

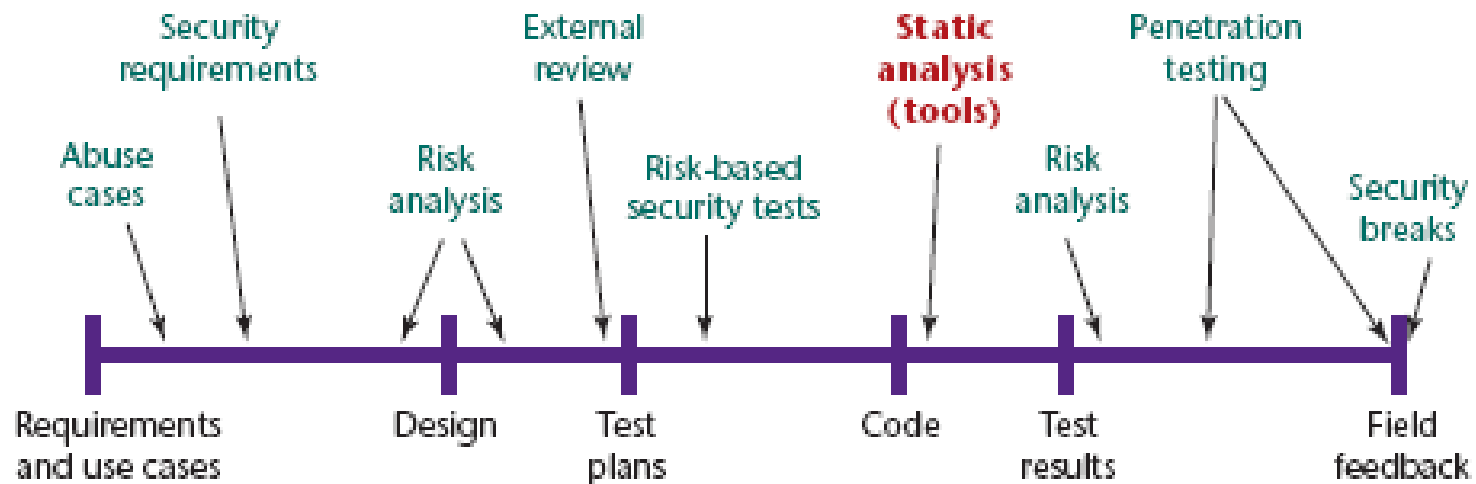
University of Rochester
CSC290B
Introduction to Computer Security

Vulnerabilities

February 3, 2009

Where are we?

- The software development life cycle (SDLC)
- Awareness of common weaknesses assists:
 - Prevention of design flaws
 - Analysis of vulnerabilities



Source: Gary McGraw, *Static Analysis for Security*

Vulnerability Analysis

- Background
- Penetration Studies
- Example Vulnerabilities
- Classification Frameworks
- Some Theory of Penetration Analysis

Overview

- What is a vulnerability?
- Penetration studies
 - Flaw Hypothesis Methodology
 - Examples
- Vulnerability examples
- Classification schemes
 - RISOS, PA, NRL Taxonomy, Aslam's Model
- Theory of penetration analysis
 - Examples

Definitions

- *Vulnerability, security flaw*: failure of security policies, procedures, and controls that allow a subject to commit an action that violates the security policy
 - Subject is called an *attacker*
 - Using the failure to violate the policy is *exploiting the vulnerability* or *breaking in*

Formal Verification

- Mathematically verifying that a system satisfies certain constraints
- *Preconditions* state assumptions about the system
- *Postconditions* are result of applying system operations to preconditions, inputs
- Required: postconditions satisfy constraints

Penetration Testing

- Testing to verify that a system satisfies certain constraints
- Hypothesis stating system characteristics, environment, and state relevant to vulnerability
- Result is compromised system state
- Apply tests to try to move system from state in hypothesis to compromised system state

Notes

- Penetration testing is a *testing* technique, not a verification technique
 - It can prove the *presence* of vulnerabilities, but not the *absence* of vulnerabilities
- For formal verification to prove absence, proof and preconditions must include *all* external factors
 - Realistically, formal verification proves absence of flaws within a particular program, design, or environment and not the absence of flaws in a computer system (think incorrect configurations, etc.)

