Security Testing Web Applications throughout Automated Software Tests

Stephen de Vries

stephen.de.vries@corsaire.com

Corsaire Ltd. 3 Tannery House, Tannery Lane, Send, Surrey GU23 7EF
United Kingdom

Abstract. Testing software during the development phase has become an important part of the development lifecycle and is key to agile methodologies. Code quality and maintainability is increased by adopting an integrated testing strategy that stresses unit tests, integration tests and acceptance tests throughout the project. But these tests are typically only focused on the functional requirements of the application, and rarely include security tests. Implementing security in the unit testing cycle means investing more in developer awareness of security and how to test for security issues, and less in specialised external resources. This is a long-term investment that can vastly improve the overall quality of software, and reduce the number of vulnerabilities in web applications, and consequently, the associated risks.

1 Outline

The following sections are presented below:

2 Introduction

Software development methodologies generally make a clear distinction between functional testing and security testing. A security assessment of the application is usually performed towards the end of the project; either after, or in parallel with user acceptance testing, and is almost always performed by an external security testing team. This approach has a number of serious disadvantages:

- The cost of addressing issues identified in the testing phases after the bulk of development is complete is relatively high compared to fixing bugs identified during the development phase.
- Developer involvement in testing is minimal, which means that the people with the best understanding of the code are not involved with testing it.
- Developer (and overall project) buy-in into the security process is minimised since it is perceived as an external testing exercise performed by outside experts.

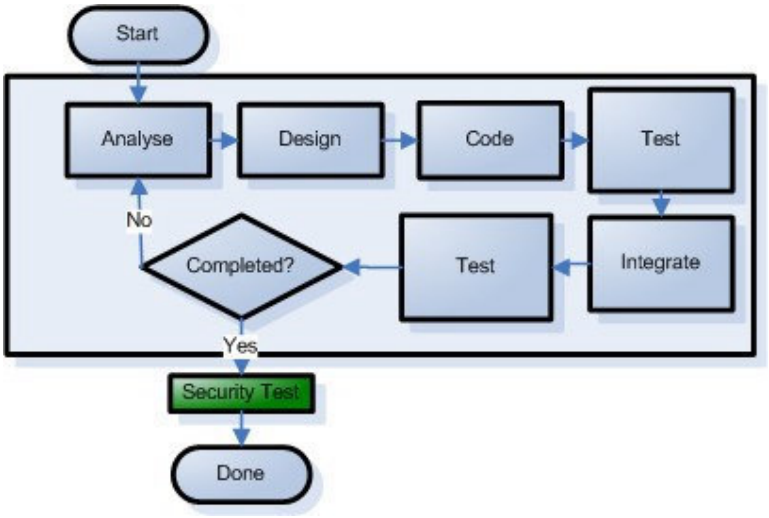

Fig. 1. Typical iterative software development cycle

Even in Agile methodologies that stress the importance of continuous and integrated testing, security is usually absent from the list of things to test. Developers tend to have their eyes fixed firmly on meeting the functional requirements without paying much attention to the security requirements. Security testing is again implemented at the end of the project, negating a lot of the benefits of an agile process.

2.1 Introducing Automated Software Testing

An automated software test is a software function used to verify that a particular module of source code is working as expected. Software tests can be written as:

- Unit Tests;
- Integration Tests; or
- Acceptance Tests.

Tests exist as distinct, self-contained source code entities that can be run against a given source code base at any time. Test cases should be written for all functions and methods so that their integrity can be tested at any point in the development process. It is important to know that a particular method functions as expected, and it is even more important to know that this method keeps functioning as expected after re-factoring and maintenance work, to prevent regressions.

Unit tests are used to test individual units of work, such as methods, functions or at most classes. These unit tests can be performed in complete isolation of both the rest of the application and also of each other. They excel at testing application and module states in exceptional conditions and not only the expected execution path. Security vulnerabilities are often introduced through software failures under precisely these exceptional conditions.

It is the thesis of this paper that security testing can, and should, be integrated into unit, integration and acceptance testing and that doing so will result in:

- A shorter security testing phase;
- More robust applications – because tests will be run on internal as well as external APIs; and
- Developer buy-in to the security process with its consequent advantages of better security in future applications.

