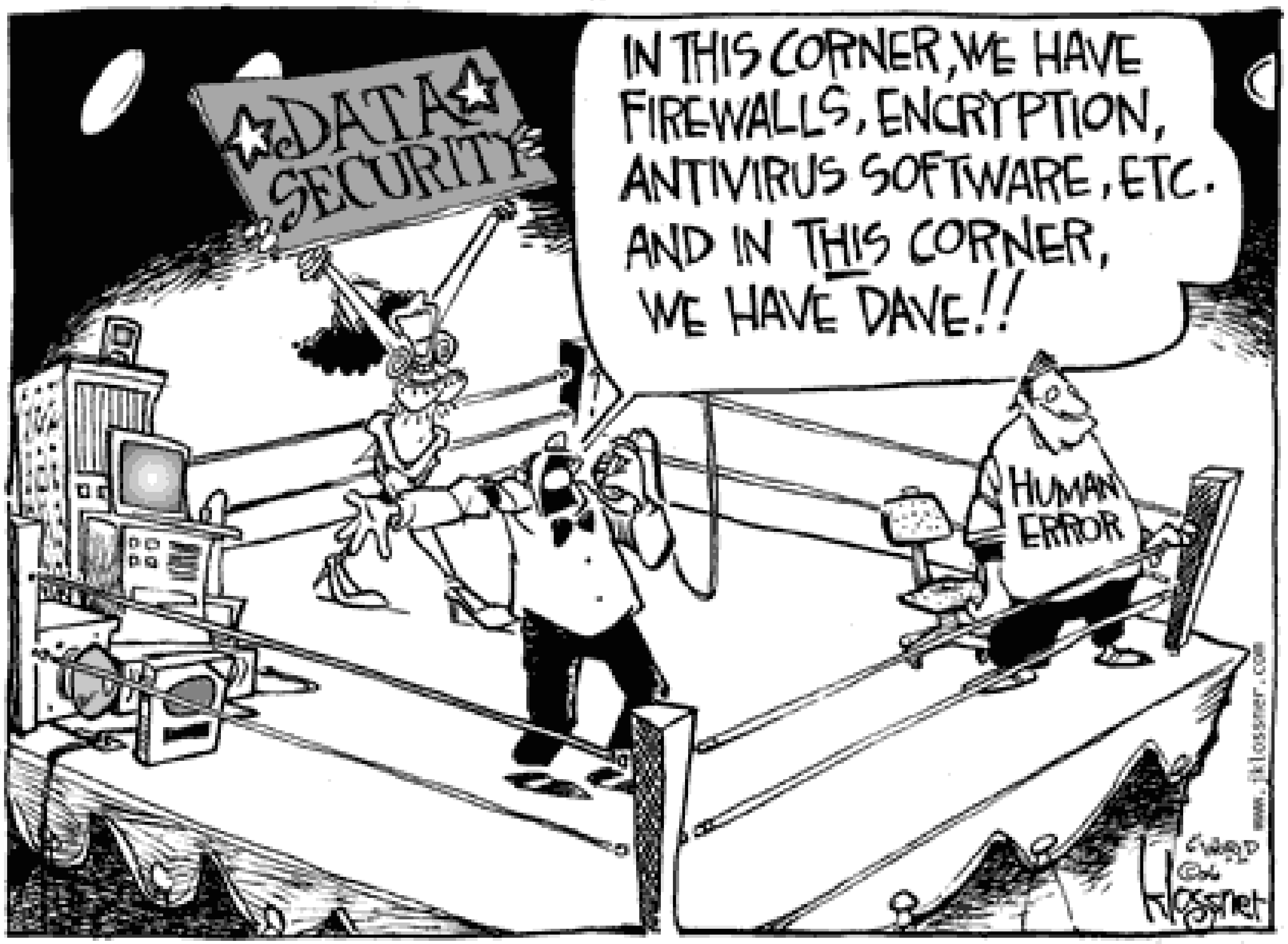
# The Software Security Problem

**Thomas L. "Trey' Jones, CISSP, CEH**

## Weakest Link


DATA SECURITY
IN THIS CORNER, WE HAVE FIREWALLS, ENCRYPTION, ANTIVIRUS SOFTWARE, ETC. AND IN THIS CORNER, WE HAVE DAVE!!
HUMAN ERROR
www.jklossner.com
Klossner

## Engineers are Taught

Pay attention to the theory and math or this happens!

