PentesterLab

Finding Needles in Haystacks

An introduction to code review

Louis Nyffenegger
@PentesterLab
louis@pentesterlab.com

Luke Jahnke
@BitcoinCTF



Agenda



01 Introduction

02 Code review

03 Lab 1: A PHP Application

04 Lab 2: A Golang Application

05 Lab 3: A Ruby on Rails Application

06 Conclusion

About Luke



Security Engineers:

- Pentester/Code Reviewer/Security consultant… at Elttam (https://www.elttam.com.au/
- Run one of the hardest web CTF: BitcoinCTF



BitcoinCTF:

- Challenges must be solved sequentially
- Prize is paid in Bitcoins
- @bitcoinctf on Twitter

About Louis

Security Engineers:

- Pentester/Code Reviewer/Security consultant/Security architect
- Run a website to help people learn security



PentesterLab:

- Platform to learn web security/penetration testing
- 100% Hands-on
- Available for individuals (free and PRO) and enterprises

Code Review



Why would you want to do code review?



- Can be faster than penetration testing
- Compliance (PCI 6.3.2)
- You're tired of penetration testing
- You want to find better bugs
- You want to check if some code is back-doored
- You want to write an exploit for a bug

Multiple ways to go about code review



- Grep for bugs
- Follow user input
- Read some random code
- Read all the code
- Check one functionality at a time (login, password reset…)

Grep for Bugs



- Just search for:
  - Potential issues
  - Dangerous functions
  - “FIXME”, “HACK”
- grep and find are your friends:

```
$ grep -R ‘system\(\$_’ *

$ find . -name \*.php -exec grep -Hn 'exec' {} \;
```

Grep for Bugs



- Pros:
  - Super fast
  - Good way to find low hanging fruits
- Cons:
  - You end up using very complex regular expressions
  - You need to know all the dangerous functions
  - Very low coverage

Follow user inputs



- Find all the routes/URI available
- Find all the way to provide data to the application (example in PHP):
  - $_POST / $_GET / $_REQUEST
  - $_COOKIE / $_SERVER
  - Data coming from the database
  - Data read from a file or from a cache
  - …

Follow user inputs



- Pros:
  - Good coverage
- Cons:
  - Need a good understanding of the framework/language
  - You find yourself reading the same code again and again

By functionality



- Pick one functionality:
  - “Password reset”
  - “Database access”
  - “Authentication”
- And review all the code associated with it

By functionality



- Pros:
  - Excellent coverage for the functionalities reviewed
  - Improve your pentest-FU
  - Work especially well across multiple applications
- Cons:
  - No coverage for the functionalities that didn’t get reviewed

Read everything



- Just start reading the code one file at the time
- Don’t try to find vulnerabilities, try to find weaknesses
- Keep (really good) notes

Read everything



- Pros:
  - Excellent coverage
  - Improve your pentest-FU
  - Work especially well across multiple applications
- Cons:
  - Hard to keep track of everything
  - Very time consuming

Top Down vs Bottom up



- Both approaches have Pro and Cons
- You can mix them as well
- Top down is probably easier for large applications
- Try both :)



What should you look for?



- Everything!
- If you don’t know a function/class/method:
  - Google it
  - Test the behaviour
- It’s going to take time (especially at the beginning):
  - The more code you review, the easier it gets

Testing a function's behaviour



- Just copy the snippet you are reviewing
- Run it (Locally/Docker/VM)
  https://medium.com/@PentesterLab/use-docker-for-your-pentesting-labs-879fe9feeca8
- Try to find some edge cases
- Save this example for the next time you see this function
- For PHP, you can use 'php -a'

Testing a function's behaviour



```
🍺 koriander ~/code/java % ls
OneTimePassword.class   TestFile.java        TestRemove.class
OneTimePassword.java    TestHashMap.class    TestRemove.java
ScriptEngine            TestHashMap.java     TestToken.class
Session.class           TestJSF.class        TestToken.java
Session.java            TestJSF.java         TestTrustFact.class
Test.class              TestLong.class       TestTrustFact.java
Test.java               TestLong.java        TestURL.class
TestAnnotation.java     TestMatch.class      TestURL.java
TestBigInt.class        TestMatch.java       TestUUID.class
TestBigInt.java         TestMatch2.class     TestUUID.java
TestCanonical.class     TestMatch2.java      int_no_sqli
TestCanonical.java      TestPattern.class    processbuilder
TestEquals.class        TestPattern.java     ssl
TestEquals.java         TestRandom.class
TestFile.class          TestRandom.java
```

What should you look for?



- Weird behaviour
- Differences between 2 functions/methods/classes
- Security checks already in place
- Comparison and conditions (if/else)
- Complexity
- Regular expressions/string matching
- What is missing?

What now?



