GitHub

Keeping Secrets: Lessons Learned From Securing GitHub

Dennis Pacewicz (@*lyninx*)
Senior Product Security Engineer
GitHub

Wei Lin Ngo (@*Creastery*)
Staff Security Engineer
Praetorian

RubyKaigi 2025

whoami

> Dennis Pacewicz (@lyninx)

```
from:
  "Toronto, Canada 🇨🇦"
company:
  "GitHub"
  "(formerly) Shopify"
team:
  "Product Security Engineering"
```

whoami

> Wei Lin Ngo (@Creastery)

```
from:
  "Singapore 🇸🇬"
company:
  "Praetorian"
  "(formerly) STAR Labs"
team:
  "Product/Application Security"
```

Agenda

1. ***(Ethically)*** **Hacking GitHub**

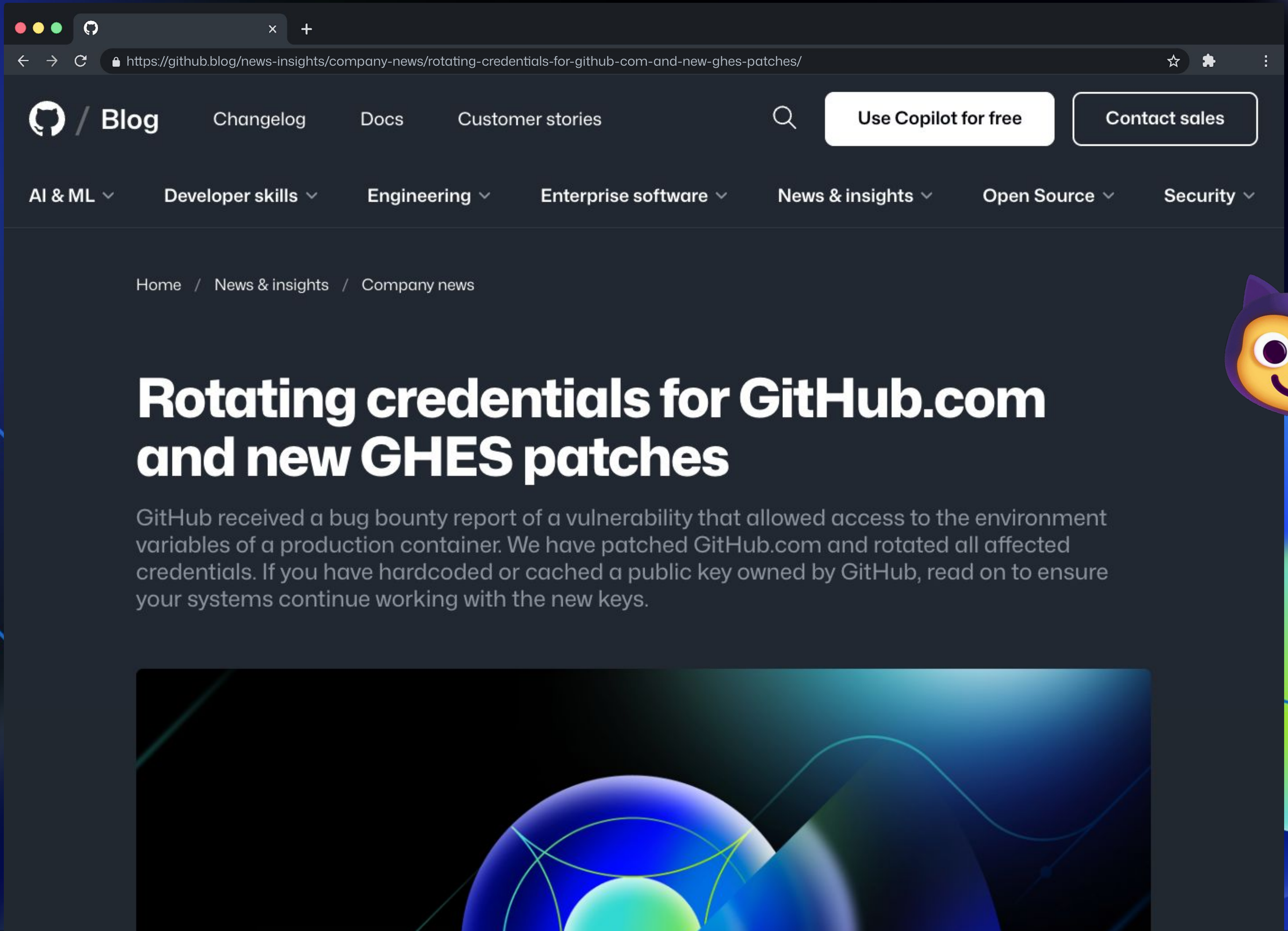
   Walkthrough of a high-impact vulnerability leading to secrets exposure

