BEYOND SCANNING:
Automated Web Application Security Testing

Stephen de Vries

...ContinuumSecurity...

www.continuumsecurity.net

INTRODUCTION

Security Testing of web applications, both in the form of automated scanning and manual security assessment is poorly integrated into the software development lifecycle (SDL) when compared to other testing activities such as Unit or Integration tests.

Agile methodologies such as Test Driven Development advocate a test first approach, where the tests themselves form the specification for the software. These effectively form an executable specification that grows with the application. If the same approach could be taken with security requirements and testing, then the security domain could also benefit from the advantages of automated integration testing.

Existing testing frameworks and web integration testing tools can be used to create an integration testing framework aimed specifically at security. This would provide security testing templates which could be applied to common web application features, a framework for writing bespoke security tests, and provide integration with existing web scanning tools.

This paper presents such a security testing framework together with the tools required to integrate it with the Burp Suite scanner.

PROBLEMS WITH SECURITY SCANNING

Security Testing is usually performed by external security consultants using broadly defined testing methodologies. The result of which is a report containing a list of security vulnerabilities - or in testing terms: test case failures. The test cases that were executed, and test case successes are not usually included in the report.

There are a number of problems with this approach. Firstly, the software owner doesn't have a clear view on exactly which tests were performed.

Secondly, the security tester can't guarantee that retesting the application will be performed consistently.

And thirdly, since the security tests to perform are selected by the external security tester they can't be used as a security specification for the software before development even starts.

The costs of performing manual retests for security assessments are also much higher than that of running automated integration tests because of the lack of the automation.

AUTOMATED INTEGRATION TESTING

For the purposes of this paper automated integration testing (and integration testing) refers to the practice of testing a web application at the web tier, such as by using Selenium or other browser driver type tools.

Integration tests can be used to define the functional and non-functional requirements for a web application by defining passing and failing scenarios. The tests are often run using popular unit testing frameworks such as JUnit and TestNG. An example of a JUnit test which tests basic login functionality in a web application using Selenium 2 (WebDriver) could be:

Two tests are defined (annotated with "@Test") that describe how the login function should behave when correct and incorrect passwords are used.

```
public void login(String username,String password) {
       driver.get("http://www.ispatula.com:8081/ispatula/shop/signonForm.do");
       driver.findElement(By.name("username")).sendKeys(username);
       driver.findElement(By.name("password")).sendKeys(password);
       driver.findElement(By.name("update")).click();
}

@Test
public void testLoginWorksForCorrectCredentials() {
       login("bob","password");
       assertTrue(driver.getPageSource().contains("Welcome"));
}

@Test
public void testLoginFailsForWrongPassword() {
       login("bob","messerschmidt");
       assertFalse(driver.getPageSource().contains("Welcome"));
}
```

This approach can also be used to perform basic security tests such as:

As more tests are added, they effectively become the functional specification for the

```
@Test
public void testLoginBypassWithSQLinjection() {
       login("bob';--","");
       assertFalse(driver.getPageSource().contains("Welcome"));
}
```

application:

```
testThatLoginWorks()
testThatLoginWithWrongPasswordFails()
testThatLogoutWorks()
testAddItemToShoppingCart()
testCheckoutWithEmptyCart()
testCheckoutWithFullCart()
etc.
```

These integration tests are usually written by the developers themselves, or by specialist testing houses. In cases, they require technical knowledge of a programming language in order to understand the test. This doesn't present a problem in small teams; but in larger teams where there are non-technical business analysts and/or security architects, who don't necessarily understand the programming language used, then the tests alone can't be used to define the application's specification.

BEHAVIOUR DRIVEN DEVELOPMENT (BDD)

BDD is an evolutionary step from Test Driven Development and offers the ability to define the behaviour of an application in a more natural language. BDD is effectively a communication tool, allowing the business and security analysts to define functional and non-functional behaviour in a natural language, while still allowing that behaviour to be captured using automated tests written by developers.

There are a number of BDD frameworks available for different languages, such as EasyB (Groovy), RSpec and Cucumber (Ruby), PyCukes (Python) and JBehave for Java.

