INF523: Computer System Assurance

Structured Design

Prof. Clifford Neuman

Lecture 4
18 Sep 2020

Announcements

September 25th – Proposal Presentations

- A 7 minute presentation on the 25th of September
  - The presentation should identify the system or class of system which has already been approved by me.
  - Explain (though not necessarily answer) the assurance issues that need to be met by the identified system.
  - Identify the consequences of security failure in such systems.
  - Discuss where one will look to answer those questions.
  - Ask for discussion from other students regarding the questions they might have for you in your final presentation.

By Thursday 24 September 6PM

Send me your PPT slides, which I will assemble into the slide deck we will use.

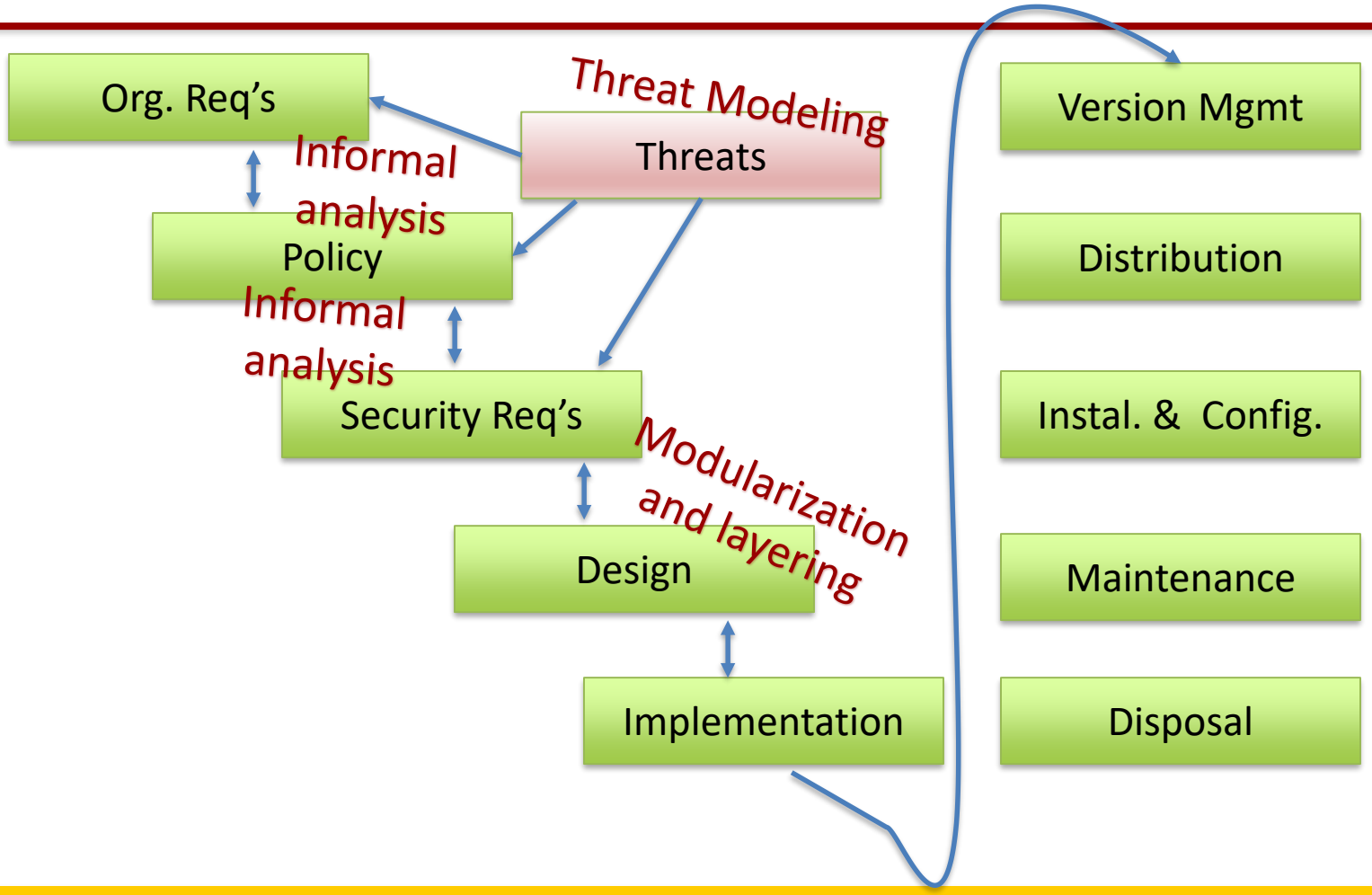
If you are a DEN student participating live, I will move through your slides as you describe them on webex.

If you cannot participate live, I can present, but only based on material specifically on the slides.

Reading for this week

- D.L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules,* 1972
- Daniel Hoffman, *On Criteria for Module Interfaces*, 1990
- Paul Karger, et. al., *A VMM security kernel for the VAX architecture*, 1990 – Section 3.7
- Final Evaluation Report, Gemini Trusted Network Processor, 1995 – Section 4.2

"Assurance Waterfall"



Structured Design

- Essential for high assurance
- Modularization and Layering
- Isolate protection-critical components; always invoke
- Minimize complexity
  - May require minimizing the number of types of objects the system supports
- What if you don't?
  - System will be vulnerable to outside malware attacks
  - More likely to contain residual errors in design and implementation (or even malicious software)

Q: why can malware disable AV?

Software Design Principles

- From IEEE *Guide to the Software Engineering Body of Knowledge*, Version 3.0

- Decomposition and Modularization
- Coupling and Cohesion
- Abstraction
- Separation of interface and implementation
- Encapsulation and Information hiding
- Sufficiency, completeness, and primitiveness

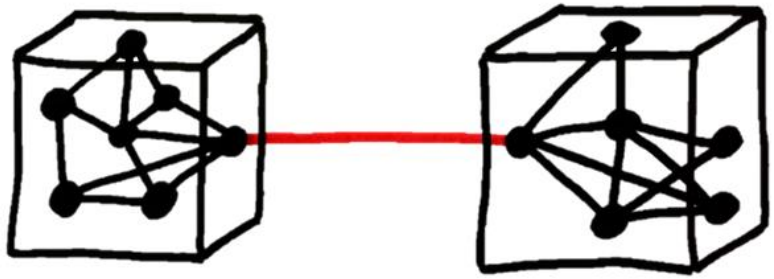
Decomposition and Modularization

- Divide large system into smaller components
  - Each has well-defined interface
- Goal is to divide by functions and responsibilities
- Modularization is good
  - Manage complexity by using smaller parts
  - System is easier to
    - Understand
    - Develop (e.g., by a team in paral
    - Test
    - Maintain

Coupling and Cohesion

- Ideas developed by Larry Constantine in late 1960s
- Coupling is measure of interdependence among modules
  - Amount of shared infrastructure
  - Amount of coordination
  - Amount of information flow
- Cohesion is measure of the degree to which elements of a module belong together
- Want low coupling and high cohesion

Low Coupling; High Cohesion



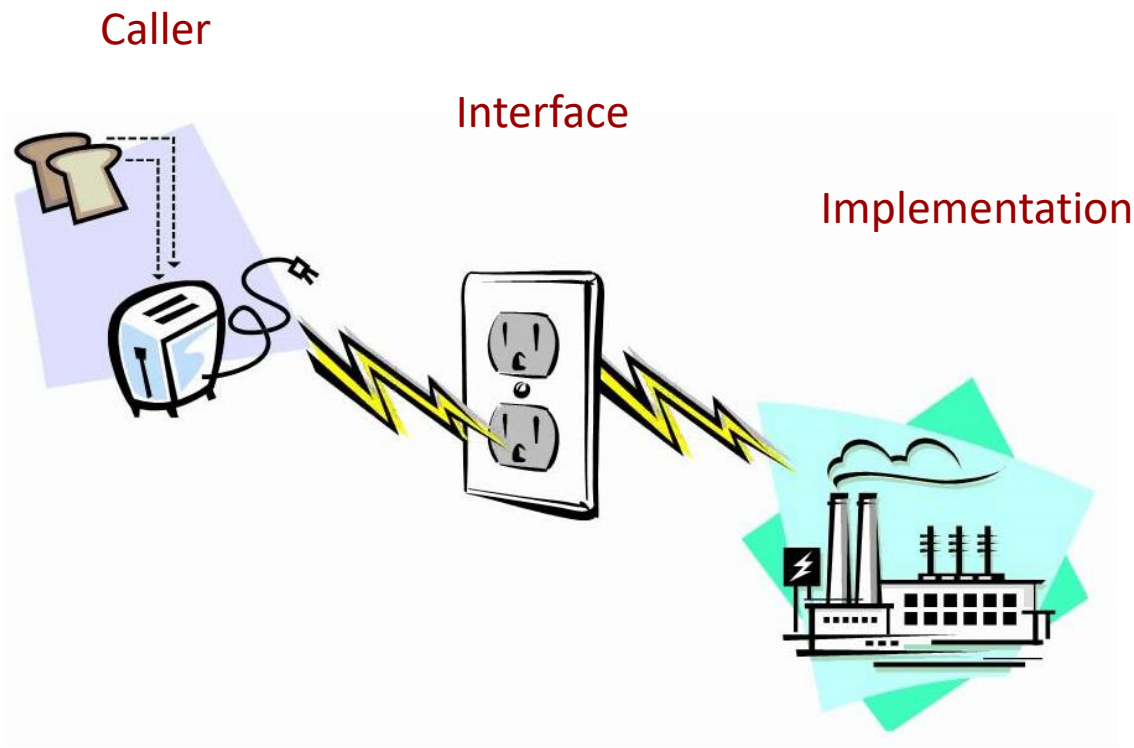
Abstraction

- View of an object
  - focuses only on relevant information
  - ignores the rest
- *Parameterization* abstracts details of data representation by using names
- *Specification abstraction* hides details of algorithm, data storage, and control by focusing on effects/results
- Want to increase abstraction at interface of modules

Separate Interface from Implementation

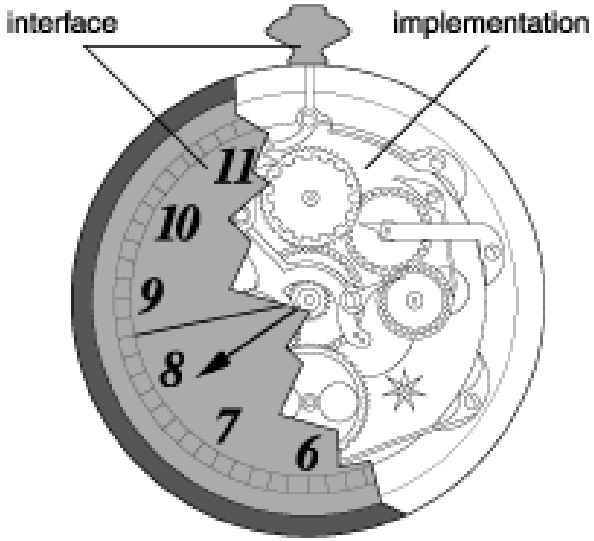
- Define module by specifying public interface
  - Parameters
  - Effects
  - Results
  - Exceptions
- Separate from details of how module is implemented
  - Algorithm
  - Data structures and storage
  - Control flow