2. **Protecting Ourselves**

   Processes and tools to improve security posture and prevent regressions

3. **Keeping Secrets**

   How to handle and secure sensitive values in production Ruby apps



Rotating credentials for GitHub.com and new GHES patches

GitHub received a bug bounty report of a vulnerability that allowed access to the environment variables of a production container. We have patched GitHub.com and rotated all affected credentials. If you have hardcoded or cached a public key owned by GitHub, read on to ensure your systems continue working with the new keys.

GitHub App Structure

- Huge monolith
- Built on Rails
  - Model-View-Controller (MVC) Architecture
- Views utilize the ViewComponent framework
  - Build component-driven UI
  - Render Ruby objects into markup

Vulnerability Discovery

```ruby
class Organizations::Settings::RepositoryItemsComponent < ApplicationComponent
  def initialize(..., repository_identifier_key: :global_relay_id, ...)
    ...
    @repository_identifier_key = repository_identifier_key
    ...
  end
  ...
  def identifier_for(repository)
    repository.send(@repository_identifier_key)
  end
  ...
end
```

(Ethically) Hacking GitHub

```ruby
class Orgs::ActionsSettings::RepositoryItemsController < Orgs::Controller
  def index
    respond_to do |format|
      format.html do
        render(Organizations::Settings::RepositoryItemsComponent.new(
          ...
          repository_identifier_key: repository_identifier_key,
          ...
        ), layout: false)
      end
    end
  end

  def repository_identifier_key
    return :global_relay_id unless rid_key.present?
    rid_key
  end

  def rid_key
    params[:rid_key]
  end
end
```

What is send()?

```ruby
class HelloWorld
  def print(*args)
    puts('Hello ' + args.join(' '))
  end
end

obj = HelloWorld.new()
obj.print('world')          # => 'Hello world'
obj.send('print', 'world') # => 'Hello world'
```

Exploiting send(😈, 😈)

```ruby
user_input1 = 'eval'
user_input2 = 'arbitrary Ruby code here'

obj.send(user_input1, user_input2)
# Example:
obj.send('eval', 'system("ls")')
```

Exploiting send(😈, ... , 😈)

```ruby
obj.send('send', 'send', 'send', 'send', 'eval', '1+1')
# will call:
obj.send('send', 'send', 'send', 'eval', '1+1')
# ...
obj.send('eval', '1+1')
# and eventually call:
eval('1+1')
```

Thinking Like A Hacker

Step 1: Identify potential vulnerabilities

Step 2: Determine exploitability

- Any safeguards present?
- Are safeguards bypassable?
- Any exploitation constraints?

Step 3: Assess security impact (later on)

```
repository.send(@repository_identifier_key)
```

"Zero-argument" arbitrary method dispatch

Thinking Like A Hacker

Exploitation Constraint: *"Zero-argument"* arbitrary method dispatch

What can we do?

- Call any methods defined in the class or those inherited from superclasses
- Call "zero-argument" methods with arity of 0 or -1:

```
def zero_args()
def positional_arg_with_default_value(arg = 'default')
def keyword_arg_with_default_value(keyword: 'default')
def splat_args(*args)
```

Finding Candidate Methods

Disclose File Names: __dir__(), caller()

Disclose Class Name: class()

Disclose Method Names: __callee__(), __method__(), methods(), etc.

