$drv.exe”)
- Result: lawsuits, flood of bad publicity, and recall of all such CDs

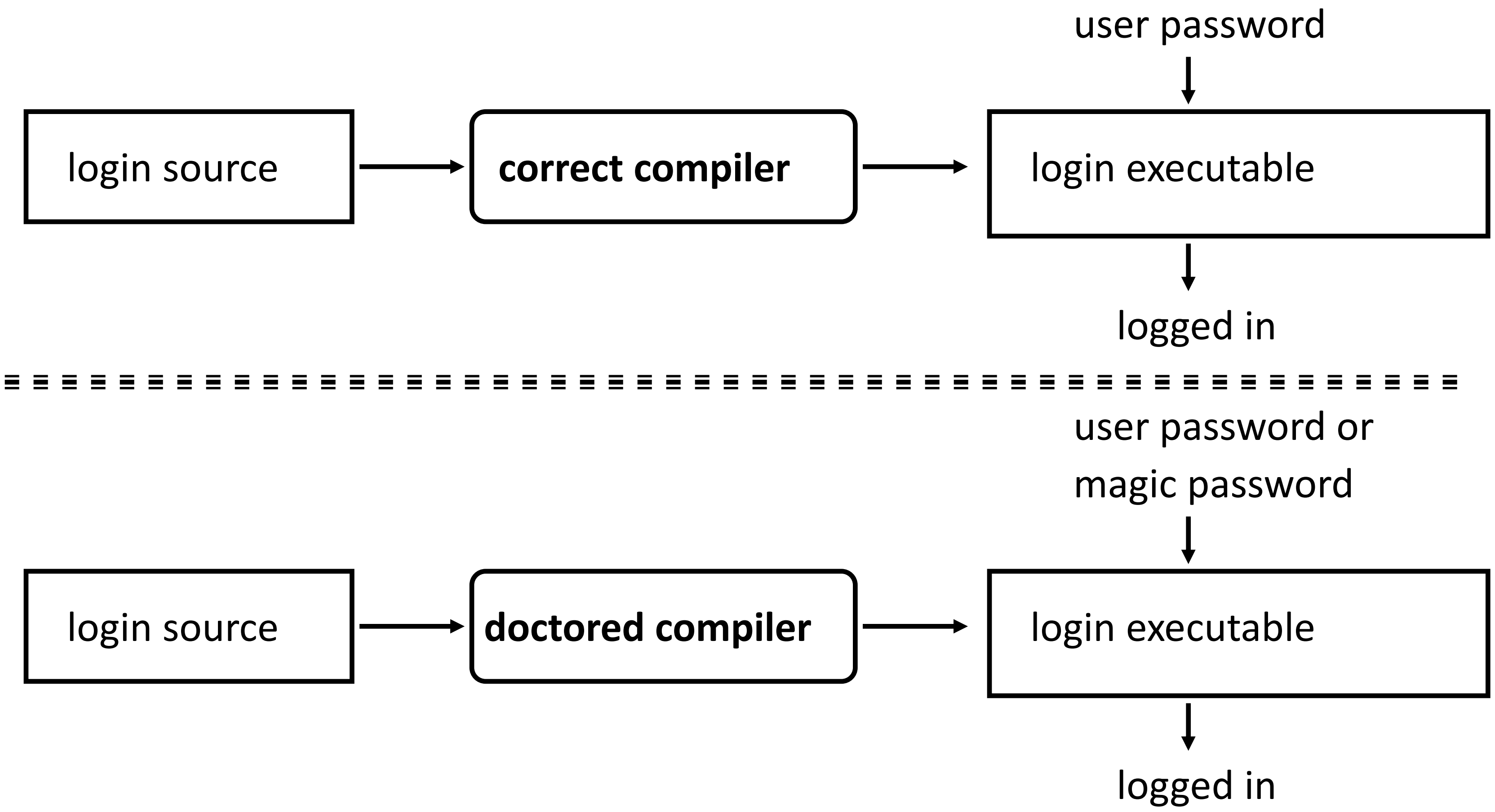
# Replicating Trojan Horse

- Trojan horse that makes copies of itself
  - Also called *propagating Trojan horse*
  - Early version of *animal* game used this to delete copies of itself
- 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

