Software Security

Defense Strategies in software design

Dr. Zeyad Safaa Younus Saffawi

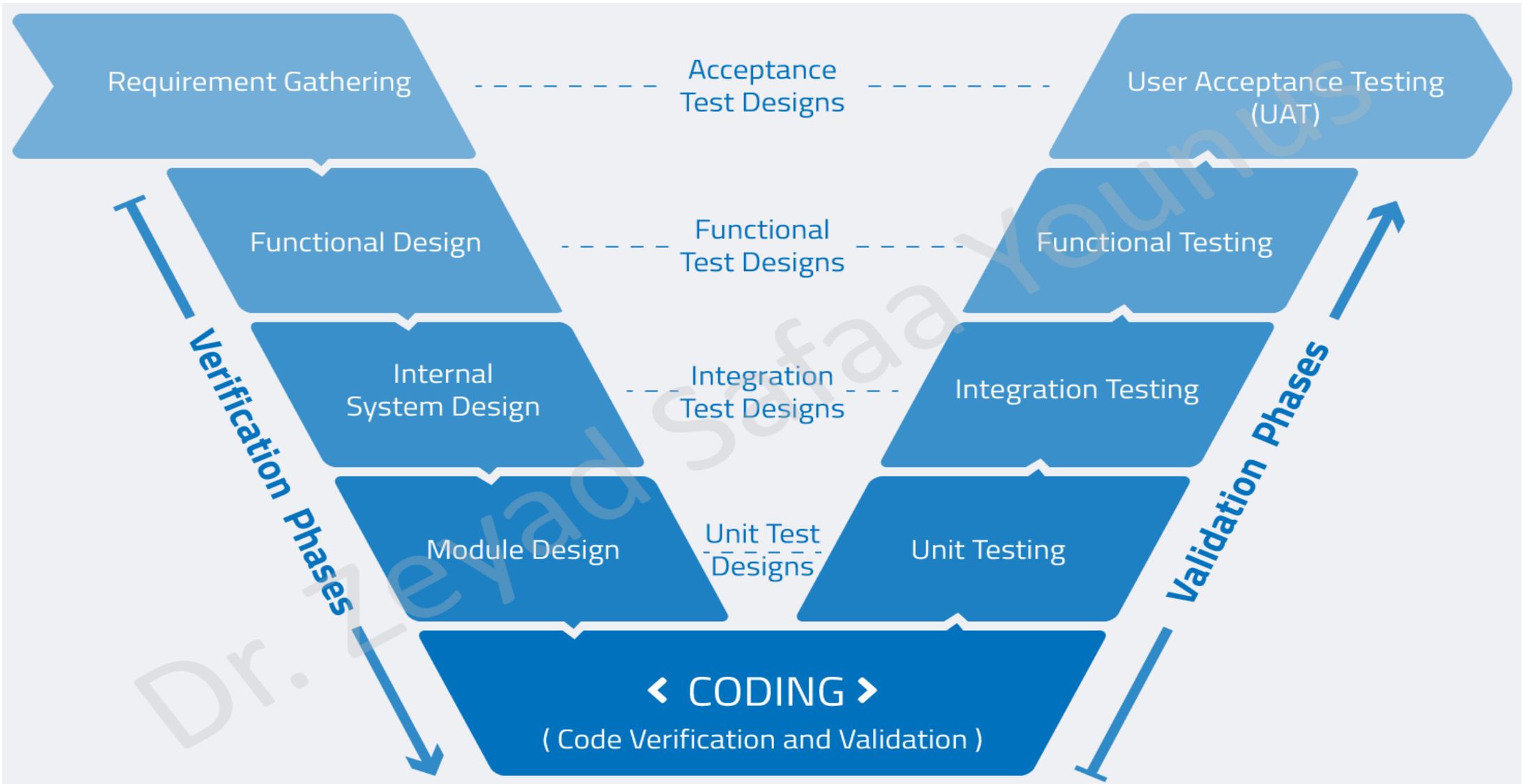
Defense Strategies in software design

- A **Defense Strategy** refers to the systematic **approach of layering defenses to enhance effectiveness and reduce costs in protecting against cyber-attacks**. It involves **implementing controls to mitigate potential impacts and improve decision-making for responding to attacks**.
- **Defense strategies in software security** aim to **protect applications and systems from various types of attacks and vulnerabilities**.
- **A multi-layered approach** is often used **to ensure comprehensive protection**.
- There are some important defense strategies such as:
  1. **Software Verification**
  2. **Language-based Security**
  3. **Manual Testing**
  4. **Sanitizers**
  5. **Fuzzing**
  6. **Symbolic Execution**

