Chapter 23: Vulnerability Analysis

• Background
• Penetration Studies
• Example Vulnerabilities
• Classification Frameworks
• 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)
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
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
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
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
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
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 ⇒ 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 ⇒ any external attacker can get full privileges
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
Michigan Terminal System

- General-purpose OS running on IBM 360, 370 systems
- Class exercise: gain access to terminal control structures
  - Had approval and support of center staff
  - Began with authorized account (level 3)
Step 1: Information Gathering

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

• Focus on segment 5
Step 2: Information Gathering

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

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

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X+2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

X  X+1 X+2
Step 3: Flaw Hypothesis

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

```
+---+---+---+
|   |   |   |
+---+---+---+
   X   X+2 ...

X   X+1   X+2
```

June 1, 2004

Computer Security: Art and Science
©2004 Matt Bishop
Step 4: Flaw Testing

• Find a system routine that:
  – Used this calling convention;
  – Took at least 2 parameters and altered 1
  – Could be made to change parameter to any value (such as an address in segment 5)

• Chose line input routine
  – Returns line number, length of line, line read

• Setup:
  – Set address for storing line number to be address of line length
Step 5: Execution

- System routine validated all parameter addresses
  - All were indeed in user segment
- Supervisor read input line
  - Line length set to value to be written into segment 5
- Line number stored in parameter list
  - Line number was set to be address in segment 5
- When line read, line length written into location address of which was in parameter list
  - So it overwrote value in segment 5
Step 6: Flaw Generalization

• Could not overwrite anything in segments 0-4
  – Protected by hardware
• Testers realized that privilege level in segment 5 controlled ability to issue supervisor calls (as opposed to system calls)
  – And one such call turned off hardware protection for segments 0-4 …
• Effect: this flaw allowed attackers to alter anything in memory, thereby completely controlling computer
Burroughs B6700

- System architecture: based on strict file typing
  - Entities: ordinary users, privileged users, privileged programs, OS tasks
    - Ordinary users tightly restricted
    - Other 3 can access file data without restriction but constrained from compromising integrity of system
  - No assemblers; compilers output executable code
  - Data files, executable files have different types
    - Only compilers can produce executables
    - Writing to executable or its attributes changes its type to data
- Class exercise: obtain status of privileged user
Step 1: Information Gathering

- System had tape drives
  - Writing file to tape preserved file contents
  - Header record indicates file attributes including type
- Data could be copied from one tape to another
  - If you change data, it’s still data
Step 2: Flaw Hypothesis

- System cannot detect change to executable file if that file is altered off-line
Step 3: Flaw Testing

• Write small program to change type of any file from data to executable
  – Compiled, but could not be used yet as it would alter file attributes, making target a data file
  – Write this to tape

• Write a small utility to copy contents of tape 1 to tape 2
  – Utility also changes header record of contents to indicate file was a compiler (and so could output executables)
Creating the Compiler

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

- Goal: determine whether corporate security measures were effective in keeping external attackers from accessing system
- Testers focused on policies and procedures
  - Both technical and non-technical
Step 1: Information Gathering

- Searched Internet
  - Got names of employees, officials
  - Got telephone number of local branch, and from them got copy of annual report

- Constructed much of the company’s organization from this data
  - Including list of some projects on which individuals were working
Step 2: Get Telephone Directory

- Corporate directory would give more needed information about structure
  - Tester impersonated new employee
    - Learned two numbers needed to have something delivered off-site: employee number of person requesting shipment, and employee’s Cost Center number
  - Testers called secretary of executive they knew most about
    - One impersonated an employee, got executive’s employee number
    - Another impersonated auditor, got Cost Center number
  - Had corporate directory sent to off-site “subcontractor”
Step 3: Flaw Hypothesis

- Controls blocking people giving passwords away not fully communicated to new employees
  - Testers impersonated secretary of senior executive
    - Called appropriate office
    - Claimed senior executive upset he had not been given names of employees hired that week
    - Got the names
Step 4: Flaw Testing

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

- Goal: gain access to system
- We know its network address and nothing else
- First step: scan network ports of system
  - Protocols on ports 79, 111, 512, 513, 514, and 540 are typically run on UNIX systems
- Assume UNIX system; SMTP agent probably *sendmail*
  - This program has had lots of security problems
  - Maybe system running one such version …
- Next step: connect to *sendmail* on port 25
### Output of Network Scan