- Three applications riddled with bugs:
  - Simple PHP application
  - Very simple Golang application
  - More complex Ruby application
- The goal is to start on your own to try to find potential issues and work from them

What now?



- No silly questions (we are all here to learn)
- Work in team if you want/like
- Don’t get stuck: if you cannot understand something:
  - Google the function/method/…
  - Ask us
  - Or move to the next line of code

What now?



- 3 stages:
  - You search on your own with no help
  - We give you the list of bugs
  - We do a walkthrough
- Today's goal is to make you confident you can review code and find cool bugs

Lab 1:
PHP Application



The application



Dropbox on a budget written in PHP

The application allows users to

- Store a file
- Retrieve a file

Language	files	blank	comment	code
CSS	2	676	11	3973
PHP	11	48	5	287
SQL	1	5	0	5
SUM:	14	729	16	4265

The application

- Get the code:
  - git clone https://github.com/PentesterLab/cr/
  - wget https://github.com/PentesterLab/cr/archive/master.zip
- Very simple application with a dozen security issues



List of weaknesses



- Hardcoded credentials
- Information leak
- Weak password hashing mechanism
- Cross-Site Scripting
- No CSRF protection

List of weaknesses



- Crypto issue
- Signature bypass
- Authentication bypass
- Authorisation bypass
- Remote Code Execution
- …

Lab 2:
Golang Application



The application



A simple multi-factor authentication

Two main API functions:

- Generate code and send code via SMS
- Verify code



The application

- Get the code:
  - git clone https://github.com/PentesterLab/cr-go/
  - wget https://github.com/PentesterLab/cr-go/archive/master.zip
- Very simple application with a dozen security issues



List of weaknesses



- Weak Random Generator
- Random Generator seeded based on current time
- Information Leak via JWT
- Using unverified JWT token
- No expiry on JWT token

List of weaknesses



- Information Leak via /_status endpoint
- Time Constant comparison of the one-time code
- Weak CORS policy
- No limit on the number of attempts
- No limit on the number of attempts due to signed sessions

Lab 3:
Ruby-on-Rails application



Ruby-on-Rails - Introduction



Commonly used web framework:

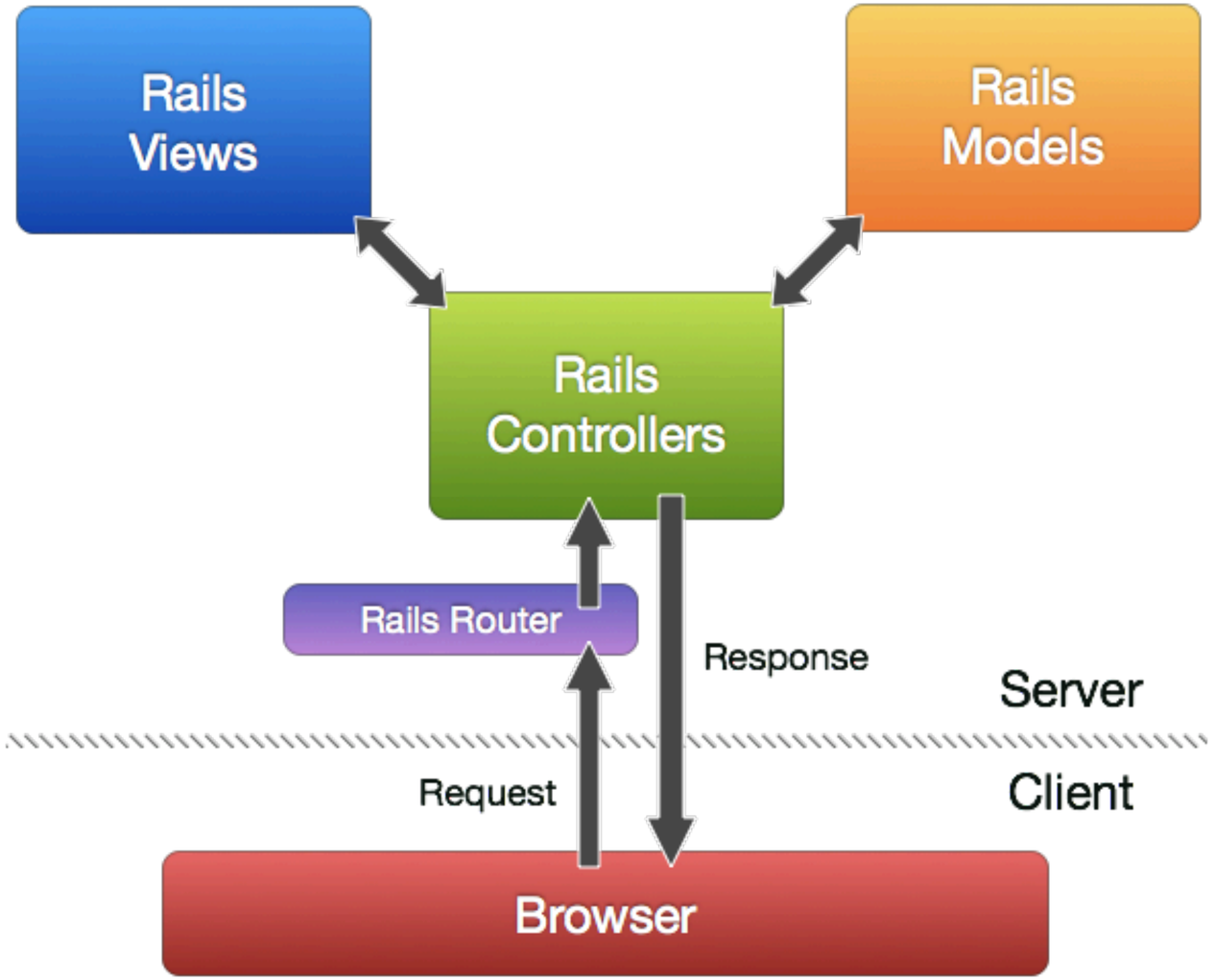
- Used a lot by startups (especially in early stage)
- Used by Github/Shopify/Gitlab/…
- A lot of very interesting vulnerabilities