(by the way, these are built-in methods for most Ruby objects)

Exploiting repository.send(😈)

Strategy:

- Drop into Rails console and gather callable methods via the send()
- Found ~3.6K possibly callable methods
- Tried invoking all methods and collected the response for analysis
- Identified two methods that disclosed 1K+ environment variables

Root Cause Analysis

```ruby
module Repository::GitDependency
  ...
  def nw_fsck(trust_synced: false)
    rpc.nw_fsck(trust_synced: trust_synced)
  end
  ...
end
```

```ruby
module GitRPC
  class Backend
    ...
    rpc_writer :nw_fsck, output_varies: true
    def nw_fsck(trust_synced: false)
      argv = []
      argv << "--connectivity-only"
      argv << "--trust-synced" if trust_synced
      spawn_git("nw-fsck", argv)
    end
    ...
  end
end
```

Root Cause Analysis

```ruby
module GitRPC
  ...
  class Native
    ...
    def spawn(argv, input = nil, env = {}, options = {})
      ...
      {
        ...
        :out       => process.out,
        :err       => process.err,
        :argv      => argv,
        :env       => env,
        :path      => @path,
        :options   => options,
        :truncated => truncated,
      }
    end
    ...
  end
end
```

Copy of ENV returned here

Escalating Impact Further

- _gh_render cookie
  - Defaults to using Marshal for serializing session data
  - Uses ENTERPRISE_SESSION_SECRET in ENV for encryption/signing
- Encrypt the marshalled payload
  - Attacker gets remote code execution in GitHub Enterprise Servers

Agenda

1. *(Ethically)* Hacking GitHub

   Walking through a high-impact vulnerability leading to secrets exposure

2. **Protecting Ourselves**

   Processes and tools to improve Ruby code security and prevent regressions

3. Keeping Secrets

   Handling and securing sensitive values in production Ruby apps

Protecting Ourselves

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

- Bug Bounty program
- Code scanning alerts
- Red team / Engineering teams
- Customer reports
- *and more!*

Protecting Ourselves

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

Protecting Ourselves

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

1. Containment / Eradication
2. Mitigation / Remediation

Protecting Ourselves

```ruby
class Organizations::Settings::RepositoryItemsComponent < ApplicationComponent
  def initialize(..., repository_identifier_key: :global_relay_id, ...)
    ...
    @repository_identifier_key = repository_identifier_key
    ...
  end

  def identifier_for(repository)
    repository.send(@repository_identifier_key)
  end






  ...
end
```

Protecting Ourselves

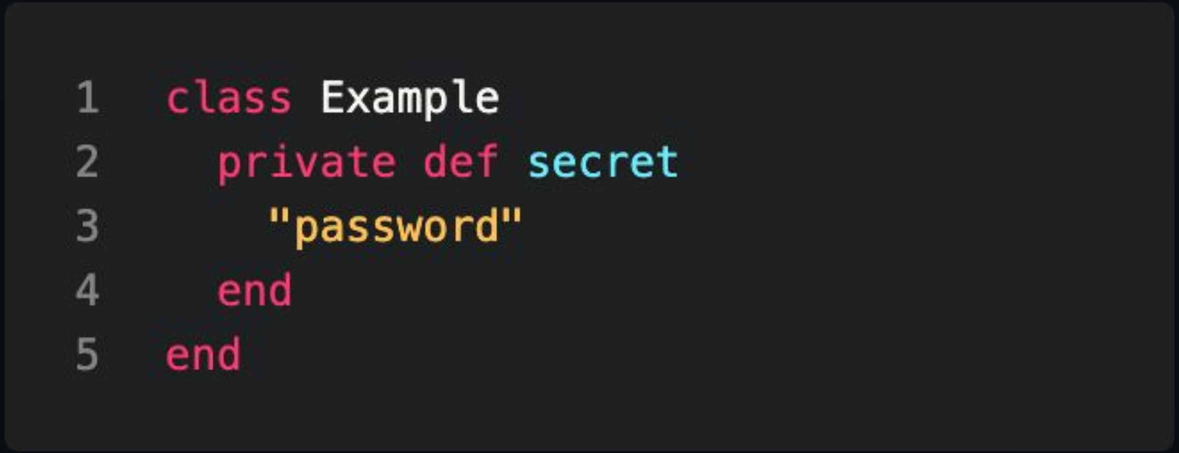
```ruby
class Organizations::Settings::RepositoryItemsComponent < ApplicationComponent
  def initialize(..., repository_identifier_key: :global_relay_id, ...)
    ...
    @repository_identifier_key = repository_identifier_key
    ...
  end

  def identifier_for(repository)
    case @repository_identifier_key
    when :id, "id"
      repository.id
    else
      repository.global_relay_id
    end
  end
  ...
end
```

