Ontario Tech
Business & IT

Software Testing and Verification

By

Dr. Madani

Winter 2024

Learning Objectives

Upon completion of this material, you should be able to

- Explain the importance of software testing.
- Describe different white-box testing.
- Describe different black-box testing.
- Distinguish between white-box vs black-box testing.

Introduction

- Software Testing
  - Crucial phase in the software development process.
  - Aimed at ensuring the quality, reliability, and performance of a software product.
  - The main goals are to identify defects, errors, or issues that may negatively impact the user experience or the software's intended functionality.
  - Three main categories:
    - **White-box Testing**: Testers have access to the software's internal structure, code, and design. This type of testing allows for a more in-depth evaluation of the software, including its logic and code quality.
    - **Black-box Testing**: Testers focus on the software's functionality without any knowledge of the underlying code or architecture. Test cases are designed based on the software's requirements and specifications.
    - **Grey-box Testing**: A hybrid approach that combines aspects of both black-box and white-box testing.

White-box Testing

- Testers have access to the source code, design documents, and other internal resources, enabling them to create test cases that cover various code paths, branches, and logic. Evaluating the internal structure, design, and implementation of a software application.
- Key Aspects:
  - **Code Coverage**: achieve high code coverage by creating test cases that exercise all possible code paths, branches, loops, and conditions within the application.
  - **Code Quality**: improve the overall quality of the code by identifying coding mistakes, design flaws, or violations of best practices and coding standards.
  - **Debugging**: localize defects or issues within the code, making it easier to diagnose and fix the problem.
  - **Security**: can help identify security vulnerabilities, such as injection attacks, buffer overflows, or insecure handling of sensitive data, by examining the code for patterns or constructs known to be associated with security issues.

White-box Testing – cont'd

- Common techniques:
  - **Unit Testing**: Testing individual units or components of a software application to ensure that they function correctly and meet their intended specifications.
  - **Integration Testing**: Testing the interactions and interfaces between different units or components to ensure that they work together as expected.

Functional Testing: focuses on validating whether the software meets the specified functional requirements.

  - **Static Code Analysis**: Analyzing the source code, bytecode, or binary code of a software application without executing it to identify potential security vulnerabilities, coding mistakes, or design flaws.
  - **Code Review**: Manually reviewing the source code to identify security vulnerabilities, coding mistakes, or design flaws that could lead to security issues or functional defects.

White-box Testing – cont’d

- Unit Testing
  - unit is the smallest testable part of an application and can be a single function, method, procedure, class, or module, depending on the programming language or development paradigm.
  - Helps to detect some defections early, improves code quality and documentation.
  - Key Aspects:
    - **Isolation**: aims to test each unit or component in isolation, which means that dependencies on other units, external systems, or resources should be minimized or eliminated.
    - **Automation**: is typically automated, as it involves a large number of small, focused test cases that can be executed quickly and frequently. Automated unit testing frameworks, such as JUnit (Java), NUnit (C#), or pytest (Python), provide tools and features to create, manage, and execute unit tests efficiently.
    - **Integration with Development Process**: is often integrated into the software development process and performed continuously throughout the development cycle.

White-box Testing – cont’d

- Unit Testing (Example)

XSS Vulnerable Code

```python
def html_tag(name, attrs):
    # Build and return an HTML fragment with attribute values
    # example: html_tag('meta', {'name':'test','content':'example'})
    # should return '<meta name="test" content="example">'

    result = f'<{name}'
    for attr in attrs:
        result += f' {attr}="{html.escape(attrs[attr])}"'

    return result + '>'
```

Unit test cases

Regular unit test case to test the required functionality.

A unit test case, testing for XSS vulnerability!

```python
1   class ExampleTestCases(unittest.TestCase):
2       def test_basic(self):
3           self.assertEqual(html_tag('meta', {'name':'test','content':'example'}),
4                            '<meta name="test" content="example"')
5
6       def test_parsed_html(self):
7           for content in ['x','x"']: #sample contents
8               result = html_tag('meta',{'name':'test', 'content':content})
9               soup = BeautifulSoup(result,'html.parser')
10              node = soup.find('meta')
11              self.assertEqual(node.get('name'),'test')
12              self.assertEqual(node.get('content'),content)
13
14
15  if __name__ == '__main__':
16      unittest.main()
```

White-box Testing – cont'd

- Static Code Analysis
  - The main goal is to identify potential issues, bugs, security vulnerabilities, or coding standard violations at an early stage in the software development process. This helps improve the overall quality, reliability, and maintainability of the software.
  - It is important to note that static code analysis has some limitations, including the possibility of false positives (issues reported that are not actual problems) or false negatives (actual problems not detected by the tool). Additionally, static analysis may not be able to detect certain runtime issues or complex interactions between components.