BDD WITH JBEHAVE

JBehave uses a similar format to describing specifications (stories) as the popular Cucumber framework for Ruby, i.e. plain text using special keywords to describe the story and code that matches those statements to executable steps in Java. For example,

```
Scenario: Test the login works with valid credentials

Given the login page
When the user logs in with username: bob and password: password
Then the word: Welcome should be on the page

Scenario: Test login fails with wrong password

Given the login page
When the user logs in with username: bob and password: blah
Then the word: Invalid should be on the page

Scenario: Test login can't be bypassed using SQL injection

Given the login page
When the user logs in with username: bob';-- and password: blah
Then the word: Invalid should be on the page
```

the login integration tests listed above could be translated into JBehave as follows:

The words “Given”, “When” and “Then” are JBehave keywords, as well as “And” which can be used to add additional statements. A JBehave story (Specification) is compromised of individual scenarios (tests) . Each step in the scenario is then mapped to Java code, e.g.:

```
@Given("the login page")
public void gotoLoginPage() {
	driver.get("http://www.ispatula.com:8081/ispatula/shop/signonForm.do");
}

@When("the user logs in with username: $username and password: $password")
public void login(String username, String password) {
	driver.findElement(By.name("username")).sendKeys(username);
	driver.findElement(By.name("password")).sendKeys(password);
	driver.findElement(By.name("update")).click();
}

@Then("the word: $matchWord should be on the page")
public void findWordInPage(@Named("matchWord") String matchWord){
	assertThat(driver.getPageSource(),containsString(matchWord));
}
```

The code would be written by a developer, while the specification could be written by a business analyst, or security architect.

DATA TABLES

Looking at the three scenarios listed above, the steps of each scenario is identical - the only difference is the data supplied. This can be used to take advantage of another JBehave feature: tabular data input. Instead of writing three separate scenarios they can

```
Scenario: Test login with valid and invalid data

Given the login page
When the user logs in with <username> and <password>
Then the <matchWord> should be on the page

Examples:
|username     |password     |matchWord    |
|bob          |password     |Welcome      |
|bob          |blah         |Invalid      |
|bob';--      |blah         |Invalid      |
```

be combined into one and a date table used to supply the data:

THE PAGE OBJECT PATTERN

THE NEED FOR ABSTRACTION

Using Selenium WebDriver directly in the tests, as shown below, means that they are tied to the navigation code. A change to the web UI, such as changing the username text field name from “username” to “uname” would mean changing the tests- potentially in more than one place.

```
@When("the user logs in with username: $username and password: $password")
public void login(String username, String password) {
	driver.findElement(By.name("username")).sendKeys(username);
	driver.findElement(By.name("password")).sendKeys(password);
	driver.findElement(By.name("update")).click();
}
```

This presents a problem for test maintenance as well as scalability of the test scripts.

SIMPLE PAGE OBJECT

The Page Object Pattern applies the concept of object oriented design to navigating web pages. Each page is represented by a class, and functions within that page are

```
@When("the user logs in with username: $username and password: $password")
public void login(String username, String password) {
	loginPage.login(username, password);
}
```

represented by methods in the class. This would allow the above code to be changed to:

The details of how the login method is implemented is hidden from the test and can be re-used in other instances of the LoginPage class. A simple example of a Page Object

```
public class LoginPageOrig {
	WebDriver driver;

	public LoginPageOrig(WebDriver driver) {
		this.driver = driver;
	}

	public LoginPageOrig open() {

	driver.get("http://www.ispatula.com:8081/ispatula/shop/signonForm.do");
		return this;
	}

	public void login(String username,String password) {
		driver.findElement(By.name("username")).sendKeys(username);
		driver.findElement(By.name("password")).sendKeys(password);
		driver.findElement(By.name("update")).click();
	}
}
```

could be:

This uses the familiar: driver.get() and driver.findElement() methods to first navigate to the page, and then set the value on certain elements.

