In *2nd USENIX Conference on Web Application Development* (WebApps 2011), Portland, OR. 15-16 June 2011.

GuardRails: A Data-Centric Web Application Security Framework

Jonathan Burket Patrick Mutchler Michael Weaver Muzzammil Zaveri David Evans
http://guardrails.cs.virginia.edu
University of Virginia

Abstract

Modern web application frameworks have made it easy to create powerful web applications. Developing a secure web application, however, still requires a developer to posses a deep understanding of security vulnerabilities and attacks. Even for experienced developers it is tedious, if not impossible, to find and eliminate all vulnerabilities. This paper presents GuardRails, a source-to-source tool for Ruby on Rails that helps developers build secure web applications. GuardRails works by attaching security policies defined using annotations to the data model itself. GuardRails produces a version of the input application that automatically enforces the specified policies. GuardRails helps developers prevent a myriad of security problems including cross-site scripting attacks and access control violations while providing a large degree of flexibility to support a range of policies and development styles.

1 Introduction

Web application frameworks have streamlined development of web applications in ways that relieve programmers from managing many details like how data is stored in the database and how output pages are generated. Web application frameworks do not, however, provide enough assistance to enable developers to produce secure web applications without a great deal of tedious effort. The goal of this work is to demonstrate that incorporating data-centric policies and automatic enforcement into a web application framework can greatly aid developers in producing secure web applications.

When developing web applications, developers typically have an idea of what security policies they want to enforce. Ranging from which users should have access to which data to how certain pieces of user input should be sanitized, these policies are rarely documented in any formal way. Code for enforcing security policies is scattered throughout the application, making access checks at a variety of locations or sanitizing strings where they might be potentially harmful. This decentralized approach to security makes it difficult, if not impossible, to be certain that all access points and data flow paths are correctly mediated. Further, security policies are rarely completely known from the start; rather they evolve along with the development of the application or in response to new threats. Changing an existing policy can be very difficult, as changes must be made across the entire application.

To alleviate these problems, we developed *GuardRails*, a source-to-source tool for Ruby on Rails applications that centralizes the implementation of security policies. GuardRails works by attaching security policies directly to data, and automatically enforcing these policies throughout the application. Developers define their policies using annotations added to their data model. GuardRails automatically generates the code necessary to enforce the specified policies. The policy decisions are centralized and documented as part of the data model where they are most relevant, and developers do not need to worry about missing security checks or inconsistent enforcement.

Like most web application frameworks, Ruby on Rails provides an object interface to data stored in database tables. This enables GuardRails to attach policies to objects and make those policies persist as data moves between the application and database. The abstract data model provided by Ruby on Rails is one key advantage over systems like DBTaint [4, 18], which have no knowledge of what the relevant data actually represents within the context of the application. While our implementation of GuardRails is specific to Ruby on Rails, we expect most of our approach could also be applied to similar frameworks for other languages.

Contributions. Our major contribution is a new approach for managing web application security focused on attaching policies directly to the data objects they control. We present a tool that demonstrates the effectiveness of this approach using a source-to-source transformation for Ruby on Rails applications. Although our approach applies to many security vulnerabilities, we focus on two of the most common and problematic security issues for web applications:

- To address *access control violations*, we provide an annotation language for specifying access control policies as part of a data model and develop mechanisms to automatically enforce those policies in the resulting application (Section 3).
- To address *injection attacks*, we implement a context-sensitive, fine-grained taint-tracking system and develop a method for automatically applying context and data-specific transformers. The transformers ensure the resulting strings satisfy the security requirements of the use context (Section 4).

To evaluate our approach, we use GuardRails on a set of Ruby on Rails applications and show that it mitigates several known access control and injection bugs with minimal effort (Section 5).

2 Overview

GuardRails takes as input a Ruby on Rails application with annotations added to define security policies on data. It produces as output a new Ruby on Rails application that behaves similarly to the input application, but includes code for enforcing the specified security policies throughout the application.

Design Goals. The primary design goal of GuardRails is to allow developers to specify and automatically enforce data policies in a way that minimizes opportunity for developer error. Hence, our design focuses on allowing developers to write simple, readable annotations and be certain that the intended policies are enforced throughout the application. We assume the developer is a benevolent partner in this goal—since the annotations are provided by the developer we cannot provide any hope of defending against malicious developers. Our focus is on external threats stemming from malicious users sending inputs to the application designed to exploit implementation flaws. This perspective drives our design decisions.

Another overarching design goal is ensuring that no functionality is broken by the GuardRails transformations and that there is no need to modify an application to use GuardRails. This allows developers to add GuardRails annotations to an application slowly or selectively use parts of GuardRails' functionality.

Ruby on Rails. We chose to target Ruby on Rails because it is a popular platform for novice developers. Rails abstracts many details of a web application including database interactions but does not provide robust security support. There is a need for a tool that abstracts data policy enforcement. Rails uses the ActiveRecord design pattern, which abstracts database columns into ordinary object instances. This makes it very easy to attach policies to data objects. In addition, all field access is through getter and setter methods that can be overridden, which provides a convenient way to impose policies.