White-box Testing – cont’d

- Static Code Analysis (cont’d)
  - Static code analysis involves checking the code against a set of predefined rules, patterns, or best practices to identify possible problems.
  - Some common issues that static analysis can detect include:
    - Syntax error and coding standard violations
    - Potential security vulnerabilities
      - Injection vulnerabilities (e.g., SQL Injection, XSS, …)
      - Buffer over/under flows
      - Improper user input/output validation and sanitization
      - Insecure error handling which may lead to information disclosure
      - Use of deprecated or insecure libraries and APIs
      - …
    - Dead or unreachable code
    - Code complexity or maintainability issues

Static Application Security Testing (SAST): *SAST tools analyze the code to detect patterns, constructs, or programming practices associated with known security vulnerabilities.*

White-box Testing – cont’d

- Static Code Analysis (cont’d)
  - SAST tools can be language-specific or support multiple programming languages. Some popular SAST tools include:
    1. Fortify Static Code Analyzer (SCA) by Micro Focus
    2. Checkmarx
    3. Veracode Static Analysis
    4. SonarQube
    5. Coverity by Synopsys
    6. Clang Static Analyzer (for C/C++/Objective-C)
    7. FindBugs or SpotBugs (for Java)
    8. Bandit (for Python)
    9. Brakeman (for Ruby on Rails)

```
bandit — -bash
> bandit examples/yaml_load.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.8.2
Run started:2022-02-15 19:18:52.689821

Test results:
>> Issue: [B506:yaml_load] Use of unsafe yaml load. Allows instantiation of arbitrary objects. Consider yaml.safe_load().
   Severity: Medium   Confidence: High
   Location: examples/yaml_load.py:7:8
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b506_yaml_load.html
6           ystr = yaml.dump({'a': 1, 'b': 2, 'c': 3})
7           y = yaml.load(ystr)
8           yaml.dump(y)

--------------------------------------------------

Code scanned:
        Total lines of code: 12
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 0.0
                Medium: 1.0
                High: 0.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 1.0
Files skipped (0):
>
```

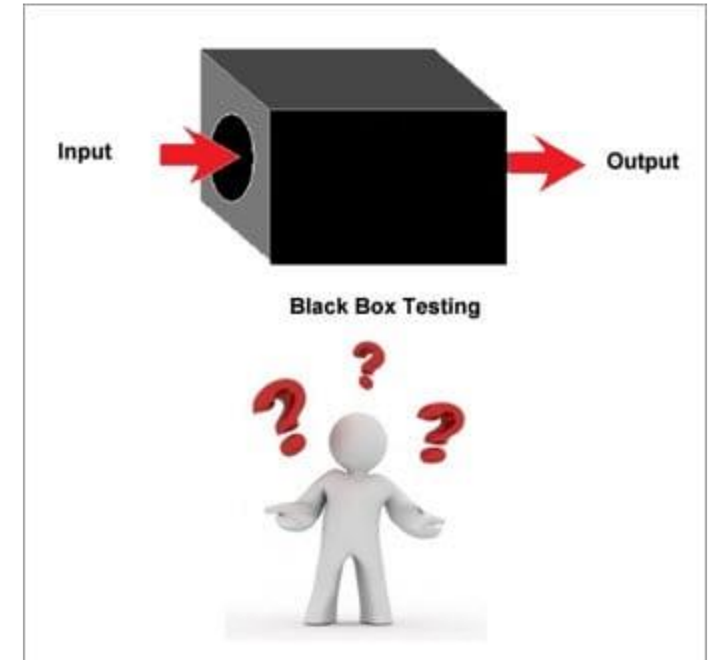
White-box Testing – cont'd

- Code Review
  - …manual process

Error Mechanism	Review Tasking
Inefficient Code	If code is obfuscated or overly complex, simplify it.
SANS Top 25 or OWASP Top 10 Previously discussed errors	All code should be reviewed to specifically prevent recurrence of previous errors. Common error patterns from SANS and OWASP lists should be checked as well.
Errors and exception handling	All functions, all procedures, and all components should fully check and handle exceptions.
Injection flaws	Input validation checks.
Cryptographic calls.	Approved libraries, good random numbers.
Unsafe and deprecated function calls	Use only approved functions.
Logging	Ensure proper logging of errors and conditions.
Secure key information	Handling of keys, PII, and other sensitive data.

