Behaviour-Driven Security Testing

Many penetration tests are poorly documented. Even if we are following standards and checklists, reports often do not state exactly what has been tested. The following technique from software integration testing helps out.

Most of the security testers I know do not have a strong background in software development. Yes, they maybe know how to hack java, reverse-engineer binaries and bypass protection. These skills are often acquired in self-study and I think that is the reason why the security testing process itself often suffers.

Testing like developers

The solution is to test like a developer would. He uses integration tests to check whether everything within the application glues together on the higher level. He spends initial efforts on specifying and writing tests, but the actual testing is then run automatically and is available for the rest of the application's lifecycle.

In counterpart, most penetration testers smash their automated tools against the application first, and then look manually for additional vulnerabilities using experience and gut feeling. If lucky, the client gets a table with findings and a vague list of tested points.

This approach is not really satisfying – not for the client nor for the tester in the long run. Would it not be nice to 'order' a penetration test once, being able to re-run it every time something is changed on the application or infrastructure?

Behaviour-Driven Security Testing

Behaviour-driven development (BDD) is used in agile development. It creates applications defined by their behaviour (what a application should be able to do) instead of low-level unit tests.

BDD uses a business language to specify requirements. The standard here is Gherkin, which is easy to read and therefore a great interface between technology and business. Here is a simple scenario in Gherkin for testing a web application's logout feature.

```
Given I am logged in
 When I logout
 Then I should see "Successfully logged out"
```

If the web application does not show "Successfully logged out", the test fails. As security guys we would probably change the scenario to determine if the session is really terminated from the user's perspective:

```
Given I am logged in
 When I logout
  And I visit "/settings"
 Then I should see "Please login"
```

Gherkin groups multiple scenarios into features. Each feature has its own text file with a header. Most web applications probably have somewhere a feature definition similar to:

```
Feature: Authentication
  In order to access the application
  As an existing user
  I should be able to login and logout

  Scenario: Existing user logs in
  Given I am logged out
   When I login as "user1" with the "correct" password
   Then I should see "Welcome, user1"

   Scenario: ...
```

Implementing the Steps

Of course, a text file alone does nothing yet: That is where Cucumber jumps in. Cucumber is a framework for BDD, based on Ruby and able to interpret Gherkin and perform actions, which are referenced as 'steps'.

A step definition is basically a regular expression that is matched steps within scenarios. Here is an example definition catching the login:

```
When /^I login as "([^"]*)" with the "([^"]*)"
                    password$/ do |username, password|
  # todo: implement actions
  pending
end
```

So when it is time to execute the step as defined in the scenario "Existing user logs in", Cucumber matches the step definition and passes 'username' and 'password' as arguments.

Next is to implement the actions. Selenium is able to automate Firefox, which is practical for testing JavaScript-loaded sites and fun to watch. First we have to make sure the user is on the login page:

```
  visit "/login"
```

Then we fill the fields with our variables

```
  fill_in "username",     :with => username
  fill_in "password",     :with => password
```

And finally click the button:

```
  click_button "Log in"
```

The instructions above assume that there is a text-field with id or name "login" and a button labeled "Log in". If one of these is missing, the test will fail. Of course we could directly submit a POST request, but this might not work in all circumstances.

So this is how our complete step definition looks now:

```
When /^I login as "([^"]*)" with the "([^"]*)"
                    password$/ do |username, password|
  fill_in "username",     :with => username
  fill_in "password",     :with => password
  click_button "Log in"
end
```

The next step is the final state in this scenario, expecting a specific page content (Welcome, user1). The definition might look like this:

```
Then /^I should see "([^"]*)"$/ do |content|
  page.has_content?(content).should be_true
end
```

If the page showing up after login in does not contain "Welcome, user1", the test will fail. It requires some experience to write clean scenarios and steps, but it's no rocket science.

Reporting

A report of a behaviour driven security test contains not only findings and countermeasures, it states exactly what has been tested. A sample test protocol may look similar to this:

```
Feature: Session Login
  User is logged out after a time of inactivity     [ok]
  The session token is stored as cookie             [ok]
  The session token is renewed after login      [FAILED]
  The session token is set with the "HttpOnly" flag  [ok]
  The session token is set with the "secure" flag  [FAILED]
  ....

Feature: Password Change
   User cannot change password without entering the old one [ok]
   User cannot change the password using the old one  [FAILED]
    User "user1" cannot use "" as new password       [ok]
    User "user1" cannot use "user1" as new password  [ok]
    ....

 Feature: Account Settings
   User "user1" cannot update settings for "user2"      [ok]
     - remark: tested by faking the 'id' parameter
  ....
```