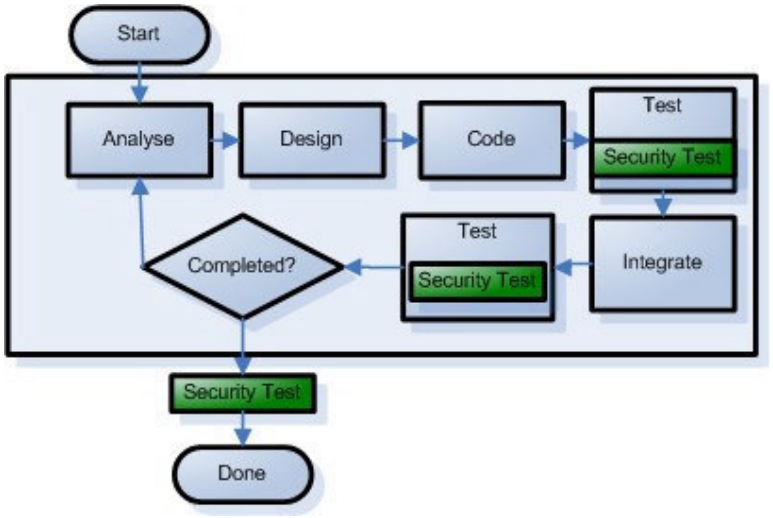

Fig. 2. Software development with integrated security tests

2.2 Use cases and Abuse cases[1]

Software testing is usually aimed at testing only the functional aspects of an application. It is generally assumed that the application will be used normally, consequently it is only the normal conditions that are tested. This is precisely the kind of thinking that contributes to the proliferation of security vulnerabilities because the actions of a user with malicious intent was never considered when designing, building or testing the application.
In addition to testing the normal functional aspects of an application, it is essential that the abnormal abuse cases also be tested. Abuse cases can be derived from a formal risk analysis of the application and specific controls to mitigate the risks can be built into the application. This should be standard practice for secure development.
In addition to formal approaches, developers could also play an active role in identifying and mitigating abuse cases by always considering the abuse potential of even small pieces of code. Once a risk has been identified, it can be mitigated and the appropriate software test written to confirm its efficacy.

3 Taxonomy of Software Tests

Software tests can be divided into groups based on their granularity and which elements of the application are tested. This also brings us to the subject of "Test Coverage" which refers to how much of the code is tested. Where only QA testing is performed on the application, only those specific execution paths exposed by the external API will be tested. This is a form of shallow testing which could allow subtle and future bugs to go undetected. Security testing is likewise, typically performed only on the functional external API. In contrast, unit and integration testing operates at multiple layers and can allow virtually every method and every class in the application to be tested which results in a high degree of test coverage.

3.1 Unit Tests

Unit tests are performed on individual classes and methods to ensure that they properly satisfy their API contracts with other classes. At this level, unit tests must be tested as isolated units without any interaction or dependency on other classes or methods. Since applications are naturally dependent on other code techniques such as stubbing or using mock objects allow test developers to stub out the dependencies so that the subject class can be tested in isolation.

Unit tests are typically written by the developers themselves to verify the behaviour of their code. These tests provide an excellent control that the internals of a class behave as expected, but because of their limited scope they cannot test the integration between modules or classes.

[1]See reference: McGraw, 2006

3.2 Integration Tests

Integration tests aim to test the integration of several classes as opposed to testing the classes in isolation. In J2EE environments, the web or EJB container provides a lot of important functionality and integration testing would therefore have to be conducted in the container, or by stubbing the relevant functions provided by the container. This class of tests could test interaction across the application tiers such as access to databases, EJBs and other resources.

Integration tests are also typically written by developers but are not executed as often as unit tests.

3.3 Acceptance Tests

Acceptance tests are at the far end of the spectrum and can be defined as the group of tests which ensure that the contract between the application API and the end user is properly satisfied. This group of tests is typically performed on the completed and deployed application and can be used to verify each use-case that the application must support. While it provides the least test coverage, it is essential in testing the complete integration of all the tiers of an application, including the services provided by application containers and web servers.

Acceptance tests are typically written by QA testers rather than by developers as the tests are far removed from the code and operate on the external API.

4 Introducing JUnit

JUnit is a Java framework for performing unit tests based on the original Smalltalk SUnit framework. Martin Fowler has said of JUnit: "Never in the field of software development was so much owed by so many to so few lines of code."

JUnit itself is a very simple framework, but the impact it has on software development is where its true value lies. On its own, JUnit is used to perform unit tests, but integrated with other testing tools it can be used to perform integration and acceptance testing.

A simple example of using JUnit to test a method from a shopping cart class follows. Consider the following interface for a shopping cart that is implemented by the Cart class:

```
interface CartInterface {
    Iterator getAllCartItems();//Returns all the items in the cart
    int getNumberOfItems(); //Returns the number of items in the cart
    boolean containsItemId(String itemId); //Checks whether an item is
already in the cart
    void addItem(Item item, boolean isInStock); //Adds an item
    Item removeItemById(String itemId); //Remove an item given its ID
    void incrementQuantityByItemId(String itemId); //Increment the quantity
of an item
    void setQuantityByItemId(String itemId, int quantity); //Set the
quantity of an item
    double getSubTotal(); //Calculate and return the subtotal
}
```

Below is the implementation detail of the addItem method that accepts an item and a Boolean flag as arguments and then adds the item to the cart. If the item is not in the cart, it is created and if it already exists a quantity counter is incremented.

```
public void addItem(Item item, boolean isInStock) {
        CartItem cartItem = (CartItem) itemMap.get(item.getItemId());
        if (cartItem == null) {
            cartItem = new CartItem();
            cartItem.setItem(item);
            cartItem.setQuantity(0);
            cartItem.setInStock(isInStock);
            itemMap.put(item.getItemId(), cartItem);
            itemList.getSource().add(cartItem);
        }
        cartItem.incrementQuantity();
    }
```

If we were to design a unit test for this method, the following tests should be considered:

- Test that a new cart has 0 items in it.
- Test whether adding a single item results in that item being present in the cart.

