Functional Testing

Software testing is performed to primarily attest the functionality of the software as expected by the business or customer. Functional testing is also referred to as *reliability* testing. We test to check if the software is reliable, a.k.a. is functioning as it is supposed to, according to the requirements specified by the business owner.

Unit Testing

Although unit tests are not conducted by software testers but by the developers, it is the first process to ensure that the software is functioning properly, according to specifications. It is performed during the implementation phase (coding) of the SDLC. It is performed by breaking the functionality of the software into smaller parts and each part is tested in isolation from the other parts for build and compilation errors as well as functional logic. If the software is architected with modular programming in mind, conducting unit tests are easier because each of the features would already be isolated as discreet units (high cohesiveness) and have few dependencies (loose coupling) with other units.

In addition to functionality validation, unit testing can be used to find Quality of Code (QoC) issues as well. By stepping through the units of code methodically one can uncover inefficiencies, cyclomatic complexities and vulnerabilities in code. Some common examples of code that are inefficient include dangling code, code in which objects are instantiated but not destroyed, and infinite loop constructs that cause resource exhaustion and eventually DoS. Within each module, code that is complex in logic with circular dependencies on other code modules (not being linearly independent) is not only a violation of the least common mechanisms design principle, but is also considered to be cyclomatically complex (covered in the secure software implementation chapter). Unit testing is useful to find out the cyclomatic complexities in code. Unit testing can also help uncover common coding vulnerabilities such as hard coding values and sensitive information such as passwords and cryptographic keys inline code.

Unit testing can start as soon as the developer completes coding a feature. However, software development is not done in a silo and there are usually many developers working together on a single project. This is especially true with the current day agile programming methodologies such as extreme programming (XP) or Scrum. Additionally, a single feature that the business wants may be split into multiple modules and assigned to different developers. In such a situation, unit testing can be a challenge. For example, the feature to calculate

the total price can be split into different modules; one to get the shipping rate (getShippingRate()), one to calculate the Tax (calcTax()), another to get the conversion rate for international orders (getCurrencyConversionRate()), and one to compute any discounts (calcDiscount) offered. Each of these modules can also be assigned to different developers and some modules may be dependent on others. In our example, the getShippingRate() is dependent on the completion of the getCurrencyConversionRate() and before its operation can complete, it will need to invoke the getCurrencyConversionRate() method and expect the output from the getCurrencyConversionRate() method as input into its own operation. In such situations, unit testing one module that is related or dependent on other modules can be a challenge, particularly when the method that is being invoked has not yet been coded. The developer who is assigned to the code the getShippingRate() method has to wait on the developer who is assigned the getCurrencyConversionRate() for the unit test of getShippingRate() to be completed. This is where *drivers and stubs* can come in handy. Implementing drivers and stubs is a very common approach to unit testing. Drivers simulate the calling unit while stubs simulate the called unit. In our case, the getShippingRate() method will be the driver, because it calls the getCurrencyConversionRate() method which will be the stub. Drivers and stubs are akin to mock objects that alleviate unit testing dependencies. Drivers and stubs also mitigate a very common coding problem, which is the hard coding of values inline code. By calling the stub, the developer of the driver does not have the need to hard code values within the implementation code of the driver method. This helps with the integrity (reliability) of the code. Additionally, drivers and stubs programming eases the development with 3rd party components when the external dependencies are not completely understood or known ahead of time.

Unit testing also facilitates collective code ownership in agile development methodologies. With the accelerated development efforts and multiple software teams collectively responsible for the code that is released, unit testing can help in identifying any potential issues raised by a programmer on the shared code base before it is released.

Unit testing provides many benefits. Some of these include the ability to:

- validate functional logic.
- find out inefficiencies, complexities and vulnerabilities in code, because the code is tested after being isolated into units, as opposed to it being integrated and tested as a whole. It is easier to find the needle in the haystack when the code is isolated into manageable units and tested.

- automate testing processes by integrating easily with automated build scripts and tools.
- extend test coverage.
- enable collective code ownership in agile development.

Logic Testing

Logic testing validates the accuracy of the software processing logic. Most developers are very intelligent and good ones tend to automate recurring tasks by leveraging and reusing existing code. In this effort, they tend to copy code from other libraries or code sets that they have written. When this is done, it is critically important to validate the implementation details of the copied code for its functionality and logic. For example, if code that performs the addition of two numbers is copied to multiply two numbers, the copied code needs to validated to make sure that the sign within the code that multiples two numbers is changed from '+' to 'x' as shown in *Figure 5.2*. Line by line manual validation of logic or step by step debugging (which is a means to unit test) ensure that the code is not only reliably functioning, but also provides the benefit of extended test coverage to uncover any potential issues with the code.

