TESTING

Software Testing Life Cycle (STLC)

The Software Testing Life Cycle (STLC) is a systematic approach to testing a software application to ensure that it meets the requirements and is free of defects. It is a process that follows a series of steps or phases, and each phase has specific objectives and deliverables. The STLC is used to ensure that the software is of high quality, reliable, and meets the needs of the end-users.

The main goal of the STLC is to identify and document any defects or issues in the software application as early as possible in the development process. This allows for issues to be addressed and resolved before the software is released to the public.

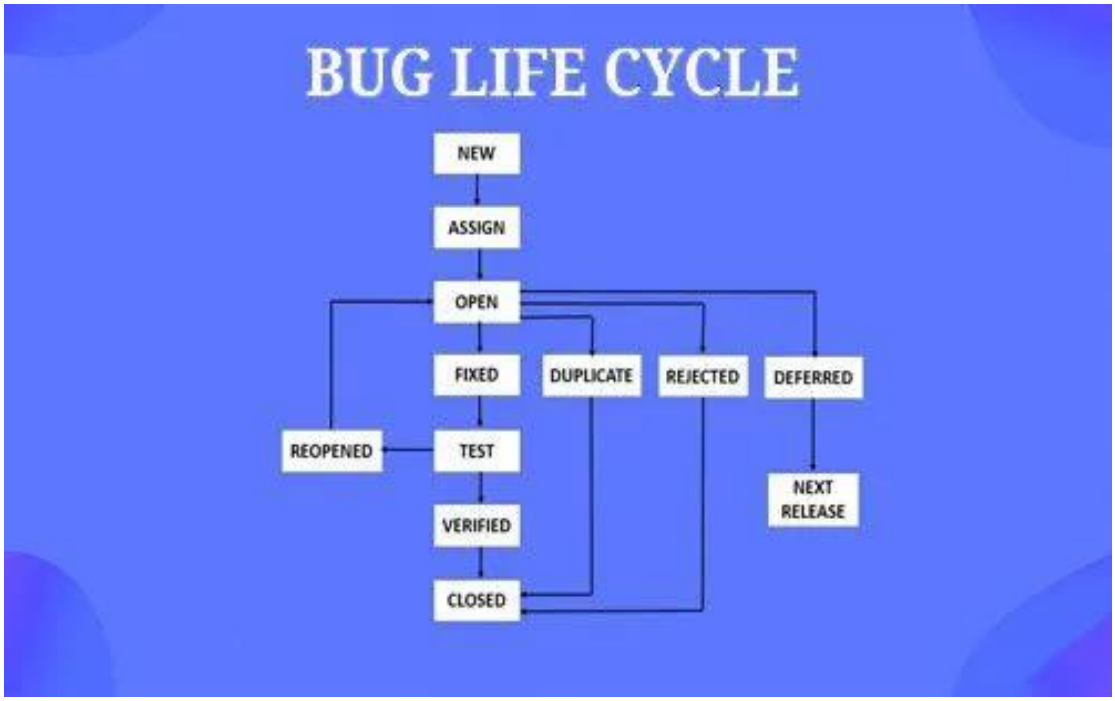
The stages of the STLC include Test Planning, Test Analysis, Test Design, Test Environment Setup, Test Execution, Test Closure, and Defect Retesting. Each of these stages includes specific activities and deliverables that help to ensure that the software is thoroughly tested and meets the requirements of the end users.

Overall, the STLC is an important process that helps to ensure the quality of software applications and provides a systematic approach to testing. It allows organizations to release high-quality software that meets the needs of their customers, ultimately leading to customer satisfaction and business success.

STLC Model Phases

- Requirement Analysis
- Test Planning
- Test case development
- Test Environment setup
- Test Execution
- Test Cycle closure

Bug Life Cycle In Software Testing



A defect in simple terms could be understood as ab Flo or an error in an application that restricts the normal flow of an application by matching the expected behavior of an application with the actual one. The defect occurs when a developer makes any mistake during a design or building of an application, and when a tester finds this flaw, it is termed as a defect.

It is the tester's responsibility to perform precise testing of an application for finding as many defects as possible to ensure that a quality product is going to reach the customer. It is essential to understand the defect life cycle before moving to the defect's workflow and different states.

Hence, let's know more about the defect life cycle. So far, we have discussed the meaning of defect and its related context to the testing activity. Now let's move towards the defect life cycle in software testing life cycle and understand the flaws workflow and the different states of a defect.

BUG/Defect Life Cycle in Detail

A defect life cycle is also known as a bug life cycle, is a cycle of the defect from which date passes through, covering the various states in its entire life. It starts as

soon as any new defect is discovered by the tester and comes to an end when the tester clauses that defect assuring it won't get generated again.

Defect Workflow

It is now the time for understanding the actual workflow of a defect life cycle:

Defect States

New:

It is the first state of a defect in the defect life cycle. When any new defect is discovered, it falls under the new condition, and validations and testing are performed on this defect in the later stages.

Assigned:

A newly created defect is given to the development team for working on the defect in this stage. It is appointed by the project lead or the manager of the testing team to a developer.

Open:

The developer commences analyzing the defect and works towards fixing it if required. Suppose the developer feels that the defect is not appropriate. In that case, it could get transferred to any of the four states, namely, duplicate, deferred, rejected, or not a bug depending upon a particular reason.

Fixed:

While the developer finishes the task of fixing a defect by making the required changes, they can mark the defect status as fixed.

Pending retest:

After the defect is fixed, the developer allot the defect to the tester for retesting the defect at their end until the tester works on protecting the defect; the state of the defect remains in the pending retest.

Retest:

The tester commences working on the defect's retesting to verify if the defect is fixed carefully by the developer as per the requirements or not at this point.

Reopen:

If any issue continues in the defect, it will be assigned to the developer again for testing, and the situation of the defect gets changed to reopen.

Verified:

If any issue in the defect after being assigned to the developer for retesting is not found. They feel that the defect status is assigned to verify if the defect has been fixed accurately.

Closed:

When the defect does not exist any longer, the tester changes the defect status to closed.

Rejected:

If the developer's defect is not considered a genuine difference, it could be marked as rejected by the developer.

Duplicate:

If the developer finds the fault the same as any other defect or if the defect matches any other defect, then the defect's state is changed to duplicate.

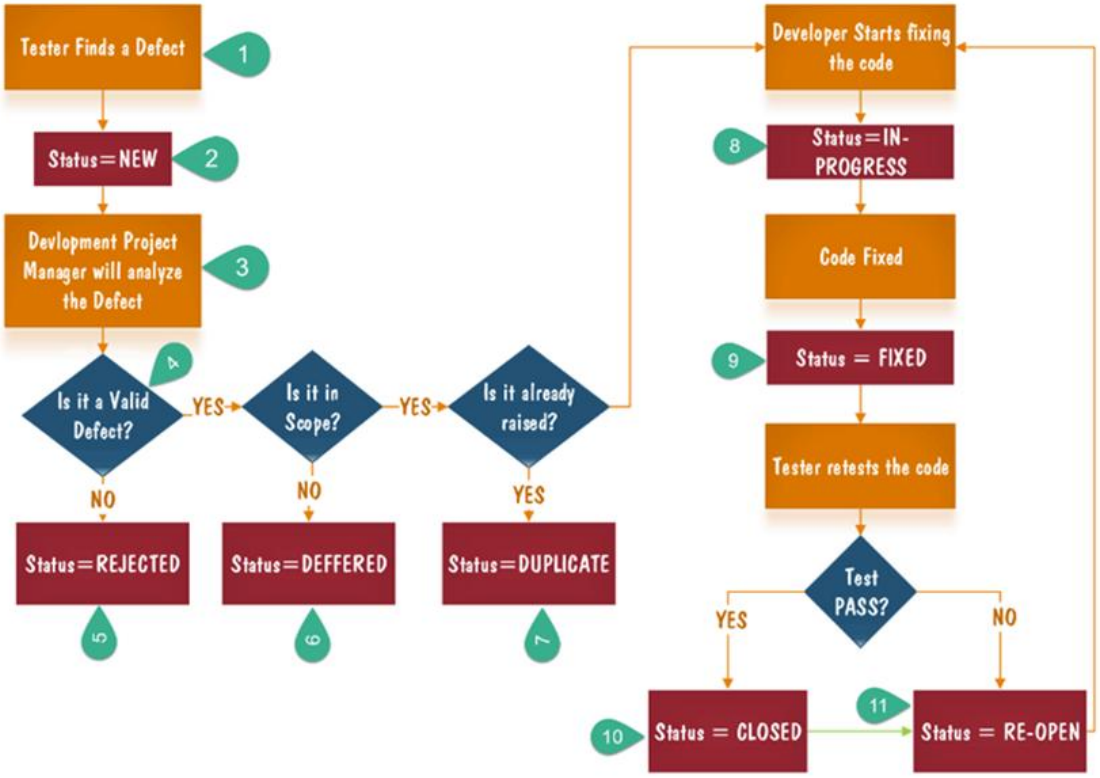
Deferred:

