Lecture 18: More Assurance

- Reviews of assurance evidence
- Security testing
- Penetration testing

Reviews of Assurance Evidence

- Reviewers given guidelines for review
- Other roles:
  - Scribe: takes notes
  - Moderator: controls review process
  - Reviewer: examines assurance evidence
  - Author: author of assurance evidence
  - Observer: observe process silently
- Important: managers may *only* be reviewers, and only then if their technical expertise warrants it

Setting Review Up

- Moderator manages review process
  - If not ready, moderator and author's manager discuss how to make it ready with author
  - May split it up into several reviews
  - Chooses team, defines ground rules
- Technical Review
  - Reviewers follow rules, commenting on any issues they uncover
    - May request moderator to stop review, send back to author
  - General and specific comments to author

Review Meeting

- Moderator is master of ceremonies
  - Grammatical issues presented first
  - General and specific comments next
  - Goal is to collect comments on entity, *not* to resolve differences
  - Scribes write down comments and who made it (anyone can see it, help scribe, verify comment made)

Conflict Resolution

- After meeting, scribe creates Master Comment List
  - Reviewers mark “Agree” or “Challenge”
  - All comments that everyone “Agree”s are put on Official Comment List
  - Rest must be resolved by reviewers
- Moderator, reviewers then:
  - Accept as is
  - Accept with changes on OCL
  - Reject

Conflict Resolution

- Author takes OCL, makes changes as sees fit
- Author then meets with reviewers
  - Explains how each comment made by reviewer was handled
  - All must be resolved to satisfaction of author, reviewer
- Review completed

Implementation Assurance

Considerations that support assurance

- Modular, with minimum of well-defined interfaces
  - Remove non-security functionality from modules enforcing security functionality
- Good choice of programming language
  - Especially those providing built-in features to help avoid common problems
- Follow good coding standards

Implementation Management

- *Configuration management*: control of changes made throughout development, operational life cycle
  - Hardware, software, firmware
  - Documentation, test documentation
  - Testing, test fixtures

Tools and Processes

- Version control and tracking
  - Enable rolling back to earlier versions, comparison of changes among versions
- Change authorization
  - Prevent conflicts, ensure specific people check things in
- Integration procedures
  - Define steps to select appropriate versions to generate system
- Tools for product generation
  - Generate system from proper versions provided by integration procedures

Justification

- How do you show implementation meets design?
  - Code reviews
  - Requirements tracing
  - Informal correspondence
  - Security testing
  - Formal proof techniques

Security Testing

- Functional testing: tests how well an entity meets its specification
  - Called *black box testing*
- Structural testing: tests based on analysis of code in order to develop test cases
  - Called *white box testing*

Components

3 components to security testing

- Security functional testing
  – Functional testing specific to security issues described in relevant specification
- Security structural testing
  – Structural testing specific to security implementation found in relevant code
- Security requirements testing
  – Security functional testing specific to security requirements found in requirements specification

When Testing Occurs

- Unit testing
  - Testing on code module before integration
  - Done by developer
- System testing
  - Functional testing of integrated modules
  - Done by integration team
- Third-party testing (independent testing)
  - Testing performed by a group outside development organization
- Security Testing
  - Testing addressing the product security

Security Functional Testing

- Differs from ordinary functional testing
  - Ordinary functional testing focuses on most commonly used functions
  - Security functional testing focuses on functions that invoke security mechanisms
    - Especially the *least* used aspects

Test Coverage

- Describes how completely entity has been tested against its functional specification
  - Security testing needs broader coverage
  - Completed test coverage analysis provides evidence that external interfaces have been tested
  - Interim test coverage analysis shows what else needs to be tested