Code was NOT unit tested

```
public int Add(int p_iA, int p_iB)
{
    return p_iA + p_iB;
}

public int Multiply(int p_iA, int p_iB)
{
    //Code without logic validation
    //Functionally this is the same as the Add function
    //although the method name is 'Mulitply'
    return p_iA + p_iB;
}
```

Code was unit tested

```
public int Add(int p_iA, int p_iB)
{
    return p_iA + p_iB;
}

public int Multiply(int p_iA, int p_iB)
{
    //Code with logic validation
    //'+' is changed to 'x'
    return p_iA x p_iB;
}
```

Figure 5.2 – **Unit Testing for Logic Validation**

Logic testing also includes the testing of predicates. A predicate is something that is affirmed or denied of the subject in a proposition in logic. Software which has a high measure of cyclomatic complexity must undergo logic testing before being shipped or released. If the processing logic of the software is dependent on user input, logic testing must not be ignored or avoided.

Boolean predicates return a true or false depending on whether the software logic is met or not. Logic testing is usually performed by negating or mutating (varying) the intended functionality. Variations in logic can be created by

applying operators ('AND', 'OR', 'NOT EQUAL TO', 'EQUAL TO', etc.) to Boolean predicates. The source of Boolean predicates can be one of more of the following:

- Functional requirements specifications like UML diagrams, RTM, etc.
- Assurance (security) requirements
- Looping constructs (for, foreach, do while, while)
- Preconditions (if-then)

Testing for blind SQL Injection is an example of logic testing in addition to being a test for error and exception handling.

Integration Testing

Just because unit testing results indicate that the code tested is functional (reliable), resilient (secure) and recoverable, it does not necessarily mean that the system itself will be secure. The security of the *sum of all parts* should also be tested. When individual units of code are aggregated and tested, it is referred to as integration testing. Integration testing is the logical next step after unit testing to validate the software's functionality, performance and security. It helps to identify problems that occur when units of code are combined. If individual code units have successfully passed unit testing, but fail when they are integrated, then it is a clear cut indication of software problems upon integration. This is why integration testing is necessary.

Regression Testing

Software is not static. Business requirements change and newer functionality is added to the code as newer versions are developed. Whenever code or data is modified, there is a likelihood for those changes to break something that was previously functional. Regression testing is performed to validate that the software did not break previous functionality or security and regress to a non-functional or insecure state. It is also known as *verification* testing.

Regression testing is primarily focused on implementation issues over design flaws. A regression test must be written and conducted for each fixed bug or database modification. It is performed to ensure that:

- it is not merely the symptoms of bugs that are fixed but that the root cause of bugs is addressed.
- the fixing of bugs does not inadvertently introduce any new bugs or errors.

- the fixing of bugs does not make old bugs that were once fixed recur.
- modifications are still compliant with specified requirements.
- unmodified code and data have not been impacted.

It is not only functionality that needs to be tested, but the security of the software as well. Sometimes implementation of security itself can deny existing functionality to valid users. An example of this is that a menu option that was previously available to all users is no longer available upon the implementation of role based access control of menu options. Without proper regression testing, legitimate users will be denied functionality. It is also important to recognize that data changes and database modification can have side effects, reverting functionality or reducing the security of the software and so this needs to be tested for regression as well.

Adequate time needs to be allocated for regression testing. It is recommended that a library of tests are developed which includes a predefined set of tests that are to be conducted before the release of any new version. The challenge with this approach is determining what tests should be part of the predefined set. At a bare minimum tests that involve boundary conditions and timing should be included. Determining the Relative Attack Surface Quotient (RASQ) for newer versions of software with the RASQ values of the software before it was modified can be used as a measure to determine the need for regression testing and the tests that need to be run.

Usually, the software quality assurance teams are the ones who perform regression testing but since the changes that need to be made are often code related, changes that need to be made are costly and project timelines can be affected. It is, therefore, advisable that regression testing be performed by developers, after integration testing, for code related changes and also performed in the testing phase before release.