Interface vs. Implementation



Encapsulation and Information Hiding

- Grouping and packaging internal details of modules
- Creating an abstraction
- Make the internal details inaccessible from outside



Sufficiency, Completeness, and Primitiveness

- *Sufficiency*: Module has enough characteristics of abstraction to be useful
- *Completeness*: Module implements entirety of abstraction
  - Otherwise, feature likely in some other module and high coupling results
- *Primitiveness*: Operations require access to underlying representation
  - Want "building blocks" that can be combined into higher-level patterns

Software “Architecture”

- *Software Architecture in Practice (2nd edition)*, by Bass, Clements, and Kazman : Architecture is
  - The structure or structures of the system, which comprise
    - *software elements*,
    - the *externally visible properties of those elements*, and
    - the *relationships among them*.
  - Architecture is concerned with the public side of interfaces
    - Not details having to do solely with internal implementation

Architectural Considerations

- Division of functions
  - Modules
  - Information hiding
- Distribution of functions (in processes or systems)
  - Concurrency - Modules run in parallel
    - Synchronization is an issue
  - May be driven by underlying system architecture
- Dependencies
  - Which modules need to call which modules
- Interfaces (externally visible properties of elements)
  - Data types and methods; effects

Strategy for Modularization

- Manage complexity by using smaller parts
- But many possible ways of modularizing a system
- Some ways are better than others
  - … which leads us to Parnas' paper

KWIK Index Production System

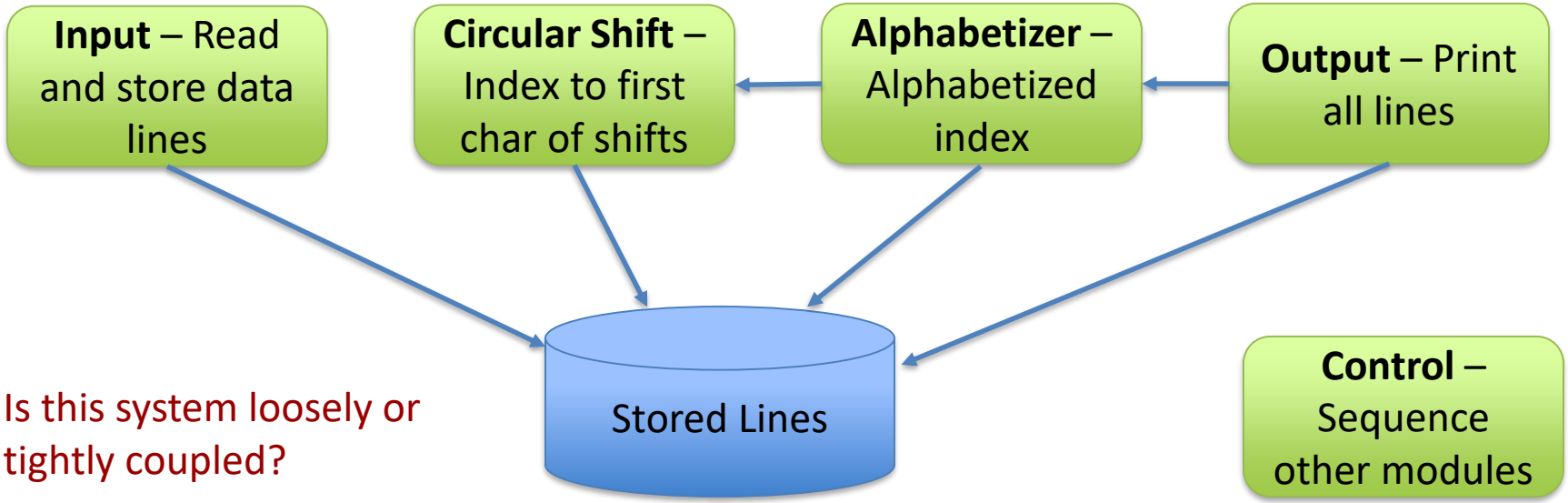
- Accepts ordered set of lines
- Each line contains ordered set of words
- Each word contains ordered set of characters
- Lines circularly-shifted
  - Move first word to end making a new line
  - Do for all words in each line
- Outputs a listing of all circular shifts of all lines in alphabetical order

KWIK Example

- Original lines: BCF, ADE →
  - ADE (ambiguous if 0 shifts included)
  - BCF
  - CFB
  - DEA
  - EAD
  - FBC

Functional Modularization

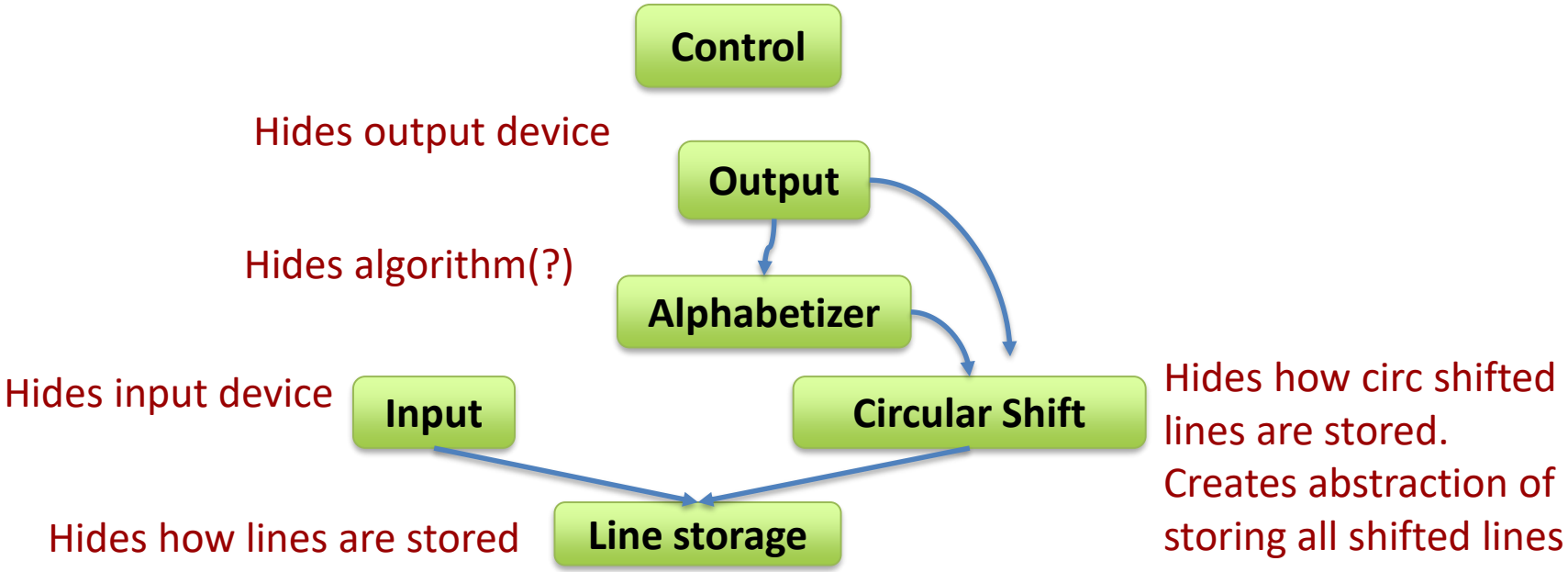
- System modeled as data flow, flow chart
- Each module implements one function in flow
- E.g., first modularization in Parnas' paper



Is this system loosely or tightly coupled?

“Information Hiding” Modularization

- System modeled on hiding implementation decisions
- Each module hides “a secret”



Information Hiding and Abstractions

- Module creates abstraction
- Examples:
  - Abstract data types: Users operate on the data without knowing its representation
  - GUI creation environments: Users construct GUIs without knowing details of how to display
    - E.g., X-Windows, MS VB
  - Protocols: Users send and receive data without knowing details of channel operation
  - Methods: Users invoke methods without knowing class's algorithms

An Advantage of Information Hiding

- Can metaphorically “lift” the interface and slide a new implementation under it
  - Take advantage of new technology but disrupt only one module
- Choose modules based on design decisions that are likely to change
  - Make that the hidden “secret”

“Secrets” and Changes

Secret	Typical Change
How to monitor a sensor	New type (more reliable, higher resolution, etc.) of sensor
How to control a device	New type (faster, larger, etc.) of device
Platform characteristics	New processor, multiprocessor, more memory, different chipset
How to control a display	Reorganization of user interface
How to exchange data	Protocol change
Database physical structure	Fields added or changed, optimized storage
Algorithm	Different time-space tradeoff, greater accuracy

Courtesy David Weiss, Iowa State Uni.

Differences between Approaches

- Different in way work is divided
- Different in interfaces
- Different in maintainability when changes made
  - If method or format of line storage changes
    - Approach 1: EVERY module must change
    - Approach 2: Only “line storage” module must change
  - If change method or timing of circular shift
    - Approach 1: Circular shift and Alphabetizer modules change
    - Approach 2: Only Circular shift module must change
  - If change method or timing of alphabetization
    - Approach 1: Alphabetizer and output modules change
    - Approach 2: Only Alphabetizer module must change

Reasonable Conclusion

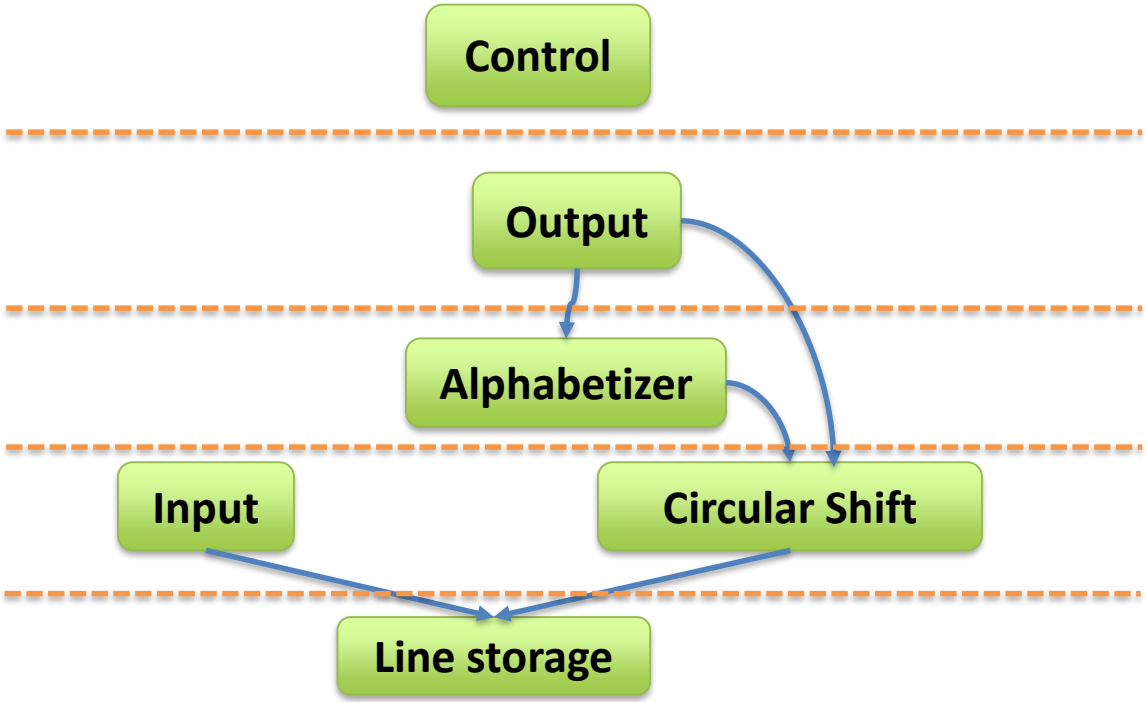
- Decompose a system into modules using information hiding
  - General criterion: each module hides design decision from rest of system (esp. likely-to-change decision)
- DON'T decompose by function
  - E.g., using a flow chart or DFD
- But don't over-modularize!
  - E.g., if need same data structures, put in same module

