Ruby, Rails, and Gems developer tools

Security Resources

Great tools and resources are available to help you write secure Ruby on Rails code. Kurt examines some tools and offers some tips. By Kurt Seifried

I'll admit it: I'm old, and increasingly I find tech trends that surprise me. Ruby on Rails is one of those trends that it seems everyone is using now. A friend gave me a good explanation of why Ruby on Rails has sneaked up on me, despite the fact that it's been around for a decade.

Ruby on Rails is a great prototype system, so it's easy to sell in the sense of "why don't we make a rapid prototype using Rails?" But, of course, once you have a working prototype, you also have something you can sell to customers. So, given a choice between writing a new "real" product or tidying up the prototype and shipping it, people usually pick the second option. Fortunately, Ruby on Rails actually allows for scaling and other production issues, so you can usually get away with this approach.

But, as with any language and framework, the majority of developers using Ruby on Rails are paid to make working code – not secure code. This is unfortunate, because several great resources and tools can help you write secure Ruby on Rails code, and that's where Ruby really shines.

Built-In Tools

Ruby on Rails offers a number of built-in security features that are all too often ignored or not used correctly. The first thing Rails can help you do securely is session handling. In my experience, almost everyone that tries to roll their own session handling code fails. You need to create secure session IDs, you need to hand them to the client securely (which usually means cookies), you need to protect them from JavaScript shenanigans, you need to expire them, and you need to prevent things like replay attacks.

Rail's built-in session handling code does all these things. It also offers some nifty features like encoding all the access data into the cookie so that you do not need a back-end storage mechanism to keep copies of all the cookies and their state. This feature can also allow you to scale applications. Because there is no need for shared back-end hosting of the session states, you can load balance sessions and move them from one server to another with no problem (as long as the cookie is valid). Rails can also enforce the use of SSL for an application by setting:

```
config.force_ssl = true
```

It really is that easy. Now, if you accidentally forgot to redirect or block standard HTTP connections, your application will force the use of HTTPS.

Rails also has cross-site request forgery (CSRF) [1] protection capabilities for POST requests, which is what you should be using if you need to pass data

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

back to the server. Simply enable CSRF protection with:

```
protect_from_forgery :secret => ⮐
  "123456789012345..."
```

Using Safe RubyGems

Of course, you probably don't write all of your Ruby code from scratch. You use gems. As with CPAN, PEAR, Hackage, and PyPi, it's almost always better to use a well-engineered wheel rather than reinvent your own. However, the bad news is that, like any software, many RubyGems contain security flaws. So, you might depend on a gem with no security issues right now, but in the future, who knows? Asking users of your program to check each gem to make sure it has no security flaws is probably not the best approach. Instead, you can use bundler-audit [2] to audit the code within your project. It uses the RubySec database, which is basically a list of security advisories for gem files hosted on rubygems.org. The RubySec people are now tracking several sources of security information, including the Open Source Security List and the Open Source Vulnerability Database.

Sanitize Everything

Speaking of RubyGems, which ones should you be using? Why the ones that provide data sanitization, of course. Like most software, in order to do something useful, Ruby on Rails applications need to take user input and then do something with it. The bad news is that, in the modern world, most of the useful file types and data formats are increasingly complex (XML, multipart email, web pages, etc.). Rather than try to clean everything yourself, you can choose from a number of RubyGems that are available. My advice is to go to rubygems.org and search for keywords like "sanitize" and whatever it is you're trying to clean up (e.g., "html" or "mail").

Embedded Code

One piece of good news is that RubyGems generally don't embed copies of other gems' code. For example, in the C/C++ world, quite a few open source programs embed copies of XML parsers like `libxml2` and `expat`. This practice ensures the program can be compiled and made to work on systems with broken or no `libxml2/expat` (e.g., Windows). However, this can cause problems because these embedded copies rarely get updated, and you end up with out-of-date software that contains security flaws.

Luckily, in the Ruby world, this practice doesn't happen as much, but there are still some RubyGems (e.g., `crack`, which is required by `webmock`) that copy code from Rails. A lot of Ruby on Rails applications are not so disciplined and do use copies of code from other projects, which is fine if the open source license allows it. So, this is something to be mindful of – and something that makes me sorry that Google code search is gone.

Brakeman

Brakeman [3] is a great static analysis tool. Basically, it scans all your Ruby source code for a wide variety of potential problems ranging from mass assignments (which can be a problem if an attacker inserts a custom value like `admin = true`) to SQL injections and specific known security flaws in code. Installing Brakeman is trivial:

```
gem install brakeman
```

To run it, you simply `cd` into the directory of your Rails applications and run `brakeman`.

Brakeman can also be called as a library; for example, following the agile/continuous development model, you could have Brakeman automatically run as part of your tests to ensure that nothing bad has happened. By comparing it to previous runs, you can quickly eliminate false positives and zero in on potential problems. You might also consider some other gems, like `flog` and `flay`. The `flay` gem analyzes code for similarities, ignoring things like white space and formatting. I especially recommend it for finding embedded code, as mentioned above in the "Embedded Code" section. The `flog` gem analyzes code for readability and "pain" (i.e., convoluted spaghetti code). In general, your code should not be painful to read.

Ruby and Rails Security Docs

Several excellent Ruby and Ruby on Rails security resources are available, such as the Ruby on Rails Security Guide website [4] and the *Ruby on Rails Security* book [5], which you can purchase online or download for free. It's a bit old (the last update was in 2008), but the basics still apply. Additionally, you can check out the Ruby on Rails Cheatsheet [6], which is basically an up-to-date subset of the security book.

Practicing Security Response

One consistent problem I've noticed among many smaller open source projects is that, generally, they are not very experienced at handling security issues. This is not always a bad thing; in general, the quality of open source code is high enough that most projects don't have to deal with a lot of security flaws, but a handy site for learning more is the Security Release Practice [7] project on GitHub; you can clone it and practice fixing security issues.

Conclusion

Ruby and Ruby on Rails are starting to mature, and some excellent security resources are available for developers. However, as with all security tools, they don't help if people don't use them.

If you're using Ruby or building and using Ruby on Rails applications, my advice is at least to run your apps through the Brakeman analysis tool and deal with the issues that arise.

Of course, if you want to learn more, the security resources can help guide you. You can also check out *#rubysec* on *irc.freenode.net* for additional security pointers. ■■■

INFO

[1] Attack of the CSRF: *http://www.linuxpromagazine.com/Issues/2009/99/Security-Lessons/*

[2] Bundler-audit: *https://github.com/postmodern/bundler-audit*

[3] Brakeman: *http://brakemanscanner.org/*

[4] Ruby on Rails Security Guide: *http://guides.rubyonrails.org/security.html*

[5] *Ruby on Rails Security*: *http://www.rorsecurity.info/the-book/*

[6] OWASP Ruby on Rails Cheat Sheet: *https://www.owasp.org/index.php/Ruby_on_Rails_Cheatsheet*

[7] Security Release Practice: *https://github.com/steveklabnik/security_release_practice*