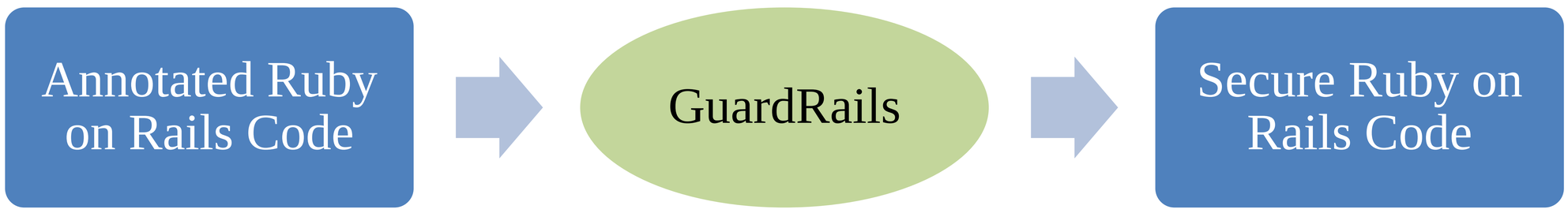
GuardRails: A (Nearly) Painless Solution to Insecure Web Applications

Jonathan Burket, Patrick Mutchler, Michael Weaver, Muzzammil Zaveri, David Evans
University of Virginia



GuardRails is a source-to-source tool that uses annotations to produce secure Ruby on Rails applications with minimal effort from the developer.

Problem

Modern web applications run in a universally privileged mode with unlimited access to potentially private data. To prevent security flaws, developers need to include checking code throughout the application to ensure the data policies are satisfied, often in an ad-hoc and haphazard fashion. This makes code difficult to read and security bugs easy to insert.

Solution

Instead of relying on the programmer to explicitly insert all the correct access control checks in the code, we let the data protect itself from the application by knowing its own security policies. We propose GuardRails, a source-to-source tool that transforms code annotated with high level data policies into a secure Ruby on Rails application. To use GuardRails, a developer simply writes typical Ruby on Rails code and adds annotations that describe data policies ranging from access permissions to sanitization routines.

SQL Injection

While the threat of SQL injection attacks has been greatly reduced with better Ruby on Rails support and the use of prepared statements, applications can still be vulnerable to this kind of attack.

```
match_group = Group.find(:first, :conditions =>
                              "group_name like '"
                              + params[:name] + "'")
```

An example from out test application *PaperTracks* of a SQL query that leaves the site open to an injection attack

This example vulnerability could easily be fixed by using a prepared query, but an inexperienced developer might miss the problem entirely. The consequences of a mistake like this are usually quite severe, often allowing an attacker to view arbitrary data from the database or delete all the contained tables.

Cross-Site Scripting

Cross-site scripting vulnerabilities are among the most prevalent security issues for modern web applications. Protecting against cross-site scripting attacks typically involves manually inserting sanitization code throughout the application so that potentially harmful content is not fully rendered in the response. The complexity of cross-site attacks, however, means that it is hard for a developer to ensure that an output truly is secure. Even if all input goes through sanitization routines, an application can still be vulnerable to second-order injection attacks.

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6
                                  blockquote br cite sub sup ins p)
user_input = sanitize(user_input, :tags => tags, ... )
send(user_input)
```

The above example taken from guide.rubyonrails.org describing countermeasures to security flaws.

Without the proper sanitization routines, a malicious user can execute arbitrary scripts in the browsers of other users of the site. Even one poorly sanitized field can lead to a total compromise of the system.

Faulty Access Control

With many types of user roles and rights to access and edit protected data, proper access control becomes a vital part of any secure web application. Currently, access rules are applied to each method that could access sensitive data. This creates repetitive code and makes it easy to forget to apply proper authorization checks.

```
conditions = ["#{Project.table_name}.id
                              IN (#{ids.join(',')})
                              AND #{Project.visible_by}"]
Issue.send( ... :find => { :conditions => conditions})
```

The above example taken from a real security bug in the Redmine application.

Without the Project.visible_by condition, unauthorized users were able to see the issues of private projects. This shows the potential damage of omitting just one authorization check.

Data Policies

To prevent many of the common web application vulnerabilities and to save developers from having to write tedious security-checking code, GuardRails allows developers to define security policies using annotations. These policies are then automatically enforced throughout the application without the need for the developer to write additional code.

Annotation Syntax:

 # @ policy type, fields, expression

- *policy type* represents the nature of the policy, such as whether it pertains to creation of the object, or being allowed to view it.
- *fields* describe which attributes of the object the policy pertains to and can be left blank to describe an entire class.
- *expression* is either a keyword or a function that dictates how the policy is enforced.

Before

The following code snippets are from different files within the application Redmine and all pertain to ensuring that users can only see projects they are authorized to see. Typically , code like this must be scattered throughout the application and forgetting a single one is enough to jeopardize the application's security.