- Test whether adding a single item results in the cart having a total of 1 items in it.
- Test whether adding two items results in both items being present in the cart.
- Test whether adding two items results in the cart having a total of 2 items in it.
- Test whether adding a null item results in an exception and nothing being set in the cart.

This can be implemented as a JUnit test case as follows:

```
public class CartTest extends TestCase {

    public CartTest(String testName) {
        super(testName);
    }

    protected void setUp() throws Exception {
        //Code here will be executed before every testXXX method
    }

    protected void tearDown() throws Exception {
        //Code here will be executed after every testXXX method
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(CartTest.class);
        return suite;
    }

    public void testNewCartHasZeroItems() {
        Cart instance = new Cart();
        assertEquals("0 items in new cart", instance.getNumberOfItems(), 0);
    }

    public void testAddSingleItem() {
        Cart instance = new Cart();
        boolean isInStock = true;

        Item item = new Item();
        item.setItemId("item01");
        instance.addItem(item, isInStock);
        boolean result = instance.containsItemId("item01");
        assertTrue("Add single item", result);
        assertEquals("1 item in cart", instance.getNumberOfItems(), 1);
    }

    public void testAddTwoItems() {
        Cart instance = new Cart();
        boolean isInStock = true;

        //Add a single item
        Item item = new Item();
        item.setItemId("item01");
        instance.addItem(item, isInStock);

        //Test adding a second item
        Item item2 = new Item();
        item2.setItemId("item02");
        instance.addItem(item2, isInStock);

        //Check whether item01 is in the cart
        boolean result = instance.containsItemId("item01");
        assertTrue("First item is in cart", result);

        //Check whether item02 is in the cart
        result = instance.containsItemId("item02");
        assertTrue("Second item is in cart", result);
        //Check that there are 2 items in the cart
        assertEquals("2 items in cart", instance.getNumberOfItems(), 2);
    }

    public void testAddNullItem() {
        Cart instance = new Cart();
        boolean isInStock = true;

        try {
            instance.addItem(null, isInStock);
            fail("Adding a null item did not throw an exception");
        } catch (RuntimeException expected) {
            assertTrue("null Item caught",true);
```

```
            assertEquals("Null not in cart", instance.getNumberOfItems(), 0);
        }
    }
}
```

When this code is executed, JUnit will iterate through all the methods that start with the word "test", then first execute the setUp() method, then the "test" method, followed by the tearDown() method as illustrated below.

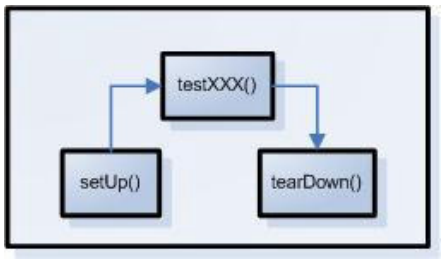

Fig. 3. JUnit's execution of each testXXX method

JUnit can execute the test methods in any order. A closer look at the testAddTwoItems() method will illustrate how JUnit works. Firstly, a new shopping cart is created, then a new item is created and added to the cart. Similarly, a second item is created and added to the cart. Next the containsItemId method is called and the result stored in a variable. An "assertTrue" statement is made to ensure that the return value was true. The same method call and assert statement is performed for the second item. Finally another assert statement, this time "assertEquals", checks that the number of items in the cart is exactly 2.

The "assert" statements make assertions about the code, should any assertion fail, it would mean that the particular test failed.

The testAddNullItem Method is an example of performing a simple test for exceptional conditions. It is important to know how the cart will behave if a null item is added to it. The test checks to ensure that an exception is thrown and that nothing was added to the cart.

