Chapter 26: Program Security

- Introduction
- Requirements and Policy
- Design
- Refinement and Implementation
- Common Security-Related Programming Problems
- Testing, Maintenance, and Operation
- Distribution

Introduction

- Goal: implement program that:
  - Verifies user's identity
  - Determines if change of account allowed
  - If so, places user in desired role
- Similar to *su*(1) for UNIX and Linux systems
  - User supplies his/her password, not target account's

Why?

- Eliminate password sharing problem
  - Role accounts under Linux are user accounts
  - If two or more people need access, *both* need role account's password
- Program solves this problem
  - Runs with *root* privileges
  - User supplies his/her password to authenticate
  - If access allowed, program spawns command interpreter with privileges of role account

Requirements

1. Access to role account based on user, location, time of request
2. Settings of role account's environment replaces corresponding settings of user's environment, but rest of user's environment preserved
3. Only *root* can alter access control information for access to role account

More Requirements

4. Mechanism provides restricted, unrestricted access to role account
    - Restricted: run only specified commands
    - Unrestricted: access command interpreter
5. Access to files, directories, objects owned by role account restricted to those authorized to use role account, users trusted to install system programs, *root*

Threats

- Group 1: Unauthorized user (UU) accessing role accounts
    1. UU accesses role account as though authorized user
    2. Authorized user uses nonsecure channel to obtain access to role account, thereby revealing authentication information to UU
    3. UU alters access control information to gain access to role account
    4. Authorized user executes Trojan horse giving UU access to role account

Relationships

threat	requirement	notes
1	1, 5	Restricts who can access role account, protects access control data
2	1	Restricts location from where user can access role account
3	3	Restricts change to trusted users
4	2, 4, 5	User's search path restricted to own or role account; only trusted users, role account can manipulate executables

More Threats

- Group 2: Authorized user (AU) accessing role accounts

5. AU obtains access to role account, performs unauthorized commands
6. AU executes command that performs functions that user not authorized to perform
7. AU changes restrictions on user's ability to obtain access to role account

Relationships

threat	requirement	notes
5	4	Allows user restricted access to role account, so user can run only specific commands
6	2, 5	Prevent introduction of Trojan horse
7	3	*root* users trusted; users with access to role account trusted

Design

- Framework for hooking modules together
  - User interface
  - High-level design
- Controlling access to roles and commands
  - Interface
  - Internals
  - Storage of access control data

User Interface

- User wants unrestricted access *or* to run a specific command (restricted access)
- Assume command line interface
  - Can add GUI, etc. as needed
- Command

```
role role_account [ command ]
```

where

  - *role_account* name of role account
  - *command* command to be run (optional)

High-Level Design

1. Obtain role account, command, user, location, time of day
    - If command omitted, assume command interpreter (unrestricted access)
2. Check user allowed to access role account
    a) at specified location;
    b) at specified time; and
    c) for specified command (or without restriction)

    If user not, log attempt and quit

High-Level Design (*con't*)

3. Obtain user, group information for role account; change privileges of process to role account
4. If user requested specific command, overlay process with command interpreter that spawns named command
5. If user requested unrestricted access, overlay process with command interpreter allowing interactive use

Ambiguity in Requirements

- Requirements 1, 4 do not say whether command selection restricted by time, location
  - This design assumes it is
    - Backups may need to be run at 1AM and only 1AM
    - Alternate: assume restricted only by user, role; equally reasonable
  - Update requirement 4 to be: Mechanism provides restricted, unrestricted access to role account
    - Restricted: run only specified commands
    - Unrestricted: access command interpreter

  Level of access (restricted, unrestricted) depends on user, role, time, location

Access to Roles, Commands

- Module determines whether access to be allowed
  - If it can’t get user, role, location, and/or time, error; return failure
- Interface: controls how info passed between module, caller
- Internal structure: how does module handle errors, access control data structures

Interface to Module

- Minimize amount of information being passed through interface
  - Follow standard ideas of information hiding
  - Module can get user, time of day, location from system
  - So, need pass only command (if any), role account name
- `boolean accessok(role rname, command cmd)`
  - *rname*: name of role
  - *cmd*: command (empty if unrestricted access desired)
  - returns *true* if access granted, *false* if not (or error)

