Software Security

Sven Schäge

About the Lecturer…

Logistics

- Lectures
  - New content
  - In-class discussion
- Quizzes
  - Accompany lectures
- Challenges
  - Give bonus points of up to 10 % of (exam) grade
- Reading material
  - In-depth knowledge
  - New perspective
  - Scientific material
  - Preparation for next week
- References to additional, interactive toy challenges
- To pass the course, a student needs to score 5.5 or higher for the written exam **excluding** bonus points.
- Final Grade= Grade of Written Exam + Bonus Points

Content

- Understand various ways that (bad/buggy) software can pose a threat to security
- Understand strategies and techniques to improve software security

Concrete Topics

- Introduction to Software, (In)Secure Software, and the 7+1 Kingdoms
- Introduction to Software Development
- Memory Corruption Attacks
- Static Code Analysis
- Dynamic Code Analysis/Fuzzing
- Safe Languages
- Introduction to Web Technology
- Web Security
- Side-Channel Attacks and Encapsulation
- Race Condition Vulnerabilities

How to Learn ‘Secure Software Development‘?

- Clearly understand root causes of problems
  - Raises awareness in other contexts
- Know common ideas of attacks (abstractly)
- Know common ideas of security countermeasures (abstractly)
- Clearly understand limitations of countermeasures (abstractly)
  - What can and do software security tools bring to the table?
  - What do they cost?
- Goal:
  - Obtain information that allows an educated decision on how to strategically approach the software development process
  - Gain awareness of problems that need to be taken care of using additional measures
  - Gain awareness of the remaining risks

Limitation of this Lecture Series

- We only get to cover a selection of
  - Concrete (historical) attacks
  - Counter-measures
  - Software security tools
- We prioritize wide-spread security problems that are characteristic of Secure Software Development
- We focus on concepts

Prerequisite

- Formal prerequisite
  - None
- Helpful background knowledge
  - Programming Language experience: C, C++, Java, Assembler, Rust
  - Knowledge on Computer Architectures
  - Cryptography

A Note on Pre-Existing Background Knowledge

- Experience has shown that the pre-existing background knowledge among the students is **very** diverse, ranging from little experience to in-depth background knowledge. Accordingly the expectations vary.
- The course needs to provide a strong basis for all students.
- Advanced students are particularly encouraged to:
  - Read the additional material (especially the scientific papers and references therein).
  - Take part in the more abstract discussion e.g. on the right approach to building programming languages.
  - Thoroughly do the challenges.
  - Try out concrete tools to put their knowledge into practice.

Literature

- Books
  - The Cyber Security Body of Knowledge (https://www.cybok.org/media/downloads/CyBOK_v1.1.0.pdf)
  - Gary McGraw: Software Security: Building Security In, 2006
  - Robert C. Seacord: Secure Coding In C and C++, 2013
  - Michael Howard, David LeBlanc, John Viega: 24 Deadly Sins of Software Security, 2010

- Selected (scientific) papers to be read for class

Introduction

Sven Schäge

What Does ‘Software Security’ Deal With?

- Software, and the role it plays for security!

Why Should We Study Software?

Software Development

Execution of Software

Users interact with Software

- The use of software is very widespread.
  - Key ingredient of modern information systems
  - Very cost-efficient way to design systems as computing platforms can be re-used for different purposes
  - Allows for more standardized computing platforms which in turn are cheaper

Why Should We Consider Software Security Specifically?

- Many software solutions are used very widespread.
  - Attractive for attackers as they might have higher pay-offs from attacking software that is used by many people.
- Software usually has to work with inputs from other (remote) parties.
  - This guarantees a potential for access by attacker.
- Software has a high level of control of machines of other parties.
  - Software tells machines of other parties what to do - on a very specific level and via thousands or even millions of precise commands.
  - This is attractive for attackers as the potential pay-off is big – controlling the machine and data of others.

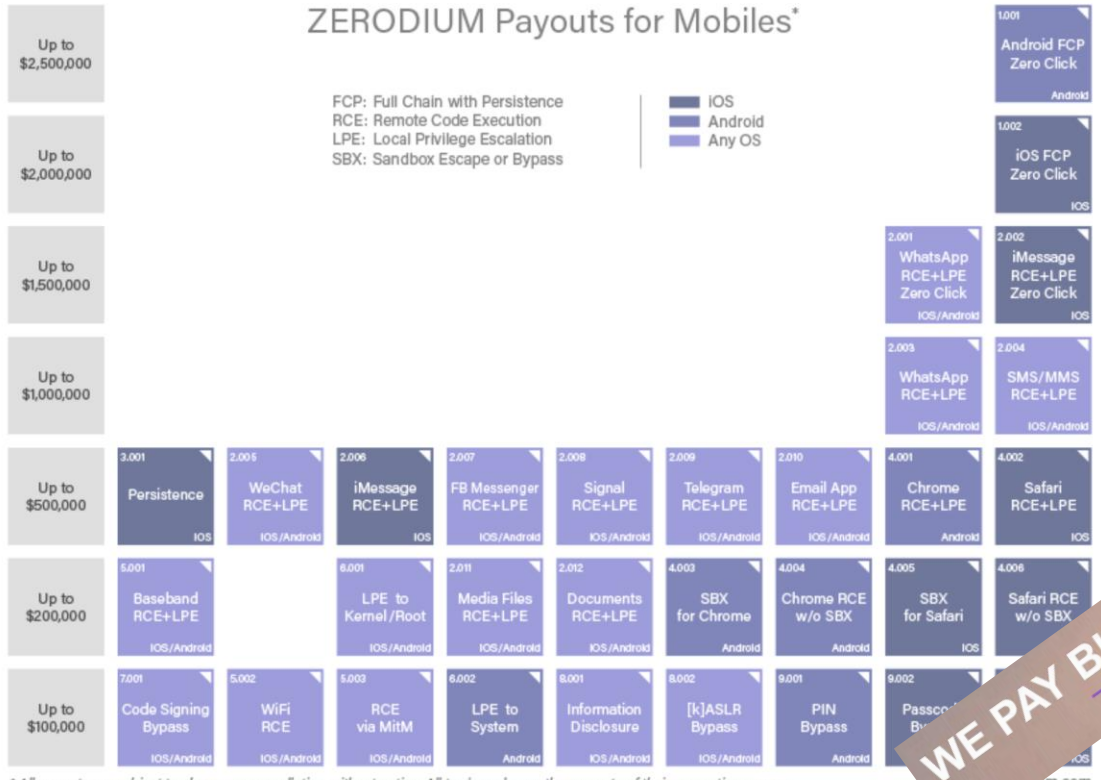
Why Does Software Seem Particularly Vulnerable to Attacks?

