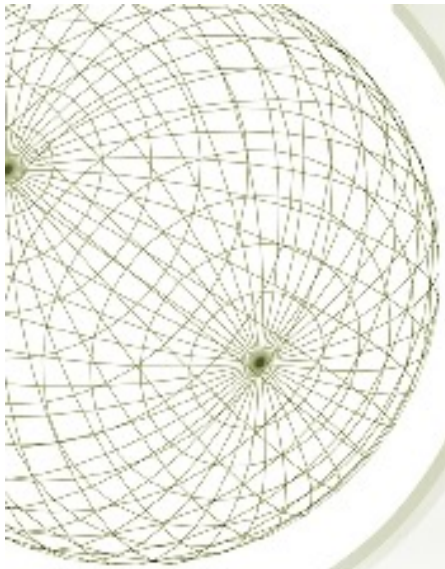
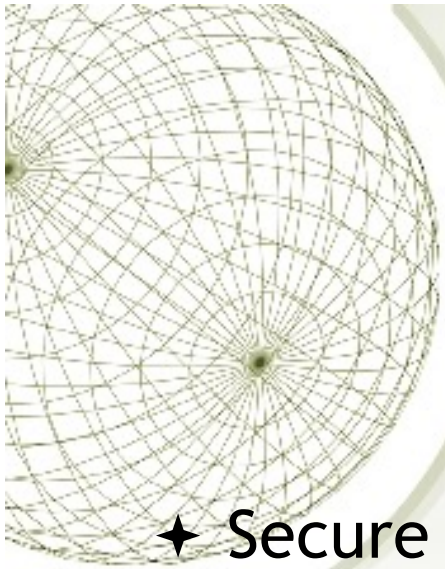


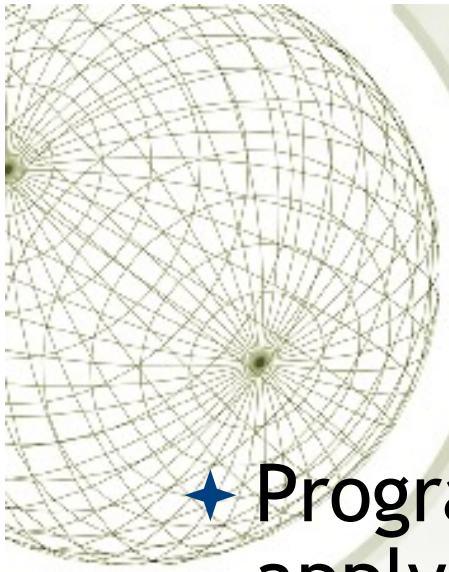
Program Security

Ola Flygt
Linnaeus University, Sweden
http://homepage.lnu.se/staff/oflmsi/
Ola.Flygt@lnu.se
+46 470 70 86 49



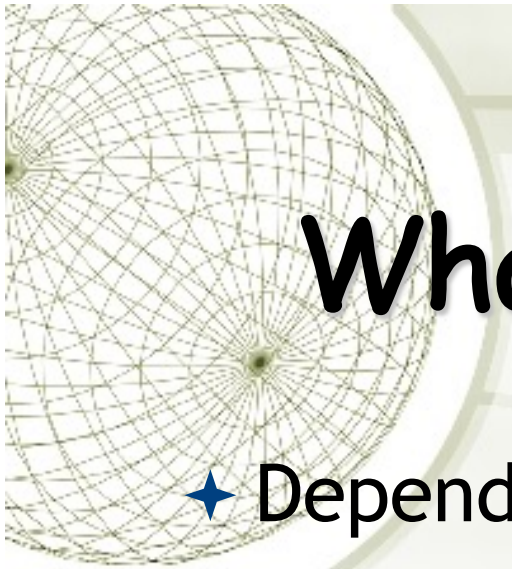
Program Security – Outline

- Secure Programs – Defining & Testing
- Non malicious Program Errors
- Malicious Code
- Controls for Security



Program Security

- Program security – Our first step on how to apply security to computing
- Protecting programs is the heart of computer security
- All kinds of programs, from apps via OS, DBMS, networks etc.
- Issues:
  - How to keep programs free from flaws
  - How to protect computing resources from programs with flaws

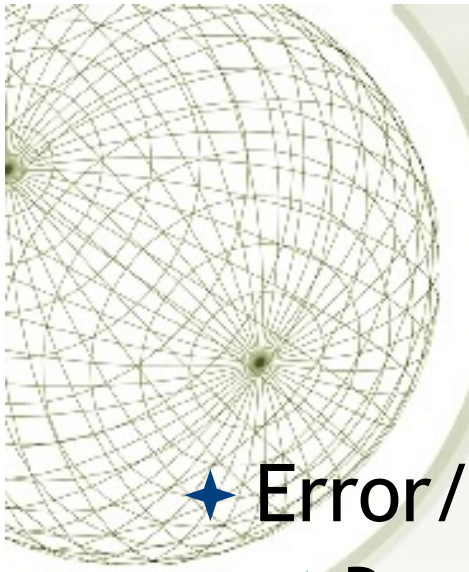


What is Program Security?

- Depends on who you ask
  - user - fit for his task
  - programmer - passes all her tests
  - manager - conformance to all specs
- Developmental criteria for program security include:
  - Correctness of security & other requirements
  - Correctness of implementation
  - Correctness of testing

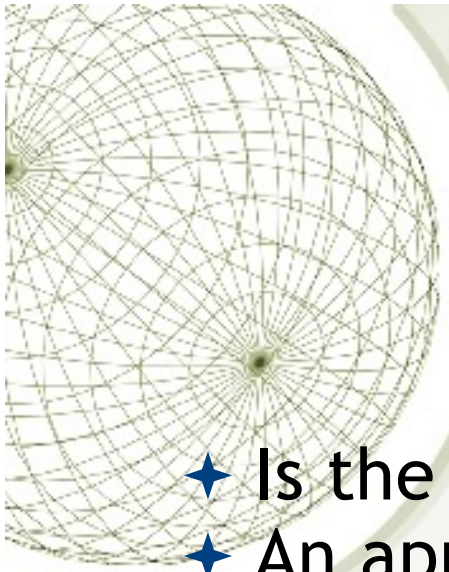
Fault (bug) terminology

- Error - may lead to a fault
- Fault - cause for deviation from intended function
- Failure - system malfunction caused by fault
  - Faults - seen by ”insiders” (e.g., programmers)
  - Failures - seen by „outsiders” (e.g., independent testers, users)



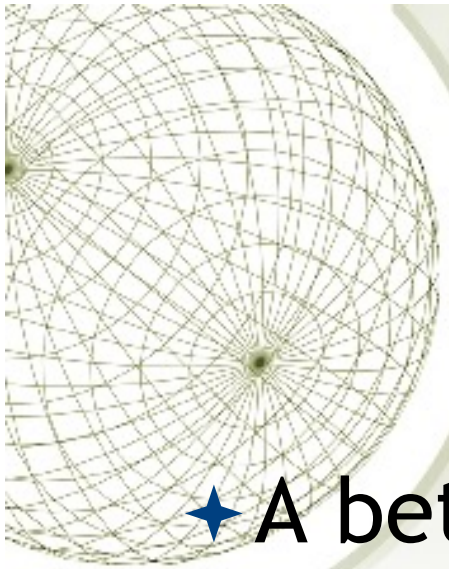
Fault terminology

- Error/fault/failure example:
  - Programmer's indexing error, leads to buffer overflow fault
  - Buffer overflow fault causes system crash (a failure)
- Two categories of faults w.r.t. duration
  - Permanent faults
  - Transient faults – can be much more difficult to diagnose