JUnit is well supported in almost all Java IDEs as well as build tools such as Ant and Maven, this facilitates the execution of tests as a simple extension to the debug cycle rather than a distinct testing phase. Executing the above test case results in the following output:

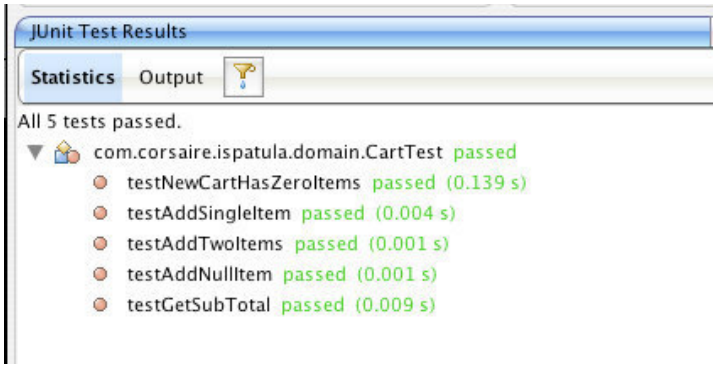

Fig. 4. Output of JUnit test case in the NetBeans IDE

The key to unit testing is that the writing and execution of tests is integrated into the development process, rather than being a distinct phase, and they can be executed at any time to ensure that code changes have not introduced regressions.

4.1 Advantages of using Unit Tests

Writing unit tests takes time and effort, but the benefits are substantial:

- They provide more test coverage of the code than QA testing which only tests the application from an external perspective.

- They allow re-factoring of the code and prevent regression. Since they are automated, it is very easy to run a test suite to ensure that all internal and external APIs function as expected after code or component changes.
- They allow teams of developers to work in parallel without having to wait for one team to complete required modules.
- They improve the design of the application by encouraging loosely coupled, pluggable components.
- They serve as living developer documentation to the code.
- They reduce the time spent debugging because component flaws are easily and quickly identified.
- They improve code quality because they encourage the developer to test for exceptional states that could cause the code to fail, instead of only concentrating on the expected execution path.

4.2 Unit testing frameworks for popular languages[2]

- Java – JUnit, (www.junit.org), TestNG (http://beust.com/testng/), JTiger (www.jtiger.org)
- Microsoft .NET - NUnit (www.nunit.org), .NETUnit (http://sourceforge.net/projects/dotnetunit/), ASPUnit (http://aspunit.sourceforge.net/), CSUnit (www.csunit.org) and MS Visual Studio Team Edition.
- PHP – PHPUnit (http://pear.php.net/package/PHPUnit), SimpleTest (www.simpletest.org)
- Coldfusion – CFUnit (http://cfunit.sf.net), cfcUnit (www.cfcunit.org)
- Perl – PerlUnit (http://perlunit.sf.net), Test::More (included with Perl)
- Python – PyUnit (http://pyunit.sf.net), doctest (included in standard library)
- Ruby – Test::Unit (included in the standard library)
- C – CUnit (http://cunit.sf.net), check (http://check.sf.net)
- C++ - CPPUnit (http://cppunit.sf.net), cxxtest (http://cxxtest.sf.net)

5 Web Application Security Standards and the Coverage Offered by Unit Tests

The JUnit example in the previous chapter demonstrates the typical use in ensuring that the functional requirements of application components are met. In some cases this could include obvious security functions such as authentication and authorisation, but there are many more security requirements that are typically not included in the functional requirements. These non-functional security requirements should be included in unit tests so as to provide a fast, accurate and repeatable view of the security of the application at any point during the development process.
The security requirements of an application should be captured in an internal Standards document. Such a standard would be derived from the organisation wide security policy and from a risk assessment performed on the application. Depending on the requirements, a Security Standard could be derived for each web application, or an organisation wide Standard for all web applications could be adopted.

4.3 Example Web Application Security Standard

The matrix below presents an extract from an example security standard for a web application; and indicates which type of software test is able to verify each of the controls. A security standard such as this is essential in defining exactly how the application's security functionality should behave.

Ref.	*Category*	*Control Question*	*Unit*	*Integration*	*Acceptance*
AU-LO	Lockout	Is there an effective account lockout?		X	X
AU-S	Storage	Are authentication credentials stored securely?		X	
CO-AU	Authorisation	Does the application properly manage access to protected resources?		X	X
CO-PM	Manipulation	Does the application successfully enforce its access control model?		X	X
CO-LO	Logout/Log off	Is a logout function provided and effective?		X	X
SM-TR	Transport	Are Session IDs always passed and stored securely?			X
SM-CT	Cookie Transport	Where cookies are used, are specific secure directives used?		X	X

[2] A more list can be found at http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Ref.	Category	Control Question	Unit	Integration	Acceptance
SM-E	Expiration	Are session expiration criteria reasonable and complete?		X	X
DV-I	Input Validation	Is all client-side input (including user, hidden elements, cookies etc.) adequately checked for type, length and reasonableness?	X	X	X
DV-SC	Special Characters	Are special characters handled securely?	X	X	X
DV-H	HTML Injection	Is HTML code as input handled securely?			X
DV-S	Active script injection	Is the application resilient to script commands as input?			X
DV-OS	OS Injection	Is access to underlying OS commands, scripts and files prevented?		X	X
DV-SQ	SQL Injection	Is the application resilient to SQL command insertion?		X	X
PT-L	Legacy data	Has all legacy data been removed from the server?			X
PT-E	Error Messages	Are all error messages generic to prevent information leakage?		X	X

It is clear that the vast majority of controls can be tested using functional, and to a lesser extent, integration testing techniques. Unit tests are only able to test a limited number of controls due to the fact that, in typical applications, a lot of security functionality is provided by other modules, the web server, or web container.
The next three sections will provide more detail on how to perform security tests in unit, integration and acceptance tests.