Parnas' Specific Criteria

- Data structure and accessing/modifying procedures
- Sequence of preparation steps to call routine and routine itself
- Control blocks
  - e.g., structure of actual records in a queue; Process Control Block in OS
- Character codes, alphabetic orderings, etc.
- Sequencing – the order things are done

Layering

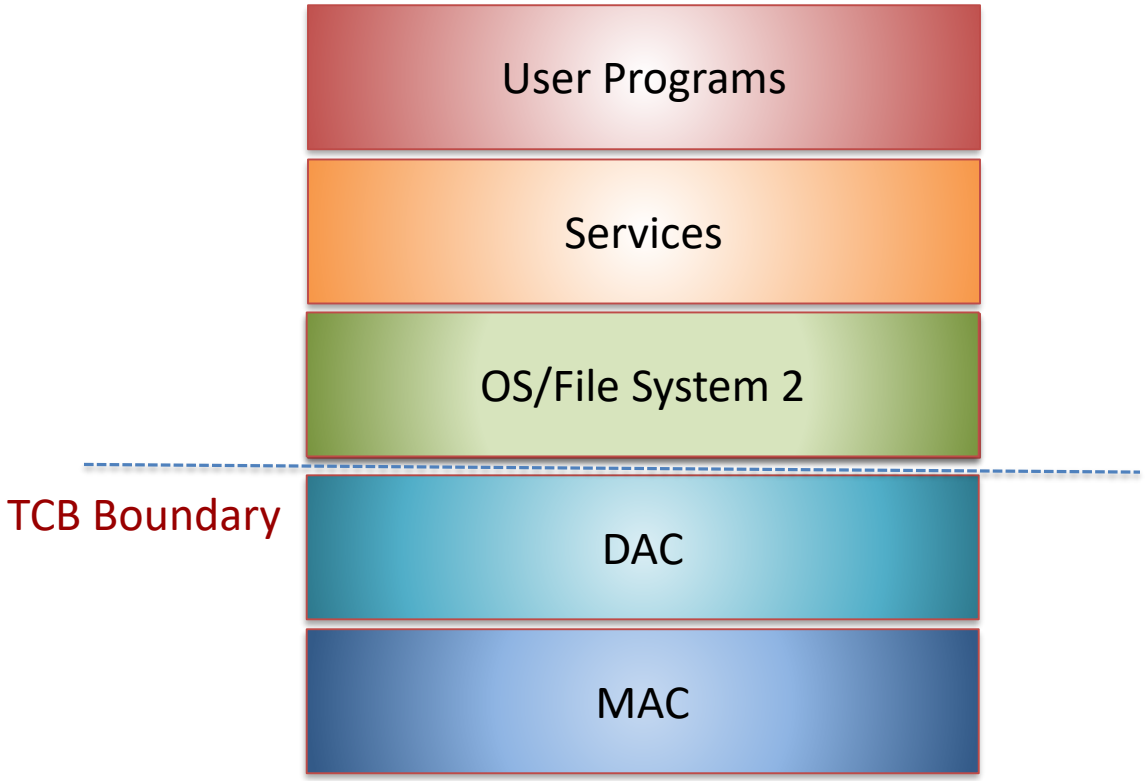
- Second modularization method (by hiding) is layered



Hierarchical Structure

- Layering – Hierarchy of layers
- Partial ordering of “uses” or “depends on”
- Lower layers provide abstract machines/data types
  - E.g., Line storage provides abstract original lines
  - Circular shifter provides abstraction of all shifted lines
- Permits simplification of higher layers
- Lower layers provide usable basis for new system
- But note: Hierarchical layers and good modularization are independent properties
  - Information hiding does not guarantee layering

Example of Layering



Layering in an OS

- External device characteristics
  - E.g., keyboard/display I/O
- External system characteristics
  - E.g., network communications protocols
- Resource allocation
  - E.g., process and thread management
  - Process scheduling
  - Memory management
- Janson developed idea of levels of abstraction in security kernel design (MIT Dissertation, 1976)
  - Used in VAX security kernel and GTNP
  - Each layer implements abstraction in part by calling lower layers

VAX Security Kernel

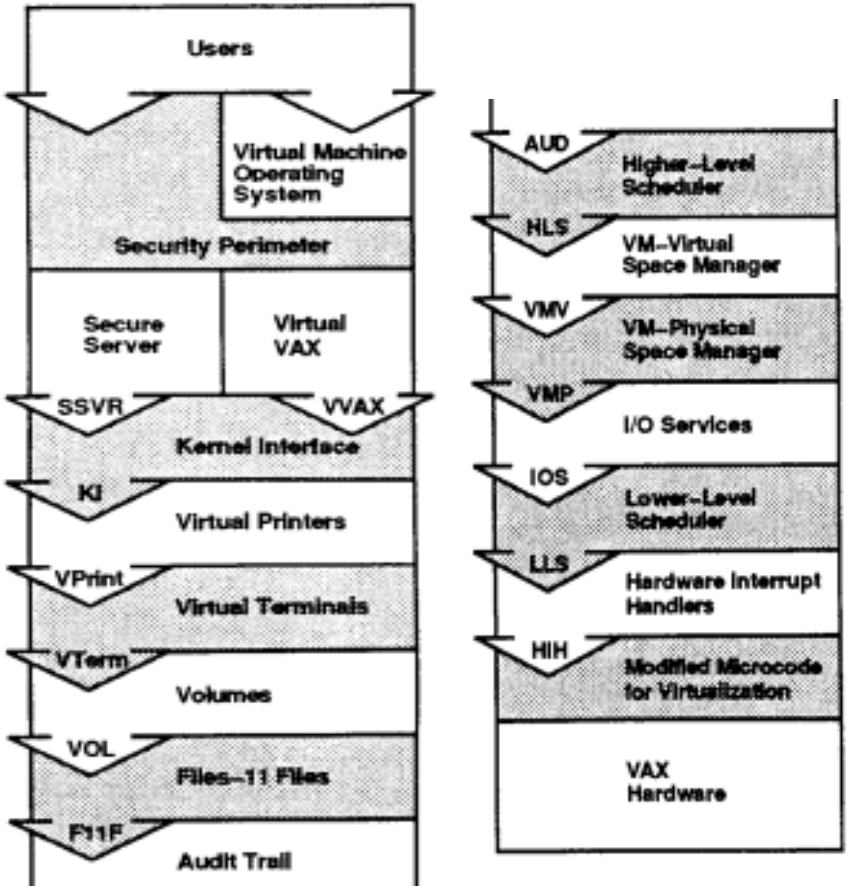
- VMM that runs on VAX processors
- Creates isolated virtual VAX processors
  - VMs run VMS or Ultrix (Unix variant) OSes
- Security labels (simplified):
  - Subjects: VMs – Each has an access class
  - Objects: Virtual disks – Access classes and ACLs
- Two-layer scheduler for performance (Reed, MIT):
  - Level 1: Small set of processes, per-process DBs all in memory
  - Level 2: User processes, require bringing per-process DBs from disk to load in Level 1 process

Example OS layering: VAX Security Kernel

Note two-layer scheduler:

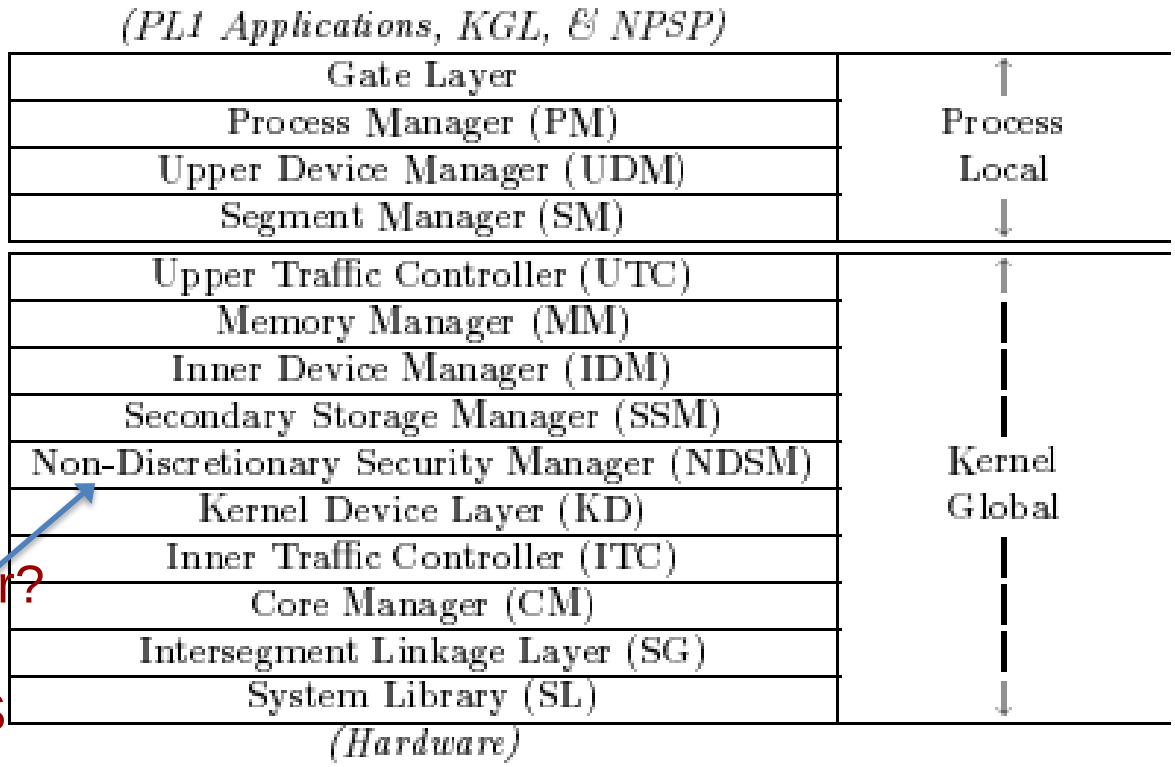
LLS – Assigns layer 1 virtual CPUs (vp1s) to physical CPUs – some vp1s reserved for kernel processes

HLS – Schedules layer 2 virtual CPUs (vp2s) to vp1s – some vp2s used for VAX VMs



Example OS Layering: GEMSOS

- Security kernel of GTNP
- Note 2-level scheduling
- ITC provides VM abstraction
- UTC schedules processes on abstract VMs
- UTC doesn't know when VM blocks
- Where is DAC layer?
- Right! DAC outside Ring 0 in GEMSOS



Rings

- Hardware rings enforce layering
- Calls only to lower layers
- Calls only to restricted entry points
- Interface must be carefully specified
- Entry points must be carefully code
  - E.g., to sanitize/filter/normalize inputs
  - E.g., to handle violations of interface specification

Module Interfaces

- Public part of module
- Hides details: data structures, control sequences, etc.
- Can metaphorically “lift” the interface and put a new implementation under it
- Treat module as “black box”

Module Interface Specification

- Module interface specification defines
  - Entry points
  - Syntax
  - Parameters
  - Data types
  - Constants
  - Exceptions
  - Semantics of state change
    - When is access call legal?
    - What effect on other calls?
- Makes all assumptions explicit

Ideal Interface Specs

- Written before implementation, not after!
- Easy to read and maintain
- Describe how to verify (test) behavior of module
  - Module must conform to spec
  - Spec says exact effects
  - Module can do that, and only that

Some Benefits

- Supports partitioning of system into modules
- Defines expected behavior of module
- Permits parallel development
- Gives verification requirements
  - Test requirements
  - Acceptance criteria
- Helps find errors

Example Interface Specification

AUDIBLE SIGNAL (1)

1. Introduction

The Audible Signal device generates a tone that can be heard within the Aircraft cockpit. The device has three states: on steady, on intermittently (beep), or off.