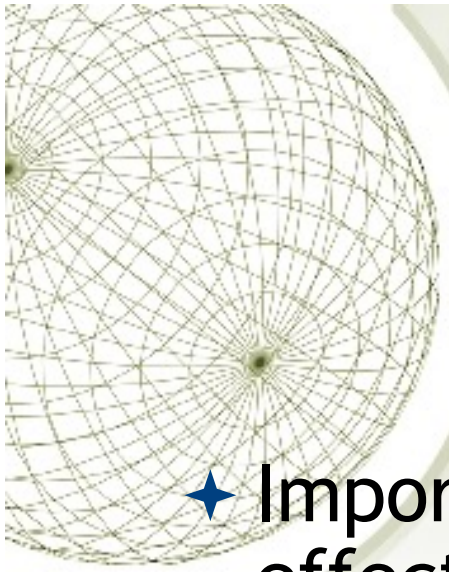
Testing Security

- Is the system/application I am reviewing secure?
- An approach to judge s/w security: penetrate and patch
  - Red Team / Tiger Team tries to crack s/w. If you withstand the attack => security is good. Is this true? Rarely.
- Too often developers try to quick-fix problems discovered by Tiger Team
  - Quick patches can introduce new faults due to:
    - Pressure – causing narrow focus on fault, not context
    - Non-obvious side effects
    - System performance requirements not allowing for security overhead



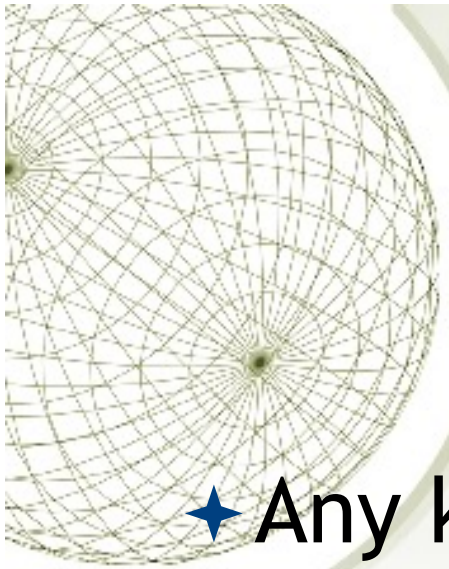
Testing Security

- A better approach to judging s/w security: testing pgm **behaviour**
  - Compare behaviour vs. requirements
  - Program security flaw = inappropriate behaviour caused by a pgm fault or failure
  - Flaw detected as a fault or a failure



Testing Security

- Important: If flaw detected as a failure (an effect), look for the underlying fault (the cause)
- Recall: fault seen by insiders, failure – by outsiders
- If possible, detect faults before they become failures

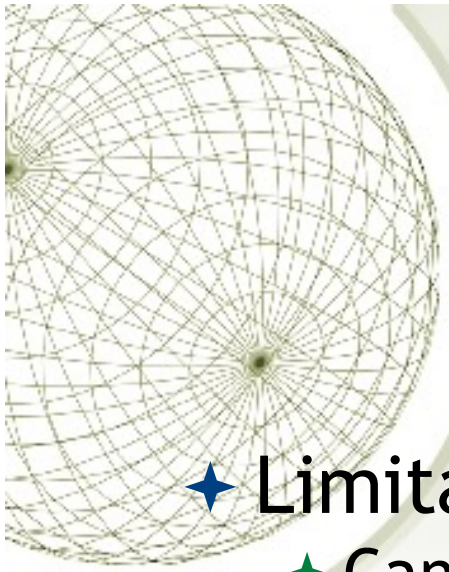


Testing Security

- Any kind of fault/failure can cause a security incident
  - Misunderstood requirements / error in coding / typing error
  - In a single pgm / interaction of k pgms
  - Intentional flaws or accidental (inadvertent) flaws

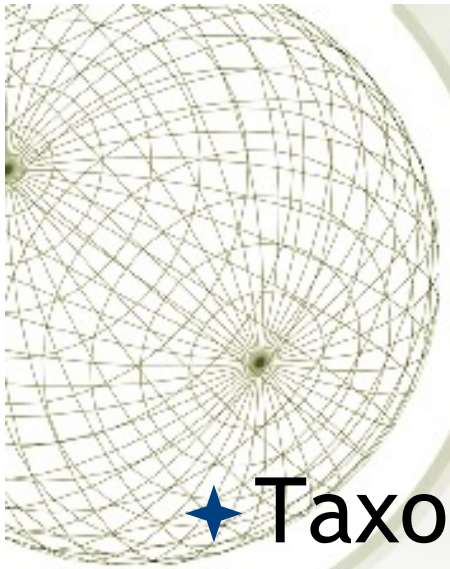
Testing Security

- Therefore, we must consider security consequences for all kinds of detected faults/failures
  - Even inadvertent faults / failures
  - Inadvertent faults are the biggest source of security vulnerabilities exploited by attackers



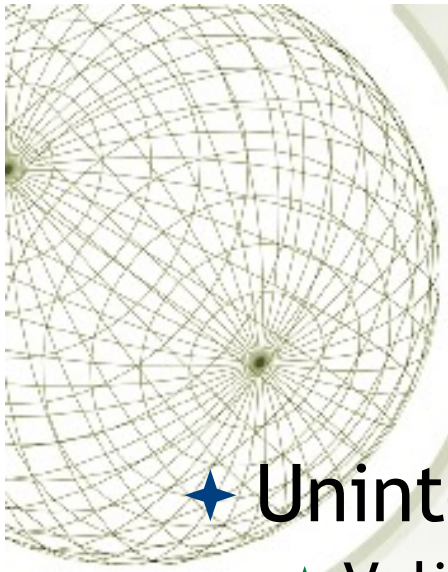
Problems with program behaviour testing

- Limitations of testing
  - Can't test exhaustively
  - Testing checks what the pgm should do – often can't test what the pgm should not do
- Complexity – malicious attacker's best friend
  - Too complex to model / to test
  - Exponential # of pgm states / data combinations
- Evolving technology
  - New s/w technologies appear
  - Security techniques have difficulty catching up with s/w technologies



Types of Program Flaws

- Taxonomy of program flaws
  - Unintentional
  - Intentional
    - Malicious
    - Non malicious



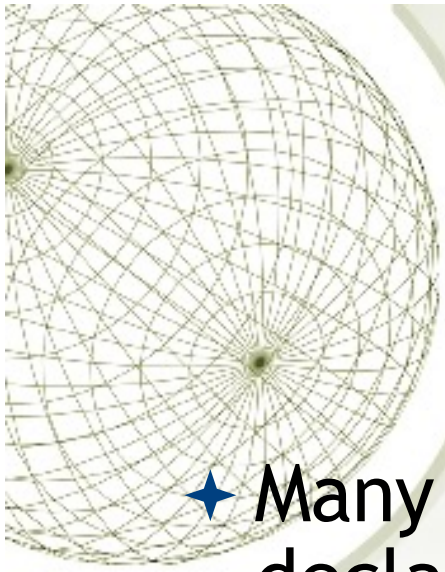
Types of Pgm Flaws

- Unintentional
  - Validation error (incomplete or inconsistent)
  - Domain error (variable value outside of its domain)
  - Serialization and aliasing
    - serialization – e.g., in DBMSs or OSs
    - aliasing - one variable or some reference, when changed, has an indirect (usually unexpected) effect on some other data
  - Inadequate identification and authentication (Section 2.1)
  - Boundary condition violation
  - Other exploitable logic errors

Common Non Malicious Program Errors

- According to a recent study[1] the most common vulnerabilities in software are Cross-Site Scripting (XSS), Input Validation, Permissions, Privileges and Access Control; and Information Leak/Disclosure
- We will look more closely at these types:
  - Buffer overflows
  - Incomplete mediation
  - Time-of-check to time-of-use errors