Penetration Studies

- Test for evaluating the strengths and effectiveness of all security controls on system
 - Also called *tiger team attack* or *red team attack*
 - Goal: violate site security policy
 - Not a replacement for careful design, implementation, and structured testing
 - Tests system *in toto*, once it is in place
 - Includes procedural, operational controls as well as technological ones

Goals

- Attempt to violate specific constraints in security and/or integrity policy
 - Implies metric for determining success
 - Must be well-defined
- Example: subsystem designed to allow owner to require others to give password before accessing file (i.e., password protect files)
 - Goal: test this control
 - Metric: did testers get access either without a password or by gaining unauthorized access to a password?

Goals

- Find some number of vulnerabilities, or vulnerabilities within a period of time
 - If vulnerabilities categorized and studied, can draw conclusions about care taken in design, implementation, and operation
 - Otherwise, list helpful in closing holes but not more
- Example: vendor gets confidential documents, 30 days later publishes them on web
 - Goal: obtain access to such a file; you have 30 days
 - Alternate goal: gain access to files; no time limit (a Trojan horse would give access for over 30 days)

Layering of Tests

1. External attacker with no knowledge of system
 - Locate system, learn enough to be able to access it
2. External attacker with access to system
 - Can log in, or access network servers
 - Often try to expand level of access
3. Internal attacker with access to system
 - Testers are authorized users with restricted accounts (like ordinary users)
 - Typical goal is to gain unauthorized privileges or information

Layering of Tests (con't)

- Studies conducted from attacker's point of view
- Environment is that in which attacker would function
- If information about a particular layer irrelevant, layer can be skipped
 - Example: penetration testing during design, development skips layer 1
 - Example: penetration test on system with guest account usually skips layer 2

Methodology

- Usefulness of penetration study comes from documentation, conclusions
 - Indicates whether flaws are endemic or not
 - It does not come from success or failure of attempted penetration
- Degree of penetration's success also a factor
 - In some situations, obtaining access to unprivileged account may be less successful than obtaining access to privileged account

Flaw Hypothesis Methodology

1. Information gathering
 - Become familiar with system's functioning
2. Flaw hypothesis
 - Draw on knowledge to hypothesize vulnerabilities
3. Flaw testing
 - Test them out
4. Flaw generalization
 - Generalize vulnerability to find others like it
5. (*maybe*) Flaw elimination
 - Testers eliminate the flaw (usually *not* included)

1. Information Gathering

- Devise model of system and/or components
 - Look for discrepancies in components
 - Consider interfaces among components
- Need to know system well (or learn quickly!)
 - Design documents, manuals help
 - Unclear specifications often misinterpreted, or interpreted differently by different people
 - Look at how system manages privileged users

2. Flaw Hypothesizing

- Examine policies, procedures
 - May be inconsistencies to exploit
 - May be consistent, but inconsistent with design or implementation
 - May not be followed
- Examine implementations
 - Use models of vulnerabilities to help locate potential problems
 - Use manuals; try exceeding limits and restrictions; try omitting steps in procedures

2. Flaw Hypothesizing (*con't*)

- Identify structures, mechanisms controlling system
 - These are what attackers will use
 - Environment in which they work, and were built, may have introduced errors
- Throughout, draw on knowledge of other systems with similarities
 - Which means they may have similar vulnerabilities
- Result is list of possible flaws

3. Flaw Testing

- Figure out order to test potential flaws
 - Priority is function of goals
 - Example: to find major design or implementation problems, focus on potential system critical flaws
 - Example: to find vulnerability to outside attackers, focus on external access protocols and programs
- Figure out how to test potential flaws
 - Best way: demonstrate from the analysis
 - Common when flaw arises from faulty spec, design, or operation
 - Otherwise, must try to exploit it

3. Flaw Testing (*con't*)

- Design test to be least intrusive as possible
 - Must understand exactly why flaw might arise
- Procedure
 - Back up system
 - Verify system configured to allow exploit
 - Take notes of requirements for detecting flaw
 - Verify existence of flaw
 - May or may not require exploiting the flaw
 - Make test as simple as possible, but success must be convincing
 - Must be able to repeat test successfully

4. Flaw Generalization

- As tests succeed, classes of flaws emerge
 - Example: programs read input into buffer on stack, leading to buffer overflow attack; others copy command line arguments into buffer on stack \Rightarrow these are vulnerable too
- Sometimes two different flaws may combine for devastating attack
 - Example: flaw 1 gives external attacker access to unprivileged account on system; second flaw allows any user on that system to gain full privileges \Rightarrow any external attacker can get full privileges