ENHANCED PAGE OBJECT

Selenium provides some convenience methods to implementing this pattern- and the BDD framework takes it further by creating a "Page" base class which leads to more

```
public class IspatulaLoginPage extends Page {
	static final String url =
"http://www.ispatula.com:8081/ispatula/shop/signonForm.do";

	WebElement username;
	WebElement password;
	WebElement update;

	public IspatulaLoginPage(WebDriver driver) {
		super(url,driver);
	}

	public void login(String usernameParam,String passwordParam) {
		username.sendKeys(usernameParam);
		password.sendKeys(passwordParam);
		update.click();
	}
}
```

terse page objects:

The WebElements are silently initialised in the parent class's open() method using WebDriver's PageFactory class. If they're not found, then an exception is thrown. By default the field names are matched to the element IDs- and if this fails, then they're matched to element names. If an alternate matching strategy is required, this can be

```
	@FindBy(how = How.LINK_TEXT, using = "click me")
	WebElement username;
```

implemented using annotations:

In addition to the benefit of abstraction, page objects can also be reused by different testing teams. The integration and functional testing team, who ensure that the application behaves as expected- and that it renders correctly in multiple browsers could define the page objects and these could then be re-used by the security or performance testing teams. When the UI or navigation flow is changed, it only has to be changed in the page object itself, without having to change the tests that use it.

ONE MORE LEVEL OF ABSTRACTION

In order to write security testing templates that can easily be applied to different web applications, it's convenient to apply another layer of abstraction and encapsulate specific features into predefined interfaces that are implemented by a single class. To this end, BDD-Security defines the following interfaces:

```
public interface ILogout {
	Page logout();
}
```

More generic features will be added in the future to cater to a wider variety of web applications, e.g.

- ITwoStepLogin for applications that require a secondary authentication step, after the primary.
- IWizardSession for applications that don't have a login, but still maintain a session which is initiated after the user enters some data.

All web applications should derive from a base class, and then implement the

```
public class IspatulaApplication extends WebApplication implements
ILogin,ILogout {

	...
}
```

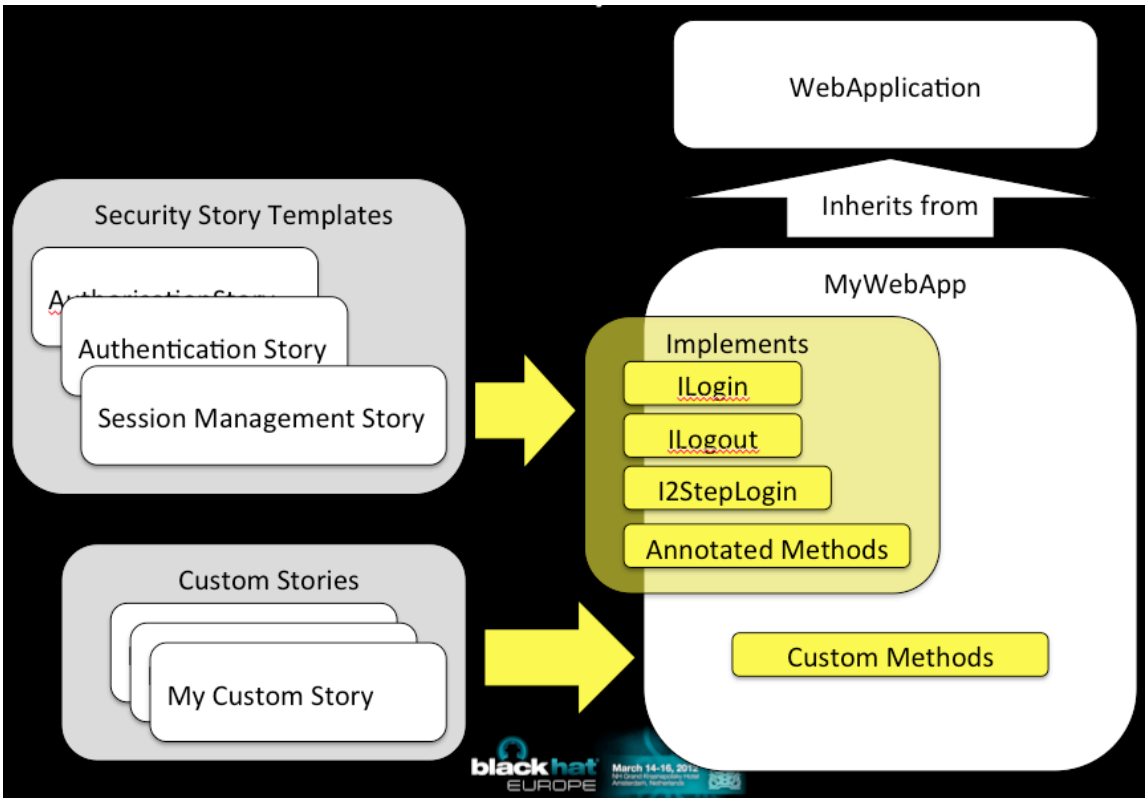
appropriate methods, e.g.:

GENERIC SECURITY TESTS

BDD-SECURITY

BDD-Security is a BDD based testing framework for web applications. It's built on JBehave and includes a number of predefined security specifications for web applications. It was designed in order to make use of re-usable page objects so that the tests should not require modification between different web applications.

ARCHITECTURE



The security story templates make use of the user defined application class which must inherit from the parent WebApplication class. This class should then implement the predefined interfaces as is appropriate for its functionality.

CONFIGURATION

Each project should have a config.xml file defining various configuration settings for the tests:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<web-app>
	<defaultDriver>HtmlUnit</defaultDriver>
	<burpDriver>BurpHtmlUnit</burpDriver>
	<baseUrl>http://www.ispatula.com:8081/ispatula/</baseUrl>
	<class>IspatulaApplication</class>

	<sessionIds>
		<name>JSESSIONID</name>
	</sessionIds>

	<users>
		<user username="bob" password="password">
			<role>user</role>
		</user>
		<user username="alice" password="password">
			<role>user</role>
		</user>
		<user username="admin" password="password">
			<role>user</role>
			<role>admin</role>
		</user>
	</users>

	<burpHost>127.0.0.1</burpHost>
	<burpPort>8080</burpPort>
	<burpWSUrl>http://127.0.0.1:8181/</burpWSUrl>
	<storyUrl>src/main/stories/</storyUrl>
	<reportsDir>reports</reportsDir>
	<latestReportsDir>target/jbehave/</latestReportsDir>

</web-app>
```

The more important user configurable elements are detailed in the following table:

Element	Description
defaultDriver	The default WebDriver implementation to use. BDD-Security has subclassed popular drivers and added HTTP inspection features which are missing from the stock WebDriver implementations. Currently HtmlUnit and Firefox are the only supported options.
burpDriver	The WebDriver used during automated testing through Burp. Currently only HtmlUnit and Firefox are supported using the values: BurpHtmlUnit and BurpFirefox
baseUrl	The base URL for the application under test.
Class	The custom Java class that extends WebApplication.
burpHost	The host running resty-burp

Element	Description
burpPort	Burp's proxy port
burpWSUrl	The WS Url used to communicate to resty-burp
sessionIDs	The cookie names used to store the session IDs.
user	The valid users of the application, currently only supports username and password authentication.
role	The roles that the user belongs to.

In addition to the configuration file, the custom Java class defining the application should also be defined, for example:

```
public class IspatulaApplication extends WebApplication implements ILogin,ILogout {

	public IspatulaApplication(WebDriver driver) {
		super(driver);
	}

	public LoginPage openLoginPage() {
		return new LoginPage(driver).open();
	}

	public Page login(Credentials credentials) {
		 return openLoginPage().login(credentials);
	}

	// Convenience user/pass login method
	public Page login(String username,String password) {
		return login(new UserPassCredentials(username,password));
	}

	public Page logout() {
		return new LogoutPage(driver).open().logout();
	}

	public Page search(String query) {
		return new SearchPage(driver).open().search(query);
	}

	// Check whether the given role is currently logged in.
	// In it's simplest form, it ignores the role.
	public boolean isLoggedIn(String role) {
		try {
			new AccountInfoPage(driver).open();
			return true;
		} catch (Exception e) {
			return false;
		}

	}

}
```

Access Control Tests

BDD-Security also provides a vertical access control test that can be used to test authorisation logic between users in different roles. In order to use this, two items should be configured:

Firstly, the config.xml file should be updated so that users belong to specified roles (see example above).

Secondly, methods that should only be visible to certain roles should be annotated with the "@Roles" annotation in the Java class, for example:

```
public class IspatulaApplication extends WebApplication implements
ILogin,ILogout {
	...
	@Roles({"admin"})
	public ListOrdersPage listOrders() {
		return new ListOrdersPage(driver).open();
	}
	...