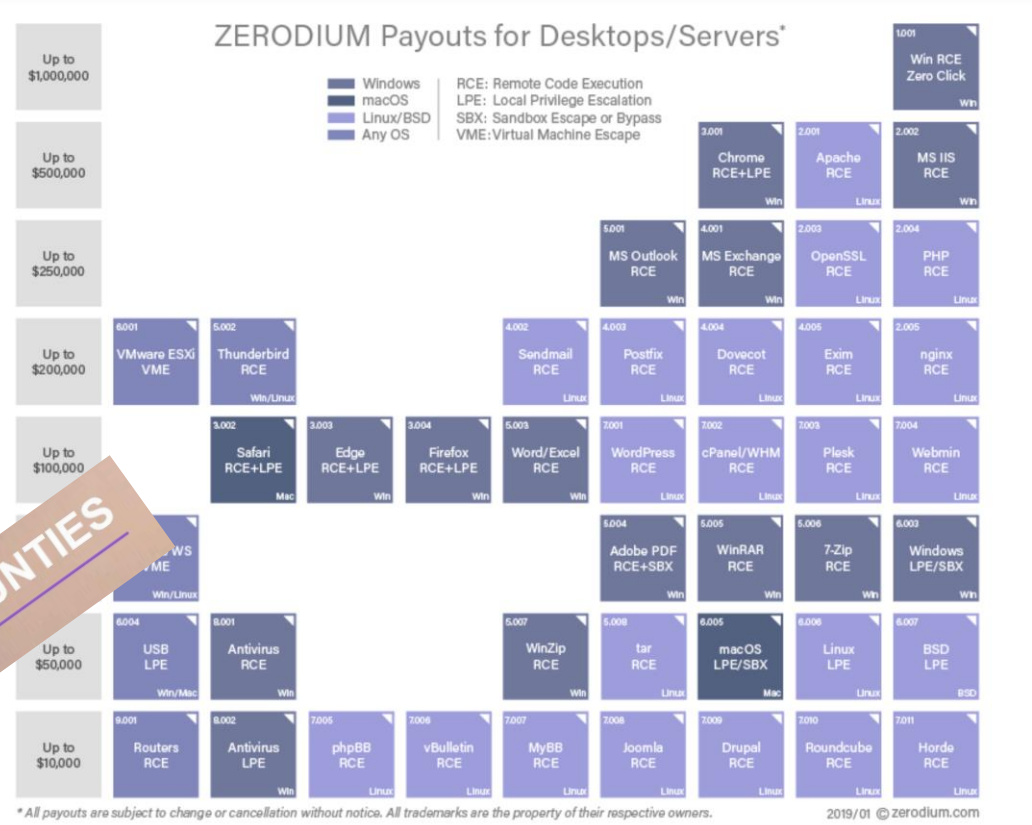
- Building software is a process that involves noting thoughts in some formal language. Often these thoughts are purported solutions to problems.
  -> **Developing software is error-prone** - as problem-solving is.
  - Larger software projects make occurrence of software errors (bugs) more likely
  - When several people are involved in the software building process, **silent expectations** on code of others might deviate from reality
- Software is complex
  - Software is so complex that in general we can assume it to have bugs. Attackers ‘simply’ need to find them.
- Core problem: many software bugs do transfer to **security vulnerabilities**.
- Often security vulnerabilities are exploited in **attacks**

What Makes the Situation Worse as Compared to Other Industries?

- Project management does traditionally not provide incentives for writing extremely secure code - neither do quality management policies.
  - This is because software can easily be updated (patched) to fix problems later **on demand**.
- Attacks can be automated and launched from remote locations
  - In general, lower risk of being caught
- Attackers can find and exploit vulnerabilities without being noticed
- Patches can introduce new vulnerabilities or other problems
- Patches often go unapplied by customers
- Various financial incentives to attack software

Incentives to Attack Software





Motivation for Secure Software: The Cost of Cybercrime

- 2021 estimations show up to USD 10.5 trillion for year 2025 for cybercrime costs globally
- Typically costs include
  - Damage and destruction of data
  - Stolen money
  - Lost productivity
  - Theft of intellectual property
  - Theft of personal and financial data
  - Embezzlement
  - Fraud
  - Post-attack disruption to the normal course of business
  - Forensic investigation
  - Restoration and deletion of hacked data and systems
  - Reputational harm
- Hidden Costs for Attacked Companies
  - Insurance premium increases
  - Increased cost to raise debt
  - Lost value of customer relationships
  - Value of lost contract revenue
  - Regulatory and compliance fines
  - Legal and public relations fees
  - Notification, identity theft repair and credit monitoring for affected parties

CNBC report: https://youtu.be/P6x4GhjDVHY

Why is the Problem Still So Acute?