5. Flaw Elimination

- Usually not included as testers are not best folks to fix this
 - Designers and implementers are
- Requires understanding of context, details of flaw including environment, and possibly exploit
 - Design flaw uncovered during development can be corrected and parts of implementation redone
 - Don't need to know how exploit works
 - Design flaw uncovered at production site may not be corrected fast enough to prevent exploitation
 - So need to know how exploit works

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

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

Vulnerability Classification

- Describe flaws from differing perspectives
 - Exploit-oriented
 - Hardware, software, interface-oriented
- Goals vary; common ones are:
 - Specify, design, implement computer system without vulnerabilities
 - Analyze computer system to detect vulnerabilities
 - Address any vulnerabilities introduced during system operation
 - Detect attempted exploitations of vulnerabilities

Example Flaws

- Use these to compare classification schemes
- First one: race condition (*xterm*)
- Second one: buffer overflow on stack leading to execution of injected code (*fingerd*)
- Both are very well known, and fixes available!
 - And should be installed everywhere ...

Flaw #1: xterm

- *xterm* emulates terminal under X11 window system
 - Must run as *root* user on UNIX systems
 - No longer universally true; reason irrelevant here
- Log feature: user can log all input, output to file
 - User names file
 - If file does not exist, *xterm* creates it, makes owner the user
 - If file exists, *xterm* checks user can write to it, and if so opens file to append log to it

File Exists

- Check that user can write to file requires special system call
 - Because *root* can append to any file, check in *open* will always succeed

Check that user can write to file “/usr/tom/X”

```
if (access("/usr/tom/X", W_OK) == 0){
```

Open “/usr/tom/X” to append log entries

```
    if ((fd = open("/usr/tom/X", O_WRONLY|O_APPEND)) < 0){
```

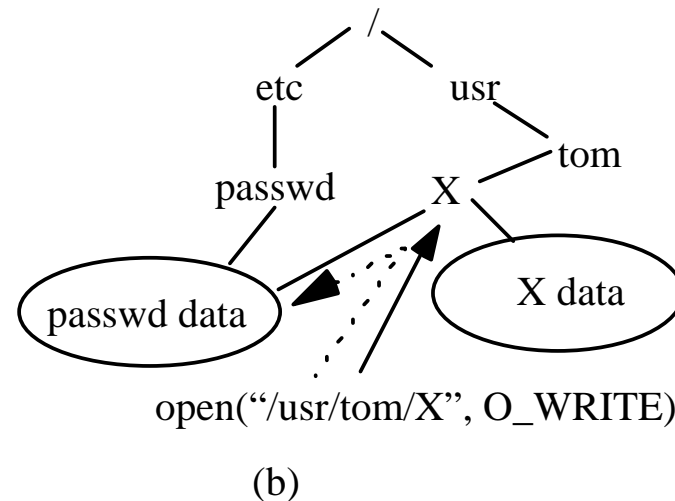
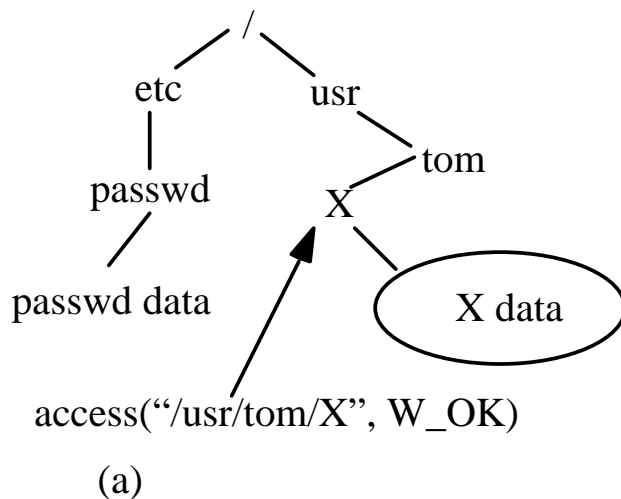
```
        /* handle error: cannot open file */
```

```
    }
```

```
}
```

Problem

- Binding of file name “/usr/tom/X” to file object can change between first and second lines
 - (a) is at *access*; (b) is at *open*
 - Note file opened is *not* file checked

