INF523: Computer System Assurance

Testing

Prof. Clifford Neuman

Lecture 7
9 October 2020

INF523: Computer System Assurance

Testing (continued)

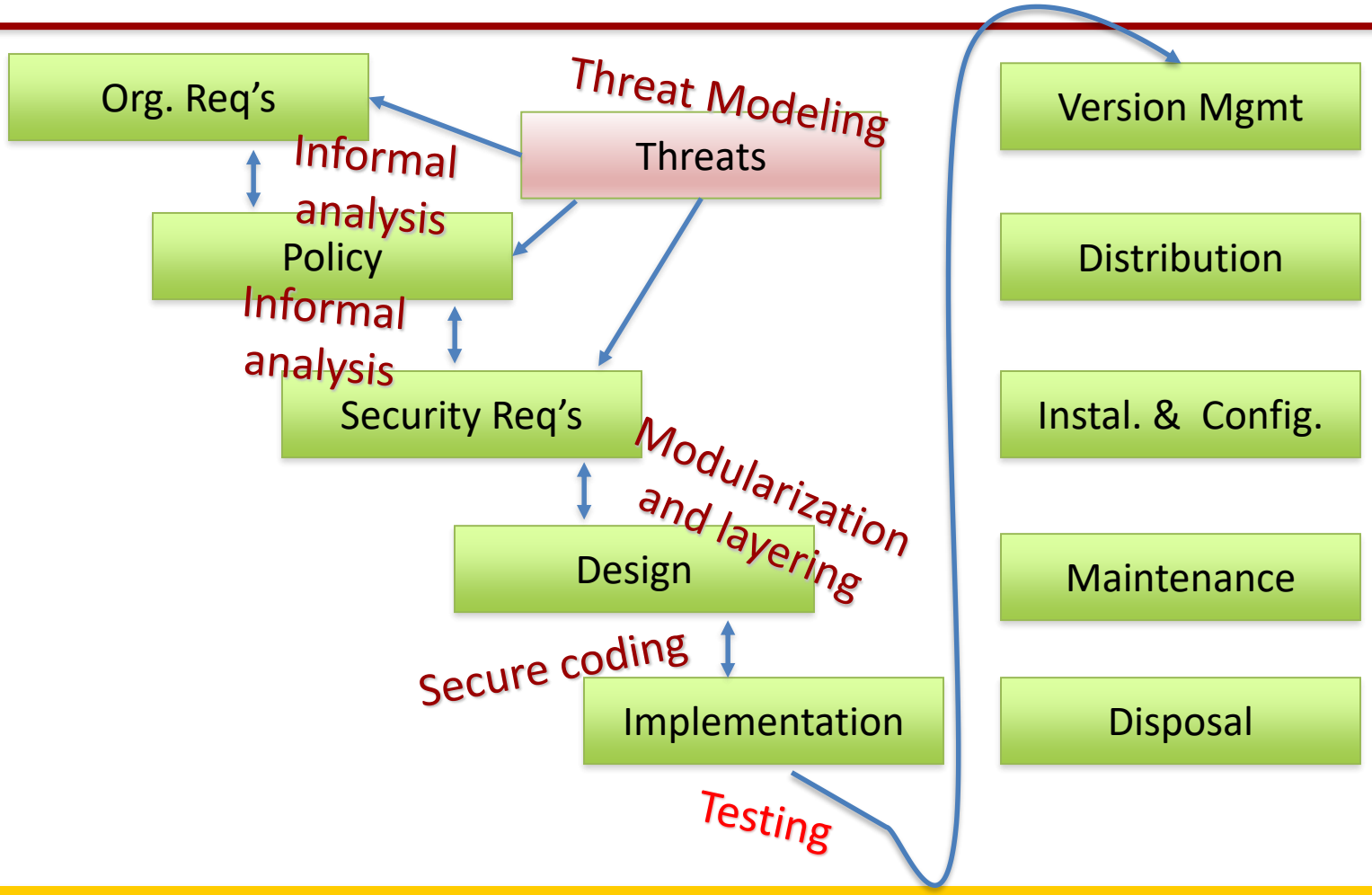
Prof. Clifford Neuman

Lecture 7
9 October 2020

Reading for This Class

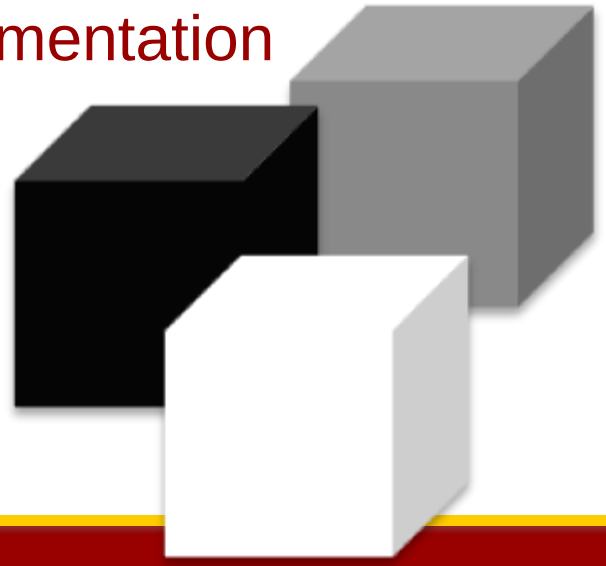
- Bishop book, Chapter 23, “Vulnerability Analysis”, pp. 645-660 (penetration testing)
- *Analysis Techniques for Information Security*, pp. 5-10 (static testing)
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, *Using static analysis to find bugs*, IEEE Software,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

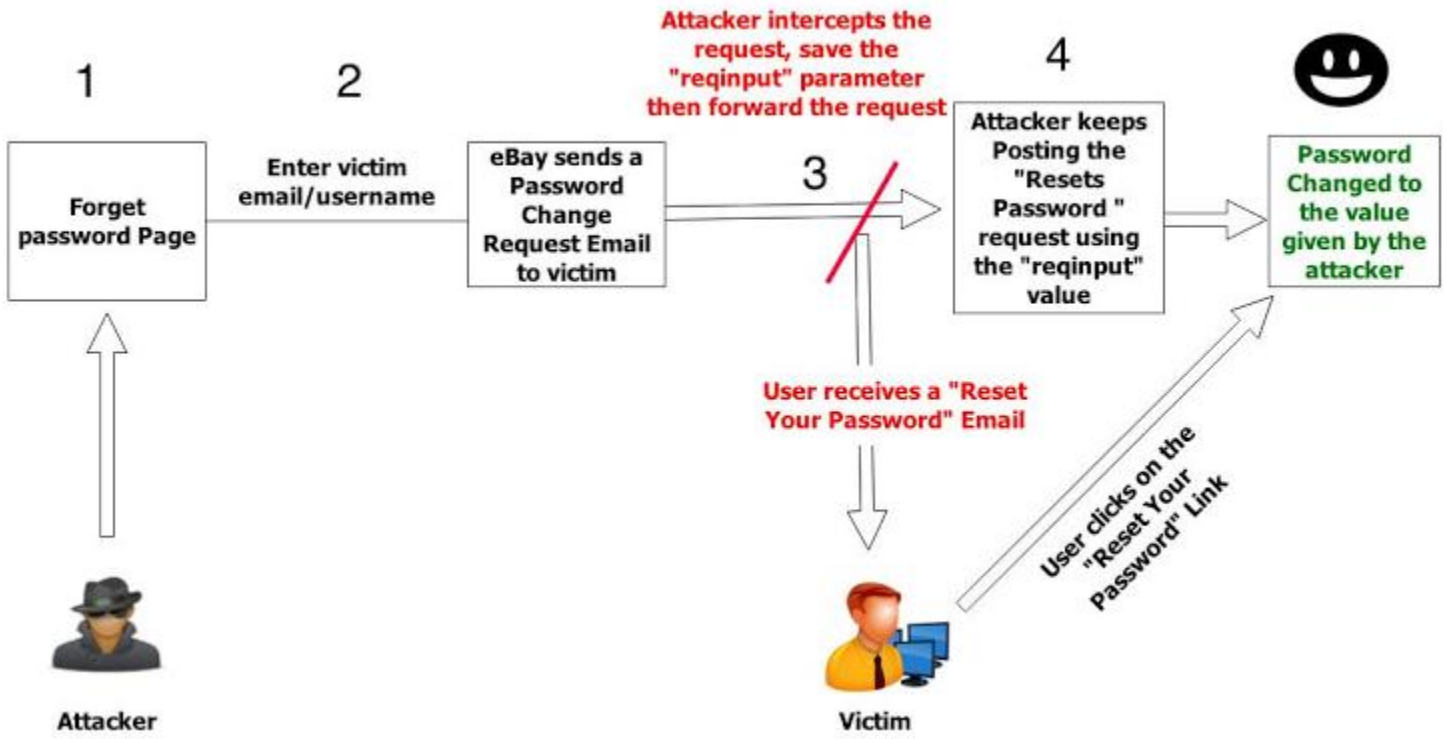
- Coding errors
  - E.g., use of gets() function and other unchecked buffers
- Logical errors
  - E.g., time of check to time of use (“TOUCTOU”)
  - Race condition where, e.g., authorization changes but

