INF523: Computer System Assurance

Secure Programming
(continued from 2 weeks ago)

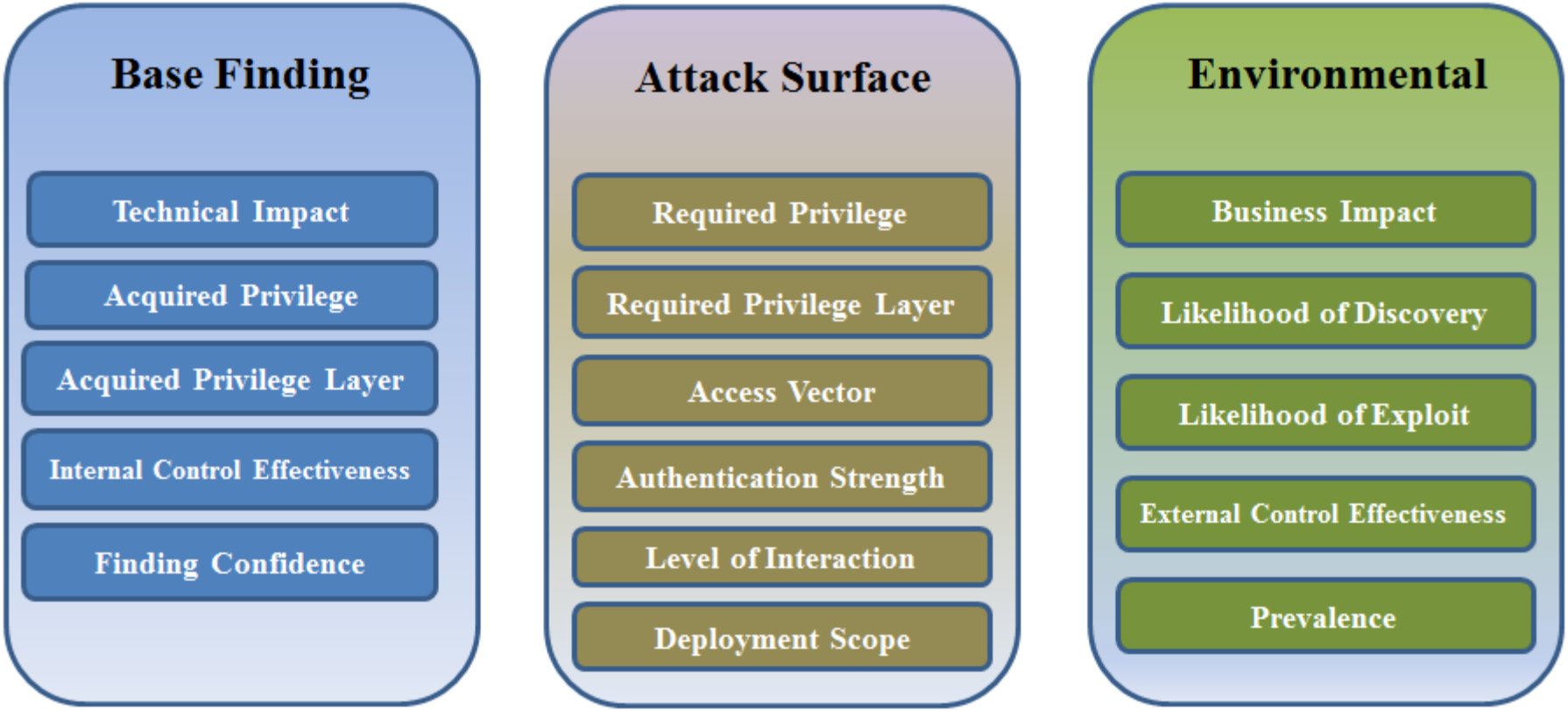
Prof. Clifford Neuman

Lecture 5
2 October 2020

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

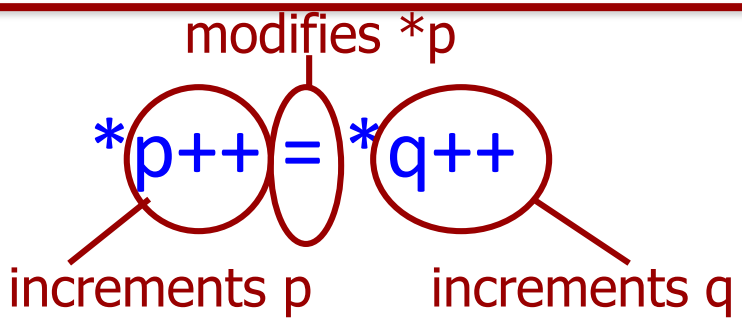
But are there Secure Languages?