6 Testing Security in Unit Tests

Testing individual classes and methods provides a fine-grained approach to testing code functionality. Unit tests should be performed on individual classes and methods without a dependency on other classes. This limits the types of security tests that can be performed, but allows the tests to be executed very early in the development process.

6.1 Testing in Isolation

Unit tests should only test a single class and should not rely on helper or dependent classes. Since few classes exist in such a form of isolation it is usually necessary to create a "stub" or "mock" of the helper class that only does what is expected by the calling class and no more. Using this technique has the added benefit of allowing developers to complete modules in parallel without having to wait for dependent modules to be completed. To enable this form of testing it is important that the code is pluggable, this can be achieved by using the Inversion of Control (IoC) or Service Locator[3] design patterns. Pluggable code using these patterns is a worthy goal in itself and the ease with which they allow tests to be performed is just one of their many advantages.

6.2 Vulnerability Testing Coverage

The number of security controls that can be verified through unit tests will depend largely on how security services are implemented in the application. A control concerning the ciphers used for the SSL session, for example, cannot be tested at the unit level since this is provided entirely by the application server or web server. Similarly, services such as meta-character encoding and access control could be implemented in the code, provided by a framework or by the application server. Usually only the former case can be tested in isolated unit tests.

6.3 Example: Testing Input Validation

Validation of user-supplied data can be performed in a number of different areas in a web application. To support modular application design, it is recommended that data validation be performed in the

[3] Inversion of Control Containers and the Dependency Injection Pattern by Martin Fowler: http://www.martinfowler.com/articles/injection.html

domain object itself. The validation rules remain portable and will be executed even when the front end of the application is changed. An example of performing validation testing using the Spring framework and JUnit is provided below:

```
public class AccountValidatorTest extends TestCase {
    private Account acc = null;
    private AccountValidator validator = null;
    private BindException errors = null;

    public AccountValidatorTest(String testName) {
        super(testName);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(AccountValidatorTest.class);
        return suite;
    }

    public void setUp() {
        acc = new Account();
        validator = new AccountValidator();
        errors = new BindException(acc, "Account");
    }

    public void testValidPhoneNumbers() {
        //Test valid input
        String number = "232321";
        acc.setPhone(number);

        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
errors.hasFieldErrors("phone"));

        number = "+23 232321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
errors.hasFieldErrors("phone"));

        number = "(44) 32321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
errors.hasFieldErrors("phone"));

        number = "+(23)232 - 321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertFalse(number+" caused a validation error.",
errors.hasFieldErrors("phone"));
    }

    //Test invalid input
    public void testIllegalCharactersInPhoneNumber() {
        String number = "+(23)';[]232 - 321";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertTrue(number+" did not cause a validation error.",
errors.hasFieldErrors("phone"));
    }

    public void testAlphabeticInPhoneNumber() {
        String number = "12a12121";
        acc.setPhone(number);
        validator.validate(acc, errors);
        assertTrue(number+" did not cause a validation error.",
errors.hasFieldErrors("phone"));

    }

}
```

When testing security functionality it is important that both valid input is accepted (a functional requirement), and also that invalid and potentially dangerous data is rejected. Testing boundary and unexpected conditions is essential for security tests.

6.4 Discussion

Implementing security tests at the Unit level is preferable to implementing the same tests at the integration or acceptance level because the tests are executed very early on in the development cycle. However, the number of security controls that can be tested as unit tests is limited by the fact that the majority of security issues facing web applications are simply not visible at the single class level.

7 Testing Security in Integration Tests

Integration tests aim to test the functionality of collaborating classes, including functionality provided by the Application server. Integration tests can be conducted using Mock objects or by running the tests within the container. In-container testing has the benefit of allowing developers to test the security services provided by the container such as access control and encryption. Compared to unit tests, many more security controls can be tested using integration tests.

7.1 Testing Strategies

There are primarily two ways to perform integration tests; using mock objects to provide a mock implementation of the application server API, or by running the tests in an application server (in-container testing). A number of projects[4] ease the process of writing mock objects and provide mechanisms for mocking common API's such as the Java, Servlet and EJB APIs. Mock objects provide a general way to perform integration testing in any environment.
In-container testing requires specific tools for specific containers, consequently there are fewer options in this space. For J2EE testing popular choices are Apache Cactus (http://jakarta.apache.org/cactus/) and TESTARE (http://www.thekirschners.com/software/testare/testare.html).