No more `repository.send()`

Protecting Ourselves

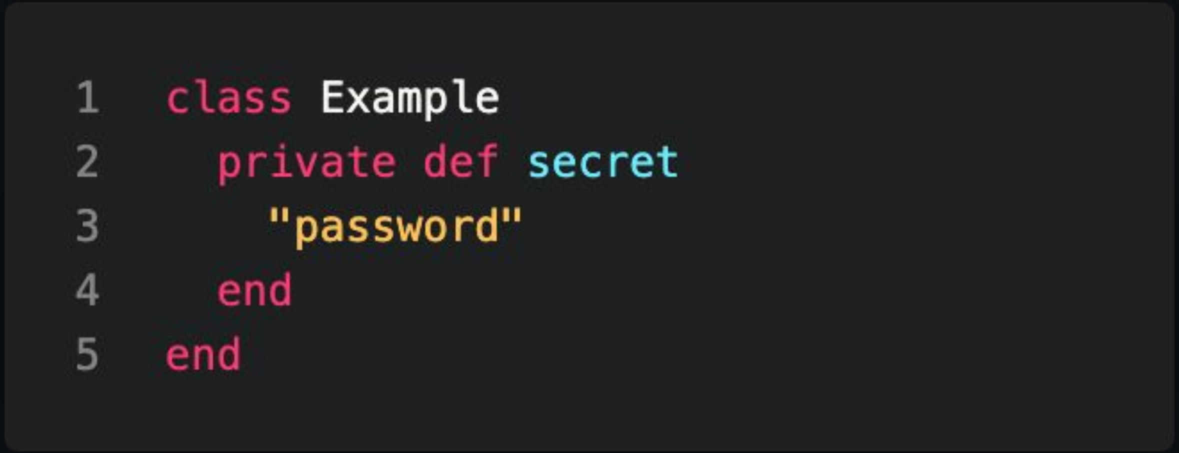
Can we use `Object.send()` safely?



```ruby
class Example
  private def secret
    "password"
  end
end
```

```
irb(main):006> Example.new.public_send(:secret)
(irb):6:in `public_send': private method `secret' called for an
instance of Example (NoMethodError)
```

Can we use `Object.send()` safely?



```ruby
class Example
  private def secret
    "password"
  end
end
```

```
irb(main):006> Example.new.public_send(:secret)
(irb):6:in `public_send': private method `secret' called for an
instance of Example (NoMethodError)

irb(main):007> Example.new.public_send(:send, :secret)
=> "password"
```

Can we use `Object.send()` safely?

```ruby
# variable method
method = params[:method] == 1 ? :method_a : :method_b
result = User.send(method, *args)

# variable target
target = params[:target] == 1 ? Account : User
result = target.send(:method, *args)
```

(please make sure to handle potentially unsafe additional arguments!)