Internals of Module

- Part 1: gather data to determine if access allowed
- Part 2: retrieve access control information from storage
- Part 3: compare two, determine if access allowed

Part 1

- Required:
  - user ID: who is trying to access role account
  - time of day: when is access being attempted
    - From system call to operating system
  - entry point: terminal or network connection
  - remote host: name of host from which user accessing local system (empty if on local system)
    - These make up location

Part 2

- Obtain handle for access control file
  - May be called a "descriptor"
- Contents of file is sequence of records:

```
role account
user names
locations from which the role account can be accessed
times when the role account can be accessed
command and arguments
```

- Can list multiple commands, arguments in 1 record
  - If no commands listed, unrestricted access

Part 3

- Iterate through access control file
  - If no more records
    - Release handle
    - Return failure
  - Check role
    - If not a match, skip record (go back to top)
  - Check user name, location, time, command
    - If *any* does not match, skip record (go back to top)
  - Release handle
  - Return success

Storing Access Control Data

- Sequence of records; what should contents of fields be?
  - Location: *any*, *local*, *host*, *domain*; operators not, or (‘,’)

    ```
    *local* , control.fixit.com , .watchu.edu
    ```

  - User: *any*, *user name*; operators not, or (‘,’)

    ```
    peter , paul , mary , joan , janis
    ```

  - Time: *any*, *time range*

Time Representation

- Use ranges expressed (reasonably) normally

  `Mon-Thu 9AM-5PM`

  – Any time between 9AM and 5PM on Mon, Tue, Wed, or Thu

  `Mon 9AM-Thu 5PM`

  – Any time between 9AM Monday and 5PM Thursday

  `Apr 15 8AM-Sep 15 6PM`

  – Any time from 8AM on April 15 to 6PM on September 15, on any year

Commands

- Command plus arguments shown

  `/bin/install *`

  – Execute /bin/install with any arguments

  `/bin/cp` log /var/inst/log

  – Copy file log to /var/inst/log

  `/usr/bin/id`

  – Run program id with no arguments
- User need not supply path names, but commands used *must* be the ones with those path names

Refinement and Implementation

- First-level refinement
- Second-level refinement
- Functions
  - Obtaining location
  - Obtaining access control record
  - Error handling in reading, matching routines

First-Level Refinement

- Use pseudocode:

```
boolean accessok(role rname, command cmd);
   stat ← false
   user ← obtain user ID
   timeday ← obtain time of day
   entry ← obtain entry point (terminal line, remote host)
   open access control file
   repeat
     rec ← get next record from file; EOF if none
     if rec ≠ EOF then
               stat ← match(rec, rname, cmd, user, timeday, entry)
   until rec = EOF or stat = true
   close access control file
return stat
```

Check Sketch

- Interface right
- Stat (holds status of access control check) false until match made, then true
- Get user, time of day, location (entry)
- Iterates through access control records
  - Get next record
  - If there was one, sets stat to result of match
  - Drops out when stat true or no more records
- Close file, releasing handle
- Return stat

Second-Level Refinement

- Map pseudocode to particular language, system
  - We’ll use C, Linux (UNIX-like system)
  - Role accounts same as user accounts
- Interface decisions
  - User, role ID representation
  - Commands and arguments
  - Result

Users and Roles

- May be name (string) or uid_t (integer)
  - In access control file, either representation okay
- If bogus name, can't be mapped to uid_t
- Kernel works with uid_t
  - So access control part needs to do conversion to uid_t at some point
- Decision: represent all user, role IDs as uid_t
- Note: no design decision relied upon representation of user, role accounts, so no need to revisit any

Commands, Arguments, Result

- Command is program name (string)
- Argument is sequence of words (array of string pointers)
- Result is boolean (integer)

Resulting Interface

```
int accessok(uid_t rname, char *cmd[]);
```

Second-Level Refinement

- Obtaining user ID
- Obtaining time of day
- Obtaining location
- Opening access control file
- Processing records
- Cleaning up