If the developer feels that the defect is not very prior and could be fixed in the next release, then the defect's status is changed as deferred.

Not a bug:

Not a bug: If it does not affect the functionality of the application then the status assigned to a bug is "Not a bug".

Defect/Bug Life Cycle Explained



Tester finds the defect

Status assigned to defect- New

A defect is forwarded to Project Manager for analyze

Project Manager decides whether a defect is valid

Here the defect is not valid- a status is given "Rejected."

So, project manager assigns a status rejected. If the defect is not rejected then the next step is to check whether it is in scope. Suppose we have another function- email functionality for the same application, and you find a problem with that. But it is not a part of the current release when such defects are assigned as a postponed or deferred status.

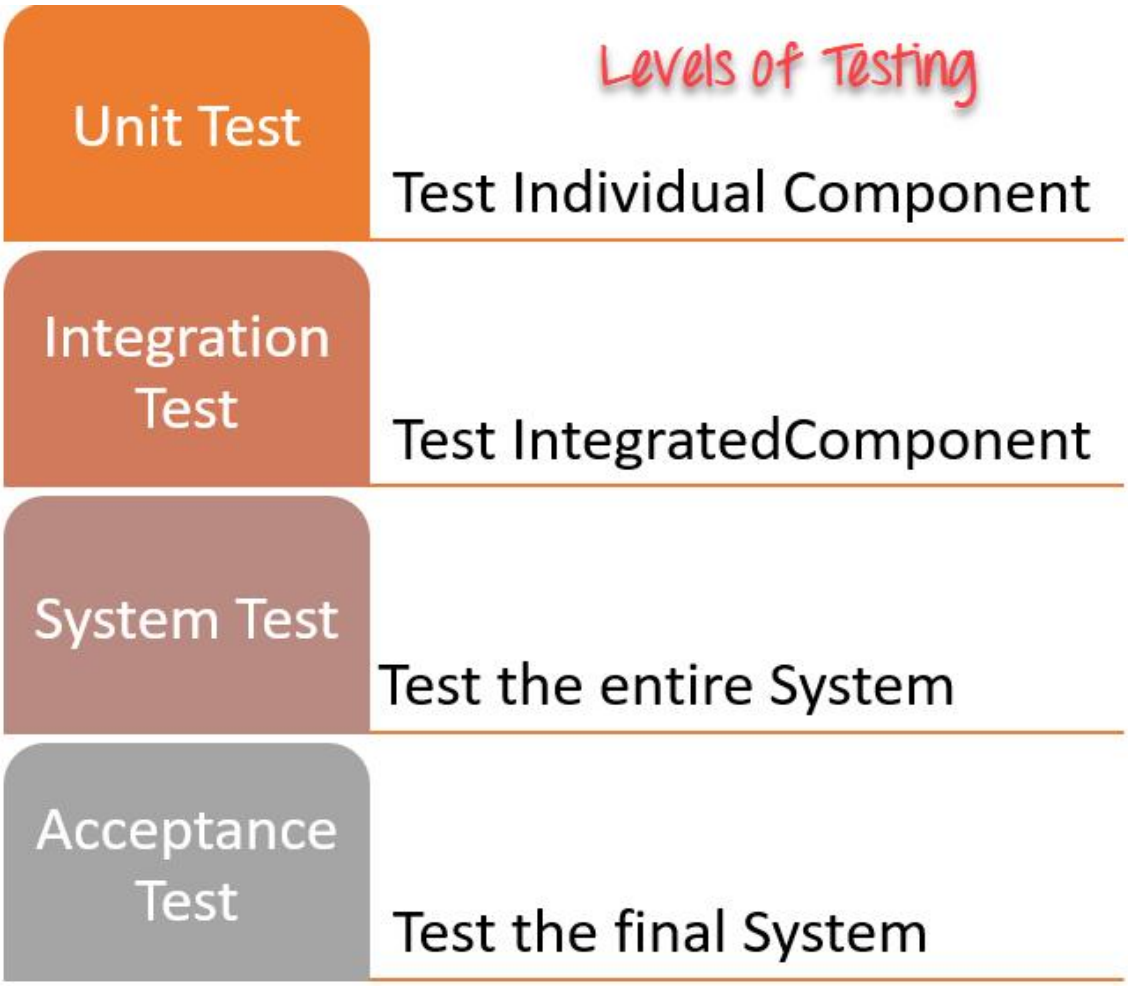
Next, the manager verifies whether a similar defect was raised earlier. If yes defect is assigned a status duplicate.

If no the defect is assigned to the developer who starts fixing the code. During this stage, the defect is assigned a status in- progress.

Once the code is fixed. A defect is assigned a status fixed

Next, the tester will re-test the code. In case, the Test Case passes the defect is closed. If the test cases fail again, the defect is re-opened and assigned to the developer.

Consider a situation where during the 1st release of Flight Reservation a defect was found in Fax order that was fixed and assigned a status closed. During the second upgrade release the same defect again re-surfaced. In such cases, a closed defect will be re-opened.



METHODS OF TESTING

Black Box Testing focuses on testing the system's functionality without considering the internal implementation. Testers have no knowledge of the internal code and solely focus on input, output, and system behavior.

White Box Testing involves testing the system's internal structure, code, and logic. Testers have full knowledge of the internal code and perform tests to achieve code coverage and validate the internal aspects of the system.

Grey Box Testing combines elements of both black box and white box testing. Testers have partial knowledge of the internal code and use that knowledge to design specific tests, focusing on particular areas or integration points of the system.

Differtiation

Testing Type	Definition	Test Knowledge	Testing Approach	Test Design Focus	Examples
Black Box Testing	Testing technique where the internal structure, design, and implementation of the system are unknown to the tester.	No knowledge of internal code	Focus on input/output and behavior	Validation of functional aspects	User Acceptance Testing, System Testing
White Box Testing	Testing technique that considers the internal structure, design, and implementation of the system.	Full knowledge of internal code	Focus on code coverage and logic	Validation of internal aspects	Unit Testing, Integration Testing
Grey Box Testing	Testing technique that combines elements of black box and white box testing.	Partial knowledge of internal code	Focus on specific areas and integration	Validation of specific aspects	API Testing, Database Testing, Penetration Testing

BLACKBOX TESTING

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

Black-box testing is a software testing technique where the internal workings of the system being tested are not known to the tester. The focus is on testing the system's functionality and behavior based on its specifications. There are several black-box testing techniques that can be used to ensure comprehensive test coverage. Here are popular black-box testing techniques:

1. Equivalence Partitioning:

Definition: Equivalence Partitioning is a black-box testing technique that divides the input domain into groups or partitions to reduce the number of test cases while ensuring adequate coverage. Test cases are designed to represent each partition, considering both valid and invalid data.