## Programmers are Taught

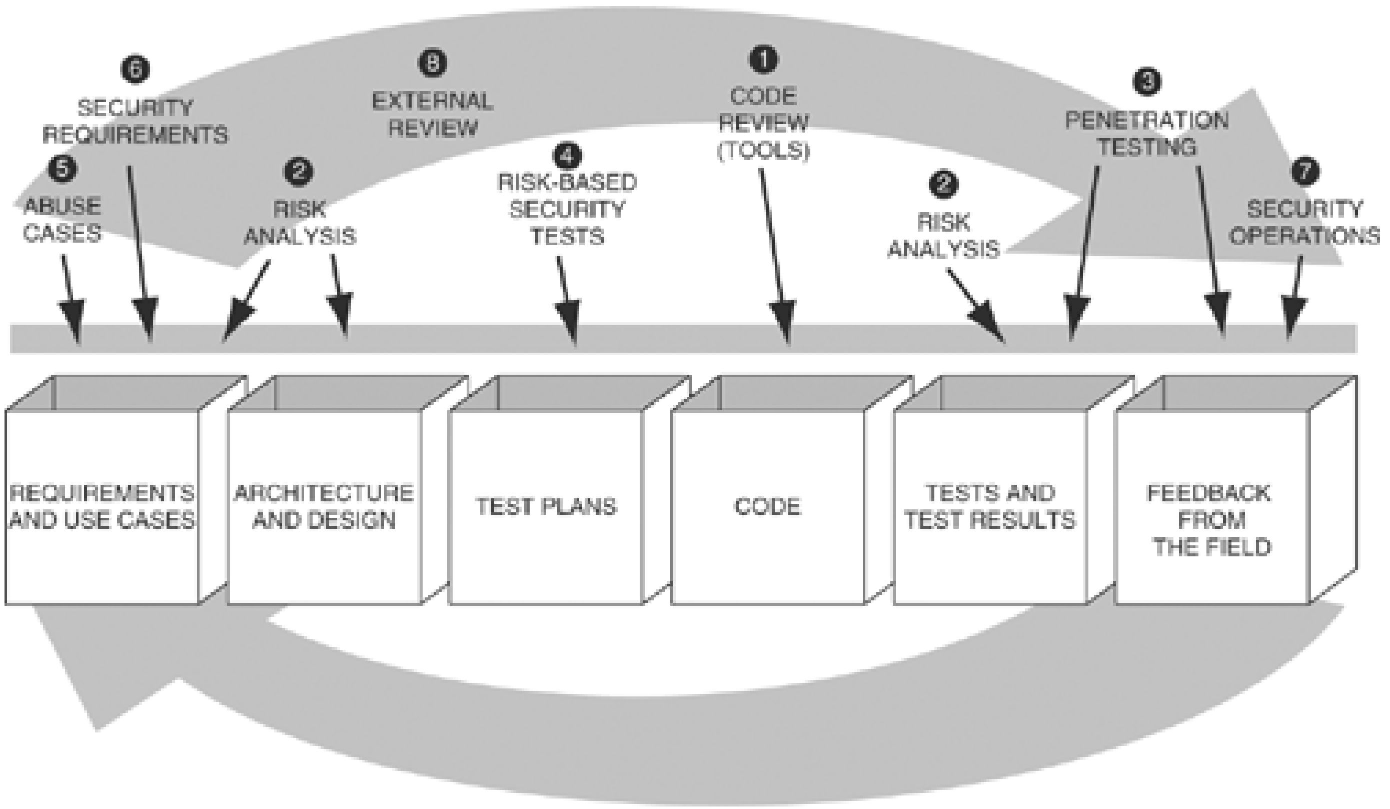
Look how easy it is to program!

```
#include <stdio.h>

int main()
{
    printf("Hello World");
    return 42;
}
```

**Most software is fragile and barely executes.**

## The Seven Touchpoints

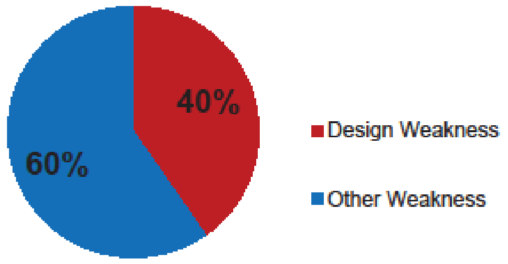


Touchpoints identify the order in which a software development team should shift to producing more secure code.

