Secure Code Guide: Example Organization

Introduction

This guide provides practical, organization-specific standards and best practices to help developers write secure, maintainable software. It focuses on topics such as authorization enforcement, data sensitivity classification, logging practices, and secure coding principles for a Ruby on Rails monolith.

Use the following ways to contact the security team:

Slack - #acme-security-team

Email - securit.slay@hotmail.com

Fax - lol, just kidding

Secure Code Guide located at: https://acme.co.notreal/secure-code-guide

1. Authorization in REST APIs

Guidelines

1. **Centralize Authorization Logic**:
    - Use `Pundit` or `CanCanCan` gems to define and enforce authorization policies in a centralized, consistent manner.
    - Example: Define a `Policy` for each resource and ensure controllers check it:

```
Unset
class ProjectPolicy < ApplicationPolicy
  def update?
    user.admin? || record.owner == user
  end
end
```

2. **Avoid Relying Solely on Client-Side Controls**:
   - Do not depend on front-end logic to enforce authorization (e.g., hiding buttons or disabling forms). Always validate permissions server-side.
3. **Use Declarative Filters in Controllers**:
   - Use `before_action` callbacks to ensure authorization checks are applied consistently:

```
Unset
before_action :authorize_project, only: [:update, :destroy]

def authorize_project
  authorize @project
end
```

4. **Deny by Default**:
   - Ensure your default policy is to deny access unless explicitly allowed.
5. **Enforce Scoping**:
   - Use `scope` to restrict data based on user roles.

```
Unset
class ProjectPolicy < ApplicationPolicy
  class Scope < Scope
    def resolve
      user.admin? ? scope.all : scope.where(owner: user)
    end
  end
end
```

---

2. Data Sensitivity Classification

Classification Levels

1. **Public**:
    - Non-sensitive information that can be shared freely (e.g., public blog posts, marketing material).
2. **Internal**:
    - Data intended for internal use but not critical (e.g., non-public project descriptions).
3. **Confidential**:
    - Sensitive data requiring strong protections (e.g., user PII, passwords, API tokens).
4. **Restricted**:
    - Highly sensitive data with limited access (e.g., financial records, proprietary algorithms).

Implementation

- **Label Data Fields**:
    - Annotate models to indicate classification levels:

```
Unset
class User < ApplicationRecord
  # @classification: confidential
  attr_encrypted :ssn, key: ENV["SSN_ENCRYPTION_KEY"]
end
```

- **Apply Role-Based Access**:
    - Ensure restricted data is only accessible by authorized roles.

---

3. Logging Best Practices

Guidelines

1. **Do Not Log Sensitive Data**:
    - Mask or omit sensitive fields like passwords, credit card numbers, and SSNs.

```
Unset
Rails.logger.info("User #{user.id} logged in") # Do not log PII
```

2. **Use Structured Logging**:
   - Use JSON or another structured format for logs to improve readability and parsing:

```
Unset
logger.info({ event: "user_login", user_id: user.id, timestamp:
Time.now }.to_json)
```

3. **Log Security Events**:
   - Include events like failed login attempts, authorization failures, and role changes.
4. **Rotate and Protect Logs**:
   - Configure log rotation with `logrotate` or similar tools to prevent unbounded log growth.
   - Set strict file permissions for log files:

```
Unset
chmod 640 /path/to/logs/*
```

5. **Avoid Debug Logging in Production**:
   - Do not enable verbose or debug-level logging in production environments.

---

4. Preferred Best Practices

Secure Development

- **Input Validation**:

    - Use strong parameter filtering in controllers:

```
Unset
params.require(:user).permit(:name, :email, :role)
```

- **Output Encoding**:
    - Use Rails helpers like `html_escape` to prevent XSS:

```
Unset
<%= html_escape(user.name) %>
```

Dependency Management

- Use `bundler-audit` to detect vulnerable dependencies:

```
Unset
bundle exec bundler-audit
```

- Regularly update gems and prioritize patching high-severity vulnerabilities.

Environment Configuration

- Store secrets and credentials securely in environment variables or secret managers:

```
Unset
export DATABASE_PASSWORD=super_secure_password
```

- Use Rails' encrypted credentials (`config/credentials.yml.enc`) to manage sensitive configurations.

---

5. Code Review Checklist

Authorization

- Have all routes been reviewed for proper authorization checks?
- Are `before_action` callbacks implemented consistently?

Data Handling

- Are sensitive fields encrypted at rest (e.g., PII)?
- Is sensitive data excluded from logs?

Error Handling

- Are error messages generic to avoid leaking system details?

Dependency Risks

- Have dependency vulnerabilities been checked and addressed?

---

6. Secure CI/CD Practices

Static Analysis

- Run `brakeman` for Rails security analysis:

```
Unset
brakeman -A
```

Secrets Scanning

- Use tools like `trufflehog` or `git-secrets` to prevent committing secrets.

Automated Tests

- Ensure security tests are part of your CI pipeline:
  - Authorization tests.
  - Input validation tests.
  - Business logic tests.

This guide can be used to simulate a realistic organization's secure coding policies and practices. Let me know if you'd like to expand any sections or add more scenarios!