Purpose: It aims to minimize redundancy in test cases by selecting representative values from each partition, as behavior within a partition is assumed to be the same.

Identify different input data classes or ranges that are expected to produce similar results.

Divide the input domain into equivalence partitions or groups.

Select representative test cases from each partition to test the system.

Ensure that test cases cover both valid and invalid partitions.

For example, if testing a login feature, test cases can be designed for valid username/password combinations, invalid username/password combinations, and boundary cases.

Example: Testing a registration form that accepts ages between 18 and 60.

Equivalence partitions:

Valid partitions: 18-25, 26-40, 41-60

Invalid partitions: <18 and >60

Test cases:

Valid partitions: 20, 30, 50 (selecting representative values from each valid partition)

Invalid partitions: 17, 61 (testing values just outside the valid range)

2. Boundary Value Analysis:

Definition: Boundary Value Analysis is a black-box testing technique that focuses on testing the boundary values of input variables. It involves selecting test cases at or just beyond the boundaries to uncover defects that may arise due to the system's boundary behavior.

Purpose: It aims to identify issues related to off-by-one errors, boundary handling, and limit cases that may result in system failures or unexpected behavior.

Identify the boundaries or limits of input variables.

Determine the valid and invalid boundary values.

Test the system using test cases that focus on the boundaries and just beyond them.

Include test cases for both lower and upper boundaries.

For example, if testing a form that accepts a numeric value from 1 to 100, test cases can be designed with values like 0, 1, 2, 99, 100, and 101.

Example: Testing a form that accepts a password between 8 and 12 characters.

Boundary values: 7, 8, 9, 11, 12, 13 (focus on the lower and upper boundaries)

Test cases:

Lower boundary: 7 (one below the acceptable range)

On the boundary: 8, 9, 12 (testing the minimum, within range, and maximum values)

Upper boundary: 13 (one above the acceptable range)

3. Decision Table Testing:

Definition: Decision Table Testing is a black-box testing technique that models complex business logic or decision rules. It involves creating a table that maps different combinations of conditions and corresponding actions, enabling the tester to design test cases that cover all possible combinations.

Purpose: It helps ensure that the system behaves as expected by testing all possible conditions and actions, leading to comprehensive coverage and identification of defects in decision-making processes.

Identify the different combinations of conditions and actions that affect the system's behavior.

Create a decision table to represent these combinations.

Define test cases that cover all possible combinations of conditions and actions.

Execute the test cases and verify if the system behaves as expected.

For example, if testing a shopping cart checkout process, conditions can include payment method, shipping method, and order total, while actions can include order confirmation, payment processing, and shipping calculation.

Example: Testing a login feature with different combinations of username and password requirements.

Conditions: Username length, Password length, Captcha enabled

Actions: Login success, Login failure

Decision Table:

Condition	Action
Username length	
< 4	Login failure
>= 4	
Password length	
< 6	Login failure
>= 6	
Captcha enabled	
Yes	
No	Login failure

Test cases:

Username length: 3, Password length: 5, Captcha enabled: Yes (testing a combination that leads to login failure)

Username length: 5, Password length: 8, Captcha enabled: No (testing a combination that leads to login success)

4. State Transition Testing:

Definition: State Transition Testing is a black-box testing technique used to verify the system's behavior as it transitions between different states. It involves modeling the system as a finite state machine and designing test cases to cover various state transitions.

Purpose: It aims to identify defects related to state changes, initialization, and handling of different system states, ensuring that the system behaves correctly throughout its lifecycle.

Example: Testing a user authentication system with different states.

Identify the different states: Logged Out, Logged In.

Define the possible transitions between states: Login, Logout.

Test cases:

Initial state: Logged Out

Test case 1: Perform Login action, expect transition to Logged In state.

Test case 2: Perform Logout action, expect no transition (stays in Logged Out state).

Initial state: Logged In

Test case 3: Perform Login action, expect no transition (stays in Logged In state).

Test case 4: Perform Logout action, expect transition to Logged Out state.

5. Error Guessing:

Definition: Error Guessing is a black-box testing technique that relies on the tester's experience, intuition, and creativity to guess and design test cases based on potential errors that could occur in the system.

Purpose: It aims to uncover defects that may not be captured by formal test design techniques by leveraging the tester's knowledge and insights, exploring potential weak areas in the system.

Example: Testing an e-commerce checkout process for potential errors.

Leverage the tester's experience and intuition to guess possible errors.

Test cases:

Test case 1: Proceed to checkout without providing a shipping address, expect an error message.

Test case 2: Enter an invalid credit card number, expect an error message.

Test case 3: Attempt to purchase a quantity exceeding the available stock, expect an error message.

6. Cause-Effect Graphing:

Definition: Cause-Effect Graphing is a black-box testing technique that represents the relationship between input causes and their corresponding output effects. It involves creating a graphical model that captures different combinations of causes and the expected effects or system behavior.

Purpose: It helps ensure comprehensive test coverage by identifying all possible combinations of causes and effects, allowing the tester to design test cases that cover the different scenarios and uncover defects related to the system's behavior.

Example: Testing a calculator application with different inputs.

Identify the inputs (causes) and corresponding outputs (effects).

Create a cause-effect graph to map the relationships between causes and effects.

Test cases:

Input: Addition operation

Test case 1: Enter positive integers, expect the correct addition result.

Test case 2: Enter negative integers, expect the correct addition result.

Input: Division operation

Test case 3: Divide by zero, expect an error or undefined result.

Test case 4: Divide by a non-zero number, expect the correct division result.

7. User Acceptance Testing (UAT):

Definition: User Acceptance Testing is a black-box testing technique performed by end-users or stakeholders to determine if the system meets their requirements and is acceptable for deployment. It involves validating the system against predefined acceptance criteria.

Purpose: It aims to evaluate the system from the end-user's perspective, ensuring that it meets their needs, is intuitive to use, and fulfills the intended business requirements.

Example: Testing a newly developed customer relationship management (CRM) system.

Involve end-users or stakeholders to evaluate the system's usability and functionality against predefined acceptance criteria.

Test cases:

Test case 1: Create a new customer profile, input relevant information, and ensure it is saved correctly.

Test case 2: Generate a report with specific filters and verify the accuracy of the report data.

Test case 3: Perform a search for customer records using various search parameters and validate the search results.

8. All-pairs testing

All-pairs testing, also known as pairwise testing or combinatorial testing, is a black-box testing technique that focuses on generating a minimal set of test cases that cover all possible combinations of pairs of input parameters. It aims to achieve maximum coverage while minimizing the number of test cases needed.

The principle behind all-pairs testing is that most defects are caused by the interaction between pairs of input parameters rather than by individual parameters alone. By testing all possible pairs of parameter values, this technique helps uncover defects that may occur due to specific combinations of inputs.

Example:

Suppose we have a software system with three input parameters: A, B, and C. Each parameter can take on one of four possible values: 1, 2, 3, or 4. To apply all-pairs testing, we create a matrix that represents all possible combinations of pairs:

| Test Case | A | B | C |

|-----------|---|---|---|

| 1 | 1 | 1 | 1 |

| 2 | 2 | 2 | 2 |

| 3 | 3 | 3 | 3 |

| 4 | 4 | 4 | 4 |

| 5 | 1 | 2 | 3 |

| 6 | 2 | 3 | 4 |

| 7 | 3 | 4 | 1 |

| 8 | 4 | 1 | 2 |

In this example, we have a total of 16 possible combinations (4 x 4 x 4). However, by using all-pairs testing, we can reduce the number of test cases needed to achieve coverage. The matrix represents eight test cases that cover all possible pairs of parameter values. Each pair appears exactly once in the matrix, ensuring that all pairs are tested at least once.

By applying all-pairs testing, we can achieve efficient test coverage while significantly reducing the number of test cases required. This technique is particularly useful when dealing with a large number of input parameters or when combinatorial interactions are likely to introduce defects.

