Lecture 15
October 30, 2024

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

Standards

- Descriptive databases used to identify vulnerabilities and weaknesses
- Examples:
  - Common Vulnerabilities and Exposures (CVE)
  - Common Weaknesses and Exposures (CWE)

CVE

- Goal: create a standard identification catalogue for vulnerabilities
  - So different vendors can identify vulnerabilities by one common identifier
  - Created at MITRE Corp.
- Governance
  - CVE Board provides input on nature of specific vulnerabilities, determines whether 2 reported vulnerabilities overlap, and provides general direction and very high-level management
  - Numbering Authorities assign CVE numbers within a distinct scope, such as for a particular vendor
- CVE Numbers: CVE-*year-number*
  - *Number* begins at 1 each year, and is at least 4 digits

Structure of Entry

Main fields:

- CVE-ID: *CVE identifier*
- Description: *what is the vulnerability*
- References: *vendor and CERT security advisories*
- Date Entry Created: *year month day as a string of 8 digits*

Example: Buffer Overflow in GNU C Library

CVE-ID: CVE-2016-3706
Description: Stack-based buffer overflow in the getaddrinfo function in sysdeps/posix/getaddrinfo.c in the GNU C Library (aka glibc or libc6) allows remote attackers to cause a denial of service (crash) via vectors involving hostent conversion. NOTE: this vulnerability exists because of an incomplete fix for CVE-2013-4458
References:

- CONFIRM:https://sourceware.org/bugzilla/show_bug.cgi?id=20010
- CONFIRM:https://sourceware.org/git/gitweb.cgi?p=glibc.git;h=4ab2ab03d4351914ee53248dc5aef4a8c88ff8b9
- CONFIRM:http://www-01.ibm.com/support/docview.wss?uid=swg21995039
- CONFIRM:https://source.android.com/security/bulletin/2017-12-01
- SUSE:openSUSE-SU-2016:1527
- URL:http://lists.opensuse.org/opensuse-updates/2016-06/msg00030.html
- SUSE:openSUSE-SU-2016:1779
- URL:http://lists.opensuse.org/opensuse-updates/2016-07/msg00039.html
- BID:88440
- URL:http://www.securityfocus.com/bid/88440
- BID:102073
- URL:http://www.securityfocus.com/bid/102073

Assigning CNA: N/A
Date Entry Created: 20160330

CVE Use

- CVE database begun in 1999
  - Contains some vulnerabilities from before 1999
- Currently over 82,000 entries
- Used by over 150 organizations
  - Security vendors such as Symantec, Trend Micro, Tripwire
  - Software and system vendors such as Apple, Juniper Networks, Red Hat, IBM
  - Other groups such as CERT/CC, U.S. NIST, and internationally

CWE

- Database listing weaknesses underlying CVE vulnerabilities
  - Developed by CVE list developers, with help from NIST, vulnerabilities research community
- Organized as a list
  - Can also be viewed as a graph as some weaknesses are refinements of others
  - Not a tree as some nodes have multiple parents

Types of Entries

- *Category entry*: identifies set of entries with a characteristic of the current entry
- *Chain entry*: sequence of distinct weaknesses that can be linked together within software
  - One weakness can create necessary conditions to enable another weakness to be exploited
- *Compound element composite entry*: multiple weaknesses that must be present to enable an exploit
- *View entry*: view of the CWE database for particular weakness or set of weaknesses.
- *Weakness variant entry*: weakness described in terms of a particular technology or language
- *Weakness base entry*: more abstract description of weakness than a weakness variant entry, but in sufficient detail to lead to specific methods of detection and remediation
- *Weakness class*: describes weakness independently of any specific language or technology.

Examples

- CWE-631, Resource-Specific Weaknesses (a view entry)
  - Child: CWE-632, Weaknesses that Affect Files or Directories
  - Child: CWE-633, Weaknesses that Affect Memory
  - Child: CWE-634, Weaknesses that Affect System Processes
- CWE-680, Integer Overflow to Buffer Overflow (a chain entry)
  - Begins with integer overflow (CWE-190)
  - Leads to failure to restrict some operations to bounds of buffer (CWE-119)
- CWE-61, UNIX Symbolic Link (Symlink) Following (a composite entry)
  - Requires 5 weaknesses to be present before it can be exploited
  - CWE-362, CWE-340, CWE-216, CWE-386, CWE-732