- 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

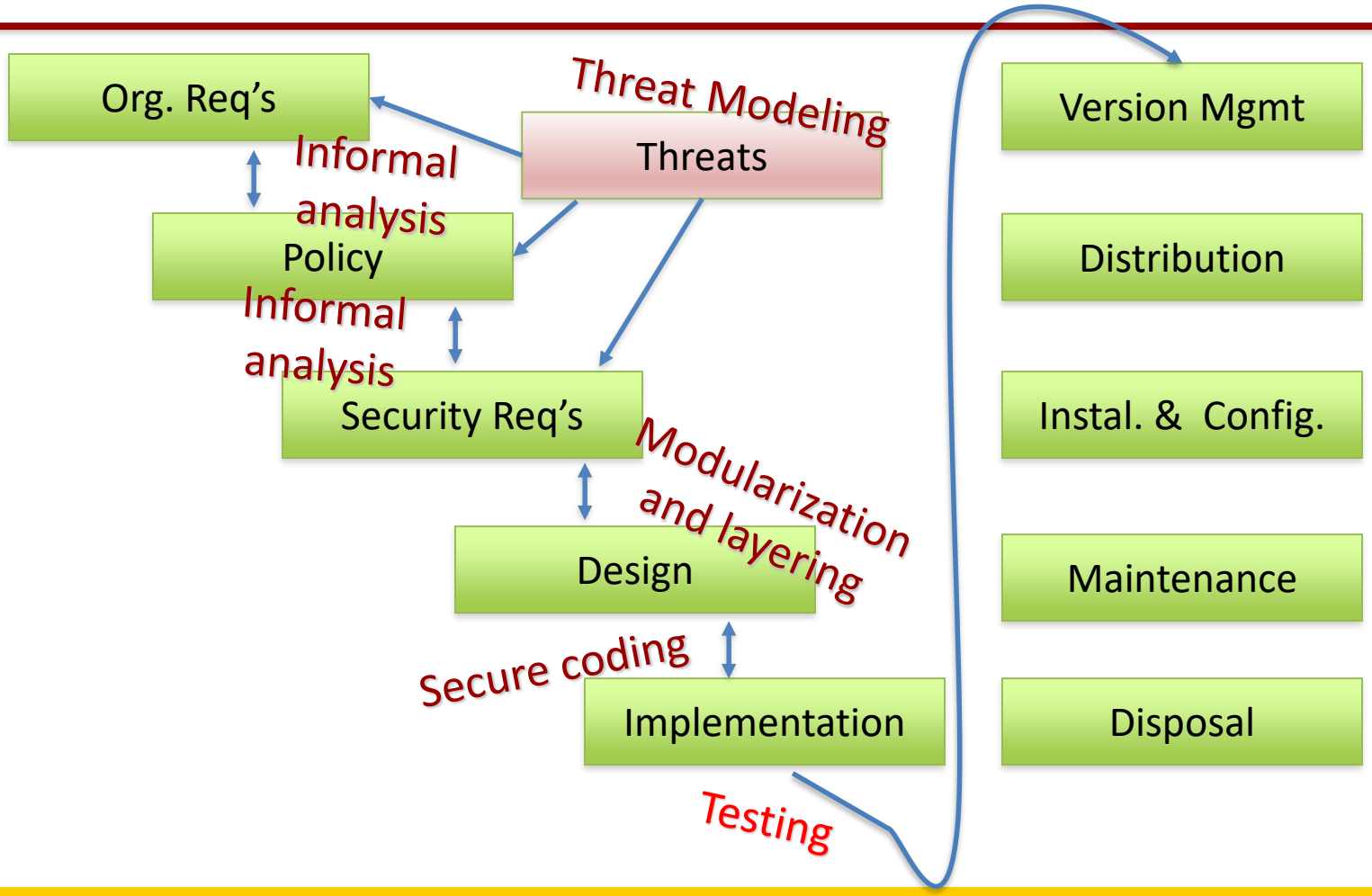
Testing

Prof. Clifford Neuman

Reading for This Class

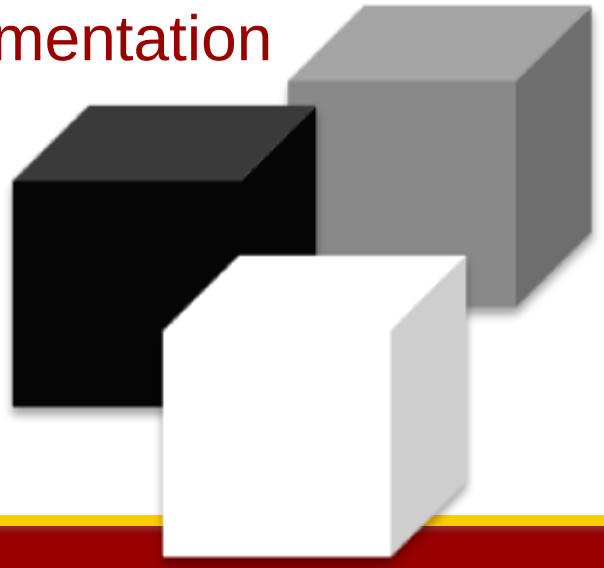
- Bishop book, Chapter 23, "Vulnerability Analysis", pp. 645-660 (penetration testing)
- *Analysis Techniques for Information Security*, pp. 5-10 (static testing)
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, *Using static analysis to find bugs*, IEEE Software, 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

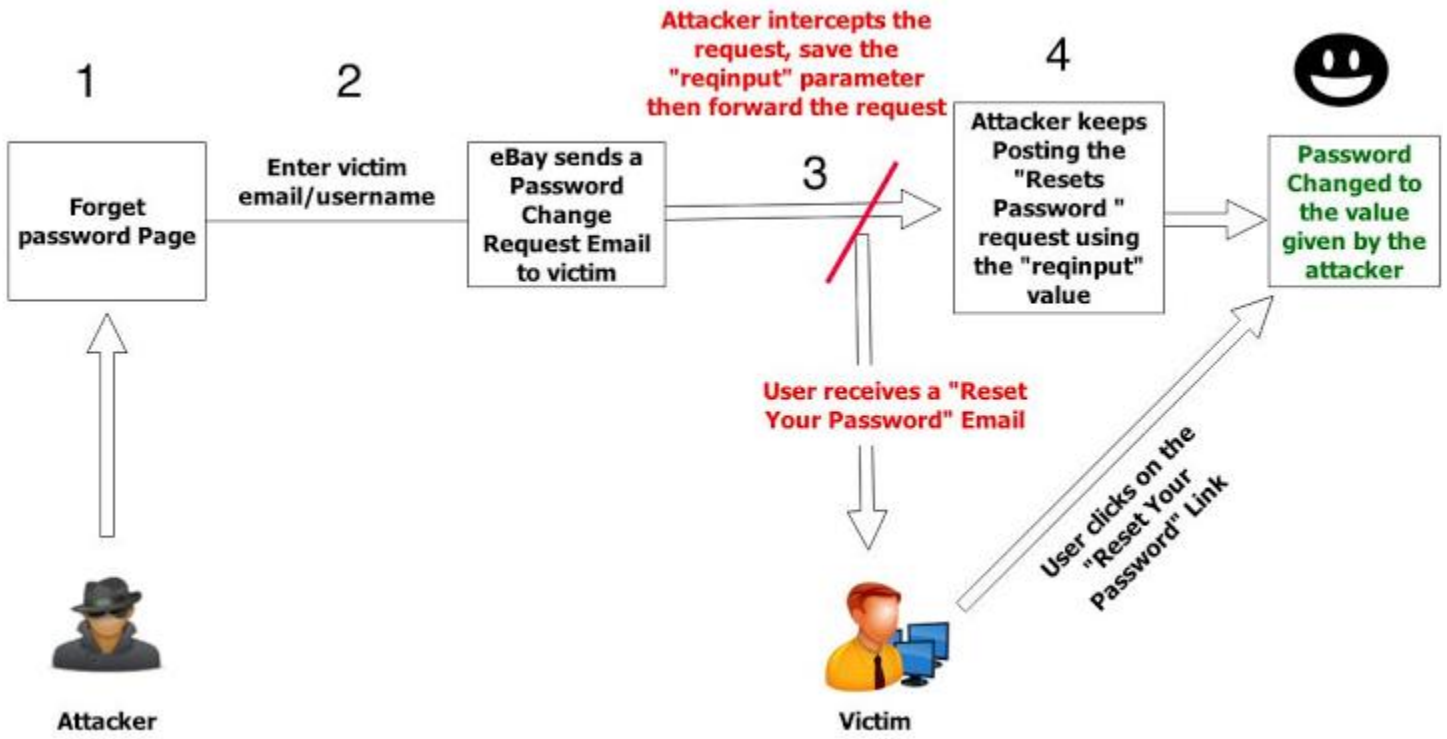
- Coding errors
  - E.g., use of gets() function and other unchecked buffers
- Logical errors
  - E.g., time of check to time of use ("TOUCTOU")
  - Race condition where, e.g., authorization changes but

