

Chapter 19: Malicious Logic

- What is malicious logic
- Types of malicious logic
- Defenses

Overview

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

Malicious Logic

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

Example

- Shell script on a UNIX system:

```
cp /bin/sh /tmp/.xyzy
```

```
chmod u+s,o+x /tmp/.xyzy
```

```
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: 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

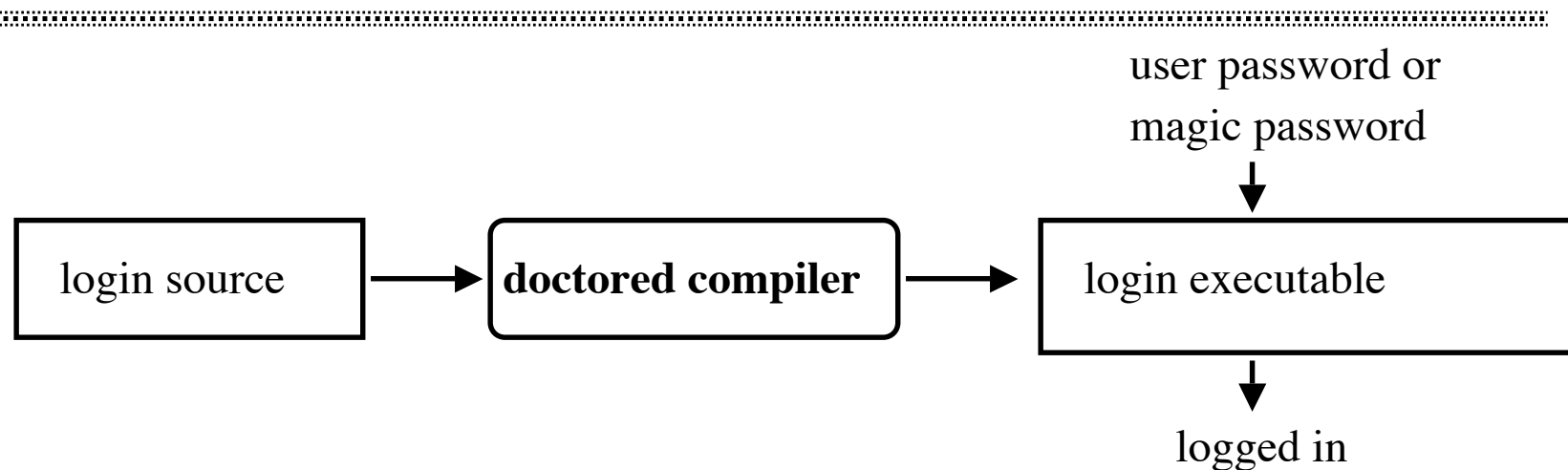
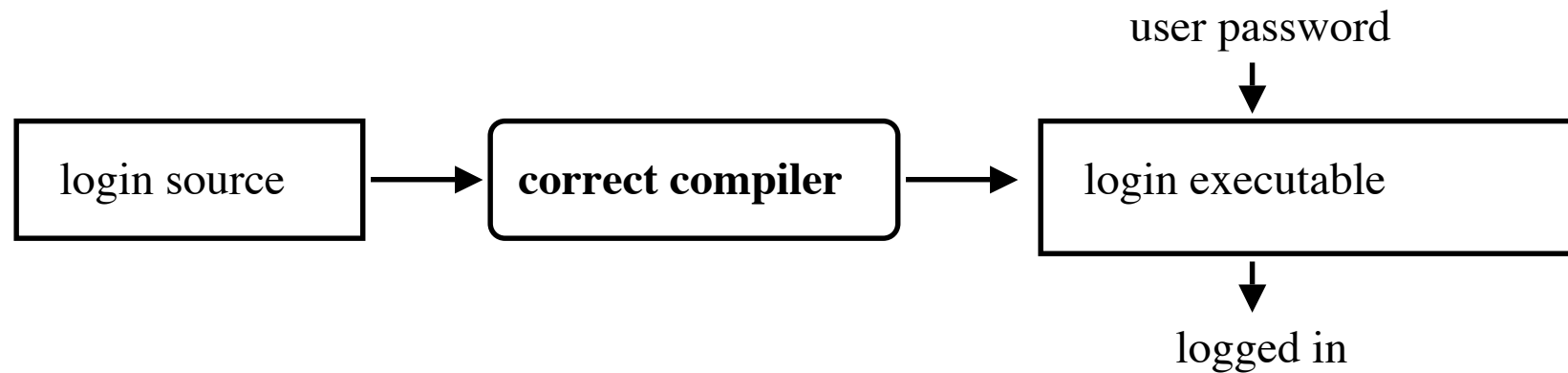
Replicating Trojan Horse

- Trojan horse that makes copies of itself
 - Also called *propagating Trojan horse*
 - 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

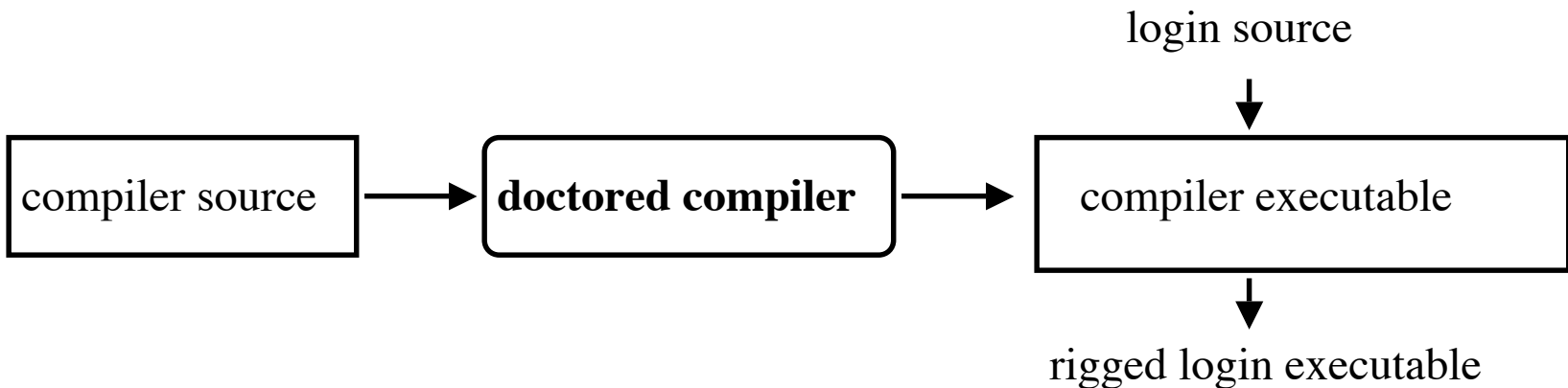
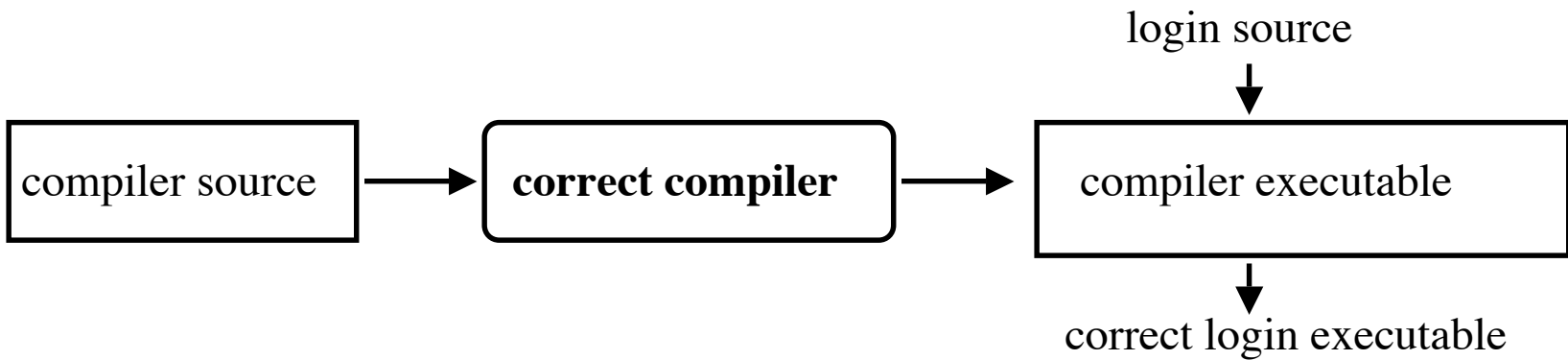
Thompson's Compiler

- 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

History

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

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: 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)
