Railsのセキュリティ

前田 修吾

ネットワーク応用通信研究所

2007-07-05

自己紹介

名前

前田 修吾

所属

ネットワーク応用通信研究所(NaCl)
基盤研究グループ

肩書

主任研究員

Rubyとの出会い

- 1997年
- JavaHouse ML
- 高木浩光先生

Rubyとの関わり

- 各種提案
  - callcc
  - protected
- アプリケーション/ライブラリ
  - mod_ruby
  - net/ftp.rb, net/imap.rb

Rubyとの関わり(2)

- インフラの管理
  - 公式Webサイト
  - 開発レポジトリ

Railsとの出会い

- 2005年4月
- 仕事のため
  - SOAPがメイン

Railsとの関わり

- パッチ
  - 文字コード関連
  - 悲観的ロック
  - バグ・パフォーマンスチューニング
- AWDwR監訳
  - 第2版作業中
  - ぜんぜん進んでません…

今日のテーマ

- Ruby/Railsの紹介
- Railsのセキュリティ

アンケート

- Rubyを知っている人?
- Railsを知っている人?

Ruby

- Rubyとは?
- コンセプト
- 特徴

Rubyとは?

- プログラミング言語
- まつもとゆきひろさん作
- 14才(思春期)

Rubyのコンセプト

- プログラマに最適化
  - プログラマ = まつもとさん
  - プログラマ != 初心者
- プログラムを簡潔に
  - 「言語仕様を簡潔に」ではない

Rubyの特徴

- スクリプト言語
- オブジェクト指向言語
- 動的言語
- 関数型言語?

スクリプト言語

- インタープリタ
- 簡潔な記法
- テキスト処理機能

インタープリタ

```
$ ruby hello.rb
$ ruby -e 'puts "hello world"'
$ ruby -pe 'gsub(/perl/, "ruby")'
```

簡潔な記法

```
class Hello
  def say(whom = ”world”)
    puts ”hello ” + whom
  end
end
hello = Hello.new
hello.say(”shugo”) #=> hello shugo
hello.say          #=> hello world
```

テキスト処理機能

```
s = "perl is cool".sub(/perl/, "ruby")
s = "hello world¥n".chop
s = "  hello wolrd  ".strip
words = "ruby perl".split
word = "ruby perl".slice(/¥w+/)
name = "def foo".slice(/def (¥w+)/, 1)
```

オブジェクト指向言語

- 純粋
- クラスベース
- Mix-in
- オブジェクトベース

純粋

- すべてのデータがオブジェクト
  - 数値
  - 文字列
  - 配列

クラスベース

- オブジェクトはかならずクラスに属する
- クラスによってオブジェクトの振舞が決まる

Mix-in

- 限定された多重継承
  - 複数のクラスは継承できない
  - モジュールなら複数継承できる
- モジュールとは
  - クラスと同じようなもの
  - インスタンス化できない

オブジェクトベース

- クラス定義は必須ではない

```
duck = Object.new
def duck.quack
  puts ”クワックワッ”
end
duck.quack
```

動的言語

- 静的型はない
  - 変数に型は指定しない
  - コンパイル時に型情報は得られない
- ほとんどのことを実行時に行う
  - クラス定義
  - メソッド定義

Duck Typing

- あひるのように歩き、あひるのように泳ぎ、あひるのように鳴くものは、あひるとみなす
- アドホックな多態

関数型言語?

- ブロックをメソッドに渡せる

```
p ["1", "2", "3"].collect { |s|
  s.to_i
} #=> [1, 2, 3]
```

関数型言語?(2)

- オブジェクト化することも可能

```
plus = lambda { |x, y|
  x + y
}
p plus.call(2, 3) #=> 5
```

関数型言語?(3)

- Ruby 1.9ではこんな書き方も

```
Y = ->(f) {
  ->(x) {
    f[->(arg) { x[x][arg] }]
  }[
    ->(x) {
      f[->(arg) { x[x][arg] }]
    }
  ]
}
```

Rails

- Railsとは?
- コンセプト
- 特徴
- コンポーネント

Railsとは?

