# Secure Software Systems

CYBR 200 | Fall 2018 | University of the Pacific | Jeff Shafer

# Assurance

*Content adapted from CS 5430 (System Security), Cornell University, Dr. Michael Clarkson*

# Schedule

## This Week

- Tue September 4
  - Beyond the Attacks
  - Goals and Requirements
- Thur September 6
  - Assurance

## Next Week

- Tue September 11
  Thur September 13
  - *Architectural Approaches to Security*

# Trivia

# Trusted Computing Base (TCB)

- Set of all hardware + firmware + software components that are critical to security
  - A vulnerability *inside* the TCB could jeopardize assets of *entire* system
    - **Examples in a commodity system?**
  - A vulnerability *outside* the TCB cannot jeopardize any more assets than those granted by security policy
    - **Examples in a commodity system?**
- **Want the TCB to be as small as possible!**
  - Security evaluation focuses on TCB

# Access Control

- **Discretionary Access Control (DAC)**
  - *Ability to restrict access to objects based on the identity of subjects and/or groups to which they belong*
  - Why discretionary? A subject (owner) with a certain access permission can decide whether or not to pass that permission on to other subjects
  - Example: File stored in OS has owner; owner can elect to make file readable/writable to other users or groups

# Access Control

- **Mandatory Access Control (MAC)**
  - *Any operation by any subject on any object is verified against authorization rules (i.e. policy) before proceeding*
  - The system (not the owner) decides whether or not to grant access
  - Subject/user cannot override, only a central policy administrator (“mandatory”)
  - Examples
    - Linux – AppArmor and SELinux
    - Windows – Integrity Levels
    - FreeBSD – TrustedBSD project

# Assurance

# Recap

- Aspects of Security
  - **C**onfidentiality, **I**ntegrity, **A**vailability
- Key Concepts
  - Harm, threat, vulnerability, attack, countermeasure
- Principles
  - Accountability, least privilege, defense in depth, …
- Goals and Requirements
  - What the system should and should not do

# Assurance

- How do you [developer] convince yourself that your system is secure?
  - **How do you convince others?**
- **Assurance** is evidence that system will not fail in particular ways
  - Development process
    (e.g. formal methods, deliberate fault injection, …)
  - Skill of developers
  - Experience with deployed systems
- **Evaluation** is process of establishing assurance
  - Developers, QA teams, third-party testing

# Economics > Security

- Companies race to ship innovative products sooner than competitors
  - Little security or wrong security
- Security is “bolted on” later in product development as NEW FEATURE™!
  - Customers already locked in
  - Product already deployed (legacy code)
  - Architectural/design changes very challenging at this stage

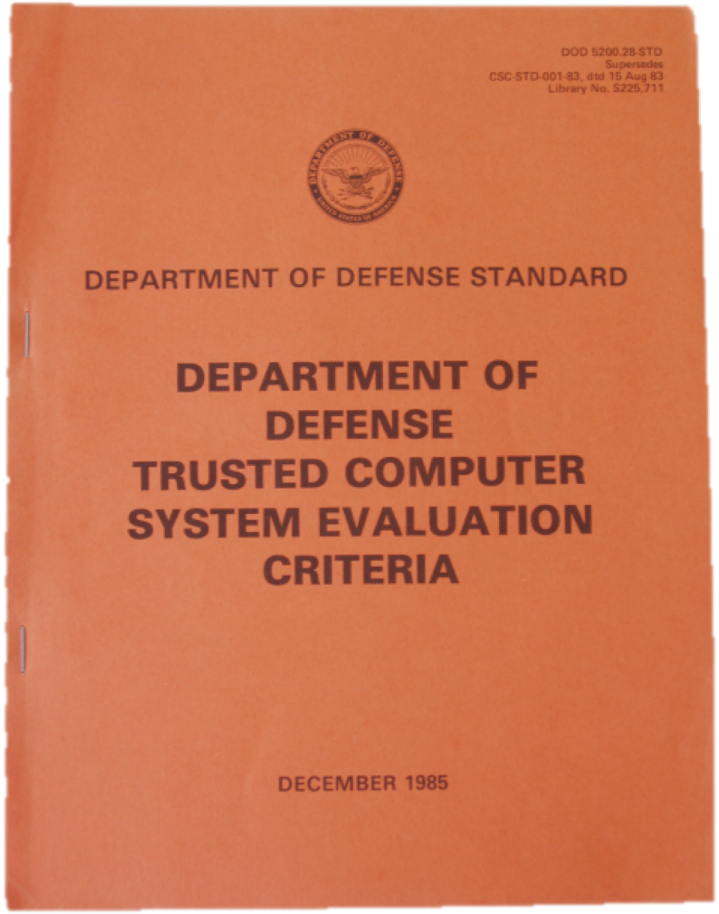
# Day 1

- **Integrate security functionality from the beginning of development**
  - During requirements engineering
  - During system design
  - During testing
- Accumulate evidence of security as development proceeds
  - Documentation
  - Analysis: By humans, by machines
  - Test suites