A Fundamental Limitation: The Halting Problem

- The halting problem roughly states that it is impossible to construct a program TESTHALT that **perfectly** decide whether an arbitrary given program P runs infinitely on some input I - or not.
- Put differently, there is no general machine TESTHALT that given arbitrary input P,I decides whether P’ terminates/halts on I.

Proof Sketch of the Halting Problem

- Assume there exists TESTHALT that on input arbitrary P,I decides whether P,I terminates.
- Construct a new program Z that computes on input I the following:
- Z:

  Line 1: If TESTHALT(I)==true loop forever
- Now consider what happens if we input to Z its own description, calling Z(Z)
- If such an TESTHALT exists, then this would always lead to a contradiction:
  - If the condition TESTHALT(Z)==true in Line 1 holds, then the entire program runs forever because of the loop forever statement which implies that TESTHALT(Z)=false - a contradiction.
  - If the condition TESTHALT(Z)==true in Line 1 does not hold, then the program terminates immediately after the If-statement – again a contradiction.
- The proof shows thus shows that such an TESTHALT cannot exist and works by contradiction using a self-reference in the sense of „This sentence is incorrect.“

Rice‘s Theorem

- Rice’s theorem shows us the limitations of automatic testing.
- Statement of Theorem: Any functional property X of a program cannot be perfectly decidable!
- Functional property X of a program
  - Describes input/output behavior (how the output to some program relates to its input). This is called a semantic property. It is a property of the language computed by the computer/Turing machine not a property of the computer/Turing machine itself.
  - There are programs that have property X while others do not have it (non-triviality).
- Put differently: Rice‘s Theorem is a fundamental limitation for all question where we want to know, whether a given program ultimately has functional property X – or not.
- As a consequence, all automatic solutions in practice must be approximate (meaning they have false positives or false negatives), heuristic or only applicable to a part of the problem.
- No ideal testing!

Proof Sketch of Rice‘s Theorem

- Proof idea (by contradiction): Assume there is general program TESTX that can decide whether an arbitrary input program Q on input J has a functional property X, then we can decide the halting problem. However, the halting problem is undecidable. So TESTX cannot exists.
- Proof sketch:
  Assume we want to solve the halting problem and decide whether program P on input I terminates. Assume there is general program TESTX that can decide whether a given arbitrary program Q on input J has a functional property X.
- Let us now describe how we can decide whether P terminates on input I by building program TESTHALT that does the following when given P, I and access to TESTX:
  - Construct program T that on input J has the following steps:
    - Run P on input I.
    - Clear all memory (restart machine).
    - Run Z on J, where Z is an arbitrary program that actually has property X. Use the output of Z as the output of T.
  - Feed T,J to TESTX and output the result as the final result of TESTHALT.
- Now, if TESTX can decide on input T,J if T has property X on input J, we implicitly solve the halting problem:
  - If TESTX outputs NO, we must have that Z has never been reached. This happens if P on input I does not terminate.
  - If TESTX outputs YES in the execution of T we will have that Z has been run on J. However, this means that P has terminated on I.

Rice‘s Theorem in Practice

- Which of the following programs P are decidable?
- P is defined as follows:
  - Always returns 1.
  - Needs poly-time.
  - Is a virus.
  - Outputs the secret password for some specific input.
  - Will never try to access certain files.
  - Will never try to escalate privileges.
  - Returns a memory unit.
  - Corrupts files.
  - Deletes files.
  - Installs without asking.
  - Disrupts other programs.
  - Needs at least X many states.

Limitations of Rice‘s Theorem

- Rice‘s theorem does only rule out an ideal automatic test algorithm that works for arbitrary given input programs.
  - For a specific program we might have enough knowledge about it to answer some of these questions.
  - This situation is similar to our knowledge about the termination of programs. Although in general the question is undecidable, we should ideally write programs where we are actually sure that they do terminate!
    - Otherwise consider writing clearer code.
- Rice’s theorem:
  There is no fully automated software testing without trade-off!

Decidability and Application to Finding Software Errors

	complete	incomplete
sound	reports all errors reports no false alarms **undecidable**	reports all errors may report false alarms **decidable**
unsound	may not report all errors reports no false alarms **decidable**	may not report all errors may report false alarms **decidable**

Practical Reasons

- Lack of awareness
- Lack of knowledge
- Lazyness
- Too little incentives to build software more securely
- Priority of functionality over security