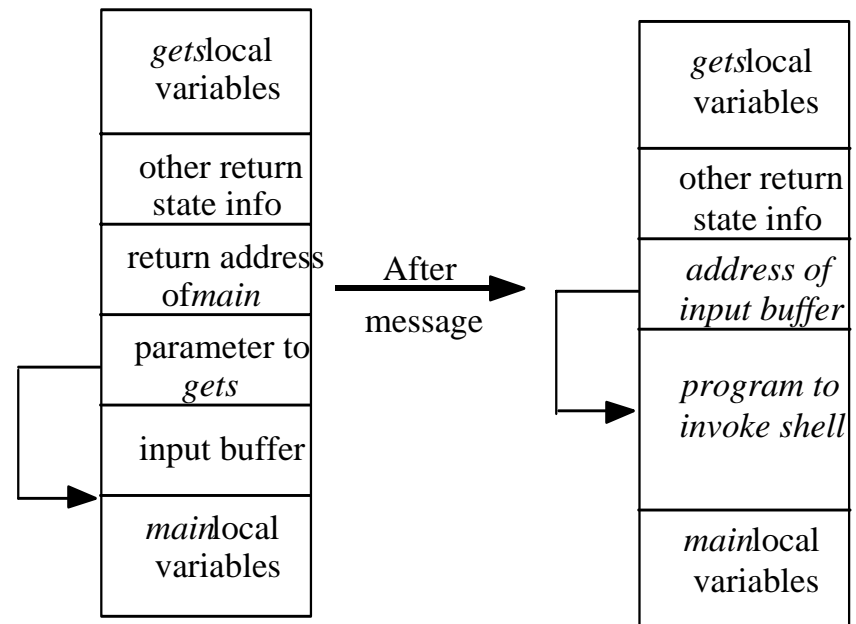


Flaw #2: *fingerd*

- Exploited by Internet Worm of 1988
 - Recurs in many places, even now
- *finger* client send request for information to server *fingerd* (*finger* daemon)
 - Request is name of at most 512 chars
 - What happens if you send more?

Buffer Overflow

- Extra chars overwrite rest of stack, as shown
- Can make those chars change return address to point to beginning of buffer
- If buffer contains small program to spawn shell, attacker gets shell on target system



Frameworks

- Goals dictate structure of classification scheme
 - Guide development of attack tool \Rightarrow focus is on steps needed to exploit vulnerability
 - Aid software development process \Rightarrow focus is on design and programming errors causing vulnerabilities
- Following schemes classify vulnerability as n-tuple, each element of n-tuple being classes into which vulnerability falls
 - Some have 1 axis; others have multiple axes

Research Into Secure Operating Systems (RISOS)

- Goal: aid computer, system managers in understanding security issues in OSeS, and help determine how much effort required to enhance system security
- Attempted to develop methodologies and software for detecting some problems, and techniques for avoiding and ameliorating other problems
- Examined Multics, TENEX, TOPS-10, GECOS, OS/MVT, SDS-940, EXEC-8

RISOS Classification Scheme

- * Incomplete parameter validation
- * Inconsistent parameter validation
- * Implicit sharing of privileged/confidential data
- * Asynchronous validation/inadequate serialization
- * Inadequate
identification/authentication/authorization
- * Violable prohibition/limit
- * Exploitable logic error

* Incomplete Parameter Validation

- Parameter not checked before use
- Example: emulating integer division in kernel (RISC chip involved)
 - Caller provided addresses for quotient, remainder
 - Quotient address checked to be sure it was in user's protection domain
 - Remainder address *not* checked
 - Set remainder address to address of process' level of privilege
 - Compute $25/5$ and you have level 0 (kernel) privileges
- Check for type, format, range of values, access rights, presence (or absence)

* Inconsistent Parameter Validation

- Each routine checks parameter is in proper format for that routine but the routines require different formats
- Example: each database record 1 line, colons separating fields
 - One program accepts colons, newlines as pat of data within fields
 - Another program reads them as field and record separators
 - This allows bogus records to be entered

* Implicit Sharing of Privileged / Confidential Data

- OS does not isolate users, processes properly
- Example: file password protection
 - OS allows user to determine when paging occurs
 - Files protected by passwords
 - Passwords checked char by char; stops at first incorrect char
 - Position guess for password so page fault occurred between 1st, 2nd char
 - If no page fault, 1st char was wrong; if page fault, it was right
 - Continue until password discovered

* Asynchronous Validation / Inadequate Serialization

- Time of check to time of use (TOCTOU) flaws, intermixing reads and writes to create inconsistencies
- Example: *xterm* flaw discussed earlier