 - Small virus placed on UNIX system, spread to 46 systems in 8 days
 - Wrote a Bourne shell script virus
- 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

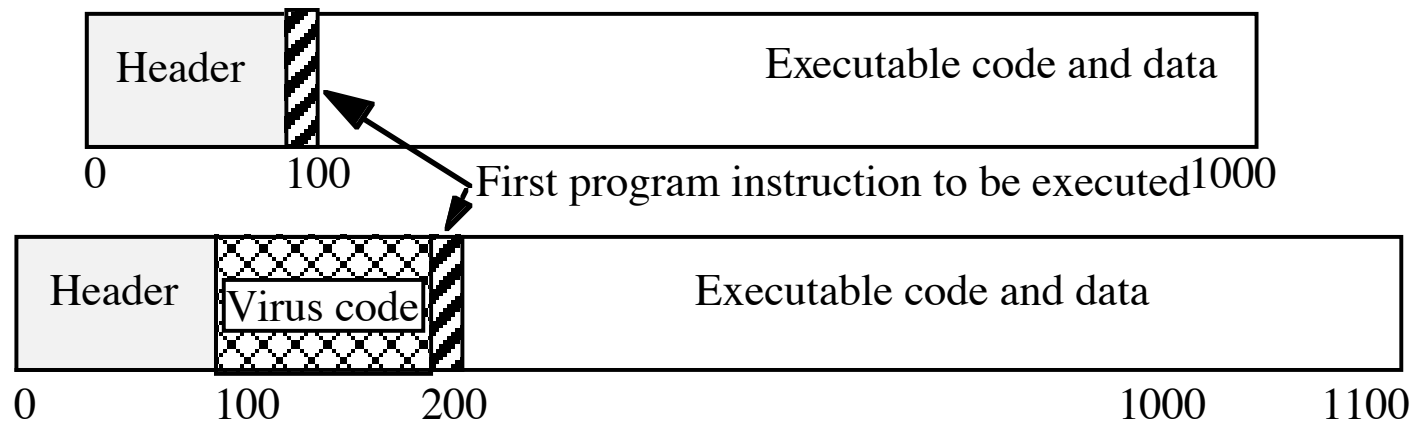
Types of Viruses

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

Boot Sector Infectors

- 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

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
 - Error: doesn't set signature when .EXE executes
 - So .EXE files continually reinfected

Multipartite Viruses

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

TSR Viruses

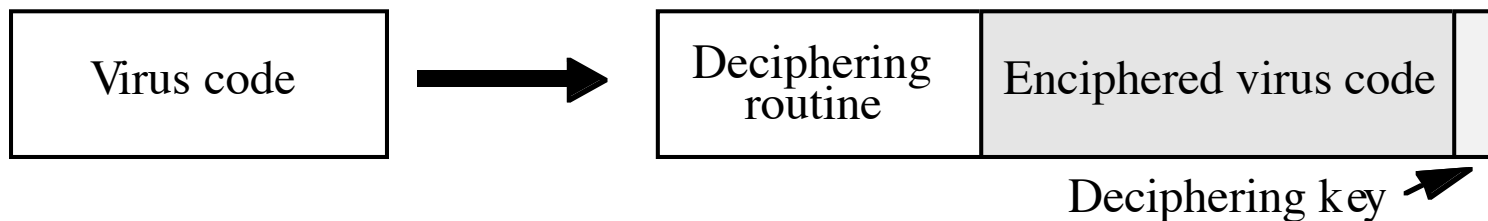
- A virus that stays active in memory after the application (or bootstrapping, or disk mounting) is completed
 - TSR is “Terminate and Stay Resident”
- Examples: Brain, Jerusalem viruses
 - Stay in memory after program or disk mount is completed

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



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
```