1) Source: https://www.techrepublic.com/article/the-3-least-secure-programming-languages/ (2019)

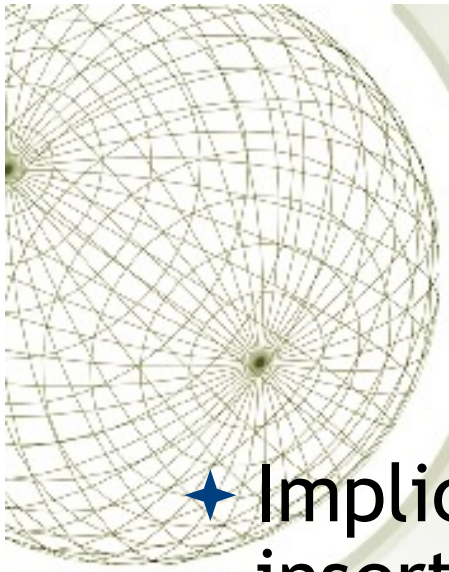


Buffer Overflows

- Many languages require buffer size declaration
  - C language statement: char sample[10];
  - Execute statement: sample[i] = ‘A’; where i=10
  - Out of bounds (0-9) subscript – buffer overflow occurs
  - Some compilers don’t check for exceeding bounds
- Similar problem caused by pointers. No reasonable way to define limits for pointers

Buffer Overflows, cont.

- Where does ‘A’ go? Depends on what is adjacent to ‘sample[10]’
  - Affects user’s data - overwrites user’s data
  - Affects users code - changes user’s instruction
  - Affects OS data - overwrites OS data
  - Affects OS code - changes OS instruction
- This is a case of aliasing



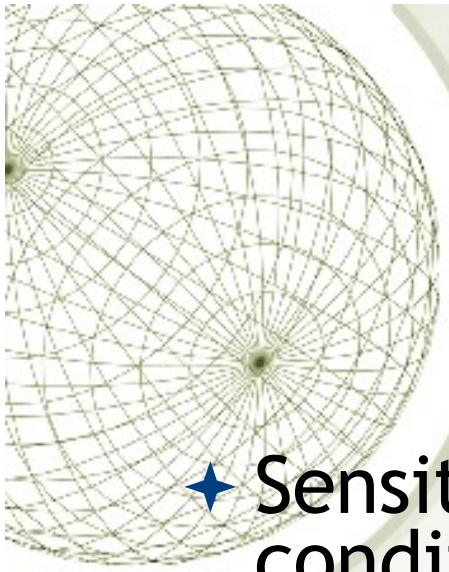
Buffer Overflows, cont.

- Implications of buffer overflow: Attacker can insert malicious data values and/or instruction codes into ”overflow space”
- Suppose buffer overflow affects OS code area:
  - Attacker code executed as if it were OS code
    - Attacker might need to experiment to see what happens when he inserts A into OS code area
  - Can raise attacker’s privileges (to OS privilege level) when ‘A’ is an appropriate instruction
  - Attacker can gain full control of OS

Buffer Overflows, cont.

- Other types of overflow
  - Buffer overflow affects a call stack area
  - Web server attack similar to buffer overflow attack: pass very long string to web server
- Buffer overflows still common
  - Used by attackers
    - to crash systems
    - to exploit systems by taking over control

Large # of vulnerabilities due to buffer overflows



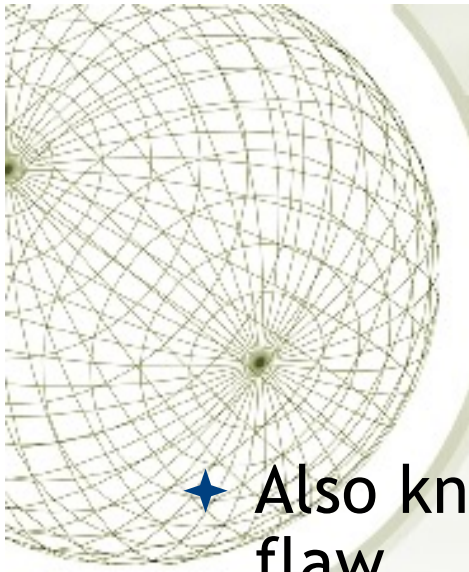
Incomplete Mediation

- Sensitive data are in exposed, uncontrolled condition
- Example
  - URL to be generated by client's browser to access server, e.g.: `http://www.things.com/order/final&custID=101&part=555A&qy=20&price=10&ship=boat&shipcost=5&total=205`
  - Instead, user edits URL directly, changing price and total cost as follows: `http://www.things.com/order/final&custID=101&part=555A&qy=20&price=1&ship=boat&shipcost=5&total=25`

Incomplete Mediation, cont.

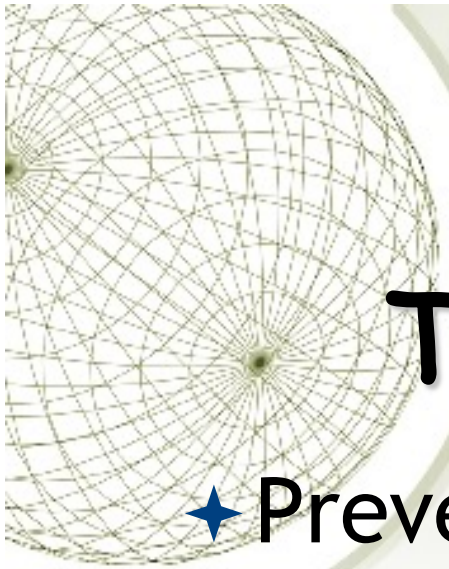
- Unchecked data are a serious vulnerability!

- Possible solution: anticipate problems
  - Don't let client return a sensitive result (like total) that can be easily recomputed by server
  - Use drop-down boxes / choice lists for data input
  - Prevent user from editing input directly
  - Check validity of data values received from client



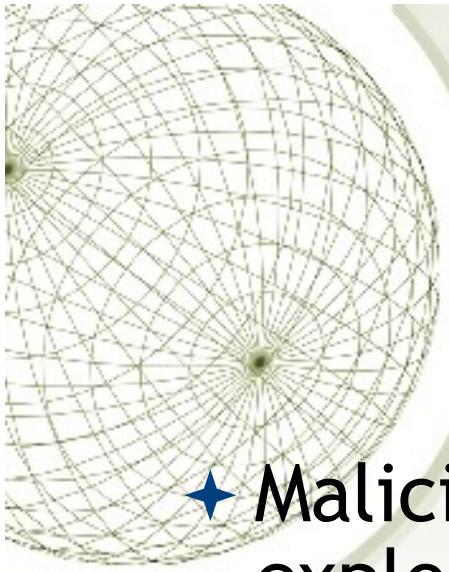
Time-of-check to Time-of-use Errors

- Also known as synchronization flaw / serialization flaw
- Synchronization - mediation with “bait and switch” in the middle
- Non-computing example:
  - Swindler shows buyer real Rolex watch (bait)
  - After buyer pays, switches real Rolex to a forged one
- In computing:
  - Change of a resource (e.g., data) between time access checked and time access used



Time-of-check to Time-of-use Errors, cont.

- Prevention of Toc-t-Tou errors
  - Be aware of time lags
  - Use digital signatures and certificates to ”lock” data values after checking them



Malicious Code – are also programs!