Source-to-source. Implementing GuardRails as a source-to-source tool rather than using Ruby's metaprogramming features offers several advantages. Some features of GuardRails, like annotation target inference, could not be implemented using metaprogramming. Location-aware annotations are important because they encourage developers to write policy annotations next to object definitions, which further establishes the connection between data and policies. A source-to-source tool also reduces the runtime overhead incurred. Instead of editing many methods and classes during each execution the changes are made once at compile time. Finally, a source-to-source approach can be more effectively applied to frameworks other than Ruby on Rails. We do use some of Ruby's metaprogramming features as an implementation convenience but we believe that all key aspects of GuardRails could implemented for any modern web framework.

3 Access Control Policies

Access control policies are security policies that dictate whether or not a particular principal has the right to perform a given action on some object. For example, a photo gallery application might have a policy to only allow a photo to be deleted by the user who uploaded it. Poorly implemented access control policies are a major security threat to web applications. Unlike content spoofing and cross-site request forgery, however, access control cannot be handled generically since access control policies are necessarily application-specific and data-dependent.

Traditionally, access control policies are implemented in web applications by identifying each function (or worse, each SQL query) that could violate a policy and adding code around the function to check the policy. This approach rarely produces secure web applications since even meticulous developers are likely to miss some needed checks. As applications evolve, it is easy to miss access control checks needed in new functions. Further, this approach leads to duplicated code and functional code cluttered with policy logic. It is difficult to tell what policies have been implemented just by looking at the code, and to tell if the desired properties are enforced throughout the application.

Policy	Annotation
Only admins can delete User objects	**@delete**, *User*, :**admin**, :**to_login**
Only admins can delete the inferred object	**@delete**, :**admin**, :**to_login**
Users can only change their own password	**@edit**, *pswrd*, **self**.*id* == *user.id*, :**to_login**
Log creation of new User objects	**@create**, *User*, *log_function*; **true**, :**nothing**

Table 1: Access policies and corresponding annotations

To address this problem, we propose a major change in the way access control policies are implemented in web applications. Instead of applying data policies to functions, developers define access control policies as part of the data objects they protect. These *data-centric policies* are specified using annotations added to data model declarations. The code needed to enforce the policies is generated automatically throughout the application. By putting policies close to data and automating enforcement, we reduce the burden on the developer and limit the opportunity for implementation error.

3.1 Annotations

To specify access control policies, developers add *access control annotations* of the form:

@<policytype>, *<target>*, *<mediator>*, *<handler>*

to ActiveRecord class definitions.

The four parameters define: (1) which one of the five data access operations the policy concerns (read, edit, append, create, destroy); (2) the object this annotation concerns (either instances of the annotated class or individual variables within the class); (3) the mediator function that checks if the protected action is allowed; and (4) the handler function that is invoked when an unauthorized action is attempted. If the target field is omitted, GuardRails infers the annotation's target from the location of the annotation (see the second annotation in Table 1).

Keywords are defined for mediators and handlers to define common policies. For example, the :**admin** keyword indicates a mediator policy that checks whether the requesting user has administrator privileges (Section 3.2 explains how developers configure GuardRails to identify the current user object and determine if that user is an administrator). In addition to making it easy to translate policies from a design document into annotations, using keywords makes it easy to define and recognize common policies.

Some policies are application and data-specific. To define these more specific policies, developers use a Ruby expression instead of a keyword. The expression is evaluated in the context of the object being accessed so it can access that object (as **self**) as well as any global application state. The expressions also have access to the current user object, which is made visible to the entire application at the beginning of each execution.

Some examples of data policies are shown in Figure 1. Each annotation would be included as a comment in a data model file above the class definition.

Privileged Functions. Our annotation system is flexible enough to support nearly all policies found in web applications. Policies that depend on the execution path to the protected action, however, cannot be specified because the policy functions do not have access to the call stack. We argue that such policies should be avoided whenever possible—they provide less clear restrictions than policies tied solely to the data and global state (e.g., the logged in user) that can be defined using annotations.

One exception is the common "forgot my password" feature that allows an unauthenticated user to reset their password. To a stateless data policy, however, there is no observable difference between a safe password modification using the "forgot my password" routine and an unsafe password modification.

To handle cases like this, we allow developers to mark functions as privileged against certain policies. To create this password edit policy, the developer annotates the password local variable in the User class with the general policy that a users cannot change a password other their own (see Table 1 for the actual annotation) and adds an annotation to the declaration of the "forgot my password" function to indicate that it is privileged. Since we are assuming developers are not malicious, this simple solution seems preferable to the more complex alternative of providing access to the execution stack in data policy annotations.

3.2 Policy Enforcement

GuardRails enforces access policies by transforming the application. It creates *policy mappings*, objects that map data access operations to mediator and handler functions, for each class or variable. All objects in the generated web application contain policy mappings. By default, policy objects map all accesses to True.

To enforce policies correctly, policies must propagate to local variables. For example, if an ActiveRecord object has a serialized object as a local variable, then edits to the local variable should be considered edits to the ActiveRecord object for policy purposes. To address this we compose the container object's policy with the contained object's policy and assign this new policy to the contained object. Most propagation can be done at compile time but some must be done at runtime since we do not know the complete shape of the data structures.

Once the policy objects have been built, GuardRails modifies the data access methods (generally getters and setters) for each class and adds code to call the appropriate policy function. Because all data access in Ruby is done through getter or setter methods, this is sufficient to enforce data access policies. The code below illustrates how this is done for an example variable:

```
alias old_var= var=
def var=(val)
  if eval_policy(:edit) and
       var.eval_policy(:edit)
    old_var=(val)
  end
end
```

Database Access. No database object should be accessed by the application without first checking its read policy. However, it is not possible to determine in advance which objects will be returned by a given query. This means that we cannot check the read policy before a database access. Instead, GuardRails performs the check *after* the object is retrieved but *before* it is accessible to the application. Since we only want to modify the specific application, not the entire Ruby on Rails framework, we cannot modify the database access functions directly. Instead, we take advantage of the fact that all database accesses are done through static methods defined in the ActiveRecord class.