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*

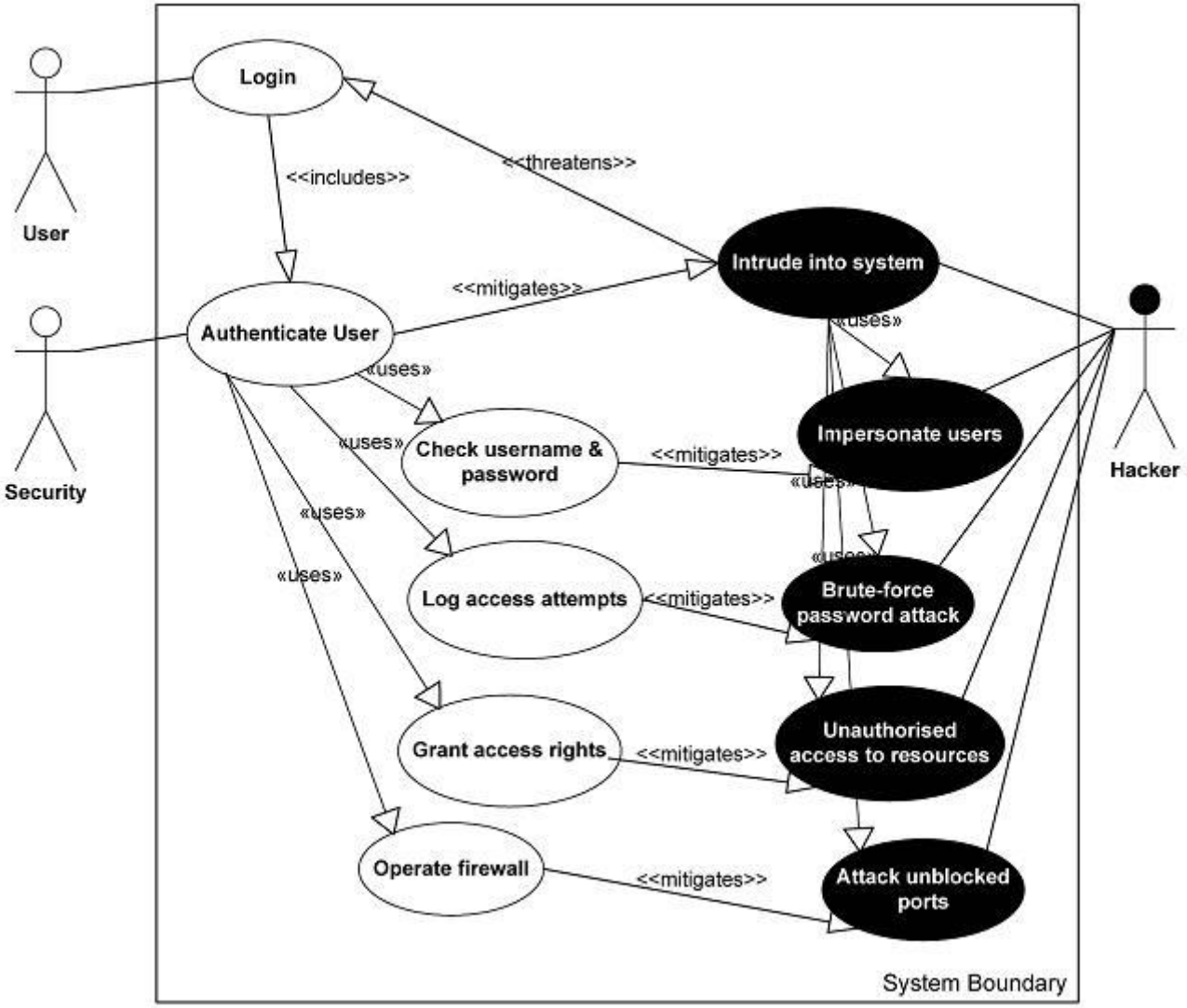
Prioritizing Risks and Tests

- 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

INF523: Computer System Assurance

Testing (continued)

Prof. Clifford Neuman

Lecture 7
9 October 2020

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

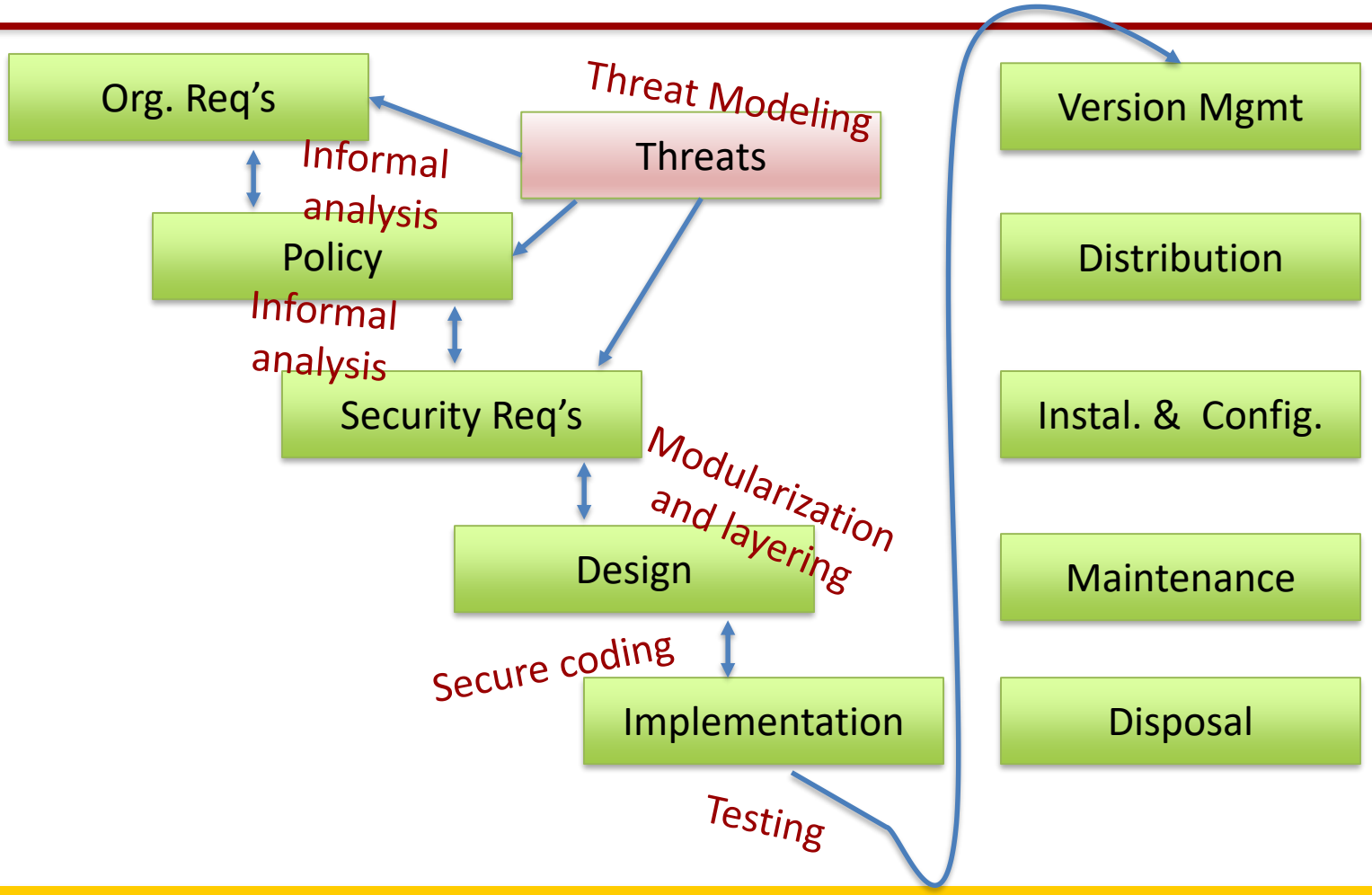
- Informal, non-rigorous, semi-systematic
  - Depends on skill of testers
- Not comprehensive
  - Proves at least one path, not all
  - When find one way to achieve a step, go on to next step
- Does not prove lack of path if unsuccessful
- But, performed by experts
  - Who are not the system developers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

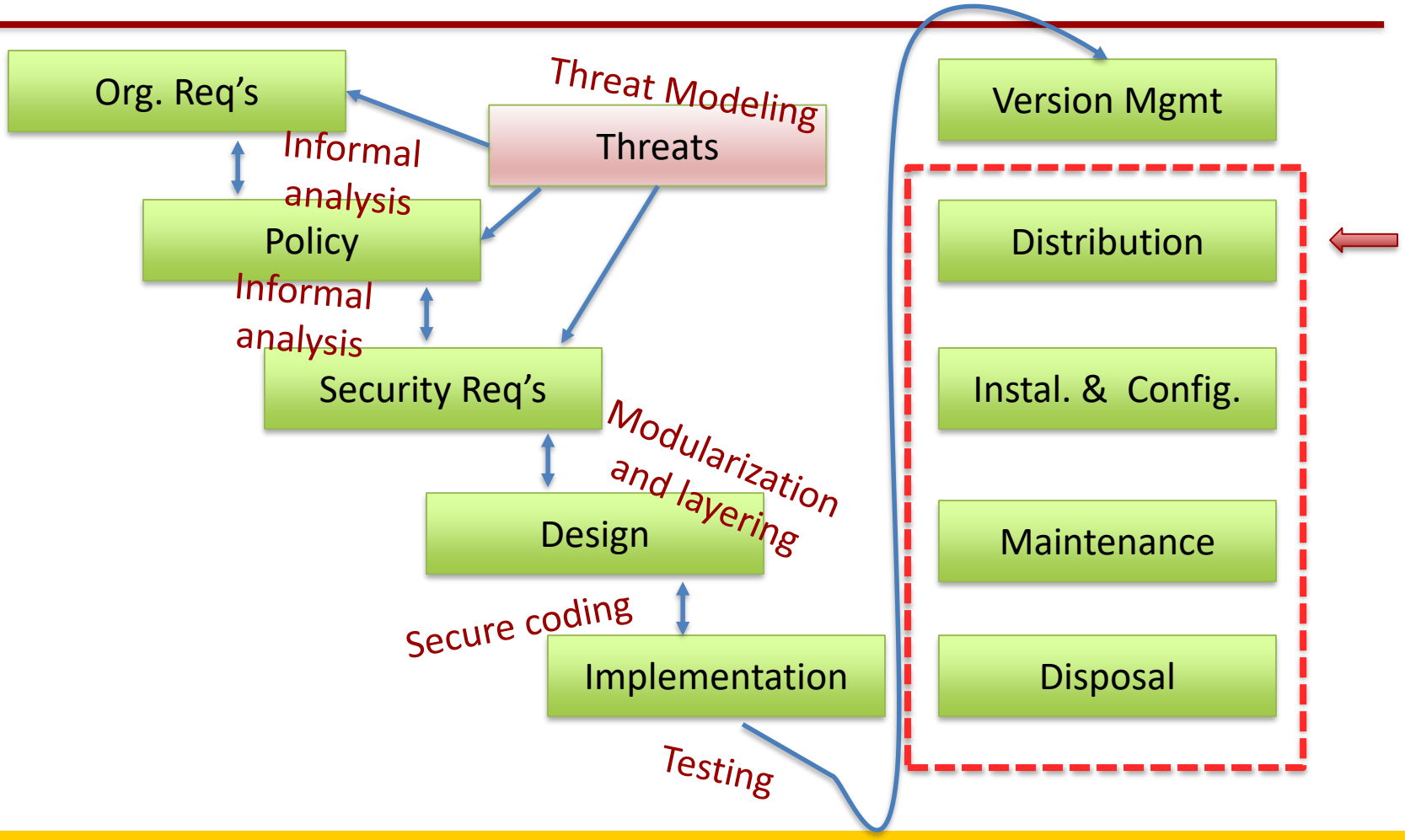


What's Left?