7.2 Apache Cactus

Apache Cactus has become a standard testing tool for in-container testing of Java web applications. It allows testing of web and EJB applications and includes convenience plugins for Jetty, Ant, Maven and Eclipse. The disadvantage of using Cactus is that a container has to be started and stopped for the tests to run, for lightweight products such as Jetty this takes a few seconds, but for full blown J2EE containers this may be a lot longer.

7.3 Example: Testing Container Managed Access Control

Consider a web application that uses container managed security to restrict access to the /admin/* resource to only those users that are in the administrators role. To ensure that only those users can access the resource the following code performs three tests: first to verify that admin users can access the resource, then to verify that unauthenticated users cannot access the resource and lastly it checks that users from other roles cannot access the resource.

```
public class TestAccessControl extends ServletTestCase {
    public TestAccessControl(String theName) {
        super(theName);
    }

    public static Test suite() {
        return new TestSuite(TestAccessControl.class);
    }

    public void beginAdminAccessControl(WebRequest theRequest) {
        theRequest.setAuthentication(new BasicAuthentication("admin",
"admin"));
    }

    public void testAdminAccessControl() throws IOException,
javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
```

[4] See: http://www.mockobjects.com for more information

```
    }

    public void endAdminAccessControl(WebResponse theResponse) throws
IOException {
        int position = theResponse.getText().indexOf("Welcome
administrator");
        assertTrue("Administrator can view /admin", position > -1);
        assertTrue("false", false);
    }

    public void testUnauthenticatedAccessControl() throws IOException,
javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
    }

    public void endUnauthenticatedAccessControl(WebResponse theResponse)
throws IOException {
        assertTrue("Unauthenticated users must not be able to access
/admin", theResponse.getStatusCode() == 401);
    }

    public void beginUnprivilegedUserAccessControl(WebRequest theRequest) {
        theRequest.setAuthentication(new BasicAuthentication("user",
"password"));
    }

    public void testUnprivilegedUserAccessControl() throws IOException,
javax.servlet.ServletException {
        AdminServlet admin = new AdminServlet();
        admin.doGet(request, response);
    }

    public void endUnprivilegedUserAccessControl(WebResponse theResponse)
throws IOException {
        assertTrue("Normal users must not be able to access /admin",
theResponse.getStatusCode() == 401);
    }
}
```

Cactus works by implementing tests in the whole HTTP request-response conversation. The life cycle of a single Cactus test is illustrated below.

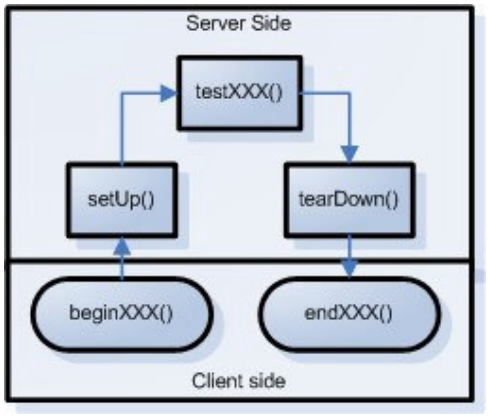

Fig. 5. Single Cactus test

For each testXXX method specified in the test case, the following actions are taken:

Step 1. Execute beginXXX methods which setup client side data needed for the test, for example the beingUnprivilegedUserAccessControl method above sets the authentication credentials for HTTP BASIC authentication.

Step 2. Execute the setUp() method on the server side if it exists. This method is used execute test code that is common to all server side tests. In the test case above, there is no such common code.

Step 3. Execute the testXXX() method on the server side. This is where the core of the server side testing is done. Tests are JUnit tests and follow the familiar format.

Step 4. Execute the tearDown() method which contains the common server side code to be executed when a test has completed. In the example above, none was required.

Step 5. Execute the endXXX methods on the client side. In this step the results returned from the server can be tested. The endUnauthenticatedAccessControl() method in the above test case makes an assertion to ensure that the HTTP status code for the response was 401 (Unauthorised access).

7.4 Discussion

The integration layer offers many opportunities to test for security vulnerabilities since the complete security feature set of the application is exposed and can be tested. Security tests that can be performed at this layer include Injection flaws, Authentication bypass and Access Control tests.

In container testing is a powerful form of integration testing that allows realistic tests to be run against the application because the testing is performed in a real application server. But this approach suffers from the overhead of starting and stopping the application server, as well as a limited number of testing frameworks. Although the number of security tests that can be performed at the integration layer is much more than at the unit testing layer, there are still some issues which can't be tested such as Cross Site Scripting and services provided by a web server (e.g.: SSL, URL filtering).