<table>
<thead>
<tr>
<th>Service</th>
<th>Port Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp</td>
<td>21/tcp</td>
<td>File Transfer</td>
</tr>
<tr>
<td>telnet</td>
<td>23/tcp</td>
<td>Telnet</td>
</tr>
<tr>
<td>smtp</td>
<td>25/tcp</td>
<td>Simple Mail Transfer</td>
</tr>
<tr>
<td>finger</td>
<td>79/tcp</td>
<td>Finger</td>
</tr>
<tr>
<td>sunrpc</td>
<td>111/tcp</td>
<td>SUN Remote Procedure Call</td>
</tr>
<tr>
<td>exec</td>
<td>512/tcp</td>
<td>remote process execution (rexecd)</td>
</tr>
<tr>
<td>login</td>
<td>513/tcp</td>
<td>remote login (rlogind)</td>
</tr>
<tr>
<td>shell</td>
<td>514/tcp</td>
<td>rlogin style exec (rshd)</td>
</tr>
<tr>
<td>printer</td>
<td>515/tcp</td>
<td>spooler (lpd)</td>
</tr>
<tr>
<td>uucp</td>
<td>540/tcp</td>
<td>uucpd</td>
</tr>
<tr>
<td>nfs</td>
<td>2049/tcp</td>
<td>networked file system</td>
</tr>
<tr>
<td>xterm</td>
<td>6000/tcp</td>
<td>x-windows server</td>
</tr>
</tbody>
</table>
Output of `sendmail`

```
220 zzz.com sendmail 3.1/zzz.3.9, Dallas, Texas, ready
   at Wed, 2 Apr 97 22:07:31 CST
   Version 3.1 has the “wiz” vulnerability that recognizes
   the “shell” command … so let’s try it
   Start off by identifying yourself
   helo xxx.org
   250 zzz.com Hello xxx.org, pleased to meet you
   Now see if the “wiz” command works … if it says “command
   unrecognized”, we’re out of luck
   wiz
   250 Enter, O mighty wizard!
      It does! And we didn’t need a password … so get a shell
   shell
   #
   And we have full privileges as the superuser, root
```
Penetrating a System (Revisited)

• Goal: from an unprivileged account on system, gain privileged access
• First step: examine system
  – See it has dynamically loaded kernel
  – Program used to add modules is loadmodule and must be privileged
  – So an unprivileged user can run a privileged program
    … this suggests an interface that controls this
  – Question: how does loadmodule work?
loadmodule

- Validates module ad being a dynamic load module
- Invokes dynamic loader *ld.so* to do actual load; also calls *arch* to determine system architecture (chip set)
  - Check, but only privileged user can call *ld.so*
- How does *loadmodule* execute these programs?
  - Easiest way: invoke them directly using *system*(3), which does not reset environment when it spawns subprogram
First Try

• Set environment to look in local directory, write own version of `ld.so`, and put it in local directory
  – This version will print effective UID, to demonstrate we succeeded

• Set search path to look in current working directory *before* system directories

• Then run `loadmodule`
  – Nothing is printed—darn!
  – Somehow changing environment did not affect execution of subprograms—why not?
What Happened

- Look in executable to see how ld.so, arch invoked
  - Invocations are “/bin/ld.so”, “/bin/arch”
  - Changing search path didn’t matter as never used
- Reread system(3) manual page
  - It invokes command interpreter sh to run subcommands
- Read sh(1) manual page
  - Uses IFS environment variable to separate words
  - These are by default blanks … can we make it include a “/”?
    - If so, sh would see “/bin/ld.so” as “bin” followed by “ld.so”, so it would look for command “bin”
Second Try

• Change value of **IFS** to include “/”
• Change name of our version of *ld.so* to *bin*
  – Search path still has current directory as first place to look for commands
• Run *loadmodule*
  – Prints that its effective UID is 0 (root)
• Success!
Generalization

- Process did not clean out environment before invoking subprocess, which inherited environment
  - So, trusted program working with untrusted environment (input) … result should be untrusted, but is trusted!