- 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

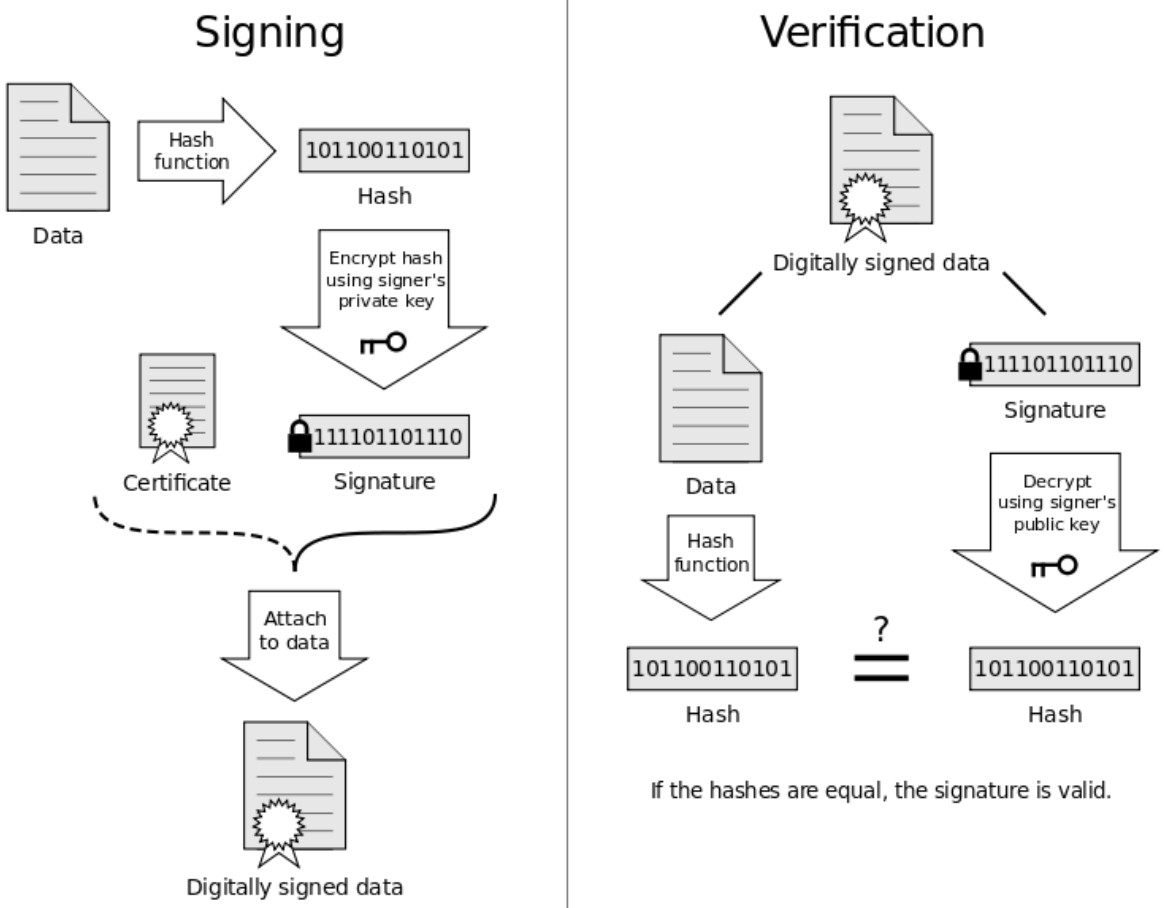
Are Checksums Reliable?

- 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

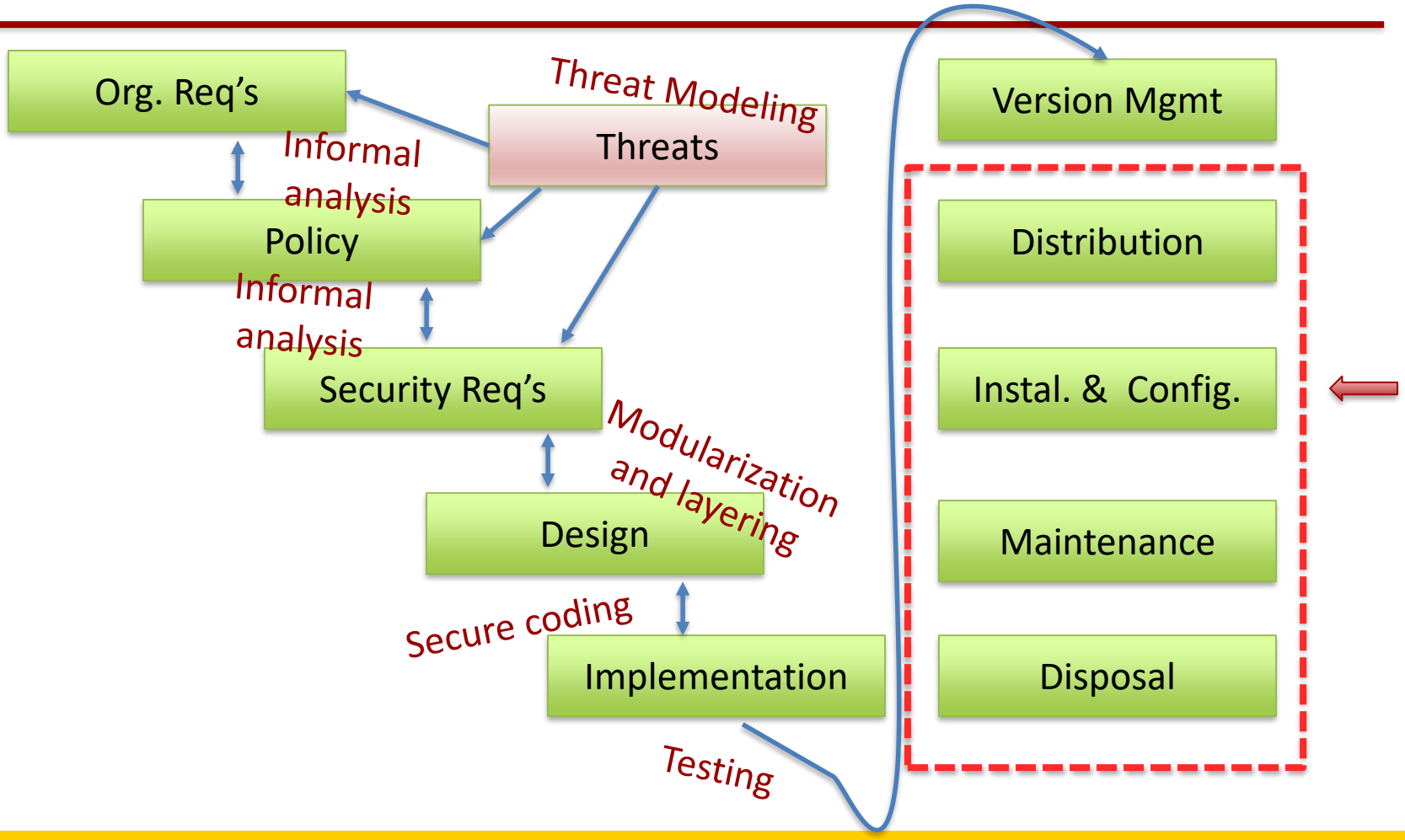
- E.g., GTNP FER (page 142)
  - Based on *cryptographic seals and data encryption*. All kernel segments are encrypted and sealed. Formatting information on distributed volumes is sealed but not encrypted. Keys to check seals and decrypt are shipped separately [i.e., sent out of band; no certification 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