2. Interface Overview

2.1 Access Program Table

Function name	Parm type	Parm info	Exceptions
G/S_AUDIBLE_SIGNAL	p1: ind_cntrl; O	!Aud Signal!	None
S_BEEP_RATE	p1: integer; I	rate (beeps/sec)	

2.2 Events Signalled: None

3. Services Provided

1. Signal. Provide users with a signal, controlled by the software, that can be heard by the pilot. This signal can be turned on steady, off, or it can alternate between on and off at a variable frequency. An example of this type of signal is a tone or whistle.

4. Interface Design Issues: None

5. Local Types

Type Name	Type Definition
Ind_cntrl	Enumerated: On, Off, $Intermittent$

6. Local Dictionary

!Aud Signal!	The current state of the audible signal.

7. Exception Dictionary: None

Example Interface Specification (cont.)

AUDIBLE SIGNAL (2)

8. **Access Program Effects:**

S_ AUDIBLE_SIGNAL	if p1 = On then audible signal turned on; if p1 = Off then audible signal turned off; if p1 = $Intermittent$ then audible signal alternated between on and off at rate specified by S_BEEP_RATE
S_BEEP_RATE	Sets the rate in beeps/second that the tone beeps when S_AUDIBLE_SIGNAL is called with p1 = $Intermittent$

9. **Configuration Parameters**

Beep_rate_default	The default beep rate for the audible signal in beeps per second. Range: .1 <= Beep_rate-default <= 1 Normal value: .5 beeps per second

10. **Information Hidden**

1. The value encoding of the data word to the device.
2. The method used to cause the signal to beep on and off.

11. **Implementation Notes**: None.

Interface Criteria

- Consistent
  - Naming conventions, parameter passing, exception
  - People tend to skip details that look familiar, so inconsistencies will cause problems
- Essential
  - Omit needless features
  - Don’t duplicate functions
- General
  - Support usage for many purposes
- Minimal (primitive)
  - If independent features, consider using independent calls
- Opaque
  - Apply information hiding
  - Interface should be unlikely to change when implementation does

“Bad” Example

Program name	Inputs	Outputs	Exceptions
s_push	integer		overflow
sg_pop		integer	empty

- Stack module interface
- Sg_pop sets and gets at same time
- How to examine top element w/o changing stack?
- Violation and how to fix?
- Minimality – separate into s_pop and g_top

Another "Bad" Example

Program name	Inputs	Outputs	Exceptions
sg_getc		character	
s_ungetc	character		overflow

- Character input module interface
- Sg_getc removes next char from input and returns its value
- Want to check if end of token
- If not end, must use s_ungetc to put char back
- Violation and how to fix?
- Minimality – separate into s_next and g_cur
  - S_ungetc no longer needed

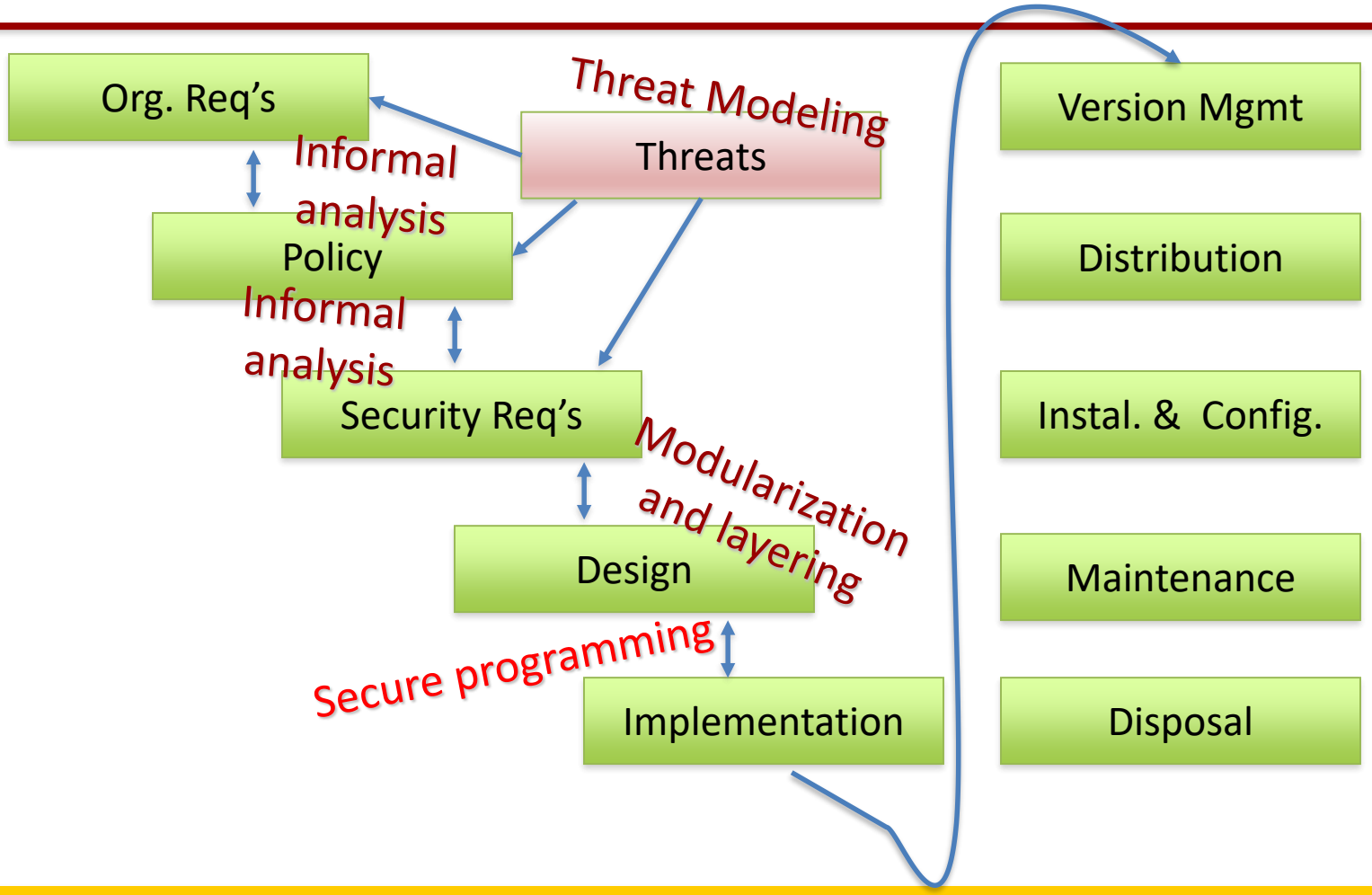
Good Example of Non-minimality

- Unix malloc call
- Allocates space and returns pointer
- Always want to do both together
- No need for separate “set” and “get” calls

Tradeoffs

- Generality increases number of entry points
  - Consistency of calls for possibly unused functions
- Minimality increases number of entry points and number of calls to them
  - Primitives tend to be called more often
- Sometimes opacity must be violated due to implementation concerns
  - Example of initialization call to create matrix representation

"Assurance Waterfall"



INF523: Computer System Assurance

Secure Programming

Prof. Clifford Neuman

Lecture 5
2 October 2020

Today's Outline

- What is Secure Programming?
  - Common software weaknesses
  - Secure coding practices
- "Secure Languages"
- Bug Tracking

Secure Programming

- Practice of developing software in a way that helps prevent vulnerabilities
  - Actually a set of practices
- Sometimes called “defensive programming”
- Characterized by having few assumptions about inputs or the environment

IEEE Top 10 Security Design Flaws

- Attempt to “shift focus from finding bugs to identifying common design flaws”
- 2014 workshop participants discussed types of flaws
- Ad hoc (like most software development)
  - No claim about completeness, for example
  - An opinion poll; no underlying engineering or theory
  - Unknowingly repeats RM principles in unstructured way
- But nevertheless applicable to high-assurance
- Illuminating (and distressing) point:
  - *Many of the flaws that made the list have been well known for decades, but continue to persist*

IEEE Top 10 Security Design Flaws (1)

1. **Earn or give, but never assume, trust**

- Don't trust clients to behave in particular way
- Don't expect data sent to a client to be protected
- Assume data sent by untrusted clients compromised until proven otherwise
- Properly validate all data received from client before processing
- *Integrity issue*
  - Consider OS calls or ring crossings
  - Source of many web application vulnerabilities

IEEE Top 10 Security Design Flaws (2)

2. Use an authentication mechanism that cannot be bypassed or tampered with

- *Non-bypassable*, hmmm… ring any bells?
- Unlike RM, conjoins identity and access control
- For users and other machines
- Require authentication; don't assume identity
- Use unforgeable credentials
- Protect credentials from theft
- Limit lifetime of session
- Use a single, well-proven mechanism/framework

IEEE Top 10 Security Design Flaws (3)

3. Authorize after you authenticate

- Authorization may change (e.g., revocation)
- Authorization may depend on context (e.g., time)
- Identity alone is insufficient to determine authorization

IEEE Top 10 Security Design Flaws (4)

4. **Strictly separate data and control instructions, and never process control instructions received from untrusted sources**

- I.e., don't trust input from untrusted sources
- Could lead to injection attacks
- Assembling insufficiently validated, untrusted data with trusted control instructions
  - E.g., shellshock Bash vulnerability
  - E.g., SQL injection
  - E.g., cross-site scripting

IEEE Top 10 Security Design Flaws (5)

5. Define an approach that ensures all data are explicitly validated

- Don't make assumptions about data
- Attackers may subvert and invalidate assumptions
  - Injection, bypass, memory corruption, resource exhaustion
- Examine, sanitize, filter before processing
- Design software so security reviewer can *verify correctness* and comprehensiveness of data validation (ring any bells?)
  - Use common, standard validation mechanism
  - Transform data to canonical form
- Re-do checks at module boundaries
  - State requirements in interface description

IEEE Top 10 Security Design Flaws (6)

6. Use cryptography correctly

- Use standard, vetted algorithms, libraries, and frameworks
- Don’t roll you own
- Protect the keys
  - Don’t hard-code embed them in a program
  - Permit revocation and rotation
  - Use strong keys
  - Use strong distribution mechanisms
- Design to allow for use of new algorithms (ring a bell?)

