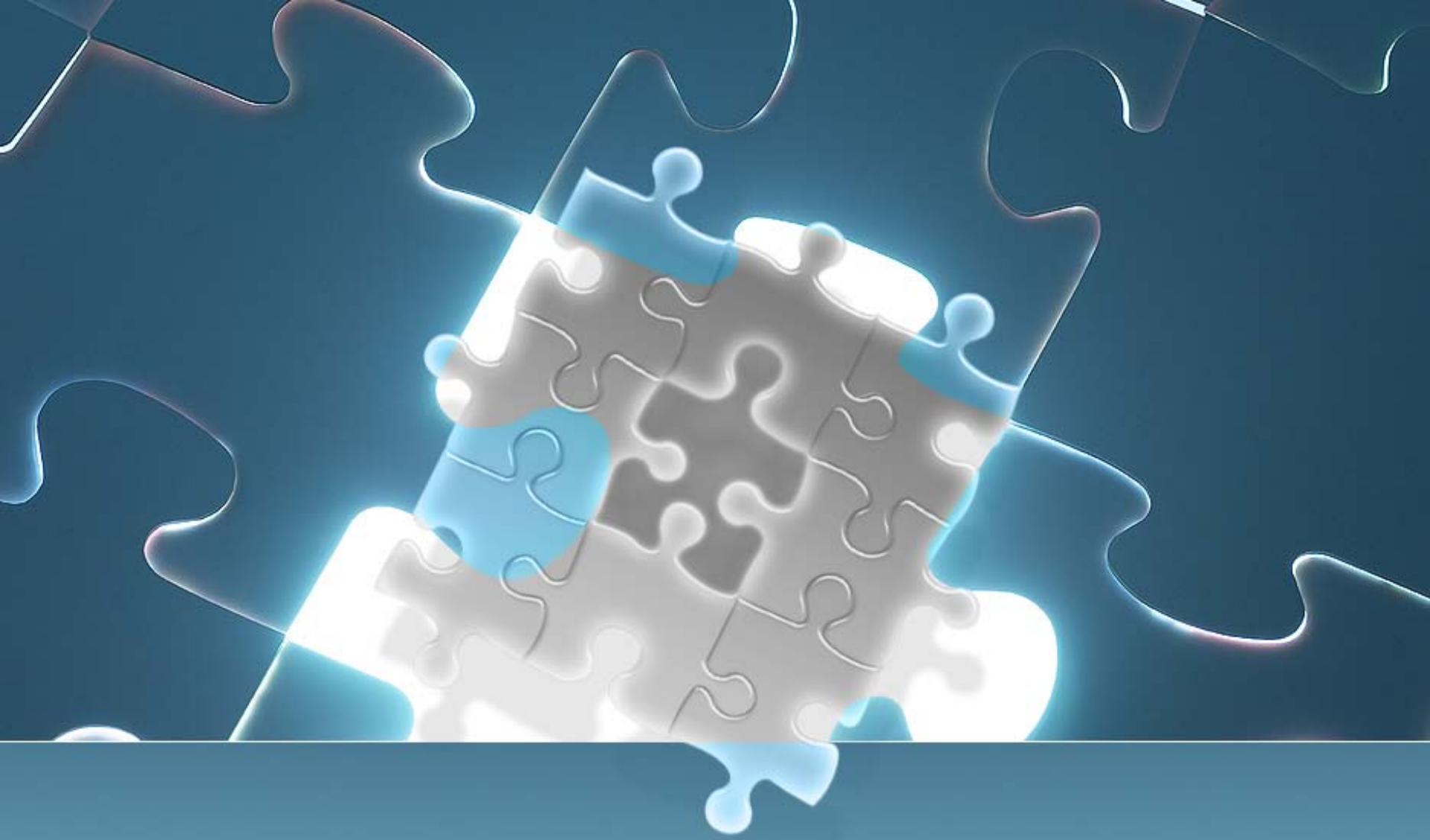

Rails Security : It's Not Just a Good Idea

By David Keener

Introduction

- Why security matters
- Realistic goals
- Learning good practices

  ...by looking at some bad practices
- Resources
- Some final thoughts

Who Am I?