Polymorphic Viruses

- A virus that changes its form each time it inserts itself into another program
- Idea is to prevent signature detection by changing the “signature” or instructions used for deciphering routine
- At instruction level: substitute instructions
- At algorithm level: different algorithms to achieve the same purpose
- Toolkits to make these exist (Mutation Engine, Trident Polymorphic Engine)

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

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

Computer Worms

- A program that copies itself from one computer to another
- Origins: distributed computations
 - Schoch and Hupp: 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: Internet Worm of 1988

- Targeted Berkeley, Sun UNIX systems
 - Used virus-like attack to inject instructions into running program and run them
 - To recover, had to disconnect system from Internet and reboot
 - To prevent re-infection, several critical programs had to be patched, recompiled, and reinstalled
- Analysts had to disassemble it to uncover function
- Disabled several thousand systems in 6 or so hours

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

Rabbits, Bacteria

- A program that absorbs all of some class of resources
- Example: for UNIX system, shell commands:

```
while true
do
    mkdir x
    chdir x
done
```
- Exhausts either disk space or file allocation table (inode) space

Logic Bombs

- A program that performs an action that violates the site security policy when some external event occurs
- 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

Defenses

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

Data vs. Instructions

- Malicious logic is both
 - Virus: written to program (data); then executes (instructions)
- Approach: treat “data” and “instructions” as separate types, and require certifying authority to approve conversion
 - Keys are assumption that certifying authority will *not* make mistakes and assumption that tools, supporting infrastructure used in certifying process are not corrupt