- Malicious code or rogue pgm is written to exploit flaws in pgms (or incautious users)
- Malicious code can do anything a pgm can
- Malicious code can change
  - data
  - other programs
- Malicious code has been ”officially” defined by Cohen in 1984 but virus behaviour known since at least 1970

Types of Malware

Code Type	Characteristics
Virus	Code that causes malicious behavior and propagates copies of itself to other programs
Trojan horse	Code that contains unexpected, undocumented, additional functionality
Worm	Code that propagates copies of itself through a network; impact is usually degraded performance
Rabbit	Code that replicates itself without limit to exhaust resources
Logic bomb	Code that triggers action when a predetermined condition occurs
Time bomb	Code that triggers action when a predetermined time occurs
Dropper	Transfer agent code only to drop other malicious code, such as virus or Trojan horse
Hostile mobile code agent	Code communicated semi-autonomously by programs transmitted through the web
Script attack, JavaScript, Active code attack	Malicious code communicated in JavaScript, ActiveX, or another scripting language, downloaded as part of displaying a web page

Types of Malware (cont.)

Code Type	Characteristics
RAT (remote access Trojan)	Trojan horse that, once planted, gives access from remote location
Spyware	Program that intercepts and covertly communicates data on the user or the user's activity
Bot	Semi-autonomous agent, under control of a (usually remote) controller or "herder"; not necessarily malicious
Zombie	Code or entire computer under control of a (usually remote) program
Browser hijacker	Code that changes browser settings, disallows access to certain sites, or redirects browser to others
Rootkit	Code installed in "root" or most privileged section of operating system; hard to detect
Trapdoor or backdoor	Code feature that allows unauthorized access to a machine or program; bypasses normal access control and authentication
Tool or toolkit	Program containing a set of tests for vulnerabilities; not dangerous itself, but each successful test identifies a vulnerable host that can be attacked
Scareware	Not code; false warning of malicious code attack

History of Malware

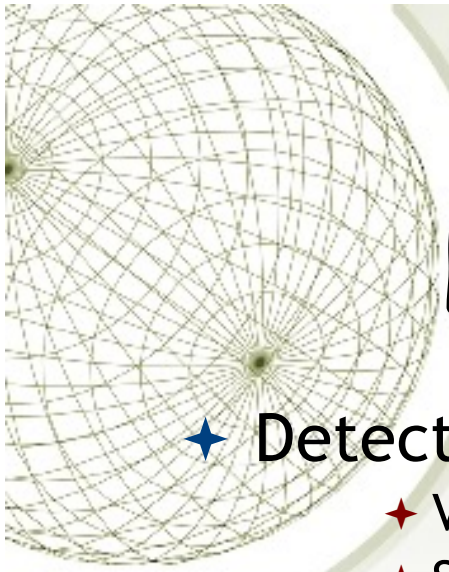
Year	Name	Characteristics
1982	Elk Cloner	First virus; targets Apple II computers
1985	Brain	First virus to attack IBM PC
1988	Morris worm	Allegedly accidental infection disabled large portion of the ARPANET, precursor to today's Internet
1989	Ghostballs	First multipartite (has more than one executable piece) virus
1990	Chameleon	First polymorphic (changes form to avoid detection) virus
1995	Concept	First virus spread via Microsoft Word document macro
1998	Back Orifice	Tool allows remote execution and monitoring of infected computer
1999	Melissa	Virus spreads through email address book
2000	IloveYou	Worm propagates by email containing malicious script. Retrieves victim's address book to expand infection. Estimated 50 million computers affected.
2000	Timofonica	First virus targeting mobile phones (through SMS text messaging)
2001	Code Red	Virus propagates from 1st to 20th of month, attacks whitehouse.gov web site from 20th to 28th, rests until end of month, and restarts at beginning of next month; resides only in memory, making it undetected by file-searching antivirus products

History of Malware (cont.)

Year	Name	Characteristics
2001	Code Red II	Like Code Red, but also installing code to permit remote access to compromised machines
2001	Nimda	Exploits known vulnerabilities; reported to have spread through 2 million machines in a 24-hour period
2003	Slammer worm	Attacks SQL database servers; has unintended denial-of-service impact due to massive amount of traffic it generates
2003	SoBig worm	Propagates by sending itself to all email addresses it finds; can fake From: field; can retrieve stored passwords
2004	MyDoom worm	Mass-mailing worm with remote-access capability
2004	Bagle or Beagle worm	Gathers email addresses to be used for subsequent spam mailings; SoBig, MyDoom, and Bagle seemed to enter a war to determine who could capture the most email addresses
2008	Rustock.C	Spam bot and rootkit virus
2008	Conficker	Virus believed to have infected as many as 10 million machines; has gone through five major code versions
2010	Stuxnet	Worm attacks SCADA automated processing systems; zero-day attack
2011	Duqu	Believed to be variant on Stuxnet
2013	CryptoLocker	Ransomware Trojan that encrypts victim's data storage and demands a ransom for the decryption key

How Viruses Work

- Attach
  - Append to program or e-mail
    - Executes with program
  - Surrounds program
    - Executes before and after program
    - Erases its tracks
  - Integrates or replaces program code
- Gain control
  - Virus replaces target
- Reside
  - In boot sector
  - Memory
  - Application program
  - Libraries



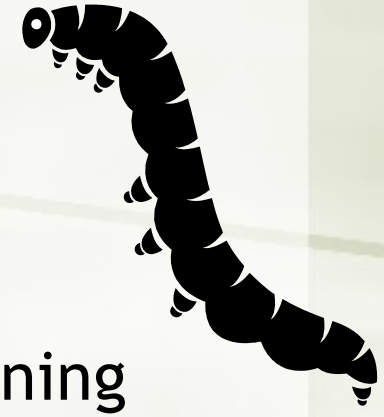
How Viruses Work, cont.

- Detection
  - Virus signatures
  - Storage patterns
  - Execution patterns
  - Transmission patterns
- Prevention
  - Don’t share executables
  - Use commercial software from reliable sources
  - Test new software on isolated computers
  - Open only safe attachments
  - Keep recoverable system image in safe place
  - Backup executable system file copies
  - Use virus detectors
  - Update virus detectors often

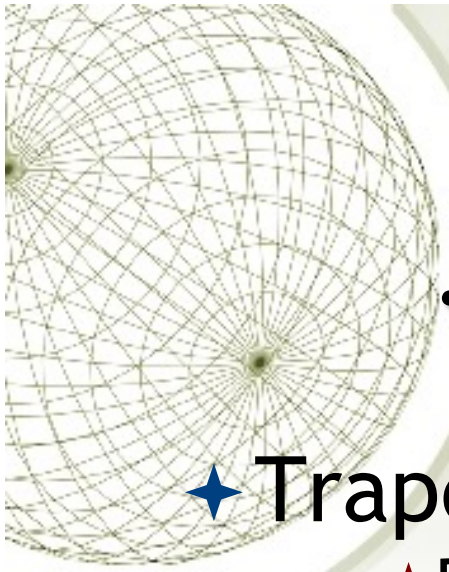
Virus Cause/Effect Analysis

Virus Effect	How it is caused
Attach to executable	✦Modify file directory ✦Write to executable program file
Attach to data/control file	✦Modify directory ✦Rewrite data ✦Append to data ✦Append data to self
Remain in memory	✦Intercept interrupt by modifying interrupt handler address table ✦Load self in non-transient memory area
Infect disks	✦Intercept interrupt ✦Intercept OS call (to format disk, for example) ✦Modify system file ✦Modify ordinary executable program
Conceal self	✦Intercept system calls that would reveal self and falsify results ✦Classify self as “hidden” file
Spread self	✦Infect boot sector ✦Infect systems program ✦Infect ordinary program ✦Infect data ordinary program reads to control its executable
Prevent deactivation	✦Activate before deactivating program and block deactivation ✦Store copy to reinfect after deactivation

