THE UNIVERSITY OF BRITISH COLUMBIA

Developing Secure Software

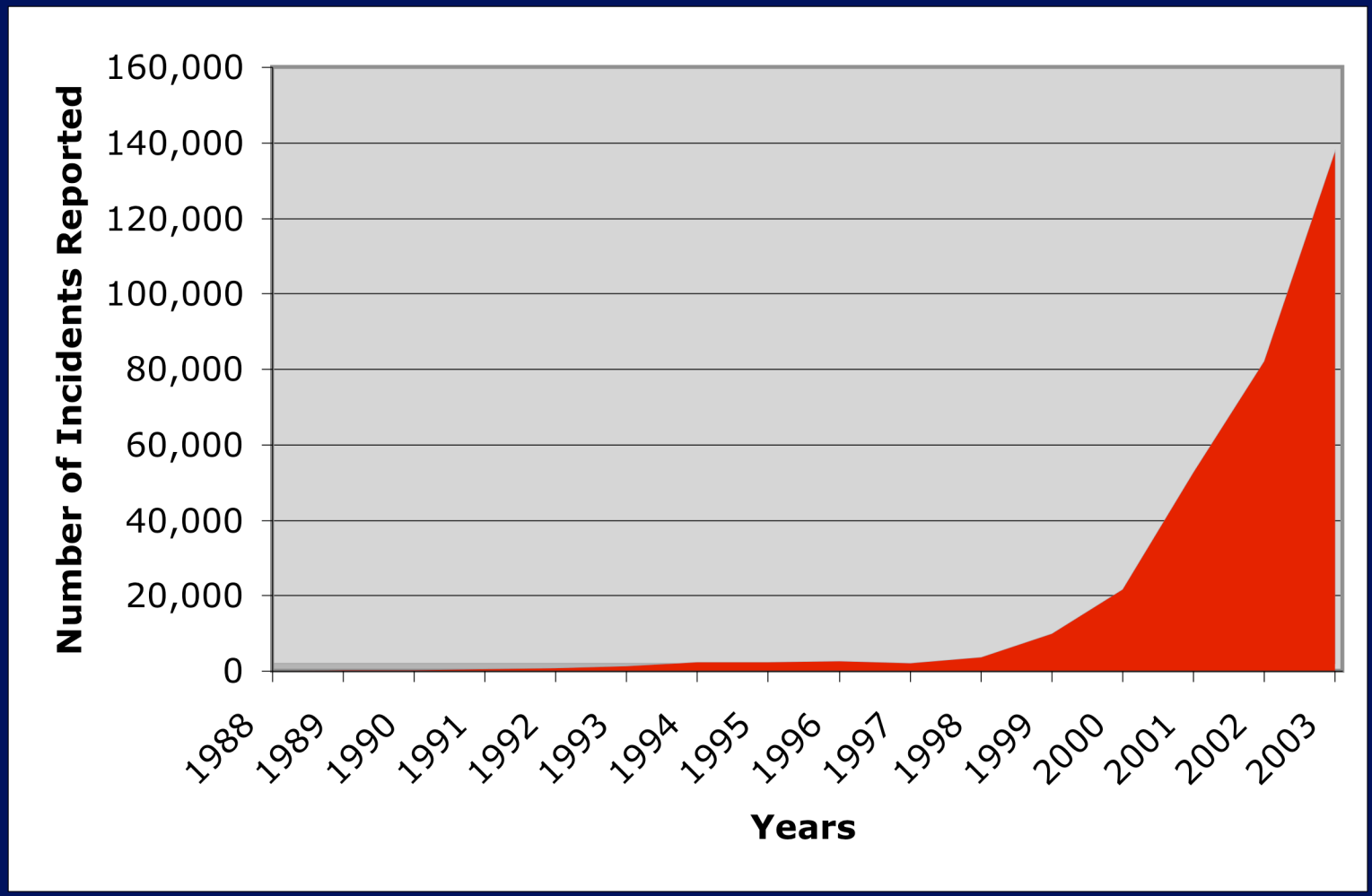
EECE 412

Session 21

THE UNIVERSITY OF BRITISH COLUMBIA

What's cell phones, ATMs, air traffic control systems, emergency service systems, healthcare equipment, and PDAs have in common?