WHITE BOX TESTING

White-box testing, also known as structural testing or glass-box testing, is a software testing technique that examines the internal structure, design, and implementation of the system being tested. Unlike black-box testing, white-box testing requires knowledge of the internal workings of the system and its codebase.

The goal of white-box testing is to ensure that the system's internal components, such as functions, methods, and modules, function correctly, and adhere to design specifications and coding standards. It focuses on exercising the code paths, logic branches, and data flows within the system to uncover errors, bugs, and vulnerabilities.

Techniques used in white-box testing include:

Certainly! Here are the definitions of each white-box testing technique with an example for each:

1. Statement Coverage:

- Definition: Statement Coverage is a white-box testing technique that aims to execute every statement in the source code at least once.

- Example: Suppose you have a function that calculates the factorial of a number. To achieve statement coverage, you would design test cases that ensure each line of code within the function is executed during testing. For example, you could have test cases that pass different input values to the factorial function and verify that all statements within the function are executed.

2. Branch Coverage:

- Definition: Branch Coverage is a white-box testing technique that aims to test all possible branches or decision points within the code.

- Example: Consider a function that determines if a given number is positive or negative. To achieve branch coverage, you would design test cases that ensure both the true and false branches of the decision point are exercised. For example, you could have test cases where the input number is positive, negative, or zero, ensuring that all branches of the decision point are tested.

3. Path Coverage:

- Definition: Path Coverage is a white-box testing technique that aims to test all possible paths through the code.

- Example: Imagine you have a function that calculates the square root of a number. To achieve path coverage, you would design test cases that cover all unique paths within the code, including loops and conditional statements. For instance, you could have test cases where the input number is positive, negative, or zero, and ensure that all the different paths within the function are exercised.

4. Condition Coverage:

- Definition: Condition Coverage is a white-box testing technique that focuses on testing the Boolean conditions within the code.

- Example: Consider a function that checks if a given string is a palindrome. To achieve condition coverage, you would design test cases that ensure each condition within the code evaluates to both true and false. For example, you could have test cases with palindromic and non-palindromic strings to verify that the function handles both scenarios correctly.

5. Decision/Condition Coverage:

- Definition: Decision/Condition Coverage is a white-box testing technique that combines branch coverage and condition coverage.

- Example: Suppose you have a function that determines whether a given year is a leap year. To achieve decision/condition coverage, you would design test cases that ensure all possible combinations of conditions are tested. For instance, you could have test cases for years that are divisible by 4, years divisible by 100, and years divisible by both 4 and 100 to cover all decision outcomes.

6. Modified Condition/Decision Coverage (MC/DC):

- Definition: MC/DC is a white-box testing technique that focuses on testing every independent condition and decision in the code.

- Example: Consider a function that calculates the grade based on a student's score and attendance. To achieve MC/DC, you would design test cases that cover each condition and decision independently, ensuring that changing one condition affects the decision. For instance, you could have test cases with different score values, different attendance values, and combinations of the two to verify the correct grading logic.

7. Path Testing:

- Definition: Path Testing is a white-box testing technique that involves designing test cases to execute specific paths or sequences of statements within the code.

- Example: Imagine a function that sorts an array of integers using a specific algorithm. To perform path testing, you would design test cases that traverse different paths within the sorting algorithm. This includes selecting test cases with varying input array sizes, elements in sorted or reverse order, and testing the different branches and loops present in the sorting logic.

These examples illustrate how each white-box testing technique can be applied to different scenarios, allowing testers to thoroughly examine the internal structure, logic, and conditions within the codebase and ensure its correctness.

GREY BOX TESTING

Grey box testing is a software testing technique that combines elements of both black box and white box testing. In grey box testing, the tester has partial knowledge of the internal workings of the system, such as access to the code, design documents, or system architecture. This limited knowledge allows the tester to design more targeted and efficient tests.

Grey box testing techniques involve leveraging this partial knowledge to design test cases that focus on specific areas of the system. By understanding the internal structure and design, testers can identify critical paths, potential vulnerabilities, and areas where defects are more likely to occur.

Here are some grey box testing techniques along with their definitions and examples:

1. API Testing:

- Definition: API (Application Programming Interface) testing involves testing the functionality, reliability, and security of APIs. Testers use their knowledge of the API specifications and internals to design tests that verify the correctness of API interactions.

- Example: Testing an e-commerce website by sending various API requests to add items to the shopping cart, check out, and retrieve order details.

2. Database Testing:

- Definition: Database testing focuses on verifying the integrity, accuracy, and efficiency of database operations. Testers use their understanding of the database schema, tables, and queries to design tests that validate data storage, retrieval, and manipulation.

- **Example:** Verifying that data is correctly inserted, updated, and retrieved from a database using SQL queries and examining the database tables and relationships.

3. Penetration Testing:

- Definition: Penetration testing, also known as ethical hacking, aims to identify vulnerabilities and security weaknesses in the system. Testers with knowledge of the system's architecture and security protocols simulate real-world attacks to assess the system's resilience.

- Example: Conducting simulated attacks on a web application, attempting to exploit potential vulnerabilities such as SQL injection, cross-site scripting (XSS), or session hijacking.

4. Requirements-Based Testing:

- Definition: Requirements-based testing involves designing tests based on the requirements and specifications of the system. Testers with access to the requirements documents can create test cases that directly align with the intended functionality.

- **Example:** Designing test cases to validate that a banking application satisfies specific functional requirements, such as transferring funds between accounts, generating account statements, and handling overdraft scenarios.

5. Model-Based Testing:

- **Definition:** Model-based testing uses models or diagrams to represent the system's behavior and generate test cases. Testers with knowledge of the system's design can create models that capture the expected behavior and generate test cases automatically.

- **Example:** Creating a state-transition diagram to model the behavior of an elevator system, identifying different states and transitions, and generating test cases to validate the system's responses in various scenarios.

Grey box testing combines the advantages of both black box and white box testing techniques, allowing testers to focus their efforts on critical areas while still validating the system from an end-user perspective. It offers a balance between the knowledge of the system's internals and the ability to test it from a functional and non-functional standpoint.

Advantages of Grey Box Testing:

Efficient Test Coverage

Increased Test Effectiveness

Improved Bug Detection

Reduced Dependence on Documentation

Realistic User Perspective

Disadvantages of Grey Box Testing:

Limited Internal Visibility

Time and Resource Constraints

Dependency on Tester Skills

Potential Bias and Assumptions

Limited Scope of Coverage

Advantages of Grey Box Testing:

Efficient Test Coverage: Grey box testing allows testers to focus their efforts on specific areas of the system that are more likely to contain defects. This targeted approach ensures efficient test coverage and reduces redundant testing.

Increased Test Effectiveness: With partial knowledge of the internal workings, grey box testers can design test cases that exercise critical paths and potential vulnerabilities. This increases the effectiveness of testing by targeting areas that are more likely to uncover defects.

Improved Bug Detection: Grey box testing can help detect defects that may not be easily identified through black box testing alone. Testers can leverage their understanding of the system's architecture and design to identify potential areas of weakness and thoroughly test them.

Reduced Dependence on Documentation: Grey box testing can be beneficial in situations where documentation is limited or outdated. Testers can rely on their partial knowledge of the system to design tests, reducing the dependency on comprehensive documentation.

Realistic User Perspective: Grey box testing takes into account the perspective of an end user while also considering the internal system structure. This approach ensures that the system is tested from both functional and non-functional aspects, enhancing the user experience.

Disadvantages of Grey Box Testing:

Limited Internal Visibility: Grey box testers have partial knowledge of the system, which may result in overlooking certain aspects or potential defects that may exist in the unexplored areas of the system.