Barrier to entry:

- You need to understand the framework to get efficient at finding issues

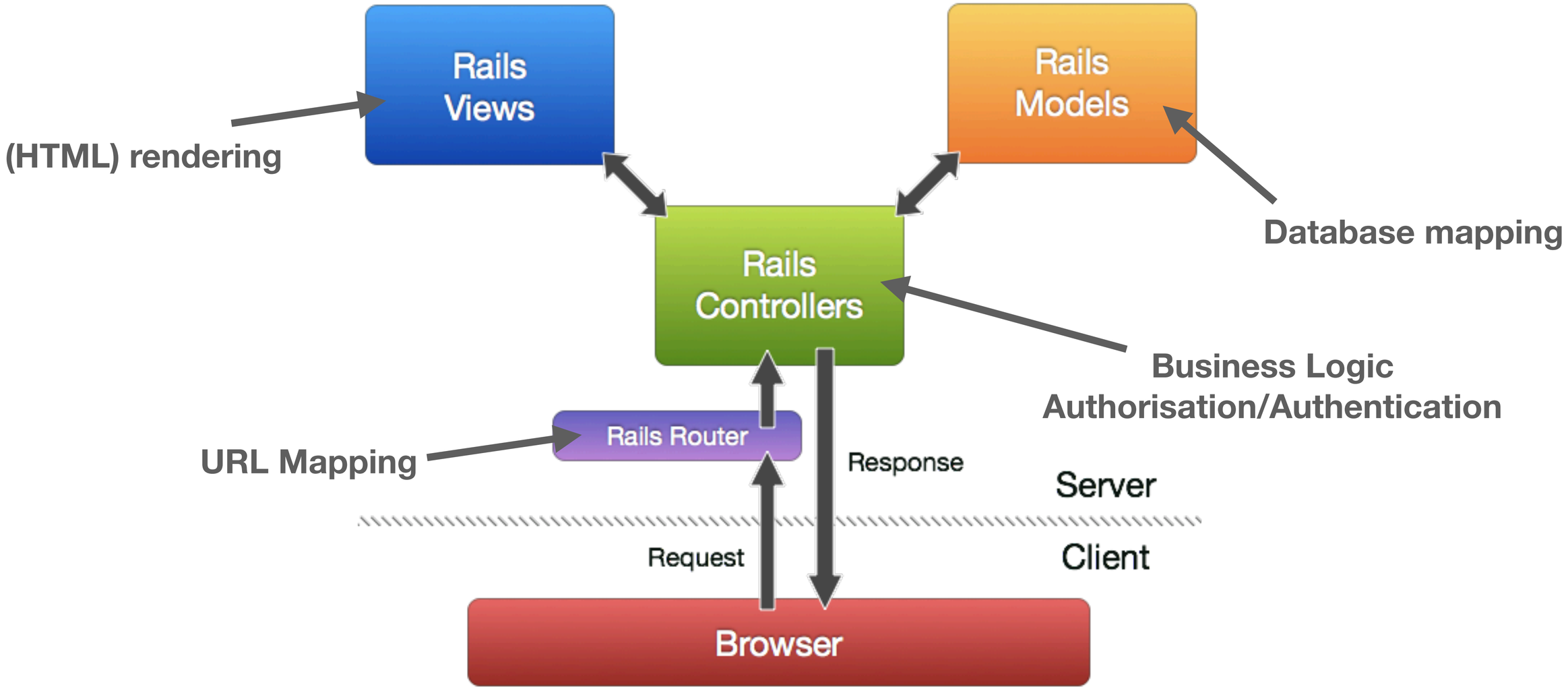


Ruby-on-Rails - MVC



Source: http://blog.ifuturz.com/ruby-on-rails/ruby-on-rails-mvc-learn-with-fun.html