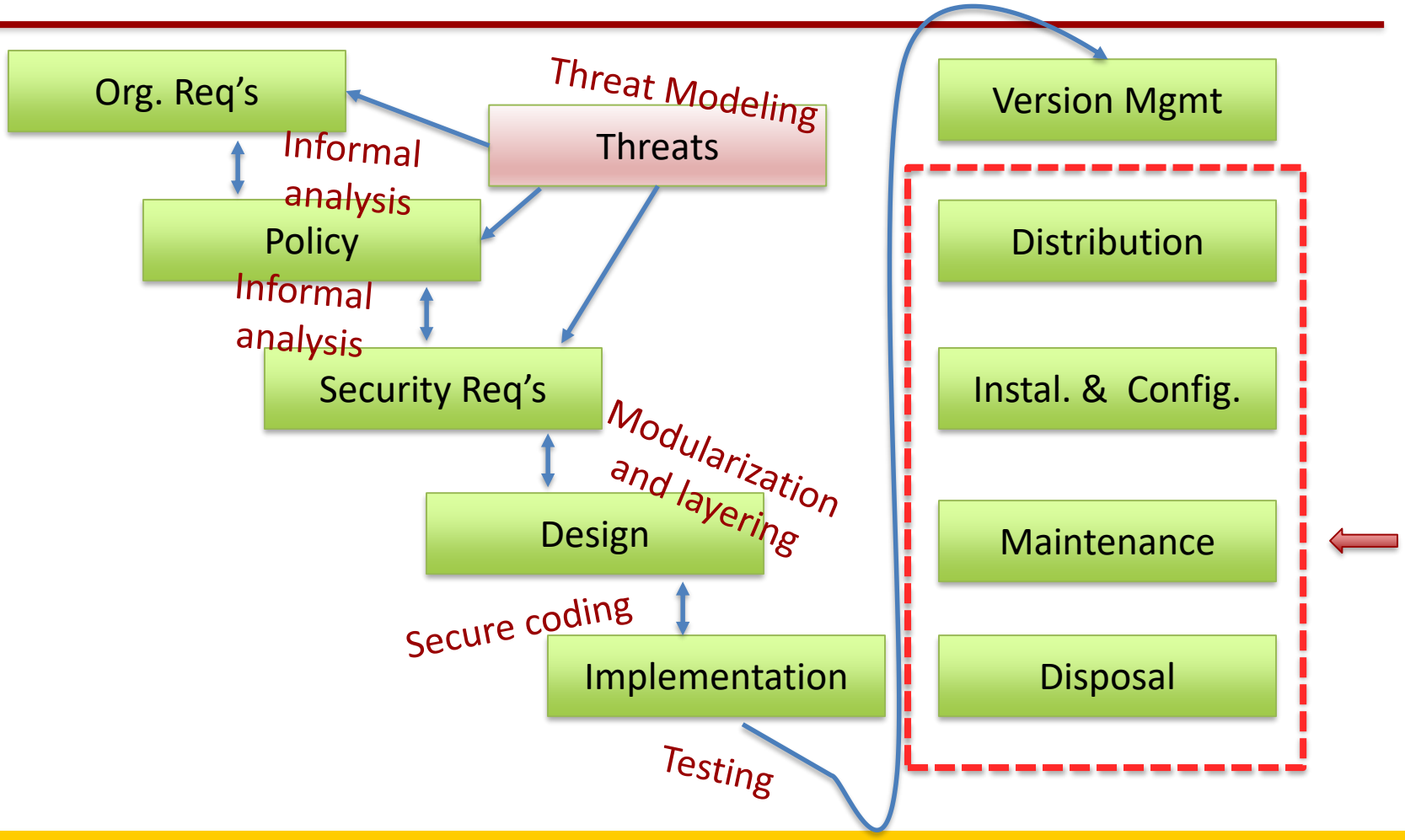
- Evaluated systems are certified
  - Under specific environmental criteria
  - (e.g., for TCSEC, criteria listed in Trusted Facility Manual)
- But environmental criteria must be satisfied for accreditation
  - E.g., security only under assumption that network is physically isolated
  - If instead use 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

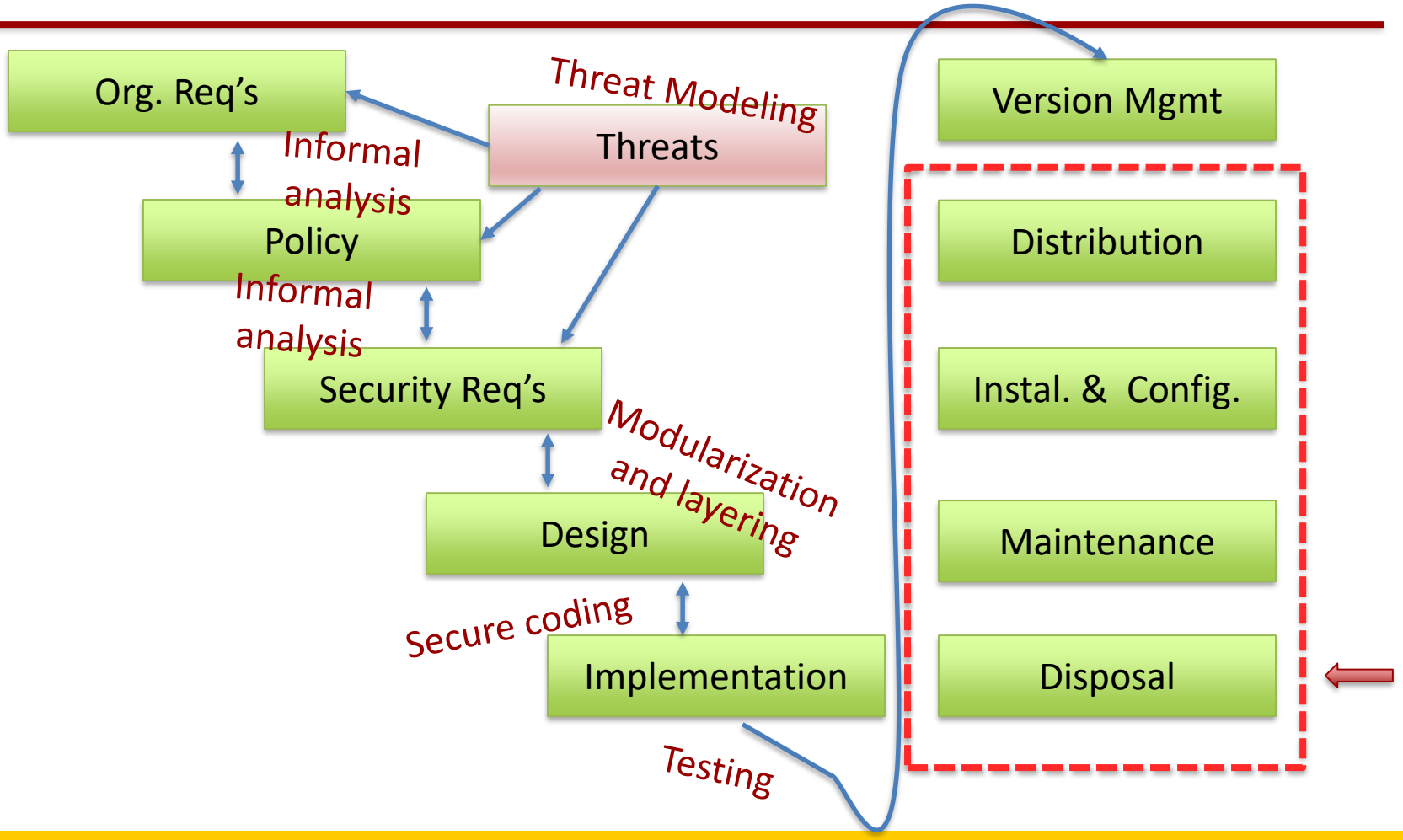
- To manage workload:
  - Determine key assets
  - Prioritize alerts

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*

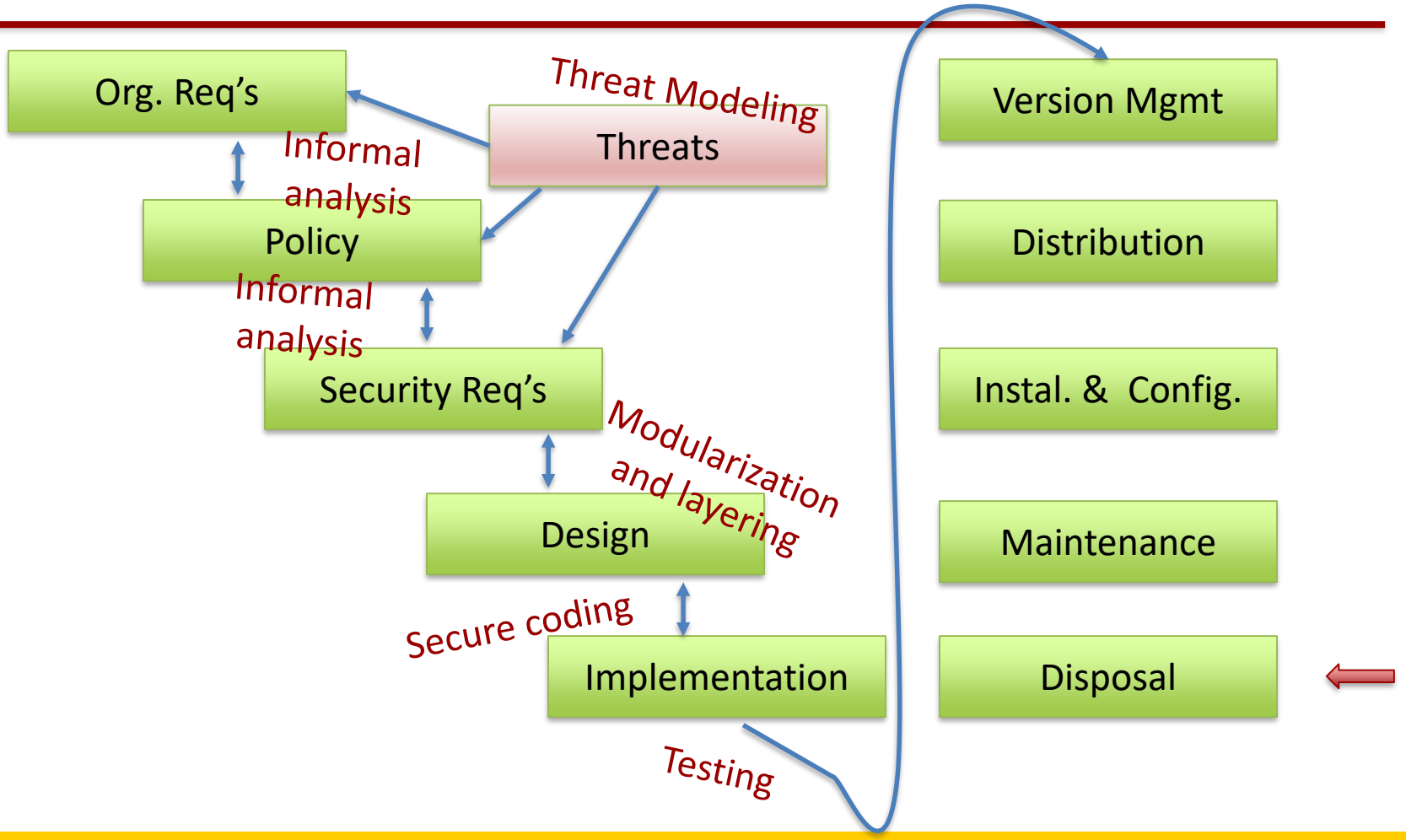
Secure Disposal

- 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

Mid-Term Exam is Friday October 23rd

Exam will be 100 minutes, from 1PM to 2:40PM.
  - A lecture will follow from 3PM to 4:20PM
- Format of the exam
  - The exam is open book, open note and online
  - 2016 and 2017 exams are posted on the class website http://ccss.usc.edu/523
    - The 2018 exams will be posted soon
  - Material to be covered will be the start of the semester through Secure Operation
  - You are responsible for material discussed in lecture, and also all assigned readings.

