Hands-On Gauntlt

Security Testing for Developers

James Wickett

Hands-on Gauntlt: Security Testing for Developers

James Wickett

This book is for sale at http://leanpub.com/hands-on-gauntlt

This version was published on 2015-02-09

Tweet This Book!

Please help James Wickett by spreading the word about this book on Twitter!

The suggested hashtag for this book is #gauntltbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#gauntltbook

Contents

Hello World

Rather than break the tradition that has been handed down to us from the software elders, lets go ahead and start with a simple hello world attack with gauntlt. The attack is not something you would actually run, however it helps you get familiar with the gauntlt syntand flow.

Take a moment and read through our `hello_world.attack` to get an understanding of what this attack is doing.

Hello World - code/hello_world/first.attack

```
@final
Feature: hello world with gauntlt using the generic command line attack
  Scenario: Check for the existence of the root user in /etc/passwd
    When I launch a "generic" attack with:
      """
      cat /etc/passwd
      """
    Then the output should contain:
      """
      root
      """
```

Since this is our first attack we are running, we are going to break it down line by line.

Intro to Tags

On the first line, we see `@final`. Anything with the `@` is a tag in gauntlt and it is one way to organize your attacks. Here we are tagging it with the word `@final` for purposes of the book. As the book was in development the attacks were tagged `@draft` and then promoted to `@final` when ready.

Tags are great and there are a lot of different ways to use them. You can use multiple tags as well, and you might want to have your first line of your attack file be `@web @prod @slow`. Feel free to mix in tags that have organizational meaning like `@prod` with tags that describe attack type, like `@web`. It is very common to want add tags describing the types of attacks you are running and how long they might take. Organizing your attacks with tags like `@slow` and `@reallyslow` are helpful too. Later when we are adding gauntlt to a build pipeline, we can choose to only run attacks that are not tagged as `@really_slow`.

Tags in Gauntlt

Gauntlt has two tags that are available by default: `@slow` and `@reallyslow`. These allow the attack to run for 30 seconds and 5 minutes respectively. Use these with caution as long running attacks make for bad citizens of the Continuous Integration and Continuous Delivery pipeline. One of the best pieces of advice you can follow is the coffee rule, if it takes longer to run your tests than it takes to get a cup of coffee, then you need to change your tests.

Now Introducing Feature

The next line of our example reads:

```
Feature: hello world with gauntlt using the generic command line attack
```

Each attack should say what it is doing. We use the `Feature` keyword to express what the attack is doing. This may seem a little weird at first, but take this space to give a good discription to what you are doing in this attack. You will have the chance to be more expressive later in your `Scenario` statements but don't neglect your `Feature` statement.

What makes a good `Feature` statement?

A `Feature` statement tells the reader what is being attacked. Later when doing reports the scenarios roll up under the feature so it is best to make sure that your `Feature` statement is:

- Descriptive to the purpose of the attack
- States what is being attacked or tested
- Decribes what tooling it is using to do the attacks

Examples of a few `Feature` statements:

- `Feature: checking for XSS on the login page`
- `Feature: fuzzing the web server for forceful browsing using the common wordlist dirb.`
- `Feature: testing for SQLi on our forum signup page using sqlmap`

Notice that not all of the examples say what attack tooling they are using (XSS), but many of them do. Sometimes you may be checking for XSS using multiple tools so don't get hung up on that part. The important part here is to convey meaning inside your organization and your dev, ops and security teams.

The Scenario

While you can think of `Feature` statements of the top-level description, `Scenario` statements are meant to encapsulate what each especfic attack is doing.

In our example we see:

```
Scenario: Check for the existence of the root user in /etc/passwd
```

This statement tells the reader clearly what is happening. The importance of a good scenario description cant be overstated, this is where you can facilitate collaboration between security, dev, and ops teams.

Take for example this attack code which we will use in a later chapter:

```
arachni --modules=xss --depth=1 --link-count=10 --auto-redundant=2 <url>
```

To a person who has used arachni before, it is pretty easy to see this is a simple single page scan of one URL using all of the XSS modules built into arachin. However to anyone else in the organization it is completely meaningless.

If you were to pick this up without any prior knowledge you would have to ask, "What is arachni?" Shortly thereafter, you would be reading arachni documentation about `--modules`, `--depth` and all the other flags being passed to it. Multiply that experience by all security attack tooling and you can see where there can be a huge learning curve. And this isnt a learning curve that is useful other than to understand what is happening in the attack!

Use `Scenario` statements to clearly say what is being attacked and what is in scope.

Back to our example:

```
Scenario: Check for the existence of the root user in /etc/passwd
```

This clearly explains what is in scope and how we are attacking or testing it.

We will cover scenarios later, but one other important aspect of scenarios is that they are run independent of each other. Gauntlt runs them as if they are the only attack and cleans up any output between scenarios. There are exceptions and ways to get around this if you need to, but you should think of a scenario as a wholly contained attack.

Given, When, Then

At the heart of gauntlt there is a simple structure of `Given`, `When`, `Then`. This is Gherkin syntax and is core to cucumber which gauntlt uses as the execution engine for the attacks.

- Given: the setup and pre-conditions of the attack
- When: the execution of the attack tooling
- Then: the expected output from the attack

```
When I launch a "generic" attack with:
  """
  cat /etc/passwd
  """
Then the output should contain:
  """
  root
  """
```

In our example we skipped the `Given` step and moved straight into `When` and `Then` steps. Our example is using the generic attack adapter. This can be used for anything being run on the command line.

We will get more into `Given`, `When`, `Then` and how gauntlt runs the steps in future chapters. The point here is to introduce them and see that they are easily readable. You can do all sorts of parsing with the `Then` step using a mix of exact matching, conditional logic and regex parsing.

Run gauntlt

Lets run the first attack.

```
gauntlt ./code/hello_world/first.attack
```

If all succeeded, you should see output matching this:

Hello World - code/hello_world/first.attack.out

```
@final
Feature: hello world with gauntlt using the generic command line attack

  Scenario: Check for the existence of the root user in /etc/passwd # manuscript\
/code/hello_world/first.attack:3
    When I launch a "generic" attack with:                          # gauntlt-1.\
0.10/lib/gauntlt/attack_adapters/generic.rb:1
      """
      cat /etc/passwd
      """
    Then the output should contain:                                 # aruba-0.5.\
4/lib/aruba/cucumber.rb:147
      """
      root
      """

1 scenario (1 passed)
2 steps (2 passed)
0m0.010s
```

In the next chapter we will look at running a more meaningful attack.