It is appropriate and common for developers to write integration tests, and in agile methodologies it is also recommended that integration tests be executed at least daily[5]which will help identify issues early on. Skilled developers who have undergone training in security testing techniques and who understand software security issues will be well equipped to write security tests at this layer.

8 Security Testing in Acceptance Tests

Acceptance testing is at the far end of the testing spectrum and can be considered an automated form of QA testing. Acceptance tests are performed against the whole application as deployed in the application server. This allows complete testing of all application security functions, but does not offer as much test coverage as integration or unit tests. Writing security tests at the acceptance testing level is more appropriate for security or QA testers as opposed to developers.

8.1 Testing tools

There are a number of testing tools available; the Java based tools typically use the HTTP functions provided by the J2SE API or a custom HTTP client to perform the tests. They differ in how they handle the presentation tier and the degree of low-level access to HTTP they offer. Since they act as external HTTP clients, the language used by the client and that of the web application need not be the same. Popular tools in this space, include HttpUnit, jWebUnit, HtmlUnit, Canoo Webtest and the Ruby based WATIR project[6].

Testing at the functional layer is more natural for a dedicated tester than a developer and since the tests require full end-to-end functionality they can reasonably only be run towards the end of the development process. Security testers are able to use the features of the functional tools to verify and reproduce the results of a manual security assessment. This could greatly reduce the time needed for retesting, since all the discovered vulnerabilities could be scripted during the initial assessment.

8.2 Example: Testing HTML injection with jWebUnit

The test case below checks to make sure that the search field of a web application is not susceptible to HTML injection.

```
public class XSSinSearchFieldTest extends WebTestCase {
    public XSSinSearchFieldTest(String name) {
        super(name);
    }

    public void setUp() throws Exception {
        getTestContext().setBaseUrl("http://localhost:8084/ispatula/");
    }
```

[5] http://www.martinfowler.com/articles/continuousIntegration.html
[6] For a more complete list see: http://opensourcetesting.org/functional.php

```
    public void testHtmlInjection() throws Exception {
        beginAt("/index.html");
        assertLinkPresentWithText("Enter the Store");
        clickLinkWithText("Enter the Store");
        assertFormPresent("searchForm");
        setFormElement("query", "<a id=\"injection\"
href=\"http://www.google.com>Injection</a>");
        submit();
        //If the link is present, it means injection succeeded, therefore
our test should fail
        assertLinkNotPresent("injection");
    }
}
```

jWebUnit follows the familiar JUnit format for tests and extends this with HTML and HTTP aware functions. jWebUnit maintains an internal conversation state which keeps track of which page is being viewed and manipulated. In the test case above, the testHtmlInjection method performs the testing, most of the method calls are self explanatory. The setFormElement method is used to set the value of a form field, in this case an attempt is made to insert an HTML link into the "query" value. If the link is present after the form is submitted, then the test case should fail since the function is susceptible to HTML injection.

8.3 WATIR

The Web Application Testing in Ruby tool takes a different approach to the aforementioned tools in that it does not use its own HTTP client but instead drives an instance of Internet Explorer. This approach means that tests are representative of how real world web clients behave. But it has the disadvantage that low level tests (such as those testing at the HTTP level) have to be coded manually.

Ruby is a high level dynamic scripting language which can be understood by non-developers and programmers alike.

8.4 Example: Testing for SQL injection in a login form

```
require 'unittests/setup'
require 'watir'

$APP_HOME = 'http://localhost:8080/ispatula'
$USERNAME = 'corsaire1'
$PASSWORD = 'corsaire1'
$SQL_CONCAT_USERNAME = 'corsaire\'+\'1'

class SQL_Injection_Test < Test::Unit::TestCase
    include Watir

    def test_SQL_Blind_Injection()
        $ie.goto($APP_HOME)
        $ie.link(:url, /signonForm.do/).click
        $ie.text_field(:name, 'username').set($USERNAME+'\' OR 1=1--')
        $ie.form(:action, "/ispatula/shop/signon.do").submit
        assert($ie.contains_text('Signon failed'));
    end

    def test_SQL_Injection_String_Concat()
        $ie.goto($APP_HOME)
        $ie.link(:url, /signonForm.do/).click
        $ie.text_field(:name, 'username').set($SQL_CONCAT_USERNAME)
        $ie.text_field(:name, 'password').set($PASSWORD)
        $ie.form(:action, "/ispatula/shop/signon.do").submit
        assert($ie.contains_text('Signon failed'));
    end
end
```

As with jUnit, test methods are labeled by starting the method name with the word "test". The $ie object holds a reference to Internet Explorer and provides access to the entire IE DOM.
The test_SQL_Blind_Injection method first navigates to the logon form by calling $ie.goto to access the application start page, then it finds a link in the page that matches the regular expression signonForm.do and clicks it. The text field with the name "username" is set to the value of: `corsaire1' OR 1=1–` and the form is submitted. Next, an assertion is made to ensure that the page returned contains the text

"Signon failed", if it does the test passes and we know that the form is not susceptible to this form of SQL injection.
The test_SQL_Injection_String_Concat method also tests for SQL injection, but uses a different technique, that of concatenating strings in an SQL statement. It first navigates to the correct page, then signs on with a username of: corsaire'+'1'. If the application is vulnerable to SQL injection then that username will be concatenated to: "corsaire1" which is a valid username. Consequently, if the login is successful, then the application is vulnerable to SQL injection and the test case should fail.