Complexity:
Many Weaknesses are Known

☹

https://cwe.mitre.org/data/definitions/699.html

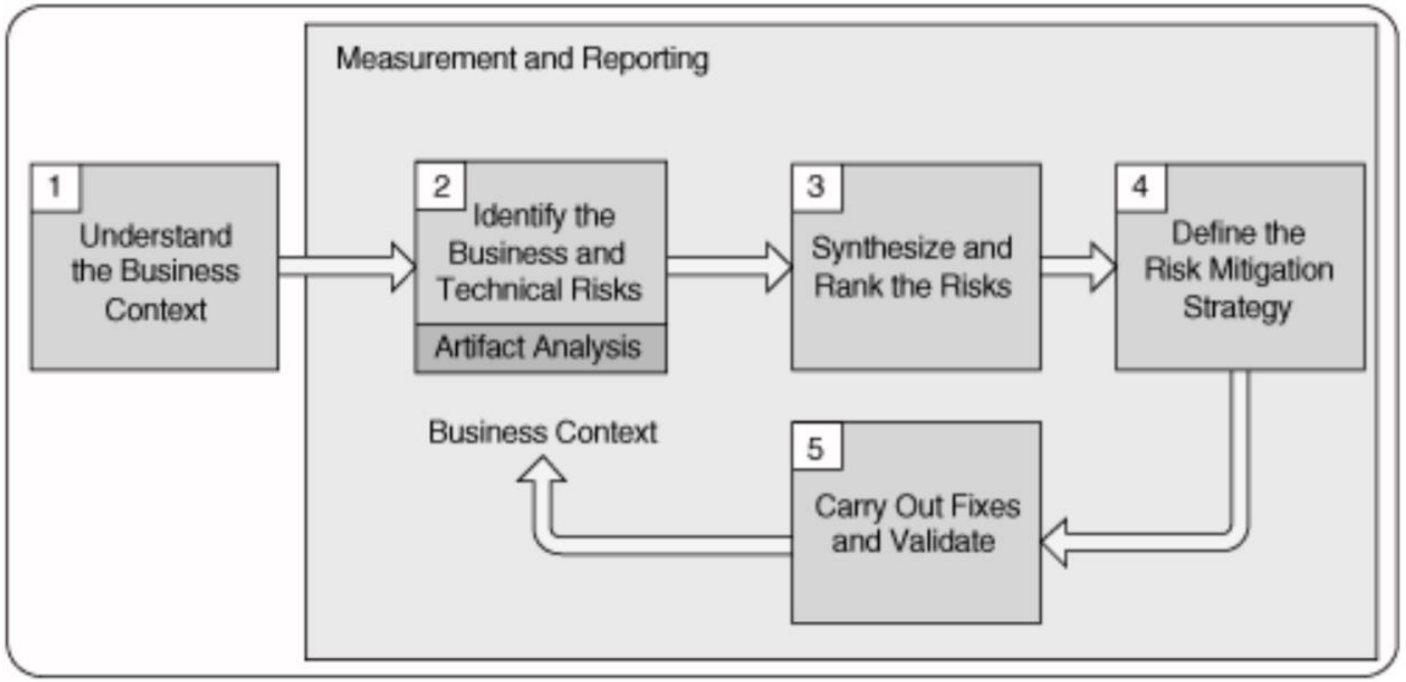
Security In The Software Life Cycle

Sven Schäge

Three Pillars of Software Security

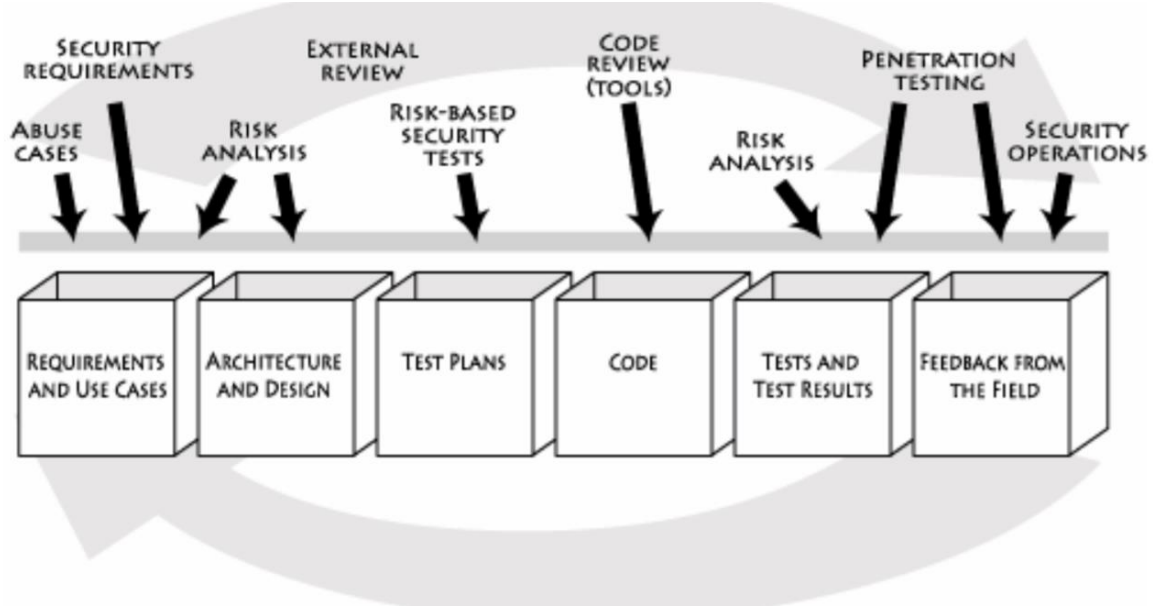
Gary McGraw: Software Security 2006

Risk Management: 5 Stages



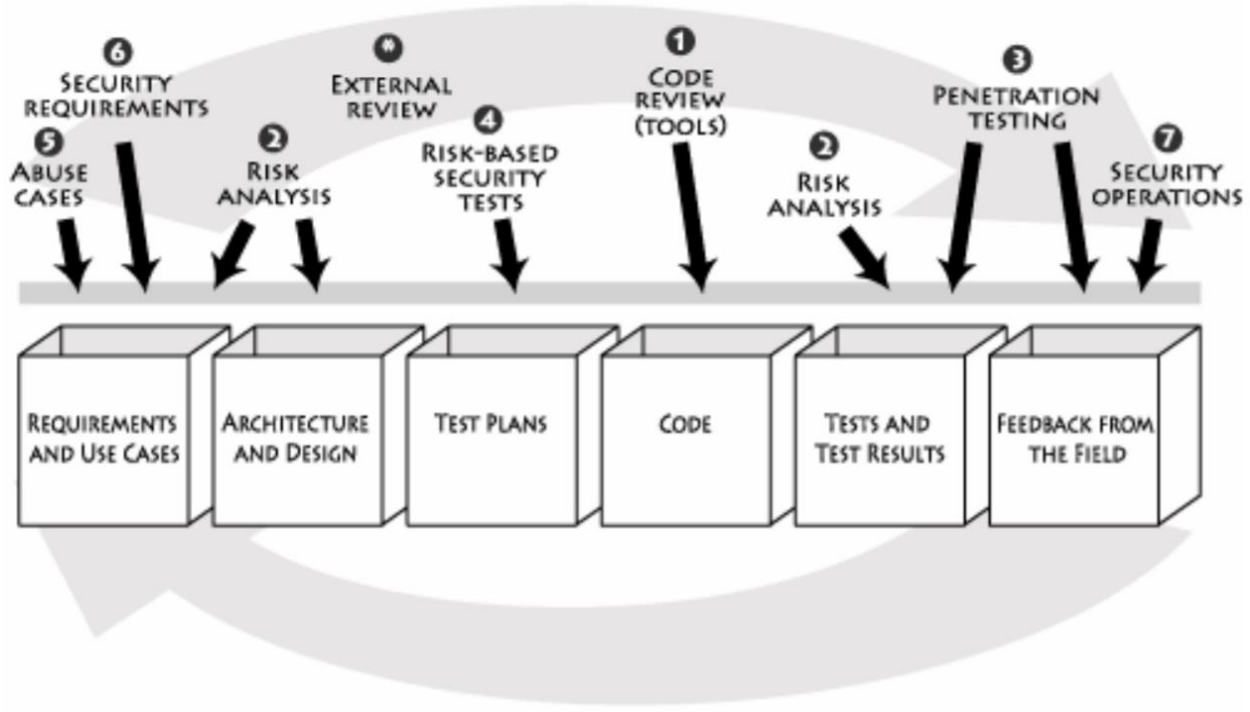
Gary McGraw: Software Security 2006

Touchpoints: Security Best Practices Meet the Software Development Process



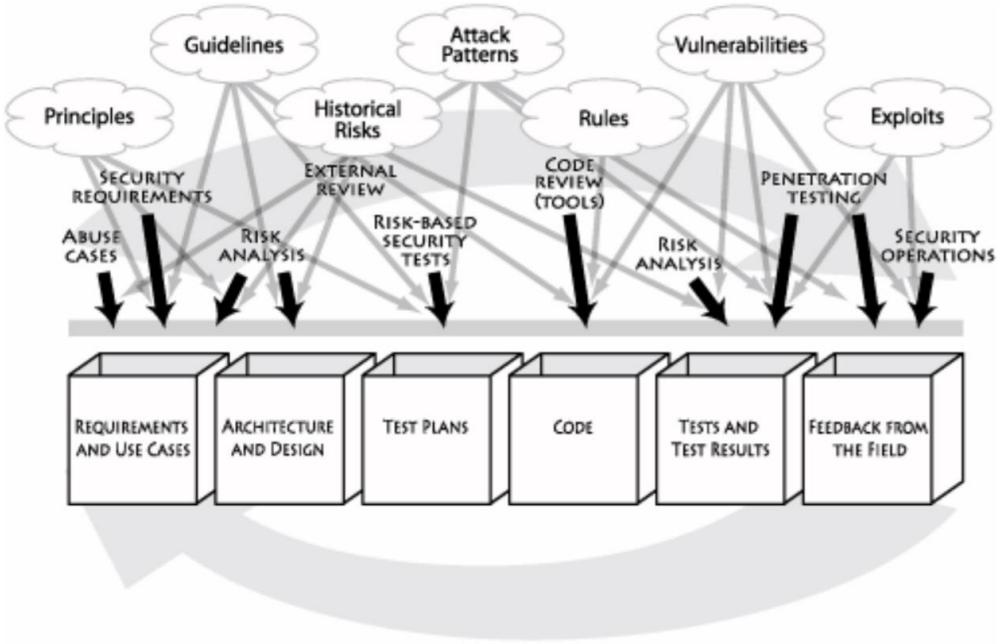
Gary McGraw: Software Security 2006

The Order of Effectiveness



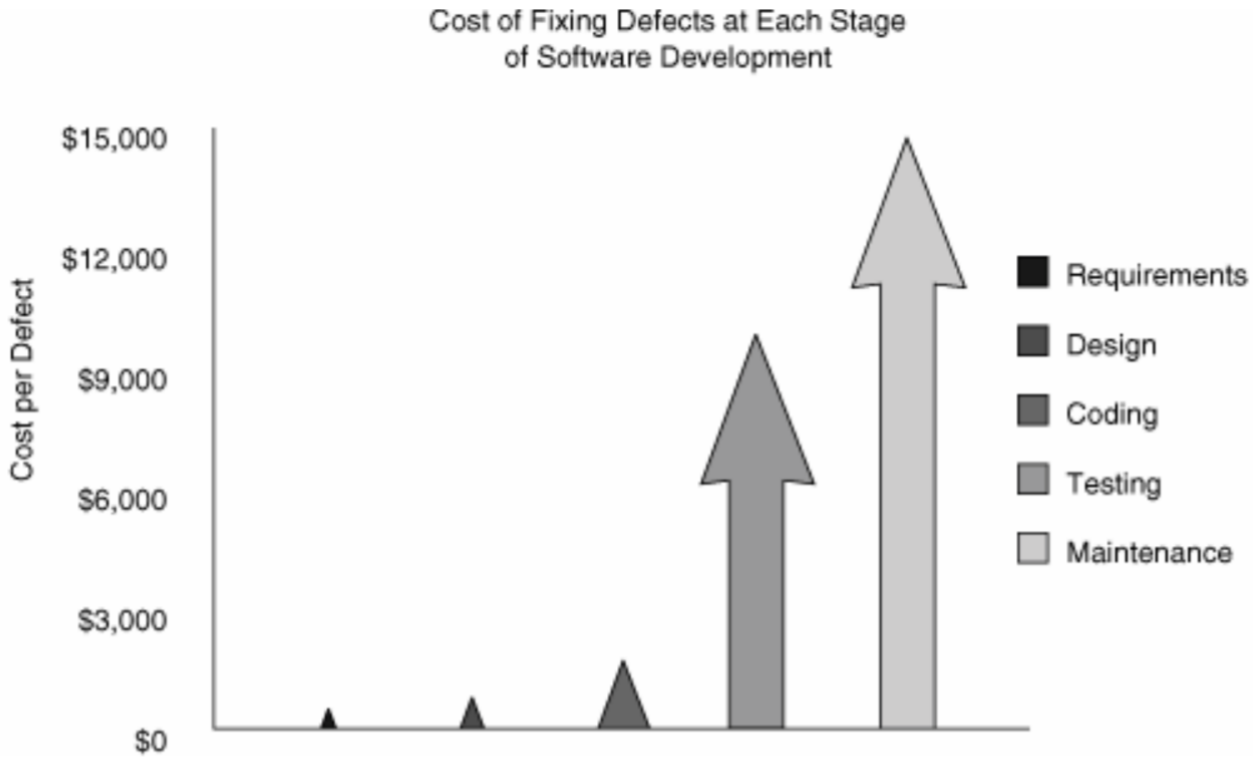
Gary McGraw: Software Security 2006

Adding Knowledge and Experience



Gary McGraw: Software Security 2006

Moving Security Fixes to Earlier Stages



Gary McGraw: Software Security 2006

Indispensable Ingredients of Security Analyses

Abuse Cases => Attacker Model

- Attacker‘s Motivation
- Attacker‘s Capabilities
- Security Assumptions
  - Which building blocks are supposed to be trusted?

Security Requirements

- What does security mean for a system?
- Use more precise language and avoid overloaded, too simplistic ‘secure’
- A corresponding vocabulary is also widespread in cryptography
  - Confidentiality/ Secrecy
  - Authentication
  - Integrity
  - Privacy
  - Anonymity
  - Availability
  - Non-Repudiation
  - Deniability

7+1 Kingdoms

Sven Schäge

The 7+1 Kingdoms Taxonomy

- Categorizes weaknesses
- Easily memorable
- Allows to search for weaknesses relatively comprehensively
- Helps to organize high number of known weaknesses
- Allows to speak about weaknesses more abstractly
- Describes weaknesses in a technology-independent way
