# Leveraging Convention over Configuration for Static Analysis in Dynamic Languages

David Worth*, Justin Collins†
*Highgroove Studios
123 Krog St., Atlanta, Georgia 30332–0250
Email: dave@highgroove.com
†Twitter
Email: @presidentbeef

***Abstract*—Static analysis in dynamic languages is a well known difficult problem in computer science, with a great deal of emphasis being put on type inference [1]. The problem is so difficult that Holkner and Harland's paper on static analysis in Python opens immediately with, "The Python programming language is typical among dynamic languages in that programs written in it are not susceptible to static analysis." [2] Dynamic languages such as Ruby provide impressive programming power thanks to expressive language constructs and flexible typing. Ruby, in particular, is strongly leveraged in the web development ecosystems thanks to well known and supported frameworks such as Ruby on Rails and Sinatra.**

**Web application security is a particularly difficult area for a number of reasons including, the low-barrier to entry for new developers combined with the high-demand for their services, the increasing complexity of the web-based ecosystem, and the traditional languages and frameworks for web-development not adopting a strong defensive stance as their default. Ruby on Rails adopts the "convention over configuration" policy aimed at aiding developers of all levels in building robust web applications with a minimum of configuration. The goal is for the framework to simply "do the right thing" by default, and more sophisticated features and technologies are to be explicitly applied by developers with those more advanced requirements and understanding. Much of the power in the Ruby on Rails framework stems from careful use of "magic" functions: dynamically generated functions using Ruby's powerful metaprogramming structures. As a side effect, many of the methods called by developers are not available to a static analysis tool by simply examining the code on disk. We are able to leverage the consistency of the language and framework to perform static analysis on Ruby on Rails applications, and reason about their attack surface. This is done by analyzing the abstract syntax tree, and sometimes the configuration (generally simply library versions) of the program itself and by comparing it to a pre-compiled library of known security issues exposed by the Ruby on Rails framework.**

## I. Introduction

Ruby on Rails is a popular web framework which provides a Model-View-Controller architecture along with many ancillary tools to engineer complicated web applications with a minimum of code as well as a minimum of exposed complexity. The fundamental principal employed to reduce complexity is "convention over configuration", meaning that standardized methods are used to achieve standard functionality. In many ways this philosophy resembles that of the Python community and its "There's only one way to do it" philosophy. The advantage to such a philosophy is that one can successfully rely on the conventions to expose large families of security vulnerabilities present in modern web applications. The Ruby on Rails Security Guide [5] lists a comprehensive collection of general web application security vulnerabilities, and a number of Rails-specific vulnerabilities, and their mitigations as provided for by the framework. Justin Collins, the original author of the Brakeman Scanner for Ruby on Rails applications, exploited exactly this convention-based approach, and these Rails-specific issues, in designing the scanner.

## II. Brakeman Scanner Architecture

At a high level Brakeman treats Ruby as an "acceptable Lisp" and uses existing parsers to decompose the code into S-expressions [7]. Each s-expression can then be interrogated for its type, for example a function definition, a collection of arguments, a Ruby block, a function call, or a string interpolation. These basic building blocks forming an abstract syntax tree are the fundamental objects used by Brakeman, along with its knowledge of Rails conventions, to analyze a given Rails application for potential security vulnerabilities.

Moreover, some basic "taint flow analysis" can be performed based on the conventions within Rails. The means by which user-input enters the system is fairly consistent, with three of the major sources being the cookie collection, a parameters hash available in, and used as the main source of input to, controller actions, and the request object which wraps up the context of a given request to the application. Due to the consistency of the sources we may reason concisely about the danger presented by relying upon input from those sources directly in contexts which have any potential security implications.

## III. Insecure User Input

The standard adage of "never trust user(-controlled) input" is easily proffered but one of the fundamental realities of developing applications is that we *must* act upon their data in some way. As such it is key to limit the ways in which we trust user-controlled data rather than simply attempting to avoid the problem.

