#! Anatomy of a Bug

Autopsy of CVSS 10.0

React2Shell

The Patient

- **Name:** React2Shell
- **CVE:** CVE-2025-55182
- **Diagnosis:** Insecure Deserialization **(RCE)**
- **System:** Next.js / React Server Components
- **Severity:** 🔥 10.0 (Critical)

What does 10.0 mean?

- **No Auth:** No authentication required.
- **Zero Interaction:** No user clicks required.
- **Remote:** A single HTTP request is sufficient.
- **Total Loss:** Full shell access on the server.

#1 The Architecture

The Limitations of JSON

#1 The Architecture

👉 React Server Components (RSC) require **asynchronous streaming;** JSON is too static for this purpose.

👉 The solution is the **Flight protocol,** a **dynamic streaming** format.

👉 Flight sends chunks that have gaps and reference each other using ***$id*** **references.**

👉 These references are **resolved at runtime.**

👉 A feature, not a bug: The parser **dynamically reconstructs the structure** based on these IDs.

#1 The Architecture

A small example of the protocol:

```
files = {
  "0": (None, '["$1"]'),          # Chunk 0 references Chunk 1
  "1": (None, '{"data":"$2"}'),   # Chunk 1 references Chunk 2
  ...
}
```

#2 The Mindset

JavaScript as an open nervous system

#2 The Mindset

👉 Attackers view JS as a **network of references**, not as code.

👉 They ask: **Which references can I traverse along?**

👉 Via ***__proto__*** → ***constructor***, you eventually land at the ***Function constructor*** (***eval***-like).

👉 React does not check for ***$id:property*** if the property was **truly its own.**

👉 This unintentionally allows for the **traversal of the prototype graph.**

#3 The Kill-Chain

The “Raw Chunk” Bypass

#3.1 The Kill-Chain

Thenable Trap: A Promise That *Promises Nothing*

👉 The RSC parser automatically treats any object possessing a *.then* property as a Promise (a so-called *Thenable*).

👉 As a result, the parser executes its *then()* method, *even if* it contains no genuine Promise logic.

👉 This means: The mere existence of this property alone constitutes a side-effect, causing the Parser's control flow to deviate into specific execution paths.

👉 An artificial *.then* property thus triggers processes that are only intended for real *Promises*.

#3.2 The Kill-Chain

$@ — Accessing the "Raw" Chunk

👉 Normally, the Flight parser recursively resolves chunks (e.g., *$1:field→value*).

👉 $@, however, means: Give me the chunk directly, still unparsed.

👉 A raw chunk contains internal properties (e.g., *status, value, reason, internal methods*) that are otherwise hidden by parsing.

👉 This makes it possible to direct the parser engine towards objects that were never intended to originate from user input.

#3.3 The Kill-Chain

Status Manipulation: A Forced Code Branch

👉 Every chunk internally has a **status:** *pending, resolved_model, resolved_module, blocked,* etc.

👉 The parser executes different functions **depending on the status.**

👉 If the status is artificially set to *resolved_model,* the parser calls the *initializeModelChunk* function—which normally runs **after deserialization.**

👉 This **bypasses** the normal **deserialization and validation flow.**

#3.4 The Kill-Chain

Blob Gadget ($B): A Harmless Feature Being Misused

👉 *$B* is a Flight mechanism for Blob/FormData data that internally accesses *_response._formData.get()*.

👉 The attacker controls *_response*, meaning the parser suddenly interprets an attacker-controlled object as the FormData source.

👉 The parser calls a method that should actually originate from a genuine FormData object...
but is now controlled by the attacker!

#3.4 The Kill-Chain

Official Payload:

```
crafted_chunk: {
    "then": "$1:__proto__:then",   // Hook the Promise-Flow
    "status": "unresolved_model",  // Force Initialization
    "reason": -1,
    "value": '{"then": "$B0"}',    // Trigger the Blob-Gadget ($B)
    "_response": {                 // Payload injection at `_prefix`
        "_prefix": "process.mainModule.require('child_process').execSync('calc'); //",
        "_formData": {
            "get": "$1:constructor:constructor", // The `Weapon`
        }
    }
}
```

#3.5 The Kill-Chain

The Constructor Chain: Why It Is Critical

👉 In JavaScript, the following chain leads to a function creation primitive:

obj→_proto_→constructor→constructor

👉 Normally this is harmless, but the parser calls the attacker-controlled *_formData.get()*.

👉 If *_formData.get()* is part of this chain, it is executed without verification.

👉 This is how the attacker indirectly gains access to the *Function* constructor.

#3.6 The Kill-Chain

The Crux: The Parser Becomes the Attack Vector

👉 The chain works because:

👉 *Thenables* are automatically executed.

👉 $@ makes internal structures accessible.

👉 *Status* is manipulable.

👉 Gadgets like *$B* open powerful code paths.

👉 React does not check if properties were truly its own → the prototype graph is open.

👉 Result: Not a single vulnerability, but a chain reaction of multiple design weaknesses.

Result:

The Server executes `Function("...")()`.

RCE.

A Masterclass in Insecure Deserialization.

#4 Developer's Takeaway

#4 Developer's Takeaway

Never trust the input

👉 **Trust Boundaries:** Data and logic must never mix. A data format that allows the execution of functions *(Thenables)* during parsing is inherently risky.

👉 **Prototype Hardening:** Object.freeze(Object.prototype) can prevent entire classes of attacks.

👉 **Explicit Checks:** Never rely on a property not being present. Explicitly check whether it exists on the instance.

#5 The Fix.

#5 The Fix.

Here is the exact code change that stopped the RCE:

```
@@ -78,7 +80,10 @@ export function preloadModule<T>(

 export function requireModule<T>(metadata: ClientReference<T>): T {
   const moduleExports = parcelRequire(metadata[ID]);
-  return moduleExports[metadata[NAME]];
+  if (hasOwnProperty.call(moduleExports, metadata[NAME])) {
+    return moduleExports[metadata[NAME]];
+  }
+  return (undefined: any);
 }
```

#5.1 The Fix.

A note on JS

JavaScript is too helpful—that is the problem.

Before the Fix: The Prototype Chain

👉 Prototype search via *moduleExports[metadata[NAME]]*.

👉 JS automatically climbs the Prototype Chain.

👉 It returns the inherited constructor.

👉 Result: The attacker gains access to *Function* → RCE.

#5.2 The Fix.

Property Check

After the Fix: The Property Check

👉 *hasOwnProperty.call(...)* now queries for own properties exclusively, not inherited ones.

👉 JS inspects the object itself → no own constructor property found.

👉 STOP. The lookup process does not traverse up the Prototype Chain.

👉 Result: Access denied, the attacker receives false.

Status

Patch immediately to Next.js 14.2.19+ or 15.0.5+.
Understand your dependencies.
Understand your serialization.

#! Anatomy of a Bug

Technical Credits:

msanft & maple3142

Author: @tralsesec

#RedTeam #AppSec #ReactJS #NextJS #ExploitDev #RCE #CVE202555182 #SecureCoding