The Morris Worm

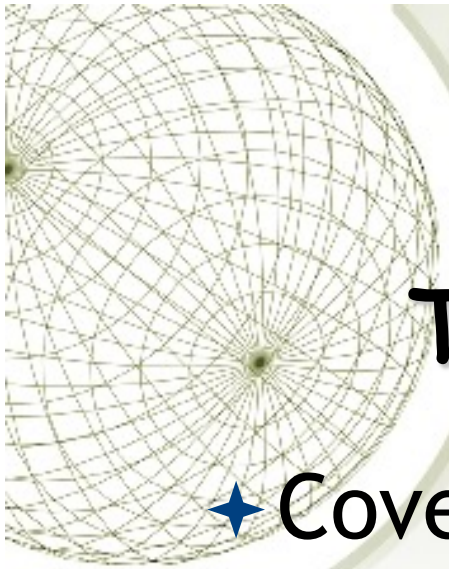


- Case Study:
  - 1988
  - Invaded VAX and Sun-3 computers running versions of Berkeley UNIX
  - Used their resources to attack still more computers
  - Within hours spread across the U.S
  - Infected hundreds / thousands of computers
  - Made many computers unusable



Exploitation of Flaws: Targeted Malicious Code

- Trapdoors
  - Program stubs during testing
  - Intentionally or unintentionally left
    - Forgotten
    - Left for testing or maintenance
    - Left for covert access
- Salami attack
  - Merges inconsequential pieces to get big results
  - Ex: deliberate diversion of fractional cents
    - Too difficult to audit

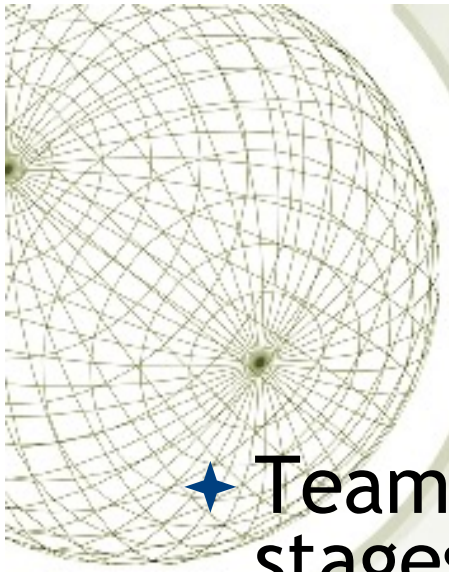


Exploitation of Flaws: Targeted Malicious Code, cont.

- Covert Channels
  - Programs that leak information
    - Trojan horse
  - Discovery
    - Analyse system resources for patterns
    - Flow analysis from a program's syntax (automated)
  - Difficult to close
    - Not much documented
    - Potential damage is extreme

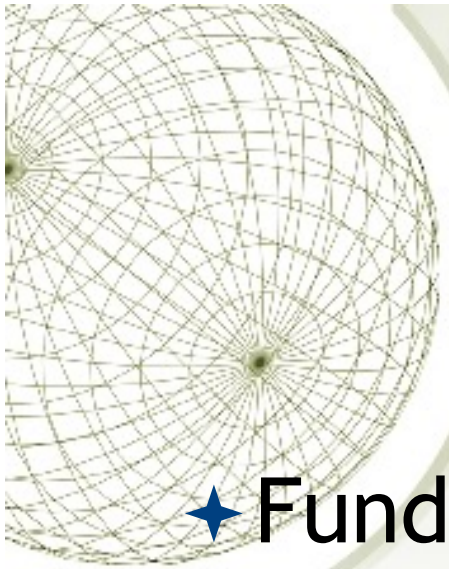
Preventing security flaws

- We have seen a lot of possible security flaws. How to prevent (at least some of) them?
- Software engineering concentrates on developing and maintaining quality s/w
  - We'll take a look at some techniques useful specifically for developing/maintaining secure s/w
- Three types of controls against pgm flaws:
  - Developmental controls
  - OS controls
  - Administrative controls



Developmental Controls Collaborative effort

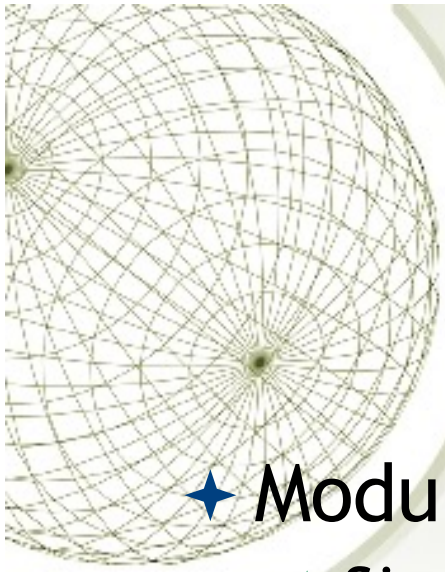
- Team of developers, each involved in ≥ 1 of stages:
  - Requirement specification
    - Regular req. specs: ”do X”
    - Security req. specs: ”do X and nothing more”
  - Design
  - Implementation
  - Testing
  - Documenting at each stage
  - Reviewing at each stage
  - Managing system development through all stages
  - Maintaining deployed system (updates, patches, new versions, etc.)



Developmental Controls, cont.

- Fundamental principles of s/w engineering
  - Modularity
  - Encapsulation
  - Information hiding

- You will go deeper into these when you study Object orientation

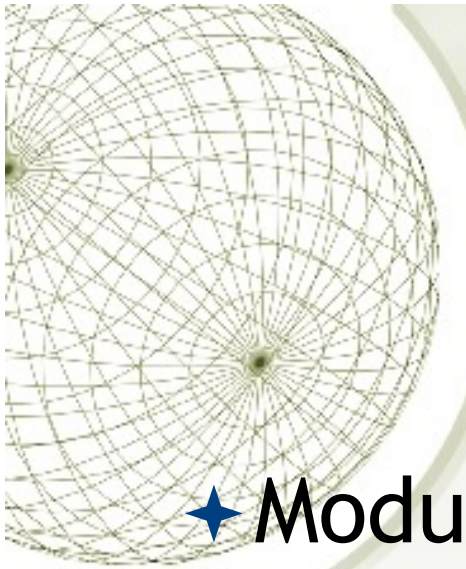


Developmental Controls, Modularity

- Modules should be:
  - Single-purpose - logically/functionally
  - Small - for a human to grasp
  - Simple - for a human to grasp
  - Independent – high cohesion, low coupling
    - High cohesion – highly focused on (single) purpose
    - Low coupling – free from interference from other modules
- Modularity should improve correctness
  - Fewer flaws => better security

Developmental Controls, Encapsulation

- Minimizing information sharing with other modules
  - Limited interfaces reduce # of covert channels
- Well documented interfaces
- ”Hiding what should be hidden and showing what should be visible.”



Developmental Controls, Information hiding

- Module is a black box
  - Well defined function and I/O
- Easy to know what module does but not how it does it
- Reduces complexity, interactions, covert channels, ...