Obtaining User ID

- Which identity?
  - Effective ID: identifies privileges of process
    - Must be 0 (*root*), so not this one
  - Real ID: identifies user running process

```
userid = getuid();
```

Obtain Time of Day

- Internal representation is seconds since epoch
  - On Linux, epoch is Jan 1, 1970 00:00:00

```
timeday = time(NULL);
```

Obtaining Location

- System dependent
  - So we defer, encapsulating it in a function to be written later

```
entry = getlocation();
```

Opening Access Control File

- Note error checking and logging

```
if ((fp = fopen(acfile, “r”)) == NULL){
    logerror(errno, acfile);
    return(stat);
}
```

Processing Records

- Internal record format not yet decided
  - Note use of functions to delay deciding this

```
do {
    acrec = getnextacrec(fp);
    if (acrec != NULL)
        stat = match(rec, rname, cmd, user,
                                 timeday, entry);
} until (acrec == NULL || stat == 1);
```

Cleaning Up

- Release handle by closing file

```
(void) fclose(fp);
return(stat);
```

Getting Location

- On login, Linux writes user name, terminal name, time, and name of remote host (if any) in file *utmp*
- Every process may have associated terminal
- To get location information:
  - Obtain associated process terminal name
  - Open *utmp* file
  - Find record for that terminal
  - Get associated remote host from that record

Security Problems

- If any untrusted process can alter *utmp* file, contents cannot be trusted
  - Several security holes came from this
- Process may have no associated terminal
- Design decision: if either is true, return meaningless location
  - Unless location in access control file is *any* wildcard, fails

getlocation() Outline

```
hostname getlocation()
   myterm ← name of terminal associated with process
   obtain utmp file access control list
   if any user other than root can alter it then
        return "*nowhere*"
   open utmp file
   repeat
        term ← get next record from utmp file; EOF if none
        if term ≠ EOF and myterm = term then stat ← true
        else stat ← false
   until term = EOF or stat = true
   if host field in utmp record = empty then
                                     host ← "localhost"
   else host ← host field of utmp record
   close utmp file
return host
```

Access Control Record

- Consider match routine
  - User name is uid_t (integer) internally
    - Easiest: require user name to be uid_t in file
    - Problems: (1) human-unfriendly; (2) unless binary data recorded, still need to convert
    - Decision: in file, user names are strings (names or string of digits representing integer)
  - Location, set of commands strings internally
    - Decision: in file, represent them as strings

Time Representation

- Here, time is an interval
  - May 30 means “any time on May 30”, or “May 30 12AM-May 31 12AM
- Current time is integer internally
  - Easiest: require time interval to be two integers
  - Problems: (1) human-unfriendly; (2) unless binary data recorded, still need to convert
  - Decision: in file, time interval represented as string

Record Format

- Here, *commands* is repeated once per command, and *numcommands* is number of *commands* fields

```
record
  role rname
  string userlist
  string location
  string timeofday
  string commands[]
  …
  string commands[]
  integer numcommands
end record;
```

- May be able to compute *numcommands* from record

Error Handling

- Suppose syntax error or garbled record
- Error cannot be ignored
  - Log it so system administrator can see it
    - Include access control file name, line or record number
  - Notify user, or tell user why there is an error, different question
    - Can just say "access denied"
    - If error message, need to give access control file name, line number
  - Suggests error, log routines part of *accessok* module

Implementation

- Concern: many common security-related programming problems
  - Present management and programming rules
  - Use framework for describing problems
    - NRL: our interest is technical modeling, not reason for or time of introduction
    - Aslam: want to look at multiple components of vulnerabilities
    - Use PA or RISOS; we choose PA

Improper Choice of Initial Protection Domain

- Arise from incorrect setting of permissions or privileges
  - Process privileges
  - Access control file permissions
  - Memory protection
  - Trust in system

Process Privileges

- Least privilege: no process has more privileges than needed, but each process has the privileges it needs
- Implementation Rule 1:
  - **Structure the process so that all sections requiring extra privileges are modules. The modules should be as small as possible and should perform only those tasks that require those privileges.**