# Evaluation

# Evaluation

1. Trusted Computer System Evaluation
   - “Orange Book”
   - 1983-2005
2. Common Criteria (CC)
   - 2009+




http://csrc.nist.gov/publications/history/dod85.pdf

**A nice relaxing read!**

# Evaluation – Orange Book

# Evaluation – Orange Book

- Trusted Computer System Evaluation Criteria
  - US Department of Defense standard
  - Released in 1983, deprecated in 2005
  - Standards to evaluate computer systems used for the processing of sensitive or classified data
- Four *divisions* (D, C, B, A) that provide different levels of trust for the evaluated system

# Evaluation – Orange Book

- Division **D** – Minimal protection
  - System was evaluated, failed to meet higher standards ☹
  - Rare certification
    (why submit to evaluation if you know you will fail?)

# Evaluation – Orange Book

- Division **C** – Discretionary protection
  - *Discretionary protection* applies to Trusted Computer Base (TCB) with optional object (file, directory, devices etc.) protection
    **C1** – Discretionary Security Protection
    - Identification and authentication
    - Separation of users and data
    - Discretionary Access Control (DAC) capable of enforcing access limitations on an individual basis
    - Required System Documentation and user manuals

# Evaluation – Orange Book

- Division **C** – Discretionary protection
  - **C2** – Controlled Access Protection
    - *All of C1, plus…*
    - More finely grained DAC
    - Individual accountability through login procedures
    - Audit trails
    - Object reuse
    - Resource isolation
    - Certified OS's: DEC VMS, Novell NetWare, IBM OS/400, Windows NT

# Evaluation – Orange Book

- Division **B** – Mandatory Protection
  - TCB protection systems are mandatory, not discretionary
  - **B1** – Labelled Security Protection
    - Informal security policies, mandatory access control (multilevel security)
    - Certified OS: HP-UX BLS, Cray Research Trusted Unicos 8.0, Digital SEVMS, Harris CS/SX, SGI Trusted IRIX
  - **B2** – Structured Protection
    - Formal security policies, clearly defined TCB, covert channel analysis
  - **B3** – Security Domains
    - Minimal TCB with complete mediation, automated intrusion detection
    - Certified OS:  Getronics/Wang Federal XTS-300

# Evaluation – Orange Book

- Division **A** – Verified Protection
  - **A1** – Verified Protection
    - Formal methods and proof of integrity of TCB
    - Certified OS's:
      - Boeing MLS LAN
      - Gemini Trusted Network Processor (RTOS) https://www.nist.gov/sites/default/files/documents/2016/09/15/aesec_rfi_mls-rtos.pdf
      - Honeywell SCOMP http://www.dtic.mil/dtic/tr/fulltext/u2/a229523.pdf (actual 1985 report granting A1 status!)

# Legacy of Orange Book

**Have you heard of most of those operating systems?**

- Evaluation didn't succeed in commercial market
  - Too costly – customer had to pay
  - Too slow – Over 1 year to complete evaluation, by which time software is out of date
- "One size fits all" requirements for all systems
- Unpopular security features mandated by higher levels
  - In Usability vs Security, security won (here)

# Legacy of Orange Book

- Raised awareness of security for vendors and governments
  - Major operating systems incorporated discretionary access control – would they have done so without government prodding?
- Few systems incorporated multilevel security specified by higher Orange Book divisions
- Lead to international standards for evaluation

# Evaluation – Common Criteria

# Common Criteria (CC)

- Developed by the governments of Canada, France, Germany, the Netherlands, the UK, and the U.S.
- Unified existing standards
  - Orange Book (US)
  - ITSEC (Europe, 1990's)
  - CTCPEC (Canada, 1990's)
- International standard: ISO/IEC 15408

# Common Criteria (CC)

- Not one-size-fit-all like Orange Book
- Protection Profile (PP) and Security Target (ST)
  - Customized security goals and requirements
  - Ex: For OS, for smartphone, for VPN client, …
- Increasingly strict evaluation criteria for how well system meets profile (PP) and target (ST)
- Evaluation done by independent labs

