

LEVERAGING CONVENTION OVER CONFIGURATION FOR STATIC ANALYSIS IN DYNAMIC LANGUAGES

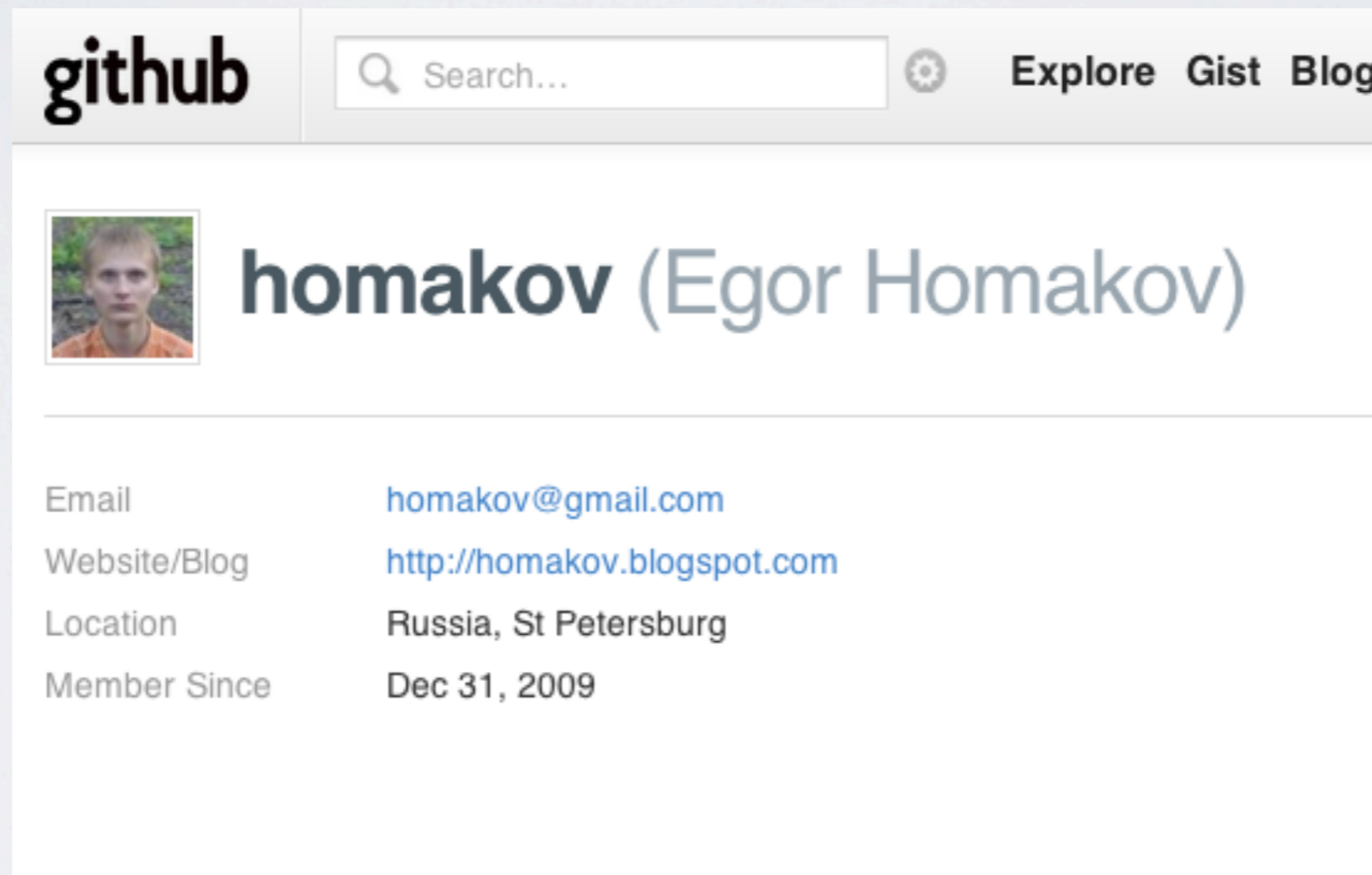
David Worth

dave@highgroove.com (email) - @highgroovedave (twitter)


highgroove
studios



Or why it's ok to write simple frameworks for complicated tasks



github [Explore](#) [Gist](#) [Blog](#)

 **homakov** (Egor Homakov)

Email	homakov@gmail.com
Website/Blog	http://homakov.blogspot.com
Location	Russia, St Petersburg
Member Since	Dec 31, 2009

ABOUT ME



- Developer and Security Engineer for Highgroove Studios in Atlanta GA, USA
- Background in computer science, mathematics, and systems security (not web!)
- Contributor to the Brakeman project since 2011

THIS IS NOT A RAILS TALK!