Note that even if you do not write any step definitions at all, it is still a lucid way for reporting security tests. It can be used for basic network assessments and vulnerability scans as well, for example:

```
Feature: Basic Network Assessment
  nmap full syn scan should only reveal port 80,443  [ok]
  nmap full udp scan should not reveal any ports  [CANCELED]
  nmap well known udp port scan should not reveal any ports [ok]
  nessus should not report any security issues  [FAILED]
    - output/nessus-report.html
  System should not answer to icmp timestamps       [ok]
  SSLv2 should be disabled on port 443              [ok]
  RSA public key size should be 2048 on port 44     [ok]
  ...
```

With such a report alone, any skilled penetration tester is able to replay the whole test. Did you notice the udp full scan above? It seems it took too long, so the tester had to restart the scan with a reduced port set. This happens often, and here it is documented. A resulting measure could be to instruct the client to review the firewall rules and check if there is no high-range udp port open – better than waiting days for a scan to finish.

Variants

So how does a behaviour-driven approach affect web application audits? It depends on the amount of step definitions. Basically, there are 3 possibilities:

1) No step definitions
   - Testing remains manual.
   - Advantage: Reproducible test, improved documentation
2) Failed-only
   - Step definitions are only created for failed scenarios.
   - Advantage: Easy post-fix testing.
3) Full steps
   - Step definitions are created for all tested scenarios.
   - Advantage: Tests can be run automatically anytime.

All three variants deliver a full scenario list, including the result of each scenario. 'Failed-only' should be considered if it is already planned to re-test the application after implementing the measures from the first run.

Full step definitions are connected with more initial efforts and ask for strategic thinking. Assume a brand-new web application that should be tested for security: Let's set the lifecycle to 10 years, with an security testing interval of two years. This makes five security audits. Assume $5.000 per test, the client would pay $25.000 or $2.500 per year for testing the security.

Now to behaviour-driven testing: Assume the initial effort for creating full step definitions is 1:2, so the first test costs $10.000. As web applications are agile, we have to consider the refactoring of the tests and maybe new features with $1.000 per year. This example lowers the total costs for security testing to $1.900 per year while simultaneously allowing the client to run a full scale security audit whenever he wants: A no-brainer for anyone who has a real interest in keeping applications secure.

Conclusion

Penetration testing is still a young, sometimes chaotic discipline, while software engineering is more sophisticated in standards and organization. But security testers and software engineers always have learned from each other, a real symbiosis that results in something good: Software that is more secure.

Tutorial

This tutorial will show you how to create a simple security test using cucumber.

Preparation

Before you start, you need a working Ruby environment. If you are new to Ruby and have no environment yet, I recommend setting it up using Ruby Version Manager (see "links").

Our tutorial needs besides of cucumber some additional gems. Install them with the following commands:

```
$ gem install cucumber
$ gem install rspec
$ gem install webrat
$ gem install mechanize
```

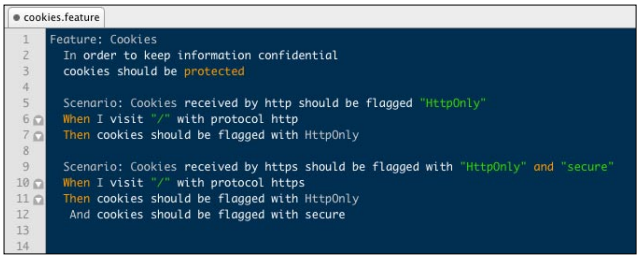

Figure 1. *A Feature Definition in Gherkin*

Create a new directory with the following subdirectories:

```
$ mkdir -p features/step_definitions
$ mkdir features/support
```

You should be able to run cucumber now from the current directory and get an output like:

```
$ cucumber
0 scenarios
0 steps
0m0.000s
```

Before writing a feature, our testing environment needs some setup, which is usually done in `features/support/env.rb`. Create this with the following contents:

```
# features/support/env.rb:
require 'rubygems'
require 'rspec'
require 'webrat'
require 'mechanize'

Webrat.configure do |config|
  config.mode = :mechanize
end

World(Webrat::Methods)
World(Webrat::Matchers)

HOST = "localhost"
```

Replace `localhost` with an IP or hostname you are allowed to test on. When running cucumber again, you should still get the same output as before, without errors.

The First Feature

Now it is time to write our first feature. We want to examine cookies, so create the feature file `./features/cookies.feature` as shown in Figure 1.

As you see, we are testing whether cookies are set with the 'HttpOnly' flag to protect them from JavaScript access. We also want to know if the 'secure' flag is set when the site is visited using an encrypted channel.

Run cucumber again and see that there is already something happening: You are getting error messages regarding undefined steps, including snippets for missing definitions, see Figure 2.