Black-box Testing

- Black-box testing is a testing technique that focuses on evaluating a software based on its inputs, outputs, and expected behavior.
  - tester treats the software as a "black box" without knowledge of the internal code or architecture.
  - The primary goal is to validate that the software meets the specified requirements and functions correctly in various scenarios.



https://www.softwaretestinghelp.com/black-box-testing/

Black-box Testing – cont’d

- Common Techniques:
  - **Functional Testing**: type of testing focuses on verifying that the software's features and functions work as intended and meet the specified requirements.
  - **Non-functional Testing**: testing evaluates the software's non-functional attributes, such as performance, usability, reliability, and security.
  - **Regression Testing**: performed after any modifications or updates are made to the software to ensure that the changes do not introduce new defects or cause previously resolved issues to reappear.
  - **Smoke Testing**: involves conducting a preliminary test of the software's critical functionalities to ensure that it is stable enough for further testing.
  - **Fuzzing**: is an automated testing technique in which the tester provides random, malformed, or unexpected inputs to the software to identify potential issues, such as crashes, memory leaks, or security vulnerabilities. https://www.fuzzingbook.org/

Black-box Testing – cont’d

- Regression Testing *(as compared to execution of series of unit tests)*
  - Regression testing is focused on ensuring the stability and quality of the software after changes have been made, while unit testing is aimed at verifying the correctness and functionality of individual components during the development phase.
  - Regression testing involves re-running previously executed test cases, which can include functional, integration, and system tests, on the modified software to validate the existing functionality. It usually covers a broader scope of the software, including its interactions with other components and external systems.

https://katalon.com/resources-center/blog/regression-testing

Black-box Testing – cont’d

- Fuzzing (much broader term than “Software Fuzzing”)
  - Fuzzing, also known as fuzz testing, is a dynamic testing technique used to discover vulnerabilities, bugs, and other issues in software applications by providing malformed, unexpected, or random inputs.
    - It is used to test software, hardware and network protocols.
    - is particularly useful for discovering vulnerabilities in software that processes complex input data like network protocols, or web applications. It is an important technique for improving the security and robustness of a software application.
  - Two Common Categories:
    - **Generation-based Fuzzing**: the tester creates test inputs from scratch using a predefined set of rules or a specification. This technique is especially useful when testing software that processes structured input data, such as network protocols.
    - **Mutation-based Fuzzing**: involves taking existing valid inputs and altering them to create new test cases. The mutations can include changing, deleting, or inserting data in the original input.

Black-box Testing – cont’d

- Fuzzing – Cont’d (https://www.fuzzingbook.org/)
  - Generation-based (also known as generative) Fuzzing
    - **Grammar-based**: test inputs are generated based on a grammar or syntax that defines the structure of the input data. A grammar is a set of production rules that describe the valid combinations of elements in the input format. By applying these rules, the fuzzer can create a wide range of valid and invalid test cases that adhere to or violate the input format's structure.
      - Example: testing XML parser, a grammar-based fuzzer would generate valid test inputs according to XML syntax rules!

```
US_PHONE_GRAMMAR: Grammar = {
    "<start>": ["<phone-number>"],
    "<phone-number>": ["(<area>)<exchange>-<line>"],
    "<area>": ["<lead-digit><digit><digit>"],
    "<exchange>": ["<lead-digit><digit><digit>"],
    "<line>": ["<digit><digit><digit><digit>"],
    "<lead-digit>": ["2", "3", "4", "5", "6", "7", "8", "9"],
    "<digit>": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
}
```