Protecting Ourselves

Remediation

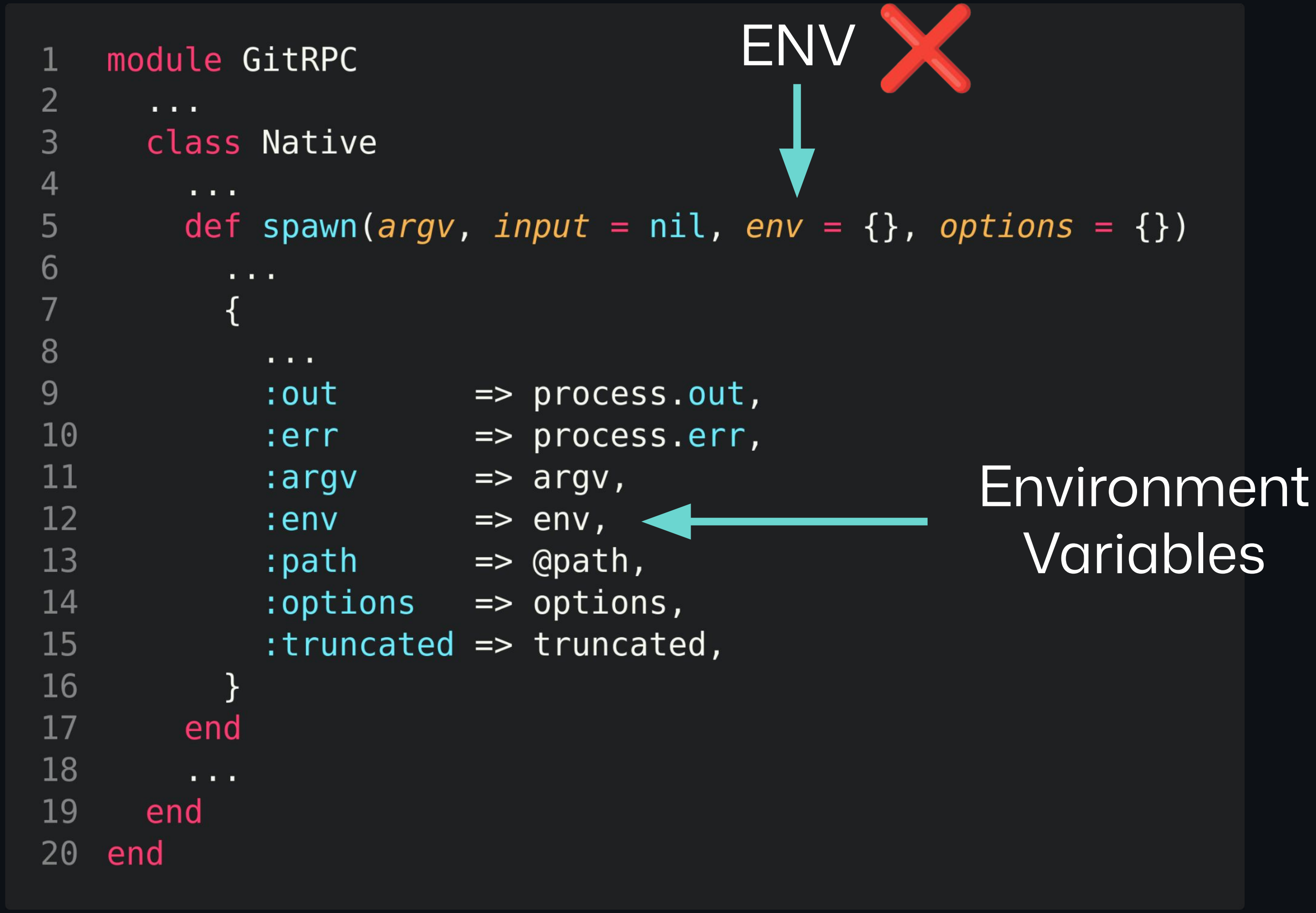
```ruby
def rid_key
  params[:rid_key]
end
```

```ruby
class Orgs::ActionsSettings::RepositoryItemsController < Orgs::Controller
  ...
  def repository_identifier_key
    return :global_relay_id unless rid_key.present?
    rid_key
  end

  def rid_key
    case params[:rid_key]
    when :global_relay_id, "global_relay_id"
      :global_relay_id
    when :id, "id"
      :id
    else
      nil
    end
  end
  ...
end
```

Protecting Ourselves

Remediation



```ruby
module GitRPC
  ...
  class Native
    ...
    def spawn(argv, input = nil, env = {}, options = {})
      ...
      {
        ...
        :out       => process.out,
        :err       => process.err,
        :argv      => argv,
        :env       => env,
        :path      => @path,
        :options   => options,
        :truncated => truncated,
      }
    end
    ...
  end
end
```

Protecting Ourselves

Remediation

- Already moved away from using Marshal for cookie serialization
- _gh_render was no longer used (part of a deprecated service)

Protecting Ourselves

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

```
- [x] Patch the vulnerable code
- [ ] Rotate all of the secrets
```