  => better security

Developmental Controls, building solid software

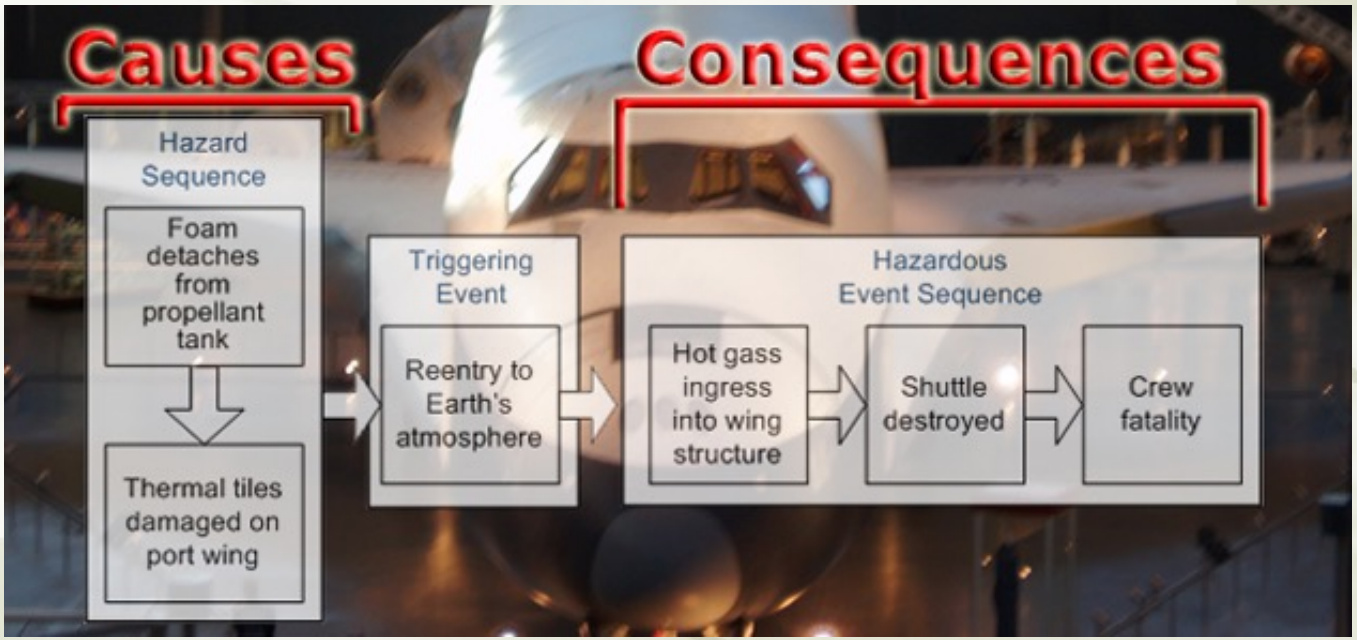
- Peer reviews
- Hazard analysis
- Testing
- Good design
- Risk prediction & management
- Static analysis
- Configuration management
- Additional developmental controls

Developmental Controls, Peer reviews

- Three types of reviews
  - Reviews
    - Informal
    - Team of reviewers
    - Gain consensus on solutions before development
  - Walk-throughs
    - Developer walks team through code/document
    - Discover flaws in a single design document
  - Inspection
    - Formalized and detailed
    - Statistical measures used

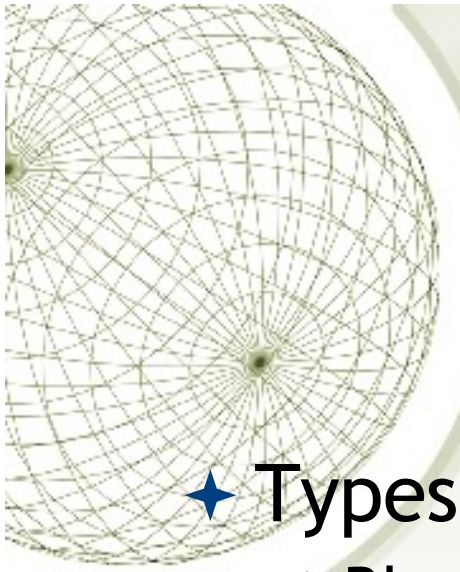
Developmental Controls, Hazard analysis

- Systematic exposure of hazardous system states
- Technique
  - Hazard lists
  - What-if scenarios
  - System-wide view (not just code)
  - Begins Day 1
  - Continues throughout SDLC
- Tools
  - HAZOP
  - FMEA
  - FTA



Developmental Controls, Testing

- Testing phases:
  - Module/component/unit, testing of indiv. modules
  - Integration testing of interacting (sub)system modules
  - (System) function testing checking against requirement specs
  - (System) performance testing
  - (System) acceptance testing – with customer against customer's requirements — on seller's or customer's premises
  - (System) installation testing after installation on customer's system
  - Regression testing after updates/changes to s/w

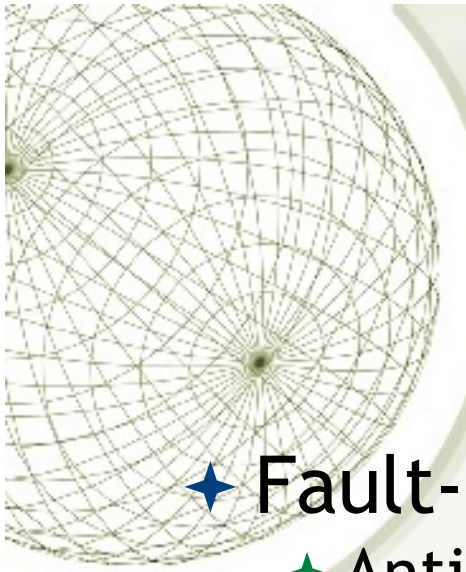


Developmental Controls, Testing, cont.

- Types of testing
  - Black Box testing – testers can't examine code
  - White Box / Clear box testing – testers can examine design and code, can see inside modules/system

Developmental Controls, Good design

- Good design uses:
  - Modularity / encapsulation / info hiding (as discussed earlier)
  - Fault tolerance
  - Consistent failure handling policies
  - Design rationale and history
  - Design patterns

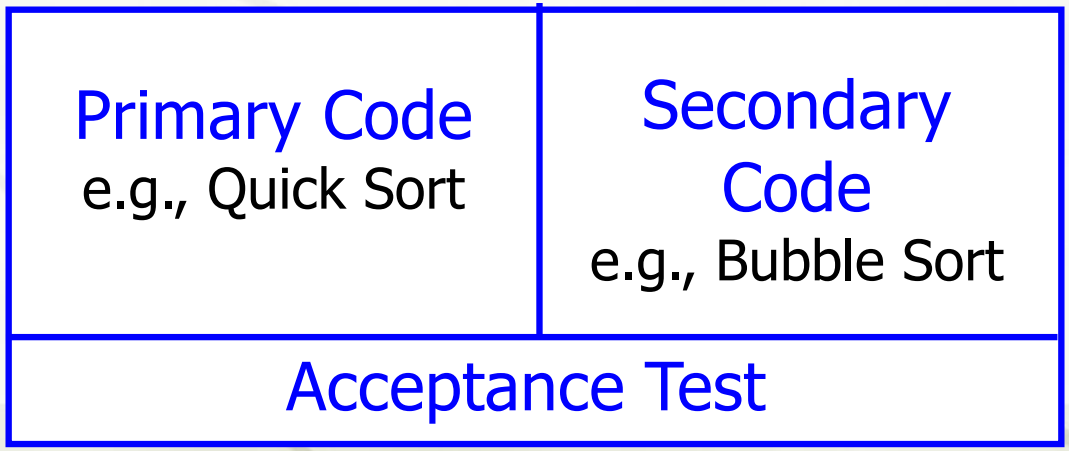


Developmental Controls, Good design, cont.

- Fault-tolerant approach:
  - Anticipate faults
    - (car: anticipate having a flat tire)
  - Active fault detection rather than passive fault detection
    - (e.g., by use of mutual suspicion: active input data checking)
  - Use redundancy
    - (car: have a spare tire)
  - Isolate damage
  - Minimize disruption
    - (car: replace flat tire, continue your trip)

Developmental Controls, Good design, cont.