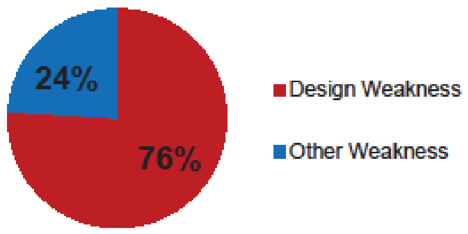
## SEI Study

### Importance of Good Design

940 Total CWEs*



Top 25 CWEs
(Most Dangerous)



*MITRE's Common Weakness Enumeration (CWE)

Source: http://cwe.mitre.org/ as of Feb 9, 2014

## Software vs Application Security

- Gary McGraw has provided an excellent description of the difference between these two terms:
  - **Software** security is about building secure software
    - Designing software to be secure
    - Making sure that software is secure
    - Educating software developers, architects, and users about how to build security in
  - **Application** security is about protecting software and the systems that software runs in a post facto, only after development is complete

## System Weaknesses

### (from the "safety domain")

- Bugs *(sometimes called "flaws or weaknesses" in the security domain)*
  - Occur at Implementation (lower) level
  - Only exist in code
  - Can often be fixed in a single line of code
- Flaws
  - Can exist at all (code / design / requirements) levels
  - Subtle problems that are instantiated in code, but are also present (or missing) in the design
  - Design (or requirement) flaws may require a redesign that can affect multiple areas in the system
- <u>Defects</u> encompass both implementation (bugs) and design (flaws) problems
  - May lie dormant for years and surface later in a fielded system
  - Give way to major consequences

**Airbus Auto-land Incident (N-version programming)**
**https://www.atsb.gov.au/media/24388/aair200705576_Prelim.pdf**

## Security Definition

Basic definition of Vulnerability

- refers to the inability to withstand the effects of a hostile environment
- open to attack or damage

Defenders can only control these!

Cyber Vulnerability (CISSP BoK)

1. A flaw* (aka weakness) exists in the system
2. Attacker has access to the flaw, and
3. Attacker has capability to exploit the flaw
    - Examples
        - Lack of security patches
        - Lack of current virus definitions
        - Software Bug
        - Lax physical security

*e.g. Buffer Overflow is still on SANS Top 25 (#3). Industry has known and discussed since 1988!

## Code vs Design and Architecture

- Code review is a necessary but not sufficient practice for achieving secure software.
- The best a code review can uncover is around 50% of the security problems.
- Architectural problems are very difficult (and mostly impossible) to find by staring at code.

## The Software Security Problem

- To build a strong system, you have to understand how the system is likely to fail [Petroski, 1985].
- Mistakes are inevitable, but you have a measure of control over your mistakes.
- Common pitfalls sound like a good way to avoid falling prey to them
  - "i before e except after c," yet believe is still misspelled.
- Understanding vs. Applying
  - Spell checkers help reinforce the spelling rules.
  - Static analysis helps programmers find common security errors in source code

## The Software Security Problem (cont’d)

- The term static analysis - any process for assessing code without executing it.
- Static analysis is powerful because it allows for the quick consideration of many possibilities.
- A static analysis tool can explore a large number of “what if” scenarios without all the computations necessary to execute the code.
- Static analysis particularly well suited to security because:
  - many security problems occur in corner cases, or
  - hard-to-reach states that can be difficult to exercise by actually running the code.
- Good static analysis tools provide a fast way to get a consistent and detailed evaluation of a body of code.

## Defensive Programming Is Not Enough

- Historically it has referred only to the practice of coding with the mindset that errors are inevitable
  - "Writing the program so it can cope with small disasters" [Kernighan and Plauger, 1981]
- Good defensive programming requires adding code to check one's assumptions.
- Defensive programming:
  - is a good first step: <u>LEADS to Quality Code!</u>
  - focuses on reducing inadvertent errors/mistakes
  - does not guarantee secure software
- Secure programming creates code that behaves correctly even in the presence of malicious behavior.

# Defensive Programming Example

## Non-Defensive (Typical Buffer Overflow):

```
void printMsg(FILE* file, char* msg) {
  fprintf(file, msg);
}
```

## Defensive (Checks for non-threat based mistakes):

```
void printMsg(FILE* file, char* msg) {
  if (file == NULL) {
    logError("attempt to print message to null file");
  } else if (msg == NULL) {
    logError("attempt to print null message");
  } else {
    fprintf(file, msg);
  }
}
```

## But what if the string is not empty

- User input for msg variable:

```
AAA1_%08x.%08x.%08x.%08x.%08x.%n
```

- This input is malicious.
- This is a form of a Format String Attack (a.k.a. “Buffer Overflow Attack”)
  - Hex codes can be executed to produce malicious results.
- This could lead to the attacker taking control of the program.