}
```

In order for the tests to work, the restricted method must throw an UnexpectedPageException if an unauthorised user attempts to load the page. In the example above, this is implemented by checking the text on the ListOrdersPage. In this simple online shopping applicatoin, if authorised users in the "admin" role view this page then the page contains the text "All Orders". If an unauthorised user in any other role attempts to view the page, then that text will not be present. This behaviour is coded

```
public class ListOrdersPage extends Page {
	...
	public final static String expectedText = "All Orders";

	@Override
	public ListOrdersPage open() {
		return (ListOrdersPage)super.open();
	}

	@Override
	public void verify() {
		if (!getSource().contains((expectedText))) {
			throw new UnexpectedPageException("Did not find the
expected text: "+expectedText);
		}
	}
}
```

into the page object as follows:

The Page.open() method calls verify() in order to verify that it is on the correct page. By overriding the verify() method in the ListOrdersPage, the method will throw an exception if the text is not present.

SECURITY TESTING STORY TEMPLATES

The following predefined security stories are included in BDD-Security:

- Authentication Story: Security specifications that define how the authentication system should behave.
- Authorisation Story: A single scenario that tests vertical access control.
- Session Management Story: A number of scenarios describing how session management should behave.
- Automated Scanning: A number of scenarios using Burp as the scanner

AUTOMATED SCANNING

Automated security scanning is a cost effective way to identify certain types of vulnerabilities in web applications. The Burp Suite testing tool is popular for manual assessments and includes plugin functionality which allows it to be extended.

RESTY-BURP

BDD-Security provides the overall framework for running tests, it would be convenient to be able to execute automated scans from those tests. To this end, Resty-Burp was developed, which is a burp extension that starts up an HTTP server within the Burp instance and offers control of certain Burp features over a REST/JSON API.

This allows Burp to be controlled from many other languages, not just Java. The Resty-Burp software is released with a Java client by default and supports the following functions:

- scan $target
- getPercentComplete
- getIssues
- getProxyHistory
- get/set/update Config
- reset

INTEGRATION WITH BDD-SECURITY

BDD-Security uses the resty-burp service to perform automated scanning. In order to support this, the application class must implement the IScanWorkflow interface and