- Makes weaknesses **and** countermeasures transferable to other technologies

1. Input Validation and Representation Problems -Maliciously Crafted Inputs

- Problems caused by
  - Meta characters
  - Alternate encodings
  - Numeric representations
  - No input validation
- Examples:
  - Buffer overflows
  - Cross-site scripting attacks
  - SQL injection
- White-listing approach for input validation strongly recommended

2. API Abuse
-Implicit Assumptions about Communication Partners to Behave as Agreed

- APIs (application programming interfaces) are formalized agreements that define how one software may offer services to another.
  - Microservices
  - APP communication with servers
- API-based communication features the exchange of highly structured data
  - Can reveal valuable information, e.g. on data model (organization of stored data), private user data
- API abuse in general means that one communication partner (the attacker!) does not behave as agreed
  - Not delivering the service as agreed
  - Not using the services as agreed
- Example:
  - Identify expensive services and launch a distributed denial of service attack (on the application level)
  - Spoofing GPS data to order a pizza to a wrong location
- Can partly be dealt with using cryptography

3. Problems when Using Security Features/Tools -Correct Combination of Correct Security Mechanisms

- Combining distinct security tools/features to manage access control, authentication, privilege management, cryptography is error-prone.

- Example:
  - Using bad or outdated cryptography.
  - Requiring too weak authentication techniques.

4. Parallelism and Consistency Problems - Time and State