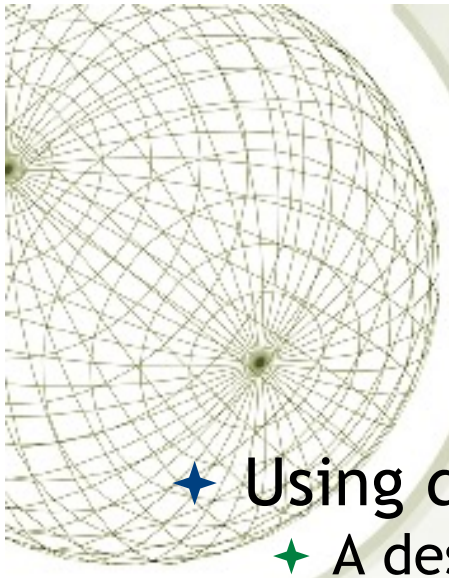
- Fault-tolerant Example 1: Majority voting (using h/w redundancy)
  - 3 processor running the same s/w
  - E.g., in a spaceship
  - Result accepted if results of ≥ 2 processors agree
- Fault-tolerant Example 2: Recovery Block (using s/w redundancy)



Quick Sort –
– new code (faster)
Bubble Sort –
– well-tested code

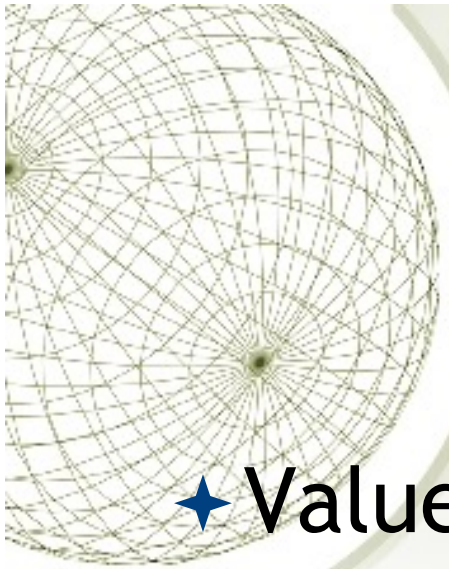
Developmental Controls, Good design, cont.

- Using consistent failure handling policies
- Each failure handled in one of three ways:
  - Retrying
    - Restore previous state, redo service using different „path”
    - E.g., use secondary code instead of primary code
  - Correcting
    - Restore previous state, correct sth, run service using the same code as before
  - Reporting
    - Restore previous state, report failure to error handler, don’t rerun service



Developmental Controls, Good design, cont.

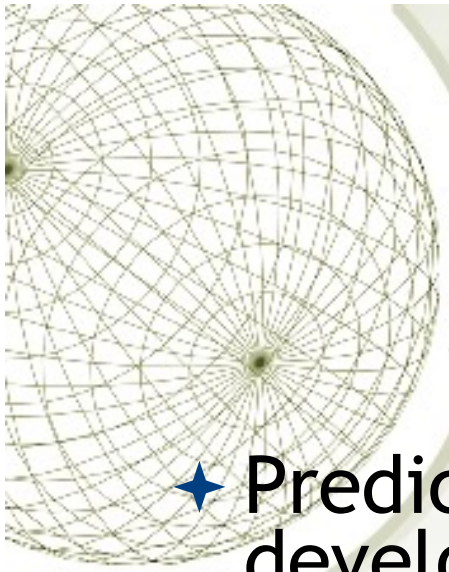
- Using design rationale and history
  - A design rationale is the explicit listing of decisions made during a design process, and the reasons why those decisions were made
  - Knowing it (incl. knowing design rationale and history for security mechanisms) helps developers modifying or maintaining system
- Using design patterns
  - Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code
  - Knowing it enables looking for patterns showing what works best in which situation



Developmental Controls, Good design, cont.

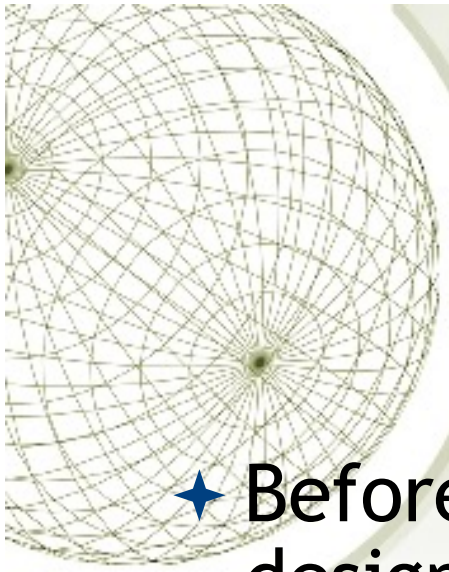
- Value of Good Design
  - Easy maintenance
  - Understandability
  - Reuse
  - Correctness
  - Better testing

=> translates into (saving) BIG bucks and a reduction in security problem



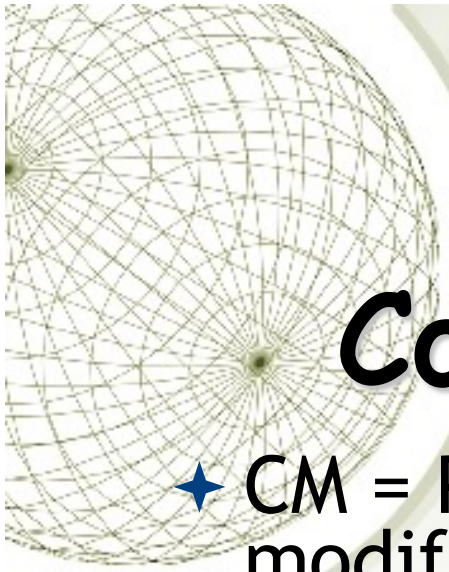
Developmental Controls, Risk prediction & management

- Predict and manage risks involved in system development and deployment
  - Make plans to handle unwelcome events should they occur
- Risk prediction/management are especially important for security
  - Because unwelcome and rare events can have security consequences
- Risk prediction/management helps to select proper security controls (e.g., proportional to risk)



Developmental Controls, Static analysis

- Before system is up and running, examine its design and code to locate security flaws
  - More than peer review
- Examines
  - Control flow structure (sequence in which instructions are executed, incl. iterations and loops)
  - Data flow structure (trail of data)
  - Data structures
- Automated tools available



Developmental Controls, Configuration management

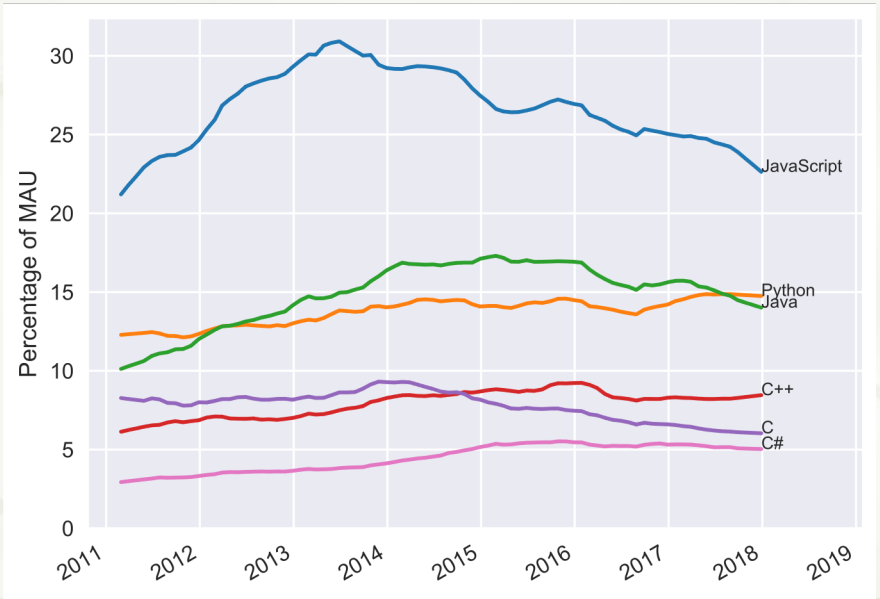
- CM = Process of controlling system modifications during development and maintenance
  - Offers security benefits by scrutinizing new/changed code
- Problems with system modifications
  - One change interfering with other change
    - E.g., neutralizing it
  - Proliferation of different versions and releases
    - Older and newer
    - For different platforms
    - For different application environments

Developmental Controls, Additional developmental controls

- Learning from mistakes
  - Avoiding such mistakes in the future enhances security
- Proofs of program correctness
  - Formal methods to verify pgm correctness
  - Problems with practical use of pgm correctness proofs
    - Esp. for large pgms/systems
  - Most successful for specific types of apps
    - E.g. for communication protocols & security policies
- Even with all these developmental controls – still no security guarantees!