Internet security incidents reported to CERT

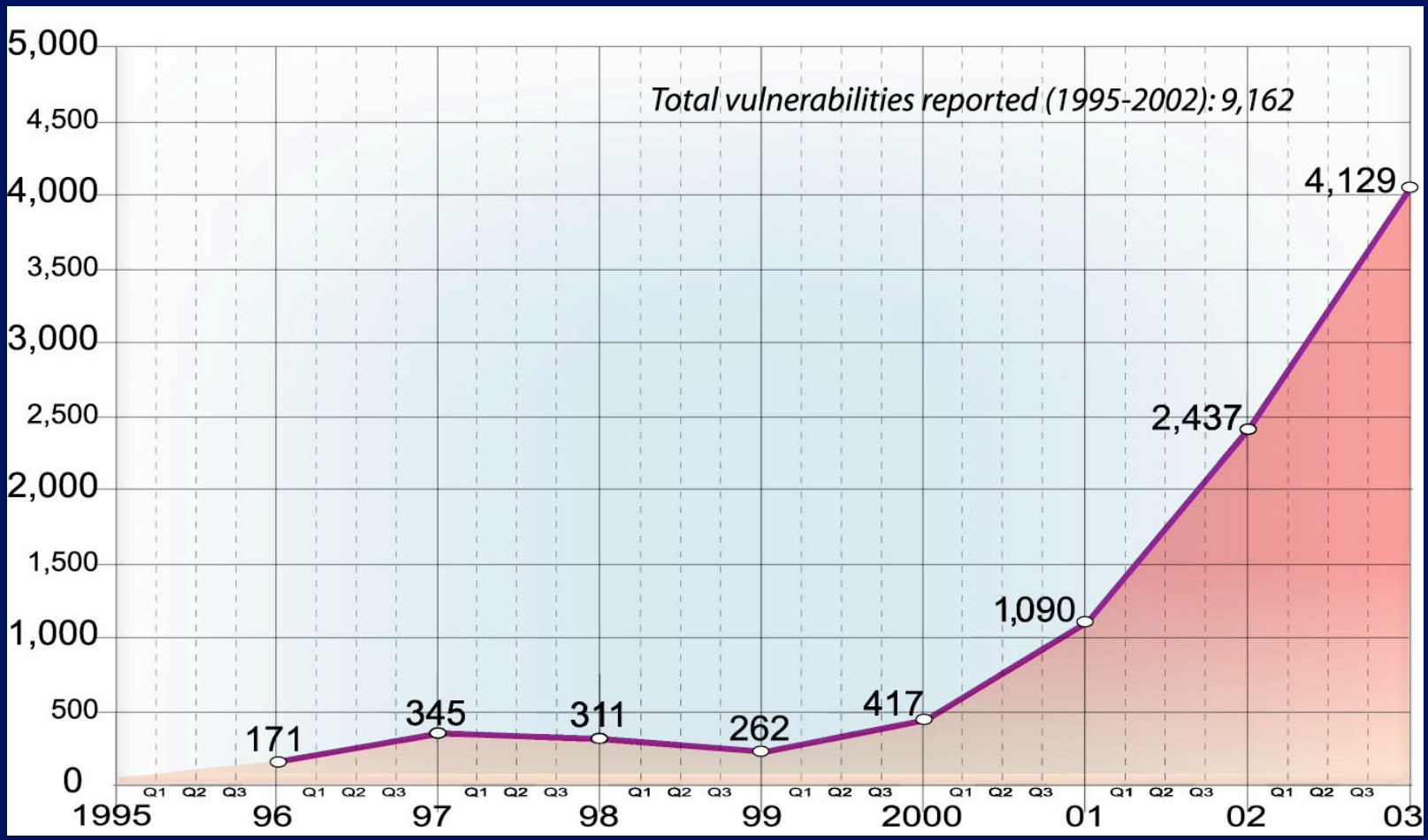


Security break-ins are all too prevalent

THE UNIVERSITY OF BRITISH COLUMBIA

Even if your computer is not compromised, how costly is it to keep it “secure” enough?

Vulnerability Report Statistics



The Scale in Human Terms

What does it mean to have 4,129 vulnerabilities reported in 2002?

- Read the descriptions
  - 4,129 vulnerabilities * 15 minutes = 129 days
- Affected by 10% of the vulnerabilities?
  - Install patches on one system
    - 413 vulnerabilities * 1 hour = 52 days
  - Reading reports and patching a single system costs 129 + 52 = 181 days
- Which vulnerability should I patch first? Remote root in DNS? Web server? Desktop systems? Network equipment?

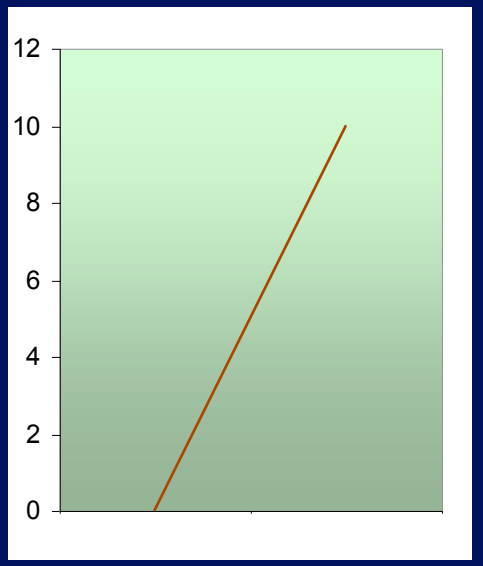
Outline

- Why software systems are prone to vulnerabilities?
- How vulnerabilities are different from defects?
- How to develop secure software?

THE UNIVERSITY OF BRITISH COLUMBIA

Why are there so many vulnerabilities in software?