- Software that uses distributed computations has to explicitly consider measures to ensure consistent states and solve time-dependent (order-dependent) race-conditions.

- Distributed software is often programmed with a linear execution model in mind.
- In reality software is executed quite differently
  - To allow for the parallel execution of tasks processors switch work on task regularly.
  - Threads are supposed to represent independent executions.
  - When two supposedly independent threads access the same resource,
- Examples:
  - Keeping databases consistent
- Mechanisms for appropriate concurrency control should be implemented
  - Semaphores
  - Transactions in dabase mangement systems

5. Error Handling/Output Problems -Error-Handling is Error-Prone

- Forgetting to handle errors
- Outputting overly detailed error messages
- Outputting overly informative outputs

- Often implemented via ‘dangerous’ goto structures in program code
- By sending incorrect input, the attacker can test new parts of the code (error-handling routines) for vulnerabilities
- Correct error handling often logically complex
- May output critical information in a very subtle way (side-channels)

- Example:
  - Error message discloses if first half of password is correct
  - Error type may give attacker valuable information

6. Code Quality Problems

- Leads to unpredictable behaviour
- Poor code quality is an opportunity for an attacker to stress the system
- Mistakes are harder to spot

7. Encapsulation/Isolation Problems

- Setting-up appropppriate boundaries and barriers
- Encapsulation should follow a carefully crafted trust model