Protecting Ourselves

Vulnerability Lifecycle

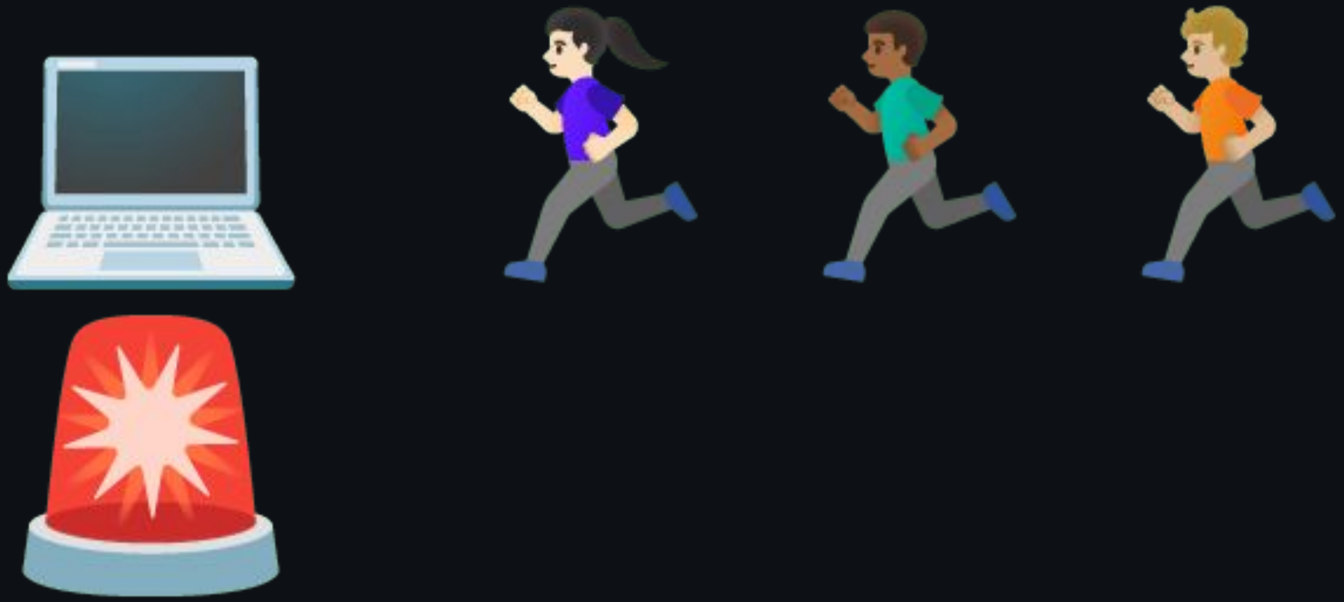
- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

Protecting Ourselves

Vulnerability Lifecycle

- Intake
- Triage
- Remediation
- Variant Analysis
- Disclosure

GitHub Engineers 0.5s after a new vulnerability is reported



Code Scanning Tools

- Brakeman (Rails)
  - Run at any stage in development
- RuboCop
  - Easy to write and use + lots of community support
  - *PublicSend* Cop (from GitLab Security)
- Semgrep / Opengrep
  - More accurate AST parsing to identify vulnerable code paths
- CodeQL
  - Easy to start using with our default query set
  - Can be used to write very accurate queries
- *and more!*

Takeaways

1. Use powerful language features with great care
2. Utilize and customize your code scanning tools
3. Always validate user controlled inputs in your code

Agenda

1. *(Ethically)* Hacking GitHub

   Walking through a high-impact vulnerability leading to secrets exposure

2. Protecting Ourselves

   Processes and tools to improve Ruby code security and prevent regressions

3. **Keeping Secrets**

   Handling and securing sensitive values in production Ruby apps

How Nimble Are Your Secrets?

Challenges of rotating secrets

- Separate config and secrets
- Identifying owning teams and impact of rotation
- Automating secrets rotation
- How long will things take?

Have a playbook / rotation plan (and actually test it!)

How To Keep Secrets

Storage Mechanisms

- .env
- Rails Credentials
- Networked secrets store (HashiCorp Vault, Azure Key Vault, etc)
    - Auditability
    - JIT access
    - Least privilege
    - Secrets versioning

Can we protect secrets within a Ruby process?

Goal

- Achieve a minimal footprint for sensitive data in memory

Strategies

- Overloading methods and blocking language features?
- Moving away from ENV
  - Using subprocesses
  - External secrets store
  - Custom class for managing secrets

Further Reading

- Send()-ing Myself Belated Christmas Gifts – GitHub.com's Environment Variables & GHES Shell
- GitHub: How GitHub uses CodeQL to secure GitHub
- Phrack Magazine Issue 0x45: Attacking Ruby on Rails Applications
- RubyKaigi 2024: Remembering (ok, not really Sarah) Marshal
- CodeQL zero to hero part 1
- RailsConf: Stop Hacker From Reading Your Data (ActiveRecord::Encryption)

https://gh.io/rubykaigi-2025

Call to Hacktion

GitHub Security runs a bug bounty program to engage with security researchers, providing a legal safe harbor for ethical hacking and vulnerability disclosures to GitHub.

Learn more at bounty.github.com.

Thank you!

https://gh.io/rubykaigi-2025

Dennis Pacewicz (@*lyninx*)
Senior Product Security Engineer
GitHub

Wei Lin Ngo (@*Creastery*)
Staff Security Engineer
Praetorian

RubyKaigi 2025