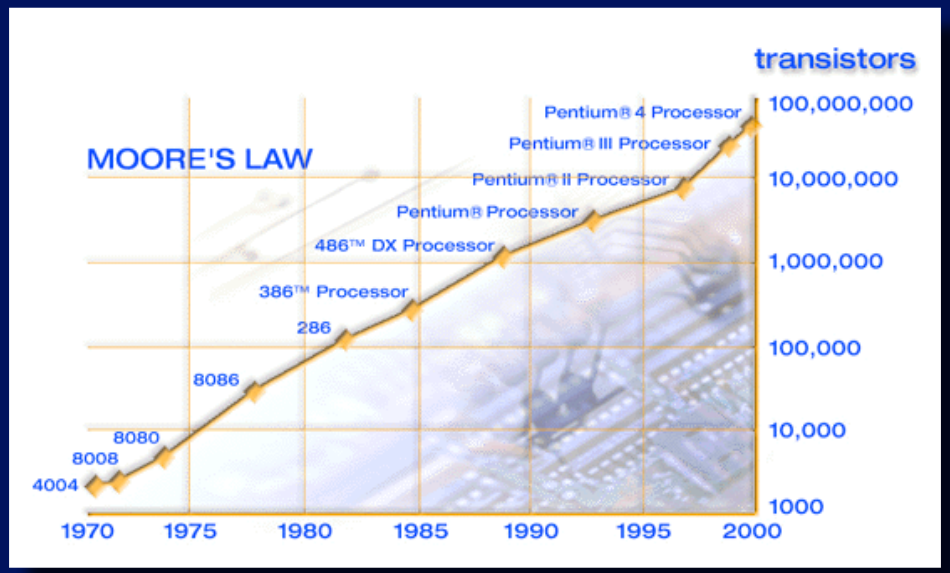
What will happen in a moment?

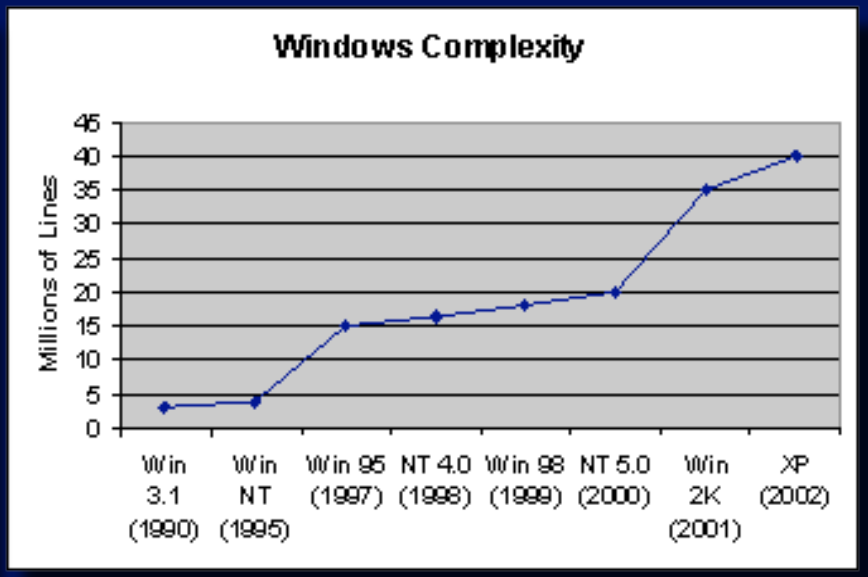


What makes simple mechanical systems predictable?

- Linearity (or, piecewise linearity)
- Continuity (or, piecewise continuity)
- Small, low-dimensional statespaces

Systems with these properties are
(1) easier to analyze, and (2) easier to test.



