Ruby Central RubyGems.org

Security Assessment

December 11, 2024

Prepared for:
Martin Emde
Ruby Central

Prepared by: **Paweł Płatek, Artur Cygan, and Matt Schwager**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Keith Hoodlet, Engineering Director, Application Security
keith.hoodlet@trailofbits.com

The following consultants were associated with this project:

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Matt Schwager, Consultant
matt.schwager@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 16, 2024	Pre-project kickoff call
August 29, 2024	Status update meeting #1
September 09, 2024	Delivery of report draft
September 09, 2024	Report readout meeting
December 11, 2024	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Ruby Central engaged Trail of Bits to review the security of RubyGems.org. RubyGems is a gem hosting service for the Ruby community. It provides a web interface, an API, storage needs, and other functionality necessary to operate as the de facto Ruby package management solution.

A team of three consultants conducted the review from August 19 to September 6, 2024, for a total of five engineer-weeks of effort. Our testing efforts focused on common web application security vulnerabilities, authentication and authorization controls, infrastructure security concerns, and a comparative analysis of other package management solutions. With full access to source code and documentation, we performed static and dynamic testing of the RubyGems.org repository, associated infrastructure, and web application using automated and manual processes.

Observations and Impact

The RubyGems application is a three-tier web application consisting of a front end, back end, and database layers. Both the application and its associated infrastructure use industry-standard libraries, frameworks, and tooling. This made our review straightforward and demonstrates that the development process is building on a solid foundation.

We identified one high-severity finding related to optional StartTLS encryption in the SMTP mailer (TOB-RGM-5) and a noteworthy medium-severity finding related to missing multi-party approval for production deployments (TOB-RGM-31). These findings are unrelated and are not indicative of systemic, high-severity vulnerabilities in RubyGems' functionality.

Although generally of lesser severity, the other code review findings cluster around two categories: configuration (TOB-RGM-1, TOB-RGM-2, and TOB-RGM-6) and data validation (TOB-RGM-3, TOB-RGM-8, TOB-RGM-9, TOB-RGM-11, and TOB-RGM-13). We recommend further testing, whether internal or external, to audit for these specific classes of vulnerabilities.

Regarding infrastructure, most issues result from manual management of cloud resources (TOB-RGM-16, TOB-RGM-20, TOB-RGM-29). The Ruby Central team usually maintains the infrastructure via Terraform, and should work toward limiting manual, ad-hoc manipulations. Moreover, projects should have better separation (TOB-RGM-21).

We found an issue related to the development of Terraform files that may represent a single point of failure: the lack of a dual approval requirement for GitHub pull requests enables compromise of a single GitHub account, which can be escalated to achieve

complete RubyGems compromise (TOB-RGM-31). This issue, while of medium severity, is a challenge to remediate due to the small size of the Ruby Central team.

We observed that some AWS API access keys are not refreshed for a long time (TOB-RGM-32). This observation implies the need for regular inventorization of credentials, and rotation of old ones.

The results of our competitive analysis and corresponding recommendations are given in appendix D.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Ruby Central take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Make tickets for recommendations given in competitive analysis appendix D.** We recommend addressing the high-priority findings in the near future. Lower-priority findings can be addressed over the longer term, but should not be forgotten before implementation or dismissed without discussion inside the RubyGems team.
- **Consider investing in custom static analysis rules and expertise.** Static analysis is a useful methodology for finding variants of bugs and ensuring that you do not make the same mistake twice. Simple "disable security" flags, such as those found in TOB-RGM-2, TOB-RGM-4, and TOB-RGM-5, are ideal candidates for static analysis rules. For this reason, we recommend developing custom static analysis rules and expertise using a tool like Semgrep or CodeQL to help find these types of bugs.
- **Consider deprecating and removing any reliance on Marshal data.** Both TOB-RGM-3 and TOB-RGM-9 relate to Marshal data. Although these findings are informational, they pose a long-term risk that could lead to higher-severity vulnerabilities like remote code execution in the future. The risks of using Marshal are well-known within the Ruby community, and this data format can generally be replaced with safer alternatives such as JSON.
- **Reduce the number of AWS users and roles with write access to production resources.** Ideally, only the CI/CD pipeline and a break-glass user should have such access (see TOB-RGM-16). Access to some services should be limited even for the pipeline (TOB-RGM-23). Additionally, we recommend isolating projects into different AWS accounts (TOB-RGM-21) and cleaning up unused resources.

- **Enable and configure Amazon's services for logging, monitoring and alerting**: Security Hub, CodeGuru, GuardDuty, VPC Flow Logs, CloudTrail. The first three services will help keep AWS secure over time, and the last two will facilitate incident response. These services should have monitoring and alerting enabled.

- **Review and improve personal security of users with administrative access to AWS, GitHub, and other services.** This includes, for example: enabling and/or enforcing strong (phishing-resistant) two-factor authentication (2FA); securing development machines (e.g., timely software updates, full-disk encryption, and separate machines for development); using hardware keys for management of SSH keys; and periodically reviewing existing API keys.

- **Consult the following documents for additional AWS security recommendations.** While this audit covered the most important security gaps, many minor or more complex guidelines could also be implemented. These include Cloud Native Security Controls Catalog v1.0, Best practices for using the Terraform AWS Provider, and aSecureCloud library.

- **Review the security of Kubernetes deployments.** Both RubyGems and Shipit configurations should be audited, as they live in the same cluster. The open-source Shipit application itself should be audited, as it may have not been yet (except potentially as part Shopify's bug bounty program).

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	7
Low	10
Informational	15
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	9
Authentication	3
Configuration	4
Cryptography	4
Data Exposure	3
Data Validation	7
Error Reporting	2
Patching	1

Project Goals

The engagement was scoped to provide a security assessment of RubyGems.org. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is RubyGems susceptible to common web vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), SQL injection (SQLi), and server-side request forgery (SSRF)?
- Does RubyGems properly configure web application defenses such as the `Content-Security-Policy` and `Strict-Transport-Security` HTTP headers, and the `SameSite` and `HTTPOnly` cookie parameters?
- Can an attacker bypass authentication in the RubyGems web interface?
- Can unauthenticated users perform unauthorized operations in the RubyGems web interface?
- Are access controls properly enforced?
- Is 2FA implemented securely?
- Is proper isolation achieved in the RubyGems infrastructure?
- Are internal and privileged APIs hardened against external and unauthorized access?
- Does RubyGems deserialize untrusted data securely?
- Is RubyGems resilient against common denial-of-service vectors?
- Are secrets managed and stored securely?
- Are AWS services securely configured?

Project Targets

The engagement involved a review and testing of the targets listed below.

RubyGems.org

Repository	https://github.com/rubygems/rubygems.org
Version	`21895a522076d15a3cb8d072229e2cfdbdb873e8`
Type	Ruby, JavaScript
Platform	Linux

rubygems-terraform

Repository	https://github.com/rubygems/rubygems-terraform
Version	`9c22354c60ac59de9a2cad32c3f061e196350c29`
Type	Terraform, Varnish Configuration Language, GitHub Actions
Platform	Linux, AWS, Fastly, GitHub

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- A manual review of cryptographic functionality within the application
- A manual review of GitHub Actions CI/CD jobs
- A review of the RubyGems web interface, with a focus on:
    - Common web application security concerns like XSS, SQLi, CSRF, and SSRF
    - Common HTTP security defenses such as `Content-Security-Policy` and `Strict-Transport-Security`
    - Dynamic testing for authorization issues
- Static analysis using Semgrep
- Secrets scanning using Semgrep
- A review of Kubernetes manifests
- A review of Rails configuration files
- A review of controllers and Pundit policies for authorization issues
- A review for deserialization, remote code execution, and mass assignment bugs
- A review of the Trusted Publishing implementation
- A review of authentication implementation
- A review of Terraform files and deployed AWS resources in the following services: IAM, EKS, ECR, S3, EC2, RDS, VPC
- A review of the Fastly configuration

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not audit the configuration of the Shipit deployment application due to time constraints. We also did not audit the security of the configuration of the `quay.io` organization, which stores Shipit Docker images.

- While our review of the EKS configurations found nothing inherently insecure, the time spent on the review of the actual Kubernetes configurations was minimal. We recommend focusing any subsequent security audits on the system's Kubernetes and Shipit components.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix E.1
Ruzzy	A coverage-guided fuzzer for pure Ruby code and Ruby C extensions	Appendix F.1
Burp Suite Professional	A web security toolkit that helps identify web vulnerabilities both manually and semi-automatically through the embedded scanner	Appendix G

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We did not review any RubyGems code that extensively uses mathematical operations.	**Not Considered**
Auditing	Auditing was not a significant focus of this engagement, and further investigation is required to reach a conclusion.	**Further Investigation Required**
Authentication / Access Controls	We spent a significant portion of this audit reviewing authentication procedures and access controls. This included basic web application login functionality, administrative functionality, and infrastructure access controls. We found a single, informational application-related issue (TOB-RGM-7). Additionally, we found many infrastructure issues related to authentication and access controls (TOB-RGM-16, TOB-RGM-17, TOB-RGM-18, TOB-RGM-20, TOB-RGM-21, TOB-RGM-22, TOB-RGM-23, TOB-RGM-24, TOB-RGM-27, TOB-RGM-31, and TOB-RGM-33).	**Weak**
Complexity Management	The system manages complexity well. It is built on industry-standard technologies such as Rails, Terraform, and Kubernetes. This encourages the system to follow best practices and provides a well-trodden path for most functionality. However, as with most systems that have existed for many years, there is a moderate amount of legacy code and functionality. The volume of security and code quality findings indicate there is room for improvement. However, the severity of these findings is generally low, which indicates that the RubyGems team has maintained a security mindset throughout the lifetime of the application.	**Satisfactory**

Configuration	Configuration management is inline with industry standards and mainstream tooling. The system's configuration is generally well-delineated between files, environment variables, command-line arguments, and information gathered at runtime, such as secrets. Although we found a number of configuration-related issues, they were generally of lower severity.	**Satisfactory**
Cryptography and Key Management	Key and secret management is generally well-designed within the application and associated infrastructure. There were a moderate number of cryptographic findings, including one high-severity finding regarding SMTP encryption in transit (TOB-RGM-5). The application avoids building its own cryptographic primitives and instead relies on mainstream implementations. This is a good practice and should be encouraged and continued in the future.	**Moderate**
Data Handling	Data validation and handling can be improved, although this is a challenging task because the application ingests a wide array of inputs. Issues we discovered related to data handling include server-side request forgery (SSRF) (TOB-RGM-8 and TOB-RGM-12), Marshal deserialization (TOB-RGM-3 and TOB-RGM-9), JWT handling (TOB-RGM-11, TOB-RGM-13, and TOB-RGM-14), and cross-site scripting (XSS) (TOB-RGM-28). Consider additional review and investments in this area.	**Weak**
Documentation	The system is well-documented and has good documentation regarding public RubyGems functionality. The repository also includes Markdown files that document development workflows. We found a few minor documentation discrepancies (TOB-RGM-7 and Code Quality #11). Consider reviewing public APIs for appropriate documentation (e.g., web hook functionality).	**Strong**
Maintenance	Overall, the system maintenance is good. The application leverages Dependabot for dependency updates. However, we found one low-severity issue regarding patching (TOB-RGM-19) and a number of code quality findings and findings due to the use of legacy functionality (TOB-RGM-7 and TOB-RGM-9). None of	**Satisfactory**

	these findings imposes significant risk, but they do represent opportunity for improvement.	
Memory Safety and Error Handling	Due to the dynamic, garbage-collected nature of the Ruby language, memory safety bugs were not a focus of this audit. However, we did perform fuzz testing on a Ruby C extension used by the application (appendix F.1). Additional fuzz testing should be considered for any gem dependencies that include a C extension component. Error handling is generally well-implemented. We found a few issues related to error handling (TOB-RGM-10, TOB-RGM-15, and TOB-RGM-30), but all are informational.	**Strong**
Testing and Verification	We did not review in detail the project's unit and integration tests. We did note the use of automated testing, CI/CD processes, and static analysis tooling, which indicates some level of investment in this area. However, we did not review the efficacy of these processes.	**Further Investigation Required**

Summary of Findings

The tables below summarize the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Docker Compose ports exposed on all interfaces	Configuration	Low
2	Rails cookies lack SameSite protections	Configuration	Low
3	Memcached cache store may allow Marshal deserialization	Data Validation	Informational
4	HSTS includeSubDomains disabled in production environment	Cryptography	Low
5	SMTP mailer may fallback on unencrypted communication	Cryptography	High
6	Avo controller sets Content-Security-Policy unsafe_inline	Configuration	Informational
7	Any user can fire webhooks for the gemcutter gem	Access Controls	Informational
8	SSRF to internal URLs in web hook functionality	Data Validation	Medium
9	Service stores spec Marshal data alongside gem file	Data Validation	Informational
10	Unhandled exception in trusted publisher exchange token request	Error Reporting	Informational
11	Insufficient JWT validation in trusted publisher exchange token request	Data Validation	Informational
12	SSRF risk while refreshing OIDC providers	Data Validation	Informational
13	The jti claim uniqueness is not enforced by database	Data Validation	Informational

14	Provider key age not checked	Cryptography	**Informational**
15	Overly verbose error returned to the user	Data Exposure	**Informational**
16	Redundant IAM users and permissions	Authentication	**Medium**
17	MFA is not enforced for IAM user	Authentication	**Medium**
18	Lack of policy conditions leading to cross-service confused deputy attacks	Access Controls	**Low**
19	External GitHub CI actions are not pinned	Patching	**Low**
20	Terraform OIDC provider has insecure configuration	Access Controls	**Low**
21	RubyGems and RubyAPI projects are under a single AWS account but different Terraform configurations	Access Controls	**Medium**
22	SQS policy has S3-based policy without account	Access Controls	**Low**
23	GitHub Action Terraform policy is overly broad	Access Controls	**Low**
24	Networking security concerns	Access Controls	**Low**
25	GitHub token may be exposed in packed data in a release-gem action	Data Exposure	**Informational**
26	GitHub Action redundant persist-credentials	Data Exposure	**Informational**
27	Fastly uses long-lived credentials instead of OIDC	Authentication	**Informational**
28	Stored XSS in gems.rubygemsusercontent.org	Data Validation	**Low**
29	OIDC S3 buckets are not claimed but are configured in Fastly	Configuration	**Medium**
30	IAM fastly user cannot list S3 objects	Error Reporting	**Informational**

31	No dual approval requirement for production deployments	Access Controls	**Medium**
32	The app-production user's API key has not been rotated for a long time	Cryptography	**Medium**
33	Staging deployments are performed with a highly privileged K8s role	Access Controls	**Informational**

Detailed Findings

1. Docker Compose ports exposed on all interfaces

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-RGM-1
Target: `rubygems.org/docker-compose.yml`	

Description

To specify Docker ports, `docker-compose.yml` configuration files use the `ports` configuration option of, for example, `5432:5432` for the Postgres container (figure 1.1). This means that these ports are accessible not just to other processes running on the same computer, but also from other computers on the same network.

```
db:
  image: postgres:13.14
  ports:
    - "5432:5432"
  environment:
    - POSTGRES_HOST_AUTH_METHOD=trust
```

Figure 1.1: Ports exposed on all interfaces (`rubygems.org/docker-compose.yml:2–7`)

Exploit Scenario

A Ruby Central developer runs this `docker-compose.yml` file while on a public Wi-Fi network. An attacker on the same network connects to the Postgres database running on the developer's computer; this database is available on port 5432 and uses the default password `postgres`. The attacker can then read and modify any data in the developer's database.

Recommendations

Short term, modify these configuration values, setting them to `127.0.0.1:5432:5432` instead of `5432:5432`.

Long term, use the `port-all-interfaces` Semgrep static analysis rule to detect and flag instances of this configuration pattern.

References

- Compose file version 3 reference: ports

.

2. Rails cookies lack SameSite protections

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-RGM-2
Target: `rubygems.org/lib/github_oauthable.rb`, `rubygems.org/config/initializers/session_store.rb`	

Description

As stated in MDN, the `SameSite` cookie attribute provides the following functionality:

> Controls whether or not a cookie is sent with cross-site requests, providing some protection against cross-site request forgery attacks (CSRF).

RubyGems disables strict `SameSite` protections in a number of locations by setting the attributes value to `Lax`. For example, the following location sets this value explicitly:

```
def log_in_as(user:, expires: 1.hour)
  cookies.encrypted[admin_cookie_name] = {
    value: user.id,
    expires: expires,
    same_site: :lax
  }
end
```

Figure 2.1: Admin cookie setting `SameSite=Lax` (`rubygems.org/lib/github_oauthable.rb#72–78`)

This value is also implicitly set on the Rails session cookie in the following location:

```
Rails.application.config.session_store :cookie_store, key: '_rubygems_session'
```

Figure 2.2: Session cookie implicitly setting `SameSite=Lax` (`rubygems.org/config/initializers/session_store.rb#8`)

By default, when using `cookie_store`, the session cookie will set `SameSite=Lax`. This is better than setting the value to `None`, but it still ultimately introduces the risk of CSRF attacks. Because Rails provides built-in CSRF protections, this is a defense-in-depth improvement; this finding's severity is therefore low.

Exploit Scenario

An attacker discovers a GET-based CSRF vulnerability in the RubyGems application. They then discover a bypass for built-in protections, or discover a controller that has set

`skip_forgery_protection`. They are then able to carry out the CSRF attack due to a lack of `SameSite` protections on RubyGems cookies.

Recommendations

Short term, configure `SameSite=Strict` on all application cookies.

Long term, incorporate the `rails-cookie-attributes` Semgrep rule from appendix E.2 into your CI systems.

3. Memcached cache store may allow Marshal deserialization

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RGM-3
Target: `rubygems.org/config/environments/production.rb`	

Description

Deserialization of attacker-controlled Marshal data may result in remote code execution. The RubyGems application uses Memcached as its cache store in the following location:

```
config.cache_store = :mem_cache_store, ENV['MEMCACHED_ENDPOINT'], {
  failover: true,
  socket_timeout: 1.5,
  socket_failure_delay: 0.2,
  compress: true,
  compression_min_size: 524_288,
  value_max_bytes: 2_097_152 # 2MB
}
```

Figure 3.1: Rails Memcached cache store (rubygems.org/config/environments/production.rb#116–123)

The `:mem_cache_store` symbol maps to the `MemCacheStore`, which subclasses `ActiveSupport::Cache::Store`. This parent class allows a `serializer` to be configured. The default serializer for Rails 7.1 is `:marshal_7_1`. The serializer can be configured as `:message_pack`, which will default to a more secure serialization format; however, it will fallback to the insecure Marshal format if it receives Marshal data. A custom serializer is needed to completely mitigate this attack vector.

Exploit Scenario

An attacker gains unauthorized access to the Memcached instance(s) used by the RubyGems application. This could occur due a separate vulnerability such as server-side request forgery (SSRF), or improper access controls configured on the Memcached instance(s). The attacker could then add malicious, serialized Marshal objects to the instance(s) that are later deserialized by the application, resulting in code execution.

At this time, we are unaware of any entrypoints to add malicious Marshal objects to production Memcached instance(s), so we have marked this finding as informational. However, similar attack chains have been observed. For example, GitHub Enterprise saw a similar vulnerability chain presented in A New Era of SSRF, included in the References section below.

Recommendations

Short term, create a custom JSON or `MessagePack` serializer that does not fallback on deserializing Marshal data. `ActiveSupport::Cache::Store` supports specifying a custom serializer module. Something similar to `MessagePackWithFallback` can be used, except without the fallback.

Long term, incorporate the `rails-cache-store-marshal` Semgrep rule from appendix E.2 into your CI systems.

References

- A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages!

4. HSTS includeSubDomains disabled in production environment

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-RGM-4
Target: `rubygems.org/config/environments/production.rb`	

Description

The HTTP `Strict-Transport-Security` (HSTS) response header ensures that subsequent connections to a site are encrypted. Additionally, the `includeSubDomains` directive applies the same configuration to all of a domain's subdomains. The production environment disables this functionality in the following location:

```
# Force all access to the app over SSL, use Strict-Transport-Security, and use
secure cookies.
config.force_ssl = true
config.ssl_options = {
  hsts: { expires: 365.days, subdomains: false },
  redirect: {
    exclude: ->(request) { request.path.start_with?('/internal') }
  }
}
```

Figure 4.1: HSTS `includeSubDomains` disabled (`rubygems.org/config/environments/production.rb#53–60`)

Note that the staging environment repeats this configuration.

Exploit Scenario

An attacker can perform a downgrade attack against a target connecting to a subdomain of RubyGems.org. They may also passively intercept subsequent HTTP connections after an initial HTTPS connection to subdomains of RubyGems.org.

Recommendations

Short term, enable the `includeSubDomains` directive for HSTS response headers.

Long term, incorporate the `action-dispatch-insecure-ssl` Semgrep rule from appendix E.2 into your CI systems.

5. SMTP mailer may fallback on unencrypted communication

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-RGM-5
Target: `rubygems.org/config/initializers/sendgrid.rb`	

Description

Rails uses `ActionMailer` to dynamically send emails. It uses the `smtp_settings` option to configure its SMTP deliveries. The RubyGems application configures these settings in the following location:

```
unless Rails.env.local?
  ActionMailer::Base.smtp_settings = {
    address:              'smtp.sendgrid.net',
    port:                 587,
    user_name:            ENV['SENDGRID_USERNAME'],
    password:             ENV['SENDGRID_PASSWORD'],
    domain:               'mailer.rubygems.org',
    authentication:       :plain,
    enable_starttls_auto: true
  }
  ActionMailer::Base.delivery_method = :smtp
end
```

Figure 5.1: Rails SMTP configuration (`rubygems.org/config/initializers/sendgrid.rb#1–13`)

The StartTLS protocol command is used to establish encrypted communication with the SMTP server. The `enable_starttls_auto` setting attempts to establish an encrypted channel, but falls back on unencrypted communication if it fails. However, the `enable_starttls` setting requires a successful StartTLS and fails if it is unsupported.

Exploit Scenario

An attacker is in a privileged network position between the RubyGems application server and the remote SMTP server. When the application server attempts to connect to the SMTP server, the attacker removes StartTLS commands during the initial handshake. They then return an unsupported error to the application server to downgrade the connection to an unencrypted stream.

Recommendations

Short term, change the `enable_starttls_auto` setting to `enable_starttls`.

Long term, incorporate the `action-mailer-insecure-tls` Semgrep rule from appendix E.2 into your CI systems.

6. Avo controller sets Content-Security-Policy unsafe_inline

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-RGM-6
Target: `rubygems.org/config/initializers/avo.rb`	

Description

The `Content-Security-Policy` HTTP header allows web servers to specify content that the web page will load. It is often used to protect against attacks like XSS and clickjacking. The RubyGems application configures the Avo controller's `style-src` to include `unsafe_inline`:

```
Avo::ApplicationController.content_security_policy do |policy|
  policy.style_src :self, "https://fonts.googleapis.com", :unsafe_inline
end
```

Figure 6.1: Avo controller configures `style-src` with `:unsafe_inline` (`rubygems.org/config/initializers/avo.rb#141–143`)

This allows including arbitrary inline `style` elements or attributes on elements (e.g., in combination with HTML injection). We are unaware of any vulnerabilities that are associated or could be chained with this functionality. However, privacy concerns or unknown attack vectors may still exist, as `style` elements may import additional stylesheets.

Recommendations

Short term, remove the `:unsafe_inline` directive.

Long term, if the above recommendation is not feasible, consider using `style-src` nonces (included in the References section below). These nonces allow a web page to verify the integrity of the stylesheet data it is receiving.

References

- CSP style-src unsafe inline styles
- CSS Injection Attacks
- Exfiltration via CSS Injection
- Better Exfiltration via HTML Injection

7. Any user can fire webhooks for the gemcutter gem

Severity: **Informational**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-RGM-7
Target: `app/controllers/api/v1/web_hooks_controller.rb`	

Description

The RubyGems service offers web hook functionality for a user's gems. The documentation for this functionality is limited, and appears to be available only in the API documentation. Web hooks have a "fire" API endpoint that allows a user to test their web hook:

```
def fire
  webhook = @api_key.user.web_hooks.new(url: @url)
  @rubygem ||= Rubygem.find_by_name("gemcutter")

  authorize webhook

  if webhook.fire(request.protocol.delete("://"), request.host_with_port,
                  @rubygem.most_recent_version, delayed: false)
    render plain: webhook.deployed_message(@rubygem)
  else
    render_bad_request webhook.failed_message(@rubygem)
  end
end
```

Figure 7.1: Web hook test fire functionality (rubygems.org/app/controllers/api/v1/web_hooks_controller.rb#32–44)

If the string "*" is provided as the gem name, then the application will fire a web hook for the `gemcutter` gem. Any user can initiate this API call and send the result to a URL of their choosing. We observed the following request sent to a URL of our choosing:

```
POST / HTTP/1.1
User-Agent: Faraday v2.10.1
Authorization: <REDACTED>
Hr_target_url: https://rhcixkbny57mb6prujw4r20nve15pvdk.oastify.com
Hr_max_attempts: 3
Content-Type: application/json
...
Host: rhcixkbny57mb6prujw4r20nve15pvdk.oastify.com
Content-Length: 1560

{"name":"gemcutter","downloads":1546928,"version":"0.7.1", ...}
```

Figure 7.2: Web hook request for `gemcutter` gem

Note that this finding was originally marked as low severity, but has been changed to informational in the final comprehensive report. Any RubyGems user can configure web hooks for any gem, so an attacker firing web hooks for the `gemcutter` gem is not an additional privilege.

Recommendations

Short term, remove the `gemcutter` web hook fire fallback.

Long term, considering documenting the web hook functionality or removing it if it is no longer needed.

8. SSRF to internal URLs in web hook functionality

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RGM-8
Target: `rubygems.org/app/models/web_hook.rb`, `rubygems.org/app/jobs/notify_web_hook_job.rb`	

Description

Web hook test "fire" functionality allows specifying sensitive hostnames like `localhost` and IPs like `10.0.0.1`. This allows an attacker to perform SSRF attacks against internal hosts. This functionality exists as an Active Job in the following locations:

```
def fire(protocol, host_with_port, version, delayed: true)
  job = NotifyWebHookJob.new(webhook: self, protocol:, host_with_port:, version:)

  if delayed
    job.enqueue
  else
    job.perform_now
  end
end
```

Figure 8.1: Active Record model for firing web hooks (rubygems.org/app/models/web_hook.rb#23–31)

This functionality ultimately makes an HTTP POST request to an attacker-specified URL in the following location:

```
def payload
  rubygem.payload(version, protocol, host_with_port).to_json
end
...
def post(url)
  Faraday.new(nil, request: { timeout: TIMEOUT_SEC }) do |f|
    f.request :json
    f.response :logger, logger, headers: false, errors: true
    f.response :raise_error
  end.post(
    url, payload,
    {
      "Authorization"   => authorization,
      "HR_TARGET_URL"   => webhook.url,
      "HR_MAX_ATTEMPTS" => "3"
    }
```

```
    )
end
```

Figure 8.2: HTTP POST request to URL (`rubygems.org/app/jobs/notify_web_hook_job.rb#77–90`)

Tests to most internal URLs seemingly fail, which is not observable to the outside user. However, firing a web hook to `http://localhost:3000/api/v1/gems` appears to succeed, which indicates that SSRF attacks to internal hosts are possible.

Exploit Scenario

An attacker discovers an SSRF vulnerability in the web hook functionality. If they can combine this with a CRLF injection in the underlying HTTP library, and a deserialization vulnerability like TOB-RGM-3, then they may be able to inject arbitrary data into the cache store and later achieve remote code execution. This attack chain is well-documented in A New Era of SSRF, included in the References section below.

Similarly, an attacker may be able to make POST requests to arbitrary hosts on the internal network. This is often not an exploit in and of itself, but SSRF vulnerabilities can commonly be combined with other issues to achieve higher impact.

Recommendations

Short term, proxy web hook HTTP requests through an application like Smokescreen. This will prevent requests from reaching internal address space.

Long term, write automated tests to ensure that all web hook requests are proxied and cannot reach internal address space.

References

- A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages!
- OWASP Server-Side Request Forgery Prevention Cheat Sheet

9. Service stores spec Marshal data alongside gem file

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RGM-9
Target: `rubygems.org/app/models/pusher.rb`, `rubygems.org/lib/tasks/helpers/gemcutter_tasks_helper.rb`, `rubygems.org/lib/tasks/gemcutter.rake`	

Description

In Ruby, Marshal data presents a security risk because it can result in deserialization bugs, which can lead to remote code execution. The Marshal module includes the following warning:

> By design, Marshal.load can deserialize almost any class loaded into the Ruby process. In many cases this can lead to remote code execution if the Marshal data is loaded from an untrusted source.
>
> As a result, Marshal.load is not suitable as a general purpose serialization format and you should never unmarshal user supplied input or other untrusted data.
>
> If you need to deserialize untrusted data, use JSON or another serialization format that is only able to load simple, 'primitive' types such as String, Array, Hash, etc. Never allow user input to specify arbitrary types to deserialize into.

When users upload new gems to the RubyGems service, the application stores the gem itself and a Marshaled representation of the spec data in S3:

```
def write_gem(body, spec_contents)
  gem_path = "gems/#{@version.gem_file_name}"
  gem_contents = body.string
  ...
  spec_path = "quick/Marshal.4.8/#{@version.full_name}.gemspec.rz"
  ...
  RubygemFs.instance.store(gem_path, gem_contents, checksum_sha256: version.sha256)
  RubygemFs.instance.store(spec_path, spec_contents, checksum_sha256: version.spec_sha256)

  Fastly.purge(path: gem_path)
  Fastly.purge(path: spec_path)
end
```

Figure 9.1: Marshaled spec data (`rubygems.org/app/models/pusher.rb#246–258`)

The RubyGems application never uses this Marshaled spec data in an automated way. However, it is used when executing one-off Rake tasks such as `gemcutter:metadata:backfill` or `gemcutter:required_ruby_version:backfill`:

```
def get_spec_attribute(version_full_name, attribute_name)
  key = "quick/Marshal.4.8/#{version_full_name}.gemspec.rz"
  file = RubygemFs.instance.get(key)
  return nil unless file
  spec = Marshal.load(Gem::Util.inflate(file))
  spec.send(attribute_name)
rescue StandardError => e
  Rails.logger.info("[gemcutter:required_ruby_version:backfill] could not get
required_ruby_version for version: #{version_full_name} " \
                    "error: #{e.inspect}")
  nil
end
```

Figure 9.2: Marshal load of spec data (rubygems.org/lib/tasks/helpers/gemcutter_tasks_helper.rb#31–41)

The Marshaled spec data is also used in the `rubygems` repository (i.e., the `gem` and `bundler` commands). Although this project is out of scope, and uses a `SafeMarshal` implementation, it is still interesting to consider. The Marshaled spec data is used in the `Gem::Source` and `Bundler::Fetcher` functionality. The former is commonly employed to exploit Ruby Marshal deserialization bugs. If an attacker can find a bypass in the `SafeMarshal` implementation, or otherwise convince a program to load that data, then they could achieve code execution. Additionally, if a client that is not using the `SafeMarshal` implementation—like the RubyGems Rake tasks—accesses that data, then it is not protected.

While this Marshal functionality does not currently appear to be exploitable, removing it and its associated functionality would reduce risk within the Ruby community as a whole; this would remove both the ability to accidentally unmarshal potentially malicious data and a common exploitation pattern for Ruby code.

Exploit Scenario

An attacker gains access to the S3 bucket storing Marshaled spec data. This could occur via access control misconfiguration, SSRF, data validation issues during the gem upload process, or some other attack vector. The attacker then uploads or modifies Marshaled data such that when it is loaded, it exploits a deserialization bug.

Recommendations

Short term, ensure that any clients accessing these Marshaled `.rz` files under the `rubygems` GitHub organization are using the `SafeMarshal` implementation.

Long term, consider removing functionality associated with these `.rz` files, or moving to a safer serialization format such as JSON. Alternatively, if the RubyGems service needs to

provide this information externally, then consider storing this metadata in the production database instead of serialized data files in S3.

10. Unhandled exception in trusted publisher exchange token request

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-RGM-10
Target: `app/controllers/api/v1/oidc/trusted_publisher_controller.rb`	

Description

Calling the `exchange_token` action from the `Api::V1::OIDC::TrustedPublisherController` controller without any parameters results in a 500 error due to an unhandled exception (figure 10.1). This occurs because the `ActionController::ParameterMissing` error handler code from `ApplicationController` (figure 11.2) calls a wrong `render_bad_request` function. The handler code is executed in the context of the deriving controller, which overrides the `render_bad_request` function (figure 10.3). The overridden function has no arguments, but the error handler calls `render_bad_request` with an argument, which causes an `ArgumentError` to be thrown and an internal server error.

```
curl -XPOST https://rubygems.org/api/v1/oidc/trusted_publisher/exchange_token.json
{"status":500,"error":"Internal Server Error"}
```

Figure 10.1: A controller crash triggered by a request without parameters

```
rescue_from(ActionController::ParameterMissing) do |e|
  render_bad_request "Request is missing param '#{e.param}'"
end
```

Figure 10.2: Error handler calls `render_bad_request` with an argument (rubygems.org/app/controllers/application_controller.rb#L54–L56)

```
def render_bad_request
  render json: { error: "Bad Request" }, status: :bad_request
end
```

Figure 10.3: Overridden `render_bad_request` expects no arguments (rubygems.org/app/controllers/api/v1/oidc/trusted_publisher_controller.rb#L69–L71)

Recommendations

Short term, remove the `render_bad_request` override from the `Api::V1::OIDC::TrustedPublisherController` controller. There is already an equivalent `render_bad_request` in `Api::BaseController` that will be used by the exception handler.

Long term, add test cases to cover all error paths in controllers.

11. Insufficient JWT validation in trusted publisher exchange token request

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RGM-11
Target: `app/controllers/api/v1/oidc/trusted_publisher_controller.rb`	

Description

The JWT claims received from trusted publishers are insufficiently validated and result in triggerable unhandled exceptions. For instance, a missing `nbf` claim results in a `NoMethodError: undefined method `+' for nil` exception and `nbf` containing a string instead of an integer `ArgumentError: bad value for range` exception.

```
def verify_signature
  raise UnsupportedIssuer, "Provider is missing jwks" if @provider.jwks.blank?
  raise UnverifiedJWT, "Invalid time" unless
(@jwt["nbf"]..@jwt["exp"]).cover?(Time.now.to_i)
  @jwt.verify!(@provider.jwks)
end
```

Figure 11.1: JWT claims used without any validation (rubygems.org/app/controllers/api/v1/oidc/trusted_publisher_controller.rb #L52–L56)

Recommendations

Short term, validate all the JWT claims in the controller and add tests exercising missing and incorrect claim values.

Long term, always validate all untrusted data as soon as it arrives into the system.

12. SSRF risk while refreshing OIDC providers

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RGM-12
Target: `app/jobs/refresh_oidc_provider_job.rb`	

Description

The `RefreshOIDCProviderJob` uses the provider-controlled `provider.configuration.jwks_uri` to perform a GET request (figure 12.1). As a result, an attacker can perform SSRF attacks against internal hosts. The attack vector is limited since it relies on a compromised trusted provider.

```
def perform(provider:)
  connection = Faraday.new(provider.issuer, request: { timeout: 2 }, headers: {
"Accept" => "application/json" }) do |f|
    f.request :json
    f.response :logger, logger, headers: false, errors: true, bodies: true
    f.response :raise_error
    f.response :json, content_type: //
  end
  resp = connection.get("/.well-known/openid-configuration")

  provider.configuration = resp.body
  provider.configuration.validate!
  provider.jwks = connection.get(provider.configuration.jwks_uri).body

  provider.save!
end
```

Figure 12.1: Calling a URL provided by a third party (rubygems.org/app/jobs/refresh_oidc_provider_job.rb#L7–L21)

Exploit Scenario

An attacker takes control of the provider's `/.well-known/openid-configuration` endpoint and sets `jwks_uri` to perform GET requests on the internal network. While this is typically not an exploit on its own, SSRF vulnerabilities can commonly be combined with other issues to achieve higher impact.

Recommendations

Short term, proxy web hook HTTP requests through an application like Smokescreen. This will prevent requests from reaching internal address space.

Long term, write automated tests to ensure that all web hook requests are proxied and cannot reach internal address space.

13. The jti claim uniqueness is not enforced by database

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-RGM-13
Target: `app/models/oidc/id_token.rb`	

Description

The `jti` claim from OIDC ID token is validated for uniqueness (figure 13.1), but this constraint is not enforced by the database (figure 13.2), which makes the application vulnerable to race conditions.

```
def jti_uniqueness
  relation = self.class.where("(jwt->>'claims')::jsonb->>'jti' = ?", jti)
  relation = relation.provider_id(api_key_role.oidc_provider_id) if api_key_role
  return unless relation.where.not(id: self).exists?
  errors.add("jwt.claims.jti", "must be unique")
end
```

Figure 13.1: The `jti` claim uniqueness validation in application code (rubygems.org/app/jobs/refresh_oidc_provider_job.rb#L7–L21)

```
create_table "oidc_id_tokens", force: :cascade do |t|
  t.bigint "oidc_api_key_role_id", null: false
  t.jsonb "jwt", null: false
  t.bigint "api_key_id"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index ["api_key_id"], name: "index_oidc_id_tokens_on_api_key_id"
  t.index ["oidc_api_key_role_id"], name:
"index_oidc_id_tokens_on_oidc_api_key_role_id"
end
```

Figure 13.2: Missing constraint ensuring that the `jti` value is unique (rubygems.org/db/schema.rb#L360–L368)

Recommendations

Short term, add a check constraint to the `oidc_id_tokens` table to ensure that the `jti` value is unique even in face of a race condition in the application code.

Long term, always add uniqueness constraints or indexes to the database to ensure data consistency.

14. Provider key age not checked

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-RGM-14
Target: `app/controllers/api/v1/oidc/trusted_publisher_controller.rb`, `app/controllers/api/v1/oidc/api_key_roles_controller.rb`	

Description

The OIDC provider keys are periodically refreshed in a scheduled job. The application code does not check the key age before verifying the signature (figures 14.1, 14.2). If the scheduled refresh job fails for an extended period of time, the JWT signature verification could operate on an outdated, possibly compromised key.

```
def verify_signature
  raise UnsupportedIssuer, "Provider is missing jwks" if @provider.jwks.blank?
  raise UnverifiedJWT, "Invalid time" unless
(@jwt["nbf"]..@jwt["exp"]).cover?(Time.now.to_i)
  @jwt.verify!(@provider.jwks)
end
```

Figure 14.1: Signature verification without checking the key age (rubygems.org/app/controllers/api/v1/oidc/trusted_publisher_controller.rb#L52–L56)

```
def decode_jwt
  raise UnverifiedJWT, "Provider missing JWKS" if @api_key_role.provider.jwks.blank?
  @jwt = JSON::JWT.decode_compact_serialized(params.permit(:jwt).require(:jwt),
@api_key_role.provider.jwks)
rescue JSON::ParserError
  raise UnverifiedJWT, "Invalid JSON"
end
```

Figure 14.2: Signature verification without checking the key age (rubygems.org/app/controllers/api/v1/oidc/api_key_roles_controller.rb#L72–L77)

Recommendations

Short term, add a sanity check for the key age before verifying the JWT signature.

15. Overly verbose error returned to the user

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-RGM-15
Target: `app/controllers/api/v1/oidc/trusted_publisher_controller.rb`, `app/controllers/api/v1/oidc/api_key_roles_controller.rb`	

Description

The `Pusher::pull_spec` method contains a rescue clause that catches `StandardError`, which is almost equivalent to catching all exceptions. The exception message is included in the HTTP response. It is in principle unsafe to share arbitrary exception messages with users, as the message might contain sensitive information.

```
rescue StandardError => e
  notify <<~MSG, 422
    RubyGems.org cannot process this gem.
    Please try rebuilding it and installing it locally to make sure it's valid.
    Error:
    #{e.message}
  MSG
```

Figure 15.1: The exception message is included in error message presented to users (rubygems.org/app/models/pusher.rb#L87–L93)

```
gemcutter = Pusher.new(@api_key, request.body, request:)
gemcutter.process
render plain: response_with_mfa_warning(gemcutter.message), status: gemcutter.code
```

Figure 15.2: The message from figure 15.1 is rendered in the response (rubygems.org/app/controllers/api/v1/rubygems_controller.rb#L37–L39)

Recommendations

Short term, remove the exception message from the `StandardError` handling in `Pusher::pull_spec`. Alternatively, add more granular exception handling with specialized error messages.

Long term, avoid showing arbitrary error messages to untrusted parties, as they could contain sensitive information.

16. Redundant IAM users and permissions

Severity: **Medium**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-RGM-16
Target: AWS IAM	

Description

The system contains a few IAM resources that appear not to be used and should be removed. Additionally, a few resources that could be more restricted; while not easily exploitable, these resources do not follow the principle of least privilege.

The below IAM users appear not to be used:

- `chef`: this account has some write privileges to EC2 and Route53, does not have any access credentials, has MFA disabled, and has never been used (according to Access Advisor).
- `nick`: this account has administrative access, has only one inactive access key configured, has MFA disabled, and has never been used (according to Access Advisor).

The following IAM users were used long time ago and may be no longer needed:

- `evan`: this account has administrative access, does not have any access credentials, MFA is disabled, was used 2 years ago last time
- `dwradcliffe`: this account has administrative access and was used 2 years ago last time
- `hsbt`: this account has privileges for ECS and production S3 buckets, was used 1 year ago for S3 access and never used other privileges
- `marty`: this administrative account is actively used, but with an old and unused access key.
- `mux`: this account with read-write S3 privileges, MFA is disabled
    - The account has redundant S3 `:CreateJob` permission for all resources
    - Other write permissions are insecurely restricted to `arn:aws:s3:::*videos*` resources: this gives access to, for example, production gems with “videos” in their name

The following roles appear to have been created for OpsWorks service that reached end of life:

- `aws-opsworks-ec2-role`
- `aws-opsworks-service-role`

Please note that the chef IAM user could be created for this service as well.

Multiple customer-managed policies do not have any entities attached:

- `KarpenterController-20240217152813154100000000b`
- `es-staging-app`
- `rubyapi-secrets-manager-read-write`
- `RubyAPITerraformCloud` (not used after fix for TOB-RGM-20)
- `aws-load-balancer-controller` (may be important for Helm configuration)

Additionally, we noted the following issues:

- The `everything-backup` policy (inlined in the `dwradcliffe` user) grants all privileges on all resources, instead of read-only privileges.
- The `Setup` user group has no users assigned, so it can probably be removed.
- The `shipit` IAM user is included in the K8s `system:masters` group, but the IAM user does not exist.

Exploit Scenario 1
The `evan` user reuses his AWS password for another popular service. The other service suffers from data breach, and passwords are leaked. Mallory finds `evan`'s password in the data dump; because MFA is disabled, Mallory successfully logs in as `evan` with only his password. Mallory has admin access to all rubygems production resources and silently installs backdoors in popular gems.

Exploit Scenario 2
The Mux user becomes malicious and backdoors all 23 gems with "videos" in their name.

Recommendations
Short term, review the IAM resources pointed out in this finding, remove redundant resources, and restrict privileges of other resources according to the principle of least privilege. The precise actions required to do this depend on the business context that is unknown to Trail of Bits; nevertheless, consider the following actions:

- For IAM users:
    - Remove the `check` and `nick` IAM users.
    - Enable MFA for the `evan` and Mux users.
    - Remove `dwradcliffe`'s `everything-backup` policy.
    - Limit `hsbt`'s privileges to S3 buckets.

    - Remove the `marty` user's access key.
    - Remove Mux's S3 `:CreateJob` permission.
    - Restrict Mux's write permissions to specific S3 buckets and objects (instead of the broad `*videos*` pattern).
- Remove OpsWorks-related roles. Remove unused policies. Remove the `Setup` user group.
- Ensure that the root account has MFA enabled.
- Adjust EKS' `kubernetes_config_map` (in eks.tf file) and `kubernetes_role_binding` resources (`k8s-rbac.tf` file) to match current IAM resources. Add a step to the development and PR review processes that would require review of K8s configurations whenever IAM resources are changed (and vice versa).
- Long term, periodically review IAM resources and remove redundant ones.
- Move away from manual management of IAM resources. Migrate the management to Terraform. Once the migration is complete, limit users and roles with write access to production resources; ideally, only CI/CD pipeline and a break-glass user should have access.
- When creating or updating IAM resources, ask for peer review of the new configurations. This review should be made a required step in GitHub, and should check if the new permissions are minimal.
- Whenever possible, migrate inlined and attached policies to groups and roles. This will reduce the possibility of granting over-privileged accesses.
- Add AWS Config rules that would automate maintenance: remove inactive IAM resources and disallow insecure configurations. Review asecurecloud's conformance packs for additional rules.

17. MFA is not enforced for IAM user

Severity: **Medium**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-RGM-17
Target: AWS IAM, AWS Config, AWS Identity Center	

Description
MFA is not enforced on the account or organization level. IAM users are allowed to perform actions in AWS without MFA enabled.

While all active users currently have MFA configured, a global enforcement of MFA is strongly recommended.

Please note that the `Force_MFA` policy exists, but it does not serve the purpose of global enforcement: it is attached only to the SRE user group. Moreover, it contains a bug: the `s3:ListBuckets` permission is incorrect (should be `s3:ListBucket`).

Exploit Scenario
A new user is added to the AWS IAM. The user forgets to configure MFA. Some time later, Mallory compromises the user's password via phishing. Mallory logs in to AWS.

Recommendations
Short term, create a AWS Config check to enforce MFA or enable IAM Identity Center and enable enforcement. Ensure that root accounts have strong MFA enabled.

Long term, use only hardware security keys for MFA, as these types of authenticators provide the strongest security. Administrators especially should use this MFA option. Configure at least two keys: one for day-to-day use and one as a backup.

18. Lack of policy conditions leading to cross-service confused deputy attacks

Severity: **Low**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-RGM-18
Target: AWS IAM roles	

Description
Some AWS services are granted `sts:AssumeRole` permission without any policy condition, and are potentially vulnerable to cross-service confused deputy attacks.

It is not clear if the issues are really exploitable, and if other services with the `sts:AssumeRole` permission are not vulnerable; AWS documentation states that only some services support mitigations for this attack vector. For the three services indicated below, we found explicit documentation for mitigating the confused deputy attack.

- arn:aws:iam::048268392960:role/rds-monitoring-role
- arn:aws:iam::048268392960:role/vpc-flow-log-role-20240501052116925800000002
- arn:aws:iam::048268392960:role/aws-opsworks-service-role

Other services are also missing policy conditions, but we found no similar AWS recommendations and therefore assume that no steps are needed. We still recommend mitigating the attack for extra safety, although implementing mitigations in some services may break the system and should be preceded by careful investigation.

Exploit Scenario
An attacker creates a new AWS account and configures a RDS service to write logs to RubyGems' RDS logs.

Recommendations
Short term, mitigate the confused deputy attacks by adding relevant `aws:` conditions to the policies of roles listed in the Description section above.

Consider implementing mitigations for the following lambda roles. (Note that mitigations for lambda service may be not necessary and are not documented by AWS, but should still work.)

- `arn:aws:iam::048268392960:role/lambda_s3_exec_role`

- `arn:aws:iam::048268392960:role/lambda-basic-exec-role`
- `arn:aws:iam::048268392960:role/SlackNotifier-LambdaFunctionRole-12R6ZJ2BX3B1W`
- `arn:aws:iam::048268392960:role/SlackNotifier-LambdaFunctionRole-1941TUMK699WF`

Long term, review security documentation for AWS services that will be enabled for RubyGems. Some important configuration options may be specific to the service and "hidden" inside its documentation.

References

- `aws:SourceArn` documentation
- Summit Route, 2019.04.03, "Advanced AWS policy auditing - Confused deputies with AWS services"

19. External GitHub CI actions are not pinned

Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-RGM-19
Target: `rubygems.org` `lint.yml` and `rubygems-terraform` GitHub Actions	

Description

The `rubygems.org` GitHub Action pipeline uses a third-party `kubeconform` action. This action is part of the supply chain for CI/CD and can execute arbitrary code in the CI/CD pipelines. The action is pinned by version, and not by commit hash.

```
- name: kubeconform
  uses: docker://ghcr.io/yannh/kubeconform:v0.6.3
  with:
    entrypoint: "/kubeconform"
    args: "-strict -summary -output json --kubernetes-version ${{
matrix.kubernetes_version }} config/deploy/${{ matrix.environment }}.rendered.yaml"
```

Figure 19.1: The action pinned by version (rubygems.org/.github/workflows/lint.yml#67–71)

Moreover, no third-party actions in `rubygems-terraform` configuration are pinned by hash, although all of these actions are trusted.

Exploit Scenario

An attacker uses social engineering to take over a private GitHub account with write permissions for one of the untrusted GitHub actions. For example, a user uses an already-leaked password and is convinced to send a 2FA code to the attacker. The attacker updates the GitHub action to include code to exfiltrate all secrets in CI/CD pipelines that use the action.

Recommendations

Short term, pin all external and third-party actions to a Git commit hash. Avoid pinning to a Git tag, as these can be changed after creation. We also recommend using the `pin-github-action` or `frizbee` tool to manage pinned actions. GitHub dependabot is capable of updating GitHub Actions that use commit hashes.

Long term, audit all pinned actions or replace them with a custom implementation.

20. Terraform OIDC provider has insecure configuration

Severity: **Low**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-RGM-20
Target: `AWS IAM roles`	

Description

Two Terraform OIDC configurations are insecure: they do not validate the `sub` fields of authentication tokens:

- `arn:aws:iam::048268392960:role/RubyAPITerraformProvisioner`
- `arn:aws:iam::048268392960:role/RubyAPITerraformCloud`

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Federated":
"arn:aws:iam::048268392960:oidc-provider/app.terraform.io"
            },
            "Action": "sts:AssumeRoleWithWebIdentity",
            "Condition": {
                "StringEquals": {
                    "app.terraform.io:aud": "aws.workload.identity"
                }
            }
        }
    ]
}
```

Figure 20.1: Insecure role configuration

The `RubyAPITerraformCloud` role has no permission attached, but the `RubyAPITerraformProvisioner` has the `RubyAPITerraformCloud` permission policy attached. It gives the role partial read-only access to EC2 and EKS resources, as well as the `ec2:createVpc` permission. More importantly, write permissions are given to EC2 and EKS resources tagged as "`rubyapi`".

The severity of this finding is low because the `rubyapi` project is out of this audit's scope; for that project, the severity is high.

Exploit Scenario

Mallory creates a new account in HashiCorp. Due to the lack of sub claim validation, her HashiCorp account is able to configure the OIDC to assume RubyGems' `RubyAPITerraformProvisioner` role.

Mallory creates a new VPC configuration that later is incidentally attached to production EC2 instances. The new VPC enables public access to private, internal services.

Recommendations

Short term, remove the two insecure OIDC configurations.

Long term, migrate all configurations to Terraform and do not manually create or update AWS resources. Doing so will enforce that all changes go through GitHub code-update workflows like peer reviews and approvals.

References

- Eduard Agavriloae, Aug 15, 2024, "Addressed AWS defaults risks: OIDC, Terraform and Anonymous to AdministratorAccess"
- HashiCorp documentation, "Dynamic Credentials with the AWS Provider"

21. RubyGems and RubyAPI projects are under a single AWS account but different Terraform configurations

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-21
Target: `AWS account`	

Description
AWS resources for the RubyGems project are managed with Terraform. Another project, `rubyapi`, is not managed via Terraform (or its configurations are stored in a different GitHub repository), but is deployed within the same AWS account as RubyGems. This design does not follow the principle of least privilege and creates risks of lateral movement: bugs in one project may be used to compromise the other one.

Exploit Scenario
A vulnerability in `rubyapi`'s AWS configuration enables attackers to modify all EC2 resources. The attacker exploits this bug to make malicious changes in RubyGems' EC2 instances.

Recommendations
Short term, implement one of the following recommendations:

1. Migrate all projects that use the single AWS account to Terraform.
2. Move `rubyapi` (and possibly other projects) to a separate AWS account.

The decision of which solution to choose depends mostly on business context like the structure of the Ruby Central team. However, we recommend the second option, as it provides stronger isolation and will help prevent not only attacks but also configuration issues.

If the second option is chosen, then disable unused AWS Regions after projects are separated. Unused but enabled regions may be used by attackers for stealth. Resources assigned to regions can be found with, for example, AWS Tag Editor.

Long term, list and review all Ruby Central projects and AWS accounts in the organization, and ensure that every project is either centrally managed or in a separate AWS account. When creating new projects, move them to new AWS accounts from the beginning.

22. SQS policy has S3-based policy without account

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-22
Target: `AWS SQS policy`	

Description

The `fastly_logs_production` (and staging) SQS allows the `rubygems-fastly-downloads-production` S3 bucket to send messages to the SQS. The bucket is not limited by the `aws:sourceAccount` condition. If the bucket is removed and claimed by an attacker in any AWS account, the attacker will be able to send messages to the queue.

```
{
  "Version": "2008-10-17",
  "Id": "policy-allow-from-s3",
  "Statement": [
    {
      "Sid": "sid-allow-from-S3",
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "SQS:SendMessage",
      "Resource": "${sqs_arn}",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "${bucket_arn}"
        }
      }
    }
  ]
}
```

Figure 22.1: SQS configuration, `aws:SourceArn` condition does not contain AWS account (`rubygems-terraform/modules/rubygems-org-app/policies/fastly_logs_sqs_policy.tpl#1–21`)

Please note that Fastly is permanently vulnerable to similar attacks, since this service does not validate the account of an S3 owner before reading data or writing logs. That is, if a S3 bucket is removed by Ruby Central and claimed by an attacker while the Fastly configuration remains unchanged, then the attacker could control data exposed on Ruby

Central HTTP domains. We are not aware of any mitigations that could be configured in Fastly.

Exploit Scenario

The Ruby Central team changes the bucket allowed to write to SQS but forgets to create the new bucket. An attacker notices the issue and claims the bucket. The attacker can write Fastly logs.

Recommendations

Short term, add the `aws:SourceAccount` condition to the `policy-allow-from-s3` policy. Change the `Principal` field of the policy to be equal to `bucket_arn` as a defense-in-depth protection.

Long term, create a GitHub workflow that lists S3 buckets referenced in the Fastly configuration, and check if the owner of all the buckets is the expected AWS account (see also TOB-RGM-29). Alternatively, reference all S3 buckets by variables and not with hard-coded names in the Fastly configuration; this change should prevent usage of buckets not managed via Terraform.

23. GitHub Action Terraform policy is overly broad

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-23
Target: `rubygems-terraform/iam.tf`	

Description

GitHub Actions for `rubygems-terraform` repository (master branch) are granted all permissions to all AWS resources. The permissions should not include full access to some services, including:

- Logging services like VPC Flow Logs and `CloudWatch`. Compromise of the repository should not enable attackers to remove all logs.
- Monitoring and alerting services like `GuardDuty`. Compromise of the repository should not enable attackers to prevent the Ruby Central team from detecting the incident in a timely manner.
- Global configurations like AWS Config. Compromise of the repository should not enable an attacker to disable global security configurations like the MFA requirement.
- Creation of unexpected AWS services should not be possible. If a new service is used, then permissions to it should be explicitly granted through a change to the `github-actions-terraform-policy`. While not a strict security boundary, this recommendation should help detect malicious code changes in the repository.

Moreover, management of other projects in the account (like RubyAPI) should not be possible from the repository (see also TOB-RGM-21).

```
data "aws_iam_policy_document" "github-actions-terraform-policy" {
  statement {
    effect = "Allow"
    condition {
      test     = "ForAnyValue:StringLike"
      variable = "token.actions.githubusercontent.com:sub"
      values   = ["repo:rubygems/rubygems-terraform:ref:refs/heads/master"]
    }
    resources = ["*"]
    actions   = ["*"]
  }
```

```
  statement {
    effect    = "Allow"
    resources = ["arn:aws:secretsmanager:us-west-2:048268392960:secret:terraform/*"]
    actions   = ["secretsmanager:GetSecretValue"]
  }
}
```

Figure 23.1: Overly broad permissions for GitHub Actions (`rubygems-terraform/iam.tf#132–148`)

To prevent GitHub Actions for `rubygems-terraform` repository from abusing IAM permissions for privilege escalation, there are two possible solutions:

- Access policies boundaries: IAM resources created via Terraform would be restricted with the same allowlist/denylist used to restrict the `github-actions-terraform-policy`.
- Separate repository for IAM management: if changes to IAM are expected to be less frequent and less urgent than changes to other services, then having a separate code control repository with higher AWS privileges (to manage IAM resources), but with stricter GitHub security configuration (like strict branch protection enabled), would help to mitigate the privilege escalation attacks.

Exploit Scenario

Mallory compromises the `rubygems-terraform` repository. She turns off monitoring and alerting AWS services, and then installs backdoors in a few gems. Later, when the attack is detected, she uses the repository to purge all AWS data, including all logs. It is much harder to perform incident response and investigate the attack.

Recommendations

Short term, limit permissions in the `github-actions-terraform-policy` policy:

- Only allow access to services that are actually managed via Terraform.
- Deny destructive access to logging, monitoring, alerting, and configuration services. These services generally should be set up once and not disabled or modified afterwards. For examples of access denylists, see the links in the Description section above.

Additionally, restrict resources created/managed via Terraform with the same restrictions as stated above. To do this, either use access policy boundaries or remove IAM permissions from the policy and create a separate, more hardened GitHub Terraform repository and role for only IAM management.

Long term, periodically review existing policies and roles and minimize privileges to the necessary ones. Trim other full-access policies according to the recommendations above.

24. Networking security concerns

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-24
Target: `AWS networking`	

Description

There are a few insecure configurations related to networking. While none are currently exploitable, they may result in severe vulnerabilities in the future.

Ingress SSH access on port 22 is enabled for CIDR blocks `0.0.0.0/0` in the following security groups:

- `rubygems-common-all`
  - This group is used by the `arn:aws:elasticache:us-west-2:048268392960:cluster:shipit2` cluster. The cluster is not exposed outside of VPC, but opening port 22 is redundant, or at least the CIDR block is too open.
- `rubygems-common-chef`
  - This group is not used and appears to be a leftover.

The Amazon EKS (K8s) API endpoint is reachable through the public internet. The `endpoint_public_access` should be set to `false` to limit access to K8s admin endpoint.

```
vpc_config {
  security_group_ids      = [aws_security_group.eks-cluster.id]
  endpoint_private_access = true
  endpoint_public_access  = true
```

Figure 24.1: Configuration enabling public K8s endpoint (`rubygems-terraform/eks.tf#57–60`)

EKS-managed nodes have the `associate_public_ip_address` flag set to `true`. This results in public IPs being assigned to the nodes. The issue is not exploitable, as the external access is blocked by security groups; however, public IPs increase the chance of exploitable misconfigurations.

```
resource "aws_launch_template" "eks-managed-nodes" {
  name = "eks-managed-nodes"
```

```
  network_interfaces {
    associate_public_ip_address = true
    security_groups = [
      aws_security_group.eks-node.id
    ]
  }
```

Figure 24.2: Configuration enabling automatic association of public IP addresses (rubygems-terraform/eks.tf#416–424)

All three subnets configured via Terraform have the `map_public_ip_on_launch` flag set to `true`. This configuration makes the AWS assign public IP addresses to new resources, which may unintentionally expose them to the internet. Please note that this flag and the `associate_public_ip_address` flag may overwrite each other.

```
resource "aws_subnet" "main-a" {
  vpc_id                  = aws_vpc.rubygems.id
  cidr_block              = "172.30.0.0/24"
  availability_zone       = "us-west-2a"
  map_public_ip_on_launch = true
```

Figure 24.3: Configuration enabling automatic mapping of public IP addresses (rubygems-terraform/subnets.tf#1–5)

Currently all resources with public IPs are protected by security groups and are not remotely reachable. For example, IP `52.38.177.176` (network interface `eni-0bdca602657463234`) has a security group that blocks any internet access.

Finally, network ACLs are not used. These should be configured and should correspond to security groups of the VPC as a defense-in-depth mechanism.

Exploit Scenario

A vulnerability in Kubernetes is exploited by a third party by directly accessing the EKS K8s management endpoint. The attacker takes control of the cluster and installs backdoors into popular gems.

Recommendations

Short term, harden the network configuration by implementing the following recommendations:

- Remove ingress port 22 from `rubygems-common-all` security group, or at least restrict the CIDR block to a few IP addresses. Remove the `rubygems-common-chef` security group.
- Disable public access to EKS management endpoint. See this EKS control for more information.

- Set the `associate_public_ip_address` flag to `false` in the EKS configuration. See this CodeGuru detector for more information.
- Set the `map_public_ip_on_launch` flag to `false` for all subnets. See this EC2 control for more information. For Load Balancers (that must have public IPs), either make the IP map explicit or move them to a separate subnet with the flag set to `true`.
- Review public IP addresses assigned to AWS resources and remove them if they are not needed.

Long term, add network ACLs that would create an additional layer of defense in case of misconfigurations.

25. GitHub token may be exposed in packed data in a release-gem action

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-RGM-25
Target: `release-gem/action.yml`	

Description

The `release-gem` GitHub Action stores GitHub authentication tokens in the filesystem in the `.git/config` file. This may result in the tokens being leaked if action's user creates an artifact from the repository's root.

```
steps:
  - name: Set remote URL
    run: |
      # Attribute commits to the last committer on HEAD
      git config --global user.email "$(git log -1 --pretty=format:'%ae')"
      git config --global user.name "$(git log -1 --pretty=format:'%an')"
      git remote set-url origin "https://x-access-token:${{ github.token
}}@github.com/$GITHUB_REPOSITORY"
```

Figure 25.1: Action saving GitHub token in the filesystem (`release-gem/action.yml#18–24`)

The severity of this finding is only informational, as a breaking change was recently introduced in the `upload-artifact` action that prevents uploads of hidden folders.

Exploit Scenario

Alice has a GitHub workflow for publishing her gem. Her workflow uses the `release-gem` action, and it produces an artifact after the action but before the workflow finishes. Mallory, an unprivileged user, steals GitHub tokens during workflow runs and uses them to install backdoors in Alice's gem.

Recommendations

Short term, remove the GitHub token from the "Set remote URL" step. If the token is necessary for some further steps, then insert the credentials explicitly in those steps in a way that will not make them persist in the filesystem (e.g., via the step's environment variables).

References

- Yaron Avital, 13 August, 2024, "ArtiPACKED: Hacking Giants Through a Race

Condition in GitHub Actions Artifacts”
- GitHub documentation, “Automatic token authentication”

26. GitHub Action redundant persist-credentials

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-RGM-26
Target: `rubygems.org` and `rubygems-terraform` GitHub Actions	

Description

All workflows (except the `scorecards`) in the `rubygems.org` and `rubygems-terraform` repositories use the default configuration for the `actions/checkout` action. The default configuration for the `persist-credentials` flag is `true`. This makes the action persist GitHub tokens in a filesystem.

While this issue is not exploitable since the filesystem is never publicly exposed, the tokens' persistence seems to be not required by subsequent steps.

Recommendations

Short term, add explicit `persist-credentials` flags with `false` values to all `actions/checkout` actions used in workflows. If some steps need the GitHub credentials, then plumb them explicitly. Add the flag to the `releasing-gem` documentation so that users who are copy-pasting the code use the more secure configuration.

References

- The actions/checkout Issue #485

27. Fastly uses long-lived credentials instead of OIDC

Severity: **Informational**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-RGM-27
Target: `rubygems-terraform/fastly`	

Description

Terraform configures the Fastly service to use long-lived AWS credentials for logging. OIDC is the recommended way of authenticating Fastly to AWS. In other words, a new AWS role should be created for Fastly, instead of an AWS user.

```
logging_s3 {
  name           = "downloads"
  format         = "%h now req.url %>s req.http.User-Agent"
  format_version = 1
  bucket_name    = var.downloads_bucket_name
  s3_access_key  = var.downloads_s3_access_key
  s3_secret_key  = var.downloads_s3_secret_key
```

Figure 27.1: Configuration using long-lived AWS credentials for Fastly logging (`rubygems-terraform/fastly/modules/main.tf#279–285`)

Unfortunately, Fastly does not allow OIDC-like authentication for accessing private S3 buckets as data sources (origins). Using long-lived AWS credentials is non-avoidable.

Recommendations

Short term, consider implementing authentication for Fastly logs using AWS roles. This change should slightly improve security. However, this change requires creating both user and roles for Fastly, which would increase overall complexity.

References

- Fastly documentation, "Log streaming: Amazon S3"

28. Stored XSS in gems.rubygemsusercontent.org

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RGM-28
Target: `AWS S3 buckets and Fastly`	

Description

The `gems.rubygemsusercontent.org` domain serves user-controlled files with either user-controlled or automatically deduced content types. This behavior enables users to create persistent XSS exploits in the domain.

The domain is different from the more important RubyGems.org, so the severity of this issue is very limited. However, users may trust the domain (as it belongs to Ruby Central), and instances of XSS inside it may be unexpected. Moreover, there seems to be no business value in serving user files with insecure content types.

The `gems.rubygemsusercontent.org` domain serves files from the `contents.oregon.production.s3.rubygems.org` S3 bucket (figures 28.1–2). The bucket holds objects created from gems source files (controlled by gems creators). The files are served with the HTTP response `Content-Type` header taken from the object's metadata (figure 28.3).

```
contents = {
  host        = "gems.rubygemsusercontent.org"
  bucket_name = "contents.oregon.production.s3.rubygems.org",
}
```

Figure 28.1: Part of the Fastly configuration (rubygems-terraform/fastly/main.tf#126–129)

```
condition {
  name      = "is_contents_s3_request"
  priority  = 9
  statement = "req.http.host == \"${var.contents.host}\""
  type      = "REQUEST"
}
```

Figure 28.2: Part of the Fastly configuration (rubygems-terraform/fastly/modules/main.tf#185–190)

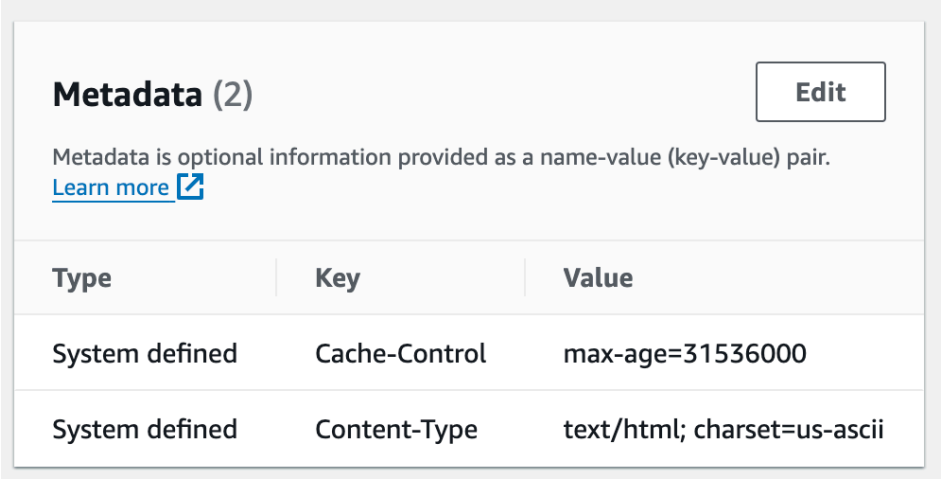

Figure 28.3: Example AWS metadata entries

The metadata is either controlled by gems owners or set automatically by AWS. In either case, publishing a gem with an HTML file is sufficient to create an XSS payload.

As an example, the `sogilis/csv_fast_importer` repository stores a `422.html` file. The file's reference can be found with this link. Then, if opened with this URL, the HTML file is served with the HTML content type.

Exploit Scenario

Mallory publishes a gem with a malicious HTML file. The file contains JavaScript payload for BeEF. Mallory easily phishes users to visit the website, as the users believe the domain to be trusted.

Recommendations

Short term, find how the `Content-Type` metadata is created when uploading objects to S3 buckets. If the metadata is user-provided, then create an allowlist of content types. If the metadata is automatically deduced, then add an automatic request (sent after object creation) that would update the `Content-Type` metadata to text/plain or octet/stream.

In either case, configure Fastly to overwrite content types of HTTP responses when serving files from S3 buckets with text/plain and octet/stream.

29. OIDC S3 buckets are not claimed but are configured in Fastly

Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-RGM-29
Target: `rubygems-terraform/fastly/oidc-api-tokens.tf`	

Description

The Fastly configuration (figure 29.1) exposes a few subdomains of RubyGems.org with S3 buckets as data sources. The buckets are not claimed by anybody. An attacker can claim them, effectively taking control over the subdomains.

```
contents = {
  host        = "gems.oidc-api-token.rubygemsusercontent.org"
  bucket_name = "contents.oregon.oidc-api-token.s3.rubygems.org",
}

compact_index = {
  bucket_name = "compact-index.oregon.oidc-api-token.s3.rubygems.org"
}
compact_index_host = "oidc-api-token.rubygems.org"
```

Figure 29.1: Vulnerable S3 buckets and subdomains (`rubygems-terraform/fastly/oidc-api-tokens.tf#34–42`)

The issue can be observed here.

Exploit Scenario

Mallory finds out that the `compact-index.oregon.oidc-api-token.s3.rubygems.org` S3 bucket is not registered. She registers it and stores malicious JavaScript payloads there. Mallory can conduct XSS-like attacks, try to circumvent CSRF protections, and exploit possible misconfigurations of the RubyGems.org application.

Recommendations

Short term, either claim the OIDC-related S3 buckets or remove relevant Fastly configurations.

Long term, create a GitHub workflow that lists the S3 buckets referenced in the Fastly configuration. Check if the buckets are registered and if the owner of all the buckets is the expected AWS account.

For the first task (list S3 buckets), the `terraform` command can be used (the command is already used in the `terraform.yml` workflow). The second task requires AWS credentials; however, the `terraform.yml` workflow already has the `github-actions-terraform` AWS role, so it has necessary permissions for ownership checks.

Alternatively, manage all S3 buckets exclusively through Terraform. With this change, all buckets will have to be referenced via variables (and not hard-coded ARN names), and if a bucket is deleted, it would not be possible to reference it.

30. IAM fastly user cannot list S3 objects

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-RGM-30
Target: AWS IAM, Fastly	

Description

The AWS `fastly` user does not have the `s3:ListBucket` permission for contents and compact-index buckets (in production and staging). As a result, an error is returned when opening, for example, the `https://gems.rubygemsusercontent.org/` URL. Moreover, the error is verbose and returns potentially sensitive information like AWS account ID and bucket name.

Figure 30.1: The error returned by Fastly for some websites

Recommendations

Short term, either add the `s3:ListBucket` permission to the fastly user or add a configuration to Fastly that would make the returned error less explicit.

31. No dual approval requirement for production deployments	
Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-31
Target: AWS IAM, Fastly	

Description
The RubyGems system is deployed in AWS based on Terraform files stored in the `rubygems-terraform` GitHub repository. Changes to the repository do not require dual approval (confirmation of changes by at least two people).

Moreover, as the Wiki/Deploys page informs, application deployments to the EKS via the Shipit application also do not require dual approval.

Exploit Scenario 1
A GitHub user with access to the `rubygems-terraform` repository is compromised. The attacker makes changes to the Terraform files and pushes them to master. The GitHub Actions run the workflow that automatically updates AWS production resources. The attack goes unnoticed for a few days.

Exploit Scenario 2
An honest user with access to the Shipit application makes a mistake and chooses a wrong commit for production deployment. A broken application is deployed. The RubyGems application is down for a few hours.

Recommendations
Short term, ensure that all users with access to the `rubygems-terraform` repository and Shipit application have 2FA enabled and follow GitHub's other security recommendations. Ideally, developers should use only security keys/WebAuthn 2FA (instead of TOTP), and should manage their SSH keys with hardware security keys.

Implement branch protection rules for `rubygems-terraform`. At a minimum, require pull requests and an approval before merging, e.g.:

- Require a pull request before merging
    - At least one approval
    - Dismiss stale pull request approvals when new commits are pushed

        - Require approval of the most recent reviewable push
    - Block force pushes

Because of the small size of the Ruby Central team, the above recommendation may not be viable: emergency changes would be paused until other team members—likely living in different time zones—would come online. On the other hand, emergency events should be rare.

For a more balanced solution, we recommend configuring the protections above and enabling "break-glass" access. The break-glass would allow bypassing branch protections, but in a noisy way: alerts to out-of-band services (like Slack or emails) would be sent, and other team members would be asked to review the PRs merged without approval when they come online. These notifications should be more visible than normal events (like regular PR merging).

This recommendation would not prevent attacks in case of GitHub account compromise, but would help the team to detect and respond to such incidents.

A possible drawback is a user fatigue problem: if noisy alerts are sent too often, developers will start ignoring them. Therefore, this recommendation is only useful if the emergency PRs would be rare. If that is not currently the case, these recommendations should be reviewed again once the development process becomes more stable.

As for possible implementation, the `emergency-pull-request-probot-app` application could be used. Its security posture is undetermined, but since it is GitHub's application, it should be fine to use.

Finally, there is the problem of a compromised GitHub account changing branch protection rules. If all developers are given full administrative rights to the repository, then all recommendations given above are effectively not useful: an attacker can first disable the protections, then introduce malicious changes, and then re-enable protections. We were unaware of the specific structure of the Ruby Central team during the audit, more consideration is required to determine the viability of the above recommendations. If the `rubygems-terraform` repository has too few maintainers, it is possible that organization-level rulesets can be used to protect the repository while making the "break-glass" solution useful.

If possible, configure dual approval requirements in the Shipit application, possibly with a similar "break-glass" mechanism.

32. The app-production user's API key has not been rotated for a long time

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-RGM-32
Target: AWS IAM	

Description
The AWS access keys for the `app-production` and `app-staging` users were created about six years ago. These keys give write access to production/staging S3 buckets. Keys are mounted in K8s pods. The keys should be rotated as soon as possible. Moreover, the key should be rotated regularly at least once every two years.

Exploit Scenario
The `app-production` user's API key was compromised a few years ago via undisclosed vulnerabilities in old versions of the RubyGems Rails application. An attacker kept the key unused until now. The attacker backdoors a few popular gems by manipulating S3 buckets' content.

Recommendations
Short term, manually rotate the API keys as soon as possible. Create a process for rotating the keys regularly, at least once every two years. Ideally, key rotation should be automated and done during every new deployment. Such automation will mitigate the risk of not rotating the key after a vulnerability fix.

33. Staging deployments are performed with a highly privileged K8s role

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RGM-33
Target: AWS EKS, `shipit`	

Description

The Shipit deployment application uses the `shipit` `ServiceAccount` to manage K8s deployments. The account is in the `cluster-admin` groups of both staging and production namespaces. The `staging` GitHub branch is automatically deployed to the K8s staging environment. If a buggy configuration is pushed to the branch, it may impact production K8s resources.

Exploit Scenario 1

A buggy K8s configuration is pushed to the `staging` branch. Shipit automatically deploys the changes. Production resources are accidentally impacted, and rubygems is down for a few hours.

Recommendations

Short term, create separate K8s accounts for staging and production environments. Make the staging deployments use the lower-privileged account. Details of the implementation depend on the Shipit features and the available configuration options.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications or that were discovered but not fully investigated due to time constraints or scope limitations.

1. **The `ingress` and `egress` arguments are not recommended in Terraform files.** The `aws_vpc_security_group_{ingress,egress}_rule` resources should be used instead.

2. **Terraform files should consistently use the recommended `jsonencode` function for inline policies.** A "hand-written" JSON policy, as in figure C.2.1, is not recommended.

```
 assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
```

Figure C.2.1: Example policy written in the not recommended style (`rubygems-terraform/eks.tf#99–112`)

3. **The `/app/tmp/local_secret.txt` file is not removed from Docker images published in the RubyGems' ECR.** The images are stored in a private ECR registry, but with read permissions for all AWS users; they should therefore be treated as publicly available. Since secret is used only in Rails testing builds, leaking it does not have serious security consequences; nevertheless, the exposure is unnecessary, and we recommend updating the Docker build script so that the file is removed.

4. **The `is_s3_request` condition in Fastly configuration should use only the request's path and not the whole URL (with query parameters).** For example, the following URL would result in a match: `/gems/example-1.0.0?some-param&.gem`. This issue may result in exposure of unintended files from the S3 bucket.

5. **The `request_conditions` in the Fastly configuration should be mutually exclusive.** For example, the `is_s3_request` and `is_contents_s3_request` conditions may both be true for a single request, making it possible to download gems from both gems.rubygemsusercontent.org and RubyGems.org.

6. **Infinite client-side redirect loop.** Fastly and/or nginx are misconfigured, resulting in an infinite redirect loop for some requests. An example command to observe the issue is given in figure C.6.1.

```
curl -L -k -v
'https://754b69fa-rubygemsproductio-4184-1999396523.us-west-2.elb.amazonaws.com/
```

Figure C.6.1: Example command to trigger infinite redirects loop

7. **Not all RubyGems AWS resources are managed by Terraform.** For example, the `rubygems-eks-role` is created manually, but is referenced in the `eks.tf` file.

8. **"Your account has been deleted from %{host}" email title.** After an account is removed from the RubyGems.org website, an email with a title containing a placeholder is sent. Review the configuration of email templates and update them to correctly replace the host placeholder.

9. **The `config/initializers/cookie_rotator.rb` initializer may no longer be needed.** When upgrading to Rails 7.0, a cookie rotator initializer may be necessary. This file was added in February 2023 to the RubyGems codebase. If the Rails session cookie time to live (TTL) is less than that, then this file is no longer necessary.

10. **The `xml-simple` dependency appears unused.** Searching the codebase for `XmlSimple` yields no results, so we assume it is unused. The only evidence of its use we could find was #73. Further, many models implement a `to_xml` method, but it appears that this functionality is also unused. If both this dependency and functionality are indeed unused, consider removing them to reduce the attack surface and maintenance burden.

11. **The `dependencies` API is still documented**, but it appears that it is no longer available (figure C.11.1). Consider removing the documentation of this functionality.

```
$ curl https://rubygems.org/api/v1/dependencies?gems=rails
The dependency API has gone away. See
https://blog.rubygems.org/2023/02/22/dependency-api-deprecation.html for more
information
```

Figure C.11.1: Dependencies API deprecation and deletion

12. **The `/api/v1/versions/[GEM NAME]/latest.json` API endpoint will serve JSONP data (see figure C.12.1).** This is due to the `callback` parameter used in

figure C.12.2. JSONP has been supplanted by CORS, which provides security and robustness advantages over JSONP, such as better XSS resilience. Consider removing the `callback` parameter, and thus JSONP support, in favor of a CORS-based solution.

```
$ curl -s -I https://rubygems.org/api/v1/versions/rails/latest.json | grep
content-type:
content-type: application/json; charset=utf-8
$ curl -s -I https://rubygems.org/api/v1/versions/rails/latest.json?callback=test |
grep content-type:
content-type: text/javascript; charset=utf-8
```

Figure C.12.1: API endpoint serving JSONP data

```
render json: { "version" => number || "unknown" }, callback: params["callback"]
```

Figure C.12.2: The `callback` parameter enables JSONP responses (`rubygems.org/app/controllers/api/v1/versions_controller.rb#30`)

13. **The Docker image built by the `docker.yml` workflow could be signed.** The signature could be then verified by AWS EKS. This feature would prevent the EKS from loading malicious images. Interoperability of these signing and verification features was not determined during the audit.

14. **A password to Shipit's database is hard coded in the GitHub repository.** We did not determine the usefulness of the password during the audit. If the credentials are used in production, then rotate them, move to secure storage, and perform actions according to the relevant incident response plan.

D. Competitive Analysis

This appendix compares the state of the RubyGems security compared to other similar systems. Specifically, we were looking to answer the following questions:

- Are there any security-relevant features or functionality missing in the Ruby package manager?
- Does the Ruby package manager adhere to best practices observed in other package managers?

As a baseline for the comparison, we used OpenSSF's Principles for Package Repository Security document and Recent Package Repository Security Capabilities spreadsheet.

Authentication

Level 1	Email address verification	Yes
	Account recovery documentation	Partial: there is documentation for MFA recovery codes, but none for password recovery.
	Strong MFA available	Yes
	Email notifications for security events	Yes
	Authentication brute-force preventions (rate-limiting)	Yes (rate limits documentation)
Level 2	Detection of abandoned email domains	Yes
	Phishing-resistant MFA available	Yes
	MFA requirement for top (most popular) gems	Yes (MFA requirement announcement)
	Leaked credential databases are consulted	Yes (unpwn validator)
Level 3	Passwordless authentication is supported	Yes
	MFA requirement for all maintainers	No. However, increasing the MFA requirement is actively discussed.

	Phishing-resistant MFA requirement for top (most popular) gems	No

Authorization

Level 1	API keys scoped to gems are supported	Yes
	API keys have repository-specific prefix	Yes (`rubygems_` prefix)
Level 2	RBAC for gem owners	No
Level 3	OIDC for gems management is supported	Yes (Trusted Publishing)
	Integration with third-party code scanning services	Yes: • GitHub • GitLab (without autorevocation) • TruffleHog, GitGuardian, etc.
	Support for provenance (SigStore integration, SLSA compliance)	No

General capabilities

Level 1	Vulnerability disclosure policy is published	Yes (security documentation)
	Typosquatting mitigations	Yes (levenshtein distance PR)
Level 2	Packages versions cannot be replaced, deleted (yanked) packages cannot be reclaimed	Yes (packages can be reclaimed after an account is removed, but only new versions can be published)
	Users can report suspicious or malicious packages via UI, CLI, and API	No. Such packages are reported via email or GitHub issues.
	Automatic malware detection	No. However, some third-party services like Mend may be doing this task.

	Warnings about known security vulnerabilities in UI	No. The bundler-audit tool can be used to detect vulnerable dependencies, but warnings are not displayed in RubyGems' UI.
Level 3	Undergo periodic security reviews of package repository infrastructure	Yes (not necessarily periodic)
	Event transparency log is published	No. There is Mend, but it does not expose API for automation and large-scale queries.
	Advisories for malicious packages published in machine-readable format	Yes (`malicious-packages` repository)

CLI tooling

Level 1	Dependency pinning via versions, hashes, etc.	OK
Level 2	CLI warnings when installing packages with known vulnerabilities	Partial: the `bundler-audit` tool has this functionality, but gem CLI does not.
Level 3	SBOM generation	No
	Automatic upgrades for know vulnerable dependencies	No
	Advanced SAST for false-positives reduction when reporting known vulnerabilities	No

Other concerns

	Manifest confusion attack is not possible	Yes (but more testing is needed)
	Dependency confusion attack is mitigated	Yes ("RubyGems dependency confusion attack side of things" article)
	API keys can be restricted by source IP address	No
	API keys have default and suggested expiration	No

	times during generation	
	Security events log in user accounts	Yes (January 2024 changelog)
	CLI stores API keys in a secure location	No
	CLI is not vulnerable to common filesystem attacks (e.g., softlinks, umask)	Undetermined
	CLI helps to detect unmaintained packages	No
	CLI helps to review known security audits of dependencies	No
	Trusted Publishing UI is safe (e.g., all important fields are not optional)	Yes
	Package Index Integrity / RSTUF	Yes (RSTUF PR)
	Binary Transparency Log	No. However, this is only an experimental, work-in-progress feature.

Recommendations

Based on the results of the competitive analysis, we recommend taking the following actions to increase RubyGems' security. We have divided the recommendations into two sections: ones that are more important and should be focused on in the near future, and ones that are less urgent and can be planned in the long term.

High priority

- Continue to increase adoption of MFA among gem owners:
    - Continuously expand the MFA requirement to less popular gems.
    - Broaden the MFA requirement to not only popular gems, but also the runtime dependencies of those gems. At a minimum, the dependencies of gems' most recent versions should be covered. The requirement is not necessary for dependencies locked to specific versions (as the published versions are immutable), but recommended anyway.
    - Ideally, the MFA requirement should cover all gem owners.

- Start enforcing phishing-resistant MFA requirements for maintainers of the most popular packages. That would mean an enforcement of security devices instead of OTP tokens. When this requirement is implemented, the maintainers should be strongly advised to register at least two security devices (one for a backup).
- Add support for provenances. This recommendation has two sub-tasks: support for attestations in RubyGems API, and automatic verification by the registry (on import to repo) and by CLI tools (on install). This recommendation would enable deprecation of the gem signing feature. An example of in-progress implementation is PEP 740.
- Add support for more vendors to the Trusted Publishing feature. Selection of specific vendors should be consulted with the community.
- Add documentation in the RubyGems.org website for reporting malware and suspicious gems. Currently, only documentation for reporting vulnerabilities in gems and RubyGems.org exists. If the malware reports should be sent as GitHub issues, then create a dedicated template for that. Otherwise, create a dedicated web page on the website (with HTML form) or point to the email address and instruct users what the email message with the report should contain. Create a "report as malware" button in the UI. In the future, consider implementing the future in CLI and API. For an example implementation, see PyPI's security page and its "report as malware" button and NPM documentation; for ideas, see the announcement of PyPI Malware Reporting and Response project.
- Improve UX around API key expiration times.
    - When generating the keys in UI, the expiration time should be set to some default value like 30 days.
    - When automatically generating API keys in CLI for sign in, the keys should be set with some default expiration time.
    - Users could be presented with some common expiration options in the UI (like 7 days, 30 days, 90 days, infinite), in addition to the calendar.
- Review the security of the CLI against common filesystem-related attacks. At least the `gem install` command should not follow symlinks when extracting repositories, and should respect the local umask settings.

Low–medium priority

- Add documentation for password recovery. This recovery process is usually straightforward, consisting of an email with a password change link. However, the doc may cover more complex cases:

    - What is needed for password recovery when MFA is enabled?
    - What should a user do if they lose access to their email?
    - What should the user do if their email inbox is compromised?
    - Can the Ruby team help in these cases and if so how to contact it?
- Implement role-based access controls (RBAC) for maintainer accounts. This could consist of two roles: owners (allowed to manage other accounts/roles and gems) and maintainers (allowed to manage gems). Alternatively, roles could correspond to granular privileges of API keys (owners with all privileges who can give a subset of privileges to other owners).
- Integrate with GitLab's Secret Detection to automatically revoke leaked API tokens.
- Add warnings in the UI for dependencies with known vulnerabilities. At a minimum, owners of a gem should be shown a warning, but ideally that information should be visible to all visitors. See crates.io issue #6397 for ideas.
- Publish an event transparency log (like `replicate.npmjs.com`, which is an instance of CouchDB). The log will help users to find and investigate security issues at scale.
- Make the `gem install` command (and other relevant commands) warn users about known vulnerabilities.
- Improve the gems CLI to reach level 3 of the Principles for Package Repository Security:
    - Add an SBOM generation command (like the `npm-sbom`).
    - Add a feature to `gem` or `bundler` for automatic updates of vulnerable dependencies.
    - Reduce the false positives rate in `bundler-audit` by leveraging static analysis (like `govulncheck`).
- Improve the gems CLI with additional security features:
    - Add a command for checking security audits done for dependencies, similar to `cargo-vet` and `cargo-crev`.
    - Add a command for listing all implicit trust relationships in dependencies, similar to `cargo-supply-chain`.

    - Add a command for checking if a package is unmaintained, similar to `cargo-audit` and `cargo-unmaintained`. This feature may require creating new types of entries in `ruby-advisory-db`.

- Add a new restriction to API keys: allowlisted source IP addresses. Users should be able to set the allowed IP address range when generating tokens in the UI/CLI/API. Consider automatically setting the IP allowlist when the gems CLI generates new tokens for user authentication (`gem signin` command); note that this would require the CLI to detect the user's public IP.

- Consider storing API keys generated by the CLI in a secure location instead of a file. That location could be a keychain, keyring, keystore, or credential manager. This would prevent API key leaks in case of a stolen device (with full-disk encryption not enabled) and mitigate attacks by malware.

E. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings. For more information, see the Trail of Bits Testing Handbook section on static analysis.

For any of the tools below that support it, we used SARIF Explorer—a VSCode extension that enhances the triaging process—to review the results.

E.1 Semgrep

We installed Semgrep by following Semgrep's installation guide. Then, we installed the Semgrep Pro Engine, which includes cross-file (interfile) and cross-function (interprocedural) analysis. To run Semgrep with the Pro Engine, we used the following commands:

```
semgrep login
semgrep install-semgrep-pro
```

To run Semgrep on the codebase, we ran the following command in the root directory of all provided source code directories:

```
semgrep --pro --config=auto --sarif > semgrep.sarif
```

To run Semgrep to identify potential secrets in the codebase, we used the following command:

```
semgrep -c p/secrets
```

To run Semgrep with every rule—leading to more findings, many of which are false positives—the following command can be used:

```
semgrep --pro --config=r/all --sarif > semgrep.sarif
```

The following public Semgrep rulesets were used during this audit:

- `/p/ruby`
- `/p/secrets`
- `/p/dockerfile`
- `/p/trailofbits`
- `/p/terraform`

E.2 Semgrep Rules

```yaml
rules:
  - id: rails-cookie-attributes
    message: |
      Found Rails cookie set with insecure attribute.
    languages: [ruby]
    severity: WARNING
    metadata:
      category: security
      cwe: "CWE-345: Insufficient Verification of Data Authenticity"
      subcategory: [audit]
      confidence: HIGH
      likelihood: HIGH
      impact: LOW
      references:
        - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie
        - https://api.rubyonrails.org/classes/ActionDispatch/Session/CookieStore.html
    patterns:
      - pattern-either:
          - pattern-inside: cookies[$ANY] = ...
          - pattern-inside: cookies. ... .$ATTR[$ANY] = ...
      - pattern-either:
          - pattern: "{..., same_site: :none, ...}"
          - pattern: "{..., same_site: :lax, ...}"
          - pattern: "{..., httponly: false, ...}"
          - pattern: "{..., secure: false, ...}"
```

Figure E.2.1: Semgrep rule for insecure Rails cookie attributes

```yaml
rules:
  - id: rails-cache-store-marshal
    message: |
      Found Rails cache store configured to allow Marshaling. As of Rails 7.1
      the default serializer is `:json_allow_marshal`. If an attacker can inject
      data into the cache store (SSRF, etc.), then they can achieve code
      execution when the object is later deserialized.
    languages: [ruby]
    severity: WARNING
    metadata:
      category: security
      cwe: "CWE-502: Deserialization of Untrusted Data"
      subcategory: [audit]
      confidence: MEDIUM
      likelihood: LOW
      impact: HIGH
      references:
        -
https://guides.rubyonrails.org/caching_with_rails.html#activesupport-cache-memcachestore
        -
https://guides.rubyonrails.org/configuring.html#config-active-support-message-serializer
        - https://api.rubyonrails.org/v7.1.3.4/classes/ActiveSupport/Cache/MemCacheStore.html
        - https://api.rubyonrails.org/v7.1.3.4/classes/ActiveSupport/Cache/Store.html
    patterns:
      - pattern-inside: |
          Rails.application.configure do
          ...
          end
```

```
    - pattern-either:
        - patterns:
            # These patterns must use two different metavariable names.
            # In Ruby, $STORE will match the entire line here, so these two
            # patterns effectively cancel each other out if they share a
            # metavariable name. To avoid that, use separate names.
            - pattern: "config.cache_store = $STORE"
            - pattern-not: "config.cache_store = $STORE2, ..., { ... }"
        - patterns:
            - pattern: "config.cache_store = $STORE, ..., $OPTIONS"
            - metavariable-pattern:
                metavariable: $OPTIONS
                patterns:
                  - pattern: "{ ... }"
                    - pattern-not: "{ ..., serializer: :json, ... }"
                    - pattern-not: "{ ..., serializer: :message_pack, ... }"
    - metavariable-pattern:
        metavariable: $STORE
        pattern-either:
          - pattern: ":file_store"
          - pattern: ":mem_cache_store"
          - pattern: ":redis_cache_store"
```

Figure E.2.2: Semgrep rule for Rails Marshal cache serialization

```
rules:
  - id: action-dispatch-insecure-ssl
    message: |
      Found Rails application with insecure SSL setting.
    languages: [ruby]
    severity: WARNING
    metadata:
      category: security
      cwe: "CWE-295: Improper Certificate Validation"
      subcategory: [audit]
      confidence: HIGH
      likelihood: HIGH
      impact: HIGH
      references:
        - https://api.rubyonrails.org/v7.1.3.4/classes/ActionDispatch/SSL.html
    patterns:
      - pattern-inside: |
          Rails.application.configure do
            ...
          end
      - pattern-either:
          - pattern: "config.force_ssl = false"
          - pattern: "config.ssl_options = { ..., secure_cookies: false, ... }"
          - pattern: "config.ssl_options = { ..., hsts: false, ... }"
          - pattern: "config.ssl_options = { ..., hsts: { ..., subdomains: false, ... }, ... }"
```

Figure E.2.3: Semgrep rule for insecure `ActionDispatch ssl_options`

```
rules:
  - id: action-mailer-insecure-tls
    message: |
      Found ActionMailer SMTP configuration with insecure TLS setting. These
```

```
      settings do not require a successful, encrypted, verified TLS connection
      is made. Set `enable_starttls: true` and `openssl_verify_mode` to verify
      peer
    languages: [ruby]
    severity: WARNING
    metadata:
      category: security
      cwe: "CWE-295: Improper Certificate Validation"
      subcategory: [audit]
      confidence: HIGH
      likelihood: HIGH
      impact: HIGH
      references:
        - https://guides.rubyonrails.org/action_mailer_basics.html#action-mailer-configuration
    pattern-either:
      - pattern: "ActionMailer::Base.smtp_settings = { ..., openssl_verify_mode:
OpenSSL::SSL::VERIFY_NONE, ... }"
      - pattern: "ActionMailer::Base.smtp_settings = { ..., openssl_verify_mode: 'none', ... }"
      - pattern: "ActionMailer::Base.smtp_settings = { ..., enable_starttls_auto: true, ... }"
```

Figure E.2.4: Semgrep rule for insecure Rails SMTP TLS configuration

F. Fuzz Testing

This appendix describes the setup of the fuzz testing tools used during this audit.

We recommend fuzz testing code that handles untrusted input, such as code that performs parsing, decoding, deserializing, or other types of heavy processing of input. Fuzzing often finds memory corruption bugs or other unintended crashes. For more information, see the Trail of Bits Testing Handbook section on fuzzing.

F.1 Ruzzy

We installed Ruzzy by following Ruzzy's installation guide. We identified the Ruby `cbor` library as a good candidate for fuzzing due to its deserialization of binary data and its C extension component. The `cbor` library is used by the WebAuthn functionality within RubyGems. We used the following fuzzing harness to fuzz `cbor`:

```
# frozen_string_literal: true

require 'cbor'
require 'ruzzy'

test_one_input = lambda do |data|
  begin
    CBOR.decode(data)
  rescue Exception
    # We're looking for memory corruption, not Ruby exceptions
  end
  return 0
end

Ruzzy.fuzz(test_one_input)
```

Figure F.1: The `cbor` fuzzing harness

We used this harness and Ruzzy's standard procedure for fuzzing Ruby C extensions. We fuzzed the target for several hours, which yielded no findings. Consider applying this process to additional input processing targets used by the application.

G. Dynamic Analysis Using Burp Suite Professional

This appendix describes the setup of Burp Suite Professional, the dynamic analysis tool used during this audit.

We dynamically scanned the RubyGems web interface with Burp Suite Professional, which allowed us to identify several issues.

Strategy

- **Scanning using a live task:** We interacted with the web interface in RubyGems and had underlying requests processed by a live task and sent to the active scanner with the maximum coverage set and extensions enabled.
- **Active Burp scanner:** When interacting with the RubyGems web interface through Burp Repeater, we sent underlying requests to the active scanner with the maximum coverage setting.
- **Manual interaction in Burp Repeater:** Analyzing the particular functionality of the targeted application, we sent a given request to Burp Repeater, changed the request, and observed the potentially malicious output.

Extensions

During the assessment, we also used the following Burp Suite Professional extensions to enhance the functionality of our auditing process:

- Turbo Intruder allows users to send large numbers of customized HTTP requests. It is intended to supplement Burp Intruder by handling attacks that require extreme speed or complexity. We used this extension to check whether the API is vulnerable to race conditions.
- Active Scan++ enhances the default active and passive scanning capabilities of Burp Suite Professional. It adds checks for vulnerabilities that the default scanner might miss (e.g., blind code injection via Ruby's `open` function and suspicious transformations, such as `7*7 => '49'`).
- Backslash Powered Scanner extends the active scanner capabilities by trying to identify known and unknown classes of server-side injection vulnerabilities.