Defense Strategies in software design

1. Software Verification

- **Software verification is a crucial process in ensuring that software systems meet their specified requirements and function correctly**. **It involves various techniques and methods to confirm that the software behaves as intended and is free of critical errors**.
- **Software verification** ensure that software requirements are clear, complete, and feasible before development begins **using Static and Dynamic Analysis tool.**
  - **Static Code Analysis**: Use tools to analyze source code without executing it. This helps in identifying potential issues such as coding standards violations, security vulnerabilities, and other defects.
  - **Dynamic Verification**: Test individual components or units of code in isolation to ensure they work correctly. Test the interactions between integrated components or systems to identify issues related to their interaction. Verify the complete and integrated system against the requirements. This includes functional testing, performance testing, and security testing.
- There are many **techniques** such as:
  - **Model-Based Verification**: **Create and analyze models** of the software to **verify that it meets its requirements and behaves correctly.**
  - **Formal Specification**: Use **formal methods** to specify and interpret **about software behavior mathematically**.
  - **Simulation and Emulation**: **Test software** in a **simulated or emulated environment** to **verify its behavior under controlled conditions**.



Defense Strategies in software design

2. Language-based security focuses on **using programming languages and formal methods to enhance the security of software systems**. This approach involves **designing languages and language features that help prevent security vulnerabilities**, such as **buffer overflows or injection attacks**.

- **Here are some important concepts:**
  1. **Type Systems**: Strong type systems can help **catch errors at compile time**, **preventing many common vulnerabilities**.
  2. **Formal Verification**: This involves mathematically proving that a **program adheres to its specification and is free from certain types of errors**. Languages designed with **formal verification in mind**, such as Ada or Coq, can ensure higher levels of reliability and security.
  3. **Safe Programming Constructs**: Languages can provide **constructs that limit the kinds of operations that can be performed, reducing the risk of errors**. For instance, languages like Java and C# provide **automatic garbage collection to avoid memory leaks and dangling pointers**.
  4. **Secure Language Design**: Some languages **are designed with security as a primary concern**. For example, languages like Haskell and Scala have features that support functional programming, which can make it easier to reason about and ensure the security of code.
  5. **Security-focused Libraries and Frameworks**: **Many modern languages come with libraries and frameworks designed to prevent common security issues**, like SQL injection or cross-site scripting (XSS).

Defense Strategies in software design

3. Manual software testing involves a **tester manually executing test cases without the use of automated tools**.

- The **goal is to identify bugs or issues in the software by simulating the end-user experience**.
- Here's a **basic outline** of how it **typically works**:

  1**. Test Planning**: Define the **scope and objectives of testing**. **Create a test plan that outlines what to test, how to test, and the resources required**.

  2. **Test Case Design**: Develop detailed test cases based on the software requirements and specifications. Each test case should have clear steps, expected results, and conditions.

  3. **Test Execution**: Manually execute the test cases on the software application. This involves navigating through the application, entering data, and verifying that the software behaves as expected.

  4. **Defect Reporting**: When issues or bugs are found, document them in a defect tracking system. Include details such as steps to reproduce, expected vs. actual results, and severity.