Before blindly copying and pasting, we should think a moment about where to place these definitions. The "When I visit something with protocol http" step will be surely used in other features, so it might be a good idea to place this one in a 'shared' step file and edit it as follows:

```
$ cucumber
Feature: Cookies
  In order to keep information confidential
  cookies should be protected

  Scenario: Cookies received by http should be flagged "HttpOnly" # features/cookies.feature:5
    When I visit "/" with protocol http                           # features/cookies.feature:6
    Then cookies should be flagged with HttpOnly                  # features/cookies.feature:7

  Scenario: Cookies received by https should be flagged with "HttpOnly" and "secure" # features/cookies.feature:9
    When I visit "/" with protocol https                                             # features/cookies.feature:10
    Then cookies should be flagged with HttpOnly                                     # features/cookies.feature:11
    And cookies should be flagged with secure                                        # features/cookies.feature:12

2 scenarios (2 undefined)
5 steps (5 undefined)
0m0.004s

You can implement step definitions for undefined steps with these snippets:

When /^I visit "([^"]*)" with protocol http$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^cookies should be flagged with HttpOnly$/ do
  pending # express the regexp above with the code you wish you had
end

When /^I visit "([^"]*)" with protocol https$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^cookies should be flagged with secure$/ do
  pending # express the regexp above with the code you wish you had
end

$
```

Figure 2. *Cucumber Showing Errors when Steps Are Not Defined*

cookies.feature | cookies_steps.rb | common_steps.rb

```
When /^I visit "([^"]*)" with protocol (.*)$/ do |path, protocol|
  @page = request_page("#{protocol}://#{HOST}#{path}", 'get', nil)
end
```

Figure 3. *A Step Definition for Cucumber*

cookies.feature | cookies_steps.rb | common_steps.rb

```
Then /^cookies should be flagged with (.*)$/ do |flag|
  @page.header.each do |key, value|
    if key.downcase == "set-cookie"
      value.include?(flag).should == true
    end
  end
end
```

Figure 4. *This Step Definition Is Checking Cookies for Flags*

Links

- Cucumber: *http://cukes.info/*
- Selenium: *http://seleniumhq.org/*
- Ruby Version Manager: *http://beginrescueend.com/*
- Tutorial on github: *https://github.com/symontech/entrance*

webrat's `request_page` method and storing the response into the `@page` variable for access in succeeding steps, see Figure 3.

The definition for reading the cookie flags contains our test, see Figure 4.

```
$ cucumber
Feature: Cookies
  In order to keep information confidential
  cookies should be protected

  Scenario: Cookies received by http should be flagged "HttpOnly" # features/cookies.feature:5
    When I visit "/" with protocol http                           # features/step_definitions/common_steps.rb:1
    Then cookies should be flagged with HttpOnly                  # features/step_definitions/cookies_steps.rb:1

  Scenario: Cookies received by https should be flagged with "HttpOnly" and "secure" # features/cookies.feature:9
    When I visit "/" with protocol https                                             # features/step_definitions/common_steps.rb:1
    Then cookies should be flagged with HttpOnly                                     # features/step_definitions/cookies_steps.rb:1
    And cookies should be flagged with secure                                        # features/step_definitions/cookies_steps.rb:1
      expected: true
           got: false (using ==) (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/cookies_steps.rb:4:in `block (2 levels) in <top (required)>'
      ./features/step_definitions/cookies_steps.rb:2:in `each'
      ./features/step_definitions/cookies_steps.rb:2:in `/^cookies should be flagged with (.*)$/'
      features/cookies.feature:12:in `And cookies should be flagged with secure'

Failing Scenarios:
cucumber features/cookies.feature:9 # Scenario: Cookies received by https should be flagged with "HttpOnly" and "secure"

2 scenarios (1 failed, 1 passed)
5 steps (1 failed, 4 passed)
0m0.876s
$
```

Figure 5. *Sample Cucumber Output with our Implemented Steps*

```
# features/step_definitions/common_steps.rb:

When /^I visit "([^"]*)" with protocol (.*)$/ do |path,
                    protocol|
  pending
end
```

As you see, the protocol is now an argument too, which makes the third snippet obsolete. We do the same with the remaining snippets:

```
# features/step_definitions/cookies_steps.rb:

Then /^cookies should be flagged with (.*)$/ do |flag|
  pending
end
```

The 'pending' keyword is a reminder for us and avoids accidentally passing incomplete tests. It is a good habit to use this keyword by default, until a step is complete.

The next task is to replace all the 'pendings' with real code. We start with the shared step, which is calling

The http header section is iterated, while looking for 'set-cookie'. If it is found, the test looks fore the presence of the flag. The solution above might not be ideal yet, because multiple cookies and cookies with the same name or contents as the flag might slip through.

Running the test

Now you are ready to run the test on your web server. You might get a similar output as shown in Figure 5.

The code from this tutorial is also available on github. Please feel free to contribute.

SIMON WEPFER

Simon Wepfer is CEO of netsense; an independent swiss IT security company specialized in vulnerability management, http://netsense.ch/en