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*

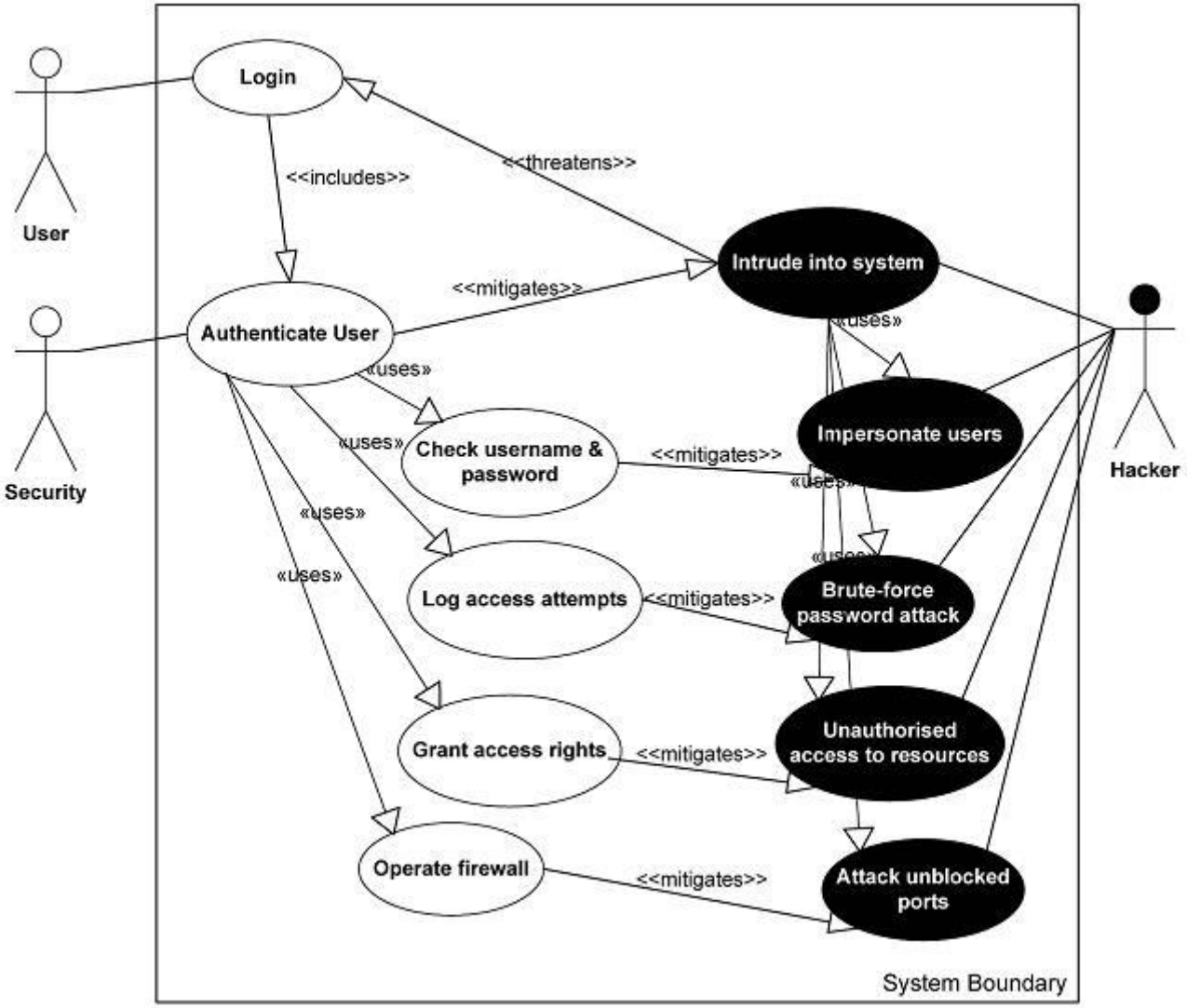
Prioritizing Risks and Tests

- 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

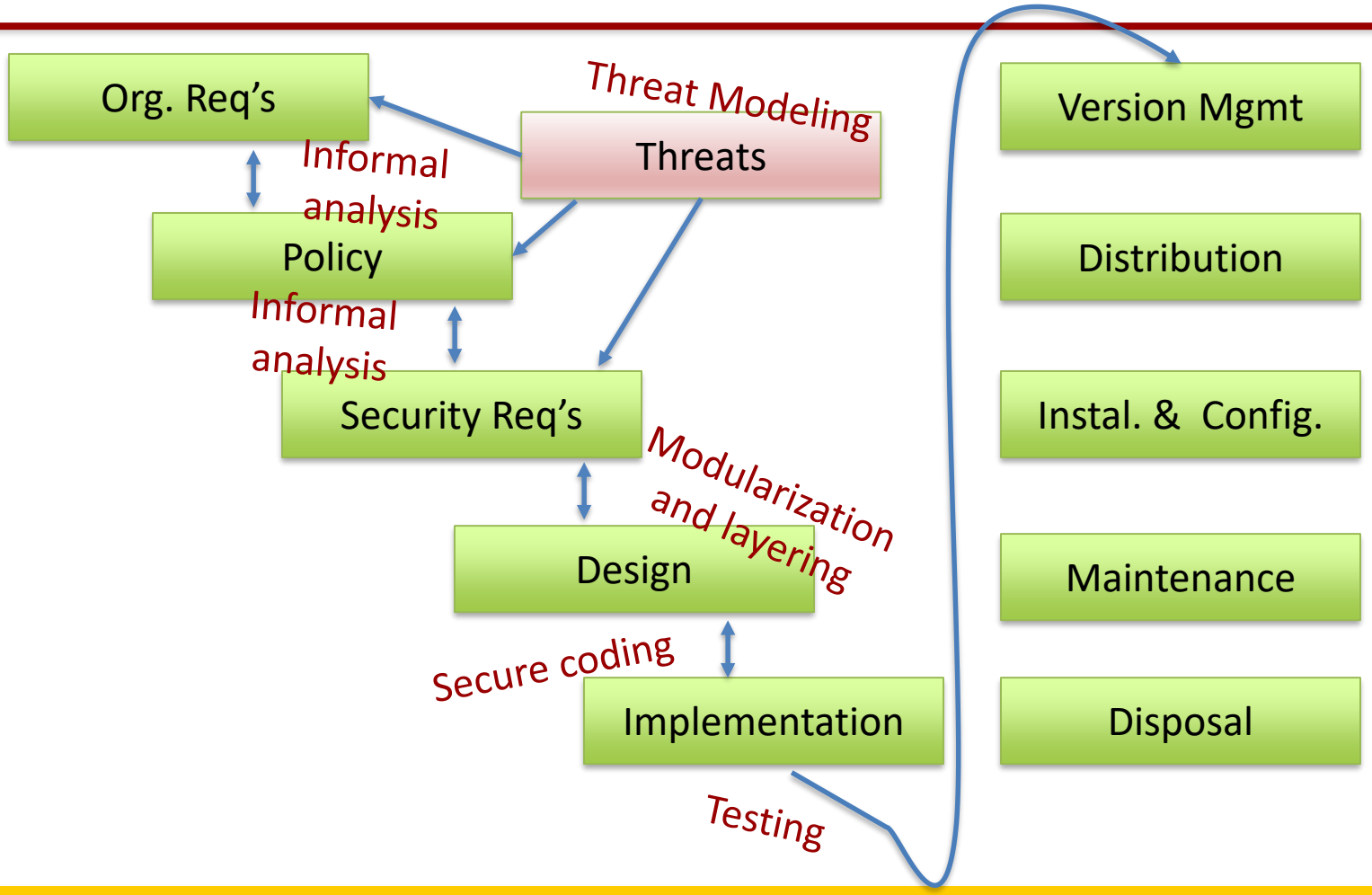
- Informal, non-rigorous, semi-systematic
  - Depends on skill of testers
- Not comprehensive
  - Proves at least one path, not all
  - When find one way to achieve a step, go on to next step
- Does not prove lack of path if unsuccessful
- But, performed by experts
  - Who are not the system developers
  - 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