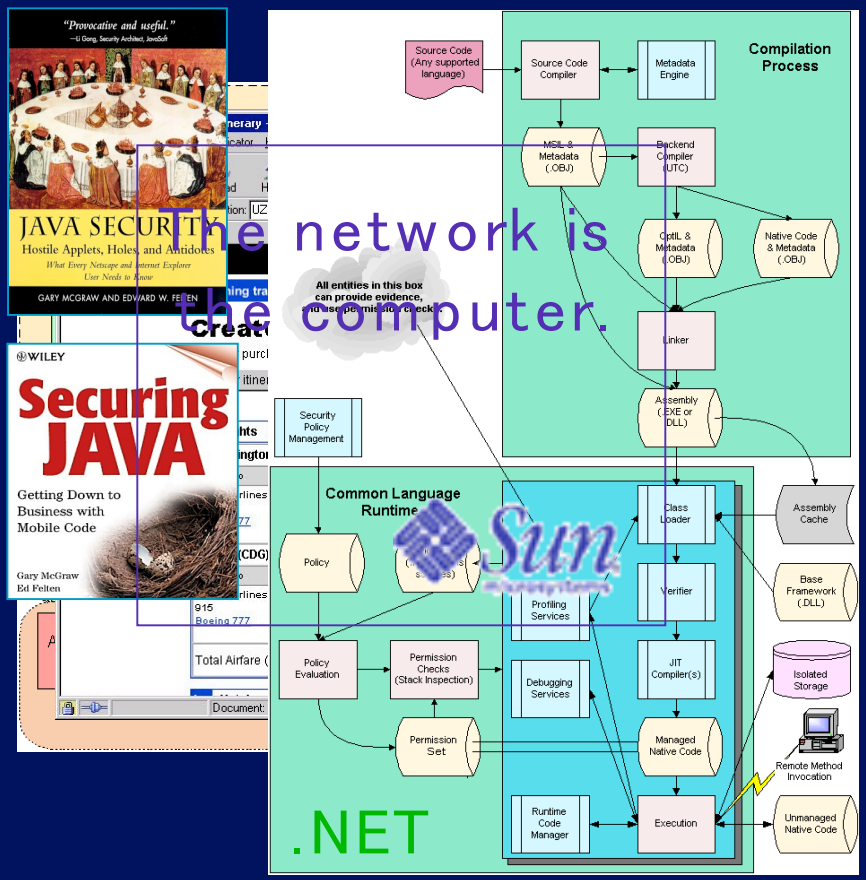


- Computers enable highly complex systems
- Software is taking advantage of this
  - Highly non-linear behavior; large, high-dim. state spaces

Other software properties make security difficult

The Trinity of Trouble

- **Connectivity**
  - The Internet is everywhere and most software is on it
- **Complexity**
  - Networked, distributed, mobile, feature-full
- **Extensibility**
  - Systems evolve in unexpected ways and are changed on the fly



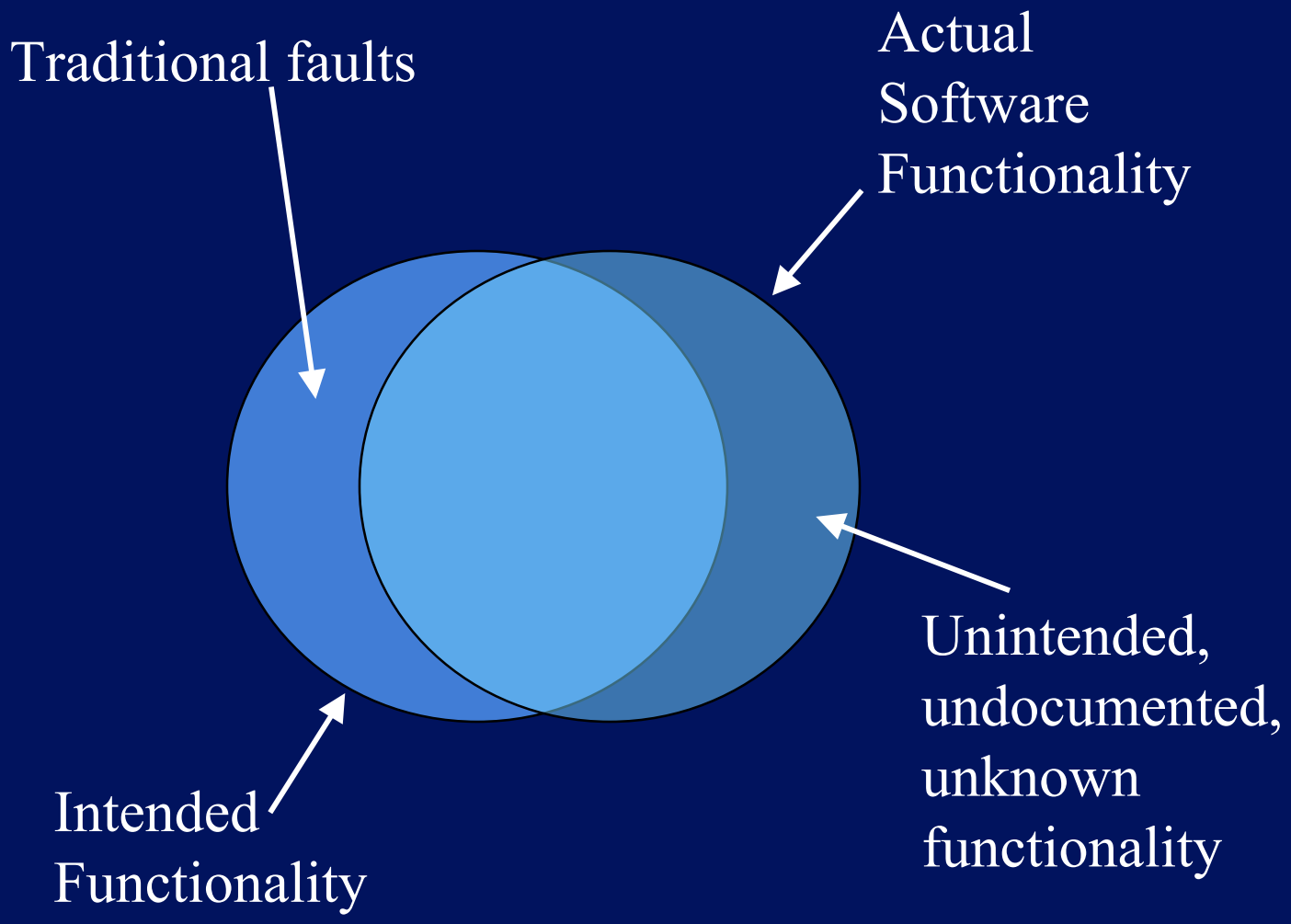
THE UNIVERSITY OF BRITISH COLUMBIA

How Are Security Bugs Different?