- Really we're not talking about Ruby on Rails
- Rails does a lot right that helps us to secure our applications
- Those same things help us to automatically reason about them
- They are essentially well codified calling conventions hidden under the hood
- This can be done in any language!

“CONVENTION”

- sensible defaults in behavior and settings with an explicit “right way” for common idioms
- allows developers to think less about easy things and more about hard things (domain/business logic)
- makes hard things easier by guiding the way, makes impossibly complicated things tractable

... OVER “CONFIGURATION”

- Contrast convention against configuration of complicated web frameworks (think Java’s Struts or Python’s Django)
- Fewer knobs to turn making the “right way” more obvious and easy
- Only dig deep, turn knobs, do it yourself if you’ve exhausted the provided tools

RUBY



Ruby
A Programmer's Best Friend

- Object-Oriented “dynamic” language with inspiration from SmallTalk, Perl, Lisp, ...
- “duck typing” is preferred to strict typing
- flexible syntax allows for easy and powerful Domain Specific Languages (DSLs)

RUBY ON RAILS



- A convention-over-configuration (CoC) MVC framework for applications working over HTTP/S
- Not just for “web” apps anymore (thinks RESTful APIs)
- An entire flexible stack allowing hooking at instrumenting at any number of levels
- This stack also allows a separation of concerns for auditors

RAILS UNDER THE HOOD

- Rails is a Ruby domain-specific language for writing HTTP/S exposed applications
- “Magic” methods are always clever, but also consistent, method calls
- Conventions are always preferred over being “clever”

STATIC ANALYSIS

- The process (art) of reasoning about a program without executing it
- The goal may be proving correctness, optimization, or finding potential bugs
- May be used for analyzing malware as part of your Reverse Engineering stack
- In dynamic languages, operate at the same level as a compiler: the Abstract Syntax Tree (AST)

STATIC ANALYSIS OF RUBY



- Use Racc to generate a Ruby grammar
- Use RubyParser to use that grammar to parse a given Ruby program into an AST
- Use SexpProcessor or custom code to analyze specific “interesting” parts of the AST.

STATIC ANALYSIS OF RUBY



Ruby
A Programmer's Best Friend

- Or use Ripper for Ruby 1.9 (part of Ruby core) - s-expression generation is explicitly listed as “Experimental”
- Rubinius Melbourne (and Melbourne 19) gems

STATIC ANALYSIS OF PYTHON



- Python supports static analysis via the Python Language Tools in StdLib
- Tools using it today: PyChecker, PyLint, PyFlakes
- No security specific tools

WEB SECURITY

- “Trivial” - Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), SQL-Injection (SQLi)
- Non-trivial - application specific relying on the specific combination of technologies in the app to exploit (“chained” exploits)
- Other - OWASP’s list of attacks includes as many systems issues as “web”



WEB APPLICATION VULNERABILITY SCANNERS

Dynamic / Runtime

- SkipFish - Michael Zalewski (Google)
- w3af - Web Application Attack and Audit Framework (Andrés Riancho)
- Retina (eEye Digital Security)

Static Analysis

- Brakeman - Justin Collins, Neil Matatall (Twitter), Many Others (myself included)
- Scanny - OpenSUSE team

TRACKING DOWN AN XSS

- XSS is always “trivial”, but spotting one during an audit might not be

SUBTLE XSS

```
class UsersController < ApplicationController
  before_filter :get_users

  def index; end

  private
  def get_users
    @users = User.all #where User is a model
  end
end
```

SUBTLE XSS (con't)

```
<%= index.html.erb %>
<%= render partial: "user", collection: @users %>

<%= _user.html.erb %>
Stuff about the user...
<%= render partial: 'bio', locals: { user_bio: raw(user.bio) } %>

<%= _bio.html.erb %>
Awesome bio: <%= user_bio %>
```

SUBTLE XSS (con't)

Brakeman Results:*

View Warnings:

Template	Warning Type	Message
users/_bio	XSS	Unescaped model attribute near line 1: (Unresolved Model).new.bio

* edited for screen space

SUBTLE XSS (con't)

```
<%# list.html %>
<%= render :partial => "user", :collection => @users %>

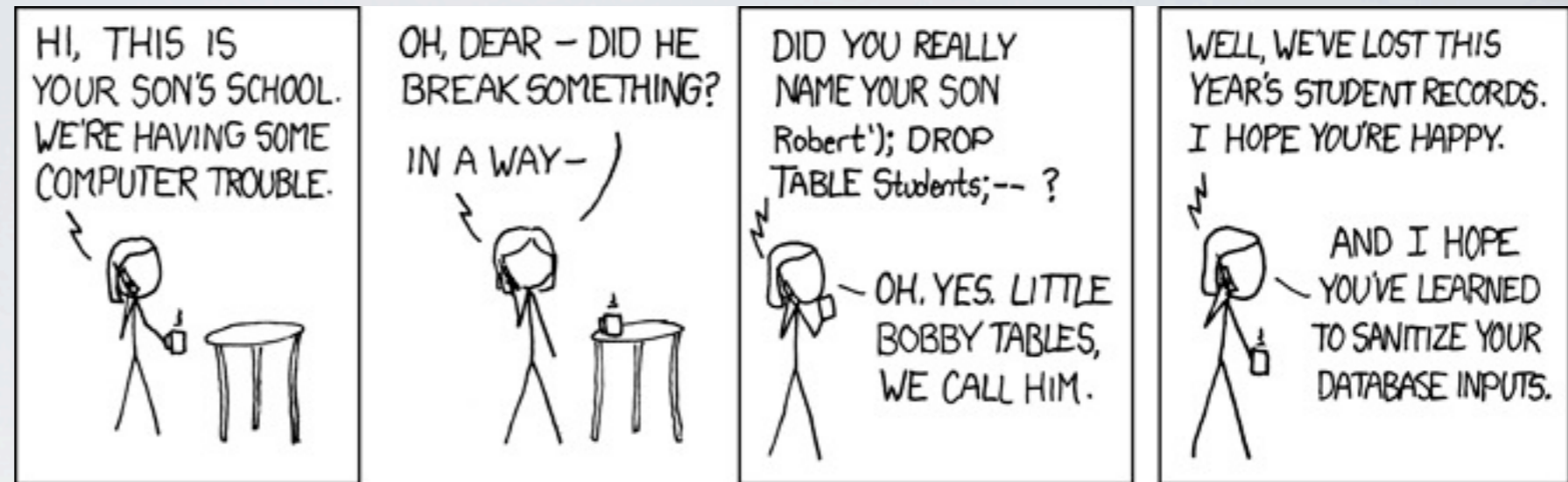
<%# _user.html.erb %>
Stuff about the user...
<%= render partial: 'bio', :locals => { :user_bio => raw(user.bio) } %>

<%# _bio.html.erb %>
Awesome bio: <%= user_bio %>
```

SUBTLE XSS (con't)

- Rails 3 assumes all output is evil and escapes it during output
- Rails 2 requires developers to mark output as HTML “safe”

SQLi



- Isn't that problem pretty much solved? That comic isn't even funny anymore (not since 2008? Maybe?)
- Why static analysis for SQLi tracking?

How about **\$grep -rin "where.*#{"**?
or **\$grep -rin params ***

SQLi (con't)

- “Semantic grep” is hard, and often vague
- It often works for obviously bad things, but not for nuanced problems
- Checking for **#{** in a scope is not sufficient!

SQLi (con't)

```
class User < ActiveRecord::Base
  attr_accessible :name, :bio

  scope :name_like,
    lambda { |first_name| where("name LIKE ?", "#{first_name}%")}

  scope :bad_name_like,
    lambda { |first_name| where("name LIKE '#{first_name}%'")}
end
```


LEVERAGING RAILS' CONVENTIONS

- Input sources for actions are very standard:
 - **params** - HTTP query parameters
 - **cookies** - the cookie object
 - **request** - the HTTP request object

USER INPUT IN RAILS

Tracing user input, or input derived from user-input, is easy

```
class UsersController < ApplicationController
  def sender
    # This is flagged even with our attempts to whitelist
    method = cookies[:method].gsub("[^a-z]", ' ')+ "appended"
    User.first.send(method)

    # dangerous parameter host is caught
    redirect_to params.merge(action: :index)
  end
end
```