### A. SQLi Detection

SQL Injection (SQLi), along with cross-site scripting (XSS) and cross-site request forgery (CSRF), represent the

most well-known, common, and language independent web vulnerability classes. Ruby on Rails, like any other web framework, can be made susceptible to such flaws, though it does provide a number of mitigations and best practices to prevent them. Moreover, many of the methods which automatically generate SQL are not explicitly defined by either Ruby on Rails itself or by the user, but rather are automatically generated at runtime by Rails. The generation of these methods is based on interrogation of the database which backs a given model wherein each column in the database is mapped to an attribute of the model via the Object Relational Model (ORM). Thus for a given user model with two database columns "name" and "email", the User class itself responds to a variety of methods, some explicitly defined such as `User.all`, which returns all users in the system, and dynamic methods based on its attributes such as `User.find_by_name`, `User.find_by_email`, `User.find_by_name_and_email`, and even `User.find_all_by_name_and_email`. While using these finders generates appropriate SQL automatically, and the parameters themselves are correctly isolated, this is a perfect example of Rails conventions that are so consistent they can be trivially identified in a static analysis context.

A naive approach one might take is to identify these finders is with a simple regular expression. Brakeman opts to use the abstract syntax tree from a Ruby parser as it provides a deeper view into the application, but the above would work quickly for a given model. This consistent, convention-based, dynamic function generation is easily and deterministically identified allowing for further reasoning about the functionality of the code.

Compounding slightly the complexity of isolating SQLi is that the parameters to various finders may very well contain user-controlled data, in this case passed in via a parameters hash(-table). Thus a simple, incorrect and explicitly recommended against [5], idiom for dynamically finding a user by name would be `User.where(``name = #{params[user_name]}'')`. During the process of analysis we are able to reason simply about the potential safety of such an expression. The call signature of the `where` method on an active record model only has a few different permutations. In the case where the first variable is a string it is to be interpreted as "raw" SQL. By examining the s-expression of the first parameter we can quickly determine if *any* (potentially user-controller) variable is interpolated, and if so it is flagged as potentially dangerous. In the case where the user cannot control the data the flagged expression may not be exploitable but is still an indication of an expression that should be refactored to follow the best-practice of passing parameters to the ORM outside of the query logic [6].

If a programmer attempts a "clever" independent re-implementation of such dynamic finders as described above, it is fundamentally more difficult to reason about the safety of a given expression. This is because there are no well understood and accepted conventions about call signatures and handling as have been created during the development of the Ruby on Rails framework. Those conventions alone allow us to reason concretely about the safety of a given expression.

### B. Insecure Redirect and Rendering

A common idiom for concluding a controller action in a Ruby on Rails application is to set some sort of notification to be displayed to the user, then a destination is specified. The destination to which the user is redirected is specified by an options hash, and must not be user-controlled. If the user does control the parameters to the redirect call then they may manipulate the `host` attribute, which can be used to send a user, after completing the initial action, to a remote malicious host [8]. The user will likely trust a given link to a trusted site without realizing the host manipulation is occurring. To increase the confidence of the victim, the attacker may obfuscate the vulnerability by making the site to which they redirect them look identical to the trusted site if they control the destination. Identification of such manipulation at runtime is simply done by identifying any variable parameters to the redirect. In that case the redirect is flagged as vulnerable. The Rails framework provides mitigations for such redirection via the `only_path` option, which can be easily applied by a developer upon being alerted to the vulnerability.

Similarly, allowing a user to control the view or template to be rendered in a given action may result in an information leak about the system by effectively bypassing any authorization controls in place. Rendering within Rails is done via the `render` [10] method, which expects to receive the name of a partial as a string and an options hash. The path to the rendered view template should never contain user-controlled parameters to avoid this vulnerability. Validating that it does not simply means checking for string interpolation in the first parameter, and checking for its source. If the source is the cookie collection, the parameters hash as created by Rails, or the request object then the source is deemed to be tainted and is flagged. Much like the taint-safety of Perl, once a variable from one of those sources has been manipulated by the developer it is deemed untainted and can no longer be reasoned about.