When is a security bug not like a bug?

- Traditional non-security bugs -- often defined as a violation of a specification.
- Security bugs -- additional behavior, not originally intended
  - Meanwhile, it is doing what it is supposed to do
  - Traditional techniques not good at finding
  - Even in inspections, tend to look for
    - missing behavior
    - incorrect behavior
  - Neglect to look for ... undesirable side-effects

Intended vs. Implemented Behavior



Traditional faults

- Incorrect
  - Supposed to do A but did B instead
- Missing
  - Supposed to do A *and* B but did only A.

Security Bugs

- Side effects
  - Supposed to do A, and it did.
  - In the course of doing A, it *also* did B
- Monitoring for side effects and their impact on security can be challenging
  - Side effects can be subtle and hidden
  - Examples: file writes, registry entries, extra network packets with unencrypted data

Security problems are complicated

Implementation Flaws

- Buffer overflow
  - String format
- Race conditions
  - TOCTOU (time of check to time of use)
- Unsafe environment variables
- Unsafe system calls
  - System()
- Untrusted input problems

Design Flaws

- Misuse of cryptography
- Compartmentalization problems in design
- Privileged block protection failure (DoPrivilege())
- Catastrophic security failure (fragility)
- Type safety confusion error
- Insecure auditing
- Broken or illogical access control
- Method over-riding problems (subclass issues)

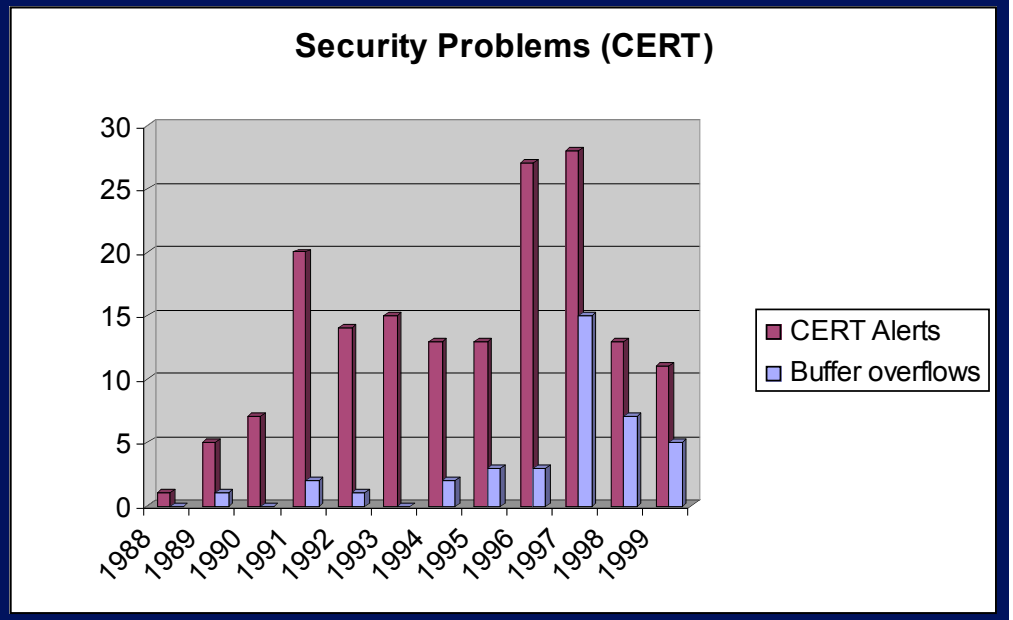
Which ones are more frequent?

The BUG: buffer overflow

- Overwriting the bounds of data objects
- Allocate some bytes, but the language doesn't care if you try to use more
  - `char x[12 ];`

    `x[12 ] = '\0';`
- Why was this done? Efficiency!

The most pervasive security problem today



Pervasive C problems lead to BUGS

```
void main() {
  char
  buf[1024];
  gets(buf);
}
```

- How not to get input
  - Attacker can send an infinite string!

- Calls to watch out for

Instead of:	Use:
gets(buf)	fgets(buf, size, stdin)
strcpy(dst, src)	strncpy(dst, src, n)
strcat(dst, src)	strncat(dst, src, n)
sprintf(buf, fmt, a1,…)	snprintf(buf, fmt, a1, n1,…) (where available)
*scanf(…)	Your own parsing

- Hundreds of such calls
- Use static analysis to find these problems
  - ITS4, SourceScope
- Careful code review is necessary

THE UNIVERSITY OF BRITISH COLUMBIA

How to Develop Secure Software?