Non-Functional Testing

In addition to testing for the functional (reliable) aspects of the software, software testing must be performed to assure the non-functional aspects of the software. Non-functional testing covers testing for the recoverability and environmental aspects of the software. These tests are conducted to check if the software will be available when required and that it has appropriate replication, load balancing, interoperability and disaster recovery mechanisms functioning properly. Recoverability testing validates that the software meets the customer's Maximum Tolerable Downtime (MTD) and Recovery Time Objective (RTO) levels.

Performance testing (load testing, stress testing) and scalability testing are examples of common recoverability testing, which are covered in the following section.

Performance Testing

Testing should be conducted to ensure that the software is performing to the SLA and expectations of the business. The implementation of secure features can have a significant impact on performance and this must be taken into account. Having smaller cache windows, complete mediation, and data replication are examples of security design and implementation features that can adversely impact performance. However, performance testing is not performed with the intent of finding vulnerabilities (bugs or flaws) but with the goal of determining bottlenecks that are present in the software. It is used to find and establish a baseline for future regression tests (covered later in this chapter). The results of a performance test can be used to tune the software to organizational or established industry benchmarks. Bottlenecks can be reduced by tuning the software. Tuning is performed to optimize resource allocation. You can tune the software code and configuration, the Operating System or the hardware. Examples of configuration tuning include setting the connection pooling limits in a database server, setting the maximum number of users allowed in a web server or setting time limits for sliding cache windows.

The two common means to test for performance are load testing and stress testing the software.

Load Testing

In the context of software quality assurance, load testing is the process of subjecting the software to volumes of operating tasks or users until it cannot handle any more, with the goal of identifying the maximum operating capacity for the software. Load testing is also referred to as longevity or endurance or volume testing. It is important to understand that load testing is an iterative process. The software is not subjected to maximum load the very first time a load test is performed. The software is subjected to incremental load (tasks or users). Generally the normal load is known and in some cases the peak (maximum) load is known as well. When the peak load is known, load testing can be used to validate it or determine areas of improvement. When the peak load is not already known, load testing can be used to find it by identifying the threshold limit at which the software no longer meets the business SLA.

Stress Testing

If load testing is to determine the zenith point at which the software can operate with maximum capacity, stress testing is taking that test one step further. It is mainly aimed to determine the breaking point of the software, i.e., the point at which the software can no longer function. In stress testing, the software is subjected to extreme conditions such as maximum concurrency, limited computing resources, or heavy loads.

It is performed to determine the ability of the software to handle loads beyond its maximum capabilities and is primarily performed with two objectives. The first is to find out if the software can recover gracefully upon failure, when the software breaks. The second is to assure that the software operates according to the design principle of failing securely. For example, if the maximum number of allowed authentication attempts has been passed, then the user must be notified of invalid login attempts with a specific non-verbose error message, while at the same time, that user's account needs to be locked out, as opposed to automatically granting the user access, even if it is only low privileged guest access. Stress testing can also be used to find timing and synchronization issues, race conditions, resource exhaustion triggers and events and deadlocks.

Scalability Testing

Scalability testing augments performance testing. It is a logical next step from performance testing the software. Its main objectives are to identify the loads (which can be obtained from load testing) and to mitigate any bottlenecks that will hinder the ability of the software to scale to handle more load or changes in business processes or technology. For example, if order_id, which is the unique identifier in the ORDER table, is set to be of an integer type (Int16), with the growth in the business, there is a high likelihood that the orders that are placed after the order_id has reached the maximum range (65535) supported by the Int16 datatype will fail. It may be wiser to set the datatype for order_id to be a long integer (Int32) so that the software can scale with ease and without failure. Performance test baseline results are usually used in testing for the effectiveness of scalability. Degraded performance upon scaling implies the presence of some bottleneck that needs to be addressed (tuned or eliminated).

Environment Testing

Another important aspect of software security assurance testing includes the testing of the security of the environment itself in which the software will operate. Environment testing needs to verify the integrity of not just the

configuration of the environment but also that of the data. Trust boundaries demarcate one environment from another and end-to-end scenarios need to be tested. With the adoption of Web 2.0 technologies, the line between the client and server is thinning and in cases where content is aggregated from various sources (environments) as in the case of Mashups, testing must be thorough to assure that the end user is not subject to risk. Interoperability testing, simulation testing and Disaster Recovery (DR) testing are important verification exercises that must be performed to attest the security aspects of software.