# Protection Profile (PP)

- Written for a category of products that meet specific consumer needs
  - Smart cards? Network firewalls? Databases?
  - Hundreds written - http://www.commoncriteriaportal.org/
- Implementation independent!
- Security environment
  - Assumptions about intended usage
  - *Threats* of concern
- Security **goals and requirements**
- PP can be evaluated (complete, consistent, technically sound)

# Security Target (ST)

- Argues (w/ evidence) how the system meets the security goals and requirements
  - Assurance argument
- Created from scratch or based on multiple protection profiles
- Customized to a specific product or system
  - Target of Evaluation (TOE)

# Evaluation Assurance Level (EAL)

- **EAL1** – Functionally Tested
  - Analysis of specifications, documentation w/ independent testing
  - Some confidence desired but threat is not serious
- **EAL2** – Structurally Tested
  - Analysis of high-level design and developer's testing w/vulnerability analysis
  - Low level of assurance – used for legacy systems?
- **EAL3** – Methodically Tested and Checked
  - Requires use of developer environment controls and configuration management

# Evaluation Assurance Level (EAL)

- **EAL4** – Methodically Designed, Tested, and Reviewed
  - Also analyze low-level design, some of the implementation
  - Developers must provide informal model of product or security policy
  - Moderate level of assurance, probably highest likely to achieve for pre-existing system
  - Common level for commercial OS

**EAL5,6,7**

  - Increasing demand for formal verification, penetration testing, and independent testing

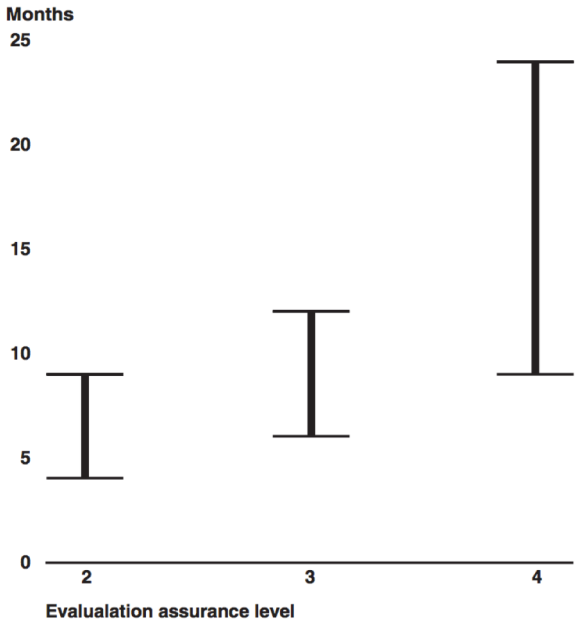
- Higher EAL does not mean more secure, it means the assurance in claimed security is based on stronger evidence

# Legacy of Common Criteria

- "When presented with a security product, you must always consider whether the salesman is lying or mistaken." – Ross Anderson
- Does the PP specify the product you actually want?
- Is the evaluation facility trustworthy?
  - Paid by developer
  - Controlled by governments
- What vulnerabilities have been discovered after the evaluation?

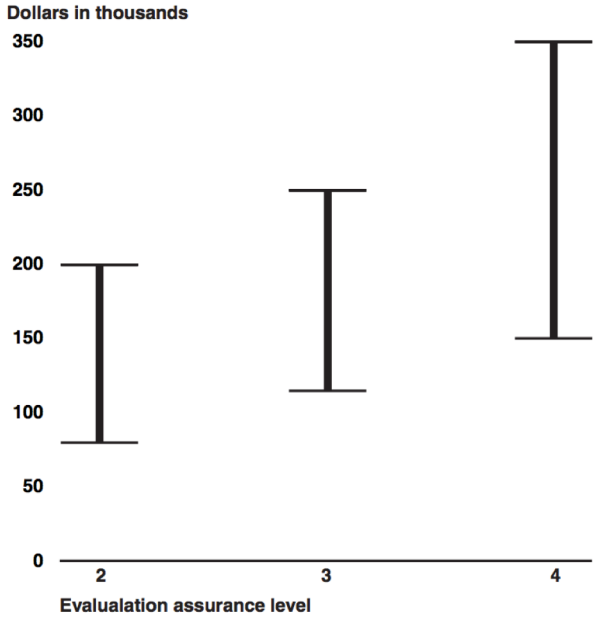
# Cost and Time

**Figure 2: Range of Sample Cost of NIAP Evaluations to Vendors by Evaluation Assurance Level**




Source: GAO analysis of data provided by laboratories.