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

Limiting Accessibility

- Basis: a user (unknowingly) executes malicious logic, which then executes with all that user's privileges
 - Limiting accessibility of objects should limit spread of malicious logic and effects of its actions
- Approach draws on mechanisms for confinement

Information Flow Metrics

- Idea: limit distance a virus can spread
- Flow distance metric $fd(x)$:
 - Initially, all info x has $fd(x) = 0$
 - Whenever info y is shared, $fd(y)$ increases by 1
 - Whenever y_1, \dots, y_n used as input to compute z , $fd(z) = \max(fd(y_1), \dots, fd(y_n))$
- Information x accessible if and only if for some parameter V , $fd(x) < V$

Example

- Anne: $V_A = 3$; Bill, Cathy: $V_B = V_C = 2$
- Anne creates program P containing virus
- Bill executes P
 - P tries to write to Bill's program Q
 - Works, as $fd(P) = 0$, so $fd(Q) = 1 < V_B$
- Cathy executes Q
 - Q tries to write to Cathy's program R
 - Fails, as $fd(Q) = 1$, so $fd(R)$ would be 2
- Problem: if Cathy executes P, R can be infected
 - So, does not stop spread; slows it down greatly, though

Implementation Issues

- Metric associated with *information*, not *objects*
 - You can tag files with metric, but how do you tag the information in them?
 - This inhibits sharing
- To stop spread, make $V = 0$
 - Disallows sharing
 - Also defeats purpose of multi-user systems, and is crippling in scientific and developmental environments
 - Sharing is critical here

Reducing Protection Domain

- 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

Example: ACLs and C-Lists

- s_1 owns file f_1 and s_2 owns program p_2 and file f_3
 - Suppose s_1 can read, write f_1 , execute p_2 , write f_3
 - Suppose s_2 can read, write, execute p_2 and read f_3
- s_1 needs to run p_2
 - p_2 contains Trojan horse
 - So s_1 needs to ensure p_{12} (subject created when s_1 runs p_2) can't write to f_3
 - Ideally, p_{12} has capability $\{ (s_1, p_2, x) \}$ so no problem
 - In practice, p_{12} inherits s_1 's rights—bad! Note s_1 does not own f_3 , so can't change its rights over f_3
- Solution: restrict access by others

Authorization Denial Subset

- Defined for each user s_i
- Contains ACL entries that others cannot exercise over objects s_i owns
- In example: $R(s_2) = \{ (s_1, f_3, w) \}$
 - So when p_{12} tries to write to f_3 , as p_{12} owned by s_1 and f_3 owned by s_2 , system denies access
- Problem: how do you decide what should be in your authorization denial subset?

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 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

Guardians, Watchdogs

- 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

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

Sandboxing

- Sandboxes, virtual machines also restrict rights
 - Modify program by inserting instructions to cause traps when violation of policy
 - Replace dynamic load libraries with instrumented routines

Example: Race Conditions

- Occur when successive system calls operate on object
 - Both calls identify object by name
 - Rebind name to different object between calls
- Sandbox: instrument calls:
 - Unique identifier (inode) saved on first call
 - On second call, inode of named file compared to that of first call
 - If they differ, potential attack underway ...

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

Multilevel Policies

- Put programs at the lowest security level, all subjects at higher levels
 - By *-property, nothing can write to those programs
 - By ss-property, anything can read (and execute) those programs
- Example: DG/UX system
 - All executables in “virus protection region” below user and administrative regions

Detect Alteration of Files

- 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, cryptographic checksums chosen from among MD4, MD5, HAVAL, SHS, CRC-16, CRC-32, etc.)

Assumptions

- Files do not contain malicious logic when original signature block generated
- Pozzo & Grey: implement Biba's model on LOCUS 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 special command to run program

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

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
 - Different algorithms with enough equal intermediate results may be infeasible
 - Especially for malicious logic, where you would check file accesses

Proof-Carrying Code

- Code consumer (user) specifies safety requirement
- Code producer (author) generates proof code meets this requirement
 - Proof integrated with executable code
 - Changing the code invalidates proof
- Binary (code + proof) delivered to consumer
- Consumer validates proof
- Example statistics on Berkeley Packet Filter:
proofs 300–900 bytes, validated in 0.3 –1.3 ms
 - Startup cost higher, runtime cost considerably shorter

Detecting Statistical Changes

- 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!
- Other attributes: more conditionals than in original; look for identical sequences of bytes not common to any library routine; increases in file size, frequency of writing to executables, etc.
 - Denning: use intrusion detection system to detect these

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