INF523: Computer System Assurance

Secure Operation

Prof. Clifford Neuman

"Assurance Waterfall"

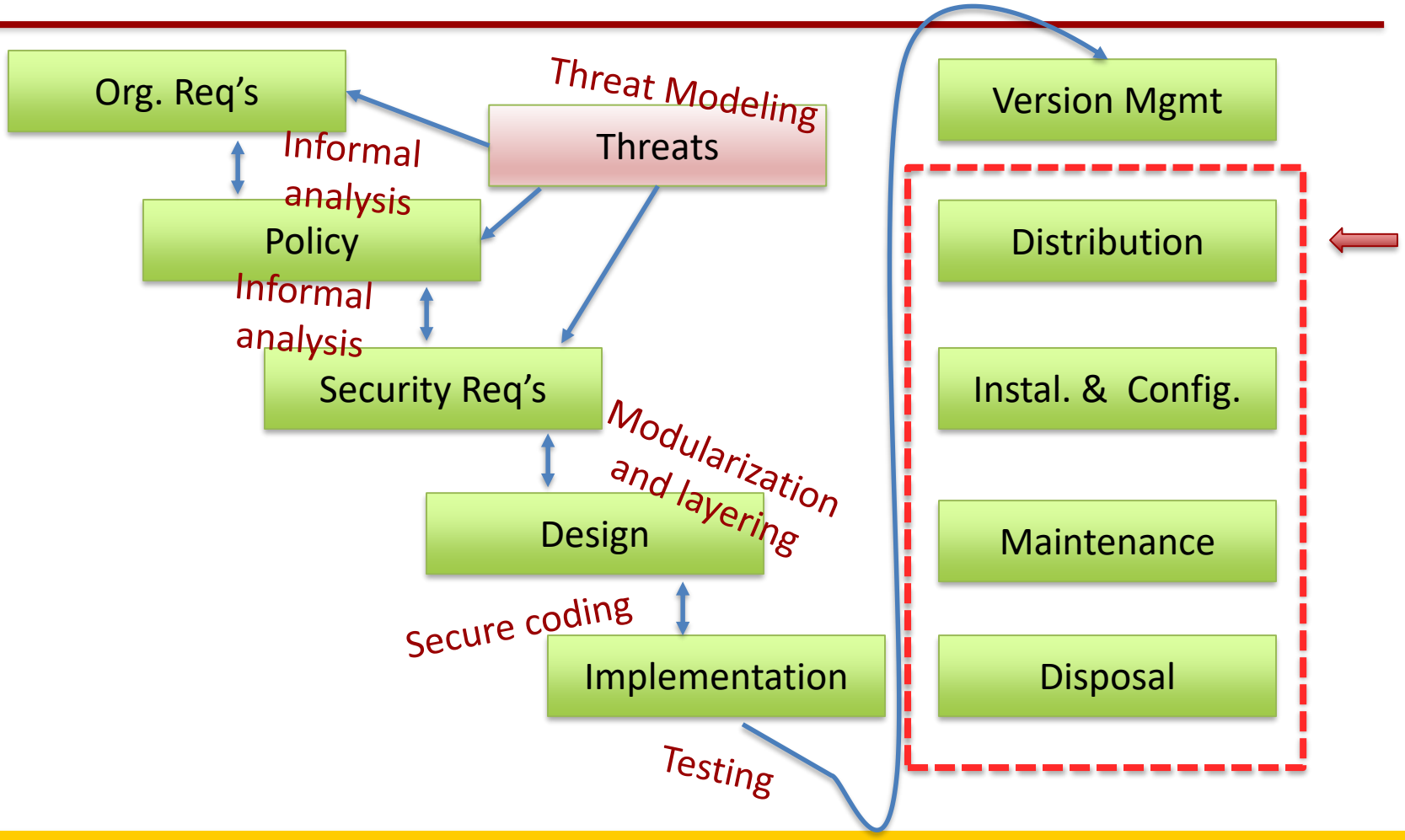


What's Left?

- Secure distribution
- Secure installation and configuration
- Patch management
- System audit and integrity monitoring
- Secure disposal

- For very high-assurance systems:
  - Covert channel analysis
  - Formal (mathematical) methods:
    - Specification and proofs
    - FSPM, FTLS, DTLS

"Assurance Waterfall"



Secure Distribution

- Problem: Integrity of distributed software
  - How can you “trust” distributed software?
  - **Watch out for subversion!**
  - **How is this accomplished for iOS?**
    - **Hint: It is in the news this week.**
- Is this the actual program from the vendor?
- … or did someone substitute or tamper with it?
  - Who might want to do that?

