

Rails et la sécurité

Bruno Michel

3 septembre 2008

Introduction

La sécurité des applications web est souvent un sujet délicat : peu de temps à y consacrer, mais cela peut avoir des conséquences assez graves. Pas de panique ! [Ruby on Rails](#) est bien armé et avec un peu de rigueur, on peut se protéger sans trop de difficultés. Nous allons voir les principaux types d'attaques et comment les éviter.

Injections SQL

Commençons par un grand classique : [les injections SQL](#). Une injection SQL consiste simplement à envoyer des données non prévues dans une requête SQL. Prenons comme exemple une application web où les utilisateurs sont authentifiés de la manière suivante :

```
@current_user = User.find(:first, :conditions =>
  "login='#{params[:login]}' AND password='#{params[:password]}'")
```

En temps normal, quand Joe s'authentifie, la requête SQL suivante est exécutée :

```
SELECT * FROM users WHERE login='Joe'
  AND password='0521bc575b0ff61daa62494c7ae9c5b6' LIMIT 1;
```

Mais supposons maintenant que Kevin, un Script Kiddie, passe dans le coin et décide de mettre "Joe' ; -" dans le champ login. La requête SQL va alors ressembler à :

```
SELECT * FROM users WHERE login='Joe'; --'  
AND password='00000000000000000000000000000000' LIMIT 1;
```

Kevin a réussi à se faire passer pour Joe sans connaître son mot de passe!

Heureusement, Active Record permet de [nous en protéger assez facilement](#). Pour cela, il suffit d'utiliser les formes échappées comme cela :

```
@current_user = User.find(:first, :conditions =>  
  ["login=? AND password=?", params[:login], params[:password]])
```

ou de façon équivalente :

```
@current_user = User.find(:first, :conditions =>  
  {:login => params[:login], :password => params[:password]})
```

Dans les 2 cas, Active Record rajoutera un caractère `'\'` devant chaque apostrophe de façon à éviter les injections SQL.

L'authentification et la gestion des droits

Pour la grande majorité des projets, l'authentification (et la gestion des droits qui vont avec) est un passage obligé. Pour cela, il existe un certain nombre de points importants à respecter comme le chiffrement des mots de passe stockés en base de données (ce que nous n'avons pas fait dans l'exemple précédent). Les erreurs sont vite arrivées, aussi je vous recommande d'utiliser des plugins reconnus comme [Restful Authentication](#), [OpenID Authentication](#) et [Authorization](#).

Il ne vous reste plus qu'à faire attention à un dernier détail : mettre en cache des pages nécessitant une authentification est une mauvaise idée. En effet, ces pages vont alors être servies par le serveur web sans passer Rails, et donc sans vérification de l'authentification.

Se protéger des données forgées

L'étape suivante consiste à bien sécuriser l'accès aux données, aussi bien en lecture qu'en écriture. En effet, Rails possède quelques raccourcis très pratiques, mais qui peuvent poser problème quand ils sont mal maîtrisés. Le plus courant est [l'affectation de masse](#), technique qui consiste à créer un

objet Active Record directement depuis les paramètres de la requête HTTP. Par exemple, la création d'un compte utilisateur pourra s'effectuer de la façon suivante :

```
@user = User.create(params[:user])
```

Supposons maintenant que la table 'users' comporte un champ 'admin' qui vaut 0 par défaut ou 1 pour les super-utilisateurs. Un utilisateur malveillant pourrait forger la requête HTTP pour ajouter un paramètre `user[admin]=1` afin de gagner les pouvoirs réservés aux admins. La première solution pour se protéger de cette attaque consiste à écrire explicitement quels sont les paramètres autorisés :

```
@user = User.create(  
  :login => params[:user][:login],  
  :email => params[:user][:email],  
  :password => params[:user][:password],  
  :cgu => params[:user][:cgu])
```

Mais ceci peut vite devenir pénible quand on commence à avoir des formulaires un peu conséquents. C'est pourquoi on lui préfère généralement la deuxième solution : la déclaration dans le modèle de la liste des attributs qui ne peuvent pas être modifiés. Cette déclaration se fait à l'aide de la méthode `attr_protected` comme suit :

```
class User < ActiveRecord::Base  
  attr_protected :admin  
  ...  
end
```

Nous pouvons de nouveau utiliser l'affectation de masse sans craindre qu'un utilisateur se fasse passer pour un admin, Rails s'occupe de filtrer les paramètres.

Dans le même style, un attaquant peut essayer de [forger des URL](#). Si, par exemple, l'utilisateur authentifié peut supprimer l'item `umero123` qui lui appartient, en appelant l'URL `/items/delete/123`, alors que se passera-t-il s'il appelle la même URL pour l'item `umero456` qui ne lui appartient pas ? La réponse dépend du code de la méthode `delete`. Une implémentation de base pourrait ressembler à :

```
class ItemsController < ApplicationController
```

```

    def delete
      Item.delete(params[:id])
    end
  end
end

```

Pour se protéger des URL forgées, on pourrait la transformer en :

```

class ItemsController < ApplicationController
  def delete
    @item = @current_user.items.find(params[:id])
    @item.delete if @item
  end
end

```

Ce n'est pas parfait (on pourrait vérifier que c'est bien une requête de type POST), mais c'est déjà beaucoup mieux.

