Leveraging Convention over Configuration for Static Analysis in Dynamic Languages

David Worth*, Justin Collins†
*Highgroove Studios
112 Krog St, Suite 6 Atlanta, GA 30307
Email: dave@highgroove.com Twitter: @highgroovedave
†Twitter
Email: collins@twitter.com
Twitter: @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.

A. Static Analysis of Ruby

The fundamental idea behind Brakeman is that Rails is a framework, or a domain specific language (DSL), for web development. As such, tokens within Ruby and Rails which appear to be analogous to "keywords" of other languages are actually method calls in Ruby. Rails is often attributed with "magic" functionality which to an experienced Rails developer is a side effect of a well thought out, non-intrusive, DSL.

To convert a Ruby program, or Rails application, into S-Expressions for analysis we use the combination of Racc [12] and RubyParser [13]. Racc is a parser generator, analogous to Yacc [15] for converting grammars into parsers for the grammar definition. RubyParser provides two grammars, one each for the unique syntactical structures of Ruby 1.8 and 1.9. RubyParser then relies on the parsers generated by Racc to provide generate S-Expressions upon which we can operate. The node-types in the S-Expressions produced by RubyParser are the node types as defined by Ruby [16] in its virtual machine.

It is interesting to note that the Ruby Parser and any given Ruby implementation may actually differ in their interpretation or parsing of a given program, as could any two implementations. The Ruby ecosystem is fairly unique in that the definitive reference for correctness of Ruby is Matz Ruby Implementation (MRI), as the implementation moves very quickly and the definitive RubySpec has not been fully translated out of Japanese. An advantage of RubyParser being implemented separately from a given implementation is that it is not susceptible to any implementation bugs of a given implementation; By the same token, it may implement its own.

SexpProcessor [14] provides a very flexible, convention based S-Expression processing framework. An S-Expression Processor inherits from the SexpProcessor class and simply defines methods of the form $process\_\langle\ ruby\ node\ type\rangle$ and $rewrite\_\langle\ ruby\ node\ type\rangle$.

Brakeman extends SexpProcessor's building in processor base-class to quickly capture important, and potentially vulnerable, points of execution context. It does this in two phases: a pre-processing collection phase, during which potentially vulnerable s-expressions are collected based on the type of vulnerability being checked for, and an analysis phase where the pre-stored S-Expressions are analyzed for actual vulnerabilities, and if one is found, it is rated in its confidence.

For example, in pre-processing templates, for Cross-Site Scripting it examines local variable assignments, and any point where there is an output from a Ruby expression. It is able to collect those expressions via a call to $process\_output$, and store them for later analysis. Later, in the analysis phase, we retrieve the output points and check them for user controlled data. It stores the results in a struct which contains the type of user controlled data along with the S-Expression which represents the vulnerable code.

Finally these matches can be analyzed individually for the actual vulnerabilities, and classify them appropriately in preparation for the reporting phase. This analysis phase requires walking the vulnerable S-Expressions and interrogating them for use of particular variables, or copies of particular variables. For example, the data-structure which carries the context of a given HTTP request in Rails is the `params` hash. If this hash is output directly without validation the program is vulnerable to a cross-site scripting vulnerability, which is recorded and reported. This is the power of convention within a web framework such as Rails. If there were no convention, but rather programmer defined naming schemes for request parameters, a great deal more work would be required in terms of data-flow analysis and taint-checking to determine if a program is susceptible to cross-site scripting. By leaning upon best practices and convention we can easily reason about the safety, or lack there-of, of various expressions.

B. The General Case

Given a language with a well-defined grammar and a parser, we can construct an abstract syntax tree (AST) for a program written in that language. Examining that AST will allow us to reason about the properties of the given program. The strength of Brakeman comes less from the nature of Ruby and its abstract syntax tree, or its representation as S-Expressions, but rather from leveraging convention over configuration. One such convention, mentioned above, is that all HTTP request parameters are exposed in a consistently named variable: `params` within a controller action handling the request. In this case, there is nothing unique about Ruby or Rails in our ability to reason about the contents of `params`.

The Python language includes in its Standard Library the necessary tools for parsing Python and interrogating its abstract syntax tree via the "Python Language Services" [18] tools. The Grok [17] project touts itself as a convention over configuration web framework with much of the same functionality as Rails. With this pair of powerful tools similar static analysis should be possible to that which we perform in the Brakeman scanner. In examining the constructs provided by Grok there are some differences which make such analysis more difficult than in the Ruby/Rails case. For example, HTTP request parameters are named by the developer rather than by the framework, so detecting XSS, for example, does require some variety of data-flow analysis and taint-checking. This short-coming is not a short-coming of Grok as a powerful framework, but rather a failing of its *convention* based design. Such a framework could be driven towards a more convention based design to enhance its ability to be automatically analyzed and reasoned about.