Checksums

- Compare hashes on downloaded files with published value (e.g., on developer's web site)
  - If values match, good to go
  - If values do not match, don't install!
- Often download site different from publisher
  - So checksum is control on distribution
- Use good hash algorithms
  - MD5 – compromised (can reliably make duplicates)
  - SHA-1 – no demonstration of compromise, but feared
  - SH-256 (aka SHA-2) still OK

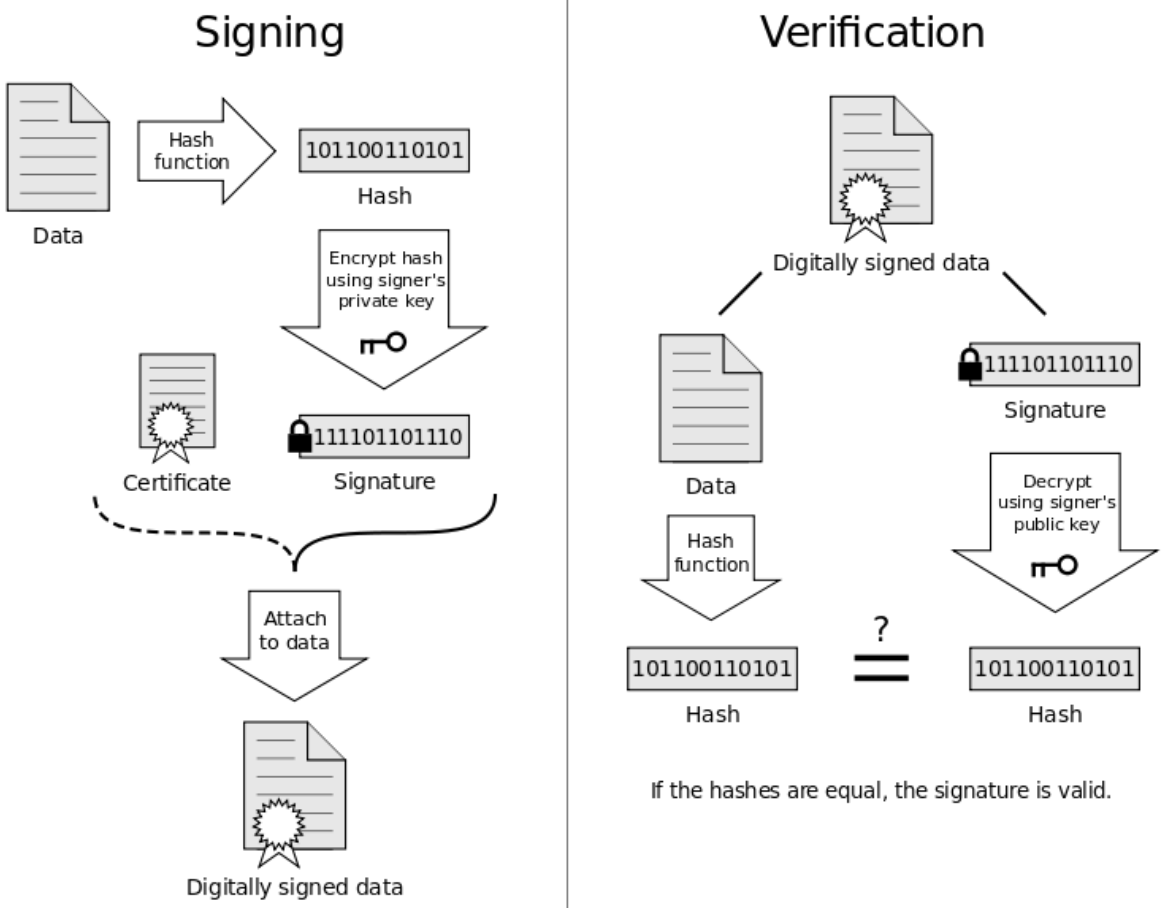
Are Checksums Reliable?

- Don’t run install from distribution point
  - Download, calculate and compare checksum first
- Make sure connected to right hash source
  - What if visit spoofed site?
  - How do you know you are on the right site?
- What if download file and checksum from same site?
  - What use is the checksum?
- Make sure connection to hash source is tamperproof
  - What if MITM attack?
  - How do you know your connection hasn’t been compromised?

Cryptographic Signing

- Solves checksum reliability problems?
- Typically uses PKI cryptography
- Signing algorithm:
  - Calculate checksum (hash) on object
  - Encrypt checksum using signer's private key
  - Attach seal to object (along with certificate of signer)
- Verification algorithm:
  - Calculate checksum on object
  - Decrypt encrypted checksum using signers' public key
  - Compare calculated and decrypted checksums

Cryptographic Signing



Source: Wikipedia

Cryptographic Signing

- Solves checksum reliability problems?
- Typically uses public/private key cryptography
- Signing algorithm:
  - Calculate checksum (hash) on object
  - Encrypt checksum using signer's private key
  - Attach seal to object (along with certificate of signer)
- Verification algorithm:
  - Calculate checksum on object
  - Decrypt encrypted checksum using signers' public key
  - Compare calculated and decrypted checksums
- Missing step: Check signer's certificate

Do You Trust the Certificate?

- You trust a source because the calculated checksum matches the checksum in the seal
- Certificate contains signer's public key
- You use public key to decrypt seal
- How do you know that signer is trustworthy?
- Certificates (like for SSL), testify as to signer identity
- Based on credibility of certificate authority
- But what if fake certificate?
  - E.g., Stuxnet