Ruby-on-Rails - MVC



Source: http://blog.ifuturz.com/ruby-on-rails/ruby-on-rails-mvc-learn-with-fun.html

Ruby-on-Rails - Code Structure



/

/config/routes.rb

/app/controllers/

/app/views/

/app/models/



Ruby-on-Rails - Code Structure



/app/models/test.rb

/app/controllers/tests_controller.rb

/app/views/tests/index.html.erb

/app/views/tests/show.html.erb

/app/views/tests/show.json.jbuilder

/app/views/tests/…



Ruby-on-Rails - example of routes.rb



```
Rails.application.routes.draw do
  resources :users

  get 'register'  => 'users#register', as:
'register'
  post 'register'  => 'users#post_register',
as: ‘post_register'
  […]
end
```

See: http://guides.rubyonrails.org/routing.html

Ruby-on-Rails - example of a model



```
class Todo < ApplicationRecord
  belongs_to :user
end
```



Ruby-on-Rails - database structure



```
ActiveRecord::Schema.define(version: 2018_07_12_063817) do
  create_table "todos", options: "ENGINE=InnoDB DEFAULT
CHARSET=utf8", force: :cascade do |t|
    t.string "title"
    t.string "data"
    t.bigint "user_id"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["user_id"], name: "index_todos_on_user_id"
  end
```



Ruby-on-Rails - Example of a controller



```
class ApplicationController < ActionController::Base
  # GET /todos/1
  # GET /todos/1.json
  def show
  end

  # GET /todos/new
  def new
    @todo = Todo.new
  end
[…]
end
```



Ruby-on-Rails - ApplicationController



```
class ApplicationController < ActionController::Base
  before_action :myfunction
  […]
  def myfunction
  end
end
```



Ruby-on-Rails - example of a view



```
<div class="container">
<p id="notice"><%= notice %></p>
<h1>Todos</h1>

<table class="table">
  <tbody>
    <% @todos.each do |todo| %>
      <tr> <td><%= todo.title %></td> <td><%=
todo.data %></td> </tr>
    <% end %>
  </tbody>
</table>
```

Ruby-on-Rails - Filters



before_action (formerly before_filter):

- Happens before an action takes place
- Great to enforce permissions or authentication

after_action (formerly after_filter):

- Happens after an action takes place
- Rarely used (sometimes for logging or queuing data)

Ruby-on-Rails - Filters



You can also disable a before_action (example):

```
skip_before_action :verify_authenticity_token,
                   :only => [:my_csrfable_function]
```

Ruby-on-Rails - configurations



Database configuration in /config/database.yml

By default sessions are signed. The secret is stored in config/initializers/secret_token.rb

Rails 5.2 introduces encrypted credentials



Ruby-on-Rails - Scaffolding



Rails allows developer to generate a lot of code automatically using scaffolding (views/model/controller):

```
$ rails generate scaffold test
      name:string order:integer
```



Ruby-on-Rails - Scaffolding



Scaffolding generates a lot of code that can potentially have security impacts:

- Information leak via HTML/JSON mapping
- Unnecessary actions
- Unauthenticated actions

Ruby-on-Rails - Brakeman



A static analysis tool (SAST) for Ruby on Rails application:

https://brakemanscanner.org/

Amazing coverage for low hanging fruits and code hygiene



Ruby-on-Rails - Getting started



- Start with the db/schema.rb file to get an idea of the data/application
- Check the config/routes.rb to see what methods are available
- Focus on the controllers for this review
- Look at the models&views as well



The application

- Get the code:
  - git clone https://github.com/PentesterLab/cr-rails/
  - wget https://github.com/PentesterLab/cr-rails/archive/master.zip
  - Very simple application with a dozen security issues



List of weaknesses



- Hardcoded and trivial credentials
- Using high-privilege account to connect to the database
- Password leak
- Weak algorithm used for password storage
- CSRF to disable MFA
- Insecure connection to MFA backend

List of weaknesses



- Mass-Assignment on users to become admin
- No unique constraint on email
- Authentication bypass
- Host injection in the link for the password reset
- Password reset bypass

Conclusion



Recommendations



- Keep reviewing code:
  - Even just to understand it
  - (Hacking) Tools you use everyday?
- Few hundred lines a day, keeps boredom away
- The more you do the easier it gets



Conclusion



- Code Review is fun and simple :)
- Keep practicing
- Make sure you checkout:
  - PentesterLab and PentesterLab PRO
  - BitcoinCTF



THANKS

FOR YOUR TIME !

Any questions?

louis@pentesterlab.com / PentesterLab.com / @PentesterLab