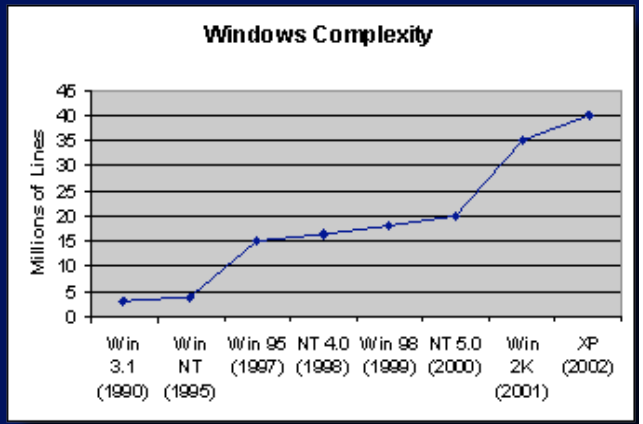
Some Guidelines

1. Reduce the number of all defects by order of magnitude
2. Build security in your development process from beginning
3. Practice principles of designing secure systems
4. Know how systems can be compromised
5. Develop and use guidelines and checklists
6. Choose safer languages, VMs, OSs, etc.
7. Provide tool support

Why Software Quality is Important?

According to CERT/CC:

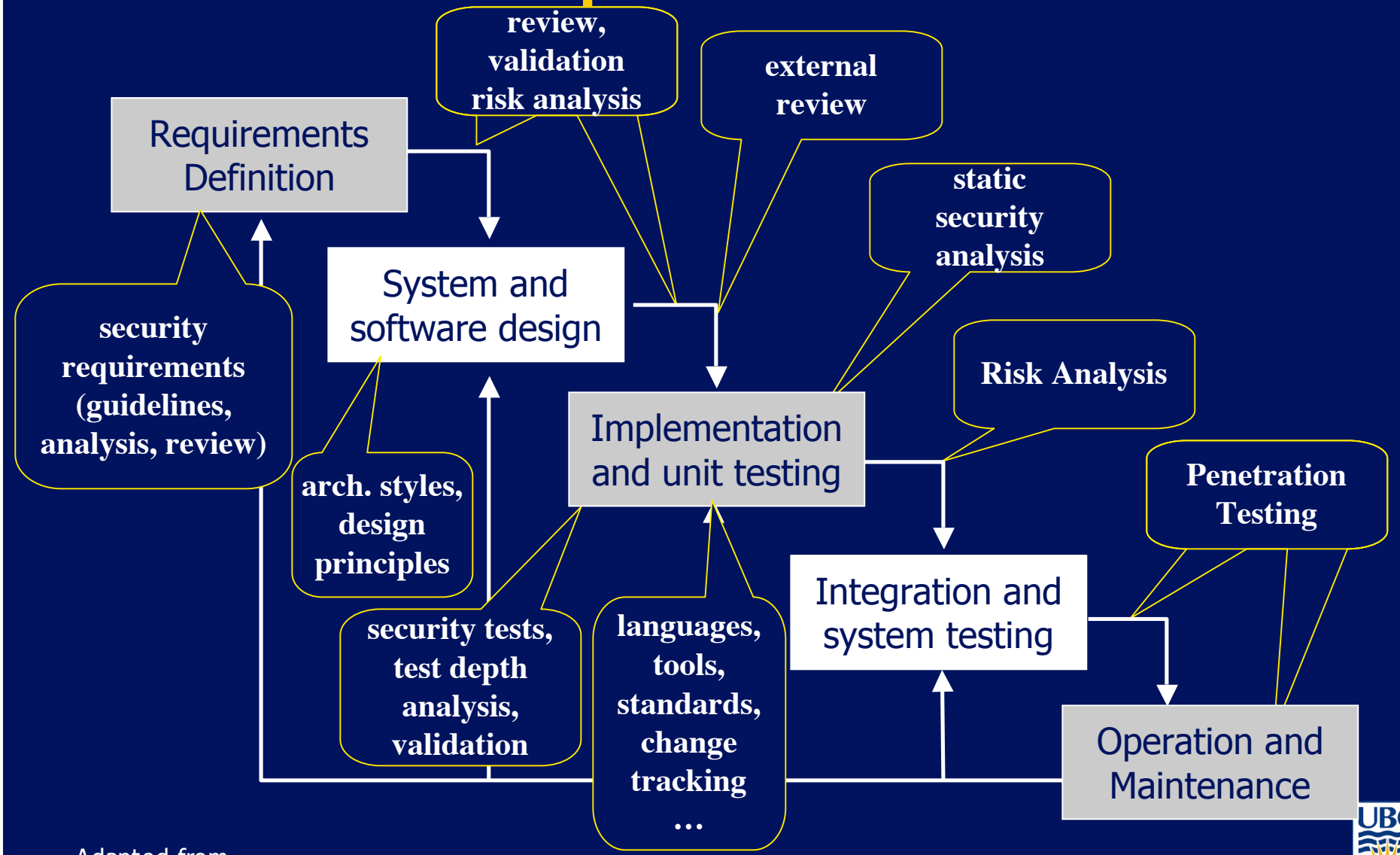
- over 90% of software security vulnerabilities are caused by known software defect types
- most software vulnerabilities arise from common causes
  - top ten account for 75% of vulnerabilities
- One design or implementation defect is injected for every 7 to 10 lines of new and changed code produced
  - Even if 99% is removed, 1/1K left (40K defects in Win XP)



1. Produce Quality Software

- Use well structured effective processes
  - e.g., Capability Maturity Model (CMM), *-CMM
- Use precise requirements and specifications
  - Formal methods
    - e.g., Praxis Critical Systems approach
      - 0.75-0.04 defects/KLOC
    - CleanRoom
      - 0.08 defects/KLOC

2. Build Security into Development Process



Adapted from
D. Verdon and G. McGraw, "Risk analysis in software design," *IEEE Security & Privacy*, vol. 2, no. 4, 2004, pp. 79-84.