Time and Resource Constraints: Obtaining and maintaining partial knowledge of the system can require additional time and effort. Testers need to strike a balance between gaining the necessary insights and meeting project timelines**.**

Dependency on Tester Skills: The effectiveness of grey box testing relies heavily on the tester's ability to utilize their partial knowledge effectively. Inadequate understanding or misinterpretation of the system's internals may lead to ineffective test coverage.

Potential Bias and Assumptions: Testers with partial knowledge may unintentionally introduce bias or assumptions into the test design process. These biases can impact the effectiveness and comprehensiveness of the testing effort.

Limited Scope of Coverage: Grey box testing, by its nature, focuses on specific areas of the system. While this allows for efficient coverage in targeted areas, it may not provide comprehensive coverage of the entire system.

/PERFORMANCE TESTING\

DEFINITION:

Performance testing is a type of software testing that evaluates how well a system performs under different conditions, such as varying user loads or stress levels. Its main goal is to identify potential performance issues, measure response times, and ensure the system can handle expected workloads efficiently and reliably.

Performance Testing Attributes:

Speed:

It determines whether the software product responds rapidly.

Scalability:

It determines amount of load the software product can handle at a time.

Stability:

It determines whether the software product is stable in case of varying workloads.

Reliability:

It determines whether the software product is secure or not.

TYPES OF PERFORMANCE TESTING:

Here are the types of performance testing along with some popular tools commonly used for each type:

1. Load Testing:

Load testing evaluates the system's performance under expected user loads. It aims to determine how the application behaves when multiple users access it simultaneously. The test measures response times, resource utilization, and overall system behavior under varying loads.

- Tools:

- Apache JMeter: Open-source tool for load testing web applications, APIs, and more.

- **LoadRunner:** Performance testing tool by Micro Focus, supporting various applications and protocols.

- **NeoLoad:** Load testing tool designed for web and mobile applications.

2. Stress Testing:

- Stress testing pushes the system beyond its normal operating capacity to assess its robustness and stability. It helps identify the breaking point and measures the system's ability to recover from failures.

- **Tools:**

- **Apache JMeter:** Can be used for stress testing by pushing the system to its limits.

- **LoadRunner:** Suitable for simulating stress scenarios and measuring system response.

3. Spike Testing:

Spike testing examines how the system handles sudden spikes or bursts of user activity. It simulates a rapid increase in user load to assess whether the system can handle abrupt surges in traffic.

- **Tools:**

- **Apache JMeter:** Configurable to simulate sudden spikes in user load.

- **LoadRunner:** Capable of simulating sudden load spikes to test application behavior.

4. Endurance Testing (Soak Testing):

Endurance testing involves subjecting the system to sustained loads over an extended period to identify performance degradation, memory leaks, or resource exhaustion.

- **Tools:**

- **Apache JMeter:** Can be used for endurance testing by running tests for an extended duration.

- **LoadRunner:** Suitable for soak testing to evaluate performance over time.

5. Scalability Testing:

- Scalability testing evaluates the system's ability to handle increasing user loads or resource demands. It assesses how well the system can scale up or down to accommodate growth.

- **Tools:**

- Apache JMeter: Can be used for scalability testing by gradually increasing virtual users.

- LoadRunner: Suitable for scalability testing to evaluate application scaling.

6. Volume Testing:

- Volume testing assesses the system's performance when dealing with a large volume of data. It helps identify how the system manages and processes significant amounts of data without performance issues.

- Tools:

- Apache JMeter: Can be used for volume testing by generating large amounts of data.

- **LoadRunner:** Suitable for volume testing to evaluate performance with large data sets.

7. Parallel Testing:

Parallel testing evaluates the system's performance by running multiple instances of the application concurrently to assess how it handles parallel processing and distributed workloads.

- Tools:

- Selenium Grid: Allows parallel testing by distributing test execution across multiple machines or browsers.

8. Mobile Performance Testing:

- Purpose: Evaluates how well mobile applications perform on various devices and network conditions.

- **Tools:**

- Apache JMeter: Can be used for mobile performance testing by simulating mobile device traffic.

- NeoLoad: Specifically designed for mobile and web application performance testing.

9. Database Performance Testing:

- Purpose: Measures the speed and efficiency of database operations.

- Tools:

- Apache JMeter: Can be used for database performance testing to measure query response times.

- Apache Bench (ab): Command-line tool for load testing web servers that can indirectly assess database performance.

10. Network Performance Testing:

- Purpose: Measures network bandwidth, throughput, and latency between client and server.

- **Tools:**

- iPerf: Command-line tool to measure network performance between client and server.

The performance testing workflow involves several stages to effectively plan, execute, and analyze the performance testing process. Below is a typical performance testing workflow:

1. Requirement Gathering:

- Understand the performance requirements and objectives from stakeholders, such as response times, throughput, and expected user loads.

2. Test Planning:

- Define the scope, objectives, and performance metrics to be measured during testing.

- Identify the performance testing types to be conducted (load testing, stress testing, etc.).

- Determine the test environment, hardware, software, and network configurations.

- Create test scenarios that represent real-world usage patterns.

3. Test Design:

- Create t0est scripts and scenarios based on the planned performance tests.

- Define test data required for the test scenarios.

- Set up test data, if necessary, in the test environment.

4. Test Execution:

- Execute the performance tests according to the designed test scenarios.

- Monitor and measure system performance metrics during test execution.

- Record performance data for analysis and reporting.

- Observe application behavior under varying workloads and conditions.

5. Data Collection and Analysis:

- Collect performance data, including response times, CPU utilization, memory usage, etc.

- Analyze performance metrics to identify bottlenecks, performance degradation, or potential issues.

- Use various performance testing tools and dashboards to visualize and interpret data.

6. Results Interpretation:

- Interpret the test results to identify performance issues and areas for improvement.

- Compare the actual results with defined performance requirements and objectives.

7. Defect Reporting:

- Document and report any performance-related issues or defects found during testing.

- Include detailed information about the identified problems and their impact on the system.

8. Performance Tuning:

- Work with developers, architects, and system administrators to address performance bottlenecks and optimize the system.

- Make necessary changes to improve the system's performance and stability.

9. Re-Testing and Validation:

- Perform re-testing after performance tuning to ensure that the identified issues are resolved.

- Validate that the changes made during performance tuning have positively impacted system performance.

10. Reporting:

- Prepare comprehensive performance test reports with test results, analysis, and recommendations.

- Share the performance test reports with stakeholders, development teams, and project management.

11. Retesting (Regression Testing):

- Perform regression testing to ensure that performance improvements or changes have not introduced new issues.

The performance testing workflow is iterative, and the process may need to be repeated multiple times to fine-tune the application's performance and ensure it meets the defined performance requirements. The ultimate goal is to deliver a high-performing, reliable, and scalable application that provides an optimal user experience.

Advantages of Performance Testing:

1. Identifies Performance Bottlenecks: Performance testing helps identify performance bottlenecks, such as high response times, resource constraints, or inefficient code, enabling timely optimizations.

2. Enhances System Scalability: By simulating various user loads, performance testing ensures that the system can handle increasing demands without performance degradation.

3. Improves User Experience: Optimized performance leads to faster response times and smoother user interactions, resulting in a better user experience.

4. Reduces Downtime and Failures: Performance testing helps uncover potential issues that could lead to system crashes or downtime, allowing teams to proactively address them.

5. Cost-Effective: Early detection of performance issues reduces the cost of fixing problems in later stages of development or after deployment.

6. Validates Performance Requirements: Performance testing verifies whether the system meets the specified performance requirements and ensures compliance with SLAs.

7. Mitigates Risks: By simulating real-world scenarios, performance testing helps uncover risks associated with the application's performance before it goes live.

Disadvantages of Performance Testing:

1. Time-Consuming: Proper performance testing requires careful planning, execution, and analysis, making it time-consuming, especially for complex systems.

2. Resource Intensive: Performance testing demands considerable computing resources, including hardware, software, and skilled personnel.