To enforce access policies on database methods, we replace all static references to ActiveRecord classes with *proxy objects*. These objects intercept function calls and pass them to the intended ActiveRecord class. If the result is an ActiveRecord object or list of ActiveRecord objects, the proxy object checks that the result is readable before returning it to the original caller. This allows GuardRails to enforce access policies for all of the database methods without needing to modify the Ruby on Rails framework.

List Violations. An interesting situation arises when a program attempts to access a list of protected objects, some of which it is not permitted to access. GuardRails supports three ways of resolving this situation: treating it as a single violation for the list object; treating each object that violates the policy as a violation individually; or not handling any violations but silently removing the inaccessible objects from the list. Often, the same choice should be used in all cases of a certain access type so we let developers specify which option to use for each access type. The default choice is to silently remove inaccessible objects from lists, following our goal of providing as little disruption as possible to application functionality.

Configuration. To enable GuardRails to support a wide range of applications, it uses a configuration file to specify application-specific details. For example, in order to give the policy functions access to the current user object, the configuration file must specify how to retrieve this object (typically just a function or reference name). Some built-in policy functions also require extra information such as the **:admin** function, which needs to know how to determine if the requesting user is an administrator. The developer provides this information by adding a Ruby function to the configuration file that checks if a user object is an administrator.

3.3 Examples

We illustrate the value of data policy annotations with a few examples from our evaluation applications (see Table 3 for descriptions of the applications).

Read Protected Objects. The Redmine application contained a security flaw where unauthorized users were able to see private project issues. In response to a request for the list of issues for a selected project, the application returns the issues for the desired project and all of its subprojects with no regard to the subproject's access control list. For example, if a public project included a private subproject, all users could see the subproject's issue list by viewing the parent project's issue list.

Adding the single annotation below to the Issue model fixed this bug by guaranteeing that users cannot see private Issue objects:

```
@read, user.memberships.include? self.project,
     :to_login
class Issue
  ...
```

Note that because of the default policy to silently remove inaccessible items from a list, this policy automatically provides the desired functionality without any code modifications.

Edit Protected Attributes. The Spree application contained a security flaw where users could alter the price of a line item in their order by modifying a POST request to include an assignment to the price attribute instead of an assignment to the quantity attribute. This bug appeared because Spree unsafely used the mass assignment function *update_attributes*. Adding a single GuardRails annotation to the *Line_Item* model prevents the price attribute from being changed:

@edit, *price*, false, :**nothing**

To maintain the behavior of the application, the functions that safely modify the price attribute can be marked as privileged against this policy.

4 Context-Sensitive Sanitization

In general, an injection attack works by exploiting a disconnect between how a developer intends for input data to be used and the way it is actually used in the application. Substantial effort has been devoted to devising ways to prevent or mitigate these attacks, primarily by ensuring that no malicious string is allowed to be used in a context where it can cause harm. Despite this, injection attacks remain one of the greatest threats to modern web applications. The Open Wep Application Security Project (OWASP) Top Ten for 2010 lists injection attacks (in general) and cross-site scripting attacks (a form of injection attack) as the top two application security risks for 2010 [16].

Like access control checking, data sanitization is typically scattered throughout the application and can easily be performed in unsafe ways. To prevent a wide range of injection attacks, including SQL injection and cross-site-scripting, GuardRails uses an extensible system of fine-grained taint tracking with context-specific sanitization. Next, we describe how GuardRails maintains fine-grained taint information on data, both as it is used in the program and stored in the database. Section 4.2 describes how context-sensitive transformers protect applications from misusing tainted data.

4.1 Fine-Grained Taint Tracking

Taint tracking is a common and powerful approach to dealing with injection vulnerabilities that has frequently been applied to both web applications [2, 10, 11, 13, 15, 23, 25, 29] and other applications [1, 7, 9, 12, 17, 20, 22, 28]. A taint-tracking system marks data from untrusted sources as *tainted* and keeps track of how tainted information propagates to other objects. Taint tracking may be done dynamically or statically; we only use dynamic taint tracking. In the simplest model, a single taint bit is associated with each object, and every object that is influenced by a tainted object becomes tainted. Object-level dynamic taint-tracking is already implemented in Ruby.

The weakness of this approach, however, is that it is too simplistic to handle the complexity of how strings are manipulated by an application. When tainted strings are concatenated with untainted strings, for example, an object-level tainting system must mark the entire result as tainted. This leads to *over-tainting*, where all of the strings that interact with a tainted string become tainted and the normal functionality of the application is lost even when there is no malicious data [17]. One way to deal with this is to keep track of tainting at a finer granularity. Character-level taint systems, including PHPrevent [13], Chin and Wagner's Java tainting system [2], and RESIN [29], track distinct taint states for individual characters in a string. This solves the concatenation problem by allowing the tainted and untainted characters to coexist in the final resulting string according to their sources, but requires more overhead to keep track of the taint status of every character independently.

GuardRails provides character-level taint-tracking, but instead of recording taint bits for every character individually, groups sequences of adjacent characters into *chunks* with the same taint status. In practice, most strings in web applications exhibit *taint locality* where tainted characters tend to be found adjacent to each other. This allows GuardRails to minimize the amount of space needed to store taint information, while still maintaining the flexibility to track taint at the character level.

In our current implementation, tainting is only done on strings, meaning that if data in a string is converted into another format (other than a character or a string), the taint information will be lost.[1] We believe this decision to be well-justified, as only tracking strings is sufficient assuming a benevolent developer. A malicious developer could easily lose taint information by extracting each character of a string, converting it to its ASCII value (an integer), and then converting the resulting integers back into their ASCII characters and the original string, but such operations are not likely to be present in a non-malicious application.

Our system only marks string objects with taint information, limiting our ability to track *implicit flows*. GuardRails does not prevent the use of tainted strings in choosing what code path to traverse, such as when the contents of a string play a role in a conditional statement. While it is conceivable that an attacker might manipulate input to direct the code in a specific way, we do not believe this risk to be a large one. On the other hand, tracking implicit flows and preventing the use of tainted strings in code decisions can easily break the existing functionality of many applications. Following our design goals and aim to assist benevolent developers in producing more secure applications, it seems justified to ignore the risks of implicit flows.

4.2 Sanitization

The main distinguishing feature of our taint system is the ability to perform arbitrarily complex *transformations* on

[1]In Ruby, characters are simply strings of length one, so taint information is not lost when characters are extracted from strings and manipulated directly.

tainted strings. Rather than stopping with an error when tainted data may be misused, GuardRails provides developers with a way to apply context-sensitive routines to transform tainted data into safe data based on the *use context*. Each chunk in a taint-tracked string includes a reference to a *Transformer* object that applies the appropriate context-specific transformation to the string when it is used. We define a use context as any distinct situation when a string is used such that malicious input could affect the result. Examples of use contexts include SQL queries, HTML output, and HTML output inside a link tag, but programmers can define arbitrarily expressive and precise use contexts. The transformer method takes in a use context and a string and applies the appropriate context-specfic transformation routine to the input.

If a chunk is *untainted*, its Transformer object is the *identity transformer*, which maps every string to itself in every context. Each *tainted* chunk has a Transformer object that may alter the output representation of the string depending on the context. Taint status gives information about the *current* state of a string, whereas the Transformer objects control how the string will be transformed when it is used.

Our goal is to sanitize tainted strings enough to prevent them from being dangerous but avoid having to block them altogether or throw an error message. After all, it is not uncommon for a benign string to contain text that should not be allowed, and simply sanitizing this string resolves the problem without needing to raise any alarms. As with access policies, GuardRails seeks to minimize locations where developers can make mistakes by attaching the sanitization rules directly to the data itself. In this case, chunks of strings contain their own policies as to how they must be sanitized before being used in different contexts, contexts that are automatically established by GuardRails, as discussed in later sections. Default tainting policies prevent standard SQL injection and cross-site scripting attacks, but the transformation system is powerful enough for programmers to define custom routines and contexts to provide richer policies. Weinberger's study of XSS sanitization in different web application frameworks reveals the importance of context-sensitive sanitization with a rich set of contexts and the risks of subtle sanitization bugs when sanitization is not done carefully [26].

Example. Figure 1 shows a simple default Transformer used by GuardRails. If a chunk containing this Transformer is used in a SQL command, the sanitizer associated with the SQL context will be applied to the string to produce the output chunk. The *SQLSanitize* function (defined by GuardRails) returns a version of the chunk that is safe to use in a SQL statement, removing any text that could change the meaning of a SQL command. Similarly, if the chunk is used within a Ruby eval statement, then it will be sanitized with the *Invisible* filter, which always returns an empty string. In HTML, the applied sanitization function differs based on the use context of the string *within* the HTML. Several HTML contexts are predefined, but new contexts can be defined using an XPath expression. The default policy specifies that if a tainted string appears between `<script>` tags, then the string will be removed via the *Invisible* filter. Elsewhere, the *NoHTMLAllowed* function will only strip HTML tags from the string. The sanitization routines used in the figure (*NoHTMLAllowed*, *BoldTagsAllowed*, etc.) are provided by GuardRails, but the sanitization function can be any function that takes a string as input and returns a string as output. Similarly, the context types for HTML (LinkTag, DivTag, etc.) are predefined by GuardRails, but developers can define their own and specify new contexts using XPath expressions (as shown in the script example).

```
{ :HTML =>
  { ”//script” => Invisible,
   :default => NoHTMLAllowed },
 :SQL => SQLSanitize,
 :Ruby_eval => Invisible }
```

Figure 1: Default Transformer

The Transformer approach supports rich contexts with context-specific policies. Contexts can be arbitrarily nested, so we could, for example, apply a different policy to chunks used in an `<img>` tag that is inside a `<div>` tag with a particular attribute compared to chunks used inside other `<img>` tags.

4.2.1 Specifying Sanitization Policies

In many cases, developers want policies that differ from the default policy. Following the format of the data policies, this is done using annotations of the form:

@taint, <*field*>, <*transformer*>

As with the data policies, the *field* component may either be used to specify which field should be marked with this taint status or may be left blank and placed directly above the declaration of a field in the model. The *Transformer* specifies the context hierarchy that maps the context to a sanitization routine within a Transformer object.

One example where such policies are useful is Redmine, a popular application for project management and bug tracking. Its *Project* object has name and description fields. A reasonable policy requires the *name* field to contain only letters and numbers, while the *description* field can contain bold, italics, and underline tags. Redmine uses the RedCloth plugin to allow for HTML-like tags in the Project *description*, but GuardRails makes this both simpler and more systematic allowing developers to use annotations to specify which rules to apply to specific fields. We could specify this using the following annotation:

@taint, {:**HTML** => {:**default** => *BIU_Allowed*}}

This annotation establishes that whenever the string from the *description* field is used in HTML, it should, by default, be sanitized using the *BIU_Allowed* function, which removes all HTML except bold, italics, and underline tags. This :**default** setting replaces the one specified in the default Transformer, but preserves all other context rules, meaning the string will still be removed when used in `<script>` tags, as detailed in Figure 1. If the use context is not already present in the default Transformer, then it will be added at the top as the highest priority rule when a match occurs.

It may be the case, however, that the developer does not want to append to the default Transformer, but overwrite it instead. Adding an exclamation point to the end of a category name specifies that the default rules for this category should not be included automatically, as in the following example:

@taint, {:**HTML**! => {:**default** => *AlphaNumeric*}}

This annotation specifies that in any HTML context the *name* field will be sanitized using *AlphaNumeric*, which removes all characters that are not letters or numbers. As the :**HTML**! keyword was used none of the other HTML context rules will be carried over from the default Transformer. Because the other top-level contexts (such as :**SQL** and :**Ruby_eval**) were not mentioned in the annotation, they will still be included form the default Transformer.

4.2.2 Determining the Use Context

GuardRails inserts additional code throughout the application that applies the appropriate transformers. While our system allows for any number of different use contexts, we focus primarily on dealing with SQL commands and HTML output. We identified all the locations in Rails where SQL commands are executed and HTML is formed into an output page and added calls to the Transformer objects associated with the relevant input strings, passing in the use context.

Many SQL injection vulnerabilities are already eliminated by the original Ruby on Rails framework. By design, Ruby on Rails tries to avoid the use of SQL queries altogether. Using prepared queries when SQL is necessary also helps prevent attacks. Nonetheless, it is still possible to construct a SQL statement that is open to attack. In these scenarios, GuardRails intercepts the SQL commands and sanitizes the tainted chunks using the string's Transformer object.

A more common danger is posed by cross-site scripting vulnerabilities. To ensure that no outgoing HTML contains a potential attack, GuardRails intercepts the final generated HTML before it is sent by the server. The output page is collected into a single string, where each chunk in that string preserves its taint information. GuardRails processes the output page, calling the appropriate transformer for each string chunk. We use Nokogiri [14] to parse the HTML and determine the context in which the chunk is being used in the page. This context information is then passed to the Transformer, which applies the appropriate sanitization routine. The detailed parse tree produced by Nokogiri is what allows for the arbitrarily specific HTML contexts. Note that it is important that the transformations are applied to the HTML chunks in order, as the result of a chunk being transformed earlier in the page may affect the use context of a chunk later in the document. After all of the tainted chunks have been sanitized, the resulting output page is sent to the user. As the entire HTML page must be constructed, then analyzed in its entirety before being sent to the user, this approach may have unacceptable consequences to the latency of processing a request, but could be avoided by more aggressively transmitting partial outputs once they are known to be safe. This is discussed further in Section 5.

4.3 Implementation

GuardRails defines taint propagation rules to keep track of the transformers attached to strings as they move throughout the application. Whenever user input enters the system either through URL parameters, form data, or uploaded files, the content is immediately marked as tainted by assigning it the default Transformer. If the application receives user input in a non-conventional way (e.g. by directly obtaining content from another site), the developer can manually mark these locations to indicate that the data obtained is tainted.

We modify the Ruby String class to contain an additional field used to store taint information about the string. Note that this change is made dynamically by GuardRails within the context of Ruby on Rails and does not involve any modification to the Ruby implementation. A string can contain any number of chunks, so we use an array of pairs, one for each chunk, where the first element represents the last character index of that chunk and the second element is a reference to the corresponding Transformer. For example, the string

`<a href='profile'>`**Joe**`</a>`

generated by concatenating three strings (where underlining represents untainted data and boldface represents tainted data), would be represented using the chunks:

```
[[ 18, <Transformer::Identity>],
 [ 21, <Transformer::Default>],
 [ 25, <Transformer::Identity>]]
```

To maintain taint information as string objects are manipulated, we override many of the methods in the String class, along with some from other related classes such as *RegExp* and *MatchData*. Our goal when deciding how to propagate taint information in these string functions was to be as conservative as possible to minimize the possibility of any exploits. Generally, the string resulting from a string function will be marked as at least as dangerous as the original string. Functions like concatenation preserve the taint status of each input string in the final result, just as one would expect. Other functions like copying and taking substrings also yield the appropriate taint statuses that reflect the taint of the original string. In some cases, discussed in Section 4.3.2, the conservative approach is too restrictive.

4.3.1 Persistent Storage

Web applications often take input from a user, store it in a database, and retrieve and use that information to respond to later requests. This can expose applications to *persistent cross-site scripting* attacks, which rely on malicious strings being stored in the database and later used in HTML responses. Therefore, we need a way to store taint information persistently, as the database is outside of the scope of our Ruby string modifications.