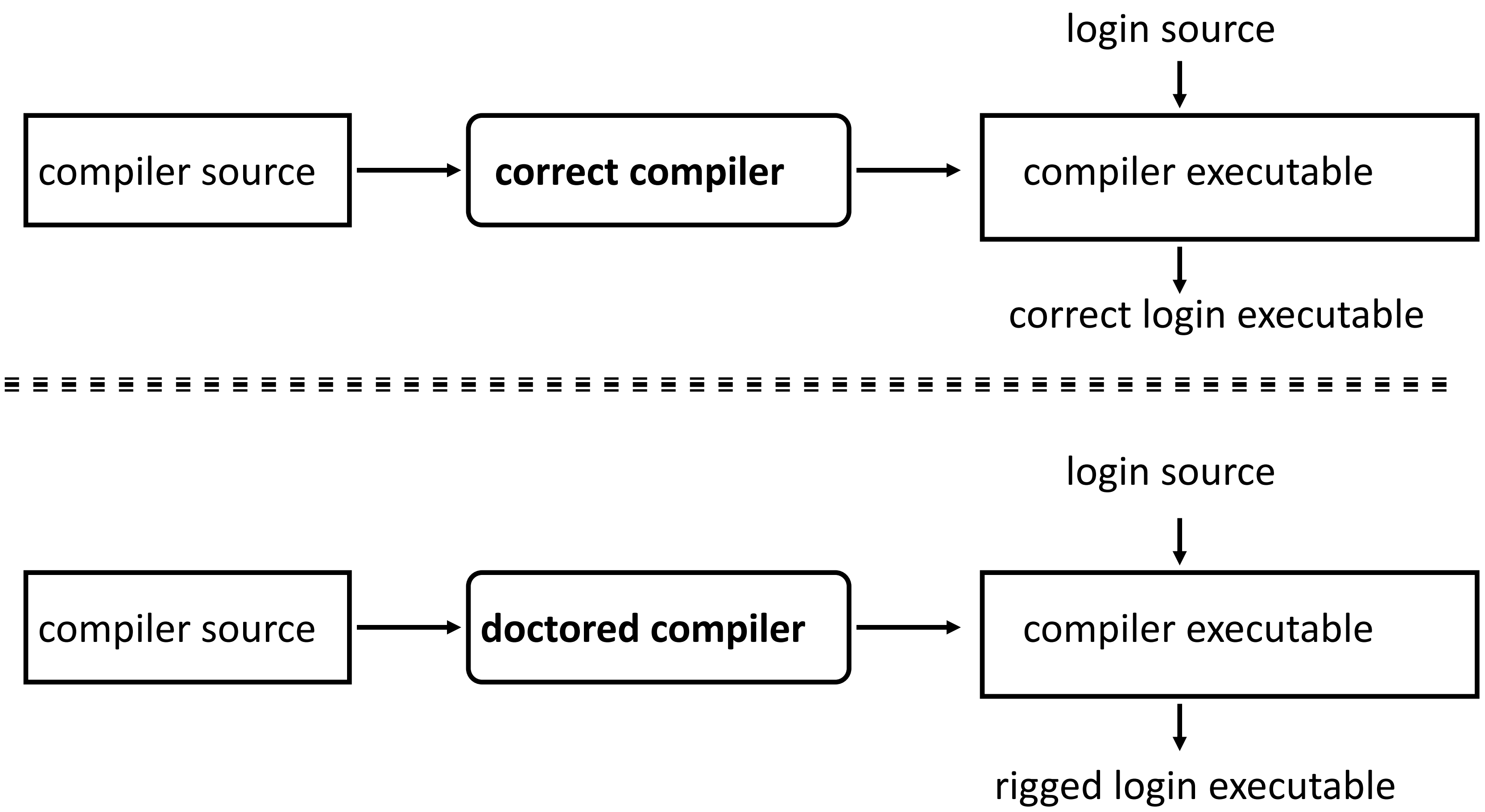
# Thompson's Compiler

- 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

# 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:
```

# Trojan Horse Or Not?

- Yes
  - Overt action = infected program's actions
  - Covert action = virus' actions (infect, execute)
- No
  - Overt purpose = virus' actions (infect, execute)
  - Covert purpose = none
- Semantic, philosophical differences
  - Defenses against Trojan horse also inhibit computer viruses