Secure Distrib in High-Assurance System

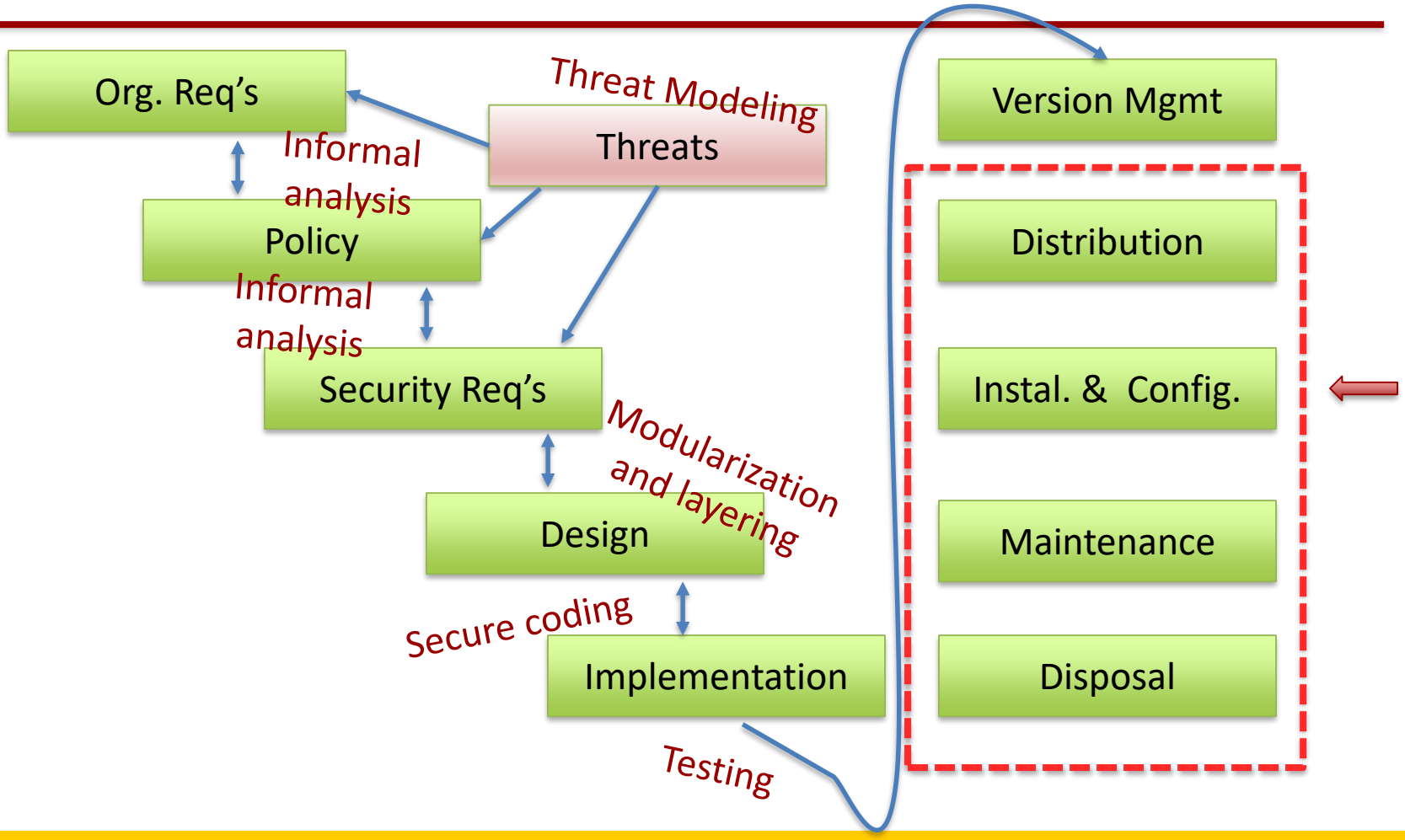
- E.g., GTNP FER (page 142)
  - Based on *cryptographic seals and data encryption*. All kernel segments are encrypted and sealed. Formatting information on distributed volumes is sealed but not encrypted. Keys to check seals and decrypt are shipped separately [i.e., sent out of band; no certification authority].
  - Hardware distribution through authenticator for each component, implemented as cryptographic seal of unique identifier of component, such as serial number of a chip or checksum on contents of a PROM [Physical HW seal and checked by SW tool]

More on GTNP Secure Distribution

- Physical seal on HW to detect tampering
- Install disk checks HW “root of trust” during install
  - Will only install on specific system

- System Integrity checks at run-time
- Multi-stage boot:
  - PROM checks checksum of boot loader
  - Boot loader checks checksum of kernel

"Assurance Waterfall"



Secure Installation and Configuration

- Evaluated, high-assurance systems come with documentation and tools for secure configuration
- Lower-assurance systems have less guidance
- Usually informal checklists
  - Benchmarks
  - Security Technical Implementation Guides (STIGs)
- Based on “best practices”
  - E.g., “change default admin password”
  - No formal assessment of effectiveness
- Not based on security policy model

E.g., Microsoft Baseline Security Analyzer

- http://www.microsoft.com/en-us/download/details.aspx?id=7558
- Standalone security and vulnerability scanner
- Helps determine security state
  - Missing patches
  - Microsoft *configuration recommendations*
- Some of the checks it does:
  - Administrative vulnerabilities
  - Weak passwords
  - Presence of known IIS and SQL vulnerabilities

STIGS

- Security Technical Implementation Guides (STIGs)
- E.g., https://web.nvd.nist.gov/view/ncp/repository
  - (Need SCAP tool to read them)
- Based on “best practices”
- Not based on security policy model

Security Content Automation Protocol

- Security Content Automation Protocol (SCAP)
  - Tools can automatically perform configuration checking using XML checklist
- Example rule for Windows 7:

```
<xccdf:Rule id="xccdf_gov.nist_rule_microsoft_network_server_disconnect_clients_when_logons_expire" selected="false" weight="10.0">
 <xccdf:title>Microsoft network server: Disconnect clients when logon hours expire</xccdf:title>
 <xccdf:description>Users should not be permitted to remain logged on to the network after they have exceeded their permitted logon hours. In many ca
 <xccdf:reference>
  <dc:type>GPO</dc:type>
  <dc:source>Computer Configuration\Windows Settings\Security Settings\Local Policies\Security Options</dc:source>
 </xccdf:reference>
 <xccdf:ident system="http://cce.mitre.org">CCE-9358-3</xccdf:ident>
 <xccdf:check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
  <xccdf:check-export export-name="oval:gov.nist.usgcb.windowsseven:var:56" value-id="xccdf_gov.nist_value_disconnect_client_whe
  <xccdf:check-content-ref href="USGCB-Windows-7-oval.xml" name="oval:gov.nist.usgcb.windowsseven:def:83"/>
 </xccdf:check>
</xccdf:Rule>
```

Configuration Management Systems

- Centralized tools and databases to manage configs
- Ideally:
  - Complete list of systems
  - Complete list of software
  - Complete list of versions
- Logs status and changes
- Can automatically push out patches/changes
- Can detect unauthorized changes
- E.g., Windows group policy management
- For more info: https://www.sei.cmu.edu/productlines/frame_report/config.man.htm

Example for High Assurance System

- E.g., GTNP FER (p. 102)
  - System Maintenance Utility – Used to define physical disk partitions, define RAM disks, allocate logical volumes to disk partitions, and modify physical device parameters
  - System Generation Utility – Used to format volumes, set volumes' read-only attribute, establish links between volumes and mount segments, define system resource limits used by the kernel, and define configuration and initial environment of initial TCB processes
- Mistakes can make system unusable, but not