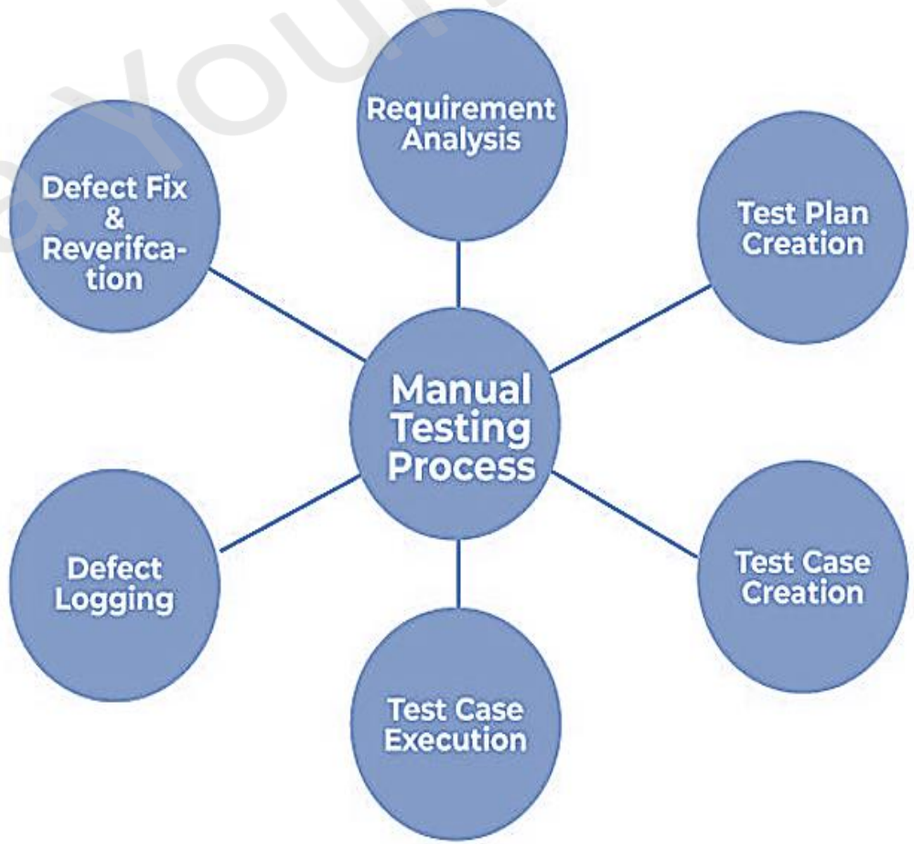
  5. **Test Closure**: Once testing is complete, review and analyze the results. Ensure all critical issues are resolved and finalize testing documentation.

  6. **Regression Testing**: After fixes are applied, retest the affected areas to ensure that the changes didn't introduce new issues.

Defense Strategies in software design

- Below the steps that used in the manual testing process:

1. **Requirement Analysis**: Study the software project documentation, guides, and Application Under Test (AUT). Analyze the requirements from SRS.
2. **Test Plan Creation**: Create a test plan covering all the requirements.
3. **Test Case Creation**: Design the test cases that cover all the requirements described in the documentation.
4. **Test Case Execution**: Review and baseline the test cases with the team lead and client. Execute the test cases on the application under test.
5. **Defect Logging**: Detect the bugs, log and report them to the developers.
6. **Defect Fix and Re-verification**: When bugs are fixed, again execute the failing test cases to verify they pass.



Defense Strategies in software design

4. Sanitizers are **tools designed to detect various types of bugs in programs**, often related to **memory safety and undefined behavior**.

- They work by **modifying the code** to **check for common issues** during **runtime**.
- Some **common types** of sanitizers include:
    1. **Address Sanitizer** (**ASan**): **Detects memory errors** such as **buffer overflows, use-after-free, and memory leaks**.
    2. **Memory Sanitizer** (**MSan**): **Identifies uninitialized memory reads**.
    3. **Thread Sanitizer** (**TSan**): **Finds data races and threading issues**.
    4. **Undefined Behavior Sanitizer** (**UBSan**): **Catches undefined behaviors**, such as **integer overflows or invalid operations**.

Defense Strategies in software design

5. Fuzzing is an **automated testing technique** that involves **sending a large volume of random or semi-random inputs** to a **program to uncover vulnerabilities or crashes**.

- It works by **generating a wide range of inputs** to **test how the software handles unexpected or malformed data**.
- **Fuzzing** can be **categorized** into different **types**:
  - **Mutation-Based Fuzzing**: Modifies existing inputs or seeds to generate new test cases.
  - **Generation-Based Fuzzing**: Creates inputs from scratch based on input formats or protocols.
  - **Coverage-Guided Fuzzing**: Uses feedback from the program's execution to guide the generation of more effective test inputs.

Defense Strategies in software design

6. Symbolic execution is a technique used to **analyze a program** by **exploring different execution paths** based on **symbolic values** rather than **concrete inputs**.

It includes:

1. **Path Exploration**: The symbolic execution engine explores possible execution paths of the program, which can lead to discovering hidden bugs and vulnerabilities.
2. **Constraint Solving**: The system generates constraints based on the symbolic inputs and solves them to find possible values that would lead to different execution paths.
3. **Error Detection**: Identifies potential issues by analyzing the constraints and the symbolic execution paths.

References

- Payer, M. (2021). *Software security: Principles, policies, and protection* (Version 0.37).