- Look for other privileged programs that spawn subcommands
  - Especially if they do so by calling `system(3)` …
Penetrating a System \textit{redux}

- Goal: gain access to system
- We know its network address and nothing else
- First step: scan network ports of system
  - Protocols on ports 17, 135, and 139 are typically run on Windows NT server systems
Output of Network Scan

<table>
<thead>
<tr>
<th>Service</th>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qotd</td>
<td>17/tcp</td>
<td>Quote of the Day</td>
</tr>
<tr>
<td>ftp</td>
<td>21/tcp</td>
<td>File Transfer [Control]</td>
</tr>
<tr>
<td>loc-srv</td>
<td>135/tcp</td>
<td>Location Service</td>
</tr>
<tr>
<td>netbios-ssn</td>
<td>139/tcp</td>
<td>NETBIOS Session Service [JBP]</td>
</tr>
</tbody>
</table>
First Try

• Probe for easy-to-guess passwords
  – Find system administrator has password “Admin”
  – Now have administrator (full) privileges on local system
• Now, go for rights to other systems in domain
Next Step

- Domain administrator installed service running with domain admin privileges on local system
- Get program that dumps local security authority database
  - This gives us service account password
  - We use it to get domain admin privileges, and can access any system in domain
Generalization

- Sensitive account had an easy-to-guess password
  - Possible procedural problem
- Look for weak passwords on other systems, accounts
- Review company security policies, as well as education of system administrators and mechanisms for publicizing the policies
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”

```c
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

(a) open("/usr/tom/X", O_WRITE)
(b) open("/usr/tom/X", O_WRITE)
Flaw #2: \textit{fingerd}

- Exploited by Internet Worm of 1988
  - Recurs in many places, even now
- \textit{finger} client send request for information to server \textit{fingerd} (\textit{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 ⇒ focus is on steps needed to exploit vulnerability
  – Aid software development process ⇒ 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
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 part 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 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 “SYS$*DLOC$” 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 (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
Program 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
Improper Choice of Initial Protection Domain

• Initial incorrect assignment of privileges, security and integrity classes

• Example: on boot, protection mode of file containing identifiers of all users can be altered by any user
  – Under most policies, should not be allowed
Improper Isolation of Implementation Detail

- Mapping an abstraction into an implementation in such a way that the abstraction can be bypassed

- Example: virtual machines modulate length of time CPU is used by each to send bits to each other

- Example: Having raw disk accessible to system as ordinary file, enabling users to bypass file system abstraction and write directly to raw disk blocks
Improper Change

- Data is inconsistent over a period of time
- Example: *xterm* flaw
  - Meaning of “/usr/tom/X” changes between *access* and *open*
- Example: parameter is validated, then accessed; but parameter is changed between validation and access
  - Burroughs B6700 allowed allowed this
Improper Naming

- Multiple objects with same name
- Example: Trojan horse
  - *loadmodule* attack discussed earlier; “bin” could be a directory or a program
- Example: multiple hosts with same IP address
  - Messages may be erroneously routed
Improper Deallocation or Deletion

• Failing to clear memory or disk blocks (or other storage) after it is freed for use by others

• Example: program that contains passwords that a user typed dumps core
  – Passwords plainly visible in core dump
Improper Validation

• Inadequate checking of bounds, type, or other attributes or values
• Example: *fingerd*’s failure to check input length
Improper Indivisibility

• Interrupting operations that should be uninterruptable
  – Often: “interrupting atomic operations”

• Example: `mkdir` flaw (UNIX Version 7)
  – Created directories by executing privileged operation to create file node of type directory, then changed ownership to user
  – On loaded system, could change binding of name of directory to be that of password file after directory created but before change of ownership
  – Attacker can change administrator’s password
Improper Sequencing

- Required order of operations not enforced
- Example: one-time password scheme
  - System runs multiple copies of its server
  - Two users try to access same account
    - Server 1 reads password from file
    - Server 2 reads password from file
