Security Advisory
SvelteKit

Created by Viktor Chuchurski
04/03/2023

Overview

This document summarizes the results of a vulnerability discovered in the SvelteKit framework while performing a security audit for one of Doyensec's customers. While security testing was not meant to be comprehensive in term of attack and code coverage, we have identified a cross-site request forgery (CSRF) protection bypass, allowing third-party domains to perform malicious request to APIs created using the SvelteKit's web server.

About Us

Doyensec is an independent security research and development company focused on vulnerability discovery and remediation. We work at the intersection of software development and offensive engineering to help companies craft secure code.

Research is one of our founding principles and we invest heavily in it. By discovering new vulnerabilities and attack techniques, we constantly improve our capabilities and contribute to secure the applications we all use.

Insufficient Cross-Site Request Forgery Protection	
Vendor	SvelteKit
Severity	Medium
Vulnerability Class	Cross-Site Request Forgery
Component	kit/src/runtime/server/respond.js#L52
Status	Closed
CVE	CVE-2023-29003
Credits	Viktor Chuchurski

Summary

SvelteKit framework offers developers an option to create simple REST APIs. This is done by defining a `+server.js` file, containing endpoint handlers for different HTTP methods.

SvelteKit provides out-of-the-box Cross-Site Request Forgery (CSRF) protection to its users. The protection is implemented at `kit/src/runtime/server/respond.js#L52`. While the implementation does a sufficient job in mitigating common CSRF attacks, the protection can be bypassed by simply specifying a different `Content-Type` header value.

If abused, this flaw will allow malicious requests to be submitted from third-party domains, which in extreme scenarios can lead to unauthorized access to users' accounts.

Technical Description

The CSRF protection is implemented using the code shown below.

```
const forbidden =
  // (1)
  request.method === 'POST' &&
  // (2)
  request.headers.get('origin') !== url.origin &&
  // (3)
  is_form_content_type(request);

if (forbidden) {
  // (4)
```

```
  const csrf_error = error(403, `Cross-site ${request.method} form submissions
are forbidden`);
  if (request.headers.get('accept') === 'application/json') {
    return json(csrf_error.body, { status: csrf_error.status });
  }
  return text(csrf_error.body.message, { status: csrf_error.status });
}
```

If the incoming request specifies a POST method (1), the protection will compare the server's origin with the value of the HTTP `Origin` header (2). A mismatch between these values signals that a potential attack has been detected. The final check is performed on the request's `Content-Type` header (3) wether the value is either `application/x-www-form-urlencoded` or `multipart/form-data`. If all the previous checks pass, the request will be rejected with an 403 error response (4).

The `is_form_content_type` validation is not sufficient to mitigate all possible variations of this type of attack. If a CSRF attack is performed with the `Content-Type` header set to `plain/text`, the protection will be circumvented and the request will be processed by the endpoint handler.

To reproduce this issue, create and run a simple server (by default running on `localhost:3000`) with a POST endpoint handler such as:

```
export async function POST({ request }) {
    console.log(await request.json())
    return new Response(String('success'));
}
```

Next, save the malicious HTML page:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
  <h1 id="name"></h1>
  <form action="http://localhost:3000/api/test" method="POST" enctype="text/
plain">
    <input type="hidden" name="&#123;&quot;name&quot;&#58;&quot;test"
value="&quot;&#44;&#13;&#10;&quot;age&quot;&#58;123&#125;&#13;&#10;" />
    <input type="submit" value="Submit" />
  </form>
</body>
</html>
```

in a file named `index.html`. Run another web server, using Python's built in `http.server` module (`python -m http.server`, by default running on `localhost:8000`), navigate to `localhist:8000/index.html` and click the `Submit` button.

Verify that the browser's URL has changed to `localhost:3000` and that the text `success` is displayed on the screen. Additionally, inspect the console of the SvelteKit web server and verify that the request body was parsed as valid JSON and printed out.

Remediation

To correctly mitigate this flaw, we recommend removing the `is_form_content_type` function call from the CSRF protection logic. As additional hardening of the CSRF protection mechanizm against potential method overrides[1], perform the validation on PUT, PATCH and DELETE methods as well.

```
const forbidden =
  + (request.method === 'POST' || request.method === 'PUT' || request.method ===
'PATCH' || request.method === 'DELETE') &&
  request.headers.get('origin') !== url.origin &&
  - is_form_content_type(request);
```

This will insure that all cross-site state altering requests are rejected by the SvelteKit server, regardless of the supplied `Content-Type` header.

Disclosure Timeline

04/03/2023	Issue identified and reported to the vendor
04/04/2023	Issue fixed and merged in `master` branch
04/04/2023	Version 1.15.1 released
04/05/2023	CVE-2023-29003 assigned and official advisory published

[1] https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions