IMMUNIO

Securing Ruby on Rails Web Applications

What You Need to Know

authored by Mike Milner
for **IMMUNIO**

IMMUNIO /eBooks

Securing Ruby on Rails Web Applications:
What You Need to Know

authored by Mike Milner
for **IMMUNIO**

Co-founder and CTO
@secretmike

Estimates say that there are 750,000 websites running on Ruby on Rails, including immensely popular and successful sites like Heroku, GitHub, Airbnb, and Hulu to name just a few. Companies use this framework to build web applications in part because it is very popular with developers. It is open source, built on the dynamic Ruby programming language which in turn is designed with developer happiness in mind. It is designed to build applications rapidly—to go quickly from concept to working prototype. It's also expandable with a large community of shared open source libraries that can do basically anything the developers need to do. That means this framework gives developers in companies of all sizes flexibility around building web applications, rapidly.

Ruby on Rails applications are vulnerable like applications written in any other language or framework.

All web applications, including those built with Ruby on Rails, suffer from the same reality of all software—they are very likely to be vulnerable to exploitation if no active measures are put in place to reduce that likelihood. Ruby on Rails applications are vulnerable like applications written in any other framework. It is important to recognize that Ruby on Rails has a strong track record of resolving security issues quickly once identified.

Application security is a critical component of any complete information security program—it includes secure coding training, software development lifecycle (SDLC), architectural reviews, source code reviews, penetration testing, and more. Application security is necessary for web applications built using any framework. In this report, we examine some of the vulnerabilities that impact web applications built on Ruby on Rails and the best way to protect against those risks. These concepts carry over for all your web applications.

"70% of the 78 Rails CVEs fall into 5 categories: SQLi, XSS, RCE, Directory Traversal, and Response Splitting" [1]

RISKS

Even with the best efforts of the developers and an adequately trained and staffed information security team, no software is completely secure. Developers and information security staff are human. They make mistakes or may lack interest or expertise in certain areas. And application security happens to be experiencing a shortage of qualified talent. Meanwhile, hackers are very creative and tenacious humans. The threat landscape is ever changing, as hackers work hard to find ways to exploit whatever vulnerabilities they can find to gain access to your system and assets.

Ruby on Rails is no exception—there are security risks with this framework. The most common publicly known vulnerabilities are known as CVEs—common vulnerabilities and errors. As of 2016, a total of 78 of them have been identified and fixed across all versions of Rails (since the first version of Rails appeared in 2005). Seventy percent of these CVEs fall into five categories: cross-site scripting (XSS), remote code execution (RCE), and SQL injection, directory traversal, and response splitting. These five threats in particular are so common that your web applications, programmed in Rails or another platform, may still be vulnerable because of other libraries you use, even if the vulnerability did not originate in Rails.

However, those are not the only 78 errors or vulnerabilities with Ruby on Rails. The CVE list shows only those that have been reported, repaired, and submitted to and published by MITRE, the central non-profit that tracks them. However, if someone doesn't take the time to report it according to a well defined protocol, no CVE is issued. That means that paying attention to CVEs is a good start. But they are only the start of a complete web application security practice.

Note that these 78 CVEs only refer to vulnerabilities affecting the Ruby on Rails framework itself, and hence include the

[1] *"2015, A Record Year For Vulnerabilities," Risk Based Security, https://www.riskbasedsecurity.com/2016/03/2015-a-record-year-for-vulnerabilities/*

COMMON ACROSS ALL WEB APPLICATIONS.

OWASP TOP TEN: [2]

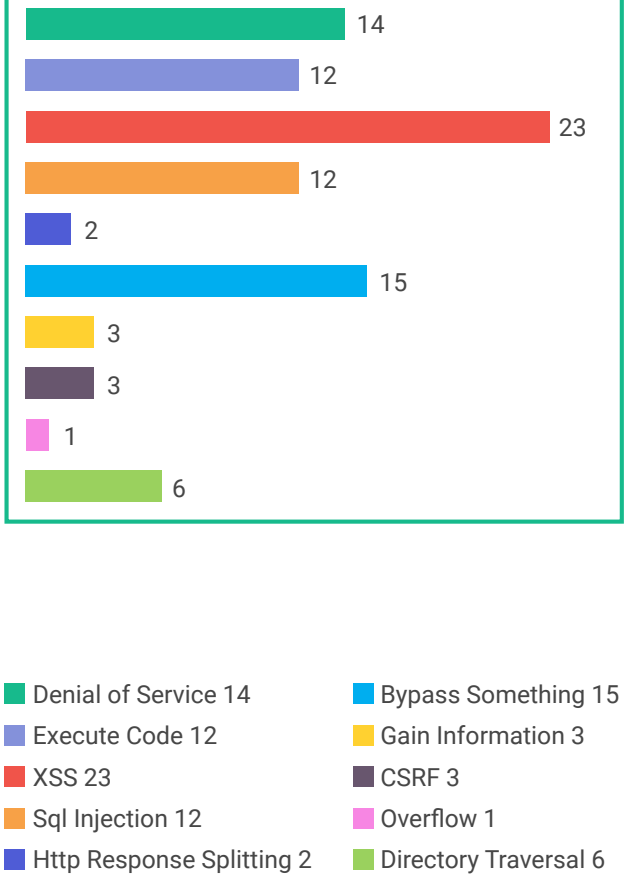
- ***A1.*** *Injection*
- ***A2.*** *Broken Authentication and Session Management*
- ***A3.*** *Cross Site Scripting (XSS)*
- ***A4.*** *Insecure Direct Object References*
- ***A5.*** *Security Misconfiguration*
- ***A6.*** *Sensitive Data Exposure*
- ***A7.*** *Missing Function Level Access Control*
- ***A8.*** *Cross Site Request Forgery (CSRF)*
- ***A9.*** *Using Components with Known Vulnerabilities*
- ***A10.*** *Unvalidated Redirects and Forwards*

[2] *"2013 Top10 List," OWASP.org, https://www.owasp.org/index.php/Top_10_2013-Top_10*

applications built using it. Since most web applications include other libraries and code that your own developers wrote, the web application is at risk from other vulnerabilities, as well.

Web applications developed on Ruby on Rails are vulnerable to the same issues as those built in any other language. Every framework, including Rails, has security features to mitigate risks baked into the framework. However, there are always subtleties and developer mistakes that account for some remaining vulnerabilities.

THE VULNERABILITIES BY TYPE[3]



[3] *"Ruby on Rails: Common Errors and Vulnerabilities," CVE Details, https://www.cvedetails.com/vendor/12043/Rubyonrails.html*

When considering the security of web applications, there are three types of code that organizations must consider: Code you didn't write, code you did write, and code that no one wrote.

CURRENT MODEL

The current model for Ruby on Rails based web applications is to upgrade the framework in your application every time there is a new security release. That is, when a new version of Rails is released that addresses a new vulnerability, your organization will have to rebuild the application with this newest version of Rails.

For many companies, it is not possible to upgrade their web applications each time there is a security upgrade. For one thing, it takes time and resources to upgrade an application, especially a major version upgrade (Ruby on Rails 2, 3, 4, 5)—we have seen companies spend six months and use the entire team to upgrade their applications from Rails 3 to 4, for example. These major releases are not necessarily backward compatible. In other cases, a web application can be frozen for new feature development, in maintenance mode, or is not scheduled to receive development work. That application will not receive security updates because it will not be updated to a newer version of Rails.

What to Worry About

When considering the security of web applications, there are three types of code that organizations must consider: Code you didn't write, code you did write, and code that no one wrote.

1. Code You Didn't Write

This refers to applications programmed in Ruby on Rails and all third-party libraries you use on your web applications. In these cases, you inherit the code, with any vulnerabilities, and have to trust it. Because your engineers are not owners of this code, they do not understand the intricacies to test its security posture and develop security fixes for it.

Currently, the best practice for dealing with code you didn't write is to run CVE checkers to identify documented errors and vulnerabilities. This is a critical part of information security, but does not provide

Only half of the vulnerabilities end up becoming CVE's.

complete security. Consider this: Existing security products that are built using the CVE list or the National Vulnerability Database will discover only half of the vulnerabilities reported in 2015.

2. Code You Did Write

This refers to applications developed in-house on top of Ruby on Rails applications. No matter how strong a programming team you have, no one is exempt from making mistakes, so errors and vulnerabilities are highly likely unless active measures are taken to reduce them.

The current best practice for securing code you wrote in-house is to run static scanners to analyze the code and highlight any coding errors or potential vulnerabilities. Static scanners designed for Ruby have limitations, as with all dynamic languages. However, there is one scanner designed specifically for Rails that stands out and should be in your security arsenal: Brakeman. After running static scanners, the next step would be to run a penetration test, where an attack is attempted to help identify vulnerabilities.

3. Code No One Wrote

This refers to security features that are not part of your code, nor the libraries and components that you are using to build the application. You do not have access to the source code, and thus you risk inheriting any vulnerabilities this code may have. For example, most login / authentication libraries do not have the security features required to resist certain types of botnet attacks, such as a million-machine botnet trying to abuse the login function (aka, credential stuffing). The current best practice in this area is to address each of these threats individually, such as special network devices for stopping botnet attacks, or user behavioral analysis for detecting stolen sessions.

The value of runtime application self-protection (RASP) is that this new class of technology is preventative.

Securing Ruby on Rails Web Applications

Securing web applications using today's best practices and technology revolves around finding vulnerabilities in the code and remediating them:

- **CVE checker**
  Analyzing the libraries used to identify one of the published common vulnerabilities and errors (CVE)
- **Static scanner**
  Analyzing source code to identify vulnerabilities
- **Dynamic testing**
  1. Penetration testing
  Attempting to attack the system as a way to identify vulnerabilities or errors
  2. Dynamic scanner
  Analyzing the application while it is running in real time
  3. Bug bounty
  Rewarding individuals who find and report errors or vulnerabilities in your web application
- **Web application firewall (WAF)**
  Filtering input to the application to identify and prevent attacks

Location of Vulnerability	Activity to identify Vulnerabilities	Addressing Vulnerability	Challenges
Your application	Static testing Dynamic testing	Code remediation	• Requires highly skilled talent • Does not usually find the subtle bugs • Fixing bugs may take months or years depending on the organization
Ruby on Rails	CVE checker	Upgrade Rails	• Not always feasible (i.e., applications in maintenance mode, major release upgrade) • 0 days
3rd party libraries	CVE checker	Upgrade library	• Not always feasible (i.e., applications in maintenance mode) • 0 days

Today's state of the art technology for dealing with security vulnerabilities shifts the focus from finding all vulnerabilities, and remediating fast, to reducing the risk of a breach by blocking the exploitation.

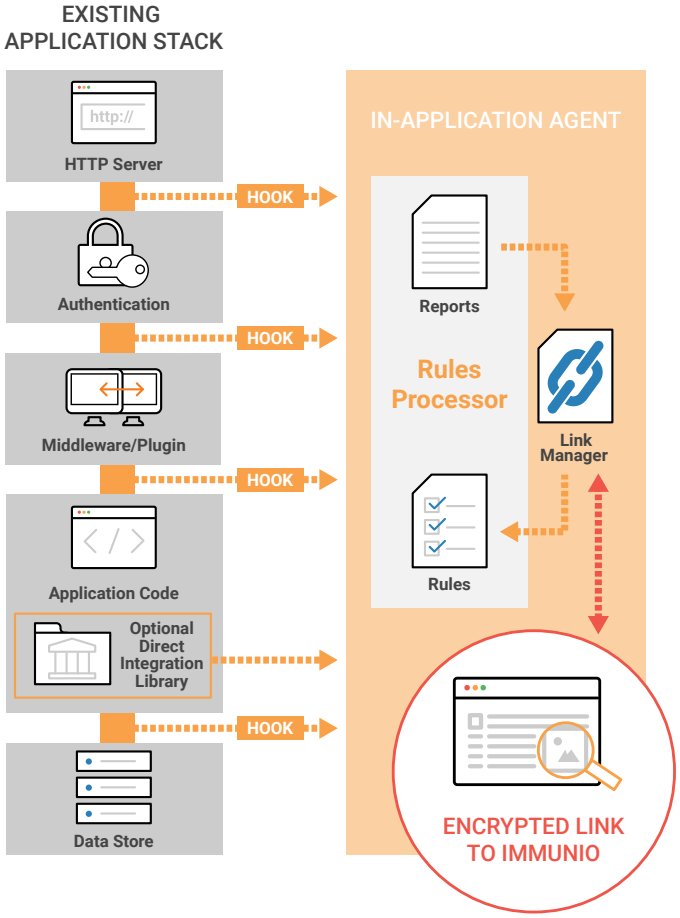
TODAY'S STATE OF THE ART: RASP

As described above, there are serious drawbacks in the current security tools for Ruby on Rails web applications. The next generation of protection—runtime application self-protection—plugs the holes those security procedures leave in your web applications.

Today's state of the art technology for dealing with security vulnerabilities shifts the focus from finding all vulnerabilities, and remediating fast, to reducing the risk of a breach by blocking the exploitation.

The value of runtime application self-protection (RASP) is that this new class of technology is preventative. It does not just identify vulnerabilities in the application; it knows how the application works when it is healthy, when it is dealing with acceptable traffic patterns. So it can easily identify attacks and unacceptable traffic without having to understand the exact nature of the threat.

How RASP Actually Works



In addition to protecting against known vulnerabilities, RASP protects web applications against the unknown.

Because it effectively wraps itself around the application, RASP protects in real time and can protect against attacks from outside the perimeter of the firewall, as well as from within inner trust boundaries.

RASP is flexible and easy to implement. It is adaptive and updates automatically when the code is updated. And it does not require access to the source code, meaning it protects all three types of code—the code you wrote, that others wrote, and that no one wrote.

RASP makes your web applications practically impenetrable. It can detect and render unexploitable all 5 of the Ruby on Rails vulnerabilities that have been disclosed in the last decade.

1. SQL injection
2. Code execution
3. XSS
4. Directory traversal
5. HTTP response splitting

In addition to protecting against known vulnerabilities, RASP protects web applications against the unknown. And secures the application until you have the chance to upgrade to a new version of Ruby on Rails.

A best in class security program includes a variety of tools to address errors and vulnerabilities in web applications.

Zero Days in Rails

Recent Ruby on Rails hacks that RASP protects against:

- **CVE-2013-0263**
  This vulnerability was caused by a timing attack and impacts anyone in the world with a Ruby on Rails server. This bug was present in the code since 2009 and fixed in 2013.
- **CVE-2013-0156**
  Multiple vulnerabilities (including SQL injection, the ability to bypass authentication, and the ability to inject and execute any code). This object injection bug threatened any site built on any version of Ruby on Rails.
- **Other CVEs in early 2016**
  8 errors and vulnerabilities to Ruby on Rails were disclosed in one day (including cross-site scripting, mass assignment, and directory traversal). These were classified as CVEs, are potentially high impact and required an upgrade to protect against.

WAF Isn't Enough

Web application firewalls (WAF) protect against many common attack vectors, such as SQL injection and cross-site scripting. Yet there are significant drawbacks.

Web application firewalls:

- Are extremely easy to bypass, since in almost all deployments they are running against known signatures.
- Are complicated to set up properly and can take weeks or months to set up for just one application.
- Work best with traditional web applications and tend to have blind spots when it comes to modern applications (built with AJAX, HTML5, WebSockets, microservices architecture, etc.).
- Need to be in the network path: one must route traffic through them, which may not always be doable when assets are on different networks or cloud providers.

When comparing even the leading WAF providers to RASP technology, the gap in

accuracy of detection and ease of bypass is apparent. In one industry benchmark, the leading WAF provider was only able to detect 60% of attempted attacks, while another vendor only detected 20%. Meanwhile, the RASP provider detected and stopped 100% of the attacks, with a much lower false positive rate than both vendors.

More info on the report can be found here.

Benefits of RASP

Runtime application self-protection has many benefits for organizations running web applications programmed with Ruby on Rails.

- Reduces the risk for a significant number of OWASP security vulnerabilities, including true protection against zero days
- Secures application beyond just code errors, including user security, botnet attacks, and monitoring
- Reduces urgency for patching or upgrading
- Enables more effective use of information security team members, and even enables some companies to have few or no dedicated information security staff
- Augments to existing information security measures like penetration testing, bug bounty programs, CVE checkers, etc.
- Difficult to bypass
- Can be deployed in minutes and configured in only hours
- Works anywhere--cloud, data center, behind the firewall, etc.

Ruby on Rails is a flexible and fast open source application that developers like to work with. There are 750,000 websites running web applications programmed with Ruby on Rails today and more to come. For information security teams, that means more potentially vulnerable web applications to secure.

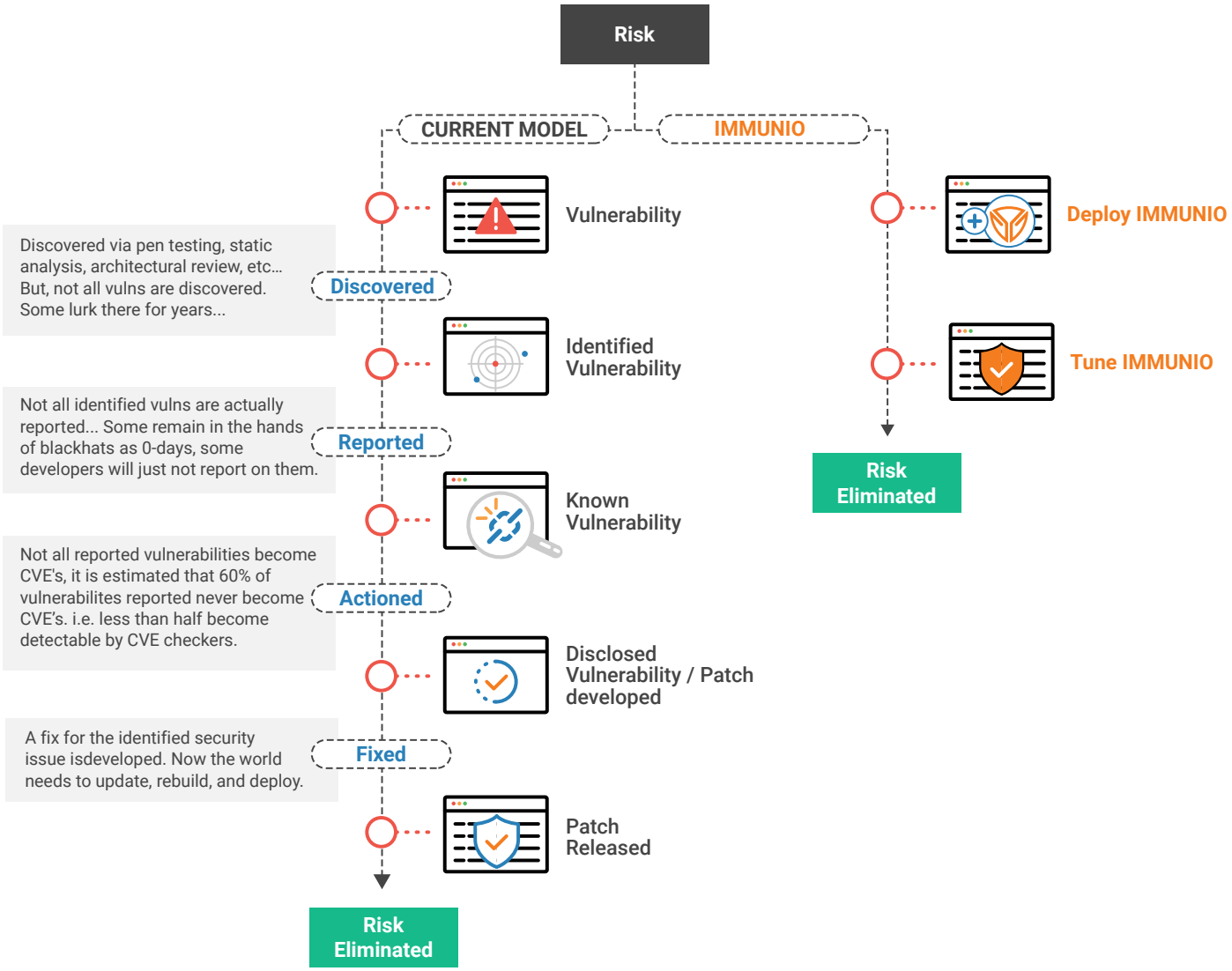
A best in class security program includes a variety of tools to address errors and vulnerabilities in web applications. That includes using runtime application self-protection on applications developed on Ruby on Rails.

CURRENT MODEL:

Usually **months or years** to eliminate risks, lots of manual effort, highly skilled labour, and relies on finding all risks

IMMUNIO MODEL:

Days or weeks, very little manual effort, no need for lots of highly skilled techies, and does not need to find all vulnerabilities



RASP protects against significant known and identified errors, as well as protecting against new and evolving threats as hackers get more creative.

If you already have an application security practice, consider adding runtime instrumentation. Adding RASP to your information security program is the best way to secure your Ruby on Rails web applications and, by extension, your users.

ABOUT IMMUNIO

IMMUNIO is a pioneer in real-time web application security (RASP), providing automatic detection and protection against application security vulnerabilities. The company's mission is to make truly effective real-time web protection technology easily available and widely deployed, and by doing so, stop the biggest source of breached data records.

For more information, visit **https://www.immun.io/** *or follow* **@immunio**.