    - Both validate typed password, allow user to log in
    - Server 1 writes new password to file
    - Server 2 writes new password to file
  - Should have every read to file followed by a write, and vice versa; not two reads or two writes to file in a row
Improper Choice of Operand or Operation

• Calling inappropriate or erroneous instructions

• Example: cryptographic key generation software calling pseudorandom number generators that produce predictable sequences of numbers
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
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
Comparison and Analysis

- **Point of view**
  - If multiple processes involved in exploiting the flaw, how does that affect classification?
    - `xterm`, `fingerd` flaws depend on interaction of two processes (`xterm` and process to switch file objects; `fingerd` and its client)

- **Levels of abstraction**
  - How does flaw appear at different levels?
    - Levels are abstract, design, implementation, etc.
xterm and PA Classification

• Implementation level
  – xterm: improper change
  – attacker’s program: improper deallocation or deletion
  – operating system: improper indivisibility
xterm and PA Classification

- Consider higher level of abstraction, where directory is simply an object
  - create, delete files maps to writing; read file status, open file maps to reading
  - operating system: improper sequencing
    - During read, a write occurs, violating Bernstein conditions
- Consider even higher level of abstraction
  - attacker’s process: improper choice of initial protection domain
    - Should not be able to write to directory containing log file
    - Semantics of UNIX users require this at lower levels
xterm and RISOS Classification

• Implementation level
  – xterm: asynchronous validation/inadequate serialization
  – attacker’s process: exploitable logic error and violable prohibition/limit
  – operating system: inconsistent parameter validation
**xterm and RISOS Classification**

- Consider higher level of abstraction, where directory is simply an object (as before)
  - all: asynchronous validation/inadequate serialization
- Consider even higher level of abstraction
  - attacker’s process: inadequate identification/authentication/authorization
    - Directory with log file not protected adequately
    - Semantics of UNIX require this at lower levels
xterm and NRL Classification

- Time, location unambiguous
  - Time: during development
  - Location: Support: privileged utilities

- Genesis: ambiguous
  - If intentional:
    - Lowest level: inadvertent flaw of serialization/aliasing
  - If unintentional:
    - Lowest level: nonmalicious: other
  - At higher levels, parallels that of RISOS
xterm and Aslam’s Classification

• Implementation level
  – attacker’s process: object installed with incorrect permissions
    • attacker’s process can delete file
  – xterm: access rights validation error
    • xterm doesn’t properly validate file at time of access
  – operating system: improper or inadequate serialization error
    • deletion, creation should not have been interspersed with access, open
  – Note: in absence of explicit decision procedure, all could go into class race condition
The Point

- The schemes lead to ambiguity
  - Different researchers may classify the same vulnerability differently for the same classification scheme
- Not true for Aslam’s, but that misses connections between different classifications
  - *xterm* is race condition as well as others; Aslam does not show this
fingerd and PA Classification

- Implementation level
  - *fingerd*: improper validation
  - attacker’s process: improper choice of operand or operation
  - operating system: improper isolation of implementation detail
fingerd and PA Classification

- Consider higher level of abstraction, where storage space of return address is object
  - operating system: improper change
  - fingerd: improper validation
    - Because it doesn’t validate the type of instructions to be executed, mistaking data for valid ones

- Consider even higher level of abstraction, where security-related value in memory is changing and data executed that should not be executable
  - operating system: improper choice of initial protection domain
fingerd and RISOS Classification

• Implementation level
  – fingerd: incomplete parameter validation
  – attacker’s process: violable prohibition/limit
  – operating system: inadequate identification/authentication/authorization
fingerd and RISOS Classification

• Consider higher level of abstraction, where storage space of return address is object
  – operating system: asynchronous validation/inadequate serialization
  – *fingerd*: inadequate identification/authentication/authorization