Un dernier petit truc pour la route avant de passer à autre chose. Si vous avez une API pour laquelle vous utilisez la sérialisation XML, il peut être intéressant de surcharger `ActiveRecord#to_xml` pour que le champ `secret_field` n'y apparaisse pas :

```

class Item < ActiveRecord::Base
  def to_xml(args={})
    super({:except => [:secret_field]}.merge(args))
  end
end

```

Cross-Site Scripting

Jusque maintenant, nous avons vu des attaques directes : un utilisateur essaye de s'en prendre à notre site. Il existe également des attaques plus pernicieuses que l'on classe sous le nom de [Cross-Site Scripting](#) (XSS en abrégé). Leur but est de s'en prendre aux utilisateurs de notre site en glissant des cochonneries sur notre site. Ceci peut aller du spammeur qui mettra une balise `<iframe>` vers son site dans tous les formulaires qui lui passent sous la main à l'injection de javascript non maîtrisé.

Par exemple, quelqu'un crée un item dont la description est la suivante :

```
<script>document.location='http://www.programmez.com/';</script>
```

Si maintenant un visiteur affiche la description de cet item, il sera redirigé vers le site www.programmez.com. Vous vous dites que c'est ennuyeux mais pas bien méchant ? Oui, mais la même technique permet de voler les cookies et donc les sessions associées. Nous allons donc chercher à nous protéger de ces failles XSS.

Pour cela, il est important de faire une distinction entre 2 cas : est-ce que le champ que vous allez afficher peut contenir des balises HTML ou non ? Pour afficher le nom d'un item, on sera dans le premier cas, à savoir pas de balises HTML : on veut juste afficher le nom tel que l'a rentré son propriétaire. Par contre, on peut souhaiter être plus souple pour la description de l'item et laisser la possibilité d'avoir un titre (balise `<h1>`), du gras (``) ou de l'italique (`<i>`). Ces 2 cas ne se traitent pas de la même façon. Pour le premier cas, Rails nous offre un moyen simple de nous en protéger : le helper `h`. En pratique, à chaque fois que l'on souhaitera afficher le titre d'un item, on procédera de la manière suivante :

```
<%=h @item.title %>
```

Ce `h` va convertir les caractères qui pourraient être interprétés par un navigateur web en l'entité HTML correspondante. Problème résolu.

Le deuxième cas est par contre plus difficile à traiter. Vous pouvez être tenté d'utiliser un moteur de formatage de texte comme [RedCloth](#). Attention, cela ne suffit pas à filtrer [toutes les attaques](#) ! Pour votre tranquillité, il vaut mieux utiliser le plugin [WhiteList](#). Depuis Rails 2.0, ce plugin fait partie du framework et peut s'utiliser de la façon suivante :

```
<%= sanitize @item.description, :tags => %w(b i h1) %>
```

Il est possible de déclarer les balises autorisées de manière globale : je vous renvoie à la [documentation officielle](#). Et pour ceux qui veulent être sûrs de ne pas oublier d'appeler à `h` ou à `sanitize`, il existe des moteurs de template alternatifs comme [Safe ERB](#) ou [Erubis](#). Ces moteurs adoptent l'approche opposée : ils filtrent par défaut tous les éléments `<%= %>`, charge au développeur d'indiquer explicitement ceux pour lequel le moteur ne fera pas de filtrage.

Cross-Site Request Forgeries

Juste avant de finir, je voudrais juste dire un mot sur un dernier type d'attaques. Les CSRF, abréviation de Cross-Site Request Forgery, sont des

attaques complexes qui visent à forcer l'utilisateur à envoyer une requête HTTP vers notre site lorsque celui-ci visitera le site de l'attaquant. Je vous renvoie à [wikipedia](#) si vous voulez comprendre comment fonctionne ce type d'attaques. Sachez que Rails vous protège de celles-ci depuis la version 2.0 et qu'il existe un plugin pour les versions plus anciennes : [CSRF-killer](#).

Conclusion

Nous avons pu voir qu'en prenant quelques bonnes habitudes, on pouvait développer des applications sûres en Rails. Il reste cependant des sujets que je n'ai pas abordés comme [l'administration de Rails](#), [apache](#) ou [MySQL \(suite\)](#), [la manipulation des fichiers](#) ou encore [le filtrage des informations sensibles dans les logs](#). Il est également important de se tenir au courant des mises à jour de sécurité de Rails et des plugins que vous utilisez.

Je remercie Benoît Sibaud pour sa relecture attentive.

Références

- <http://www.rorsecurity.info/ruby-on-rails-security-cheatsheet/>
- <http://www.quarkruby.com/2007/9/20/ruby-on-rails-security-guide>
- <http://blog.innerwut.de/2008/1/3/24c3-ruby-on-rails-security>
- http://www.owasp.org/index.php/OWASP_AppSec_FAQ
- http://www.owasp.org/index.php/Top_10_2007