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

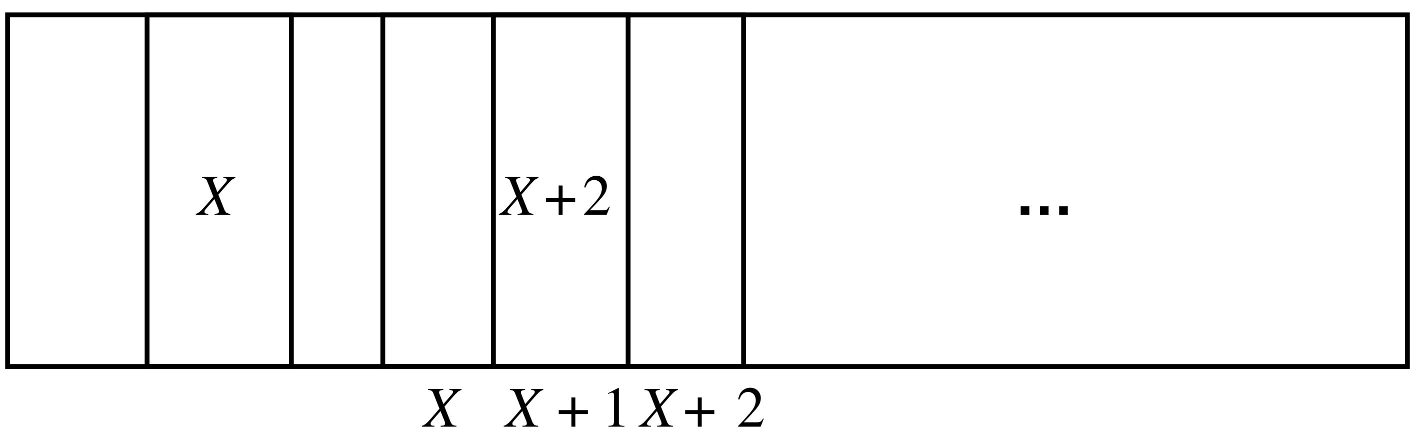
- Learn details of system's control flow and supervisor
  - When program ran, memory split into segments
  - 0-4: supervisor, system programs, system state
    - Protected by hardware mechanisms
  - 5: system work area, process-specific information including privilege level
    - Process should not be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

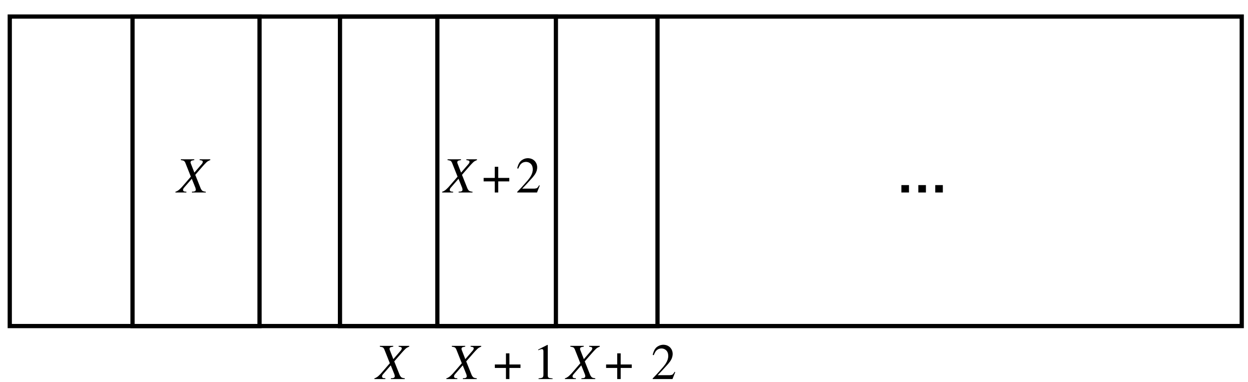
How to Make a Supervisor Call

- 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



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 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

```
ftp              21/tcp File Transfer
telnet           23/tcp Telnet
smtp             25/tcp Simple Mail Transfer
finger            79/tcp Finger
sunrpc          111/tcp SUN Remote Procedure Call
exec            512/tcp remote process execution (rexecd)
login           513/tcp remote login (rlogind)
shell           514/tcp rlogin style exec (rshd)
printer         515/tcp spooler (lpd)
uucp            540/tcp uucpd
nfs            2049/tcp networked file system
xterm          6000/tcp x-windows server
```

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 *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

```
qotd             17/tcp  Quote of the Day
ftp              21/tcp  File Transfer [Control]
loc-srv         135/tcp  Location Service
netbios-ssn     139/tcp  NETBIOS Session Service [JBP]
```

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