```
public class IspatulaApplication extends WebApplication implements
IScanWorkflow {
	...
	public void navigateAll() {
		search("hello");
		openLoginPage().login(
			Config.instance().getUsers().getDefaultCredentials("user"));
		new AccountInfoPage(driver).open();
	}
	...
```

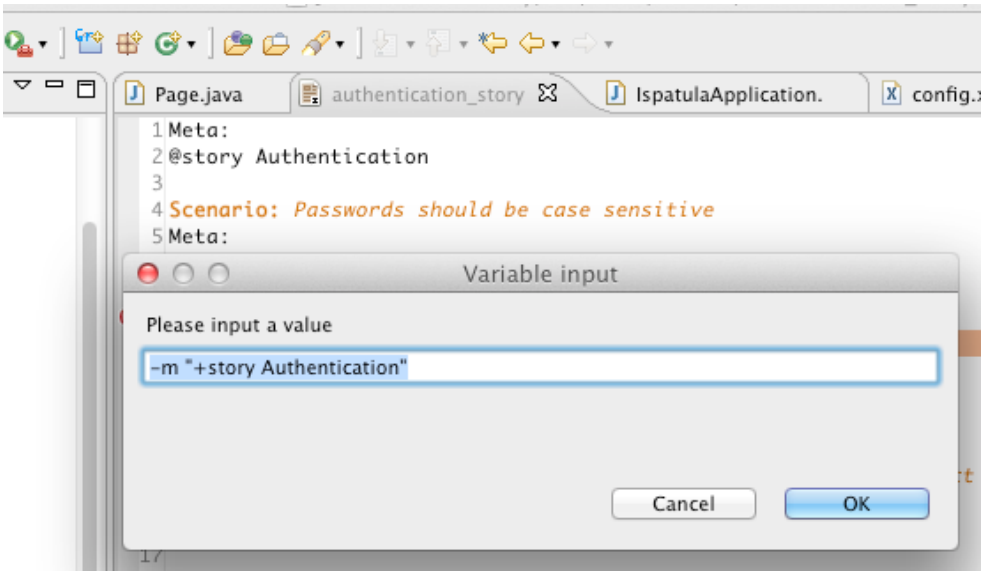
define the navigateAll() method:

The navigation code included in the navigateAll() method is what is used to populate Burp's proxy logs -and therefore the scanner. Only the part of the application included in the navigateAll() method will be scanned by Burp.

The automated scanning performed by BDD-Security is defined in the Automated Scanning Story file.

PUTTING IT ALL TOGETHER

BDD-Security can be run from the command line through maven, or from within an IDE by executing the StoryRunner class. Specific scenarios or entire stories can be included or excluded using meta tags. For example, to only execute the Authentication story:



HTML, XML and TXT reports are generated after each run and copied both to:

bdd-security/target/jbehave

as well as timestamped directory in:

bdd-security/reports

The reports.html file provides a summary of the test execution, how many stories were included/excluded and how many scenarios passed and failed.

Story Reports

Stories		Scenarios				
Name	Excluded	Total	Successful	Pending	Failed	Excluded
AfterStories	0	0	0	0	0	0
Authentication Story	0	7	3	0	4	0

Each story then has its own HTML report describing the scenarios in detail and their results:

authentication_story.story

Meta:
@story Authentication

Scenario: Passwords should be case sensitive

Meta:
@id auth_case

Given the default user logs in:

users.table

Then the user is logged in
When the case of the password is changed
And the user logs in from a fresh login page
Then the user is not logged in

Scenario: Login should be secure against SQL injection bypass attacks in the username field

Meta:
@id auth_sql_bypass2

Examples:

Given the login page
And the default username from: users.table
When an SQL injection <value> is appended to the username
And the user logs in
Then the user is not logged in

value
'--
' OR '1'='1

Example: {value='--}

Given the login page
And the default username from:

users.table

When an SQL injection '-- is appended to the username
And the user logs in
Then the user is not logged in (FAILED)
java.lang.AssertionError:
Expected: is <false>
got: <true>

CONCLUSION

Behaviour Driven Development provides a very useful concept for defining and then executing automated security tests. The improved readability of the specifications make it suitable for use instead of, or in addition to traditional specification documents. By implementing security tests in a BDD framework, they can take advantage of the features of automated testing such as:

- Reliable regression testing
- Low cost retesting
- Integration into existing build tools

One of the biggest advantages is that the security requirements can be defined up front by non-developers, and then interpreted and implemented by development staff.

The Page Object pattern allows testing code to stay free of navigation code, thereby improving maintainability and scalability. Page Objects can be re-used by different teams, but centrally managed and maintained. Changes to the page objects should not require changes to the tests that use those objects.

Resty-Burp provides a standard client-server interface to the Burp Suite tool using a REST/JSON API. This allows automated test scripts to use Burp as a security scanner.

The BDD-Security framework ties these concepts together to provide a generic set of executable security requirements that should be applicable to a wide variety of web applications. Furthermore, it provides a framework to easily extend the existing tests with security tests that are specific to the web application under test.

TOOLS

- BDD-Security: http://www.continuumsecurity.net/bdd-intro.html
- Resty-Burp: http://www.continuumsecurity.net/resty-intro.html

ABOUT THE AUTHOR

Stephen de Vries is an independent Security Consultant specialising in application security and on improving the security practices in software development.

Stephen has worked in the security field since 1998 and has spent the last 12 years focused on Security Assessment and Penetration Testing at Corsaire, KPMG and Internet Security Systems. He was a founding leader of the OWASP Java project and regularly presents talks on secure programming and security testing.