## IV. Explicit Access Controls via Exposed APIs and DSLs

Much of the Rails architecture revolves around explicit APIs, exposed to the user to perform various actions, along with domain-specific languages. Domain-specific languages (DSLs) are Ruby code which exploit the syntactic flexibility of Ruby itself to resemble a unique language intended to express programmatic needs of the developer.

### A. Mass Assignment

After the well-publicized compromise of Github by Egor Homakov [9], the subject of Mass Assignment vulnerabilities within the Rails framework came, once again, to the fore. This vulnerability had been well understood within the community, and as such standardized mitigations had been created and well documented. The mitigations themselves

were active mitigations requiring understanding of the domain models to implement. The goal was to make such mitigations simple enough they could be implemented quite quickly with fundamental understanding of the models and the use of their attributes. With the Rails 3.2.6 release, mass-assignment has been disabled in development in such a way that an exception is thrown if mass-assignment restrictions are violated at runtime as an aid to developers. The vulnerability is triggered via the very convenient `update_attributes` method which is provided to simultaneously update multiple attributes on a model. This may optionally include so-called "nested-attributes", those attributes belonging to domain models associated with the model being updated explicitly, via a single method call. This update is done without regard to the nature of those attributes, as Rails does not attempt to reason about the use of any attribute. For example, a user may have a boolean attribute to indicate the user is an administrator. Other attributes may be explicitly exposed to the user through a web-form such as "real name" and "home town". In this case the user themself might manipulate that web-form's fields to include an administrator attribute of their own, as if it were provided by the developer, effectively making themselves an administrator.

While current static analysis tools cannot prevent the above scenario, as they cannot reason about the relative sensitivity of various fields within the system, they can reveal those models which lack explicit white- or black-listing of attributes for update. Rails provides two mechanisms for attribute access control: white-listing via `attr_accessible`, and black-listing, the considerably less preferable route, via `attr_protected`. The latter mechanism is often used in "kitchen-sink" models which have so many attributes such that explicitly white-listing those attributes which should be writeable is tedious for the developer, so a black-listing of the few sensitive attributes is provided. The `attr_accessible` and `attr_protected` "keywords" are actually methods in a sophisticated DSL. We can identify calls to these methods, and from their existence, infer that the developers of a given application have at least taken steps to limit access to model attributes, and hopefully, reasoned about their relative sensitivity. Though this is not a foolproof methodology, it does indicate to a developer or external auditor those models which have not explicitly made public declarations about the sensitivity of their underlying attributes. Moreover, those models which are of sufficient complexity to warrant a black-listing via `attr_protected` may warrant a significant refactor into smaller models, each of which can have explicit access controls applied to them. In this case the mere existence of the black-list indicates a "pain-point" ripe for a refactor to improve the architecture and security of the application in one pass.

## V. External library conventions

Not only does the Rails framework itself encourage standardization in the methods by which various pieces of functionality are achieved, but the entire ecosystem has evolved with the same philosophy in mind. Some libraries and executable gems such as Bundler and Rake, have become so pervasive as to be considered an integral part of developing a modern Rails application.