Outline of Pre-Mid-Term Course Material



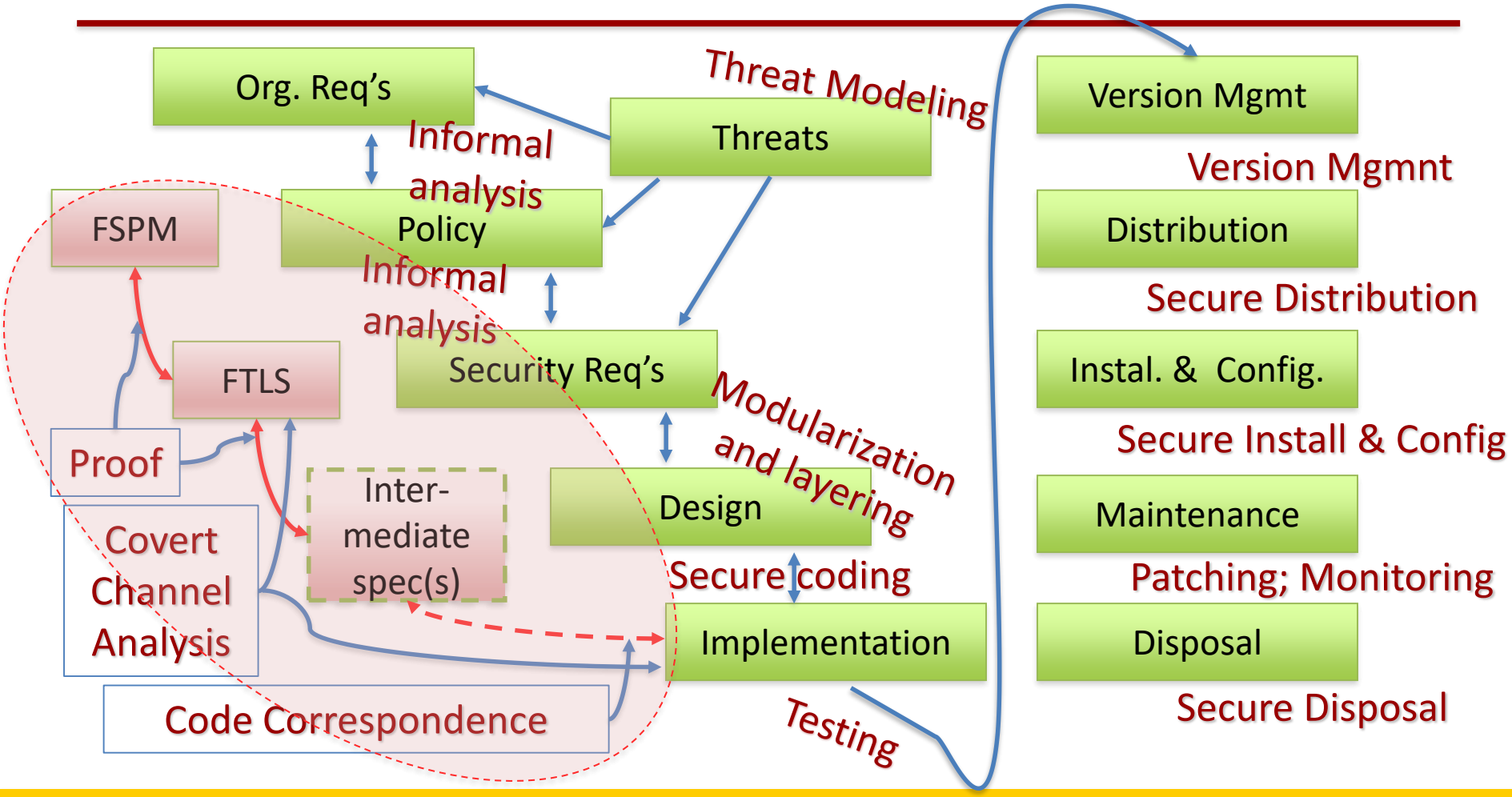
Reading for Next Time

- Jonathan K. Millen. 1976. Security Kernel validation in practice. *Comm. ACM* 19, 5 (May 1976), 243-250. DOI=10.1145/360051.360059
- T. Levin, S. Padilla, and R. Schell, Engineering Results from the A1 Formal Verification Process, in *Proceedings of the 12th National Computer Security Conference*, Baltimore, Maryland, 1989. pp. 65-74

Reading for This Time

- Bishop, pp. 545-551
  - A Specifier's Introduction to Formal Methods, Jeannette M. Wing
  - Formal Specifications, a Roadmap, Axel van Lamsweerde

"Assurance Waterfall"



Formal Methods

- *Formal* means *mathematical*
- Tools and methods for reasoning about correctness
  - *Correctness* means system design satisfies some properties
  - Security, but also safety and other types of properties
- Useful way to think completely, precisely, and unambiguously about the system
  - Help delimit boundary between system and environment
  - Characterize system behavior under different conditions
  - Identify assumptions
  - Identify necessary invariant properties
- Often find flaws just from writing formal specification

Informal vs. Formal Specifications

- Informal
  - Human language, descriptive
  - E.g., "The value of variable *x* will always be less than 5"
  - Often vague, ambiguous, self-contradictory, incomplete, imprecise, and doesn't handle abstractions well
    - **All of which can easily lead to unknown flaws**
  - But, relatively easy to write
- Formal
  - Mathematical
  - E.g., $\forall t. \forall x. (t >= x \wedge (sys\_init(x))) \rightarrow x(t) < 5$
  - Easily handles abstractions, concise, non-ambiguous, precise, complete, etc.
  - But, requires lots of training and experience to do right

Always? What about before the system is (re)initialized?

Formal vs. "Informal" Verification

- "Informal" verification:
  - **Testing** of various sorts
    - Finite, can never can be complete, only demonstrates cases
- Formal verification:
  - Application of formal methods to "prove" a design satisfies some requirements (properties)
    - A.k.a. "demonstrating correctness"
  - Can "prove" a system is secure
    - I.e., that the system design satisfies some properties that are the definition of "security" for the system
    - I.e., that a system satisfies the security policy

Some Uses of Formal Methods

- Prove certain properties
  - E.g., invariants, such as BLP always in secure state
- Prove that certain combinations of states never occur
- Prove value of certain variable never exceeds bounds
- Prove absence of information flows
  - E.g., for transitive closure of shared resource matrix

- Very widely used for hardware
- Not currently widely used for software

Types of Formal Verification

- Theorem proving (semi-automated)
  - Proving of mathematical theorems
    - E.g., that FTLS satisfies FSPM
  - Complex, prone to error if done totally by hand
  - Must use automated (mechanized) theorem proving tools
    - Can solve some simple proofs automatically using heuristics
    - Non-trivial proofs require lots of human input
- Model checking (automated)
  - Specify system as FSM, properties as valid states
    - Exhaustively compare possible system states to specification to show all states satisfy spec
  - May run a long time for complex state
    - Use heuristics in advance to prune state space

Steps in Security Formal Verification

1. Develop FSPM (e.g., BLP)
2. Develop Formal Top-Level Spec (FTLS)
   - Contrast with Descriptive Top-Level Specification (DTLS)
     - Natural language, not mathematical, specification
3. Proof (formal or informal) that FTLP satisfies FSPM
4. (Possibly intermediate specs and proofs)
   - At different levels of abstraction
5. Show implementation "corresponds" to FTLS
   - Code proof beyond state of the art (but see https://sel4.systems/)
   - Generally informal arguments
   - Must show how every part of code fits

Attributes of Formal Specifications

- States what system does, but not how
  - I.e., like module interfaces from earlier this semester
  - Module interfaces are (probably informal) specifications
- Precise and complete definition of effects
  - Effects on system state
  - Results returned to callers
  - All side-effects, if any
- Not the details of *how*
  - Not how the data is stored, etc.
  - I.e., abstraction
- Formal specification language is not code

Parts of a Formal Specification

- Basic types of entities
  - E.g., in BLP, subjects and objects, access modes
- State variables
  - E.g., b, M, f, and H
- Defined concepts and relations
  - In terms of entities and state variables
  - E.g., dominance, SSC, *-property
- Operations
  - E.g., get_read
  - Relations of inputs to outputs – e.g., R, D, W
  - State changes

Bell-La Padula Formal Policy Model

- From "Secure Computer System: Unified Exposition and Multics Interpretation", Appendix

Rule 1 (R1): get-read

Domain of R1: all $R_k = (g, S_i, O_j, r)$ in $R^{(1)}$. (Denote domain of R_i by dom (R_i).)

Semantics: Subject S_i requests access to object O_j in read-only mode (r).

*-property function: $*1\ (R_k, v) = \text{TRUE} \Leftrightarrow f_c(S_i) \propto f_o(O_j)$.