Follow Best Practices

- These best practices should be applied throughout the lifecycle
- Tendency is to "start at the end" (penetration testing) and declare victory
  - Not cost effective
  - Hard to fix problems
- Start as early as possible
- Abuse cases
- Security requirements analysis
- Architectural risk analysis
- Risk analysis at design
- External review
- Test planning based on risks
- Security testing (malicious tests)
- Code review with static analysis tools

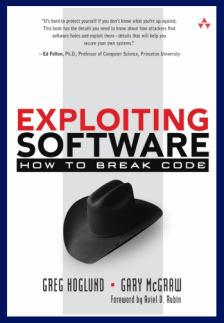
3. Practice principles of designing secure systems

Principles of Designing Secure Systems

1. Least Privilege
2. Fail-Safe Defaults
3. Economy of Mechanism
4. Complete Mediation
5. Open Design
6. Separation of Privilege
7. Least Common Mechanism
8. Psychological Acceptability
9. Defense in depth
10. Question assumptions

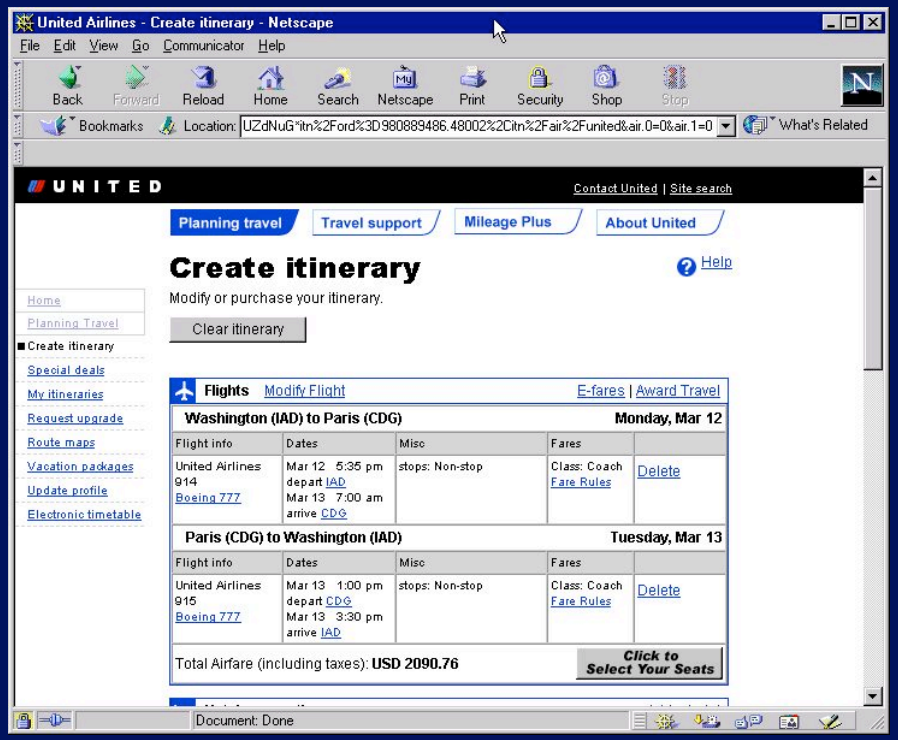
4. Know How Systems Can Be Compromised

1. Make the Client Invisible
2. Target Programs That Write to Privileged OS Resources
3. Use a User-Supplied Configuration File to Run Commands That Elevate Privilege
4. Make Use of Configuration File Search Paths
5. Direct Access to Executable Files
6. Embedding Scripts within Scripts
7. Leverage Executable Code in Nonexecutable Files
8. Argument Injection
9. Command Delimiters
10. Multiple Parsers and Double Escapes
11. User-Supplied Variable Passed to File System Calls
12. Postfix NULL Terminator
13. Postfix, Null Terminate, and Backslash
14. Relative Path Traversal
15. Client-Controlled Environment Variables
16. User-Supplied Global Variables (DEBUG=1, PHP Globals, and So Forth)
17. Session ID, Resource ID, and Blind Trust
18. Analog In-Band Switching Signals (aka "Blue Boxing")
19. Attack Pattern Fragment: Manipulating Terminal Devices
20. Simple Script Injection
21. Embedding Script in Nonscript Elements
22. XSS in HTTP Headers
23. HTTP Query Strings
24. User-Controlled Filename
25. Passing Local Filenames to Functions That Expect a URL
26. Meta-characters in E-mail Header
27. File System Function Injection, Content Based
28. Client-side Injection, Buffer Overflow
29. Cause Web Server Misclassification
30. Alternate Encoding the Leading Ghost Characters
31. Using Slashes in Alternate Encoding
32. Using Escaped Slashes in Alternate Encoding
33. Unicode Encoding
34. UTF-8 Encoding
35. URL Encoding
36. Alternative IP Addresses
37. Slashes and URL Encoding Combined
38. Web Logs
39. Overflow Binary Resource File
40. Overflow Variables and Tags
41. Overflow Symbolic Links
42. MIME Conversion
43. HTTP Cookies
44. Filter Failure through Buffer Overflow
45. Buffer Overflow with Environment Variables
46. Buffer Overflow in an API Call
47. Buffer Overflow in Local Command-Line Utilities
48. Parameter Expansion
49. String Format Overflow in syslog()