* Inadequate Identification / Authorization / Authentication

- Erroneously identifying user, assuming another's privilege, or tricking someone into executing program without authorization
- Example: OS on which access to file named “SYSS*DLOC\$” meant process privileged
 - Check: can process access any file with qualifier name beginning with “SYS” and file name beginning with “DLO”?
 - If your process can access file “SYSA*DLOC\$”, which is ordinary file, your process is privileged

* Violable Prohibition / Limit

- Boundary conditions not handled properly
- Example: OS kept in low memory, user process in high memory
 - Boundary was highest address of OS
 - All memory accesses checked against this
 - Memory accesses not checked beyond end of high memory
 - Such addresses reduced modulo memory size
 - So, process could access $(\text{memory size})+1$, or word 1, which is part of OS ...

* Exploitable Logic Error

- Problems not falling into other classes
 - Incorrect error handling, unexpected side effects, incorrect resource allocation, etc.
- Example: unchecked return from monitor
 - Monitor adds 1 to address in user's PC, returns
 - Index bit (indicating indirection) is a bit in word
 - Attack: set address to be -1 ; adding 1 overflows, changes index bit, so return is to location stored in register 1
 - Arrange for this to point to bootstrap program stored in other registers
 - On return, program executes with system privileges

Legacy of RISOS

- First funded project examining vulnerabilities
- Valuable insight into nature of flaws
 - Security is a function of site requirements and threats
 - Small number of fundamental flaws recurring in many contexts
 - OS security not critical factor in design of OSeS
- Spurred additional research efforts into detection, repair of vulnerabilities

Protection Analysis (PA)

- Goal: develop techniques to find vulnerabilities
- Tried to break problem into smaller, more manageable pieces
- Developed general strategy, applied it to several OSes
 - Found previously unknown vulnerabilities

Classification Scheme

- Improper protection domain initialization and enforcement
 - Improper choice of initial protection domain
 - Improper isolation of implementation detail
 - Improper change
 - Improper naming
 - Improper deallocation or deletion
- Improper validation
- Improper synchronization
 - Improper indivisibility
 - Improper sequencing
- Improper choice of operand or operation

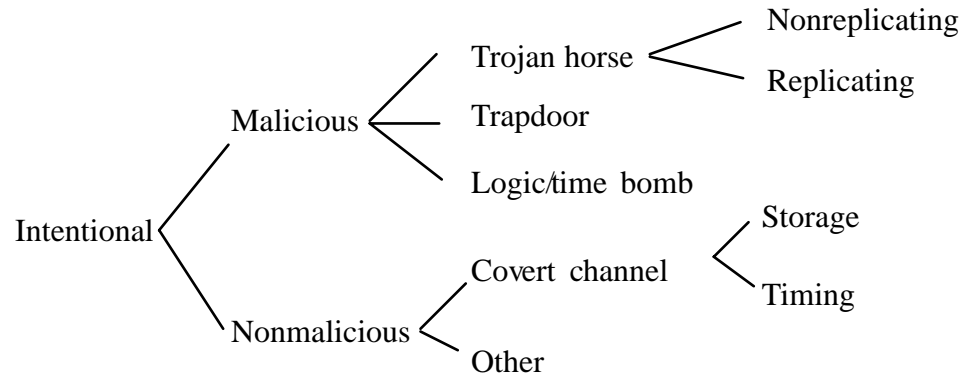
Legacy

- First to explore automatic detection of security flaws in programs and systems
- Methods developed but not widely used
 - Parts of procedure could not be automated
 - Complexity
 - Procedures for obtaining system-independent patterns describing flaws not complete
- *See the text, Bishop, 23.4.2*

NRL Taxonomy

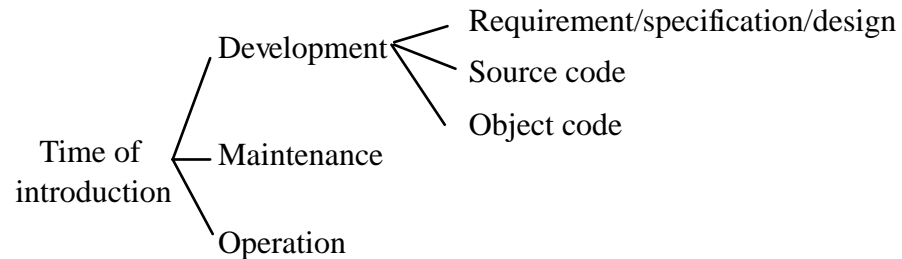
- Goals:
 - Determine how flaws entered system
 - Determine when flaws entered system
 - Determine where flaws are manifested in system
- 3 different schemes used:
 - Genesis of flaws
 - Time of flaws
 - Location of flaws