- Webアプリケーションフレームワーク
- David Hainemeier Hanssonさん作
- 3才

Railsのコンセプト

- DRY
  - Don't Repeat Yourself
- CoC
  - Convention over Configuration
- 生産性 > 柔軟性

DRY

- 重複を排除する
  - コピペは悪

CoC

- 設定より規約
- 命名規約により設定を省略
  - ファイル名をクラス名から推測
  - テーブル名をクラス名から推測
- 使いやすいデフォルト
  - 設定で上書きも可能

生産性 > 柔軟性

- 思い切った割り切り
- 柔軟性よりも生産性を重視
- 80%のWebアプリケーションを効率的に書ければよい

Railsの特徴

- オールインワン
- 自動生成
- プラグイン

オールインワン

- MVCをセットで提供
  - 密結合
- その他のユーティリティ
  - テストのサポートなど

自動生成

- scaffold
  - シンプルなCRUD
  - いじりやすい

プラグイン

- サードパーティ製のプラグイン
- フレームワークの機能を拡張
- Rails本体に取り込まれることも

コンポーネント

- ActiveRecord
- ActionView
- ActionController

ActiveRecord

- モデルを担当
- O/Rマッパー
- PofEAAの同名のパターンに由来

対応

テーブル	クラス
行	オブジェクト
列	属性

例

```
class Post < ActiveRecord::Base
end

post = Post.new(:title => ”テスト”,
  :body => ”テストです。¥n”)
post.save
test = Post.find(:first,
  :conditions => ”title = ’テスト’”)
p test.body
```

ActionView

- ビューを担当
- デフォルトのビューはeRuby

例

```
<p><%= h(@message) %></p>
```

ActionController

- コントローラを担当

例

```
class PostsController < ApplicationController
  def index
    @posts = Post.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @posts.to_xml }
    end
  end
end
```

セキュリティ

- Rubyのセキュリティ
- Railsのセキュリティ

Rubyのセキュリティ

- 長所
- 短所

長所

- バッファオーバーフローがない
- $SAFEによるセキュリティ機構
- 高い可読性

Buffer Overflow

- バッファは必要に応じて自動的に拡張
  - String#concat, Array#push
- Ruby自体や拡張ライブラリにバグがある場合は除く
  - アプリケーションレベルでは気にしなくてよい

$SAFE

- スレッドローカル変数
- $SAFEの値によってセーフレベルが変わる

$SAFE == 1

- 実行対象のプログラムが外部の環境から保護される
- たとえば、Webアプリケーションを悪意のあるユーザから保護

$SAFE == 4

- プログラムの実行環境が、実行対象のプログラムから保護される
- Sandbox
- 今回は触れない

オブジェクトの汚染

```
# オブジェクトに汚染フラグがある
# 外部からの入力データは汚染されている
p ARGV[0].tainted? #=> true

# 汚染は他のオブジェクトに伝播する
p (ARGV[0] + "baz").tainted? #=> true
```

フラグの設定/除去

```
# 汚染フラグの設定
s.taint
p s.tainted? #=> true

# 汚染フラグの除去
s.untaint
p s.tainted? #=> false
```

注意

- taint/untaintは破壊的操作
- 場合によってはコピーが必要

```
t = s.dup.untaint
```

危険な操作の禁止

- 危険な操作を汚染されたオブジェクトに適用すると、SecurityError例外が発生

```
p s.tainted?  #=> true
system(s)     #=> SecurityError
```

高い可読性

- Rubyのコードは読みやすい
- コードレビューのコストが低い
- 脆弱性も発見しやすい

短所

- eval
- スタックオーバーフロー
- 静的解析が難しい

eval

- 文字列をRubyプログラムとして評価
- 高い柔軟性
  - 多くの場合高すぎる…

Stack Overflow

- 現在のRubyはスタック消費が激しい
- とくに深い再帰は危険
  - 再帰の深さが入力データによるものなど
- 最悪の場合、SEGV
  - 運が良ければSystemStackError例外

静的解析

- 機械にとってはRubyは読みにくい
  - 変数に型がない
  - ほとんどのことが実行時に行われる
- 規約をヒントにできる?

Railsのセキュリティ

- 長所
- 短所
- セキュアな書き方

長所

- Rubyのセキュリティ上の長所を受け継ぐ
- 書き方が強制される
  - フレームワーク全般の性質
- Rails自体やプラグインによるサポート
  - 前提条件が多いためサポートしやすい

短所

- 成熟度が低い
  - 枯れたフレームワークにくらべて
  - 1.1.4では2度に渡る脆弱性対応
- $SAFE == 1に未対応
  - evalの利用も多い
  - プラグインでカバー

セキュアな書き方

- モデル
- ビュー
- コントローラ
- セッション
- プラグイン

モデル

- 検索条件の指定
- 属性の保護

検索条件の指定

- Railsでは:conditionsパラメータで指定

```
post = Post.find(:first,
  :conditions => "title = 'hello'")
```

危険な例

- 変数などの値を文字列に直接埋め込む

```
post = Post.find(:first,
  :conditions => "title = '#{params[:title]}'")
```