• Consider even higher level of abstraction, where security-related value in memory is changing and data executed that should not be executable
  – operating system: inadequate identification/authentication/authorization
fingerd and NRL Classification

- Time, location unambiguous
  - Time: during development
  - Location: support: privileged utilities
- Genesis: ambiguous
  - Known to be inadvertent flaw
  - Parallels that of RISOS
fingerd and Aslam Classification

• Implementation level
  – *fingerd*: boundary condition error
  – attacker’s process: boundary condition error
  • operating system: environmental fault
    – If decision procedure not present, could also have been access rights validation errors
Theory of Penetration

• Goal: detect previously undetected flaws
• Based on two hypotheses:
  – Hypothesis of Penetration Patterns
  – Hypothesis of Penetration-Resistant Systems
• Idea: formulate principles consistent with these hypotheses and check system for inconsistencies
Hypothesis of Penetration Patterns

System flaws that cause a large class of penetration patterns can be identified in system (i.e., TCB) source code as incorrect/absent condition checks or integrated flows that violate the intentions of the system designers.

– Meaning: an appropriate set of design, implementation principles will prevent vulnerabilities
Hypothesis of Penetration-Resistant Systems

A system (i.e., TCB) is largely resistant to penetration if it adheres to a specific set of design properties.

Example properties:

- Users must not be able to tamper with system
- System must check all references to objects
- Global objects belonging to the system must be consistent with respect to both timing and storage
- Undesirable system and user dependencies must be eliminated
Flow-Based Model

- Focus on flow of control during parameter validation
- Consider `rmdir(fname)`
  - Allocates space for copy of parameter on stack
  - Copies parameter into allocated storage
- Control flows through 3 steps:
  - Allocation of storage
  - Binding of parameter with formal argument
  - Copying formal argument (parameter) to storage
- Problem: length of parameter not checked
Model

- System is sequence of states, transitions
- Abstract cell set $C = \{ c_i \}$
  - Set of system entities that hold information
- System function set $F = \{ f_i \}$
  - All system functions user may invoke
  - $Z \subseteq F$ contains those involving time delays
- System condition set $R = \{ r_i \}$
  - Set of all parameter checks
More Model

• Information flow set $IF = C \times C$
  – Set of all possible information flows between pairs of abstract cells
  – $(c_i, c_j)$ means information flows from $c_i$ to $c_j$

• Call relationship set $SF = F \times F$
  – Set of all possible information flows between pairs of system functions
  – $(f_i, f_j)$ means $f_i$ calls $f_j$ or $f_i$ returns to $f_j$

• These capture flow of information, control throughout system
System-Critical Functions

- Functions that analysts deem critical with respect to penetration
  - Functions that cause time delays, because they may allow window during which checked parameters are changed
  - Functions that can cause system crash
- System-critical function set $K$
- System entry points $E$
  - Gates through which user processes invoke system functions
rmdir

TCB entry point → rmdir(fname)

\[ \text{dststr} = \text{local buffer buf} \]

\[ \text{buf} \rightarrow \text{dststr} \]

\[ \text{srcstr} = \text{fname} \]

\[ \text{fname} \rightarrow \text{srcstr} \]

function call to strcpy

copy from \( \text{srcstr} \) to \( \text{dststr} \)
**rmdir and Model**

- **fname ∈ C**
  - Points to global entity
- **rmdir ∈ F, rmdir ∈ E**
  - System function and also entry point
- **fname cannot be illegal address**
  - \( \text{islegal}(\text{fname}) \in R \)
- **length of fname less than that of buf**
  - \( \text{length}(\text{fname}) < \text{spacefor}(\text{buf}) \in R \)
$rmdir$ and Model

- $strcpy \in K$
  - Because $strcpy$ does not check source, destination bounds
- $(fname, buf) \in IF$
  - Because information flows from $fname$ to $buf$
- $(rmdir, strcpy) \in SF$
  - Because $rmdir$ calls $strcpy$
More Model

• Alter set $AC = \{ (c_i, R_i) \}, R_i \subseteq R$
• View set $VC = \{ (c_i, R'_i) \}, R_i \subseteq R$
  – Set of abstract cells that can be altered/viewed and conditions that must be validated first
• $Element(c_i, R_i)$ predicate
  – Conditions in $R_i \subseteq R$ must be checked before $c_i$ viewed or altered
• Critical function set $KF = \{ (k_i, R''_i) \}, R_i \subseteq R$
• Entry point set $EF = \{ (e_i, R'''_i) \}, R_i \subseteq R$
  – Analogous to $AC$
More `rmdir`

- `strcpy` must validate `fname`’s address as legal before viewing `fname`
- `strcpy` must validate that size of `fname` is small enough to fit in `buf` before altering `buf`