To solve this problem, we use a method similar to that used by RESIN [29] and DBTaint [4]. For every string that is stored in the database, we add an additional column that contains that string's taint information. We then modify the accessors for that string so that when the string is saved to the database, it is broken into its raw content and taint information, and when it is read from the database, both the content and the taint are recombined back into the original string. We also modify several other functions that interact with the database to ensure that the taint information is always connected to the string. This solution makes more sense than serializing the entire object, as it does not disrupt any existing lookup queries made by the application that search for specific text in the database.

4.3.2 Problematic Functions

In our tests, we found that there are some cases where being overly safe can result in overtainting in a way that interferes with the behavior of the application. Our rules are slightly more relaxed in these situations, but only when necessary and the security risk is minimal. Next, we discuss several of these cases and others where determining the appropriate tainting is more complex.

Pattern Substitution. Ruby provides the *sub* and *gsub* procedures that provide regular expression substitution in strings. With these functions, the contents of one chunk affect different parts of the output string in complex ways. If the input string is tainted and the replacement is untainted, then the resulting taint status is ambiguous, as the tainted string affects where the untainted string is placed.

A maximally conservative approach might consider the untainted replacement as tainted, as its location was specifically dictated by the contents of the tainted string. While our is generally to take a conservative approach to tainting, we found in our test applications that this approach frequently leads to overtainting. Hence, we adopt a more relaxed model where output characters are tainted only when they directly result from a tainted chunk. Figure 2 illustrates some examples of how taint information is manipulated in commands such as *gsub*.

Composing Transformers. Another set of special cases are those functions that blend multiple tainted chunks in a way where it is difficult or impossible to keep the resulting taint statuses separate. One such function is *squeeze*, which replaces identical consecutive characters in a string with a single copy of that character (see Figure 2 for examples). If the repeated characters have the same taint status then there is no issue: the resulting single character should also have the same taint status. If, however, each of the repeated characters has a different taint status, the resulting character depends on both inputs. Picking one of the two taint statuses could potentially leave the application vulnerable, so we mix the different taint statues by composing the transformers. A Transformer object can simply be considered a function that takes in a string and a context and returns a sanitized string for that context. This means that we can combine Transformers simply by composing their respective functions. When the composed Transformer is given a string and context, it applies the first Transformer with the given context, then applies the second Transformer with the same context to the result of the first.

As the order in which the two Transformers are applied might affect the final result, we perform the transformations in both possible orders and check that the results are the same. If they are not, then GuardRails acts as conservatively as possible, either throwing an error or emptying the contents of the string. In practice, this issue is not particularly problematic as only a few uncommonly used string functions (*squeeze*, *next*, *succ*, *upto*, and *unpack*) need to compose Transformers and the majority of Transformers produce the same result regardless

String Command	Result
"**foobar**".gsub("o","0")	"**f**00**bar**"
"**medium**".gsub(/(datu|mediu|agendu|bacteriu)m/,"\1a")	"**medi**a"
"uto**pia**".gsub(/(a|e|i|o|u)/) { |x| x.swapcase }	"UtO**pIA**"
"**foot**ball".squeeze	"**fot**bal"
"**bat**tle".squeeze	"**ba***t*le"

Table 2: Example String Commands with Taint
Underlined and bold text in these examples indicate different taint statuses. The second *gsub* example is very similar to the matching used by Ruby on Rails in the *pluralize* function, which converts words to their plural form. The second *squeeze* example demonstrates how taint must be merged in cases where the value of a chunk comes from multiple sources.

of the order in which they are applied.

String Interpolation. One key advantage of taint tracking system employed by GuardRails is that modifies string operations dynamically, with no need to directly alter any Ruby libraries. *String interpolation*, a means of evaluating Ruby code in the middle of a string, is managed by native Ruby C code, however, and cannot be changed dynamically. To resolve this problem, the source-to-source transformation done by GuardRails transforms all instances in the web application where interpolation is used with syntactically equivalent concatenation. Additionally, because Ruby on Rails itself also uses interpolation, GuardRails runs the same source-to-source transformation on the Ruby on Rails code, replacing all uses of interpolation with concatenation.

4.4 Examples

We illustrate how one taint-tracking system eliminates vulnerabilities and simplifies application code by describing a few examples from our evaluation applications.

SQL Injection. Substruct, an e-commerce application, handles most forms safely but one particular set of fields was left vulnerable to SQL injection [21]. The issue lay with the use of the function *update_all*, which performs direct database updates and can easily be used unsafely, as in the following code written by the developers:

```
update_all("value = '#{value}'", "name = '#{name}'")
```

This code directly includes the user-provided *value* string in the SQL command. A malicious user could exploit this to take control of the command and potentially take control of the database.

GuardRails prevents this vulnerability from being exploited. Since *update_all* is known to be a way of passing strings directly into SQL commands, GuardRails modifies the function to transform the provided strings for the SQL use context. As form data is marked automatically with the default Transformer, the potentially harmful strings will be sanitized to remove any dangerous text that might modify the SQL command. Note that for this example, the vulnerability is eliminated by using GuardRails even if no annotations are provided by the developer.

Cross-Site Scripting. Onyx correctly sanitizes input that is used to post comments on images, but does not perform the same checks on input to certain administrator fields. This allows any administrator to inject code into the application to attack application users or other site administrators. In the case of the *site_description* field, simply putting in a double quote as part of the input is enough to break the HTML structure of the resulting pages. These are examples of simple persistent cross-site scripting issues, where the offending string is first saved in the database, then attacks users when later loaded from the database and placed into HTML responses.