Basis

- Reference monitor
  - Verifiable: here, modules are small and simple
  - Complete: here, access to privileged resource only possible through privileges, which require program to call module
  - Tamperproof: separate modules with well-defined interfaces minimizes chances of other parts of program corrupting those modules
- ***Note***: this program, and these modules, are ***not*** reference monitors!
  - We're approximating reference monitors …

More Process Privileges

- Insufficient privilege: denial of service
- Excessive privilege: attacker could exploit vulnerabilities in program
- Management Rule 1:
  - **Check that the process privileges are set properly.**

Implementation Issues

- Can we have privileged modules in our environment?
  - No; this is a function of the OS
  - Cannot acquire privileges after start, unless process started with those privileges
- Which role account?
  - Non-*root*: requires separate program for each role account
  - *Root*: one program can handle all role accounts

Program and Privilege

- Program starts with *root* privileges
- Access control module called
  - Needs these privileges to read access control file
- Privileges released
  - But they can be reacquired …
- Privileges reacquired for switch to role account
  - Because *root* can switch to any user
- Key points: privileges acquired *only* when needed, and relinquished once *immediate* task is complete

Access Control File Permissions

- Integrity of process relies upon integrity of access control file
- Management Rule 2:
  - **The program that is executed to create the process, and all associated control files, must be protected from unauthorized use and modification. Any such modification must be detected.**

Program and File

- Program checks integrity of access control file whenever it runs
- Check dependencies, too
  - If access control file depends on other external information (like environment variables, included files, etc.), check them
  - Document these so maintainers will know what they are

Permissions

- Set these so only *root* can alter, move program, access control file
- Implementation Rule 2:
  - **Ensure that any assumptions in the program are validated. If this is not possible, document them for the installers and maintainers, so they know the assumptions that attackers will try to invalidate.**

UNIX Implementation

- Checking permissions: 3 steps
  - – Check *root* owns file
  - – Check no group write permission, or that *root* is single member of the group owner of file
    - Check list of members of that group first
    - Check password file next, to ensure no other users have primary GID the same as the group; these users need not be listed in group file to be group members
  - – Check no world read, write permission

Memory Protection

- Shared memory: if two processes have access, one can change data other relies upon, or read data other considers secret
- Implementation Rule 3
  - **Ensure that the program does not share objects in memory with any other program, and that other programs cannot access the memory of a privileged process.**

Memory Management

- Don’t let data be executed, or constants change
  - Declare constants in program as `const`
  - Turn off execute permission for data pages/segments
  - Do not use dynamic loading
- Management Rule 3:
  - **Configure memory to enforce the principle of least privilege. If a section of memory is not to contain executable instructions, turn execute permission off for that section of memory. If the contents of a section of memory are not to be altered, make that section read-only.**

Trust

- What does program trust?
  - System authentication mechanisms to authenticate users
  - UINFO to map users, roles into UIDs
  - Inability of unprivileged users to alter system clock
- Management Rule 4:
  - **Identify all system components on which the program depends. Check for errors whenever possible, and identify those components for which error checking will not work.**

Improper Isolation of Implementation Detail

- Look for errors, failures of mapping from abstraction to implementation
  – Usually come out in error messages
- Implementation Rule 4:
  – **The error status of every function must be checked. Do not try to recover unless the cause of the error, and its effects, do not affect any security considerations. The program should restore the state of the system to the state before the process began, and then terminate.**

Resource Exhaustion, User Identifiers

- Role, user are abstractions
  - The system works with UIDs
- How is mapping done?
  - Via user information database
- What happens if mapping can't be made?
  - In one mail server, returned a default user—so by arranging that the mapping failed, anyone could have mail appended to any file to which default user could write
  - Better: have program fail

Validating Access Control Entries

- Access control file data implements constraints on access
  - Therefore, it's a mapping of abstraction to implementation
- Develop second program using same modules as first
  - Prints information in easy-to-read format
  - Must be used after each change to file, to verify change does what was desired
  - Periodic checks too