Attack pattern example: Make the client invisible

- Remove the client from the communications loop and talk directly to the server
- Leverage incorrect trust model (never trust the client)
- Example: hacking browsers that lie



5. Develop Guidelines and Checklists

Example from Open Web Application Security Project (www.owaspp.org):

- Validate Input and Output
- Fail Securely (Closed)
- Keep it Simple
- Use and Reuse Trusted Components
- Defense in Depth
- Security By Obscurity Won't Work
- Least Privilege: provide only the privileges absolutely required
- Compartmentalization (Separation of Privileges)
- No homegrown encryption algorithms
- Encryption of all communication must be possible
- No transmission of passwords in plain text
- Secure default configuration
- Secure delivery
- No back doors

Secure Programming How-Tos

- David Wheeler's Secure Programming for Linux and UNIX How-To
  - http://www.dwheeler.com/secure-programs
- Secure UNIX Programming FAQ
  - http://www.whitefang.com/sup/secure-faq.html
- OWASP (Open Web Application Security Project) Guide
  - http://www.owasp.org
- Etc... (Google "secure programming")

6. Choose Safer Languages, VMs, OSs, etc.

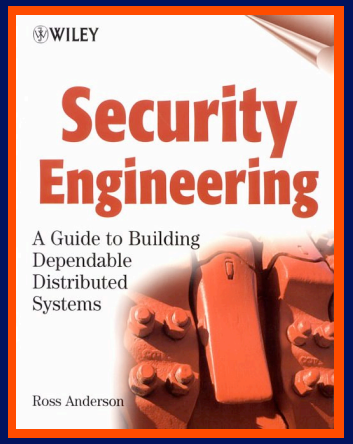
- C or C++?
- Java or C++?
- Managed C++ or vanilla C++?
- .NET CLR or JVM?
- Windows XP or Windows 2003?
- Linux/MacOS/Solaris or Windows?

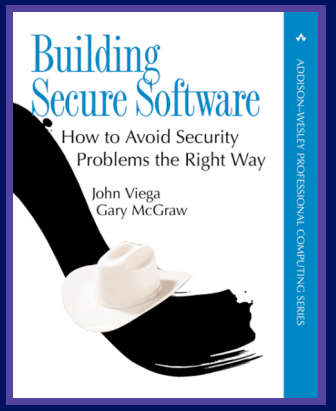
7. Make Developers' Life Easier: Give Them Good Tools

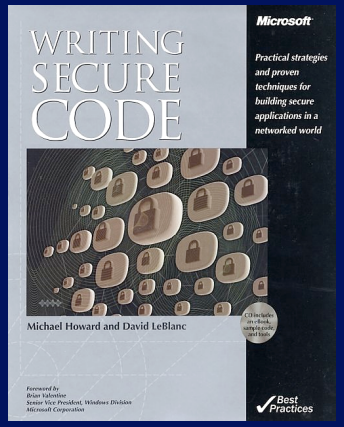
- automated tools for formal methods
  - http://www.comlab.ox.ac.uk/archive/formal-methods.html
- code analysis tools
  - RATS http://www.securesw.com/rats
  - Flawfinder http://www.dwheeler.com/flawfinder
  - ITS4 http://www.cigital.com/its4
  - ESC/Java http://www.niii.kun.nl/ita/sos/projects/escframe.html
  - PREfast, PREfix, SLAM www.research.microsoft.com
  - Fluid http://www.fluid.cmu.edu
  - JACKPOT research.sun.com/projects/jackpot
  - Many more …

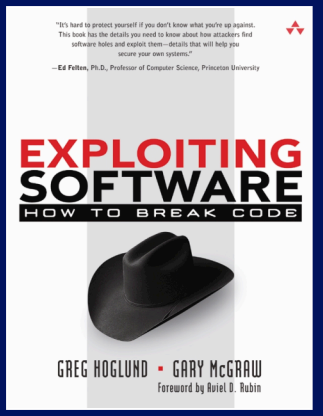
Relevant Books

- High Level
  - Secure Coding, Principles and Practices (M.G. Graff and K.R. Van Wyk 2003)
- Technical
  - Secure Programming Cookbook (J. Viega and M. Messier)
  - Writing Secure Code, 2nd Edition (Howard and Leblanc)









Free Relevant Books

- Improving Web Application Security: Threats and Countermeasures Roadmap
  - J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla and Anandha Murukan
    Microsoft Corporation
  - MSDN Library, June 2003
  - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp

Summary

- Why software systems are prone to vulnerabilities?
- How vulnerabilities are different from defects?
- How to develop secure software?