Applying GuardRails fixes these cross-site scripting vulnerabilities. Recall that the default Transformer (Figure 1) removes all tags when an unsafe string is used in an HTML context. Thus, when the attacker submits the malicious string in the web form, it is immediately assigned the default Transformer. When that string is later used in HTML, it is sanitized using the *NoHTMLAllowed* filter. The taint information is preserved when strings are saved and loaded from the database (Section 4.3.1), so the vulnerability is not exploitable even though it involves strings read from the database.

5 Evaluation

We conducted a preliminary evaluation of GuardRails by testing it on a variety of Ruby on Rails applications. Table 3 summarizes the test applications. These applications are diverse in terms of complexity and cover a range of application types and design styles. We were able to use GuardRails effectively on all the applications without any modifications.

As detailed in Section 3.3, we have had success at preventing known access control issues with simple policy annotations. Our system of fine-grained taint tracking

Application	Description	Source	Lines of Code
Onyx	image gallery	http://www.hulihanapplications.com/projects/onyx	680
Spree	shopping cart	http://spreecommerce.com/	11561
Substruct	shopping cart	http://code.google.com/p/substruct/	5556
Redmine	project management	http://www.redmine.org/	30747
PaperTracks	publication and citation tracker	developed ourselves	1980

Table 3: Test Applications

also succeeded at blocking SQL injection and cross-site scripting attacks, as explained in Section 4.4.

While performance was not a major design goal, it is still a practical concern. To estimate the overhead imposed by our system, we transformed the image gallery application Onyx with various configurations of GuardRails and measured the average throughput for 50 concurrent users. Table 4 summarizes the results.

As currently implemented, GuardRails does impose a significant performance cost. But, we believe most of this performance overhead is due to limitations of our prototype system rather than intrinsic costs of our approach.

Performing the access control checking decreases throughput by around 25 percent. Most of the performance overhead comes from the code needed to assign policies dynamically. This code is independent of the number of policies, so the performance does not greatly depend on the number of annotated policies. We could reduce this overhead by using static analysis to determine which policies can be assigned statically instead of dynamically.

Taint tracking incurs substantial overhead, reducing throughput by more than 75 percent for some requests. Our taint tracking implementation replaces the native C string implementations provided by Ruby with interpreted Ruby implementations. Since the Ruby string implementations are highly optimized, and interpreting Ruby code is much slower than native C, it is not surprising that this incurs a substantial performance hit. Complex functions like *gsub*, *split*, *delete*, and *slice* require more code to ensure that taint status is handled correctly. The *split* method, for example, took 0.14 seconds to run 400 times without the taint system applied in one test. With the taint system applied, the same test took 0.15 seconds when operating on untainted strings but nearly 5 seconds to split tainted strings. In future work, we hope both to optimize string methods both by rewriting them in C and using more efficient algorithms, and we are optimistic that much of the performance overhead imposed by GuardRails could be eliminated by doing this.

6 Related Work

Much research has been done towards the goal of improving security of web applications and developing access control policies. Here, we review the most closely related work on data policy enforcement and taint tracking.

6.1 Data Policy Enforcement

Aspect-oriented programming is a design paradigm that centralizes code that would normally be spread throughout an application, often referred to as cross-cutting concerns [8]. Data policy enforcement is such a concern and several authors have suggested using aspect-oriented programming to implement security policy enforcement [24, 27]. Like our project, this work seeks to reduce implementation errors and improve readability by centralizing information about security policies.

Automated data policy enforcement is becoming a popular method for preventing security vulnerabilities. Some projects let developers specify data policies, assign the policies to object instances explicitly, and enforce the policies using a runtime system [29, 19]. It is often difficult to define security policies in a clear and concise format. Some projects attempt to remedy this by creating a policy description language [5] while others aim to infer appropriate policies [3] without developer input.

The most similar previous work is RESIN, a tool that enforces developer-specified policies on web applications [29]. GuardRails and RESIN differ in several fundamental ways. RESIN handles policies attached to individual object instances, so developers must manually add the policies on each object instance they want to protect. GuardRails instead associates policies with data models (classes), so the appropriate policy is automatically applied to all instances of the class. Additionally, GuardRails automates much more of the work required to build a secure web application than RESIN does. RESIN requires the developer to write an entire class for each security policy while GuardRails only requires a small annotation.

Transformation Status	Homepage	Login Interface	Image Gallery
Original Application	8.9	9.6	9.2
Access Control Only	7.1	7.1	6.6
Taint Tracking w/o HTML Parsing	2.5	2.8	2.5
Full System	2.0	2.5	2.2

Table 4: Performance Measurements from Onyx
Each number indicates the number of transactions per second for the given request and configuration.

6.2 Taint Tracking

Taint tracking techniques have been used to find format string vulnerabilities [20, 22, 28], prevent buffer overflows [22, 28], improve signature generation [12], and even to track information flow at the operating system level [7]. Several systems, like the GIFT framework [9], are designed, like GuardRails, to be extensible to prevent many types of injection attacks [1, 15]. As mentioned in Section 4.1, some recent research has focused on solving the over/undertainting problem with character-by-character taint tracking [2, 13, 29]. Many systems are limited to using boolean taint states [22, 28] or make use of the compiler, making them difficult to directly apply to a dynamic, interpreted language like Ruby [1, 11].