- Long-time Ruby/Rails developer
- Founder/Organizer – RubyNation & DevIgnition conferences
- Last 3 projects...
  - Online Video Contest: With $10K-plus prizes...
  - Bank: Online charitable donations via credit card
  - Cyber Security: <cannot discuss details>
- “School of hard knocks” for security

Why Security Matters

- More personal data is moving online than ever before
  - Social networks, retail sites, mobile devices, etc.
- The community of “bad actors” is growing and it’s international
  - The Internet is everywhere
- If the data in your app has value, someone WILL try to get to it
  - RSA is THE ultimate example
- There are serious legal ramifications to security breaches

Realistic Goals

- Security is a BIG topic
- Nobody can cover it all in one talk

Goal 1: Illustrate how important security is

Goal 2: Demonstrate some good practices

(and a few bad mistakes)

Goal 3: Point you to some awesome resources

“…out of 300 audited sites,
97% are vulnerable to attack”

- From a Gartner Group survey

1. A Silly Vulnerability

```ruby
class SessionController < ApplicationController

  def login
    # Render login page
  end

  def create
    # Process login request
  end

  .
  .
  .

end
```

A newbie mistake…

```ruby
class ApplicationController < ActionController::Base
  before_filter :login_required, :except => [:login, :create]

  .
  .
  .


end
```

…here's the exploit:

curl –d 'forum_id=1&message[subject]=foobar' http://yoursite.com/forums

1. Mitigation

- Option 1: Include the before filter in proper controller
- Option 2: Create a NoAuth controller...
  - ApplicationController: before_filter :login_required
  - NoAuthController: skip_before_filter :login_required

General Rule: “Default Deny” is your friend.

Also: Be aware of code smells such as out-of-control before-filter stacks, filters with non-obvious side effects, etc.

2. Mass Assignment

How many times have we seen this:

```ruby
class UsersController < ApplicationController

  def create
    @user = User.new(params[:user])
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      flash[:notice] = "User was successfully updated."
      redirect_to(users_path)
    else
      flash[:notice] = "User update failed."
      render 'edit'
    end
  end

  .
  .
  .

end
```

Mass assignment is convenient…but it's not safe.

2. Mitigation

- Mass assignment – only for fields with no security impact
- ALL models should use attr_accessible to specify fields that can be mass-assigned

```
attr_accessible :first_name, :last_name, :email
```

- Other fields can be individually assigned if needed

3. Regexes

What's wrong with this code?

```
1  class IpAddress
2    validate :valid_ip_address
3
4    def valid_ip_address
5      ip_regex = /^([1-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])(\.([0-9]|[1-9][0-9]|1[0-9]
   [0-9]|2[0-4][0-9]|25[0-5])){3}$/
6      if ip_address.present && ip_address.match(ip_regex).nil?
7        errors.add(:base, "Invalid IP Address")
8      end
9    end
10
11 end
```

3. Mitigation

- The regular expression uses ^ and $ to match the start and end of the string
- In Ruby, this only matches a single line if multi-line input is provided

Use \A and \z instead for input validation

4. File Uploads

```erb
<h1>Upload File</h1>

<%= form_tag({:action => :create}, :multipart => true) do %>
  <p><%= file_field_tag 'upload_file' %></p>
  <p><span class='button'><%= submit_tag("Upload") %></span></p>
<% end %>
```

views/uploads/new.html.erb

```ruby
class UploadsController < ApplicationController
  UPLOAD_PATH = 'public/data'

  def create
    upload = params['upload_file']
    path = File.join(UPLOAD_PATH, upload.original_filename)
    File.open(path, "wb") { |x| x.write(upload.read) }
  end

end
```