The rule:

$$R1(R_k,v) = \begin{cases} (?, v) & \text{if } R_k \notin \text{dom } (R1); \\ (\text{yes}, (b \cup (S_i,O_j,r)^{\dagger}, M, f, H)) & \text{if } [R_k \in \text{dom } (R1)] \ \& \ [r \in M_{ij}] \ \& \ [f_s(S_i) \propto f_o(O_j)] \ \& \ [S_i \in S_T \text{ or } *1(R_k,v)]; \\ (\text{no}, v) & \text{otherwise.} \end{cases}$$

New state

Error if invalid call or not a get_read call, and no change to state

Discretionary and mandatory policy requirements

If valid get_read call but does not satisfy discretionary or mandatory policy, no change to state

Algorithm for R1:

if $R_k \notin$ dom (R1) then $R1(R_k, v) = (?, v)$; else if $r \in M_{ij}$ and $\langle [S_i \in S' \text{ and } *1(R_k, v)] \text{ or } [S_i \in S_T \text{ and } f_s(S_i) \propto f_o(O_j)] \rangle$
then $R1(R_k, v) = (\text{yes}, (b \cup (S_i, O_j, r), M, f, H))$;
else $R1(R_k, v) = (\text{no}, v)$;

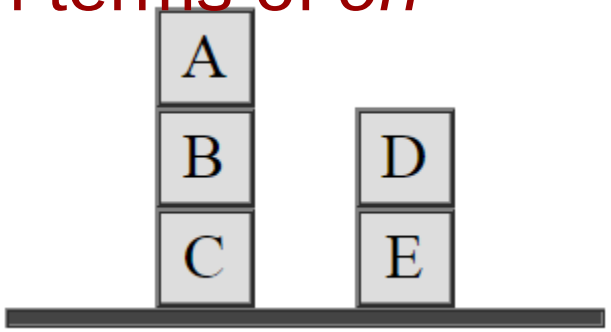
end;

Formal Top-Level Specification

- Represents interface of the system
  - In terms of exceptions, error messages, and effects
  - Must be shown to accurately reflect TCB interface
  - Include HW/FW operations, if affect state at interface
  - TCB “instruction set” consists of HW instructions accessible at interface and TCB calls
- Describe external behavior of the system
  - precisely,
  - unambiguously, and
  - in a way amenable to computer processing for analysis
  - Without describing or constraining implementation

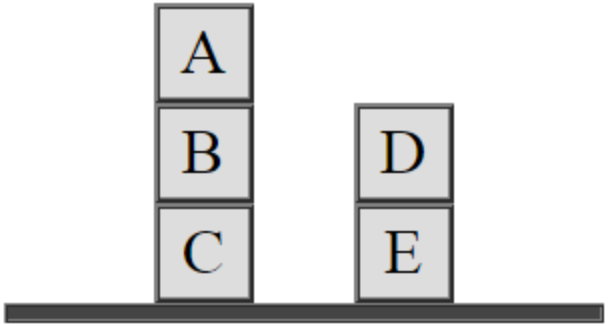
Creating a Formal Specification

- Example, “blocks world”
- 5 objects *{a,b,c,d,e}*
  - Table is not an object in this example
- Relations *{on,above,stack,clear,ontable}*
- *on(a,b); on(b,c); on(d,e)*
- ¬*on(a,a)*, ¬*on(b,a)*, … , etc.
- Define all of the other relations in terms of *on*



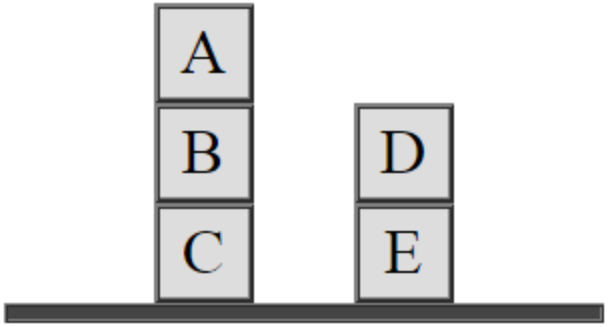
Creating a Formal Specification

- Define all of the other relations in terms of *on*
- ∀y.(clear(y) ⇔
  ¬∃x.on(x,y))
- ∀*x*.(*ontable*(*x*) ⇔
  ¬∃*y*.*on*(*x*,*y*))
- ∀x.∀y.∀z.(stack(x,y,z) ⇔
  on(x,y) ∧ on(y,z))
- ∀*x*.∀*z*.(*above*(*x*,*z*) ⇔
  on(*x*,*z*) ∨ ∃*y*.(*on*(*x*,*y*) ∧ *above*(*y*,*z*)))
  - We are missing something for *above*. What is it?
    ∀x.¬above(x,x)



Alternative Specification

- Define all of the other relations in terms of *above*
- ∀x.(ontable(x) ⇔ ¬∃y.above(x,y))
- ∀x.(clear(x) ⇔ ¬∃y.above(y,x))
- ∀x.∀y.(on(x,y) ⇔
  above(x,y) ∧ ¬∃z.(above(x,z) ∧
  above(z,y))
- What about *stack*?
  - Can define in terms of on, as before
- Need other axioms about *above*:
- ∀x.¬above(x,x)
- ∀x.∀y.∀z. above(x,y) ∧ above(y,z) => above(x,z)
- ∀x.∀y.∀z. above(x,y) ∧ above(x,z) =>
  y=z \/ above(y,z) \/ above(z,y)
- ∀x.∀y.∀z. above(y,x) ∧ above(z,x) =>
  y=z \/ above(y,z) \/ above(z,y)



Observation

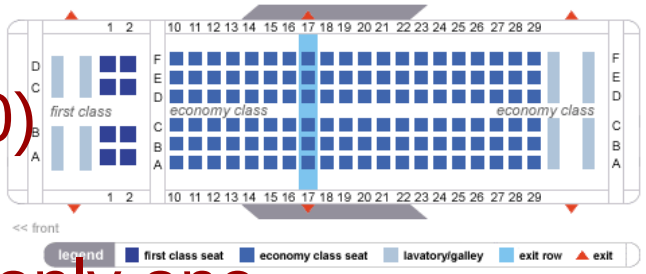
- Many ways to specify the same system
- Not every way is equally good
- If pick less good way, may create lots of complexity
- E.g., consider how to specify a FIFO queue
  1. Infinite array with index of current head and tail
     - Not very abstract – specifies "how"
  2. Simple, recursive, add and remove functions and axioms
     - E.g., ∀x. remove(add(x,EMPTY)) = x
- The first is tedious to reason with
  - Lots of "overhead" to keep track of indexes
- The second is easy and highly automatable

Formal System Specifications

- Previous example used first-order logic (FOL)
  - ∀ and ∃
- For complex systems, FOL may not be enough
- Want "higher-order" logic (HOL), which can take functions as arguments
- E.g., [Rushby PVS phone book example]
  - http://www.csl.sri.com/papers/wift-tutorial/slides.pdf

Homework (due 11/1)

- Write a formal spec for seating in an airplane:



- An airplane has 100 seats (1..100)
- Every passenger gets one seat
- Any seat with a passenger holds only one passenger
- The state of a plane P is a function [N -> S]
  - Maps a passenger name to a seat number
- Two functions: assign_seat and deassign_seat
- Define the functions
- Show some lemmas that demonstrate correctness

Start of Homework Solution

- Types:
  - N : type (of passenger)
  - S : type (of seat number)
  - A : type (of airplane function) [N -> S]
  - e0 : N (represents an empty seat)

- Variables:
  - nm : var N (a passenger)
  - pl : var A (an airplane function)
  - st : var S (a seat number)

What you Need to Do

1. Define the axioms for the two functions:
   - assign_seat : [A x N x S -> A]
   - deassign_seat : [A x S -> A]
2. Be careful that the spec covers all requirements:
   - Can someone have “e0” as their seat number?
   - Can a passenger have more than one seat?
   - Can a seat have more than one passenger?
3. Identify some lemmas that demonstrate that the system specification describes what is intended and sketch the proof

Formal Verification is Not Enough

- Formal verification complements, but does not replace testing (informal verification)
- Requires abstraction which
  - May leave out important details (stuff missing)
  - May make assumptions that code does not support (extra stuff)
- Even if "proven correct", may still not be correct

- "Beware of bugs in the above code; I have only proved it correct, not tried it." -Knuth

INF523: Assurance in Cyberspace as Applied to Information Security

Case Studies of Formal Specification and Proofs

Reading for This Class

- Jonathan K. Millen. 1976. Security Kernel validation in practice. *Commun. ACM* 19, 5 (May 1976), 243-250. DOI=10.1145/360051.360059
- T. Levin, S. Padilla, and R. Schell, Engineering Results from the A1 Formal Verification Process, in *Proceedings of the 12th National Computer Security Conference*, Baltimore, Maryland, 1989. pp. 65-74

DEC PDP 11

- Sold by DEC
- 1970s-1990s
- Most popular minicomputer ever
- Smallest mini-computer for a decade that could run Unix

Millen: PDP 11/45 Proof of Correctness

- Proof of correctness for PDP 11/45 security kernel
- Correctness defined as proper implementation of security policy model (BLP)
- Security policy model defined as set of axioms
  - Axioms are propositions from which properties are derived
  - E.g., in BLP, SSC and *-property
- Proof is that all operations available at the interface of the system preserve the axioms
- Also considered covert storage channels
  - Method did not address timing channels

Millen: PDP 11/45 Proof of Correctness

- Security policy model defined as set of axioms
  - Simple security condition
    - If a subject has “read” access to an object, level of subject dominates level of object
  - *-property
    - If a subject has “read” access to one object and “write” access to a second object, level of second object dominates level of first object
  - Tranquility principle for object levels
    - Level of active object will not be changed
  - Exclusion of read access to inactive objects
  - Rewriting (scrubbing) of objects that become active

Layers of Specification and Proof

- Four stages
- Each stage more detailed and closer to machine implementation than the one before

1. FSPM (BLP)
2. FTLS – The interface of the system
   - Includes OS calls and
   - PDP 11/45 instructions available outside kernel
3. Algorithmic specification – High-level code that represents machine language
   - Semantics of language must be well-understood
4. Machine itself: Running code and HW

Why Four Proof Stages?

