Abusing Client-Side Desync on Werkzeug to perform XSS on default configurations

Kévin GERVOT (Mizu)
kevin.mizu@protonmail.com

Abstract. Werkzeug is a python Web Server Gateway Interface (WSGI) library for website development. It provides a simple way to set up an operational HTTP server for developers and is mostly present in Flask in development mode.
This article highlight an interesting Client-Side Desync attack (CVE-2022-29361 [11]) which can be used to perform Cross-Site Scripting (XSS) attack on Werkzeug. The full attack leverages 2 vulnerabilities, an HTTP request smuggling and an open redirect vulnerability present on the Werkzeug core. After performing these chained attacks, a malicious JavaScript file will be cached in the victim's browser, allowing to trigger XSS on every page of the website.

Introduction

Werkzeug is a python Web Server Gateway Interface (WSGI) [10] library for website development. It provides a simple way to set up an operational HTTP server for developers and is mostly present in Flask [15] in development mode. In latest versions, Werkzeug use python [16] library to handle most parts of the HTTP protocol.

In this paper, we will deep dive into an interesting case of Client-Side Desync (`CVE-2022-29361` [11]) on Werkzeug versions `2.1.0` to `2.1.1` (included). Using this vulnerability on a vulnerable host could lead to a full account takeover exploit via XSS.

1 Setting up a vulnerable environment

All information about setting up a vulnerable environment can be found on the following github repository:
`https://github.com/kevin-mizu/Werkzeug-CVE-2022-29361-PoC`

2 HTTP request parsing error in Werkzeug

2.1 Finding the vulnerable commit

As Werkzeug is a development Web Server Gateway Interface (WSGI), Pallets Projects [3] frequently updates the code of the Werkzeug core to facilitate its usage. Among the changes, the commit `4795b9a7` (released in january 2022) aims to `enable HTTP/1.1 when server has multiple workers`. This commit is special as it forces Werkzeug to use `keep-alive` connections when `threaded` or `processes` options are enabled. At first sight, this modification isn't an issue, but still creates new possible attack vectors on Werkzeug.

This commit was merged into Werkzeug production branch in commit `9a3a981d70d2e9ec3344b5192f86fcaf3210cd85` [19] and later available in release `2.1.0`. After this commit, issues `#2380` [9] and `#4507` [17] involving bugs in the query handler were opened.

2.2 Understanding the issue

In impacted versions, when performing a `POST` request with parameters that aren't properly handled in the Flask application, it will break the next HTTP request. From the developer's point of view, this was more annoying than dangerous and was not interpreted as a security issue. But is it really not a security issue?

```
(sstic) [19:10] /sstic$ python app.py
127.0.0.1 - - [30/Mar/2023 19:11:56] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [30/Mar/2023 19:11:56] "key=valueGET /static/js/main.js HTTP/1.1" 405 -
```

Fig. 1. $2.1.0 \leq$ Werkzeug $\leq 2.1.1$ improper handling of `POST` parameters [19].

As we can see on figure 1, it is possible to control arbitrary bytes in the next request from the body of a `POST` request. As explained in the issue `#2546` [6], this behavior comes from python `http.server` [16] module which doesn't properly handle `keep-alive` connections. Therefore, when not handled in the Flask application, `POST` parameters are left in the connection queue and are still usable at the beginning of the next request. Moreover, all queries made to the server are sent over the same connection (ID) that is used for local ressources access which gives an interesting context to perform Client-Side Desync attacks, as seen in figure 2.

Name ▲	Status	Type	Size	Connection ID
localhost	200	document	196 B	227387
main.js	304	script	241 B	227387

Fig. 2. Same connection ID is used for multiple ressources access.

3 Client-Side Desync to the rescue

3.1 What are Client-Side Desync attacks?

Client-Side Desync attacks are a subset of request smuggling attacks, which occur between the browser and the web server without proxy. This vulnerability is made possible when a web server doesn't properly handle the request's body during `keep-alive` connections. James Kettle (`@albinowax`) published an excellent article on the subject last summer which describe them in very specific details [7].