IEEE Top 10 Security Design Flaws (7)

7. **Identify sensitive data and how they should be handled**

- *This sure sounds like a need for a policy, doesn't it?*
- Consider laws, regulations, company policy, contractual obligations (e.g., NDAs), and user expectation
- Protect data at rest and in transit when designing controls

IEEE Top 10 Security Design Flaws (8)

8. Always consider the users

- Some users are sophisticated and interested in using a system securely, but most are not
- Security is not an add-on; it is a property emerging from how the system is built and operated
- Make controls easy to deploy, configure, use, and update
- Also consider needs of programmers
- This is such a sprawling “flaw” that the recommendations are sometimes contradictory and overwhelming

IEEE Top 10 Security Design Flaws (9)

9. Understand how integrating external components changes your attack surface

- E.g., off-the-shelf or open source libraries
- Reusing components means inheriting their security weaknesses
- Attempt to isolate external components
- Validate provenance (integrity) of components
- Authenticate data-flow, validate inputs

IEEE Top 10 Security Design Flaws (10)

10. Be flexible when considering future changes to objects and actors

- Design for change
  - *I.e., modularize, hide secrets, layer*
- Design for secure updates
- Design so security components can be easily updated
  - E.g., keys, passwords

Common Weakness Enumeration

- The CWE is a “dictionary” of common software security flaws
  - http://cwe.mitre.org/
  - Approximately 1000 currently
  - Many are clusters of similar or related weaknesses
    - E.g., more than 30 related to “Path Traversal” or “Path Equivalence”

CWE Listings

- CWE entries are multi-page listings that consist of
  - Description
  - Consequences
  - Likelihood of exploit
  - Detection methods
  - Short examples
  - Observed examples in the CVE database
  - Potential mitigations
  - Relationships to other weaknesses
  - Research notes
  - Attack patterns that exploit the weakness
  - References

CWE Example (just the description)

CWE-116: Improper Encoding or Escaping of Output

Weakness ID: 116 *(Weakness Class)* **Status:** Draft

Description

Summary

The software prepares a structured message for communication with another component, but encoding or escaping of the data is either missing or done incorrectly. As a result, the intended structure of the message is not preserved.

Extended Description

Improper encoding or escaping can allow attackers to change the commands that are sent to another component, inserting malicious commands instead.

Most software follows a certain protocol that uses structured messages for communication between components, such as queries or commands. These structured messages can contain raw data interspersed with metadata or control information. For example, "GET /index.html HTTP/1.1" is a structured message containing a command ("GET") with a single argument ("/index.html") and metadata about which protocol version is being used ("HTTP/1.1").

If an application uses attacker-supplied inputs to construct a structured message without properly encoding or escaping, then the attacker could insert special characters that will cause the data to be interpreted as control information or metadata. Consequently, the component that receives the output will perform the wrong operations, or otherwise interpret the data incorrectly.

Alternate Terms

Output Sanitization

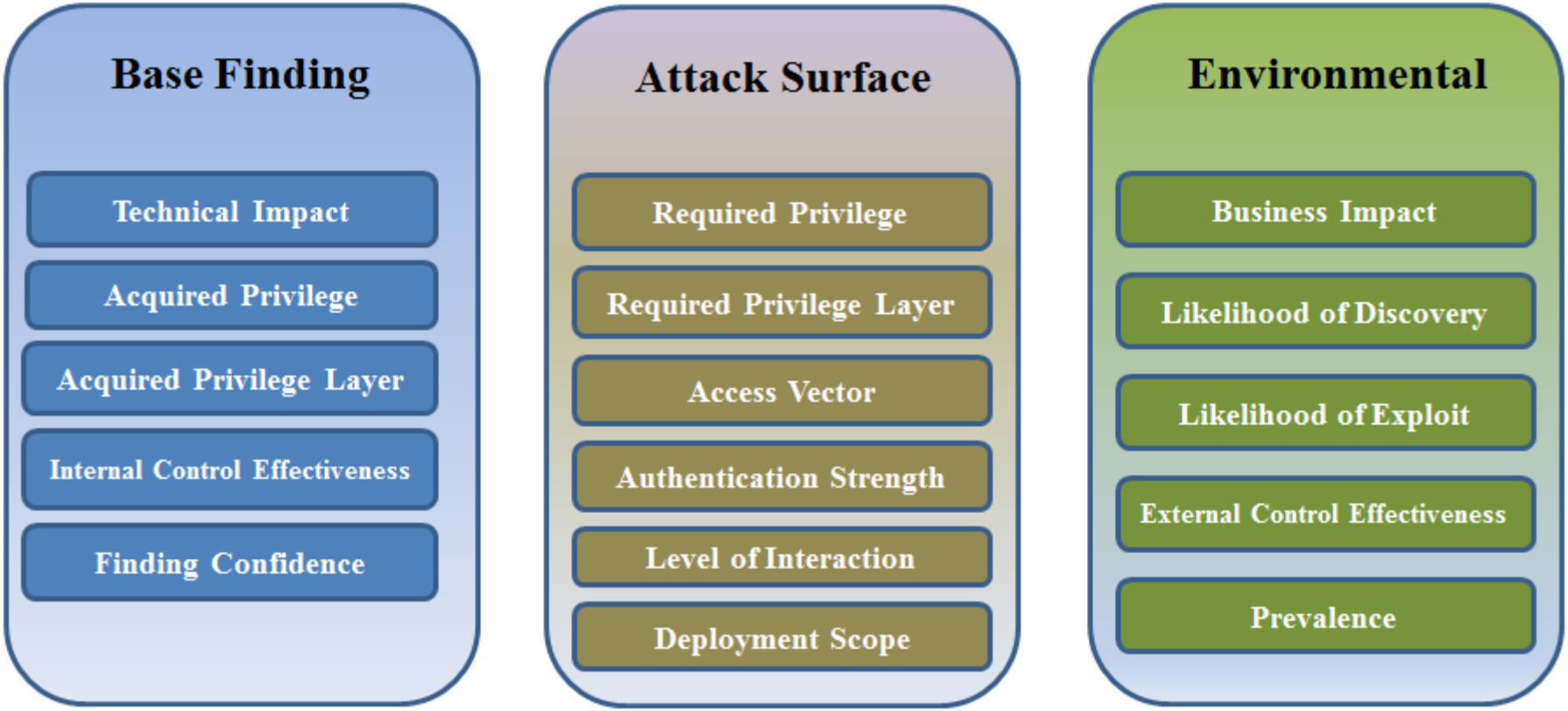
Output Validation

Output Encoding

Prioritizing Weaknesses

- Which bugs should you fix first?
- Common Weakness Scoring System (CWSS) (http://cwe.mitre.org/cwss/cwss_v1.0.1.html)
- Helpful rankings:
  - OWASP Top 10 (https://www.owasp.org/index.php/OWASP_Top_10)
    - Web application focus
    - Mapped to CWE, but uses OWASP Risk Rating Methodology
  - SANS Top 25 Software Errors (https://www.sans.org/top25-software-errors/)
    - SANS uses the CWSS

CWSS Metric Groups



CWSS Weights

- Weight for each value based on estimates of risk, confidence, or other hard to quantify attributes

Proven True	T	1.0	The weakness is reachable by the attacker.
Proven Locally True	LT	0.8	The weakness occurs within an individual function or component whose design relies on safe invocation of that function, but attacker reachability to that function is unknown or not present. For example, a utility function might construct a database query without encoding its inputs, but if it is only called with constant strings, the finding is locally true.
Proven False	F	0.0	The finding is erroneous (i.e. the finding is a false positive and there is no weakness), and/or there is no possible attacker role.
Default	D	0.8	Median of the weights for Proven True, Proven Locally True, and Proven False.
Unknown	UK	0.5	There is not enough information to provide a value for this factor. Further analysis may be necessary. In the future, a different value might be chosen, which could affect the score.

CWSS Score Formula

- A CWSS 1.0 score can range between 0 and 100. It is calculated as follows:
  - BaseFindingSubscore * AttackSurfaceSubscore * EnvironmentSubscore
- E.g., the Base Finding subscore (BaseFindingSubscore) is calculated as follows:
  - Base = [(10 * TechnicalImpact + 5*(AcquiredPrivilege + AcquiredPrivilegeLayer) + 5*FindingConfidence) * f(TechnicalImpact) * InternalControlEffectiveness] * 4.0
  - f(TechnicalImpact) = 0 if TechnicalImpact = 0; otherwise f(TechnicalImpact) = 1
- The other metric groups are similarly complex
- Precision when using estimated values?

OWASP Top 10

- A1-Injection
- A2-Broken Authentication and Session Management
- A3-Cross-Site Scripting (XSS)
- A4-Insecure Direct Object References
- A5-Security Misconfiguration
- A6-Sensitive Data Exposure
- A7-Missing Function Level Access Control
- A8-Cross-Site Request Forgery (CSRF)
- A9-Using Components with Known Vulnerabilities

A1, A3, A4, A8, A10: Unvalidated inputs

OWASP Mapping to CWE

- E.g., A1-Injection maps to following CWE items:
  - CWE Entry 77 on Command Injection
  - CWE Entry 89 on SQL Injection
  - CWE Entry 564 on Hibernate Injection
    - Hibernate is framework for mapping Java to relational db

SANS Top 25 (the first 9)

Rank	Score	ID	Name
[1]	93.8	**CWE-89**	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type

CERT Top 10 Secure Coding Practices

- https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices
  - Validate inputs – esp. from untrusted data sources
  - Heed compiler warnings – Use highest warning level
    - Use static and dynamic analysis tools
  - Architect and design for security policies
  - Keep it simple
  - Default deny
  - Use principle of least privilege
  - Sanitize data sent to other systems
    - (such as command shells!)
  - Practice defense in depth
  - Use effective quality assurance techniques
  - Adopt a secure coding standard
  - Bonus: Define security requirements and model threats

SANS-CWE "Monster Mitigations"

- http://cwe.mitre.org/top25/index.html#Mitigations

ID	Description
M1	**Establish and maintain control over all of your inputs**. (CWE-20)
M2	**Establish and maintain control over all of your outputs**. (CWE-116)
M3	Lock down your environment. (CWE-250 Execution with Unnecessary Privileges)
M4	Assume that external components can be subverted, and your code can be read by anyone.
M5	Use industry-accepted security features instead of inventing your own.
GP1	(general) Use libraries and frameworks that make it easier to avoid introducing weaknesses.
GP2	(general) Integrate security into the entire software development lifecycle.
GP3	(general) Use a broad mix of methods to comprehensively find and prevent weaknesses.
GP4	(general) Allow locked-down clients to interact with your software.