Restricting Protection Domain

- Use overlays rather than spawning child
  - Overlays replace original protection domain with that of overlaid program
  - Programmers close all open files, reset signal handlers, changing privileges to that of role
  - Potential problem: saved UID, GID
    - When privileges dropped in usual way, can regain them because original UID is saved; this is how privileges restored
    - Use *setuid* system call to block this; it changes saved UID too

Improper Change

- Data that changes unexpectedly or erroneously
- Memory
- File contents
- File/object bindings

Memory

- Synchronize interactions with other processes
- Implementation Rule 5:
  - **If a process interacts with other processes, the interactions should be synchronized. In particular, all possible sequences of interactions must be known and, for all such interactions, the process must enforce the required security policy.**

More Memory

- Asynchronous exception handlers: may alter variables, state
  - Much like concurrent process
- Implementation Rule 6:
  - **Asynchronous exception handlers should not alter any variables except those that are local to the exception handling module. An exception handler should block all other exceptions when begun, and should not release the block until the handler completes execution, unless the handler has been designed to handle exceptions within itself (or calls an uninvoked exception handler).**

Buffer Overflows

- Overflow *not* the problem
- Changes to variables, state caused by overflow is the problem
  - Example: *fingerd* example: overflow changes return address to return into stack
    - Fix at compiler level: put random number between buffer, return address; check before return address used
  - Example: *login* program that stored unhashed, hashed password in adjacent arrays
    - Enter any 8-char password, hit space 72 times, enter hash of that password, and system authenticates you!

Problem

- Trusted data can be affected by untrusted data
  - Trusted data: return address, hash loaded from password file
  - Untrusted data: anything user reads
- Implementation Rule 7:
  - **Whenever possible, data that the process trusts and data that it receives from untrusted sources (such as input) should be kept in separate areas of memory. If data from a trusted source is overwritten with data from an untrusted source, a memory error will occur.**

Our Program

- No interaction except through exception handling
  - Implementation Rule 5 does not apply
- Exception handling: disable further exception handling, log exception, terminate program
  - Meets Implementation Rule 6
- Do not reuse variables used for data input; ensure no buffers overlap; check all array, pointer references; any out-of-bounds reference invokes exception handler
  - Meets Implementation Rule 7

File Contents

- If access control file changes, either:
  - File permissions set wrong (Management Rule 2)
  - Multiple processes sharing file (Implementation Rule 5)
- Dynamic loading: routines *not* part of executable, but loaded from libraries when program needs them
  - Note: these may not be the original routines …
- Implementation Rule 8:
  - **Do not use components that may change between the time the program is created and the time it is run.**

Race Conditions

- Time-of-check-to-time-of-use (TOCTTOU) problem
  - Issue: don't want file to change after validation but before access
  - UNIX file locking advisory, so can't depend on it
- How we deal with this:
  - Open file, obtaining file descriptor
  - Obtain status information *using file descriptor*
  - Validate file access
    - UNIX semantics assure this is same as for open file object; no changing possible

Improper Naming

- Ambiguity in identifying object
- Names interpreted in context
  - Unique objects cannot share names within available context
  - Interchangeable objects can, provided they are truly interchangeable
- Management Rule 5:
  - **Unique objects require unique names. Interchangeable objects may share a name.**

Contexts

- Program must control context of interpretation of name
  - Otherwise, the name may not refer to the expected object
- Example: *loadmodule* problem
  - Dynamically searched for, loaded library modules
  - Executed program *ld.so* with superuser privileges to do this
  - Default context: use “/bin/ld.so” (system one)
  - Could change context to use “/usr/anyone/ld.so” (one with a Trojan horse)

Example

- Context includes:
  - Character set composing name
  - Process, file hierarchies
  - Network domains
  - Customizations such as search path
  - Anything else affecting interpretation of name
- Implementation Rule 9:
  - **The process must ensure that the context in which an object is named identifies the correct object.**

Sanitize of Not?