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 $\Rightarrow$ 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

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

Versions

- These supply details the Flaw Hypothesis Methodology omits
- Information Systems Security Assessment Framework (ISSAF)
  - Developed by Open Information Systems Security Group
- Open Source Security Testing Methodology Manual (OSSTMM)
- Guide to Information Security Testing and Assessment (GISTA)
  - Developed by National Institute for Standards and Technology (NIST)
- Penetration Testing Execution Standard

ISSAF

- Three main steps
  - *Planning and Preparation Step*: sets up test, including legal, contractual bases for it; this includes establishing goals, limits of test
  - *Assessment Phase*: gather information, penetrate systems, find other flaws, compromise remote entities, maintain access, and cover tracks
  - *Reporting and Cleaning Up*: write report, purge system of all attack tools, detritus, any other artifacts used or created
- Strength: clear, intuitive structure guiding assessment
- Weakness: lack of emphasis on generalizing new vulnerabilities from existing ones

OSSTMM

- Scope is 3 classes
  - *COMSEC*: communications security class
  - *PHYSSEC*: physical security class
  - *SPECSEC*: spectrum security class
- Each class has 5 channels:
  - *Human channel*: human elements of communication
  - *Physical channel*: physical aspects of security for the class
  - *Wireless communications channel*: communications, signals, emanations occurring throughout electromagnetic spectrum
  - *Data networks channel*: all wired networks where interaction takes place over cables and wired network lines
  - *Telecommunication channel*: all telecommunication networks where interaction takes place over telephone or telephone-like networks

OSSTMM (con't)

- 17 modules to analyze each channel, divided into 4 phases
  - *Induction*: provides legal information, resulting technical restrictions
  - *Interaction*: test scope, relationships among its components
  - *Inquest*: testers uncover specific information about system
  - *Intervention*: tests specific targets, trying to compromise them

  These feed back into one another
- Strength: organization of resources, environmental considerations into classes, channels, modules, phases
- Weakness: lack of emphasis on generalizing new vulnerabilities from existing ones

GISTA

- GISTA has 4 phases:
  - *Planning*, in which testers, management agree on rules, goals
  - *Discovery*, in which testers search system to gather information (especially identifying and examining targets) and hypothesizing vulnerabilities
  - *Attack*, in which testers see whether hypotheses can be exploited; any information learned fed back to discovery phase for more hypothesizing
  - *Reporting*, done in parallel with other phases, in which testers create a report describing what was found and how to mitigate the problems
- Strength: feedback between discovery and attack phases
- Weakness: quite generic, does not provide same discipline of guidance as others

PTES

- 7 phases
  - *Pre-engagement interaction*: testers, clients agree on scope of test, terms, goals
  - *Intelligence gathering*: testers identify potential targets by examining system, public information
  - *Thread modeling*: testers analyze threats, hypothesize vulnerabilities
  - *Vulnerability analysis*: testers determine which of hypothesized vulnerabilities exist
  - *Exploitation*: testers determine whether identified vulnerabilities can be exploited (using social engineering as well as technical means)
  - *Post-exploitation*: analyze effects of successful exploitations; try to conceal exploitations
  - *Reporting*: document actions, results
- Strengths: detailed description of methodology
- Weakness: lack of emphasis on generalizing new vulnerabilities from existing ones