## Secure Programming

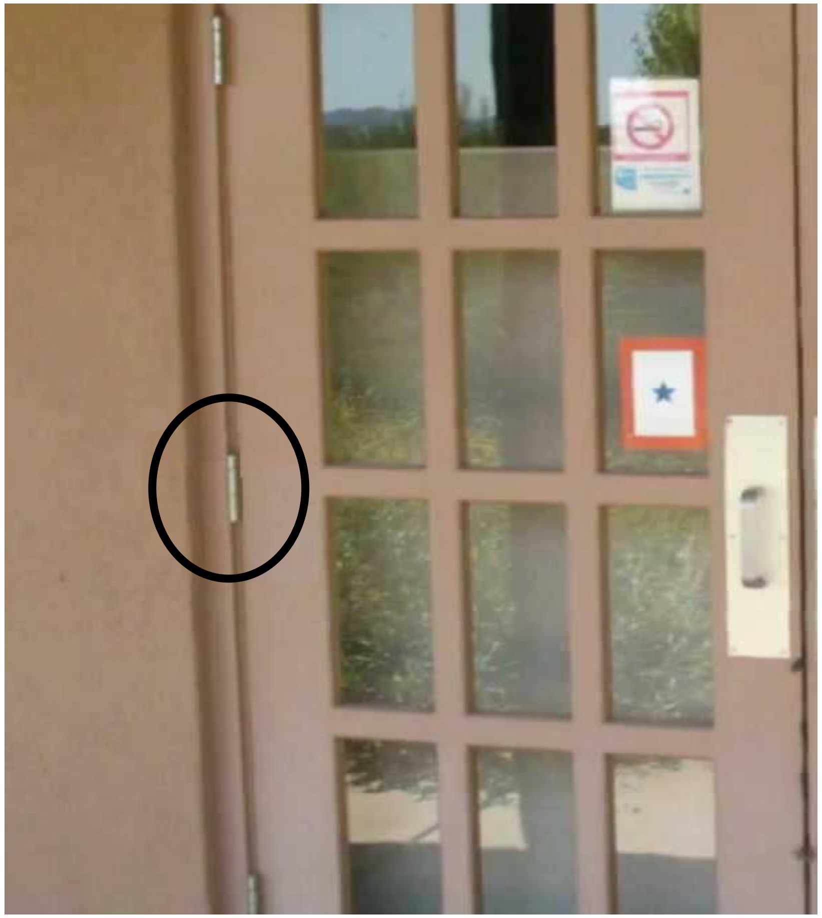
```
void printMsg(FILE* file, char* msg) {
  if (file == NULL) {
    logError("attempt to print message to null file");
  } else if (msg == NULL) {
    logError("attempt to print null message");
  } else {
    fprintf(file, "%.128s", msg);
  }
}
```

## Security Features != Secure Features

For a program to be secure, all portions of the program must be secure, not just the bits that explicitly address security. In many cases, security failings are not related to security features at all. A security feature can fail and jeopardize system security in plenty of ways, but there are usually many more ways in which defective nonsecurity features can go wrong and lead to a security problem. Security features are (usually) implemented with the idea that they must function correctly to maintain system security, but nonsecurity features often fail to receive this same consideration, even though they are often just as critical to the system's security.