- Replace context with known, safe one on start-up
  - Program controls interpretation of names now
- File names (access control file, command interpreter program)
  - Use absolute path names; do not create any environment variables affecting interpretation
- User, role names
  - Assume system properly maintained, so no problems
- Host names
  - No domain part means local domain

Improper Deallocation, Deletion

- Sensitive information can be exposed if object containing it is reallocated
  - Erase data, then deallocate
- Implementation Rule 10:
  - **When the process finishes using a sensitive object (one that contains confidential information or one that should not be altered), the object should be erased, then deallocated or deleted. Any resources not needed should also be released.**

Our Program

- Cleartext password for user
  - Once hashed, overwritten with random bytes
- Access control information
  - Close file descriptor before command interpreter overlaid
    - Because file descriptors can be inherited, and data from corresponding files read
- Log file
  - Close log file before command interpreter overlaid
    - Same reasoning, but for writing

Improper Validation

- Something not checked for consistency or correctness
  - Bounds checking
  - Type checking
  - Error checking
  - Checking for valid, not invalid, data
  - Checking input
  - Designing for validation

Bounds Checking

- Indices: off-by-one, signed vs. unsigned
- Pointers: no good way to check bounds automatically
- Implementation Rule 11:
  - **Ensure that all array references access existing elements of the array. If a function that manipulates arrays cannot ensure that only valid elements are referenced, do not use that function. Find one that does, write a new version, or create a wrapper.**

Our Program

- Use loops that check bounds in our code
- Library functions: understand how they work
    - Example: copying strings
        - In C, string is sequence of chars followed by NUL byte (byte containing 0)
        - *strcpy* never checks bounds; too dangerous
        - *strncpy* checks bounds against parameter; danger is not appending terminal NUL byte
    - Example: input user string into buffer
        - *gets* reads, loads until newline encountered
        - *fgets* reads, loads until newline encountered or a specific number of characters are read

Type Checking

- Ensure arguments, inputs, and such are of the right type
  - Interpreting floating point as integer, or shorts as longs
- Implementation Rule 12:
  - **Check the types of functions and parameters.**

Compilers

- Most compilers can do this
  - Declare functions before use; specify types of arguments, result so compiler can check
  - If compiler can't do this, usually other programs can—use them!
- Management Rule 6:
  - **When compiling programs, ensure that the compiler flags report inconsistencies in types. Investigate all such warnings and either fix the problem or document the warning and why it is spurious.**

Error Checking

- Always check return values of functions for errors
  - If function fails, and program accepts result as legitimate, program may act erroneously
- Implementation Rule 13:
  - **Check all function and procedure executions for errors.**

Our Program

- Every function call, library call, system call has return value checked unless return value doesn't matter
  - In some cases, return value of *close* doesn't matter, as program exits and file is closed
  - Here, only true on denial of access or error
    - On success, overlay another program, and files must be closed before that overlay occurs

Check for Valid Data

- Know what data is valid, and check for it
  - Do not check for invalid data unless you are *certain* all other data will be valid for as long as the program is used!
- Implementation Rule 14:
  - **Check that a variable's values are valid.**

Example

- Program executed commands in very restrictive environment
  - Only programs from list could be executed
- Scanned commands looking for metacharacters before passing them to shell for execution
  - Old shell: ‘`’ ordinary character
  - New shell: ‘`x`’ means “run program *x*, and replace `x` with the output of that program
- Result: you could execute any command

Our Program

- Checks that command being executed matches authorized command
  - Rejects anything else
- Problem: can allow all users except a specific set to access a role (keyword "not")
  - Added because on one key system, only system administrators and 1 or 2 trainees
  - Used on that system, but recommended against on all other systems

Handling Trade-Off

- Decision that weakened security made to improve useability
  - Document it and say why
- Management Rule 7:
  - **If a trade-off between security and other factors results in a mechanism or procedure that can weaken security, document the reasons for the decision, the possible effects, and the situations in which the compromise method should be used. This informs others of the trade-off and the attendant risks.**

Checking Input

- Check all data from untrusted sources
  - Users are untrusted sources
- Implementation Rule 15:
  - **Check all user input for both form and content. In particular, check integers for values that are too big or too small, and check character data for length and valid characters.**