- Simplify proof work
- Big jump from machine to FSPM
  - FSPM has subjects, objects, *-property, …
  - Machine has code and hardware
- Intermediate layers are closer to each other
- First prove FTLS is valid interpretation of FSPM
- Then further proofs only need to show that lower stages implement FTLS
  - Lower-level proofs don't need abstractions of subjects and objects and *-property

Stages 1 and 2 Specification Format

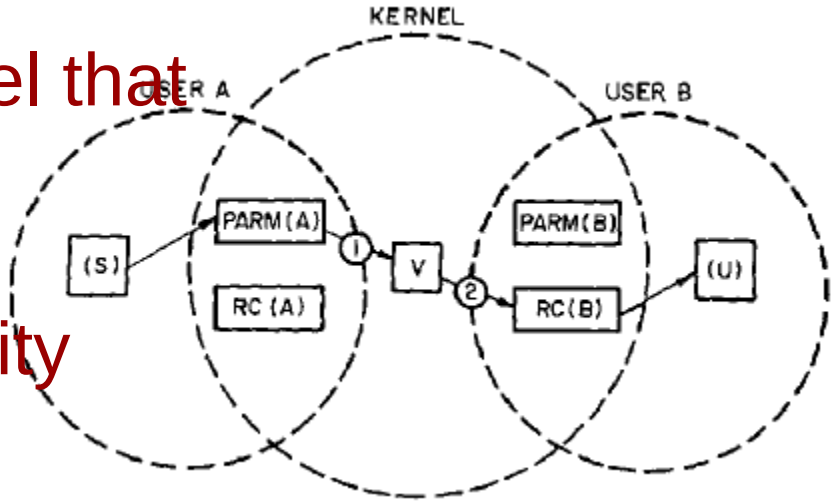
- Both FSPM and FTLS are state machines
  - States and transitions
  - E.g., BLP state is (b, M, f, H)
  - FTLS transitions:
    - Create (activate) object
    - Delete (deactivate) object
    - Get access to an object for a subject
    - Release access to an object for a subject
    - Put a subject in an object's ACL
    - Remove a subject from an object's ACL
    - PDP-11/45 instructions available at interface

V- and O-functions

- State variables and kernel operations are functions
  - State variables are represented as V-functions
    - All V-functions are references to objects
  - Operations are O-functions
    - By subjects to objects
    - Accesses are due to O-function executions
- O-functions have effects on state variables
  - Indicated by values of V-functions before and after
  - E.g., $\neg$`(PS_SEG_INUSE(TCP,dseg))` $\wedge$ `RC(TCP) = NO`
  - I.e., if the object is not in use by the subject then the return code is “NO”

“Shared Resource Problems”

- Covert storage (not timing) channels
- User A at high level
- Modifies kernel state variable V
- User B at low level
- Receives value from kernel that was influenced by V
- Detect by assigning security level to internal variables like V



Proof Example in Paper

- Verification that DELETE enforces *-property
- Original spec at right
- Effect statements are labeled “A”, “B”, etc.
  - Used to simplify statement form for proof
- x = ‘y’ means subject has read access to y and write access to x

```
   Function:  DELETE
   parameters:  dseg,entry
   let:  dpn = PS_SEG(TCP,dseg)
   effect:
A    if not PS_SEG_INUSE(TCP,dseg) or
        not AST_WAL(dpn,TCP)  or
B       AST_TYPE(dpn,TCP) ≠ DIRECTORY  or
          ´DIR_SIZE´(dpn,entry) = 0
C    then RC(TCP) = NO
     else RC(TCP) = YES  and
          DIR_SIZE(dpn,entry) = 0  and
          if ´DIR_ACL_HEAD´(dpn,entry) ≠ 0
D         then ACL_CHAIN(dpn,FINDEND(dpn,entry))
                 = ´ACL_CHAIN´(dpn,0)  and
                 ACL_CHAIN(dpn,0) = ´DIR_ACL_HEAD´(dpn,entry)
          end
          and (¥pn)
E(pn)     if pn ≥ dpn.entry
          then if pn = apn.aentry
                  and DIR_TYPE(apn.aentry) = DIRECTORY
               then (¥bentry) DIR_SIZE(pn,bentry) = 0 and
                    (¥acle) ACL_CHAIN(pn,acle)
F(pn)                        = acle + 1 (mod acle_max + 1)
               else VM(pn) = 0
               end  and
               WAIT_SET(pn) = ∅ and
               SMFR_COUNT(pn) = 1
               and (¥aproc)
```

Abbreviated DELETE Specification

- Statements abstracted to show structure
- Bottom version is in "conjunctive form"
- "Else" sometimes replaced by negation of the "If" condition
- Statements in form **`if f then g else h end`** sometimes converted to **`(f ∧ g) ∨ (¬f ∧ h)`**

```
If A pr B
then C
else D and
          (¥pn)
          if E(pn)
          then F(pn) and
                    (¥aproc)
                    if G(pn,aproc)
                    then H(pn,aproc)
                    end
          end and
          (¥aproc)
          if J(aproc)
          then K(aproc)
          end
end
```

```
          if A then C else if ~A and B then C else D end
and (¥pn) if ~A and ~B and E(pn) then F(pn) end
and (¥aproc)
          if ~A and ~B
          then (¥pn)
                    if E(pn) and G(pn,aproc)
                    then H(pn,aproc)
                    end
          end
and (¥aproc) if ~A and ~B and J(aproc) then K(aproc) end
```

Proof Technique: Security Levels

- Object levels
  - Level based on pathname *pn* of object in hierarchy
  - Level of object at pathname *pn* is *L(pn)*
  - V-functions that take *pn* as parameter have level *L(pn)*
  - Constant V-functions (no parameters) have sys-low level
- Level of subject with process number *proc* is *PL(proc)*
  - *PL(proc)* is level where subject can both read and write
  - V-functions for subjects (i.e., that read state values for processes, as opposed to system state) have level *PL(proc)*
  - O-functions have process numbers as parameters
  - O-functions and their parameters have level *PL(proc)*

Property Cases and Security Levels

```
   Function:  DELETE
   parameters:  dseg,entry
   let:  dpn = PS_SEG(TCP,dseg)
   effect:
A    if not PS_SEG_INUSE(TCP,dseg) or
B       not AST_WAL(dpn,TCP)  or
        AST_TYPE(dpn,TCP) ≠ DIRECTORY  or
          'DIR_SIZE'(dpn,entry) = 0
C    then RC(TCP) = NO
     else RC(TCP) = YES  and
          DIR_SIZE(dpn,entry) = 0  and
          if 'DIR_ACL_HEAD'(dpn,entry) ≠ 0
D         then ACL_CHAIN(dpn,FINDEND(dpn,entry))
                 = 'ACL_CHAIN'(dpn,0)  and
                 ACL_CHAIN(dpn,0) = 'DIR_ACL_HEAD'(dpn,entry)
          end
          and (¥pn)
E(pn)     if pn ≥ dpn.entry
          then if pn = apn.aentry
                  and DIR_TYPE(apn.aentry) = DIRECTORY
               then (¥bentry) DIR_SIZE(pn,bentry) = 0 and
                    (¥acle) ACL_CHAIN(pn,acle)
F(pn)                     = acle + 1 (mod acle_max + 1)
               else VM(pn) = 0
               end  and
               WAIT_SET(pn) = ∅ and
               SMFR_COUNT(pn) = 1
               and (¥aproc)
```

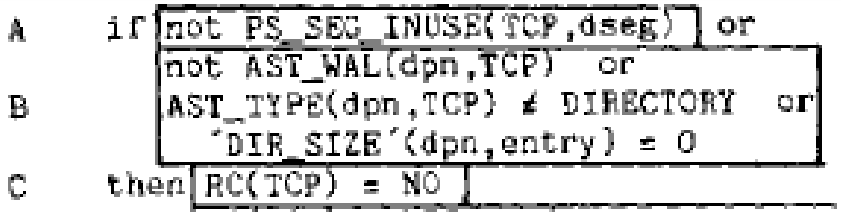
Table I. *-Property Cases.

Subject/ Case	Condi- tions	Ef- fects	Read Levels	Write Levels
S_1/I	A	C	$PL(TCP)$	$PL(TCP)$
S_1/II	$\bar{A}\ B$	C	$PL(TCP), L(dpn)$	$PL(TCP)$
S_1/III	$\bar{A}\ \bar{B}$	D	$PL(TCP), L(dpn)$	$PL(TCP), L(dpn)$
$S_2(pn)$	$\bar{A}\ \bar{B}\ E$	F	$PL(TCP), L(dpn), L(apn)$	$L(pn)$
$S_3(aproc)$	$\bar{A}\ \bar{B}\ E\ G$	H	$PL(TCP), L(dpn), PL(aproc), L(pn)$	$PL(aproc)$
$S_4(aproc)$	$\bar{A}\ \bar{B}\ J$	K	$PL(TCP), L(dpn), PL(aproc)$	$PL(aproc)$

Table II. Security Levels of V-Function References in Labeled Statements.

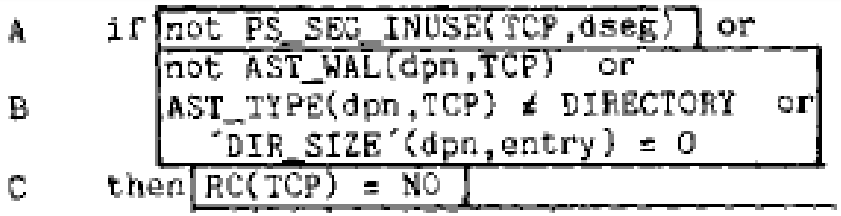
Statement	Read Levels	Write Levels
A	$PL(TCP)$	—
B	$PL(TCP), L(dpn)$	—
C	—	$PL(TCP)$
D	$PL(TCP), L(dpn)$	$PL(TCP), L(dpn)$
E (pn)	$PL(TCP)$	—
F (pn)	$L(apn)$	$L(pn)$
G $(pn, aproc)$	$PL(aproc)$	—
H $(pn, aproc)$	$PL(aproc), L(pn)$	$PL(aproc)$
J $(aproc)$	$PL(aproc)$	—
K $(aproc)$	$PL(aproc)$	$PL(aproc)$

Proof Case Example: Explanation of A, B, C



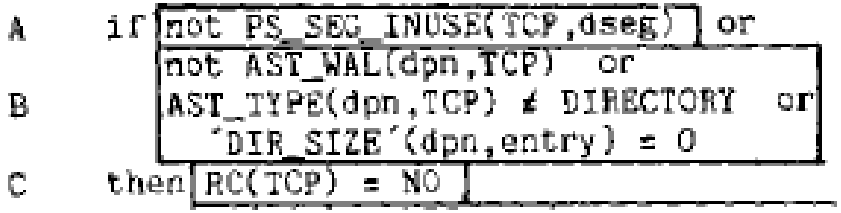
- Delete(dseg,entry)
  - Erases a segment from a directory
  - *Dseg* is directory segment, *entry* is index in directory
- (A) If the local segment number is not in use, or
- (B) If the process does not have write access to the directory or the directory entry is empty
- (C) Return “NO”

Proof Case Example: Function Explanation



- *dpn* is abbreviation for *PS_SEG(TCP,dseg)*
  - Directory path name
- **Active segment table (AST) is part of global state**
- **AST entry numbers must be invisible to avoid channel**
- PS_SEG(TCP,dseg) maps process-local segment numbers to active segment table AST entries
- PS_SEG_INUSE indicates whether or not an element in PS_SEG is in use
- AST_WAL is active segment table write access list

Proof Case Example: Proof Goal



Subject/ Case	Condi- tions	Ef- fects	Read Levels	Write Levels
S_1/I	A	C	$PL(TCP)$	$PL(TCP)$
S_1/II	$\bar{A}$ B	C	$PL(TCP), L(dpn)$	$PL(TCP)$

- Second case: ¬A ∧ B ∧ C
  - PS_SEG_INUSE(TCP,dseg) = TRUE and
  - AST_WAL(dpn,TCP) = FALSE or… or …
- Prove second case does not violate *-property
- Process is reading from the directory and writing to the response code, so must prove: L(dir) ≤ L(RC)
- I.e., **L(dpn) ≤ PL(TCP)**

Proof Case Example: Proof

Two general relations suffice to complete the proof of the inequality:

R1: $(\forall proc)(\forall seg)$ if $PS\_SEG\_INUSE(proc, seg) = TRUE$
then $AST\_CPL(PS\_SEG(proc,seg), proc) = TRUE$.