Michael Howard, Microsoft Security Team

## What’s wrong with this?



**Hinges are a functional feature, incorrectly implemented they become a security vulnerability!**

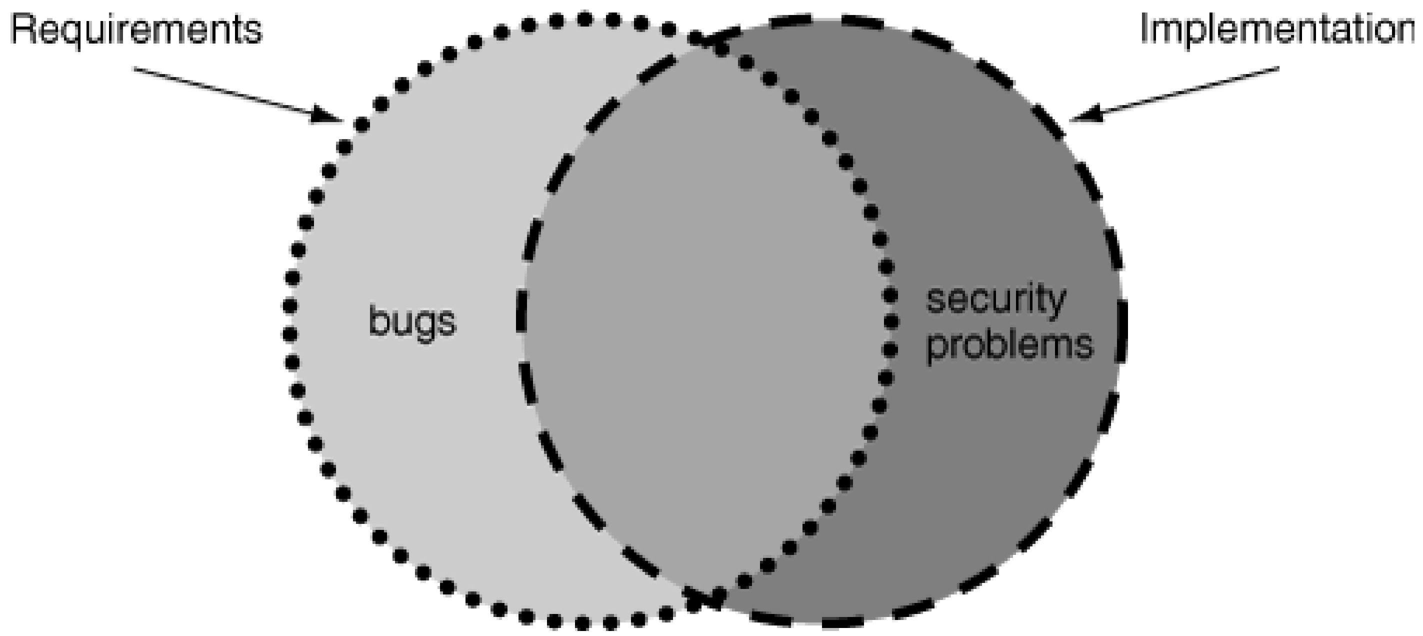
## The Quality Fallacy

- Software Quality - find the bugs that will affect the most users in the worst ways.
  - Functionality testing ensures that software meets typical users' expectations and satisfaction.
  - Software Testing compares the implementation to the requirements.
  - This approach is inadequate for finding security problems.
  - If the software fails to meet a particular requirement, you've found a bug.