The Groovy [19] programming language is a dynamic language built on the Java Virtual Machine built with the goals of implementing Ruby style DSLs and other dynamic language techniques with the advantages of static types and compilation. The Grails [20] web framework is a convention over configuration based web framework meant to very closely resemble the Rails framework. Its resemblance to Rails is so strong in fact that many of the language constructs appear to be copied directly from Rails into Groovy with the most minimal syntax changes possible. As such, HTTP request parameters are passed to controller actions as `params`, and thus their use can be reasoned about in a similar fashion as we do in Brakeman for Rails applications. The CodeNarc [21] project provides some static analysis tools for Groovy, though it is aimed at best-practices, and other Groovy specific issues. It would be an interesting exercise for the Groovy/Grails community to port some of the functionality from Brakeman to CodeNarc in an effort to enhance the security of Grails applications.

While the above comparisons focus on a single, fairly trivial use of convention over configuration, specifically the use of a single name for the HTTP parameters passed to a given controller action, it is illustrative. If all system inputs are provided in easily recognized, and automatically constructed fashions, then the onus of identification of potentially dangerous data within an application is taken from the developer or their tools. The focus of tools can then be placed on other functionality such as detecting dangerous uses of said data.

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.

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 susceptible to such flaws, though it does provide a number of mitigations and best practices to prevent them.

A. SQLi Detection

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. Cross-Site Scripting

Within a given Rails application there are two sources of potential Cross-Site Scripting vulnerabilities: Rails itself and user code. Brakeman is Rails-version aware, and adds checks for vulnerabilities within the Rails framework as they are made public. When such vulnerabilities are created with a CVE, as they generally are, the warnings produced reference the given CVE and the minimum version to which a developer must upgrade in order to no longer be vulnerable to the given bug.

The second phase of detecting XSS, briefly outlined above, involves parsing the templates within the system and examining them for outputs. Each templating language may have its own mechanism for denoting outputs but two of the most common, Erb, and Haml, denote output as following a single = token. By identifying all of the points where output is produced from a Ruby expression we can focus on identifying potentially dangerous inputs. This is the pre-processing phase discussed in general above. Each new templating language introduced to the Rails ecosystem must have a custom parser implemented to support such scanning since their semantics may vary radically.

The bulk of the work in the analysis stage is performed by the `has_immediate_user_input?` and `has_immediate_model?` predicates, which are called on all un-escaped output nodes detected by the scanner. The first of these predicates recursively walks the entire expressions looking for access to the `params`, `cookies`, or `request` hashes, as well as string interpolations and other potential sources of user-controlled data. Upon finding them a Match struct is returned with the vulnerable expression. The second predicate simply looks for calls directly on a model class. For

example, if `Model.find_by_dangerous_attribute` is called directly with user-controlled input, then a user may circumvent security mechanisms or cause the system to leak secure data. This may be converted to a XSS if the returned data can be forced by the user to refer to a given model stored in the database which they control. For example, if the user is able to store dangerous Javascript in their profile in a social-networking site, then with a carefully crafted URL they might force another user to view their profile which executes that Javascript.

Note that in the case of Rails 3.x, output in views is automatically escaped so the checks become relatively trivial as the built-in escaping method is called automatically. With that in mind, there have been XSS vulnerabilities within Rails itself, and the new mitigations should not be viewed as a claim that Rails is somehow immune to this classic vulnerability.

C. 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 strips all protocol, host, and port information from the emitted URL, leaving only the resource path behind. This mitigation can be easily and quickly be 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, 3

[6] *Ruby on Rails Security Guide* [Online]. http://guides.rubyonrails.org/security.html#sql-injection (URL) 3

[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) 4
[9] Egor Homakov (2012, March 4) *wow how come I commit in master? O_o* [Online]. https://github.com/rails/rails/commit/b83965785db1eec019edf1fc272b1aa393e6dc57 (URL) 4
[10] *Rails API Documentation* - `ActionView::Template#render` [Online]. http://apidock.com/rails/ActionView/Template/render (URL) 4
[11] *Rails API Documentation* - `Module::private` [Online]. http://apidock.com/ruby/Module/private (URL) 5
[12] Aaron Patterson. *Racc* [Online]. Available: https://github.com/tenderlove/racc (URL) 1
[13] SeattleRB (Ryan Davis). *RubyParser* [Online]. Available: https://github.com/seattlerb/ruby_parser (URL) 2
[14] SeattleRB (Ryan Davis). *SexpProcessor* [Online]. Available: https://github.com/seattlerb/sexp_processor (URL) 2
[15] Stephen C Johnson. *Yacc* [Online]. Available: http://dinosaur.compilertools.net/yacc/index.html (URL) 2
[16] Ruby core team. *Ruby source code* [Online]. Available: https://github.com/ruby/ruby/blob/trunk/node.c#L100-886 (URL) 2
[17] Grok core team. *Grok* [Online]. Available: http://grok.zope.org/ (URL) 2
[18] Python core team. *Python Language Services* [Online]. Available: http://docs.python.org/library/language.html (URL) 2
[19] Groovy core team. *Groovy* [Online]. Available: http://groovy.codehaus.org/ (URL) 2
[20] Grails core team. *Grails* [Online]. Available: http://grails.org/ (URL) 2
[21] CodeNarc team. *CodeNarc* [Online]. Available: http://codenarc.sourceforge.net/ (URL) 2