8.5 Example: Testing Access Control

Consider an application that allows only administrative users to view all orders placed by accessing the URL: /shop/listOrders.do. Should regular users attempt to access this resource they should be redirected to the login page. The following test methods could be used to verify this behaviour:

```
def test_Access_Control_List_Orders_Unauthorised()
    #First check unauthenticated access
    logout
    $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
    assert('Please enter your username')
end

def test_Access_Control_List_Orders_Normal_User()
    #Check normal user access
    login('corsaire1','corsaire1')
    $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
    assert('Please enter your username')
end

def test_Access_Control_List_Orders_Admin()
    #Check administrator access
    login('admin','password')
    $ie.goto("https://localhost:8443/ispatula/shop/listOrders.do")
    assert('Administrative functions available')
end
```

8.6 Example: Testing for XSS

The example application is susceptible to Cross Site Scripting in the search field, to test this, the script will insert Javascript that opens a new window. Watir will then attempt to attach to the new window, if the window does not exist then an exception will be thrown. Using the unit testing framework's "assert_raises" function, it's possible to check whether this exception is raised or not.

```
def test_XSS_In_Search
        $ie.goto('http://localhost:8080/ispatula/shop/index.do')
        $ie.text_field(:name,
'query').set('<script>window.open("http://localhost:8080/ispatula/help.html"
)</script>')
        $ie.form(:action, /Search.do/).submit
        assert_raises(Watir::Exception::NoMatchingWindowFoundException,
"Search field is susceptible to XSS") {
          ie2 = Watir::IE.attach(:url,
"http://localhost:8080/ispatula/help.html")
        }
    end
```

8.7 Discussion

Security tests integrated into the acceptance test layer do not allow for as granular approach to testing as unit or integration tests, but they do allow for full testing of the external API of the application. Anything that can be tested during a black box security assessment of an application, can be tested for at the acceptance test layer.

External security testers that perform penetration tests and security assessments of applications can use acceptance testing tools to script common vulnerabilities. This is not as useful in detecting vulnerabilities, as it is in documenting and retesting vulnerabilities.

9 Conclusions

Unit testing of functional requirements of applications is already a well-established process in many development methodologies and is strongly emphasised by the Agile methods. If developers are trained in security then existing testing tools and techniques can be used to perform security testing. External security testers can also make use of testing tools to document and automate discovered vulnerabilities.
Security tests could be implemented:

- At the unit test layer by developers to ensure that the security requirements of the application are met, and that all failure and exceptional conditions are tested.
- At the integration test layer by developers and/or QA staff using either an in-container testing or mock object strategy to ensure that security services provided by components and the application server function as required and are free from common security vulnerabilities.
- At the acceptance test layer by security consultants and/or trained QA staff to ensure that the external API of the application is free from security issues. The creation of functional tests can accompany a manual security assessment of the application to provide an automated means of verifying all vulnerabilities discovered. Automated security retests could then be executed at any time with minimal overhead.

Implementing security tests in this manner provides a number of benefits:

- Developers are more aware of security issues and understand how to test for them in their code.
- Security issues are discovered early on when development and debugging is ongoing meaning that issues are addressed rapidly.
- Code is more robust since the tests can be executed at any point to confirm that changes have not adversely affected the security of the application.
- Code auditing is improved since the security standards or policies for web applications can be compared to the tests executed against the application.
- Overall security awareness is improved since it is an integral part of the process rather than an add-on.

Well-tested code that includes security tests results in an end product that is more robust, easier to maintain, naturally self-documenting and more secure.

10 References

MASSOL, Vincent; HUSTED, Ted: *JUnit in Action* (Manning Publications co.; Greenwich, CT, 2004)
MCGRAW, Gary: *Software Security* (Addison-Wesley; 2006)
FOWLER, Martin: "Inversion of Control Containers and the Dependency Injection Pattern" (http://www.martinfowler.com/articles/injection.html)
- "Continuous Integration" (http://www.martinfowler.com/articles/continuousIntegration.html)

jWebUnit (http://jwebunit.sourceforge.net)
WATIR (http://wtr.rubyforge.org)
"Open source testing tools, news and discussion" (http://opensourcetesting.org)