Semgrep

XSS Prevention for Ruby on Rails

Semgrep ruleset for this cheatsheet: https://semgrep.dev/p/minusworld.ruby-on-rails-xss

This is a cross-site scripting (XSS) prevention cheat sheet by r2c. It contains code patterns of potential XSS in an application. Instead of scrutinizing code for exploitable vulnerabilities, the recommendations in this cheat sheet pave a safe road for developers that mitigates the possibility of XSS in your code. By following these recommendations, you can be reasonably sure your code is free of XSS.

Exploitation Conditions:

User input +

1. Server code: Unescaped variable enters template engine in Python code
2. Server code: Bypassing the template engine
3. Templates: Variable explicitly unescaped
4. Templates: Variable in dangerous location

Check your project for these conditions:

```
$ semgrep --config https://semgrep.dev/p/minusworld.ruby-on-rails-xss
```

---

1. Server code: Unescaped variable enters template engine in Python code

1.A. Using `html_safe()`

`html_safe()` marks the supplied string as "safe for HTML rendering." This bypasses HTML escaping and potentially creates XSS vulnerabilities.

Recommendation: If needed, review each usage and exempt with `# nosem`.

Code example:

```
html = "<div>#{name}</div>".html_safe
```

References:

- Brakeman scanner - Cross-site scripting
- Preventing XSS in Ruby on Rails

1.B. Using `content_tag()`

`content_tag()`'s escaping behavior has changed between Rails 2 and 3. In Rails 2, no supplied content is escaped. In Rails 2 and 3, attribute names are not escaped. Further, the returned value is marked as "safe," the same as if `html_safe()` had been used. This confusing behavior makes it difficult to use `content_tag()` properly; improper use can create XSS vulnerabilities in your application.

Code example:

```
content_tag :p, "Hello, #{name}"
```

References:

- Brakeman scanner - Content tag

Recommendation: If necessary, prefer `html_safe()` due to its straightforward behavior.

1.C. Using `raw()`

`raw()` disables HTML escaping for the returned content. This permits raw HTML to be rendered in a template, which could create a XSS vulnerability.

Recommendation: Prefer `html_safe()` if necessary.

Code example:

```
raw @user.name
```

References:

- `raw()` documentation
- Preventing XSS in Ruby on Rails

1.D. Disabling of `ActiveSupport#escape_html_entities_in_json`

`ActiveSupport#escape_html_entities_in_json` is a setting which determines whether `Hash#to_json()` will escape HTML characters. Disabling this could create XSS vulnerabilities.

Recommendation: If HTML is needed in JSON, use `JSON.generate()` and review each usage carefully. Exempt each case with `# nosem`.

Code example:

```
config.active_support.escape_html_entities_in
```

References:

- `escape_html_entities_in_json` documentation
- Brakeman scanner - Cross-site scripting (JSON)
- How to disable HTML escaping for JSON, but keep enabled for views?

---

2. Server code: Bypassing the template engine

2.A. Manually creating an ERB template

Manually creating an ERB template could create a server-side template injection (SSTI) vulnerability if it is created with user input. (This could also result in XSS.) Due to the severity of this type of vulnerability, it is better to use a template file instead of creating templates in code.

Recommendation: Use ERB template files

Code example:

```
ERB.new("<div>#{@user.name}</div>").result
```

References:

- Brakeman scanner - Template injection
- Ruby ERB template injection

2.B. Rendering an inline template with `render inline:`

`render inline:` is the same as creating a template manually and is therefore susceptible to the same vulnerabilities as manually creating an ERB template. This can result in a SSTI or XSS vulnerability.

Code example:

```
render inline: "<div>#{@user.name}</div>"
```

References:

- Zen Rails Security Checklist

Recommendation: Use ERB template files

- Inline renders - even worse than XSS!

2.C. Using `render text:`

`render text:` unintuitively sets the Content-Type to text/html. This means anything rendered through `render text:` will be interpreted as HTML. Templates rendered in this manner could create a XSS vulnerability.

Recommendation: Use ERB template files

Code example:

```
render text: "<div>#{@user.name}</div>"
```

References:

- Inline renders - even worse than XSS!

---

3. Templates: Variable explicitly unescaped

3.A. Using `html_safe()`

`html_safe()` marks the supplied string as "safe for HTML rendering." This bypasses HTML escaping and potentially creates XSS vulnerabilities.

Recommendation: Prefer using `html_safe()` in Ruby code instead of templates.

Code example:

```
<%= name.html_safe %>
```

References:

- Brakeman scanner - Cross-site scripting
- Preventing XSS in Ruby on Rails

3.B. Using `content_tag()`

`content_tag()`'s escaping behavior has changed between Rails 2 and 3. In Rails 2, no supplied content is escaped. In Rails 2 and 3, attribute names are not escaped. Further, the returned value is marked as "safe," the same as if `html_safe()` had been used. This confusing behavior makes it difficult to use `content_tag()` properly; improper use can create XSS vulnerabilities in your application.

Recommendation: If necessary, prefer `html_safe()` in Ruby code due to its straightforward behavior.

Code example:

```
<%= content_tag :p, "Hello, #{name}" %>
```

References:

- Brakeman scanner - Content tag

3.C. Using `raw()`

`raw()` disables HTML escaping for the returned content. This permits raw HTML to be rendered in a template, which could create a XSS vulnerability.

Code example:

```
<%= raw @user.name =>
```

Recommendation: Prefer `html_safe()` in Ruby code if necessary.

