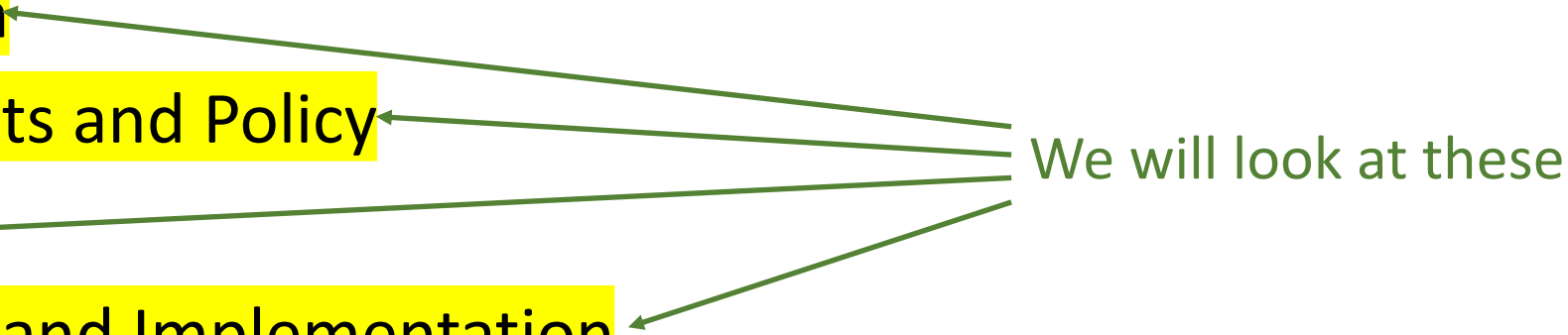
ECS 235B Module 62
Program Security

Program Security Components

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

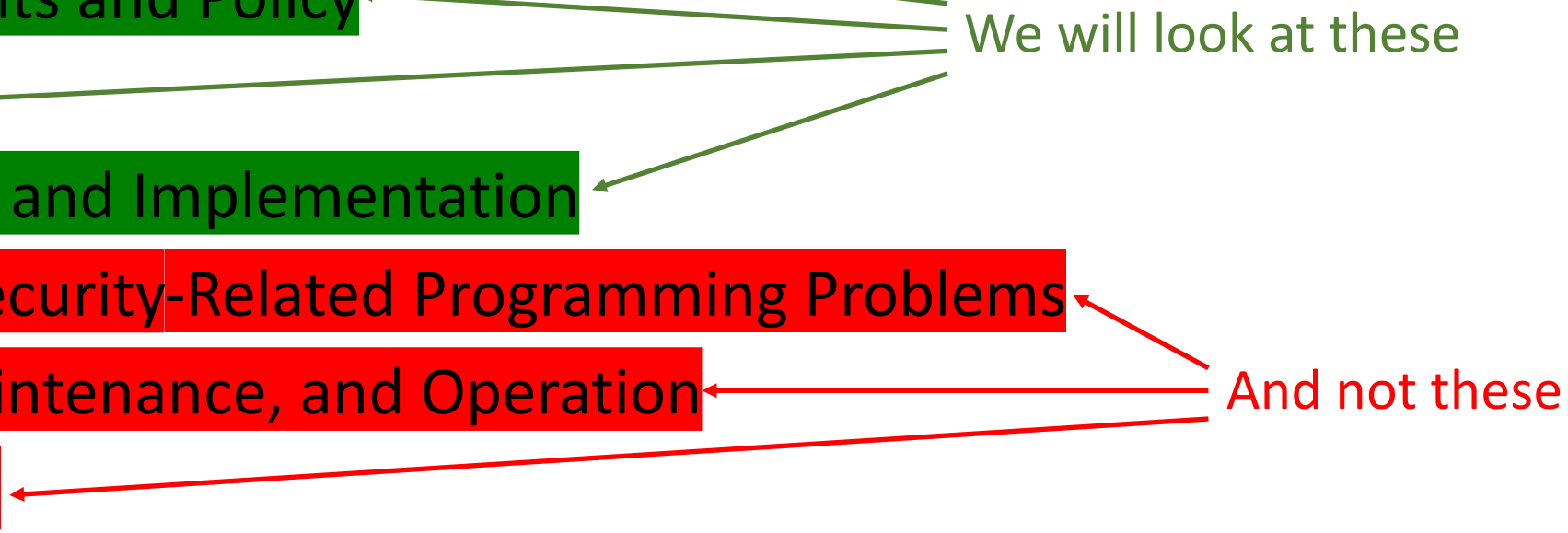
Program Security Components

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



Program Security Components

- 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
  - Like *sudo*(1) but offers different constraints

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*

    ```
    Monday-Thursday 9a.m.-5p.m.
    ```

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

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