**Figure 4: Range of Time Required for Completing Product Evaluations at Various Evaluation Assurance Levels**



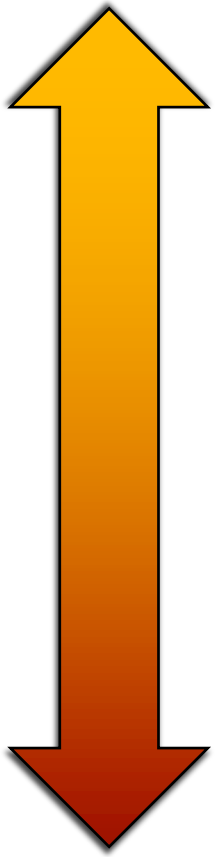

Source: GAO analysis of data provided by laboratories.

GAO report on Information Assurance, 2006
GAO-06-392
http://www.gao.gov/new.items/d06392.pdf

# Verification and Testing

# Approaches to Reliability

- Social
  - Code review
  - Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - Static analysis
  - Fuzzers
- Mathematical
  - Sound type systems
  - Formal verification



Less Formal – Techniques may miss problems in programs

All methods should be used! Even formal methods can have holes, e.g. Did you prove the right thing? Do your assumptions match reality?

More Formal – Eliminate *with certainty* as many problems as possible

# Testing vs Verification

- Testing
  - Cost effective
  - Guarantee that the program is correct on **tested** inputs and in **tested** environments
- Verification
  - Expensive
  - Guarantee that program is correct on **all** inputs and in **all** environments

# Formal Verification

- Idea: Prove system correct w/r/t mathematical models
- Typically done for small or safety-critical systems
- Modern examples
  - CompCert – Verified C compiler
    - http://compcert.inria.fr/
  - seL4 – Verified microkernel OS
    - https://sel4.systems/
  - Ynot – Verified DBMS and web service
    - http://ynot.cs.harvard.edu

# Verification

- Options for lightweight verification?
- Type systems
  - Guarantee certain misbehaviors won't occur
  - Good tradeoff of usability vs guarantees
- Static analysis
  - Inspect source code or object/class files and look for suspect patterns
  - Example: `FindBugs` for Java class files

# Bugs?

- "Bugs" imply that something just wandered in
- The truth


- Fault: Result of human error
  - Implementation doesn't match design, or design doesn't match requirements
  - End user might never notice
- Failure: Violation of requirements
  - End user notices

# FindBugs

- Looks for *patterns* in code that are likely *faults* and (if un-fixed) are likely to cause *failures*
- Categorizes and prioritizes bugs for presentation to developer
- **`FindBugs`** - http://findbugs.sourceforge.net/
  - Bug descriptions - http://findbugs.sourceforge.net/bugDescriptions.html
  - Video presentation by Dr. Bill Pugh (creator) - https://www.youtube.com/watch?v=8eZ8YWVl-2s

# Testing

- Goal is to expose existence of faults, so that they can be fixed
- **Unit testing**: isolated components
- **Integration testing**: combined components
- **System testing**: functionality, performance, acceptance

# Testing

- **When do you stop testing?**
- Bad answer: when time is up
- Bad answer: what all tests pass
- Better answer: when methodology is complete (code coverage, paths, boundary cases, etc.)
- Future answer: statistical estimation says Pr[undetected faults] is low enough (active research)

# Penetration Testing

- Testing for **security**
- Experts attempt to attack
  - Internal vs. external
  - Overt vs. covert
- Typical vulnerabilities exploited
  - Passwords (cracking)
  - Buffer overflows
  - Bad input validation
  - Race conditions / TOCTOU
  - Filesystem misconfiguration
  - Kernel flaws

# Fuzz Testing

- Generate random inputs and feed them to programs:
  - Crash? hang? terminate normally?
  - Of ~90 utilities in '89, crashed about 25-33% in various Unixes
  - Crash implies buffer overflow potential
- Since then, “fuzzing” has become a standard practice for security testing

# Fuzz Testing

- Testing strategy? Purely random only gets low hanging fruit
- Better testing:
  - Use grammar to generate inputs
  - Randomly mutate good inputs in small ways
    - Especially for testing of network protocols
- Research: use analysis of source code to guide mutation of inputs