Certification and Accreditation

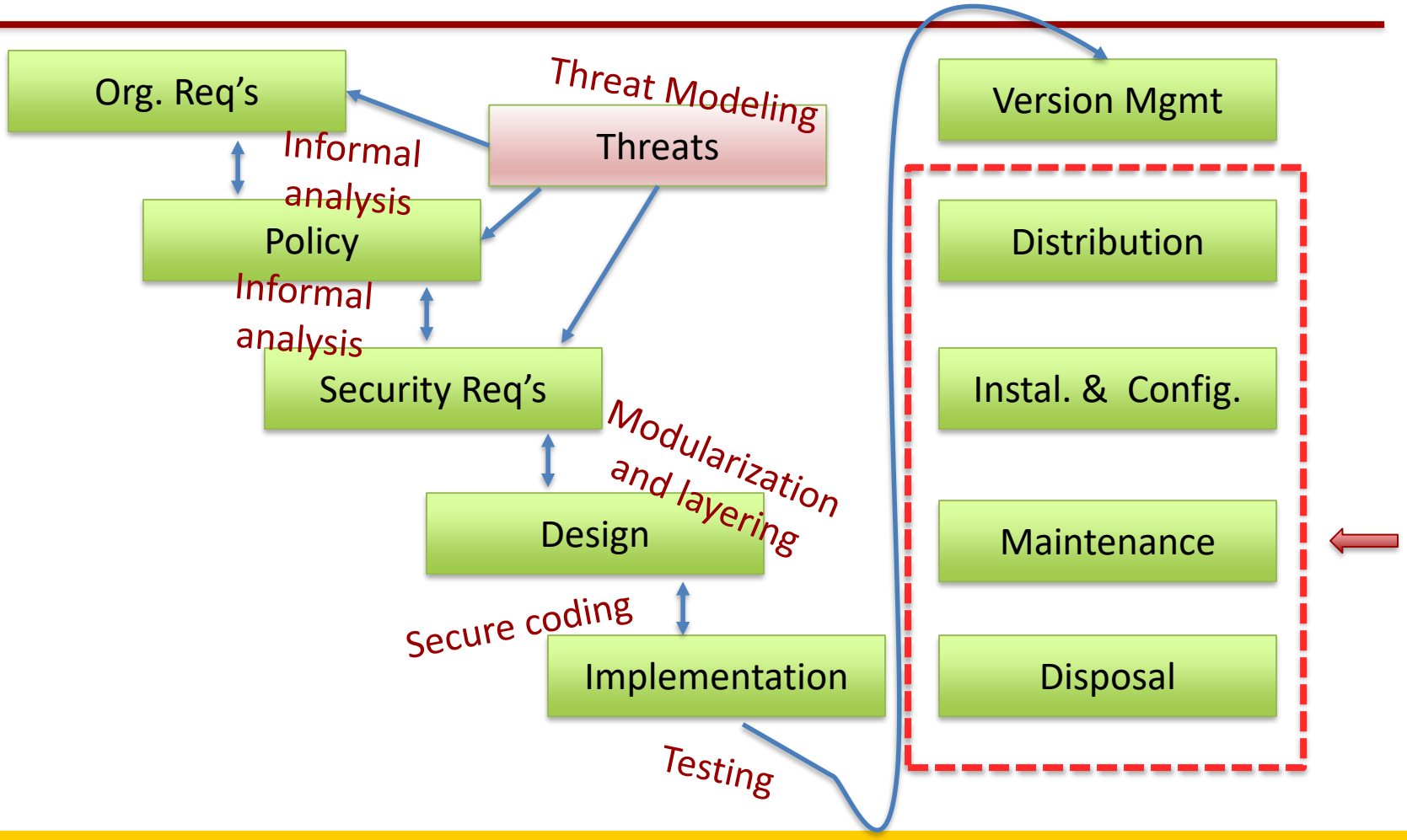
- Evaluated systems are certified
  - Under specific environmental criteria
  - (e.g., for TCSEC, criteria listed in Trusted Facility Manual)
- But environmental criteria must be satisfied for accreditation
  - E.g., security only under assumption that network is physically isolated
  - If instead use public Internet, cannot be accredited

Operational Environment and Change

- Must “configure” environment
- **Not enough to correctly install and configure a system if the environment is out of spec**

- What if the system and environment start out correctly configured, but then change?

"Assurance Waterfall"



Maintenance

- System is installed and configured correctly
- Environment satisfies requirements
- Will they stay that way?
- Maintenance needs to
  1. Preserve known, secure configuration
  2. Permit necessary configuration changes
     - E.g., patching

Patch Management

- All organizations use low-assurance systems
- Low-assurance systems have lots of bugs
- A “patch” is a security update to fix vulnerabilities
  - Maybe to fix bugs introduced in last patch
- Constant “penetrate-and-patch” cycle
  - Must constantly acquire, test, and install patches
- Patch management:
  - Strategy and process of determining
    - what patches should be applied,
    - to which programs and systems, and
    - when

Risk of Not Applying Patches

- Ideally, install patches ASAP
- Risk goes way up when patches are not installed
  - System then has known vulnerabilities
  - "Assurance" of system is immediately very low
  - Delay is dangerous – live exploits often within hours
- But is there risk of installing patches too soon?

Patch Management Tradeoffs

- Delay means risk
- But patches may break applications
  - Custom applications or old, purchased applications
- Patches may even break the system
  - Microsoft, for example, “recalls” patches
  - (Microsoft Recalls Another Windows 7 Update Over Critical Errors http://www.techlicious.com/blog/faulty-windows-7-update-kb3004394/)
- Must balance the two risks
  - Sad fact: Security often loses in these battles
  - Must find other mitigating controls

Patch Testing and Distribution

- Know what patches are available
- Know what systems require patching

- Test patches before installing
  - On non-production systems
  - Test as completely as possible with operational environ.