(1) Describing what a valid US phone number looks like

```
>>> [simple_grammar_fuzzer(US_PHONE_GRAMMAR) for i in range(5)]
['(692)449-5179',
 '(519)230-7422',
 '(613)761-0853',
 '(979)881-3858',
 '(810)914-5475']
```

(2) Generate random valid US phone numbers to be used as inputs to test our phone book app!

https://www.fuzzingbook.org/html/Grammars.html#

Black-box Testing – cont'd

- Fuzzing – Cont'd (https://www.fuzzingbook.org/)
  - Generation-based (also known as generative) Fuzzing
    - **Protocol-based fuzzing**: testing software that uses specific communication protocols, such as HTTP, FTP, or SMTP. Test inputs are generated according to the rules and conventions of the target protocol, with the fuzzer creating both valid and malformed messages to test the software's behavior and resilience.

    - Use Scapy (a python library) for network protocol fuzzing: https://www.youtube.com/watch?v=yrmPRYSEdg0

Black-box Testing – cont'd

- Fuzzing – Cont'd (https://www.fuzzingbook.org/)
  - Generation-based (also known as generative) Fuzzing
    - **Template-based fuzzing:** uses predefined templates as a starting point for generating test inputs. These templates are modified or expanded upon to create a diverse set of test cases. This technique can be useful when the input format is complex, and manual generation of test cases would be time-consuming or error-prone.

```
>>> sql_fuzzer = SQLInjectionFuzzer(httpd_url, "DELETE FROM orders")
>>> sql_fuzzer.fuzz()
"/order?item=lockset&name=+&email=0%404&city=+'+)%3b+DELETE+FROM+orders%3b+--&zip='+OR+1%3d1--'&terms=on&submit="
```

https://www.fuzzingbook.org/html/WebFuzzer.html

Black-box Testing – cont’d

- Fuzzing – Cont’d
  - Mutation-based Fuzzing (https://www.fuzzingbook.org/html/MutationFuzzer.html)
    - Mutation-based fuzzing is particularly useful when dealing with complex input formats or when a valid input sample set is already available.
    - it is essential to have a good set of seed inputs that cover different aspects of the input format or protocol. A diverse and representative seed input set increases the chances of discovering new issues and achieving better code coverage during fuzz testing.

    - **Bit-flipping (or Byte flipping):** individual bits in the input data are flipped, generating test cases with single-bit alterations. For example, if the original data contains the byte 01101010, bit flipping might produce 11101010 or 01001010. This technique is useful for identifying issues arising from minor data corruptions.

    - **Block Swapping:** blocks of data within the input are swapped, rearranged, or shuffled to generate new test cases. This can help identify issues related to the ordering of data elements, such as incorrect parsing or data handling in the software.

Black-box Testing – cont’d

- Fuzzing – Cont’d
  - Mutation-based Fuzzing (https://www.fuzzingbook.org/html/MutationFuzzer.html)
    - **Random Mutations:** involve introducing random changes to the input data, such as adding, deleting, or modifying elements. This approach aims to cover a wide range of possible input variations to expose potential issues that might not be discovered using more structured or deterministic mutation techniques.

```
>>> seed_input = "http://www.google.com/search?q=fuzzing"
>>> mutation_fuzzer = MutationFuzzer(seed=[seed_input])
>>> [mutation_fuzzer.fuzz() for i in range(10)]
['http://www.google.com/search?q=fuzzing',
 'http://wwBw.google.com/searh?q=fuzzing',
 'http8//wswgoRogle.am/secch?qU=fuzzing',
 'ittp://www.googLe.com/serch?q=fuzzingZ',
 'httP://wgw.google.com/seasch?Q=fuxzanmgY',
 'http://www.google.cxcom/search?q=fuzzing',
 'hFttp://ww.-g\x7fog+le.com/s%arch?q=f-uzz#ing',
 'http://www\x0egoogle.com/seaNrch?q=fuZzing',
 'http//www.Ygooge.comsarch?q=fuz~Ijg',
 'http8//ww.goog5le.com/sezarc?q=fuzzing']
```

https://www.fuzzingbook.org/html/MutationFuzzer.html

Black-box Testing – cont'd

- Fuzzing – Cont'd
  - **Code Coverage**
    - in the context of fuzzing, refers to the extent to which the software's source code is exercised or executed by the generated test inputs during the fuzz testing process.
    - is a metric used to evaluate the effectiveness of the fuzzing efforts by measuring the percentage of code paths, branches, or statements that have been traversed by the test inputs.
    - the analysis can help identify portions of the code that have not been tested, allowing testers to focus their efforts on those areas or create new test cases to improve coverage.
    - Fuzzing tools, such as American Fuzzy Lop (AFL) and honggfuzz, often include built-in mechanisms for monitoring and reporting code coverage during the fuzz testing process. These tools use techniques like instrumentation, which involves adding probes to the code to track its execution, or binary analysis, which examines the compiled binary to determine code coverage without modifying the source code.