OWASP Secure Coding Practices Checklist- Areas

- **Input Validation**
- **Output Encoding**
- Authentication and Password Management
- Session Management
- Access Control
- Cryptographic Practices
- Error Handling and Logging
- Data Protection
- Communication Security
- System Configuration
- Database Security
- File Management
- Memory Management
- General Coding Practices

https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

OWASP Secure Coding Practices Checklist- Checks

- Each area has several pages of specific checks
- E.g., for Output Validation
  - Conduct all encoding on a trusted system (e.g., The server)
  - Utilize a standard, tested routine for outbound encoding
  - *Contextually output encode* all data returned to the client that originated outside the application's *trust boundary*. Encode all characters unless they are known to be safe for the intended interpreter
  - Contextually *sanitize* all output of un-trusted data to queries for SQL, XML, and LDAP
  - *Sanitize* all output of un-trusted data to operating system commands

Microsoft Coding Best Practices

- Use the latest compiler and supporting tools
  - E.g., highest warning level
- Make use of defenses provided by the compiler
  - E.g., Buffer security check, safe exception handling, DEP
- Use source-code analysis tools
  - Static testing
- Do not use banned functions
  - Legacy functions with known exploits
- Reduce potentially exploitable constructs
  - Static checking doesn't always catch these
- Use a secure coding checklist

Good Coding Practices

- Art, not science
- Different groups have different standards
- But many similarities
  - Major focus: control of inputs and outputs

Outline

- What is Secure Programming?
  - Common software weaknesses
  - Secure coding practices
- "Secure Languages"
- Bug Tracking

“Secure Languages”

- Are there programming languages that are more secure than others?
- How would you measure that?
- How do we know if that is due to the language?
  - May be due to coding practices at company
  - May be due to skill of the programmers
- Which types of vulnerabilities are preventable by a language and which are independent of language?

Example Attempt to Answer Those Q's

- WhiteHat security 2014 survey of languages used to implement web site applications
- Compared vulnerabilities on sites with languages used to implement those sites
- Languages, in order of popularity:
  - .NET
  - Java
  - ASP
  - PHP
  - ColdFusion
  - Perl

Results of Survey: # Vulnerabilities

- # detected vulnerabilities (For each site? For each application?) ranged from 11 to 6
  - Highest: .Net, Java, ASP: 11; PHP: 10
  - Lowest: Perl and ColdFusion: 6 and 7
- Conclusion: Language choice has little effect on # vulnerabilities detected
  - Maybe the problem is their detection method?
- #1 vulnerability type for almost every language: XSS
  - Close #2: information leakage (revealing system data or debugging information through an output stream)

Results of Survey: # Types of Vulnerabilities

- #1 vulnerability type for almost every language: XSS
  - Close #2: information leakage (revealing system data or debugging information through an output stream)
- Median days to remediate XSS ranged from 184 for Perl sites down to 49 for PHP sites
  - This has to largely be due to policies and staffing at the companies running the servers
- Conclusion: Language choice has little effect of types of vulnerabilities
- Language choice probably has nothing to do with time to remediate

Results of Survey: Conclusions

- Language choice does not matter (for commonly used web app programming languages)
- SDLC processes matter
- Testing matters
- Developer skill matters
- Management of server and environment matters
  - Inventory of assets
  - Policy enforcement

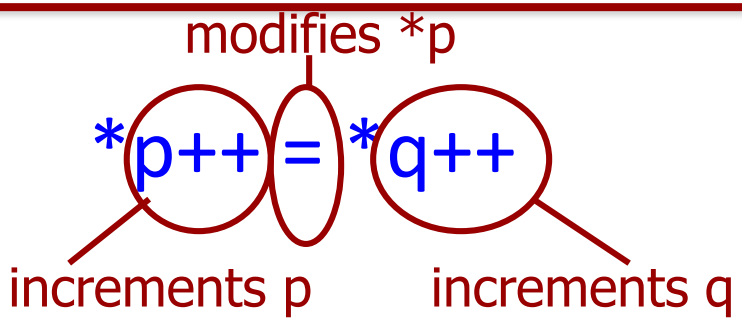
But are there Secure Languages?

- The WhiteHat survey was for commonly used scripting languages
- Overwhelmingly, the vulnerabilities were based on incorrectly validating/sanitizing input data or revealing too much specific data about the system
- These are language-agnostic problems
  - more likely due to lack of training for programmers
- Are there features of programming languages to help create more secure code?
- Specifically, what languages are suitable for developing high-assurance systems?

Good Language Features

- Clear syntax; conceptual simplicity
- Modularity, data abstraction and objects
- Program behavior is the same on different systems
- Type safety - Type errors are detected
- Run-time errors properly trapped
- Memory leaks prevented
- Program analysis
  - Automated error detection, programming environments, compilation checks
- Isolation and special security features
  - Sandboxing, language-based security, …

What Does this C Statement Mean?



Does this mean...

```
*p = *q;
++p;
++q;
```

... or

```
*p = *q;
++q;
++p;
```

... or

```
tp = p;
++p;
tq = q;
++q;
*tp = *tq;
```

Example from Vitaly Shmatikov, U. Texas

Good Java Features

- Modularity and information hiding
- Array bounds checking – data cannot be accessed from area outside of allocated array
  - ArrayOutOfBoundsException
- Exception handling
  - But must correctly handle exceptions or can get DoS
- Managed memory to prevent memory leaks
- Code signing
  - Use cryptography to establish origin of class file
    - This info can be used by Java Security Manager

What about System Programming?

- Java is good for applications, but not for system programming
  - Can't use interpreter; must be native code
  - Can't use sandbox
- Most OS, even JVM, written in C or C++
  - Even some assembly language, when can't avoid it
- But we know C/C++ have problems wrt security
- Want language to help increase assurance
- How to choose?

Choosing a Language for System Programming

- Language features that increase assurance:
  - Strongly-typed
  - Information hiding and static data
  - Semantics reflected by syntax
    - Binary clearly reflects source
  - Unambiguous semantics
    - e.g., no pointer arithmetic
  - Indirect referencing w/o pointers
  - Compiled
- Most A1 systems were implemented in Pascal

Active Area of Research

- Many research projects to create programming-language technology for software security
- E.g.,
  - **Manifest Security** at U. Penn. and CMU
    SOL at U Penn.
    The Grey Project at CMU.
    SELinks at U. Maryland, College Park
    Jif at Cornell University.
    FlowCaml at INRIA.
    Polymer at Princeton
    Cryptyc at DePaul University
    OPA at OWASP
- Most allow programmers to specify information flow and access control security policies on data
- Most based on Java, or on encapsulation/monitoring, or explicitly for applications, so can't be used for system programming

Outline

- What is Secure Programming?
  - Common software weaknesses
  - Secure coding practices
- "Secure Languages"
- Bug Tracking

Bug Tracking

- Necessity for software development assurance
- Use bug tracking tool
- Many systems for tracking bugs
  - E.g. Bugzilla
- Ideally, can integrate with version control systems, like “subversion”, “Git”, and “CVS”
- Database of known bugs
  - Date discovered
  - Current status
  - Assignment to programmer to remediate
- Careful not to reintroduce fixed bugs

INF523: Computer System Assurance

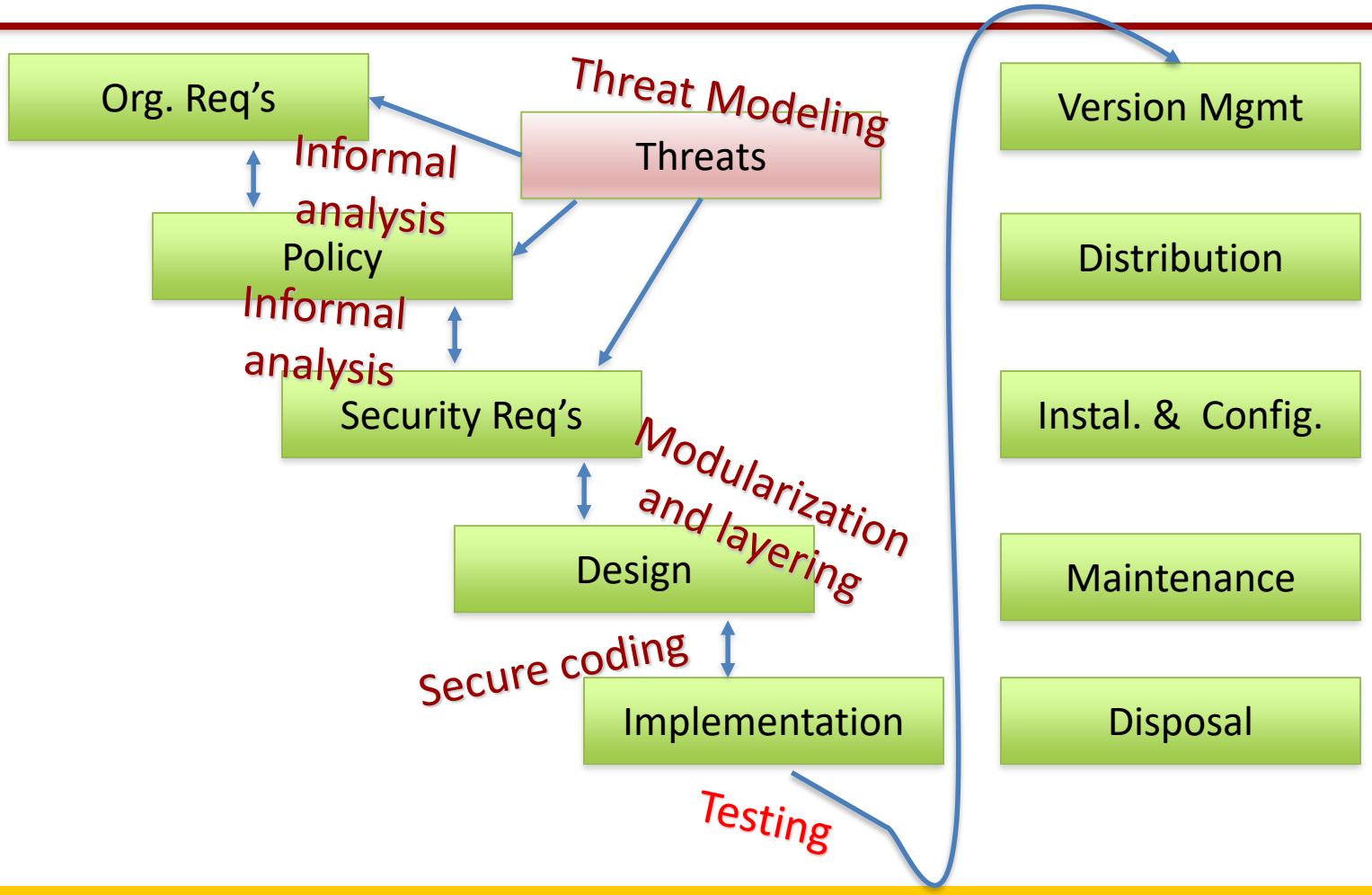
Testing

Prof. Clifford Neuman

Reading for This Class

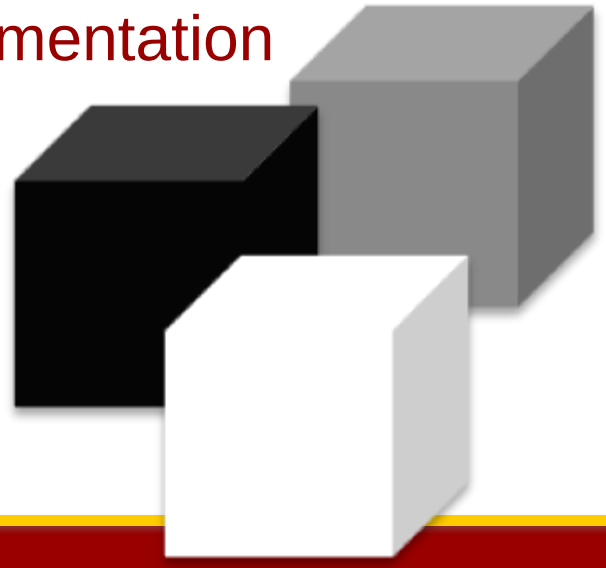
- Bishop book, Chapter 23, “Vulnerability Analysis”, pp. 645-660 (penetration testing)
- *Analysis Techniques for Information Security*, pp. 5-10 (static testing)
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, *Using static analysis to find bugs*, IEEE Software, vol. 25, no. 5, pp. 22–29, Sep./Oct. 2008
- P. Oehlert, *Violating assumptions with fuzzing*, 2005 (fuzzing/dynamic testing)
- Jose Fonseca, et. al., *Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks*, 2007 (vulnerability scanning)
- *The Design and Implementation of Tripwire: A File System Integrity Checker*, Gene Kim, 1993