Example

- Setting variables while printing
  - *i* contains 2, *j* contains 21

    ```
    printf(“%d %d%n %d\n%n”, i, j, &m, i, &n);
    ```

    stores 4 in *m* and 7 in *n*
- Format string attack
  - User string input stored in *str*, then

    ```
    printf(str)
    ```

    User enters “log%n”, overwriting some memory location with 3
    - If attacker can figure out where that location is, attacker can change the value in that memory location to any desired value

Designing for Validation

- Some validations impossible due to structure of language or other factors
  - Example: in C, test for NULL pointer, but not for valid pointer (unless “valid” means “NULL”)
- Design, implement data structures in such a way that they can be validated
- Implementation Rule 16:
  - **Create data structures and functions in such a way that they can be validated.**

Access Control Entries

- Syntax of file designed to allow for easy error detection:

```
role name
  users comma-separated list of users
  location comma-separated list of locations
  time comma-separated list of times
  command command and arguments
  …
  command command and arguments
endrole
```

- Performs checks on data as appropriate
  - Example: each listed time is a valid time, etc.

Improper Indivisibility

- Operations that should be indivisible are divisible
  - TOCTTOU race conditions, for example
  - Exceptions can break single statements/function calls, etc. into 2 parts as well
- Implementation Rule 17:
  - **If two operations must be performed sequentially without an intervening operation, use a mechanism to ensure that the two cannot be divided.**

Our Program

- Validation, then open, of access control file
  - Method 1: do access check on file name, then open it
    - Problem: if attacker can write to directory in full path name of file, attacker can switch files after validation but before opening
  - Method 2 (program uses this): open file, then *before reading from it* do access check on file descriptor
    - As check is done on open file, and file descriptor cannot be switched to another file unless closed, this provides protection
  - Method 3 (not implemented): do it all in the kernel as part of the open system call!

Improper Sequencing

- Operations performed in incorrect order
- Implementation Rule 18:
  - **Describe the legal sequences of operations on a resource or object. Check that all possible sequences of the program(s) involved match one (or more) legal sequences.**

Our Program

- Sequence of operations follow proper order:
  - User authenticated
  - Program checks access
  - If allowed:
    - New, safe environment set up
    - Command executed in it
- When dropping privileges, note ordinary user cannot change groups, but *root* can
  - Change group to that of role account
  - Change user to that of role account

Improper Choice of Operand or Operation

- Erroneous selection of operation or operand
- Example: *su* used to access *root* account
  - Requires user to know *root* password
  - If no password file, cannot validate entered password
  - One program assumed no password file if it couldn't open it, and gave user *root* access to fix problem
    - Attacker: open all file descriptors possible, spawn *su*—as open file descriptors inherited, *su* couldn't open any files—not even password file
    - Improper operation: should have checked to see if no password file or no available file descriptors

Assurance

- Use assurance techniques
  - Document purpose, use of each function
  - Check algorithm, call
- Management Rule 8:
  - **Use software engineering and assurance techniques (such as documentation, design reviews, and code reviews) to ensure that operations and operands are appropriate.**

Our Program

- Granting Access
  - Only when entry matches *all* characteristics of current session
    - When characteristics match, verify access control module returns true
    - Check when module returns true, program grants access and when module returns false, denies access
- Consider UID (type uid_t, or unsigned integer)
  - Check that it can be considered as integer
    - If comparing signed and unsigned, then signed converted to unsigned; check there are no comparisons with negative numbersr

Our Program (*con't*)

- Consider location
  - Check that match routine correctly determines whether location passed in matches pattern in location field of access control entries, and module acts appropriately
- Consider time (type time_t)
  - Check module interprets time as range
  - Example: 9AM means 09:00:00—09:59:59, not 09:00:00
    - If interpreted as *exactly* 9:00:00, almost impossible for user to hit exact time, effectively disabling the entry; violates Requirement 4

Our Program (*con't*)

- Signal handlers
  - Signal indicates: error in program; or request from user to terminate
  - Signal should terminate program
  - If program tries to recover, and continues to run, and grants access to role account, either it continued in face of error, or it overrode user's attempt to terminate program
    - Either way, choice of improper operation