3. Expensive: Acquiring and maintaining performance testing tools, infrastructure, and expertise can be expensive, particularly for small projects or teams.

4. Complex Test Scenarios: Designing realistic test scenarios that accurately simulate real-world user behavior can be challenging and require significant effort.

5. Limited Scope: Performance testing may not cover all possible scenarios or user behaviors, leading to the possibility of overlooking certain performance issues.

6. Dependency on Test Environment: Accurate performance testing results depend on a reliable and representative test environment, which may not always be easy to replicate.

7. Inaccurate Test Data: Performance testing relies on accurate test data, and using inadequate or unrealistic data can impact the validity of test results.

Security Testing

Security testing is a systematic and comprehensive evaluation of a software application, network, or system to identify vulnerabilities, weaknesses, and potential security threats. The primary aim of security testing is to assess the application's ability to withstand attacks, unauthorized access, and protect sensitive data from potential breaches. By simulating various attack scenarios and testing for potential vulnerabilities, security testing helps ensure that appropriate security measures are in place to safeguard the application and its users from potential cyber threats.

The objectives of security testing include:

- Identifying and fixing security vulnerabilities before the software is deployed to protect sensitive data and prevent potential breaches.
- Ensuring compliance with security standards, regulations, and industry best practices.
- Assessing the application's resilience against attacks and unauthorized access.
- Enhancing user trust and confidence in the application's security.

Security testing is an essential part of the software development life cycle, helping organizations protect their applications and the data they handle from potential security risks and threats.

Authentication and Authorization

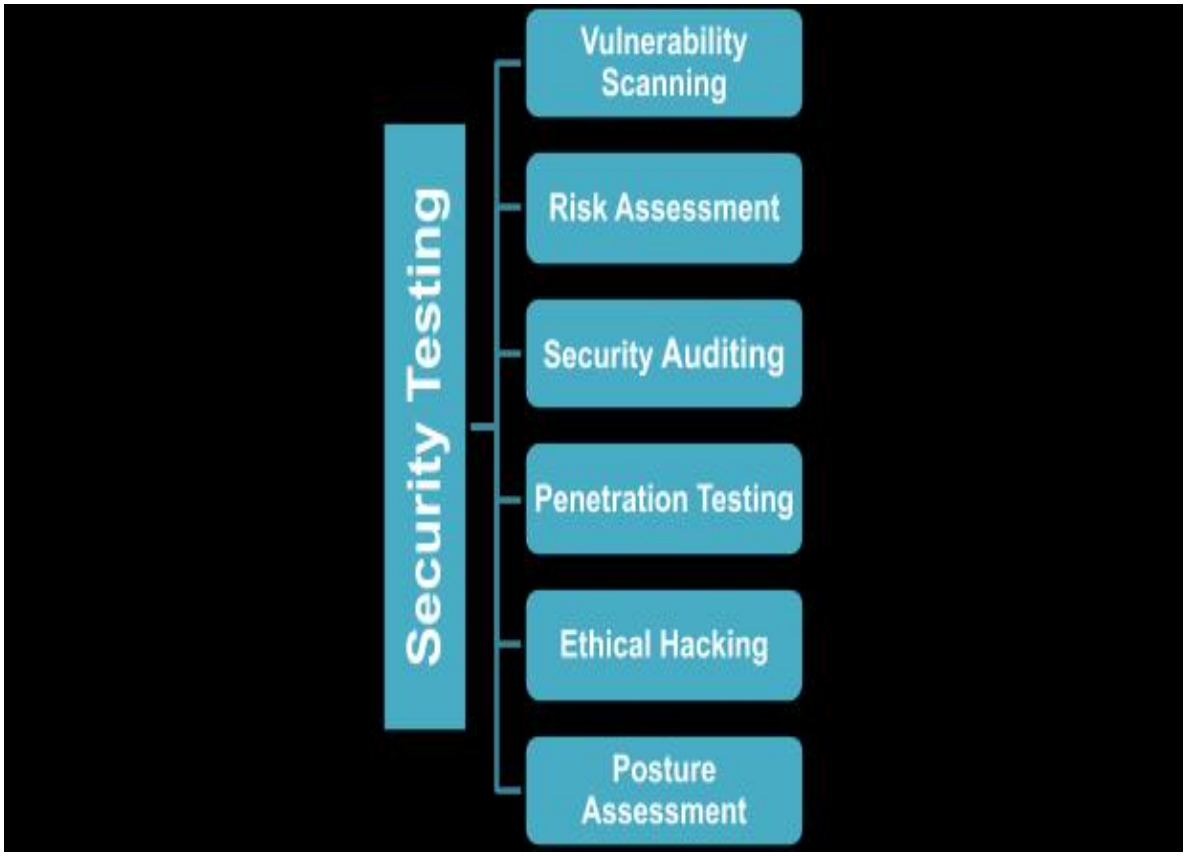
Authentication: Verifying the identity of a user or entity trying to access a system.

Authorization: Determining the permissions and privileges granted to an authenticated user or entity within the system.

Authentication verifies who the user is, while **Authorization** determines what the user is allowed to do once their identity is verified. Both authentication and authorization are essential components of access control and information security in various systems and applications.

Vulnerability

In short form, a vulnerability is a weakness or flaw in a system, software, or network that can be exploited by attackers to gain unauthorized access or cause harm.



Types of Security Testing

There are various types of security testing, each focusing on different aspects of security evaluation. Some of the common types of security testing include:

1. Vulnerability Assessment:

Identifying and assessing vulnerabilities within the application or system.

Tools: OpenVAS, Nessus, Qualys Vulnerability Management

2. Penetration Testing (Pen Testing):

Simulating real-world attacks to uncover security weaknesses and potential exploits.

Tools: Metasploit, Burp Suite, OWASP ZAP (Zed Attack Proxy)

3. Security Code Review:

Analyzing the source code for security vulnerabilities and coding errors.

Tools: SonarQube, Checkmarx, Fortify

4. Security Configuration Review:

Evaluating the security settings and configurations of servers, network devices, and applications.

Tools: CIS-CAT, Lynis, OpenSCAP

5. Security Authentication Testing:

Verifying the effectiveness of authentication mechanisms and identifying authentication-related issues.

Tools: OWASP WebGoat, THC-Hydra, Patator

6. Authorization Testing:

Assessing the access control and privileges granted to users and roles.

Tools: AccessChk, BeEF, Aclpwn

7. Session Management Testing:

Evaluating how the application handles and protects user sessions.

Tools: OWASP WebScarab, OWASP ZAP, Burp Suite

8. Data Security Testing:

Ensuring the confidentiality and integrity of sensitive data.

Tools: SQLMap, IBM Security AppScan, Acunetix

9. Cryptographic Testing:

Verifying the implementation and effectiveness of cryptographic algorithms.

Tools: Nmap, OpenSSL, TestSSLServer

10. **Security Logging and Monitoring Testing:**

    Evaluating the effectiveness of security logs and monitoring mechanisms.

    Tools: OSSEC, ELK Stack (Elasticsearch, Logstash, Kibana), Splunk

11. **Error Handling and Input Validation Testing:**

    Assessing how the application handles errors and validates user inputs.

    Tools: Peach Fuzzer, OWASP AppSensor, FuzzDB

12. **Business Logic Testing:**

    Evaluating the application's business logic for potential security flaws.

    Tools: OWASP Security Shepherd, YASAT, Wapiti

Each type of security testing contributes to the overall security posture of the application or system, helping identify and mitigate potential security risks and vulnerabilities. Organizations often combine multiple types of security testing to ensure a comprehensive security assessment.

Basic security concepts

Basic security concepts form the foundation of information security and are crucial for safeguarding data, systems, and networks from potential threats and unauthorized access. Here are some fundamental security concepts:

Confidentiality: Ensuring that sensitive information is accessible only to authorized individuals or entities and protected from unauthorized disclosure.