- Distribute using signed checksum
  - Watch out for subversion, even inside the organization

Challenges in Patch Management

- NIST Guide to Patch Management Technologies (http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r3.pdf)
- Timing, prioritization, and testing
- Ideally, deploy immediately
  - But must test first! Possible side-effects
  - Testing takes time and resources
  - May require rebooting/restarting, so more delay
- Vendors may bundle patches so less frequent
  - But then window of exposure is much longer
- Difficult to keep track of installed patches
  - System vuln scanners or config mgmt. tools can help

Patching High-Assurance Systems

- Distribution same as for original system:
  - Cryptographic seals and data encryption
  - Keys to check seals and decrypt are shipped separately
- Advantage of high-assurance system:
  - Lots of effort to “get it right” from the beginning
  - Modularization, layering, proper testing, FSPM, etc.
- No TCSEC Class A1 system ever needed security patch (per Roger Schell)

Preserve Known, Secure Configuration

- Two steps:

1. Document that installation and initial configuration are correct
   - Don't forget environment
   - Update documentation as necessary after patching
2. Periodically check that nothing has changed in system (or environment)
   - Compare results of check to documentation

System Audit and Integrity Monitoring

- Static audit: scan systems and note discrepancies
  - Missing patches
  - Mis-configurations
  - Changed, added, or deleted system files
  - Changed, added or deleted applications
  - Added or deleted systems!
- Dynamic system integrity checking
  - Same as static, but continuous
- Example: Tripwire (http://www.tripwire.com/)

Tripwire

- Used to create checksums of
  - user data,
  - executable programs,
  - configuration data,
  - authorization data, and
  - operating system files
- Saves database
- Periodically calculates new checksums
- Compares to database to detect unauthorized or unexpected changes

Continuous Monitoring

- Static audit is good, but systems may be out of compliance almost immediately
- Goal: Real-time detection and mediation
  - Sad reality: minutes to days to detect, maybe years to resolve
- Need to automate monitoring
- See, e.g.,
  - SANS Whitepaper: http://www.sans.org/reading-room/whitepapers/analyst/continuous-monitoring-is-needed-35030
  - NIST 800-137 Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations http://csrc.nist.gov/publications/nistpubs/800-137/SP800-137-Final.pdf

Inventory of Systems and Software

- IT operations in constant state of flux
  - New services, legacy hardware and software, failure to follow procedures and document changes
- Make a list of authorized systems, software, and versions (and patches)
  - Create baseline
  - Discovery using administrative efforts, active and passive technical efforts
- Regularly scheduled scans to look for deviations
  - Continuously update as new approved items added or items deleted

Other things to Monitor

- System configurations
- Network traffic
- Logs
- Vulnerabilities
- Users

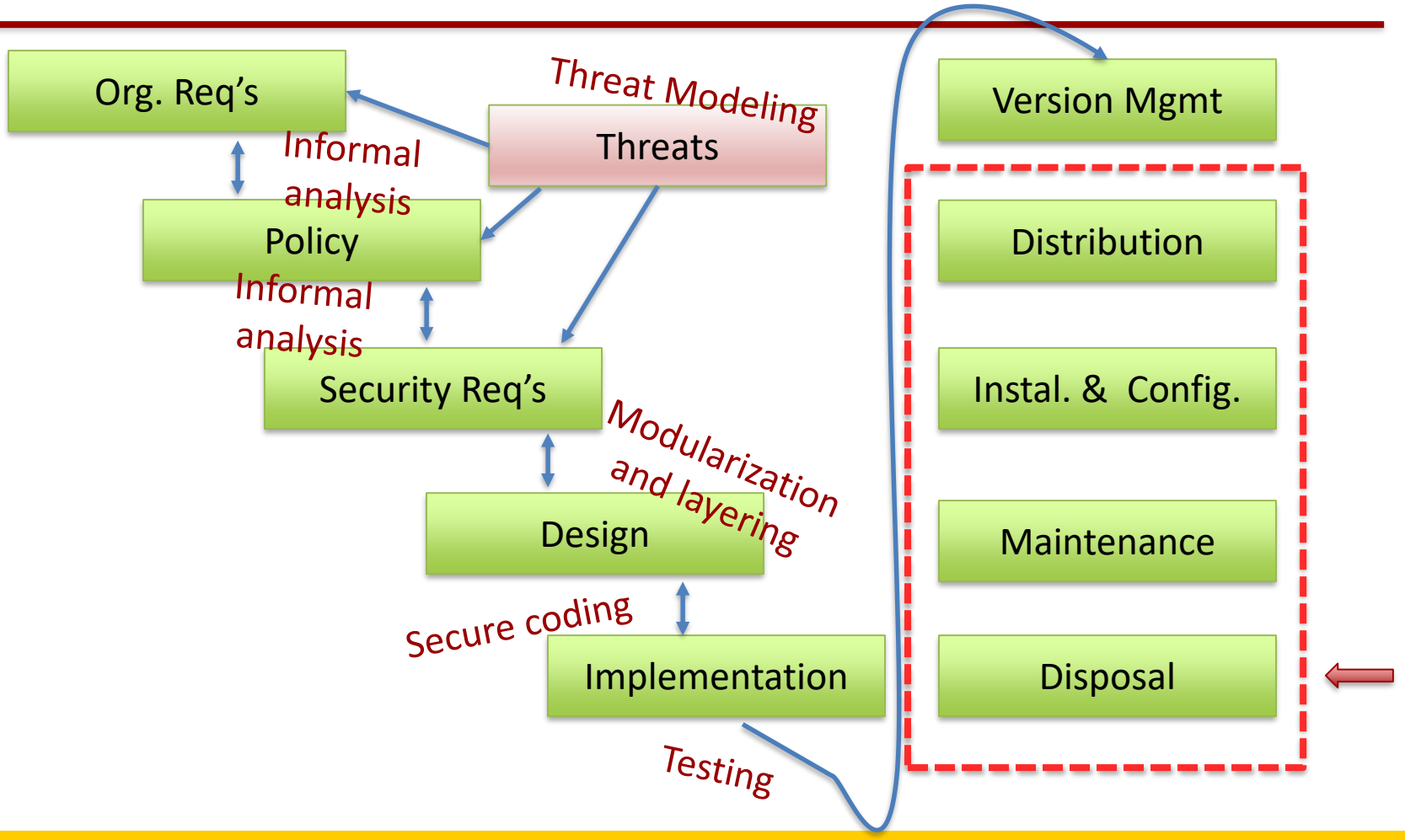
- To manage workload:
  - Determine key assets
  - Prioritize alerts

System Integrity in High Assurance System

- E.g., GTNP FER (p. 121-123)
  - HW and SW integrity tests at boot time
  - Continuously running diagnostic tests (system idle)

- Obviously standalone
- Not part of a larger networked environment

"Assurance Waterfall"



Secure Disposal Requires Attention

- Delete sensitive data on systems before disposal
  - Not always obvious where media is
- E.g., copy machines have hard drives
  http://www.cbsnews.com/news/digital-photocopiers-loaded-with-secrets/
- E.g., mobile phones not properly erased
  http://www.theguardian.com/technology/2010/oct/12/mobile-phones-personal-data
  - *50% of second-hand mobile phones contain personal data*

Secure Disposal

- User proper disposal techniques
  - E.g., shred drives or other storage media for best results
  - Degaussing of magnetic media not enough
  - SSDs even harder to erase