```
named_scope :visible, lambda { { :conditions => Project.visible_by(User.current) } }
```

```
@subprojects = @project.children.visible
```

```
@rows = @project.descendants.visible
```

```
find (:all, :limit => count, :conditions => visible_by(user),
                                        :order => "created_on DESC")
```

After

With GuardRails, all of the repeated checks in the code above are unnecessary. Instead a single annotation is sufficient to apply the visibility policy throughout the entire application.

```
# @ read_access, lambda {|user| Project.visible_by(user)}
class Project…
```

Expressive Taint Tracking

Sanitization policies are combined with taint tracking to prevent injection attacks

GuardRails also attaches sanitization policies to strings themselves. Strings derived from user input or other potentially malicious sources are marked as tainted and are assigned rules as to how they should be sanitized or displayed in different contexts. These rules are then automatically enforced in places where injection attacks might occur, such as in SQL Queries or HTML output.

Annotation	Policy
# @ taint, username, AlphaNumeric	A username can contain only letters and numbers
# @ taint, full_name, NoHTML, TitleTag: LettersAndSpaces	A full name can contain any non-HTML characters, but can only contain letters and spaces in a title tag
# @ taint, profile, BoldItalicUnderline, "//script[@language='javascript']": Invisible	A profile can contain bold, italic, and underline tags, but no other HTML. The string will not be displayed if it appears in javascript tags.

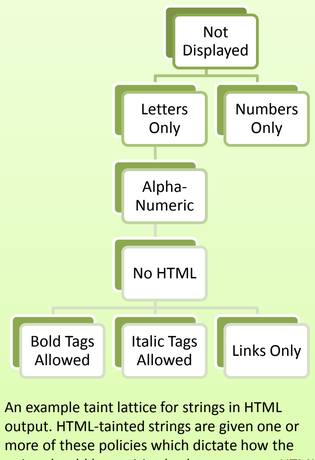
While GuardRails automatically assigns strings a set of default policies, these rules can easily be expanded to include a wide range of custom behavior. Developers can use annotations to specify different sanitization policies for different fields (see examples above). These policies may rely on custom sanitization routines and can be set to behave differently based on the context of where the string appears.

Taint Tracking with Chunks:

Taint tracking of strings typically is handled by having a single bit dictating the taint status for the entire string. This often means that in cases where strings are combined, benign values can be marked as tainted or vice versa. Instead of having one taint bit for the whole string, we divide the string into *chunks*, each of which can have its own taint status.

Concatenation with chunks: "**foo**" + "bar" → "**foo**bar"

On the opposite end of the spectrum, there has been recent work in tracking taint at a character level. Because chunks can be any length, our approach is just as powerful as per character tainting, but less costly. Additionally, our system can support *arbitrary taint information*, including separate taint bits for whether a string is SQL-safe or HTML-safe and rules about how the string needs to be sanitized.



An example taint lattice for strings in HTML output. HTML-tainted strings are given one or more of these policies which dictate how the string should be sanitized when output to HTML.

Authorization Logic

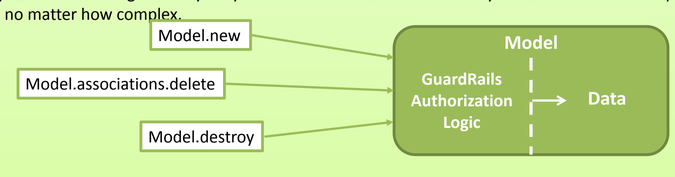
Rules for accessing data are linked directly to the data objects themselves

By associating data policies directly with data models, we can insert checking code to ensure that an object is not being accessed from an improper setting. The object places the authorization logic between itself and any function accessing it by intercepting the getters and setters as well as all other access vectors.

Annotation	Policy
# @ delete access, :admin class Workgroup...	Only administrators can delete Workgroups
# @ write access, password, lambda{|user| user.id == self.id } class User...	A user can only change his/her own password
# @ read access, lambda{|user| self.friends.contains?(user) } class Profile...	Only a user's friends may see his/her profile

Developers can specify access policies by annotating model files with a simple policy specification language. From this specification, GuardRails produces the necessary checking code to implement the policies. This approach ensures that developers no longer have to write convoluted and error prone access control statements all over an application. In addition, the system has an added benefit of making authorization policies easy to find, read, and comprehend.

The policy language used by GuardRails includes keywords to let developers easily specify common policies, but also has a extensive system for including arbitrary ruby code. This ensures that almost any access check can be implemented using GuardRails, no matter how complex.



This work is supported by the Air Force Office of Scientific Research through a MURI award

Guardrail image from www.galaxyfences.com