"Assurance Waterfall"



Black Box and White Box Testing

- Black box testing
  - Tester has no information about the implementation
  - Good for testing independence
  - Not good for test coverage
  - Hard to test individual modules
- White box testing
  - Tester has information about the implementation
  - Knowledge guides tests
  - Simplifies diagnosis of problem
  - Can zero in on specific modules
  - Possible to have good coverage
  - May compromise tester independence



Layers of Testing

- Module testing
  - Test individual modules or subset of system
- Systems integration
  - Test collection of modules
- Acceptance testing
  - Test to show that system meets requirements
  - Typically focused on functional specifications

Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Security Testing

- A process to find system flaws that would lead to violation of the security policy
  - Find flaws in security mechanisms
  - Find flaws that could bypass security mechanisms
- Focus is on security policy, not function

Security Testing

- Functional testing: Does system do what it is supposed to do?
  - In the presence of good inputs
- Security testing: Does the system do what it is supposed to do, *and nothing more*?
  - For good *and bad* inputs
  - E.g., I can only get access to my data after I log in
    - But can I get access to only my data?
- Security testing assumes intelligent adversary
  - Test functional and non-functional security requirements
  - Test as if you were an attacker

Testing Security Mechanisms

- Security mechanisms thought of as "non-functional"
  - Often not tested during system testing!
- But many security mechanisms do have functional specifications
- Must test security mechanisms as if they were the subject of functional testing
  - E.g., test identification and authentication mechanisms
  - Do they correctly enforce the policy?
  - What if malicious inputs?
  - Do they "fail safe"?

What to Test in Security Testing

- Violation of assumptions
  - About inputs
    - Behavior of system with “bad” inputs
    - Inputs that violate type, size, range, …
  - About environment
  - About operational procedures
  - About configuration and maintenance
- Often due to
  - Ambiguous specifications
  - Sloppy procedures
- Special focus on Trust Boundaries

Types of Flaws – Implementation Bugs

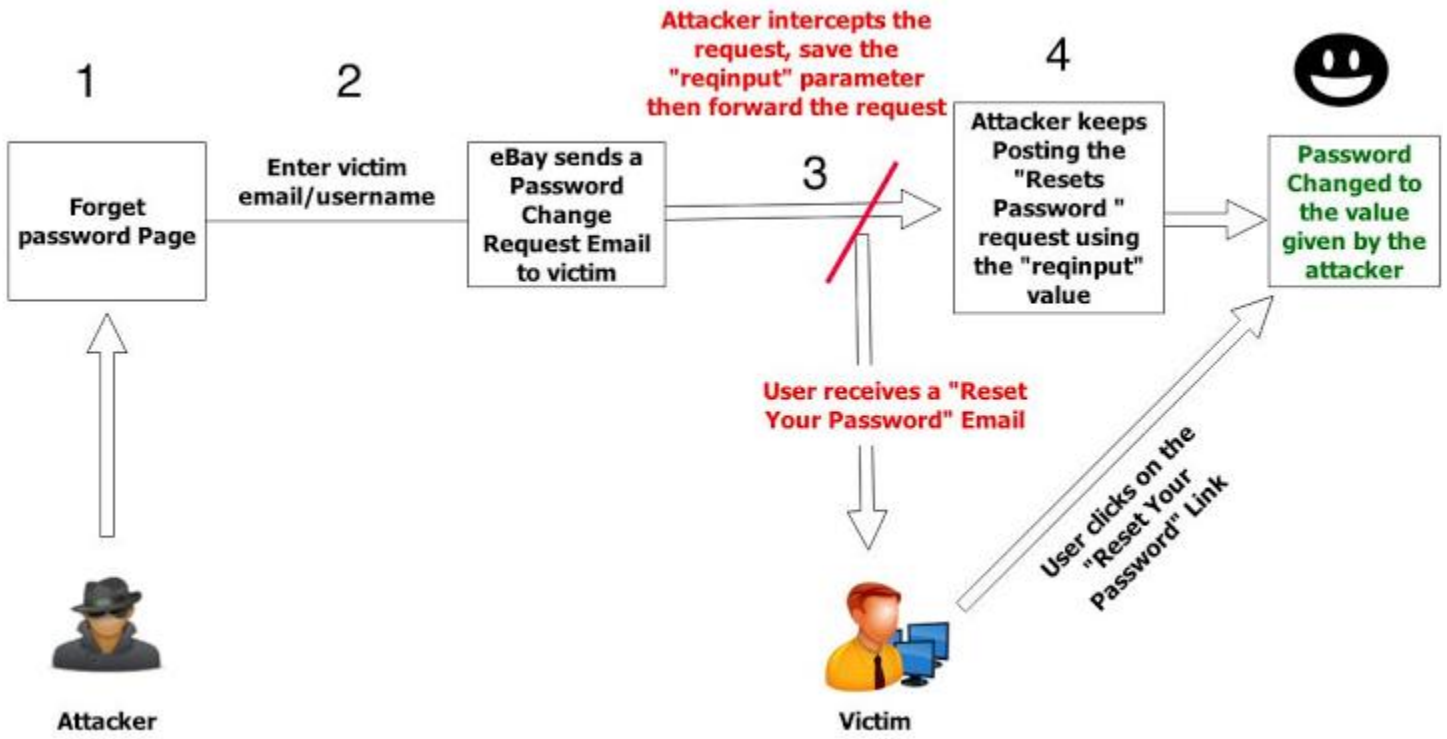
- Coding errors
  - E.g., use of gets() function and other unchecked buffers
- Logical errors
  - E.g., time of check to time of use (“TOUCTOU”)
  - Race condition where, e.g., authorization changes but

Victim	Attacker

`if (access("file", W_OK) != 0) { exit(1); }`
 	
`fd = open("file", O_WRONLY);`
`write(fd, buffer, sizeof(buffer));`
	`// After the access check,`
	`symlink("/etc/passwd", "file");`
	`// Before the open, "file" points to the password database`

eBay Password Reset Bug

- Reported Nov 2014 (http://thehackernews.com/2014/09/hacking-ebay-accounts.html)
- Programming error - used wrong “secret code”



Types of Flaws – Design Flaws

- Error handling - E.g., failure in insecure states
- Transitive trust issues (typical of DAC)
- Unprotected data channels
- Broken or missing access control mechanisms
- Lack of audit logging
- Concurrency issues (timing and ordering)

- Design flaws are likely hardest to detect
- Usually most critical
- Probably most prevalent

A Fundamental, "Unsolvable" Problem

- Fundamental problem: lack of reference monitor
  - Entire system ("millions of lines of code") vulnerable
  - Buffer overflow in GUI is as serious as bug in access control mechanism
  - No way to find the numerous flaws in all of that code
- Reference monitor is "small enough to be verifiable"
  - Helps bound testing
- But testing still required for reference monitor

Limits of Testing

- "Testing can prove the presence of errors, but not their absence" – Edsger W Dijkstra
- How much testing is enough?
  - Undecidable
  - Never "enough" because never know if found all bugs
  - But resources, including time, are finite
- Testing would probably miss eBay flaw, for example
  - Requires deep understanding of flaw and precise test
- Subversion? Find a trap-door? Forget about it.
- Must *prioritize*

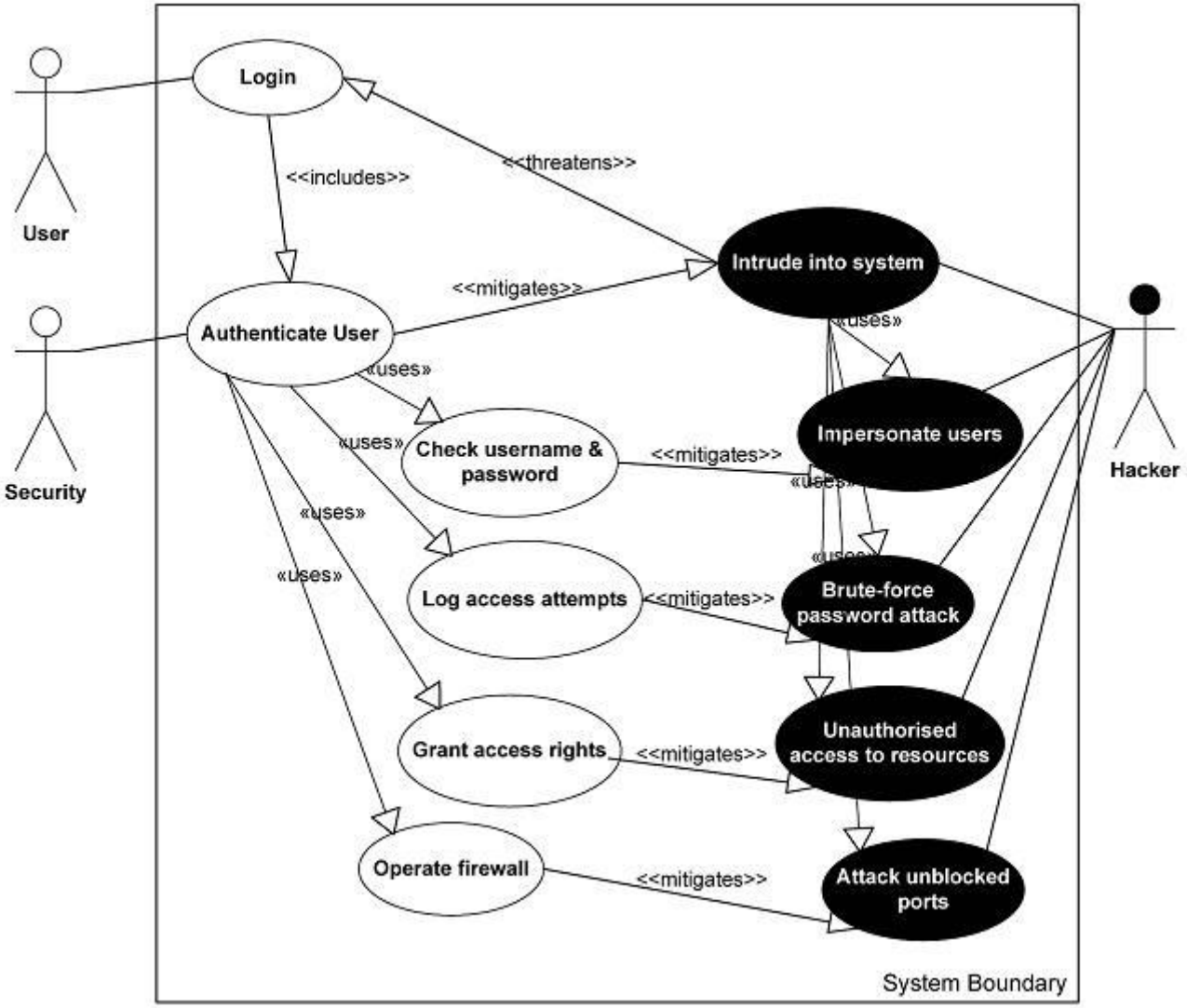
Prioritizing Risks and Tests

- Create security misuse cases
  - I.e., threat assessment
- Identify security requirements
  - Use identified threats with policy to derive reqs
- Perform architectural risk analysis
  - Where will I get the biggest bang for my buck?
  - Trust boundaries are very interesting here
- Build risk-based security test plans
  - Test the “riskiest” things
- Perform the (now limited, but focused) testing