- conditionsの値はそのままSQLに埋め込まれる
- SQLインジェクションの危険性

対策

- バインド変数を使う
- 名前付きバインド変数を使う
- 動的ファインダを使う

バインド変数

```
# ?の部分にparams[:title]がクォートされて入る
post = Post.find(:first,
  :conditions => [
    "title = ?",
    params[:title]
  ])
```

クォートの仕組み

- データの型(オブジェクトのクラス)に応じてクォートされる
- 実際のクォートの仕方は、DBMSごとのアダプタによって異なる
  - database.ymlのencodingの設定によっても挙動が変わる(SJIS)

クォートの仕組み(2)

- モデルの属性を保存する際などは、列の型情報も用いる
  - バイナリデータには特殊なエスケープを施すなど

名前付きバインド変数

```
title = params[:title]
post = Post.find(:first,
  :conditions => [
    "title = :title",
    { :title => title }
  ])
```


動的ファインダ

```
title = params[:title]
# find_by_titleは属性名に応じて作られる
post = Post.find_by_title(title)
```


動的ファインダ(2)

```
# allを付けるとすべての結果を配列で返す
posts = Post.find_all_by_title(title)
# andで複数の属性のAND検索
post = Post.find_by_title_and_author(title, author)
```


属性の保護

- ActiveRecordでは属性の一括代入が可能

```
@post = Post.find(params[:id])
@post.update_attributes(params[:post])
```


危険な例

```
class User < ActiveRecord::Base
end

# 実際にはクライアントからの入力
params = {
  :user => {
    ...,
    :admin => "true" # admin属性を改竄
  }
}
# ユーザの権限が昇格
user.update_attributes(params[:user])
```


対策

- attr_protected
- attr_accesible

attr_protected

```
class User < ActiveRecord::Base
  # 以下で指定した属性は一括代入で無視
  attr_protected :admin
end

# 以下のような明示的な変更は可能
user.admin = true
p user.admin? #=> true
```


attr_accesible

```
class User < ActiveRecord::Base
  # 以下で指定した属性のみ一括代入可能
  attr_accesible :name, :nickname, :age
end
```


ビュー

- エスケープ

エスケープ

- HTMLにデータを埋め込む際にはエスケープに注意する必要がある。

危険な例

```
<p>
<%= @message %>
</p>
```


対策: hを使う

```
<%= h(@message) %>

# ヘルパーを使う場合も注意
<%= link_to h(label), :action => ”hello” %>
```


注意点

- テキストはhでエスケープ
- HTML断片はそのまま
  - エスケープは必要な箇所で一度だけ
- 場所によってはhではダメ
  - <script>要素の内容
  - onclick属性などの値

コントローラ

- 非公開メソッド
- 権限のチェック

非公開メソッド

- コントローラのpublicメソッドはすべてアクションとして公開される!

危険な例

```
class UsersController < ApplicationController
  def clear
    raise "error" if current_user.admin?
    destroy_users
  end

  def destroy_users
    User.delete_all
  end
end
```


対策: private

```
class PostsController < ApplicationController
  def clear
    raise "error" if current_user.admin?
    destroy_posts
  end

  private

  def destroy_posts
    Post.delete_all
  end
end
```


private/protected

- 両方ともサブクラスからアクセス可能
- 違いはprivateではレシーバを省略した形式でしか呼び出せないこと

protected

- 2項演算子などを定義する際に利用
- privateメソッドの呼び出しより若干遅い
- 多くのRailsコードは間違い

権限のチェック

- 権限のチェックはプログラマの責任で行う必要がある
- チェックの洩れが起こりがち

対策

- フィルタ
- with_scope
- ScopedAccess

フィルタ

- アクションの前後で実行される
  - before_filter
  - after_filter
  - around_filter

before_filter

```
class PostsController < ApplicationController
  before_filter :login_required, :except => [:list, :show]

  private

  def login_required
    if session[:user_id]
      return true
    else
      redirect_to(:controller => "login", :action => "login")
      return false
    end
  end
end
```


with_scope

- ActiveRecordのメソッドにパラメータを自動的に追加
- ブロックを受け取る
- CRUDに応じて条件を指定

with_scope(2)

```
class PostsController < ApplicationController
  def list
    Post.with_scope(:find => {
                            :conditions => [
                              "owner = ?",
                              session[:user_id]
                            ]}) do
      @posts = Post.find(:all)
    end
  end
end
```


DRY with_scope

```
class PostsController < ApplicationController
  private

  def with_current_user_scope(&block)
    Post.with_scope({
      :find => {
        :conditions => [
          "user_id = ?",
          session[:user_id]
        ]
      },
      :create => {
        :user_id => session[:user_id]
      }}, &block)
  end
end
```