The role of the program language for Program Security

- Here's how the seven most widely-used coding languages stack up when it comes to the total open source security vulnerabilities per language[1] and popularity in languages[2]
  - C (47%)
  - PHP (17%)
  - Java (11%)
  - JavaScript (10%)
  - Python (5%)
  - C++ (5%)
  - Ruby (4%)



1) Source: https://www.techrepublic.com/article/the-3-least-secure-programming-languages/ (2019)
2) Source: https://www.benfrederickson.com/ranking-programming-languages-by-github-users/

Operating System Controls for Security

- Developmental controls not always used

OR:

- Even if used, not foolproof

  => Need other, complementary controls, incl. OS controls

- Such OS controls can protect against some pgm flaws

Operating System Controls for Security, Trusted software

- OS code is rigorously developed an analysed so we can trust that it does all and only what specs say
- Trusted code establishes foundation upon which untrusted code runs
  - Trusted code establishes security baseline for the whole system
- In particular, OS can be trusted s/w

Operating System Controls for Security, key characteristics

- Key characteristics determining if OS code is trusted
  - Functional correctness
    - OS code consistent with specs
  - Enforcement of integrity
    - OS keeps integrity of its data and other resources even if presented with flawed or unauthorized commands
  - Limited privileges
    - OS minimizes access to secure data/resources
    - Trusted pgms must have ”need to access” and proper access rights to use resources protected by OS
    - Untrusted pgms can’t access resources protected by OS
  - Appropriate confidence level
    - OS code examined and rated at appropriate trust level

Operating System Controls for Security, increasing security

- Similar criteria used to establish if s/w other than OS can be trusted

- Ways of increasing security if untrusted pgms present:
  - Mutual suspicion
  - Confinement
  - Access log

Operating System Controls for Security, Mutual suspicion between programs

- Distrust other pgms – treat them as if they were incorrect or malicious
  - Pgm protects its interface data
    - With data checks, etc.

Operating System Controls for Security, Confinement

- OS can confine access to resources by suspected pgm
- Example 1: strict compartmentalization
  - Pgm can affect data and other pgms *only* within its compartment
- Example 2: sandbox for untrusted pgms
- Can limit spread of viruses

Operating System Controls for Security, Audit log / access log

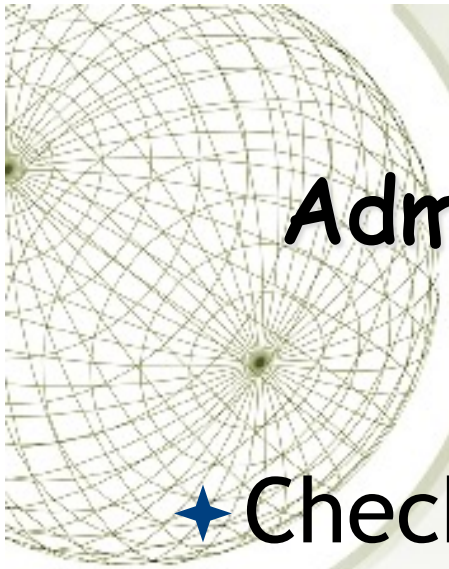
- Records who/when/how (e.g., for how long) accessed/used which objects
  - Events logged: logins/logouts, file accesses, pgm ecxecutions, device uses, failures, repeated unsuccessful commands (e.g., many repeated failed login attempts can indicate an attack)
- Audit frequently for unusual events, suspicious patterns
- Forensic measure not protective measure
  - Forensics – investigation to find who broke law, policies, or rules

Administrative Controls for Security

- They prohibit or demand certain human behaviour via policies, procedures, etc.
- They include:
  - Standards of program development
  - Security audits
  - Separation of duties

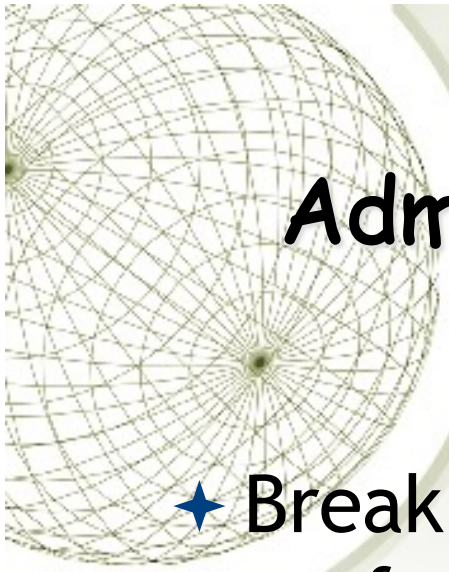
Administrative Controls for Security,
Standards and guidelines (S&G) for program development

- Capture experience and wisdom from previous projects
- Facilitate building higher-quality s/w (incl. more secure)
- They include:
  - Design S&G – design tools, languages, methodologies
  - S&G for documentation, language, and coding style
  - Programming S&G - incl. reviews, audits
  - Testing S&G
  - Configuration management S&G



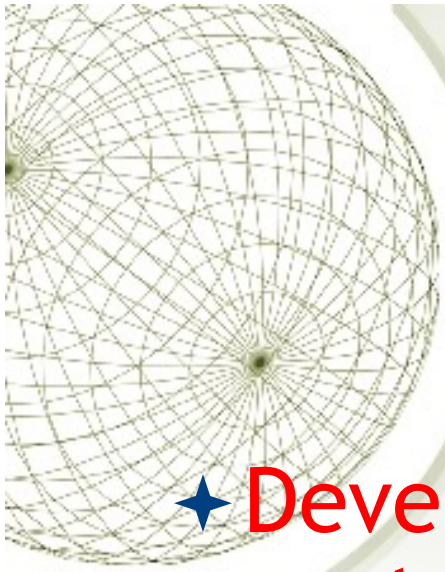
Administrative Controls for Security, Security audits

- Check compliance with S&G
- Scare potential dishonest programmer from including illegitimate code (e.g., a trapdoor)



Administrative Controls for Security, Separation of duties

- Break sensitive tasks into ≥ 2 pieces to be performed by different people (learned from banks)
- Example 1: modularity
  - Different developers for cooperating modules
- Example 2: independent testers
  - Rather than developer testing her own code



Conclusions
(for Controls for Security)

- Developmental / OS / administrative controls help produce/maintain higher-quality (also more secure) s/w
- Art and science - no ”silver bullet” solutions
- ”A good developer who truly understands security will incorporate security into all phases of development.”

Summary

Control	Purpose	Benefit
Develop-mental	Limit mistakes Make malicious code difficult	Produce better software
Operating System	Limit access to system	Promotes safe sharing of info and computing resources
Adminis-trative	Limit actions of people	Improve usability, reusability and maintainability