Summary

- Approach differs from using checklist of common vulnerabilities
- Approach is design approach
  - Apply it at each level of refinement
  - Emphasizes documentation, analysis, understanding or program, interfaces, execution environment
  - Documentation will help other analysts, or folks moving program to new system with different environment

Testing

- Informal validation of design, operation of program
  - Goal: show program meets stated requirements
  - If requirements drive design, implementation then testing likely to uncover minor problems
  - If requirements ill posed, or change during development, testing may uncover major problems
    - In this case, *do not* add features to meet requirements! Redesign and reimplement …

Process

- Construct environment matching production environment
  - If range of environments, need to test in all
    - Usually considerable overlap, so not so bad …
  - If repeated failures, check developer assumptions
    - May have embedded information about development environment—one different than testing environment!

Steps

- Begin with requirements
  - Appropriate?
  - Does it solve the problem?
- Proceed to design
  - Decomposition into modules allows testing of each module, with stubs to take place of uncompleted modules
- Then to implementation
  - Test each module
  - Test interfaces (composition of modules)

Philosophy

- Execute all possible paths of control
    - Compare results with expected results
- In practise, infeasible
    - Analyze paths, order them in some way
        - Order depends on requirements
    - Generate test data for each one, check each
- Security testing: also test least commonly used paths
    - Usually not as well checked, so miss vulnerabilities
- First check modules, then check composition

Testing Module

- Goal: ensure module acts correctly
  - If it calls functions, correctly regardless of what functions return
- Step 1: define "correct behavior"
  - Done during refinement, when module specified
- Step 2: list interfaces to module
  - Use this to execute tests

Types of Tests

- Normal data tests
  - Unexceptional data
  - Exercise as many paths of control as possible
- Boundary data tests
  - Test limits to interfaces
  - Example: if string is at most 256 chars, try 255, then 256, then 257 chars
  - Example: in our program, try UID of –1 in parameter list
    - Is it rejected or remapped to 2^{31}–1 or 2^{16}–1?

Types of Tests (*con't*)

- Exception tests
  - How module handle interrupts, traps
  - Example: send program signal to cause core dump, see if passwords visible in that file
- Random data tests
  - Give module data generated randomly
  - Module should fail but restore system to safe state
  - Example: in one study of UNIX utilities, 30% crashed when given random inputs

Testing Composed Modules

- Consider module that calls other modules
- Error handling tests
  - Assume called modules violate specifications
  - See if this module violates specification
- Example: logging via mail program
  - Program logs connecting host by mail

    ```
    mail –s hostname netadmin
    ```

  - Gets host name by mapping IP address using DNS
  - DNS has fake record: `hi nobody; rm -rf *; true`
  - When mail command executed, deletes files

Testing Program

- Testers assemble program, documentation
- New tester follows instructions to install, configure program and tries it
  - This tester should *not* be associated with other testers, so can provide independent assessment of documentation, correctness of instructions
- Problems may be with documentation, installation program or scripts, or program itself

Distribution

- Place program, documentation in repository where only authorized people can alter it and from where it can be sent to recipients
- Several factors afftct how this is done

Factors

- Who can use this program?
  - Licensed to organization: tie each copy to the organization so it cannot be redistributed
- How can availability be ensured?
  - Physical means: distribute via CD-ROM, for example
    - Mail, messenger services control availability
  - Electronic means: via ftp, web, etc.
    - Ensure site is available

Factors (*con't*)

- How to protect integrity of master copy?
  - Attacker changing distribution copy can attack everyone who gets it
  - Example: *tcp_wrappers* altered at repository to incluse backdoor; 59 hosts compromised when they downloaded and installed it
  - Damages credibility of vendor
  - Customers may disbelieve vendors when warned

Key Points

- Security in programming best done by mimicing high assurance techniques
- Begin with requirements analysis and validation
- Map requirements to design
- Map design to implementation
  - Watch out for common vulnerabilities
- Test thoroughly
- Distribute carefully