### A. Bundler

With the introduction of the Bundler gem the process of creating, updating, and maintaining library dependencies for Rails applications, a process which had previously been notoriously difficult, has become trivial. Bundler interrogates a `Gemfile` and determines which versions of a given gem and each of its dependencies (recursively) are required. For example, an application may require a gem by name, by name and explicit version (useful for specifying a bug-fix level of a gem upon which an application relies), or an optimistic version which will never be less than the specified version, but may choose minor versions up to the next major version of a gem. These options allow the system to avoid API changes that may be introduced into a well semantically-versioned gem while staying abreast of bugfixes. The results of the dependency resolution process are stored in a `Gemfile.lock` which can be interrogated for specific vulnerabilities. This interrogation process does rely on some sort of external library of vulnerabilities but does have some strong advantages. The Rails framework itself is installed as a gem from the `Gemfile`. As vulnerabilities are found within Rails itself, CVEs are created, and new versions are released, the Brakeman scanner can be updated to check that a given application is not vulnerable to those bugs fixed in various patch releases. This is not formal "static analysis" in the context of analyzing executable code but does follow the same philosophy of simply relying on developers and tools to act consistently and openly such that the code produced may be checked for correctness with a maximum of ease.

### B. Extension of Rails by Domain-Specific Languages

Much of Rails which appears at first glance to be made of "keywords", as a developer might be experienced with from other languages, is actually a function call within the current class (or one of its ancestors). Examples of this are the `private` [11] "keyword" for marking methods as inaccessible to call from outside of the current class or module, and `attr_accessible` and `attr_protected` for limiting access to a models attributes. Thus, writing specific handlers to analyze the parameters to these apparent keywords is no different than writing handlers for an other function call. Moreover, this means that extending Brakeman to support external gems which introduce DSL methods of their own is of the same difficulty as implementing handlers for the attribute control methods. For example, the MetaWhere gem by Ernie Miller which exposes some attribute-level searching of ActiveRecord models, provides the `assoc_searchable` and `assoc_unsearchable` methods to white and blacklist attribute in much the same fashion as Rails attribute access methods.

## VI. Conclusion

By leveraging conventions within modern languages and frameworks, one can more easily write sophisticated static analysis tools than one might in a *laissez-faire* language which encourages each developer to implement fundamental operations on their own. Following the conventions, we are able to reason strongly about a developer's use of given methods and constructs with a minimum of overhead, and often without resorting to extremely low-level analysis. Ruby's type-system, while being intentionally flexible, lends itself to analysis of its abstract syntax tree, and thus to determining when various pieces of data may be user-controlled and when not. Just as the Rails framework continues to evolve, so does the Brakeman scanner with constant improvements in identifying security vulnerabilities in an extremely popular web framework.

## Acknowledgments

The authors would like to thank the entire Brakeman Scanner team for their support in improving the scanner constantly. Moreover, David Worth would like to thank the project's maintainer and original implementor Justin Collins for his encouragement to contribute to and participate in the project via contributions to, and papers such as this.

## References

[1] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1859–1866. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529700 1

[2] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of python applications," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ser. ACSC '09. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, pp. 19–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1862659.1862665 1

[3] Google Scholar results for "Static Type Inference for Ruby"

[4] Justin Collins. *BrakemanScanner* [Online]. Available: http://brakemanscanner.org (URL)

[5] *Ruby on Rails Security Guide* [Online]. http://guides.rubyonrails.com/security.html (URL) 1, 2

[6] *Ruby on Rails Security Guide* [Online]. http://guides.rubyonrails.org/security.html#sql-injection (URL) 2

[7] Magnus Holm. (2010, February 4). *Sexp for Rubyists* [Online]. http://blog.rubybestpractices.com/posts/judofyr/sexp-for-rubyists.html (URL) 1

[8] *OWASP Top 10 - Redirect* [Online]. https://www.owasp.org/index.php/Top_10_2010-A10 (URL) 2

[9] Egor Homakov (2012, March 4) *wow how come I commit in master? O_o* [Online]. https://github.com/rails/rails/commit/b83965785db1eec019edf1fc272b1aa393e6dc57 (URL) 2

[10] *Rails API Documentation* - `ActionView::Template#render` [Online]. http://apidock.com/rails/ActionView/Template/render (URL) 2

[11] *Rails API Documentation* - `Module::private` [Online]. http://apidock.com/ruby/Module/private (URL) 3