- Hence:
  \[
  (\text{strcpy}, \text{islegal}(\text{fname}) \land \text{length}(\text{fname}) < \text{spacefor}(\text{buf})) \in KF
  \]
History of Transitions

- Altered cells set $ACS = \{ (c_i, e_i, pc_i) \}$
- Viewed cells set $VCS = \{ (c_i, e_i, pc_i') \}$
  - $c_i$ has been altered/viewed by invoking entry point $e_i$, and $pc_i, pc_i' \subseteq IF \cup SF \cup R$ sequence of information flows, function flows, conditions along path
- Critical functions invoked set $KCS = \{ (k_i, e_i, pc_i'') \}$
  - Like $ACS$, but $k_i$ has been invoked by invoking entry point $e_i$
- $(ACS, VCS, KCS)$ make up state of system
Penetration-Resistant State: Idea

- If the system function checks all conditions on the global variables to be altered or viewed, and all conditions on the system-critical functions, then system cannot be penetrated using a technique that exploits failure to check conditions
  - Need to check on entry
  - Need to check conditions on memory locations or system-critical functions
  - Need to check changes in previously checked parameters as result of time delay caused by a function
Penetration-Resistant State

State that meets the following requirements:

1. For all states \((c, e, p) \in ACS\):
   a) Conditions associated with \(e \in EF\) subset of conditions checked in \(p\)
   b) Conditions associated with cell \(c \in AC\) subset of conditions checked in \(p\)
   c) A subsequence of \(p\) contains the last element of \(p\), the conditions in part b, and does not contain any elements \((f, g) \in SF\) with \(f \in Z\) or \(g \in Z\)

2. Requirement 1, but for \(VCS\) rather than \(ACS\)

3. Requirement 1, but for \((k, e, p) \in KFS\) rather than \(ACS\)
State Transition Rules

- Control updating of information as system changes
- \( \tau \) state transition function
- \( \Sigma = (ACS, VCS, KCS) \)
- \( \tau(\Sigma) = \Sigma' = (ACS', VCS', KCS') \)
- Functions are \textit{alter\_cell}, \textit{view\_cell}, \textit{invoke\_crit\_func}
Altering Cells

- $alter_{\text{cell}}(c, e, p)$
  - Check:
    - $c \in C$, $e \in E$, $p \subseteq IF \cup SF \cup R$
    - Requirement 1 holds
  - If so:
    - $ACS' = ACS \cup \{(c, e, p)\}$
    - $VCS' = VCS$
    - $KCS' = KCS$
  - If not, new state is not penetration-resistant
Viewing Cells

• $\text{view\_cell}(c, e, p)$
  – Check:
    • $c \in C$, $e \in E$, $p \subseteq IF \cup SF \cup R$
    • Requirement 2 holds
  – If so:
    • $ACS' = ACS$
    • $VCS' = VCS \cup \{ (c, e, p) \}$
    • $KCS' = KCS$
  – If not, new state is not penetration-resistant
Invoking Critical Functions

- `invoke_crit_func(k, e, p)`
  - Check:
    - $k \in K$, $e \in E$, $p \subseteq IF \cup SF \cup R$
    - Requirement 3 holds
  - If so:
    - $ACS' = ACS$
    - $VCS' = VCS$
    - $KCS' = KCS \cup \{(k, e, p)\}$
  - If not, new state is not penetration-resistant
Penetration Resistance

- *Theorem*: Let the system be in a state that is penetration-resistant. Then if a state transition function is applied to the current state, the resulting state will also be penetration-resistant.
**rmdir Again**

- Assume system in penetration-resistant state
- `invoke_crit_func(strcpy, rmdir, p)`
- Requirement 3 must hold
  - No conditions associated with entry point `rmdir`, so 3a holds
  - Conditions for `strcpy` not checked within TCB, so `{ islegal(fname) ∧ length(fname)<spacefor(buf) } ∉ p`
  - Requirement 3 does not hold
- System no longer in penetration-resistant state
Automated Penetration Analysis Tool

- APA performed this testing automatically
  - *Primitive flow generator* reduces statements to Prolog facts recording needed information
  - *Information flow integrator, function flow integrator* integrate execution path derived from primitive flow statements
  - *Condition set consistency prover* analyzes conditions along execution path, reports inconsistencies
  - *Flaw decision module* determines whether conditions for each entry point correspond to penetration-resistant specs (applies Hypothesis of Penetration Patterns)
Questions

• Can this technique be generalized to types of flaws other than consistency checking?
• Can this theory be generalized to classify vulnerabilities?
Summary

• Classification schemes requirements
  – Decision procedure for classifying vulnerability
  – Each vulnerability should have unique classification

• Above schemes do not meet these criteria
  – Inconsistent among different levels of abstraction
  – Point of view affects classification
Key Points

• Given large numbers of non-secure systems in use now, unrealistic to expect less vulnerable systems to replace them

• Penetration studies are effective tests of systems provided the test goals are known and tests are structured well

• Vulnerability classification schemes aid in flaw generalization and hypothesis