**Reliable software does what it is supposed to do.**

## Unintended Functionality

Security problems are often not violations of the requirements



**Secure Software DOESN'T DO ANYTHING ELSE!**

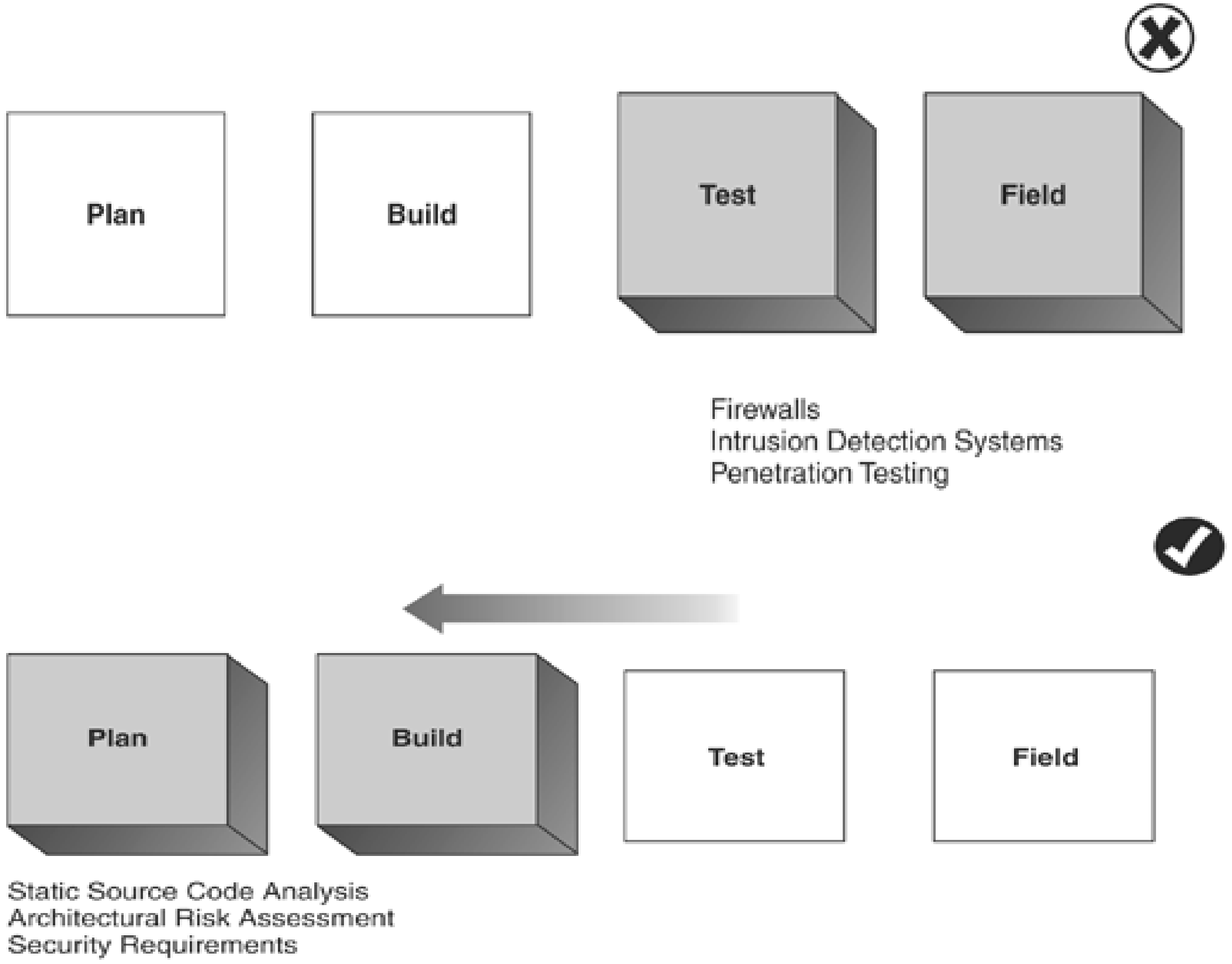
## Penetration Testing

- A form of Black Box Testing
- Presumes the attackers have no knowledge of the system
- Testing does not begin until system is built and released.
  - After the release, attackers and defenders are on equal footing
  - Attackers are now able to test and study the software, too.
  - The sum total of all attackers can easily have more hours to spend hunting for problems than the defenders have hours for testing.

## Static Analysis in the Big Picture

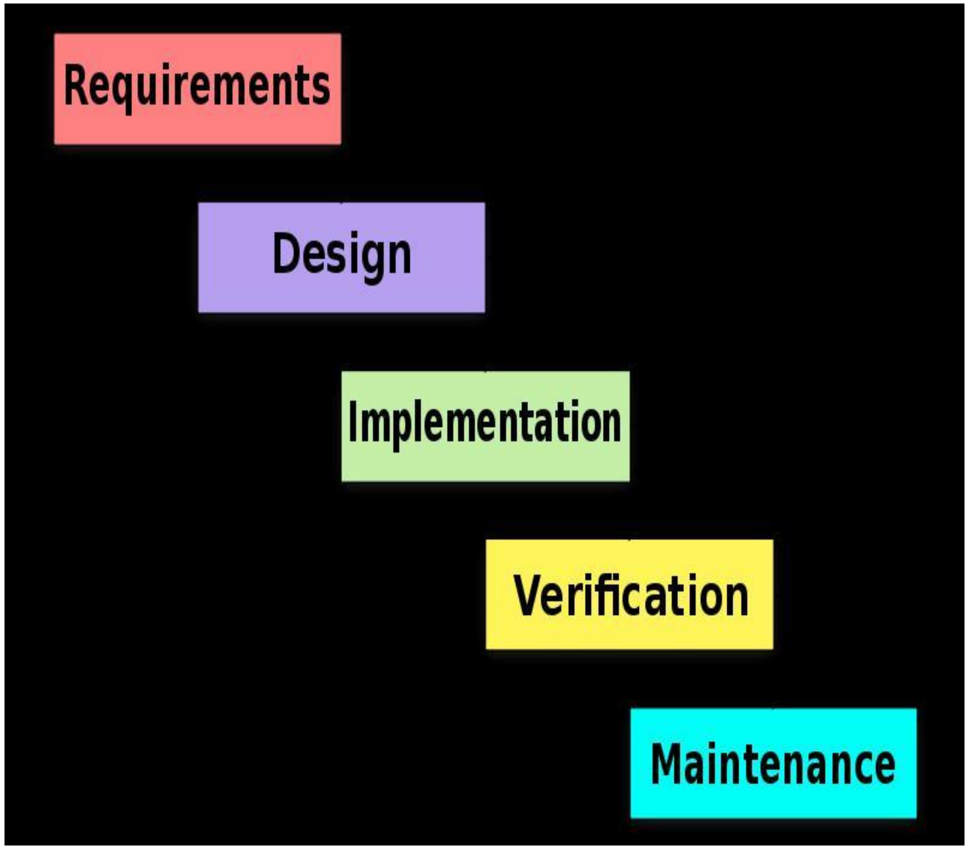
- All software methodologies use a form of this:
  1. **Plan** — Gather requirements, create a design, and plan testing.
  2. **Build** — Write the code and the tests.
  3. **Test** — Run tests, record results, and determine the quality of the code.
  4. **Field** — Deploy the software, monitor its performance, and maintain it as necessary

## The Correct Approach



## Comparison

SEI Software Engineering Waterfall



CISSP BoK Process

1. Project initiation
2. Functional design analysis and planning
3. System design specification
4. Software development
5. Installation/implementation
6. Operational/maintenance
7. Disposal