DRY with_scope(2)

```
class PostsController < ApplicationController
  def list
    with_current_user_scope do
      @posts = Post.find(:all)
    end
  end

  def create
    with_current_user_scope do
      Post.create(params[:post])
    end
  end
end
```


注意

- アソシエーションの場合には
  うまくいかないことがある

```
with_current_user_scope do
  category = Category.find(params[:category_id])
  @posts = category.posts
end
```

- SQLが実際に発行されるの
  は@postsが使われる時点

注意(2)

- :includeを使った場合も原因は違うがうまくいかない

```
with_current_user_scope do
  category = Category.find(params[:category_id],
                           :include => :posts)
  @posts = category.posts
end
```


ScoppedAccess

- with_scopeのフィルタ化
- アクション毎の指定がいらない。

ScoppedAccess(2)

```
class PostsController < ApplicationController
  around_filter ScopedAccess::Filter.new(Post)

  private

  def method_scoping
    return {
      :find => {
        # ...
      },
      :create => {
        # ...
      }
    }
  end
end
```


セッション

- Railsのセッション
- Session Fixation
- CSRF

Railsのセッション

- セッションIDでセッションを識別
- セッションIDの保持にはCookieを利用
  - プラグインによりURLに含めることができる
- セッションデータは定期的に掃除する必要がある

Session Fixation

1. 攻撃者がセッションIDを取得
2. 攻撃者が上記のセッションIDを用いて被害者をログイン画面に誘導
3. 被害者がログイン
4. 攻撃者が上記のセッションIDを使ってセッションを乗っ取る

条件

- ログイン前からセッションIDが発行されている
  - Railsも該当
- ログイン後も同一のセッションIDが利用される

対策

- セッションIDの再発行
- セッションを使わない

セッションIDの再発行

```
class LoginController < ApplicationController
  def authenticate
    if user = User.authenticate(params[:username],
                                params[:password])
      reset_session
      redirect_to(:controller => "posts",
                  :action => "index")
    else
      redirect_to(:action => "index")
    end
  end
end
```


セッションを使わない

- そもそも認証を行わないような場合
  - 不特定多数対象のアンケートなど
- hiddenでデータを受け渡す

CSRF

1. 被害者がログイン
2. 攻撃者は被害者があるURLにアクセスするよう誘導
3. 被害者が認証された状態でURLにアクセス
4. 記事の投稿や商品の注文などの処理が通常のアクセスとして実行される

対策

- リクエストに攻撃者が予測できないようなトークンを含め、処理の前にトークンをチェックする
- プラグインを使う

プラグイン

- security_extension
- Safe ERB
- SafeRecord

security_extension

- CSRF対策用プラグイン
- いわゆる高木方式

インストール

```
$ ./script/plugin install ¥
  http://svn.aviditybytes.com/rails/plugins/security_extensions
```


コントローラの修正

```
class PostsController < ApplicationController
  verify_form_posts_have_security_token :only =>
    [:create, :update, :destroy]
end
```


ビューの修正

```
<% form_for(:post, :url => posts_path) do |f| %>
  <%= hidden_field_tag(:session_id_validation,
                       security_token) %>
...
<% end %>
```


Safe ERB

- taint機構を使ってHTMLのエスケープ洩れをチェック
- 汚染されたオブジェクトを出力に含めると例外
- hでエスケープする際にuntaint

インストール

- 以下のURLからダウンロード
  <URL:http://www.kbmj.com/~shinya/rails/safe_erb-0.2.zip>
- vendor/plugins/に展開

動作

```
<%= @message %>          <!-- 例外 -->
<%= h(@message) %>       <!-- OK -->

<%= @post.title %>       <!-- 例外 -->
<%= h(@post.title) %> <!-- OK -->
```


SafeRecord

- taint機構を使ってSQLのエスケープ洩れをチェック
- 汚染されたオブジェクトがSQLに含まれると例外
- DBMS毎のアダプタのクォート処理時にuntaint
- Safe ERBに依存

インストール

```
$ ./script/plugin install ¥
  http://projects.netlab.jp/svn/rails_plugins/safe_record
```


動作

```
# 例外
post = Post.find(:first,
  :conditions => "title = '#{params[:title]}'")

# OK
post = Post.find(:first,
  :conditions => [
    "title = ?",
    params[:title]
  ])
# OK
post = Post.find_by_title(params[:title])
```


注意

- 先週末に思い付きで作ったのでたぶん不具合が…
- とりあえず開発環境で試してください

まとめ

- Railsの特徴を理解してセキュアに
- Rails自体やプラグインの機能を活用

おまけ

- Rabbitにもエスケープ洩れが!
  - スライドタイトルに_があると「Jump to」で…

終り

ご静聴ありがとうございました