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

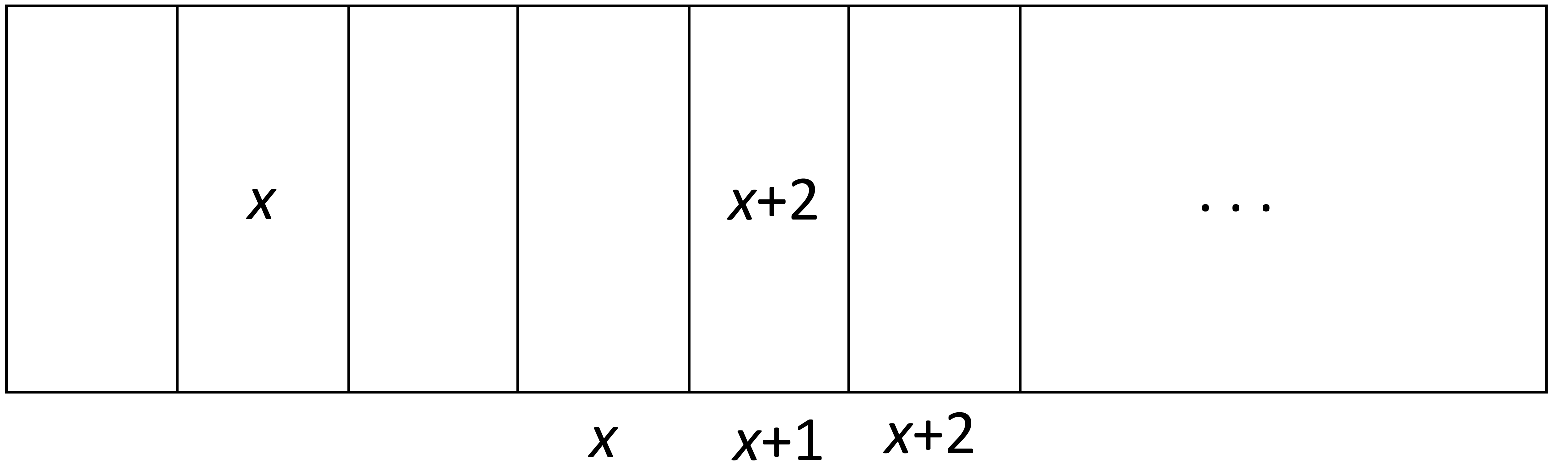
- Learn details of system's control flow and supervisor
  - When program ran, memory split into segments
  - 0-4: supervisor, system programs, system state
    - Protected by hardware mechanisms
  - 5: system work area, process-specific information including privilege level
    - Process should not be 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

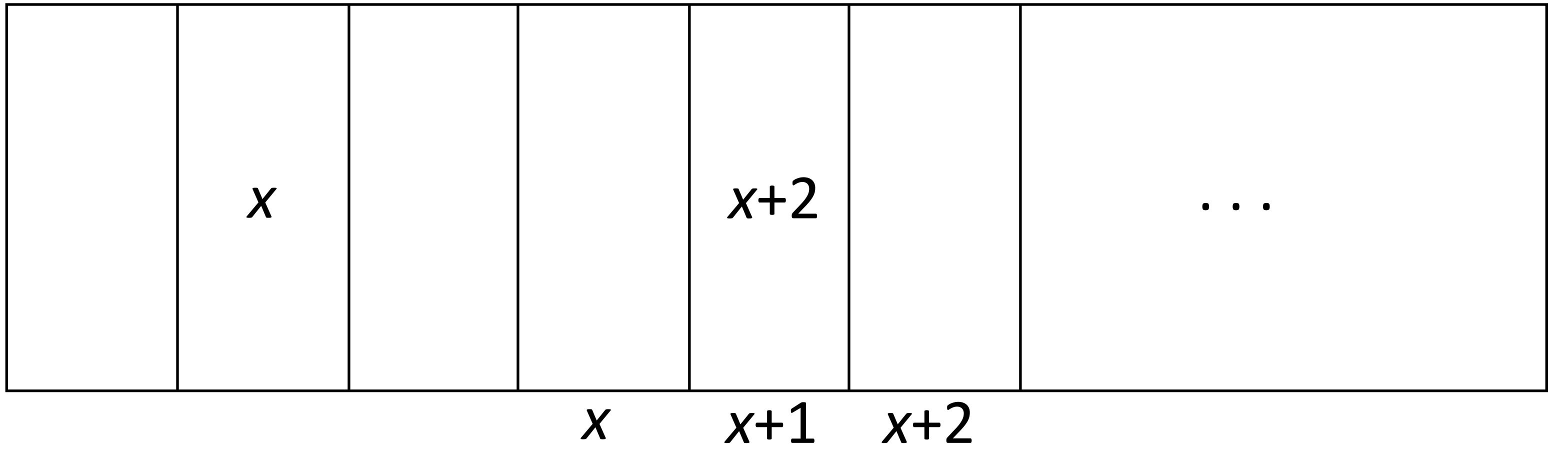
How to Make a Supervisor Call

- 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



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



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 4: 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 5: 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

Step 5: Flaw Generalization

- Other human (social engineering) methods would get more information
- Problem here is human
  - Inadequate training
  - Inadequate validation of claims to be in the company
  - Not checking where information is sent
  - Not checking where information is came from

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