Interoperability Testing

When software operates in disparate environments, it is imperative to verify the resiliency of the interfaces that exist between the environments. This is particularly important if credentials are shared for authentication purposes between these environments as is the case with single sign-on. The following is a list of interoperability testing that can be performed to verify that:

- security standards (such as WS-Security for web services implementation) are used,
- complete mediation is effectively working to ensure that authentication cannot be bypassed,
- tokens used for transfer of credentials cannot be stolen, spoofed and replayed, and
- authorization checks post authentication are working properly.

It is also important and necessary to check the software's upstream and downstream dependency interfaces. For example, it is important to verify that there is secure access to the key by which a downstream application can decrypt data that was encrypted by an application upstream in the chain of dependent applications. Furthermore, tests to verify that the connections between dependent applications are secure are to be conducted.

Disaster Recovery (DR) Testing

An important aspect of environment testing is the ability of the software to restore its operation after a disaster happens. DR testing verifies the recoverability of the software. It also uncovers data accuracy, integrity and system availability issues. DR testing can be used to gauge the effectiveness of error handling and auditing in software as well. Does the software fail securely and how does it report errors upon downtime? Is there proper logging of the failure in place? These are important questions to answer using DR testing. Failover testing is part of disaster testing and the accuracy of the tests is dependent on how close

a real disaster can be simulated. Since this can be a costly proposition, proper planning, resource and budget allocation is necessary and testing by simulating disasters must be undertaken for availability assurance.

Simulation Testing

The effectiveness of least privilege implementation and configuration mismatches can be uncovered using simulation testing. A common issue faced by software teams is that the software functions as desired in the development and test environments but fails in the production environment. A familiar and dangerous response to this situation is that the software is configured to run with administrative or elevated privileges. The most probable root cause for such varied behavior is that the configuration settings in these environments differ. When production systems cannot be mirrored, assurance can still be achieved by simulation testing. By simulating the configuration settings between these environments, configuration mismatch issues can be determined. Additionally, the need to run the software in elevated privileges in the production environment can be determined and appropriate least privilege implementation measures can be taken.

It is crucially important to test data issues as well, but this can be a challenge. It may be necessary to test cascading relationships but data to support that relationship may not be available in the test environment. Simulation tests for data is covered in more detail under the Test Data Management section in this chapter.

Other Testing

Privacy Testing

Software should be tested to assure privacy. For software that handles personal data, privacy testing must be part of the test plan. This should include the verification of organizational policy controls that impact privacy. It should also encompass the monitoring of network traffic and the communication between end-points to assure that personal information is not disclosed. Tests for the appropriateness of notices and disclaimers when personal information is collected must also be conducted. This is critically important when collecting information from minors or children and privacy testing of protection data, such as the age of the child and parental controls, cannot be ignored in such situations. Both Opt-in and Opt-out mechanisms need to be verified. The privacy escalation response mechanisms upon a privacy breach must also be tested for accuracy of documentation and correctness of processes.

User Acceptance Testing (UAT)

Prior to software release, during the software acceptance phase, the end user needs to be assured that the software meets their specified requirements. This can be accomplished using user acceptance testing (UAT) which is also known as end user testing or smoke testing. UAT must be the penultimate step before software is released. It is a gating mechanism used to determine if the software is ready for release or not and can help with security, because it gives the opportunity to prevent insecure software from being released into production or to the end user. Additionally, it brings the benefit of extending software testing to the end users as they are the ones who perform the UAT before accepting it. The results of the UAT are used to provide the end user with confidence of the software's reliability. It can also be used to identify design flaws and implementation bugs that are related to the usability of the software.

Prerequisites of UAT include the following:

- The software must have exited the development (implementation) phase
- Other quality assurance and security tests such as unit testing, integration testing, regression testing, software security testing, etc. must be completed.
- Functional and security bugs need to be addressed.
- Real world usage scenarios of the software are identified and test cases to cover these scenarios are completed.

UAT is generally performed as a black box test which focuses primarily on the functionality and usability of the application. It is most useful if the UAT test is performed in an environment that most closely simulates the real world or production environment. Sometimes UAT is performed in a real production environment post deployment to get a more accurate picture of the software's usability. However, when this is the case, the test should be conducted within an approved change window with the possibility of rolling back.

The final step in the successful completion of an UAT is a go/no go decision, best implemented with a formal sign off. The decision is to be captured in writing and is the responsibility of the signature authority representing the end users.