Review

• **Aspects of security:** confidentiality, integrity, availability

• **Concepts:** Harm, threat, vulnerability, attack, countermeasure

• **Principles:** Accountability, least privilege, defense in depth, open design, ...

• **Goals and Requirements:** What system should not and should do for security

**Today:** assurance and evaluation
**Assurance**

- How do you convince yourself that 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 and discovery)
  - Skill of developers
  - Experience with deployed system
- **Evaluation** is process of establishing assurance
  - developers
  - QA teams
  - third party labs
Economics is against us

• Companies race to ship innovative product sooner than competitors
  – Little security
  – Wrong security

• Later security is “bolted on” as additional features
  – But incentive is to lock in customers
  – Product is already deployed; too late for major design changes that might be necessary
Build security in

• 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
Orange Book evaluation
Orange Book evaluation

- Used approx. 1985-2000 for US government systems
- Evaluation classes (selected traits):
  - **D**: meets no higher requirements
  - **C1**: DAC & authentication (but maybe not at the level of individual users), TCB with integrity verification, security testing, documentation of security features/testing/design
  - **C2**: improved DAC (at the level of single users, failsafe defaults, limits on propagation), audit (of specified security relevant events and details of those events)
  - IBM mainframes and Windows NT got this certification
Orange Book evaluation

• Evaluation classes, continued:
  – B1: informal security policies, mandatory access control (multilevel security)
  – B2: formal security policies, clearly defined TCB, covert channel analysis
  – B3: minimal TCB with complete mediation, automated intrusion detection
  – A1: formal verification of design
    • only a handful of systems ever achieved this level
Legacy of Orange Book

• Evaluation didn't succeed in commercial market
  – Too costly; costs diverted to government and customers
  – Too long to get evaluated (>1 year) compared to short product cycles
• Raised awareness of security for vendors and government
  – Major operating systems did incorporate discretionary access control; would that have happened without evaluation?
  – But few systems ever incorporated the multilevel security the US DoD wanted
• Unpopular security features mandated by higher levels
  – Research still ongoing on how to make such features usable
• Led to international standards for evaluation...
Common Criteria (CC)

• Evolved in the 1990s out of criteria in Europe, Canada, and US

• Different evaluation model:
  – Define protection profile and security target
    • think of these as customized security goals/requirements
    • e.g., for OS, for smartphone, for VPN client
    • not one-size-fits-all like Orange Book
  – Increasingly strict evaluation criteria for how well system meets profile/target

• Evaluation done by independent labs
Protection profile (PP)

• Written for a category of products or systems that meet specific consumer needs
• Implementation independent
• Security environment:
  – assumptions about intended usage
  – threats of concern
• Security goals and requirements [using our terminology]
• PP itself can be evaluated (complete, consistent, technically sound)
Security target (ST)

- Can be based on multiple protection profiles, or created from scratch
- Customized to a specific Target of Evaluation (TOE), i.e., product or system
- Argues (provides evidence) how the system meets the security goals and requirements
  — Assurance argument
Evaluation Assurance Level (EAL)

• **EAL1**: Functionally Tested
  – Analysis of specifications, documentation; independent testing
  – Some confidence desired but threat is not serious

• **EAL2**: Structurally Tested
  – Analysis also of high-level design, of developer's testing; vulnerability analysis
  – Low level of assurance, perhaps for legacy systems

• **EAL3**: Methodically Tested and Checked
  – Also requires use of development 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 systems
  – Common level for commercial OS

• **EAL5 through EAL 7**
  – Increasing demands for formal verification, penetration testing, independent testing

• Higher EAL does not mean more secure—rather, means 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

• Is the PP really what you want?

• Is the evaluation facility trustworthy?
  – Paid by developer
  – Controlled by governments

• What vulnerabilities have been discovered after evaluation?
Evaluation Assurance Level (EAL)

VERIFICATION AND TESTING
Approaches to reliability

- Social
  - Code reviews
  - Extreme/Pair programming

- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking

- Technological
  - Static analysis ("lint" tools, FindBugs, …)
  - Fuzzers

- Mathematical
  - Sound type systems
  - Formal verification

Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:
- 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 program is correct on tested inputs and in tested environments

Verification:
• Expensive
• Guarantee that program is correct on all inputs and in all environments
Edsger W. Dijkstra

Turing Award Winner (1972)

*For eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness*

"Program testing can at best show the presence of errors but never their absence."
Verification

Formal verification: prove system correct w.r.t. mathematical models

• Typically done for small and/or safety-critical systems

• In the 1970s, scaled to about tens of LOC

• Now, research projects scale to real software:
  – **CompCert**: verified C compiler
  – **seL4**: verified microkernel OS
  – **Ynot**: verified DBMS, web services
Verification

Lightweight kinds of verification:

• Type systems
  – Guarantee certain misbehaviors won't occur
  – Good tradeoff of usability vs. guarantees

• Researchers continue working to find other sweet spots

• For lightweight security verification?
  – **FindBugs**: project from UMD, used by Google, Oracle, Wells Fargo, Bank of America, etc.
  – Eclipse plugin and standalone tool
  – A subset of Fortify SCA tool
  – What does it do...?
Bugs

"bug": suggests something just wandered in

[IEEE 729]

- **Fault**: result of human error in software system
  - E.g., implementation doesn't match design, or design doesn't match requirements
  - Might never appear to end user
- **Failure**: violation of requirement
  - Something goes wrong for end user
FindBugs

• Looks for *patterns* in code that are likely *faults* and that are likely to cause *failures*

• Categorizes and prioritizes bugs for presentation to developer
  
  – [Sample output on JDK 7 source](https://www.youtube.com/watch?v=8eZ8YVVI-2s)

• Watch video of Prof. Bill Pugh, developer of FindBugs, present it to a Google audience:
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
• **Fun fact:** $\Pr[\text{undetected faults}]$ increases with $\#$ detected faults [Myers 1979, 2004]
• **Better answer:** when methodology is complete (code coverage, paths, boundary cases, etc.)
• **Future answer:** statistical estimation says $\Pr[\text{undetected faults}]$ is low enough (active research)

Testing for security?
Penetration testing

• 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

[Barton Miller, 1989, 2000, 2006]
• "It was a dark and stormy night..."
• 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
• Results have been repeated for X-windows system, Windows NT, Mac OS X
  – Results keep getting worse in GUIs but better on command line
Fuzz testing

Testing strategy:
• Purely random no longer so good, just gets low-hanging fruit
• Better:
  — Use grammar to generate inputs
  — Or randomly mutate good inputs in small ways
    • especially for testing of network protocols
  — Research: use analysis of source code to guide mutation of inputs
Upcoming events

• [today] Add deadline
• [next Wed] A1 due

One unerring mark of the love of truth is not entertaining any proposition with greater assurance than the proofs it is built upon will warrant.

– John Locke