Let's deep dive into a step-by-step example of a Client-Side Desync:

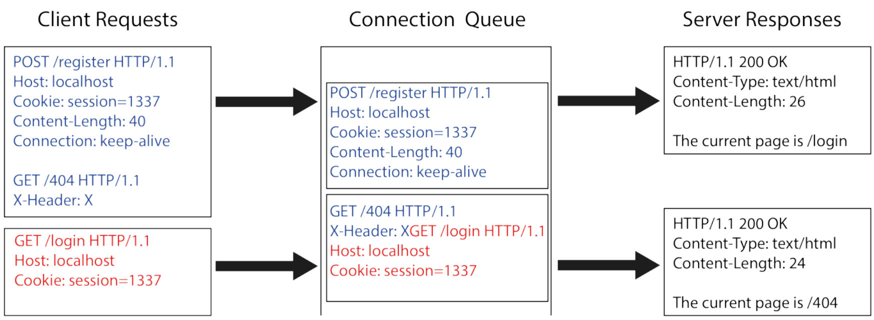

Fig. 3. Incorrect server-side parsing leads to Client-Side Desync.

In the figure 3, the client sends a `POST` request in `keep-alive` mode which contains the beginning of another `GET` request in the body. If the web server is vulnerable, it will not process the request body and leave it in the connection queue. Then, when the browser sends another request, it will read the previous `POST` request body and the newly received `GET` request. Thus, the client will expect to receive the content of `/login`, but instead the web server will answer with `/404`.

3.2 Where do they occur?

Client-Side Desync mainly occurs on endpoints that don't require data to be sent. As an example, a static image file or a server side redirection endpoint may be good candidates as they usually don't require user to provide information.

3.3 How to abuse them?

With this kind of vulnerabilities, real problems happened when it is possible to perform cross-site attacks and keep the user's session thanks to CORS [1] or cookie missconfiguration [5]. Under this particular conditions and depending on the website features, it might be possible to abuse them to leak the `Cookie` header of the second query. A good example of this attack can be found on PortSwigger Academy [14].

4 Exploit Chain

In section 2, we exposed a request smuggling vulnerability in Werkzeug `2.1.0` to `2.1.1`, without exposing any security risk. In section 3, we learned what Client-Side Desync are and how to use them. A notable difference in the Werkzeug context is its connection management. In fact, in vulnerable versions, it will keep the same connection ID for each query, this is really interesting as it allows to potentially desync a request to a ressource initiated by the browser.

Therefore, if the first ressource is a script file, it might be possible to control its content thanks to the Client-Side Desync vulnerability. As the vulnerable application hasn't any file upload feature, it is not possible to control a file on the server. It is necessary to find an open redirect vulnerability inside the Werkzeug core, to use it to change the script file location.

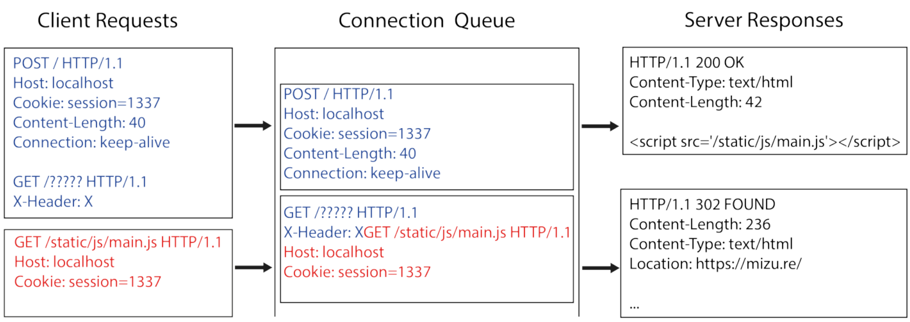

Fig. 4. Abuse open redirect to change script location.