References:

- `raw()` documentation
- Preventing XSS in Ruby on Rails

3.D. Using `<%== ... %>`, which is an alias for `html_safe()`

The double-equals `==` is an ERB alias for `html_safe()`. This will mark the contents as "safe for rendering" and may introduce an XSS vulnerability.

Recommendation: Prefer `html_safe()` in Ruby code if necessary.

Code example:

```
<%== @user.name %>
```

References:

- Alias for `html_safe()`
- Raw vs. html_safe

---

4. Templates: Variable in dangerous location

4.A. Unquoted variable in HTML attribute

Unquoted template variables rendered into HTML attributes is a potential XSS vector because an attacker could inject JavaScript handlers which do not require HTML characters. An example handler might look like: `onmouseover=alert(1)`. HTML escaping will not mitigate this. The variable must be quoted to avoid this.

Recommendation: Always use quotes around HTML attributes.

Code example:

```
<div class=<%= classes %></div>
```

References:

- Flask cross-site scripting considerations - unquoted variable in HTML attribute

4.B. Variable in `href` attribute

Template variables in a `href` value could still accept the `javascript:` URI. This could be a XSS vulnerability. HTML escaping will not prevent this. Use `link_to` beginning with a literal forward slash to generate links.

Recommendation: Use `url_for` to generate links.

Code example:

```
<a href="<%= link %>"></a>
```

References:

- Flask cross-site scripting considerations - variable in `href`

4.C. Using `link_to` with unrestricted URL scheme

Detected a template variable used in 'link_to'. This will generate dynamic data in the 'href' attribute. This allows a malicious actor to input the 'javascript:' URI and is subject to cross- site scripting (XSS) attacks. If using a relative

Code example:

```
<%= link_to "Here", @link %>
```

References:

URL, start with a literal forward slash and concatenate the URL, like this: `<%= link_to "Here", "/"+@link %>`. You may also consider setting the Content Security Policy (CSP) header.

Recommendation: If you must use this, add a literal forward-slash at the beginning to create a relative url.

- OWASP Cheatsheet - Ruby on Rails XSS
- Brakeman scanner - link_to

4.D. Variable in `<script>` block

Template variables placed directly into JavaScript or similar are now directly in a code execution context. Normal HTML escaping will not prevent the possibility of code injection because code can be written without HTML characters. This creates the potential for XSS vulnerabilities, or worse.

Recommendation: If necessary, use the the `escape_javascript` function or its alias, `j`. Review each usage carefully and exempt with `# nosem`.

Code example:

```
<script>var name = <%= name %>;</script>
```

References:

- Template engines: Why default encoders are not enough
- Protecting against XSS in Rails - JavaScript contexts
- `escape_javascript` documentation

Mitigations

Item	Name	Semgrep rule	Recommendation
1.A.	Ban `html_safe()`	ruby.rails.security.audit.xss.avoid-html-safe.avoid-html-safe	If needed, review each usage and exempt with `# nosem`.
1.B.	Ban `content_tag()`	ruby.rails.security.audit.xss.avoid-content-tag.avoid-content-tag	If necessary, prefer `html_safe()` due to its straightforward behavior.
1.C.	Ban `raw()`	ruby.rails.security.audit.xss.avoid-raw.avoid-raw	Prefer `html_safe()` if necessary.
1.D.	Ban disabling of `ActiveSupport#escape_html_entities_in_json`	ruby.lang.security.json-entity-escape.json-entity-escape	If HTML is needed in JSON, use `JSON.generate()` and review each usage carefully. Exempt each case with `# nosem`.
2.A.	Ban template creation in code	ruby.rails.security.audit.xss.manual-template-creation.manual-template-creation	Use ERB template files
2.B.	Ban `render inline:`	ruby.rails.security.audit.xss.avoid-render-inline.avoid-render-inline	Use ERB template files
2.C.	Ban `render text:`	ruby.rails.security.audit.xss.avoid-render-text.avoid-render-text	Use ERB template files
3.A.	Ban `html_safe()`	ruby.rails.security.audit.xss.templates.avoid-html-safe.avoid-html-safe	Prefer using `html_safe()` in Ruby code instead of templates.
3.B.	Ban `content_tag()`	ruby.rails.security.audit.xss.templates.avoid-content-tag.avoid-content-tag	If necessary, prefer `html_safe()` in Ruby code due to its straightforward behavior.
3.C.	Ban `raw()`	ruby.rails.security.audit.xss.templates.avoid-raw.avoid-raw	Prefer `html_safe()` in Ruby code if necessary.
3.D.	Ban `<%== ... %>`, which is an alias for `html_safe()`	ruby.rails.security.audit.xss.templates.alias-for-html-safe.alias-for-html-safe	Prefer `html_safe()` in Ruby code if necessary.
4.A.	Flag unquoted HTML attributes ERB expressions	ruby.rails.security.audit.xss.templates.unquoted-attribute.unquoted-attribute	Always use quotes around HTML attributes.
4.B.	Flag template variables in `href` attributes	ruby.rails.security.audit.xss.templates.var-in-href.var-in-href	Use `url_for` to generate links.
4.C.	Flag `link_to` in templates	ruby.rails.security.audit.xss.templates.dangerous-link-to.dangerous-link-to	If you must use this, add a literal forward-slash at the beginning to create a relative url.

Item	Name	Semgrep rule	Recommendation
4.D.	Ban template variables in `<script>` blocks.	ruby.rails.security.audit.xss.templates.var-in-script-tag.var-in-script-tag	If necessary, use the the `escape_javascript` function or its alias, `j`. Review each usage carefully and exempt with `# nosem`.