Integrity: Maintaining the accuracy and consistency of data and ensuring that it is not altered or tampered with by unauthorized parties.

Availability: Ensuring that data and resources are accessible and available to authorized users when needed and not subject to disruptions or denial of service.

Authentication: Verifying the identity of users or entities attempting to access a system to ensure that they are who they claim to be.

Authorization: Determining the level of access and permissions granted to authenticated users or entities based on their roles or privileges.

Encryption: The process of converting data into a secure code to prevent unauthorized access during transmission or storage.

Firewalls: Security mechanisms that control and monitor network traffic to protect systems from unauthorized access and external threats.

Intrusion Detection and Prevention: Systems that detect and respond to unauthorized attempts to access or compromise a network or system.

Least Privilege: The principle of granting users the minimum level of access necessary to perform their tasks, reducing the risk of unauthorized access to sensitive information.

Patch Management: Regularly updating software and systems with the latest security patches and updates to address known vulnerabilities.

Social Engineering: The use of psychological manipulation to deceive individuals into divulging sensitive information or granting unauthorized access.

Phishing: A type of cyber-attack that uses fraudulent communication to trick individuals into providing sensitive information or clicking on malicious links**.**

Malware: Malicious software designed to damage, disrupt, or gain unauthorized access to computer systems.

Backups: Regularly creating and storing copies of important data to recover from data loss due to cyber incidents or system failures.

Security Policies: Established rules and guidelines that define how an organization addresses security concerns and ensures compliance with security standards and regulations.

Encryption & Decryption

Encryption is the process of converting plaintext data into ciphertext to protect it from unauthorized access. It involves using encryption algorithms and keys.

Decryption is the reverse process of converting ciphertext back to plaintext using decryption algorithms and keys. It ensures authorized access to encrypted data.

Encryption and decryption are fundamental concepts in security testing and play a critical role in ensuring data confidentiality and integrity. Let's take a closer look at both processes:

Encryption:

Encryption is the process of converting plaintext (human-readable data) into ciphertext (encrypted data) using an algorithm and an encryption key. The primary purpose of encryption is to protect sensitive information from unauthorized access during transmission or storage.

In security testing, the encryption process is assessed to ensure that it is robust and adequately implemented. Testers check the following aspects:

1. Encryption Algorithms: Evaluate the strength of the encryption algorithms used. Strong and widely accepted algorithms like AES (Advanced Encryption Standard) are preferred.

2. Encryption Key Management: Verify how encryption keys are generated, stored, and managed. Proper key management is crucial to prevent unauthorized access to the keys.

3. Data Transmission: Check that sensitive data is encrypted before being transmitted over networks, especially in scenarios like HTTPS for web applications.

4. Data Storage: Assess how sensitive data is encrypted when stored in databases or files, protecting it from potential data breaches.

5. Key Length: Check the length of encryption keys used. Longer key lengths enhance security by making brute force attacks more difficult.

6. Randomization and Initialization Vector (IV): Ensure that randomization techniques and Initialization Vectors are used, enhancing the security of encrypted data.

Decryption:

Decryption is the reverse process of encryption, where encrypted data (ciphertext) is converted back to its original plaintext form using a decryption algorithm and the appropriate decryption key.

In security testing, decryption is assessed to ensure that sensitive information can be securely restored to its original form only by authorized users or systems. Testers check the following aspects:

1. Decryption Correctness: Verify that the decryption process accurately restores the original data without any loss or corruption.

2. Access Control: Ensure that only authorized users or systems have access to the decryption keys and can perform decryption.

3. Key Protection: Assess the security of decryption keys and ensure that they are adequately protected from unauthorized access or theft.

4. Decryption Errors: Test the application's response to incorrect or invalid decryption attempts, verifying that it handles errors securely.

By thoroughly testing the encryption and decryption processes, security testers can ensure that sensitive data remains protected and secure throughout its lifecycle, safeguarding it from potential unauthorized access and data breaches.

Key Terms used in Security Testing

1. Vulnerability

This is the weakness of the web application. The cause of such “weakness” can be due to the bugs in the application, an injection (SQL/ script code), or the presence of viruses.

2. URL Manipulation

Some web applications have an additional feature to communicate between the browser and the server in the URL.

Changing some information in the URL may sometimes lead to unintended behavior by the server and this termed URLManipulation.

3. SQL injection

This is the process of inserting SQL statements through the web application user interface into some query that is then

executed by the server.

4. XSS (Cross-Site Scripting)

When a user inserts HTML/client-side script in the user interface of a web application, this insertion is visible to other users and it is termed as XSS.

Sample Test Cases for Security Testing

Session Management Testing:

Test Case: Verify that the application generates a new session ID after successful login and invalidates the old session ID to prevent session fixation attacks.

Password Policy Testing:

Test Case: Check if the application enforces password complexity requirements (e.g., minimum length, alphanumeric, special characters) during user registration and password changes.

Brute Force Attack Protection:

Test Case: Test if the application locks out user accounts after a specified number of unsuccessful login attempts to prevent brute force attacks.

Error Handling and Information Leakage:

Test Case: Attempt to cause intentional errors and verify that error messages displayed to users do not reveal sensitive information about the application or server.

Data Input Validation:

Test Case: Submit invalid inputs (e.g., strings in numeric fields, large data entries) to validate if the application correctly handles and rejects such inputs.

HTTP Security Headers:

Test Case: Verify if the application uses proper security headers (e.g., Content Security Policy, Strict-Transport-Security) to enhance security against various web-based attacks.

File Inclusion Testing:

Test Case: Attempt to include unauthorized files using techniques like Local File Inclusion (LFI) or Remote File Inclusion (RFI) to check if the application is protected against such attacks.

Third-Party Library and Component Security:

Test Case: Assess the security of third-party libraries and components used in the application to ensure they are up to date and free from known vulnerabilities.

API Security Testing:

Test Case: Validate that API endpoints have proper authentication and authorization mechanisms, and access is restricted based on user roles and permissions.

User-Input Sanitization:

Test Case: Test for various input vectors, including HTML, JavaScript, and SQL, to ensure the application sanitizes user inputs effectively.

HTTPS/SSL Certificate Testing:

Test Case: Verify if the application uses a valid SSL certificate and establishes secure connections over HTTPS.

Mobile App Permissions Testing:

Test Case: Assess the permissions requested by a mobile app and ensure they are necessary and aligned with the app's functionality.

User Access Controls:

Test Case: Test different user roles (e.g., admin, regular user) to ensure that access controls are correctly implemented, restricting unauthorized actions.

Data Protection and Anonymization:

Test Case: Validate if the application adequately protects sensitive data and anonymizes personal information when necessary.

Session Timeout Testing:

Test Case: Check if the application logs out inactive users after a defined period of inactivity to prevent unauthorized access to sensitive data.

SQL Injection:

Test Case: Attempt to input malicious SQL code into input fields to check if the application prevents unauthorized access to the database.

Cross-Site Scripting (XSS):

Test Case: Inject malicious scripts into user inputs (e.g., comments, form fields) to verify if the application properly sanitizes and validates user data.

Broken Authentication and Session Management:

Test Case: Check if the application enforces strong password policies, session timeouts, and properly handles user login/logout sessions.

Sensitive Data Exposure:

Test Case: Verify that sensitive information like passwords or credit card numbers are appropriately encrypted during transmission and storage.

Security Misconfiguration:

Test Case: Attempt to access default pages, directories, and files to ensure that sensitive information is not exposed due to misconfiguration.

Cross-Site Request Forgery (CSRF):

Test Case: Create a malicious webpage that triggers actions on the target application to verify if CSRF tokens are properly implemented.

Insecure Direct Object References:

Test Case: Attempt to access restricted resources directly through manipulated URLs to check if access controls are properly enforced.

File Upload Testing:

Test Case: Upload files with malicious content (e.g., scripts, executables) to verify that the application restricts certain file types and properly scans uploads.