## System Development Process

- Developing software consists of several key areas
  - Role of security management during creation
  - Seven key life cycle phases
  - Appropriate change and control measures
    - Ensures integrity of the system during development

## System Development Process

- Life cycle phases broken down several stages
  - Different software development models integrate these phases in one way or another
    - Project initiation
    - Functional design analysis and planning
    - System design specification
    - Software development
    - Installation/implementation
    - Operational/maintenance
    - Disposal

## Project Initiation

- Stage 1
- Where the product or concept is born
- Conceptual definition is created
  - Identify features and benefits
- Includes risk management and risk analysis
  - Attempts to identify threats and weakness of the project or product

## Functional Design

- Stage 2
- Includes the following characteristics
  - Security checkpoints
  - Quality assurance of security controls
  - Configuration and change control process is identified
  - Formal functional baseline is created
  - Generation of design document

## System Design Phase

- Stage 3
- Creation of models to explain the design
  - Informational
    - Type of information and how it will be processed
  - Functional
    - Tasks and functions the application needs to carry out
  - Behavioral
    - State the application will be in during key states
- Access control mechanisms are chosen

## Software Development Phase

- Stage 4
- Where the programmers begin coding
- Development considerations
  - Buffer overflow issues
  - Debugging and code review
  - Formal and informal testing
  - Software hooks inserted need to be removed
- Security tests should be run

## Installation Phase

- Stage 5
- Product is put into production
- Configured for appropriate level of security
- Accreditation and certification may occur at this stage
- Auditing is normally implemented at this stage

## Maintenance Phase

- Stage 6
- Ongoing analysis of security and integrity is performed at this stage
  - Vulnerability tests
  - Monitoring
  - Auditing

## Disposal Phase

- Stage 7
- Product is discarded
- Data may need to be backed up, archived or securely removed

## Software Development Methods

- Waterfall
- Spiral Model
- Joint Application Development (JAD)
- Rapid Application Development (RAD)
- Cleanroom
- Agile

## System Development Process

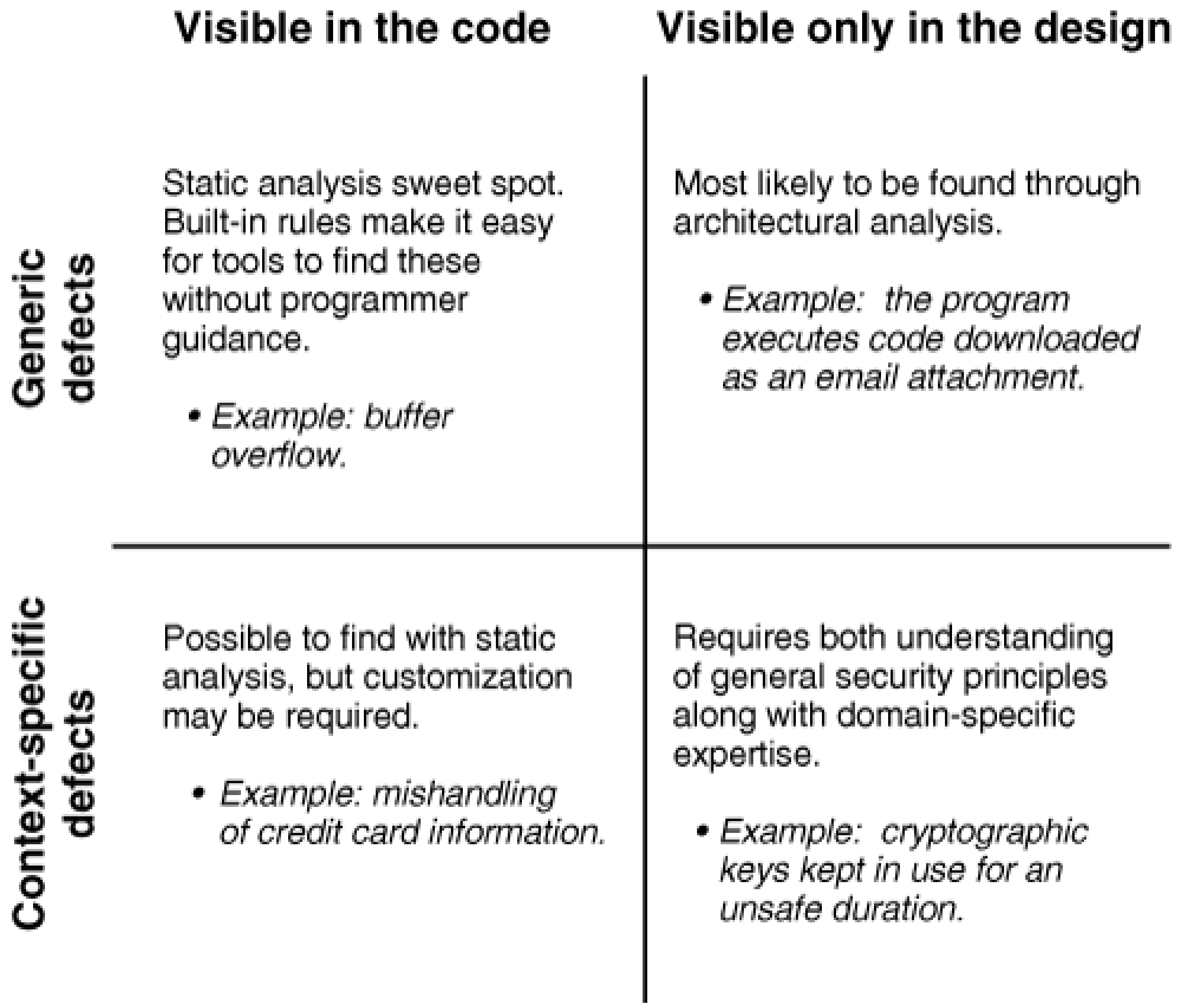
- Security needs to be included during development
  - A security plan should be drawn up at the beginning of the development process
    - Needs to be updated as the project grows and develops during the development cycle
    - Will most likely be used by the security officer/auditor to ensure compliance of the application