5 Finding an open redirect

5.1 Old reported vulnerabilities

Werkzeug is a development WSGI which makes it more focused on usability than security. Therefore, it is important to take a look to newly added features or old vulnerability fixes and reports. Among them, an 8 years old open redirect inside Werkzeug core reported on `#822` [18] (`CVE-2020-28724` [8]) is a good start to go. This vulnerability was firstly reported on Flask repository and occured when using an URL path that starts by 2 slashes. When trying to access it with a double slash path we successfully get redirected to the remote ressource.

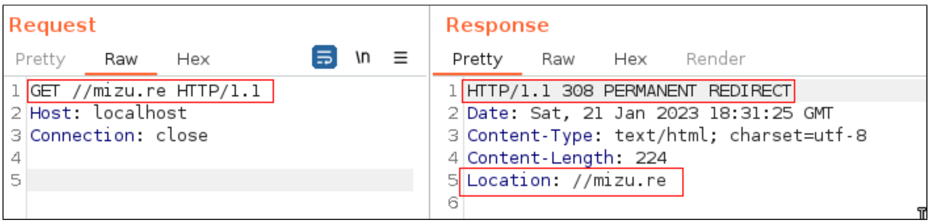

Fig. 5. Open redirect on Werkzeug < `0.11.6`.

5.2 Understanding the fix

The Werkzeug project has fixed this vulnerability in the commit `556bdcb13516617335c10efdedf3c1bd50b31b6d` [13]. They ensure that the `scheme` in the `url_parse` output is not empty with a valid `netloc`. This is a good way to fix it has there is impossible to create an URL with those conditions on the browser side. This would be like trying to go to `https://domain.comhttps://mizu.re` which makes no sense.

```
class WSGIRequestHandler(BaseHTTPRequestHandler):
    # ...
    if request_url.scheme and request_url.netloc:
        environ['HTTP_HOST'] = request_url.netloc
```

Fig. 6. Werkzeug commit `556bdcb13516617335c10efdedf3c1bd50b31b6d` [13].

5.3 Bypassing the fix

Even if the fix prevents the abuse of the open redirect in normal browser's usage, the redirection will still be present. Indeed, as we have a Client-Side Desync in Werkzeug, and this kind of attacks allows to control arbitrary bytes of the next request, it is possible to abuse it to recreate the open redirect payload from a malicious HTTP request.

In addition, it is important to notice that the redirect isn't a simple `302` redirect, but a `308` permanent redirect. This type of redirect will force the browser to cache the actual location of the ressource for further usage. Therefore, successfully achieving the full chain exploit would poison the location of the script for each loading page, even if the victim user doesn't trigger the attack again.

6 Wrapping up everything

6.1 Creating the client-side exploit

To create the client-side exploit, we need to find a way to send the payload cross-site with one request which will change the first resource location. The necessary condition for this exploit is that the connection of the malicious request must be in `keep-alive` mode. If this condition is not met, the connection will immediately be closed and no exploit would be possible. Therefore, the best way to achieve our exploit will be to use a `<form>` with `method="POST"` using `target="http://vulnerable-website/"`. As we want to control the first bytes of the next query, we will need to use space and line return (CR.LF). In order to wrap this kind of payload into a `<form>` POST data, we need to insert it inside the attribute `name` value.

```
<form action="http://vulnerable-website/" method="POST">
<textarea name="GET http://rogue-web-server HTTP/1.1
Foo: x">Mizu</textarea>
</form>              <script> x.submit() </script>
```

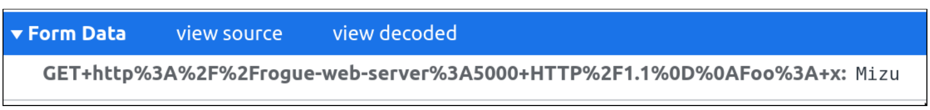

Fig. 7. Simple form with Client-Side Desync payload. URL encoded body content, the payload is invalid.