Business Logic Flaws:

Test Case: Analyze the application's business logic to identify potential vulnerabilities in processes such as payment handling or access control.

Insecure Cryptographic Implementation:

Test Case: Assess how the application handles encryption, hashing, and key management to ensure cryptographic best practices are followed.

User Privilege Escalation:

Test Case: Test if a regular user can escalate their privileges and access restricted functionality or data.

Denial of Service (DoS) Testing:

Test Case: Launch DoS attacks against the application to evaluate its resilience and response to such attacks.

Security Logging and Monitoring:

Test Case: Verify if security-related events are properly logged, and monitoring systems are capable of detecting suspicious activities.

Mobile Application Security:

Test Case: Assess mobile app security, including secure data storage, authentication mechanisms, and API security.

Social Engineering Testing:

Test Case: Conduct social engineering attempts (e.g., phishing) on employees to gauge their awareness and adherence to security policies.

TDM & TEM

Test Data Management (TDM) and Test Environment Management (TEM) are two distinct but closely related aspects of software testing that play essential roles in ensuring effective and efficient testing processes. Let's briefly explain each of them together:

Test Data Management (TDM)

Test Data Management (TDM) is a critical aspect of software testing that involves the planning, creation, storage, and maintenance of data used in testing environments. It ensures that the right test data is available to execute test cases effectively and efficiently. Test data plays a crucial role in software testing, as it allows testers to verify the functionality, performance, and reliability of the software.

Key objectives of Test Data Management include:

1. **Data Privacy and Security:** Ensuring that sensitive and confidential data is handled securely, and data masking or anonymization techniques are applied to protect sensitive information during testing.

2. **Data Reusability:** Creating test data in a way that it can be reused across different test cases and testing cycles, saving time and effort in test data creation.

3. **Data Accuracy:** Providing accurate and realistic data that reflects real-world scenarios and conditions, thus increasing the validity of the testing process.

4. **Data Subset and Generation:** Selecting subsets of production data or generating synthetic data to meet specific testing requirements and scenarios.

5. **Data Refresh and Cleanup:** Regularly refreshing test data from production systems to maintain relevance and cleanliness, and removing obsolete or unnecessary data.

6. **Data Versioning and Tracking:** Keeping track of different versions of test data and associating them with specific test cases to maintain consistency and traceability.

Test Data Management is particularly important when dealing with complex applications and databases. Properly managed and relevant test data can lead to more effective testing, accurate defect identification, and a reduction in software failures when the application is in production.

Many organizations use dedicated Test Data Management tools or frameworks to streamline the process and maintain data integrity throughout the testing lifecycle.

Test Environment Management (TEM)

Test Environment Management (TEM) is the process of effectively planning, setting up, configuring, maintaining, and controlling the test environments used in software testing. It ensures that the testing environments are stable, reliable, and accurately represent the production environment to conduct various types of testing.

Key aspects of Test Environment Management include:

1. **Environment Planning:** Identifying the types of test environments needed, their configurations, and the resources required for testing different aspects of the software.

2. **Environment Setup:** Creating and configuring the test environments, which may involve setting up hardware, software, databases, and network configurations to match the production environment or specific test scenarios.

3. **Data Management:** Managing the test environment data, which includes ensuring the availability of appropriate test data (real or synthetic), masking sensitive information, and refreshing data as needed

4. **Environment Monitoring:** Continuously monitoring the health and status of the test environments to identify and address any issues that may affect the testing process

5. **Environment Versioning and Control:** Managing different versions of the test environments and ensuring consistency across test cycles to maintain reproducibility and traceability.

6. **Environment Reporting:** Providing reports and metrics related to the test environments, such as usage statistics, performance data, and issues encountered, to aid in decision-making and process improvement.

Effective Test Environment Management is crucial for successful software testing as it minimizes the risk of test disruptions due to faulty environments, ensures accurate and reliable test results, and supports collaborative testing efforts among team members. It also helps in reducing the overall testing cycle time and promotes higher-quality software releases.

Together:

Test Data Management and Test Environment Management go hand-in-hand to facilitate successful software testing. Having a well-managed test environment is crucial for executing test cases, and having the right test data is equally important to validate different aspects of the software. When integrated effectively, these two processes ensure that testers have the required resources, including accurate and relevant test data, to conduct tests in a controlled and representative environment,

resulting in comprehensive and reliable testing outcomes. Both TDM and TEM contribute significantly to improving the overall quality and efficiency of software testing.

What is Exploratory Testing?

“Exploratory testing” – as the name suggests, is a simultaneous learning, test design, and test execution process. We can say that in this testing test planning, analysis, design and test execution, are all done together and instantly. This testing is about exploring the system and encouraging real-time and practical thinking of a tester.

Exploratory Testing is an informal, creative approach where testers simultaneously design and execute tests to explore the software and uncover defects. It relies on the tester's experience and intuition rather than pre-defined test plans.

Test Data Management (TDM) and Test Environment Management (TEM) work together:

Example Scenario: Online Shopping Website Testing

Suppose there is an online shopping website that needs to be tested before its official release. The testing team is responsible for validating various functionalities, such as user registration, product search, cart management, and checkout process.

Test Data Management (TDM):

Creating Test Data: The TDM team creates a variety of test data to cover different test scenarios. This includes user accounts with varying roles (regular users, premium users, admins), different product categories, various payment methods, and shipping addresses.

Data Privacy: As the website deals with user information, the TDM team ensures that any sensitive data, such as user passwords or payment details, is masked or anonymized to protect privacy during testing.

Data Reusability: Test data is organized in a way that it can be reused for various tests across different test cycles, making the testing process more efficient.

Test Environment Management (TEM):

Setting Up Test Environment: The TEM team prepares a dedicated test environment that closely replicates the production environment. It includes configuring the required servers, databases, web servers, and network settings.

Version Control: The TEM team maintains version control of the test environment to ensure that all testers are working on the same configuration during testing, minimizing inconsistencies.

Monitoring and Maintenance: The TEM team continuously monitors the health of the test environment, addressing any issues promptly to avoid interruptions in testing activities.

What is Test Basis?

Basis for the tests is called the Test Basis.

It could be a system requirement, a technical specification, the code itself, or a business process.

The test basis is the information needed in order to start the test analysis and create our Test Cases.

From a testing perspective, tester looks at the test basis in order to see what could be tested. In other words, Test basis is defined as the source of information or the document that is needed to write test cases and also for test analysis.

It should be well defined and adequately structured so that one can easily identify test conditions from which test cases can be derived.

Reviewing Test Basis is a very important activity of V- Model in SDLC. It is also an activity during the phase of Test Analysis and Design in the Testing Process. As it is most likely to identify gaps and ambiguities in the specifications, as reviewer tries to identify precisely what happens at each point in the system, and this also pre-vents defects appearing in the code.

Typical Test Basis:

Requirement document
Test Plan
Codes Repository
Business Requirement

Possible Test Basis are:

- System Requirement Document (SRS)
- Functional Design Specification
- Technical Design Specification
- User Manual

- Use Cases
- Source Code
- Business Requirement Document (BRD)

What is a Test Case?

A Test Case is a set of actions executed to verify a particular feature or functionality of your software application.

A Test Case contains test steps, test data, precondition, postcondition developed for specific test scenario to verify any requirement.

The test case includes specific variables or conditions, using which a testing engineer can compare expected and actual results to determine whether a software product is functioning as per the requirements of the customer.

What is a Test Scenario?

A Test Scenario is defined as any functionality that can be tested. It is also called Test Condition or Test Possibility. As a tester, you should put yourself in the end user's shoes and figure out the real-world scenarios and use cases of the Application Under Test.

Scenario Testing

Scenario Testing in software testing is a method in which actual scenarios are used for testing the software application instead of test cases. The purpose of scenario testing is to test end to end scenarios for a specific complex problem of the software. Scenarios help in an easier way to test and evaluate end to end complicated problems.