R2: $(\forall pn)(\forall proc)$ if $AST\_CPL(pn, proc) = TRUE$
then $L(pn) \leq PL(proc)$.

- R1: If PS_SEG_INUSE is true then it must be the case that the process is in the AST "connected process list" (CPL) for that segment
- R2: If a process is in the AST_CPL then it must be the case that **`L(pn) ≤ PL(proc)`**
- Relations proven inductively over all operations

GEMSOS Verification

- PDP 11/45 verification before TCSEC
- GEMSOS developed to meet TCSEC class-A1
- Gemini Trusted Network Processer (GTNP) developed to be TNI M-component (multilevel)
  - Based on GEMSOS
- Evaluation on GTNP
- This paper, however, about GEMSOS TCB only

GEMSOS A1 Formal Verification Process

- FSPM, FTLS written in InaJo specification language
- BLP BST proven using FDM theorem prover
  - FSPM was not “pure” BLP, but the GEMSOS interpretation of BLP
- Conformance of FTLS to model also proven
- FTLS also used for code correspondence and covert storage channel analysis

Value of Formal Verification Process

- "Provided formulative and corrective guidance to the TCB design and implementation"
- I.e., just going through the process helped prevent and fix errors in the design and implementation
- Required designers/developers to use clean designs
  - So could be more easily represented in FTLS
  - Prevents designs difficult to evaluate and understand

GEMSOS TCB Subsets

- Ring 0: Mandatory security kernel
- Ring 1: DAC layer
- Policy enforced at TCB boundary is union of subset policies

Each Subset has its own FTLS and Model

- Each subset was verified through a separate Model and FTLS
- Separate proofs, too
- TCB specification must reflect union of subset policies

Where in SDLC?

- Model and FTLS written when interface spec written
- Preliminary model proofs, FTLS proofs, and covert channel analysis performed when implementation spec and code written
- Code correspondence, covert channel measurements, and final proofs performed when code is finished

- Formal verification went on simultaneously with development

Goal of GEMSOS TCB Verification

- To provide assurance that TCB implements the stated security policy
- Through chain of formal and informal evidence
  - Statements about TCB functionality
  - Each at different levels of abstraction
    - Policy
    - Model
    - Specification
    - Source
    - TCB itself (hardware and software)
  - Plus assertions that each statement is valid wrt next more abstract level

Chain of Verification Evidence

- Notes:
  - Model-to-policy argument is informal
  - Spec to model argument is both formal and informal
  - Source to spec argument is code correspondence
  - TCB to source means HW and compiler validation
    - I.e., object code
    - Considered "beyond state of the art"

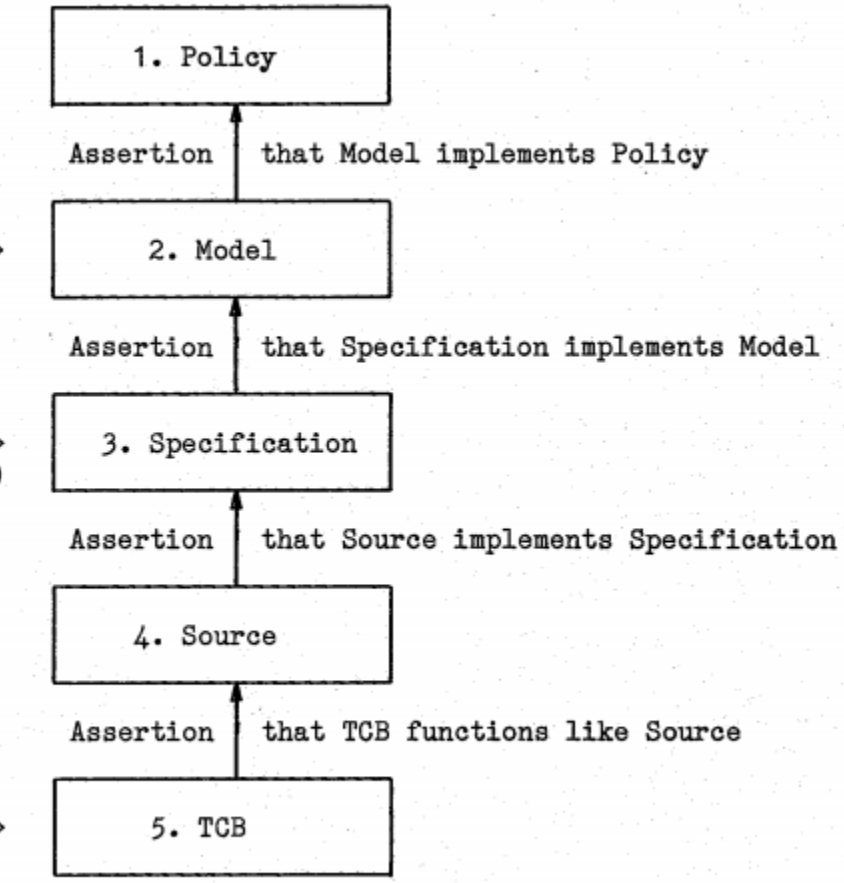


Figure 1. Chain of Verification Evidence

Model

- Mathematical statement of access control policy
- “Interpretation” of BLP
- Security defined as axioms
- Must prove all model transforms preserve axioms
  - SSC
  - *-property
  - (and probably others, as with PDP 11/45)
- Proof of model shows model upholds policy

Key Characteristic of Model

- Not just formal statement of policy or functions
- A model of a reference monitor
  - "Linchpin" of security argument
- If show that TCB satisfies reference monitor model then have shown that it is secure
  - Implies that anything outside TCB cannot violate policy
- What if did not model reference monitor?
  - May be "correct" wrt functions, but not necessarily secure

FDM “Levels”

- I.e., levels of abstraction
- InaJo language has method of formally mapping elements of one level to the elements of the next level
  - Top level: Model
  - Second level: FTLS

FTLSs

- One each for kernel and for TCB
- Exceptions, error messages, and effects visible at interface
- Transform for each call and
- Transforms for HW “read” and “write” operations
  - Other opcodes are irrelevant for access control security
- Proof maps each transform of FTLS to transform in model
- Each call specified as conditional statement
  - Last case contains any change statements
  - Exceptions specified in order of possible occurrence in code
    - Important for Covert Channel Analysis
  - Very end specifies everything else unchanged

Code Correspondence

- Three parts:
  1. Description of correspondence methodology
  2. Account of non-correlated source code
  3. Map between elements of FTLS and TCB code
- FTLS must accurately describe the TCB
- TCB must be valid interpretation of FTLS
- All security-relevant functions of TCB must be represented in FTLS
  - Prevent deliberate or accidental “trap door”

Example of Value of Formal Proof

- Subject is process/ring
- Subject can have range of access classes (trusted subject)
- Subjects in outer rings can have access class ranges "smaller" than subjects of the same process ir

```
Ring 3 Subject Range          READ WRITE
Ring 2 Subject Range      READ         WRITE
Ring 1 Subject Range  READ                 WRITE

(labels to the left dominate labels to the right)
(subject n is more privileged than subject n+1)
```

- Formal proof "stuck" trying to prove this

Example Formal Spec Detected Problem

- If range of subject in outer ring not within range of inner ring, move the outer ring access class to be within the range
- Original spec and code didn't take into account non-comparable access classes

```
dominates (ring_3_read_class, new_ring_2_read_class)
        then move (ring_3_read_class)
and
dominates (new_ring_2_write_class, ring_3_write_class)
        then move (ring_3_write_class)
```

- How to fix?

```
~dominates (new_ring_2_read_class, ring_3_read_class)
        then move (ring_3_read_class)
and
~dominates (ring_3_write_class, new_ring_2_write_class)
        then move (ring_3_write_class)
```

2nd Example Formal Spec Detected Problem

- Adjusting the access classes depends on the "move" function
- But it was found that the move function did not correctly ensure that the access class range of the outer ring subject was correct (i.e., that the "read" class dominated the "write" class)

Example of Value of Code Correspondence

- Code correspondence of kernel to spec found flaws in code:
  1. Access to segments in new child processes being checked using parent's privileges, not child's
  2. Segment descriptor in Local Descriptor Table not being set until segment brought into RAM
     - Not clear if this just meant inconsistent with model or was a real security problem

Example of Value of Covert Channel Analysis

- Two unexpected covert storage channels discovered
- Both related to “dismount_volume” call
- Dismount_volume used to (temporarily) remove set of segments from the segment structure
- Originally, any process whose access class range spanned range of volume could dismount the volume
- What if volume has only Unclassified segments?
  - TS process has made_known some of those segments
  - Unclassified process tries to dismount the volume, but gets an error message
- Fix?
  - Require caller’s range from volume low to sys-high

2nd Covert Channel

- Order of error checking
- Errors about volume could be reported to the calling subject even if subject did not have access to dismount the volume
- Fix: check label range before returning errors related to volume attributes