Similar to our context-specific transformers, the Context-Sensitive String Evaluation (CSSE) [15] system treats tainted strings differently depending on the context of their use. CSSE uses meta-data tags to allow for complex taint statuses. CSSE, however, focuses on propagating information about where the content originated from, with the context-specific code dealing with the tainted strings at the location of their use based on this origin information. The Auto Escape mode in Google's Template System is another similar system that uses different sanitization routines depending on the context of a string in HTML [6]. Without taint-tracking, however, Auto Escape cannot distinguish between safe and unsafe strings without explicit specifications from the developer, so it is necessary to explicitly identify templates that should use auto escape mode.

Other systems do not modify the web application itself or the underlying platform, but instead operate between the application's key entry and exit points. Sekar developed one such tool [18] that records the input received by the application, and later uses *taint inference* in output and database commands to find similar strings that may have been derived from this input. The tool also focuses on looking for changes in syntax of important commands that might be indicative of an injection attack. Another system, DBTaint [4] works outside of the application, helping to preserve arbitrary taint information given from an arbitrary application in the database. Both of these tools have the advantage of being largely platform-independent, and neither needs any application modifications.

7 Conclusion

GuardRails seeks to reduce the effort required to build a secure web application by enforcing security policies defined with the data model, in particular, access control policies and context-sensitive string transformations. The main novelty of GuardRails is the way policies are tied directly to data models which fits developer understanding naturally, provides a large amount of expressiveness, and centralized policies in a way that minimizes the likelihood of missing necessary access control checks. Our early experience with GuardRails provides cause for optimism that application developers can be relieved of much of the tedious and error-prone work typically required to build a secure web application. Although the performance overhead is prohibitive for large scale commercial sites, many web applications can tolerate fairly poor performance. Further, although our current prototype implementation incurs substantial overhead, we believe many of techniques we advocate could be implemented more efficiently if they are more fully integrated into the underlying framework implementation, and that reducing developer effort and mitigating security risk will become increasingly important in rapid web application development.

Availability

GuardRails is available under an open source license from http://guardrails.cs.virginia.edu/.

Acknowledgements

This work was funded in part by grants from the National Science Foundation and a MURI award from the Air Force Office of Scientific Research. The authors thank Armando Fox for his helpful comments and suggestions, and thank Dawn Song, Prateek Saxena, and the attendees at RubyNation for helpful discussions about this work.

References

[1] Chang, W., Streiff, B., and Lin, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 39–50.

[2] Chin, E., and Wagner, D. Efficient Character-level Taint Tracking for Java. In *2009 ACM Workshop on Secure Web Services* (2009).

[3] Dalton, M., Kozyrakis, C., and Zeldovich, N. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 267–282.

[4] Davis, B., and Chen, H. DBTaint: Cross-application Information Flow Tracking via Databases. In *2010 USENIX Conference on Web Application Development* (2010), WebApps'10.

[5] Efstathopoulos, P., and Kohler, E. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 301–313.

[6] Google. Auto escape. http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html, 2010.

[7] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), EuroSys '06, ACM, pp. 29–41.

[8] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming* (1997).

[9] Lam, L. C., and Chiueh, T.-c. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 463–472.

[10] Livshits, V. B., and Lam, M. S. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[11] Nanda, S., Lam, L.-C., and Chiueh, T.-c. Dynamic Multi-process Information Flow Tracking for Web Application Security. In *2007 ACM/IFIP/USENIX International Conference on Middleware Companion* (2007).

[12] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploitson commodity software. In *Proceedings of the 12th AnnualNetwork and Distributed System Security Symposium* (2005), NDSS05.

[13] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically Hardening Web applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing* (2005).

[14] Patterson, A., Dalessio, M., Nutter, C., Arbeo, S., Mahoney, P., and Harada, Y. Nokogiri: an HTML, XML, SAX, and Reader Parser. http://nokogiri.org/, 2008.

[15] Pietraszek, T., and Berghe, C. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds., vol. 3858 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 124–145.

[16] Project, O. W. A. S. OWASP Top 10 — The Ten Most Critical Web Application Security Risks. http://www.owasp.org/index.php/Top_10, 2010.

[17] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy (Oakland)* (2010).

[18] Sekar, R. An Efficient Black-box Technique for Defeating Web Application Attacks. In *16th Annual Network and Distributed System Security Symposium (NDSS)* (2009).

[19] Seo, J., and Lam, M. S. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium* (2010).

[20] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10* (Berkeley, CA, USA, 2001), SSYM'01, USENIX Association, pp. 16–16.

[21] Substruct Developer. Preference.save_settings is insecure. http://code.google.com/p/substruct/issues/detail?id=36, Mar. 2008.

[22] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ASPLOS-XI, ACM, pp. 85–96.

[23] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: Effective Taint Analysis of Web Applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 87–97.

[24] Viega, J., Bloch, J. T., and Ch, P. Applying aspect-oriented programming to security. *Cutter IT Journal 14* (2001), 31–39.

[25] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '07)* (2007).

[26] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., and Song, D. An empirical analysis of xss sanitization in web application frameworks. Tech. Rep. UCB/EECS-2011-11, EECS Department, University of California, Berkeley, Feb 2011.

[27] Win, B. D., Vanhaute, B., and Decke, B. D. Developing secure applications through aspect-oriented programming. *Advances in Network and Distributed Systems Security* (2001), 125–138.

[28] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS '06)* (2006).

[29] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving Application Security with Data Flow Assertions. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).