EXTENSIBILITY TO GEMS

- Ruby libraries come in “gems”
- many gems for Rails apps strive to provide conventions within the Rails idioms

RAILS DEPENDENCY CHECKING

- Modern Rails apps rely on Bundler to manage gem dependencies
- Bundler tracks version numbers for dependency reasons, can be used to check for vulnerable versions of Rails
- May be extended to track vulnerable gems (requires gems to use CVEs or another tracking system)

GEMS (CON'T)

- Ernie Miller's MetaSearch provides security methods to mirror `attr_accessible` and `attr_protected`:
`attr_searchable`, `attr_unsearchable`,
`assoc_searchable`, `assoc_unsearchable`
- Not currently supported by Brakeman but is a fairly simple extension (similar to checking for attribute restrictions)

IT ISN'T ABOUT RAILS!

- Really we're not talking about Rails
- Rails does a lot right that helps us to secure our applications
- Those same things help us to automatically reason about them
- They are essentially well codified calling conventions hidden under the hood
- This can be done in any language!

BUT THIS IS A SECURITY CONFERENCE...

- Should we be talking about attacking web apps instead of securing them?
- CoC also allows for more efficient and accurate auditing
- By understanding what conventions exist we also know in what ways we can attack those conventions (when used by less skilled developers)
- There's no magic bullet

FULL STACK FRAMEWORKS

- Full stack MVC Frameworks implement all components needed for a web application, including user-input, routing and business logic, and a persistence layer for “model” or other data

DJANGO



- Full-Stack (MTV) framework but not-CoC written in Python
- Developers must decide on organization of applications

DJANGO



- Focus on security various layers:
 - persistence layer to prevent SQLi via querysets
 - presentation layer to prevent XSS via escaping
 - request layer via anti-CSRF nonce - but must remember to include it (though a warning is displayed if it is not)!

GROK



- Python based CoC framework
- No convention for user controlled data in actions. All parameters to an action must be treated as user controlled... but what about in methods lower down the call tree?

GRAILS



- Groovy, a JVM based dynamic language (with static typing and compilation)
- Grails - Rails reimplemented in Groovy
- CodeNarc static analysis engine leverages the strength of the JVM, Grails exposes user data very consistently in the same fashion as Rails

MICROFRAMEWORKS



- Ruby - Sinatra, Cuba
- Python - Flask

MICROFRAMEWORKS



- Lightweight - may only implement a subset of the MVC stack
 - Commonly only routing and presentation are implemented, letting persistence be implemented elsewhere if needed
- This is fantastic for small apps or API endpoints

SINATRA



- From their README: “Sinatra is a DSL for quickly creating web applications in Ruby with minimal effort”
- Also great for writing lightweight API endpoints
- Does not carry with it the heavyweight baggage of ActiveRecord, etc.

SINATRA (con't)



- routes are defined with the **get** or **post** methods
- parameters may be passed in multiple ways

SINATRA (con't)



Some are easy to reason about:

```
get '/hello/:name' do
  "Hello #{params[:name]}!"
end
```

SINATRA (con't)



Some are *harder* (but still possible) to reason about:

```
get  '/hello/:name' do |n|  
    "Hello #{n}!"  
end
```

We assume that all block parameters are user controlled

ADDING CONVENTIONS

- To implement conventions effectively in a web framework we must
 - identify boundaries: user-input, persistence, output
 - make entry and exit points to those boundaries extremely consistent and sufficiently powerful that developers don't *attempt* to bypass them
 - lean on those methods to identify vulnerabilities

FULL-STACK FRAMEWORKS

- Rails, Grails, Django, and Grok are all “full-stack” frameworks
- These frameworks must concern themselves with user-input, persistence, and output
- Since they provide all of these layers they can
 - provide sane defaults that may be reasoned about
 - focus on securing the interfaces between the layers

MICROFRAMEWORKS

- Micro-frameworks may implement only some concerns
- Reasoning about those interfaces is more difficult if a developer rolls her own persistence layer
- Generalizing from the beginning may lead to YAGNI
- Not generalizing leads to difficult to reason about resource / security boundaries

QUESTIONS?



MERCI!