controllers/uploads.rb

Uploads are prone to numerous potential security issues…

4. Problems

- The original file is left in the /tmp directory
  - Under a name like Rack*multipart*{random stuff}
  - Executable files could theoretically be executed by someone (BAD)
- The uploaded file is copied to $RAILS_ROOT/public/data
  - Under a name like Rack*multipart*{random stuff}
  - The file is web-accessible
  - Embedded JavaScript will have server access (BAD)
  - Could potentially see files uploaded by other users (PRIVACY)
- Uploaded files are never cleaned up
- File names can have collisions

4. Mitigations

- File in /tmp directory
  - Remove the file immediately after it has been copied
- File is copied to $RAILS_ROOT/public/data
  - Copy files to a non-web-accessible dir
  - Validate file types and eliminate undesirable files
  - Ensure that files are never left as executable
- Uploaded files never cleaned up
  - Delete files when no longer needed
- File names can have collisions
  - Add a unique ID as a filename prefix to prevent name collisions

5. Uploading XML Files

What if I want to ...

upload and parse an XML File?

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE foo [
3 <!ELEMENT foo ANY>
4 <!ENTITY disclaimer SYSTEM "/home/dkeener/sandbox/myapp/config/database.yml">]>
5 <things>
6   <thing id="Godzilla">
7     <description>&disclaimer;</description>
8   </thing>
9   <thing id="Mothra">...</thing>
10 </things>
```

It’s called an XML External Entity Expansion Attack…

5. Mitigation Options

- Option 1: Before parsing, regex for !DOCTYPE, !ELEMENT or !ENTITY and immediately reject the file
  - No need to be nice to the user
- Option 2: Disable entity expansions
  - Differs based on XML parser used

5. Mitigation - Nokogiri

```ruby
def parse_with_nokogiri(xml, xsd)
  xsd_doc = Nokogiri::XML::Schema(File.read(xsd))
  doc = Nokogiri::XML(xml) do |config|
    config.noent.nonet
  end

  xsd_doc.validate(doc).each do |err|
    @errors << err.message
  end

  doc
end
```

Parse settings

- Validates the XML against the XSD
- noent => No entity expansions
- nonet => No network access
- Does not actually do external entity expansions, but the infrastructure is there

5. Mitigation - REXML

```
1
2  def parse_with_rexml(xml)
3    REXML::Document.entity_expansion_limit = 0
4    REXML::Document.new(xml)
5  end
```

- Non-validating parser
- entity_expansion_limit: raises exception if it finds any entity expansions
- Not actually required to do entity expansions, but it seems to have some of the infrastructure

Some Best Practices

Here are a few more best practices

- Always sanitize user-provided input
- Rolling your own authentication is an anti-pattern...think twice
  - Use Devise, restful_authentication, CanCan, etc.
- Use database-backed session storage
- “Default Deny” is your friend
- Use SSL for secure logins in production

Resources

- Rails Security Guide
  http://guides.rubyonrails.org/security.html
- OWASP Ruby on Rails Security Guide V2
  https://www.owasp.org/index.php/Category:OWASP_Ruby_on_Rails_Security_Guide_V2
- DHS Sensitive Systems Policy Directive 4300A
  http://www.uscg.mil/hq/cg9/NAIS/RFP/SectionJ/dhs-4300A-policy.pdf

Conclusion

- Security matters...
  Compromising PII or financial info will always be BAD
- It's easier to build security in from the beginning than to retrofit it later
- Make good security practices second nature now...they will pay off later
- The security of your app must be TESTED
  RSpec, Cucumber and similar tools are essential


It’s smooth sailing
If you build good
security practices
into your app...

Questions

Feel free to contact me:

dkeener@keenertech.com
david.keener@gd-ais.com
Twitter: dkeener2010
Blog: http://www.keenertech.com

GENERAL DYNAMICS

We're also looking for some good Rubyists...