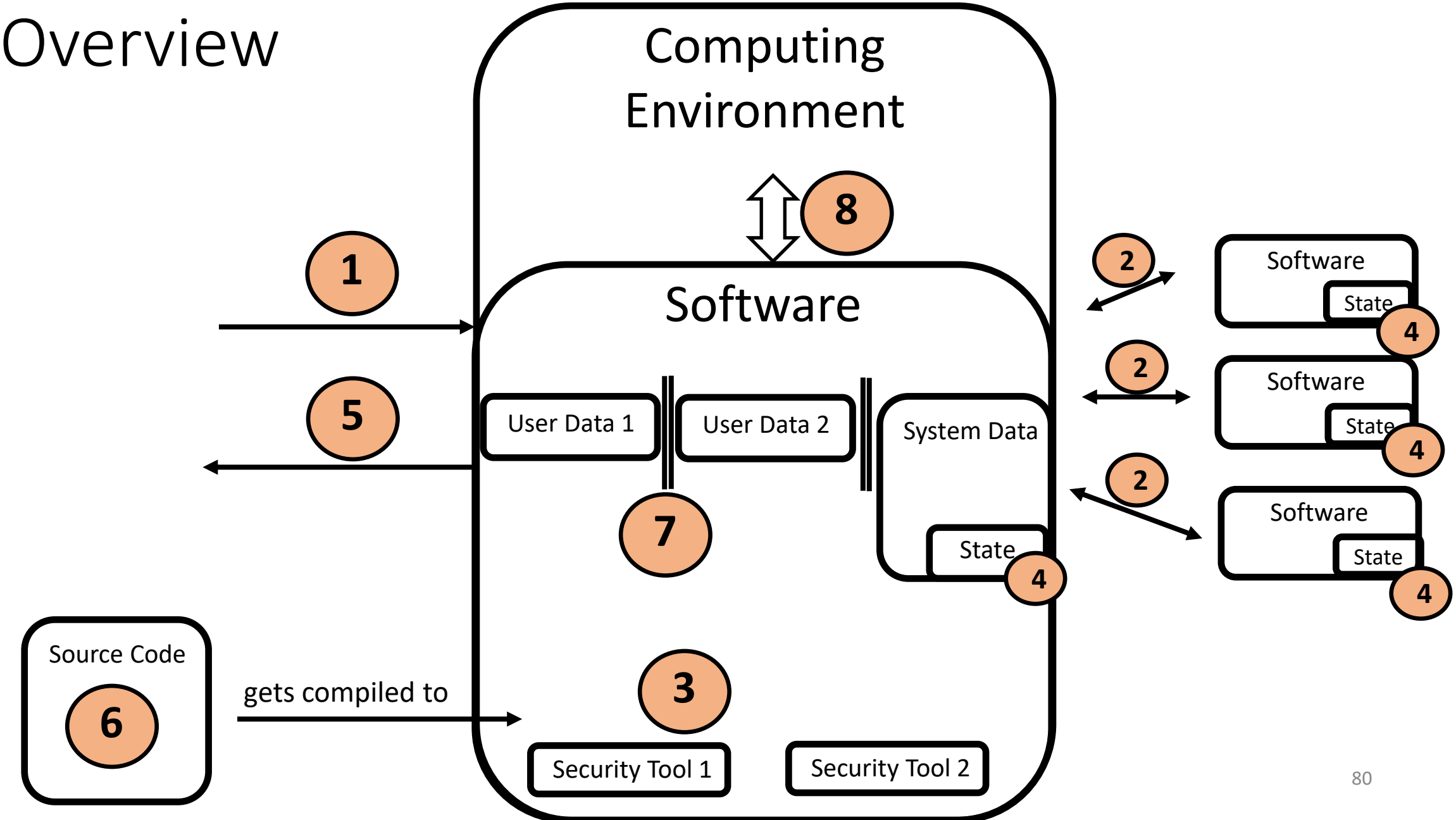
- Example:
  - No separation of authenticated data vs. unauthenticated data
  - Input of User 1 should not have access to input of User 2
  - Output reveals parts of a secret

8. Dependence on Environment

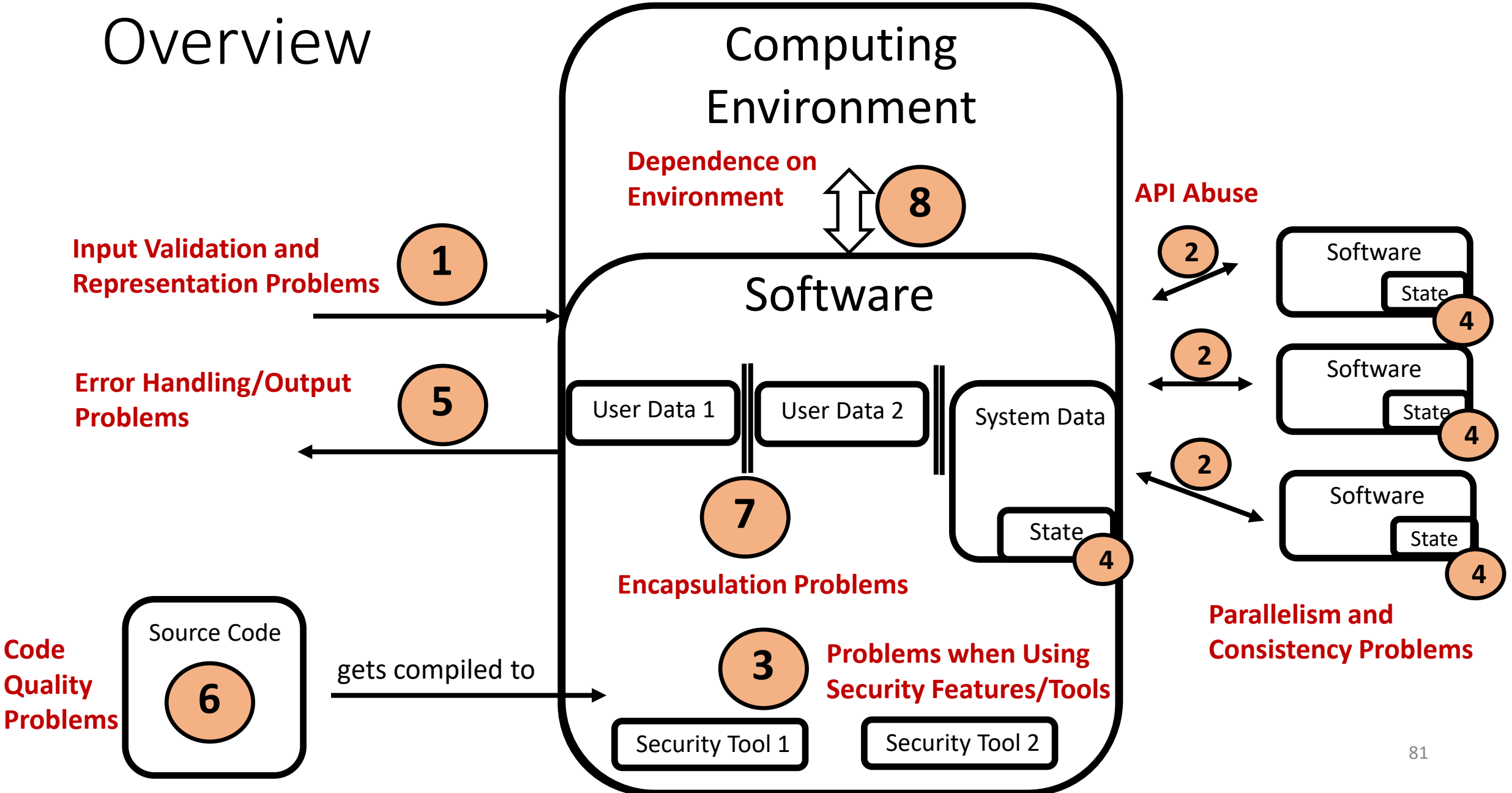
- Requirements on the computing environment that are critical for the security of a software