Unfortunately, by default, requests made by the HTML `<form>` use `application/x-www-form-urlencoded` MIME Type [4] which break the payload. This could look like a dead cause, but reading the MDN documentation [12] about `<form>` tag and interesting attributes can be found. To change the previous request MIME Type to `text/plain`, the `enctype` attribute [2] can be used in the HTML `<form>` tag.

Fig. 8. Simple form with Client-Side Desync payload using `text/plain` encoding.

6.2 Prepare the rogue web server

To perform this exploit chain, it is necessary to setup a rogue server which will have one route that return the malicious JavaScript content and another that deliver the exploit payload to the victim. To do so, PoC can be found on the following github repository: `https://github.com/kevin-mizu/Werkzeug-CVE-2022-29361-PoC`

6.3 Perform the final exploit chain

Finally, sending the exploit URL to the victim will perform everything described earlier and execute the XSS. In addition, each time a new page is opened containing the same script file, the XSS will be triggered. This leads to a full compromise of the website thanks to the cached malicious javascript file in the user's browser. A complete video of the exploit can be found here: `https://www.youtube.com/watch?v=HJWafpbMcbA`

Conclusion

We have demonstrated an efficient Client-Side Desync attack on Werkzeug WSGI. This attack allows to perform XSS on a vulnerable instance without any requirements. Moreover, even if the challenge was to find an exploit with no requirements, this full chain attack could be performed in a much more easier way if other vulnerabilities are already present in the web application.

While this paper only focus on vulnerability research on Werkzeug which is only used in development server, it would be interesting to conduct the same research on production WSGI.

Acknowledgements

I would like to thank Remi GASCOU (@podalirius_) for helping me on vulnerability report stages and reviewing this paper.

References

1. Cors access control allow origin. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin.
2. enctype form attribute. https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/enctype.
3. Pallets projects. https://github.com/pallets.
4. Post requests mime-types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.
5. Samesite cookie attribute. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.
6. abergmann. Issue 2546: Http request smuggling inside the development server. https://github.com/pallets/werkzeug/issues/2546.
7. James Kettle (@albinowax). Browser-powered desync attacks: A new frontier in http request smuggling. https://portswigger.net/research/browser-powered-desync-attacks.
8. Ramin Frajpour Cami. Werkzeug open redirect cve-2020-28724. https://nvd.nist.gov/vuln/detail/CVE-2020-28724.
9. ImreC. Issue 2380: Http request smuggling inside the development server. https://github.com/pallets/werkzeug/issues/2380.
10. Web Server Gateway Interface. What is wsgi? https://wsgi.readthedocs.io/en/latest/what.html.
11. Kevin GERVOT (Mizu). Werkzeug request smuggling cve-2022-29361. https://nvd.nist.gov/vuln/detail/cve-2022-29361.
12. Mozilla. Developer network docs. https://developer.mozilla.org/en-US/.
13. PalletsTeam. Werkzeug 0.11.6 open redirect fix. https://github.com/pallets/werkzeug/commit/556bdcb13516617335c10efdedf3c1bd50b31b6d.
14. PortSwigger. Lab: Client-side desync. https://portswigger.net/web-security/request-smuggling/browser/client-side-desync/lab-client-side-desync.
15. Pallets Projects. Flask. https://flask.palletsprojects.com/.
16. Python. http.server - http servers. https://docs.python.org/3/library/http.server.html.
17. tangbinyeer. Issue 4507: Flask 2.1.0 can't handle request method properly when sending post repeatedly with an empty body. https://github.com/pallets/flask/issues/4507.
18. ThiefMaster. Issue 822: dev server sets wrong http_host when path starts with a double slash. https://github.com/pallets/werkzeug/issues/822.
19. Werkzeug. Commit introducing the vulnerability. https://github.com/pallets/werkzeug/commit/9a3a981d70d2e9ec3344b5192f86fcaf3210cd85.