**Build Security In - not Bolted On!**

## Classifying Vulnerabilities

- Two loose groups: generic and context specific
  - A generic defect is a problem that can occur in almost any program written in the given language.
    - e.g. Buffer Overflow
  - Context-specific defects, on the other hand, requires a specific knowledge about the semantics of the program at hand.
    - Payment Card Industry (PCI) Data Protection Standard,

# Defect Categories



| | Visible in the code | Visible only in the design |
|---|---|---|
| Generic defects | Static analysis sweet spot. Built-in rules make it easy for tools to find these without programmer guidance.<br>• *Example: buffer overflow.* | Most likely to be found through architectural analysis.<br>• *Example: the program executes code downloaded as an email attachment.* |
| Context-specific defects | Possible to find with static analysis, but customization may be required.<br>• *Example: mishandling of credit card information.* | Requires both understanding of general security principles along with domain-specific expertise.<br>• *Example: cryptographic keys kept in use for an unsafe duration.* |

## The Seven Pernicious Kingdoms

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Error Handling
6. Code Quality
7. Encapsulation

* Environment

## Input Validation and Representation

- Input validation and representation problems are caused by
  - metacharacters,
  - alternate encodings, and
  - numeric representations.
- Security problems result from trusting input. The issues include
  - buffer overflow,
  - cross-site scripting,
  - SQL injection, …..

**Problems related to input validation and representation are the most prevalent and the most dangerous category of security defects in software today.**

## API Abuse

- An API is a contract between a caller and a callee.
- The most common forms of API abuse are caused by the caller failing to honor its end of this contract.
- Callee (program) must defend by considering as many ways as caller can "abuse" API.

**▪Example: Failing to call chdir() after calling chroot() violates the contract that specifies how to change the active root directory in a secure fashion.**

## Security Features

- Must get the security features right.
  - authentication,
  - access control,
  - confidentiality,
  - cryptography, and
  - privilege management.

**Hard-coding a database password in source code is an example of a security feature (authentication) gone wrong.**

## Time and State

- Programmers like to think of their code as being executed in an orderly, uninterrupted, and linear fashion.
- Multitasking operating systems running on multicore, multi-CPU, or distributed machines don't play by these rules.
- Defects caused by unexpected interactions between
  - threads,
  - processes,
  - time, and
  - data.
- These interactions happen through shared state:
  - semaphores, variables, the file system, and anything that can store information

## Error Handling

- Two ways to introduce an error-related security vulnerability.
  - The first (and most common) is to handle errors poorly or not at all.
  - The second is to produce errors that either reveal too much or are difficult to handle safely.

## Code Quality

- Poor code quality leads to unpredictable behavior.
  - User’s perspective: manifests as poor usability.
  - Attacker’s perspective: provides an opportunity to stress the system in unexpected ways.
- Can lead to denial of service (DOS) attack which affects availability of the system.

## Encapsulation

- Encapsulation is about drawing strong boundaries
  - Differentiation between validated data and unvalidated data
  - Between one user's data and another's
  - Between data that users are allowed to see and data that they are not allowed to see

## *Environment

- Includes everything outside the source code but is still critical to the security of the product being created.
    - The configuration files that govern the program’s behavior
    - The compiler flags used to build the program

## Conclusion

- Questions?