Genesis of Flaws



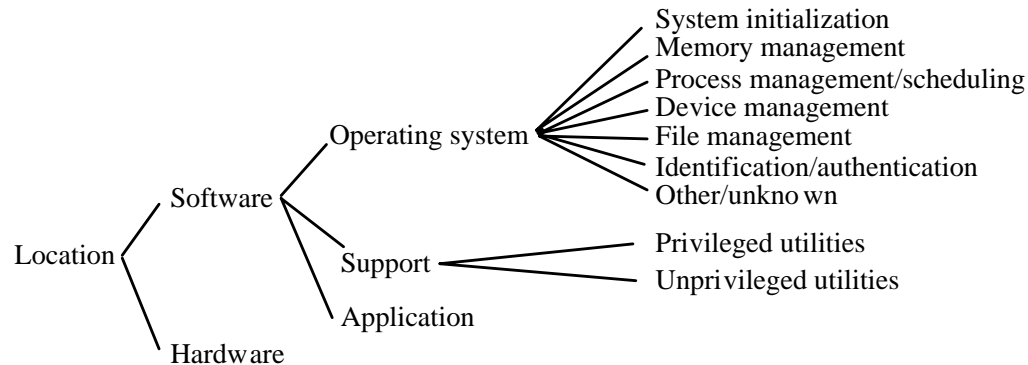
- Inadvertent (unintentional) flaws classified using RISOS categories; not shown above
 - If most inadvertent, better design/coding reviews needed
 - If most intentional, need to hire more trustworthy developers and do more security-related testing

Time of Flaws



- Development phase: all activities up to release of initial version of software
- Maintenance phase: all activities leading to changes in software performed under configuration control
- Operation phase: all activities involving patching and not under configuration control

Location of Flaw



- Focus effort on locations where most flaws occur, or where most serious flaws occur

Legacy

- Analyzed 50 flaws
- Concluded that, with a large enough sample size, an analyst could study relationships between pairs of classes
 - This would help developers focus on most likely places, times, and causes of flaws
- Focused on social processes as well as technical details
 - But much information required for classification not available for the 50 flaws

Aslam's Model

- Goal: treat vulnerabilities as faults and develop scheme based on fault trees
- Focuses specifically on UNIX flaws
- Classifications unique and unambiguous
 - Organized as a binary tree, with a question at each node. Answer determines branch you take
 - Leaf node gives you classification
- Suited for organizing flaws in a database

Top Level

- Coding faults: introduced during software development
 - Example: *fingerd*'s failure to check length of input string before storing it in buffer
- Emergent faults: result from incorrect initialization, use, or application
 - Example: allowing message transfer agent to forward mail to arbitrary file on system (it performs according to specification, but results create a vulnerability)

Coding Faults

- Synchronization errors: improper serialization of operations, timing window between two operations creates flaw
 - Example: *xterm* flaw
- Condition validation errors: bounds not checked, access rights ignored, input not validated, authentication and identification fails
 - Example: *fingerd* flaw

Emergent Faults

- Configuration errors: program installed incorrectly
 - Example: *tftp* daemon installed so it can access any file; then anyone can copy any file
- Environmental faults: faults introduced by environment
 - Example: on some UNIX systems, any shell with “-” as first char of name is interactive, so find a setuid shell script, create a link to name “-gotcha”, run it, and you has a privileged interactive shell

Legacy

- Tied security flaws to software faults
- Introduced a precise classification scheme
 - Each vulnerability belongs to exactly 1 class of security flaws
 - Decision procedure well-defined, unambiguous

Resources

- CWE (Common Weakness Enumeration) at cwe.mitre.org
 - “provides a unified, measurable set of software weaknesses that is enabling more effective discussion, description, selection, and use of software security tools and services that can find these weaknesses in *source code* and operational systems as well as better understanding and management of software weaknesses related to architecture and design.”
- CVE (Common Vulnerabilities and Exposures) at cve.mitre.org
 - a dictionary of publicly known information security vulnerabilities and exposures

Acknowledgements

- Substantial portions of these slides are courtesy of Matt Bishop and used with permission