- Example:
  - DNS
  - TLS Tunnels
  - Good Randomness

Overview


Computing Environment
8
1
Software
2
Software
State
4
2
Software
State
4
5
User Data 1
User Data 2
System Data
7
State
4
2
Software
State
4
Source Code
6
gets compiled to
3
Security Tool 1
Security Tool 2

Overview


Computing Environment
Dependence on Environment
8
Input Validation and Representation Problems
1
Software
Error Handling/Output Problems
5
User Data 1
User Data 2
System Data
State
4
7
Encapsulation Problems
API Abuse
2
Software
State
4
Parallelism and Consistency Problems
Source Code
6
Code Quality Problems
gets compiled to
3
Problems when Using Security Features/Tools
Security Tool 1
Security Tool 2

Organizing Existing Weaknesses According to 7+1 Kingdoms

CWE VIEW: Seven Pernicious Kingdoms

View ID: 700
Type: Graph

Downloads: Booklet | CSV | XML

Objective

This view (graph) organizes weaknesses using a hierarchical structure that is similar to that used by Seven Pernicious Kingdoms.

Audience

Stakeholder	Description
Software Developers	This view is useful for developers because it is organized around concepts with which developers are familiar, and it focuses on weaknesses that can be detected using source code analysis tools.

Relationships

The following graph shows the tree-like relationships between weaknesses that exist at different levels of abstraction. At the highest level, categories and pillars exist to group weaknesses. Categories (which are not technically weaknesses) are special CWE entries used to group weaknesses that share a common characteristic. Pillars are weaknesses that are described in the most abstract fashion. Below these top-level entries are weaknesses are varying levels of abstraction. Classes are still very abstract, typically independent of any specific language or technology. Base level weaknesses are used to present a more specific type of weakness. A variant is a weakness that is described at a very low level of detail, typically limited to a specific language or technology. A chain is a set of weaknesses that must be reachable consecutively in order to produce an exploitable vulnerability. While a composite is a set of weaknesses that must all be present simultaneously in order to produce an exploitable vulnerability.

Show Details: ☐

Expand All | Collapse All

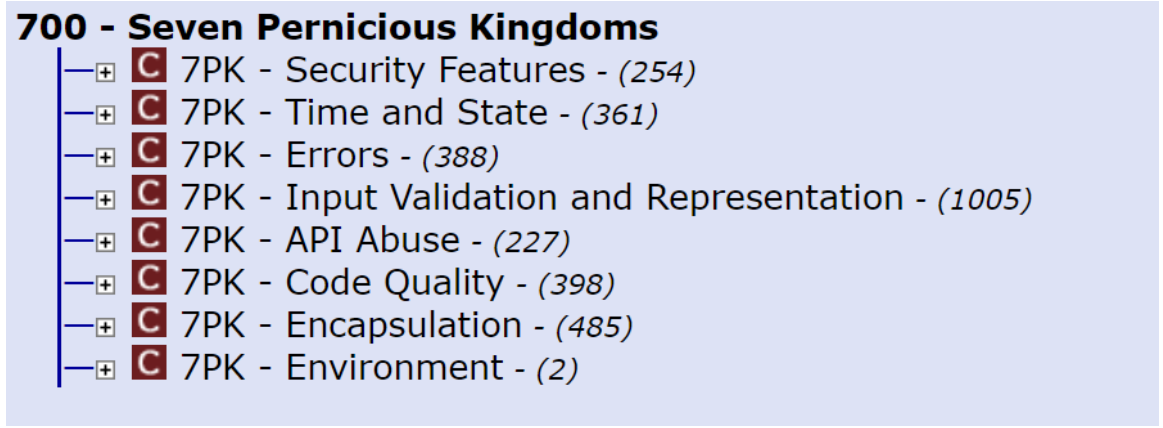
700 - Seven Pernicious Kingdoms
- C 7PK - Security Features - *(254)*
- C 7PK - Time and State - *(361)*
- C 7PK - Errors - *(388)*
- C 7PK - Input Validation and Representation - *(1005)*
- C 7PK - API Abuse - *(227)*
- C 7PK - Code Quality - *(398)*
- C 7PK - Encapsulation - *(485)*
- C 7PK - Environment - *(2)*

BACK TO TOP

Notes

Other

The MITRE CWE team frequently uses "7PK" as an abbreviation for Seven Pernicious Kingdoms.



https://cwe.mitre.org/data/definitions/700.html

Prioritizing Widespread Problems

The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html