Black-box Testing – cont’d

- Fuzzing – Cont’d
  - **Code Coverage**
    - Code coverage, as a metric, can guide the random mutation process for more comprehensive testing.

```
>>> seed_input = "http://www.google.com/search?q=fuzzing"
>>> mutation_fuzzer = MutationFuzzer(seed=[seed_input])
>>> [mutation_fuzzer.fuzz() for i in range(10)]
['http://www.google.com/search?q=fuzzing',
 'http://wwBw.google.com/searh?q=fuzzing',
 'http8//wswgoRogle.am/secch?qU=fuzzing',
 'ittp://www.googLe.com/serch?q=fuzzingZ',
 'httP://wgw.google.com/seasch?Q=fuxzanmgY',
 'http://www.google.cxcom/search?q=fuzzing',
 'hFttp://ww.-g\x7fog+le.com/s%arch?q=f-uzz#ing',
 'http://www\x0egoogle.com/seaNrch?q=fuZzing',
 'http//www.Ygooge.comsarch?q=fuz~Ijg',
 'http8//ww.goog5le.com/sezarc?q=fuzzing']
```

v.s.

```
>>> mutation_fuzzer = MutationCoverageFuzzer(seed=[seed_input])
>>> mutation_fuzzer.runs(http_runner, trials=10000)
>>> mutation_fuzzer.population[:5]
['http://www.google.com/search?q=fuzzing',
 'http://wwv.oogle>co/search7Eq=fuzing',
 'http://wwv\x0eOogleb>co/seakh7Eq\x1d;fuzing',
 'http://wwv\x0eoglebkooqeakh7Eq\x1d;fuzing',
 'http://wwv\x0eoglekol=oekh7Eq\x1d\x1bf~ing']
```

https://www.fuzzingbook.org/html/MutationFuzzer.html

Summary

- Software testing is a critical part of the software development process, ensuring the quality, functionality, and reliability of the software. Here are some key points to consider when thinking about software testing:
  1. **Objective**: The primary objective of software testing is to identify and rectify defects, issues, or vulnerabilities in the software, ensuring it meets the specified requirements and functions as expected.
  2. **Types of testing**: Software testing can be broadly categorized into functional testing, which focuses on verifying the software's functionality against the requirements, and non-functional testing, which evaluates aspects like performance, usability, and security.
  3. **Levels of testing**: There are several levels of software testing, including unit testing, integration testing, system testing, and acceptance testing. Each level focuses on different aspects of the software and is performed at different stages of the development process.

Summary

4. **Testing methodologies**: Two common software testing methodologies are black-box testing, where the tester has no knowledge of the internal workings of the software, and white-box testing, where the tester has access to and understands the internal structure and implementation of the software.
5. **Manual vs. automated testing**: Software testing can be performed manually or using automated testing tools. While manual testing is essential for exploratory testing and usability testing, automation can improve the efficiency and accuracy of repetitive tasks, such as regression testing.
6. **Test case design**: Test cases should be designed to cover a wide range of possible input scenarios, edge cases, and error conditions. This includes both positive testing (testing with valid inputs) and negative testing (testing with invalid inputs).
7. **Code coverage**: Code coverage is an important metric in software testing, as it measures the extent to which the software's source code is executed during testing. High code coverage increases the likelihood of discovering hidden vulnerabilities or issues.

Summary

8. **Continuous testing**: Integrating software testing into the development process as part of a Continuous Integration/Continuous Deployment (CI/CD) pipeline can help identify and fix issues early, reducing the overall cost and effort of software development.
9. **Regression testing**: Regression testing is essential for maintaining the software's quality and stability over time, as it involves re-running previously executed test cases after changes have been made to the software.