Misuse Cases

- "Negative scenarios"
  - I.e., threat modeling
- Define what an attacker would want
- Assume level of attacker abilities/skill
  - Helps determine what steps are possible and risk
- Imagine series of steps an attacker could take
  - Attack-defense tree or requires/provides model
  - Or "Unified Modeling Language" (UML)
- Identify potential weak spots and mitigations

Example of UML



Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Static Testing

- Analyze code (and documentation)
  - Usually only source code, but sometimes object
  - Program not executed
  - Testing abstraction of the system
- Code analysis, inspection, reviews, walkthroughs
  - Human techniques often called “code review”
- Automated static testing tools
  - Checks against coding standard (e.g., formatting)
  - Coding flaws
  - Potentially malicious code
  - May also refer to formal proof of code correctness

Static Testing Techniques

- Many Static Testing techniques based on compiler technology
- Some techniques:
  - Type analysis
  - Abstract Interpretation
  - Data-flow analysis
  - Taint analysis

Type Analysis

- Type analysis
  - For languages without strong typing, like JavaScript
  - Program analyzed against type constraints
  - Each construct has derived type, or expected type
  - May have false positives

```
function onlyWorksOnNumbers(x) {
    return x * 10;
}

onlyWorksOnNumbers('Hello, world!');
```

Abstract Interpretation

- Abstract Interpretation
  - Partial execution using an interpreter
  - Map variable values to ranges or relations
    - E.g., map pointer values to “points-to” relation
  - For control or data flow, without performing calculations
  - Abstraction can be sound or unsound
    - Sound – never false negatives but may be false positives
      - “Over-abstraction” may include unreachable states
      - Usually slower tools
    - Unsound – may have false negatives and false positives
      - Over- and Under-abstraction possible
      - Time trade-off so faster

Data Flow Analysis

- Data-flow analysis
  - Gathers information about possible set of variable values at specific points in the program
  - Uses control flow graph (CFG) and lattice theory
  - Examples:
    - Liveness
    - Dead variables
    - Uninitialized variables
    - Sign analysis
    - Lower and upper bounds

```
1.if b==4 then
2.   a = 5;
3.else
4.   a = 3;
5.endif
7.if a < 4 then
8....
```

The “reaching” definition of variable “a’” at line 7 is the set of assignments a=5 at line 2 and a=3 at line 4.

Taint Analysis

- Taint analysis
  - Tries to identify variables affected by user input
  - Tracks flow of data dependencies in program
  - If tainted variables are ever passed to sensitive functions, flag

```
1  x := 2*get_input(·)
2  y := 5 + x
3  goto y
```

Line #	Statement	Δ	τ_Δ	Rule	*pc*
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \rightarrow 40\}$	$\{x \rightarrow \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-GOTO	*error*

“Lint-like” Tools

- Finds “suspicious” software constructs
  - E.g., Variables being used before being initialized
  - Divide by zero
  - Constant conditions
  - Calculations outside the range of a type
- Language-dependent
- Can check correspondence to style guidelines

Example Static Testing Tool

- Splint – Modern version of classic “lint” tool

```c
#include <stdio.h>
int main()
{
  char c;
  while (c != 'x');
  {
    c = getchar();
    if (c = 'x')
      return 0;
    switch (c){
    case '\n':
    case '\r':
      printf("Newline\n");
    default:
      printf("%c",c);
    }
  }
  return 0;
}
```

Splint's output:

```
* Variable c used before definition
* Suspected infinite loop. No value
used in loop test (c) is modified by
test or loop body.
* Assignment of int to char: c =
getchar()
* Test expression for if is assignment
expression: c = 'x'
* Test expression for if not boolean,
type char: c = 'x'
* Fall through case (no preceding
break)
```

Limitations of Static Testing

- Lots of false positives and false negatives
- Automated tools seem to make it easy, but it takes experience and training to use effectively
- Misses many types of flaws
- Won’t find vulnerabilities due to run-time environment

Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Dynamic Testing

- Test running software in “real” environment
  - Contrast with static testing
- Techniques
  - Simulation – assess behavior/performance
  - Error seeding – bad input, see what happens
    - Use extremes of valid/invalid input
    - Incorrect and unexpected input sequences
    - Altered timing
  - Performance monitoring – e.g., real-time memory use
  - Stress tests – e.g., abnormally high workloads

Limits to Dynamic Testing

- From outside, cannot test all software paths
- Cannot even test all hardware faults
- May not find rare events (e.g., due to timing)

Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Fuzzing

- Tool used by both security testers and attackers
- Form of dynamic testing, usually automated
- Provide many invalid, unexpected, often random inputs to software
  - Extreme limits, or beyond limits, of value, size, type, ...
  - Can test command line, GUI, config, protocol, format, file contents, …
- Observe behavior – if unexpected result, a flaw!
  - Crashes or other bad exception handling
  - Violations of program state (assertions)
  - Memory leaks
- Flaws could conceivably be exploited
- Fix, and re-test

Fuzzing Examples

- Testing for integer overflows
  - -1, 0, 0x100, 0x3fffffff, 0x7ffffffe, 0x7fffffff, 0xffffffff, etc.
- Testing for buffer overflows
  - ‘A’ x Z, where Z is in domain {1, 5, 33, 129 257, 513, etc.}
- Testing for format string errors
  - %s%p%x%d, .1024d, %d%d%d%d, %%20s, etc.

Fuzzing Methods

- Mutation-based
  - Mutate existing test data, e.g., by flipping bits
- Generation-based
  - Generate test data based on models of input
  - Use a specification
- Black box – no reference to code
  - Useful for testing proprietary systems
- White (or gray) box – use code as a guide of what to test
- Recursive – enumerate all possible inputs
- Replacive – use only specific values

Limits of Fuzzing

- Random sample of behavior
- Usually finds only simple flaws
- Best for rough measure of software quality
  - If find lots of problems, better re-work the code
- Also good for regression testing, or comparing versions
- Demonstrates that program handles exceptions
- Not a comprehensive bug-finding tool
- Not a proof that software is correct

Fuzzers

- Lots of different fuzzing programs available
- SPIKE, framework for protocol fuzzing (linux)
  - http://www.immunitysec.com/resources-freesoftware.shtml
  - Intro to use: http://resources.infosecinstitute.com/intro-to-fuzzing/
- Peach (Windows, Mac, linux)
  - http://sourceforge.net/projects/peachfuzz/
  - Data definitions written in XML
- CERT Basic Fuzzing Framework (BFF)
  - https://www.cert.org/vulnerability-analysis/tools/bff.cfm
- Or not hard to roll your own, at least for simple random fuzzing

Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Vulnerability Scanning

- Another tool used by attackers and defenders alike
- Automated
- Look for flaws using database of known flaws
  - Contrast with fuzzing
- As comprehensive as database of vulnerabilities is
- Different types of vulnerability scanners (example):
  - Port scanner (NMAP)
  - Network vulnerability scanner (Nessus)
  - Web application scanner (Nikto)
  - Database (Scuba)
  - Host security audit (Lynis)

Vulnerability Scanning Methods

- Passive – probe without any changes
  - E.g., Check version and configuration, “rattle doors”
  - Do nothing that might crash the system
- Active – attempt to see if actually vulnerable
  - Run *exploits* and monitor results
  - Might disrupt, crash, or even damage target
  - Always get explicit permission (signed agreement) before running active scans

Example Nessus Output

Taking the following actions across 10 hosts would resolve 20% of the vulnerabilities on the network:

Action to take	Vulns	Hosts
OpenSSH LoginGraceTime / MaxStartups DoS: Upgrade to OpenSSH 6.2 and review the associated server configuration settings.	12	3
Samba 3.x < 3.5.22 / 3.6.x < 3.6.17 / 4.0.x < 4.0.8 read_nttrans_ea_lis DoS: Either install the patch referenced in the project's advisory, or upgrade to version 3.5.22 / 3.6.17 / 4.0.8 or later.	9	1
Dropbear SSH Server < 2013.59 Multiple Vulnerabilities: Upgrade to the Dropbear SSH 2013.59 or later.	6	3
MS05-051: Vulnerabilities in MSDTC Could Allow Remote Code Execution (902400) (uncredentialed check): Microsoft has released a set of patches for Windows 2000, XP and 2003.	4	1
Firewall UDP Packet Source Port 53 Ruleset Bypass: Either contact the vendor for an update or review the firewall rules settings.	4	2

Limits of Vulnerability Scanning

- Passive scanning only looks for known vulnerabilities
  - Or potential vulnerabilities (e.g., based on configuration)
- Passive scanning often simply checks versions
  - then reports known vulnerabilities in those versions
  - and encourages updating
- Active scanning can crash or damage systems
- Active scanning potentially requires a lot of "hand-holding"
  - Due to unpredictable system behavior
  - E.g., system auto-log out

Outline

- Security testing
- Static testing
- Dynamic testing
- Fuzzing
- Vulnerability scanning
- Penetration testing

Penetration Testing

- Actual attacks on a system carried out with the goal of finding flaws
  - Called a “test”, when used by defenders
  - Called an “attack” when used by attackers
- Human, not automated
- Usually goal driven – stop when achieve
- Step-wise (like requires/provides)
  - When find one way to achieve a step, go on to next step
- Identifies vulnerabilities that may be impossible for automated scanning to detect
- Shows how different low-risk vulns can be combined into successful exploit
- Same precautions as for other forms of active testing
  - Explicit permission; don’t interfere with production

Flaw-Hypothesis Methodology

- Five steps:

1. Information gathering
   - Become familiar with the system's design, implementation, operating procedures, and use
2. Flaw hypothesis
   - Think of flaws the system might have
3. Flaw testing
   - Test for exploitable flaws
4. Flaw generalization
   - Generalize vulnerabilities that can be exploited
5. Flaw elimination (often skipped)

Limits of Penetration Testing

- Informal, non-rigorous, semi-systematic
  - Depends on skill of testers
- Not comprehensive
  - Proves at least one path, not all
  - When find one way to achieve a step, go on to next step
- Does not prove lack of path if unsuccessful
- But, performed by experts
  - Who are not the system developers
  - Who think like attackers
- Tests developer and operator assumptions
  - Helps locate shortcomings in design and implementation
  - Probably only test technique that would find eBay bug